

BIOSIGNAL and MEDICAL IMAGE PROCESSING

Third Edition

**JOHN L. SEMMLOW
BENJAMIN GRIFFEL**

**BIO SIGNAL
and
MEDICAL
IMAGE
PROCESSING**

Third Edition

**BIOSIGNAL
and
MEDICAL
IMAGE
PROCESSING**

Third Edition

**JOHN L. SEMMLOW
BENJAMIN GRIFFEL**



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

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

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2014 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20140117

International Standard Book Number-13: 978-1-4665-6737-5 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

*This book is dedicated to the memory of Larry Stark, who showed
me the many possibilities and encouraged me to take them.*

Contents

Preface.....	xv
Acknowledgments.....	xix
Authors	xxi
Chapter 1 Introduction	1
1.1 Biosignals	1
1.2 Biosignal Measurement Systems.....	3
1.3 Transducers.....	4
1.4 Amplifier/Detector	6
1.5 Analog Signal Processing and Filters.....	7
1.5.1 Filter Types	8
1.5.2 Filter Bandwidth.....	9
1.5.3 Filter Order	9
1.5.4 Filter Initial Sharpness	12
1.6 ADC Conversion.....	13
1.6.1 Amplitude Slicing	15
1.6.2 Time Slicing.....	18
1.6.3 Edge Effects.....	22
1.6.4 Buffering and Real-Time Data Processing	24
1.7 Data Banks.....	24
1.8 Summary.....	24
Problems	25
Chapter 2 Biosignal Measurements, Noise, and Analysis.....	29
2.1 Biosignals	29
2.1.1 Signal Encoding.....	30
2.1.2 Signal Linearity, Time Invariance, Causality	31
2.1.2.1 Superposition.....	32
2.1.3 Signal Basic Measurements	33
2.1.4 Decibels	37
2.1.5 Signal-to-Noise Ratio	37
2.2 Noise	38
2.2.1 Noise Sources	39
2.2.2 Noise Properties: Distribution Functions	40
2.2.2.3 Electronic Noise.....	42
2.3 Signal Analysis: Data Functions and Transforms.....	44
2.3.1 Comparing Waveforms.....	45
2.3.1.1 Vector Representation	46

Contents

2.3.1.2	Orthogonality	48
2.3.1.3	Basis Functions.....	50
2.3.2	Correlation-Based Analyses.....	52
2.3.2.1	Correlation and Covariance	53
2.3.2.2	Matrix of Correlations.....	54
2.3.2.3	Cross-Correlation	56
2.3.2.4	Autocorrelation	59
2.3.2.5	Autocovariance and Cross-Covariance	62
2.3.3	Convolution and the Impulse Response.....	64
2.4	Summary.....	70
	Problems	71
Chapter 3	Spectral Analysis: Classical Methods	77
3.1	Introduction.....	77
3.2	Fourier Series Analysis.....	79
3.2.1	Periodic Functions.....	81
3.2.1.1	Symmetry	88
3.2.2	Complex Representation.....	88
3.2.3	Data Length and Spectral Resolution.....	92
3.2.3.1	Aperiodic Functions	94
3.2.4	Window Functions: Data Truncation.....	95
3.3	Power Spectrum	101
3.4	Spectral Averaging: Welch's Method	105
3.5	Summary.....	111
	Problems	111
Chapter 4	Noise Reduction and Digital Filters.....	119
4.1	Noise Reduction	119
4.2	Noise Reduction through Ensemble Averaging	120
4.3	Z-Transform	123
4.3.1	Digital Transfer Function	124
4.4	Finite Impulse Response Filters.....	127
4.4.1	FIR Filter Design and Implementation.....	131
4.4.2	Derivative Filters: Two-Point Central Difference Algorithm	141
4.4.2.1	Determining Cutoff Frequency and Skip Factor	144
4.4.3	FIR Filter Design Using MATLAB.....	145
4.5	Infinite Impulse Response Filters	148
4.5.1	IIR Filter Implementation.....	150
4.5.2	Designing IIR Filters with MATLAB	151
4.6	Summary.....	155
	Problems	155
Chapter 5	Modern Spectral Analysis: The Search for Narrowband Signals.....	163
5.1	Parametric Methods.....	163
5.1.1	Model Type and Model Order	164

5.1.2	Autoregressive Model.....	166
5.1.3	Yule–Walker Equations for the AR Model.....	169
5.2	Nonparametric Analysis: Eigenanalysis Frequency Estimation	174
5.2.1	Eigenvalue Decomposition Methods	175
5.2.2	Determining Signal Subspace and Noise Subspace Dimensions.....	176
5.2.3	MATLAB Implementation	176
5.3	Summary.....	185
	Problems	186
Chapter 6	Time–Frequency Analysis	193
6.1	Basic Approaches	193
6.2	The Short-Term Fourier Transform: The Spectrogram	193
6.2.1	MATLAB Implementation of the STFT	194
6.3	The Wigner–Ville Distribution: A Special Case of Cohen’s Class.....	200
6.3.1	The Instantaneous Autocorrelation Function	200
6.3.2	Time–Frequency Distributions.....	202
6.3.3	The Analytic Signal	207
6.4	Cohen’s Class Distributions.....	208
6.4.1	The Choi–Williams Distribution.....	208
6.5	Summary.....	212
	Problems	214
Chapter 7	Wavelet Analysis	217
7.1	Introduction.....	217
7.2	Continuous Wavelet Transform	218
7.2.1	Wavelet Time–Frequency Characteristics.....	220
7.2.2	MATLAB Implementation	222
7.3	Discrete Wavelet Transform.....	225
7.3.1	Filter Banks.....	226
7.3.1.1	Relationship between Analytical Expressions and Filter Banks	230
7.3.2	MATLAB Implementation	231
7.3.2.1	Denoising	234
7.3.2.2	Discontinuity Detection.....	236
7.4	Feature Detection: Wavelet Packets	237
7.5	Summary.....	243
	Problems	244
Chapter 8	Optimal and Adaptive Filters	247
8.1	Optimal Signal Processing: Wiener Filters	247
8.1.1	MATLAB Implementation	250
8.2	Adaptive Signal Processing	254
8.2.1	ALE and Adaptive Interference Suppression.....	256
8.2.2	Adaptive Noise Cancellation.....	257
8.2.3	MATLAB Implementation	258

Contents

8.3	Phase-Sensitive Detection.....	263
8.3.1	AM Modulation	263
8.3.2	Phase-Sensitive Detectors.....	265
8.3.3	MATLAB Implementation.....	267
8.4	Summary.....	269
	Problems	270
Chapter 9	Multivariate Analyses: Principal Component Analysis and Independent Component Analysis	273
9.1	Introduction: Linear Transformations	273
9.2	Principal Component Analysis.....	276
9.2.1	Determination of Principal Components Using Singular-Value Decomposition.....	278
9.2.2	Order Selection: The Scree Plot.....	279
9.2.3	MATLAB Implementation.....	279
9.2.3.1	Data Rotation.....	279
9.2.4	PCA in MATLAB	280
9.3	Independent Component Analysis.....	284
9.3.1	MATLAB Implementation.....	291
9.4	Summary.....	293
	Problems	294
Chapter 10	Chaos and Nonlinear Dynamics	297
10.1	Nonlinear Systems	297
10.1.1	Chaotic Systems	298
10.1.2	Types of Systems	301
10.1.3	Types of Noise	302
10.1.4	Chaotic Systems and Signals	303
10.2	Phase Space	304
10.2.1	Iterated Maps.....	308
10.2.2	The Hénon Map	308
10.2.3	Delay Space Embedding	311
10.2.4	The Lorenz Attractor.....	313
10.3	Estimating the Embedding Parameters.....	315
10.3.1	Estimation of the Embedding Dimension Using Nearest Neighbors	316
10.3.2	Embedding Dimension: SVD	323
10.4	Quantifying Trajectories in Phase Space: The Lyapunov Exponent	324
10.4.1	Goodness of Fit of a Linear Curve	326
10.4.2	Methods of Determining the Lyapunov Exponent.....	327
10.4.3	Estimating the Lyapunov Exponent Using Multiple Trajectories	330
10.5	Nonlinear Analysis: The Correlation Dimension	335
10.5.1	Fractal Objects	335
10.5.2	The Correlation Sum	337
10.6	Tests for Nonlinearity: Surrogate Data Analysis.....	345

Contents

10.7 Summary.....	353
Exercises.....	353
Chapter 11 Nonlinearity Detection: Information-Based Methods.....	357
11.1 Information and Regularity	357
11.1.1 Shannon's Entropy Formulation.....	358
11.2 Mutual Information Function	359
11.2.1 Automutual Information Function.....	364
11.3 Spectral Entropy	370
11.4 Phase-Space-Based Entropy Methods.....	374
11.4.1 Approximate Entropy	374
11.4.2 Sample Entropy.....	378
11.4.3 Coarse Graining.....	381
11.5 Detrended Fluctuation Analysis.....	385
11.6 Summary.....	389
Problems	390
Chapter 12 Fundamentals of Image Processing: The MATLAB Image Processing Toolbox.....	393
12.1 Image-Processing Basics: MATLAB Image Formats.....	393
12.1.1 General Image Formats: Image Array Indexing	393
12.1.2 Image Classes: Intensity Coding Schemes	395
12.1.3 Data Formats	396
12.1.4 Data Conversions	397
12.2 Image Display	399
12.3 Image Storage and Retrieval.....	404
12.4 Basic Arithmetic Operations.....	404
12.5 Block-Processing Operations	411
12.5.1 Sliding Neighborhood Operations	411
12.5.2 Distinct Block Operations	415
12.6 Summary.....	416
Problems	418
Chapter 13 Image Processing: Filters, Transformations, and Registration	421
13.1 Two-Dimensional Fourier Transform	421
13.1.1 MATLAB Implementation	423
13.2 Linear Filtering.....	425
13.2.1 MATLAB Implementation	426
13.2.2 Filter Design	427
13.3 Spatial Transformations.....	433
13.3.1 Affine Transformations.....	434
13.3.1.1 General Affine Transformations	436
13.3.2 Projective Transformations	437
13.4 Image Registration.....	441
13.4.1 Unaided Image Registration	442
13.4.2 Interactive Image Registration.....	445

Contents

13.5 Summary.....	447
Problems	448
Chapter 14 Image Segmentation.....	451
14.1 Introduction.....	451
14.2 Pixel-Based Methods	451
14.2.1 Threshold Level Adjustment	452
14.2.2 MATLAB Implementation	455
14.3 Continuity-Based Methods	456
14.3.1 MATLAB Implementation	457
14.4 Multithresholding.....	463
14.5 Morphological Operations	465
14.5.1 MATLAB Implementation	466
14.6 Edge-Based Segmentation	472
14.6.1 Hough Transform.....	473
14.6.2 MATLAB Implementation	474
14.7 Summary.....	477
Problems	479
Chapter 15 Image Acquisition and Reconstruction.....	483
15.1 Imaging Modalities	483
15.2 CT, PET, and SPECT	483
15.2.1 Radon Transform.....	484
15.2.2 Filtered Back Projection.....	486
15.2.3 Fan Beam Geometry	489
15.2.4 MATLAB Implementation of the Forward and Inverse Radon Transforms: Parallel Beam Geometry.....	489
15.2.5 MATLAB Implementation of the Forward and Inverse Radon Transforms: Fan Beam Geometry	492
15.3 Magnetic Resonance Imaging.....	494
15.3.1 Magnetic Gradients	497
15.3.2 Data Acquisition: Pulse Sequences.....	497
15.4 Functional MRI.....	499
15.4.1 fMRI Implementation in MATLAB	501
15.4.2 Principal Component and ICA.....	503
15.5 Summary.....	506
Problems	508
Chapter 16 Classification I: Linear Discriminant Analysis and Support Vector Machines.....	511
16.1 Introduction.....	511
16.1.1 Classifier Design: Machine Capacity	514
16.2 Linear Discriminators.....	515
16.3 Evaluating Classifier Performance	520
16.4 Higher Dimensions: Kernel Machines.....	525
16.5 Support Vector Machines	527
16.5.1 MATLAB Implementation	530

Contents

16.6	Machine Capacity: Overfitting or “Less Is More”	534
16.7	Extending the Number of Variables and Classes	536
16.8	Cluster Analysis	537
16.8.1	<i>k</i> -Nearest Neighbor Classifier	538
16.8.2	<i>k</i> -Means Clustering Classifier	540
16.9	Summary.....	544
	Problems	545
Chapter 17	Classification II: Adaptive Neural Nets.....	549
17.1	Introduction.....	549
17.1.1	Neuron Models	549
17.2	Training the McCullough–Pitts Neuron.....	552
17.3	The Gradient Decent Method or Delta Rule.....	557
17.4	Two-Layer Nets: Back Projection.....	561
17.5	Three-Layer Nets.....	565
17.6	Training Strategies.....	568
17.6.1	Stopping Criteria: Cross-Validation.....	568
17.6.2	Momentum.....	569
17.7	Multiple Classifications.....	573
17.8	Multiple Input Variables	575
17.9	Summary.....	577
	Problems	578
Appendix A:	Numerical Integration in MATLAB.....	581
Appendix B:	Useful MATLAB Functions	585
Bibliography.....		589
Index		593

Preface

Signal processing can be broadly defined as the application of analog or digital techniques to improve the utility of data. In biomedical engineering applications, improved utility usually means that the data provide better diagnostic information. Analog techniques are sometimes applied to data represented by a time-varying electrical signal, but most signal-processing methods take place in the digital domain where data are represented as discrete numbers. These numbers may represent a time-varying signal, or an image. This book deals exclusively with signal processing of digital data, although the Introduction briefly describes analog processes commonly found in medical devices.

This book should be of interest to a broad spectrum of engineers, but it is written specifically for biomedical engineers (called bioengineers in some circles). Although the applications are different, the signal-processing methodology used by biomedical engineers is identical to that used by electrical and computer engineers. The difference for biomedical engineers is in the approach and motivation for learning signal-processing technology. A computer engineer may be required to develop or modify signal-processing tools, but for biomedical engineers these techniques are tools to be used. For the biomedical engineer, a detailed understanding of the underlying theory, while always of value, is not usually needed. What is needed is an understanding of what these tools do and how they can be employed to meet biomedical challenges. Considering the broad range of knowledge required to be effective in biomedical engineering, encompassing both medical and engineering domains, an in-depth understanding of all of the useful technology is not realistic. It is most important to know what tools are available, have a good understanding of what they do (if not how they do it), be aware of the most likely pitfalls and misapplications, and know how to implement these tools given available software packages. The basic concept of this book is that just as the cardiologist can benefit from an oscilloscope-type display of the ECG without a deep understanding of electronics, so a biomedical engineer can benefit from advanced signal-processing tools without always understanding the details of the underlying mathematics.

In line with this philosophy, most of the concepts covered in this book can be broken down into two components. The first component is a presentation of a general understanding of the approach sufficient to allow intelligent application of the concepts. Sometimes this includes a description of the underlying mathematical principles. The second section describes how these tools can be implemented using the MATLAB® software package and several of its toolboxes.

This book was originally written for a single-semester course combining signal and image processing. Classroom experience indicates that this ambitious objective is possible for most graduate formats, although eliminating a few topics may be desirable. For example, some of the introductory or basic material covered in Chapters 1 and 2 could be skipped or treated lightly for students with the appropriate prerequisites. In addition, topics such as Advanced Spectral Methods (Chapter 5), Time–Frequency Analysis (Chapter 6), Wavelets (Chapter 7), Advanced Filters (Chapter 8), and Multivariate Analysis (Chapter 9) are pedagogically independent and can be covered as desired without affecting the other material. With the inclusion of Chapters

Preface

10 and 11 on nonlinear signal processing, the material on image processing could be deleted and the course restricted to one-dimensional signal processing.

Nonlinear signal analysis is an emerging area within signal processing that recognizes that real biomedical signals do not always conform to our assumptions of linearity. For such signals, nonlinear analysis can reveal additional information that linear analysis cannot. New techniques and algorithms are being continuously added, but Chapters 10 and 11 provide a foundation for those interested in nonlinear signal analyses. These techniques have gained a foothold in the area of heart rate and EEG analysis, where there is evidence of nonlinear behavior. Other new biomedical applications include DNA sequence analysis and speech processing. Nonlinear signal processing can only grow in importance as the techniques are improved offering greater speed and better immunity to noise, and as new biomedical applications are discovered.

This third edition of this textbook was undertaken to improve clarity and understanding by expanding both examples and problems. Although much of the material covered here will be new to most students, the book is not intended as an “introductory” text since the goal is to provide a working knowledge of the major signal-processing methods without the need for additional course work. The challenge of covering a broad range of topics at a useful, working depth is motivated by current trends in biomedical engineering education, particularly at the graduate level where a comprehensive education must be attained with a minimum number of courses. This has led to the development of “core” courses to be taken by all students. This book was written for just such a core course in the Graduate Program of Biomedical Engineering of Rutgers University. It is also quite suitable for an upper-level undergraduate course and would also be of value for students in other disciplines that would benefit from a working knowledge of signal and image processing.

It is not possible to cover such a broad spectrum of material to a depth that enables productive application without heavy reliance on MATLAB-based examples and problems. In this regard, the book assumes that the student has some knowledge of MATLAB programming and has the basic MATLAB software package at hand, including the Signal Processing and Image Processing Toolboxes. (MATLAB also has toolboxes for wavelets and adaptive neural nets, but these sections are written so as not to require additional toolboxes, primarily to keep the number of required toolboxes to a minimum.) The problems are an essential part of this book and often provide a discovery-like experience regarding the associated topic. The code used for all examples is provided in the software package that accompanies this book, available at <http://www.crcpress.com/product/isbn/9781466567368>. Since many of the problems are extensions or modifications of examples given in the book, time can be saved by starting with the code of a related example. The package also includes support routines and data files used in the examples and problems, and the code used to generate many of the figures.

For instructors, there is an Education Support Package available that contains the problem solutions and PowerPoint® presentations for each of the chapters. These PowerPoint slides include figures, equations, and text and can be modified by the instructor as desired.

In addition to heavy reliance on MATLAB problems and examples, this book makes extensive use of simulated data. Examples involving biological signals are only occasionally used. In our view, examples using biological signals provide motivation, but they are not generally very instructive. Given the wide range of material to be presented at a working depth, emphasis is placed on learning the tools of signal processing; motivation is left to the reader (or the instructor).

Organization of the book is straightforward. Chapters 1 through 4 are fairly basic. Chapter 1 covers topics related to analog signal processing and data acquisition, while Chapter 2 includes topics that are basic to all aspects of signal and image processing. Chapters 3 and 4 cover classical spectral analysis and basic digital filtering, topics fundamental to any signal-processing course. Advanced spectral methods are covered in Chapter 5 and are important due to their widespread use in biomedical engineering. Chapter 6 and the first part of Chapter 7 cover topics related to spectral analysis when the signal’s spectrum is varying in time, a condition often found in biological signals. Chapter 7 addresses both continuous and discrete wavelets, another popular technique used in the analysis of biomedical signals. Chapters 8 and 9 feature advanced

topics. In Chapter 8, optimal and adaptive filters are covered: the latter's inclusion is also motivated by the time-varying nature of many biological signals. Chapter 9 introduces multivariate techniques, specifically Principal Component Analysis and Independent Component Analysis, two analysis approaches that are experiencing rapid growth with regard to biomedical applications. Chapters 10 and 11 cover nonlinear signal-processing methods. The next three chapters are about image processing with the first of these, Chapter 12, covering the conventions used by MATLAB's Imaging Processing Toolbox. Image processing is a vast area and the material included here is limited primarily to areas associated with medical imaging: image acquisition (Chapter 15); image filtering, enhancement, and transformation (Chapter 13); and segmentation and registration (Chapter 14). The last two chapters concern classification: linear discriminators and support vector machines in Chapter 16, and adaptive neural nets in Chapter 17.

Some chapters cover topics that may require an entire book for thorough presentation. Such chapters involve a serious compromise and omissions are extensive. Our excuse is that classroom experience with this approach seems to work: students end up with a working knowledge of a vast array of signal- and image-processing tools. A few of the classic or major books on these topics are cited in a bibliography at the end of the book. No effort has been made to construct an extensive bibliography or reference list since more current lists are readily available off the Web.

Textbook Protocols

Some early examples of MATLAB code are presented in full, but in most examples some of the routine code (such as for plotting, display, and labeling operation) is omitted. Nevertheless, we recommend that students carefully label (and scale when appropriate) all graphs done in the

Table I Textbook Conventions

Symbol	Description/General Usage
$x(t), y(t)$	General functions of time, usually a waveform or signal
k, n	Data indices, particularly for digitized time data
N	Number of points in a signal (maximum index or size of a data set)
L	Length of a data set (alternate) such as a set of filter coefficients
$x[n], y[n]$	Waveform variable, usually digitized time variables (i.e., a discrete variable)
m	Index of variable produced by transformation, or the index of specifying the member number of a family of functions (i.e., $f_m(t)$)
$X(f), Y(f)$	Frequency representation (usually complex) of a time function
$X[m], Y[m], X(f)$	Frequency representation (usually complex) of a discrete variable
$h(t)$	Impulse response of a linear system
$h[n]$	Discrete impulse response of a linear system
$b[n]$	Digital filter coefficients representing the numerator of the discrete transfer function, hence, the same as the impulse response
$a[n]$	Digital filter coefficients representing the denominator of the discrete transfer function
$b[m,n]$	Coefficients of a two-dimensional, image filter
<code>Courier font</code>	MATLAB command, variable, routine, or program
<code>'Courier font'</code>	MATLAB filename or string variable

Preface

problems. Some effort has been made to use consistent notation as described in Table I. In general, lower-case letters n and k are used as data indices, and capital letters (i.e., N or L) are used to indicate the signal length (or maximum subscript value) or other constants. In two-dimensional data sets, lower-case letters m and n are used to indicate the row and column subscripts of an array, while capital letters M and N are used to indicate vertical and horizontal dimensions, respectively. The letter m is also used as the index of a variable produced by a transformation, or as an index indicating a particular member of a family of related functions.* As is common, analog or continuous time variables are denoted by a “ t ” within parentheses (i.e., $x(t)$), whereas brackets enclose subscripts of digitized signals composed of discrete variables (i.e., $x[n]$). Other notation follows either standard or MATLAB conventions.

Italics is used to introduce important new terms that should be incorporated into the reader’s vocabulary. If the meaning of these terms is not obvious from their use, they are explained where they are introduced. All MATLAB commands, routines, variables, and code are shown in *courier typeface*. Single quotes are used for string variables following MATLAB conventions. These and other protocols are summarized in Table I.

MATLAB® and Simulink® are registered trademarks of The MathWorks, Inc. For product information, please contact:

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098 USA
Tel: 508 647 7000
Fax: 508-647-7001
E-mail: info@mathworks.com
Web: www.mathworks.com

* For example, m would be used to indicate the harmonic number of a family of harmonically related sine functions; that is, $f_m(t) = \sin(2\pi mt)$.

Acknowledgments

John L. Semmlow wishes to thank Susanne Oldham who, from the quaint little village of Yellow Springs, Ohio, managed to edit this book, and provided strong, continuing encouragement and support. He would also like to acknowledge the patience and support of Peggy Christ and Lynn Hutchings. Benjamin Griffel would like to particularly thank Professor Tony Shinbrot for discussions on chaos and nonlinear dynamics that contributed greatly to Chapters 10 and 11. Both authors are deeply indebted to Raymond Muzic of Case Western University for his thoughtful comments, corrections, and insights regarding many of the topics in this book. Finally, we both would like to thank our students who provided suggestions and whose enthusiasm for the material provided much-needed motivation.

Authors

John L. Semmlow was born in Chicago, in 1942, and received BSEE from the University of Illinois in Champaign in 1964. Following several years as a design engineer for Motorola, Inc., he entered the Bioengineering Program at the University of Illinois Medical Center in Chicago, and received his PhD in 1970. He has held faculty positions at the University of California, Berkeley, the University of Illinois, Chicago, and a joint Professorship in Surgery, Robert Wood Johnson Medical School and Biomedical Engineering, School of Engineering, Rutgers University in New Jersey. In 1985 he was a NSF/CNRS (National Science Foundation/Centre National de la Recherche Scientifique) Fellow in the Sensorimotor Control Laboratory of the University of Provence, Marseille, France. He was appointed a Fellow of the IEEE (Institute of Electrical and Electronics Engineers) in 1994 in recognition of his work on acoustic detection of coronary artery disease. He was elected Fellow of AIMBE (American Institute for Medical and Biological Engineering) in 2003 and Fellow of BMES (Biomedical Engineering Society) in 2005. He was founding chair of the International Conference on Vision and Movement in Man and Machines, first held in Berkeley in 1995. His primary research interests include discovering strategies for how the brain controls human motor behavior such as eye movements and the development of medical instrumentation for noninvasive detection of coronary artery disease.

He is an avid, if somewhat dangerous, skier, and an enthusiastic, but mediocre, tennis player. He teaches folk dance and is president of the Board of the New Brunswick Chamber Orchestra. He especially enjoys travel and the slow, but nonetheless expensive, sport of sailing. He recently purchased a sailboat (currently in the British Virgin Islands) and he soon hopes to launch his second career in computer-controlled kinetic art.

Benjamin Griffel was born in Brooklyn, New York. He received bachelor's and PhD degrees from Rutgers University in biomedical engineering in 2005 and 2011, respectively, and MS from Drexel University in 2007. His thesis topic explored the use of chaotic and entropy-based signal analysis to diagnose coronary artery disease using acoustic cardiovascular recordings. He is currently an INSPIRE (IRACDA New Jersey/New York for Science Partnerships in Research and Education) fellow at Rutgers University and a teaching partner at New Jersey City University. INSPIRE is an NIH (National Institutes of Health) fellowship designed to help young researchers develop teaching skills and to increase diversity within the areas of STEM (science, technology, engineering, and mathematics). His current research examines the use of nonlinear and chaotic signal analysis to analyze the effects of inflammation on heart rate recordings. In his spare time, Benjamin enjoys guitar playing, arm chair philosophizing, and spending time with his wife.

1

Introduction

1.1 Biosignals

Much of the activity in biomedical engineering, be it clinical or research, involves the measurement, processing, analysis, display, and/or generation of biosignals. Signals are variations in energy that carry information, and the search for information from living systems has been a preoccupation of medicine since its beginnings (Figure 1.1). This chapter is about the modification of such signals to make them more useful or more informative. To this end, this chapter presents tools and strategies that use digital signal processing to enhance the information content or interpretation of biosignals.

The variable that carries information (the specific energy fluctuation) depends on the type of energy involved. Biological signals are usually encoded into variations of electrical, chemical, or mechanical energy, although, occasionally, variations in thermal energy are of interest. For communication within the body, signals are primarily encoded as variations in electrical or chemical energy. When chemical energy is used, encoding is usually done by varying the concentration of the chemical within a “physiological compartment,” for example, the concentration of a hormone in blood. Bioelectric signals use the flow or concentration of ions, the primary charge carriers within the body, to transmit information. Speech, the primary form of communication between humans, encodes information as variations in pressure. Table 1.1 summarizes the different types of energy that can be used to carry information and the associated variables that encode this information. Table 1.1 also shows the physiological measurements that involve these energy forms.

Outside the body, information is commonly transmitted and processed as variations in electrical energy, although mechanical energy was used in the seventeenth and early eighteenth centuries to send messages. The semaphore telegraph used the position of one or more large arms placed on a tower or high point to encode letters of the alphabet. These arm positions could be observed at some distance (on a clear day) and relayed onward if necessary. Information processing can also be accomplished mechanically, as in the early numerical processors constructed by Babbage. Even mechanically based digital components have been attempted using variations in fluid flow. Modern electronics provides numerous techniques for modifying electrical signals at very high speeds. The body also uses electrical energy to carry information when speed is important. Since the body does not have many free electrons, it relies on ions, notably Na^+ , K^+ , and Cl^- , as the primary charge carriers. Outside the body, electrically based signals are so useful that signals carried by other energy forms are usually converted into electrical energy when significant transmission or processing tasks are required. The conversion of physiological



Figure 1.1 Information about internal states of the body can be obtained through the acquisition and interpretation of biosignals. Expanding such information is an ongoing endeavor of medicine and medical research. It is also the primary motivation of this chapter.

Table 1.1 Energy Forms and Associated Information-Carrying Variables

Energy	Variables (Specific Fluctuation)	Common Measurements
Chemical	Chemical activity and/or concentration	Blood ion, O ₂ , CO ₂ , pH, hormonal concentrations, and other chemistry
Mechanical	Position Force, torque, or pressure	Muscle movement, cardiovascular pressures, muscle contractility, valve, and other cardiac sounds
Electrical	Voltage (potential energy of charge carriers) Current (charge carrier flow)	EEG, ECG, EMG, EOG, ERG, EGG, and GSR
Thermal	Temperature	Body temperature and thermography

1.2 Biosignal Measurement Systems

energy into an electric signal is an important step, often the first step, in gathering information for clinical or research use. The energy conversion task is done by a device termed a *transducer*,* specifically, a *biotransducer*. The biotransducer is usually the most critical component in systems designed to measure biosignals.

With the exception of this chapter, this book is limited to topics on digital signal and image processing. To the extent possible, each topic is introduced with the minimum amount of information required to use and understand the approach along with enough information to apply the methodology in an intelligent manner. Strengths and weaknesses of the various methods are also explored, particularly through discovery in the problems at the end of the chapter. Hence, the problems at the end of each chapter, most of which utilize the MATLAB® software package (Waltham, MA), constitute an integral part of the book and a few topics are introduced only in the problems.

A fundamental assumption of this book is that an in-depth mathematical treatment of signal-processing methodology is not essential for effective and appropriate application of these tools. This book is designed to develop skills in the application of signal- and image-processing technology, but it may not provide the skills necessary to develop new techniques and algorithms. References are provided for those who need to move beyond the application of signal- and image-processing tools to the design and development of new methodology. In the subsequent chapters, the major topics include sections on implementation using the MATLAB software package. Fluency with the MATLAB language is assumed and is essential for the use of this book. Where appropriate, a topic area may also include a more in-depth treatment, including some of the underlying mathematics.

1.2 Biosignal Measurement Systems

Biomedical measurement systems are designed to measure and usually record one or more biosignals. A schematic representation of a typical biomedical measurement system is shown in Figure 1.2. The term biomedical measurement is quite general and includes image acquisition

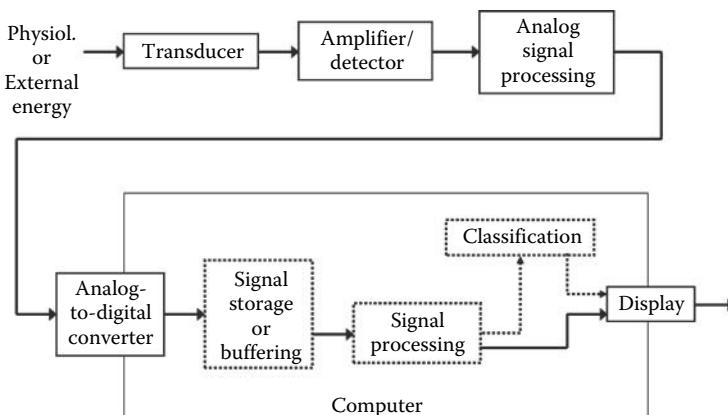


Figure 1.2 Schematic representation of a typical biomedical measurement system.

* Learning the vocabulary is an important part of mastering a discipline. In this book, the commonly used terms are highlighted using italics. Sometimes, the highlighted term is described when it is introduced, but occasionally, the determination of its definition is the responsibility of the reader.

Biosignal and Medical Image Processing

and the acquisition of different types of diagnostic information. Information from the biological process of interest must first be converted into an electric signal via the *transducer*. Some analog signal processing is usually required, often including amplification and lowpass (or bandpass) filtering. Since most signal processing is easier to implement using digital methods, the analog signal is converted into a digital format using an analog-to-digital converter (ADC). Once converted, the signal is often stored or *buffered* in the memory to facilitate subsequent signal processing. Alternatively, in *real-time* applications, the incoming data are processed as they come in, often with minimal buffering, and may not be permanently stored. Digital signal-processing algorithms can then be applied to the digitized signal. These signal-processing techniques can take on a wide variety of forms with varying levels of sophistication and they make up the major topic areas of this book. In some applications such as diagnosis, a classification algorithm may be applied to the processed data to determine the state of a disease or the class of a tumor or tissue. A wide variety of classification techniques exist and the most popular techniques are discussed in Chapters 16 and 17. Finally, some sort of output is necessary in any useful system. This usually takes the form of a display, as in imaging systems, but it may be some type of effector mechanism such as in an automated drug delivery system. The basic elements shown in Figure 1.2 are discussed in greater detail in the next section.

1.3 Transducers

A transducer is a device that converts energy from one form to another. By this definition, a lightbulb or a motor is a transducer. In signal-processing applications, energy is used to carry information; the purpose of energy conversion is to transfer that information, not to transform energy as with a lightbulb or a motor. In measurement systems, all transducers are so-called input transducers: they convert nonelectrical energy into an electronic signal. An exception to this is the electrode, a transducer that converts ionic electrical energy into electronic electrical energy. Usually, the output of a biomedical transducer is voltage (or current) whose amplitude is proportional to the measured energy (Figure 1.3).

Input transducers use one of the two different fundamental approaches: the input energy causes the transducer element to generate voltage or current, or the input energy creates a change in an electrical property (the resistance, inductance, or capacitance) of the transducer material. Most optical transducers use the first approach. Photons strike a photo sensitive material producing free electrons (or holes) that can then be detected as external current flow. Piezoelectric devices used in ultrasound also generate a charge when under mechanical stress. Many examples can be found in the use of the second category, a change in some electrical property. For example, metals (and semiconductors) undergo a consistent change in resistance with changes in temperature and most temperature transducers utilize this feature. Other examples include the strain gage, which measures mechanical deformation using the small change in resistance that occurs when the sensing material is stretched.

The energy that is converted by the input transducer may be generated by the physiological process itself, it may be energy that is indirectly related to the physiological process, or it may be energy produced by an external source. In the last case, the externally generated energy interacts with, and is modified by, the physiological process and it is this alteration that produces

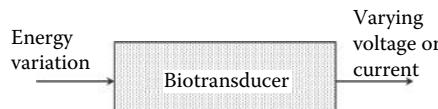


Figure 1.3 General systems representation of a transducer.

Table 1.2 Energy Forms and Related Direct Measurements

Energy	Measurement
Mechanical Length, position, and velocity Force and pressure	Muscle movement, cardiovascular pressures, muscle contractility valve, and other cardiac sounds
Heat	Body temperature and thermography
Electrical	EEG, ECG, EMG, EOG, ERG, EGG, and GSR
Chemical	Ion concentrations

the measurement. For example, when externally generated x-rays are transmitted through the body, they are absorbed by the intervening tissue and a measurement of this absorption is used to construct an image. Many diagnostically useful imaging systems are based on this external energy approach.

In addition to images created by external energy that is passed through the body, some images are generated using the energy of radioactive emissions of radioisotopes injected into the body. These techniques make use of the fact that selected or “tagged,” molecules will collect in the specific tissue. The areas where these radioisotopes collect can be mapped using a γ -camera or with certain short-lived isotopes, which are localized better using positron emission tomography (PET).

Many physiological processes produce energy that can be detected directly. For example, cardiac internal pressures are usually measured using a pressure transducer placed on the tip of a catheter introduced into the appropriate chamber of the heart. The measurement of electrical activity in the heart, muscles, or brain provides other examples of the direct measurement of physiological energy. For these measurements, the energy is already electrical and only needs to be converted from ionic into electronic current using an “electrode.” These sources are usually given the term ExG, where “x” represents the physiological process that produces electrical energy, ECG—electrocardiogram, EEG—electroencephalogram, EMG—electromyogram, EOG—electrooculogram, ERG—electroretinogram, and EGG—electrogastrogram. An exception to this terminology is the galvanic skin response (GSR), the electrical activity generated by the skin. The typical physiological measurements that involve the conversion of other energy forms into electrical energy are shown again in Table 1.2. Figure 1.4 shows the early ECG machine where the interface between the body and the electrical monitoring equipment was buckets filled with saline (“E” in Figure 1.4).

The biotransducer is often the most critical element in the system since it is the interface between the subject or life process and the rest of the system. The transducer establishes the risk or “noninvasiveness,” of the overall system. For example, an imaging system based on differential absorption of x-rays, such as a computed tomography (CT) scanner, is considered more *invasive* than an imaging system based on ultrasonic reflection since CT uses ionizing radiation that may have an associated risk. (The actual risk of ionizing radiation is still an open question and imaging systems based on x-ray absorption are considered *minimally invasive*.) Both ultrasound and x-ray imaging would be considered less invasive than, for example, monitoring internal cardiac pressures through cardiac catheterization, where a small catheter is threaded into the heart chambers.

Many critical problems in medical diagnosis await the development of new approaches and new transducers. Indeed, many of the outstanding problems in biomedical measurement, such as noninvasive measurement of internal cardiac pressures or the noninvasive measurement of intracranial pressure, await an appropriate, and undoubtedly clever, transducer mechanism.

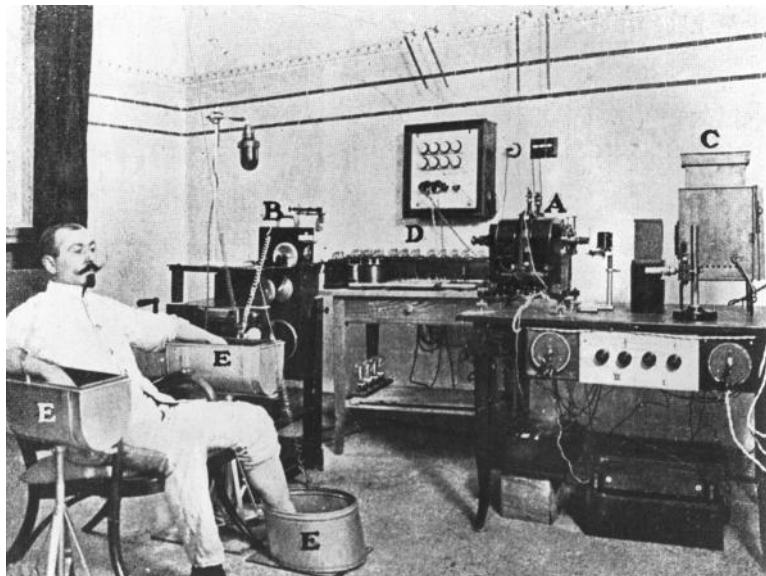


Figure 1.4 An early ECG machine using buckets filled with a saline solution as electrodes (marked "E" in this figure).

1.4 Amplifier/Detector

The design of the first analog stage in a biomedical measurement system depends on the basic transducer operation. If the transducer is based on a variation in electrical property, the first stage must convert that variation into a variation in voltage. If there is only one sensing element in the transducer, the transducer signal is said to be *single ended* and a constant current source can be used as a detector (Figure 1.5). A typical transducer that would use this type of detector circuit is a temperature-measurement device based on a thermistor, a resistor-like element that changes its resistance with temperature. Thermistors have a well-defined relationship between temperature and resistance. The detector equation for the circuit in Figure 1.5 follows Ohm's law:

$$V_{\text{out}} = I(R + \Delta R) \quad \text{where } \Delta R = f(\text{input energy}) \quad (1.1)$$

If the transducer can be configured differentially so that one element increases with increasing input energy while the other element decreases, the bridge circuit is commonly used as a detector. Figure 1.6 shows a device made to measure intestinal motility using strain gages.

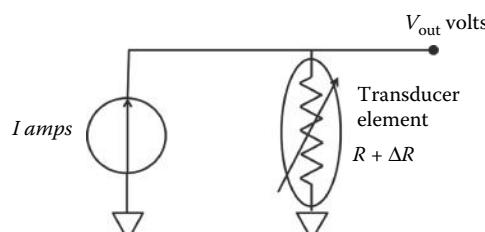


Figure 1.5 A constant current source is used in the detector circuit for a transducer based on an energy-induced variation in resistance. One example is a temperature-measuring transducer based on a thermistor, a semiconductor that changes resistance with temperature in a well-defined manner.

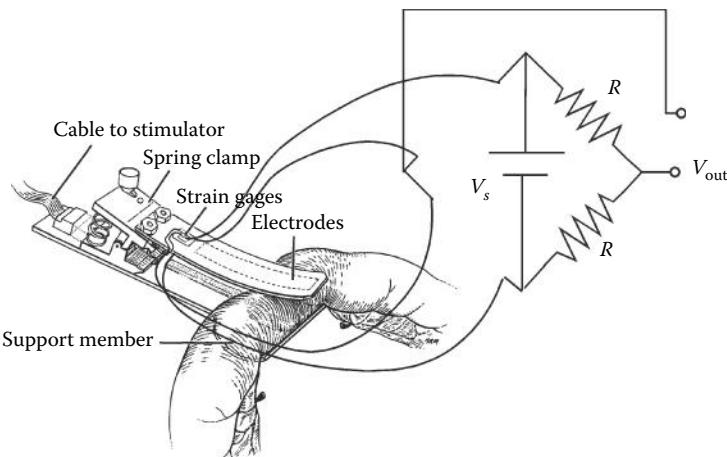


Figure 1.6 A strain gage probe used to measure motility of the intestine. The bridge circuit is used to convert the differential change in resistance from a pair of strain gages into a change in voltage.

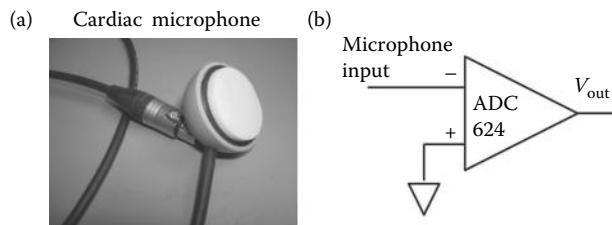


Figure 1.7 (a) A cardiac microphone that uses a piezoelectric element to generate a small voltage when deformed by sound pressure. (b) The detector circuit for this transducer is just a low-noise amplifier.

A bridge circuit detector is used in conjunction with a pair of differentially configured strain gages: when the intestine contracts, the end of the cantilever beam moves downward and the upper strain gage (visible) is stretched and increases in resistance whereas the lower strain gage (not visible) compresses and decreases in resistance. The output of the bridge circuit can be found from simple circuit analysis to be $V_{\text{out}} = V_s R/2$, where V_s is the value of the source voltage. If the transducer operates based on a change in inductance or capacitance, the above techniques are still useful except a sinusoidal voltage source must be used.

If the transducer element is a voltage generator, the first stage is usually an amplifier. Figure 1.7a shows a cardiac microphone used to monitor the sounds of the heart that is based on a piezoelectric element. The piezoelectric element generates a small voltage when sound pressure deforms this element; so, the detector stage is just a low-noise amplifier (Figure 1.7b). If the transducer produces a current output, as is the case in many electromagnetic detectors, then a current-to-voltage amplifier (also termed a transconductance amplifier) is used to produce a voltage output. In some circumstances, additional amplification beyond the first stage may be required.

1.5 Analog Signal Processing and Filters

While the most extensive signal processing is performed on digitized data using algorithms implemented in software, some analog signal processing is usually necessary. The most common

Biosignal and Medical Image Processing

analog signal processing restricts the frequency range or *bandwidth* of the signal using *analog filters*. It is this filtering that usually sets the bandwidth of the overall measurement system. Since signal bandwidth has a direct impact on the process of converting an analog signal into an equivalent (or nearly equivalent) digital signal, it is often an essential element in any biomedical measurement system. Filters are defined by several properties: filter type, bandwidth, and attenuation characteristics. The latter can be divided into initial and final characteristics. Each of these properties is described and discussed in the next section.

1.5.1 Filter Types

Analog filters are electronic devices that remove selected frequencies. Filters are usually termed according to the range of frequencies they *do not* suppress. Thus, *lowpass* filters allow low frequencies to pass with minimum attenuation while higher frequencies are attenuated. Conversely, *highpass* filters pass high frequencies, but attenuate low frequencies. *Bandpass* filters reject frequencies above and below a *passband* region. An exception to this terminology is *bandstop* filters that pass frequencies on either side of a range of attenuated frequencies.

These filter types can be illustrated by a plot of the filter's *spectrum*, that is, a plot showing how the filter treats the signal at each frequency over the frequency range of interest. Figure 1.8 shows stereotypical frequency spectra or frequency plots of the four different filter types described above. The filter gain is the ratio of output signal amplitude to input signal amplitude as a function of frequency:

$$\text{Gain}(f) = \frac{\text{Output values}(f)}{\text{Input values}(f)} \quad (1.2)$$

The lowpass filter has a filter gain of 1.0 for the lower frequencies (Figure 1.8). This means that the output equals the input at those frequencies. However, as frequency increases, the gain drops, indicating that the output signal also drops for a given input signal. The highpass filter has exactly the opposite behavior with respect to frequency (Figure 1.8). As frequency increases, the gain and output signal increase so that at higher frequency, the gain is 1.0 and the output

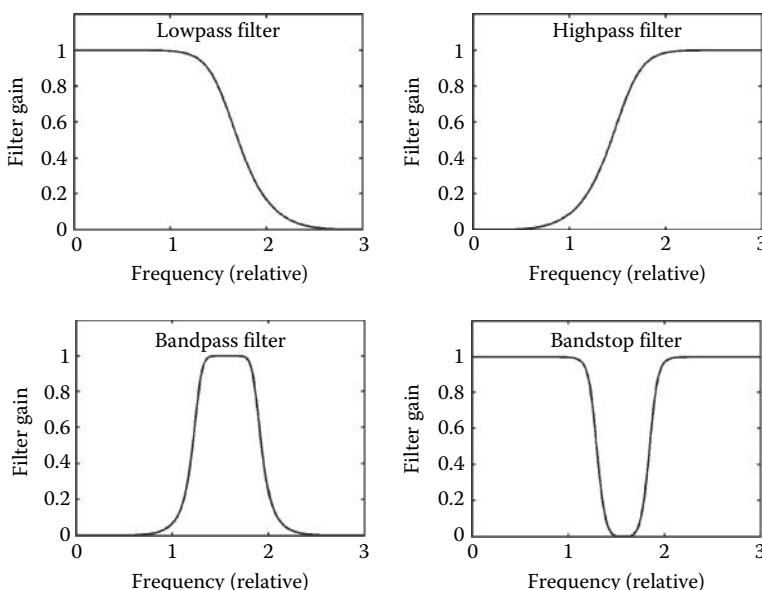


Figure 1.8 Influence on signal frequency of the four basic filter types.

equals the input. The bandpass filter is a combination of these two filters; so, the gain and output increase with frequency up to a certain frequency range where the gain is constant at 1.0, then gain decreases with further increases of frequency (Figure 1.8). The spectrum of the band-stop filter is the inverse of the bandpass filter (Figure 1.8).

Within each class, filters are also defined by the frequency ranges that they pass, termed the filter bandwidth, and the sharpness with which they increase (or decrease) attenuation as frequency varies. (Again, band-stop filters are an exception as their bandwidth is defined by the frequencies they reject.) Spectral sharpness is specified in two ways: as an initial sharpness in the region where attenuation first begins, and as a slope further along the attenuation curve. These various filter properties are best described graphically in the form of a frequency plot (sometimes referred to as a *Bode* plot), a plot of filter gain against frequency. Filter gain is simply the ratio of the output voltage divided by the input voltage, $V_{\text{out}}/V_{\text{in}}$, often taken in dB. (The dB operation is defined in Section 2.1.4, but is simply a scaled log operation.) Technically, this ratio should be defined for all frequencies for which it is nonzero, but practically, it is usually stated only for the frequency range of interest. To simplify the shape of the resultant curves, frequency plots sometimes plot gain in dB against the log of frequency.* When the output/input ratio is given analytically as a function of frequency, it is termed the *transfer function*. Hence, the frequency plot of a filter's output/input relationship can be viewed as a graphical representation of its transfer function (Figure 1.8).

1.5.2 Filter Bandwidth

The bandwidth of a filter is defined by the range of frequencies that are not attenuated. These unattenuated frequencies are also referred to as *passband* frequencies. Figure 1.9a shows the frequency plot of an ideal filter, a filter that has a perfectly flat passband region and an infinite attenuation slope. Real filters may indeed be quite flat in the passband region, but will attenuate with a gentler slope, as shown in Figure 1.9b. In the case of an ideal filter (Figure 1.9a), the bandwidth (the region of unattenuated frequencies) is easy to determine: specifically, the bandwidth ranges between 0.0 and the sharp attenuation at f_c Hz. When the attenuation begins gradually, as in Figure 1.9b, defining the passband region is problematic. To specify the bandwidth in this filter, we must identify a frequency that defines the boundary between the attenuated and unattenuated portions of the frequency curve. This boundary has been somewhat arbitrarily defined as the frequency when the attenuation is 3 dB.[†] In Figure 1.9b, the filter would have a bandwidth of 0.0 to f_c Hz, or simply f_c Hz. The filter whose frequency characteristics are shown in Figure 1.9c has a sharper attenuation characteristic, but still has the same bandwidth (f_c Hz). The bandpass filter whose frequency characteristics are shown in Figure 1.9d has a bandwidth of $f_h - f_l$ Hz.

1.5.3 Filter Order

The slope of a filter's attenuation curve is related to the complexity of the filter: more complex filters have a steeper slope, approaching the ideal filter as shown in Figure 1.9a. In analog filters, complexity is proportional to the number of energy storage elements in the circuit. These could be either inductors or capacitors, but are generally capacitors for practical reasons. Using standard circuit analysis, it can be shown that each independent energy storage device leads to an additional order of a polynomial in the denominator of the transfer function equation (Equation 1.2) that describes the filter. (The denominator of the transfer function is also

* When gain is plotted in dB, it is in logarithmic form, since the dB operation involves taking the log (see Section 2.1.4). Plotting gain in dB against log frequency puts the two variables in similar metrics and results in more straight-line plots.

[†] This defining point is not entirely arbitrary because when the signal is attenuated at 3 dB, its amplitude is 0.707 ($10^{-3/20}$) of what it was in the passband region and it has half the power of the unattenuated signal since $0.707^2 = 1/2$. Accordingly, this point is also known as the *half-power point*.

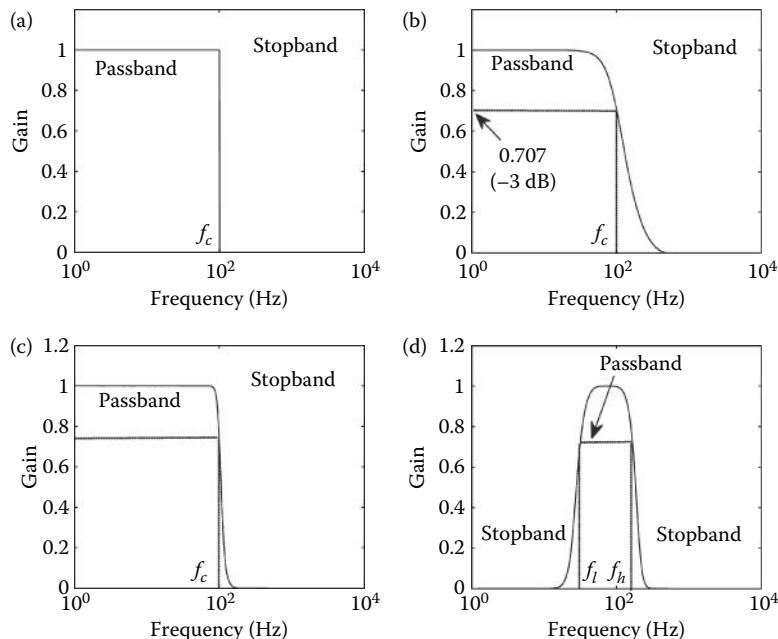


Figure 1.9 Frequency plots of ideal and realistic filters. Each of the frequency plots shown here has a linear vertical axis, but often, the vertical axis is plotted in dB. The horizontal axis is in log frequency. (a) Ideal lowpass filter. (b) Realistic lowpass filter with a gentle attenuation characteristic. (c) Realistic lowpass filter with a sharp attenuation characteristic. (d) Bandpass filter.

referred to as the *characteristic equation* because it defines the basic characteristics of the related system.) As with any polynomial equation, the number of roots of this equation will depend on the order of the equation; hence, filter complexity (i.e., the number of energy storage devices) is equivalent to the number of roots in the denominator of the transfer function. In electrical engineering, it has long been common to call the roots of the denominator equation *poles*. Thus, the complexity of a filter is also equivalent to the number of poles in the transfer function. For example, a *second-order* or *two-pole* filter has a transfer function with a second-order polynomial in the denominator and would contain two independent energy storage elements (very likely two capacitors).

Applying an asymptote analysis to the transfer function, it can be shown that for frequencies much greater than the cutoff frequency, f_c , the slope of most real-world filters is linear if it is plotted on a log-versus-log plot. Figure 1.10 shows the frequency characteristics of the transfer function of a first-order (single-pole) filter with a cutoff frequency, f_c , of 5 Hz plotted in both linear (Figure 1.10a) and dB versus log frequency (Figure 1.10b) format. Converting the vertical axis to dB involves taking the log (see Section 2.1.4); so Figure 1.10b is a log–log plot. At the cutoff frequency of 5 Hz, the frequency characteristic is curved, but at higher frequencies, above 10 Hz, the curve straightens out to become a downward slope that decreases 20 dB for each order of magnitude, or decade, increase in frequency. For example, at 50 Hz, the frequency characteristic has a value of -20 dB and at 500 Hz the value would be -40 dB although this is not shown in the graph. Plotting dB versus log frequency leads to the unusual units for the slope of dB/decade. Nonetheless, this type of plot is often used because it generates straight-line segments for the frequency characteristics of real-world filters and because taking logs extends the range of values presented on both axes. Both linear and dB versus

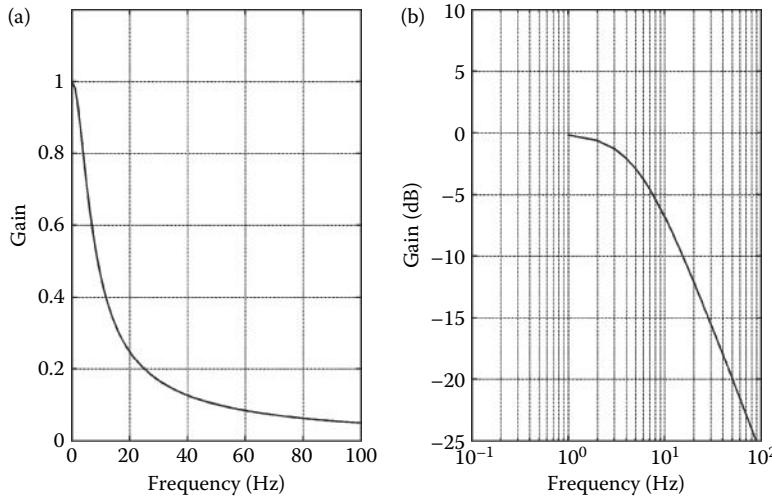


Figure 1.10 Two representations of the gain characteristics (i.e., transfer function) for a first-order filter. (a) A linear plot of gain against frequency. (b) The same curve is plotted with gain in dB, a log function, against log frequency. The attenuation slope above the cutoff frequency becomes a straight line with a slope of 20 dB/decade.

log frequency plotting is used in this book; the axes will describe which type of plot is being presented.

The downward slope of 20 dB/decade seen for the first-order filter shown in Figure 1.10b generalizes, in that for each additional filter pole added (i.e., each increase in filter order), the slope is increased by 20 dB/decade. (In a lowpass filter, the downward slope is sometimes referred to as the filter's *roll-off*.) Figure 1.11 shows the frequency plot of a second-order, two-pole filter

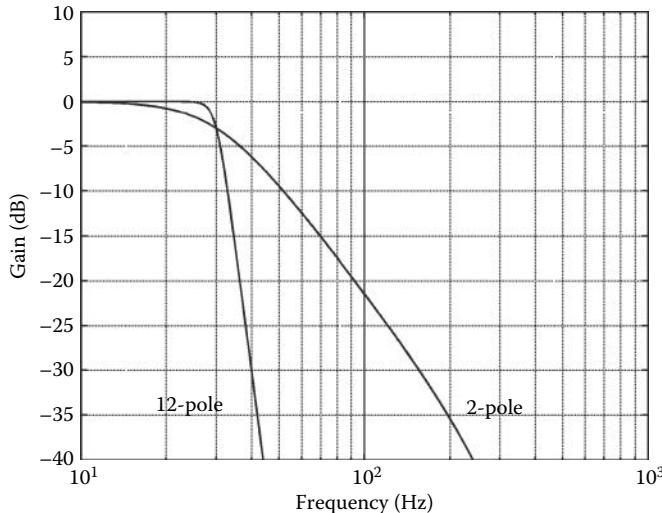


Figure 1.11 A dB versus log frequency plot of a second-order (two-pole) and a 12th-order (12-pole) lowpass filter having the same cutoff frequency. The higher-order filter more closely approaches the sharpness of an ideal filter.

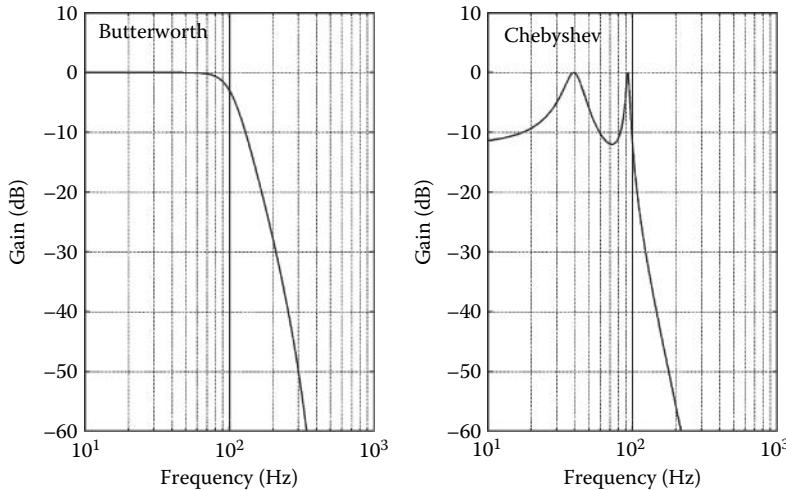


Figure 1.12 Two filters that have the same cutoff frequency (100 Hz) and the same order (four-pole), but differing in the sharpness of the initial slope. The filter labeled Chebyshev has a steeper initial slope, but contains ripples in the passband region.

with a slope of 40 dB/decade and a 12th-order lowpass filter. Both filters have the same cutoff frequency, f_c , hence the same bandwidth. The steeper slope or roll-off of the 12-pole filter is apparent. In principle, a 12-pole lowpass filter would have a slope of 240 dB/decade (12×20 dB/decade). In fact, this frequency characteristic is theoretical because in real analog filters, parasitic components and inaccuracies in the circuit elements limit the actual attenuation that can be obtained. The same rationale applies to highpass filters, except that the frequency plot decreases with decreasing frequency at a rate of 20 dB/decade for each highpass filter pole.

1.5.4 Filter Initial Sharpness

As shown above, both the slope and the initial sharpness increase with filter order (number of poles), but increasing filter order also increases the complexity and hence the cost of the filter. It is possible to increase the initial sharpness of the filter's attenuation characteristics without increasing the order of the filter, if you are willing to accept some unevenness or *ripple* in the passband. Figure 1.12 shows two lowpass, fourth-order filters having the same cutoff frequency, but differing in the initial sharpness of the attenuation. The one-marked Butterworth has a smooth passband, but the initial attenuation is not as sharp as the one marked Chebyshev, which has a passband that contains ripples. This property of analog filters is also seen in digital filters and is discussed in detail in Chapter 4.

EXAMPLE 1.1

An ECG signal of 1 V peak to peak has a bandwidth from 0.01 to 100 Hz. (Note that this frequency range has been established by an official standard and is meant to be conservative.) It is desired to reduce any noise in the signal by at least 80 dB for frequencies above 1000 Hz. What is the order of analog filter required to achieve this goal?

Solution

Since the signal bandwidth must be at least 100 Hz, the filter's cutoff frequency, f_c , must be not less than 100 Hz, but the filter must reduce the signal by 80 dB within 1 decade. Since typical

analog filters reduce the gain by 20 dB/decade for each pole, and an 80-dB reduction is desired, the filter must have at least $80/20 = 4$ poles.

Similar, but more involved, problems of this sort are explored in Chapter 4 that covers digital filters.

1.6 ADC Conversion

The last analog element in the typical measurement system shown in Figure 1.2 is the interface between the analog and digital world: the ADC. As the name implies, this electronic component converts an analog voltage into an equivalent digital number. In the process of ADC conversion, an analog or a continuous waveform, $x(t)$, is converted into a discrete waveform, $x[n]$,^{*} a function of real numbers that are defined as integers at discrete points in time. These numbers are called *samples* and the discrete points in time are usually taken at regular intervals termed the sample interval, T_s . The sample interval can also be defined by a frequency termed the *sample frequency*:

$$f_s = \frac{1}{T_s} \text{ Hz} \quad (1.3)$$

To convert a continuous waveform into a digital format requires slicing the signal in two ways: slicing in time and in slicing amplitude (Figure 1.13). So, the continuous signal $x(t)$ becomes just a series of numbers, $x[1], x[2], x[3], \dots, x[n]$ that are the signal values at times $1T_s, 2T_s, 3T_s$, and nT_s . In addition, if the waveform is longer than the computer memory, only a portion of the analog waveform can be converted into digital format. Segmenting a waveform to fit in computer memory is an operation termed *windowing*. The consequences of this operation are discussed in Chapter 3. Note that if a waveform is sampled at T_s seconds for a total time, T_T seconds, the number of values stored in the computer will be

$$N = \frac{T_T}{T_s} \text{ points} \quad (1.4)$$

The relationship between a sample stored in the computer and the time at which it was sampled is determined by its sample index, n , and the sample interval or sampling frequency:

$$t = nT_s = \frac{n}{f_s} \quad (1.5)$$

Each of these modifications, time slicing and amplitude slicing, has an impact on the resulting digital signal. Comparing the analog and digital signal in Figure 1.13 (left upper and lower graphs), it is evident that the digitized signal is not the same as the original. The question is: Does the difference matter, and if so, how does it matter? Intuitively, we might expect that if the time and amplitude slices were very small, then “for all practical purposes” (that great engineering expression), the digital and analog signals are essentially the same. The question then becomes: how small must we make the time and amplitude slices? This question is addressed separately for the two slicings in the following sections. Before approaching these questions, we present a simple MATLAB example of the generation and display of a digitized waveform.

* It is common to use brackets to represent discrete or digital functions whereas parentheses are used in continuous or analog functions. See Section 1.3.1.

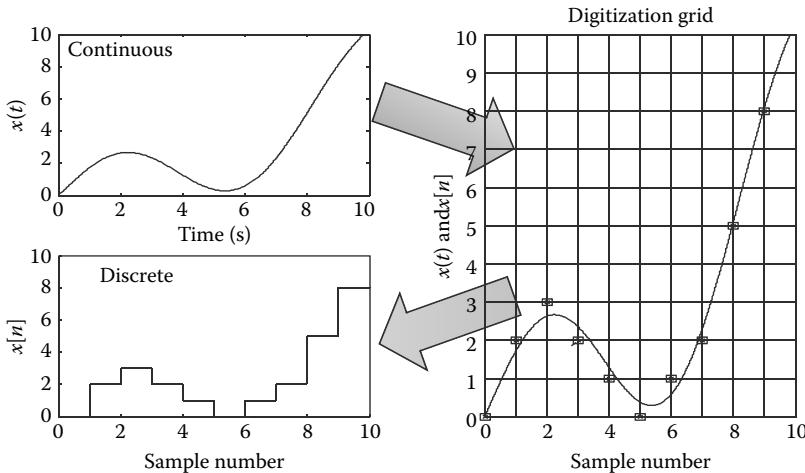


Figure 1.13 Digitizing a continuous signal, upper left, requires slicing the signal both in time and amplitude, right side. The result is a series of discrete numbers (squares) that approximate the original signal. The resultant digitized signal, lower left, consists of a series of discrete numerical values sampled at discrete intervals of time. In this example, $x[n] = 2, 3, 2, 1, 0, 2, 3, 5$, and 8 .

EXAMPLE 1.2

Generate a discrete 2-Hz sine wave using MATLAB. Assume a sample time of 0.01 s and use enough points to make the sine wave 1-s long, that is, the total time, T_T , should be 1 s. Plot the result in seconds. Label the time axis correctly (as is always expected in MATLAB problems).

Solution

In this example, we are given sample intervals and the total time; so, the number of discrete points that will be needed is fixed. From Equation 1.5, the number of points in the array will be

$$N = T_T/T_s = 100 \text{ points}$$

In many MATLAB problems, the number of points is arbitrary and we need to pick some reasonable value for N . The best way to solve this problem in MATLAB is first to generate a time vector* that is 1-s long with increments of T_s seconds: $t = 0:T_s:1$. We do not even have to find the number of points explicitly, although it turns out to be 100. We can use this time vector in conjunction with the MATLAB `sin` function to generate the sine wave.

```
% Example 1.2 Generate a discrete 2 Hz sine wave using MATLAB.
% Assume a sample time of 0.01 sec. and use enough points to make
% the sine wave 1 sec. long; i.e., the total time, TT should be 1 sec.
%
clear all; close all;
Ts = .01; % Define Ts = .01 sec
TT = 1; % Define total time = 1 sec
f = 2; % Define frequency = 2 Hz
t = 0:Ts:1; % Generate time vector
x = sin(2*pi*f*t); % Generate desired sine wave
plot(t,x,'k'); % Plot sine wave as discrete points
xlabel('Time (sec)'); % and label
ylabel('x(t)');
```

* The reason a MATLAB sequence of numbers or an array is sometimes called a vector is discussed in Section 2.3.1.1.

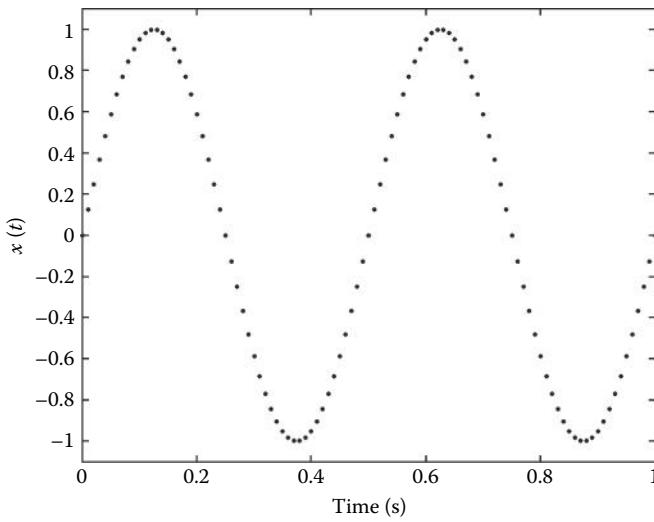


Figure 1.14 Sequence of points generated by the code in Example 1.2. The sine wave pattern of these points is evident. If these points were connected by straight lines, the graph would look very much like a sine wave.

Results

The program produces the sequence of points shown in Figure 1.14. The sequence of points clearly varies in a sinusoidal manner; if the points were connected by lines (as done by the plot unless an alternative is requested), the plot would look just like a sine wave. This is examined in Problem 1.5. Note how the MATLAB command that produces the sine wave looks the same as an equivalent mathematical statement: $x = \sin(2\pi ft)$.

1.6.1 Amplitude Slicing

Amplitude slicing, slicing the signal amplitude into discrete levels, is termed *quantization*. The equivalent numerical value of the *quantized* signal can only approximate the level of the analog signal and the degree of approximation will depend on the number of different values that are used to represent the signal. Since digital signals are represented as binary numbers, the bit length of binary numbers used determines the quantization level. The minimum voltage that can be resolved and the amplitude slice size is known as the quantization level, q . The slice size in volts is the voltage range of the converter divided by the number of available discrete values, assuming all bits are converted accurately. If the converter produces a binary number having b accurate bits, then the number of nonzero values is $(2^b - 1)$, where b is the number of bits in the binary output. If the voltage range of the converter varies between 0 and V_{MAX} volts, then the quantization step size, q , in volts, is given as

$$q = \frac{V_{MAX}}{2^b - 1} \text{ V} \quad (1.6)$$

where V_{MAX} is the range of the ADC and b is the number of bits converted.

EXAMPLE 1.3

The specifications (*specs*) of a 12-bit ADC advertise an accuracy of \pm the least significant bit (LSB). If the input range of the ADC is 0–10 V, what is the resolution of the ADC in analog voltage?

Solution

If the input range is 10 V, then the analog voltage represented by the LSB can be found using Equation 1.6:

$$V_{\text{LSB}} = \frac{V_{\text{MAX}}}{(2^{\text{bits}} - 1)} = \frac{10}{(2^{12} - 1)} = \frac{10}{4095} = 0.0024 \text{ V}$$

Hence, the resolution would be ± 0.0024 V.

Typical converters feature 8-, 12-, and 16-bit outputs, although some high-end audio converters use 24 bits. In fact, most signals do not have sufficient signal-to-noise ratio to justify a higher resolution; you are simply obtaining a more accurate conversion of the noise in the signal. For example, given the quantization level of a 12-bit ADC, the dynamic range is $2^{12} - 1 = 4095$; in dB, this is $20 \times \log(4095) = 72$ dB. Since typical signals, especially those of biological origin, have dynamic ranges rarely exceeding 40–50 dB and are often closer to 30 dB, a 12-bit converter with a dynamic range of 72 dB is usually adequate. Microprocessor chips often come with built-in 12-bit ADCs. A 16-bit converter with a theoretical range of 96 dB may be used for greater dynamic range or to provide more *headroom* so that the signal can be comfortably less than the maximum range of the ADC. Alternatively, an 8-bit ADC still offers a dynamic range of 48 dB and may be sufficient, and the converted 8-bit value requires only 1 byte of storage. Note that in most ADCs, the accuracy of the conversion is \pm the LSB; so, the accuracy of real ADCs is based on the number of bits converted minus 1.

The effect of quantization can be viewed as a noise added to the original signal (Figure 1.15). This noise depends on the quantization level, q , in Equation 1.6. If a sufficient number of quantization levels exist (say, >64), the quantization error can be modeled as additive-independent white noise with a uniform distribution between $\pm q/2$ and zero mean. The variance or mean square error of this noise can be determined using the expectation function from basic statistics

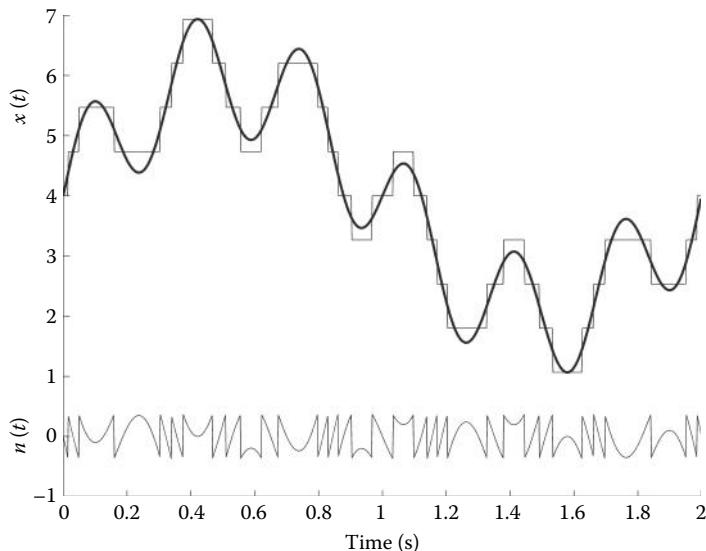


Figure 1.15 The effect of quantization on the original signal can be viewed as noise added to the signal. The amplitude of this noise depends on the quantization level, q , which in turn depends on the number of bytes employed by the ADC. The variance of this noise, which is approximately the same as the RMS value, is given in Equation 1.8.

$$\sigma^2 = \bar{e^2} = \int_{-\infty}^{\infty} e^2 \text{PDF}(e) de \quad (1.7)$$

where $\text{PDF}(e)$ is the uniform probability density function and e is the error voltage (the bottom trace in Figure 1.13). The $\text{PDF}(e)$ for a uniform distribution between $\pm q$ is simply $1/q$. Substituting into Equation 1.7:

$$\sigma^2 = \int_{-q/2}^{q/2} e^2 (1/q) de = \frac{(1/q)e^3}{3} \Big|_{-q/2}^{q/2} = \frac{q^2}{12} \quad (1.8)$$

EXAMPLE 1.4

Write a MATLAB program to evaluate Equation 1.8 through simulation. Generate a 4-Hz sine wave in a 1000-point array ($N = 1000$). Assume a sample interval of $T_s = 0.002$. Quantize this sine wave array into a 6-bit binary number using the routine `quantization.m`.* The calling structure for this routine is

```
signal_out = quantization(signal_in, bits)
```

where `signal_in` is the original signal, `bits` is the number of bits for quantization (`bits = 6` in this case), and `signal_out` is the quantized signal. Find the noise signal by subtracting the original signal from the quantized signal. Plot this signal and take the variance to quantify the noise. Then evaluate the theoretical noise using Equations 1.6 and 1.8 to find the theoretical variance and compare the two results.

Solution

Since the number of points desired is given, it is easier to generate a 1000-point time vector and then multiply it by T_s , that is, $t = (0:999)*Ts$. Then use that vector to generate the desired 4-Hz sine wave signal as in Example 1.2. (Note that almost any waveform and sample interval will work as long as it includes a fair number of points: the sine wave is just a handy waveform.) Quantize the signal into 6 bits using the routine `quantization.m`. Subtract the original signal from the quantized signal and take the variance of the result as the simulated noise variance. Then evaluate Equation 1.6 with `bits = 6` to find the quantization level q , and use that value of q in conjunction with Equation 1.8 to find the theoretical value of noise variance. Compare the two results. Use a formatted output to ensure that the results are displayed in an adequate number of decimal places.

```
% Example 1.4 Evaluate the quantization equation, Eq. 1.8 using simulated
data.
%
f = 4; % Desired frequency
N = 1000; % Number of points
Ts = 0.002; % Ts
bits = 6; % Quantization level
t = (0:N-1)*Ts; % Vector used to generate 1-cycle sine wave
signal_in = sin(2*pi*f*t); % Generate signal
signal_out = quantization(signal_in, bits); % Quantize signal
noise_signal = signal_out - signal_in; % Determine quantization error
```

* The MATLAB routine `quantization.m`, like all auxiliary routines, can be found in this book's website. For clarity, MATLAB variables, files, instructions, and routines are shown in courier typeface throughout the book.

Biosignal and Medical Image Processing

```
q_noise=var(noise_signal); % Variance of quantization noise
q=1/(2^bits - 1);          % Calculate quantization level (Eq. 1.6)
theoretical=(q^2)/12;       % Theoretical quantization error (Eq. 1.8)
disp(' Quantization Noise')
disp('Bits      Emperical      Theoretical')
out=sprintf('%2d %5e %5e', bits, q_noise, theoretical); % Format output
disp(out)
```

Results

The results produced by this program are shown in Table 1.3. The noise variance determined empirically by the program is quite close to the theoretical value determined from Equations 1.6 and 1.8. This evaluation is extended to four different bit values in Problem 1.7.

It is relatively easy and common to convert between the analog and digital domains using electronic circuits specially designed for this purpose. Many medical devices acquire the physiological information as an analog signal and convert it to a digital format using an ADC for subsequent computer processing. For example, the electrical activity produced by the heart can be detected using properly placed electrodes and, after amplification of the resulting signal, the ECG is an analog-encoded signal. This signal might undergo some *preprocessing* or *conditioning* using analog electronics such as those described above. The signal is then converted into a digital signal using an ADC for more complex, computer-based processing and storage. In fact, conversion to digital format would usually be done even if the data are only to be stored for later use. Conversion from the digital to the analog domain is also possible using a *digital-to-analog converter* (DAC). Most personal computers (PCs) include both ADCs and DACs as part of a sound card. This circuitry is specifically designed for the conversion of audio signals, but can be used for other analog signals in some situations. Universal serial bus (USB)-compatible data-transformation devices designed as general-purpose ADCs and DACs are readily available; they offer greater flexibility than sound cards with regard to sampling rates and conversion gains. These cards provide multichannel ADCs (usually 8–16 channels) and several channels of DAC. MATLAB has a toolbox that will interface directly with either a PC sound card or a number of popular converters.

1.6.2 Time Slicing

Slicing the signal into discrete points in time is termed *time sampling* or simply *sampling*. Time slicing samples the continuous waveform, $x(t)$, at discrete points in time, $1T_s, 2T_s, 3T_s, \dots, nT_s$, where T_s is the sample interval. Since the purpose of sampling is to produce an acceptable (for all practical purposes) copy of the original waveform, the critical issue is how well does this copy represent the original? Stated in another way, can the original be reconstructed from the digitized copy? If so, then the copy is clearly adequate. The answer to this question depends on the frequency at which the analog waveform is sampled relative to the frequencies that it contains.

The question of what sampling frequency should be used can best be addressed using a simple waveform, a single sinusoid.* In Chapter 3, we show that all finite, continuous waveforms

Table 1.3 Results from Example 1.2 Comparing the Noise Variance Predicted by Equation 1.10 with Those Determined by Digitizing a Sine Wave Signal and Finding the Digitization Error

Bits	Empirical	Theoretical
6	1.878702e-005	2.099605e-005

* A sinusoid has a straightforward frequency domain representation: it is defined only by a single magnitude and phase (or a single complex point) at the frequency of the sinusoid. The classical methods of frequency analysis described in Chapter 3 make use of this fact.

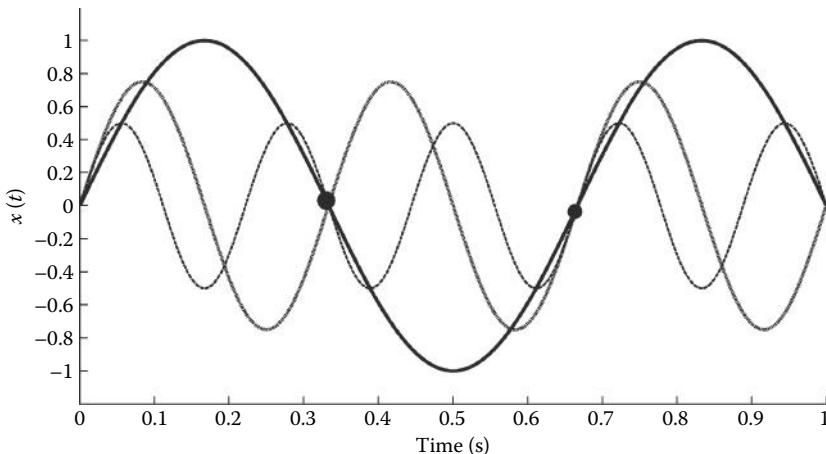


Figure 1.16 A sine wave (solid line) is sampled at two locations within one period. According to the Shannon sampling theorem, there are no other sine waves that pass through these two points at a lower frequency. The lowest-frequency sine wave is uniquely defined. However, there are an infinite number of higher-frequency sine waves that could pass through these two points. Two of these higher-frequency sine waves are shown: a sine wave at double the frequency (dotted line) and one at triple the frequency of the original sine wave.

can be represented by a series of sinusoids (possibly an infinite series); so, if we can determine the appropriate sampling frequency for a single sinusoid, we have also solved the more general problem. The *Shannon sampling theorem* states that any sinusoidal waveform can be uniquely reconstructed provided it is sampled at least twice in one period. (Equally spaced samples are assumed.) That is, the sampling frequency, f_s , must be $>2f_{\text{sinusoid}}$. In other words, only two equally spaced samples are required to uniquely specify a sinusoid and these can be taken anywhere over the cycle. Figure 1.16 shows a sine wave (solid line) defined by two points ("*") covering a time period slightly <1 cycle of the sine wave. According to the Shannon sampling theorem, there are no other sinusoids of a *lower frequency* that can pass through these two points; hence, these two points uniquely define a lowest-frequency sine wave. Unfortunately, there are a lot of *higher-frequency* sine waves that are also defined by these two points, for example, all those points at frequencies that are multiples of the original, two of which are shown in Figure 1.16. If you continue to go up in frequency, there are, in theory, an infinite number of high-frequency sine waves that can be defined by these two points.

Since an infinite number of other higher frequencies can be represented by the two sample points in Figure 1.16, the sampling process can be thought of as actually generating all these additional sinusoids. In other words, you cannot rule out the possibility that these two points also represent all those other sine waves. A more comprehensive picture of the influence of sampling on the original signal can be found by looking at the frequency characteristics of a hypothetical signal before and after sampling. Figure 1.17 presents an example of a signal spectrum before (Figure 1.17a) and after (Figure 1.17b) sampling. As discussed in Chapter 3, an individual point in a spectrum represents a single sinusoidal* waveform; so, before sampling, the signal is a mixture of seven sinusoids at frequencies of 1, 2, 3, 4, 5, 6, and 7 Hz. After sampling, the original spectrum is still there, but in addition, it is now found in many other places in the spectrum, specifically, on either side of the sampling frequency, f_s . Additional sine waves are found reflected about multiples of f_s and added sine waves are even found at negative frequencies. Negative

* A sinusoidal waveform is a sine wave, or cosine wave, or a mixture of the two at the same frequency.

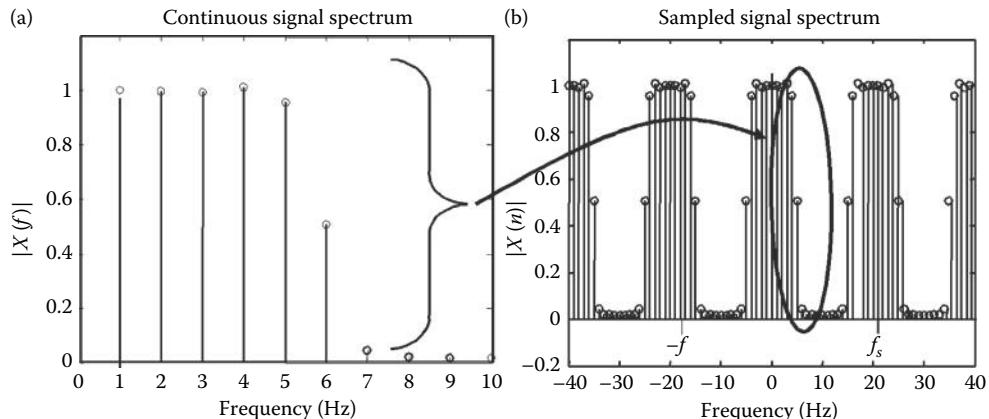


Figure 1.17 The effect of sampling viewed in the frequency domain. (a) An original spectrum before sampling. (b) The spectrum after sampling. The original spectrum has been duplicated and reflected around f_s and at all multiples of f_s including negative frequencies.

frequencies are generated by mathematics that describes the sampling operation. Although they are mathematical constructs, they do have an influence on the sampled signal because they generate the added sine wave reflected to the left of f_s (and to the left of multiples of f_s). Moreover, Figure 1.17b only shows a portion of the sampled signal's spectrum because, theoretically, there are an infinite number of higher frequencies and so the spectrum continues to infinity.

The comparison of frequency characteristics presented in Figure 1.17 clearly demonstrates that the sampled signal is *not* the same as the original, but the critical question is: Can we possibly recover the original signal from its sampled version? This is an extremely important question because if we cannot get an accurate digital version of an analog signal, then digital signal processing is a lost cause. This crucial question is best answered by looking closely at the frequency characteristics of a signal before and after sampling. If we can reconstruct the original unsampled spectrum from the sampled spectrum, then the digital signal is an adequate representation of the original.

Figure 1.18 shows a blowup of just one segment of the spectrum shown in Figure 1.17b, the period between 0 and f_s Hz. Comparing this spectrum to the spectrum of the original signal (Figure 1.17a), we see that the two spectra are the same for the first half of the spectrum up to $f_s/2$ and the second half is just the mirror image (a manifestation of these theoretical negative frequencies). It would appear that we could obtain a frequency spectrum that was identical to the original if we somehow get rid of all frequencies above $f_s/2$. In fact, we can get rid of the frequencies above $f_s/2$ by filtering as described previously. As long as we can get back to the original spectrum, our sampled computer data are a useful reflection of the original data. The frequency $f_s/2$ is so important that it has its own name: the *Nyquist frequency**.

This strategy of just ignoring all frequencies above the Nyquist frequency ($f_s/2$) works well and is the approach that is commonly adopted. But it can only be used if the original signal does not have spectral components at or above $f_s/2$. Consider a situation where four sinusoids with frequencies of 100, 200, 300, and 400 Hz are sampled at a sampling frequency of 1000 Hz. The spectrum produced after sampling actually contains eight frequencies (Figure 1.19a): the four original frequencies plus the four mirror image frequencies reflected about $f_s/2$ (500 Hz). As long as we know, in advance, that the sampled signal does not contain any frequencies above the Nyquist frequency

* Nyquist was one of the most prominent engineers to hone his skills at the former Bell Laboratories during the first half of the twentieth century. He was born in Sweden, but received his education in the United States.

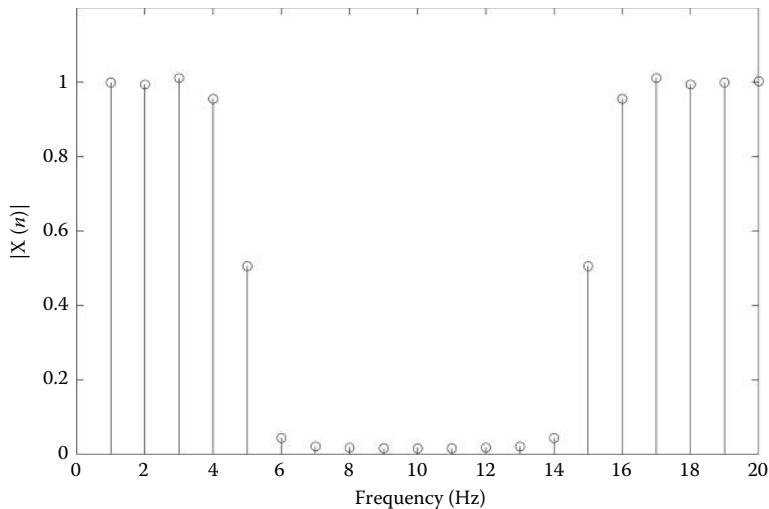


Figure 1.18 A portion of the spectrum of the sampled signal is shown in Figure 1.17b. Note that the added frequencies above $f_s/2$ (10 Hz in this example) are distinct from, and do not overlap, the original frequencies. If we were to eliminate those frequencies with a lowpass filter, we would have the original spectrum back.

(500 Hz), we will not have a problem: we know that the first four frequencies are those of the signal and the second four frequencies, above the Nyquist frequency, are the reflections that can be ignored. However, a problem occurs if the signal contains frequencies higher than the Nyquist frequency. The reflections of these high-frequency components will be reflected back into the *lower* half of the spectrum. This is shown in Figure 1.19b, where the signal now contains two additional

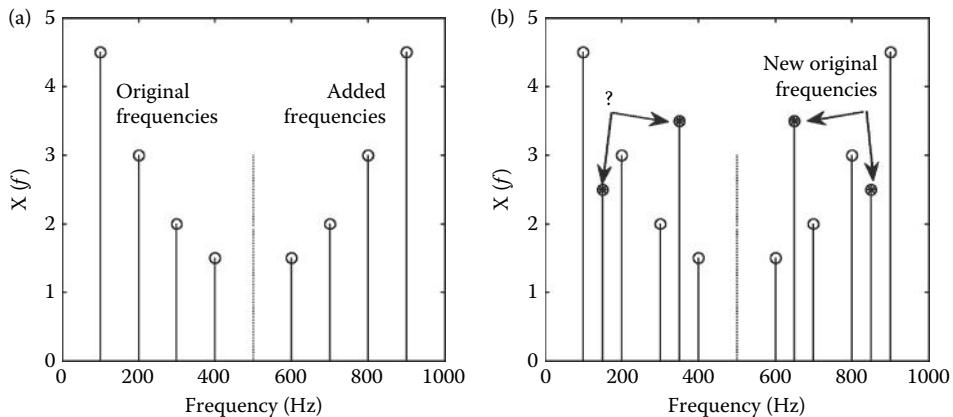


Figure 1.19 (a) Four sine waves between 100 and 400 Hz are sampled at 1 kHz. Only one period of the sampled spectrum is shown, the period between 0 and f_s Hz. Sampling essentially produces new frequencies that are not in the original signal. Because of the periodicity and even symmetry of the sampled spectrum, the additional frequencies are a mirror image reflection around $f_s/2$, the Nyquist frequency. If the frequency components of the sampled signal are all below the Nyquist frequency as shown here, then the upper frequencies do not interfere with the lower spectrum and can be filtered out or simply ignored. (b) If the sampled signal contains frequencies above the Nyquist frequency, they are reflected into the lower half of the spectrum (circles with “*”). It is no longer possible to determine which frequencies belong where, an example of aliasing.

Biosignal and Medical Image Processing

frequencies at 650 and 850 Hz. These frequency components have their *reflections* in the lower half of the spectrum at 350 and 150 Hz, respectively. Now, it is no longer possible to determine if the 350- and 150-Hz signals are part of the true spectrum of the signal (i.e., the spectrum of the signal before it was sampled) or whether these are reflections of signals with frequency components greater than $f_s/2$ (which, in fact, they are). Both halves of the spectrum now contain mixtures of frequencies above and below the Nyquist frequency, and it is impossible to know where they really belong to. This confusing condition is known as *aliasing*. The only way to resolve this ambiguity is to ensure that all frequencies in the original signal are less than the Nyquist frequency.

If the original signal contains frequencies above the Nyquist frequency, then the digital signal in the computer is hopelessly corrupted. Fortunately, the converse is also true. If there are no corrupting frequency components in the original signal (i.e., the signal contains no frequencies above half the sampling frequency), the spectrum in the computer can be altered by filtering to match the original signal spectrum. This leads to a common representation of the famous sampling theorem of Shannon: the original signal can be recovered from a sampled signal provided the sampling frequency is more than twice the *maximum* frequency contained in the original:

$$f_s > 2 f_{\max} \quad (1.9)$$

In practical situations, f_{\max} is taken as the frequency above which negligible energy exists in the analog waveform. The sampling frequency is generally under software control and it is up to the biomedical engineer doing the data acquisition to ensure that f_s is high enough. To make elimination of the unwanted higher frequencies easier, it is common to sample at three to five times f_{\max} . This increases the spacing between the frequencies in the original signal and those generated by the sampling process (Figure 1.20). The temptation to set f_s higher than really necessary is strong, and it is a strategy often pursued. However, excessive sampling frequencies lead to larger data storage and signal-processing requirements that could unnecessarily burden the computer system.

1.6.3 Edge Effects

As mentioned above, the details of truncating a long data set to fit within the computer memory are discussed in Chapter 3, but one obvious consequence is that there are end points. Many signal-processing algorithms work on multiple sequential samples or sample segments, and a problem arises

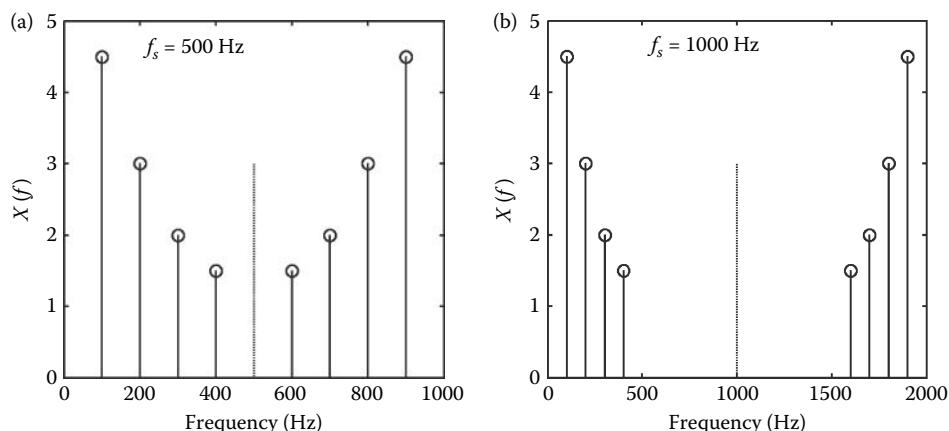


Figure 1.20 The same signal sampled at two different sampling frequencies. (a) A spectrum of a signal consisting of four sinusoids sampled at 500 Hz. (b) The same signal sampled at 1000 Hz. The higher sampling frequency provides greater separation between the original signal spectrum and the spectral components added by sampling.

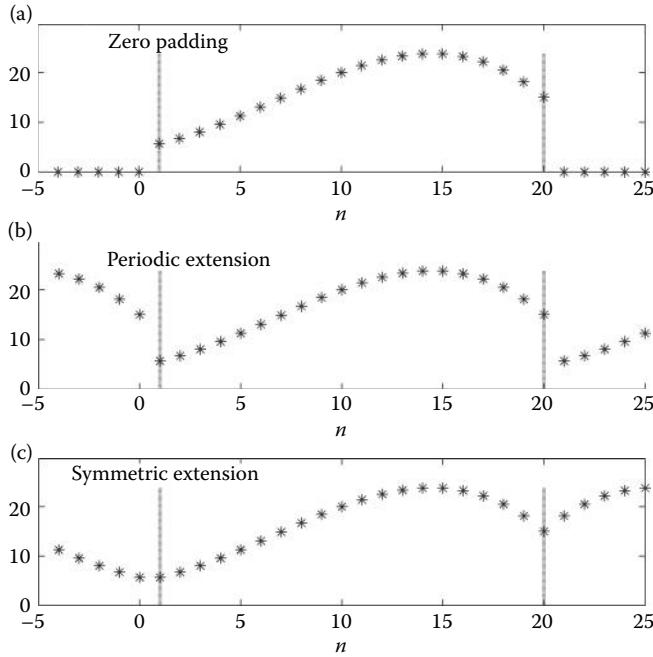


Figure 1.21 Three strategies for extending the length of data: (a) zero padding: zeros are added at the ends of the data set, (b) periodic or wraparound: the waveform is assumed to be periodic; so, the end points are added at the beginning and beginning points are added at the end, and (c) symmetric: points are added at the ends in reverse order. With this last strategy, the edge points may, or may not, be repeated at the beginning and end of the data set.

when an end point is encountered. There are three common strategies for dealing with situations where you come to the end of the data set yet the algorithm needs additional samples: extending with zeros (or a constant) termed *zero padding*, using periodicity or *wraparound*, and extending by reflection also known as *symmetric extension*. These options are illustrated in Figure 1.21.

In the zero-padding approach, zeros are added at the end or beginning of the data sequence (Figure 1.21a). This approach is frequently used in spectral analysis and is justified by the implicit assumption that the waveform is zero outside the sample period anyway. A variant of zero padding is *constant padding* where the data sequence is extended using a constant value, often the last (or first) value in the sequence. If the waveform can be reasonably thought of as 1 cycle of a periodic function, then the *wraparound* approach is justified (Figure 1.21b). Here, the data are extended by tacking on the initial data sequence at the end of the data set and vice versa. These two approaches will, in general, produce a discontinuity at the beginning or end of the data set; this can lead to an artifact for some algorithms. The symmetric-reflection approach eliminates this discontinuity by tacking on the end points in reverse order (or beginning points if the extending is done at the beginning of the data sequence) (Figure 1.21c).*

To reduce the number of points in cases in which an operation has generated additional data, two strategies are common: simply eliminate the additional points at the end of the data set, or eliminate data from both ends of the data set, usually symmetrically. The latter is used when the

* When using this extension, there is a question as to whether or not to repeat the last point in the extension: either strategy will produce a smooth extension. The answer to this question will depend on the type of operation being performed and the number of data points involved; determining the best approach may require empirical evaluation.

Biosignal and Medical Image Processing

data are considered periodic, and it is desired to retain the same timing or when other similar concerns are involved. An example of this strategy is found in periodic convolution often used in wavelet analysis as described in Chapter 7.

1.6.4 Buffering and Real-Time Data Processing

Real-time data processing simply means that the data are processed and results are obtained in sufficient time to influence some ongoing process. This influence may come directly from the computer or through human intervention. The processing time constraints naturally depend on the dynamics of the process of interest. Several minutes might be acceptable for an automated drug delivery system, while information on the electrical activity of the heart usually needs to be immediately available.

The term *buffer*, when applied to digital technology, usually describes a set of memory locations used temporarily to store incoming data until enough data are acquired for efficient processing. When data are being acquired continuously, a technique called *double buffering* can be used. Incoming data are alternatively sent to one of the two memory arrays and the one that is not being filled is processed (that may simply involve transfer to disk storage). Most ADC software packages provide a means for determining which element in an array has most recently been filled, to facilitate buffering. They frequently also have the ability to determine which of the two arrays (or which half of a single array) is being filled to facilitate double buffering.

1.7 Data Banks

With the advent of the World Wide Web, it is not always necessary to go through the ADC process to obtain digitized data of physiological signals. A number of *data banks* exist that provide physiological signals such as ECG, EEG, gait, and other common biosignals in digital form. Given the volatility and growth of the web and the ease with which searches can be made, no attempt is made here to provide a comprehensive list of appropriate websites. However, a good source of several common biosignals, particularly the ECG, is the “Physio Net Data Bank” maintained by MIT: <http://www.physionet.org>. Some data banks are specific to a given set of biosignals or a given signal-processing approach. An example of the latter is the ICALAB Data Bank in Japan, <http://www.bsp.brain.riken.go.jp/ICALAB/>, which includes data that can be used to evaluate independent component analysis (see Chapter 9) algorithms.

Numerous other data banks containing biosignals and/or images can be found through a quick search on the web and many more are likely to come online in the coming years. This is also true for some of the signal-processing algorithms we describe in more detail later. For example, the ICALAB website mentioned above also has algorithms for independent component analysis in MATLAB’s *m*-file format. A quick web search can provide both signal-processing algorithms and data that can be used to evaluate a signal-processing system under development. The web is becoming an even more useful tool in signal and image processing, and a brief search on the web can save considerable time in the development process, particularly if the signal-processing system involves advanced approaches.

1.8 Summary

From a traditional reductionist viewpoint, living things are described in terms of component systems. Many traditional physiological systems such as the cardiovascular, endocrine, and nervous systems are quite extensive and are composed of many smaller subsystems. Biosignals provide communication between systems and subsystems, and are our primary source of information on the behavior of these systems. Interpretation and transformation of signals are a major focus of this chapter.

Biosignals, like all signals, must be “carried” by some form of energy. The common biological energy sources include chemical, mechanical, and electrical energy (in the body, electrical signals are carried by ions, not electrons). In many diagnostic approaches, an external energy source is used to probe the body, as in ultrasound and CT scanners. Signals can be encoded in a number of formats, but most biosignals are generally analog in nature: they vary continuously over time and the desired information is embedded in the amplitude or instantaneous value of the signal.

Measuring biosignals involves complex multicomponent systems, but the lead element is the biotransducer, which provides the interface between the living system and the measurement system. It converts biological energy (or external energy in some systems) into an electric signal compatible with analog and digital processing. The biotransducer usually sets the noise level of the measurement and often limits the ultimate utility of the system. For these reasons, the biotransducer can be the most critical component of a measurement system. Since most signal processing is performed in the digital domain, measurement systems usually include an ADC to transform the continuously time-varying biosignals into a sequence of numbers suitable for digital processing. Analog signal processing is common before conversion, including amplification (boosting the signal values) and filtering. The latter is used to reduce noise and, as shown in Chapter 3, to ensure accurate conversion to digital format. Filters vary in function, but most filters can be fully defined by their frequency characteristics or spectrum. Specifically, three frequency attributes describe the important features of a filter: the filter type (lowpass, highpass, bandpass, and bandstop), attenuation slope, and initial sharpness. The frequency characteristics of filters (and other analog system components) are often displayed as a plot in dB versus log frequency since these log-log-type plots are often easier to interpret.

The conversion of analog signals to digital (or discrete time) signals requires slicing the continuous signal into discrete levels of both amplitude and time. Amplitude slicing adds noise to the signal; the amount of noise added is dependent on the size of the slice. Typically, the slices are so small that the noise from amplitude slicing is usually ignored. Not so for time slicing, which produces such a complicated transformation that it adds additional frequencies to the digitized version of the signal. While the digitized signal is quite different in spectral content from the original analog signal, the added frequencies will all be above those in the original, provided the sampling frequency is greater than twice the *maximum* frequency in the original signal. This rule is known as Shannon’s sampling theorem.

In addition to time and amplitude slicing, many real-world signals must be truncated if they are to be stored in computer memory. The details of truncation or data windowing are discussed in Chapter 3, but one consequence of finite data is often the need for a strategy for dealing with the end points. Three popular methods exist: zero padding (or constant padding) that adds zeros at the end, periodic extension which wraps data around using beginning points at the end and ending points at the beginning, and symmetric extension that reflects the last few points around the end point in a symmetric manner.

The multitudinous advantages of the World Wide Web extend to signal processors as well; many resources including algorithms and data banks are available and continually expanding.

PROBLEMS

- 1.1 A lowpass filter is desired with a cutoff frequency of 10 Hz. This filter should attenuate a 100-Hz signal by a factor of at least 78. What should be the order of this filter?
- 1.2 Since sine waves have energy at only one frequency (see Chapter 3), they can be used to probe the frequency characteristics of a filter (or other linear systems). Here, sine waves at several different frequencies are used as input to an analog filter. The input

Biosignal and Medical Image Processing

sine waves have a value of 1.0 V root mean square (RMS) and the output measured at each frequency is given below:

Frequency (Hz)	2	10	20	60	100	125	150	200	300	400
V_{out} RMS (V)	0.15×10^{-7}	0.1×10^{-3}	0.002	0.2	1.5	3.28	4.47	4.97	4.99	5.0

Use MATLAB to plot the filter's frequency characteristic. (As in all plots, label the axes correctly.) What type of filter is it? What is the cutoff frequency and order of this filter? Use the MATLAB grid function (`grid on;`) to help find the slope and cutoff frequency from the plot. [Hint: Since sine waves have energy at only one frequency, the frequency characteristics of the filter can be determined from the ratio of output to input (Equation 1.2). Take $20 \log(V_{\text{out}}/V_{\text{in}})$ where $V_{\text{in}} = 1.0$ to get the gain in dB and use MATLAB's `semilogx` to get a plot gain against log frequency. This will allow you to estimate the spectral slope in dB/decade better.]

- 1.3. The MATLAB routine `analog_filter1.m` found in the support material associated with this chapter is a simulated analog filter. (Calling structure: `output = analog_filter1(input);`). Repeat the strategy used in Problem 1.2, but generate the input sine waves in MATLAB and plot the output. Make $T_s = 0.001$ s and the number of points $N = 1000$. To generate sine waves, define the time vector $t = (0:999)*Ts$ as in Example 1.3. Use this vector to generate the input sine waves at various frequencies: `input = sin(2*pi*f*t);`. Use frequencies of 2, 10, 15, 20, 30, 40, 50, 60, 80, 90, 100, 150, 200, 300, and 400 Hz.

Find the amplitude of the output of `analog_filter1.m` using MATLAB's `max` operator and plot the resulting values in dB versus log frequency. This gives an approximation of the filter's frequency characteristics or spectrum. Use this spectrum with the grid function enabled to determine the type, bandwidth, and attenuation slope of the filter. [Hint: Put the desired input frequencies in an array and use a for-loop to generate the input sine waves. Store the maximum values of the filter's output in an array for plotting. Plot the 20 log of the output values against the frequency array using the `semilogx` routine.]

- 1.4 Repeat Problem 1.3 but use the routine `analog_filter2.m` for the filter (same calling structure). In this problem, you are to select the frequencies. Use the minimum number of frequencies that allow you to accurately measure the filter's cutoff frequency and slope. Limit frequencies to be between 2 and 400 Hz.
- 1.5 Generate a discrete 2-Hz sine wave using MATLAB as in Example 1.2. Use sample intervals of 0.05, 0.01, and 0.001 s and again use enough points to make the sine wave 1-s long, that is, the total time, T_T , should be 1 s. Plot with lines connecting the points and on the same plot, the individual points superimposed on the curves as in Example 1.2. Plot the sine waves using the three sample intervals separately, but use `subplot` to keep the three plots together. Note that even the plot with very few points looks sinusoidal. Chapter 3 discusses the minimum number of points required to accurately represent a signal.
- 1.6 Write a MATLAB problem to test Equation 1.6 through simulation. Generate a 4-Hz, 1000-point sine wave as in Example 1.4 assuming a sample interval of $T_s = 0.002$. Use `quantization.m` to digitize it using 4-, 8-, 12-, and 16-bit ADC. Then, as in Example 1.4, subtract the original signal from the quantized signal to find the error signal. The amplitude of the error signal should be equal to the quantization level, q .

in Equation 1.6. Use MATLAB's `max` and `min` functions to find this amplitude and compare it with the value predicted by Equation 1.6. Put this code in a for-loop and repeat the evaluations for the four different bit levels requested. Be sure to display the results to at least four decimal places to make an accurate comparison. [Hint: Most of the code needed for this problem will be similar to that in Example 1.4.]

- 1.7 Extend the code in Example 1.4 to include quantization levels involving 4, 8, 10, and 12 bits. Compare the theoretical and simulated noise for these four different quantization levels. Present the output in a format with sufficient resolution to illustrate the small differences between the simulated and theoretical noise.
- 1.8 Write a MATLAB program to generate 1 s of a 5-Hz sine wave in a 1000-point array. (Use Equation 1.4 to determine the equivalent T_s .) Plot the sine wave. Simulate the sampling of this waveform at 7 Hz by plotting a point (such as an “*”) at intervals of $T_s = 1/f_s = 1/7$ s. (Use a for-loop to generate the sample time, T_s , insert it into the equation for the 5-Hz sine wave, and plot the resulting point.) Aliasing predicts that these sampled points should fall on a 2-Hz (7–5 Hz) sine wave. Can you find a 2-Hz sine wave that includes all seven points? [Hint: The sine wave could be phase shifted by 180°.]
- 1.9 Repeat Problem 1.8 using a simulated sample interval of 9 Hz ($T_s = 1/f_s = 1/9$ s) and find the appropriate sine wave produced by aliasing.

2

Biosignal Measurements, Noise, and Analysis

This chapter presents four major topics that permeate signal processing: the nature of biosignals, their basic measurements, the characteristics of noise, and an overview of signal-processing methodology. The discussion of signal processing introduces the concept of transforms and presents fundamental equations that underlie analytical techniques developed in four subsequent chapters.

2.1 Biosignals

This book concerns digital signals, but we often think about these signals in terms of their analog equivalents. Most of the concepts presented apply equally well to analog or digital signals, the analytical operations performed on both are similar, and there is a correspondence between the equations for these similar digital and analog operations. Often both equations are presented to emphasize this similarity. Since most of the examples and problems are carried out in MATLAB on a computer, they are implemented in the digital domain even if they are phrased as analog domain problems.

Digital signals can be classified into two major subgroups: *linear* and *nonlinear* (Figure 2.1). Linear signals derive from linear processes while nonlinear signals arise from nonlinear processes such as chaotic or turbulent systems. Each of these two signal classes can be further divided into *stationary* and *nonstationary* signals. Stationary signals have consistent statistical properties over the data set being analyzed. There is a formal mathematical definition of stationarity based on the signal's probability distribution function, but basically stationary signals have constant means, standard deviations, and the same average correlations between data points. It is important to know which type of signal you are dealing with, as the analytical tools that apply are quite different. Figure 2.1 indicates the chapters that cover signal-processing tools appropriate to the various signal types.

Much of this book is devoted to signals that are linear and stationary since an extensive set of powerful signal-processing tools has been developed for these signals. While biological signals are often nonlinear and nonstationary, these tools are so useful that assumptions or approximations are made so that linear techniques can be applied. Linearity can be approximated by using small-signal conditions where many systems behave more-or-less linearly. Alternatively, piecewise linear approaches can be used where the analysis is confined to operating ranges over which the system behaves linearly. For nonstationary signals that have a varying mean, *detrending* techniques that subtract out the time-varying mean can be used. A popular approach

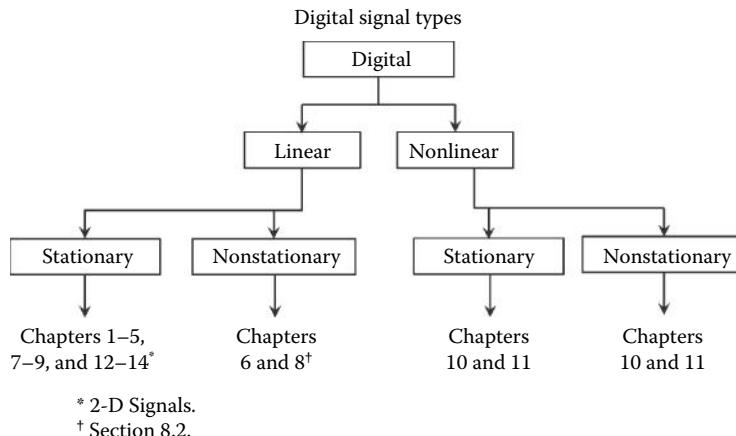


Figure 2.1 Classification of digital signals. The chapters that cover the different signal types are shown.

to dealing with nonstationary signals is to study the signal within a time frame or series of time frames that are short enough so that the signal can be considered stationary during that period. When an assumption or approximation of linearity is clearly untenable, several nonlinear signal-processing tools are available as described in Chapters 10 and 11. These nonlinear signals are often assumed to be stationary as their analysis requires long data sets. For this reason, nonlinear signal processing is particularly challenging for nonstationary signals.

2.1.1 Signal Encoding

Concepts covered in this book are applied to signals that already exist in digital format; the assumption is that they have been detected by a biotransducer, appropriately preprocessed, and correctly converted to digital signals (see Section 1.6). All signals involve some type of encoding scheme. Encoding schemes vary in complexity: human speech is so complex that decoding can still challenge voice recognition computer programs. Yet the same information can be encoded into a simple series of long and short pulses known as the Morse code, easily decoded by a computer.

Most encoding strategies can be divided into two broad categories or domains: continuous and discrete. The discrete domain is used almost exclusively in computer-based technology, as such signals are easier to manipulate electronically. Discrete signals are usually transmitted as a series of pulses at even (synchronous transmission) or uneven (asynchronous transmission) intervals. These pulses may be of equal duration, or the information can be encoded into the pulse length. Within the digital domain, many different encoding schemes can be used. For encoding alphanumeric characters, those featured on the keys of a computer keyboard, the ASCII code is used. Here each letter, the numbers 0 through 9, and many other characters are encoded into an 8-bit binary number. For example, the letters a through z are encoded as 97 (for a) through 122 (for z) while the capital letters A through Z are encoded by numbers 65 through 90. The complete ASCII code can be found on numerous Internet sites.

In the continuous domain, information is encoded in terms of signal amplitude, usually the intensity of the signal at any given time. For an *analog* electronic signal, this could be the value of the voltage or current at a given time. Note that all signals are by nature *time varying*, since a single constant value contains no information.* For example, the temperature in a room can

* Modern information theory makes explicit the difference between information and meaning. The latter depends upon the receiver, that is, the device or person for which the information is intended. Many students have attended lectures with a considerable amount of information that, for them, had little meaning. This book strives valiantly for information with the expectation that it will also have meaning.

be encoded so that 0 V represents 0.0°C, 5 V represents 10°C, 10 V represents 20°C, and so on. Assuming a *linear* relationship, the encoding equation for the transducer would be:

$$\text{Voltage amplitude} = \text{temperature}/2 \text{ V} \quad (2.1)$$

This equation relating the input (temperature) to the output (voltage) follows the classic linear relationship:

$$y = mx + b \quad (2.2)$$

where m is the slope of the input–output relationship and b is the offset which in this case is 0.0. The temperature can be found from the voltage output of the transducer as

$$\text{Temperature} = 2^* \text{ voltage } ^\circ\text{C} \quad (2.3)$$

Analog encoding was common in consumer electronics such as early high-fidelity amplifiers and television receivers, although most of these applications now use discrete or digital encoding. (Vinyl records that are still in vogue among some audiophiles are notable exceptions.) Nonetheless, analog encoding is likely to remain important to the biomedical engineer, if only because many physiological systems use analog encoding, and most biotransducers generate analog encoded signals.

When a continuous analog signal is converted into the digital domain, it is represented by a series of numbers that are discrete samples of the analog signals at specific points in time (see Section 1.6.2):

$$X[n] = x[1], x[2], x[3], \dots, x[n] \quad (2.4)$$

Usually this series of numbers would be stored in sequential memory locations with $x[1]$ followed by $x[2]$, then $x[3]$, and so on. In this case, the memory index number, n , relates to the time associated with a given sample. Converting back to relative time is achieved by multiplying the index number, n , by the sampling interval (Equation 1.5), repeated here:

$$t = nTs = \frac{n}{fs} \quad (2.5)$$

Recall that it is common to use brackets to identify a discrete variable (i.e., $x[n]$), and standard parentheses to notate an analog variable (i.e., $x(t)$). The MATLAB programming also uses brackets, but in a somewhat different context: to define a series of numbers (e.g., $\mathbf{x} = [2 \ 4 \ 6 \ 8]$).

2.1.2 Signal Linearity, Time Invariance, Causality

In the example of a transducer defined by Equation 2.1, the relationship between input temperature signal and output voltage signal was assumed to be linear. The concept of linearity has a rigorous definition, but the basic concept is one of proportionality. If you double the input into a linear system, you will double the output. One way of stating this proportionality property mathematically is: if the independent variables of linear function are multiplied by a constant, k , the output of the function is simply multiplied by k :

Assume $y = f(x)$ where f is a linear function. Then

$$ky = f(kx) \quad (2.6)$$

Also, if f is a linear function:

$$f(x_1(t)) + f(x_2(t)) = f(x_1(t) + x_2(t)) \quad (2.7)$$

In addition, if $y = f(x)$ and $z = df(x)/dx$, then

$$\frac{df(kx)}{dx} = k \left(\frac{df(x)}{dx} \right) = kz \quad (2.8)$$

Similarly, if $y = f(x)$ and $z = \int f dx$, then

$$\int f(kx) dx = k \int f(x) dx = kz \quad (2.9)$$

So, the derivation and integration are linear operations. Systems that contain derivative and integral operators along with other linear operators (a wide class of systems) produce linear signals. When these operators are applied to linear signals, the signals preserve their linearity.

If a system has the same behavior at any given time, that is, its basic response characteristics do not change over time, it is said to be *time invariant*.^{*} Time invariance could be thought of as a stricter version of stationarity since a time-invariant system would also be stationary. The mathematical definition of a time-invariant function, f , is given as

$y = f(x)$ where f is a linear function, then for time invariance:

$$y(t - T) = f(x(t - T)) \quad (2.10)$$

If a system is both linear and time invariant, it is sometimes referred to as a *linear time-invariant (LTI)* system. The LTI assumptions allow us to apply a powerful array of mathematical tools known collectively as linear systems analysis or linear signal analysis. Of course, most living systems change over time, they are adaptive, and they are often nonlinear. Nonetheless, the power of linear systems analysis is sufficiently seductive that assumptions or approximations are made so that these tools can be used. We sometimes impose rather severe simplifying constraints to be able to use these tools.

If a system responds only to current and past inputs, the system is termed *causal*. Real-world systems are causal and you might think that all systems must be causal; How can a system possibly respond to a future stimulus? However, if a signal is stored in computer memory, operations that act over a given time period can act on both past and future values of a signal. For example, a digital filter could adjust its output at any given sample time based on data points both before and after that sample time. So digital filters can be *noncausal* but analog filters which do not have access to future inputs are causal.

2.1.2.1 Superposition

Response proportionality, or linearity, is required for the application of an important concept known as *superposition*. Superposition states that if there are two (or more) stimuli acting on the system, the system responds to each as if it were the only stimulus present. The combined influence of the multiple stimuli is just the addition of each stimulus acting alone. This allows a “divide and conquer” approach, in which complex stimuli can be broken down into simpler components and an input–output analysis performed on each component as if the others did not exist. The actual output is the sum of all the outputs to the individual components. This approach, combined with other analytical tools that rely on time invariance, can greatly simplify and enhance the value of signal analysis. Approaches that rely on superposition and the LTI assumption are found throughout this book.

* Note the difference in meaning between “time-invariant” and “time-varying.” A time-varying signal may or may not be time invariant. Time invariant relates to a signal’s properties, while time varying describes the moment-to-moment fluctuation of signals.

2.1.3 Signal Basic Measurements

One of the most straightforward of signal measurements is the assessment of its *mean* or average value. To determine the average of a digital signal, simply add the individual values and divide by N , the number of samples in the signal:

$$x_{avg} = \bar{x} = \frac{1}{N} \sum_{n=1}^N x_n \quad (2.11)$$

where n is an index number indicating the sample number, the specific position of a number in the series.* While we will be working only with digital signals, there is an equivalent equation that is used to evaluate the mean of continuous time-varying signals, $x(t)$:

$$\bar{x}(t) = \frac{1}{T} \int_0^T x(t) dt \quad (2.12)$$

Comparing Equations 2.11 and 2.12 shows some common differences between digital and analog domain equations: the conversion of summation into integration and the use of a continuous variable, t , in place of the discrete integer, n . These conversion relationships are generally applicable, and most digital domain equations can be transferred to continuous or analog equations in this manner and vice versa.

Although the mean is a basic property of a signal, it does not provide any information about the variability of the signal. The root mean-squared (RMS) value is a measurement that includes both the signal's variability and its average. Obtaining the RMS value of a signal is just a matter of following the measurement's acronym: first squaring the signal, then taking its mean, and finally taking the square root of the mean:

$$x_{rms} = \left[\frac{1}{N} \sum_{n=1}^N x_n^2 \right]^{1/2} \quad (2.13)$$

The continuous form of the equation is obtained by following the rules described above.

$$x(t)_{rms} = \left[\frac{1}{T} \int_0^T x(t)^2 dt \right]^{1/2} \quad (2.14)$$

These equations describe discrete (Equation 2.13) and continuous (Equation 2.14) versions of the RMS operation and also reflect the two approaches required for solution. Continuous equations require *analytical* solutions implemented manually (i.e., paper and pencil), while discrete equations could be done manually but are almost always implemented by the computer. Example 2.1 uses both the analytical and digital approach to find the RMS of a sinusoidal signal.

EXAMPLE 2.1

Find the RMS value of the sinusoidal signal using both analytical (Equation 2.14) and digital (Equation 2.13) approaches.

Solution, Analytical

Since this signal is periodic, with each period the same as the one before it, it is sufficient to apply the RMS equation over a single period. This is true for all operations on sinusoids. Neither the

* The bar over the x in Equation 2.11 stands for “the average of ...” In this book, the letters k and i are also used to indicate sample number. In some mathematical formulations, it is common to let the summation index n range from 0 to $N-1$, but in MATLAB the index must be nonzero, so here most summations range between 1 and N .

Biosignal and Medical Image Processing

RMS value nor anything else about the signal will change from one period to the next. Applying Equation 2.14:

$$\begin{aligned}
 \bar{x}(t)_{rms} &= \left[\frac{1}{T} \int_0^T x(t)^2 dt \right]^{1/2} = \left[\frac{1}{T} \int_0^T \left(A \sin\left(\frac{2\pi t}{T}\right) \right)^2 dt \right]^{1/2} \\
 &= \left[\frac{1}{T} \frac{A^2}{2\pi} \left(-\cos\left(\frac{2\pi t}{T}\right) \sin\left(\frac{2\pi t}{T}\right) + \frac{\pi t}{T} \right) \Big|_0^T \right]^{1/2} \\
 &= \left[\frac{A^2}{2\pi} (-\cos(2\pi)\sin(2\pi) + \pi + \cos 0 \sin 0) \right]^{1/2} \\
 &= \left[\frac{A^2 \pi}{2\pi} \right]^{1/2} = \left[\frac{A^2}{2} \right]^{1/2} = \frac{A}{\sqrt{2}} = 0.707A
 \end{aligned}$$

Solution, Digital

Generate a 1-cycle sine wave. Assume a sample interval of $T_s = 0.005$ s and make $N = 500$ points long (both arbitrary). So solving Equation 1.4 for the total time: $T = NT_s = 0.005(500) = 2.5$ s. To generate a single cycle given these parameters, the frequency of the sine wave should be $f = 1/T = 1/2.5 = 0.4$ Hz. Set the amplitude of the sine wave, $A = 1.0$.

```

N = 500; % Number of points for waveform
Ts = .005; % Sample interval = 5 msec
t = (1:N)*Ts; % Generate time vector (t = N Ts)
f = 1/(Ts*N); % Sine wave freq. for 1 cycle
A = 1; % Sine wave amplitude
x = A*sin(2*pi*f*t); % Generate sine wave
RMS = sqrt(mean(x.^2)); % Take the RMS value and output.

```

Results

The value produced by the MATLAB program is 0.7071, which is very close to the theoretical value of $1/\sqrt{2}$ (rounded here to 0.707) multiplied by the amplitude of the sine wave, A . Hence there is a proportional relationship between the peak amplitude of a sinusoid (± 1.0 in this example) and its RMS value. This is only true for sinusoids. For other waveforms, Equation 2.13 (or Equation 2.14 for continuous functions) must be used to find the RMS value.

A statistical measure related to the RMS value is the *variance*, σ^2 . The variance is a measure of signal variability irrespective of its average. The calculation of variance for both discrete and continuous signals is given as

$$\sigma^2 = \frac{1}{N-1} \sum_{n=1}^N (x_n - \bar{x})^2 \quad (2.15)$$

$$\sigma^2 = \frac{1}{T} \int_0^T (x(t) - \bar{x})^2 dt \quad (2.16)$$

where \bar{x} is the mean or signal average (Equation 2.13 or 2.14). In statistics, the variance is defined in terms of an estimator known as the *expectation* operation and is applied to the probability

distribution function of the data. Since the distribution of a real signal is rarely known in advance, the equations given here are used to calculate variance in practical situations.

The *standard deviation* is another measure of a signal's variability and is simply the square root of the variance:

$$\sigma = \left[\frac{1}{N-1} \sum_{n=1}^N (x_n - \bar{x})^2 \right]^{1/2} \quad (2.17)$$

$$\sigma = \left[\frac{1}{T} \int_0^T (x(t) - \bar{x})^2 dt \right]^{1/2} \quad (2.18)$$

In determining the standard deviation and variance from discrete or digital data, it is common to normalize by $1/(N - 1)$ rather than $1/N$. This is because the former gives a better estimate of the actual standard deviation or variance when the data are samples of a larger data set that has a Gaussian distribution. Signals rarely have a Gaussian distribution but, as shown below, noise often is Gaussian. If the data have zero mean, the standard deviation is nearly the same as the RMS value except for the normalization factor in the digital calculation. (Problem 2.7 explores this similarity on a typical waveform.) Nonetheless they come from very different traditions (statistics versus measurement) and are used to describe conceptually different aspects of a signal: signal magnitude for RMS and signal variability for standard deviation.

Calculating the mean, variance, or standard deviation of a signal using MATLAB is straightforward. Assuming a signal vector x , these measurements are determined using one of the code lines below.

```
xm = mean(x); % Evaluate mean of x
xvar = var(x); % Variance of x normalizing by N-1
xstd = std(x); % Evaluate the standard deviation of x,
```

If x is a matrix instead of a vector, then the output is a row vector resulting from application of the calculation (mean, variance, or standard deviation) to each column of the matrix. The next example uses measurement of mean and variance to determine if a signal is stationary.

EXAMPLE 2.2

Evaluate a signal to determine if it is stationary. If not, attempt to modify the signal to remove the nonstationarity, if possible. The file `data_c1.mat` contains the signal in variable x . The signal is 1000 points long and the sample interval is $T_s = 0.001$ s.

Solution, Part 1

In Part 1 of this example, we want to determine if the signal is stationary. If it is not, we will modify the signal in Part 2 to make it approximately stationary. There are a number of approaches we could try, but one straightforward method is to segment the data and evaluate the mean and variance of the segments. If they are the same for all segments, the signal is probably stationary. If these measures change segment-to-segment, the signal is clearly nonstationary.

Biosignal and Medical Image Processing

```
% Example 2.2, Part 1. Evaluate a signal for stationarity.  
load data_c1; % Load the data. Signal in variable x  
for k=1:4 % Segment signal into 4 segments  
    m=250*(k-1)+1; % Index of first segment sample  
    segment=x(m:m+249); % Extract segment  
    avg(k)=mean(segment); % Evaluate segment mean  
    variance(k)=var(segment); % and segment variance  
end  
disp('Mean Segment 1 Segment 2 Segment 3 Segment 4') % Display heading  
disp(avg) % Output means  
disp('Variance Segment 1 Segment 2 Segment 3 Segment 4') % Heading  
disp(variance) % Output variance
```

Results, Part 1

The output of this program is shown in Table 2.1. The signal is clearly nonstationary since the mean is different in every segment, although the variance seems to be consistent (allowing for statistical fluctuations).

Solution, Part 2

One trick is to transform the signal by taking the derivative: the difference between each point. Since only the mean is changing, another approach is to *detrend* the data; that is, subtract out and estimation of the changing mean. If the change in mean is linear, MATLAB's *detrend* operator, which subtracts the best-fit straight line, could make the data linear. More complicated procedures could be developed to remove variations in mean that were nonlinear. The first approach is used in this example and the second approach is assessed in Problem 2.8.

```
% Example 2.8, Part 2. Modify a nonstationary signal to become stationary  
%  
y=[diff(x),0]; % Take difference between points.  
for k=1:4 % Segment signal into 4 segments  
    m=250*(k-1)+1; % Index of first segment sample  
    segment=y(m:m+249); % Extract segment  
    avg(k)=mean(segment); % Evaluate segment mean  
    variance(k)=var(segment); % and segment variance  
end  
..... Output avg and variance as above .....
```

Results, Part 2

The *diff* operator produces an output that is the difference between sequential data points, but the output signal is necessarily one point shorter than the input. To make the output the same length as the input, the program adds a 0 at the end of the signal. The trick of adding zeros to a signal to extend its length is a well-established approach and is discussed in detail in Chapter 3. The output of this code is shown in Table 2.2. The means of all 4 segments are now

Table 2.1 Results from Example 2.2, Part 1

	Segment 1	Segment 2	Segment 3	Segment 4
Mean	0.0640	0.4594	0.6047	0.8095
Variance	0.1324	0.1089	0.0978	0.1021

Table 2.2 Results of the Analysis in Example 2.2 after Differencing the Signal

	Segment 1	Segment 2	Segment 3	Segment 4
Mean	0.0020	-0.0008	0.0004	0.0004
Variance	0.0061	0.0057	0.0071	0.0066

approximately the same; in fact, they are near zero. The variances are roughly the same, but slightly less than the original variances, not surprising as the changing mean might be expected to add to the variance.

2.1.4 Decibels

It is common to compare the intensity of two signals using ratios, $V_{\text{Sig}1}/V_{\text{Sig}2}$, particularly if the two are the signal and noise components of a waveform (see next section). Moreover, these signal ratios are commonly represented in units of *decibels*. Actually, decibels (dB) are not really units, but are simply a logarithmic scaling of a dimensionless ratio. The decibel has several advantageous features: (1) the log operation compresses the range of values (e.g., a range of 1–1000 becomes a range of 1–3 in log units); (2) when numbers or ratios are to be multiplied, they are simply added if they are in log units; and (3) the logarithmic characteristic is similar to human perception. It was this latter feature that motivated Alexander Graham Bell to develop the logarithmic unit called the *Bel*. Audio power increments in logarithmic Bels were perceived as equal increments by the human ear. The Bel turned out to be inconveniently large, so it has been replaced by the decibel: $\text{dB} = 1/10 \text{ Bel}$. While originally defined only in terms of a ratio, dB units are also used to express the power or intensity of a single signal.

When applied to a power measurement, the decibel is defined as 10 times the log of the power ratio:

$$P_{\text{dB}} = 10 \log\left(\frac{P_2}{P_1}\right) \text{dB} \quad (2.19)$$

Power is usually proportional to the square of voltage. So when a signal voltage, or voltage ratio, is being converted into dB, Equation 2.19 has a constant multiplier of 20 instead of 10 since $\log x^2 = 2 \log x$:

$$V_{\text{dB}} = 10 \log\left(\frac{V_{\text{sig}1}}{V_{\text{sig}2}}\right)^2 = 20 \log\left(\frac{V_{\text{sig}1}}{V_{\text{sig}2}}\right) \text{ for a ratio of signals} \quad (2.20)$$

or

$$V_{\text{dB}} = 10 \log(V_{\text{sig}})^2 = 20 \log(V_{\text{sig}}) \text{ for a single signal} \quad (2.21)$$

If a ratio of signals is involved, then decibels are dimensionless, otherwise they have the dimensions that are used to define the signal intensity (i.e., volts, dynes, etc.).

2.1.5 Signal-to-Noise Ratio

Most waveforms consist of signal plus noise mixed together. As noted previously, signal and noise are relative terms, relative to the task at hand: the signal is that portion of the waveform you are interested in while the noise is everything else. Often the goal of signal processing is to separate out signal from noise, or to identify the presence of a signal buried in noise, or to detect features of a signal buried in noise.

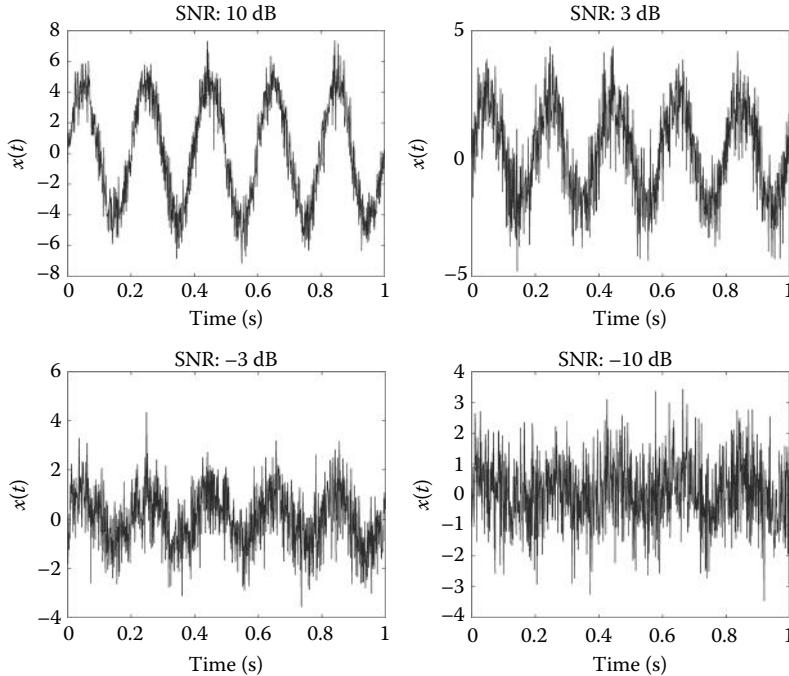


Figure 2.2 A 5-Hz sine wave with varying amounts of added noise. The sine wave is barely discernible when the SNR is -3 dB, and not visible when the SNR is -10 dB.

The relative amount of signal and noise present in a waveform is usually quantified by the signal-to-noise ratio, or SNR. As the name implies, this is simply the ratio of signal to noise, both measured in the same units (usually volts RMS). The SNR is often expressed in dB:

$$\text{SNR} = 20 \log \left(\frac{\text{Signal}}{\text{Noise}} \right) \text{dB} \quad (2.22)$$

To convert from a dB scale into a linear scale:

$$\text{SNR}_{\text{linear}} = 10^{\text{dB}/20} \quad (2.23)$$

For example, a ratio of 20 dB means that the RMS value of the signal was 10 times the RMS value of the noise ($10(20/20) = 10$); +3 dB indicates a ratio of 1.414 ($10(3/20) = 1.414$), 0 dB means the signal and noise are equal in RMS value, -3 dB means that the ratio is $1/1.414$, and -20 dB means the signal is $1/10$ of the noise in RMS units. Figure 1.17 is meant to give some feel to a range of SNR values and shows a sinusoidal signal and added white noise for different SNR values. It is difficult to detect the presence of the signal visually when the SNR is -3 dB, and impossible when the SNR is -10 dB. The ability to detect signals with low SNR is the goal and motivation for many of the signal-processing tools described in this book (Figure 2.2).

2.2 Noise

Biomedical signal processors are called upon to enhance the interpretation of the data or even to bring meaning to data that have previously been of little value. Yet all too often our task is

to improve the reliability or consistency of a measurement, and that entails reducing noise or variability. Variability really is noise, but is often used to indicate fluctuation between, as opposed to within, measurements. Fortunately, a variety of signal-processing tools exist to reduce noise, but noise is the enemy and it is useful to know something about your enemy. The more that is known about the noise, the more powerful the signal-processing methods that are applicable.

2.2.1 Noise Sources

In biomedical measurements, measurement variability has four different origins: (1) physiological variability; (2) environmental noise or interference; (3) transducer artifact; and (4) electronic noise. These sources of noise or variability along with their causes and possible remedies are presented in Table 2.3. Note that in three out of four instances, appropriate transducer design may reduce variability or noise. This demonstrates the important role of the transducer in the overall performance of the instrumentation system.

Physiological variability occurs when the information you desire is based on a measurement subject to biological influences other than those of interest. For example, assessment of respiratory function based solely on the measurement of blood pO_2 could be confounded by other physiological mechanisms that alter blood pO_2 . Physiological variability can be a very difficult problem to solve, and sometimes it requires a totally different approach.

Environmental noise can come from sources external or internal to the body. A classic example is the measurement of fetal ECG where the desired signal is corrupted by the mother's ECG. Since it is not possible to describe the specific characteristics of environmental noise, typical noise reduction techniques such as filtering are not usually successful. Sometimes environmental noise can be reduced using adaptive techniques such as those described in Chapter 8 since these techniques do not require prior knowledge of noise characteristics. Indeed, one of the approaches described in Chapter 8, adaptive noise cancellation, was initially developed to reduce interference from the mother in the measurement of fetal ECG.

Transducer artifact is produced when the transducer responds to energy modalities other than those desired. For example, recordings of internal electrical potentials using electrodes placed on the skin are sensitive to *motion artifact* where the electrodes respond to mechanical movement as well as the desired electrical signal. Transducer artifacts can sometimes be successfully addressed by modifications in transducer design. For example, aerospace research has led to the development of electrodes that are quite insensitive to motion artifact. Noise associated with the electronic components of a measurement system is the only noise that is well defined. Section 2.2.3 describes the sources and provides a quantitative description of electronic noise.

Table 2.3 Sources of Variability

Source	Cause	Potential Remedy
Physiological variability	Measurement only indirectly related to variable of interest	Modify overall approach
Environmental (internal or external)	Other sources of similar energy form	Noise cancellation, transducer design
Artifact	Transducer responds to other energy sources	Transducer design
Electronic	Thermal or shot noise	Transducer or electronic design

2.2.2 Noise Properties: Distribution Functions

Noise can be represented as a random variable. Since the variable is random, describing it as a function of time is not useful. It is more common to discuss other properties of noise such as its probability distribution, its variance (Equations 2.17 and 2.18), or its frequency characteristics. While noise can take on a variety of different probability distributions, the Central Limit Theorem implies that most noise will have a *Gaussian* or *normal* distribution.* The Central Limit Theorem states that when noise is generated by a large number of independent sources, it will have a Gaussian probability distribution regardless of the probability distribution of the individual sources. Figure 2.3a shows the distribution of 20,000 *uniformly* distributed random numbers between 0 and +1 (generated using MATLAB's `rand` function). Uniformly distributed means the noise variable has an equal probability of any value between 0 and 1, so its distribution is approximately flat between these limits as expected. Figure 2.3b shows the distribution from the same size data set where each number in the data set is now the average of two uniformly distributed random numbers. Random numbers made by averaging two very non-Gaussian random numbers together have a distribution that is much closer to Gaussian (Figure 2.3b, upper right). When eight uniformly distributed random numbers are averaged to produce a value within a 20,000 point data set, the distribution function is quite close to Gaussian (Figure 2.3d).

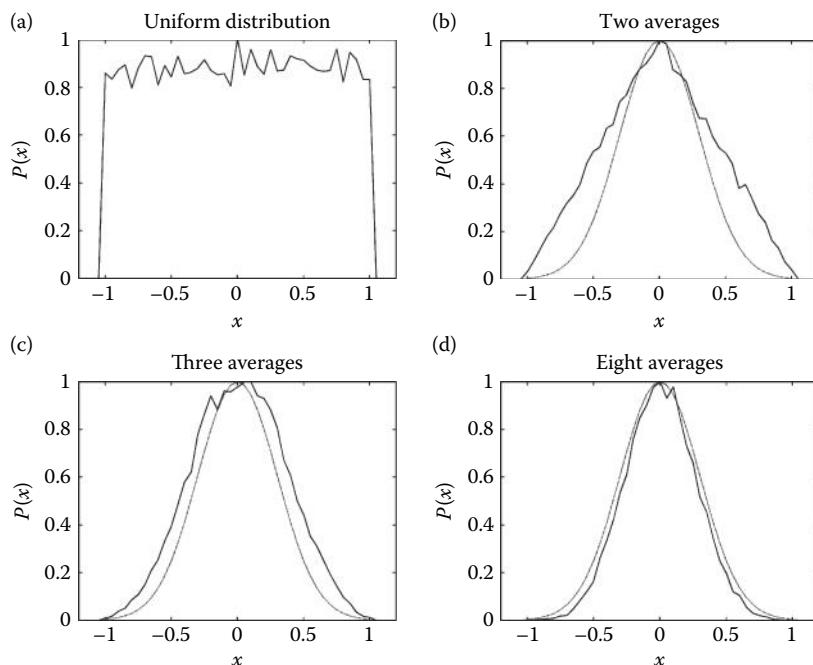


Figure 2.3 (a) The distribution of 20,000 uniformly distributed random numbers. The distribution function is approximately flat between 0 and 1. (b) The distribution of the same size data set where each number is the average of two uniformly distributed numbers. The dashed line is the Gaussian distribution. (c) Each number in the data set is the average of three uniformly distributed numbers. (d) When eight uniformly distributed numbers are averaged to produce one number in the data set, the distribution is very close to Gaussian.

* Both terms are commonly used. We use the term “Gaussian” to avoid the value judgment implied by the word “normal”!

The probability distribution of a Gaussianly distributed variable, x , is specified in the well-known equation:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-x^2/2\sigma^2} \quad (2.24)$$

This is the distribution produced by the MATLAB function `randn`.

EXAMPLE 2.3

Use a large data set generated by `randn` to determine if these data have a Gaussian probability distribution. Also estimate the probability distribution of the data produced by `rand`.

Solution

A straightforward way to estimate the distribution function of an existing data set is to construct a *histogram* of the data set. A histogram is a tabulation over the data set of the number of occurrences of a given range of values. Counts of values that fall within a given range are stored in *bins* associated with that range. The user usually decides how many bins to use. Histograms are then plotted as counts against the range. Usually the mean value of a range is used and is plotted on the horizontal axis with counts on the vertical axis.* Bar-type plots are commonly used for plotting histograms. Histograms will be very useful in image processing to graph the distribution of intensity values in an image.

The MATLAB graphics routine `hist` evaluates the histogram and as typical of MATLAB, it has a number of options. The most useful calling structure for this example is

```
[ht,xout] = hist(x,nu_bins); % Calculate the histogram of data in x
```

where the inputs are the data set vector, x , and nu_bins are the number of bins desired. The outputs are the histogram vector, ht and a vector, $xout$, which gives the mean of the ranges for the bins used. This vector is useful in correctly scaling the horizontal axis when plotting. Helpful plotting vectors are often found as outputs in MATLAB routines.

This example first constructs a large (20,000 point) data set of Gaussianly distributed random numbers using `randn`, then uses `hist` to calculate the histogram and plot the results. This procedure will be repeated using `rand` to generate the data set.

```
% Example 2.3 Evaluation of the distribution of data produced by MATLAB's
% rand and randn functions.
%
N=20000;                                % Number of data points
nu_bins=40;                               % Number of bins
y=randn(1,N);                            % Generate random Gaussian noise
[ht,xout]=hist(y,nu_bins);                % Calculate histogram
ht=ht/max(ht);                           % Normalize histogram to 1.0
bar(xout, ht);                           % Plot as bar graph
..... Label axes and title .....
.....Repeat for rand .....
```

* The vertical axis of a histogram may also be labeled "Number" (or N), "Occurrences," "Frequency of occurrence," or shortened to "Frequency." The latter term should not be confused with "frequency" when describing cycles per second which is the most common use of this word, at least in signal processing.

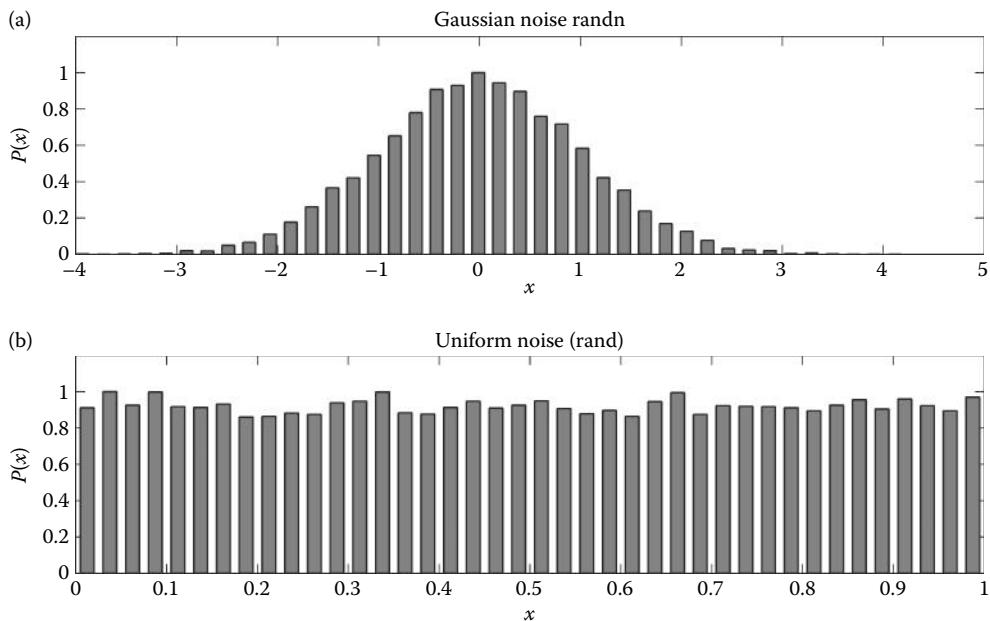


Figure 2.4 (a) The distribution of a 20,000-point data set produced by the MATLAB random number routine `randn`. As is seen here, the distribution is quite close to the theoretical Gaussian distribution. (b) The distribution of a 20,000-point data set produced by `rand`.

Results

The bar graphs produced by this example are shown in Figure 2.4 to be very close to the Gaussian distribution for the `randn` function and close to flat for the `rand` function. Problem 2.14 explores the distributions obtained using different data lengths.

2.2.3 Electronic Noise

Electronic noise is the only category of measurement variability that has well-identified sources and characteristics. Electronic noise falls into two broad classes: *thermal*, also known as *Johnson's noise*, and *shot noise*. The former is produced primarily in a resistor or resistance materials while the latter is related to voltage barriers associated with semiconductors. Both types are produced by the combined action of a large number of individual sources (think individual electrons) so they both have Gaussian distributions. Both sources produce noise with a broad range of frequencies often extending from 0 Hz* to 10^{12} – 10^{13} Hz. Such broad-spectrum noise is referred to as *white noise* since it, like white light, contains energy at all frequencies, or at least all the frequencies of interest to biomedical engineers. Figure 2.5 shows a plot of power density versus frequency for white noise calculated from a noise waveform (actually an array of random numbers) using the spectral analysis methods described in Chapter 3. Note that the energy of this simulated noise signal is fairly constant over the range of frequencies shown.

Johnson's or thermal noise is produced by resistance sources, and the amount of noise generated is related to the resistance and to the temperature:

$$V_j = \sqrt{4kTRBW} \quad \text{volts} \quad (2.25)$$

* 0 Hz is also known as *DC* or direct current, a somewhat inappropriate term that means “not time varying.”

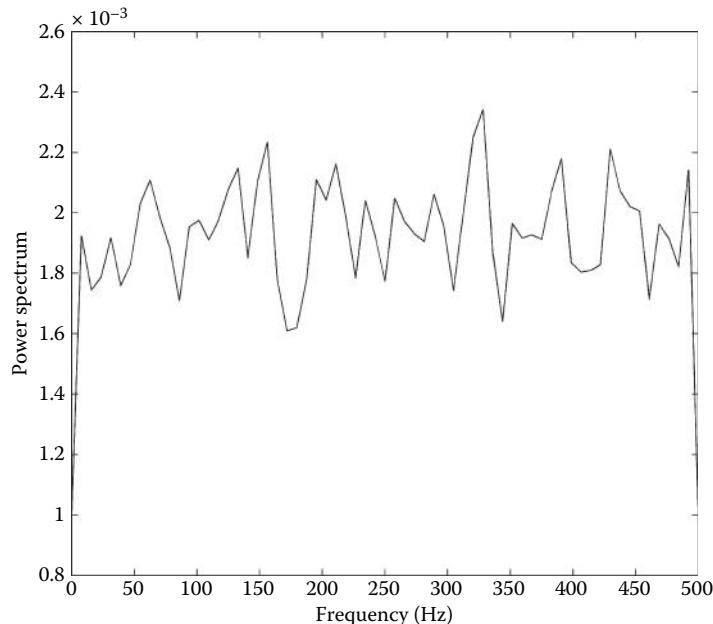


Figure 2.5 Power density (power spectrum) of white noise showing a fairly constant value over all frequencies. This constant energy feature is a property of both thermal and shot noise.

where R is the resistance in ohms, T is the temperature in degrees Kelvin, and k is Boltzman's constant ($k = 1.38 \times 10 - 23 \text{ J}^\circ\text{K}$).^{*} BW is the bandwidth of the measurement system. The system bandwidth is usually determined by the analog filters in the measurement system.

If noise current is of interest, the equation for Johnson noise current can be obtained from Equation 2.25 in conjunction with Ohm's law ($(i = v/R)$):

$$I_j = \sqrt{4kTBW/R} \quad \text{amps} \quad (2.26)$$

Since Johnson's noise is spread evenly over all frequencies (at least in theory), it is not possible to calculate a noise voltage or current without specifying BW , the bandwidth of the measurement system. Since the bandwidth is not always known in advance, it is common to describe a relative noise; specifically, the noise that would occur if the bandwidth were 1.0 Hz. Such relative noise specification can be identified by the unusual units required: volts// $\sqrt{\text{Hz}}$ or amps// $\sqrt{\text{Hz}}$.

Shot noise is defined as current noise and is proportional to the baseline current through a semiconductor junction:

$$I_d = \sqrt{2qI_dBW} \quad \text{amps} \quad (2.27)$$

where q is the charge on an electron ($1.662 \times 10^{-19} \text{ C}$), and I_d is the baseline semiconductor current. In photodetectors, the baseline current that generates shot noise is termed the *dark current*; hence, the subscript d in symbol I_d in Equation 2.27. Again, since the noise is spread across all

* A temperature of 310 K is often used as worst-case room temperature, in which case $4kT = 1.7 \times 10^{-20} \text{ J}$.

Biosignal and Medical Image Processing

frequencies, the bandwidth, BW , must be specified to obtain a specific value, or a relative noise can be specified in amps/ $\sqrt{\text{Hz}}$.

When multiple noise sources are present, as is often the case, their voltage or current contributions to the total noise add as the square root of the sum of the squares, assuming that the individual noise sources are independent. For voltages:

$$V_T = (V_1^2 + V_2^2 + V_3^2 + \dots + V_N^2)^{1/2} \quad (2.28)$$

This equation is verified empirically using computer-generated noise in Problem 2.19. A similar equation applies to current.

2.3 Signal Analysis: Data Functions and Transforms

The basic measurements presented in Section 2.1.3 are important, but they really do not tell us much about the signal. For example, the two segments of an EEG signal shown in Figure 2.6 have the same mean, RMS, and variance values, but they are clearly different. We would like some method to capture the differences between these two signals, and preferably to be able to quantify these differences. With the help of other functions or waveforms, we can define these differences.

To mathematicians, the term “function” can take on a wide range of meanings, but in signal processing, functions fall into two categories: (1) data including waveforms and images, and (2) entities that *operate* on data. Operations that modify data are often referred to as *transformations*. Transformations are used to: (1) improve data quality by removing noise as in filtering (Chapter 4); (2) make the data easier to interpret as with the Fourier transform (Chapter 3); or (3) reduce the size of the data by removing unnecessary components as with the wavelet transform (Chapter 7) or principal component analysis (Chapter 9). The type of transformation applicable depends on the type of data. Figure 2.7 summarizes the transformations presented in this book and the type of data to which these transformations apply. Single observation data means that the signal comes from only one source; with multiple observation data, the

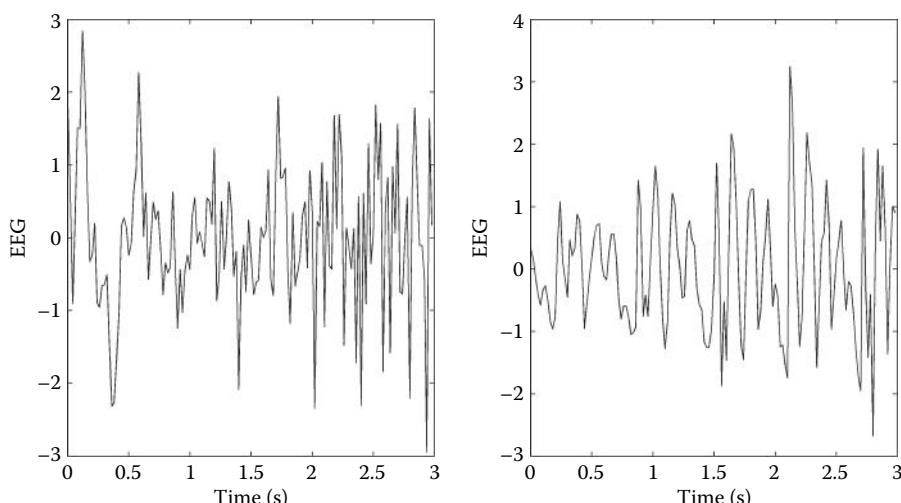


Figure 2.6 Two segments of an EEG signal with the same mean, RMS, and variance values, yet the signals are clearly different.

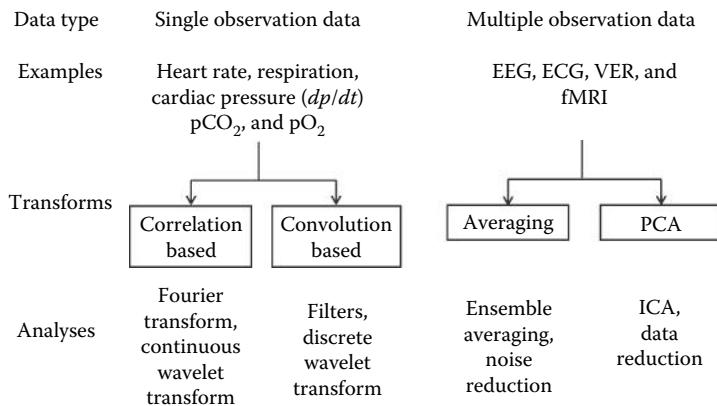


Figure 2.7 A summary of some of the analyses techniques covered in this book, the basic strategy used, and the related data type.

biological process could be monitored by several signals such as those produced by multiple electrodes detecting the EEG or ECG signals. Alternatively, the same process could be recorded multiple times in response to repeated stimuli as with the visual-evoked response (VER, see Section 4.1).

In this section, we concentrate on the two basic strategies used to transform single observation data: correlation-based methods and convolution-based methods. These methods are also frequently applied to individual records of multiple observation data. Both strategies involve multiplication of the data with some type of function either to compare the data with the signal or to produce a modified data set.

2.3.1 Comparing Waveforms

Comparisons between functions* form the basis of correlation-based analyses. *Correlation* seeks to quantify how much one function is like another. Mathematical correlation does a pretty good job of describing similarity, but once in a while it breaks down so that some functions that are conceptually similar, such as sine and cosine waves, have a mathematical correlation of zero. All correlation-based approaches involve taking the sum of the sample-by-sample product of the two functions:

$$\begin{aligned}
 r_{xy} &= x[1]y[1] + x[2]y[2] + x[3]y[3] + \dots + x[N]y[N] \\
 &= \sum_{n=1}^N x_n y_n
 \end{aligned} \tag{2.29}$$

where r_{xy} is used to indicate correlation and the subscripts x and y indicate what is being correlated. Equation 2.29 uses the summation of the sample-by-sample product, but often the mean of the summation is taken, as

$$r_{xy} = \frac{1}{N} \sum_{n=1}^N x_n y_n \tag{2.30}$$

* The terms function and signal mean the same in this context. The term signal-functions seems awkward, so throughout this discussion we use the term “function” to mean either function, signal, or waveform. Correlation-based approaches usually compare waveforms with other waveforms.

Biosignal and Medical Image Processing

Other normalization may also be used as described in Section 2.3.2. Before discussing correlation-based methods in greater detail, some preliminary concepts are in order.

2.3.1.1 Vector Representation

In some applications, it is advantageous to think of a function (of whatever type) not just as a sequence, or array, of numbers, but as a *vector*. In this conceptualization, the series of N numbers $x[n] = x[1], x[2], x[3], \dots, x[N]$ is a single vector defined by a single point: the endpoint of the vector in N -dimensional space (Figure 2.7). This somewhat curious and highly mathematical concept has the advantage of unifying some signal-processing operations and fits well with matrix methods. It is difficult for most people to imagine higher-dimensional spaces and even harder to present these spaces graphically, so operations and functions in higher-dimensional space are usually described in 2 or 3 dimensions and the extension to higher dimensions is left to the imagination of the reader. (This task is difficult for ordinary humans: try and imagine even a short data sequence of 16 samples represented as a single vector in 16-dimensional space.) Figure 2.8 shows a very short series of 3 data points $(2, 3, 2.5)$ as a 3-dimensional (3-D) vector.

The concept of thinking about a waveform (i.e., a sequence of numbers) as a vector is useful when comparing functions. If two functions are highly correlated, their vector representations project closely on one another. For this reason, correlation of two functions is often referred to as projecting one function on the other, or simply *projection*. The term correlation is more commonly used when comparing two waveforms, while projection is used when comparing a waveform with a function, but they really are the same operation. Figure 2.9 shows the 2-D vector representations of four waveform pairs having varying degrees of correlation.

The projection of one vector on another is found by taking the *scalar product* of the two vectors.* This operation demonstrates the relationship between vector projection and correlation. The scalar product is defined as

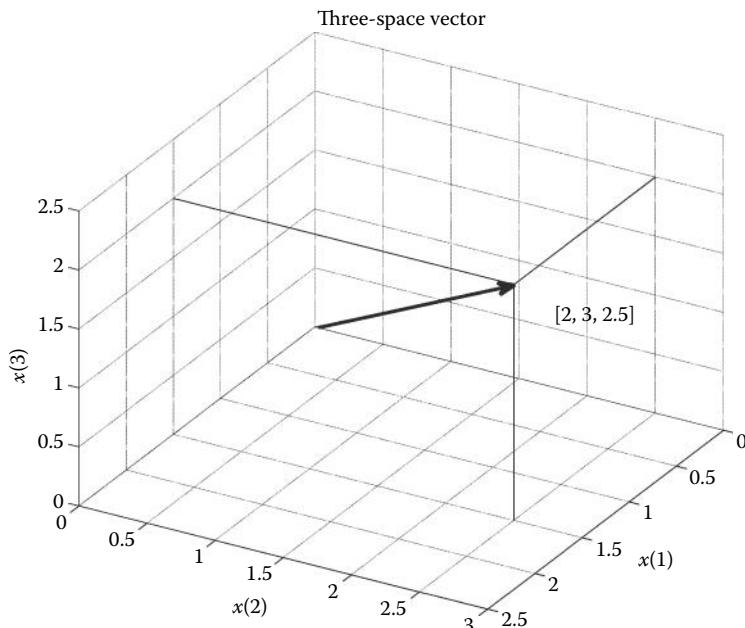


Figure 2.8 The data sequence $x[n] = [2, 3, 2.5]$ represented as a vector in 3-D space.

* The scalar product is also termed the *inner product*, the *standard inner product*, or the *dot product*.

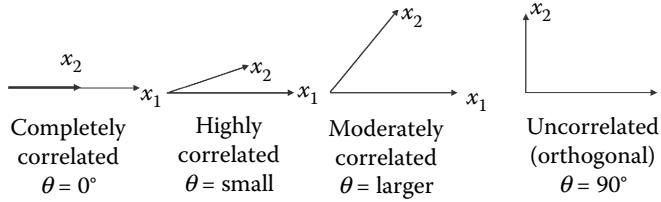


Figure 2.9 The correlation of two waveforms (or functions) can be viewed as a projection of one on the other when the signals are represented as vectors.

$$\begin{aligned}
 \text{Scalar product of } x \& y \equiv \langle x, y \rangle &= \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} \\
 &= x_1 y_1 + x_2 y_2 + \cdots + x_N y_N \\
 &= \sum_{n=1}^N x_n y_n
 \end{aligned} \tag{2.31}$$

This is the same as the basic correlation equation (Equation 2.29). So taking the scalar product of two vectors that represent functions is mathematically the same as correlating these functions and can be interpreted as projection of one vector on another. Note that the scalar product results in a single number (i.e., a scalar), not a vector. The scalar product can also be defined in terms of the magnitude of the two vectors and the angle between them:

$$\text{Scalar product of } x \& y = \langle x, y \rangle = |x||y|\cos\theta \tag{2.32}$$

where θ is the angle between the two vectors. Projection (correlation) using Equation 2.29 or 2.30 is an excellent way to compare two functions or, commonly, to compare a signal with a “probing” or “test” waveform. Such comparisons are common in signal analysis and some general examples are given later in this chapter.

EXAMPLE 2.5

Find the angle between two short signals represented as vectors. Give the angle in deg. The signals are:

$$x = [1.7, 3, 2.2] \quad \text{and} \quad y = [2.6, 1.6, 3.2]$$

Since these signals are short, plot the two vector representations in three dimensions.

Solution

Construct the two vectors and find the scalar product using Equation 2.29. When multiplying the two vectors, be sure to use MATLAB’s `*` operator to implement a point-by-point multiplication. Recall that the magnitude of a vector can be found using:

$$|x| = \sqrt{x[1]^2 + x[2]^2 + x[3]^2} \tag{2.33}$$

Biosignal and Medical Image Processing

Solve Equation 2.32 for the angle:

$$\langle x, y \rangle = |x||y|\cos\theta; \quad \cos\theta = \frac{\langle x, y \rangle}{|x||y|} \quad \theta = \cos^{-1}\left(\frac{\langle x, y \rangle}{|x||y|}\right) \quad (2.34)$$

```
% Example 2.5 Find the angle between two vectors
%
x = [1.7, 3, 2.2]; % Generate the vectors
y = [2.6, 1.6, 3.2];
sp = sum(x.*y); % Take the scalar (dot) product
mag_x = sqrt(sum(x.^2)); % Calculate magnitude of x vector
mag_y = sqrt(sum(y.^2)); % Calculate magnitude of y vector
cos_theta = sp/(mag_x*mag_y); % Calculate the cosine of theta
angle = acos(cos_theta); % Take the arc cosine and
angle = angle*360/(2*pi); % convert to degrees
hold on;
plot3(x(1),x(2),x(3),'k*'); % Plot x vector end point
plot3([0 x(1)], [0 x(2)], [0 x(3)]); % Plot x vector line
plot3(y(1),y(2),y(3),'k*'); % Plot y vector end point
plot3([0 y(1)], [0 y(2)], [0 y(3)]); % Plot vector y line
title(['Angle = ', num2str(angle, 2), ' (deg)']); % Output angle
grid on;
```

Result

The plot representing the two vectors is shown in Figure 2.10. The angle between the two vectors is calculated to be 26°. If these were signals, the fairly small angle would indicate some correlation between them.

2.3.1.2 Orthogonality

Orthogonal signals and functions are very useful in a variety of signal-processing tools. In common usage, “orthogonal” means perpendicular: if two lines are orthogonal, they are perpendicular. Indeed in the vector representation of signals just described, orthogonal signals would have orthogonal vector representations. The formal definition for orthogonal signals is that their correlation (or scalar product) is zero:

$$\sum_{n=1}^N x[n]y[n] = 0 \quad (2.35)$$

An important characteristic of signals that are orthogonal (i.e., uncorrelated) is that when they are combined or added together they *do not interact* with one another. Orthogonality simplifies many calculations and some analyses could not be done, at least not practically, without orthogonal signals. Orthogonality is not limited to two signals. Whole families of signals can be orthogonal to all other members in the family. Such families of orthogonal or *orthonormal** signals are called *orthogonal* or *orthonormal sets*.

* Orthonormal vectors are orthogonal, but in addition have unit length.

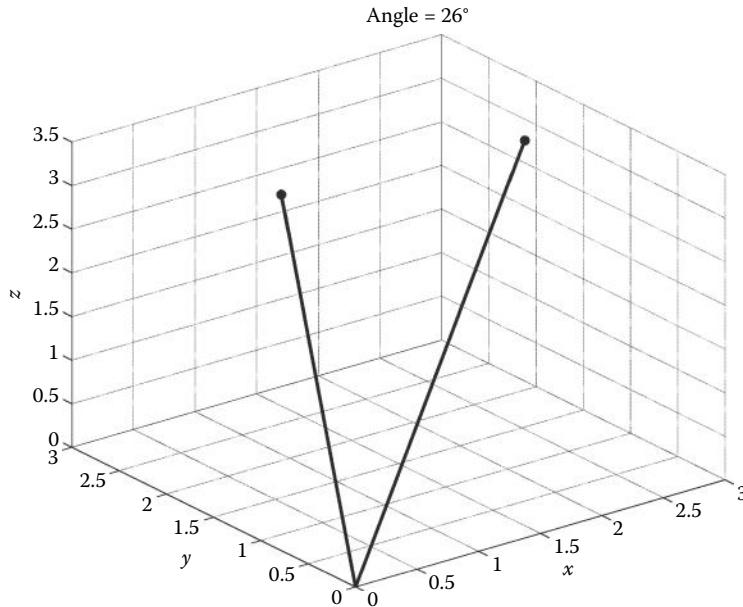


Figure 2.10 Two vectors used in Example 2.5. Using Equations 2.29, 2.32, and 2.33 the angle between these vectors is determined to be 26°. If these vectors represent signals (short signals to be sure), the small angle indicates some correlation between them.

EXAMPLE 2.6

Generate a 500 point, 2 Hz sine wave and a 4 Hz sine wave of the same length. Make $T_T = 1$ s. Are these two waveforms orthogonal?

Solution

Generate the waveforms. Since $N = 500$ and $T_T = 1$ s, $T_s = T_T/N = 0.002$ s. Apply Equation 2.29 to these waveforms. If the result is near zero, the waveforms are orthogonal.

```
% Example 2.6 Evaluate 2 waveform for Orthogonality.
%
Ts = 0.002; % Sample interval
N=500; % Number of points
t = (0:N-1)*Ts; % Time vector
f1 = 2; % Frequency of sine wave 1
f2 = 4; % Frequency of sine wave 2
x=sin(2*pi*f1*t); % Sine wave 1
y=sin(2*pi*f2*t); % Sine wave 2
Corr=sum(x.*y); % Eq. 2.29
disp(Corr)
```

Result

The correlation value produced by application of Equation 2.29 to the two sine waves is $1.9657e-014$. This is very close to zero and shows the waveforms are orthogonal. This is expected, since harmonically related sines (or cosines) are known to be orthogonal. This is one of the reasons for the utility of sinusoids in the Fourier transform as detailed in Chapter 3.

2.3.1.3 Basis Functions

A transform can be thought of as a re-mapping of the original data into something that provides more information.* The Fourier transform described in Chapter 3 is a classic example, as it converts the original time-domain data into frequency-domain data that often provide greater insight into the nature and/or origin of the signal. Many of the transforms described in this book achieved by comparing the signal of interest with some sort of probing function or a whole family of probing functions termed a *basis*. Usually the probing function or basis is simpler than the signal, for example, a sine wave or series of sine waves. (As shown in Chapter 3, a sine wave is about as simple as a waveform gets.) A quantitative comparison can tell you how much your complicated signal is like a simpler basis or reference family. If enough comparisons are made with a well-chosen basis, these comparisons taken together can provide an alternative representation of the signal. We hope the new representation is more informative or enlightening than the original.

To compare a waveform with a number of different functions that form the basis requires modification of the basic correlation equation (Equation 2.29) so that one of the functions becomes a family of functions, $f_m[n]$. If the comparison is made with a family of functions, a series of correlation values is produced, one for each family member:

$$X[m] = \sum_{n=1}^N x[n] f_m[n] \quad (2.36)$$

The multiple correlations over the entire basis result in the series $X[m]$, where m indicates the specific basis member. This analysis can also be applied to continuous functions:

$$X(m) = \int_{-\infty}^{\infty} x(t) f_m(t) dt \quad (2.37)$$

where $x(t)$ is now a continuous signal and $f_m(t)$ is the set of continuous basis functions.

Equations 2.36 and 2.37 assume that the signal and basis are the same length. If the length of the basis, $f_m[n]$, is shorter than the waveform, then the comparison can only be carried out on a portion of $x[n]$. The signal can be segmented by truncation, cutting out the desired portion, or by multiplying the signal by yet another function that is zero outside the desired portion. A function used to segment a waveform is termed a *window* function and its application is illustrated in Figure 2.11. Note that simple truncation can be viewed as multiplying the function by a *rectangular window*, a function whose value is 1 for the excised portion of the waveform and 0 elsewhere. The influence of different window functions is discussed in Chapter 3. If a window function is used, Equation 2.37 becomes:

$$X[m] = \sum_{n=1}^N x[n] f_m[n] W[n] \quad (2.38)$$

where $W[n]$ is the window function. In this equation, all the vectors are assumed to be the same length. Alternatively the summation can be limited to the length of the shortest function; that is, $N = \text{length of the shortest function}$. If $W[n]$ is a rectangular function, then $W[n] = 1$ for $1 \leq n \leq N$ and it can be omitted from the equation: a rectangular window is implemented implicitly by limits on the summation.

* Some definitions would be more restrictive and require that a transform be *bilateral*, that is, it must be possible to recover the original data from the transformed data. We use the looser definition and reserve the term *bilateral transform* to describe reversible transformations.

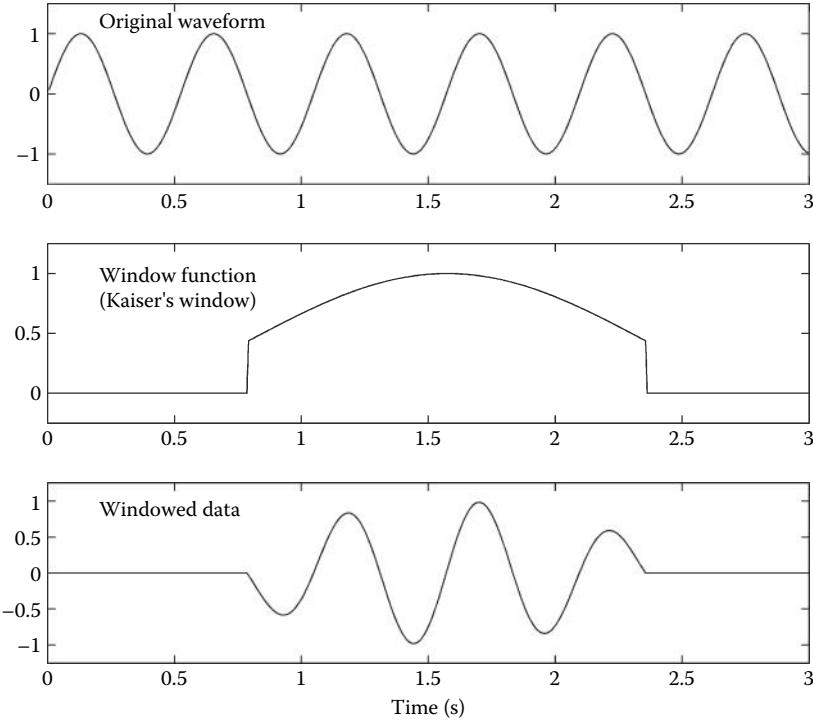


Figure 2.11 A waveform (upper plot) is multiplied by a window function (middle plot) to create a truncated version (lower plot) of the original waveform. The window function is shown in the middle plot. This particular window function is called the Kaiser window, one of many popular window functions described in Chapter 3.

If the probing function or basis is shorter than the waveform, then it might be appropriate to translate or slide it over the waveform. Then the correlation operation can take place at various relative positions along the waveform. Such a sliding correlation is shown in Figure 2.11 where a single probing function slides along a longer signal. At each position, a single correlation value is calculated:

$$X[k] = \sum_{n=1}^N x[n] f[n+k] \quad (2.39)$$

The output is a series of correlation values as a function of k which defines the relative position of the probing function. If the probing function is a basis (i.e., a family of functions), then the correlation values are a function of both k and family member m :

$$X[m,k] = \sum_{n=1}^N x[n] f_m[n+k] \quad (2.40)$$

where the variable k indicates the relative position between the two functions and m defines the specific family member of the basis. Equation 2.40 is used to implement the continuous wavelet transform described in Chapter 7 (Figure 2.12).

A variation of this approach can be used for long, or even infinite, probing functions, provided the probing function itself is shortened by windowing to a length that is less than the

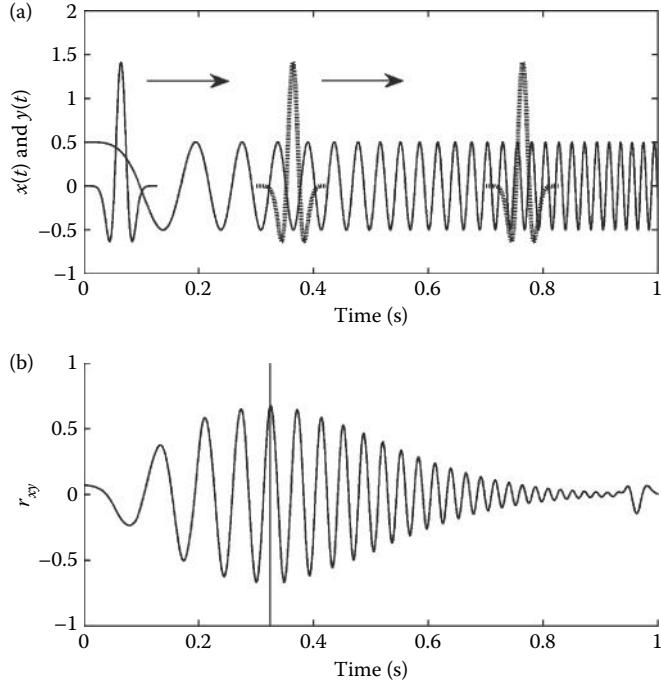


Figure 2.12 (a) The probing function slides over the signal and at each position the correlation between probe and signal is calculated using Equation 2.39. In this example, the probing function is one member of the *Mexican Hat* family (see Chapter 7) and the signal is a sinusoid that increases its frequency linearly over time known as a *chirp*. (b) The result shows the correlation between the waveform and the probing function as it slides across the waveform. Note that this correlation varies sinusoidally as the phase between the two functions varies, but reaches a maximum around 2.5 s, the time when the signal is most like the probing function.

waveform. Then the shortened probing function can be moved across the waveform in the same manner as a short probing function. The equation for this condition becomes

$$X[m, k] = \sum_{n=1}^N x[n] (W[n+k] \quad f_m[n]) \quad (2.41)$$

where $f_m[n]$ are basis functions shortened or restricted by the sliding window function, $W[n+k]$. The variables n and k have the same meaning as in Equation 2.40: basis family member and relative position. This is the approach taken in the short-term Fourier transform described in Chapter 6.

Note that Equations 2.36 through 2.41 all involve correlation between the signal and a fixed or sliding probing function or basis. A fixed or sliding window function may also be involved. A series of correlations is generated, possibly even a two-dimensional series. We will encounter each of these equations again in different signal analyses, illustrating the importance and fundamental nature of the basic correlation equation (Equation 2.29).

2.3.2 Correlation-Based Analyses

Correlation, essentially the summed product of two functions, is at the heart of many signal-processing operations. As is mentioned above, mathematical correlation does a good job of

quantifying the similarity of two functions, but sometimes can be misleading. For example, the correlation between a sine wave and a cosine wave is zero despite the fact that they have similar oscillatory behavior. When exploring the oscillatory behavior of waveforms, simple correlation may not suffice. Fortunately, some variations of correlation described here can accurately capture oscillatory and other behavior that might be missed with simple correlation (Equation 2.29). In addition, different approaches to normalizing the correlation sum are presented.

2.3.2.1 Correlation and Covariance

Covariance computes the variance that is shared between two (or more) waveforms. The equation is similar to that of basic correlation (Equation 2.30) except that the means are removed from the two waveforms and the correlation sum is normalized by $N - 1$:

$$\sigma_{xy} = \frac{1}{N-1} \sum_{n=1}^N (x_n - \bar{x})(y_n - \bar{y}) \quad (2.42)$$

where σ_{xy} is the covariance between x and y and \bar{x} and \bar{y} are the means of x and y .

A popular variation of the basic correlation equation normalizes the correlation sum so that the outcome lies between ± 1 . This correlation value, known as the Pearson correlation coefficient, is obtained using a modification of the covariance equation (Equation 2.42):

$$r_{xy\ Pearson} = \frac{1}{(N-1)\sigma_x\sigma_y} \sum_{n=1}^N (x_n - \bar{x})(y_n - \bar{y}) \quad (2.43)$$

where σ_x and σ_y are the variances, and \bar{x} and \bar{y} are the means, of x and y . Again r_{xy} is the common symbol used for correlations between x and y . This equation could be embedded in any of the general equations defined in the last equation. Note that if the means of the waveforms are removed, the Pearson correlation coefficient can be obtained from the basic correlation equation using

$$r_{xy\ Pearson} = \frac{r_{xy}}{\sqrt{\sigma_1^2\sigma_2^2}} \quad (2.44)$$

where the variances are the same as used in Equation 2.43. This will make the correlation value equal to $+1$ when the two signals are identical and -1 if they are exact opposites. In this book, the term “Pearson correlation coefficient” or just “correlation coefficient” implies this normalization, while the term “correlation” is used more generally to mean normalized or un-normalized correlation.

EXAMPLE 2.7

Find the Pearson correlation coefficient between the two waveforms shown in Figure 2.13. These waveforms are stored as variables x and y in file `Ex2_7.mat`.

Solution

Load the file and find the unnormalized correlation using Equation 2.29 as in Example 2.5 and apply Equation 2.44. Subtract the means of each signal before correlation.

```
load Ex2_7 % Load the file
N = length(x); % Find N
rxy = sum((x-mean(x)).*(y-mean(y))); % Subtract means and apply Eq. 2.29
rxy = rxy/((N-1)*sqrt(var(x)*var(y))); % Apply Eq. 2.44
title(['Correlation: ', num2str(rxy)]); % Output correlation
```

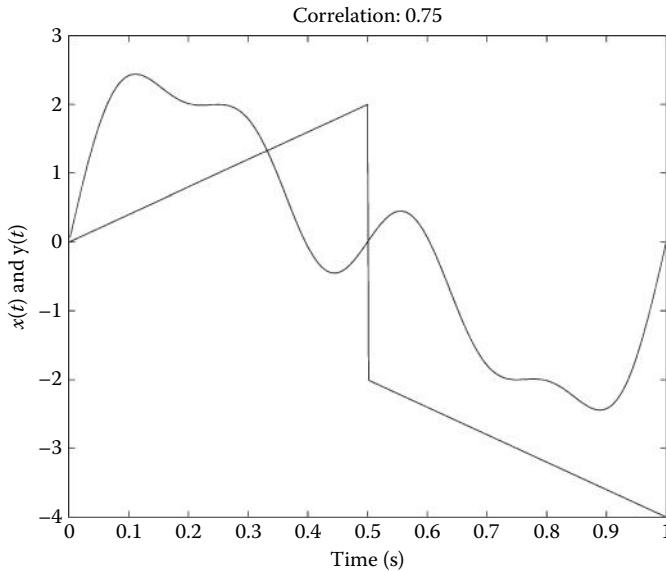


Figure 2.13 Two waveforms used in Example 2.7. The Pearson correlation was found to be 0.75, reflecting the similarity between the two waveforms.

2.3.2.2 Matrix of Correlations

MATLAB has functions for determining the correlation and/or covariance that are particularly useful when more than two waveforms are compared. A matrix of correlation coefficients or covariances between different combinations of signal can be calculated using:

```
Rxx = corrcoef(X); % Signal correlations
S = cov(X); % Signal covariances
```

where X is a matrix that contains the various signals to be compared in columns. Some options are available as explained in the associated MATLAB help file. The output, R_{xx} , of `corrcoef` is an $n \times n$ matrix Pearson correlation coefficients where n is the number of waveforms arranged as columns in the input matrix X :

$$R_{xx} = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,N} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ r_{N,1} & r_{N,2} & \cdots & r_{N,N} \end{bmatrix} \quad (2.45)$$

The diagonals of this matrix represent the correlation coefficient of the signals with themselves so they are 1. The off-diagonals represent the correlation coefficients of the various combinations. For example, r_{12} is the correlation between signals 1 and 2. Since the correlation of signal 1 with signal 2 is the same as that of signal 2 with signal 1, $r_{12} = r_{21}$, and since $r_{m,n} = r_{n,m}$ the matrix will be symmetrical. The `cov` routine produces a similar output, except the diagonals are the variances of the various signals and the off-diagonals are the covariances (Equation 2.46):

$$S = \begin{bmatrix} \sigma_{1,1}^2 & \sigma_{1,2}^2 & \cdots & \sigma_{1,N}^2 \\ \sigma_{2,1}^2 & \sigma_{2,2}^2 & \cdots & \sigma_{2,N}^2 \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{N,1}^2 & \sigma_{N,2}^2 & \cdots & \sigma_{N,N}^2 \end{bmatrix} \quad (2.46)$$

Example 2.8 uses covariance and correlation analysis to determine if sines and cosines of different frequencies are orthogonal. Recall that if two signals are orthogonal, they will have zero correlation. Either covariance or correlation could be used to determine if signals are orthogonal. Example 2.8 uses both.

EXAMPLE 2.8

Determine if a sine wave and cosine wave at the same frequency are orthogonal and if sine waves at harmonically related frequencies are orthogonal. The term “harmonically related” means that sinusoids are related by frequencies that are multiples. Thus, the signals $\sin(2t)$, $\sin(4t)$, and $\sin(6t)$ are harmonically related.

Solution

Generate a 500-point, 1.0-s time vector as in the last example. Use this time vector to generate a data matrix where the columns represent 2- and 4-Hz cosine and sine waves. Apply the covariance and correlation MATLAB routines (i.e., cov and corrcoef) and display results.

```
% Example 2.8 Application of the covariance matrix to sinusoids
% that are orthogonal and a sawtooth
%
N=1000; % Number of points
Tt = 2; % desired total time
fs = N/Tt; % Calculate sampling frequency
t = (0:N-1)/fs; % Time vector
X(:,1) = cos(2*pi*t)'; % Generate a 1Hz cosine
X(:,2) = sin(2*pi*t)'; % Generate a 1Hz sine
X(:,3) = cos(4*pi*t)'; % Generate a 2Hz cosine
X(:,4) = sin(4*pi*t)'; % Generate a 1Hz sine
%
S = cov(X) % Print covariance matrix
Rxx = corrcoef(X) % and correlation matrix
```

Analysis

The program defines a time vector in the standard manner and uses this time vector to generate the sine and cosine waves. The program then determines the covariance and correlation matrices of X.

Results

The output from this program is a covariance and correlation coefficient matrix:

$S =$

```
0.5005 -0.0000 -0.0000 0.0000
-0.0000 0.5005 -0.0000 0.0000
```

Biosignal and Medical Image Processing

```

-0.0000 -0.0000  0.5005  0.0000
 0.0000  0.0000  0.0000  0.5005

Rxx =
1.0000 -0.0000 -0.0000  0.0000
-0.0000  1.0000 -0.0000  0.0000
-0.0000 -0.0000  1.0000  0.0000
 0.0000  0.0000  0.0000  1.0000

```

The off-diagonals of both matrices are zero, showing that all these sine and cosine waves are uncorrelated and hence orthogonal.

2.3.2.3 Cross-Correlation

The mathematical dissimilarity between a sine and a cosine is disconcerting and a real problem if you are trying to determine if a signal has the oscillatory pattern characteristic of a sinusoid. Correlating a waveform with a sine wave might lead you to believe it does not exhibit sinusoidal behavior even if it is, for example, a perfect cosine wave. One way to circumvent this mis-identification would be to phase shift the sine wave between 0° and 90° so that it looks like a cosine wave (a 90° shift) and every sinusoid in between, but in order to determine its similarity to a general sinusoid (i.e., $\sin(2\pi ft + \theta)$) you would need to correlate the sine wave at every possible phase shift (or, equivalently every possible time shift). In fact, this is what is done; the approach of shifting and correlating is called *cross-correlation* and is guaranteed to find the best match between any two signals.

The equation for cross-correlation is similar to Equation 2.39 except that the shifted function does not need to be shorter than the waveform. In addition, the correlation sum is normalized by $1/N$:

$$r_{xy}[k] = \frac{1}{N} \sum_{n=1}^N y[n]x[n+k] \quad (2.47)$$

where the shift value, k , specifies the number of samples shifted for a given correlation and $r_{xy}[k]$ is a series of correlations as a function of shift. The shift is often called *lags* (as in how much one shifted waveform lags the other). The maximum shift value will depend on the particular analysis, but will be $\leq \pm N$. If the cross-correlated signals were originally time functions, the lags may be converted into time shifts in seconds. Finally, it does not matter which function is shifted with respect to the other, the results will be the same.

The two waveforms need not be the same length; however, even if one waveform is much shorter, there will be points missing in one of the waveforms if the shift becomes large enough. If the two waveforms are the same length, there will also be points missing in one waveform for all nonzero lags. There are a number of different ways of dealing with missing end points, but the most common is to use zeros for the missing points. Recall that this is termed zero padding. The maximum shift value could be just the length of the longest waveform, but often the data are extended with zeros to enable correlations at all possible shift positions: positive and negative. In such cases, the maximum shift is the *combined* length of the two data sets minus one.

In MATLAB, cross-correlation could be implemented using the routine `xcorr`. The most common calling structure for this routine is

```
[rxy,lags] = xcorr(x,y,maxlags);
```

where `x` and `y` are the waveforms to be cross-correlated and `maxlags` specifies the shift range as \pm `maxlags`. If that argument is omitted, the default maximum shift is:

`length(x) + length(y) - 1`. The outputs are `rxy`, a vector of correlation values (normalized by $1/N$) and `lags`, a vector of the same length giving corresponding lag values useful for plotting. The `xcorr` function is part of the Signal Processing Toolbox. If that toolbox is unavailable, there is a similar routine, `axcor`, which can be found in the accessory material. The routine has the same calling structure except that there is no option for modifying the maximum shift: it is always `length(x) + length(y)-1`. Of course you can always limit the plots or other output to the range of interest. In addition, `axcor` uses scaling that gives the output as Pearson's correlation coefficients (as in Equation 2.44). Although cross-correlation operations usually use the scaling in Equation 2.47, `axcor` produces a cross-correlation function that ranges between ± 1 . Dividing the output of `xcorr` by the square root of the variances of each signal will give similar results.

```
[rxy, lags] = axcor(x, y); % Crosscorrelation
```

EXAMPLE 2.9

File `neural_data.mat` contains two waveforms, `x` and `y`, that were recorded from two different neurons in the brain with a sampling interval of 0.2 ms. They are believed to be involved in the same function, but separated by one or more neuronal junctions that impart a delay to the signal. Plot the original data, determine if they are related and, if so, the time delay between them.

Solution

Take the cross-correlation between the two signals using `axcor`. Find the maximum correlation and the time shift at which that maximum occurs. The former will tell us if they are related and the latter the time delay between the two nerve signals.

```
load neural_data.mat; % Load data
fs=1/0.0002; % Sampling freq. (1/Ts)
t=(1:length(x))/fs; % Time vector
subplot(2,1,1);
plot(t,y,'k',t,x,:'); % Plot original data
.....labels.....
[rxy, lags] = axcor(x, y); % Compute crosscorrelation
subplot (2,1,2);
plot(lags/fs,rxy,'k'); % Plot crosscorrelation function
[max_corr, max_shift] = max(rxy); % Find max correlation and shift
max_shift=lags(max_shift)/fs; % Convert max shift to sec
plot(max_shift,max_corr,'*k'); % Plot max correlation
disp([max_corr max_shift]) % Output delay in sec
..... labels, title, scaling.....
```

Analysis

After cross-correlation, finding the maximum correlation is straightforward using MATLAB's `max` operator. Finding the time at which the maximum value occurs is a bit more complicated. The `max` operator provides the index of the maximum value, labeled here as `max _ shift`. To find the actual shift corresponding to this index, we need to find the lag value at this shift, that is, `lags(max _ shift)`. This lag value then needs to be converted into a corresponding time shift by dividing it by the sampling frequency, `fs` (or multiplying by T_s). The position of the peak is plotted on the cross-correlation curve as a *-point in Figure 2.14.

Result

The two signals are shown in Figure 2.14a. The cross-correlation function is seen in Figure 2.14b. The time at which the peak occurs is indicated and is found to be 0.013 s after converting into time as described above. The maximum correlation is 0.45, suggesting the two signals are related.

The next example anticipates the Fourier transform presented in Chapter 3. In Example 2.10, we take the EEG signal shown in Figure 2.15 and compare the EEG signal to a sinusoid at a given frequency. Since MATLAB will do the work, we can do this comparison over a range of frequencies. Unfortunately, we cannot simply compare the signal with a sine wave using standard correlation (say, Equation 2.29), because we want to compare it to a general *sinusoid*, not only a sine wave. A sinusoid could be a sine wave, but with any given phase shift (or equivalent time shift) up to and including 90°. One approach is to use cross-correlation to compare the EEG signal to shifted sine waves and take the maximum correlation as in the last example. (If we were smart, we would limit the shift to the equivalent of $\pm 90^\circ$, but it is easier to just use the maximum shift and allow MATLAB to do the extra work.)

EXAMPLE 2.10

Compare the EEG signal shown in Figure 2.15 with sinusoids ranging in frequencies between 1.0 and 25 Hz. The sinusoidal frequencies should be in 1.0 Hz increments.

Solution

Load the ECG signal found as vector `eeg` in file `eeg_data.mat`. Use a loop to generate a series of sine waves from 0.25 to 25 Hz. Since we do not know the best frequencies to use in the

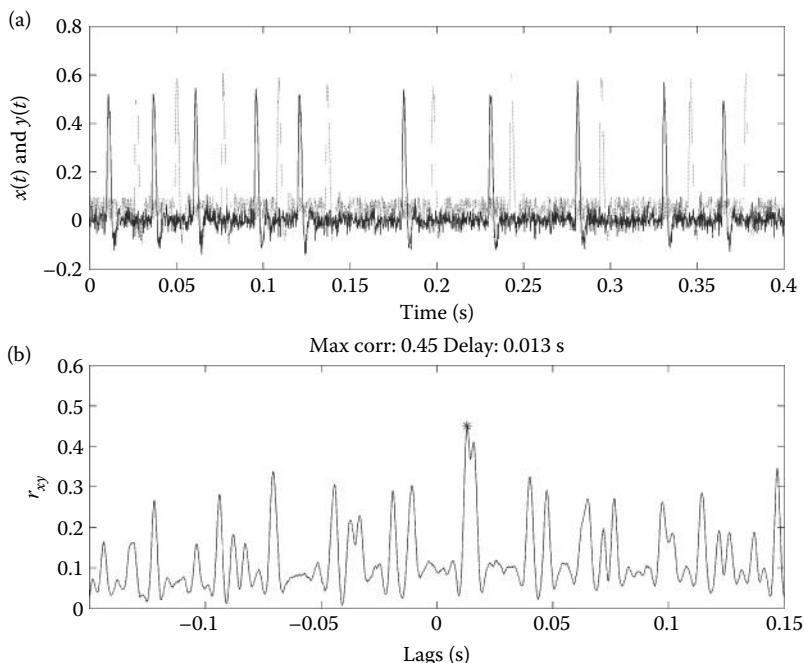


Figure 2.14 (a) Signals recorded from two different neurons (light and dark traces) used in Example 2.9. (b) To determine if they are related, the cross-correlation is taken and the delay (lag) at which the maximum correlation occurs is determined. The maximum correlation found by the `max` operator is indicated.

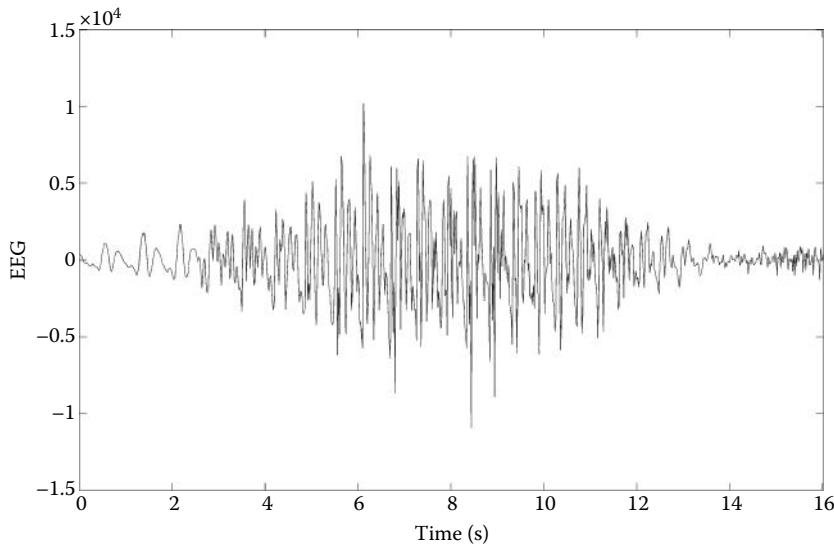


Figure 2.15 An EEG signal that is compared with sinusoids at different frequencies in Example 2.10.

correlation, we will increment the sine wave frequency in small intervals of 0.25 Hz. (Cosine waves would work just as well since the cross-correlation covers all possible phase shifts.) Cross-correlate these sine waves with the EEG signal and find the maximum cross-correlation. Plot this maximum correlation as a function of the sine wave frequency.

```
% Example 2.10 Comparison of an EEG signal with sinusoids
%
load eeg_data; % Get eeg data
fs = 50; % Sampling frequency
t = (1:length(eeg))/fs; % Time vector
for i = 1:25
    f(i) = 0.25*i; % Frequency range: 0.25–25 Hz
    x = sin(2*pi*f(i)*t); % Generate sine
    rxy = axcor(eeg,x); % Perform crosscorrelation
    rmax(i) = max(rxy); % Store max value
end
plot(f,rmax,'k'); % Plot max values as function of freq.
```

Result

The result of the multiple cross-correlations is seen in Figure 2.16, and an interesting structure emerges. Some frequencies show much higher correlation between the sinusoid and the EEG. A particularly strong peak is seen in the region of 7–9 Hz, indicating the presence of an oscillatory pattern known as the *alpha wave*. The Fourier transform is a more efficient method for obtaining the same information as shown in Chapter 3.

2.3.2.4 Autocorrelation

It is also possible to correlate a signal with other segments of itself. This can be done by performing cross-correlation on two identical signals, a process called *autocorrelation*. Autocorrelation is easy to implement: simply apply cross-correlation with the same signal as both original signal

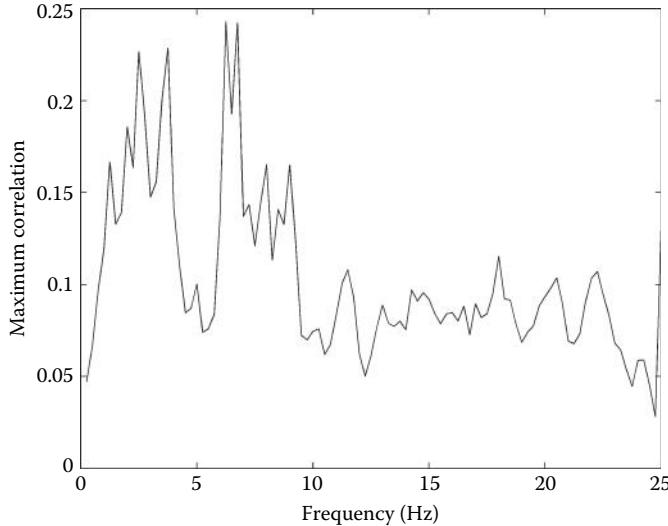


Figure 2.16 The maximum correlation between the EEG signal in Figure 2.15 and a sinusoid plotted as a function of the frequency of the sinusoid.

and reference (e.g., `axcor(x,x)`)—but it is harder to understand what the result signifies. Basically, the autocorrelation function describes how well a signal correlates with shifted versions of itself. This could be useful in finding segments of a signal that repeat. Another way of looking at autocorrelation is that it shows how the signal correlates with neighboring portions of itself. As the shift increases, the signal is compared with more distant neighbors. Determining how neighboring segments of a signal relate to one another provides some insight into how the signal was generated or altered by intervening processes. For example, a signal that remains highly correlated with itself over a period of time must have been produced, or modified, by some processes that took into account previous values of the signal. Such a process can be described as having “memory,” since it must remember past values of the signal (or input) and use this information to shape the signal’s current values. The longer the memory, the more the signal will remain partially correlated with shifted versions of itself. Just as memory tends to fade over time, the autocorrelation function usually goes to zero for large enough time shifts.

To derive the autocorrelation equation, simply substitute the same variable for x and y in Equation 2.47:

$$r_{xx}[k] = \frac{1}{N} \sum_{n=1}^N x[n]x[n+k] \quad (2.48)$$

where $r_{xx}[k]$ is the autocorrelation function.

Figure 2.17 shows the autocorrelation of several different waveforms. In all cases, the correlation has a maximum value at zero lag (i.e., no time shift) since when the lag is zero, this signal is being correlated with itself. It is common to normalize the autocorrelation function to 1.0 at lag 0. The autocorrelation of a sine wave is another sinusoid, as shown in Figure 2.17a, since the correlation varies sinusoidally with the lag, or phase shift. Theoretically the autocorrelation should be a pure cosine, but because the correlation routine adds zeros to the end of the signal to compute the autocorrelation at all possible time shifts, the cosine function decays as the shift increases.

A rapidly varying signal, as in Figure 2.17c, *decorrelates* quickly: the correlation of neighbors falls off rapidly for even small shifts of the signal with respect to itself. One could say that this signal has a poor memory of its past values and was probably the product of a process

2.3 Signal Analysis

with a short memory. For slowly varying signals, the correlation falls slowly, as in Figure 2.17b. Nonetheless, for all of these signals there is some time shift for which the signal becomes completely decorrelated with itself. For Gaussian noise, the correlation falls to zero instantly for all positive and negative lags, as in Figure 2.17d. This indicates that each signal sample has no correlation with neighboring samples.

Since shifting the waveform with respect to itself produces the same results no matter which way the waveform is shifted, the autocorrelation function will be symmetrical about lag zero. Mathematically, the autocorrelation function is an even function:

$$r_{xx}(-\tau) = r_{xx}(\tau) \quad (2.49)$$

The maximum value of r_{xx} clearly occurs at zero lag, where the waveform is correlated with itself. If the autocorrelation is normalized by the variance which is common, the value will be 1 at zero lag. (Since in autocorrelation the same function is involved twice, the normalization equation given in Equation 2.43 would include a $1/\sigma^2$.)

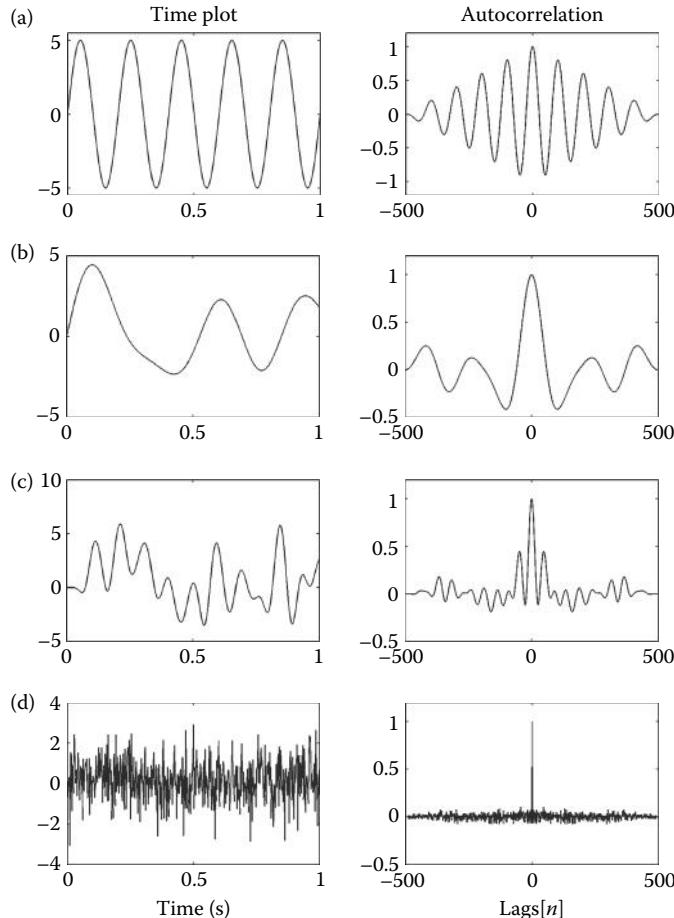


Figure 2.17 Four different signals (left side) and their autocorrelation functions (right side): (a) A truncated sinusoid. The reduction in amplitude is due to the finite length of the signal. A true (i.e., infinite) sinusoid would have a nondiminishing cosine wave as its autocorrelation function; (b) A slowly varying signal; (c) A rapidly varying signal; (d) A random signal (Gaussian's noise). Signal samples are all uncorrelated with one another.

Biosignal and Medical Image Processing

When autocorrelation is implemented on a computer, it is usually considered just a special case of cross-correlation. When only a single input vector is provided, both `xcorr` and `axcor` assume the autocorrelation function is desired.

```
[rxx, lags] = axcor(x); % Autocorrelation  
[rxx, lags] = xcorr(x, maxlags, 'coeff'); % Autocorrelation
```

The '`coeff`' option is used with MATLAB's `xcorr` routine to indicate that the output should be normalized to 1.0 at zero shift (i.e., `lags = 0`) when the signal is identically correlated with itself. The routine `axcor` does this automatically when only a single input is given. A simple application of autocorrelation is shown in the next example.

2.3.2.5 Autocovariance and Cross-Covariance

Two operations closely related to autocorrelation and cross-correlation are autocovariance and cross-covariance. The relationship between these functions is similar to the relationship between standard correlation and covariance: they are the same except with covariances in which the signal means have been removed:

$$c_{xx}[k] = \frac{1}{N} \sum_{n=1}^N (x[n] - \bar{x})(x[n+k] - \bar{x}) \quad (2.50)$$

$$c_{xy}[k] = \frac{1}{N} \sum_{n=1}^N (y[n] - \bar{y})(x[n+k] - \bar{x}) \quad (2.51)$$

The autocovariance function can be thought of as measuring the memory or self-similarity of the *deviation* of a signal about its mean level. Similarly, the cross-covariance function is a measure of the similarity of the deviation of two signals about their respective means. An example of the application of the autocovariance to the analysis of heart rate variability is given in Example 2.11.

Since auto- and cross-covariance are the same as auto- and cross-correlation if the data have zero means, they can be implemented using `axcor` or `xcorr` with the data means subtracted out; for example,

```
[cxy, lags] = axcor(x-mean(x), y-mean(y)); % Cross-covariance
```

Many physiological processes are repetitive, such as respiration and heart rate, yet vary somewhat cycle to cycle. Autocorrelation and autocovariance can be used to explore this variation. For example, considerable interest revolves around the heart rate and its beat-to-beat variations. (Remember that autocovariance will subtract the mean value of the heart rate from the data and analyze only the variation.)

Figure 2.18 shows the time plots of instantaneous heart rate in beats per minute taken under normal and meditative conditions. These data are found in the associated material as `HR_pre.mat` (preliminary) and `HR_med.mat` (meditative) and were originally from the PhysioNet database (Goldberger et al., 2000). Clearly the mediators have a higher-than-average heart rate, a different structure, and their variation is greater. Example 2.11 uses autocovariance to examine the variation in successive beats. In this example, we use autocovariance, not autocorrelation, since we are interested in correlation of heart rate *variability*, not the correlation of heart rate *per se*.

EXAMPLE 2.11

Determine if there is any correlation in the variation between the timing of successive heartbeats under normal resting conditions.

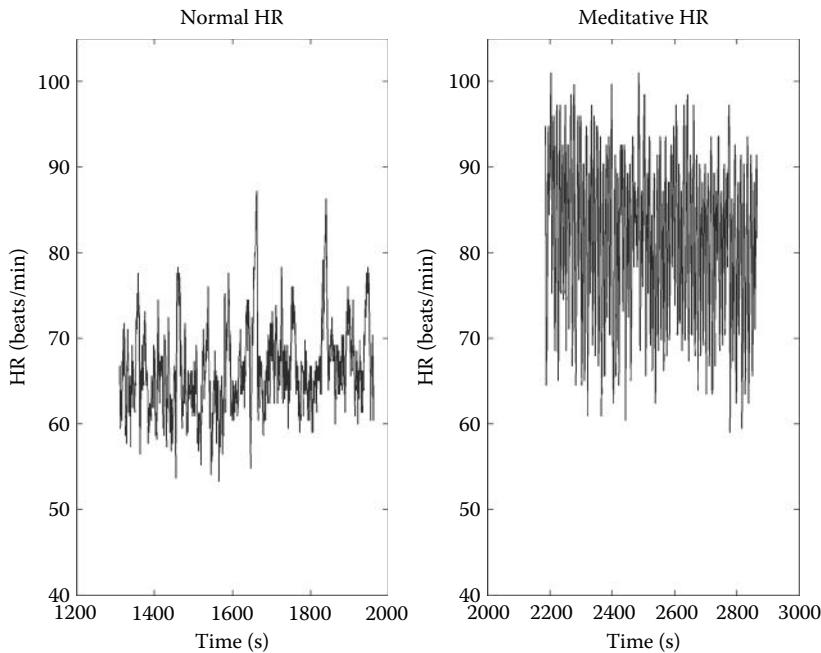


Figure 2.18 Approximately 10 min of instantaneous heart rate data taken from a normal subject and one who is meditating. The meditating subject shows a higher overall heart rate and much greater beat-to-beat fluctuations.

Solution

Load the heart rate data taken during normal conditions. The file `Hr_pre.mat` contains the variable `hr_pre`, the instantaneous heart rate. However, the heart rate is determined each time a heartbeat occurs, so it is not evenly time sampled. A second variable `t_pre` contains the time at which each beat is sampled. For this problem, we will determine the autocovariance as a function of heart beat and we will not need the time variable. We can determine the autocovariance function using `axcor` by first subtracting the mean heart rate. The subtraction can be done within the `axcor` input argument. We then plot the resulting autocovariance function and limit the `x` axis to ± 30 successive beats to better evaluate the decrease in covariance with successive beats.

```
% Example 2.11 Use of autocovariance to determine the correlation
% of heart rate variation between heart beats
%
load Hr_pre; % Load normal HR data
[cov_pre,lags_pre] = axcor(hr_pre - mean(hr_pre)); % Auto-covariance
plot(lags_pre,cov_pre,'k'); hold on; % Plot normal auto-cov
plot([lags_pre(1) lags_pre(end)], [0 0], 'k'); % Plot a zero line
axis([-30 30 -0.2 1.2]); % Limit x-axis to ± 30 beats
```

Results

The results in Figure 2.19 show that there is correlation between adjacent heartbeats all the way out to 10 beats. The autocovariance of the heart rate data during meditation is examined in Problem 2.35.

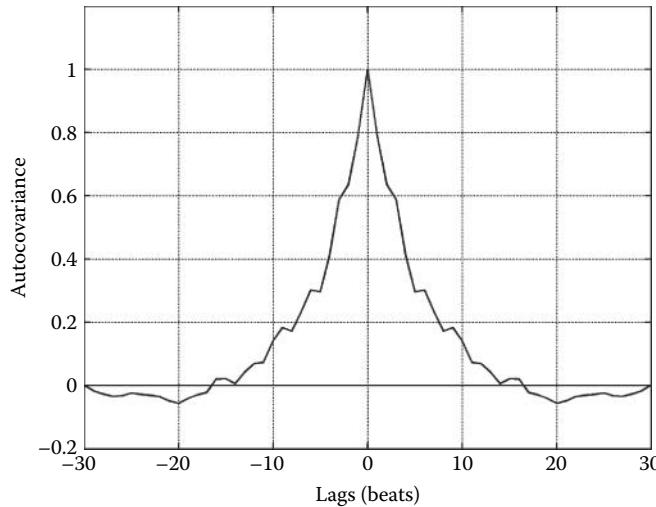


Figure 2.19 Autocovariance function of the heart rate under normal resting conditions (solid line). Some correlation is observed in the normal conditions over approximately 10 successive heartbeats.

2.3.3 Convolution and the Impulse Response

Convolution is an important concept in linear systems theory and is used as the time domain equivalent to the *transfer function*. The transfer function is a frequency-domain concept that is used to calculate the output of a linear system to any input. In the time domain, convolution is used to define a general input–output relationship for a linear, time-invariant (LTI) system. In Figure 2.20, the input, $x(t)$, and the output, $y(t)$ are linked via convolution through the function, $h(t)$. Since these are all functions of time, convolution is a time-domain operation. In the digital domain, the three time functions become sampled discrete vectors: $x[n]$, $y[n]$, and $h[n]$.

The function, $h(t)$, is known as the *impulse response*. As the name implies, it is the system's response to a standard impulse input (Figure 2.21b). An impulse input (also termed a *delta* function and commonly denoted $\delta(t)$) is a very short pulse with an area of 1.0 (in whatever units you are using) (Figure 2.21a). In theory, it is infinitely short but also of infinite amplitude, compensating in such a manner as to keep the area 1.0. It is a little challenging to generate an infinitely short pulse, so just a short pulse is used. In practice, an impulse should be short enough so that if you shorten it further the system's response (the impulse response) does not change its basic shape.

If you know the system's response to an impulse, you can determine its response to any input simply by dividing the input into a sequence of impulses (Figure 2.21b). Each time slice will generate its own little impulse response (Figure 2.22). The amplitude and position of this impulse response is determined by the amplitude and position of the associate input signal segment. If superposition and time invariance hold, then the output can be determined by summing (or integrating for continuous functions) the impulse responses from all the input signal segments.

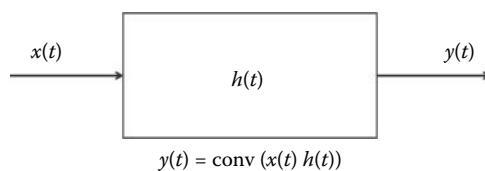


Figure 2.20 The operation of convolution is used in linear systems theory to calculate the output of an LTI system to any input signal.

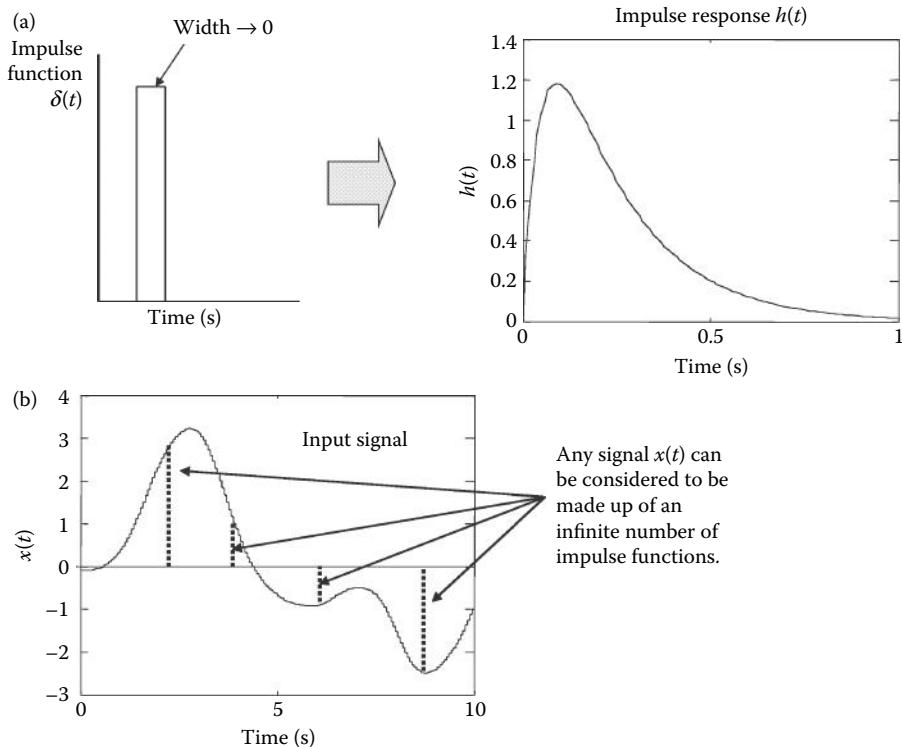


Figure 2.21 (a) If the input to a system is a short pulse (left side), then the output is termed the impulse response (right side). (b) Every signal can be thought of as composed of short pulses.

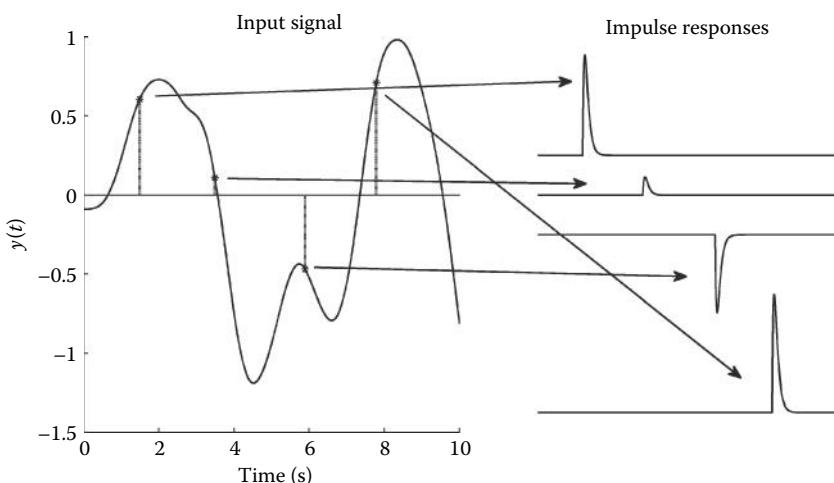


Figure 2.22 Each segment of an input signal can be represented by an impulse, and each produces its own impulse response. The amplitude and position of the impulse response is determined by the amplitude and position of the associated input segment.

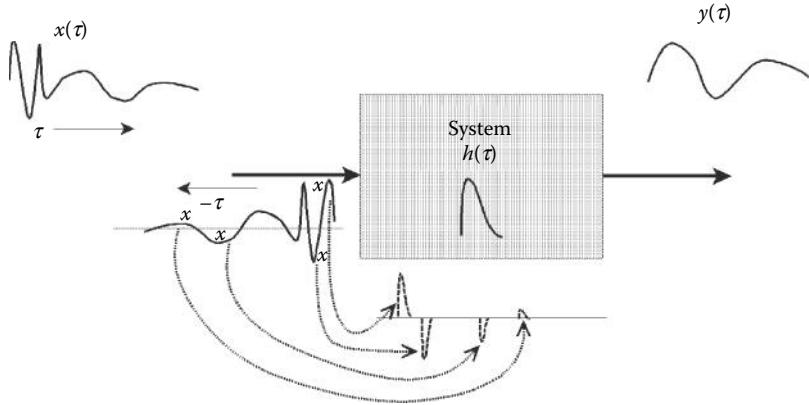


Figure 2.23 An input signal first enters the target system at the lowest (or most negative) time value. So the first impulse response generated by the input is from the left-most signal segment. As it proceeds in time through the system backwards, it generates a series of impulse responses scaled and shifted by the associated signal segment.

When implementing convolution, we have to reverse the input signal because of the way signals are graphed. The left side of the signal is actually the low-time side; it is the first portion to enter the system. So the left-most segment of the input signal produces the first impulse response and the input signal proceeds through the system backward from the way it is plotted (Figure 2.23).

The convolution process is stated mathematically as

$$y[n] = \sum_{k=0}^{K-1} h[k]x[n-k] \quad (2.52)$$

where n is the variable that slides the reversed input, $x[-k]$, across the impulse response, $h[k]$, and K is the length of the shorter function, usually $h[k]$. Equation 2.52 is called the *convolution sum*. The convolution sum, Equation 2.52, is the same as the cross-correlation equation (cf. Equation 2.47) except for the normalization and the minus sign. Despite this similarity, the intent of the two equations is completely different: cross-correlation compares two functions while convolution provides the output of an LTI system given its input and its impulse response. Since it really does not matter which function is shifted, the convolution sum can also be written as

$$y[n] = \sum_{k=0}^{K-1} x[k]h[n-k] \equiv x[k] * h[k] \quad (2.53)$$

This equation also shows a shorthand representation for convolution using the “ $*$ ” symbol. The problem is that the “ $*$ ” symbol is universally used to indicate multiplication in computer programs, including MATLAB, so its use to represent convolution is confusing. Fortunately, this shorthand representation is falling into disuse and it will not be used again here.

In the continuous domain, the summation becomes integration leading to the *convolution integral*:

$$y(t) = \int_{-\infty}^{\infty} h(t - \tau)x(\tau)d\tau = \int_{-\infty}^{\infty} h(\tau)x(t - \tau)d\tau \quad (2.54)$$

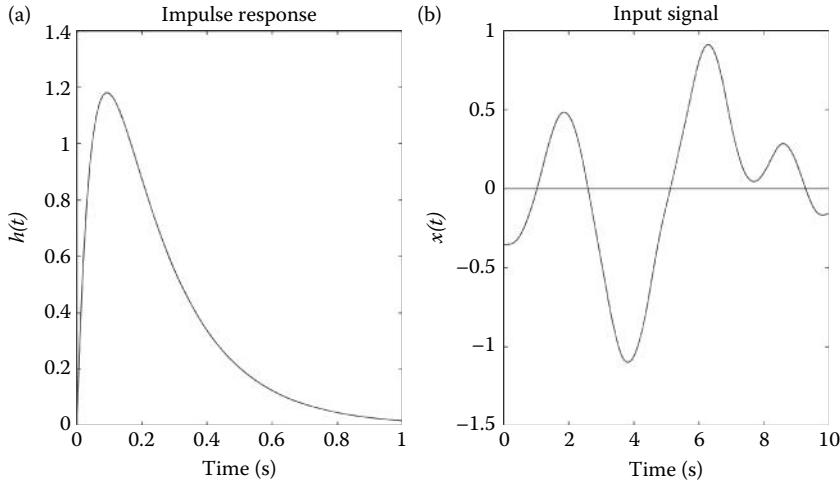


Figure 2.24 (a) The impulse response to a hypothetical system used to illustrate convolution. (b) The input signal to the hypothetical system. The impulse response is much shorter than the input signal (note the 1.0 versus 10 s time scales) as is usually the case.

An example of the convolution process is given in Figures 2.24 through 2.26. Figure 2.24a shows the impulse response of an example system and Figure 2.24b shows the input signal, $x[k]$. Note the two figures have different time scales: the impulse response is much shorter than the input signal, as is generally the case. In Figure 2.25a, the impulse responses to four signal samples at the equivalent of 2, 4, 6, and 8 s are shown. Each segment produces an impulse response which will be added to all the other impulse responses generated by all the other signal samples to produce the output signal. Each impulse response is scaled by the amplitude of the input signal at that time, and is reversed as in Equation 2.52. Some responses are larger, some negative, but they all have the reverse of the basic shape shown in Figure 2.24a. We begin to get

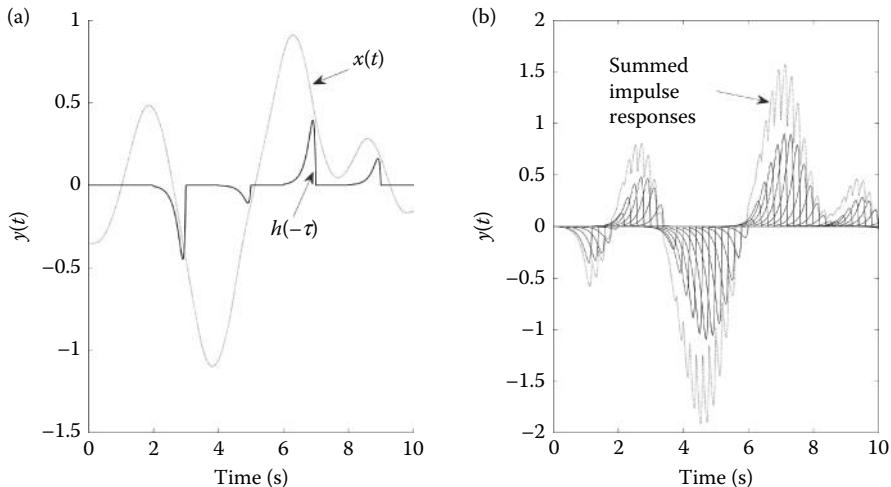


Figure 2.25 (a) Four impulse responses to instantaneous segments of the input signal ($x(t)$ in Figure 2.24a) at 2, 4, 6, and 8 s. (b) Impulse responses from 50 evenly spaced segments of $x(t)$ along with their summation. The summation begins to look like the actual output response.

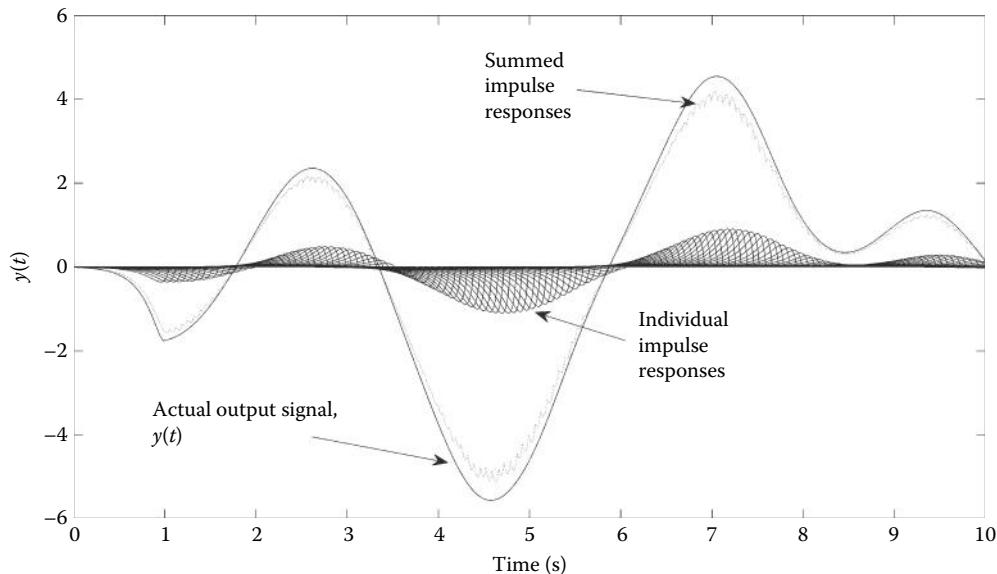


Figure 2.26 The reversed impulse responses of 150 evenly spaced segments of $x(t)$ along with the summation and the actual output signal.

the picture in Figure 2.25b where the impulse responses to 50 infinitesimal segments are shown. The summation of all these responses is also shown as a dashed line. This summation begins to look like the actual output. In Figure 2.26, 150 impulse responses are used, and the summation of all these impulse responses (dotted line) now looks quite similar to the actual output signal (solid line). (The output signal is scaled down to aid comparison.) Convolution of a continuous signal would require an infinite number of segments, but the digital version is limited to one for each data sample. (The input signal used in these figures has 1000 points.)

Convolution is easy to implement in MATLAB:

```
y = conv(h, x); % Convolution sum
```

There are some options to control how many data points are generated by the convolution, but the default produces `length(h) + length(x) - 1` data points. Usually only the first `length(x)` points are used and the additional points are ignored. There is an alternative command for implementing convolution presented in Chapter 4 that does not generate additional points.

EXAMPLE 2.12

Construct an array containing the impulse response of a first-order process. The impulse response of a first-order process is given by the equation: $h(t) = e^{-t/\tau}$ (scaled for unit amplitude). Assume a sampling frequency of 500 Hz and a time constant, τ , of 1 s. Use convolution to find the response of this system to a unit step input. (A unit step input jumps from 0.0 to 1.0 at $t = 0$.) Plot both the impulse response and the output to the step input signal. Repeat this analysis for an impulse response with a time constant of 0.2 s (i.e., $\tau = 0.2$ s).

Solution

The most difficult part of this problem is constructing the first-order impulse response, the discrete function $h[k]$. To adequately represent the impulse response, a decaying exponential,

we make the function at least 5 time constants long, the equivalent of 5 s. Thus, we need an array that is $N = T_p/T_s = T_p(f) = 5(500) = 2500$ points. A 2500-sample time vector scaled by f_s is used to generate the impulse response as well as in plotting. The step function is just an array of ones 2500 points long. Convolving this input with the impulse response and plotting is straightforward.

```
% Example 2.12 Convolution of a first-order system with a step
%
fs=500; % Sample frequency
N=2500; % Construct 5 seconds worth of data
t=(0:N-1)/fs; % Time vector 0 to 5
tau=1; % Time constant
h=exp(-t./tau); % Construct impulse response
x=ones(1,N); % Construct step stimulus
y=conv(x,h); % Get output (convolution)
subplot(1,2,1);
plot(t,h); % Plot impulse response
.....label and title.....
subplot(1,2,2);
plot(t,y(1:N)); % Plot the step response
.....label and title.....
```

Results

The impulse and step responses of a first-order system are shown in Figure 2.27. Note that MATLAB's conv function generates more additional samples (in fact, 4999 points) so only the first 2500 points are used in plotting the response.

In signal processing, convolution can be used to implement the basic filters described in Chapter 4. Like their analog counterparts, digital filters are linear processes that modify the input spectra in some desired way, for example, to reduce noise. As with all linear processes, the

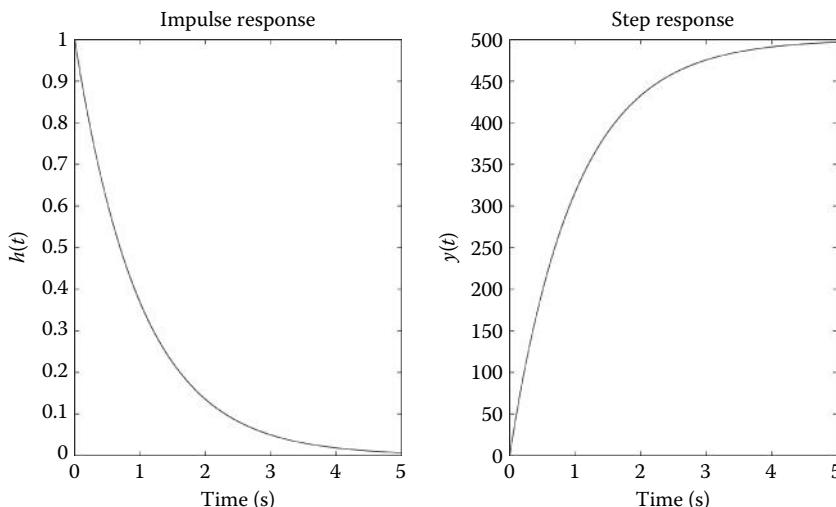


Figure 2.27 The impulse response of a first-order system and the output of this system to a step input. These plots are produced by the code in Example 2.12.

Biosignal and Medical Image Processing

filter's impulse response, $h[k]$ or $h(t)$, completely describes the filter. In Chapter 4, we learn to design $h[k]$ to implement a desired frequency characteristic.

2.4 Summary

Biosignals fall into a number of different categories which require different signal-processing approaches. Signals that are linear and time invariant, that is signals produced by a linear time-invariant (LTI) system, have the widest variety and most powerful set of processing tools. Much of this book is devoted to those tools that apply only to LTI signals. Signals are often assumed to be in, or forced into, the LTI category to take advantage of the powerful tools available. For example, a signal that is nonstationary, that is a signal that changes its basic characteristics over time, might be divided into shorter segments, each of which more-or-less fits the LTI model. A few special tools exist for signals that are either nonlinear or nonstationary, but not both. These tools are generally less powerful and may require longer data sets.

By definition signal is what you want and noise is everything else; all signals contain some noise. The quality of a signal is often specified in terms of a ratio of the signal amplitude or power, to noise amplitude or power, a ratio termed the signal-to-noise ratio (SNR), frequently given in dB. Noise has many sources and only electronic noise can be easily evaluated based on known characteristics of the instrumentation. Since electronic noise exists at all frequencies, reducing the frequency ranges of a signal is bound to reduce noise. That provides the motivation for filtering signals, as is detailed in Chapter 4.

Several basic measurements that can be applied to signals include the mean, RMS value, standard deviation, and variance, all easily implemented in MATLAB. While these measurements may give some essential information, they do not provide much information on signal content or meaning. A common approach to obtain more information is to probe a signal by correlating it with one or more reference waveforms. One of the most popular probing signals is the sinusoid, and sinusoidal correlation is covered in detail in Chapter 3. If signals are viewed as multi-dimensional vectors, then correlation between signals is the same as projecting their associated vectors on one another. Zero correlation between a signal and its reference does not necessarily mean they have nothing in common, but it does mean the two signals are mathematically orthogonal. Multiple correlations can be made between a signal and a family of related functions, known as a basis. A particularly useful basis is harmonically related sinusoids, and correlation with this basis is explored in the next chapter. Bases that are orthogonal are particularly useful in signal processing because when they are used in combination, each orthogonal signal can be treated separately: it does not interact with other family members.

If the probing signal is short or the correlation at a number of different relative positions is desired, a running correlation known as cross-correlation is used. Cross-correlation not only quantifies the similarity between the reference and the original, but also shows where that match is greatest. A signal can be correlated with shifted versions of itself, a process known as autocorrelation. The autocorrelation function describes the time period for which a signal remains partially correlated with itself, and this relates to the structure of the signal. A signal consisting of random Gaussian noise decorrelates immediately, whereas a slowly varying signal will remain correlated over a longer period. Correlation, cross-correlation, autocorrelation, and the related covariances are all easy to implement in MATLAB.

Convolution is a time-domain technique for determining the output of an LTI system in response to any general input. The approach is based on the system's impulse response: the impulse response is used as a representation of the system's response to an infinitesimal segment, or single data point, of the input. For an LTI system superposition holds and the impulse response from each input segment can be summed to produce the system's output. The convolution sum (or convolution integral for continuous signals) is basically a running correlation between the input signal and the reversed impulse response. This integration can be cumbersome

for continuous signals, but is easy to program on a computer. Convolution is commonly used in signal processing to implement digital filtering, as is shown in Chapter 4.

PROBLEMS

- 2.1 The operation of some processes is simulated by MATLAB functions `unknown.m` and `unknown1.m` found in the files associated with the chapter. The input to the processes represented by these functions are the functions input arguments. Similarly to response of the represented processes are the function's outputs:

```
output = unknown(input) :
```

Plot the output of these processes to a 2-Hz sine wave. Make the total time (T_T) 4 s and use a sample time of your choosing, but make the waveform at least 100 points. Determine as best as you can whether these processes are linear or nonlinear processes. [Hint: It is a challenge to determine if a process is truly linear, but if the output signal is proportional to the input signal over a range of input signal amplitudes, the process is linear over that range. Input the 2-Hz sine wave having a range of amplitudes (say, 1–1000) and plot the corresponding output amplitudes. You can use `max(output) – min(output)` to find the output amplitude. If it is proportional and plots as a straight line, the process is probably linear.]

- 2.2 The file `temp_cal.mat` contains the matrix variable `x` which has the output of a temperature transducer in the first column and the actual temperature in °C in the second column. The transducer is known to be linear. The file `temp_out.mat` is the output of this transducer over a 2-day period. This output is in variable `y` and is sampled every 1 min. Write a MATLAB program to take the data in `temp_out.mat` and plot actual temperature in °C as a function of time. The horizontal axis should be in hours. Correctly label the axes, as should be done for all MATLAB plots. [Hint: First find the equation for this transducer system that is equivalent to Equation 2.1 by plotting column 1 against column 2. Note that, unlike the transducer described by Equation 2.1, the transducer in this problem does have an offset; that is, b in Equation 2.2 is not 0.0. Use this equation in your program to decode the data in `temp_out` and then plot using an appropriate horizontal axis.]
- 2.3 Use Equation 2.14 to *analytically* determine the RMS value of a “square wave” with an amplitude of 1.0 V and a period 0.2 s.

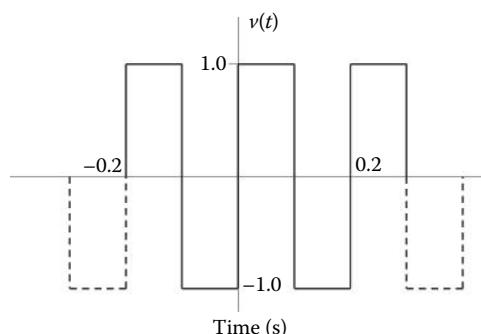


Figure P2.3 Square wave used in Problem 2.3.

Biosignal and Medical Image Processing

- 2.4 Generate one cycle of the square wave similar to the one shown above in a 500-point MATLAB array. Determine the RMS value of this waveform using Equation 2.13. [Hint: When you take the square of the data array, be sure to use a period before the up arrow so that MATLAB does the squaring point-by-point (i.e., $x.^2$).]
- 2.5 Use Equation 2.14 to *analytically* determine the RMS value of the waveform shown below with amplitude of 1.0 V and a period 0.5 s. [Hint: You can use the symmetry of the waveform to reduce the calculation required.]
- 2.6 Generate the waveform shown for Problem 2.5 above in MATLAB. Use 1000 points to produce one period. Take care to determine the appropriate time vector. (Constructing the function in MATLAB is more difficult than the square wave of Problem 2.3, but can still be done in one line of the code.) Calculate the RMS value of this waveform as in Problem 2.3. Plot this function to ensure you have constructed it correctly.
- 2.7 For the waveform of Problem 2.6, calculate the standard deviation using MATLAB's `std` and compare it to the RMS value determined using Equation 2.13.
- 2.8 Load the signal used in Example 2.2 found in file `data_c1.mat`. This signal was shown to be nonstationary in that example. Apply MATLAB's `detrend` operator and evaluate the detrended signal by calculating the means and variances of signal segments as in Example 2.2. Is the modified signal now stationary?
- 2.9 If a signal is measured as 2.5 V and the noise is 28 mV (28×10^{-3} V), what is the SNR in dB?
- 2.10 A single sinusoidal signal is found with some noise. If the RMS value of the noise is 0.5 V and the SNR is 10 dB, what is the RMS amplitude of the sinusoid?
- 2.11 The file `signal_noise.mat` contains a variable `x` that consists of a 1.0-V peak sinusoidal signal buried in noise. What is the SNR for this signal and noise? Assume that the noise RMS is much greater than the signal RMS.
- 2.12 An 8-bit ADC converter that has an input range of ± 5 V is used to convert a signal that ranges between ± 2 V. What is the SNR of the input if the input noise equals the quantization noise of the converter? [Hint: Refer back to Equation 1.8 to find the quantization noise.]
- 2.13 The file `filter1.mat` contains the spectrum of a fourth-order lowpass filter as variable `x` in dB. The file also contains the corresponding frequencies of `x` in

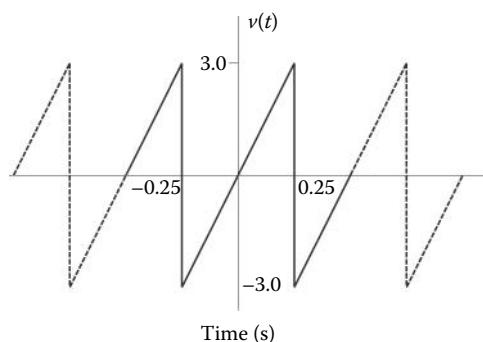


Figure P2.5 Waveform used in Problem 2.5.

- variable `freq`. Plot the spectrum of this filter both as dB versus log frequency and as linear amplitude versus linear frequency. The frequency axis should range between 10 and 400 Hz in both plots. [Hint: Use Equation 2.23 to convert.]
- 2.14 Generate a 10,000-point data set, where each value is the average of four random numbers produced by `rand`, the uniform random number generator. Plot the histogram of this data set as a bar plot. Note how this distribution function closely resembles a Gaussian distribution despite the fact that the original noise source had uniform distribution.
 - 2.15 Repeat the analysis of Example 2.3 with different data lengths. Specifically, use $N = 100, 500, 1000$, and 50,000. Use only the `randn` function.
 - 2.16 A resistor produces $10 \mu\text{V}$ noise (i.e., $10 \times 10^{-6} \text{ V}$ noise) when the room temperature is 310 K and the bandwidth is 1 kHz (i.e., 1000 Hz). What current noise would be produced by this resistor?
 - 2.17 The noise voltage out of a $1-\text{M}\Omega$ (i.e., $10^6-\Omega$) resistor is measured using a digital voltmeter as $1.5 \mu\text{V}$ at a room temperature of 310 K. What is the effective bandwidth of the voltmeter?
 - 2.18 A 3-ma current flows through both a diode (i.e., a semiconductor) and a $20,000-\Omega$ (i.e., $20-\text{k}\Omega$) resistor. What is the net current noise, i_n ? Assume a bandwidth of 1 kHz (i.e., $1 \times 10^3 \text{ Hz}$). Which of the two components is responsible for producing the most noise?
 - 2.19 Verify Equation 2.28 using simulated data. Generate two 1000-point random waveforms using `randn`. Make one waveform twice the amplitude of the other. Calculate the RMS of each, then add the two together and determine the RMS of this sum. Compare the RMS of the sum with the theoretical value determined by taking the square root of the sum of the two individual RMS values squared as predicted by Equation 2.28.
 - 2.20 Modify Example 2.6 to test if a sine wave and cosine wave, both having a frequency, $f = 1.5 \text{ Hz}$, are orthogonal. Make $T_T = 3 \text{ s}$ and $T_s = 0.01 \text{ s}$.
 - 2.21 Modify the approach used in Example 2.5 to find the angle between short signals:
- $$x = [6.6209 \ 9.4810 \ 9.8007 \ 7.4943 \ 3.1798 \ -1.9867].$$
- $$y = [4.3946 \ 4.9983 \ 4.2626 \ 2.3848 \ -0.1320 \ -2.6134].$$
- Do not attempt to plot these vectors as it would require a 5-dimensional plot!
- 2.22 Determine if the two signals, `x` and `y`, in file `correl1.mat` are correlated. Equation 2.35 is not normalized (it is just the sum) so if it results in a modest value, it may not be definitive with respect to orthogonality. Best to check to see if the angle between them is close to 90° .
 - 2.23 Use basic correlation, Equation 2.30, to test if two sine waves, having frequencies of 1 and 1.5 Hz, are orthogonal. Make $T_T = 3 \text{ s}$ and $T_s = 0.01 \text{ s}$.
 - 2.24 Repeat Problem 2.22 but solve for the Pearson correlation coefficient to determine if the two variables are orthogonal.
 - 2.25 Use Equation 2.30 to find the normalized correlation between a cosine and a square wave as shown below. This is the same as Example 2.6 except that the sine has been replaced by a cosine and the mean is taken.

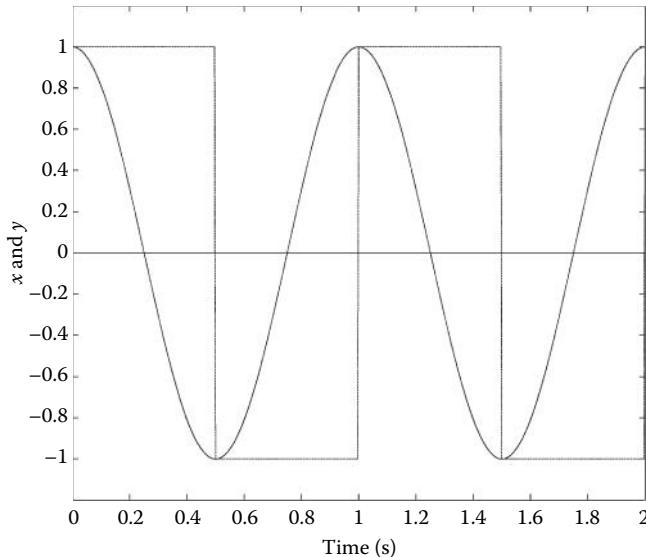


Figure P2.25 Waveforms wave used in Problem 2.25.

- 2.26 Use Equation 2.30 to find the correlation between the two waveforms shown below. [Hint the `saw` routine found with other files for this chapter can be used to generate one of the signals. See `help saw`.]
- 2.27 The file `sines.mat` contains two 10 Hz sinusoids in variable `x` and `y`. The two sinusoids are shifted by 30°. Use MATLAB and cross-correlation to find the time delay between these sinusoids. Plot the cross-correlation function and display the maximum correlation and the shift in seconds. Assume a sample frequency of 2 kHz. (Note if you use `axcor`, the function will be scaled as Pearson's correlation coefficients.) [Hint. Follow the approach used in Example 2.9 to get the time shift.]

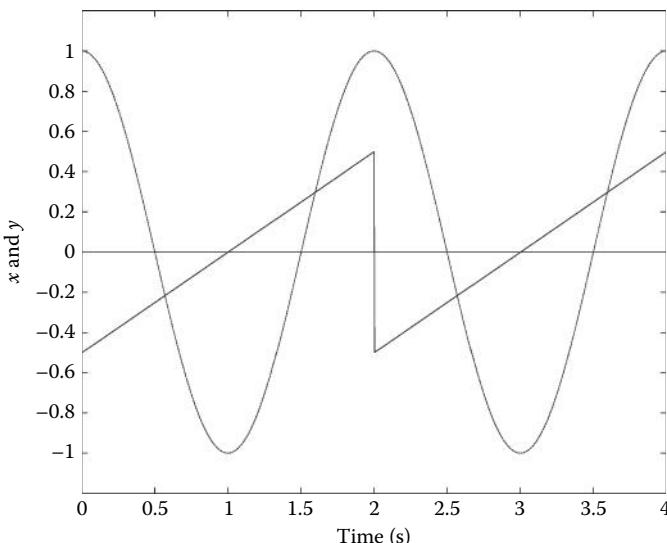


Figure P2.26 Waveforms wave used in Problem 2.26

- 2.28 Use MATLAB and cross-correlation to find the *phase shift* between 10-Hz sinusoids found as variables x and y in file `sines1.mat`. Assume a sample frequency of 2 kHz. Plot the cross-correlation function and find the lag at which the maximum (or minimum) correlation occurs. [Hint: Determine the time shift using the approach in Example 2.9. To convert that time shift into a phase shift, note that the period of $x[n]$ is 1/10 s and the phase in deg is the ratio of the time delay to the period, times 360.]
- 2.29 The file `two_var.mat` contains two variables x and y . Is either of these variables random? Are they orthogonal to each other? (Use any valid method to determine orthogonality.)
- 2.30 Use autocorrelation and random number routine `randn` to plot the autocorrelation sequence of Gaussian's white noise. Use arrays of 2048 and 256 points to show the effect of data length on this operation. Repeat for both lengths of uniform noise using the MATLAB routine `rand`. Note the difference between Gaussian's and uniform noise.
- 2.31 The file `bandwidths.mat` contains two signals having different bandwidths. The signal in x is narrowband while the signal in y is broadband. Plot and compare the autocorrelation functions of the two signals. Plot only lags between ± 30 to show the initial decrease in correlation. Note the inverse relationship between bandwidth and width of the autocorrelation function.
- 2.32 Modify Example 2.10 to probe the respiratory signal in file `resp1.mat`. The variable `resp` contains a 32-s recording of respiration ($f_s = 125$ Hz). Use cross-correlation to find the maximum correlation of this signal and a number of sinusoids ranging in frequency between 0.05 and 1.25 Hz. (The respiratory signal components are slower than those of the EEG signal so you need to modify the frequency range of the probing sinusoids.) Plot the respiratory signal and the maximum correlation as a function of the sinusoidal frequency. The major peak corresponds to the frequency of the base respiratory rate. Determine the respiratory rate in breaths/min from this peak frequency. (Note: There are 125 (32) = 4000 points in the signal array and since the code in Example 2.10 is not optimized, the program will take some time to run.) Repeat this problem correlating the signal with cosine waves.
- 2.33 The file `prob2_33_data.mat` contains a signal x ($f_s = 500$ Hz). Determine if this signal contains a 50-Hz sine wave and if so over what time periods.
- 2.34 The file `prob2_34_data.mat` contains a variable x that is primarily noise but may contain a periodic function. Plot x with the correct time axis ($f_s = 1$ kHz). Can you detect any structure in this signal? Apply autocorrelation and see if you can detect a periodic process. If so, what is the frequency of this periodic process? It may help to expand the x axis to see detail. The next chapter presents a more definitive approach for detecting periodic processes and their frequencies.
- 2.35 Develop a program along the lines of Example 2.11 to determine the correlation in heart rate variability during meditation. Load file `Hr_med.mat` which contains the heart rate in variable `hr_med` and the time vector in variable `t_med`. Calculate and plot the autocovariance. The result will show that the heart rate under meditative conditions contains some periodic elements. Can you determine the frequency of these periodic elements? A more definitive approach is presented in the next chapter which shows how to identify the frequency characteristics of any waveform.

Biosignal and Medical Image Processing

- 2.36 Construct a 512-point Gaussian noise array, then filter it by averaging segments of three consecutive samples. In other words, construct a new array in which every point is the average of the preceding three points in the noise array: $y[n] = x[n]/3 + x[n - 1]/3 + x[n - 2]/3$. You could write the code to do this, but an easier way is to convolve the original data with a function consisting of three equal coefficients having a value of 1/3, in MATLAB: $h(n) = [1/3 \ 1/3 \ 1/3]$. Construct, plot, and compare the autocorrelation of the original and filtered waveform. Limit the x -axis to ± 20 lags to emphasize the difference. Note that while the original Gaussian data were uncorrelated point to point, the filtering process imposed some correlation on the signal.
- 2.37 Repeat the process in Problem 2.36 using a 10 weight averaging filter; that is, $h(n) = [0.1 \ 0.1 \ 0.1 \ 0.1 \ 0.1 \ 0.1 \ 0.1 \ 0.1 \ 0.1 \ 0.1]$. Note that the increased correlation imposed by this filter. [Hint: Due to the number of weights in this filter, it is much easier to implement using convolution than some sort of averaging code. Simply convolve the signal with $h[n]$.]
- 2.38 Modify Example 2.12 to use a 2500-point uniformly distributed random array as the input. Use convolution to find the response of this system to a random input. Note the similarity between the response to a random input and that of a step.
- 2.39 Modify Example 2.12 to use the impulse response of a second-order, underdamped system. The impulse response of a second-order underdamped system is given by

$$h(t) = \frac{\delta}{\sqrt{1-\delta^2}} e^{-\delta 2\pi f_n t} \sin(2\pi f_n \sqrt{1-\delta^2} t)$$

This is a more complicated equation, but it can be implemented with just a slight modification of Example 2.12. Use a sampling frequency of 500 Hz and set the damping factor, δ , to 0.1 and the frequency, f_n (termed the undamped natural frequency), to 5 Hz. The array should be the equivalent of at least 2.0 s of data (i.e., $N = 1000$). Plot the impulse response to check its shape. Convolve this impulse response with a 1000-point step input signal and plot the output.

- 2.40 Repeat Problem 2.39 but make $f_n = 2$ Hz and use a 1000-point uniformly distributed random waveform as input. Plot the impulse function and the output. Note the similarity with the responses.
- 2.41 Construct four damped sinusoids similar to the signal, $h(t)$, in Problem 2.39. Use a damping factor of 0.04 and generate 2 s of data assuming a sampling frequency of 500 Hz. Two of the four signals should have an f_n of 10 Hz and the other two an f_n of 20 Hz. The two signals at the same frequency should be 90° out of phase (i.e., replace the \sin with a \cos). Are any of these four signals orthogonal? [Hint: You can use `corrcoef` to compare all four signals at the same time.]

3

Spectral Analysis *Classical Methods*

3.1 Introduction

Sometimes, the frequency content of the waveform provides more useful information than the time-domain representation. Many biological signals demonstrate interesting or diagnostically useful properties when viewed in the so-called *frequency domain*. Examples of such signals include heart rate, EMG, EEG, ECG, eye movements and other motor responses, acoustic heart sounds, stomach, and intestinal sounds. In fact, just about all biosignals have, at one time or another, been examined in the frequency domain. Figure 3.1 shows the time response of an EEG signal and an estimate of spectral content using the classical Fourier transform (FT) method described later. Several peaks in the frequency plot can be seen, indicating significant energy in the EEG at these frequencies. Most of these peaks are associated with general neurological states.

Determining the frequency content of a waveform is termed *spectral analysis* and the development of useful approaches for this frequency decomposition has a long and rich history. Spectral analysis decomposes a time-domain waveform into its constituent frequencies just as a prism decomposes light into its spectrum of constituent colors (i.e., specific frequencies of the electromagnetic spectrum).

Various techniques exist to convert time-domain signals into their spectral equivalent. Each has different strengths and weaknesses. Basically, spectral analysis methods can be divided into two broad categories: classical methods based on the Fourier transform (FT), and modern methods such as those based on models of the signal's source. The accurate determination of the waveform's spectrum requires that the signal must be periodic, of finite length, and noise free. Unfortunately, many biosignals are sufficiently long that only a portion is available for analysis. Moreover, biosignals are often corrupted by substantial amounts of noise or artifact (see Section 2.2). If only a portion of the actual signal can be analyzed and/or if the waveform contains noise along with the signal, then all spectral analysis techniques must necessarily be approximate, that is, they are *estimates* of the true spectrum. The modern spectral analyses described in Chapter 5 attempt to improve the estimation accuracy of specific spectral features, but require some knowledge, or guesswork, about the signal content.

An intelligent application of spectral analysis techniques requires an understanding of what spectral features are likely to be of interest and which methods provide the most accurate determination of these features. Two spectral features of potential interest are the overall shape of

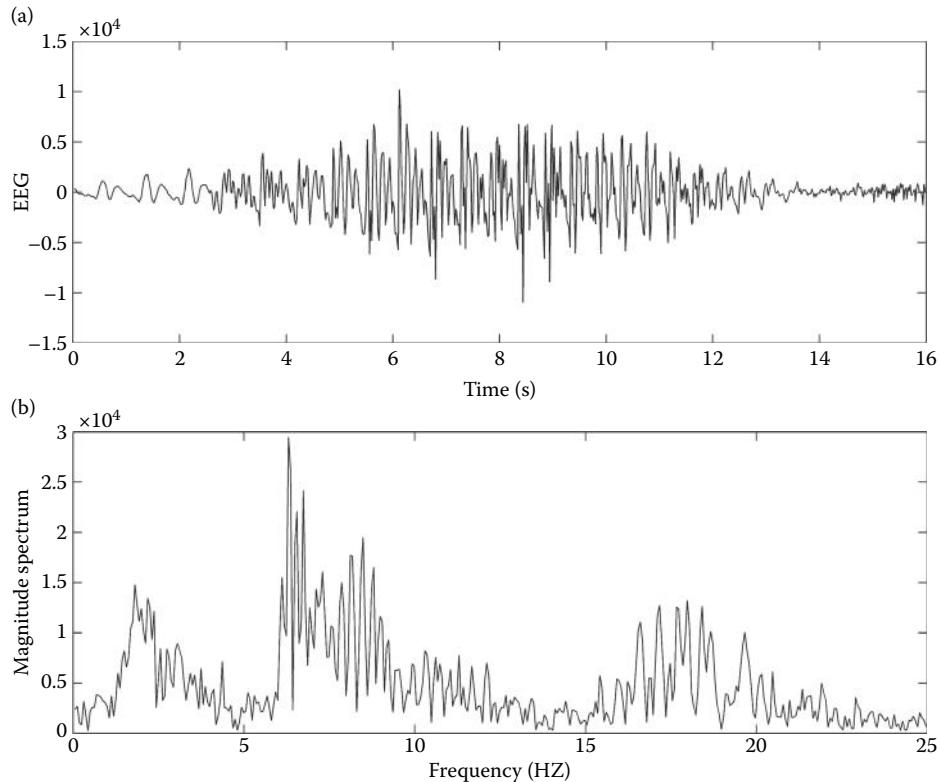


Figure 3.1 (a) Segment of an EEG time-domain signal also shown in Figure 2.16. (b) The equivalent magnitude component of the frequency-domain representation. The peaks observed in this spectrum are known to correspond with certain neurological states. This spectrum is calculated in Example 3.1.

the spectrum, termed the spectral estimate, and/or local features of the spectrum, sometimes referred to as parametric estimates. Local features would include narrowband signals whereas spectral estimates would describe the broadband characteristics of a signal. The techniques that provide good spectral estimation are poor local estimators and vice versa. Fortunately, we are not usually interested in the accurate estimation of both narrowband and broadband features, but if we are, then different analyses will be required.

Figure 3.2a shows the spectral estimate obtained by applying the traditional FT to a waveform consisting of a 100-Hz sine wave buried in white noise. The SNR is -14 dB, that is, the signal amplitude is only $1/5$ of the noise. Note that the 100-Hz sine wave is readily identified as a peak in the spectrum at that frequency, but many other smaller peaks are seen that are due to noise. Figure 3.2b shows the spectral estimate obtained by a smoothing process applied to the same signal. This smoothing operation is called the Welch method and is described later in this chapter. The resulting spectrum provides a more accurate representation of the overall spectral features (predominantly those of the white noise), but the 100-Hz signal is lost. Figure 3.2 shows that the smoothing approach is a good spectral estimator in the sense that it provides a better estimate of the dominant component (i.e., noise), but it is not a good signal detector of local features.

The classical procedures for spectral estimation are described in this chapter with particular regard to their strengths and weaknesses. These methods are easily implemented in MATLAB. Modern methods for spectral estimation are covered in Chapter 5.

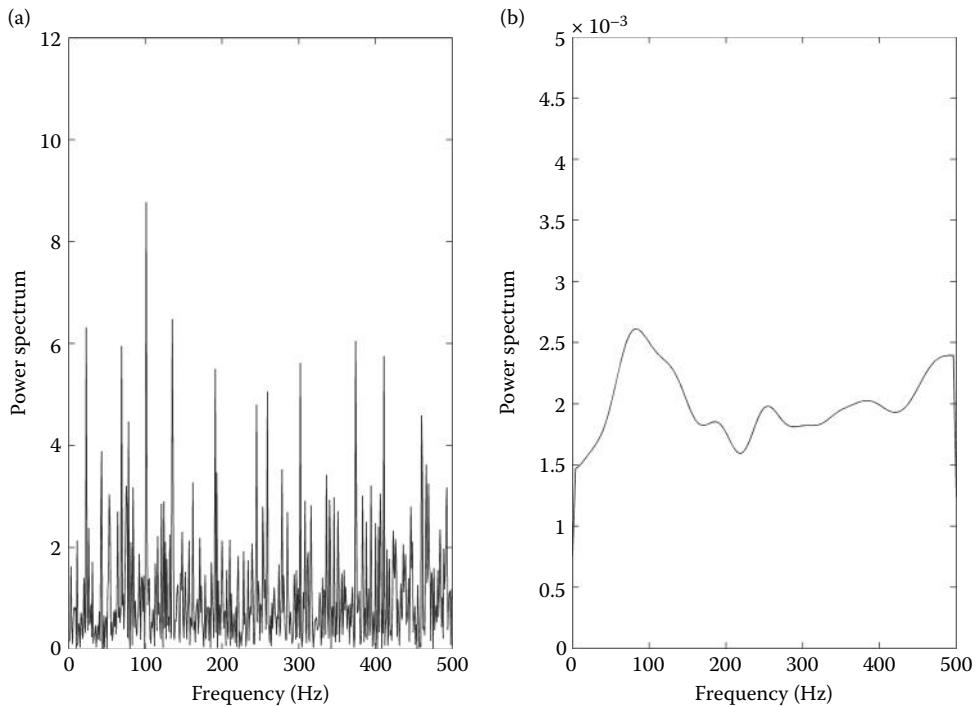


Figure 3.2 Spectra obtained from a waveform consisting of a 100-Hz sine wave and white noise using two different methods. (a) The spectrum produced by the FT clearly shows the signal as a spike at 100 Hz. However, other spikes are present that are just noise. (b) An averaging technique creates a smooth spectrum that represents the background white noise (that should be flat) better, but the 100-Hz component is no longer clearly visible.

3.2 Fourier Series Analysis

Of the many techniques used for spectral estimation, the classical method commonly known as the Fourier transform* is the most straightforward. Essentially, FT approaches use the sinusoid as a gateway between the time and frequency domains. Since sinusoids contain energy at only one frequency, a sinusoid maps to the frequency domain in a very straightforward manner. As shown in Figure 3.3, the amplitude of the sinusoid maps to a single point on the spectrum's magnitude curve and the phase of the sinusoid maps to a single point on the spectrum's phase curve. Although up to now, we have dealt with only the magnitude spectrum, a full spectrum consists of both a magnitude and phase component. Often, only the magnitude spectrum is of interest, but sometimes, we would like to know the phase characteristics. Moreover, the full spectrum is essential if we want to convert back from the frequency domain to the time domain.

If a waveform can be decomposed into sinusoids of different frequencies, this simple mapping procedure can be used to determine the magnitude and phase spectrum of the waveform. This strategy is illustrated in Figure 3.4 for a triangular waveform (left panel) that is decomposed into three sinusoids of increasing frequency (middle panel) that are then mapped to a magnitude and phase spectrum (right panels). In essence, sinusoids form a bridge between the time- and frequency-domain representations of a signal.

* The term “Fourier transform” is a general term that is loosely used to cover several related analysis techniques. These different techniques are summarized in Tables 3.2 and 3.3 along with the proper terminology.

Biosignal and Medical Image Processing

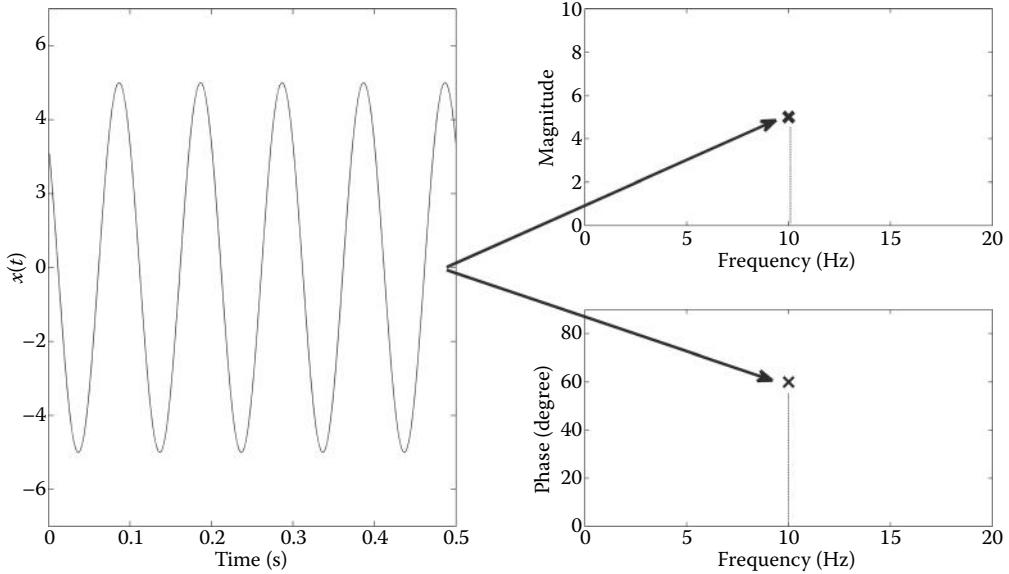


Figure 3.3 Since sinusoids contain energy at only a single frequency, they map to the frequency domain in a very simple manner. The amplitude of the sinusoid maps to a single point on the magnitude spectrum curve and the phase angle of the sinusoid maps to a single point on the phase portion of the spectrum. Both points are at the frequency of the sinusoid, in this case 10 Hz.

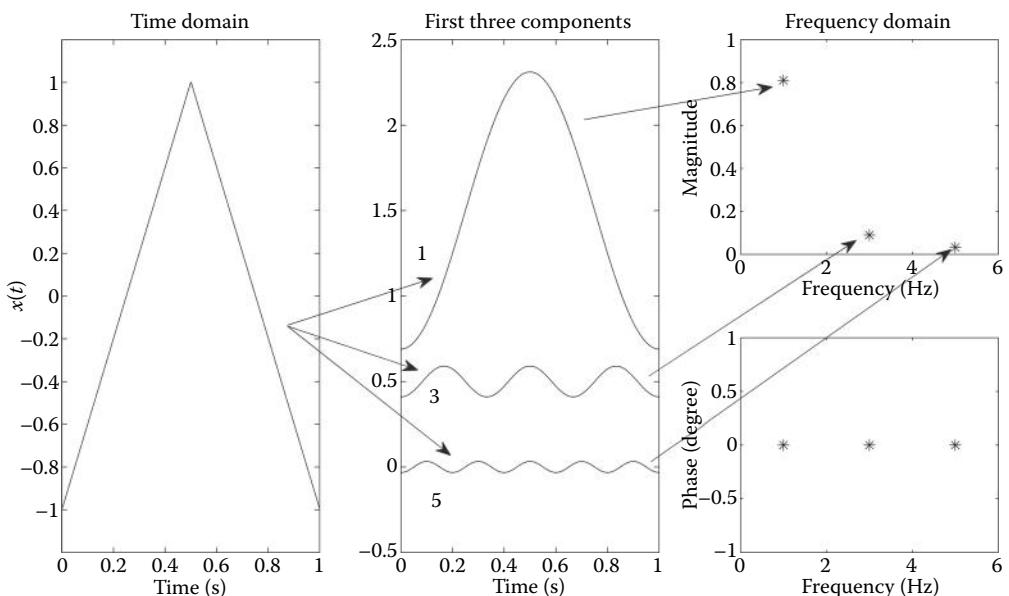


Figure 3.4 Sinusoids can be used as a gateway between a general waveform (left panel) and its frequency representation or spectrum (right panels). The triangular waveform is decomposed into three sinusoids (middle panel). These are then mapped to the magnitude spectrum (arrows) and the phase spectrum.

3.2.1 Periodic Functions

When the waveform of interest is periodic, an approach known as *Fourier series analysis* is used to convert between the time and frequency domains. It uses the “sinusoidal bridge” approach to decompose a general periodic waveform into sinusoids. The foundation of this approach is the *Fourier series theorem*, which states that any periodic waveform, no matter how complicated, can be decomposed into sinusoids that are at the same frequency as, or multiples of, the waveform’s frequency. This family or basis can be expressed as either a series of sinusoids of appropriate amplitude and phase angle or a mathematically equivalent series of sines and cosines. In either case, it is only necessary to project (i.e., correlate) the waveform with the basis (see Equation 2.37). Substituting a basis of sines and cosines for $f_m(t)$ in the continuous version of the correlation equation, Equation 2.37 leads to

$$a[m] = \frac{2}{T} \int_0^T x[t] \cos(2\pi m f_1 t) dt \quad m = 1, 2, 3, \dots \quad (3.1)^*$$

$$b[m] = \frac{2}{T} \int_0^T x[t] \sin(2\pi m f_1 t) dt \quad m = 1, 2, 3, \dots \quad (3.2)$$

where $f_1 = 1/T$, T is the period of the waveform $x(t)$, and m is a set of integers: $m = 1, 2, 3, \dots$ (possibly infinite) defining the family member. This gives rise to a basis of sines and cosines having harmonically related frequencies, mf_1 . If this concept was implemented on a computer, the discrete version of these equations would be used where integration becomes summation, $x(t) \rightarrow x[n]$, and $t \rightarrow nT_s$:

$$a[m] = \sum_{n=1}^N x[n] \cos(2\pi m f_1 n T_s) \quad m = 1, 2, 3, \dots M \quad (3.3)$$

$$b[m] = \sum_{n=1}^N x[n] \sin(2\pi m f_1 n T_s) \quad m = 1, 2, 3, \dots M \quad (3.4)$$

These equations are correctly known under two terms: the *discrete time Fourier series* or *discrete Fourier transform* (DFT).

Note that the sine and cosine frequencies in Equations 3.3 and 3.4 are related to m , the base frequency f_1 or the total time, or the sample interval and number of points, or the sample frequency and number of points. These four relationships are summarized mathematically as

$$f = mf_1 = \frac{m}{T} = \frac{m}{NT_s} = \frac{mf_s}{N} \quad (3.5)$$

For the continuous equations (Equations 3.1 and 3.2), m could go to infinity, but for the discrete equations (Equations 3.3 and 3.4), $m \leq N$. This is because when $m = N$, the period of $\sin(2\pi N f_1 n T_s)$ would be $(1/Nf_1) = (1/f_s)$ that is one sample and a sine wave cannot be represented by less than one sample. Since the number of components cannot possibly exceed the number of points ($m \leq N$), the maximum theoretical frequency would be

$$f_{\text{theoretical max}} = mf_1 = Nf_s = \frac{Nf_s}{N} = f_s \quad (3.6)$$

* These equations normalize by $2/T$, but other normalizations, such as $1/T$, are common. Unfortunately, there is no standard protocol for normalization. In the digital domain, normalization is usually not used as is the case here in Equations 3.3 and 3.4.

Biosignal and Medical Image Processing

when $m = N$, $f = f_s$. However, as described in Section 1.6.2, there must be at least two sample points within a period to define a sine wave. To have two samples within a period, the maximum period with respect to sample interval is

$$2T_s = \frac{2}{f_s} = \frac{2}{Nf_s}$$

which corresponds to a maximum frequency of $(Nf_s/2)$. So, the maximum m must be $\leq N/2$. Hence, from Equation 3.5:

$$f_{\max} < \frac{Nf_s}{2} < \frac{Nf_s}{2N} < \frac{f_s}{2} \quad (3.7)$$

which is Shannon's sampling theorem. So, the maximum frequency component that can be obtained from a digitized signal is less than $f_s/2$ and occurs for all $m < N/2$. Recall that $f_s/2$ is termed the Nyquist frequency.

These equations are the most straightforward examples of the Fourier series analysis equations, but they are not the most commonly used. (The continuous time-domain equations, Equations 3.1 and 3.2, are used in calculations done manually, but such problems are only found in textbooks.) A more succinct equation that uses complex numbers is described later in this chapter. The basis used in Fourier series analysis consists of sines and cosines having frequencies only at discrete values of mf_s , which is either the same frequency as the waveform (when $m = 1$) or higher multiples (when $m > 1$) termed *harmonics*. Thus, the Fourier series analysis decomposes a waveform by projection on the basis of harmonically related sinusoids (or sines and cosines); so, the approach is sometimes referred to as *harmonic decomposition*.

Rather than plotting sine and cosine amplitudes obtained from these equations, it is more intuitive to plot the amplitude and phase angle of a single sinusoidal waveform such as $\cos(2\pi f_s t + \theta)$. This is known as the polar rather than the rectangular representation of the harmonic components. The sine and cosine amplitudes can be converted into a single sinusoid using the well-known rectangular-to-polar equation:

$$a\cos(x) + b\sin(x) = C\cos(x - \theta) \quad (3.8)^*$$

The basic trigonometric identity for the difference of two arguments of a cosine function gives the conversion between the polar representation as a single cosine of magnitude C and angle $-\theta$, into a sine and cosine representation:

$$a = C \cos(\theta) \quad (3.9)$$

$$b = C \sin(\theta) \quad (3.10)$$

The reverse transformation, from sines and cosines to a signal sinusoid, is found algebraically from Equations 3.9 and 3.10 as

$$C = (a^2 + b^2)^{1/2} \quad (3.11)$$

$$\theta = \tan^{-1}\left(\frac{b}{a}\right) \quad (3.12)$$

Note that the θ has a negative sign in Equation 3.8; so, Equation 3.12 actually finds $-\theta$.

* Note that in engineering, it is common to specify the angle, θ , in deg, and the time component, x , in radians. Of course, MATLAB uses radians in all trigonometric functions, but conversion into deg is easy and is commonly done for phase curves.

EXAMPLE 3.1

Use Fourier series analysis to generate the magnitude and phase plot of the ECG signal originally shown in Figure 2.16 and repeated below. For these data, $f_s = 50$ Hz.

Solution

Fourier series analysis requires that the waveform must be periodic; so, we have to assume that the EEG signal is periodic. In other words, the EEG segment is assumed to be one cycle of a periodic signal as illustrated in Figure 3.5. Of course, this is beyond improbable, but it really does not matter since we know nothing about the signal before or after the segment on the computer; so, any assumption about that unknown signal is fine. Once this assumption is made, implementing Equations 3.3 and 3.4 in MATLAB is easy. Then we use Equations 3.11 and 3.12 to convert the sine and cosine components into magnitude and phase plots. We make $M = N/2$ since that corresponds to the maximum valid frequency to be found in the digitized signal (Equation 3.7). (Since this frequency is $f/2$ and $f_s = 50$ Hz, we anticipate a maximum frequency of 25 Hz.)

```
% Example 3.1 Use Fourier series analysis to generate the magnitude and
% phase plots of the ECG signal.
%
fs = 50; % Sample frequency
load eeg_data; % Get data (vector eeg)
N = length(eeg); % Get N
Tt = N/fs; % Calculate total time
f1 = 1/Tt; % Calculate fundamental frequency
t = (1:N)/fs; % Time vector for plotting
for m = 1:round(N/2)
    f(m) = m*f1; % Sinusoidal frequencies
    a = sum(eeg.*cos(2*pi*f(m)*t)); % Cosine coeff., Eq. 3.3
    b = sum(eeg.*sin(2*pi*f(m)*t)); % Sine coeff., Eq. 3.4
```

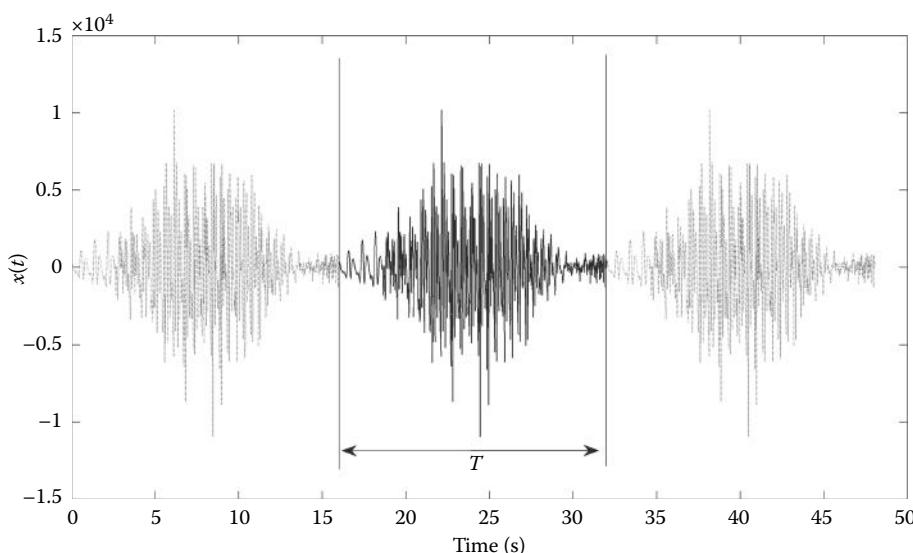


Figure 3.5 The EEG signal, originally shown in Figure 2.16, illustrated here as one period of a fictional periodic signal having period T .

Biosignal and Medical Image Processing

```
X_mag(m) = sqrt(a^2 + b^2); % Magnitude spectrum Eq. 3.6
X_phase(m) = -atan2(b,a); % Phase spectrum, Eq. 3.7
end
X_phase = unwrap(X_phase); % Compensates for shifts > pi
X_phase = X_phase*360/(2*pi); % Convert phase to deg.
subplot(2,1,1);
plot(f,X_mag,"k"); % Plot magnitude spectrum
.....Labels .....
subplot(2,1,2);
plot(f,X_phase,"k"); % Plot phase spectrum
.....Labels .....
```

Analysis

The total time, T_p , of the signal was determined by dividing the number of points, N , by f_s (equivalent to multiplying N by T_s). The fundamental frequency, f_1 , is $1/T_p$. A loop was used to correlate the sine and cosine waves with the EEG signal and their respective components were found by direct application of Equations 3.3 and 3.4. These components were converted into the more useful single sinusoidal representation of magnitude and phase using the transformation equations, Equations 3.6 and 3.7. Note that the component frequencies, mf_1 , are saved in a vector for use in plotting.

While the magnitude spectrum is straightforward, determining the phase curve has some potential pitfalls. First, note that the negative arctangent is taken as θ as given in Equation 3.12. The next problem is in the implementation of the arctangent function. Many computer and calculator algorithms, including MATLAB's atan function, do not take into account the angle quadrant; so, for example, $\tan^{-1}(-b/a)$ gives the same result as $\tan^{-1}(b/-a)$, but the former has an angle between 0 and $-\pi/2$ whereas the latter has an angle between $\pi/2$ and π . MATLAB's atan2 function checks the signs of both the numerator and denominator and adjusts the resultant angle accordingly.

```
angle = atan2(b,a); % Find the arctangent taking the quadrant in account
```

Another potential problem is that by definition, the phase angle is limited to $\pm\pi$ (or $0-2\pi$ but atan2 returns $\pm\pi$). Yet, at some frequency, it is possible, in fact likely, that the phase component will exceed those limits. When this occurs, the phase curve will "wrap around" so that $\pi + \theta$ will have the value of $-\pi + \theta$. If we were dealing with single values, there would be little we can do, but since we are dealing with a curve, we might assume that large point-to-point transitions indicate that wrap around has occurred. One strategy would be to search for transitions greater than some threshold and, if found, reduce that transition by adding $+\pi$ or $-\pi$, whichever leads to the smaller transition. While it would be easy to write such a routine, it is unnecessary as MATLAB has already done it (yet again). The routine unwrap unwraps phase data in radians by changing absolute jumps $\geq\pi$ to their 2π complement.

```
phase_unwrapped = unwrap(phase); % Unwrap phase data
```

After this routine is applied to the phase data, the phase is converted into deg that are more conventionally used in engineering. The magnitude and phase curves are then plotted as functions of frequency.

Results

The plots generated in this example are shown in Figure 3.6. The magnitude plot is roughly similar to the plot in Figure 2.16 produced in Example 2.10 by taking the maximum cross-correlation between sines and the signal. However, in that example, the sinusoidal frequencies

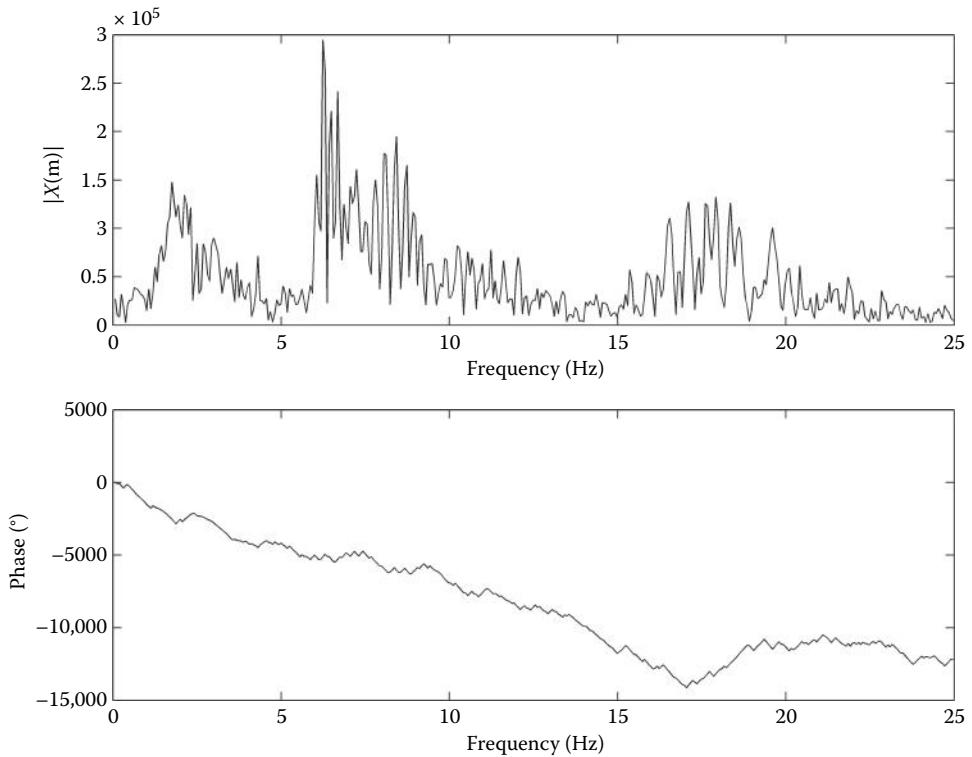


Figure 3.6 The magnitude and phase spectrum of the EEG signal shown in Figure 3.5 as determined by projecting the EEG signal on a basis of harmonically related sines and cosines ($f = mf_1$ where $m = 1, 2, 3, \dots$). Special attention is paid to the construction of the phase plot. The magnitude spectrum is also shown in Figure 3.1.

were selected arbitrarily (0.25–25 Hz in increments of 0.25 Hz) and were not related to the fundamental frequency of the EEG signal. If we check the values of Tt and f_1 in this example, we find the total time of EEG is 16.02 s and the related fundamental frequency is 0.0624 Hz. So, a major advantage in using Fourier series decomposition is that it tells you exactly what sinusoidal frequencies might exist in the waveform: specifically, the fundamental frequency, f_1 , and higher harmonics, mf_1 . This assumes that the signal is periodic but, as stated above, that assumption must be made to apply Fourier analysis to a digital signal.

There is another advantage in using Fourier series analysis as opposed to probing with sinusoids at arbitrary frequencies. The code in Example 3.1 is much faster than the code in Example 2.10. In fact, it could be made even faster by using a special algorithm that employs complex numbers and breaks the signal into smaller time segments. This fast algorithm is known, predictably, as the *fast Fourier transform* or commonly as the FFT.

The Fourier series analysis is part of an invertible transform: Fourier analysis converts a time function into the frequency domain as equivalent sinusoidal components and the frequency-domain Fourier sinusoidal components can be summed to reconstruct the time-domain waveform. In the continuous domain, the equations become:

$$x(t) = \frac{a(0)}{2} + \sum_{m=1}^{\infty} a[m] \cos(2\pi m f_1 t) + \sum_{m=1}^{\infty} b[m] \sin(2\pi m f_1 t) \quad (3.13)$$

Biosignal and Medical Image Processing

where the $a[m]$ and $b[m]$ are defined in Equations 3.1 and 3.2 and the $a(0)$ component is used to add in any nonzero DC component. The DC component is evaluated as twice the mean of the $x(t)$:

$$a(0) = \frac{2}{T} \int_0^T x(t) dt \quad (3.14)$$

The normalization is by $2/T$ to correspond to that used in Equations 3.1 and 3.2. Note that the “2” is divided back in Equations 3.13 and 3.15. Using the transformation equations given in Equations 3.11 and 3.12, the harmonic series in Equation 3.13 can also be expressed in terms of a single sinusoid family with members having different amplitudes and phases:

$$x(t) = \frac{a(0)}{2} + \sum_{m=1}^{\infty} C(m) \cos(2\pi m f_1 t + \theta(m)) \quad (3.15)$$

The operation represented by Equation 3.13 or Equation 3.15 is often termed the *inverse Fourier transform*, particularly when referring to the discrete operations. A discrete version of these equations is presented in Section 3.2.2.

Most simple waveforms can be well approximated by only a few family members. This is illustrated in the next example that decomposes a triangular waveform and then reconstructs it with limited components.

EXAMPLE 3.2

Perform a discrete Fourier series analysis on the triangular waveform defined by the equation:

$$x(t) = \begin{cases} t & 0 \leq t < 0.5 \text{ s} \\ 0 & 0.5 \leq t < 1.0 \text{ s} \end{cases}$$

Reconstruct the waveform using the first five and then the first 10 components. Plot the reconstructed waveforms superimposed on $x(t)$. Make $f_s = 500$ Hz.

Solution

After constructing the waveform in MATLAB, decompose the waveform into 10 components using the approach featured in Example 3.1. Note that the total time of the waveform is 1.0 s. Then reconstruct the two waveforms using the digital version of Equation 3.15. However, if we use Equation 3.15 for reconstruction, then we need to normalize in a manner equivalent to that used in Equations 3.1 and 3.2, that is, by $2/N$. Note that the waveform does have a DC term that should be added to the reconstructions.

```
% Example 3.2 Fourier series decomposition and reconstruction.
%
fs = 500; % Sampling frequency
Tt = 1; % Total time
N = Tt*fs; % Determine N
f1 = 1/Tt; % Fundamental frequency
t = (1:N)/fs; % Time vector
x = zeros(1,N); % Construct waveform
x(1:N/2) = t(1:N/2);
% Fourier decomposition
a0 = 2*mean(x); % Calculate a(0)
```

3.2 Fourier Series Analysis

```

for m=1:10
    f(m) = m*f1;           % Sinusoidal frequencies
    a= (2/N)*sum(x.*cos(2*pi*f(m)*t));      % Cosine coeff., Eq. 3.3
    b= (2/N)*sum(x.*sin(2*pi*f(m)*t));      % Sine coeff., Eq. 3.4
    X_mag(m) = sqrt(a^2+b^2);                 % Magnitude spectrum Eq. 3.6
    X_phase(m) = -atan2(b,a);                  % Phase spectrum, Eq. 3.7
end
x1=zeros(1,N);          % Reconstruct waveform
for m=1:5
    f(m) = m*f1;           % Sinusoidal frequencies
    x1=x1+X_mag(m)*cos(2*pi*f(m)*t+X_phase(m)); % Eq. 3.15
end
x1=x1+a0/2;            % Add in DC term
subplot(2,1,1);
plot(t,x1,'k'); hold on;
plot(t,x,'-k');        % Plot reconstructed and original waveform
.....labels and title.....
.....Repeat for 10 components.....

```

Results

Figure 3.7 shows the original triangular waveform (dashed lines) decomposed using the Fourier series analysis and reconstructed using only five or 10 components plus the DC term. The reconstruction using 10 components is reasonably close, but shows oscillations. Such oscillations will occur whenever there is a discontinuity in the waveform and a limited number of components is used to reconstruct the waveform.

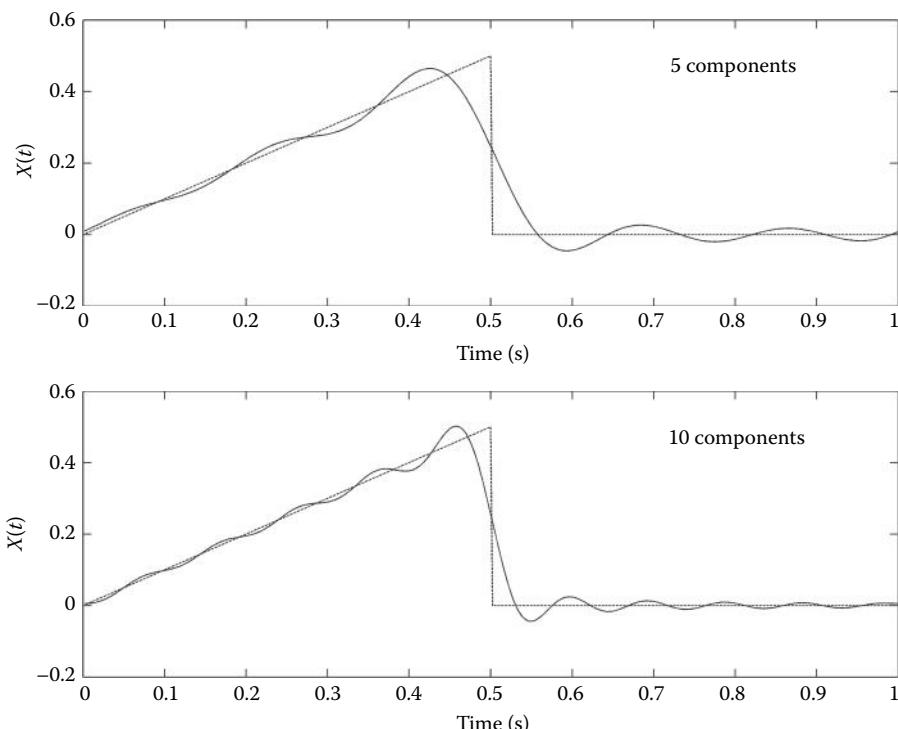


Figure 3.7 A triangular waveform (dashed lines) decomposed using the Fourier series analysis and reconstructed using only 5 or 10 components plus the DC term. The oscillations, known as the Gibbs oscillations, occur when limited components are used to reconstruct a waveform.

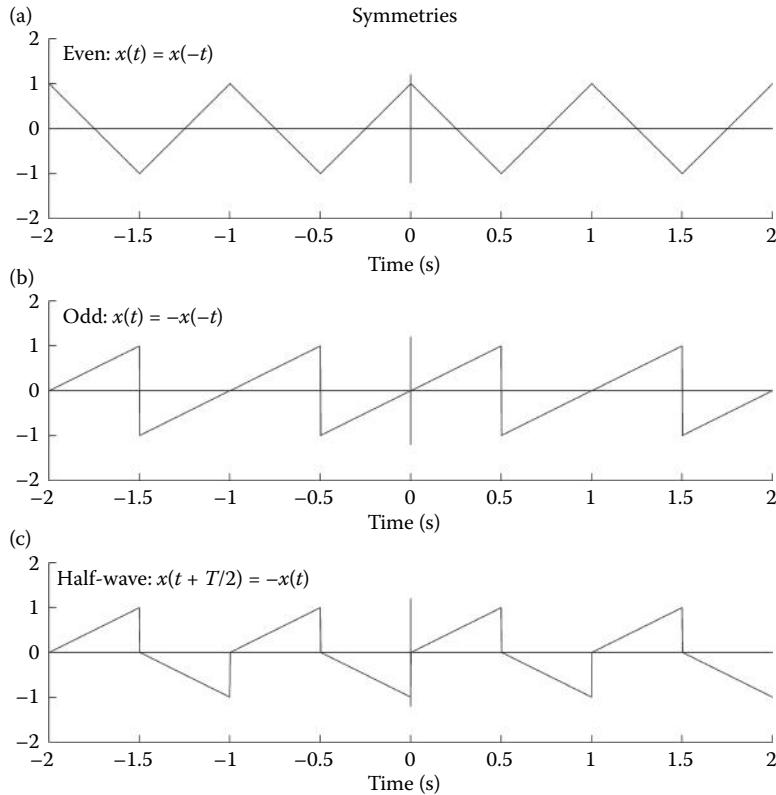


Figure 3.8 Three waveforms illustrating symmetries useful in the Fourier series analysis. Each waveform has a period of 1.0 s. (a) Even symmetry, (b) odd symmetry, and (c) half-wave symmetry.

3.2.1.1 Symmetry

Some waveforms are symmetrical or antisymmetrical about $t = 0$, so that one or the other of the components, a_m or b_m in Equations 3.1 and 3.2, will be zero. Specifically, if the waveform has mirror symmetry about $t = 0$, that is, $x(t) = x(-t)$ (Figure 3.8a), then multiplications with all sine functions will integrate or sum to zero so that the b_m terms will be zero. Such mirror symmetry functions are termed *even* functions. If the function has antisymmetry, $x(t) = -x(-t)$, (Figure 3.8b), it is termed an *odd* function and all multiplications with cosines will integrate or sum to zero so that the a_m coefficients will be zero. Finally, functions that have *half-wave* symmetry will have no even coefficients; so, both a_m and b_m will be zero for even m . These are functions where the second half of the period looks like the first half, but is inverted, that is, $x(t + T/2) = -x(t)$ (Figure 3.8c). These functions can be thought of as having the first half shifted to become the second half and then inverted. Functions having half-wave symmetry are also even functions. These symmetries are useful for simplifying the task of solving for the coefficients manually, but are also useful for checking solutions done on a computer. Table 3.1 and Figure 3.8 summarize these properties.

3.2.2 Complex Representation

The equations for computing the Fourier series analysis of digitized data are usually presented using complex variables, as this leads to more succinct equations. In addition, the FFT is coded using this representation. By using complex variables notation, the sine and cosine terms of Equations 3.3 and 3.4 can be represented by a single exponential term using Euler's identity:

$$e^{jx} = \cos x + j\sin x \quad (3.16)$$

Table 3.1 Function Symmetries

Function Name	Symmetry	Coefficient Values
Even	$x(t) = x(-t)$	$b[m] = 0$
Odd	$x(t) = -x(-t)$	$a[m] = 0$
Half-wave	$x(t) = -x(t+T/2)$	$a[m] = b[m] = 0$; for m even

(Note: Mathematicians use “ i ” to represent $\sqrt{-1}$, whereas engineers use “ j ” since “ i ” is reserved for current.) Using complex notation, the equation for the continuous FT can be derived from Equations 3.1, 3.2, and 3.16 using only algebra:

$$X(f) = \int_0^T x(t)e^{-j2\pi ft} dt \quad f = -\infty, \dots, -2f_1, -f_1, 0, f_1, 2f_1, \dots, \infty \quad (3.17)$$

where the variables are defined as in Equations 3.1 and 3.2. Again, the frequency is obtained from the value of m , that is, $f = mf_1$ where $f_1 = 1/T$. In discrete form, the integration becomes summation and $t \rightarrow nT_s$,

$$X[m] = \sum_{n=1}^N x[n]e^{(-j2\pi m f_1 n T_s)} \quad m = -N/2, \dots, -1, 0, 1, \dots, N/2 \quad (3.18)$$

This is the complex form of the DFT or discrete time Fourier series. An alternative equation can be obtained by substituting $f_1 = 1/T = 1/NT_s$ for f_1 in Equation 3.18:

$$X[m] = \sum_{n=1}^N x[n]e^{(-j2\pi mn T_s/NT_s)} = \sum_{n=1}^N x[n]e^{(-j2\pi mn/N)} \quad (3.19)$$

This is the common form of the discrete Fourier series analysis equations; it is the format found in most publications that use this analysis. The family number, m , must now be allowed to be both positive and negative when used in complex notation: $m = -N/2, \dots, N/2-1$. Although the equations look different, the game is the same: analysis by projecting the waveform ($x[n]$) on a harmonic sinusoidal basis, only the basis is now represented using complex variables. It is common to use capital letters for frequency-domain variables as in Equations 3.18 and 3.19.

The inverse FT can be calculated as

$$x[n] = \frac{1}{N} \sum_{m=1}^N X[m]e^{(-j2\pi mn/N)} \quad (3.20)$$

Applying Euler’s identity (Equation 3.16) to the expression for $X[m]$ in Equation 3.20 gives

$$X[m] = \sum_{n=1}^N x[n]\cos[2\pi mn/N] + j \sum_{n=1}^N x[n]\sin[2\pi mn/N] \quad (3.21)$$

Transforming this equation into polar form using the rectangular-to-polar transformation described in Equations 3.11 and 3.12, it can be shown that $|X[m]|$ equals $\sqrt{a[m]^2 + b[m]^2}$ (the magnitude for the sinusoidal representation of the Fourier series), whereas the angle of $X[m]$ that is $\tan^{-1}(jb[m]/a[m])$ (the phase angle for this representation of the sinusoid).

As mentioned above, $X[m]$ must be allowed to have both positive and negative values for m for computational reasons. Negative values of m imply negative frequencies, but these are only

Biosignal and Medical Image Processing

a computational necessity and have no physical meaning. In some versions of the Fourier series equations shown above, Equation 3.19 is multiplied by T_s (the sampling time) whereas Equation 3.20 is divided by T_s so that the sampling interval is incorporated explicitly into the Fourier series coefficients. Other methods of scaling these equations can be found in the literature.

Originally, the FT (or Fourier series analysis) was implemented by direct application of the above equations, usually using the complex formulation. These days, the FT is implemented by a more computationally efficient algorithm, the FFT, which cuts the number of computations from N^2 to $2 \log N$, where N is the length of the digital data. For large N , this is a substantial improvement in processor time. The implementation of the basic FT in MATLAB is

```
X = fft(x, N)
```

where x is the input waveform and X is a complex vector providing the complex sinusoidal coefficients. The argument n is optional, but is often quite useful. It is used to modify the length of the signal to be analyzed: if $N < \text{length}(x)$, then the signal that is analyzed is truncated to the first N points; if $N > \text{length}(x)$, x is padded with trailing zeros to equal n . The length of the complex output vector, X , is the same length as the adjusted (padded or truncated) length of x ; however, only the first half of the points are valid. If the signal is real, the second half of the magnitude spectrum of X is just the mirror image of the first half, and the second half of the phase spectrum is the mirror image inverted. So, the second half of both the magnitude and phase spectrum contains redundant information; however, note that the number of nonredundant points in the spectrum is the same as in the original signal, since essentially, two vectors are generated, magnitude and phase, each with half the valid number of points as the original signal. This “conservation of samples” is essential, since the FT is *invertible* and we must be able to reconstruct the original waveform from its spectrum.

The `fft` routine implements Equation 3.19 above using a special high-speed algorithm. Calculation time is highly dependent on data length and is fastest if the data length is a power of two, or if the length has many prime factors. For example, on one machine, a 4096-point FFT takes 2.1 s, but requires 7 s if the sequence is 4095-points long and 58 s if the sequence is 4097 points. If at all possible, it is best to stick with data lengths that are powers of two.

The output of the `fft` routine is a complex vector in which the real part corresponds to $a[m]$ in Equation 3.3 and the imaginary part corresponds to $b[m]$ in Equation 3.4. The first entry $X[1]$ is always real and is the DC component; thus, the first frequency component is the complex number in $X[2]$. It is easy to convert this complex number, or the entire complex vector, into polar form (i.e., magnitude and phase). To get the magnitude, take the absolute value function, `abs`, of the complex output X :

```
Magnitude = abs(X)
```

This MATLAB function simply takes the square root of the sum of the real part of X squared and the imaginary part of X squared. As with many MATLAB routines, X can be either a scalar or a vector and either a scalar or a vector is produced as the output. The phase angle can be obtained by using the MATLAB `angle` function:

```
Phase = angle(X)
```

The `angle` function takes the arctangent of the imaginary part divided by the real part of X . The output is in radians and should be converted into deg if desired.

The inverse FT (Equation 3.20) is implemented in MATLAB using the routine `ifft`. The calling structure is

```
x = ifft(X, N); % Take the inverse Fourier transform
```

An example applying the MATLAB's `fft` to an array containing sinusoids and white noise is given next. This example uses a special routine, `sig_noise`, found in the accompanying material. The routine is used throughout this book and generates data consisting of sinusoids and noise useful in evaluating spectral analysis algorithms. The calling structure for `sig_noise` used in this example is

```
[x,t] = sig_noise([f], [SNR], N);
```

where f specifies the frequency of the sinusoid(s) in Hz, N is the number of points, and SNR specifies the desired noise in dB associated with the sinusoid(s). The routine assumes a sample frequency of 1 kHz. If f is a vector, multiple sinusoids are generated. If SNR is a vector, it specifies the SNR for each frequency in f ; if it is a scalar, then all sinusoids generated have that SNR. The output waveform is in vector x ; t is a time vector useful in plotting.

EXAMPLE 3.3

Plot the magnitude and phase spectrum of a waveform consisting of a single sine wave and white noise with an SNR of -7 dB. Make the waveform 1024-points long. Use MATLAB's `fft` routine and plot only the valid points.

Solution

Specify N (1024) and f_s (1000 Hz) and generate the noisy waveform using `sig_noise`. Generate a frequency vector for use in plotting. This vector should be the same length as the signal and ranges to f_s (recall that when $m = N, f = f_s$, from Equations 3.5 and 3.6). Take the FT using `fft` and calculate the magnitude and phase from the complex output. Unwrap the phase and convert it into deg. Plot only the valid points from 1 to $N/2$. Since the first element produced by the `fft` routine is the DC component, the frequency vector should range between 0 and $N-1$ if the DC term is included in the spectral plot.

```
% Example 3.3 Determine spectrum of a noisy waveform
%N=1024; % Data length
N2 = 511; % Valid spectral points
fs=1000; % Sample frequency (assumed by sig_noise)
[x,t] = sig_noise(250,-7,N); % Generate signal (250 Hz sine plus white noise)
X=fft(x); % Calculate FFT
X_mag=abs(X); % Compute magnitude spectrum
Phase=unwrap(angle(X)); % Phase spectrum unwrapped
Phase=Phase*360/(2*pi); % Convert phase to deg
f=(0:N-1)*fs/N; % Frequency vector
subplot(2,1,1);
plot(f(1:N2),X_mag(1:N2),'k'); % Plot magnitude spectrum
.....Labels and title.....
subplot(2,1,2);
plot(f(1:N2),Phase(1:N2),'k') % Plot phase spectrum
label('Phase (deg)', 'FontSize', 14);
.....Labels and title.....
```

Results

The resulting plots are shown in Figure 3.9. The presence of a 250-Hz sine wave is easy to identify even though it is "buried" in considerable noise.

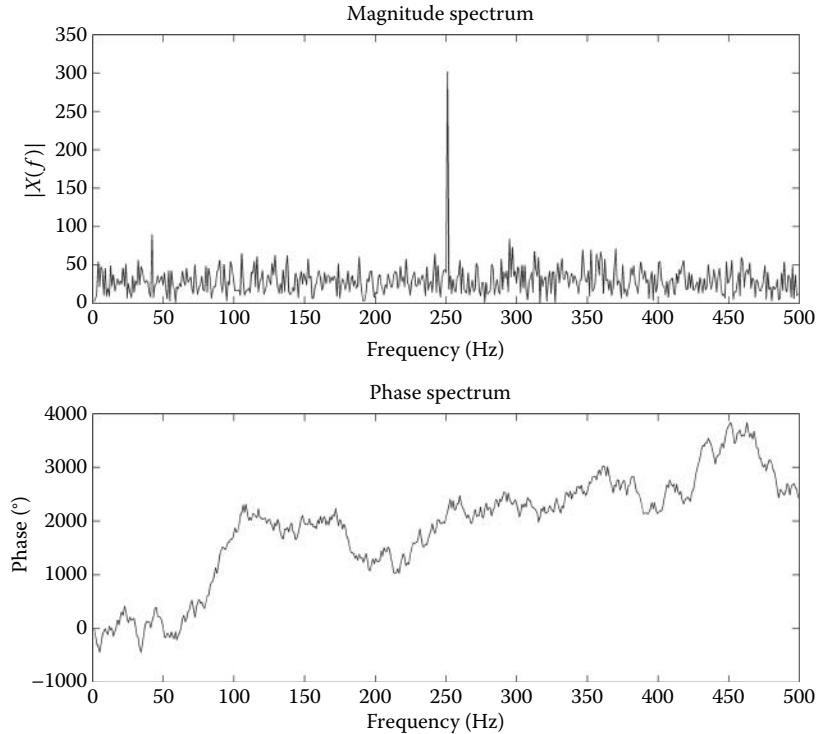


Figure 3.9 Magnitude and phase spectral plots produced by the program in Example 3.3. The peak at 250 Hz is apparent. The sampling frequency of these data is 1 kHz; hence, the spectrum is valid up to the Nyquist frequency, $f_s/2$ (500 Hz). Accordingly, only the first half of the vector produced by `fft` is plotted. (SNR = -7 dB, $N = 1024$.)

3.2.3 Data Length and Spectral Resolution

For the DFT, the length of the data, N , and the sample time, T_s (or f_s), completely determine the frequency range of the spectrum since $f_{\min} = f_1 = 1/T = 1/(NT_s)$ and $f_{\max} < f_s/2 = 1/2T_s$. So, in terms of T_s :

$$\frac{1}{NT_s} \leq f < \frac{1}{2T_s} \quad (3.22)$$

Or, using f_s :

$$\frac{f_s}{N_s} \leq f < \frac{f_s}{2} \quad (3.23)$$

These two variables, N and T_s , also determine the frequency resolution of the spectrum. Since the spectral frequencies are at $f = mf_1$ (Equation 3.5), the spectral frequencies are separated by f_1 that is equal to

$$f_{\text{Resolution}} = f_1 = \frac{1}{T_T} = \frac{1}{NT_s} = \frac{f_s}{N} \quad (3.24)$$

For a given sampling frequency, the larger the number of samples in the signal, N , the smaller the frequency increment between successive DFT data points: the more points sampled, the higher the spectral resolution.

Once the data have been acquired, it would seem that the number of points representing the data, N , is fixed, but there is a trick that can be used to increase the data length post hoc. We can increase N simply by tacking on constant values, usually zeros, that is, zero padding. This may sound like cheating, but it is justified by the underlying assumption that the signal is actually unknown outside the data segment on the computer. Zero padding gives the appearance of a spectrum with higher resolution since it decreases the distance between frequency points and the more closely spaced frequency points show more of the spectrum's details. In fact, extending the period with zeros does *not* increase the information in the signal and the resolution of the signal is really not any better. What it does is to provide an interpolation between the points in the unpadded signal: it fills in the gaps of the original spectrum using an estimation process. Overstating the value of zero padding is a common mistake of practicing engineers. Zero padding does not increase the resolution of the spectrum, only the apparent resolution. However, the interpolated spectrum will certainly look better when plotted.

EXAMPLE 3.4

Generate a 0.5-s symmetrical triangle wave assuming a sample frequency of 100 Hz so that $N = 50$ points. Calculate and plot the magnitude spectrum. Zero pad the signal so that the period is extended to 2 and 8 s and calculate and plot the new magnitude spectra. Limit the spectral plot to a range of 0–10 Hz.

Solution

First, generate the symmetrical triangle waveform. Since $f_s = 100$ Hz, the signal should be padded by 150 and 750 additional samples to extend the 0.5-s signal to 2.0 and 8.0 s. Calculate the appropriate time and frequency vectors based on the padded length. Calculate the spectrum using `fft` and take the absolute value of the complex output to get the magnitude spectra. Use a loop to calculate and plot the spectra of the three signals. As in Example 3.3, the frequency vector should range from 0 to $N-1$ since the DC term is included in the spectral plots.

```
% Example 3.4 Calculate the magnitude spectrum of a simple waveform with
% and without zero padding.
%
fs = 100; % Sample frequencies
N1 = [0 150 750]; % Padding added to signal
x = [(0:25) (24:-1:0)]; % Generate basic test signal
for k=1:3
    x1 = [x zeros(1,N1(k))]; % Zero pad signal
    N = length(x1); % Data length
    t = (1:N)/fs; % Time vector for plotting
    f = (0:N-1)*fs/N; % Frequency vector for plotting
    subplot(3,2,k*2-1);
    plot(t,x1,"k"); % Plot test signal
    .....Label and title.....
    xlim([0 t(end)]);
    subplot(3,2,k*2);
    X1=abs(fft(x1)); % Calculate the magnitude spectrum
    plot(f, X1,".k"); % Plot magnitude spectrum
    axis([0 10 0 max(X1)*1.2]);
    .....Label and title.....
end
```

Results

The time and magnitude spectrum plots are shown in Figure 3.10. All the spectral plots have the same shape, but the points are more closely spaced with the zero-padded data. Simply adding

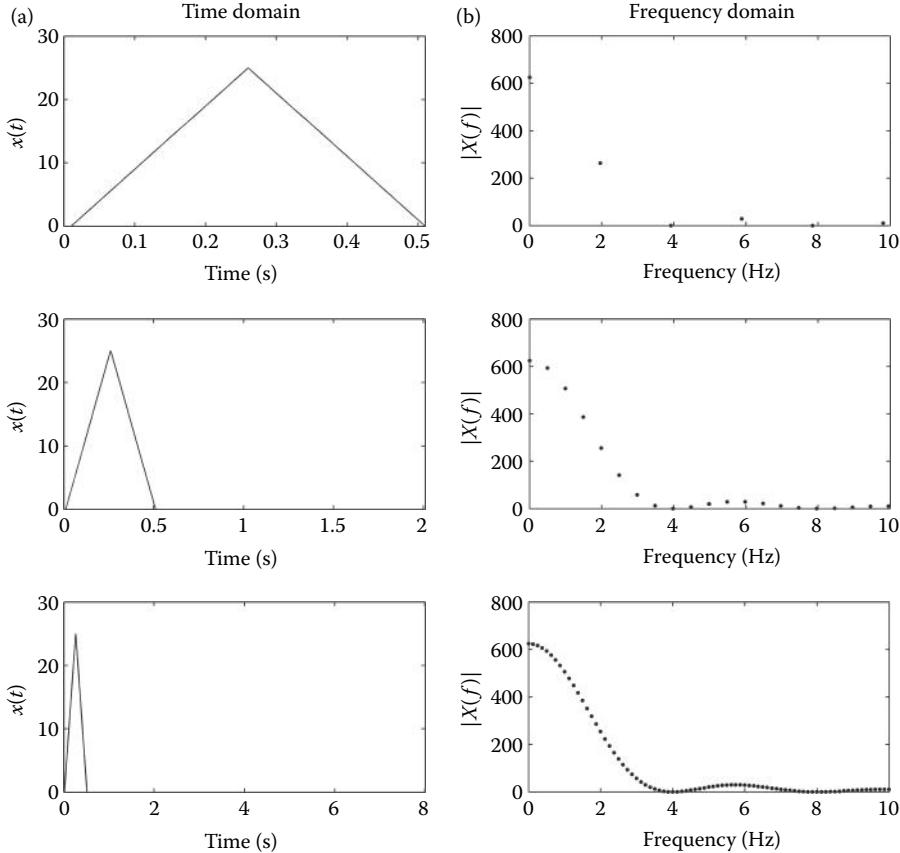


Figure 3.10 A triangular waveform padded with zeros to three different periods. Upper: $T_T = 0.5$ s (no padding). Middle: $T_T = 2.0$ s (padded with 150 zeros). Lower: $T_T = 8.0$ s (padded with 750 zeros).

zeros to the original signal produces a better-looking curve, which explains the popularity of zero padding even if no additional information is produced. In this example, the signal was padded before calling the `fft` routine since we wanted time plots of the padded signals. Otherwise, the second argument of the `fft` routine could have been used to pad the signal as described above. Also, the entire spectrum was plotted including the redundant points, but these points were eliminated by limiting the x -axis using `xlim`.

3.2.3.1 Aperiodic Functions

As mentioned above, the DFT makes the assumption that the signal is periodic with a period of $T_T = NT_s$, and the frequency spectrum produced is a set of individual numbers spaced f_s/N apart (Equation 3.24). To emphasize the individual nature of the spectral points, sometimes, the points are plotted as lines from the horizontal axis to the point value as shown in Figure 3.11a. This gives rise to the term *line spectra* for the spectra determined from periodic functions.

The frequency of the spectral point is $f = mf_1 = mf_s/N$ since $f_1 = (1/NT_s) = (f_s/N)$. As the period, NT_s , gets longer, N would increase and the distance between the spectral points, f_s/N , would get closer together. The longest period imaginable would be infinite, $T_T = NT_s \rightarrow \infty$, which would require an infinite number of points, $N \rightarrow \infty$. As the period becomes infinite, the spacing between points, $f_s/N \rightarrow 0$, and m/N goes to a continuous variable f . Equation 3.19 becomes, theoretically, a function of the continuous variable f :

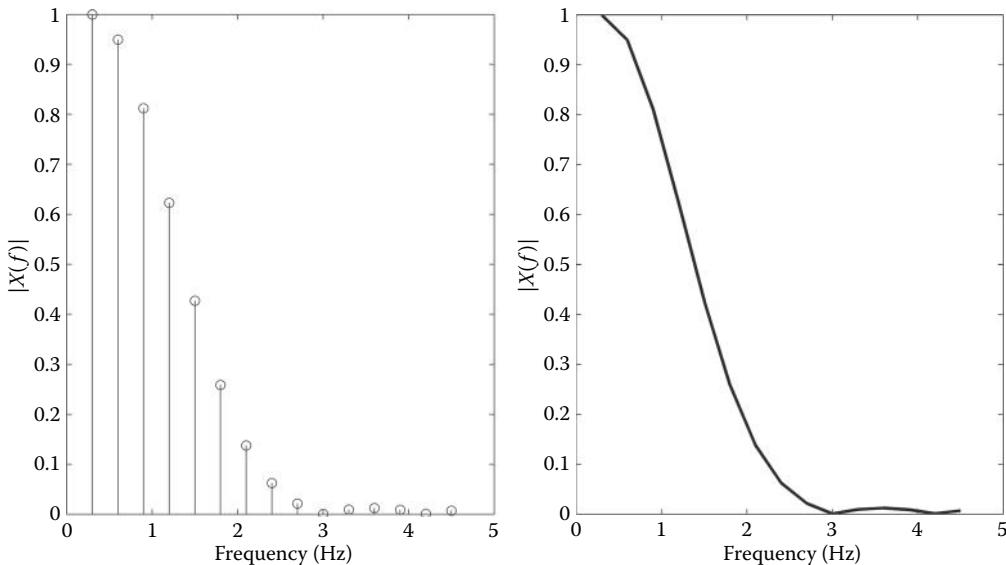


Figure 3.11 (a) The spectrum calculated by the discrete time Fourier series consists of individual points spaced f_s/N apart. (Recall that $f_1 = 1/(NT_s) = f_s/N$ and is equal to 0.3 Hz in this figure.) Sometimes, these individual points are plotted using vertical lines as shown, giving rise to the term *line spectra* for these plots. (b) If we assume that the original waveform is really aperiodic, then the distance between lines theoretically goes to 0 and the points can be connected.

$$X[f] = \sum_{n=1}^{\infty} x[n]e^{-j2\pi nf} \quad (3.25)$$

Such signals are termed *aperiodic*: they are nonzero only for a well-defined period of time and are zero everywhere else. For analog signals, it is possible to calculate the FT of an aperiodic signal. This is termed the *continuous time FT* (as opposed to the Fourier series for periodic signals). The discrete version of the FT for aperiodic signals is termed the *discrete time Fourier transform* (DTFT). This operation has only theoretical significance, since a discrete aperiodic signal requires an infinite number of points and cannot exist in a real computer. However, if we pretend that the signal was actually aperiodic, we can plot the spectrum not as a series of points, but as a smooth curve (Figure 3.11b). Although this is commonly done when plotting the spectra, you should be aware of the underlying truth: an implicit assumption is being made that the signal is aperiodic; the calculated spectrum is really a series of discrete points that have been joined together because of this assumption. Again, true aperiodic signals are not possible in a real computer and Equation 3.25 is only of theoretical interest. It “does not compute” on a digital computer.

In summary, there are four versions of “Fourier transform”: versions for periodic and aperiodic signals and for continuous and discrete signals. For each of these, there is a transform and an inverse transform. The equations for these four transforms are shown in Table 3.2 and the equations for the four inverse transforms are shown in Table 3.3. Note that in the rest of this book, only the DFT (also known as the discrete time Fourier series) and its inverse are used.

3.2.4 Window Functions: Data Truncation

The digitized versions of real biosignals are usually segments of a much longer, possibly infinite, time series. Examples are found in EEG and ECG analysis where the waveforms being analyzed continue over the lifetime of the subject. Obviously, only a portion of such waveforms can be

Table 3.2 Analysis Equations (Forward Transform)^a

<p>$X(t)$ periodic and continuous: Fourier series:</p> $X[m] = \frac{1}{T} \int_0^T x(t) e^{-j2\pi m f_t t} dt \quad m = 0, \pm 1, \pm 2, \pm 3, \dots$ $f_t = 1/T$	<p>$x[n]$ periodic and discrete: tDFT or discrete time Fourier series^{b,c}:</p> $X[m] = \sum_{n=1}^N X[n] e^{-j2\pi mn/N}$ $m = 0, \pm 1, \pm 2, \pm \dots \pm N/2$
<p>$x(t)$ aperiodic and continuous: FT or continuous time FT:</p> $X(f) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi ft} dt$	<p>$x[n]$ aperiodic and discrete: DTFT^d:</p> $X(f) = \sum_{n=-\infty}^{\infty} x[n] e^{-j2\pi nf}$

^a Often, ω is usually substituted for $2\pi f$ to make the equations shorter. In such cases, the Fourier series integration is carried out between 0 and 2π or from $-\pi$ to $+\pi$.

^b Alternatively, $2\pi m n T/N$ where T is the sampling interval is sometimes used instead of $2\pi m n/N$.

^c Sometimes, this equation is normalized by $1/N$ and then no normalization term is used in the inverse DFT.

^d The DTFT cannot be calculated on a computer since the summations are infinite. It is used in theoretical problems as an alternative to the DFT.

represented in the finite memory of the computer, and some attention must be paid to how the waveform is truncated. Often, a segment is simply cut out from the overall waveform, that is, a portion of the waveform is truncated and stored, without modification, in the computer. This is equivalent to the application of a rectangular *window* to the overall waveform and the analysis is restricted to the *windowed* portion of the waveform. The window function, $w(t)$, for a rectangular window is simply 1.0 over the length of the window and 0.0 elsewhere (Figure 3.12, middle trace). Note that a rectangular window will usually produce abrupt changes or discontinuities at the two endpoints (Figure 3.12, lower trace). Window shapes other than rectangles are possible simply by multiplying the waveform by the desired shape (sometimes, these shapes are referred to as “tapering” functions). (See Section 2.3.1.3 and Figure 2.11.)

Table 3.3 Synthesis Equations (Reverse Transform)

<p>Fourier Series:</p> $x(t) = \sum_{n=-\infty}^{\infty} X[m] e^{j2\pi m f_t t} \quad m = 0, \pm 1, \pm 2, \pm 3, \dots$ $m = 0, \pm 1, \pm 2, \pm \dots \pm N/2$	<p>DFT or discrete time Fourier series^{a,b}:</p> $x[n] = \frac{1}{N} \sum_{m=1}^N X[m] e^{j2\pi mn/N}$ $m = 0, \pm 1, \pm 2, \pm \dots \pm N/2$
<p>FT or continuous time FT:</p> $x(t) = \int_{-\infty}^{\infty} X(f) e^{j2\pi ft} df$	<p>DTFT^{c,d}:</p> $x[n] = \frac{1}{T} \int_0^T X(f) e^{j2\pi nf} df$

^a Alternatively, $2\pi m n T/N$ where T is the sampling interval is sometimes used instead of $2\pi m n/N$.

^b Sometimes, this equation is normalized by $1/N$ and then no normalization term is used in the inverse DFT.

^c The DTFT cannot be calculated on a computer since the summations are infinite. It is used in theoretical problems as an alternative to the DFT.

^d Requires an integral because the summations that produce $X(f)$ are infinite in number.

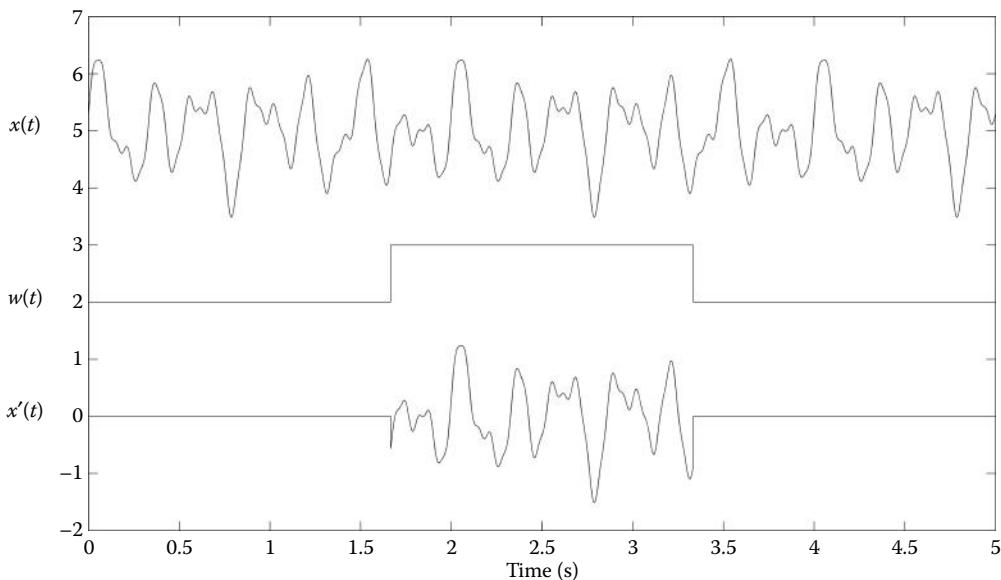


Figure 3.12 If a waveform is simply cut at the beginning and end of the portion stored in the computer, it is mathematically equivalent to multiplying the waveform, $x(t)$ by a rectangular window function, $w(t)$. This usually results in a discontinuity at the endpoints.

The act of windowing has an effect on a signal's spectrum, even if a rectangular window (i.e., simple truncation) is used. Mathematically, windowing is the multiplication of the signal with a window function. Multiplication in the time domain is equivalent to convolution in the frequency domain (and vice versa); so, the original spectrum is convolved with the spectrum of the window. Ideally, we would like this convolution to leave the signal spectrum unaltered. This will occur only if the spectrum of the window function is a unit impulse, that is, $W[f] = 1.0$ for $f = 0$ and 0.0 for all other frequencies. So, an idea of the artifact produced by a given window can be obtained from the FT of the window function itself. The deviations of the window's spectrum from a true impulse function show how it will modify the signal's spectrum.

The spectra of three popular windows are shown in Figure 3.13. The rectangular window spectrum (Figure 3.13a) is not exactly an impulse, although it does have a narrow peak at $f = 0$ Hz about 15 dB above the background peaks at $x(t)$. This peak around 0.0 Hz is called the *mainlobe*. The finite width of the mainlobe means that when it is convolved with the spectrum of the original signal, frequencies in the windowed spectrum will include an average of nearby frequencies in the original spectrum. Thus, peaks in a windowed spectrum will not be as precisely defined as when no window is used (i.e., the signal was of infinite length). Of more concern are the fairly large multiple peaks beginning at -15 dB and dropping to around -35 dB (Figure 3.13a). These are termed *sidelobes*, and they indicate the amount of distant frequencies that are merged into the windowed spectrum. They are of particular concern when a large spectral peak is near to a much smaller peak. Because of the merging produced by the side lobes, the smaller peak may be masked by the larger peak.

Other window shapes can be imposed on the data by multiplying the truncated waveform by the desired shape. There are many different window shapes and all have the effect of tapering the two ends of the data set toward zero. An example of a tapering window is the *Hamming window*, shown in Figure 3.14. This window is often used by MATLAB as a default window in routines that operate on short data sets.

Biosignal and Medical Image Processing

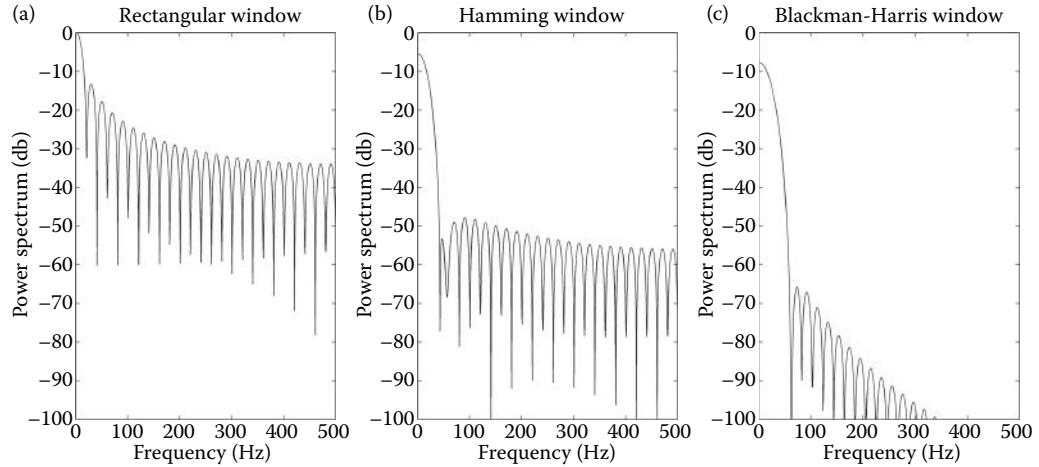


Figure 3.13 (a) The spectrum of a rectangular window. (b) The spectrum of a Hamming window. (c) The spectrum of the Blackman–Harris window.

The Hamming window has the shape of a raised half-sine wave (Figure 3.14) and when the truncated signal is multiplied by this function, the endpoint discontinuities are reduced. The equation for the Hamming function is

$$w[n] = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N-1}\right) \quad (3.26)$$

where $w[n]$ is the Hamming window function and where $-N/2 \leq n < N/2$ which makes the window length N , the length of the stored signal. As shown in Figure 3.13b, the spectrum of

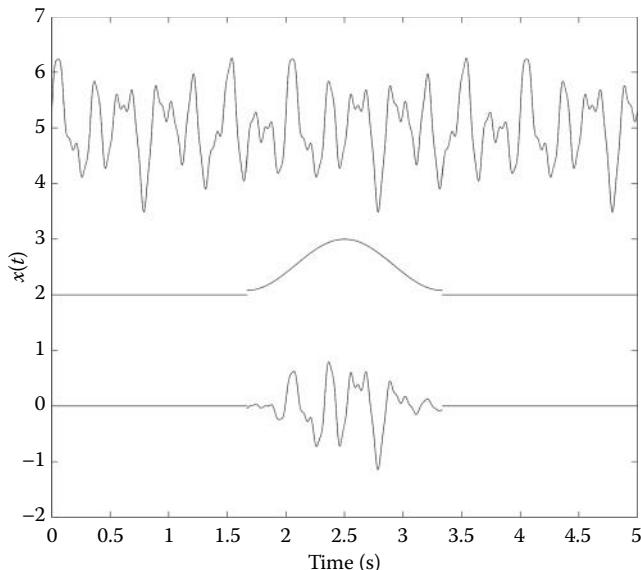


Figure 3.14 The Hamming window, shown as a time function, $w(t)$, is used to truncate a signal, $x(t)$. The Hamming window is defined in Equation 3.26.

the Hamming window has lower sidelobes than that of a rectangular window, which means the nearby peaks have less of an influence on a small peak. However, the mainlobe is wider than that of the rectangular window, leading to a decrease in spectral resolution. When the task is to precisely identify the frequency of a spectral peak, a rectangular window is better.

Many other windows exist and most are structured to reduce the sidelobes, but at the cost of a wider mainlobe. Figure 3.13c shows the spectrum of the Blackman–Harris window, one of the many windows offered in MATLAB’s Signal Processing Toolbox. The spectrum of this window has the lowest sidelobes (around -70 dB, Figure 3.13c), but it has a wider mainlobe than the other two windows. The Blackman–Harris window is based on multiple cosines:

$$w[n] = a_0 + a_1 \cos\left(\frac{2\pi}{N}n\right) + a_2 \cos\left(\frac{2\pi}{N}2n\right) + a_3 \cos\left(\frac{2\pi}{N}3n\right) \quad (3.27)$$

where $-N/2 \leq n < N/2$ to make the window length N . Typical coefficients are

$$a_0 = 0.35875, a_1 = 0.48829, a_2 = 0.14128, \text{ and } a_3 = 0.01168$$

The Blackman–Harris window suppresses the influence of nearby frequencies, but at an even greater reduction in spectral resolution. The fact that the Hamming window represents a good compromise in resolution and suppression may explain its adoption as the default window in many MATLAB routines.

Many different windows are easily implemented in MATLAB, especially with the Signal Processing Toolbox. Selecting the appropriate window, like so many other aspects of signal analysis, depends on what spectral features are of interest. If the task is to precisely define the frequency where a sharp spectral peak occurs, then a window with the narrowest mainlobe (the rectangular window) is preferred. If the task is to separate a strong and a weak sinusoidal signal at closely spaced frequencies, then a window with rapidly decaying sidelobes is preferred. Often, the most appropriate window is selected by trial and error.

MATLAB has a number of data windows available. These include the Hamming (Equation 3.26) and the Blackman–Harris (Equation 3.27) windows. The relevant MATLAB routines generate an N -point vector array containing the appropriate window shape. There are several ways of defining a window function in MATLAB along with a special tool, `wintool` for evaluating the various window functions. The most straightforward way to define a window is to call the MATLAB routine that has the same name as the window function.

```
w=window_name(N); % Generate vector w of length N
% containing the function.
```

where N is the number of points in the output vector and `window_name` is the name, or an abbreviation of the name, of the desired window. At this writing, 16 different windows are available in MATLAB’s Signal Processing Toolbox. Using `help window` will provide a list of window names. A few of the more popular windows are `bartlett`, `blackman-harris`, `gausswin`, `hamming`, `hanning`,^{*} and `kaiser`. A few of the routines have additional optional arguments. For example, `chebwin` (the Chebyshev window), which features a nondecaying, constant level of sidelobes, has a second argument to specify the sidelobe amplitude. Of course, the smaller this level is set, the wider the mainlobe becomes and the poorer the frequency resolution. The details for any given window can be found through

^{*} The `hanning` routine implements a Hann window that is very similar to the Hamming window: it is a pure half-cosine without the small offset in the Hamming window (see Figure 3.14, center trace). The similarity in the mathematical function and the name can be a source of confusion.

Biosignal and Medical Image Processing

the help command. In addition to the individual functions, all the window functions can be constructed with one call:

```
w = window(@name,N,opt); % Construct N-point window 'name.'
```

where name is the name of the specific window function (preceded by "@"), N is the number of points desired, and opt is the possible optional argument(s) required by some specific windows.

To apply a window, simply multiply, point by point, the digitized waveform by the window vector. Of course, the waveform and the window must be the same length. This operation is illustrated in the next example.

EXAMPLE 3.5

Generate a data set consisting of two sine waves closely spaced in frequency (235 and 250 Hz) with added white noise in a 128-point array sampled at 1 kHz. The SNR should be -3 dB. Apply a rectangular, Hamming, and Blackman-Harris window to the signal and take the FT. Plot the magnitude spectra (only the relevant points).

Solution

After defining f_s and N , generate the data using `sig_noise`. Also, generate a frequency vector for plotting the spectra. Take the magnitude of the FFT output for the signal with no window as the rectangular window. Then multiply the signal by a Hamming window generated by MATLAB's `hamming` routine. Note that the multiplication should be point by point using MATLAB's `.*` operator and the window function, and the signal may need to be transposed to have the same vector orientation. Repeat the analysis using the Blackman-Harris window.

```
% Example 3.5 Application of several window functions
%
fs = 1000; % Sampling freq assumed by sig_noise
N = 128;
x = sig_noise([235 250], -3, N); % Generate data
f = (0:N-1)*fs/N; % Frequency vector
%
X_mag = abs(fft(x)); % Mag. spect: rect. (no) window
subplot(3,1,1);
plot(f(1:N/2), X_mag(1:N/2)); % Plot magnitude
.....labels.....
%
x1 = x .*hamming(N)'; % Apply Hamming window (Eq. 3.26)
X_mag = abs(fft(x1)); % Mag. spect: Hamming window
subplot(3,1,2);
plot(f(1:N/2), X_mag(1:N/2), 'k'); % Plot magnitude
.....labels.....
%
x1 = x .*blackmanharris(N)'; % Apply Blackman-Harris (Eq. 3.27)
X_mag = abs(fft(x1)); % Mag. spect: Blackman-Harris window
subplot(3,1,3);
plot(f(1:N/2), X_mag(1:N/2), 'k'); % Plot magnitude
.....labels.....
```

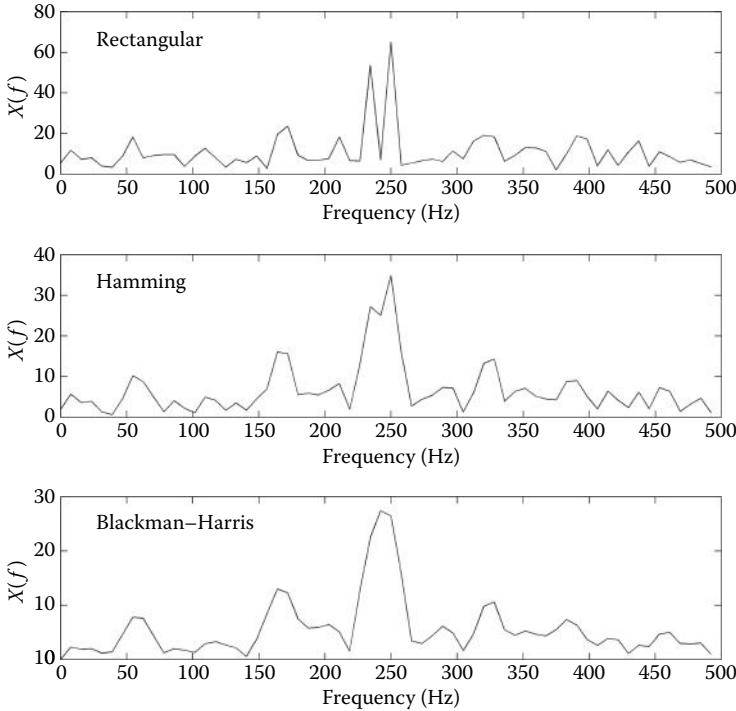


Figure 3.15 Three spectra computed from the same data but using different window functions. The data array is 128-points long and consists of two closely spaced sinusoids (235 and 250 Hz) in noise (SNR –3 dB). The influence of the three windows, with their increasing mainlobe widths, can be seen as a merging of nearby frequencies.

Results

Figure 3.15 shows three spectra obtained from Example 3.5. The differences in the three spectra are due to the windowing. The two peaks are clearly seen in the upper spectrum obtained using a rectangular function; however, the two peaks are barely discernible in the middle spectrum that used the Hamming window due to the broader mainlobe of this window that effectively averages adjacent frequencies. The effect of this frequency merging is even stronger when the Blackman–Harris window is used as only a single broad peak can be seen. The Blackman–Harris filter does provide a somewhat better estimate of the background white noise, as this background spectrum is smoother than in the other two spectra. The smoothing or averaging of adjacent frequencies by the Blackman–Harris window may seem like a disadvantage in this case, but with real data, where the spectrum may be highly variable, this averaging quality can be of benefit.

If the data set is fairly long (perhaps 256 points or more), the benefits of a nonrectangular window are slight. Figure 3.16 shows the spectra obtained with a rectangular and Hamming window to be nearly the same except for a scale difference produced by the Hamming window.

3.3 Power Spectrum

The power spectrum (PS) is commonly defined as the FT of the autocorrelation function. In continuous and discrete notations, the PS equation becomes

$$PS(f) = \frac{1}{T} \int_0^T r_{xx}(t) e^{-j2\pi mf_1 t} dt \quad m = 0, 1, 2, 3, \dots \quad (3.28)$$

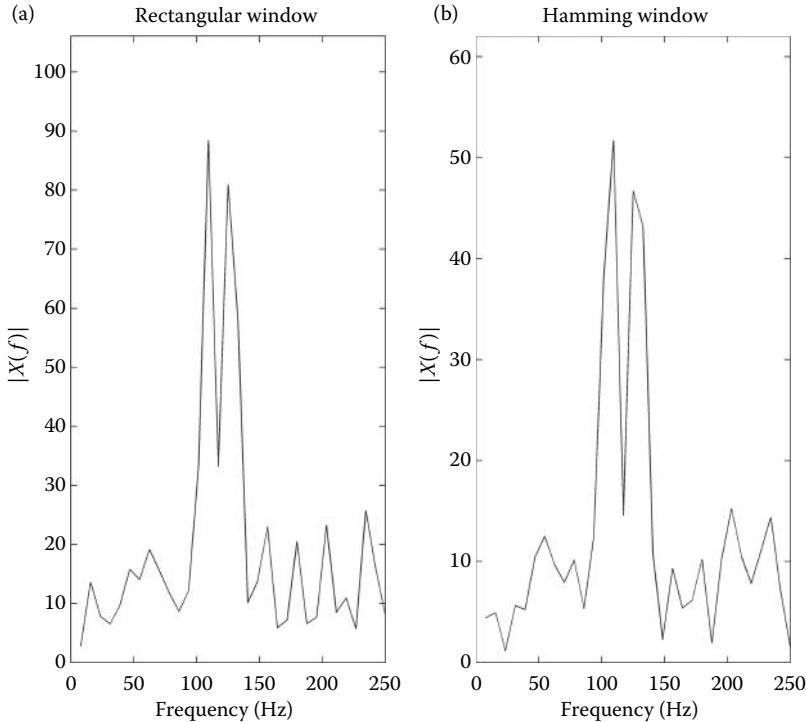


Figure 3.16 The spectra obtained from a signal containing two sinusoids and noise. The signal length is $N = 512$. The use of a nonrectangular window, in this case, a Hamming window, has little effect on the resulting spectrum. In a case where the signal consists of a fairly large number of points (>256), the rectangular window is preferred. Finally, the application of a nonrectangular window tapers the signal and reduces the energy in the signal. When this is of consequence, the scale of the magnitude spectrum can be restored by scaling the signal (or magnitude spectrum) by the ratio of the area under a rectangular window to the area under the applied window. An example of this is given in Problems 3.20 and 3.21, and in Example 3.7.

$$PS[m] = \sum_{n=1}^N r_{xx}[n] e^{-\frac{j2\pi nm}{N}} \quad m = 0, 1, 2, 3, \dots, N/2 \quad (3.29)$$

where $r_{xx}(t)$ and $r_{xx}[n]$ are autocorrelation functions as described in Chapter 2. Since the autocorrelation function has even symmetry, the sine terms of the Fourier series are all zero (see Table 3.1) and the two equations can be simplified to include only real cosine terms:

$$PS[m] = \sum_{n=0}^{N-1} r_{xx}[n] \cos\left(\frac{2\pi nm}{N}\right) \quad m = 0, 1, 2, 3, \dots, N/2 \quad (3.30)$$

$$PS(f) = \frac{1}{T} \int_0^T r_{xx}(t) \cos(2\pi mft) dt \quad m = 0, 1, 2, 3, \dots \quad (3.31)$$

Equations 3.30 and 3.31 are sometimes referred to as *cosine transforms*.

A more popular method for evaluating the PS is the *direct approach*. The direct approach is motivated by the fact that the energy contained in an analog signal, $x(t)$, is related to the magnitude of the signal squared integrated over time:

$$E = \int_{-\infty}^{\infty} |x(t)|^2 dt \quad (3.32)$$

By an extension of a theorem attributed to Parseval, it can be shown that

$$\int_{-\infty}^{\infty} |x(t)|^2 dt = \int_{-\infty}^{\infty} |X(f)|^2 df \quad (3.33)$$

Hence, $|X(f)|^2$ equals the energy density function over frequency, also referred to as the *energy spectral density*, *power spectral density* (PSD) or, simply and most commonly, the *power spectrum* (PS). In the direct approach, the PS is calculated as the magnitude squared of the Fourier transform (or the Fourier series) of the waveform of interest:

$$PS(f) = |X(f)|^2 \quad (3.34)$$

This direct approach of Equation 3.34 has displaced the cosine transform for determining the PS because of the efficiency of the FFT. (However, a variation of this approach is still used in some advanced signal-processing techniques involving time and frequency transformation.) Problem 3.18 compares the PS obtained using the direct approach of Equation 3.34 with the traditional cosine transform method represented by Equation 3.30 and, if done correctly, shows them to be identical.

Unlike the FT, the PS does not contain phase information; so, the PS is not an invertible transformation: it is not possible to reconstruct the signal from the PS. However, the PS has a wider range of applicability and can be defined for some signals that do not have a meaningful FT (such as those resulting from random processes). Since the PC does not contain phase information, it is applied in situations where phase is not considered useful or to data that contain a lot of noise, as phase information is easily corrupted by noise.

An example of the descriptive properties of the PS is given using the heart rate data shown in Figure 2.18. The heart rates show major differences in the mean and standard deviation of the rate between meditative and normal states. Applying the autocovariance to the meditative heart rate data (Example 2.11) indicates a possible repetitive structure for the variation in heart rate during meditation. The next example uses the PS to search for structure in the frequency characteristics of both normal and meditative heart rate data.

EXAMPLE 3.6

Determine and plot the power spectra of heart rate variability during both normal and meditative states.

Solution

The PS can be obtained through the Fourier transform using the direct method given in Equation 3.34. However, the heart rate data should first be converted into evenly sampled time data and this is a bit tricky. The data set obtained by downloading from the PhysioNet database provides the heart rate at unevenly spaced times, where the sample times are provided as a second vector. These interval data need to be rearranged into evenly spaced time positions. This process, known as *resampling*, is done through interpolation using MATLAB's `interp1` routine. This routine takes in the unevenly spaced $x-y$ pairs as two vectors along with a vector containing the desired evenly spaced x values. The routine then uses linear interpolation (other options are possible) to approximate the y values that match the evenly spaced x values. The details can be found in the MATLAB help file for `interp1`.

Biosignal and Medical Image Processing

In the program below, the uneven $x-y$ pairs for the normal conditions are in vectors t_pre and hr_pre , respectively, both in the MATLAB file Hr_pre . (For meditative conditions, the vectors are named t_med and hr_med and are in file Hr_med). The evenly spaced time vector is xi and the resampled heart rate values are in vector yi .

```
% Example 3.6
% Frequency analysis of heart rate data in the normal and meditative state
%
fs = 100; % Sample frequency (100 Hz)
ts = 1/fs; % Sample interval
load Hr_pre; % Load normal and meditative data
%
% Convert to evenly-spaced time data using interpolation; i.e., resampling
% First generate evenly space time vectors having one second
% intervals and extending over the time range of the data
%
xi = (ceil(t_pre(1)):ts:floor(t_pre(end))); % Evenly-spaced time vector
yi = interp1(t_pre,hr_pre,xi"); % Interpolate
yi = diff(yi); % Remove average
N2 = round(length(yi)/2);
f = (1:N2)*fs/N2; % Vector for plotting
%
% Now determine the Power spectrum
YI = abs((fft(yi)).^2); % Direct approach (Eq. 3.34)
subplot(1,2,1);
plot(f,YI(2:N2+1,"k"); % Plot spectrum, but not DC value
axis([0 .15 0 max(YI)*1.25]); % Limit frequency axis to 0.15 Hz
.....label and axis.....
%
% Repeat for meditative data
```

Analysis

To convert the heart rate data into a sequence of evenly spaced points in time, a time vector, xi , is first created that increases in increments of 0.01 s ($1/f_s = 1/100$) between the lowest and highest values of time (rounded appropriately) in the original data. A 100-Hz resampling frequency is used here because this is common in heart rate variability studies that use certain nonlinear methods, but in this example, a wide range of resampling frequencies gives the same result. Evenly spaced time data, yi , were generated using the MATLAB interpolation routine `interp1`. This example asked for the PS of heart rate *variability*, not heart rate per se; in other words, the change in beat-to-beat rate. To get this beat-to-beat change, we need to take the difference between sample values using MATLAB's `diff` operator before evaluating the PS.

After interpolation and removal of the mean heart rate, the PS is determined using `fft` and then taking the square of the magnitude component. The frequency plots (Figure 3.17) are limited to a maximum of 0.15 Hz since this is where most of the spectral energy is to be found. Note that the DC term is not plotted.

Results

The PS of normal heart rate variability is low and increases slightly with frequency (Figure 3.17a). The meditative spectrum (Figure 3.17b) shows a large peak at around 0.12 Hz, indicating that some resonant process is active at these frequencies, which correspond to a time frame of around 8 s. Speculation as to the mechanism behind this heart rate rhythm is left to the reader.

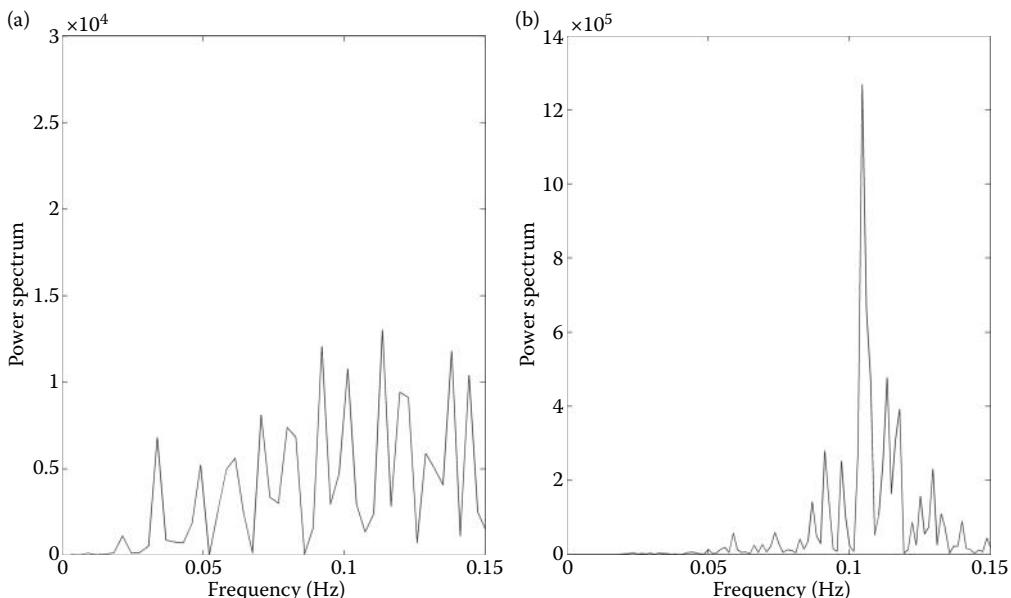


Figure 3.17 (a) The PS of heart rate variability under normal conditions. The power increases slightly with frequency. (b) The PS of heart rate variability during meditation. Strong peaks in power occur around 0.12 Hz, indicating that an oscillatory or resonant process is active around this frequency (possibly a feedback process with a time scale of 8.0 s). Note the much larger amplitude scale of this meditative spectrum.

3.4 Spectral Averaging: Welch's Method

While the PS is usually calculated using the entire waveform, it can also be applied to isolated segments of the data. The power spectra determined from each of these segments can then be averaged to produce a spectrum that represents the broadband or “global,” features of the spectrum better. This approach is popular when the available waveform is only a sample of a longer signal. In such situations, spectral analysis is necessarily an estimation process and averaging improves the statistical properties of the result. When the PS is based on a direct application of the FT followed by averaging, it is referred to as an *average periodogram*.

Averaging is usually achieved by dividing the waveform into a number of segments, possibly overlapping, and evaluating the PS on each of these segments (Figure 3.18). The final spectrum is constructed from the ensemble average of the power spectra obtained from each segment. The ensemble average is an averaged spectrum obtained by taking the average over all spectra at each frequency.* Ensemble averaging can be easily implemented in MATLAB by placing the spectra to be averaged in a matrix where each PS is a row of the matrix. The MATLAB averaging routine `mean` produces an average of each column; so, if the spectra are arranged as rows in the matrix, the routine will produce the ensemble average.

This averaging approach can only be applied to the magnitude spectrum or PS because it is insensitive to time translation. Applying this averaging technique to the standard FT would not make sense because the phase spectrum is sensitive to the segment position. Averaging phases obtained from different time positions would be meaningless.

One of the most popular procedures to evaluate the average periodogram is attributed to Welch and is a modification of the segmentation scheme originally developed by Bartlett. In

* Ensemble averaging is also used in the time domain to reduce noise. This application of averaging is described in Chapter 4.

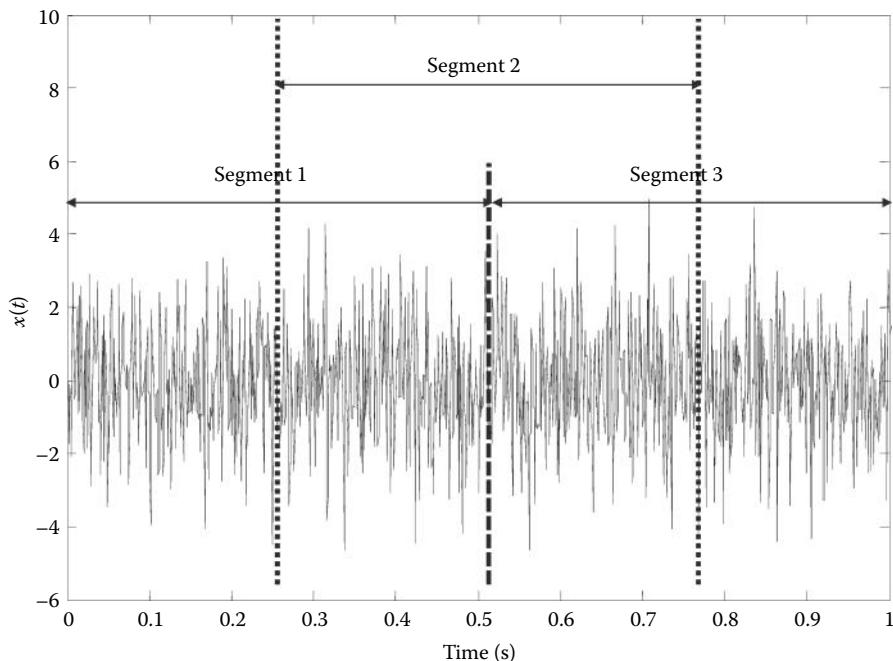


Figure 3.18 A waveform is divided into three segments with a 50% overlap. In the *Welch* method of spectral analysis, the PS of each segment is taken and an average of the three spectra is computed.

this approach, overlapping segments are used and a shaping window (i.e., a nonrectangular window) is sometimes applied to each segment. *Averaged periodograms* traditionally average spectra from half-overlapping segments, that is, segments that overlap by 50% as in Figure 3.18. Higher amounts of overlap have been recommended in applications when computing time is not a factor. Maximum overlap occurs when each segment is shifted by only one sample.

Segmenting the data reduces the number of data samples analyzed to the number in each segment. Equation 3.6 states that frequency resolution is proportional to f_s/N where N is now the number of samples in a segment. So, averaging produces a trade-off between spectral resolution, which is reduced by averaging and statistical reliability. Choosing a short segment length (a small N) will provide more segments for averaging and improve the reliability of the spectral estimate, but will also decrease frequency resolution. This trade-off is explored in the next example.

While it is not difficult to write a program to segment the waveform and average the individual power spectra, it is unnecessary as MATLAB has a routine that does this. The Signal Processing Toolbox includes the function `pwelch*` that performs these operations:

```
[PS, f] = pwelch(x, window, noverlap, nfft, fs); % Apply the Welch method
```

Only the first input argument, x , the data vector is required as the other arguments have default values. By default, x is divided into eight sections with 50% overlap, each section is

* The calling structure for this function is different in MATLAB versions <6.1. Use the help file to determine the calling structure if you are using an older version of MATLAB. Another version, `welch`, is included in the routines associated with this chapter and can be used if the Signal Processing Toolbox is not available. This version always defaults to the Hamming window but otherwise offers the same options as `pwelch`. See the associated help file.

windowed with the default Hamming window, and eight periodograms are computed and averaged. If `window` is an integer, it specifies the segment length and a Hamming window of that length is applied to each segment. If `window` is a vector, then it is assumed to contain the window function itself. Many window functions are easily implemented using the window routines described below. The segment length analyzed can be shortened by making `nfft` less than the window length (`nfft` must be less than, or equal to window length). The argument `noverlap` specifies the overlap in samples. The sampling frequency is specified by the optional argument `fs` and is used to fill the frequency vector, `f`, in the output with appropriate values. As is always the case in MATLAB, any variable can be omitted and the default can be selected by entering an empty vector, []. The output `PS` is in vector `PS` and is only half the length of the data vector, `x`, as the redundant points have been removed. Check the help file for other options. The spectral modification produced by the Welch method is explored in the following example.

EXAMPLE 3.7

Evaluate the influence of averaging power spectra on a signal made up of a combination of broadband and narrowband signals along with added noise. The data can be found in file `broadband1.mat`, which contains a white-noise signal filtered to be between 0 and 300 Hz and two closely spaced sinusoids at 390 and 410 Hz.

Solution

Load the test data file `broadband1` containing the narrowband and broadband processes. First, calculate and display the unaveraged PS using Equation 3.34. Then apply PS averaging using an averaging routine. Use a segment length of 128 points with a maximum overlap of 127 points. To implement averaging, use `pwelch` with the appropriate calling parameters.

```
% Example 3.7 Investigation of the use of averaging to improve
% broadband spectral characteristics in the power spectrum.
%
load broadband1; % Load data (variable x)
fs = 1000; % Sampling frequency
nfft = 128; % Segment length
%
% Un-averaged spectrum using the direct method of Eq. 3.34
%
PS = abs((fft(x)).^2)/length(x); % Calculate un-averaged PS
half_length = fix(length(PS)/2); % Valid points
f = (0:half_length-1)*fs/(2*half_length); % Frequency vector for plotting
subplot(1,2,1)
plot(f,PS(1:half_length),'k'); % Plot un-averaged Power Spectrum
.....labels and title.....
%
[PS_avg,f] = pwelch(x,nfft, nfft-1, [], fs); % Periodogram, max. overlap
%
subplot(1,2,2)
plot(f,PS_avg,'k'); % Plot periodogram
.....labels and title.....
```

Analysis

This example uses `pwelch` to determine the averaged PS and also calculates the unaveraged spectrum using `fft`. Note that in the latter case, the frequency vector includes the DC term. For

Biosignal and Medical Image Processing

the averaged spectrum or the periodogram, a segment length of 128 (a power of 2) was chosen along with maximal overlap. In practice, the selection of segment length and the averaging strategy is usually based on experimentation with the data.

Results

In the unaveraged PS (Figure 3.19a), the two sinusoids at 390 and 410 Hz are clearly seen; however, the broadband signal is noisy and poorly defined. The periodogram produced from the segmented and averaged data in Figure 3.19b is much smoother, reflecting the constant energy in white noise better, but the loss in frequency resolution is apparent as the two high-frequency sinusoids are hardly visible. This demonstrates one of those all-so-common engineering compromises. Spectral techniques that produce a good representation of “global” features such as broadband features are not good at resolving narrowband or “local” features such as sinusoids and vice versa.

EXAMPLE 3.8

Find the approximate bandwidth of the broadband signal, x , in file Ex3_8_data.mat ($f_s = 500$ Hz). The unaveraged PS of this signal is shown in Figure 3.20.

Solution

The example presents a number of challenges similar to those found in real-world problems. First, the PS obtained using the direct method is very noisy, as is often the case for broadband signals (Figure 3.20). To smooth this spectrum, we can use spectral averaging; this produces

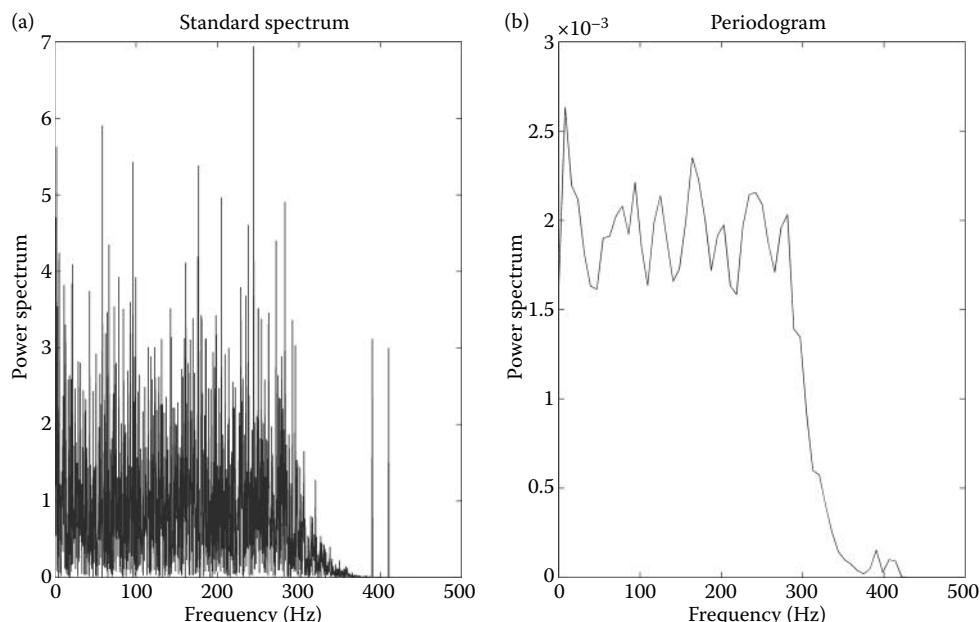


Figure 3.19 A waveform consisting of a broadband signal and two high-frequency sinusoids with noise. (a) The unaveraged PS clearly shows the high-frequency peaks, but the broadband characteristic is unclear. (b) The Welch averaging technique describes the spectrally flat broadband signal better, but the two sinusoids around 400 Hz are barely visible and would not be noticed unless you already knew they were there.

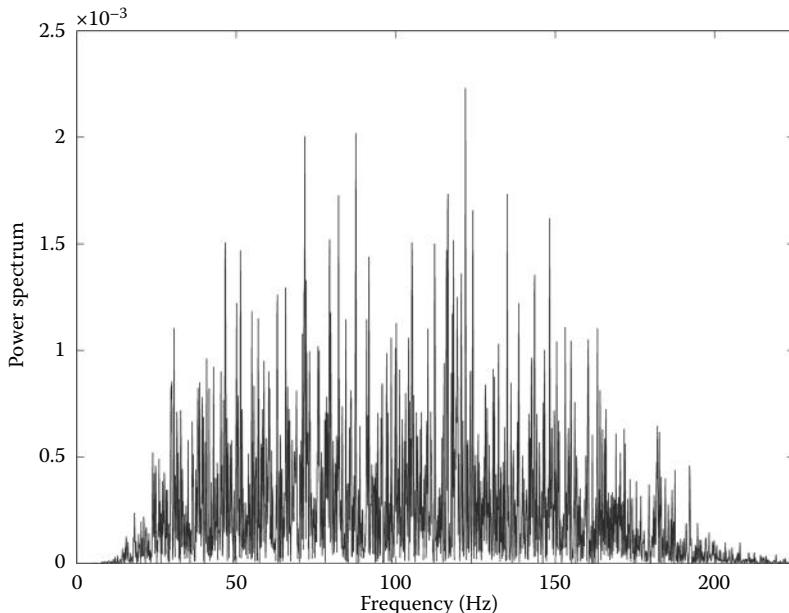


Figure 3.20 The PS of the broadband signal used in Example 3.8. Spectral averaging is used to produce a smoother spectrum (see Figure 3.21), making it easier to identify the cutoff frequencies. Since this is a PS, the cutoff frequencies will be taken at 0.5 (0.707^2), the bandpass value. This is the same as the –3-dB point in the magnitude spectrum when plotted in dB.

the cleaner spectrum shown in Figure 3.21. However, to find the bandwidth requires an estimate of spectral values in the bandpass region and there is considerable variation in this region.

To estimate the bandpass values, we take the average of all spectral values within 50% of the maximum value. This threshold is somewhat arbitrary, but seems to work well. The bandpass value obtained using this strategy is plotted as a dashed horizontal line in Figure 3.21. To find the high and low cutoff frequencies, we can use MATLAB's `find` routine to search for the first and last points >0.5 of the bandpass value. If this was the magnitude spectrum, we would take 0.707 of the bandpass value as described in Chapter 1, but since the PS is the square of the magnitude spectrum (Equation 3.34), we use 0.5 (i.e., 0.707^2) as the cutoff frequency value.

```
% Example 3.8 Find bandwidth of the signal x in file Ex3_8_data.mat
%
load Ex3_8_data.mat; % Load the data file. Data in x.
fs = 500; % Sampling frequency (given).
nfft = 64; % Power spectrum window size
[PS,f] = pwelch(x,[],nfft-1,nfft,fs); % PS using max. overlap
plot(f,PS,'r'); hold on; % Plot spectrum
bandpass_thresh = 0.50 *max(PS); % Threshold for bandpass values
bandpass_value = mean(PS(PS > bandpass_thresh)); % Use logical indexing
plot(f,PS,'k'); hold on; % Plot spectrum
plot([f(1) f(end)], [bandpass_value bandpass_value],':k'); % Bandpass est.
.....labels.....
%
cutoff_thresh = 0.5 *bandpass_value; % Set cutoff threshold
in_f1 = find(PS >cutoff_thresh, 1, 'first'); % Find index of low freq. cutoff
```

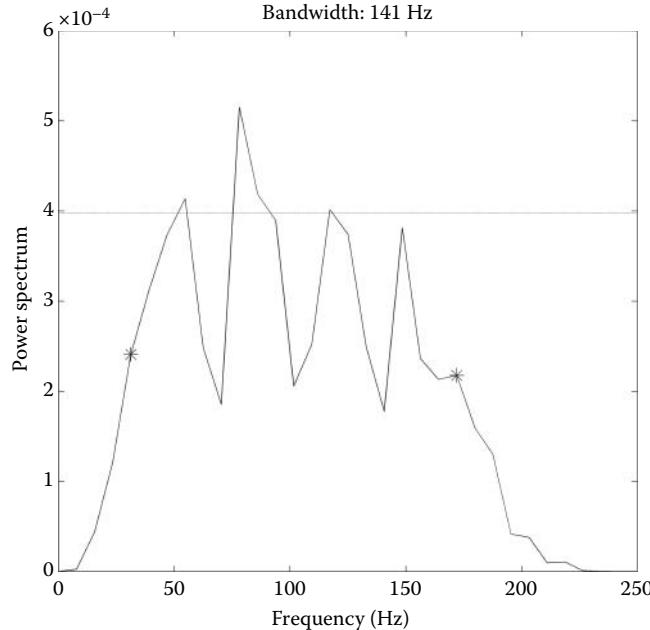


Figure 3.21 The spectrum of the signal used in Example 3.8 obtained using spectral averaging. The window size was selected empirically to be 64 samples. This size appears to give a smooth spectrum while maintaining good resolution. The bandpass value is determined by taking the average of spectral values greater than half the maximum value. The cutoff frequency points (these are the same as the -3-dB point in the magnitude spectrum) are determined by searching for the first and last points in the spectrum that are >0.5 of the average bandpass value. MATLAB's `find` routine was used to locate these points that are identified as large dots on the spectral curve. The frequency vector is used to convert the index of these points into equivalent frequencies and the calculated bandwidth is shown in the figure title.

```

in_fh = find(PS >cutoff_thresh, 1, 'last'); % Find index of high freq. cutoff
f_low = f(in_fl); % Find low cutoff freq.
f_high = f(in_fh); % Find high cutoff freq.
plot(f_low,PS(in_fl),'k*', 'MarkerSize',10); % Put marker at cutoff freqs.
plot(f_high,PS(in_fh),'k*', 'MarkerSize',10); % Put marker at cutoff freqs.
BW = f_high - f_low; % Calculate bandwidth
title(['Bandwidth: ',num2str(BW,3), 'Hz'], 'FontSize',14);

```

Results

Using the `pwelch` routine with a window size of 64 samples produces a spectrum that is fairly smooth, but still has a reasonable spectral resolution (Figure 3.21). As is so often necessary in signal processing, this window size was found by trial and error. Using *logical indexing* (`PS(PS > bandpass_thresh)`) allows us to get the mean of all spectral values $>50\%$ of the maximum power spectral value. This value is plotted as a horizontal line on the spectral plot.

Using MATLAB's `find` routine with the proper options gives us the indices of the first and last points above 0.5 of the bandpass value ($f_{low} = 31$ Hz and $f_{high} = 172$ Hz). These are plotted and superimposed on the spectral plot (large dots in Figure 3.21). The high and low sample points can be converted into frequencies using the frequency vector produced by `pwelch`. The difference between the two cutoff frequencies is the bandwidth and is displayed

in the title of the spectral plot (Figure 3.21). The estimation of the high and low cutoff frequencies might be improved by interpolating between the spectral frequencies on either side of the 0.5 values. This is done in Problem 3.36.

3.5 Summary

The sinusoid (i.e., $A \cos(\omega t + \theta)$) is a pure signal in that it has energy at only one frequency, the only waveform to have this property. This means that sinusoids can serve as intermediaries between the time-domain representation of a signal and its frequency-domain representation. The technique for determining the sinusoidal series representation of a periodic signal is known as Fourier series analysis. To determine the Fourier series, the signal of interest is correlated with sinusoids at harmonically related frequencies. This correlation provides either the amplitude of a cosine and sine series or the amplitude and phase of a sinusoidal series. The latter can be plotted against frequency to describe the frequency-domain composition of the signal. Fourier series analysis is often described and implemented using the complex representation of a sinusoid. A high-speed algorithm exists for calculating the discrete time Fourier series, also known as the DFT. The FT is invertable and the inverse FT can be used to construct a time-domain signal from its frequency representation.

Signals are usually truncated or shortened to fit within the computer memory. If the signal is simply cut off at the beginning and end, it is the same as multiplying the original signal by a rectangular window of amplitude 1.0 having the length of the stored version. Alternatively, the digital version of the signal can be multiplied by a shaping window that tapers the signal ends, providing less-abrupt endpoints. The window shape will have an influence on the resulting signal spectrum, particularly if the signal consists of a small number of data points. Window selection is a matter of trial and error, but the Hamming window is popular and is the default in some MATLAB routines that deal with short data segments. Window functions are used in the construction of some filters, as is described in the next chapter.

The DFT can be used to construct the PS of a signal. The power spectral curve describes how the signal's power varies with frequency. The PS is particularly useful for random data where phase characteristics have little meaning. By dividing the signal into a number of possibly overlapping segments and averaging the spectrum obtained from each segment, a smoothed PS can be obtained. The resulting frequency curve will emphasize the broadband or global characteristics of a signal's spectrum, but will lose the fine detail or local characteristics.

PROBLEMS

- 3.1 Extend Example 3.1 to include $m = N$ frequencies, where N is the number of points in the signal. Since $Mf_1 = f_s$, all the magnitude frequency components above $N/2$ should be reflections of frequency components below $N/2$. Note that the phase characteristics also reflect on either side of $f_s/2$ but are also inverted.
- 3.2 Modify Example 3.2 to decompose the waveform into 100 components and also the maximum number of valid components (i.e., $N/2$), and recompose it using both 100 and $N/2$ components. Plot only the reconstructed waveforms to best show the differences in reconstruction. Note the Gibbs oscillations with less than the maximum number of reconstructions.
- 3.3 Modify Example 3.2 to use the sawtooth waveform given by the equation:

$$x(t) = \begin{cases} t & 0 < t \leq 0.5 \\ 0.5 - t & 0.5 < t \leq 1.0 \end{cases}$$

Biosignal and Medical Image Processing

Also, reconstruct it using Equation 3.13 as in Example 3.2 with three and six components. Assume a sample interval of 250 Hz. Note how close the reconstruction is, even with only three components. Also note the absence of the Gibbs oscillations because there is no discontinuity in this waveform.

- 3.4 Modify Example 3.3 to plot all the points generated by the FFT algorithm. Use a signal that consists of two sinusoids at 100 and 200 Hz, both with an SNR of -10 dB. Use `sin_noise` to construct this signal. Observe that the magnitude plot is a mirror image about $f_s/2$ and the phase plot is the mirror image inverted.
- 3.5 Construct the sawtooth wave shown below and use the MATLAB `fft` routine to find the frequency spectrum of this waveform. Then reconstruct the square wave using the first 24 components using Equation 3.15. Do not plot the spectrum, but plot the original and reconstructed waveform superimposed. Make $f_s = 1024$ and $N = 1024$ to represent the 1-s periodic waveform. [Hint: The MATLAB `fft` routine does no scaling; so, you will have to scale the output to get the correct reconstructed waveform. Also, remember that the first element of the complex vector produced by the `fft` routine is the DC component (that will be 0.0 for this waveform) and the first complex sinusoidal coefficient is in the *second* element.]
- 3.6 File `sawtooth.mat` contains vector \mathbf{x} . This signal is the same waveform as in Problem 3.5 but with an added DC term. For this signal, $f_s = 1024$ and $N = 1024$. As in Problem 3.5, use the `fft` routine to find the frequency spectrum of this waveform. Then reconstruct the signal using the first 24 components and appropriately add the DC term. Do not plot the spectrum, but plot the original and reconstructed waveform superimposed. [Hint: After reconstructing the waveform, add the first `fft` term that contains the DC term, but remember to divide by 2 as indicated in Equation 3.15.]
- 3.7 Construct the waveform given by the equation below. Make $f_s = 1024$ and $N = 1024$.

$$x(t) = \begin{cases} t & 0 < t \leq 0.5 \\ -t - 0.5 & 0.5 < t \leq 1.0 \end{cases}$$

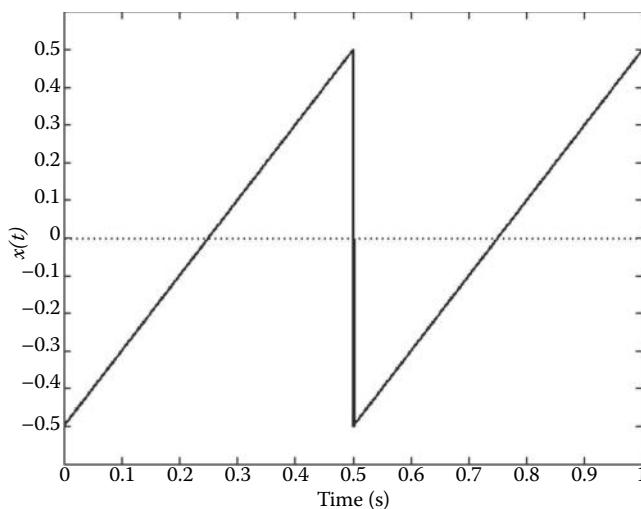


Figure P3.5 Waveform for Problem 3.5.

The second half of the waveform has the inverted version of the first; so, this waveform has half-wave symmetry. Plot the waveform to ensure that you have constructed it correctly. Use `fft` to find the frequency spectrum of this waveform and reconstruct the waveform using the first 24 components as in Problem 3.5 above. Then reconstruct the waveform using Equation 3.15 and only the first 12 odd-numbered harmonics. Remember that since the first element of the `fft` routine's output is the DC component so odd harmonics will be in elements 2, 4, 6, ... Note that the two reconstructions are the same since the even components in this half-wave symmetrical waveform are 0.0.

- 3.8 Use `randn` to construct two arrays of white noise: one 128 points in length and the other 1024 points in length. Take the FT of both and plot the magnitude of the nonredundant points. Does increasing the length improve the spectral estimate of white noise?
- 3.9 Use the routine `sig_noise` to generate a waveform containing 200- and 400-Hz sine waves with an SNR of -8 dB, that is, `x = sig_noise([200 400], -8, N)`. Make $N = 512$. Plot the nonredundant magnitude spectrum. Repeat for an SNR of -16 dB. Note that one or the other of the two sinusoids is hard to distinguish at the higher (-16 dB) noise level. Recall that with `sig_noise`, $f_s = 1 \text{ kHz}$. Rerun the program several times to get a feel for the variability in results due to noise fluctuations.
- 3.10 Use the routine `sig_noise` to generate a waveform containing 200- and 400-Hz sine waves as in Problem 3.9 with an SNR of -12 dB. Make $N = 1000$ samples ($f_s = 1 \text{ kHz}$). Again, plot only the nonredundant points in the magnitude spectrum. Repeat for the same SNR, but for a signal with only 200 points. Note that the two sinusoids are hard to distinguish with the smaller data sample. Taken together, Problems 3.9 and 3.10 indicate that both data length and noise level are important when detecting sinusoids (i.e., narrowband signals) in noise. Rerun the program several times to get a feel for the variability in results due to noise fluctuations.
- 3.11 The data file `pulses.mat` contains three signals: `x1`, `x2`, and `x3`. These signals are all 1.0 s in length and were sampled at 500 Hz. Plot the three signals and show that each signal contains a single 40-ms pulse, but at three different delays: 0, 100, and 200 ms. Calculate and plot the spectra for the three signals superimposed on a single magnitude and single phase plot. Plot only the first 20 points as discrete points, plus the DC term, using a different color for each signal's spectra. Apply the `unwrap` routine to the phase data and plot in deg. Note that the three magnitude plots are identical and while the phase plots are all straight lines, they have radically different slopes.
- 3.12 Load the file `chirp.mat` that contains a sinusoidal signal, `x`, which increases its frequency linearly over time. The sampling frequency of this signal is 5000 Hz. This type of signal is called a “chirp” signal because of the sound it makes when played through an audio system. If you have an audio system, you can listen to this signal after loading the file using the MATLAB command: `sound(x, 5000)`. Take the FT of this signal and plot magnitude and phase (no DC term). Note that the magnitude spectrum shows the range of frequencies that are present, but there is no information on the timing of these frequencies. Actually, information on signal timing is contained in the phase plot but, as you can see, this plot is not easy to interpret. Advanced signal-processing methods known as “time-frequency” methods (see Chapter 6) are necessary to recover the timing information.

Biosignal and Medical Image Processing

- 3.13 Load the file ECG_1 min.mat that contains 1 min of ECG data in variable `ecg`. Take the FT. Plot both the magnitude and phase (unwrapped) spectrum up to 20 Hz and do not include the DC term. The spectrum will have a number of peaks; however, the largest peak (that is also the lowest in frequency) will correspond to the cardiac cycle. Find the average heart rate in beats/min from this peak. The sample frequency is 250 Hz. [Hint: Get the index of that peak from the second output argument of MATLAB's `max` routine and the actual frequency from the frequency vector value at that index (i.e., $f_{\text{max}} = f(i_{\text{max}})$). The interbeat interval corresponding to that frequency in seconds. This is just the inverse of that frequency (i.e., $1/f_{\text{max}}$). The heart rate in beats/min is 60 s divided by the interbeat interval.]
- 3.14 This problem demonstrates aliasing. Generate a 512-point waveform consisting of two sinusoids at 200 and 400 Hz. Assume $f_s = 1 \text{ kHz}$. Generate another waveform containing frequencies at 200 and 900 Hz. Take the FT of both waveforms and plot the magnitude of the spectrum up to $f_s/2$. Plot the two spectra superimposed, but plot the second spectrum as dashed and in a different color to highlight the additional peak due to aliasing at 100 Hz. [Hint: To generate the sine waves, first construct a time vector, `t`, then generate the signal using $x = \sin(2\pi f_1 t) + \sin(2\pi f_2 t)$ where $f_1 = 200$ for both signals, whereas $f_2 = 400$ for one waveform and $f_2 = 900$ for the other.]
- 3.15 Another example of aliasing. Load the chirp signal, `x`, in file `chirp.mat` and plot the magnitude spectrum. Now, decrease the sampling frequency by a factor of 2, removing every other point, and recalculate and replot the magnitude spectrum. Note the distortion of the spectrum produced by aliasing. Do not forget to recalculate the new frequency vector based on the new data length and sampling frequency. (The new sampling frequency is effectively half that of the original signal.) Decreasing the sampling frequency is termed *downsampling* and can be easily done in MATLAB: `x1 = x(1:2:end)`. Note the disturbance in the downsampled signal at the higher frequencies.
- 3.16 Load the file `sample_rate.mat` that contains signals `x` and `y`. Are either of these signals oversampled (i.e., $f_s/2 \leq f_{\text{max}}$)? Alternatively, could the sampling rate of either signal be safely reduced? Justify your answer. Both signals are of the same length and $f_s = 500 \text{ Hz}$.
- 3.17 The file `short.mat` contains a very short signal of 32 samples. Plot the magnitude spectrum as discrete points obtained with and without zero padding. Zero pad out to a total of 256 points. Note the interpolation provided by zero padding.
- 3.18 Use `sig_noise` to generate a 256-point waveform consisting of a 300-Hz sine wave with an SNR of -12 dB (`x = sig_noise(300, -12, 256);`). Calculate and plot the PS using two different approaches. First, use the direct approach: take the FT and square the magnitude function. In the second approach, use the traditional method defined by Equation 3.29: take the FT of the autocorrelation function. Calculate the autocorrelation function using `xcorr`, then take the absolute value of the `fft` of the autocorrelation function. You should only use the second half of the autocorrelation function (those values corresponding to positive lags). Plot the PS derived from both techniques. The scales will be different because of different normalizations.
- 3.19 Generate a white-noise waveform using `randn`. Make $N = 1024$ and $f_s = 500 \text{ Hz}$. Construct the PS using the direct approach given by Equation 3.34. Apply the Welch methods to the waveform using a Hamming window with a window size (`nfft`) of 128 samples with no overlap. Now, change the overlap to 50% (an overlap

- of 64 samples) and note any changes in the spectrum. Submit the three frequency plots as subplots appropriately labeled.
- 3.20 This and the next problem demonstrate a reduction in energy produced by the use of tapering windows. Use `sig_noise` to generate a waveform containing two sinusoids, 200 and 300 Hz, with an SNR of -3 dB. Make the waveform fairly short: $N = 64$ ($f_s = 1$ kHz). Take the magnitude spectrum with a rectangular and a Blackman–Harris window and compare. Plot both magnitude spectra (as always, only valid points). Note the lower values for the windowed data. Rescale the time-domain signal to compensate for the reduction in energy caused by the application of the Blackman–Harris window. Replot both spectra and compare. This problem demonstrates that rescaling the time-domain signal when applying a tapering window is essential if magnitude spectra amplitude levels are to be preserved. Note that the spectrum of the rescaled windowed signal now has approximately the same amplitude as the unwindowed signal. Also note the smoothing done by the Blackman–Harris window. [Hint: Use either Equation 3.27 or MATLAB (`w = blackmanharris(N);`) to generate the window. Rescale the windowed time signal by multiplying by a scale factor. The scale factor can be determined by taking the area (i.e., sum) of a rectangular window of amplitude 1.0 divided by the area (sum) under the Blackman–Harris window. The rectangular window can be created using MATLAB's `ones` function.]
- 3.21 Repeat Problem 3.20 but compare the power spectra instead. Rescale the windowed time-domain signal before taking the windowed PS (again scale by the sum of a rectangular window divided by the sum of the Blackman–Harris window). Use the direct method to calculate both the power spectra (Equation 3.34). Again, compare the rectangular window PS with that obtained from the windowed, unscaled waveform and with the PS obtained from the windowed, scaled waveform.
- 3.22 Repeat Example 3.6 but determine the PS using the Welch method. Use the default number of segments (8) and the default overlap (50%). Plot the normal and meditative power spectra using the same scale for the vertical axis to compare the energy levels in the two signals better.
- 3.23 Generate the “filtered” waveform used in Problem 2.36. First, construct a 512-point Gaussian noise array, then filter it by averaging segments of three consecutive samples to produce a new signal. In other words, construct a new vector in which every point is the average of the preceding three points in the noise array: $y[n] = x[n]/3 + x[n-1]/3 + x[n-2]/3$. This is called a *moving average* (MA) filter. You could write the code to do this, but an easier way is to convolve the original data with a function consisting of three equal coefficients having a value of 1/3, in MATLAB: `h(n) = [1/3 1/3 1/3]`. Find the PS of this filtered noise data using the direct method (Equation 3.34). Assume $f_s = 200$ Hz for plotting. Note the lowpass characteristic of this averaging process, as some of the higher frequencies have been removed from the white noise.
- 3.24 Repeat Problem 3.23 above using the Welch averaging method with 256 and 32 segment lengths. Use the default window and maximum overlap. Use `subplot` to plot the two power spectra on the same page for comparison. The actual frequency characteristics of the MA filter begin to emerge when the short window is used.
- 3.25 This problem examines signal detection in noise using the Welch averaging method with different segment lengths. Load file `welch_ana.mat` that contains waveform `x` consisting of two sinusoids at 140 and 180 Hz with an SNR of -14 dB. $N = 256$ and $f_s = 1$ kHz. Construct the PS using window sizes of 256 (i.e., N), 128,

Biosignal and Medical Image Processing

64, and 32. Use the default window type and overlap. Note that at the smallest window, the two narrowband signals can no longer be distinguished from one another.

- 3.26 Construct the first-order process defined by the impulse response equation below.

$$h(t) = e^{(-t/\tau)}$$

To calculate the time vector, use $N = 1000$ and $f_s = 200$ Hz. Use convolution to input white noise to the process and use the Welch method to calculate the PS of the output (i.e., the result of convolution). Use a window size of 128 and the default overlap. Find and plot the PS for two time constants: $\tau = 0.2$ and 2.0 s. Limit the spectral plot to be between 0 and 10 Hz.

- 3.27 Repeat Problem 3.26, but use a second-order process given by the equation below:

$$h(t) = e^{(-2\pi t)} \sin(20\pi t)$$

Use the same variables as in Problem 3.26 to construct the equation: $N = 1000$ and $f_s = 200$ Hz. Use the Welch method to determine and plot the PS of the process to white-noise input. Use window sizes of 256 and 64. Limit the frequency range of the plot to be between 0 and 30 Hz.

- 3.28 Use `sig_noise` to generate a signal that contains a combination of 200- and 300-Hz sine waves with SNRs of -4 dB. Make the data segment fairly short, 64 samples, and recall $f_s = 1$ kHz. Plot the magnitude spectrum obtained with a rectangular, a Hamming, and a Blackman–Harris window. Generate 512-sample Hamming and Blackman–Harris windows using Equations 3.26 and 3.27. Multiply these windows point by point with the signal. Plot the magnitude spectra of the three signals. Note that the difference is slight, but could be significant in certain situations.
- 3.29 Use `sig_noise` to generate a signal that contains a combination of 280- and 300-Hz sine waves with SNRs of -10 dB. Make $N = 512$ samples and recall $f_s = 1$ kHz. Repeat Problem 3.19 and plot the magnitude spectrum obtained with a rectangular, a Hamming, and a Blackman–Harris window. Generate a 512-sample Hamming and Blackman–Harris windows using Equations 3.26 and 3.27. Multiply point by point with the signal. Plot the magnitude spectra of the three signals. Rerun the program several times and observe the variation in spectra that are produced due to variations in the random noise. Note how the Hamming window is a good compromise between the rectangular window (essentially no window) and the Blackman–Harris windows, again motivating its use as the default window in some MATLAB routines.
- 3.30 Use `sig_noise` to generate two arrays, one 128-points long and the other 512-points long. Include two closely spaced sinusoids having frequencies of 320 and 340 Hz with an SNR of -12 dB. Calculate and plot the (unaveraged) PS using Equation 3.34. Repeat the execution of the program several times and note the variability of the results, indicating that noise is noisy. Recall $f_s = 1$ kHz.
- 3.31 Use `sig_noise` to generate a 256-point array containing 320- and 340-Hz sinusoids as in Problem 3.21. Calculate and plot the unaveraged PS of this signal for an SNR of -10 , -14 , and -18 dB. How does the presence of noise affect the ability to detect and distinguish between the two sinusoids?
- 3.32 Load the file `broadband2` that contains variable `x`, a broadband signal with added noise. Assume a sample frequency of 1 kHz. Calculate the averaged PS using

- `pwelch`. Evaluate the influence of segment length using segment lengths of $N/4$ and $N/16$, where N is the length of the date of variable. Use the default overlap.
- 3.33 Load the file `eeg_data.mat` that contains an ECG signal in variable `eeg` ($f_s = 50$ Hz). Analyze these data using the unaveraged power spectral technique and an averaging technique using the `pwelch` routine. Find a segment length that gives a smooth background spectrum, but still retains any important spectral peaks. Use a 99% overlap.
 - 3.34 Load the file `broadband3.mat` that contains a broadband signal and a sinusoid at 400 Hz in variable `x` ($f_s = 1$ k Hz). Use the `pwelch` routine and apply a rectangular and a Blackman–Harris window to the PS. Analyze the spectrum of the combined broadband/narrowband signal with the two windows and with segment lengths of $N/4$ and $N/16$, where N is the length of the date of variable. Use the default (50%) overlap. Use `subplot` to group the four spectra for easy comparison. Note the trade-off between smoothing of broadband features and preservation of narrowband features. [Hint: Use `ones(1,N)` to generate a rectangular window of appropriate length.]
 - 3.35 Use the approach given in Example 3.8 to find the approximate bandwidth of the signal `x` in file `broadband2.mat`. ($f_s = 500$ Hz). Plot the unaveraged and smoothed spectra. Adjust the window to get the maximum size that still results in a reasonably smooth spectrum. As in Example 3.8, show the bandpass value and the cutoff frequency points.
 - 3.36 Load the file `broadband2.mat` and find the bandwidth of the signal in `x` using the methods of Example 3.8 combined with interpolation ($f_s = 1$ kHz). Determine the PS using `pwelch` with a window size of 64 and maximum overlap. This will give a highly smoothed estimate of the PS but a poor estimate of bandwidth due to the small number of points. Interpolation can be used to improve the bandwidth estimate from the smoothed PS. The easiest way to implement interpolation in this problem is to increase the number of points in the PS using MATLAB's `interp` routine. (E.g., `PS1 = interp(PS,5)` would increase the number of points in the PS by a factor of 5.) Estimate the bandwidth using the approach of Example 3.8 before and after expanding the PS by a factor of 5. Be sure to expand the frequency vector (`f` in Example 3.8) by the same amount. Repeat the steps to find the bandpass values and cutoff frequencies on the expanded spectrum, and plot the bandpass values and cutoff frequencies as in the example. Compare the bandwidth values obtained with and without interpolation.

4

Noise Reduction and Digital Filters

4.1 Noise Reduction

In the last chapter, specifically in Problem 3.23, we showed in a roundabout way how taking the average of three successive samples generates a new signal with reduced high-frequency components. The equation for this *moving average* method is

$$y[n] = x[n]/3 + x[n - 1]/3 + x[n - 2]/3 \quad (4.1)$$

where $x[n]$ is the original signal and $y[n]$ is the new signal.

Intuitively, we can see that this averaging process lessens the influence of outlying samples and smooths sharp transitions. Figure 4.1b shows the application of a three-sample moving average (Equation 4.1) to a signal in Figure 4.1a. The sharp transition is smoother and the amplitude of the outlier reduced. Of course, there is no reason to limit the average to three samples, and Figure 4.1c shows how this signal is changed by averaging over 10 samples. The transition is now very smooth and just a trace of the outlier remains. This shows that averaging can be a simple, yet powerful, tool to eliminate abrupt transitions from a signal. Such transitions are associated with high-frequency components, so this moving average process can be viewed as a lowpass filter. Later in this chapter, we show how to find the exact frequency characteristics of a moving average process.

Averaging is the fundamental concept behind all of the noise reduction techniques presented in this chapter. Unlike Equation 4.1, the weights* (the multiplying coefficients) need not all be the same. In addition, the averaging weights may be applied as a moving average (as in Equation 4.1), or as a recursive moving average so that samples already averaged are gone over again. Averaging can even be applied to whole signals, a technique known as *ensemble averaging*. This latter technique is quite powerful but has special requirements: it must be possible to obtain repeated observations of the same response.

Moving averaging is a process that is often described as a filter. Filters are used to improve the SNR of a signal and are often thought of in terms of reshaping a signal's spectrum to advantage. So, while the moving average filter is a time-domain process, its effects are usually conceptualized in the frequency domain. Most noise is broadband (the broadest-band noise being white noise with a flat spectrum), and most signals are narrowband. Accordingly, it should always be possible to improve the SNR by appropriately reshaping a waveform's spectrum using a filter.

* Filter weights are also called the *weighting function*, the *filter coefficients*, or just *coefficients* and these terms are used interchangeably to mean the weighting given to a sample in the moving average approach.

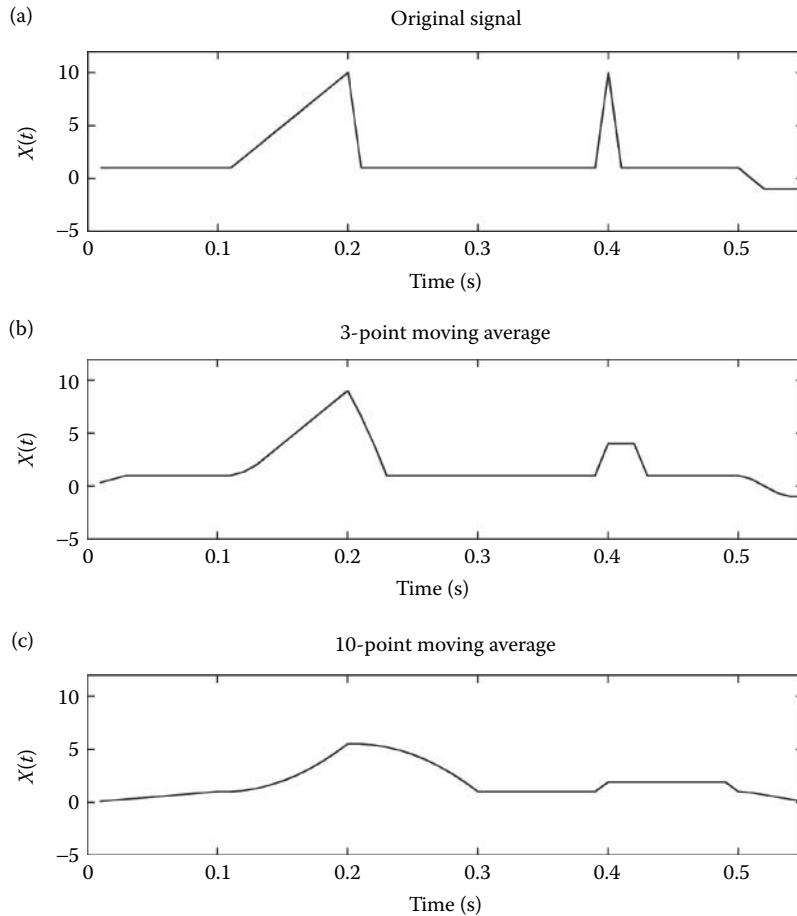


Figure 4.1 (a) An original signal segment that has a smooth ramp followed by a sharp transition at 0.12 s and an outlier at 0.4 s. (b) The signal produced by a 3-point moving average has a smoother transition and the height of the outlier reduced. (c) The signal produced by a 10-point moving average has a smooth transition and just a trace of the outlier.

A moving average can also be viewed as a linear process that acts on an input signal to produce an output signal with better (one hopes) SNR. If the filter is fixed, as are all the filters in this chapter, the filter is described as an LTI process (see Section 2.3.3). Moving average filters usually involve unequal weights (multiplying coefficients) and these weights act on the signal in the same manner as the impulse response of a linear process. Since the number of weights is finite, such moving average filters are termed *finite impulse response (FIR)* filters. Recursive filters apply weighted averaging to the input signal just as in FIR filters, but then they also implement a second set of weights to a delayed version of the “FIR-type” output to produce a new output. The impulse response of such recursive processes is effectively infinite, so they are termed *infinite impulse response (IIR)* filters. They can be particularly effective acting over only a few samples.

4.2 Noise Reduction through Ensemble Averaging

When multiple measurements are made, multiple values or signals are generated. If these measurements are combined or added together, the means add, so the combined value or signal has

4.2 Noise Reduction through Ensemble Averaging

a mean that is the average of the individual means. The same is true for the variance: the variances add, and the average variance of the combined measurement is the mean of the individual variances:

$$\bar{\sigma}^2 = \frac{1}{N} \sum_{n=1}^N \sigma_n^2 \quad (4.2)$$

When several signals are added together, it can be shown that the noise standard deviation is reduced by a factor equal to $1/\sqrt{N}$, where N is the number of measurements that are averaged.

$$\bar{\sigma}_{\text{AVG}} = \frac{\sigma}{\sqrt{N}} \quad (4.3)$$

In other words, averaging measurements from different sensors, or averaging multiple measurements from the same source, reduces the standard deviation of the measurement's variability by the square root of the number of averages. For this reason, it is common to make multiple measurements whenever possible and average the results.

In ensemble averaging, entire signals are averaged together. Ensemble averaging is a simple yet powerful signal-processing technique for reducing noise when multiple observations of the signal are possible. Such multiple observations could come from multiple sensors, but in many biomedical applications, the multiple observations come from repeated responses to the same stimulus. In ensemble averaging, several time responses are averaged over corresponding points in time. A new signal with a lower standard deviation is constructed by averaging over all signals in the ensemble. A classic biomedical engineering example of the application of ensemble averaging is the *visual-evoked response* (VER) in which a visual stimulus produces a small neural signal embedded in the EEG. Usually this signal cannot be detected in the EEG signal, but by averaging hundreds of observations of the EEG, time-locked to the visual stimulus, the visually evoked signal emerges.

There are two essential requirements for the application of ensemble averaging for noise reduction: the ability to obtain multiple observations, and a reference signal closely time-linked to the response. The reference signal shows how the multiple observations are to be aligned for averaging. Usually, a time signal linked to the stimulus is used.

An example of ensemble averaging is given in Example 4.1. In this example, a simulated VER data set is used. This data set not only contains the responses that would typically be recorded, but also the actual noise-free VER. Of course, it is impossible to get a noise-free version of the VER in the real world, but for our analysis here, the actual VER allows us to calculate the noise reduction produced by ensemble averaging. We can then compare this reduction with the theoretical prediction given by Equation 4.3.

EXAMPLE 4.1

This example evaluates the noise reduction provided by ensemble averaging. An ensemble of simulated VERs is found as matrix variable `ver` in MATLAB file `ver.mat`. The actual, noise-free VER signal is found as variable `actual_ver` in the same file. $T_s = 0.005$ s. Construct an ensemble average of 100 responses. Subtract out the noise-free VER from a selected individual VER and estimate the noise in each record. Then determine the noise in the ensemble average, and compare the reduction in standard deviation produced by averaging with that predicted theoretically.

Solution

Load the VER signals in file `ver.mat`. To average the signals in the matrix, we can use the MATLAB averaging routine `mean`. For a matrix, this routine produces an average of each

Biosignal and Medical Image Processing

column, so if the signals are arranged as rows in the matrix, the routine will produce the ensemble average. To determine the orientation of the data, we check the size of the data matrix. Normally, the number of signal samples will be greater than the number of signals. Since we want the signals to be in rows, if the number of rows is greater than the number of columns, we will transpose the data using the MATLAB transposition operator (i.e., the ' symbol).

```
% Example 4.1 Example of ensemble averaging
load ver; % Get visual evoked response data;
Ts = 0.005; % Sample interval = 5 msec
[nu,N] = size(ver); % Get data matrix size
if nu > N
    ver = ver';
    t = (1:nu)*Ts; % Generate time vector
else
    t = (1:N)*Ts; % Time vector if no transpose
end
%
subplot(2,1,1);
plot(t,ver(4,:)); % Plot individual record (Number 4)
..... Label axes .....
% Construct and plot the ensemble average
avg = mean(ver); % Take ensemble average
subplot(2,1,2);
plot(t,avg); % Plot ensemble average other data
..... Label axes .....
%
% Estimate the noise components
unaveraged_noise = ver(:, :) - actual_ver; % Noise in individual signal
averaged_noise = avg - actual_ver; % Noise in ensemble avg.
std_unaveraged = std(unaveraged_noise); % Std of noise in signal
std_averaged = std(averaged_noise); % Std of noise in ensemble avg.
theoretical = std_unaveraged/sqrt(100); % Theoretical noise reduction
disp(' Unaveraged Averaged Theroetical') % Output results
disp([std_unaveraged std_averaged theoretical])
```

Results

The VER is not really visible in a typical response (Figure 4.2a) but is clearly visible in the ensemble average (Figure 4.2b). The numerical output from this code is

Unaveraged Standard Deviation	Averaged Standard Deviation	Theoretical Averaged Standard Deviation
0.9796	0.0985	0.0979

Based on Equation 4.3, which states that the noise should be reduced by \sqrt{N} , where N is the number of responses in the average, we would expect the ensemble average to have $\sqrt{100} = 10$ times less noise than an individual record. As shown above, the standard deviation after averaging is just slightly more than the predicted value. It is not uncommon that signal-processing techniques do not reduce noise quite as much as predicted due to round-off and other errors associated with digital operations. In addition, noise is a random process, and the results of ensemble averaging are subject to some randomness. The noise reduction characteristics of ensemble averaging and its limitations are further explored in Problems 4.2 and 4.3.

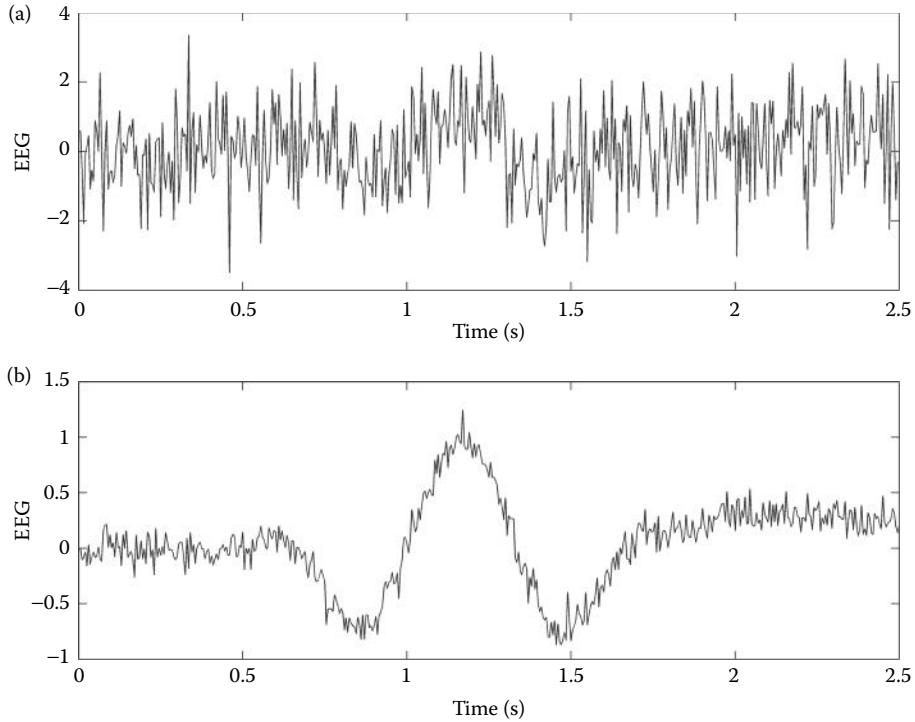


Figure 4.2 (a) The raw EEG signal showing a single response to the stimulus. (b) The ensemble average of 100 individual responses such as in graph (a) with the VER now clearly visible.

4.3 Z-Transform

The frequency-based analysis introduced in the last chapter is the most useful tool for analyzing systems or responses in which the waveforms are periodic or aperiodic, but cannot be applied to transient responses of infinite length, such as step functions, or systems with nonzero initial conditions. These shortcomings motivated the development of the *Laplace transform* in the analog domain. Laplace analysis uses the complex variable s , where $s = \sigma + j\omega$ as a representation of complex frequency in place of $j\omega$ in the Fourier transform.

$$X(\sigma, \omega) = \int_0^{\infty} x(t) e^{-\sigma t} e^{-j\omega t} dt = \int_0^{\infty} x(t) e^{-st} dt \quad (4.4)$$

The addition of the $e^{-\sigma t}$ term in the Laplace transform insures convergence for a wide range of input functions, $x(t)$, provided σ has the proper value. Like the Fourier transform, the Laplace transform is bilateral, but unlike the Fourier transform, it is not determined numerically: transformation tables and analytical calculations are used.

The *Z-transform* is a modification of the Laplace transform that is more suited to the digital domain; it is much easier to make the transformation between the discrete time domain and the Z-domain. Since we need a discrete equation, the first step in modifying the Laplace transform is to convert it to a discrete form by replacing the integral with a summation and substituting

Biosignal and Medical Image Processing

sample number n for the time variable t . (Actually, $t = nT_s$, but we assume that T_s is normalized to 1.0 for this development.) These modifications give

$$X(\sigma, \omega) = \sum_{n=-\infty}^{\infty} x[n]e^{-\sigma n}e^{-j\omega n} = \sum_{n=-\infty}^{\infty} x[n]r^n e^{-j\omega n} \quad (4.5)$$

where $r = e^\sigma$.

Equation 4.5 is a valid equation for the Z-transform, but in the usual format, the equation is simplified by defining another new, complex variable. This is essentially the idea used in the Laplace transform, where the complex variable s is introduced to represent $\sigma + j\omega$. The new variable, z , is defined as $z = re^{-j\omega} = |z| e^{-j\omega}$, and Equation 4.5 becomes

$$X(z) = \sum_{n=-\infty}^{\infty} x[n]z^{-n} = Z[x[n]] \quad (4.6)$$

This is the defining equation for the Z-transform, notated as $Z[x[n]]$. The Z-transform follows the format of the general transform equation involving projection on a basis. In this case, the basis is z^{-n} . In any real application, the limit of the summation in Equation 4.6 is finite, usually the length of $x[n]$.

As with the Laplace transform, the Z-transform is based around the complex variable, in this case, the arbitrary complex number z , which equals $|z| e^{j\omega}$. Like the analogous variable s in the Laplace transform, z is termed the *complex frequency*. As with the Laplace variable s , it is possible to substitute $e^{j\omega}$ for z to perform a strictly sinusoidal analysis.* This is a very useful property as it allows us to easily determine the frequency characteristic of a Z-transform function.

While the substitutions made to convert the Laplace transform to the Z-transform may seem arbitrary, they greatly simplify conversion between time and frequency in the digital domain. Unlike the Laplace transform, which requires a table and often considerable algebra, the conversion between discrete complex frequency representation and the discrete time function is easy as shown in the next section. Once we know the Z-transform of a filter, we can easily find the spectral characteristics of the filter: simply substitute $e^{j\omega}$ for z and apply standard techniques like the Fourier transform.

Equation 4.6 indicates that every data sample in the sequence $x[n]$ is associated with a unique power of z ; that is, a unique value of n . The value of n defines a sample's position in the sequence. If $x[n]$ is a time sequence, the higher the value of n in the Z-transform, the further along in the time sequence a given data sample is located. In other words, in the Z-transform, the value of n indicates a time shift for the associated input waveform, $x[n]$. This time-shifting property of z^{-n} can be formally stated as

$$Z[x(n - k)] = z^{-k}Z[x(n)] \quad (4.7)$$

This time-shifting property of the Z-transform makes it very easy to implement Z-transform conversion, either from a data sequence to the Z-transform representation or vice versa.

4.3.1 Digital Transfer Function

As in Laplace transform analysis, the most useful applications of the Z-transform lies in its ability to define the digital equivalent of a transfer function. By analogy to linear system analysis, the digital transfer function is defined as

$$H[z] = \frac{Y[z]}{x[z]} \quad (4.8)$$

* If $|z|$ is set to 1, then $z = e^{j\omega}$. This is called evaluating z on the unit circle. See Smith (1997) or Bruce (2001) for a clear and detailed discussion of the properties of z and the Z-transform.

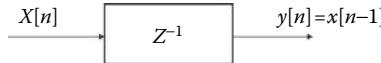


Figure 4.3 The Z-transform illustrated as a linear system. The system consists simply of a unit delay, which shifts the input by one data sample. Other powers of z can be used to provide larger shifts.

where $X[z]$ is the Z-transform of the input signal $x[n]$, and $Y[z]$ is the Z-transform of the system's output, $y[n]$. As an example, a simple linear system known as a *unit delay* is shown in Figure 4.3. In this system, the time-shifting characteristic of z is used to define the process where the output is the same as the input, but shifted (or delayed) by one data sample. The z-transfer function for this process is just $H[z] = z^{-1}$.

Most transfer functions will be more complicated than that of Figure 4.3. They can include polynomials of z in both the numerator and denominator, just as analog transfer functions contain polynomials of s :

$$H[z] = \frac{b[0] + b[1]z^{-1} + b[2]z^{-2} + \cdots + b[K]z^{-K}}{1 + a[1]z^{-1} + a[2]z^{-2} + \cdots + a[L]z^{-L}} \quad (4.9)$$

where the $b[k]$ s are constant coefficients of the numerator and the $a[\ell]$ s are coefficients of the denominator.* While $H[z]$ has a structure similar to the Laplace-domain transfer function $H[s]$, there is no simple relationship between them.[†] For example, unlike analog systems, the order of the numerator, K (Equation 4.9), need not be less than, or equal to, the order of the denominator, L , for stability. In fact, systems that have a denominator order of 1 (i.e., no zs in the denominator) are stabler than those having higher-order denominators.

Note that, just as in the Laplace transform, a linear system is completely defined by the a and b coefficients. Equation 4.9 can be more succinctly written as

$$H[z] = \frac{\sum_{k=0}^K b[k]z^{-k}}{\sum_{\ell=0}^L a[\ell]z^{-\ell}} \quad (4.10)$$

From the digital transfer function, $H[z]$, it is possible to determine the output given any input:

$$Y[z] = X[x]H[z] \quad y[n] = Y^{-1}[z] \quad (4.11)$$

where $Y^{-1}[z]$ is the inverse Z-transform.

While the input–output relationship can be determined from Equation 4.9 or 4.10, it is more common to use the difference equation. This is analogous to the time-domain equation, and can

* This equation can be found in a number of different formats. A few authors reverse the roles of the a and b coefficients, with as being the numerator coefficients and bs being the denominator coefficients. More commonly, the denominator coefficients are written with negative signs changing the nominal sign of the a coefficient values. Finally, it is common to start the coefficient series with $a[0]$ and $b[0]$ so that the coefficient index is the same as the power of z . This is the notation used here; it departs from that used by MATLAB where each coefficient in the series starts with 1 (i.e., $a[1]$ and $b[1]$). So $a[0]$ here equal to $a[1]$ in MATLAB.

[†] Nonetheless, the Z-transfer function borrows from the Laplace terminology so the term *pole* is sometimes used for denominator coefficients and the term *zeros* for numerator coefficients.

Biosignal and Medical Image Processing

be obtained from Equation 4.10 by applying the time-shift interpretation (i.e., Equation 4.7) to the term z^{-n} :

$$y[n] = \sum_{k=0}^{K-1} b[k]x[n-k] - \sum_{\lambda=1}^L a[\ell]y[n-\lambda] \quad (4.12)$$

This equation is derived assuming that $a[0] = 1$ as specified in Equation 4.9. In addition to its application to filters, Equation 4.12 can be used to implement any linear process given the a and b coefficients. We will find it can be used to represent IIR filter later in this chapter and the autoregressive moving average (ARMA) model described in Chapter 5.

Equation 4.12 is the fundamental equation for implementation of all linear digital filters. Given the a and b weights (or coefficients), it is easy to write a code that is based on this equation. Such an exercise is unnecessary, as MATLAB has again beaten us to it. The MATLAB routine `filter` uses Equation 4.12 to implement a wide range of digital filters. This routine can also be used to realize any linear process given its Z-transform transfer function as shown in Example 4.2. The calling structure is

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

where x is the input, y the output, and b and a are the coefficients in Equation 4.12. If $a = 1$, all higher coefficients are of a in Equation 4.12 are zero and this equation reduces to the convolution equation (Equation 2.55); hence, this routine can also be used to perform the convolution of b and x .

Designing a digital filter is just a matter of finding coefficients, $a[\ell]$ and $b[k]$, that provide the desired spectral shaping. This design process is aided by MATLAB routines that generate the a and b coefficients that produce a desired frequency response.

In the Laplace domain, the frequency spectra of a linear system can be obtained by substituting the real frequency, $j\omega$, for the complex frequency, s , into the transfer function. This is acceptable as long as the input signals are restricted to sinusoids and only sinusoids are needed to determine a spectrum. A similar technique can be used to determine the frequency spectrum of the Z-transfer function $H(z)$ by substituting $e^{j\omega}$ for z . With this substitution, Equation 4.12 becomes

$$H[m] = \frac{\sum_{k=0}^K b[k]e^{-j\omega k}}{\sum_{\lambda=0}^L a[\ell]e^{-j\omega \lambda}} = \frac{\sum_{k=0}^K b(k)e^{-2\pi mn/N}}{\sum_{\lambda=0}^L a[\ell]e^{-2\pi mn/N}} = \frac{FT(b[k])}{FT(a[\ell])} \quad (4.13)$$

where FT indicates the Fourier transform. As with all Fourier transforms, the actual frequency can be obtained from the harmonic number m after multiplying by f_s/N or $1/(NT_s)$.

EXAMPLE 4.2

A system is defined by the digital transfer function below. Plot its frequency spectrum: magnitude in dB and phase in degrees. Also plot the time-domain response of the system to an impulse input (i.e., the system's impulse response). Assume a sampling frequency of $f_s = 1$ kHz and make $N = 512$. These are both optional: f_s is provided so that the frequency curve can be plotted in Hz instead of relative frequency, and N is chosen to be large enough to produce a smooth frequency curve. (In this example, it is actually much larger than needed for a smooth curve: $N = 128$ would be sufficient).

$$H[z] = \frac{0.2 + 0.5z^{-1}}{1 - 0.2z^{-1} + 0.8z^{-2}}$$

Solution

In filter design, we usually work from a desired frequency spectrum, determining the a and b weights that generate a digital filter to approximate the desired spectrum. In this example, we work the other way around: we are given the coefficients in terms of the digital transfer function and are asked to find the spectrum. From the transfer function, we note that $a[0] = 1$ (as required), $a[1] = 0.2$, $a[2] = 0.8$, $b[0] = 0.2$, and $b[1] = 0.5$. We apply Equation 4.13 to find $H[m]$, then convert to frequency in Hz by multiplying m by f_s/N . To find the impulse response, we generate an impulse function (1.0 followed by zeros) and determine the output by implementing Equation 4.12 through the MATLAB filter routine.

```
% Example 4.2 Plot the Frequency characteristics and impulse
% response digital system given the digital transfer function
%
fs = 1000; % Sampling frequency (given)
N = 512; % Number of points (given)
% Define a and b coefficients based on H(z)
a = [1 - .2 .8]; % Denominator of transfer function
b = [.2 .5]; % Numerator of transfer function
%
H = fft(b,N)./fft(a,N); % Compute H(m). Eq. 4.13
Hm = 20*log10(abs(H)); % Get magnitude in dB
Theta = (angle(H))*360/(2*pi); % and phase in deg.
f = (1:N)*fs/N; % Frequency vector for plotting
.....plot H(m)and Theta(m),label axes, turn on grid.....
%
% Compute the Impulse Response
x = [1, zeros(1,N-1)]; % Generate an impulse input
y = filter(b,a,x); % Apply Eq. 4.12
.....plot first 60 points of x for clarity and label.....
```

The magnitude and phase characteristics of the digital transfer function given in Example 4.2 are shown in Figure 4.4. The impulse response of this transfer function is shown in Figure 4.5 and has the form of a damped sinusoid. The digital filters described in the rest of this chapter use a straightforward application of these linear system concepts. Again, the design of digital filters is only a question of determining the $a[n]$ and $b[n]$ coefficients to produce the desired frequency characteristics.

4.4 Finite Impulse Response Filters

FIR filters are moving average filters, generally with uneven weights. In FIR filters, the filter weights (coefficients) define the impulse response, so of course the impulse response will be finite. FIR filters are implemented by convolving the weights with the input signal, which is mathematically the same as taking a running weighted average over the input signal. So, the general equation for the implementation of an FIR filter is the convolution equation presented in Chapter 2 (Equation 2.55) and repeated here with MATLAB-type indexing (i.e., from 1 to N instead of 0 to $N - 1$):

$$y[n] = \sum_{k=1}^{K-1} b[k]x[n-k] \quad (4.14)$$

where $b[k]$ defines the weights (recall, these weights are also called the *filter coefficients* or simply *coefficients*, the *weighting function*, or *impulse response*), K is number of weights or filter length,

Biosignal and Medical Image Processing

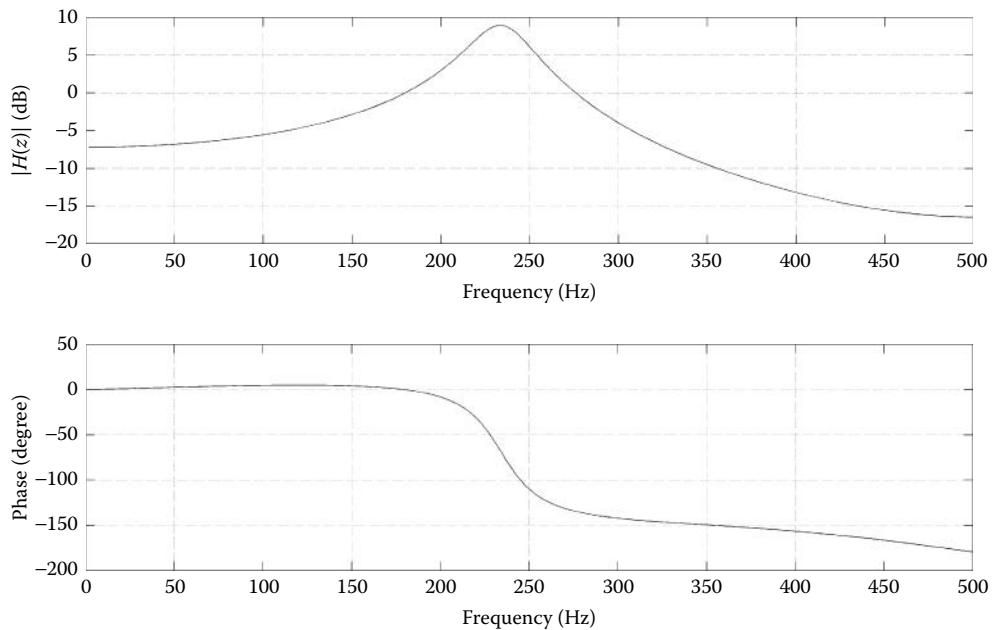


Figure 4.4 The frequency characteristic (magnitude and phase) of the digital transfer function given in Example 4.2.

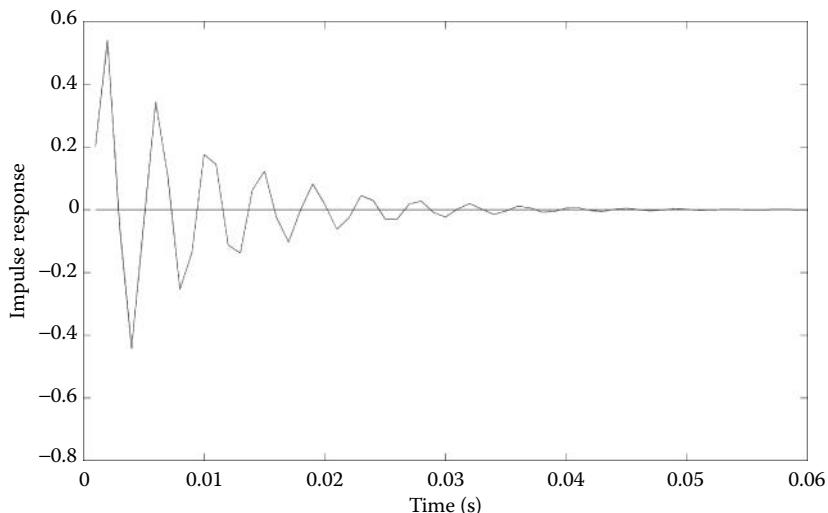


Figure 4.5 Impulse response of the system represented by the digital transfer function given in Example 4.2.

$x[n]$ is the input, and $y[n]$ is the output. FIR filters can be implemented in MATLAB using the `conv` routine. Equation 4.14 is a specific case of Equation 4.12, in which there are no a coefficients except $a[0] = 1$ (recall 0 has an index of 1 in MATLAB). So, FIR filters can also be implemented using the `filter` routine where the a coefficients are set to a value of 1.0:

```
y = filter(b,1,x); % Implementation of an FIR filter.
```

4.4 Finite Impulse Response Filters

The spectral characteristics of an FIR filter can be found from Equation 4.13. Substituting $a = a[0] = 1.0$, Equation 4.13 reduces to

$$X[m] = \sum_{k=0}^{K-1} b[k] e^{-j2\pi mn/N} = FT(b[k]) \quad (4.15)$$

This equation is supported by linear systems theory that states that the spectrum of a linear (LTI) system is the Fourier transform of the impulse response.

In the moving average FIR filter defined by Equation 4.14, each sample in the output is the average of the last N points in the input. Such a filter, using only current and past data, is termed a *causal* filter. These filters follow the cause-and-effect principle in that the output is a function of past, and perhaps present, inputs. All real-time systems are causal since they do not have access to the future. They have no choice but to operate on current and past values of a signal. However, if the data are stored in a computer, it is possible to use future signal values along with current and past values to compute an output signal, that is, future with respect to a given sample in the output signal. Filters (or systems) that use future values of a signal in their computation are *noncausal*.

The motivation for using future values in filter calculations is provided in Figure 4.6. The upper curve in Figure 4.6a is the response of the eyes to a target that jumps inward in a step-like manner. (The curve is actually the difference in the angle of the two eyes with respect to the straight-ahead position.) These eye-movement data are corrupted by 60-Hz noise riding on top of the signal, a ubiquitous problem in the acquisition of biological signals. A simple 10-point, equal-weight, moving average filter was applied to the noisy data and this filter did a good job of removing the noise as shown in Figure 4.6a (lower two curves). The 10-point, moving average filter was applied in two ways: as a causal filter using only current and past values of the eye-movement data, lower curve (Figure 4.6a), and as a noncausal filter applying the filter weights to

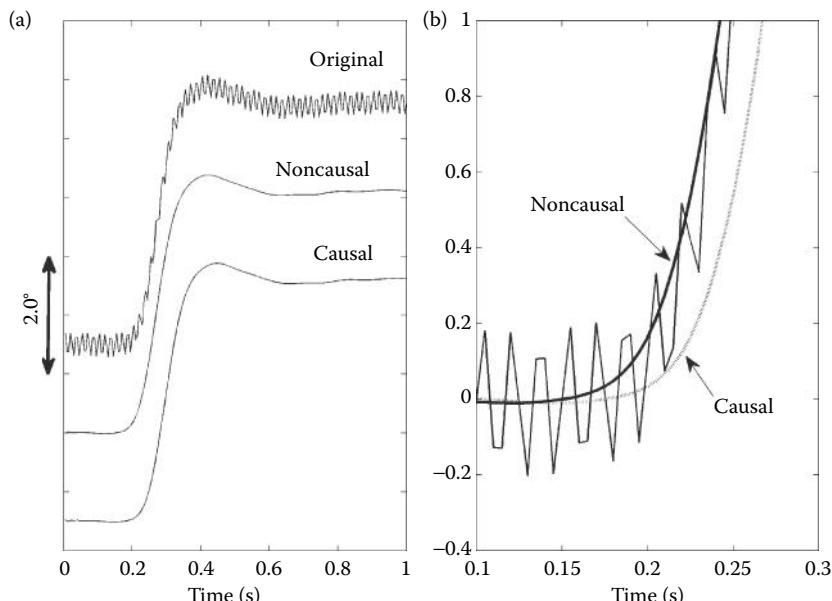


Figure 4.6 (a) Eye-movement data containing 60 Hz noise and the same response filtered with a 10-point, equal-weight, moving average filter applied in a causal and noncausal approach. (b) Detail of the initial response of the eye movement showing the causal and noncausal filtered data superimposed. The noncausal filter overlays the original data while the causal filter produces a time shift in the response.

Biosignal and Medical Image Processing

an approximately equal number of past and future values, middle curve (Figure 4.6a). The causal filter was implemented using MATLAB's `conv` routine (i.e., Equation 4.14), with the noncausal filter using the `conv` routine with the option '`same`':

```
y = conv(x, b, 'same'); % Apply a noncausal filter.
```

where x is the noisy eye-movement signal and b is an impulse response consisting of 10 equal value samples ($b = [1 1 1 1 1 1 1 1 1]/10$) to create a moving average. When the '`same`' option is invoked, the convolution algorithm returns only the center section of output, effectively shifting future values into the present. The equation for a noncausal filter is similar to Equation 4.14, but with a shift in the index of x .

$$y[n] = \sum_{k=1}^K b[k]x[n - k + K/2] \quad (4.16)$$

Both filters do a good job of reducing the 60-cycle noise, but the causal filter has a slight delay. In Figure 4.6b, the initial responses of the two filters are plotted superimposed over the original data, and the delay in the causal filter is apparent. Eliminating the delay or time shift inherent in causal filters is the primary motivation for using noncausal filters. In Figure 4.6, the time shift in the output is small, but can be much larger if filters with longer impulse responses are used or if the data are passed through multiple filters. However, in many applications, a time shift does not matter, so a noncausal filter is adequate.

The noncausal implementation of a three-point moving average filter is shown in Figure 4.7. This filter takes one past and one current sample from the input along with one future sample to construct the output sample. In Figure 4.7, a slightly heavier arrow indicates a future sample. This operation can be implemented in MATLAB using an impulse function of $b = [1/3 1/3 1/3]$ and the `conv` routine with option '`same`'.

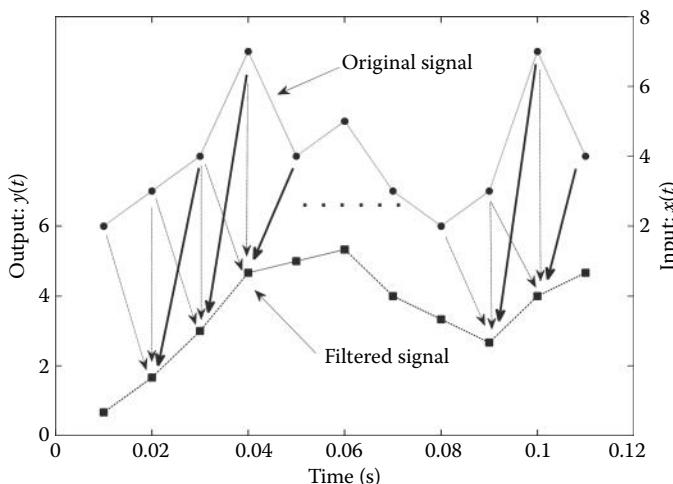


Figure 4.7 The procedure for constructing a three-point moving average. The average of each set of three points from the input makes up the output point. One point is taken from the past, one from the present, and one from the future (darker arrow). This averaging process slides across the input signal to produce the output. Endpoints are usually treated by applying zero-padding to the input signal. This process is identical to convolving the input with an impulse response of $b = [1/3 1/3 1/3]$ in conjunction with symmetrical convolution (`conv` with option '`same`'). The time axis is scaled assuming a sampling frequency of 100 Hz.

EXAMPLE 4.3

Find the magnitude and phase spectra of a 3-point moving average filter and a 10-point moving average filter, both with equal weights. Plot the two spectra superimposed. Assume a sampling frequency of 250 Hz.

Solution

The frequency spectrum of any FIR filter can be determined by taking the Fourier transform of its filter coefficients, which is also the impulse response (Equation 4.15). However, since the impulse responses of many FIR filters can be quite short (only three points for a three-point moving average filter), it is appropriate to pad the response before taking the Fourier transform. In this example, we first construct the two filter impulse responses, then take the Fourier transform using `fft`, but pad the data out to 256 points to get smooth plots. We then plot the valid points of the magnitude and phase from the complex Fourier transform using a frequency vector based on the 250 Hz sampling rate.

```
% Example 4.3 Spectra of moving-average FIR filter
%
fs = 250; % Sampling frequency
% Construct moving average filters
b3 = [1 1 1]/3;
b10 = [1 1 1 1 1 1 1 1 1 1]/10; % Construct filter impulse response
%
B3 = fft(b3,256); % Find filter system spectrum
B3_mag = abs(B3); % Compute magnitude spectrum
Phase3 = unwrap(angle(B3)); % Compute phase angle and unwrap
Phase3 = Phase3*360/(2*pi); % Phase angle in degrees
.....repeat for b10 and plot the magnitude and phase spectra.....
```

Results

Figure 4.8 shows the magnitude and phase spectra of the two filters assuming a sampling frequency of 250 Hz. Both are lowpass filters with a cutoff frequency of ~ 40 Hz for the 3-point moving average and ~ 16 Hz for the 10-point moving average filter. In addition to a lower cutoff frequency, the 10-point moving average filter has a much sharper rolloff. The phase characteristics decrease linearly from 0 to ~ -160 deg in a repetitive manner.

The moving average filter weights used in Example 4.3 consist of equal values; however, there is no reason to give the filter coefficients the same value. We can make up any impulse response we choose and quickly determine its spectral characteristics using the Fourier transform as in this example. The question is how do we make up filter weights that do what we want them to do in the frequency domain? This inverse operation, going from a desired frequency response to filter weights $b[k]$, is known as filter design. We could use trial and error, and sometimes that is the best approach, but in the case of FIR filter design, there are straightforward methods to get from the spectrum we want to the appropriate filter coefficients. The FIR filter design is straightforward in principle: since the frequency response of the filter is the Fourier transform of the filter coefficients (Equation 4.15), the desired impulse response is the *inverse Fourier transform* of the desired frequency response. As is often the case, the details are a little more involved, but FIR filter design is still fairly easy.

4.4.1 FIR Filter Design and Implementation

To design a filter, we start with the desired frequency characteristic. Then all we need to do is take the inverse Fourier transform. We only need the magnitude spectrum, because FIR filters

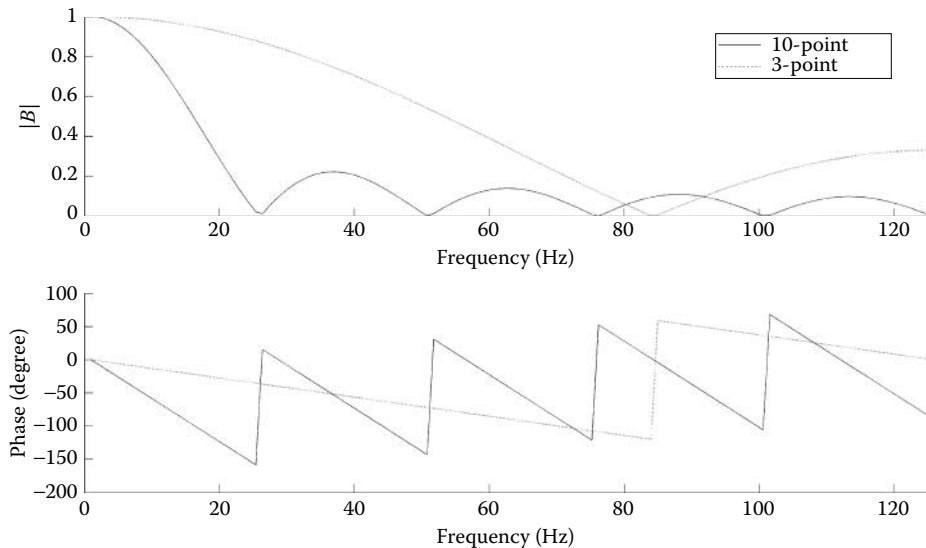


Figure 4.8 The magnitude (upper) and phase (lower) spectra of a 3-point and a 10-point moving average filter. Both filters show lowpass characteristics with a cutoff frequency, f_c , of ~ 40 Hz for the 3-point moving average and ~ 16 Hz for the 10-point moving average assuming the sampling frequency of 250 Hz. The phase characteristics change linearly and repetitively between 0 and ~ -160 degree. (Note that the MATLAB's unwrap routine has been applied, so the jumps in the phase curve are characteristic of the filter and are not due to transitions greater than 360 degree.)

have linear phase, which is uniquely determined by the magnitude spectrum. While there may be circumstances where some specialized frequency characteristic is needed, usually we just want to separate out a desired range of frequencies from everything else and we want that separation to be as sharp as possible. In other words, we usually prefer an ideal filter such as that shown in Figure 1.9a with a particular cutoff frequency. The ideal lowpass filter in Figure 1.9a has the frequency characteristics of a rectangular window and is sometimes referred to as a *rectangular window filter*.*

This spectrum of an ideal filter is a fairly simple function, so we ought to be able to find the inverse Fourier transform analytically from the defining equation given in Chapter 3 (Equation 3.17). (This is one of those cases where we are not totally dependent on MATLAB.) When using the complex form, we must include the negative frequencies so that the desired filter's frequency characteristic is as shown in Figure 4.9. Frequency is shown in radians/s to simplify the derivation of the rectangular window impulse response. (Recall $\omega = 2\pi f$.)

To derive of the impulse response of a rectangular window filter, we apply the continuous inverse Fourier transform equation (the continuous version of Equation 3.20) to the window function in Figure 4.9. Since frequency is in radians, we use $k\omega$ for $2\pi mf$:

$$b[k] = \frac{1}{2\pi} \int_{-\pi}^{\pi} B(\omega) e^{jk\omega} d\omega = \frac{1}{2\pi} \int_{-\omega_c}^{\omega_c} 1 e^{jk\omega} d\omega \quad (4.17)$$

* This filter is sometimes called a *window filter*, but the term *rectangular window filter* is used in this book so as not to confuse this filter with a “window function” as described in Chapter 3. This can be particularly confusing since, as shown in Figure 4.9, rectangular window filters use window functions!

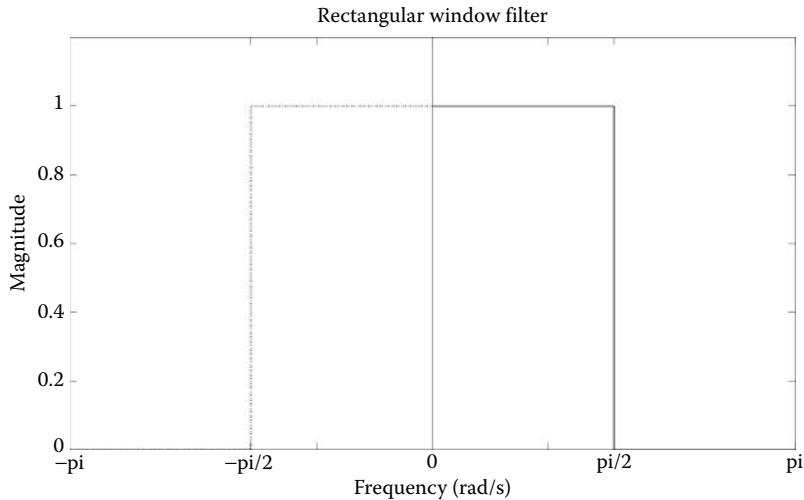


Figure 4.9 The magnitude spectrum of a rectangular window filter is an ideal filter. The negative frequency portion of this filter is also shown because it is needed in the computation of the complex inverse Fourier transform. The frequency axis is in radians per second and normalized to a maximum value of 1.0.

since the window function, $B(\omega)$, is 1.0 between $\pm\omega_c$ and zero elsewhere. Integrating and putting in the limits:

$$b[k] = \frac{1}{2\pi} \frac{e^{jk\omega}}{j} \Big|_{-\omega_c}^{\omega_c} = \frac{1}{k\pi} \frac{e^{jk\omega_c} - e^{-jk\omega_c}}{2j} \quad (4.18)$$

The term $e^{jk\omega_c} - e^{-jk\omega_c}/2j$ is the exponential definition of the sine function and equals $\sin(k\omega_c)$. So, the impulse response of a rectangular window is

$$b[k] = \frac{\sin(k\omega_c)}{\pi k} = \frac{\sin(2\pi f_c k)}{\pi k} \quad -(L-1)/2 \leq k \leq (L-1)/2 \quad (4.19)$$

where the filter length, L , the number of filter coefficients must be an odd number to make $b[k]$ symmetrical about $k=0$. When $k=0$, the denominator goes to zero and the actual value of $b[0]$ can be obtained by applying the limits and noting that $\sin(x) \rightarrow x$ as x becomes small:

$$b[0] = \lim_{k \rightarrow 0} \left| \frac{\sin(2\pi k f_c)}{\pi k} \right| = \lim_{k \rightarrow 0} \left| \frac{2\pi k f_c}{\pi k} \right| = 2f_c \quad (4.20)$$

Substituting $\omega = f/2\omega_c$ into Equation 4.20 gives the value of $b[0]$ in radians/s:

$$b[0] = 2f_c = \frac{2\omega_c}{2\pi} = \frac{\omega_c}{\pi} \quad (4.21)$$

The impulse response of a rectangular window filter has the general form of a *sinc* function: $\text{sinc}(x) = \sin(x)/x$. The impulse response, $b[k]$, produced by Equation 4.19 is shown for 65 samples (k ranges between ± 32) and two values of f_c in Figure 4.10. These cutoff frequencies are relative to the sampling frequency as explained below.

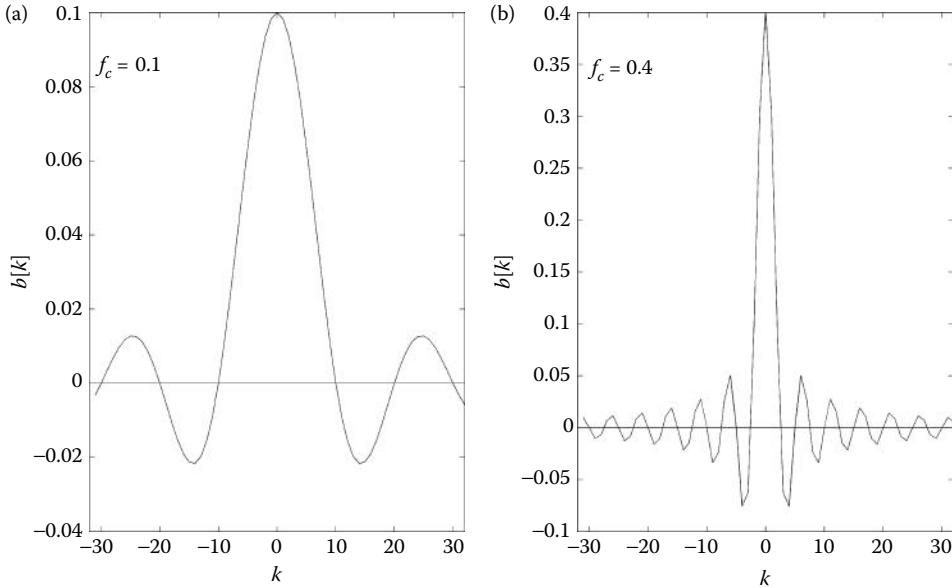


Figure 4.10 The impulse response of a rectangular window filter (Equation 4.19) for $L = 65$ coefficients (k ranges between ± 32). The cutoff frequencies are given relative to the sampling frequency, f_s , as is common for digital filter frequencies. (a) Lowpass filter with a relative cutoff frequency of 0.1. (b) Lowpass filter with a higher relative cutoff frequency of 0.4 Hz.

From Figure 4.9, we can see that the cutoff frequency, ω_c , is relative to a maximum frequency of π . Converting frequency to Hz, recall that the maximum frequency in the spectrum of a sampled signal is f_s . Frequency specifications for digital filters are usually described relative to either the sampling frequency, f_s , or to half the sampling frequency $f_s/2$. (Recall that $f_s/2$ is the Nyquist frequency.) So, the cutoff frequency, f_c , in Hz is relative to the sampling frequency, f_s :

$$f_{\text{actual}} = f_c/f_s \quad (4.22)$$

In the MATLAB filter design routines included in the Signal Processing Toolbox, the cutoff frequencies are relative to $f_s/2$, the Nyquist frequency, so

$$f_{\text{actual}} = f_c/f_s/2 \text{ (in MATLAB filter routines)} \quad (4.23)$$

The symmetrical impulse response shown in Figure 4.10 has both positive and negative values of k . Since MATLAB requires indexes to be positive, we need to shift the output index, k , by $L/2 + 1$. If L is mistakenly given as an even number, it is adjusted so that $(L - 1)/2$ is an integer as shown in the next example.

Since we are designing FIR filters with a finite number of filter coefficients (i.e., L is finite), we need an impulse response that is also finite. Unfortunately, the solution to Equation 4.19 is an impulse response that is theoretically infinite: it does not go to zero for finite values of L . In practice, we simply truncate the impulse response equation (Equation 4.19) to some finite value L . You might suspect that limiting a filter's impulse response by truncation has a negative impact on the filter's spectrum. In fact, truncation has two adverse effects: the filter no longer has an infinitely sharp cutoff, and oscillations are produced in the filter's spectrum. These adverse effects are demonstrated in the next example, where we show the spectrum of a rectangular window filter truncated to two different lengths.

EXAMPLE 4.4

Use the Fourier transform to find the magnitude spectrum of the rectangular window filter given by Equation 4.23 for two different lengths: $L = 18$ and $L = 66$. Use a cutoff frequency of 300 Hz assuming a sampling frequency of 1 kHz (i.e., a relative frequency of 0.3).

Solution

First, generate the filter's impulse response, $b[k]$, using Equation 4.19. This can be done by generating the negative half of b , adding $b[0] = 2f_c$, then adding the positive half of b by flipping the negative half. Since the requested L is even, the program rounds up to produce 19 and 67 coefficients. After calculating the impulse response, we find the spectrum by taking the Fourier transform of the response and plot the magnitude spectrum.

```
% Example 4.4 Generate two rectangular window impulse responses and
% find their magnitude spectra.
%
N = 256; % Number of samples for plotting
fs = 1000; % Sampling frequency
f = (1:N)*fs/N; % Frequency vector for plotting
fc = 300/fs; % Cutoff frequency (normalized to fs)
L = [18 66]; % Requested filter lengths
for m = 1:2 % Loop for the two filter lengths
    k = -floor(L(m)/2):-1; % Construct k for negative b[k];
    b = sin(2*pi*fc*k)./(pi*k); % Construct negative b[k]
    b = [b 2*fc, fliplr(b)]; % Rest of b
    H = fft(b,N); % Calculate spectrum
    subplot(1,2,m); % Plot magnitude spectrum
    plot(f(1:N/2), abs(H(1:N/2)), 'k');
    .....labels and title.....
end
```

The spectrum of this filter is shown in Figure 4.11 to have two artifacts associated with finite length. The oscillation in the magnitude spectrum is another example of the Gibbs artifact first encountered in Chapter 3; it is due to the truncation of the infinite impulse response. In addition, the slope is less steep when the filter's impulse response is shortened.

We can probably live with the less than ideal slope (we should never expect to get an ideal anything in the real world), but the oscillations in the spectrum are serious problems. Since the Gibbs artifacts are due to truncation of an infinite function, we might reduce them if the impulse response were tapered toward zero rather than abruptly truncated. Some of the tapering window functions described in Chapter 3 can be useful in mitigating the effects of the abrupt truncation. In Chapter 3, window functions such as the Hamming window are used to improve the spectra obtained from short data sets, and FIR impulse responses are usually short. There are many different window functions, but the two most popular and most useful for FIR impulse responses are the Hamming and the Blackman–Harris windows described in Section 3.2.4. The Hamming window equation is given in Equation 3.26 and the Blackman–Harris window in Equation 3.27, both repeated here:

$$w[n] = 0.5 - 0.46 \cos\left(\frac{2\pi n}{N}\right) \quad (4.24)$$

$$w[n] = a_0 + a_1 \cos\left(\frac{2\pi}{N}n\right) + a_2 \cos\left(\frac{2\pi}{N}2n\right) + a_3 \cos\left(\frac{2\pi}{N}3n\right) \quad (4.25)$$

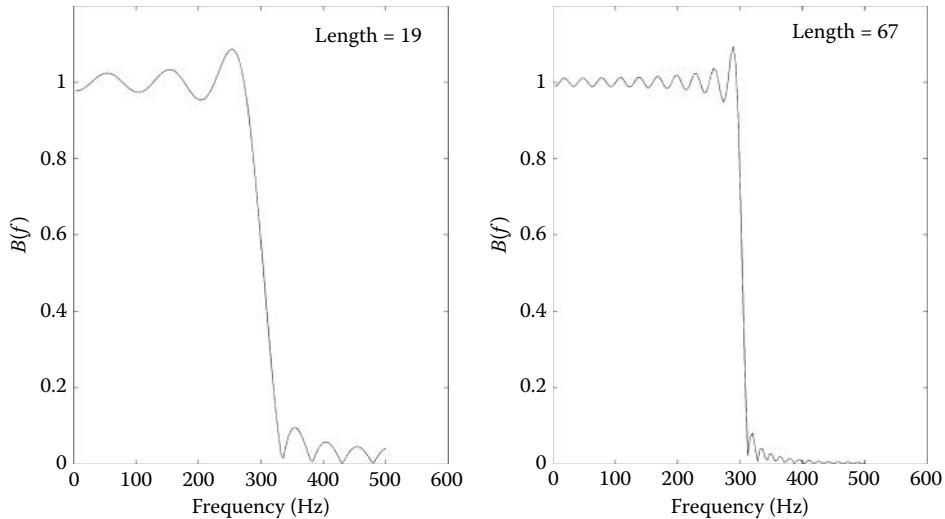


Figure 4.11 Magnitude spectra of two FIR filters based on an impulse derived from Equation 4.19. The impulse responses are abruptly truncated at 19 and 69 coefficients. The lowpass cutoff frequency is 300 Hz for both filters at an assumed sample frequency of 1 kHz. The oscillations seen are the Gibbs artifacts and are due to the abrupt truncation of what should be an infinite impulse response. Like the Gibbs artifacts seen in Chapter 3, they do not diminish with increasing filter length, but do increase in frequency.

where N is the length of the window, which should be the same length as the filter (i.e., $N = L$). Typically, $a_0 = 0.35875$; $a_1 = 0.48829$; $a_2 = 0.14128$; $a_3 = 0.01168$.

The example below applies the Blackman–Harris window the filters of Example 4.4. Coding the Hamming window is part of one of the problems at the end of this chapter.

EXAMPLE 4.5

Apply a Blackman–Harris window to the rectangular window filters used in Example 4.4. (Note that the word “window” is used in two completely different contexts in the last sentence. The first “window” is a time-domain function; the second is a frequency-domain function. Engineering terms are not always definitive.) Calculate and display the magnitude spectrum of the impulse functions after they have been “windowed.”

Solution

Generate a Blackman–Harris window equivalent to the length of the filter. Use a vector of $n = -\text{floor}(L/2):\text{floor}(L/2)$ to generate the window. Apply it to the filter impulse responses using point-by-point multiplication (i.e., using the $\cdot*$ operator). Modify the last example applying the window to the filter’s impulse response before taking the Fourier transform.

```
% Example 4.5 Apply the Blackman-Harris window to the rectangular window
impulse
% responses developed in Example 4.4.
%
.....Same code as in Example 4.4 up to.....
b = [b 2*fc, fliplr(b)]; % Rest of b
```

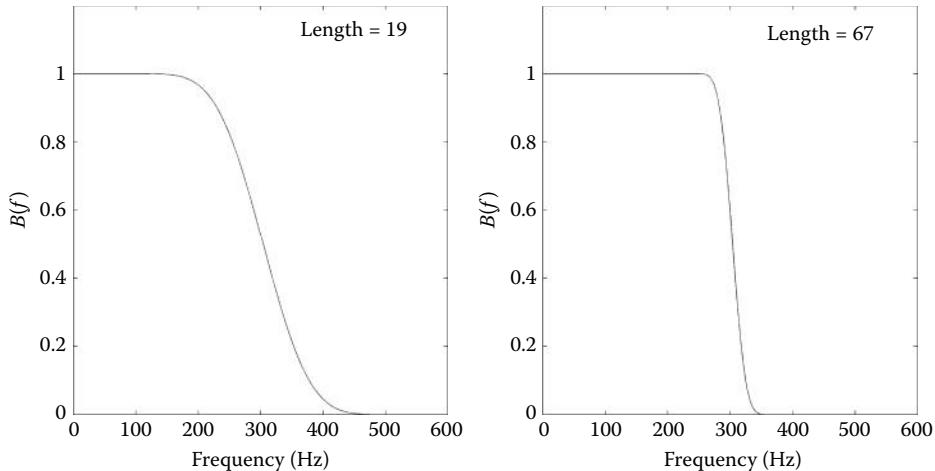


Figure 4.12 Magnitude spectrum of the FIR filters shown in Figure 4.11 except a Blackman–Harris window was applied to the filter coefficients. The Gibbs oscillations seen in Figure 4.11 are no longer visible.

```

N_w = length(b); % Window length
n = -floor(L(m)/2):floor(L(m)/2); % Window vector
w_B = 0.35875 + 0.48829*cos(2*pi*n/N_w) +...
      0.14128*cos(4*pi*n/N_w) + 0.01168*cos(6*pi*n/N_w);
b = b .* w_B; % Apply window to impulse response
H = fft(b,N); % Calculate spectrum
.....same code as in Example 4.4, plot and label.....

```

The Blackman–Harris window is easy to generate in MATLAB (even easier if using the Signal Processing Toolbox window routines) and, when applied to the impulse responses of Example 4.4, substantially reduces the oscillations as shown in Figure 4.12. The filter rolloff is still not that of an ideal filter, but becomes steeper with increased filter length. Of course, increasing the length of the filter increases the computation time required to apply the filter to a data set, a typical engineering compromise.

The next example applies a rectangular window lowpass filter to a signal of human respiration that was obtained from a respiratory monitor.

EXAMPLE 4.6

Apply a lowpass rectangular window filter to the 10-min respiration signal shown in the top trace of Figure 4.13 which can be found as variable `resp` in file `Resp.mat`. The signal is sampled at 12.5 Hz. The low sampling frequency is used because the respiratory signal has a very low bandwidth. Use a cutoff frequency of 1.0 Hz and a filter length of 65. Use the Blackman–Harris window to truncate the filter's impulse response. Plot the original and filtered signal.

Solution

Reuse the code in Example 4.4 and apply the filter to the respiratory signal using convolution. For variety, the Blackman–Harris filter will be generated using the Signal Processing Toolbox routine, `blackman`. MATLAB's `conv` routine is invoked using the option '`'same'`' to implement a causal filter and avoid a time shift in the output.

Biosignal and Medical Image Processing

```

Example 4.6 Apply a rectangular window filter to noisy data
%
load Resp;                      % Get data
N = length(resp);                % Get data length
fs = 12.5;                       % Sample frequency in Hz
t = (1:N)/fs;                    % Time vector for plotting
L = 65;                          % Filter lengths
fc = 1.0/fs;                     % Filter cutoff frequency: 1.0 Hz
plot(t,resp+1,'k');              % Plot original data, offset for clarity
k = -floor(L/2):-1;               % Construct k for negative b[k];
b = sin(2*pi*fc*k)./(pi*k);     % Construct negative b[k]
b = [b 2*fc, fliplr(b)];         % Rest of b
b = b.*blackman(L);              % Apply Blackman window
%
y = conv(resp,b,'same');          % Apply filter (causal)
plot(t,y);                        % Plot filtered data

```

Results

The filtered respiratory signal is shown in the lower trace of Figure 4.13. The filtered signal is smoother than the original signal as the noise riding on the original signal has been eliminated.

The FIR filter coefficients for highpass, bandpass, and bandstop filters can also be derived by applying an inverse FT to rectangular spectra having the appropriate associated shape. These equations have the same general form as Equation 4.19 except they may include additional terms:

$$b[k] = \begin{cases} \frac{-\sin(2\pi kf_c)}{\pi k} & k \neq 0 \\ 1 - 2f_c & k = 0 \end{cases} \quad \text{Highpass} \quad (4.26)$$

$$b[k] = \begin{cases} \frac{\sin(2\pi kf_h) - \sin(2\pi kf_l)}{\pi k} & k \neq 0 \\ 2(f_h - f_l) & k = 0 \end{cases} \quad \text{Bandpass} \quad (4.27)$$

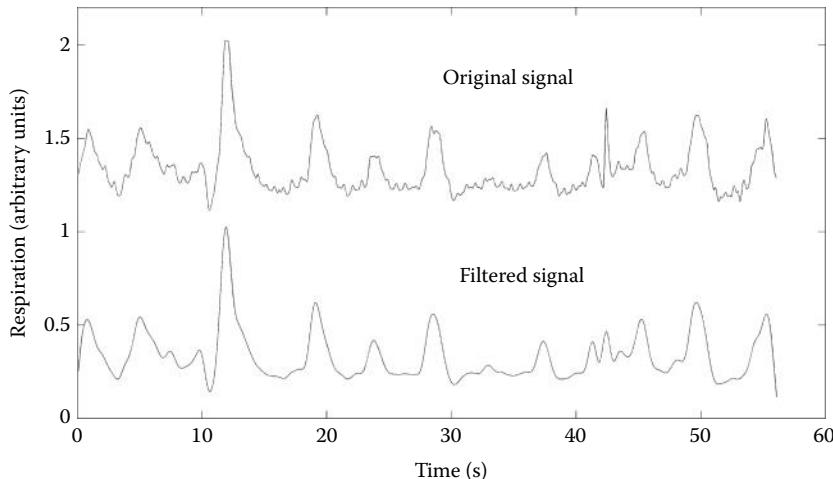


Figure 4.13 Results from Example 4.6. The output of a respiratory monitor is shown in the upper trace to have some higher-frequency noise. In the lower trace, the signal has been filtered with an FIR rectangular lowpass filter with a cutoff frequency of 1.0 Hz and a filter length of 65. (Original data from PhysioNet, Goldberger et al., 2000.)

$$b[k] = \begin{cases} \frac{\sin(2\pi kf_l)}{\pi k} - \frac{\sin(2\pi kf_h)}{\pi k} & k \neq 0 \\ 1 - 2(f_h - f_l) & k = 0 \end{cases} \quad \text{Bandstop} \quad (4.28)$$

The order of highpass and bandstop filters should always be even, so the number of coefficients in these filters should be odd. The next example applies a bandpass filter to the EEG introduced in Chapter 1.

EXAMPLE 4.7

Apply a bandpass filter to the EEG data in file ECG.mat. Use a lower cutoff frequency of 6 Hz and an upper cutoff frequency of 12 Hz. Use a Blackman–Harris window to truncate the filter's impulse to 129 coefficients. Plot the data before and after bandpass filtering. Also plot the spectrum of the original signal and superimpose the spectrum of the bandpass filter.

Solution

Construct the filter's impulse response using a bandpass equation (Equation 4.30). Use the same strategy for constructing the $b[k]$ as in the previous three examples. Note that $b[0] = 2f_h - 2f_l$. Recall that the sampling frequency of the EEG signal is 50 Hz.

After applying the filter and plotting the resulting signal, compute the signal spectrum, using the Fourier transform, and plot. The filter's spectrum is found by taking the Fourier transform of the impulse response ($b[k]$) and is plotted superimposed on the signal spectrum.

```
% Example 4.7 Apply a bandpass filter to the EEG data in file ECG.mat.
%
load EEG; % Get data
fs = 50; % Sample frequency
N = length(eeg); % Find signal length
fh = 12/fs; % Set bandpass cutoff frequencies
fl = 6/fs; % Set number of weights as 129
L = 129; % Construct k for negative b[k];
k = -floor(L/2):-1;
b = sin(2*pi*fh*k)./(pi*k) - sin(2*pi*fl*k)./(pi*k); % Neg. b[k]
b = [b 2*(fh-fl), fliplr(b)]; % Rest of b[k]
b = b .* blackman(L)'; % Apply Blackman-Harris
% window to filter coefficients
y = conv(eeg,b,'same'); % Filter using convolution
.....plot eeg before and after filtering.....
.....plot eeg and filter spectra.....
```

Bandpass filtering the EEG signal between 6 and 12 Hz reveals a strong higher-frequency oscillatory signal that is washed out by lower-frequency components in the original signal (Figure 4.14). This figure shows that filtering can significantly alter the appearance and interpretation of biomedical data. The bandpass spectrum shown in Figure 4.15 has the desired cutoff frequencies and, when compared to the EEG spectrum, is shown to reduce sharply the high- and low-frequency components of the EEG signal.

Implementation of other FIR filter types is found in the problem set. A variety of FIR filters exist that use strategies other than the rectangular window to construct the filter coefficients, and some of these are explored in the section on MATLAB implementation. One FIR filter of particular interest is the filter used to construct the derivative of a waveform, since the derivative is often of interest in the analysis of biosignals. The next section explores a popular filter for this operation.

Biosignal and Medical Image Processing

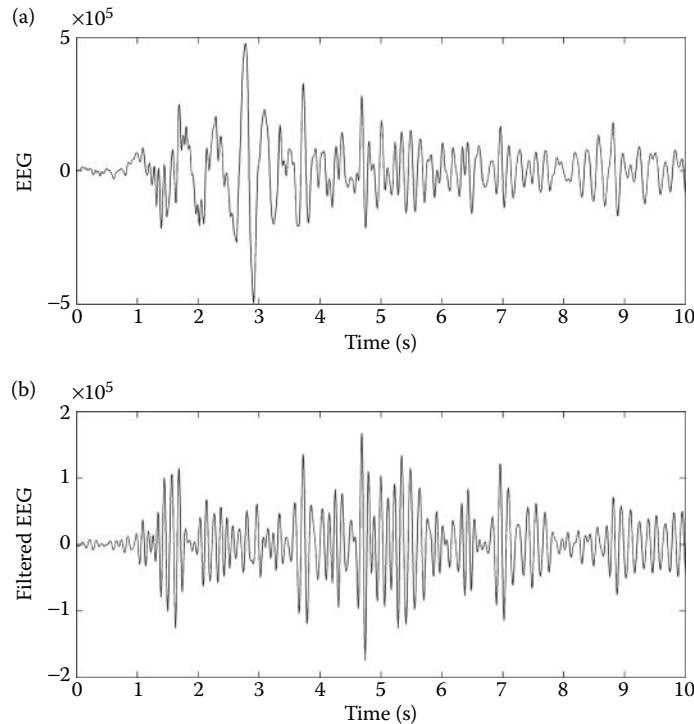


Figure 4.14 EEG signal before (a) and after (b) bandpass filtering between 6 and 12 Hz. A fairly regular oscillation is seen in the filtered signal.

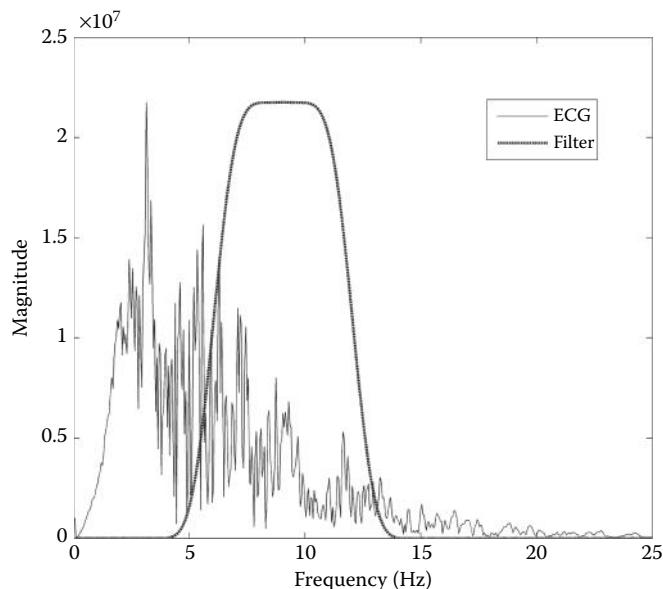


Figure 4.15 The magnitude spectrum of the EEG signal used in Example 8.5 along with the spectrum of a bandpass filter based in Equation 4.27. The bandpass filter range is designed with low and high cutoff frequencies of 6 and 12 Hz.

4.4.2 Derivative Filters: Two-Point Central Difference Algorithm

The derivative is a common operation in signal processing and is particularly useful in analyzing certain physiological signals. Digital differentiation is defined as $dx[n]/dn$ and can be calculated directly from the slope of $x[n]$ by taking differences:

$$\frac{dx[n]}{dn} = \frac{\Delta x[n]}{T_s} = \frac{x[n+1] - x[n]}{T_s} \quad (4.29)$$

This equation can be implemented by MATLAB's `diff` routine. This routine uses no padding, so the output is one sample shorter than the input. As shown in Figure 4.17, the frequency characteristic of the derivative operation increases linearly with frequency, so differentiation enhances higher-frequency signal components. Since the higher frequencies frequently contain a greater percentage of noise, this operation tends to produce a noisy derivative curve. The upper curve of Figure 4.16a is a fairly clean physiological motor response, an eye movement similar to that shown in Figure 4.6. The lower curve of Figure 4.16a is the velocity of the movement obtained by calculating the derivative using MATLAB's `diff` routine. Considering the relative smoothness of the original signal, the velocity curve obtained using Equation 4.28 is quite noisy.

In the context of FIR filters, Equation 4.29 is equivalent to a two-coefficient impulse function, $[+1, -1]/T_s$. (Note that the positive and negative coefficients are reversed by convolution, so they must be sequenced in reverse order in the impulse response.) A better approach to differentiation is to construct a filter that approximates the derivative at lower frequencies, but attenuates at higher frequencies that are likely to be only noise. The *two-point central difference algorithm* achieves just such an effect, acting as a differentiator at lower frequencies and a lowpass filter at higher frequencies. Figure 4.16b shows the same response and derivative when this algorithm is used to estimate the derivative. The result is a much cleaner velocity signal that still captures the peak velocity of the response.

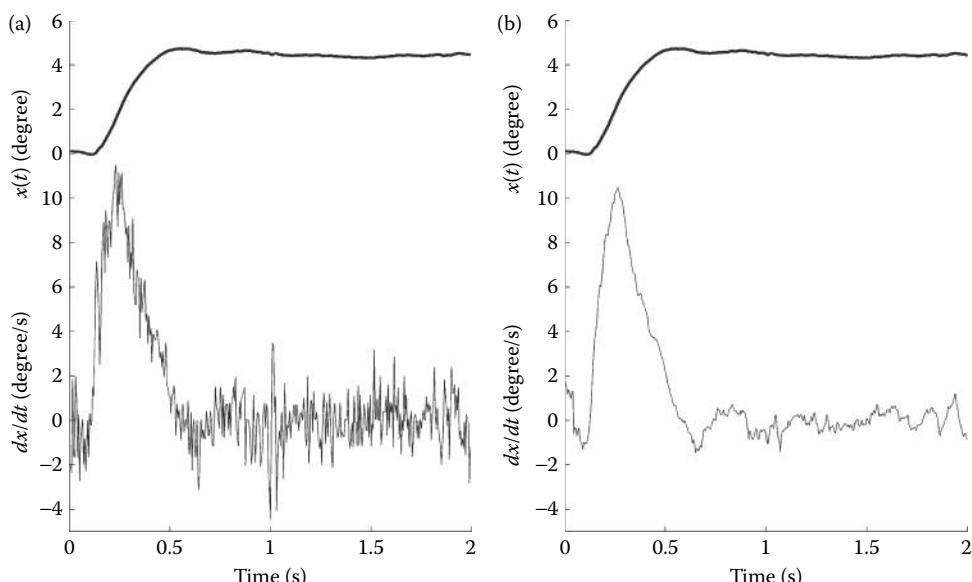


Figure 4.16 An eye-movement response to a step change in target depth is shown in the upper trace, and its velocity (i.e., derivative) is shown in the lower trace. (a) The derivative is calculated by taking the difference in adjacent points and scaling it by the sample interval (Equation 4.29). The velocity signal is noisy, even though the original signal is fairly smooth. (b) The derivative is computed using the two-point central difference algorithm (Equation 4.30) with a skip factor of 4.

Biosignal and Medical Image Processing

The two-point central difference algorithm still subtracts two points to get a slope, but the two points are no longer adjacent; they may be spaced some distance apart. Putting this in FIR filter terms, the algorithm is based on an impulse function containing two coefficients of equal but opposite sign spaced L points apart. The equation defining this differentiator is

$$\frac{dx[n]}{dn} = \frac{x[n+L] - x[n-L]}{2LT_s} \quad (4.30)$$

where L is now called the *skip factor* that defines the distance between the points used to calculate the slope and T_s is the sample interval. L influences the effective bandwidth of the filter, as is shown below. Equation 4.30 can be implemented as an FIR filter using filter coefficients:

$$b[k] = \begin{cases} 1/2LT_s & k = -L \\ -1/2LT_s & k = +L \\ 0 & k \neq \pm L \end{cases} \quad (4.31)$$

Note that the $+L$ coefficient is negative and the $-L$ coefficient is positive since the convolution operation reverses the order of $b[k]$. As with all FIR filters, the frequency response of this filter algorithm can be determined by taking the Fourier transform of $b[k]$. Since this function is fairly simple, it is not difficult to take the Fourier transform analytically as well as in the usual manner using MATLAB. Both methods are presented in the example below.

EXAMPLE 4.8

(a) Determine the magnitude spectrum of the two-point central difference algorithm analytically, and then (b) use MATLAB to determine the spectrum.

Analytical Solution

Starting with the equation for the discrete Fourier transform (Equation 3.18) substituting k for n in that equation:

$$X[m] = \sum_{n=0}^{N-1} b[k] e^{-j2\pi mnk/N}$$

Since $b[k]$ is nonzero only for $k = \pm L$, the Fourier transform, after the summation limits are adjusted for a symmetrical coefficient function with positive and negative n , becomes

$$X[m] = \sum_{k=-L}^L b[k] e^{-j2\pi mnk/N} = \frac{1}{2LT_s} e^{-j2\pi m(-L)/N} - \frac{1}{2LT_s} e^{-j2\pi mL/N}$$

$$X[m] = \frac{e^{-j2\pi m(-L)/N} - e^{-j2\pi mL/N}}{2LT_s} = \frac{-j \sin(2\pi mL/N)}{LT_s}$$

where L is the skip factor and N is the number of samples in the waveform. To put this equation in terms of frequency, note that $f = m/(NT_s)$; hence, $m = fNT_s$. Substituting in fNT_s for m and taking the magnitude of $X[m]$, now $X(f)$:

$$|X(f)| = \left| -j \frac{\sin(2\pi f LT_s)}{LT_s} \right| = \frac{|\sin(2\pi f LT_s)|}{LT_s}$$

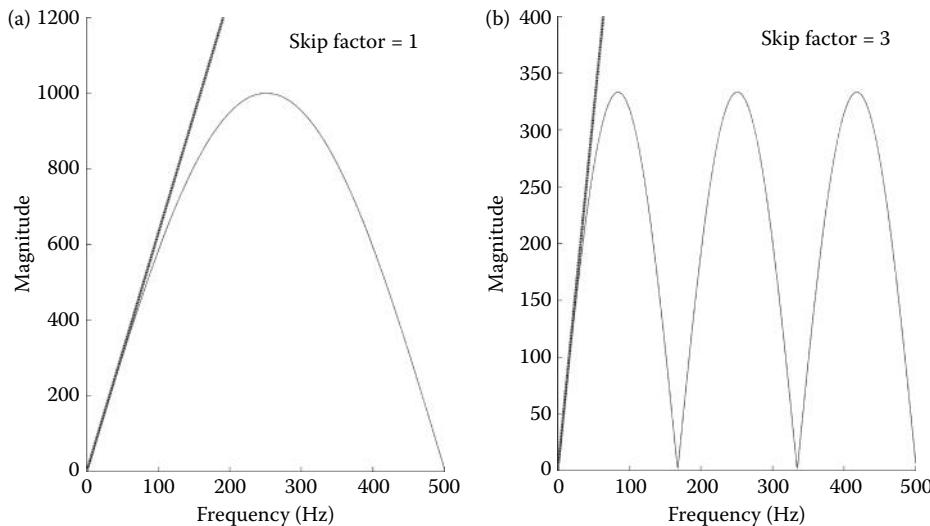


Figure 4.17 The frequency response of the two-point central difference algorithm using two different skip factors: (a) $L = 1$; (b) $L = 3$. The darker line shows the frequency characteristic of a simple differencing operation. The sample frequency is 1.0 kHz.

This equation shows that the magnitude spectrum, $|X(f)|$, is a sine function that goes to zero at $f = 1/(2LT_s) = f_s/(2L)$. Figure 4.17 shows the frequency characteristics of the two-point central difference algorithm for two different skip factors: $L = 1$ and $L = 3$.

MATLAB Solution

Finding the spectrum using MATLAB is straightforward once the impulse response is constructed. An easy way to construct the impulse response is to use brackets and concatenate zeros between the two end values, essentially following Equation 4.31 directly. Again, the initial filter coefficient is positive, and the final coefficient negative, to account for the reversal produced by convolution.

```
% Example 4.8 Determine the frequency response of
% the two-point central difference algorithm used for differentiation.
%
Ts = .001; % Assume a Ts of 1 msec.(i.e., fs = 1 kHz)
N = 1000; % Number of data points (time = 1 sec)
Ln = [1 3]; % Define two different skip factors
for m = 1:2 % Repeat for each skip factor
    L = Ln(m); % Set skip factor
    b = [1 zeros(1,2*L+1) -1]/(2*L*Ts); % Filter impulse resp.
    H = abs(fft(b,N)); % Calculate magnitude spectrum
    subplot(1,2,m); % Plot the result
    .....plot and label spectrum; plot straight line for comparison.....
end
```

The result of this program and the analytical analysis is shown in Figure 4.17. A true derivative has a linear change with frequency: a line with a slope proportional to f as shown by the dashed lines in Figure 4.17. The two-point central difference spectrum approximates a true derivative over the lower frequencies, but has the characteristic of a lowpass filter for higher frequencies. Increasing the skip factor, L , has the effect of lowering the frequency range over which the filter acts like a derivative operator as well as lowering the lowpass filter range. Note that

Biosignal and Medical Image Processing

for skip factors >1 , the response curve repeats at $f = 1/(2LT_s)$. Usually the assumption is made that the signal does not contain frequencies in this range. If this is *not* true, then these higher frequencies can be removed by an additional lowpass filter as shown in one of the problems. It is also possible to combine the difference equation (Equation 4.31) with a lowpass filter; this is also explored in one of the problems.

4.4.2.1 Determining Cutoff Frequency and Skip Factor

Determining the appropriate cutoff frequency of a filter or the skip factor for the two-point central difference algorithm can be somewhat of an art (meaning there is no definitive approach to a solution). If the frequency ranges of the signal and noise are known, setting cutoff frequencies is straightforward. In fact, there is an approach called the *Wiener filtering* that leads to an optimal filter if these frequencies are accurately known (see Chapter 8). However, this knowledge is usually not available in biomedical engineering applications. In most cases, filter parameters such as filter order (i.e., L_1) and cutoff frequencies are set empirically based on the data. In one scenario, the signal bandwidth is progressively reduced until some desirable feature of the signal is lost or compromised. While it is not possible to establish definitive rules due to the task-dependent nature of filtering, the next example gives an idea about how these decisions are approached.

In the next example, we return to the eye-movement signal. We evaluate several different skip factors to find the one that gives the best reduction of noise without reducing the accuracy of the velocity trace.

EXAMPLE 4.9

Use the two-point central difference algorithm to compute velocity traces of the eye-movement step response in file `eye.mat`. Use four different skip factors (1, 2, 5, and 10) to find the skip factor that best reduces noise without substantially reducing the peak velocity of the movement.

Solution

Load the file and use a loop to calculate and plot the velocity determined with the two-point central difference algorithm using the four different skip factors. Find the maximum value of the velocity trace for each derivative evaluation, and display it on the associated plot. The original eye movement (Figure 4.16) (upper traces) is in degrees, so the velocity trace is in degrees per second.

```
% Example 4.9 Evaluate the two-point central difference algorithm using
% different derivative skip factors
%
load eye;                                % Get data
fs = 200;                                 % Sampling frequency
Ts = 1/fs;                                % Calculate Ts
t = (1:length(eye_move))/fs;               % Time vector for plotting
L = [1 2 5 10];                           % Filter skip factors
for m = 1:4                                % Loop for different skip factors
    b = [1 zeros(1,2*L(m)-1) -1]/(2*L(m)*Ts); % Construct filter
    der = conv(eye_move,b,'same');             % Apply filter
    subplot(2,2,m);                          % Plot in different positions
    plot(t,der,'k');
    text(1,22,['L = ',num2str(L(m))],'FontSize',12); % Add text to plot
    text(1,18,['Peak = ',num2str(max(der),2)],'FontSize',12);
    .....labels and axis.....
end
```

The results of this program are shown in Figure 4.18. As the skip factor increases the noise decreases, and the velocity trace becomes quite smooth at a skip factor of 10. However, often we

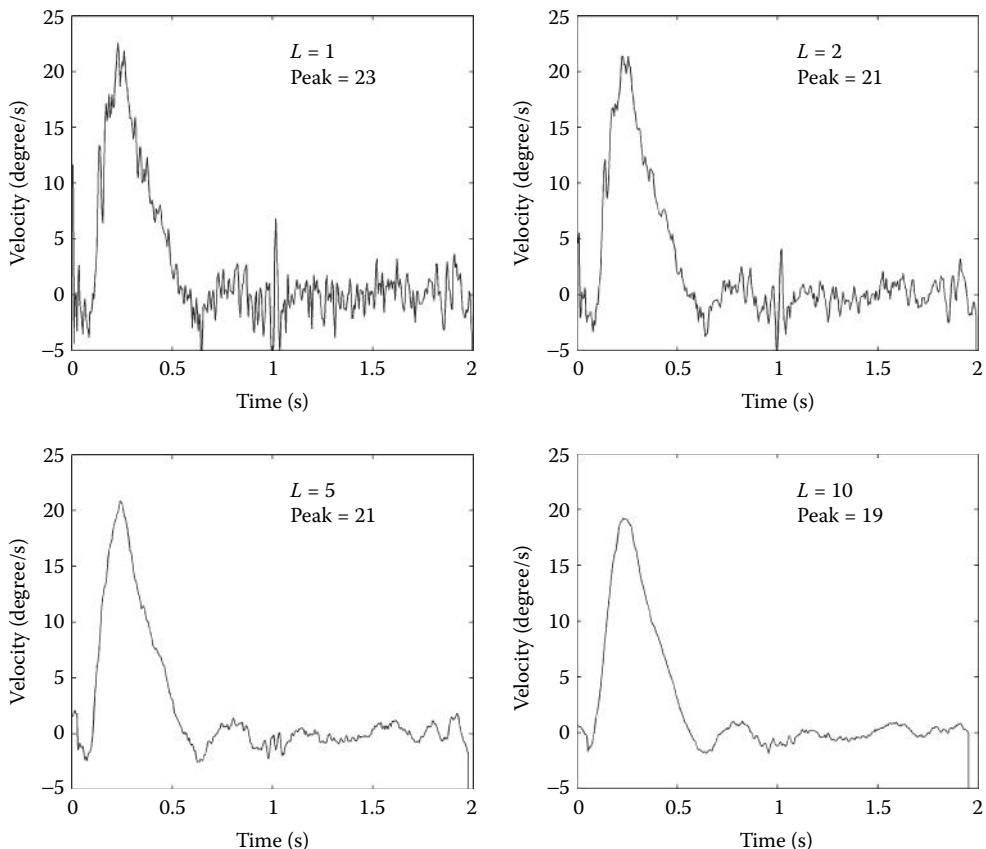


Figure 4.18 Velocity traces for the eye movement shown in Figure 4.16 (upper traces) calculated by the two-point central difference algorithm for different values of skip factor as shown. The peak velocities are shown in degrees per second.

are concerned with measuring the maximum velocity of a response, and the peak velocity also decreases with the higher skip factors. Examining the curve with the lowest skip factor ($L = 1$) suggests that the peak velocity shown, 23 deg/s, may be augmented a bit by noise. A peak velocity of 21 deg/s is found for both skip factors of two and five. This peak velocity appears to be reflective of the true peak velocity. The peak found using a skip factor of 10 is slightly reduced by the derivative filter. Based on this empirical study, a skip factor of around five seems appropriate, but this is always a judgment call. In fact, judgment and intuition are all too frequently involved in signal-processing and signal analysis tasks, a reflection that signal processing is still an art. Other examples that require empirical evaluation are given in the problem set.

4.4.3 FIR Filter Design Using MATLAB

The rectangular window equations facilitate the design of all the basic FIR filter types: lowpass, highpass, bandpass, and bandstop. In some rare cases, there may be a need for a filter with a more exotic spectrum; MATLAB offers considerable support for the design and evaluation of both FIR and IIR filters in the Signal Processing Toolbox.

Within the MATLAB environment, filter design and application occur in either one or two stages where each stage is executed by different, but related routines. In the two-stage protocol,

Biosignal and Medical Image Processing

the user supplies information regarding the filter type and desired attenuation characteristics, but *not the filter order*. The first-stage routines determine the appropriate order as well as other parameters required by the second-stage routines. The second-stage routines then generate the filter coefficients, $b[k]$, based on the arguments produced by the first-stage routines, including the filter order. It is possible to bypass the first-stage routines if you already know, or can guess, the filter order. Here, we cover only the single-stage approach, in which you specify the filter order, since trial and error is often required to determine the filter order anyway.

If a filter task is particularly demanding, the Signal Processing Toolbox provides an interactive filter design package called FDATool (for filter design and analysis tool) that uses an easy graphical user interface (GUI) to design filters with highly specific or demanding spectral characteristics. Another Signal Processing Toolbox package, the SPTool (for Signal Processing Tool), is useful for analyzing filters and generating spectra of both signals and filters. MATLAB help contains detailed information on the use of these two packages.

Irrespective of the design process, the net result is a set of b coefficients, the filter's impulse response. The FIR filter represented by these coefficients is implemented as above, using either MATLAB's `conv` or `filter` routine. Alternatively, the Signal Processing Toolbox contains the routine `filtfilt` that, like `conv` with the 'same' option, eliminates the time shift associated with standard filtering. A comparison between `filter` and `filtfilt` is given in one of the problems.

One useful Signal Processing Toolbox routine determines the frequency response of a filter, given the coefficients. We already know how to do this using the Fourier transform, but the MATLAB routine `freqz` also includes frequency scaling and plotting so while it is not essential, it is convenient.

```
[H, f] = freqz (b, a, n, fs);
```

where b and a are the filter coefficients, and n is optional and specifies the number of points in the desired frequency spectra. Only the b coefficients are associated with FIR filters, so a is set to 1.0. The input argument, fs , is also optional and specifies the sampling frequency. Both output arguments are also optional and are usually not given. If `freqz` is called without the output arguments, the magnitude and phase plots are produced. If the output arguments are specified, the output vector H is the complex frequency response of the filter (the same variable produced by `fft`) and f is a frequency vector useful in plotting. If fs is given, f is in Hz and ranges between 0 and $f_s/2$; otherwise, f is in rad/sample and ranges between 0 and π .

The MATLAB Signal Processing Toolbox has a filter design routine based on the rectangular window filters described above: `fir1`. The basic rectangular window filters provided by these routines will not be explained here, but its calling structure can easily be found in `help fir1`. Like `freqz`, these routines do things we already know how to do; they are just more convenient.

A filter design algorithm that is more versatile than the rectangular window filters already described is `fir2`, which can generate a filter spectrum having an arbitrary shape. The command structure for `fir2` is

```
b = fir2(order, f, G); % Design special FIR filter
```

where $order$ is the filter order (i.e., the number of coefficients in the impulse response), f is a vector of normalized frequencies in ascending order, and G is the desired gain of the filter at the corresponding frequency in vector f . In other words, `plot(f, G)` shows the desired magnitude frequency curve. Clearly, f and G must be the same length, but duplicate frequency points are allowed, corresponding to step changes in the frequency response. In addition, the first value of f must be 0.0 and the last value 1.0 (equal to $f_s/2$). As with all MATLAB filter design routines, frequencies are normalized to $f_s/2$ (not f): that is, an $f = 1$ is equivalent to the frequency

$f_s/2$. Some additional optional input arguments are mentioned in the MATLAB Help file. An example showing the flexibility of the `fir2` design routine is given next.

EXAMPLE 4.10

Design a double bandpass filter that has one passband between 50 and 100 Hz and a second passband between 200 and 250 Hz. Use a filter order of 65. Apply this to a signal containing sinusoids at 75 and 225 Hz in -20 dB of noise. (Use `sig_noise` to generate the signal). Plot the desired and actual filter magnitude spectra and the magnitude spectra of the signals before and after filtering.

Solution

We can construct a double bandpass filter by executing two bandpass filters in sequence, but we can also use `fir2` to construct a single FIR filter having the desired filter characteristics. First, we specify the desired frequency characteristics in terms of frequency and gain vectors. The frequency vector must begin with 0.0 and end with 1.0, but can have duplicate frequency entries to allow for step changes in gain. Then, we construct the coefficients (i.e., impulse response) using `fir2`, and apply it using `filter` or `conv`. In this example, we use `filter` for a little variety. Then the magnitude spectra of the filtered and unfiltered waveforms are determined using the `freqz` routine.

```
% Ex 4.10 Design a double bandpass filter.
%
fs = 1000; % Sample frequency
N = 2000; % Number of points
L = 65; % Filter order
f11 = 50/(fs/2); % Define cutoff freqs: first peak low
f1h = 100/(fs/2); % First peak high freq.
f12 = 200/(fs/2); % Second peak low cutoff
f2h = 250/(fs/2); % Second peak high cutoff
%
x = sig_noise([75 225], -20, N); % Generate noisy signal
%
% Design filter Construct frequency and gain vectors
f = [0,f11,f11,f1h,f1h,f12,f12,f2h,f2h,1]; % Frequency vector
G = [0, 0, 1, 1, 0, 0, 1, 1, 0, 0]; % Gain vector
subplot(2,1,1); hold on; % Set up to plot spectra
plot(f,G); % Plot the desired response
. .... labels. .....
b = fir2(L,f,G); % Construct filter
[H,f] = freqz(b,1,512,fs); % Calculate filter response
subplot(2,1,2);
plot(f,abs(H)); % Plot filter freq. response
. .... labels. .....
y = filter(b,1,x); % Apply filter
Xf = abs(fft(x)); % Compute magnitude spectra of filtered
Yf = abs(fft(y)); % and unfiltered data
. .... plot and label magnitude spectra of data. .....
```

Figure 4.19a is a plot of the desired magnitude spectrum obtained simply by plotting the gain vector against the frequency vector (`plot(f,A)`). Figure 4.19b shows the actual magnitude spectrum that is obtained by the `freqz` routine; this could have been found using the Fourier transform using Equation 4.13. The effect of applying the filter to the noisy data is shown by the magnitude spectra in Figure 4.20. The spectrum of the unfiltered data (Figure 4.20a) shows the

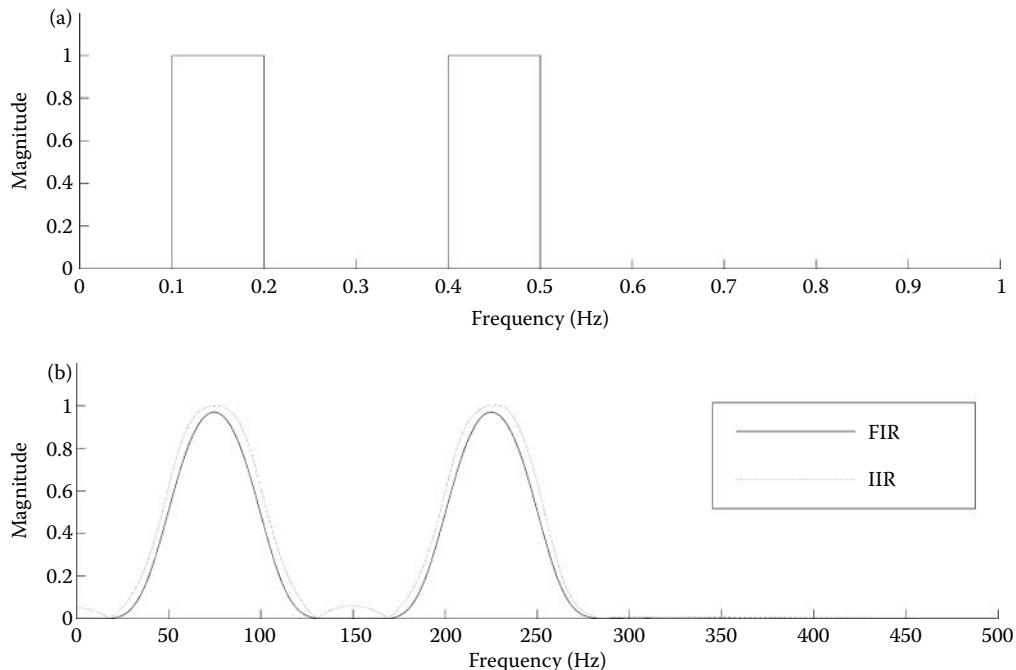


Figure 4.19 (a) Desired magnitude response for the double bandpass filter in Example 4.10. (b) Actual magnitude spectrum obtained from a 65th-order (i.e., 65 coefficients) FIR filter designed using MATLAB's `fir2` routine. A 12th-order IIR filter described in the next section and constructed in Example 4.11 is also shown for comparison.

two peaks at 75 and 225 Hz, but many other noise peaks of nearly equal amplitude are seen. In the filtered data spectrum (Figure 4.20b), the peaks are not any larger but are clearly more evident as the energy outside the two passbands has been removed. In addition, many analysis procedures would benefit from the overall reduction of noise.

4.5 Infinite Impulse Response Filters

To increase the attenuation slope of a filter, we can apply two or more filters in sequence. In the frequency domain, the frequency spectra multiply. This leads to a steeper attenuation slope in the spectrum. Figure 4.21 shows the magnitude spectrum of a 12th-order FIR rectangular window filter (dark line) and the spectra of two such filters in series (light gray line). However, for the same computational effort, you can use a single 24th-order filter and get an even steeper attenuation slope (Figure 4.21, medium gray line).

An intriguing alternative to using two filters in series is to filter the same data twice, first using a standard moving average FIR filter, then feeding back the output of the FIR filter to another filter that also contributes to the output. This becomes a moving average recursive filter with the two filters arranged as in Figure 4.22. The upper pathway is the standard FIR filter; the lower feedback pathway is another FIR filter with its own impulse response, $a[\ell]$. The problem with such an arrangement is apparent from Figure 4.22: the lower pathway receives its input from the filter's output but that input also requires the output: the output, $y[n]$, is a function of itself. To avoid the *algebraic loop* (a mathematical term for vicious circle) that such a configuration produces, the lower feedback pathway only operates on *past values* of $y[n]$; that is, values

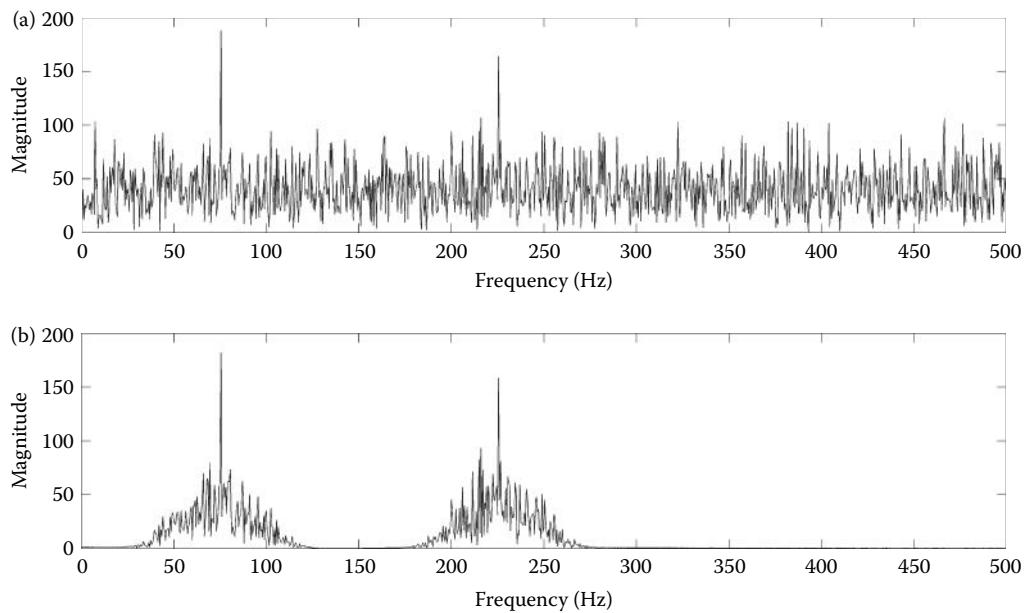


Figure 4.20 (a) Magnitude spectrum of two sinusoids buried in a large amount of noise (SNR = -20 dB). The signal peaks at 75 and 225 Hz are visible, but a number of noise peaks have the substantial amplitude. (b) The spectrum of the signal and noise after filtering with a double bandpass filter having the spectral characteristic seen in Figure 4.19b. The signal peaks are no larger than in the unfiltered data, but are much easier to see as the surrounding noise has been greatly attenuated. Other interesting filters with unusual spectra are explored in the problems.

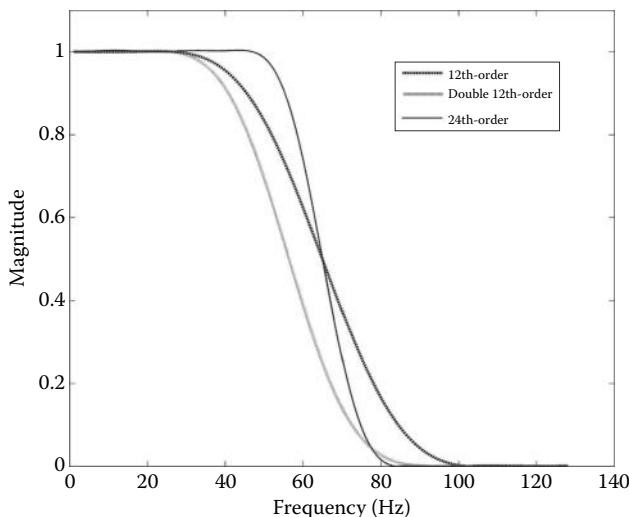


Figure 4.21 The magnitude spectrum of a 12th-order FIR filter (dark line) and the spectrum obtained by passing a signal between two such filters in series (light gray line). The two series filters produce a curve that has approximately twice the downward slope of the single filter. However, a single 24th-order filter produces an even steeper slope for the same computational effort (medium gray line).

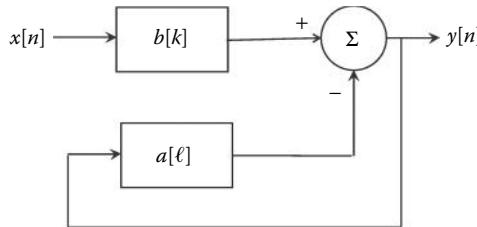


Figure 4.22 The configuration of an IIR filter containing two moving average-type FIR filters. The upper filter plays the same role as in a standard FIR filter. The lower filter operates on the output, but only past values of the output that have already been determined by the upper filter.

that have already been determined by the upper filter. (In chicken-and-egg terms, the output, or rather the *past output*, comes first.)

The equation for an IIR filter can be derived from the configuration given in Figure 4.22. The output is the convolution of the input with the upper FIR filter, minus the convolution of the lower FIR filter with the output, but only past values of the output. This tacks on an additional term to the basic FIR convolution equation given in Equation 4.14 to account for the second filter leading to Equation 4.12, and repeated here:

$$y[n] = \sum_{k=0}^K \underbrace{b[k]x[n-k]}_{\text{Upper path}} - \sum_{\ell=1}^L \underbrace{a[\ell]y[n-\ell]}_{\text{Lower path}} \quad (4.32)$$

where $b[k]$ are the upper filter coefficients, $a[\ell]$ are the lower filter coefficients, $x[n]$ is the input, and $y[n]$ is the output. Note that while the b coefficients are summed over delays beginning at $k = 0$, the a coefficients are summed over delays beginning at $\ell = 1$. Hence, the a coefficients are only summed over past values of $y[n]$. Since part of an IIR filter operates on passed values of the output, these filters are recursive filters, and the a coefficients are the *recursive coefficients*. The primary advantages and disadvantages of FIR and IIR filters are summarized in Table 4.1.

4.5.1 IIR Filter Implementation

IIR filters are applied to signals using Equation 4.32, where x is the signal and a and b are the coefficients that define the filter. While it would not be difficult to write a routine to implement this equation, it is again unnecessary as the MATLAB filter routine works for IIR filters as well:

```
y = filter(b,a,x); % IIR filter applied to signal x
```

where b and a are the same filter coefficients used in Equation 4.32, x is the input signal, and y is the output.

4.1 FIR versus IIR Filters: Features and Applications		
Filter Type	Features	Applications
FIR	Easy to design Stable Applicable to 2-D data (i.e., images)	Fixed, 1-D filters Adaptive filters Image filtering
IIR	Require fewer coefficients for the same attenuation slope, but can become unstable Particularly good for low cutoff frequencies Mimic analog (circuit) filters	Fixed, 1-D filters, particularly at low cutoff frequencies Real-time applications where speed is important

As mentioned above, the time shift of either FIR or IIR filters can be eliminated by using a noncausal implementation. Eliminating the time shift also reduces the filter's phase shift, and noncausal techniques can be used to produce zero-phase filters. Since noncausal filters use both *future* as well as past data samples (future only with respect to a given sample in the computer), they require the data to exist already in computer memory. The Signal Processing Toolbox routine `filtfilt` employs noncausal methods to implement filters with no phase shift. The calling structure is exactly the same as `filter`:

```
y = filtfilt(b,a,x); % Noncausal IIR filter applied to signal x
```

Several problems dramatically illustrate the difference between the use of `filter` and `filtfilt`.

4.5.2 Designing IIR Filters with MATLAB

The design of IIR filters is not as straightforward as that for FIR filters. However, the MATLAB Signal Processing Toolbox provides a number of advanced routines to assist in this process. IIR filters are similar to analog filters and originally were designed using tools developed for analog filters. In an analog filter, there is a direct relationship between the number of independent energy storage elements in the system and the filter's rolloff slope: each energy storage element adds 20 dB/decade to the slope. In IIR digital filters, the first *a* coefficient, $a[0]$, always equals 1.0, but each additional *a* coefficient adds 20 dB/decade to the slope. So an eighth-order analog filter and an eighth-order IIR filter have the same slope. Since the downward slope of an IIR filter increases by 20 dB/decade for each increase in filter order, determining the filter order needed for a given attenuation is straightforward.

IIR filter design with MATLAB follows the same procedures as FIR filter design; only the names of the routines are different. In the MATLAB Signal Processing Toolbox, the two-stage design process is supported for most of the IIR filter types. As with FIR design, a single-stage design process can be used if the filter order is known, and that is the approach that is covered here.

The Yule–Walker recursive filter is the IIR equivalent of the `fir2` FIR filter routine in that it allows for the specification of a general desired frequency response curve. The calling structure is also very similar to that of `fir2`.

```
[b,a] = yulewalk(order,f,G); % Find coeff. of an IIR filter
```

where `order` is the filter order, and `f` and `G` specify the desired frequency characteristic in the same manner as `fir2`: `G` is a vector of the desired filter gains at the frequencies specified in `f`. As in all MATLAB filter design routines, the frequencies in `f` are relative to $f_s/2$, and the first point in `f` must be 0 and the last point 1. Again, duplicate frequency points are allowed, corresponding to steps in the frequency response.

EXAMPLE 4.11

Design the double bandpass filter that is used in Example 4.10. The first passband has a range of 50–100 Hz and the second passband ranges between 200 and 250 Hz. Use an IIR filter order of 12 and compare the results with the 65th-order FIR filter used in Example 4.10. Plot the frequency spectra of both filters, superimposed for easy comparison.

Solution

Modify Example 4.10 to add the Yule–Walker filter, determine its spectrum using `freqz`, and plot it superimposed with the FIR filter spectrum. Remove the code relating to the signal as it is not needed in this example.

Biosignal and Medical Image Processing

```
% Example 4.11 Design a double bandpass IIR filter and compare with
% a similar FIR filter
%
% ..... same initial code as in Example 4.10.....
%
% Design filter
f = [0 50 50 100 100 200 200 250 250 fs/2]/(fs/2); % Construct desired
G = [0 0 1 1 0 0 1 1 0 0]; % frequency curve
b1 = fir2(L1,f,G); % Construct FIR filter
[H1,f1] = freqz(b1,1,512,fs); % Calculate FIR filter spectrum
[b2 a2] = yulewalk(L2,f,G); % Construct IIR filter
[H2 f2] = freqz(b2,a2,512,fs); % Calculate IIR filter spectrum
%
plot(f1,abs(H1),'k'); hold on; % Plot FIR filter mag. spectrum
plot(f2,abs(H2),':k','LineWidth',2); % Plot IIR filter magnitude spectrum
%.....labels.....
```

The spectral results from two filters are shown in Figure 4.19b. The magnitude spectra of both filters look quite similar despite that fact that the IIR filter has far fewer coefficients. The FIR filter has 65 b coefficients and 1 a coefficient while the IIR filter has 13 b coefficients and 13 a coefficients.

In an extension of Example 4.11, the FIR and IIR filters are applied to a random signal of 10,000 data points, and MATLAB's `tic` and `toc` are used to evaluate the time required for each filter operation. Surprisingly, despite the larger number of coefficients, the FIR filter is faster than the IIR filter: 0.077 ms for the FIR versus 1.12 ms for the IIR filter. However, if `conv` is used to implement the FIR filter, the FIR filter takes longer: 2.64 ms.

Several well-known analog filter types can be duplicated as IIR filters. Specifically, analog filters termed *Butterworth*, *Chebyshev* type I and II, and *Elliptic* (or Cauer) designs can be implemented as IIR digital filters and are supported in the MATLAB Signal Processing Toolbox. Butterworth filters provide a frequency response that is maximally flat in the passband and monotonic overall. To achieve this characteristic, Butterworth filters sacrifice rolloff steepness; hence, as shown in Figure 1.12, the Butterworth filter has a less sharp initial attenuation characteristic than the Chebyshev filter. The Chebyshev type I filter shown in Figure 1.12 features a faster rolloff than the Butterworth filter, but has ripple in the passband. The Chebyshev type II filter has ripple only in the stopband, its passband is monotonic, but it does not rolloff as sharply as type I. The ripple produced by Chebyshev filters is termed equiripple since it is of constant amplitude across all frequencies. Finally, elliptic filters have steeper rolloff than any of the above, but have equiripple in both the passband and stopband. While the sharper initial rolloff is a desirable feature, as it provides a more definitive boundary between passband and stopband, most biomedical engineering applications require a smooth passband, making Butterworth the filter of choice.

The filter coefficients for a Butterworth IIR filter can be determined using the MATLAB routine

```
[b,a] = butter(order,wn,'ftype') ; % Design Butterworth filter
```

where `order` and `wn` are the order and cutoff frequencies, respectively. (Of course, `wn` is relative to $f/2$.) If the '`ftype`' argument is missing, a lowpass filter is designed provided `wn` is scalar; if `wn` is a two-element vector, then a bandpass filter is designed. In the latter case, `wn = [w1 w2]`, where `w1` is the low cutoff frequency and `w2` is the high cutoff frequency. If a highpass filter is desired, then `wn` should be a scalar and '`ftype`' should be '`'high'`'. For a stopband filter, `wn` should be a two-element vector indicating the frequency ranges of the stop band and '`ftype`' should be '`'stop'`'. The outputs of `butter` are the `b` and `a` coefficients.

While the Butterworth filter is the only IIR filer you are likely to use, the other filters are easily designed using the associated MATLAB routine. The Chebyshev type I and II filters are designed with similar routines except an additional parameter is needed to specify the allowable ripple:

```
[b,a] = cheby1(order,rp,wn,'ftype'); % Design Chebyshev Type I
```

where the arguments are the same as in butter except for the additional argument, rp, which specifies the maximum desired passband ripple in dB. The type II Chebyshev filter is designed using

```
[b,a] = cheby2(order,rs,wn,'ftype'); % Design Chebyshev Type II
```

where again the arguments are the same, except rs specifies the stopband ripple again in dB but with respect to the passband gain. In other words, a value of 40 dB means that the ripple does not exceed 40 dB *below* the passband gain. In effect, this value specifies the minimum attenuation in the stopband. The elliptic filter includes both stopband and passband ripple values:

```
[b,a] = ellip(order,rp,rs,wn,'ftype'); % Design Elliptic filter
```

where the arguments presented are in the same manner as described above, with rp specifying the passband gain in dB and rs specifying the stopband ripple relative to the passband gain.

The example below uses these routines to compare the frequency response of the four IIR filters discussed above.

EXAMPLE 4.12

Plot the frequency response curves (in dB versus log frequency) obtained from an eighth-order lowpass filter using the Butterworth, Chebyshev type I and II, and elliptic filters. Use a cutoff frequency of 200 Hz and assume a sampling frequency of 2 kHz. For all filters, the ripple or maximum attenuation should be less than 3 dB in the passband, and the stopband attenuation should be at least 60 dB.

Solution

Use the design routines above to determine the *a* and *b* coefficients, use freqz to calculate the complex frequency spectrum, take the absolute value of this spectrum and convert to dB $20\log_{10}(\text{abs}(H))$. Plot using semilogx to put the horizontal axis in term of log frequency.* Repeat this procedure for the four filters.

```
% Example 4.12 Frequency response of four IIR 8th-order lowpass filters
%
N = 256; % Padding
fs = 2000; % Sampling frequency
L = 8; % Filter order
fc = 200/(fs/2); % Filter cutoff frequency
rp = 3; % Maximum passband ripple in dB
rs = 60; % Stopband ripple in dB
%
```

* Most spectral plots thus far have been in linear units. Plots of dB versus log frequency are often used for filter spectra, particularly IIR filter spectra, as the attenuation slopes are in multiples of 20 dB/decade. These slopes become straight lines in dB versus log f plots. See Figure 1.10 for a comparison of linear and dB versus log f (i.e., log-log) plots. We use both plot types in this book.

Biosignal and Medical Image Processing

```
% Determine filter coefficients
[b,a] = butter(L,fc);
[H,f] = freqz(b,a,N,fs);
H = 20*log10(abs(H)); % Convert to magnitude in dB
subplot(2,2,1);
semilogx(f,H,'k'); % Plot spectrum in dB vs. log freq.
.....labels and title.....
%
[b,a] = cheby1(L,rp,fc); % Chebyshev Type I filter coefficients
[H,f] = freqz(b,a,N,fs); % Calculate complex spectrum
H = 20*log10(abs(H)); % Convert to magnitude in dB
.....plot as above, labels and title.....
%
[b,a] = cheby2(L,rs,fc); % Chebyshev Type II filter coefficients
.....Use freqz, dB conversion and plot as above, labels and title.....
%
[b,a] = ellip(L,rp,rs,fc); % Elliptic filter coefficients
.....Use freqz, dB conversion and plot as above, labels and title.....
```

The spectra of the four filters are shown in Figure 4.23. As described above, the Butterworth is the only filter that has smooth frequency characteristics in both the passband and stopband; it is this feature that makes it popular in biomedical signal processing, both in its analog and digital incarnations. The Chebyshev type II filter also has a smooth passband and a slightly

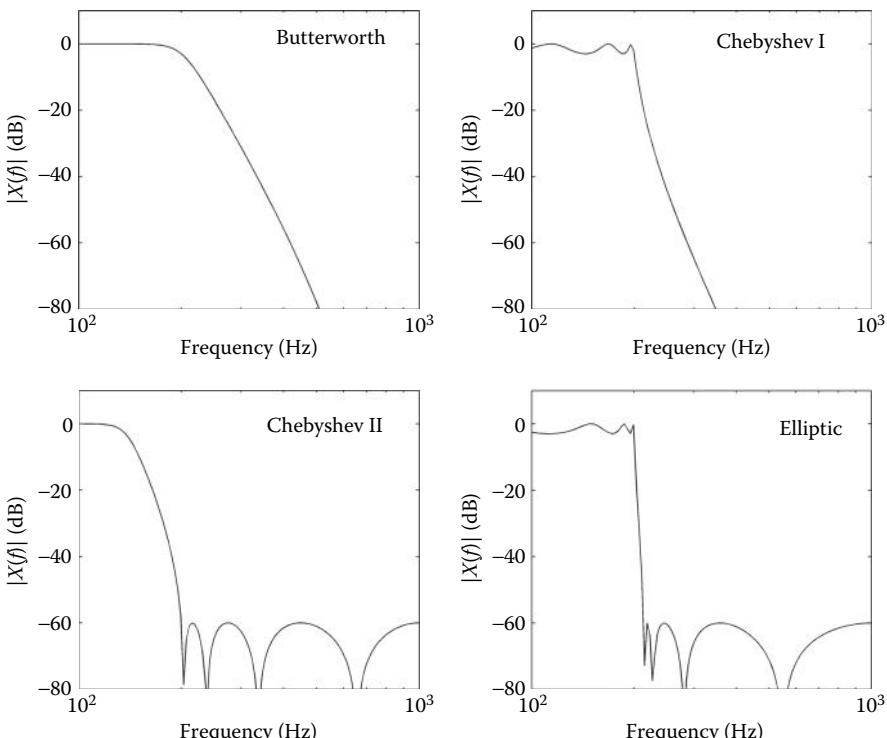


Figure 4.23 The spectral characteristics of four different eighth-order IIR lowpass filters with a cutoff frequency of 200 Hz. The assumed sampling frequency is 2 kHz. The spectral characteristics show that it is possible to increase the initial sharpness of an IIR filter significantly if various types of ripple in the spectral curve can be tolerated.

steeper initial slope than the Butterworth, but it does have ripple in the stopband, which can be problematic in some situations. The Chebyshev type I has an even sharper initial slope, but also has ripple in the passband, which limits its usefulness. The sharpest initial slope is provided by the elliptic filter, but ripple is found in both the passband and stopband. Reducing the ripple of these last three filters is possible in the design process, but this also reduces the filter's initial sharpness, another illustration of the compromises that continually arise in engineering.

Other types of filters exist in both FIR and IIR forms; however, the filters described above are the most common and are the most useful in biomedical engineering.

4.6 Summary

Most linear approaches to noise reduction are based on averaging. In ensemble averaging, entire signals are averaged together. This approach is quite effective, but requires multiple versions, or observations, of the same signal and a timing reference for the signals. If repetitive stimuli produce similar responses, time-locked to the stimulus, they are ideal for ensemble averaging. The most noted use of averaging in biosignal analysis is to separate out neuroelectric-evoked responses from the background noise in the EEG signal.

A special digital transfer function based on the Z-transform can be used to analyze digital processes such as IIR filters. The spectral characteristics of Z-domain transfer functions can easily be determined using the Fourier transform. While it is not easy to convert between the Z-domain transfer function and the Laplace transfer function, it is possible to convert to the frequency domain by substituting $e^{j\omega}$ for z .

Filters are used to shape the spectrum of a signal, often to eliminate frequency ranges that include noise or to enhance frequencies of interest. Digital filters are also based on averaging applied as a moving average and/or as a recursive moving average. FIR filters are straightforward moving average processes, usually with unequal filter weights. The filter weights correspond to the impulse response of the filter, so they can be implemented by convolving the weights with the input signal. FIR filters have linear phase characteristics and they can be applied to 2-D data such as images. FIR filters can be designed based on the inverse Fourier transform, as in rectangular window filters, but MATLAB routines also exist that simplify design.

IIR filters combine a moving average process with a recursive moving average to achieve much sharper attenuation characteristics for the same number of filter coefficients. In IIR filters, the moving average impulse response is applied to the input signal through standard convolution, while a recursive moving average is applied to a delayed version of the output also using convolution. IIR filters can be designed to mimic the behavior of analog filters; some types can produce sharp initial cutoffs at the expense of some ripple in the bandpass region. An IIR filter known as the Butterworth filter produces the sharpest initial cutoff without bandpass ripple, and is the most commonly used in biomedical applications. The design of IIR filters is more complicated than that of FIR filters, and is best achieved using MATLAB support routines.

PROBLEMS

- 4.1 Apply ensemble averaging to the data `ensemble_x.mat`. This file contains a data matrix labeled `x`. The data matrix contains 100 responses of an exponential signal buried in noise. In this matrix, each row is a separate response, so no transposition is necessary. Plot two randomly selected samples of these responses. Is it possible to analyze the buried signal in any single record? Construct and plot the ensemble average for this data. Scale the time axis correctly assuming $T_s = 0.05$.
- 4.2 Expand Example 4.1 to evaluate ensemble averaging for different numbers of individual responses. Load file `ver_problem.mat`, which contains the VER data, `ver`,

Biosignal and Medical Image Processing

along with the actual, noise-free evoked response in `actual_ver`. The visual response data set consists of 1000 responses, each 500 points long. The sample interval is 10 ms. Construct an ensemble average of 25, 100, and 1000 responses. The variables are in the correct orientation and do not have to be transposed. Subtract the noise-free variable (`actual_ver`) from an individual-evoked response and the three ensemble averages to get an estimate of the noise in the three waveforms. Compare the standard deviations of one of the unaveraged with the three averaged waveforms. Output the theoretical standard deviation based on Equation 4.3 using the unaveraged standard deviation and the appropriate number of averages. Compare the theoretical and actual noise values. Also plot one of the individual responses and, separately, the signal produced by averaging 1000 responses. Superimpose the noise-free response on the averaged signal and compare.

- 4.3 This program illustrates one of the problems that can occur with ensemble averaging: the lack of a fixed and stable reference signal. The file `ver_problem2.mat` contains three variables: `actual_ver`, which is the noise-free VER, `ver`, which consists of 100 noise records recorded with a fixed reference signal, and `ver1`, which consists of 100 records recorded with a reference signal that varies randomly by ± 150 ms. Construct ensemble averages from `ver` and `ver1` and plot separately along with the noise-free record. $T_s = 0.005$ s. What is the difference in the two averages?
- 4.4 Find the spectrum (magnitude and phase) of the system represented by the Z-transform:

$$H(z) = \frac{0.06 - 0.24z^{-1} + 0.37z^{-2} - 0.24z^{-3} + 0.06z^{-4}}{1 - 1.18z^{-1} + 1.61z^{-2} - 0.93z^{-3} + 0.78z^{-4}}$$

Use Equation 4.13 to find the spectrum (not `freqz`). Plot magnitude and phase (in deg) versus frequency (in Hz) assuming $f_s = 500$ Hz. Also find the step response of this system over a time period of 0.5 s.

- 4.5 Write the Z-transform equation for a fourth-order Butterworth highpass filter with a relative cutoff frequency of 0.3. [Hint: Get the coefficients from MATLAB's `butter` routine.]
- 4.6 Find the spectrum (magnitude and phase) of the system represented by the Z-transform:

$$H(z) = \frac{0.42x10^{-3} + 1.25x10^{-3}z^{-1} + 1.25x10^{-3}z^{-2} - 0.42x10^{-3}z^{-3}}{1 - 2.686z^{-1} + 2.42z^{-2} - 0.73z^{-3}}$$

Plot the magnitude in dB versus $\log f$ and the phase (in deg) versus $\log f$ assuming $f_s = 2000$ Hz. Limit the magnitude plot to be between 0 and -80 dB. Also find the step response of this system over a time period of 0.1 s. Use Equation 4.13 to find the spectrum (not `freqz`). Assuming $H(z)$ is a filter, what can you say about the filter from the magnitude plot?

- 4.7 Use MATLAB to find the frequency response of a 3-point moving average filter (i.e., $b = [1 1 1]/3$) and a 10-point moving average filter. Use appropriate zero-padding to improve the spectra. Plot both the magnitude and phase spectra assuming a sample frequency of 500 Hz.
- 4.8 Use `sig_noise` to generate a 20-Hz sine wave in 5 dB of noise (i.e., SNR = -5 dB) and apply the two filters from Problem 4.7 using the MATLAB `filter` routine. Plot

the time characteristics of the two outputs. Use a data length (N) of 200 in `sig_noise` and remember that `sig_noise` assumes a sample frequency of 1 kHz.

- 4.9 Find the magnitude spectrum of an FIR filter with a weighting function of $b = [.2 .2 .2 .2]$ in two ways: (a) apply the `fft` with padding to the filter coefficients as in Problem 4.7 and plot the magnitude spectrum of the result; (b) pass white noise through the filter using `conv` and plot the magnitude spectra of the output. Since white noise has, theoretically, a flat spectrum, the spectrum of the filter's output to white noise should be the spectrum of the filter. In the second method, use a 20,000-point noise array; that is, $y = \text{conv}(b, \text{randn}(20000, 1))$. Use the Welch averaging method described in Section 4.3 to smooth the spectrum. For the Welch method, use a segment size of 128 points and a 50% segment overlap. Since the `pwelch` routine produces the power spectrum, you need to take the square root to get the magnitude spectrum for comparison with the FT method. The two methods use different scaling, so the vertical axes are slightly different. Assume a sampling frequency of 200 Hz for plotting the spectra.
- 4.10 Use `sig_noise` to construct a 512-point array consisting of two closely spaced sinusoids of 200 and 230 Hz with SNR of -8 dB and -12 dB, respectively. Plot the magnitude spectrum using the FFT. Generate a 25 coefficient rectangular window bandpass filter using approach in Example 8.5. Set the low cutoff frequency to 180 Hz and the high cutoff frequency to 250 Hz. Apply a Blackman–Harris window to the filter coefficients, then filter the data using MATLAB's filter routine. Plot the magnitude spectra before and after filtering.
- 4.11 Use `sig_noise` to construct a 512-point array consisting of a single sinusoid at 200 Hz in -20 dB noise (a very large amount of noise). Narrowband filter the data around 200 Hz with a rectangular window filter. Use high and low cutoff frequencies at 180 and 220 Hz and use 25 coefficients in the filter. Plot the power spectrum obtained using the FFT direct method before and after filtering and using the Welch method. For the Welch method, use a segment length of 128 samples with 50% overlap. Note the improvement due to filtering in the ability to identify the narrowband component, particularly with the Welch method.
- 4.12 This problem compares the Blackman–Harris and Hamming windows in the frequency domain. Write a program to construct the coefficients of a lowpass rectangular window filter with a cutoff frequency of 200 Hz. Make the filter length $L = 33$ coefficients. Assume $f_s = 1000$ Hz. Generate and apply the appropriate length Hamming (Equation 4.24) and Blackman–Harris (Equation 4.25) windows to the filter coefficients, $b[k]$. Construct the two windows from the basic equations as in Example 4.5. Find and plot the spectra of the filter without a window and with the two windows. Plot the spectra superimposed to aid comparison and pad to 256 samples. As always, do not plot redundant points. Note that the Hamming window produces a somewhat steeper attenuation, but has just the slightest ripple before the cutoff.
- 4.13 This problem compares the Blackman–Harris and Hamming windows in the time domain. Construct the rectangular window filter used in Problem 4.12, but make $L = 129$ coefficients and the cutoff frequency 100 Hz. As in Problem 4.12, generate and apply the appropriate length Hamming (Equation 4.24) and Blackman–Harris (Equation 4.25) windows to the filter coefficients, $b[k]$. Determine the impulse response of these two filters using the MATLAB filter routine with an impulse input. The impulse input should consist of a 1 followed by 255 zeros. Plot the impulse responses superimposed (in different colors) and you will find they look nearly identical. Limit the time axis to 0.1 s to better observe the responses. Subtract the impulse

Biosignal and Medical Image Processing

response of the Blackman–Harris-windowed filter from the impulse response of the Hamming-windowed filter and plot the difference. Note that, while there is a difference, it is several orders of magnitude less than the impulse responses.

- 4.14 Load file ECG _ 9.mat, which contains 9 s of ECG data in variable x. These ECG data have been sampled at 250 Hz. The data have a low-frequency signal superimposed over the ECG signal, possibly due to respiration artifact. Filter the data with a highpass filter having 65 coefficients and a cutoff frequency of 8 Hz. Use the window of your choice to taper the filter weights. Plot the magnitude spectrum of the filter to confirm the correct type and cutoff frequency. Also plot the filtered and unfiltered ECG data. [Hint: You can modify a section of the code in Example 4.4 to generate the highpass filter.]
- 4.15 ECG data are often used to determine the heart rate by measuring the time interval between the peaks that occur in each cycle known as the *R wave*. To determine the position of this peak accurately, ECG data are first prefiltered with a bandpass filter that enhances the *R* wave and the two small negative peaks on either side, the *Q* and *S* wave. The three waves together are termed the *QRS complex*. Load file ECG_noise.mat, which contains 10 s of noisy ECG data in variable ecg. Filter the data with a 65-coefficient FIR bandpass filter to best enhance the *R*-wave peaks. Use the tapering window of your choice. (Note that it is not possible to eliminate all the noise, just improve the identification of the peaks.) Determine the low and high cutoff frequencies empirically, but they will be in the range of 4–24 Hz. The sampling frequency is 250 Hz.
- 4.16 This problem explores the use of multiple filters in sequence. Load the file deriv1_data.mat containing the signal variable x. Take the derivative of this signal using the two-point central difference algorithm with a skip factor of 6, implemented using the filter routine. Now add an additional lowpass filter using a rectangular window filter with 65 coefficients and a cutoff frequency 25 Hz. Use the conv routine with option 'same' to eliminate the added delay that is induced by the filter routine. Plot the original time data, the result of the two-point central difference algorithm, and the lowpass filtered derivative data. Note the clear derivative trace in the lowpass filtered data. Also plot the spectrum of the two-point central difference algorithm, the lowpass filter, and the combined spectrum. The combined spectrum can be obtained by simply multiplying the two-point central difference spectrum point-by-point with the lowpass filter spectrum.
- 4.17 Load the file deriv1_data.mat, which contains signal variable x ($f_s = 250$ Hz). Use these data to compare the two-point central difference algorithm with the combination of a difference (Equation 4.29) and a lowpass filter. Use a skip factor of 8 for the two-point central difference algorithm and with the difference operator use a 67th-order rectangular window lowpass filter with a cutoff frequency of 20 Hz. Use MATLAB's diff to produce the difference output, and scale the result by dividing by T_s . Filter this output with the lowpass filter. Note that the derivative obtained this way is smooth, but has low-frequency noise. Again, this problem shows the application of two sequential operations.
- 4.18 Use sig_noise to construct a 512-point array consisting of two widely separated sinusoids: 150 and 350 Hz, both with SNR of –14 dB. Use MATLAB's fir2 to design a 65-coefficient FIR filter having a spectrum with a double bandpass. The bandwidth of the two bandpass regions should be 20 Hz centered about the two peaks. Plot the filter's magnitude spectrum superimposed on the desired spectrum. [Recall: Referring to the calling structure of fir2, plot G versus f after scaling the frequency vector, f, appropriately.] Also plot the signal's magnitude spectrum before and after filtering.

- 4.19 The file `ECG_60HZ_data.mat` contains an ECG signal in variable x that was sampled at $f_s = 250$ Hz and has been corrupted by 60-Hz noise. The 60-Hz noise is at a high frequency compared to the ECG signal, so it appears as a thick line superimposed on the signal. Construct a 127-coefficient FIR rectangular *bandstop* filter with a center frequency of 60 Hz and a bandwidth of 10 Hz (i.e., ± 5 Hz) and apply it to the noisy signal. Implement the filter using either `filter` or `conv` with the 'same' option, but note the time shift if you use the former. Plot the signal before and after filtering and also plot the filter's magnitude spectrum to ensure the filter spectrum is what you want. [Hint: You can easily modify a portion of the code in Example 4.7 to implement the bandstop filter.]
- 4.20 This problem compares causal and noncausal FIR filter implementation. Generate the filter coefficients of a 65th-order rectangular window filter with a cutoff frequency of 40 Hz. Apply a Blackman–Harris window to the filter. Then apply the filter to the noisy sawtooth wave, x , in file `sawth.mat`. This waveform was sampled at $f_s = 1000$ Hz. Implement the filter in two ways. Use the causal `filter` routine and noncausal `conv` with the 'same' option. (Without this option, `conv` is like `filter` except that it produces extra points.) Plot the two waveforms along with the original, superimposed for comparison. Note the obvious differences. Also note that while the filter removes much of the noise, it also reduces the sharpness of the transitions.
- 4.21 Given the advantage of a noncausal filter with regard to the time shift shown in Problem 4.20, why not use noncausal filters routinely? This problem shows the downsides of non-causal FIR filtering. Generate the filter coefficients of a 33rd-order rectangular window filter with a cutoff frequency of 100 Hz, assuming $f_s = 1$ kHz. Use a Blackman–Harris window on the truncated filter coefficients. Generate an impulse function consisting of a 1 followed by 255 zeros. Now apply the filter to the impulse function in two ways: causally using the MATLAB `filter` routine, and noncausally using the `conv` routine with the 'same' option. (The latter generates a noncausal filter since it performs symmetrical convolution.) Plot the two time responses separately, limiting the x axis to 0–0.05 s to better visualize the responses. Then take the Fourier transform of each output and plot the magnitude and phase. (For a change, you can plot the phase in radians.) Use the MATLAB `unwrap` routine on the phase data before plotting. Note the strange spectrum produced by the noncausal filter (i.e., `conv` with the 'same' option). This is because the noncausal filter has truncated the initial portion of the impulse response. To confirm this, rerun the program using an impulse that is delayed by 10 sample intervals (i.e., `impulse = [zeros(1,10) 1 zeros(1,245)];`). Note that the magnitude spectra of the two filters are now the same, although the phase curves are different due to the different delays produced by the two filters. The phase spectrum of the noncausal filter shows reduced phase shift with frequency as would be expected. This problem demonstrates that noncausal filters can create artifact with the initial portion of an input signal because of the way it compensates for the time shift of causal filters.
- 4.22 Compare the step response of an 8th-order Butterworth filter and a 44th-order (i.e., $L = 44 + 1$) rectangular window (i.e., FIR) filter, both having a cutoff frequency of 0.2 f_s . Assume a sampling frequency of 2000 Hz for plotting. (Recall that MATLAB routines normalize frequencies to $f_s/2$, while the rectangular window filters equations are normalized to f_s .) Use a Blackman–Harris window with the FIR filter, then implement both filters using MATLAB's `filter` routine. Use a step of 256 samples but offset the step by 20 samples for better visualization (i.e., the step change should occur at the 20th sample). Plot the time responses of both filters. Also plot the magnitude spectra of both filters. Use Equation 4.13 to find the spectrum of both filters. (For the FIR filter, the denominator of Equation 4.13 is 1.0.) Note that although the magnitude

Biosignal and Medical Image Processing

- spectra have about the same slope, the step response of the IIR filter has more overshoot—another possible downside to IIR filters.
- 4.23 Repeat Problem 4.19, but use an eighth-order Butterworth bandstop filter to remove the 60-Hz noise. Load the file `ECG_60HZ_data.mat` containing the noisy ECG signal in variable x ($f_s = 250$ Hz). Apply the filter to the noisy signal using either `filter` or `filtfilt`, but note the time shift if you use the former. Plot the signal before and after filtering. Also plot the filter spectrum to ensure the filter is correct. [Recall that with the Signal Processing Toolbox, cutoff frequencies are specified relative to $f_s/2$.] Note how effective the low-order IIR filter is at removing the 60-Hz noise.
- 4.24 This problem demonstrates a comparison of a causal and a noncausal IIR filter implementation. Load file `Resp_noise1.mat` containing a noisy respiration signal in variable `resp_noise1`. Assume a sample frequency of 125 Hz. Construct a 14th-order Butterworth filter with a cutoff frequency of $0.15 f_s/2$. Filter that signal using both `filter` and `filtfilt` and plot the original and both filtered signals. Plot the signals offset on the same graph to allow for easy comparison. Also plot the noise-free signal found as `resp` in file `Resp_noise1.mat` below the other signals. Note how the original signal compares with the two filtered signals, in terms of the restoration of features in the original signal and the time shift.
- 4.25 This problem is similar to Problem 4.21 in that it illustrates problems with non-causal filtering, except that an IIR filter is used and the routine `filtfilt` is used to implement the noncausal filter. Generate the filter coefficients of an eighth-order Butterworth filter with a cutoff frequency of 100 Hz assuming $f_s = 1$ kHz. Generate an impulse function consisting of a 1 followed by 255 zeros. Now apply the filter to the impulse function using both the MATLAB `filter` routine and the `filtfilt` routine. The latter generates a noncausal filter. Plot the two time responses separately, limiting the x axis to 0–0.05 s to better visualize the responses. Then take the Fourier transform of each output and plot the magnitude and phase. Use the MATLAB `unwrap` routine on the phase data before plotting. Note the differences in the magnitude spectra. The noncausal filter (i.e., `filtfilt`) has ripple in the passband. Again, this is because the noncausal filter has truncated the initial portion of the impulse response. To confirm this, rerun the program using an impulse that is delayed by 20 sample intervals (i.e., `impulse = [zeros(1,20) 1 zeros(1,235)];`). Note that the magnitude spectra of the two filters are now the same. The phase spectrum of the noncausal filter shows reduced phase shift with frequency, as would be expected. However, even after changing the delay, the noncausal implementation has a small amount of ripple in the passband of the magnitude spectrum.
- 4.26 Load the data file `ensemble_data.mat`. Filter the average with a 12th-order Butterworth filter. (The sampling frequency is not given, but you do not need it to work the problem.) Select a cutoff frequency that removes most of the noise, but does not unduly distort the response dynamics. Implement the Butterworth filter using `filter` and plot the data before and after filtering. Implement the same filter using `filtfilt` and plot the resultant filter data. Compare the two implementations of the Butterworth filter. For this signal, where the interesting part is not near the edges, the noncausal filter is appropriate. Use MATLAB's `title` or `text` command to display the cutoff frequency on the plot containing the filtered data.
- 4.27 This problem compares FIR–IIR filters. Construct a 12th-order Butterworth highpass filter with a cutoff frequency of 80 Hz assuming $f_s = 300$ Hz. Construct an FIR high-pass filter having the same cutoff frequency using Equation 4.26. Apply a Blackman–Harris window to the FIR filter. Plot the spectra of both filters and adjust the order of

the FIR filter to approximately match the slope of the IIR filter. Use Equation 4.13 to find the spectra of both filters. Count and display the number of b coefficients in the FIR filter and the number of a and b coefficients in the IIR filter.

- 4.28 Find the power spectrum of an LTI system four ways: (1) use white noise as the input and take the Welch power spectrum of the output; (2) use white noise as an input and take the Fourier transform of the autocorrelation function of the output; (3) use white noise as an input and take the Welch power spectrum of the cross-correlation of the output with the input; and (4) apply Equation 4.13 to the a and b coefficients. The third approach works even if the input is not white noise. As a sample LTI system, use a fourth-order Butterworth bandpass filter with cutoff frequencies of 150 and 300 Hz. For the first three methods, use a random input array of 20,000 points. Use `xcorr` to calculate the auto- and cross-correlation and `pwelch` to calculate the power spectrum. For `pwelch`, use a window of 128 points and a 50% overlap. [Owing to the number of samples, this program may take 30 s or more to run so you may want to debug it with fewer samples initially.] Note that while the shapes are similar, the vertical scales will be different.
- 4.29 Load the file `nb2_noise_data.mat`, which contains signal variable x . This signal has two sinusoids at 200 and 400 Hz in a large amount of noise. Narrowband filter the signal around 200 and 400 Hz using a double bandpass IIR Yule–Walker filter. Use a 12th-order filter with high and low cutoff frequencies at ± 30 Hz of the center frequencies. Plot the power spectrum obtained using the Welch method before and after filtering. Use a segment length of 256 samples with 50% overlap. Note the substantial improvement in the ability to identify the narrowband signals.
- 4.30 Find the step responses of the four IIR filters presented in Example 4.12. Use the same filter parameters used in that example and implement using `filter`. Use a step signal of 90 samples. For plotting the step responses, use $f_s = 2$ kHz as in Example 4.12. Note the unusual step responses produced by the Chebyshev type 1 and elliptic filters. Also note that even the Butterworth produces some overshoot in the step response.

5

Modern Spectral Analysis

The Search for Narrowband Signals

5.1 Parametric Methods

The techniques for determining the power spectra described in Chapter 3 are based on the Fourier transform and are referred to as classical methods. These methods are the most robust of the spectral estimators. They require no assumptions about the origin or nature of the data, although some knowledge of the data is useful for window selection and averaging strategies. Fourier transform methods provide the best spectral estimate of the waveform stored in the computer, but that waveform often consists of signal *and* noise, sometimes a lot of noise. The problem with the Fourier transform is its nonjudgmental approach to a waveform: it accurately estimates the spectrum of the signal and the noise while we usually want only the spectrum of the noise-free signal. Alternative spectral analysis methods exist that, under some circumstances, may provide a better spectral estimate of just the signal.

Modern approaches to spectral analysis are designed to overcome some of the distortions produced by the classical approach. These are particularly effective if the data segments are short. More importantly, they provide some control over the kind of spectrum they produce. Modern techniques fall into two broad classes: parametric or model-based,* and nonparametric. These techniques attempt to overcome the limitations of traditional methods by taking advantage of something that is known, or can be assumed, about the signal. For example, if you know that the process that generated your waveform can only produce sinusoids, then you would prefer a spectral analysis tool that emphasizes the presence of narrowband signals. Alternatively, if you know that the biological process under study generates signals similar to those of a second-order system, you would want a spectral analysis tool that is sensitive to such signals. If you know something about a signal that is model-based, or parametric, there are methods that can make assumptions about that signal outside the data window. This eliminates the need for windowing and can improve spectral resolution and fidelity, particularly if the waveform is short or contains a large amount of noise. Any improvement in spectral resolution and fidelity will depend strongly on the appropriateness of the model selected. Accordingly, modern approaches require more judgment in their application than classical, FFT-based methods. The

* In some semantic contexts, all spectral analysis approaches can be considered model-based. For example, classic Fourier transform spectral analysis could be viewed as using a model consisting of harmonically related sinusoids. Here, we use the term *parametric* to avoid possible confusion.

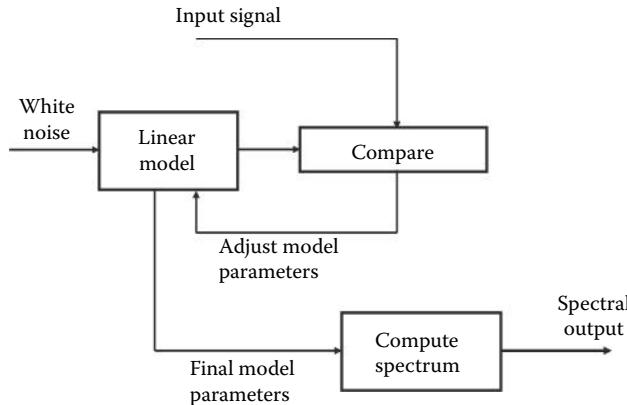


Figure 5.1 Schematic representation of model-based methods of spectral estimation.

increased flexibility of modern spectral methods is one of their major strengths, but this flexibility demands more thought by the user, and the particular model chosen must be justifiable. Finally, these methods provide only magnitude information, usually in the form of the power spectrum.

Parametric methods make use of an LTI model to estimate the power spectrum. This model is the same as the filters described in the last chapter and is totally defined by coefficients.* The basic strategy of this approach is shown in Figure 5.1.

The LTI or model as defined by Equation 4.12 in the last chapter is assumed to be driven by white noise. Recall that white noise contains equal energy at all frequencies; its power spectrum is a constant over all frequencies. The output of this model is compared with the input waveform, and the model parameters are adjusted for the best match between model output and the waveform of interest. When the best match is obtained, the defining model parameters (*as* and/or *bs*) are used to determine the waveform's spectrum using Equation 4.13 in the last chapter. The logic is that if the input to the model is spectrally flat, then the output spectrum is entirely determined by the model coefficients. So, if the waveforms match (more or less), the model spectrum must be the same as the signal spectrum. You can then determine the model spectrum from the *a* and *b* coefficients via Equation 4.13. This approach may seem roundabout, and it is, but it permits well-defined constraints to be placed on the resulting spectrum by using a specific model type and model order. The complexity of the resultant spectrum is constrained by the complexity of the model: basically the number of numerator and/or denominator coefficients.

5.1.1 Model Type and Model Order

Three model types are commonly used in this approach, distinguished by the nature of their transfer functions: *autoregressive* or *AR* models (having only *a* coefficients), *moving average* or *MA* models (having only *b* coefficients), and *autoregressive moving average* or *ARMA* models (having both *a* and *b* coefficients) (Figure 5.2). The selection of the most appropriate model requires some knowledge of the probable shape of the spectrum. The MA model is useful for evaluating spectra with valleys but no sharp peaks. The Z-transform transfer function of this model has only a numerator polynomial and so is sometimes referred to as an *all-zero* model.

* More semantics: a linear process (i.e., LTI) is referred to as a “model” in parametric spectral analysis and as a “filter” when it is used to shape a signal’s spectral characteristics. When used as a model, the coefficients that define the model (i.e., the *a* and *b* coefficients) are called “parameters.” When used as a filter, these same coefficients are called “coefficients” or “weights.” Despite the different terms, “linear models,” “filters,” “processes,” and “LTI’s” are all the same thing and are described by the basic equations (Equation 4.9 or 4.12) presented in Chapter 4.

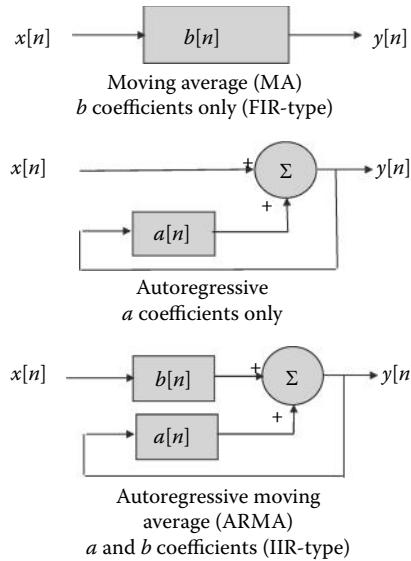


Figure 5.2 Three different linear models used in spectral estimation. The upper model uses only numerator coefficients (i.e., b s), as in the FIR filter, and is termed a moving average or MA model. The middle model uses only denominator coefficients (i.e., a s) like a modified IIR filter, and is termed an autoregressive or AR model. The lower model uses both coefficients, as in an IIR filter, and is termed an autoregressive moving average or ARMA model.

The MA model is the same as an FIR filter and has the same defining equation: Equation 4.14 repeated here with a modification in a variable name:

$$y[k] = \sum_{n=0}^{q-1} b[n]x[k-n] \quad (5.1)$$

where q is the number of b coefficients called the *model order*, $x[n]$ is the input, and $y[n]$ is the output.

The AR model has a Z-transform transfer function with only a constant in the numerator and a polynomial in the denominator; hence, this model is sometimes referred to as an *all-pole* model (Figure 5.2). The AR approach adjusts coefficients in the denominator of the transfer function. When denominator values are small, slight changes in their value can result in large peaks in the model's spectrum. In fact, if the denominator goes to zero, the peaks become infinite. Because the AR approach manipulates a denominator, the AR model is particularly useful for estimating spectra that have sharp peaks, in other words, the spectra of narrowband signals. This trick of manipulating the denominator to create spectra having sharp peaks is also used in the eigen decomposition methods of Section 5.2. Conversely, the AR approach is not very good at estimating the spectra of broadband signals, as is shown in Example 5.7.

The time-domain equation of an AR model is similar to the IIR filter equation (Equation 4.31) but with only a single numerator coefficient, $b[0]$, which is assumed to be 1:

$$y[k] = x[n] - \sum_{n=0}^{p-1} a[n]x[k-n] \quad (5.2)$$

where $x[n]$ is the input or noise function and p is the model order.* Note that in Equation 5.2, the output is the input after subtracting the convolution of the model coefficients with *past* versions

* Note that p and q are commonly used symbols for the order of AR and MA models, respectively.

of the output (i.e., $y[n - k]$). This linear process is not usually used as a filter, but it is just a variant of an IIR filter, one with a constant numerator.

If the spectrum is likely to contain both sharp peaks and valleys, then a model that combines both the AR and MA characteristics can be used. As might be expected, the transfer function of an ARMA model contains both numerator and denominator polynomials, so it is sometimes referred to as a *pole-zero* model. The ARMA model equation is the same as an IIR filter (Equation 4.32):

$$y[k] = \sum_{n=0}^{q-1} b(n)x(k-n) - \sum_{n=0}^{p-1} a(n)x(k-n) \quad (5.3)$$

where p is the number of denominator coefficients in the Z-transfer function and q is the number of numerator coefficients.

In addition to selecting the type of model to be used, we need to select the model order, p and/or q . Some knowledge of the process generating the data is most helpful. A few schemes have been developed to assist in selecting model order; these are described briefly below. The basic idea is to make the model order large enough to allow the model's spectrum to fit the signal spectrum, but not so large that it begins fitting the noise as well. In most practical situations, model order is derived on a trial-and-error basis.

5.1.2 Autoregressive Model

While many techniques exist for evaluating the parameters of an AR model, algorithms for MA and ARMA are less plentiful. In general, these algorithms involve significant computation and are not guaranteed to converge, or may converge to the wrong solution. Most ARMA methods estimate the AR and MA parameters separately, rather than jointly, and this does not lead to an optimal solution. The MA approach cannot model narrowband spectra well: it is not a high-resolution spectral estimator. This shortcoming limits its usefulness in power spectral estimation of biosignals. Because of the computational challenges of the ARMA model and the spectral limitations of the MA model, the rest of this description of model-based power spectral analysis will be restricted to AR spectral estimation. Moreover, it is the only parametric method implemented in the MATLAB Signal Processing Toolbox. The MATLAB Signal Identification Toolbox includes MA, ARMA, and even more complicated models, but such approaches are rarely needed in practice.

AR spectral estimation techniques can be divided into two categories: algorithms that process block data, and algorithms that process data sequentially. The former are appropriate when the entire waveform is available in memory; the latter are effective when incoming data must be evaluated rapidly for real-time considerations. Here we consider only block processing algorithms, as they find the largest application in biomedical engineering and are the only algorithms implemented in the MATLAB Signal Processing Toolbox.

As with the concept of power spectral density introduced in the last chapter, the AR spectral approach is usually defined with regard to estimation based on the autocorrelation sequence. Nevertheless, better results are obtained, particularly for short data segments, by algorithms that operate directly on the waveform without estimating the autocorrelation sequence.

There are a number of different approaches for estimating the AR model coefficients and related power spectra directly from the waveform. The four approaches that are supported by MATLAB are the Yule-Walker, the Burg, the covariance, and the modified covariance methods. All of these approaches to spectral estimation are implemented in the MATLAB Signal Processing Toolbox.

The most appropriate method depends somewhat on the expected (or desired) shape of the spectrum, since different methods theoretically enhance different spectral characteristics.

5.1 Parametric Methods

For example, the Yule–Walker method is thought to produce spectra with the least resolution among the four but provides the most smoothing, while the modified covariance method should produce the sharpest peaks, useful for identifying sinusoidal components in the data. The Burg and covariance methods are known to produce similar spectra. In reality, the MATLAB implementations of the four methods all produce similar spectra, as we show below and also in the problem set. The derivation of the Yule–Walker approach is also given below.

Figure 5.3 illustrates some of the advantages and disadvantages of using AR analysis as a spectral analysis tool. A test waveform is constructed consisting of a low-frequency broadband signal, four sinusoids at 100, 240, 280, and 400 Hz, and white noise. A classically derived spectrum (i.e., the Fourier transform) is shown without the added noise in Figure 5.3a and with the noise in Figure 5.3b. The remaining plots show the spectra obtained with an AR model of differing model orders. Figures 5.3c through 5.3e show the importance of model order on the resultant spectrum. The use of the Yule–Walker method with a relatively low-order model ($p = 17$) produces a smooth spectrum, particularly in the low-frequency range, but the spectrum combines the two closely spaced sinusoids (240 and 280 Hz), and does not show the 100-Hz

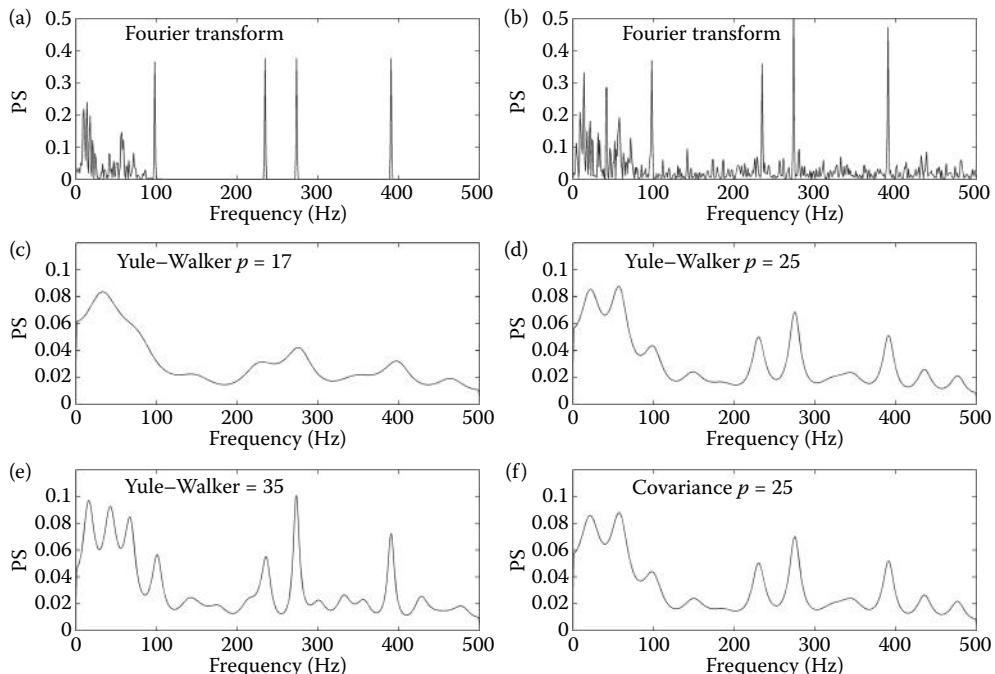


Figure 5.3 Comparison of AR and classical spectral analysis on a complicated spectrum. (a) Spectrum obtained using classical methods (Fourier transform) of a waveform consisting of four sinusoids (100, 240, 280, and 400 Hz) and a low-frequency region generated from lowpass filtered noise. (b) Spectrum obtained using the Fourier transform method applied to the same waveform after white noise has been added (SNR = -16 dB). (c–e) Spectra obtained using AR models (Yule–Walker) having three different model orders. The lowest-order model ($p = 17$) represents the broadband frequencies well, but does not show the 100-Hz sinusoid and cannot distinguish the two closely spaced sinusoids (240 and 280 Hz). The highest-order model ($p = 35$) better identifies the 100-Hz signal and the shows sharper peaks, but shows *spurious* peaks throughout the frequency range. (f) AR spectral analysis using the covariance method produces results nearly identical to the Yule–Walker method.

Biosignal and Medical Image Processing

component (Figure 5.3). The two higher-order models ($p = 25$ and 35) identify all of the sinusoidal components with the highest-order model showing sharper peaks and a better defined peak at 100 Hz (Figures 5.3d and 5.3e). However, the highest-order model ($p = 35$) also produces a less accurate estimate of the low-frequency spectral feature, showing a number of low-frequency peaks that are not present in the data. Such artifacts are termed *spurious peaks*; they occur most often when higher model orders are used. In Figure 5.3f, the spectrum produced by the covariance method is seen to be nearly identical to the one produced by the Yule–Walker method with the same model order.

The influence of model order is explored further in Figure 5.4. Four spectra are obtained from a waveform consisting of three sinusoids at 100, 200, and 300 Hz, buried in a fair amount of noise (SNR = -14 dB). Using the traditional transform method, the three sinusoidal peaks are well identified, but other, lesser, peaks are seen due to the noise (Figure 5.4a). A low-order AR model (Figure 5.4b) smooths the noise very effectively, but identifies only the two outermost peaks at 100 and 300 Hz. Using a higher-order model results in a spectrum where the three peaks are clearly identified, although the frequency resolution is moderate as the peaks are not very sharp. A still higher-order model improves the frequency resolution (the peaks are sharper), but now several spurious peaks are seen. In summary, the AR along with other model-based methods can be useful spectral estimators if the nature of the signal is known, but considerable care must be taken in selecting model order and model type. Several problems at the end of this chapter further explore the influence of model order.

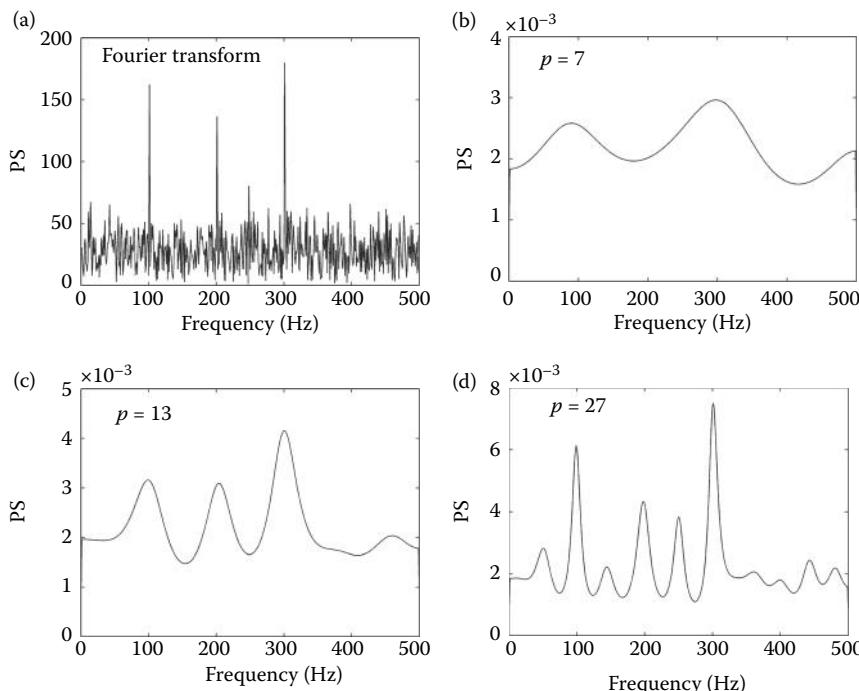


Figure 5.4 Spectra obtained from a waveform consisting of equal-amplitude sinusoids at 100, 200, and 300 Hz with white noise ($N = 1024$; SNR = -14 dB). (a) The traditional FFT method shows the three sinusoids, but also lesser peaks solely due to the noise. (b) The AR method with a low model order ($p = 7$) shows only the two outside peaks with a hint of the center peak. (c) A higher-order AR model ($p = 13$) shows the three peaks clearly. (d) An even higher-order model ($p = 27$) shows the three peaks with better frequency resolution, but also shows a number of spurious peaks.

5.1.3 Yule–Walker Equations for the AR Model

In all model-based approaches to spectral analysis, the problem is to derive the model coefficients that produce the best match between the model's white-noise output and the waveform. Since the spectrum is uniquely defined by the autocorrelation function, the a coefficients can also be found by matching the model and waveform autocorrelation functions. Once these coefficients are determined, the model's spectrum can be obtained from Equation 4.13. For an AR model with only denominator coefficients, this equation becomes

$$H[f] = \frac{K}{FFT(a[n])}, \quad \text{where } K = \sigma^2 \quad (5.4)$$

To scale the power spectrum correctly, the spectrum is scaled by the variance, σ^2 . To find the a coefficients, we begin with the basic time-domain equation (Equation 5.2) and rearrange.

$$\begin{aligned} y[k] &= x[n] - \sum_{n=1}^p a[n]y[k-n] \\ x[n] &= y[k] + \sum_{n=1}^p a[n]y[k-n] \\ \sum_{n=0}^p a[n]y[k-n] &= x[n] \end{aligned} \quad (5.5)$$

where $x[n]$ is now white noise and $a[0] = 1$. The next set of algebraic operations is used to put this equation into a format that uses the autocorrelation function. Assuming real-valued signals, multiply both sides of this equation by $y[k-m]$ and take the expectation operation.

$$E\left[\sum_{n=0}^p a[n]y[k-n]y[k-m]\right] = E[x[n]y[k-m]] \quad (5.6)$$

Simplifying the left side by interchanging the expectation and summation:

$$\sum_{n=0}^p a[n] E[y[k-n]y[k-m]] = E[x[n]y[k-m]] \quad (5.7)$$

Note that the term on the left side of the equation, $E[y[k-n]y[k-m]]$, is equal to the autocorrelation of y with multiple lags $[m-n]$:

$$\sum_{n=0}^N a[n]r_{yy}[m-n] = E[x[n]y[k-m]] \quad (5.8)$$

The right side of the equation, $E[x[n]y[k-m]]$, is equal to zero, since $x[n]$ is uncorrelated because it is white noise. The second term, $y[k-m]$, is just a delayed version of $x[n]$, so the expectation of the product of two uncorrelated variables is zero. So Equation 5.8 simplifies to

$$\sum_{n=0}^N a[n]r_{yy}[m-n] = 0 \quad m > 0 \quad (5.9)$$

This can be expanded to

$$r_{yy}[m] = -a[1] r_{yy}[m-1] - a[2] r_{yy}[m-2] - \cdots - a[p] r_{yy}[m-p] \quad m > 0 \quad (5.10)$$

Biosignal and Medical Image Processing

This is just a series of simultaneous equations for value of $m = 1, 2, \dots, p$. These equations can be written in matrix form:

$$\begin{bmatrix} r_{yy}[0] & r_{yy}[1] & \dots & r_{yy}[p-1] \\ r_{yy}[1] & r_{yy}[0] & \dots & r_{yy}[p-2] \\ \vdots & \vdots & \ddots & \vdots \\ r_{yy}[p-1] & r_{yy}[p-2] & \dots & r_{yy}[0] \end{bmatrix} \begin{bmatrix} -a[1] \\ -a[2] \\ \vdots \\ -a[p] \end{bmatrix} = \begin{bmatrix} r_{yy}[1] \\ r_{yy}[2] \\ \vdots \\ r_{yy}[p] \end{bmatrix} \quad (5.11)$$

Or in compact matrix form:

$$R(-a) = r \quad \text{or} \quad a = -R^{-1}r \quad (5.12)$$

Thus, the AR coefficients, $a[n]$, can be computed from R and r , which can be obtained using only the autocorrelation coefficients. Note that Equation 5.11 solves only for $a[1]$ through $a[p]$, as $a[0]$ was assumed to be 1.0. Other methods for computing $a[n]$ exist. Example 5.1 below illustrates the application of these equations.

Implementing the Yule–Walker equation (Equation 5.11) in MATLAB is straightforward, but there are a couple of MATLAB routines that facilitate this process. The left side of the equation is a symmetrical matrix of autocorrelation lags. This matrix appears frequently in signal-processing theory; it is called the autocorrelation matrix or just the *correlation matrix*. This matrix has a symmetrical structure termed a *Toeplitz* structure and is uniquely defined by the first row or column, which in this case is just the autocorrelation function. While it could be constructed directly from the autocorrelation function with a few lines of code, MATLAB provides some easier alternatives. The routine `toeplitz` takes a vector r and converts it to the symmetrical matrix:

```
R = toeplitz(r); % Construct a symmetrical matrix
```

where r is the input vector and R is the output matrix. To construct the autocorrelation matrix, first find the autocorrelation using `xcorr` or similar routine, then apply the `toeplitz` routine:

```
[ryy, lags] = xcorr(x, p); % Autocorrelation vector
Rxx = toeplitz(ryy(p:end)); % Autocorrelation matrix
```

where p specifies the order of the matrix and thus the order of the AR model. Equation 5.11 uses only the positive lags of the autocorrelation function. These begin at index $p + 1$ in the autocorrelation vector, ryy . This approach is used in Example 5.1; however, an easier alternative for constructing the autocorrelation matrix in a single step is to use the MATLAB routine `corrmtx`:

```
[X, Rxx] = corrmtx(x, maxlags);
```

where Rxx is the autocorrelation matrix and X is a matrix used to calculate the autocorrelation matrix (specifically, $Rxx = X' * X$). This routine has a number of options, but the defaults will work for calculating the autocorrelation matrix.

An even easier method for evaluating the AR spectrum is found in the MATLAB Signal Processing Toolbox, a routine that solves the Yule–Walker equation, removes the redundant points, and provides a frequency vector for plotting.

```
[PS, freq] = pyulear(x, p, nfft, Fs);
```

The format of `pyulear` is similar to `pwelch` described in Chapter 3. Only the first two input arguments, x and p , are required as the other two have default values. The input argument, x , is a vector containing the input waveform, and p is the order of the AR model. The input argument `nfft` specifies the length of the data segment to be analyzed, and if `nfft` is less than the length of x , averages will be taken as in `pwelch`. The default value for `nfft` is

256.* As in `pwelch`, `Fs` is the sampling frequency and, if specified, is used to appropriately fill the frequency vector, `freq`, useful in plotting. If `Fs` is not specified, the output vector `freq` varies in the range of 0 to π .

As in routine `pwelsh`, only the first output argument, `PS`, is required, and it contains the resultant power spectrum. Similarly, the redundant points are removed and the length of `PS` is either $(\text{nfft}/2)+1$ if `nfft` is even, or $(\text{nfft}+1)/2$ if `nfft` is odd. An exception is made if `x` is complex, in which case the length of `PS` is equal to `nfft`. Other optional arguments are available and are described in the MATLAB help file.

The other AR methods are implemented using similar MATLAB statements, differing only in the function name.

```
[Pxx, freq] = pburg(x,p,nfft,Fs);
[Pxx, freq] = cov(x,p,nfft,Fs);
[Pxx, freq] = pmcov(x,p,nfft,Fs);
```

The routine `pburg` uses the Burg method, `cov` the covariance method, and `pmcov` the modified covariance method.

EXAMPLE 5.1

Apply the AR spectral method to analyze a 1000-point waveform consisting of four sinusoids buried in 12 dB of noise (i.e., SNR = -12 dB). The sinusoidal frequencies should be 100, 240, 280, and 400 Hz. Use a model order of 17.

Solution

Use the Yule–Walker method to solve Equation 5.11 for the a coefficients and apply Equation 5.4 to get the power spectrum. Since both the autocorrelation *vector* and the autocorrelation *matrix* are needed, use `xcorr` followed by `toeplitz` to get these variables. Compare this with the spectrum produced by MATLAB's `pyulear` routine.

Example 5.1 AR Model using Yule-Walker Eq.

```
%  
N = 1024; % Size of arrays  
SNR = -12; % SNR  
fs = 1000; % Sample frequency  
p = 17; % AR Model order  
  
x = sig_noise([100 240 280 400], SNR, N); % Generate data  
  
[rxx, lags] = xcorr(x,p); % Autocorrel. with p lags  
rxx = rxx(p+1:end)'; % Positive lags only  
Rxx = toeplitz(rxx(1:end-1)); % Const. autocorr matrix  
a = - (Rxx\rxx(2:end)); % Solve for a's, Eq. 5.12  
a = [1; a]; % Add a[0]  
H = var(x) ./ fft(a,1024); % Get spectrum Eq. 5.4  
% Compare with pyulear and plot  
[PS,f] = pyulear(x,17,N,fs); % AR power spectrum  
.....plot and label the two spectra.....
```

* The term `nfft` is somewhat misleading since it is also used with classical MATLAB spectral routines based on the Fourier transform. Despite this potential confusion, we still use it to be consistent with MATLAB terminology.

Analysis

After the data are constructed with `sig_noise`, the a coefficients are determined using `xcorr` to produce the autocorrelation function of x , and `toeplitz` to construct the correlation matrix. In the equation solution, note the transposition of `rxx` and also that `rxx(2:end)` is used to conform to the Yule–Walker equation (Equation 5.11). This solves only for $a[1]$ through $a[p]$. Hence, $a[0]$ must be added in front of the rest of the a vector. The power spectrum is obtained by dividing the Fourier transform of the a vector into the variance. This is zero-padded out to 1024 points, many more than required for a respectable spectrum, but this is the same padding used by `pyulear` so the frequency vector generated `pyulear` could be shared.

Results

The two spectra produced by this example are shown in Figure 5.5.

The next example compares spectra produced by different AR model orders, which generated Figure 5.3. The remainder of this section provides examples of AR power spectral methods applied to various signals, and explores the influence of model order on the outcome.

EXAMPLE 5.2

Load the file `example5_2_data.mat`, which contains signal x consisting of lowpass filtered noise, four sinusoids (two of which are closely spaced), and white noise ($f_s = 1$ kHz). This example was used to generate the plots in Figure 5.3 shown previously.

Solution

After loading the data file, use `pwelch` with the window size equivalent to the signal length to compute the power spectrum. This is identical to using `fft` and taking the square, but is easier since it removes the redundant points and provides a frequency vector. Use MATLAB's `pyulear` and `pmcov` to compute the AR power spectra.

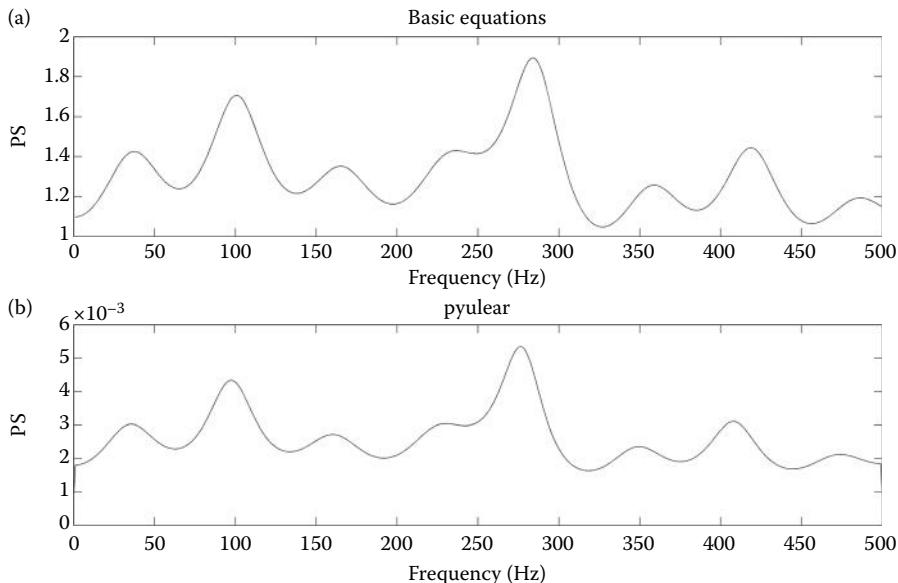


Figure 5.5 Power spectra of a waveform containing four sinusoids buried in noise (100, 240, 280, and 400 Hz). (a) Power spectrum generated using the basic Yule–Walker equations (Equations 5.4 and 5.11). (b) Power spectrum generated using MATLAB's `pyulear` routine.

```
% Example 5.2 and Figure 5.3
% Program to evaluate AR spectral methods and model order.
%
load example5_2_data.mat; % Load data
N = length(x); % Get signal length
fs = 1000; % Sample frequency
%
% FFT spectrum (Welch, but with window including entire signal)
[PS,f] = pwelch(x,N,[],[],fs);
subplot(3,2,1);
    ..... plot, labels, and axis .....
%
% Yule-Walker spectra, 11th order
[PS,f] = pyulear(x,11,N,fs);
    ..... plot, labels, and axis .....
%
% Yule-Walker spectra, 17th order
[PS,f] = pyulear(x,17, N,fs);
    ..... plot, labels, and axis .....
%
% Yule-Walker spectra, 25th order
[PS,f] = pyulear(x,25, N,fs);
    ..... plot, labels, and axis. .....
%
% Yule-Walker spectra, 35th order
[PS,f] = pyulear(x,35, N,fs);
    ..... plot, labels, and axis .....
%
% Modified covariance method, 25th order
[PS,f] = pmcov(x,25,N,fs);
    ..... plot, labels, and axis .....
```

Results

The resultant waveform is analyzed using different power spectral methods: the FFT-based method for reference, the Yule–Walker method with model orders of 11, 17, 25, and 35, and the modified covariance method with a model order of 25. As is shown in Figure 5.3, increasing model order improves spectrum resolution, but the broadband signal is not as well defined. At the highest model order, a number of spurious frequency peaks can be observed. This trade-off is also shown in Figure 5.4. The influence of noise on the AR spectrum is explored in the next example.

EXAMPLE 5.3

Explore the influence of noise on the AR spectrum specifically with regard to the ability to detect two closely related sinusoids.

Solution

The program uses `sig_noise` to generate a signal, `x`, that contains 240- and 280-Hz sine waves at four levels of SNR (0, -4, -9, and -15 dB, $N = 1000$). A loop is used to run through waveform generation and power spectrum estimations for the four noise levels. The Yule–Walker AR method is used to determine the power spectrum at each SNR, which is then plotted. A 15th-order model is used.

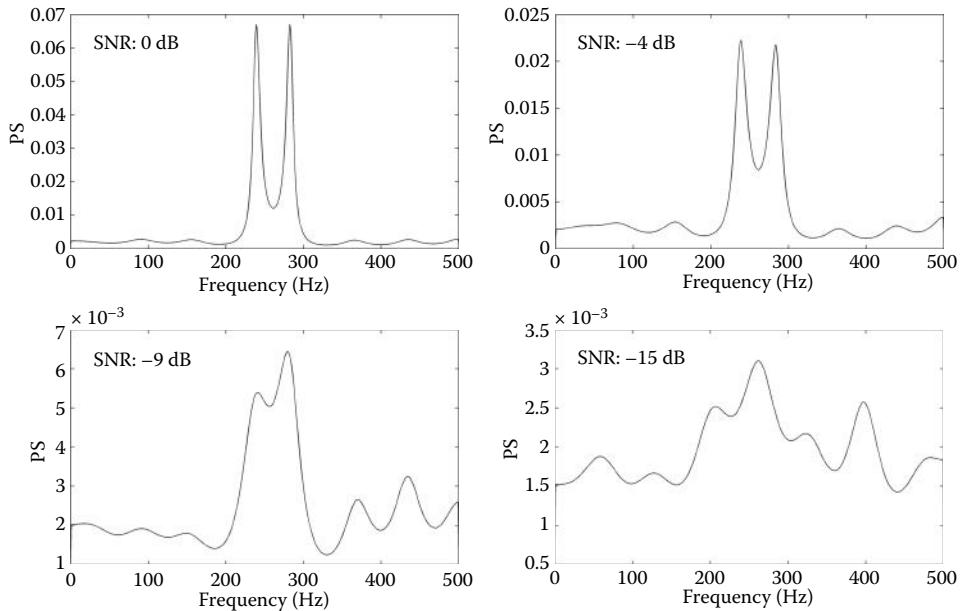


Figure 5.6 AR spectral estimation of a waveform consisting of two closely spaced sinusoids (240 and 280 Hz) buried in different levels of noise. The SNR has a strong influence on the effective spectral resolution.

```
% Example 5.3
% Program to evaluate the effect of noise on the AR spectrum
%
N = 1024; % Signal length
fs = 1000; % Sample frequency from sig_noise
p = 15; % Model order
SNR = [0 -4 -9 -15]; % Define SNR levels in dB
for k = 1:4
    x = sig_noise([240 280],SNR(k),N);
    [PS,f] = pyulear(x,p,N,fs);
    .....plots and labels .....
end
```

Results

The output of this example is presented in Figure 5.6. Note that the sinusoids are clearly identified at the two lower noise levels, but appear to merge together for the higher noise levels. At the highest noise level, only a single, broad peak can be observed at a frequency that is approximately the average of the two sine wave frequencies. Hence, with the AR model, spectral resolution depends on the SNR. The signal length, the number of samples in the signal, also strongly influences the resolution of all spectral methods. The relationship between the ability to detect narrowband signals and both signal length and SNR is explored in the problems at the end of this chapter.

5.2 Nonparametric Analysis: Eigenanalysis Frequency Estimation

Eigenanalysis spectral methods are promoted as having better resolution and better frequency estimation characteristics, especially at high noise levels. The basic idea is to separate correlated

and uncorrelated signal components using a linear algebra decomposition method termed *singular value decomposition*. This technique will also be used for evaluating principal components in Chapter 9. This so-called eigen decomposition approach is particularly effective in separating highly correlated signal components such as sinusoidal, exponential, or other *narrowband* processes from uncorrelated white noise. Conversely, it is not good at representing the spectra of broadband signals. However, when highly correlated signal components are involved (i.e., narrowband components), these eigen methods can eliminate much of the uncorrelated noise contribution. However, if the noise is not white (i.e., colored or broadband noise containing spectral features), then this noise will have some correlation, and the separation provided by singular value decomposition will be less effective. The reliance on signal correlation explains why these approaches are not very good at separating broadband processes from noise, since data from broadband processes are not highly correlated.

The key feature of eigenanalysis approaches is to divide the information contained in the waveform into two subspaces: a signal (i.e., correlated) subspace, and a noise (i.e., uncorrelated) subspace. The eigen decomposition produces a transformed data set where the components, which are called *eigenvectors*, are ordered with respect to their energy. A component's energy is measured in terms of its total variance, which is called the component's *eigenvalue*. Thus, each component or eigenvector has an associated variance measure, its eigenvalue, and these components are ranked ordered from highest to lowest eigenvalue. More importantly, the transformed components are *orthonormal* (i.e., orthogonal and scaled to a variance of 1.0). This means that if the noise subspace components are eliminated from the spectral analysis, then the influence of that noise is completely removed. The remaining signal subspace components can then be analyzed with standard Fourier analysis. The resulting spectra show sharp peaks where sinusoids or other highly correlated (i.e., narrowband) processes exist. Unlike parametric methods discussed above, these techniques are not considered true power spectral estimators, since they do not preserve signal power, nor can the autocorrelation sequence be reconstructed by applying the Fourier transform to these estimators. Better termed *frequency estimators*, they function to detect narrowband signals but in relative units.

The eigen decomposition needs multiple observations since it operates on a matrix. If multiple observations of the signal are not available, the autocorrelation matrix can be used for the decomposition. Alternatively, a single observation can be divided up into segments as is done in the Welch averaging method.

5.2.1 Eigenvalue Decomposition Methods

One of the first eigen decomposition methods to be developed is termed the Pisarenko harmonic decomposition method. A related method is the *multiple signal classification (MUSIC)* algorithm approach, which improves on the original algorithm but at the cost of great computer time. Both approaches are somewhat roundabout and actually estimate the spectrum of the noise subspace, not the signal subspace. The logic is based on the assumption that after decomposition, the signal and noise subspaces are orthogonal so that in the spectrum of the *noise subspace* will be small at frequencies where a signal is present. The estimate of the noise subspace frequency spectrum is in the denominator, so large peaks will occur at those frequencies where narrowband signals are present. By placing the frequency estimation in the denominator, we achieve the same advantage as in the AR model-based analysis: we are able to generate spectral functions having very sharp peaks, which allow us to clearly define any narrowband signals present. An investigation of the behavior of an eigen-based frequency estimation using the *signal subspace* is given in one of the problems.

The basic equation for the MUSIC narrowband estimator is

$$PS(\omega) = \frac{1}{\sum_{k=p+1}^M |V_k^H e(\omega)|^2 / \lambda_k} \quad (5.13)$$

Biosignal and Medical Image Processing

where M is the dimension of the eigenvectors, λ_k is the eigenvalue of the k th eigenvector, and V_k^H is the k th eigenvector usually calculated from the correlation matrix of the input signal. The eigenvalues, V_k^H , are ordered from the highest to the lowest power, so presumably the signal subspace is the lower dimensions (i.e., lower values of k) and the noise subspace the higher. The integer p is considered the dimension of the *signal* subspace (from 1 to p), so the summation taken from $p + 1$ to M is over the noise subspace.

The vector $e(\omega)$ consists of complex exponentials:

$$e(\omega) = [(1, e^{j\omega}, e^{j2\omega}, e^{j3\omega}, \dots, e^{j(M-1)\omega})]^T \quad (5.14)$$

so the term $V_k^H e(\omega)$ in the denominator is just the Fourier transform of the eigenvalues, and Equation 5.13 reduces to

$$PS(\omega) = \frac{1}{\sum_{k=p+1}^M |FT(V_k)|^2 \lambda_k} \quad (5.15)$$

Note that the denominator term is the summation of the scaled power spectra of noise subspace components.

Since this frequency estimator operates by finding the signals, or really absence of signals, in the noise subspace, the eigenvectors V_k^H used in the sum correspond to all those in the decomposed correlation matrix greater than p , presumably those in the noise subspace. The Pisarenko frequency estimator was a precursor to the MUSIC algorithm and is an abbreviated version of MUSIC in which only the first noise subspace dimension is used; in mathematical terms, $M = p + 1$ in Equation 5.15, so no summation is required. Accordingly, the Pisarenko method is faster since only a single Fourier transform is calculated, but considerably less stable for the same reason: only a single dimension is used to represent the noise subspace.

Since $e(\omega)$ is a complex exponential having harmonically related frequencies ($1\omega, 2\omega, 3\omega, \dots$), the inner product, $V_k^H e(\omega)$, is just the Fourier transform of the eigenvectors V_k^H as indicated in the lower equation. The magnitude from each of the Fourier transforms is squared and summed to produce the power spectrum over all noise subspace dimensions. The inverse is the noise subspace estimate, and it is taken as the frequency estimate, which leads to very sharp peaks in the spectral estimate. This equation is easy to implement in MATLAB, as shown in Example 5.4.

5.2.2 Determining Signal Subspace and Noise Subspace Dimensions

If the number of narrowband processes is known, then the dimensions of the signal subspace can be calculated using this knowledge: since each real sinusoid is the sum of two complex exponentials, the signal subspace dimension should be twice the number of sinusoids, or narrowband processes, present. In some applications, the signal subspace can be determined by the size of the eigenvalues. Here the eigenvalues (proportional to component energies) are plotted out. Noise components should all have about the same energy leading to a flat line plot, while signal components will decrease in energy (remember all components are ordered by energy level). The idea is to find the component number where the eigenvalue plot switches from a down slope to a relatively flat slope. This is explored in an example below and works well with artificial data where the noise is truly uncorrelated, but does not often work in practice, particularly with short data segments (Marple, 1987). As with the determination of the order of an AR model, the determination of the signal subspace usually relies on a trial-and-error approach.

5.2.3 MATLAB Implementation

With the aid of the `corrmtx` and `fft` routines, Equation 5.13 can be implemented directly. This is illustrated in the next example. As usual, MATLAB makes it even easier to construct power spectra based on eigenanalysis with two routines, one implementing the Pisarenko harmonic decomposition method and the other the MUSIC algorithm. As with AR, these routines

are not really necessary, but they shorten the code, eliminate redundant points, and provide a frequency vector for plotting. Both methods have the same calling structure, a structure similar to that used by the AR routines.

```
[PS,f,v,e] = peig(x,[p thresh],nfft,Fs,window,nooverlap);
[PS,f,v,e] = pmusic(x,[p thresh],nfft,Fs,window,nooverlap);
```

The arguments of the routines are the same. The last four input arguments are optional and have the same meaning as in `pwelch`, except that if `window` is either a scalar or omitted, a rectangular window is used (as opposed to a Hamming window). The first argument, `x`, is the input signal, and if multiple observations are available, `x` should be a matrix in which each row is a signal. If only a single observation is available, `x` is a vector, and the routine uses `corrmtx` to generate the correlation matrix. The examples given here and many of the problems use a single input signal, but the last two problems at the end of the chapter use the matrix input feature of these routines. The second argument is used to control the dimension of the signal (or noise) subspace. Since this parameter is critical to the successful application of the eigenvector approach, extra flexibility is provided as described in the associated help file. Usually a single number, `p`, is used and is taken as the dimension of the signal subspace. Of course `p` must be $< N$, the dimension of the eigenvectors. The dimension of the eigenvectors, `N`, is either `nfft` or, if not specified, the default value of 256.

The data argument, `x`, is also flexible. If `x` is a vector, then it is taken as one observation of the signal as in previous AR and Welch routines. However, `x` can also be a matrix, in which case the routine assumes that each row of `x` is a separate observation of the signal. For example, each row could be the output of an array of sensors, or a single response in an ensemble of responses. Data featuring multiple observations are termed *multivariate* and are discussed in Chapter 9. Finally, the input argument `x` could be the correlation matrix, for example, obtained from `corrmtx`. In this case, `x` must be a square matrix, and the argument '`corr`' should be added to the input sequence anywhere after the argument `p`. If the input is the correlation matrix, then the arguments `window` and `nooverlap` have no meaning and are ignored. If the '`corr`' option is used, then `N` is the size of the correlation matrix.

The main output argument is `PS`, which contains the power spectrum (more appropriately termed the *pseudospectrum* due to the limitations described previously). The second output argument, `f`, is the familiar frequency vector useful in plotting. The third optional output argument, `v`, is a matrix of eigenvectors spanning the noise subspace (one per column) and the final output argument, `e`, is either a vector of singular values (squared) or a vector of eigenvalues of the correlation matrix when the input argument '`corr`' is used. The output can be useful in estimating the size of the signal subspace.

EXAMPLE 5.4

Load the file `narrowband_signals.mat` that contains a signal vector `x` consisting of two sinusoids at 100 and 200 Hz in 12 dB of noise. Use eigen decomposition based on Equation 5.13 to estimate the spectrum. Compare the results with MATLAB's `pmusic` routine.

Solution

Compute the autocorrelation matrix using MATLAB's `corrmtx` routine and apply singular value decomposition using the `svd` routine. (It is also possible to use `xcorr` and `toeplitz` instead of `corrmtx` as described previously.) Then take the Fourier transform of each eigenvector, square the magnitude, divide by the eigenvalue, and sum. Take the inverse of this sum as the power spectrum and remove the redundant points. Since there are two sinusoids in the waveform, the subspace dimension should be 4 or higher, so we will try a dimension of 5. To

Biosignal and Medical Image Processing

construct the autocorrelation matrix, we use a dimension of 10, which will become the value of the noise and signal subspace. Since we are using a signal subspace of 5, the noise subspace will also be 5.

```
% Example 5.4 Spectral analysis using eigen-decomposition based on Eq. 5.13.  
%  
fs = 1000; % Sampling frequency  
p = 5; % Signal subspace dimension  
M = 10; % Number of noise and signal subspace dimensions  
load narrowband_ana; % Load data  
N = length(x); % Signal length  
%  
[cor,Rxx] = corrmtx(x,M); % Generate the autocorrelation matrix  
[U,D,V] = svd(Rxx,0); % Singular value decomposition  
eigen = diag(D); % Find the eigenvalues  
for k = p + 1:length(eigen); % Calculate the individual Fourier transforms  
    Sxx(k-p,:) = abs(fft(V(:,k),256)).^2/eigen(k);  
end  
PS = 1./sum(Sxx); % Sum Fourier transforms and invert  
PS = PS(1:round(length(PS)/2)); % Remove redundant data  
f = (0:length(PS)-1)*fs/(2*length(PS)); % Compute the frequency vector  
[PS1,f1] = pmusic(x,p,N,fs); % MATLAB eigen-decomposition  
.....plot and label.....
```

Analysis

The program sets up the parameters, including N , f_s , p , and M . The latter is the total number of dimensions used. In the MUSIC method, this should be much greater than $p + 1$ to include dimensions for the noise subspace. (In the Pisarenko method, only a single additional dimension is used to represent the noise subspace.) In this case, the total dimension is set to 11, as the routine `corrmtx` calculates the autocorrelation matrix having dimensions of $M + 1$ by $M + 1$. Increasing the total number of dimensions increases the size of the noise subspace allowing for better representation of the noise, but also increases the computational effort. A total dimension of 11 leads to a noise subspace dimension of 6 (11–5), which appears to work well.

Results

Figure 5.7 shows well-defined spectra with excellent frequency resolution obtained from both Equation 5.13 and the MATLAB routine `pmusic`. The two peaks found are unequal due to differences in scaling, but remember that these spectra do not represent actual energy, so the amplitude of a peak is not that informative. Comparisons of eigen decomposition and AR methods are found in the problem set.

In Example 5.4, the subspace dimension was 5 since it was known *a priori* that the waveform contained two sinusoids. If the number of components is not known in advance, a plot of the singular values (or eigenvalues) may be helpful in determining the number of components. This plot, known as the *Scree* plot,* will descend steeply over the signal subspace (since the singular values are ordered), but level off over the noise subspace since all noise components should have

* Scree is the loose rock on the slope of a mountain. The downward slope of the Scree plot has the appearance of a mountainside (with some imagination).

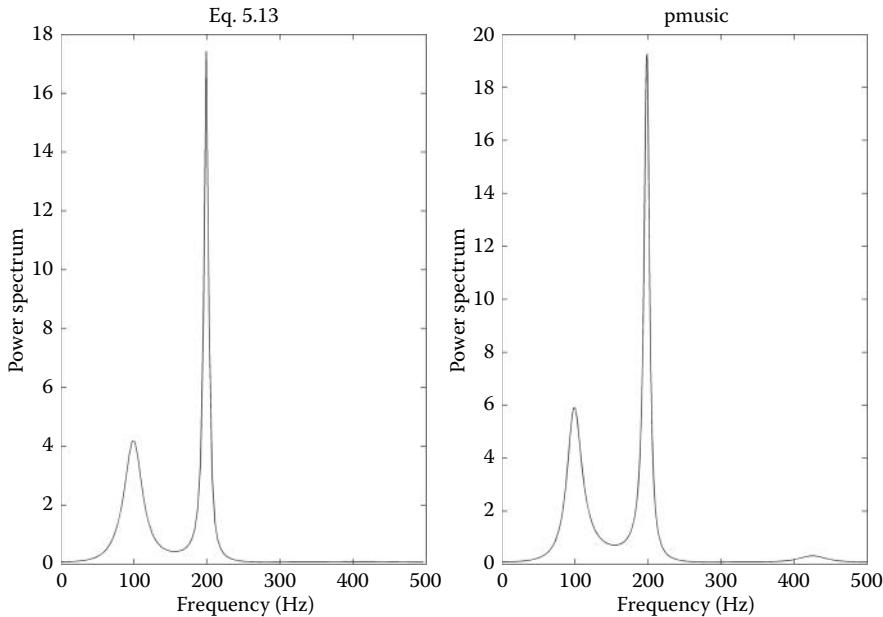


Figure 5.7 Eigenanalysis power spectra obtained using the basic equation (Equation 5.14, left side) and MATLAB's pmusic routine (right side).

about the same variance. This strategy is examined in the next example, which presents the Scree plots for waveforms containing different numbers of sinusoids.

EXAMPLE 5.5

Construct and show the Scree plots from waveforms containing 1, 2, 3, and 4 sinusoids.

Solution

Generate the four waveforms, use `corrmtx` to calculate the autocorrelation functions, and use `svd` to determine the singular values (i.e., eigenvalues). Plot the singular values as a function of their component number. Use a loop to make the code shorter.

```
% Example 5.5 Program to demonstrate the use of the Scree plot to
% determine the number of components.
N = 1200; % Size of data vector
x(1,:) = sig_noise(100,-12,N); % Generate data
x(2,:) = sig_noise([100 200],-12,N);
x(3,:) = sig_noise([100 200 300],-12,N);
x(4,:) = sig_noise([100 200 300 400],-12,N);
for i = 1:4
    [X,Rxx] = corrmtx(x,20); % Const. autocorr matrix
    [U,D,V] = svd(Rxx,0); % Calculate singular values
    eigen = diag(D); % and extract eigenvalues
    subplot(2,2,i);
    plot(eigen(1:20), 'k'); % and plot
    .....labels.....
end
```

Results

The Scree plots produced by Example 5.5 are shown in Figure 5.8. The Scree plot for one sinusoid (upper left) shows a clear change in slope or *breakpoint* at 3, and the plot for two sinusoids (upper right) shows a breakpoint at 5. Both indicate the correct number of components. The Scree plot for three sinusoids (lower left) also shows an expected breakpoint at 7. While the plot for four components (lower right) does have a breakpoint at 10, other breakpoints are seen, and the correct one is not easily determined. In real data, where the noise is not completely uncorrelated, clear breakpoints are rare; however, the Scree plot analysis is generally worth the minimal effort when the number of narrowband processes is in doubt.

The strengths and weaknesses of the eigenanalysis method and the influence of signal subspace dimension are both illustrated in Figure 5.9, which shows four spectra obtained from the same small data set ($N = 32$) consisting of two closely spaced sinusoids (150 and 200 Hz) in white noise ($\text{SNR} = -4 \text{ dB}$). Figure 5.9 (upper left) shows the spectrum obtained using the classical Fourier transform. The other three plots show spectra obtained using eigenvector analysis (MUSIC), but with different partitions between the signal and noise subspaces. In Figure 5.9 (upper right), the spectrum was obtained using a signal subspace dimension of 4. In this case, the size of the signal subspace is not large enough to differentiate between the two closely spaced sinusoids, and only a single peak is seen. When the signal subspace dimension is increased to 7 in Figure 5.9 (lower left), the two sinusoidal peaks are clearly distinguished and much of the

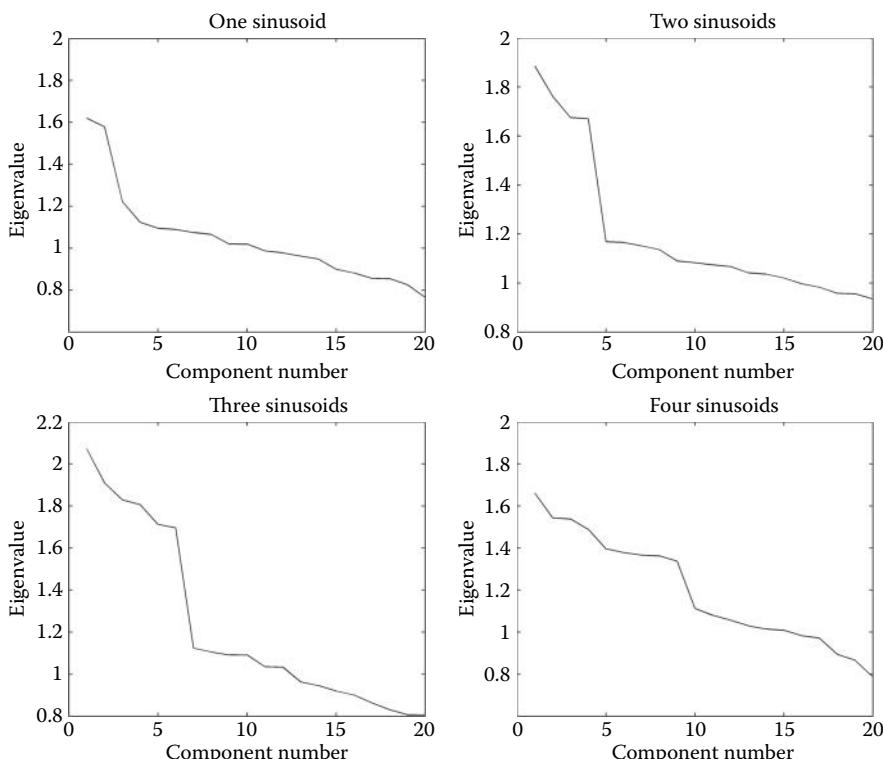


Figure 5.8 Scree's plots from waveforms containing one, two, three, and four sinusoids in 12 dB of noise. Breakpoints indicate the correct number of components in the first three cases, but when four sinusoids are present, the breakpoint is difficult to discern. With real data, breakpoints are usually vague.

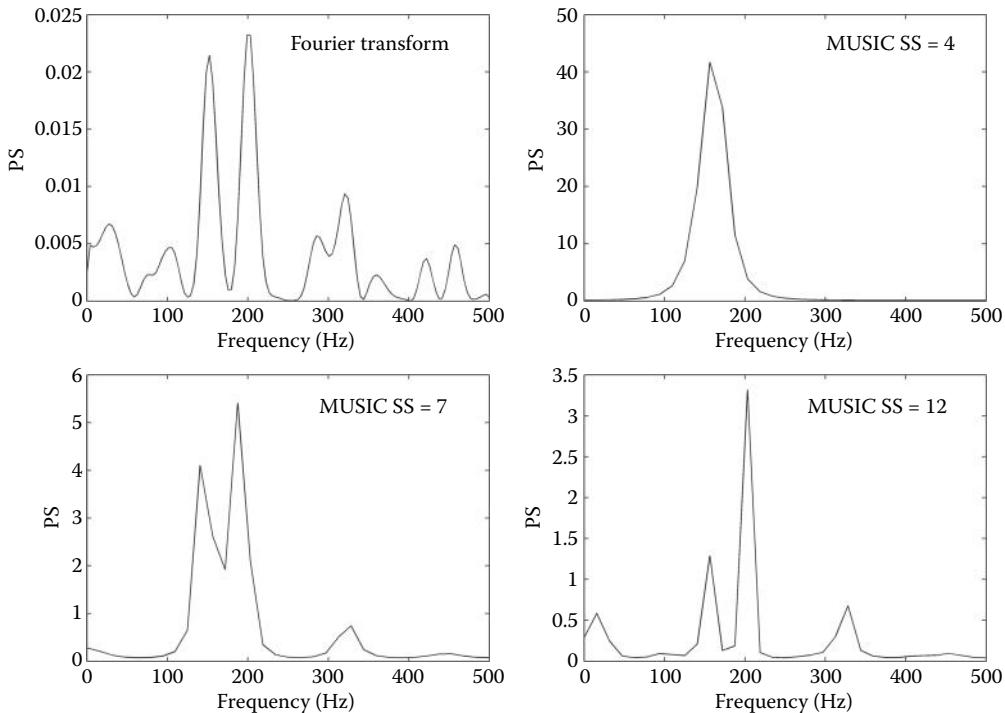


Figure 5.9 Spectra produced from a short data sequence ($N=32$) containing two closely spaced sinusoids (150 and 200 Hz) in white noise (SNR = -4 dB). The upper left plot was obtained using the Fourier transform, while the other three use eigenanalysis with different partitions between the signal and noise subspace.

white noise spectrum is absent (i.e., the energy at other frequencies is close to zero). However, when the signal subspace is enlarged further, to a dimension of 12 (Figure 5.9, lower right), the resulting spectrum contains the two peaks but also has two spurious peaks. This shows that the location of the subspace boundaries can substantially affect the resultant spectrum.

The next example compares the three spectral approaches (Fourier transform, AR modeling, and eigenanalysis) to the problem of identifying narrowband signals, specifically, four sinusoids in 12 dB of noise. Two of the signals are closely spaced. The identification of narrowband signals in noise is a frequent challenge in biomedical engineering problems.

EXAMPLE 5.6

Compare classical, AR, and eigenanalysis spectral analysis methods applied to narrowband signals in an evaluation of their ability to identify such signals. The signal should consist of sinusoids of frequencies 50, 80, 240, and 400 Hz with an SNR of 12 dB.

Solution

Generate the signal using `sig_noise`. Use `pwelch` to calculate the power spectrum using the averaging method. Use the default window size and overlap. Use the modified covariance method (`pmcov`) for the AR model and `pmusic` for the eigenvalue method. Adjust the AR model order and the MUSIC signal subspace for best detection and discrimination of the narrowband signals.

Biosignal and Medical Image Processing

```
% Example 5.6
% Compares FFT-based, AR, and MUSIC Spectral methods
%
N = 1024;                                % Size of waveform
fs = 1000;                                 % Sample frequency
SNR = -12;                                 % SNR
%
% Generate a spectra of sinusoids and noise
x = sig_noise([50 80 240 400],SNR),N;
%
% Estimate the Fourier transform spectrum
[PS,f] = pwelch(x,N,[],[],fs);
.....plot, axis, labels, title.....
% AR Modified Covariance Spectrum 15th order
[PS,f] = pmcov(x,15,N,fs);
.....plot, labels, title.....
% AR Modified Covariance Spectrum 31th order
subplot(2,2,3);
[PS,f] = pmcov(x,31,N,fs);
.....plot, labels, title.....
%
% MUSIC spectrum, Signal space = 15
[PS,f = pmusic(x,15,N,fs);
.....plot, labels, title.....
```

Results

The plots produced by Example 5.6 are shown in Figure 5.10, and the usefulness of the eigenvector method for this task is apparent. In this example, the data length is quite long ($N = 1024$), but the SNR is low (-12 dB). The signal consists of four equal-amplitude sinusoids, two of which are closely spaced (50 and 80 Hz). All three methods detect the four peaks, provided that the order of the AR model is high enough. However, the eigenvector method (Figure 5.10, lower right) shows the peaks with excellent frequency resolution and shows no background noise. Since we know that background noise was actually present, this is not an accurate spectral representation of the waveform, but if all we are interested in is narrowband detection, the eigenvalue method does very well. However, unlike the Fourier transform method, it does require specifying an important parameter, the boundary between the signal and noise subspace, and the value of this parameter has a strong influence on the resultant spectrum as shown in Example 5.5.

The eigenvector method also performs well with short data sets. The behavior of the eigenvector method with short data sets and other conditions is compared to that of other methods in the problems.

Most of the examples provided in this book evaluate the ability of the various spectral methods to identify sinusoids or narrowband processes in the presence of noise, but what of a more complex spectrum? Example 5.7 explores the ability of the classical (the Welch-Fourier transform), AR, and eigenvalue methods in estimating a more complicated spectrum. In this example, we use a linear process, one of the filters described in Chapter 4, driven by white noise, to create a more complicated, broadband spectrum. Since the filter is driven by white noise, the power spectrum of its output waveform should be approximately the same as the power spectrum of the filter (i.e., the system's frequency characteristics squared). The `fir2` routine is used to develop a filter with an initial upward slope followed by a sharp cutoff. The effective power spectrum of this filter is determined by squaring the Fourier transform of the b coefficients, and this spectrum is compared with that found by the three different methods.

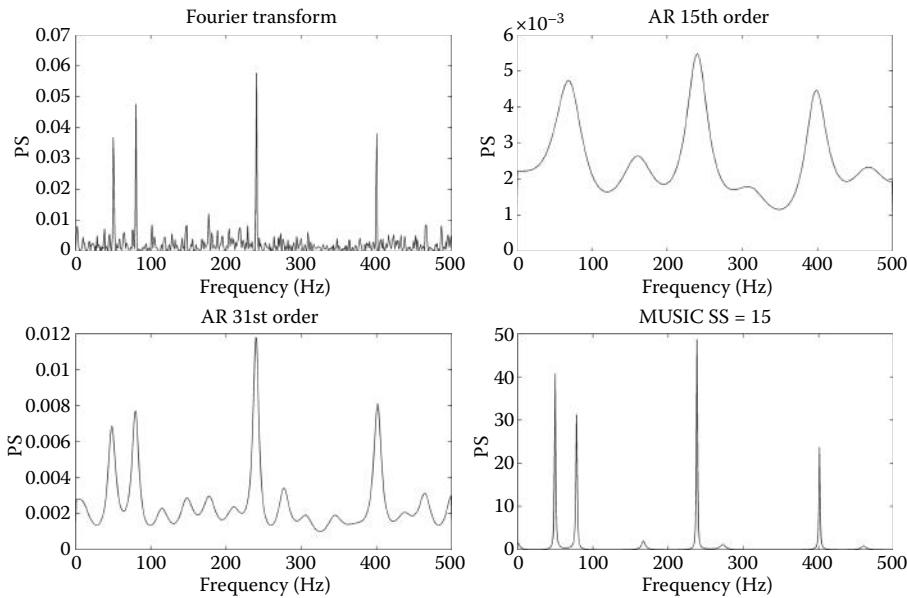


Figure 5.10 Spectra using *three* different methods applied to a waveform containing four equal-amplitude sinusoids at frequencies of 50, 80, 240, and 400 Hz, and white noise ($\text{SNR} = -12 \text{ dB}$; $N = 1024$). While all three methods detect the four peaks, the eigenvector method (lower right) identifies the four narrowband signals with the least amount of background noise.

EXAMPLE 5.7

Generate a broadband signal by applying an FIR filter to white noise. Compare the effective power spectrum of this filter with the output signal's power spectrum as found by the Welch averaging technique, the AR (modified covariance), and the MUSIC eigenvalue methods. The FIR filter should have 28 coefficients; its frequency characteristic should increase linearly with frequency up to $0.4 f_s$, then fall rapidly to zero. Determine the output of this filter to a white noise input ($N = 512$, $f_s = 200 \text{ Hz}$) and find the power spectrum of this output using the three methods. Adjust the Welch window, the AR model order, and the MUSIC signal subspace to best reflect the filter's spectrum.

Solution

Use `randn` to generate the white noise signal and `fir2` to produce the FIR filter coefficients. Square the magnitude of the Fourier transform of the padded filter coefficients to get the effective power spectrum. (Pad the b coefficients to 512 points, way more than needed for a smooth spectrum.) Use `pwelch` to find the averaged power spectrum, adjusting the window size for the best match with the filter power spectrum. Use `pmcov` and `pmusic` to generate the AR and eigenvalue power spectra, again adjusting AR model order and eigenvalue signal subspace for the best results. Use `subplot` to place the four graphs together for easy comparison.

```
% Example 5.7 Comparison of spectral methods
%
fs = 200;          % Sampling frequency
N = 512;           % Data length
fc = 0.4;          % Derivative cutoff frequency
N1 = 28;           % Filter length
```

Biosignal and Medical Image Processing

```

x = randn(N,1); % Generate white noise signal
%
% Design filter and plot magnitude characteristics
f1 = [0 fc fc + .01 1]; % Specify filter frequency curve
G = [0 1 0 0]; % Upward slope until 0.4 fs
b = fir2(N1,f1,G); % Find FIR filter coefficients
x = filter(b,1,x); % and apply filter
%
% Calculate and plot filter's power spectrum
PS = abs(fft(b,N)).^2; % Calculate and plot
..... subplot, plot PS, labels, and text.....
%
[PS,f] = pwelch(x,N/8,(N/8)-1,[ ],fs); % Welch, use 99% overlap
..... subplot, plot PS, labels, and text.....
%
[PS,f] = pmcov(x,9,N,fs); % AR - Model order = 9
.....subplot, plot PS, labels, and text.....
%
[PS,f] = music(x,9,N,fs); % MUSIC Signal subspace = 9
.....subplot, plot PS, labels, and text.....

```

Results

The plots produced by this program are shown in Figure 5.11. In Figure 5.11a, the square of the filter's magnitude transfer function is plotted so that it can be directly compared with the

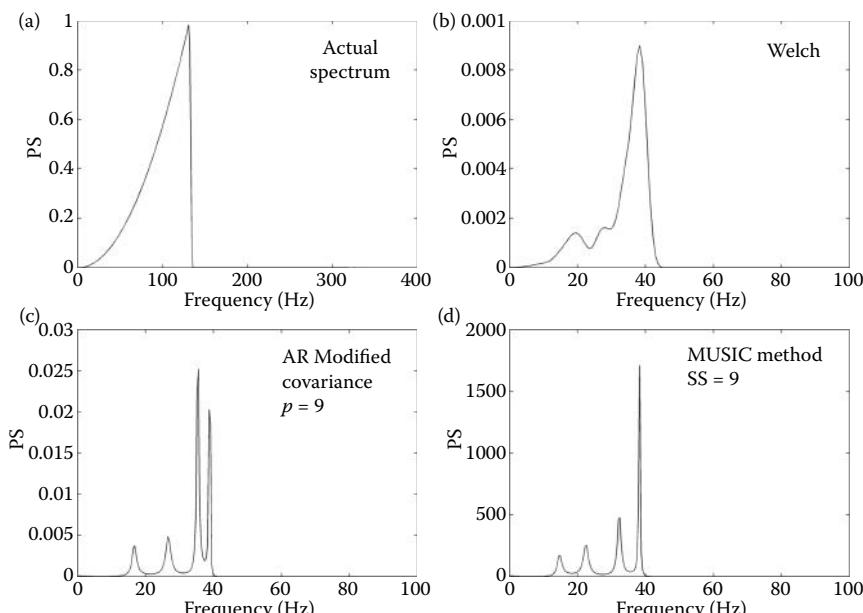


Figure 5.11 Estimation of a broadband spectrum using classical, model-based, and eigenvalue methods. (a) Transfer function of the filter used to generate the data plotted a magnitude squared. (b) Estimation of the data spectrum using the Welch method with eight averages and maximum overlap. (c) Estimation of data spectrum using AR model, $p = 9$. (d) Estimation of data spectrum using an eigenvalue method with a subspace dimension of 9.

various spectral techniques that estimate the power spectrum of the filter's output waveform. The parameters associated with the various approaches have been adjusted to produce as close a match to the filter transfer function as possible. The influence of these parameters is explored in the problems. The power spectrum produced by the classical approach is shown in Figure 5.11b. Since the signal being analyzed is filtered noise, an exact match with the actual filter spectrum should not be expected, but the Welch method does lead to a power spectrum that is similar to that of the filter. This classical approach averages eight segments with maximum overlap. Both the model-based approach with a model order of 9 and the MUSIC method with a signal subspace of 9 can only approximate the actual power spectrum using a series of narrowband peaks. As one of the problems shows, changing the model order or signal subspace only increases or decreases the number of peaks used to approximate the spectrum. This demonstrates that AR and eigenvalue methods, while well suited to identifying sinusoids in noise, are not good at estimating spectra that contain broadband features.

5.3 Summary

If you want the power spectrum of a waveform, noise, and all, classical Fourier transform methods are the way to go. However, if your waveform consists of signal and noise, and you prefer to estimate the power spectrum of only the signal, alternative methods may work better. Modern spectral methods only produce power spectra and have no phase information. They do not provide a complete representation of the signal and cannot be used bidirectionally. However, as noted in Chapter 4, often, only the power spectrum (or magnitude spectrum) is of interest. These alternatives are divided into two categories: model-based or parametric methods, and nonparametric eigen decomposition methods.

Parametric methods fit the output of a model, theoretically driven by white noise, to the spectrum of the signal being analyzed. The model's parameters can then be used to determine the spectrum of the model using equations given in Chapter 4. Since white noise has constant energy at all frequencies, the frequency alterations produced by the model must reflect those of the signal, that is, the model's spectrum must also be the spectrum of the signal. This round-about approach gives us control over the resulting spectrum. If we use a more complex model, we will get a more complex spectrum and vice versa. This control carries with it an obligation, as we must decide in advance how complicated to make the model. Trial-and-error methods can be used to find the right level of model complexity.

Models used in parametric spectral analysis are defined in terms of their digital transfer functions (i.e., Z-transform transfer functions). Models with only numerator terms in the transfer function are termed MA models; from a systems viewpoint, they are identical to the FIR filters described in Chapter 4. These models are rarely used in spectral analysis because the spectra they produce, consisting of valleys, are not very useful, and the computation required to find the appropriate match to the signal is complicated. Models with only denominator terms in the transfer function are termed AR models, and are like IIR filters with only one numerator (i.e., a) coefficient that equals 1.0. Finding the coefficients of an AR model that produce a signal match is straightforward and, since the defining function is in the denominator, spectra with sharp peaks can be generated. These features make AR models popular as model-based spectral tools. More complicated spectra can be obtained using models that combine the features of MA and AR models, logically called ARMA models. These models have digital transfer functions having both numerator and denominator coefficients (i.e., a and b coefficients). As systems, they are identical to the IIR filters of Chapter 4. Unfortunately, it is more difficult to find all the model coefficients and, since they are found in turn, they may not optimally reflect the spectrum.

AR models are studied in this chapter; they can represent spectra containing narrowband features well because of their ability to generate sharp spectral peaks. However, they are not good

Biosignal and Medical Image Processing

at representing broadband spectra, since they approximate such spectra by a series of spikes. AR spectral analysis is well supported by MATLAB through a number of special routines.

The ultimate tools for identifying narrowband processes, particularly in a noisy waveform, are the eigen decomposition methods. These approaches use a matrix decomposition method, termed singular value decomposition, and require multiple signals. Occasionally, multiple signal observations are available, such as in the visual-evoked response discussed in Chapter 4 (see Problems 5.26 and 5.27). More often, a single signal is used and multiple signals constructed from the (auto) correlation matrix of this signal. (Note that these multiple signals are created from correlations between the signal and its delayed versions.) The multiple signals are then decomposed using the technique of singular value decomposition. This linear algebra decomposition is discussed at greater length in Chapter 9, but essentially decomposes the signal into orthogonal components, ordered so the larger components come first in the decomposition. The assumption is that the larger components are part of the signal while the smaller components are the noise.

Once you have the signal and noise component of subspaces, it is up to you to decide where to place the boundary between the two subspaces. Of possible help with this decision is the Scree plot, which plots the variance of each component as a function of component number. Since singular value decomposition orders the components from highest to lowest variance, the graph drops steeply, but levels off in the noise subspace since the noise components should have more or less the same variance. However, usually trial and error is required to find the best boundary.

After you identify the signal subcomponents, you could average the power spectra obtained from the signal components using traditional methods. (This approach is used in Problem 5.19.) However, sharper peaks can be obtained by averaging power spectra obtained from the noise subspace (again using traditional Fourier transform methods), with the assumption that the noise has an absence of signal so the noise spectra represent the inverse of the signal spectral. Hence, taking the inverse of the noise power spectrum should give you the signal spectrum. In the original Pisarenko method, only one noise dimension is used (see Problem 5.18), but in the MUSIC algorithm, the noise power spectrum is calculated by taking the average of individual noise component spectra. While computationally more intensive, the averaging leads to a stabler estimate of the signal spectrum.

PROBLEMS

If the Signal Processing Toolbox is not available, all the AR problems, with the exception of Problem 5.9, can be solved using the code for the Yule–Walker equations given in Example 5.1. The eigen-spectral problems can be solved using the code in Example 5.4.

- 5.1 Use the basic Yule–Walker equations given in Example 5.1 to find the spectrum of the signal used in that example, but use a higher-order model to better resolve the four frequencies. Since this is a fairly complicated spectrum, you may need to use a very high order for high spectral resolution. Compare the AR power spectrum with that obtained using classical Fourier transform methods. As shown in Example 5.2, an easy way to calculate the Fourier transfer function is to use `pwelch` with a window that is the same length as the signal. The routine uses the “direct method” to determine the power spectrum but also provides the frequency vector and eliminates the redundant points.
- 5.2 This problem examines the effect of noise level on the ability to determine a complex spectrum. Repeat Problem 5.1 but with an SNR of -18 dB. Note that it is generally impossible to resolve the four narrowband signals. (Rerunning this problem several times will reveal the variability associated with noisy data, but it is unlikely that you will be able to resolve all four peaks.)

- 5.3 This problem further illustrates the difficulty of resolving narrowband signals in high levels of noise. Use `sig_noise` to generate a 256-point signal containing two closely spaced sinusoids at 200 and 230 Hz both with SNR of -8 dB. (Recall `sig_noise` assumes $f_s = 1$ kHz.) Find the best AR model order to distinguish these two peaks with a minimum of spurious peaks. Use the Burg method. Compare this with the Fourier transform power spectrum. (Again, you can use `pwelch` with a window the same length as the signal.) Repeat with an SNR of -14 dB. Note that at the higher noise level, it is usually difficult to resolve the two peaks. (Rerunning this problem several times will reveal the variability associated with noisy data.)
- 5.4 This problem emphasizes the effect of signal length on the ability to detect narrowband signals. Repeat Problem 5.1. Use a shorter data segment ($N = 64$), but a higher SNR (0 dB). Now, lower the SNR to -5 dB and again note the severe degradation in performance and the difficulty of resolving the two narrowband peaks at the higher noise level.
- 5.5 This problem directly explores the effect of signal length on spectral resolution using AR methods. Use `sig_noise` to generate two closely spaced, and one more distant, sinusoid ($f = 240, 260, 350$ Hz) with an SNR = -8 dB. Use the modified covariance method (`pmcov`) to determine the power spectrum of this signal. Use a model order of 35 since this is a fairly complicated spectrum. Repeat using a loop for signal lengths of 64, 128, 256, and 512. What is the minimum signal length necessary to guarantee identification of the three signals? Rerun the program several times to ensure that identification is always possible at this SNR.
- 5.6 This problem complements Problem 5.5 and directly explores the effect of SNR on spectral resolution using AR methods. Use `sig_noise` to generate the signal used in Problem 5.5 but make signal length constant, $N = 256$. Use the modified covariance method (`pmcov`) to determine the power spectrum of this signal. Use a model order of 35 since this is a fairly complicated spectrum. Repeat using a loop for SNR's of $-6, -10, -12$, and -14 dB. What is the minimum SNR necessary to guarantee identification of the three signals? Rerun the program several times to ensure identification is always possible at this SNR.
- 5.7 Expand Example 5.3 to compare the AR method with the classical Fourier transform method. For the latter, you can use `pwelch` with a window the same length as the signal. Use the signals in Example 5.3, but make the SNRs equal to $-5, -12, -16$, and -20 dB. Plot the AR and Fourier transform spectra one above (using `subplot`) the other for easy comparison, and use `subplot` to put all the plots on the same page. To determine the AR power spectrum, use the modified covariance method (`pmcov`) with a model order of 25. Are there any SNR levels for which AR shows the two sinusoids better than the Fourier transform and vice versa? You may want to rerun this problem several times to confirm your evaluation.
- 5.8 The file `ar_compare.mat` contains a signal `x` containing two sinusoids at 300 and 340 Hz and noise ($f_s = 1$ kHz). Compare the ability of the AR and Fourier transform methods to identify the narrowband signals and their frequencies correctly. Do the comparison for AR model orders of $p = 9, 15, 23$, and 32. You can use `pwelch` with a window the same length as the signal as in Example 5.2. Plot the AR and Fourier transform methods one above (using `subplot`) the other for easy comparison, and use `subplot` to put all the plots on the same page. Are there any model orders for which the AR shows the two sinusoids better than the Fourier transform and vice versa? [Hint: This problem can be solved using an easy code modification from Problem 5.7.]

Biosignal and Medical Image Processing

- 5.9 Load the file `prob5_9_data.mat` that contains a signal x consisting of three equally spaced sinusoids in noise ($f_s = 1 \text{ kHz}$). Evaluate the power spectrum of this signal using four AR methods: Yule–Walker (`pyulear`), Burg (`pburg`), covariance (`pcov`), and modified covariance (`pmcov`). Use a model order of 30. Note the subtle differences between the four methods.
- 5.10 Load the file `spectral_analysis1.mat` that contains signal x composed of three *broadband* signals ($f_s = 1 \text{ kHz}$). Find the lowest AR model order that faithfully captures the three broadband signals. Use the Burg method. Then, in a separate plot, compare the power spectrum produced by the AR model using the lowest acceptable order with that produced by the classical FFT method.
- 5.11 Repeat Problem 5.10, but compare the AR model with the power spectrum generated using the Welch method. Use the default (50%) overlap and the window size that best represents the three broadband signals.
- 5.12 File `spectral_analysis3.mat` contains signal x , which has a lower-frequency broadband component and a single higher-frequency narrowband component at 400 Hz ($f_s = 1 \text{ kHz}$). Find the order of the AR model (modified covariance) that produces a power spectrum that best fits the broadband data, but still clearly identifies the narrowband signal. Then find the power spectrum generated using the Welch method. Using the default (50%) overlap, adjust the window size to produce a spectrum that best represents the broadband signals while still identifying the narrowband signal. Signals that contain both broadband data and narrowband components along with noise present the greatest challenges in spectral analysis.
- 5.13 File `spectral_analysis2.mat` contains signal x , which has a single broadband component ($f_s = 1 \text{ kHz}$). Use AR spectral analysis to find the bandwidth of the broadband component. Use the AR method of your choice to estimate the cutoff frequencies at the half-power points. [Recall that since you are using the power spectrum, the cutoff frequency which on a linear magnitude scale would be at 0.707 of the bandpass value is now at 0.5 (0.707²) of the average bandpass value.]
- 5.14 The file `filter_coeff1.mat` contains a and b filter coefficients for some type of filter. Assume $f_s = 250 \text{ Hz}$ for the spectral plots. Pass a Gaussian noise signal ($N = 2058$) through a filter based on these coefficients and plot the power spectrum of the filter's output signal using an AR model. Use the AR method of your choice to determine the type of filter and estimate the cutoff frequency(s). A very low-order filter works best to reduce spurious peaks due to noise. Note that the “pass” region of this filter is represented by a multiple peaks.
- 5.15 Repeat Problem 5.14 using the a and b filter coefficients in file `filter_coeff2.mat`. Again assume $f_s = 250 \text{ Hz}$ for the spectral plots and pass Gaussian noise ($N = 2058$) through a filter based on these coefficients. Use the AR method of your choice to determine the type of filter and estimate the cutoff frequency(s). This filter is slightly more complicated than the one in Problem 5.14, so a moderately low-order filter works best to reduce spurious peaks due to noise. Note that the “pass” regions of this filter are represented by a multiple peaks.
- 5.16 The file `spectral_analysis5.mat` contains signal vector x composed of two broadband components ($f_s = 1 \text{ kHz}$). The two components have center frequencies of 150 and 350 Hz, and each has a bandwidth of 100 Hz. Compare the power spectrum produced by both the AR ($p = 4\text{--}8$, your choice of methods) and Welch methods for this signal. Repeat, but truncate the signal to $N = 126$ points. For the Welch method,

use a window size of 256 for the original signal, or the signal length when $N = 126$, and the default overlaps. Plot all four graphs in the same figure. Note that bandwidths are better shown with the Welch method, at least for the longer signal, but the AR method better shows the center frequencies particularly with the shorter signal.

- 5.17 The file `spectral_analysis6.mat` contains signal vector \mathbf{x} composed of a single broadband component ($f_s = 1 \text{ kHz}$). The component has a center frequency of 250 and a bandwidth of 300 Hz. Compare the power spectrum produced by AR with model orders of $p = 3, 16, 32$, and 64 . Note that the lowest model order accurately locates the center frequency; however, the AR represents the flat portion of the bandwidth as a series of frequency peaks. The higher model orders have more peaks and better represent the flat portion of the spectrum.
- 5.18 This problem compares MATLAB's `peig` with code based on a modified version of Equation 5.13. Load data file `prob5_18_dat.mat` containing signal vector \mathbf{x} , which consists of two sinusoids buried in 12-dB noise. To implement the Pisarenko method, modify Equation 5.13 and Example 5.4 so that $M = p + 1$. This requires only a minor modification of Example 5.4: set $M = p = 5$ since `corrmtx` generates $M + 1$ dimensions. The summation is now unnecessary, and should be eliminated. Finally, change the `pmusic` routine in Example 5.4 to `peig`.
- 5.19 This problem uses the *signal subspace* to evaluate the spectrum of two narrowband signals in noise. Modify Example 5.4 to average over the signal subspace; that is, take the sum from 1 to p instead of $p + 1$ to M . Also put the summation in the numerator since we are dealing with the signal subspace. As in Example 5.4, use the signal \mathbf{x} in `narrowband_signals.mat` and compare the signal subspace spectrum with that obtained using `pmusic` (which uses the noise subspace) and plot both spectra side by side for easy comparison. Note that the spectra are similar, but that the peaks are not as sharp when the signal subspace is used.
- 5.20 Load the file `signal_analysis5.mat` containing signal vector \mathbf{x} that consists of two closely spaced signals at 200 and 220 Hz ($f_s = 1 \text{ kHz}$). The SNR is fairly high (-5 dB), but the data segment is short, $N = 128$. Compare the spectrum generated by the Fourier transform, AR, and eigenvector methods. Find the best values for AR model order and eigenvalue signal subspace. Use the modified covariance AR method and the MUSIC method.
- 5.21 This problem compares the Pisarenko and MUSIC eigen decomposition methods. Load the file used in Problem 5.20 and compare the spectra generated by MATLAB's `peig` and `pmusic`. Use a signal subspace of 7 and 11. Note that the two methods produce very similar results and that a signal subspace of 7 is not enough to identify the two closely spaced peaks. Theoretically, a signal subspace dimension of 4 should be sufficient to separate out two sinusoids, but the presence of noise increases the number of peaks required.
- 5.22 This problem compares power spectrum estimation using the Welsh, AR, and eigenvector methods applied to a short signal. The file `signal_analysis6.mat` contains signal vector \mathbf{x} with two closely spaced sinusoids at 300 and 330 Hz ($f_s = 1 \text{ kHz}$). The SNR is fairly high (-3 dB) but the signal is quite short ($N = 64$). Find the power spectra using the Fourier transform, the AR (Burg), and the MUSIC eigen decomposition method. Use an AR model order and eigenvector signal space that shows the two narrowband peaks clearly. Note that the MUSIC method shows the two peaks with the least amount of background noise.

Biosignal and Medical Image Processing

- 5.23 This problem examines the ability of the Fourier transform, AR, and eigen decomposition methods to represent broadband spectra. Construct a broadband spectrum by passing white noise through a filter. Generate an IIR bandpass filter with cutoff frequencies of 20 and 80 Hz (use `butter`). Assume $f_s = 200$ Hz, and use a 12th-order Butterworth filter. Confirm the filter's spectrum directly from the coefficients. (Note you will have to take the FT of both numerator and denominator coefficients and divide, or use `freqz`.) Next, compare the spectrum produced by the three methods as in Example 5.7. For the Welch method, use a window size of 64 with maximal overlap. Use an AR model order of 30 and eigenvalue signal subspace of 20. Use signal lengths of 256 and 4000 points. Note how the Welch spectrum improves with a longer signal (because more segments are averaged), but the other two methods still represent the broadband signal as a series of peaks.
- 5.24 This problem directly examines the dependency of AR and MUSIC spectral analysis methods on signal length. Load the file `spectral_analysis7.mat`, which contains two signal vectors, `x` and `x1`. Both signals contain four sinusoids in 12 dB of noise, but `x` has 1024 samples and `x1` is a subset containing 126 samples. Sinusoidal frequencies are 100, 240, 280, and 400 Hz ($f_s = 1$ kHz). Calculate the spectrum using the AR covariance method with a model order of 26, and the MUSIC method with a signal subspace of 15. These parameters generate spectra that identify the four sinusoids in the longer signal, `x`, without generating excessive spurious peaks. Now apply these two methods to the shorter signal `x1`. Note that both methods miss one of the two closely spaced peaks. Now increase the two parameters so that both closely spaced peaks are identified. Note that spurious peaks are now present in both spectra. With the short signal and this amount of noise, it is not possible to identify all the peaks and eliminate spurious peaks.
- 5.25 This problem directly examines the dependency of AR and MUSIC spectral analysis methods on noise level. Load the file `spectral_analysis8.mat`, which contains two signal vectors, `x` and `x1`. Both signals have 256 samples and contain four sinusoids, but `x` has an SNR of 0 dB and `x1` has an SNR of -12 dB. Sinusoidal frequencies are 100, 240, 280, and 400 Hz. Calculate the spectrum using the AR covariance method with a model order of 26 and the MUSIC method with a signal subspace of 11. These parameters generate spectra that identify the four sinusoids in the low-noise signal without generating excessive spurious peaks. Now apply these two methods to the noisier signal `x1`. Note that both methods miss one of the two closely spaced peaks and may have spurious peaks. Try changing two parameters so that both closely spaced peaks are identified. Note that with the high-noise signal, it is simply not possible to identify all four peaks.
- 5.26 This problem applies the MUSIC eigen decomposition method to multiple observations of a signal as might occur in EEG analysis. Load the file `ver.mat` containing matrix variable `ver`, which consists of 100 observations of the visual-evoked response ($f_s = 100$ Hz). These signals are arranged as rows. Use the MUSIC method with a signal subspace between 9 and 15 to find the power spectrum of the visual-evoked response two ways: first, take the mean of the signals and use that average signal vector as the input signal to `pmusic`, and second, use the matrix directly as the input to `pmusic` so that the eigen decomposition is applied directly to the signal matrix and not a correlation matrix. Plot only the first 20 low-frequency components and compare the two spectra. Also plot the average signal in the time domain.
- 5.27 In Problem 5.26, it did not matter if the mean of the visual-evoked response or the signal matrix is used, because the individual-evoked responses were time-locked. In

this problem, the signals, two sinusoids, have different phases in each observation. As in Problem 5.26, the MUSIC eigen decomposition method is applied to multiple observations of a signal. Load the file `multi_observations.mat` containing matrix variable `x`, which consists of 100 observations of two sinusoids at 20 and 70 Hz in noise ($f_s = 200$ Hz). These signals are arranged as rows. Use the MUSIC method to find the spectrum of the signal two ways as in Problem 5.26: first, take the mean of the signals as input to `pmusic`, and second, input the matrix directly to `pmusic` so that the eigen decomposition is applied directly to the signal matrix. Plot all the frequency components and compare the two spectra. Note that they are not the same, and that only the spectra obtained using the signal matrix correctly show the peaks at 20 and 70 Hz.

6

Time–Frequency Analysis

6.1 Basic Approaches

The spectral analysis techniques developed so far represent powerful signal-processing tools but are not very useful if you are interested in the timing of particular events in the signal. The Fourier transform of a musical passage would tell us what notes are played, but it is extremely difficult to figure out when they are played (Hubbard, 1998). Such information must be embedded in the spectrum, since the Fourier transform is bilateral and the musical passage can be uniquely reconstructed using the inverse Fourier transform. However, timing is encoded in the phase portion of the transform and this encoding is very difficult to interpret and recover. In the Fourier transform, specific events in time are distributed across all the phase components: a local feature in time is transformed into a global feature in phase.

Classical and modern spectral methods also assume that waveforms are stationary, that is, the waveforms do not change their basic properties (their statistical properties) over the analysis period. Yet, many waveforms, particularly those of biological origin, are not stationary and change substantially over time. For example, an EEG changes radically depending on the internal mental states of the subject: if the subject is meditating, various stages of sleep, or with eyes open or closed. Frequently, these changes over time are of greatest interest. A wide range of approaches have been developed to extract both time and frequency information from a waveform. Basically, they can be divided into two groups: time–frequency methods and timescale methods. The latter are better known as *wavelet* analyses, a popular approach described in the next chapter. This chapter is dedicated to time–frequency methods.

6.2 The Short-Term Fourier Transform: The Spectrogram

The first time–frequency methods were based on the straightforward approach of slicing the waveform of interest into a number of short segments and performing an analysis on each of these segments, normally using the standard Fourier transform. A window function similar to those described in Chapter 3 is applied to a segment of data, effectively isolating that segment from the overall waveform and the Fourier transform is applied to that segment. This is termed the *spectrogram* or *short-term Fourier transform* (STFT)* since the Fourier transform is applied

* The terms STFT and spectrogram are used interchangeably but we tend to favor the use of STFT. The reader should be familiar with both terms.

Biosignal and Medical Image Processing

to a segment of data that is shorter, often much shorter, than the overall waveform. Since abbreviated data segments are used, selecting the most appropriate window length can be critical. Since the segments are short, a window function (other than a rectangular window) is usually applied. The STFT has been successfully applied in a number of biomedical applications.

The basic equation for the spectrogram in the continuous domain is

$$X(t, f) = \int_{-\infty}^{\infty} x(t)w(t - \tau)e^{-j2\pi mft} dt \quad (6.1)$$

where $w(t - \tau)$ is the window function, τ is the variable that slides the window across the waveform, $x(t)$, and $e^{-j2\pi mft}$ is the complex sinusoidal term that probes the signal in the frequency domain. The discrete version of Equation 6.1 is the same as Equation 2.43 in Chapter 2, in which the probing function, $f_m[n]$, in that equation is replaced by the family of complex sinusoids, that is, $e^{-jnm/N}$.

$$x[m, k] = \sum_{n=1}^{N} x[n](w[n - k]e^{-jnm/N}) \quad (6.2)$$

There is one major problem with the spectrogram: selecting the best window length for data segments that contain several different features may not be possible. This problem arises from the inherent time–frequency trade-off; shortening the data segment length, N , to improve time resolution will reduce frequency resolution that is approximately $1/(NT)$. Shortening the data segment can also result in the loss of low frequencies that are no longer fully included in the segment. Hence, if the window is made smaller to improve the time resolution, then the frequency resolution is degraded and vice versa. This time–frequency trade-off has been equated to the famous uncertainty principle in physics. With the STFT, the product of bandwidth, B , and time, T , must be greater than some minimum value, specifically

$$B T \geq \frac{1}{4\pi} \quad (6.3)^*$$

The trade-off between time and frequency resolution inherent in the STFT is illustrated graphically in Figure 6.1, which plots frequency resolution against time resolution for three different window sizes. The smallest time resolution of 0.25 s has the poorest frequency resolution (Figure 6.1, black rectangle). The best frequency resolution has the poorest time resolution (Figure 6.1, dashed line). These limitations have led to the development of several other time–frequency methods presented in Section 6.2 as well as the time-scale approaches discussed in Chapter 7. Despite these limitations, the STFT has been used successfully in a wide variety of problems, particularly those where only high-frequency components are of interest and frequency resolution is not critical. The area of speech processing has benefited considerably from the application of the STFT. The STFT is a simple solution that rests on a well-understood classical theory (i.e., the Fourier transform) and is easy to interpret. The strengths and weaknesses of the STFT are explored in the examples below and in the problems at the end of this chapter.

6.2.1 MATLAB Implementation of the STFT

The implementation of the time–frequency algorithms described above is straightforward and is illustrated in the examples below. The spectrogram can be generated using the standard `fft`

* The constant depends on how frequency resolution and time resolution are defined. For example, using frequency in radians, this equation can also be written as $\Delta\omega \Delta t \geq 0.5$.

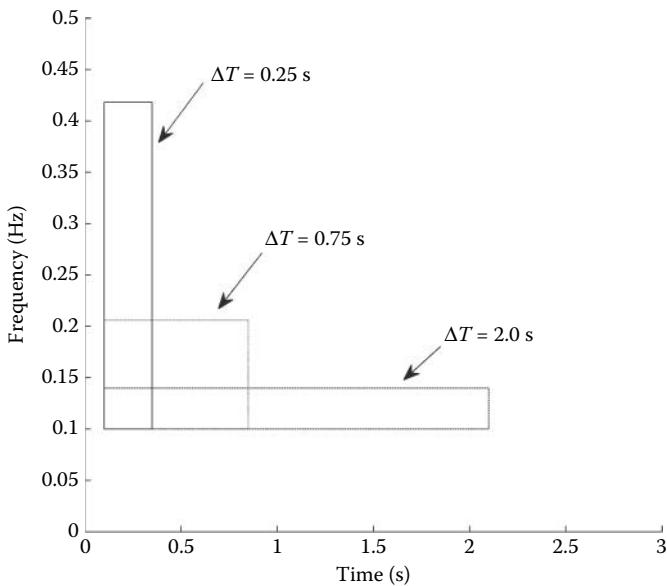


Figure 6.1 Graphical illustration of the time–frequency trade-off inherent in the STFT. The smallest time resolution gives the poorest frequency resolution (vertical rectangle) and vice versa (horizontal rectangle).

function described in Chapter 3 or using a special function of the Signal Processing Toolbox, `spectrogram`. The arguments for `spectrogram` are similar to those used for `pwelch` described in Chapter 3.

```
[B, f, t] = spectrogram(x, window, noverlap, nfft, fs, );
```

where the output, `B`, is a complex matrix containing the magnitude and phase of the STFT time–frequency spectrum, with the rows encoding the time axis and the columns representing the frequency axis. The optional output arguments, `f` and `t`, are time and frequency vectors that can be helpful in plotting. The input arguments include the data vector, `x`, and the size of the Fourier transform window, `nfft`. Three optional input arguments include the sampling frequency, `fs`, used to calculate the plotting vectors, the window function desired, and the number of overlapping points between the windows. The window function is specified as in `pwelch`. If a scalar is given, then a Hamming window of that length is used.

The output of all MATLAB-based time–frequency methods is a function of two variables, time and frequency, and requires either a three-dimensional plot (3-D) or a two-dimensional (2-D) *contour* plot. Both plotting approaches are available through MATLAB standard graphics and are illustrated in the example below.

EXAMPLE 6.1

Construct a time series consisting of two sequential sinusoids of 10 and 40 Hz, each active for 0.5 s (see Figure 6.2). The sinusoids should be preceded and followed by 0.5 s of no signal (i.e., zeros). Determine the magnitude of the STFT and plot as both a 3-D grid plot and as a contour plot. Do not use the Signal Processing Toolbox routine, but develop a code for the STFT. Use a Hamming window to isolate data segments.

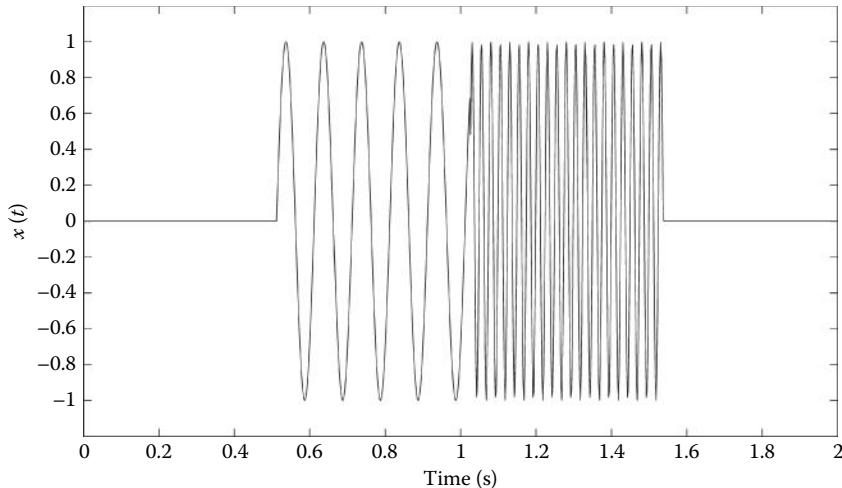


Figure 6.2 Signal consisting of two sequential sinusoids at different frequencies (10 and 40 Hz) bounded by 0.5 s of zeros. This signal is used in Example 6.1.

Solution

This example uses a function similar to MATLAB's spectrogram, except that a Hamming window is always used and all the input arguments must be specified. This function, `spectog`, has arguments similar to those in `spectrogram`. The code for this routine is given below the main program. The results are plotted both as a mesh and as a contour plot using standard MATLAB routines.

```
% Example 6.1 Example of the use of the spectrogram
%   Uses function spectog given below
%
% Set up constants
fs = 500; % Sample frequency
N = 1024; % Signal length
f1 = 10; % First frequency
f2 = 40; % Second frequency
nfft = 64; % Window size
nooverlap = 32; % Overlapping points (50%)
%
% Construct a step change in frequency
tn = (1:N/4)/fs; % Time vector for signal
x = [zeros(N/4,1); sin(2*pi*f1*tn)'; ...
      sin(2*pi*f2*tn)'; zeros(N/4,1)];
t = (1:N)/fs; % Time vector for plot
.... plot signal with labels ....
%
[B,f,t] = spectog(x,nfft,nooverlap,fs); % Calculate STFT
B = abs(B); % Spectrum magnitude
figure;
mesh(t,f,B); % Plot time-freq as 3-D mesh
.....labels and axis
figure
contour(t,f,B); % Plot time-freq as contour
.... labels and axis ....
```

6.2 The Short-Term Fourier Transform

The function `spectog` includes comprehensive comments and uses a standard MATLAB tick to ensure that the input data are arranged as a row vector. It then determines the number of samples to move the window and constructs the frequency vector based on the window size and sampling frequency. The data are zero padded at both ends to handle edge effects and a loop is used to calculate the Fourier transform at each window positions. The window time position is used to construct a time vector for use in plotting.

```

function [sp,f,t] = spectog(x,nfft,nooverlap,fs);
% function [sp,f,t] = spectog(x,nfft,nooverlap,fs);
% Function to calculate spectrogram
% Output arguments
%     sp spectrogram
%     t time vector for plotting
%     f frequency vector for plotting
% Input arguments
%     x data
%     nfft window size
%     fs sample frequency
%     nooverlap number of overlapping points
%     Uses Hanning window
%
[N xcol] = size(x);
if N < xcol
    x = x';                                % Insure that the input is a
    N = xcol;                               % row vector
end
incr = nfft - nooverlap;                  % Calculate window increment
hwin = fix(nfft/2);                      % Half window size
f = (1:hwin)*(fs/nfft);                  % Calculate frequency vector
%
% Zero pad data array to handle edge effects
x_mod = [zeros(hwin,1); x; zeros(hwin,1)];
%
j = 1;                                     % Used to index >time vector
% Calc. spectra at each position. Use Hanning window
for k = 1:incr:N
    data = x_mod(k:k + nfft - 1) .* hanning(nfft);    % Apply window
    ft = abs(fft(data));                            % Magnitude data
    sp(:, j) = ft(1:hwin);                         % Meaningful points
    t(j) = k/fs;                                  % Calculate time vector
    j = j + 1;                                    % Increment index
end

```

Figures 6.3 and 6.4 show that the STFT produces a time–frequency plot, with the step change in frequency at approximately the correct time although neither the time of the step change nor the frequencies are very precisely defined. The lack of what is called finite support in either time or frequency is particularly noticeable in the contour plot of Figure 6.4 by the appearance of energy slightly before 0.5 s and slightly after 1.5 s, and energies at frequencies other than 10 and 40 Hz. In this example, the time resolution is better than the frequency resolution. Changing the time window alters the compromise between time and frequency resolution. The exploration of the trade-off is explored in several problems at the end of this chapter.

A popular signal used to explore the behavior of time–frequency methods is a sinusoid that increases in frequency over time. This signal is called a *chirp* signal because of the sound it makes if treated as an audio signal. A sample of such a signal is shown in Figure 6.5. This signal can be generated by multiplying the argument of a sine function by a linearly increasing term as

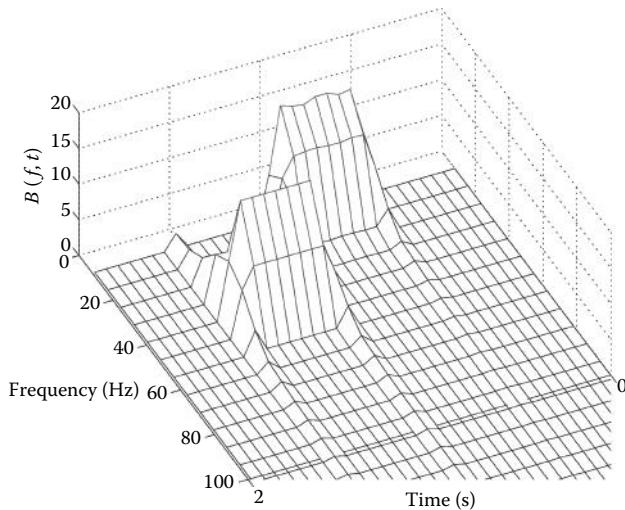


Figure 6.3 The STFT of the signal in Figure 6.2 plotted as a 3-D mesh plot.

shown in Example 6.2. Alternatively, the Signal Processing Toolbox contains a special function to generate a chirp that provides some extra features, such as logarithmic or quadratic changes in frequency. The MATLAB `chirp` routine is used in a later example. The response of the STFT to a chirp signal is demonstrated in the example below.

EXAMPLE 6.2

Generate a linearly increasing sine wave that varies between 10 and 200 Hz over a 1.0-s period. Analyze this chirp signal using the STFT program from MATLAB (i.e., `spectrogram`). Use a Hamming window (the default) and a 50% overlap (also the default). Plot the resulting spectrogram as both a 3-D grid and as a contour plot. Assume a sample frequency of 500 Hz; so, for a 1.0-s signal, $N = 500$.

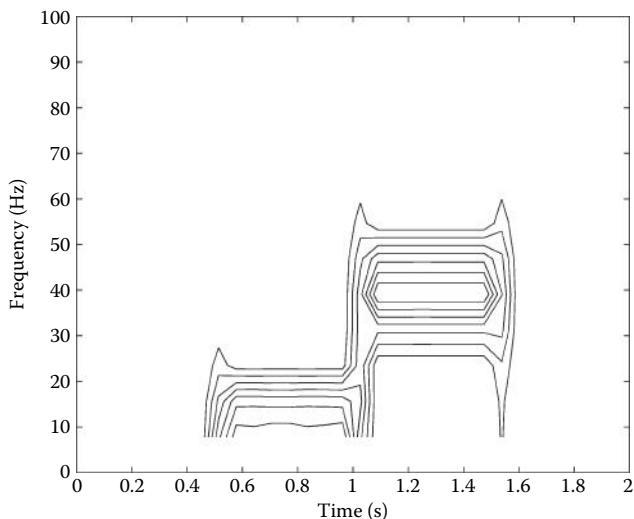


Figure 6.4 The STFT of the signal in Figure 6.2 plotted as a contour plot.

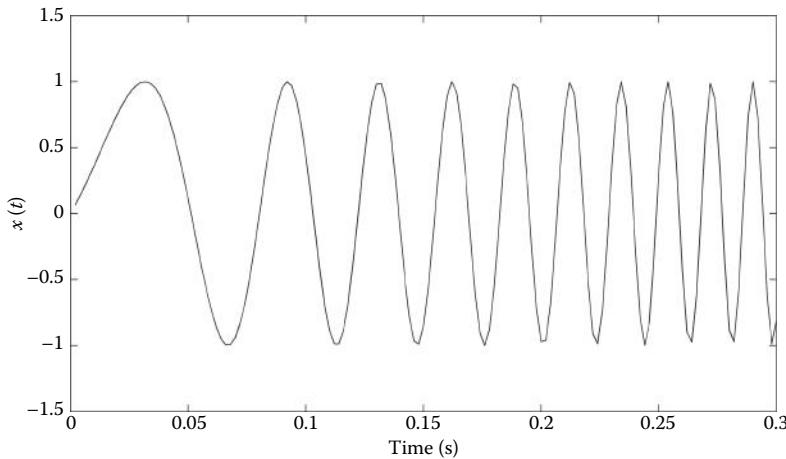


Figure 6.5 The first 0.4 s of the 1.0-s chirp signal used in Example 6.2. The continuous increase in frequency is clear.

Solution

The primary challenge of this problem is constructing the signal. To generate the chirp signal, construct a frequency vector that varies from 10 to 200 over a 500-sample interval and a time vector that varies from 0 to 1.0 over the same number of samples. Use the point-by-point product of the two vectors to generate the chirp signal, that is, $x = \sin(\pi \cdot t \cdot f_c)$. (Note that as the frequency continuously increases over time, the frequency actually generated is twice that specified by vector f_c ; so, the 2 is omitted in the sine function.) Once generated, this signal is analyzed using spectrogram and plotted.

```
% Example 6.2 Example to generate a sine wave with a linear change
% in frequency and evaluate that signal using the STFT.
%
% Constants
N = 500; % Number of samples
fs = 500; % Sample frequency
f1 = 10; % Minimum chirp frequency
f2 = 200; % Maximum chirp frequency
nfft = 32; % Window size
t = (1:N)/fs; % Time vector for chirp
fc = ((1:N)*((f2-f1)/N)) + f1; % Frequency vector for chirp
x = sin(pi*t.*fc);
%
% ..... plot the chirp and initiate new figure
% Compute spectrogram. Default is Hamming window
[B,f,t] = spectrogram(x,nfft,[],nfft,fs);
%
mesh(t,f,abs(B)); % 3-D plot
% .... labels, axis, title, and subplot ....
contour(t,f,abs(B)); % Contour plot
% .... labels, axis, and title ....
```

Results

Figure 6.6a presents a 3-D plot of the STFT of the chirp signal and shows the expected continuous increase in peak frequency as time increases from 0 to 1.0 s. The contour plot in Figure 6.6 shows that the peak spectral values follow the expected linear progression between 10 and

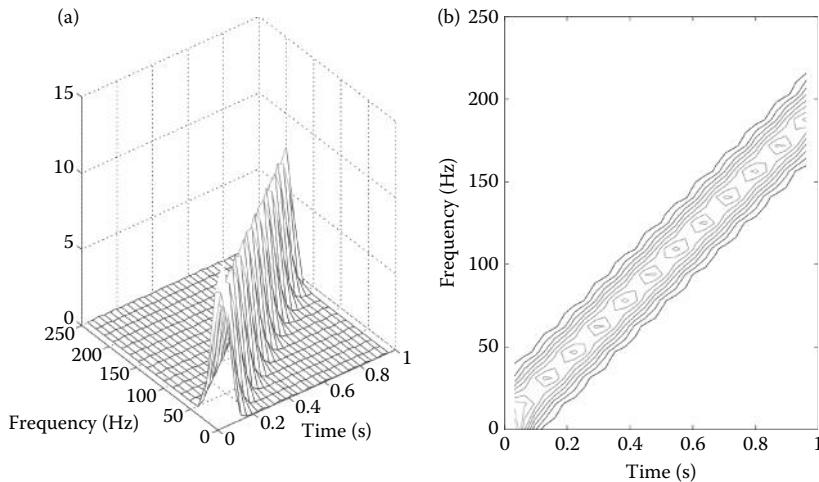


Figure 6.6 (a) 3-D plot of the STFT of a chirp signal. (b) Contour plot of the chirp signal. Both plots show the linear increase frequency over time and the energy spread over both time and frequency. Ideally, the 3-D plot should be a shaped ridge and the contour plot should be a narrow line.

200 Hz. Again, the spectrum is broadened along both time and frequency directions as predicted by the uncertainty equation (Equation 6.3). The chirp is a good signal to use to examine the time–frequency spread: the inherent trade-off between time and frequency resolution.

6.3 The Wigner–Ville Distribution: A Special Case of Cohen’s Class

A number of approaches have been developed to overcome some of the shortcomings of the spectrogram. The first of these is the *Wigner–Ville distribution*,* which is also one of the most studied and best understood of the many time–frequency methods. The approach was actually developed by Wigner for use in physics, but later applied to signal processing by Ville; hence, the dual name. We will see below that the Wigner–Ville distribution is a special case of a wide variety of similar transformations known under the heading of *Cohen’s class of distributions*.

6.3.1 The Instantaneous Autocorrelation Function

The Wigner–Ville distribution and others of Cohen’s class use an approach that harkens back to the early use of the autocorrelation function for calculating the power spectrum. As noted in Chapter 3, the classical method for determining the power spectrum is to take the Fourier transform of the autocorrelation function (Equations 3.28 and 3.29). To construct the autocorrelation function, the waveform is compared with itself for all possible relative shifts or lags. The autocorrelation equation (Equation 2.50) is repeated here in discrete form

$$r_{xx}[k] = \frac{1}{N} \sum_{n=1}^N x[n]x[n+k] \quad (6.4)$$

where k is the shift of the waveform with respect to itself and n is the time index.

* The term “distribution” in this usage should more properly be “density” since that is the equivalent statistical term (Cohen, 1990).

In the autocorrelation function, an average is taken over time (n in Equation 6.4); so, the time variable is averaged out of the result; thus, $r_{xx}[k]$, is only a function of the lag k . The Wigner–Ville, and, in fact, all Cohen’s class of distributions uses a variation of the autocorrelation function where time *remains* in the result. This is achieved by comparing the waveform with itself for all possible lags, but instead of averaging over time, the comparison is done for all possible values of time. This comparison gives rise to the *instantaneous autocorrelation* function presented in continuous form

$$R_{xx}(t, \tau) = x\left(t + \frac{\tau}{2}\right)x^*\left(t - \frac{\tau}{2}\right) \quad (6.5)$$

where t is time and τ is the lag variable.

In discrete format, Equation 6.5 becomes

$$R_{xx}[n, k] = x[n+k]x^*[k-n] \quad (6.6)$$

where n is the time variable, k is the time lag variable as in autocorrelation, and $*$ represents the complex conjugate of the signal, x . So, the instantaneous autocorrelation function becomes a function of two variables: the lag variable, k , and the time variable, n . Although actual signals are real, the instantaneous autocorrelation (Equation 6.6) is often applied to a complex version of the real signal known as the analytic signal discussed below. Note that the continuous equation (Equation 6.5) calls for fractional lag value, $\tau/2$, but the discrete version must use integer lags (Equation 6.6). Since the lag values in the discrete version are double those of the continuous equation, the effective sampling frequency of the discrete equation is reduced by a factor of 2 when instantaneous autocorrelation is used in frequency analysis.

EXAMPLE 6.3

Compute the instantaneous autocorrelation function of a 2-Hz sine wave. Make $N = 500$ pts and the signal sampling frequency $f_s = 500$ Hz. Plot the result as both a 3-D mesh plot and a contour plot.

Solution

After the data are generated, the instantaneous autocorrelation function is evaluated at every time value, n , using a loop. At each value of n , we implement Equation 6.5 over a range of positive and negative shifts k , up to some maximum shift value. We set the maximum shift to half the data length, $N/2$, but to deal with edge effects, we limit the shift to the available data. (Alternatively, we could zero pad the data.) We also construct a lag vector to be used in plotting. As requested, the instantaneous autocorrelation function is plotted as both a 3-D mesh plot and a contour plot.

```
% Example 6.3 Compute and plot the instantaneous autocorrelation function
% of a 2 Hz sine wave.
%
N = 500; % Number of points
fs = 500; % Sampling frequency
t = (1:N)/fs; % Time vector
f1 = 2; % Signal frequency
x = sin(2*pi*f1*t)'; % Construct signal as row vector
N_2 = N/2; % Half data length
%
Rxx = zeros(N,N); % Inst. auto. output array
% Compute instantaneous autocorrelation: Eq. 6.5
```

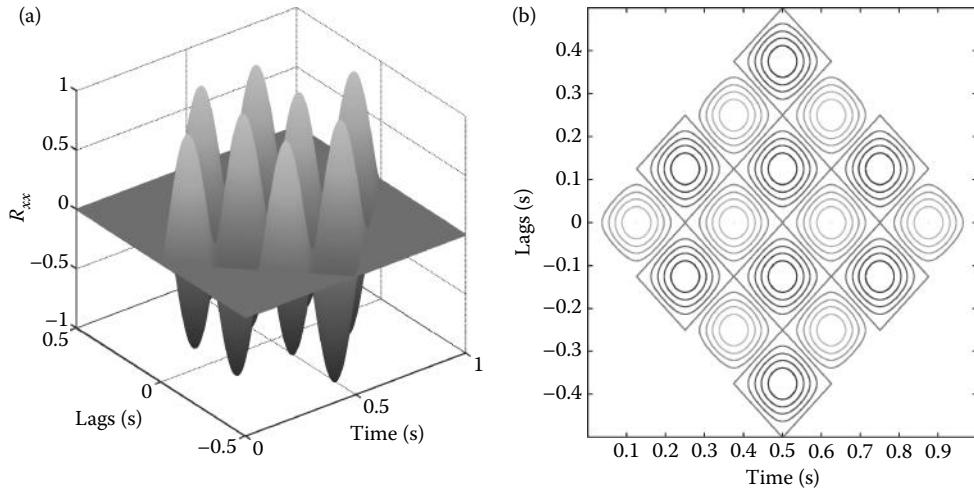


Figure 6.7 (a) The instantaneous autocorrelation function of a 2-Hz sine wave plotted in 3-D. (b) The instantaneous autocorrelation function plotted as a contour plot. The sinusoids along the diagonal are due to cross-products from the multiplications in the instantaneous autocorrelation equation (Equation 6.5).

```

for ti = 1:N % Increment over time
    taumax = min([ti-1,N-ti]); % Limit lags to available data
    tau = -taumax:taumax; % Shift tau in both directions
    Rxx(N_2 + tau,ti) = x(ti + tau) .* conj(x(ti-tau));
end
lags = (-N_2:N_2-1)/fs; % Lags for plotting
subplot(1,2,1); % 3-D plot
.....labels and colormap.....
subplot(1,2,2);
contour(t, lags, Rxx); % Contour plot
.....labels and colormap.....

```

Result

Since the instantaneous autocorrelation function retains both lag and time variables, it is a 2-D function. The output of this function to a sine wave is shown as a 3-D and contour plot in Figure 6.7. The standard autocorrelation function of a sinusoid would be a sinusoid of the same frequency; so, it is not surprising that the instantaneous autocorrelation is a sinusoid along both the time and lag axis. However, Figure 6.7 shows sinusoids along the diagonal. These are cross-products resulting from the multiplications in the instantaneous autocorrelation equation (Equation 6.5). We will find that these cross-products are a source of considerable problems for time-frequency methods based on the instantaneous autocorrelation function.

6.3.2 Time-Frequency Distributions

As mentioned above, the classical method of computing the power spectrum is to take the Fourier transform of the autocorrelation function. The Wigner-Ville distribution and others of Cohen's class echo this approach by taking the Fourier transform of the instantaneous autocorrelation function, but only along the lag (i.e., k) dimension, that is, along the vertical lines shown in Figure 6.8. The result is a function of both frequency (from the Fourier transform) and time (the remaining variable).

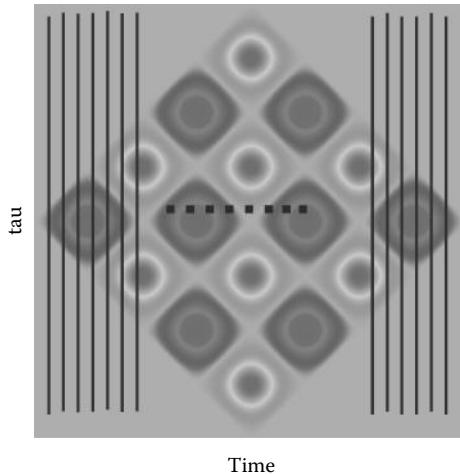


Figure 6.8 A shaded contour plot of the instantaneous autocorrelation function of a 2-Hz sine wave similar to that shown in Figure 6.7b. The Wigner–Ville and other distributions are determined by calculating the Fourier transform along the lag or τ -axis of the instantaneous autocorrelation function, that is, along the dark lines shown here. The result is a function of frequency, from the Fourier transform, and time, the remaining variable from the instantaneous autocorrelation function.

When the traditional one-dimensional (1-D) power spectrum was computed using the autocorrelation function (before the development of the fast Fourier transform), it was common to filter the autocorrelation function before taking the Fourier transform to improve the power spectrum. While the Wigner–Ville distribution does not use a filter, all the other approaches apply a filter before taking the Fourier transform. Since the instantaneous autocorrelation is 2-D, it must be a 2-D filter. In fact, the only difference between the many different distributions in Cohen’s class is simply the type of filter that is used. All the many distributions in Cohen’s class are determined by applying a 2-D filter to the instantaneous autocorrelation function, then taking the 1-D Fourier transform along the τ -axis.

The formal equation for determining a time–frequency distribution from Cohen’s class of distributions is rather formidable, but can be simplified in practice

$$C(t, f) = \iiint e^{j\pi v(u-t)} g(v, \tau) x \left(u + \frac{1}{2} \tau \right) x^* \left(u - \frac{1}{2} \tau \right) e^{-j2\pi v ft} dv du d\tau \quad (6.7)$$

where $g(v, \tau)$ provides the 2-D filtering of the instantaneous autocorrelation, also known as a *kernel*. Note that the last three terms of the integrand make up the Fourier transform of the instantaneous autocorrelation function. A more practical equation, reflecting the actual steps used to implement these distributions, is

$$C(t, f) = \text{FFT}\{G(u, \tau) \text{ conv } R_{xx}(t, \tau)\} \quad (6.8)$$

where conv represents 2-D convolution, $R_{xx}(t, \tau)$ is the instantaneous autocorrelation function (Equation 6.5), and $G(u, \tau)$ defines the 2-D filter.

For the Wigner–Ville distribution, there is no filtering and $G(u, \tau) = 1$. So, the general equation of Equation 6.7, after integration by dv , reduces to the continuous equation (Equation 6.9) or the discrete version (Equation 6.10).

$$W(t, f) = \int_{-\infty}^{\infty} x \left(t + \frac{\tau}{2} \right) x^* \left(t - \frac{\tau}{2} \right) e^{-j2\pi f t} d\tau \quad (6.9)$$

$$\begin{aligned}
 W[n,m] &= 2 \sum_{k=1}^N x[n+k]x^*[n-k] e^{-j2\pi mn/N} \\
 &= 2 \sum_{k=1}^N e^{-j2\pi mn/N} R_{xx}[n,k] = \text{FFT}[R_{xx}[n,k]]
 \end{aligned} \tag{6.10}$$

where again $R_{xx}[n,k]$ is the discrete instantaneous autocorrelation function (Equation 6.6).

The relationship between discrete and continuous variables is the same as that found previously for time, that is, $t = nT_s$. However, the relationship between frequency and harmonic number, m , differs by a factor of 2 to account for the half-lags (i.e., $\tau/2$) in Equation 6.9:

$$f = m/(2NT_s) \tag{6.11}$$

EXAMPLE 6.4

Determine and plot the Wigner–Ville distribution of the chirp signal used in Example 6.2. Also, plot the instantaneous autocorrelation function.

Solution

Generate the chirp signal using the code in Example 6.2 and then calculate the instantaneous autocorrelation function using the code in Example 6.3. Then take the Fourier transform over the lags of the instantaneous autocorrelation function. When the MATLAB `fft` routine is applied to a matrix, it takes the Fourier transform over the columns. Since the instantaneous autocorrelation code of Example 6.3 places the lag values in the matrix columns, the `fft` routine can be applied directly to the matrix produced by this code without the need for transposition.

```
% Example 6.4 Example of the use of the Wigner-Ville Distribution
%
% .....Set up chirp signal. Same code as Example 6.2,
% except fs divided in half, Eq. 6.10.....
%
% .....The following code is the same as in Example 6.3.....
N_2 = N/2; % Half data length
Rxx = zeros(N,N); % Inst. auto. output array
% Compute instantaneous autocorrelation: Eq. 6.7
for n = 1:N % Increment over time
    k_max = min([n-1,N-n]); % Limit lags to available data
    k = -k_max:k_max; % Shift tau in both directions
    Rxx(N_2 + k, n) = x(n+k) .* conj(x(n-k)); % Eq. 6.7
end
lags = (-N_2:N_2-1)/fs; % Lags (in columns) for plotting
contour(t, lags, Rxx); % Plot inst. autocorr.
%
% .....labels and figure.....
WD = abs(fft(Rxx)); % Calculate the W-V Dist.
mesh(t, f, WD); % Plot W-V Distribution
%
% .....labels, title, and figure.....
contour(t, f, WD, [25 50 75 100 125 150]); % Plot W-V as contour plot
%
% .....labels and figure.....
```

Results

The instantaneous autocorrelation is shown as a contour plot in Figure 6.9 that appears as a set of sinusoids at ever-increasing frequencies. These sinusoids appear in horizontal, vertical, and diagonal directions. Figure 6.10 shows that the Wigner–Ville distribution of a chirp signal is quite precisely defined in both time and frequency, although the amplitude is not constant

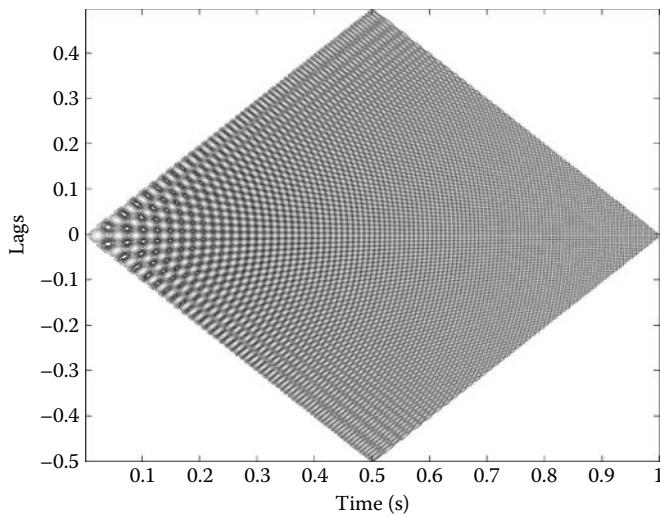


Figure 6.9 The instantaneous autocorrelation function of a chirp signal plotted as a contour plot. Sinusoids of increasing frequency are seen in the horizontal, vertical, and diagonal directions.

over time (or frequency) as it should be for this signal. This is because of the truncation of the instantaneous autocorrelation; other methods for extending this function can be used. In fact, the Wigner–Ville distribution is particularly well suited to analyzing the time–frequency characteristics of chirp signals but, as shown below, does not work as well for other signals.

While the Wigner–Ville has some advantages over the STFT, it also has a number of shortcomings. Its greatest strength is that it produces “a remarkably good picture of the time–frequency structure” (Cohen, 1992). It also has favorable *marginals* and *conditional moments*. The marginals relate the summation over time or frequency to the signal energy at that time or frequency. For example, if we sum the Wigner–Ville distribution over frequency at a fixed time,

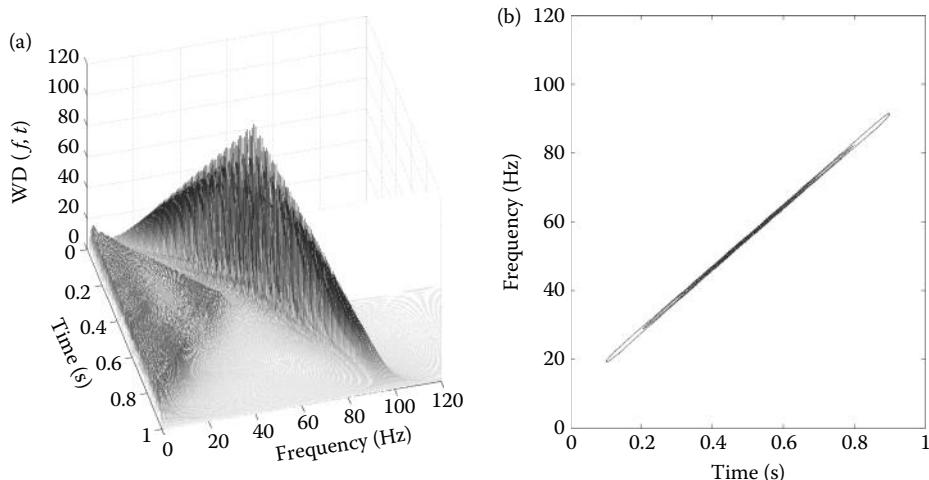


Figure 6.10 The Wigner–Ville time–frequency distribution of a chirp signal increasing in frequency from 20 to 100 Hz over a 1.0-s period. (a) 3-D plot. (b) Contour plot.

Biosignal and Medical Image Processing

we get a value equal to the energy at that point in time. Alternatively, if we fix frequency and sum over time, the value is equal to the energy at that frequency.

The Wigner–Ville distribution has a number of other properties that may be of value in certain applications. It is possible to recover the original signal, except for a constant, from the distribution and the transformation is invariant to shifts in time and frequency. For example, shifting the signal in time by a delay of T seconds produces the same distribution except when it is shifted by T on the time axis. The same can be said of a frequency shift, although biological processes that produce shifts in frequency are not as common as those that produce time shifts. These characteristics are also true for the STFT and some of the other distributions described below.

A property of the Wigner–Ville distribution not shared by the STFT is *finite support* in time and frequency. Finite support in time means that the distribution is zero before the signal starts and after it ends. Finite support in frequency means the distribution does not contain frequencies above or below those in the input signal. However, the Wigner–Ville may contain nonexistent energies due to the cross-products, as mentioned above and observed in Figure 6.8, but these are contained within the time and frequency boundaries of the original signal. Owing to these cross-products, the Wigner–Ville distribution is not necessarily zero whenever the signal is zero, a property termed by Cohen as *strong finite support*. Figure 6.11 plots the Wigner–Ville distribution of the two-frequency sine wave used in Example 6.1. While the two frequencies, 10 and 40 Hz, are well defined in both time and frequency, there is a large amount of energy between the two frequencies. This additional energy is due to the cross-products and has greater magnitude than the two real frequencies. These phantom energies have been the prime motivator for the development of other distributions that apply various filters to the instantaneous autocorrelation function to mitigate the damage done by the cross-products.

In addition, the Wigner–Ville distribution can generate negative regions that have no meaning and the distribution also has poor noise properties. Essentially, the noise is distributed across time and frequency along with the cross-products of the noise, although in some cases, the cross-products and noise influences can be reduced by using a window. In such cases, the

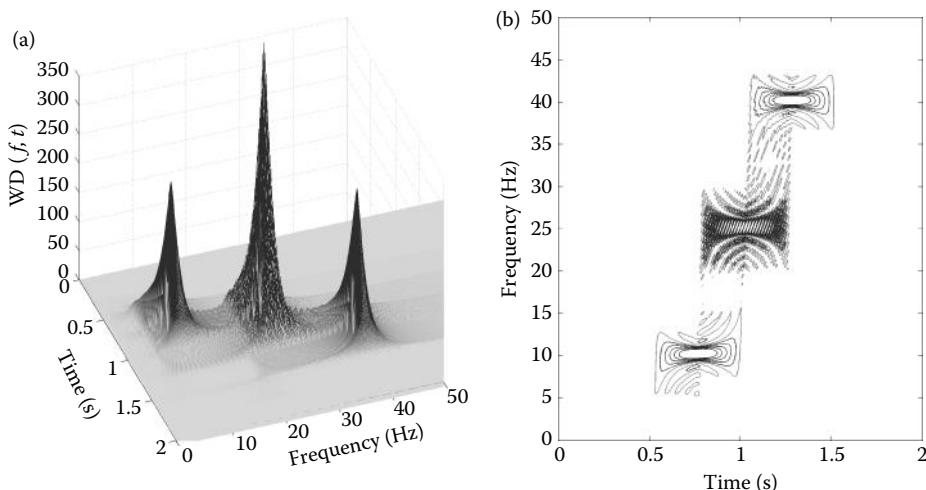


Figure 6.11 The Wigner–Ville distribution of the signal shown in Figure 6.2 consisting of two sequential sinusoids at different frequencies (10 and 40 Hz) bounded by 0.5 s of zeros. While the two sinusoids are well defined in time and frequency, there is a large cross-product that shows energy between the two frequencies (and timing) that is not present in the signal. (a) 3-D plot. (b) Contour plot. This distribution was calculated using the analytic signal described in Section 6.3.3.

desired window function is applied to the lag dimension of the instantaneous autocorrelation function similar to the way it was applied to the time function in Chapter 3. As in Fourier transform analysis, windowing reduces frequency resolution and, in practice, a compromise is sought between a reduction of cross-products and loss of frequency resolution. Noise properties and the other weaknesses of the Wigner–Ville distribution, along with the influences of windowing, are explored in the problems.

6.3.3 The Analytic Signal

All the transformations in Cohen’s class of distributions produce better results when applied to a modified version of the signal termed the *analytic signal*. The analytic signal has the same Fourier transform as the original, but without negative frequencies. Negative frequencies are equivalent to the redundant frequencies above $f_s/2$, which are mirror reflections of the components below $f_s/2$. They are nonphysical artifacts of the Fourier transform and we should be able to eliminate them without loss of signal information.

While the real signal can always be used in any distribution, the analytic signal has several advantages. Since both the positive and negative spectral terms produce cross-products, using a signal in which the negative frequency components are eliminated will reduce these cross-products. Another benefit is that if the analytic signal is used, the sampling rate can be reduced. As noted above, the discrete form of the instantaneous autocorrelation function is calculated using evenly spaced lags; so, in any frequency analysis that uses this function, the signal is effectively undersampled by a factor of 2 (see Equation 6.11). Thus, if the analytic function is not used, the data must be sampled at twice the normal minimum, that is, twice the Nyquist frequency or four times f_{MAX}^* .

Several approaches can be used to construct the analytic signal. We want to construct a signal that produces the same Fourier transform as the original but without the negative frequencies. These negative frequencies are reflections of those above $f_s/2$. Logically, we should be able to take the Fourier transform of our signal, set frequency components above $f_s/2$ to zero, and take the inverse Fourier transform of this modified spectrum. In fact, that is just what is done, with the added minor step of multiplying the remaining components (those below $f_s/2$) by 2 to keep the overall energy the same. The resulting signal is now complex, but that does not present any problems as the necessary MATLAB routines work just as well on complex signals. This approach is used by MATLAB’s *hilbert* routine.

Constructing a signal that eliminates the frequency components over $f_s/2$ can also be done using the *Hilbert transform*. In this approach, the new signal has a real part that is the same as the original signal and an imaginary part that is the Hilbert transform of the real signal

$$Z[n] = x[n] + j, \mathcal{H}\{x[n]\} \quad (6.12)$$

where \mathcal{H} denotes the Hilbert transform, which can be implemented as an FIR filter (Chapter 4) with coefficients of

$$h[n] = \begin{cases} \frac{2 \sin 2[\pi n/2]}{\pi n} & \text{for } n \neq 0 \\ 0 & \text{for } n = 0 \end{cases} \quad (6.13)$$

Although the Hilbert transform filter should have an infinite impulse response length (i.e., an infinite number of coefficients), in practice, an FIR filter length of approximately 79 samples has been shown to provide an adequate approximation (Bobashash and Black, 1987).

* If the waveform has already been sampled, the number of data points can be doubled with intervening points added using interpolation before the instantaneous autocorrelation is taken.

6.4 Cohen's Class Distributions

The existence of cross-products in the Wigner–Ville transformation has motived the development of other distributions. These other distributions apply a 2-D filter to the instantaneous autocorrelation function before taking the Fourier transform in Equation 6.8.* The use of 2-D filtering is common in image processing (see Chapter 13) and can be implemented using the MATLAB routine conv2. The filter function, referred to in these applications as the *determining function*, can be obtained from the kernel function, $g(v,\tau)$, in Equation 6.7, by integration over the variable v :

$$G(u,\tau) = \int_{-\infty}^{\infty} g(v,\tau) e^{j\pi vu} dv \quad (6.14)$$

In all of Cohen's class of distributions, the integrated function is used to filter the instantaneous autocorrelation function before taking the Fourier transform.

6.4.1 The Choi–Williams Distribution

One popular distribution is the *Choi–Williams*, which is also referred to as an *exponential distribution* since it has an exponential-type kernel. After integrating using the equation above, the Choi–Williams determining function becomes

$$G(u,\tau) = e^{-\alpha(u\tau)^2} \quad (6.15)$$

where u is another dummy time variable and α is a constant. The effect of α is to modify the rate of off-axis decrease in the filter coefficient and its influence on the resulting distribution is examined in the problems. The Choi–Williams distribution can also be used in a modified form that incorporates a window function and in this form is considered one of a class of *reduced interference distributions* (RID) (Williams, 1992). In addition to having reduced cross-products, the Choi–Williams distribution also has better noise characteristics than the Wigner–Ville distribution.

To implement the Choi–Williams distribution, we will construct a MATLAB routine that computes $G(u,\tau)$. This routine will take a convergence factor, α , and the filter size (given as the number of points along one side) as an input and produce the filter function as an output. The convergence factor describes how quickly the filter amplitude falls off lowpass. Since this and the other filters we use are symmetric about their center point, only one quadrant of the filter function is calculated and the rest is generated by flipping the array about the two axes. The filter function is implemented through a direct application of Equation 6.15.

```
function G = choi_williams(alpha,L)
% Function to calculate the Choi-Williams determining function
%
N = round(L/2); % Number of points in each quadrant
for u = 1:N
    for tau = 1:N
        G(u,tau) = exp(-alpha*((u)*(tau))^2); % Eq. 6.15
    end
end
```

* Another approach to implementing Cohen's class of distributions uses the *ambiguity function*. This function is the Fourier transform of the instantaneous autocorrelation function, but along the time axis, not the τ -axis as is used to calculate the Wigner–Ville distribution. While the theory behind Cohen's class of distribution uses the ambiguity function, implementation using 2-D convolution is easier to comprehend.

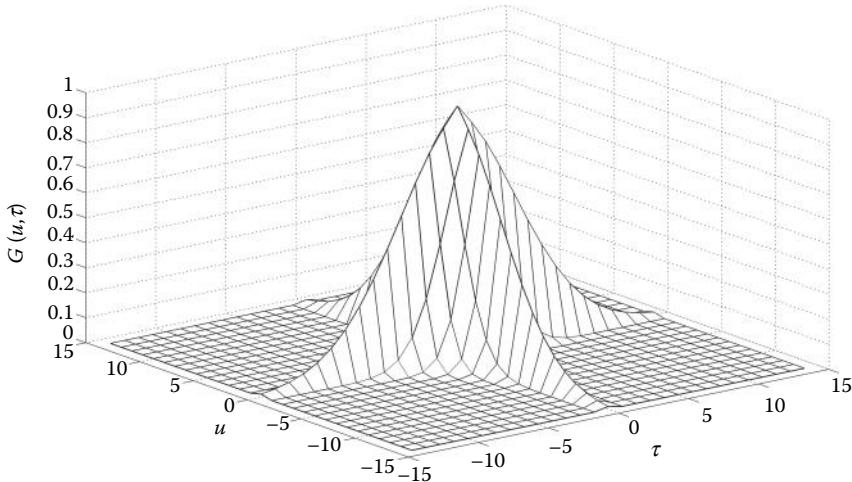


Figure 6.12 The 2-D filter of the determining function used to produce the Choi–Williams distribution ($\alpha = 0.01$). It has the general shape of a lowpass filter.

```
% Expand to 4 quadrants
G = [fliplr(G(:,2:end)) G]; % Add 2nd quadrant
G = [flipud(G(2:end,:)); G]; % Add 3rd and 4th quadrants
```

A plot of the filter coefficients produced by this routine is shown in Figure 6.12. As expected from the exponential filter equation, the filter coefficients decrease rapidly as they become more distant from the two axes. As with a 1-D filter (Chapter 4), this has the characteristic of a low-pass filter.

Since all the distributions in Cohen's class require the instantaneous autocorrelation function, we construct a routine using the relevant code in the last two examples. The calling format is

```
[Rxx,t,lags] = int_autocorr(x,fs); % Instantaneous autocorrelation
```

where the inputs are x , the signal, and fs , the sampling frequency; and the outputs are Rxx , the instantaneous autocorrelation function, and the two vectors useful in plotting, t , the time vector and $lags$, the lag vector. These two routines are employed in the next example.

EXAMPLE 6.5

Apply the Choi–William's distribution to the sequential sinusoid signal used in Example 6.1, except make the two sinusoidal frequencies 40 and 100 Hz ($f_s = 500$ Hz). Use the analytic signal. Plot the distribution as a mesh plot.

Solution

After generating the signal, use MATLAB's `hilbert` to generate the analytic signal. Construct the instantaneous autocorrelation function using `int _ autocorr` and the Choi–Williams filter using `choi _ williams`. Confirm that the filter is in the proper orientation before applying it to the instantaneous autocorrelation using `conv2`. Since the filter generates the τ -axis in the columns, as does the instantaneous autocorrelation routine, no transposition is required. Just as in 1-D convolution, the MATLAB 2-D convolution routine should be used with the option “same” to avoid generating additional points. After convolution, the Fourier transform is taken along the lags dimension as in Example 6.4 and a mesh plot of the result is produced.

Biosignal and Medical Image Processing

```
% Example 6.5 Application of the Choi-Williams distribution
%
alpha = 0.3; % Choi-Williams filter constant
L = 30; % Determining function size
. ....construct signal as in Example 6.1. .....
x = hilbert(x); % Get Analytic function
t = (1:N)/fs; % Calculate time and frequency vectors
f = (1:N) * (fs/(2*N));
%
Rxx = int_autocorr(x,fs); % Instantaneous autocorrelation
G = choi_williams(alpha,L); % Choi-Williams determining function
CD1 = conv2(Rxx,G,'same'); % 2-D convolution
CD = abs(fft(CD1)); % 1-D Fourier transform
```

Results

The 3-D plot of the sequential sinusoidal signal is shown in Figure 6.13. Compared with the Wigner–Ville distribution from this same waveform shown in Figure 6.11, the cross-product term is considerably reduced.

A number of other distributions have been developed,* but they all use the same basis strategy summarized in Equation 6.8, the only difference being the specific determining function, $G(u,\tau)$. One very simple determining function was developed by Born–Jorden–Cohen (BJC). The determining function for this filter is

$$G(u,\tau) = \begin{cases} \frac{1}{\tau+1} & \tau < u \\ 0 & \text{Otherwise} \end{cases} \quad (6.16)$$

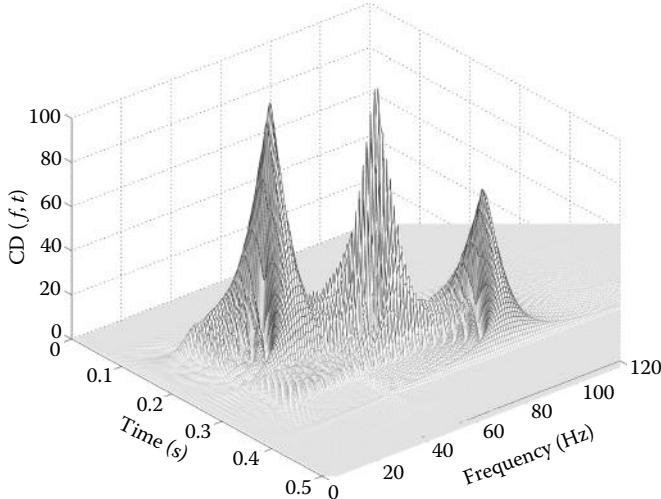


Figure 6.13 The Choi–Williams distribution for two sequential sinusoids at 40 and 100 Hz. This distribution shows a modest reduction in the central cross-product compared to the Wigner–Ville distribution of this waveform shown in Figure 6.11.

* For an extensive listing of these distributions, see the classical book by Cohen: Cohen, L., *Time–Frequency Analysis*, Prentice-Hall, New York, NY, 1995, ISBN: 978-0135945322.

7.2 Continuous Wavelet Transform

where t_0 is the center time, or first moment of the wavelet, and is given by

$$t_0 = \frac{\int_{-\infty}^{\infty} t |\Psi(t/a)|^2 dt}{\int_{-\infty}^{\infty} |\Psi(t/a)|^2 dt} \quad (7.8)$$

Similarly, the frequency range, $\Delta\omega_\Psi$, is given by

$$\Delta\omega_\Psi = \sqrt{\frac{\int_{-\infty}^{\infty} (\omega - \omega_0)^2 |\Psi(\omega)|^2 d\omega}{\int_{-\infty}^{\infty} |\Psi(\omega)|^2 d\omega}} \quad (7.9)$$

where $\Psi(\omega)$ is the frequency-domain representation (i.e., Fourier transform) of $\Psi(t/a)$, and ω_0 is the center frequency of $\Psi(\omega)$. The center frequency is given by an equation similar to Equation 7.8:

$$\omega_0 = \frac{\int_{-\infty}^{\infty} \omega |\Psi(\omega)|^2 d\omega}{\int_{-\infty}^{\infty} |\Psi(\omega)|^2 d\omega} \quad (7.10)$$

The time and frequency ranges of a given family can be obtained from the mother wavelet using Equations 7.7 and 7.9. Dilation by the variable a changes the time range simply by multiplying Δt by a . Accordingly, the time range of $\Psi_{a,0}$ is defined as $\Delta t(a) = |a|\Delta t_\Psi$. The inverse relationship between time and frequency is shown in Figure 7.3, which was obtained by applying Equations 7.7 through 7.10 to the Mexican Hat wavelet. (The code for this is given in Example 7.2.) The Mexican Hat wavelet is the second derivative of the popular Gaussian function and is given by the equation

$$\Psi(t) = (1 - 2t^2)e^{-t^2} \quad (7.11)$$

If we multiply the frequency range by the time range given by the equations above, the as cancel and we are left with a constant that depends on the specific wavelet:

$$\Delta\omega_\Psi(a)\Delta t_\Psi(a) = \Delta\omega_\Psi\Delta t_\Psi = \text{constant}_\Psi \geq 0.5 \quad (7.12)*$$

It is important to note that the CWT is subject to exactly the same restrictions on time–frequency resolution as the STFT: specifically the limitations given in Equation 7.12. As with the STFT, the resolution limits are inversely related: increasing the frequency range, $\Delta\omega_\Psi$, decreases the time range, Δt_Ψ . Since the time and frequency resolutions are inversely related, the CWT will provide better frequency resolution when a is large and the length of the wavelet (and its effective time window) is long. Conversely, when a is small, the wavelet is short and the time resolution is maximum, but the wavelet only responds to high-frequency components. The major

* This equation indicates that the product of the time and frequency ranges is invariant to dilation, a . In addition, translations, changes in b , do alter either the time or the frequency resolution; hence, the product of time and frequency resolution is also independent of the value of b .

This function is implemented in the routine BJC. Another common determining function implemented in routine BJD is again attributed to Born–Jorden:

$$G(u, \tau) = \frac{\sin(u\tau/2)}{u\tau/2} \quad (6.17)$$

EXAMPLE 6.6

Compare the Choi–Williams and BJC distributions of both the simultaneous sinusoids used in Examples 6.1 and 6.5 and the chirp signal used in Examples 6.2 and 6.4. For the Choi–Williams determining function, use an α of 0.5.

Solution

To facilitate this and other distribution comparisons, we will construct a new routine, `cohen.m`, by modifying the code in the previous example. This modification will allow for easy selection of the various determining functions.

```
function [CD,f,t] = cohen(x,fs,type,L)
    .....comments and make x row vector.....
alpha = 0.5;                                % Choi-Williams constant
t = (1:N)/fs;                                % Calculate time and frequency vectors
f = (1:N) * (fs/(2*N));
%
CD = int_autocorr(x);                      % Instantaneous autocorrelation
if type == 'c'                                % Get appropriate determining function
    G = choi_williams(alpha,L);               % Choi-Williams
elseif type = 'b'
    G = BJC(L);                            % Born-Jorden-Cohen
elseif type(1) == 'J'
    G = BJD(L);                            % Born-Jorden
elseif type(1) == 'W'
    G = 1;                                 % Wigner-Ville
else
    disp('ERROR No filter specified')
    return
end
CD = conv2(CD,G,'same');                     % 2-D convolution
CD = abs(fft(CD));                           % 1-D Fourier transform (output)
```

The solution makes use of another general routine to produce the signals. This routine combines the signals used in the examples cited above along with two other useful signals used in the problem set. The calling structure of this routine is

```
x = tf_signals(type);                      % Generate a signal for test.
```

where `x` is the signal and `type` indicates the signal type; '`ss`' specifies the sequential sinusoids used in Example 6.5; '`ch`' specifies a chirp signal such as that used in Examples 6.2 and 6.4; '`s2`' specifies two simultaneous sinusoids, but one is only for the central period; and '`ms`' specifies a sinusoid that changes in frequency sinusoidally (i.e., sinusoidal frequency modulation). For this frequency-modulated signal, the base frequency is 70 Hz and the modulation frequency is 2 Hz. This signal produces 512 samples and $f_s = 500$ Hz.

The rest of the solution is just an amalgamation of the previous examples. The program will be set up to generate either signal and select either of the two distribution functions.

Biosignal and Medical Image Processing

```
% Example 6.6 Cohen class distributions applied to both
% sequential sinusoids and a chirp signal
%
fs = 500; % Sample frequency assumed by tf_signal
L = 30; % Determining function size
%
signal_type = input('Signal type: (ss = sines; ch = chirp)', 's');
x = tf_signal(signal_type); % Construct signal
%
% Get desired distribution
type = input('Enter type (c choi-williams; b = BJC):', 's');
x = hilbert(x); % Analytic function
[CD,f,t] = cohen(x,fs,type,0.5); % Cohen's class of Transformations
.....plot and label.....
```

Results

Figure 6.14 shows the Choi–Williams and BJC distributions determined from the sequential sinusoids signal. The BJC shows smaller cross-products than the Choi–Williams approach, at least for the value of α used with the Choi–Williams determining function. (The effect of this parameter on cross-products is explored in Problem 6.12.) However, the Choi–Williams produces a much better representation of the chirp signal (Figure 6.15).

The response produced by these and other determining functions to other signals is explored in the problem set.

6.5 Summary

Signals that show interesting variations in their properties over time are common in biology and medicine. This chapter presented two different approaches to defining the spectral changes with time.

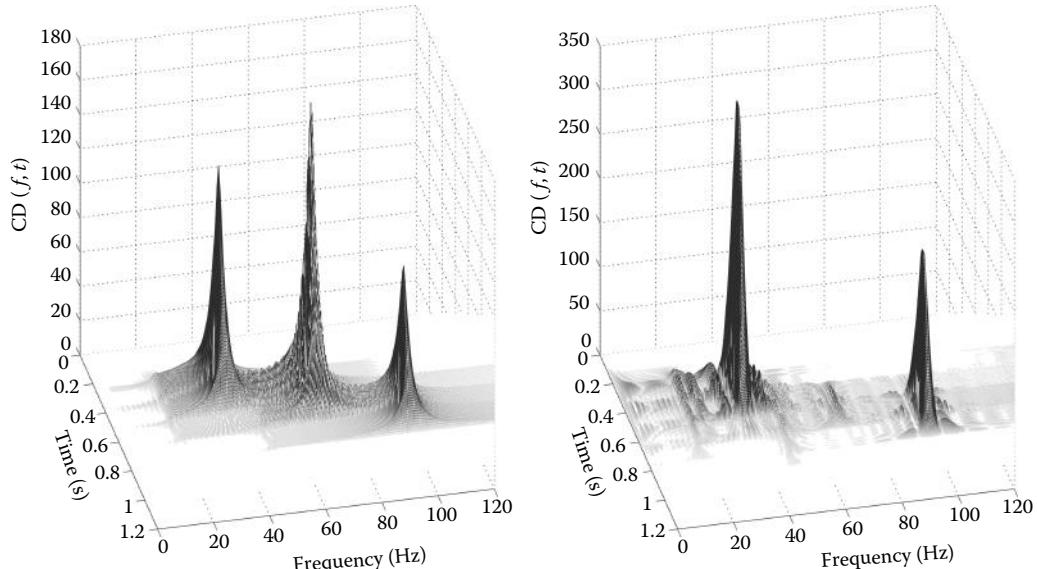


Figure 6.14 Distributions obtained using a signal containing two sequential sinusoids of 40 and 100 Hz. Each sinusoid is 0.25-s long and the transition occurs at 0.5 s. (a) The Choi–Williams determining function (Equation 6.15) is used. (b) The BJC (Equation 6.16) distribution is used.

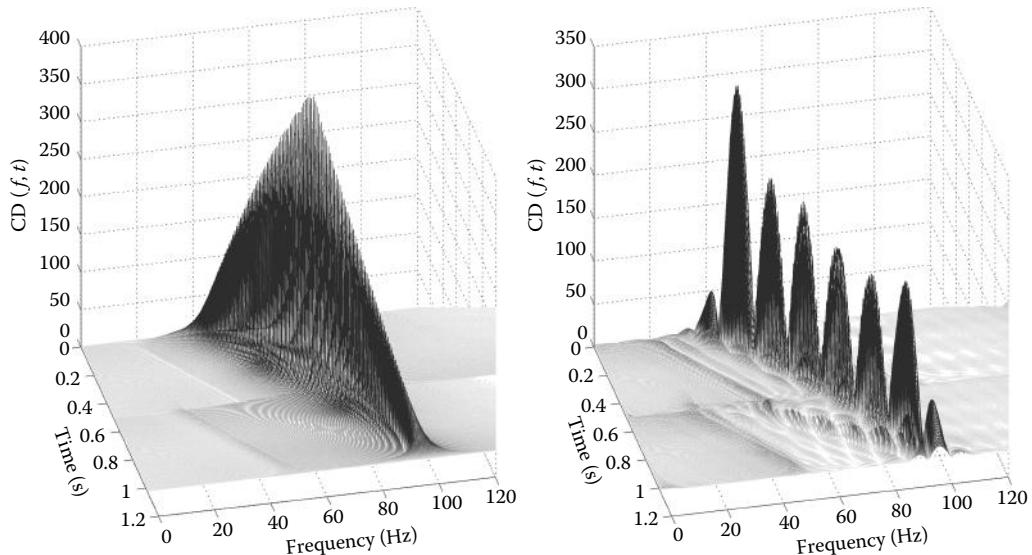


Figure 6.15 Distributions obtained using a chirp signal that increases linearly in frequency from 40 to 100 Hz. (a) The Choi–Williams determining function is used. (b) The BJC distribution is used.

The STFT is the most straightforward and best-understood approach to describing spectral changes over time. The basic idea is to divide the signal into segments (possibly overlapping) and evaluate the Fourier transform for each segment. The assumption is that the frequency characteristics are relatively consistent within a segment. Since the data segments are likely to be short (hence the “short-term” in the name), a window is frequently applied to the data before taking the Fourier transform. To have good time resolution, the data segments should be short, but to have good frequency resolution, the data segments should be long. The compromise between time and frequency resolution is inherent to the STFT and the lower limits of these resolutions are described by an uncertainty equation (Equation 6.3).

The fixed limit to time–frequency resolution has motivated the development of a very different approach to time–frequency spectral analysis based on the instantaneous autocorrelation function (Equation 6.5). This function is similar to the standard autocorrelation function introduced in Chapter 2, but the time variable is not integrated out; so, the function is 2-D, including both time and lag (i.e., τ) variables. Taking the standard 1-D Fourier transform over the lag dimension produces a 2-D function of time and frequency known as the Wigner–Ville distribution.

Unfortunately, the Wigner–Ville spectrum often shows energy where it does not exist in the signal. This artifact is due to cross-products generated by the multiplication used to determine the instantaneous autocorrelation function. Special 2-D filters called determining functions can be applied to the instantaneous autocorrelation function to reduce these cross-products. A large number of such determining functions exist, each with its own set of strengths and weaknesses. Trial and error may be needed to find the most suitable determining function for a given application.

In this chapter, we have explored only a few of the many possible time–frequency distributions and, necessarily, covered only the very basics of this extensive subject. Two of the more important topics that are not covered here are the estimation of instantaneous frequency from the time–frequency distribution, and the effect of noise on these distributions. The former is discussed in Chapter 11; the latter is touched on in the problem set below.

PROBLEMS

- 6.1 Construct a chirp signal similar to that used in Example 6.2. Evaluate the analysis characteristics of the STFT using two different window sizes. Specifically, use window sizes of 128 and 32 samples. Plot as both a 3-D mesh plot and as a contour plot using the subplot to plot all four on one graph. Note that in the contour plot, the line thickness does not change much since the decrease along one axis is offset by an increase along the other.
- 6.2 Construct the chirp signal used in Example 6.2 and Problem 6.1 ($f_s = 500$ Hz, $T_T = 1.0$ s). Evaluate the analysis characteristics of the STFT using three different window sizes and two different windows shapes. Specifically, use window sizes of 128, 64, and 32 points and apply the STFT using both the Hamming and Chebyshev windows. Modify `spectog` (in the associated files) to apply the Chebyshev window or else use MATLAB's `spectrogram`. Plot only contour plots and combine all plots on a single graph for easy comparison. Which window size produces the narrowest time-frequency spectrum? Does the type of window shape used make a difference?
- 6.3 Construct the chirp signal used in Example 6.2 ($f_s = 500$ Hz, $T_T = 1.0$ s). Evaluate the analysis characteristics of the STFT using either `spectrogram` or `spectog` with a window size of 64 and the default window shape (Hamming). Use window overlaps of 25%, 50%, 75%, and 95%. Use contour plots and combine the plots on a single graph for easy comparison. Which overlap produces the narrowest time-frequency spectrum? Note that in general, overlap does not make much difference in the thickness of the time-frequency spectral line for, as shown in Problems 6.6 and 6.7, increasing overlap improves time resolution at the expense of frequency resolution. Repeat the problem using the mesh 3-D plot and note the smoother plot with 95% overlap.
- 6.4 Load the file `time_freq1.mat` that contains variable `x`, a signal with two frequencies closely spaced in frequency and in time; specifically, they are 0.1 s and 10 Hz apart. Assume $f_s = 500$ Hz. Analyze the signal using the STFT beginning with a window size of 512 samples. Reduce the window until you can see a definite gap in time between the two signals. Use `subplot` to plot up to six contour plots on a page and find the maximum window size that shows the two signals separated in time. Approximately, how much do the two frequencies overlap at this window size?
- 6.5 Load the file `time_freq1.mat` used in Problem 6.3 containing two frequencies closely spaced in frequency and in time ($f_s = 500$ Hz). Beginning with a window size of 128 samples, increase the window size until you observe a distinct gap between the frequencies of the two signals. What is the *approximate* time overlap (in sec) between the two signals at this window size? Estimate the frequency of the two signals.
- 6.6 Load the file `time_freq1.mat` used in Problem 6.4. This file has a vector `x` that contains two frequencies closely spaced in frequency and in time ($f_s = 500$ Hz). Set the window size to 64 samples and evaluate the STFT with overlaps of 25%, 50%, 75%, and 95%. Which overlap produces the best time separation between the two signals?
- 6.7 Load the file `time_freq1.mat` used in Problem 6.4 with vector `x` containing two frequencies closely spaced in frequency and in time ($f_s = 500$ Hz). Set the window size to 240 samples and evaluate the STFT with overlaps of 25%, 50%, 75%, and 95%. Which overlap produces the best frequency separation between the two signals? Note that this is opposite to the finding in Problem 6.6.

- 6.8 Load the file `time_freq2.mat` containing signal vector \mathbf{x} that consists of two narrowband signals buried in noise. The signal is 3.3-s long and $f_s = 600$ Hz. Use the STFT to find the frequencies of the narrowband signals, their durations, and the time at which they occur (i.e., their onset time). Because of the noise, you will have to search for the window size that best defines these narrowband signals. [Hint: A contour plot will show time and frequency characteristics best.]
- 6.9 Load the file `time_freq3.mat` that contains signal vector \mathbf{x} that consists of an unknown signal buried in noise. The signal is 2000-samples long and $f_s = 500$ Hz. Use the STFT to describe the signal in as much detail as possible. Because of the noise, you will have to search for the window size that best defines the signal. Again, a contour plot will show time and frequency characteristics best.
- 6.10 Rerun Example 6.4 that applies the Wigner–Ville distribution to a chirp signal, but increases the range of the chirp signal to be between 10 and 200 Hz. Plot the distribution using the contour plot. Note that, even though $f_s = 500$ Hz and the maximum frequency of the chirp is only 200 Hz, the aliased chirp is seen in the plot. This is because the instantaneous autocorrelation reduces the effective f_s by 2. Rerun the problem using the analytic signal and demonstrate that the aliased chirp signal is no longer present. (Note: Even though you are running the program twice, you can still use the subplot to get both contour plots on the same page.)
- 6.11 Construct a chirp signal ranging from 40 to 100 Hz and convert that into an analytic signal. (a) Determine and plot (mesh) the Choi–Williams distribution. (b) Add a constant 20-Hz sine wave to the chirp signal and recalculate and plot the Choi–Williams distribution. Note how the chirp is overwhelmed by the amplitude of the continuous sine wave. (c) Calculate and plot the combined signal of Part (b) using Wigner–Ville distribution. Note the increased amplitude of the chirp signal although it is not flat and there are cross-products present.
- 6.12 Repeat Example 6.5 using a determining function constant, α , of 0.1 and 2.0. Plot the distribution as a mesh and note how little difference the value of this constant makes, at least for this signal. Keep the determining function size as 30×30 (i.e., $L = 30$). Use the analytic signal.
- 6.13 Repeat Problem 6.12, keeping α constant at 0.5, but change the determining function size. Use three sizes: 12×12 , 30×30 , and 60×60 . Plot the distribution as a mesh and note the impact of determining function size on the cross-products.
- 6.14 Load the double sine wave found in vector \mathbf{x} in file `double_sine.mat` ($f_s = 500$ Hz). Take the analytic signal. Compare the distributions obtained with the Choi–Williams and BJC distributions using mesh plots, and limit the time axis to 0–1 s and the frequency axis to 0–120 Hz. Note the complete absence of cross-products using the BJC determining function. Also, note that the distribution amplitude is not flat as it should be considering the signal characteristics. [Hint: Use `cohen.m` with the '`c`' and '`s`' options. Use a determining function size of 30×30 ($L = 30$)].
- 6.15 Repeat Problem 6.14, but add noise to the signal. Make the variance of the added noise equal to the variance of the signal. Note which determining function is more immune to the noise.
- 6.16 Repeat Problem 6.15, but use the sequential sinusoidal step signal ($f_s = 500$ Hz, $N = 512$). (You can use $\mathbf{x} = \text{tf_signal('ss')}$ to generate this signal). Again, make the variance of the added noise equal to the variance of the signal. Note which

Biosignal and Medical Image Processing

- determining function is more immune to the noise and which is more immune to the cross-products.
- 6.17 Generate a chirp signal used in Example 6.5. Add noise to the signal as in Problem 6.16 so that the variance of the signal is equal to the variance of the noise. Find the distribution of this noisy signal using the Wigner–Ville and Choi–Williams distributions. Compare the two distributions using mesh plots. Note the distribution with the greater noise immunity.
 - 6.18 Generate the two sequential sinusoid signals as in Problem 6.16. Make the variance of the noise three times the variance of the signal. Use the Choi–Williams distribution with two determining function sizes: $L = 30$ and $L = 120$. Use mesh plots to compare the two distributions and note the influence of filter size on noise immunity.
 - 6.19 Use the code in Example 6.6 to compare the BJC with the Choi_Williams determining functions as the latter was modified by one of the authors. Select the ‘m’ option in cohen which uses JLS to construct the determining function.

7

Wavelet Analysis

7.1 Introduction

Before introducing *wavelet transform*, let us review some of the concepts regarding transforms presented in Chapter 2. A transform can be thought of as a remapping of a signal into something that provides more information than the original. The Fourier transform fits this definition quite well because the frequency information it provides often leads to new insights about the original signal. However, the inability of the Fourier transform to describe both time and frequency characteristics (see Section 6.2) of the waveform leads to a number of different approaches described in the last chapter. None of these approaches is able to solve the time-frequency problem without added artifacts. The wavelet transform can be viewed as yet another way to describe the properties of a waveform that changes over time, but in this case, the waveform is divided not into sections of time, but *segments of scale*.

In the Fourier transform, the waveform is compared to a sine function, in fact a whole family of sine functions at harmonically related frequencies. This correlation (or projection) is implemented by multiplying the waveform with the sinusoidal functions, then averaging (using either integration in the continuous domain or summation in the discrete domain). This leads to Equation 3.17 repeated here:

$$X[m] = \int_{-\infty}^{\infty} x(t)e^{-j2\pi mf_it} dt \quad (7.1)$$

As discussed in Chapter 2, almost any family of functions can be used to probe the characteristics of a waveform, but sinusoidal functions are particularly popular because of their unique frequency characteristics: they contain energy at only one specific frequency; this leads to easy conversion into the frequency domain.

Other probing functions or basis can be used to evaluate some particular behavior or characteristic of the waveform. If the probing function is of finite duration, it is appropriate to translate, or slide, the function over the waveform, $x(t)$, as is done in convolution, and the STFT (Equation 6.1) repeated here:

$$\text{STFT}(t, f) = \int_{-\infty}^{\infty} x(\tau)w(t - \tau)e^{-j2\pi mf_i\tau} d\tau \quad (7.2)$$

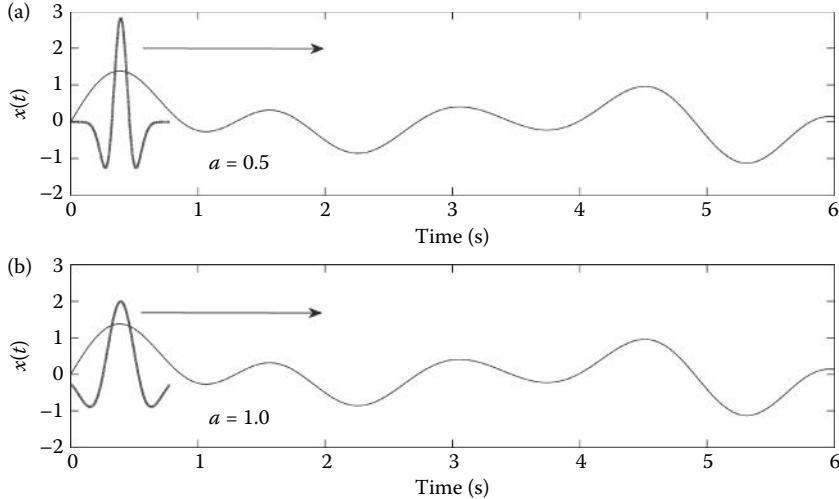


Figure 7.1 In the CWT, each family member slides over the signal and a correlation or projection is made between the family member and the signal at that position. This is essentially a series of cross-correlations between the signal, $x(t)$, and the family member, $f_m(t)$. Two example family members are shown for $\alpha = 0.5$ (a) and 1.0 (b).

where m serves as an indication of family member and $w(t - \tau)$ is a sliding window function, where t acts to translate the window over x . More generally, a translated probing function can be written as

$$X(t, m) = \int_{-\infty}^{\infty} x(\tau) f(t - \tau)_m d\tau \quad (7.3)$$

where $f_m(t)$ is some family of functions or basis, with m specifying the family member. This equation is presented in discrete form in Equation 2.42. As shown in Figure 7.1, each family member slides along $x(t)$ and a correlation or projection is performed for each position. This is the same operation as in cross-correlation (see Section 2.3.2.3).

If the family of functions, $f_m(t)$, is sufficiently large, then it should be able to represent all the information in the waveform $x(t)$. This would then allow $x(t)$ to be reconstructed from $X(t, m)$, making this transform bilateral as defined in Chapter 2. Often, the family or basis is so large that $X(t, m)$ forms a redundant set of descriptions, more than sufficient to recover $x(t)$. This redundancy can sometimes be useful, serving to reduce noise or acting as a control, but may simply be unnecessary. Note that while the Fourier transform is not redundant, most transforms in Chapter 6 (including the STFT and all the distributions) are highly redundant since they map a variable of one dimension, t , into a variable of two dimensions, t, f . Since the purpose of these transforms is signal analysis, redundancy is not an issue.

7.2 Continuous Wavelet Transform

The wavelet transform introduces an intriguing twist to the basic concept of a sliding correlation (or projection) described by Equation 6.2 and elsewhere. In wavelet analysis, a basis of family members is also used, but the family members consist of enlarged or compressed versions of the

7.2 Continuous Wavelet Transform

base function. This concept leads to the defining equation for the *continuous wavelet transform* (CWT):

$$W(a,b) = \int_{-\infty}^{\infty} x(t) \frac{1}{\sqrt{|a|}} \Psi * \left(\frac{t-b}{a} \right) dt \quad (7.4)$$

where b acts to translate the function across $x(t)$, performing the same role as t in Equations 7.2 and 7.3 and the variable a acts to vary the timescale of the probing function, Ψ . If a is greater than one, the wavelet function, $\Psi(t)$, is stretched along the time axis; if it is less than one (but still positive), it contracts the function. (Negative values of a simply flip the probing function on the time axis.) While the probing function could be any of a number of different functions, it always takes on the form of a damped oscillation, hence the term “wavelet.” The $*$ indicates the operation of complex conjugation, and the normalizing factor $1/\sqrt{|a|}$ ensures that the energy is the same for all values of a (all values of b as well, since translations do not alter wavelet energy). If $b = 0$, and $a = 1$, then the wavelet is in its natural form, which is termed the mother wavelet,* that is, $\Psi_{1,0}(t) \equiv \Psi(t)$. A mother wavelet is shown in Figure 7.2 along with some family members produced by dilation and contraction. The wavelet shown is the popular *Morlet* wavelet, named after a pioneer of wavelet analysis. It consists of a cosine tapered by a Gaussian function.

$$\Psi(t) = e^{-t^2} \cos \left(\pi \sqrt{\frac{2}{\ln 2}} t \right) \quad (7.5)$$

As shown in Problem 7.2, the Gaussian function provides the optimal trade-off between time and frequency resolutions, so it is a popular way to shape wavelets.

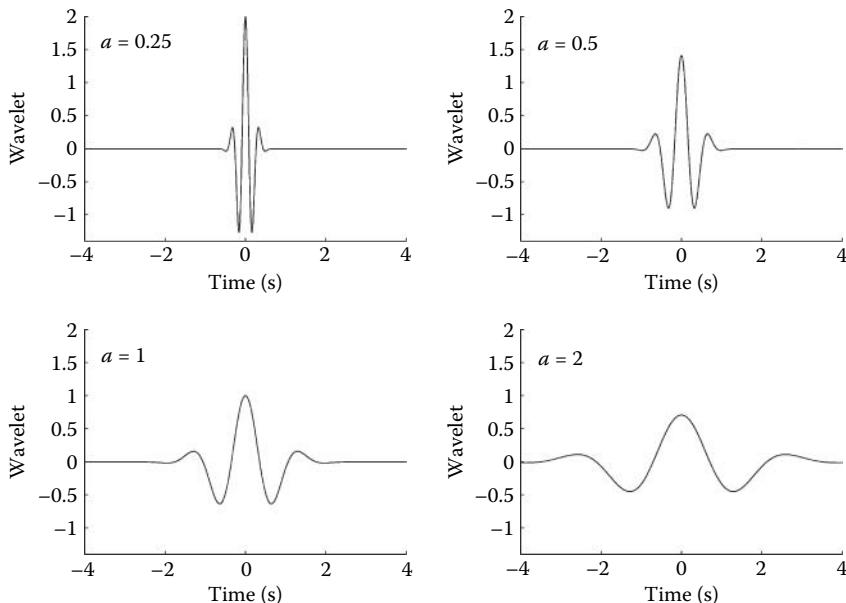


Figure 7.2 A mother wavelet ($a=1$) with one dilation ($a=2$) and two contractions ($a=0.25$ and 0.5). Note the amplitude changes that keep the area under the wavelet constant.

* Individual members of the wavelet family are specified by the subscripts a and b ; that is, $\Psi_{a,b}(t)$.

Biosignal and Medical Image Processing

The wavelet coefficients, $W(a,b)$, describe the correlation between the waveform and the wavelet at various translations and scales, that is, the similarity between the waveform and the wavelet at every combination of scale and position, a,b . Stated another way, the coefficients provide the amplitudes of a series of wavelets, over a range of scales and translations, that would need to be added together to reconstruct the original signal. From this perspective, wavelet analysis can be thought of as a search over the waveform of interest for activity that most clearly approximates the shape of the wavelet. This search is carried out over a range of wavelet sizes: the time span of the wavelet varies, but its basic shape remains the same. Since the net area of a wavelet is always zero by design, a waveform that is constant over the length of the wavelet gives rise to zero coefficients. Wavelet coefficients respond to changes in the waveform: more strongly to changes on the same scale as the wavelet, and most strongly to changes that resemble the wavelet. Although it is a redundant transformation, it is often easier to analyze or recognize patterns using the CWT. An example of the application of the CWT to analyze a waveform is given in the section on MATLAB implementation.

If the wavelet function, $\Psi(t)$, is appropriately chosen, then it is possible to reconstruct the original waveform from the wavelet coefficients just as in the Fourier transform. Since the CWT decomposes the waveform into coefficients of two variables, a and b , a double summation (or integration) is required to recover the original signal from the coefficients:

$$x(t) = \frac{1}{C} \int_{a=-\infty}^{\infty} \int_{b=-\infty}^{\infty} W(a,b) \Psi(t) da db \quad (7.6)$$

where

$$C = \int_{-\infty}^{\infty} \frac{|\Psi(\omega)|}{|\omega|} d\omega$$

where C must range between 0 and infinity (the so-called *admissibility condition*) for recovery using Equation 7.6.

In fact, reconstruction of the original waveform is rarely performed using the CWT coefficients because of the redundancy in the transform. When recovery of the original waveform is desired, the more parsimonious discrete wavelet transform is used as described later in this chapter.

7.2.1 Wavelet Time–Frequency Characteristics

Wavelets such as those shown in Figure 7.2 do not exist at a specific time or a specific frequency. In fact, wavelets provide a compromise in the battle between time and frequency localization: they are well localized in both time and frequency, but not precisely localized in either. A measure of the time range of a specific wavelet, Δt_Ψ , can be specified by the square root of the second moment of a given wavelet about its time center (i.e., its first moment) (Akansu and Haddad, 1992; Popoulis, 1977):

$$\Delta t_\Psi = \sqrt{\frac{\int_{-\infty}^{\infty} (t - t_0)^2 |\Psi(t/a)|^2 dt}{\int_{-\infty}^{\infty} |\Psi(t/a)|^2 dt}} \quad (7.7)$$

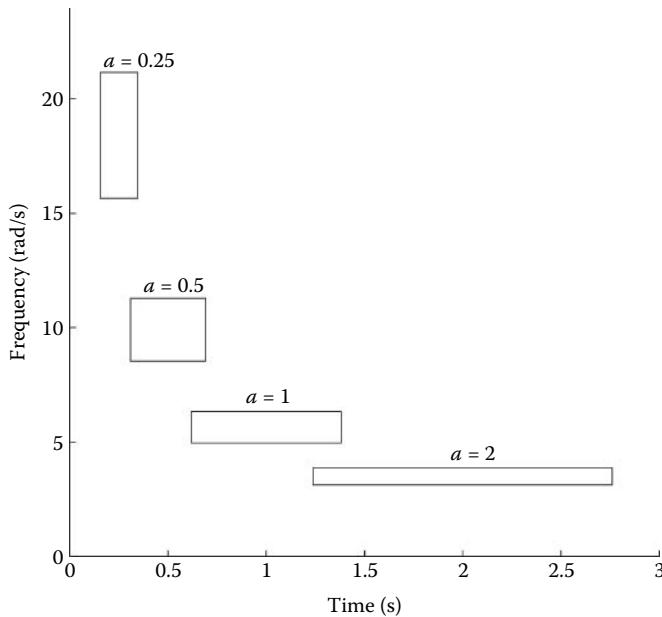


Figure 7.3 Time–frequency boundaries of the Mexican Hat wavelet for various values of a . The area of each of these boxes is constant (see Equation 7.12). The code that generates this figure is based on Equations 7.7 through 7.10 and is given in Example 7.2.

advantage of the CWT is that a is variable and changes during the analysis; hence, there is a *built-in trade-off* between time and frequency resolution that varies during the analysis. This automatic trade-off between time and frequency resolution is the key to the success of the CWT and makes it well suited to analyzing signals with rapidly varying high-frequency components superimposed on slowly varying low-frequency components.

7.2.2 MATLAB Implementation

A number of software packages exist in MATLAB for computing the CWT, including MATLAB's Wavelet Toolbox and "Wavelab," which is available free over the Internet (<http://www-stat.stanford.edu/~wavelab/>). However, it is not difficult to implement Equation 7.4 directly, as illustrated in the example below.

EXAMPLE 7.1

Write a program to construct the CWT of a signal consisting of two sequential sine waves of 10 and 20 Hz. (i.e., similar to the signal shown in Figure 6.1). Use the Morlet wavelet. Make $f_s = 200$ Hz and use 1000 samples to generate a 5.0-s signal. Plot the wavelet coefficients, as functions of a and b , as a contour and 3-D plot.

Solution

The signal waveform is the same as that constructed in Example 6.1. To generate the wavelet, a time vector, b , is used to produce the positive half of the wavelet and also for plotting. The length of this vector is chosen to produce a wavelet that is ± 5 s long. The range of a is chosen to generate wavelets that adequately probe the signal. The initial value of a begins at 0.5, its value decreasing with each iteration to produce a more compressed wavelet that probes ever-increasing frequencies in the signal. Again, a is inversely related to the frequencies probed. In this example, 120 different

7.2 Continuous Wavelet Transform

values of a are used. At each value of scale, the positive half of the Morlet wavelet is constructed using the defining equation (Equation 7.5); the negative half is generated from the positive half by concatenating a time-reversed (flipped) version with the positive side.

The wavelet coefficients at a given scale are obtained by convolution of the scaled wavelet with the signal using MATLAB's `conv` routine with the 'same' option. (Since the wavelet is symmetric, convolution is the same as cross-correlation.) As the first argument to the `conv` routine is the signal, 1000 correlation coefficients will be generated. These convolution coefficients are plotted three-dimensionally along the b axis for the given value of a .

```
% Example 7.1 Example of continuous wavelet transform.  
%  
% Set up constants  
fs = 200; % Sample frequency  
N = 1000; % Signal length and half wavelet lenght  
n = N/4; % Signal length divided by 4  
f1 = 10; % First frequency in Hz  
f2 = 20; % Second frequency in Hz  
resol_level = 120; % Number of levels of a  
decr_a = 1; % Decrement for a  
a_init = 0.5; % Initial a  
wo = pi * sqrt(2*log2(2));  
b = (1:N)/fs; % Time vector for wavelet and plotting  
%  
% Generate the signal  
tn = (1:n)/fs; % Time vector to create signal  
x = [zeros(n,1); sin(2*pi*f1*tn)'; sin(2*pi*f2*tn)'; zeros(n,1)];  
%  
% Calculate Continuous Wavelet Transform  
for k = 1:resol_level  
    a(k) = a_init/(k*decr_a); % Set scale  
    t = t1/a(k); % Time vector for wavelet  
    wav = (exp(-t.^2).*cos(wo*t))/sqrt(a(k)); % Morlet wavelet  
    psi = [fliplr(wav) wav(2:end)]; % Make symmetrical about zero  
    CW_Trans(:,k) = conv(x,psi,'same'); % Symmetrical convolution  
end
```

Results

The plot of the convolution (i.e., cross-correlation), coefficients as a function of a and b , is shown as a 3-D mesh plot in Figure 7.4 and as a contour plot in Figure 7.5. These results are similar to the time-frequency characteristics produced by the STFT shown in Figure 6.3. However, note that the higher frequencies are probed with shorter wavelets producing very high time resolution as well as good frequency resolution (Figure 7.5).

In this example, a is modified by division with a linearly increasing value. Often, wavelet scale is modified based on octaves or fractions of octaves.

A determination of the time-frequency boundaries of a wavelet through MATLAB implementation of Equations 7.7 through 7.10 is provided in the next example.

EXAMPLE 7.2

Find the time-frequency boundaries of the Mexican Hat wavelet.

Solution

For each of four values of a , the scaled wavelet is constructed using an approach similar to that found in Example 7.1. The magnitude squared of the frequency response is calculated using the

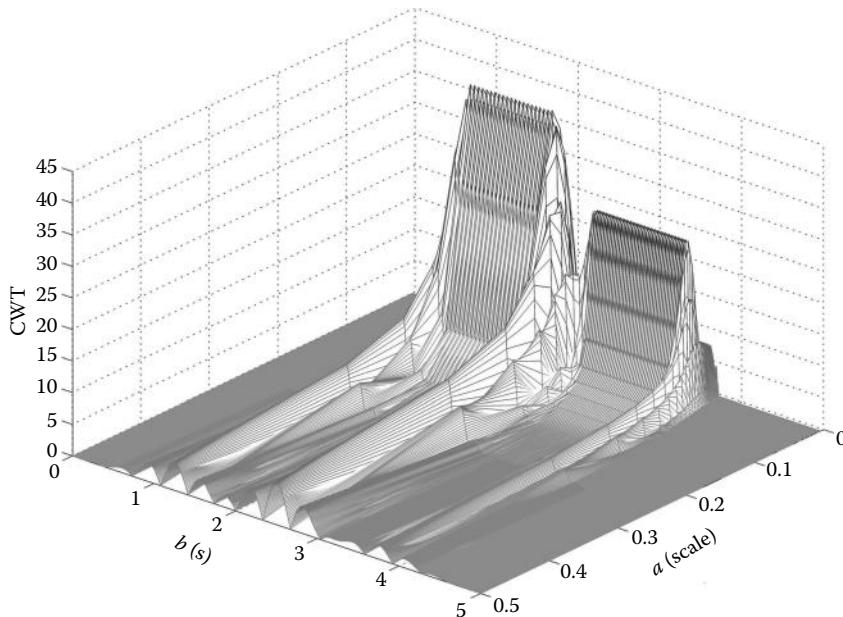


Figure 7.4 A 3-D mesh plot of wavelet coefficients obtained by applying the CWT to a signal consisting of two sequential sine waves of 10 and 20 Hz. The Morlet wavelet was used.

Fourier transform. The center time, t_0 , and center frequency, ω_0 , are constructed by direct application of Equations 7.8 and 7.10. Note that since the wavelet is constructed symmetrically about $t = 0$, the center time, t_0 , will always be zero, and an appropriate offset time, t_{01} , is added during plotting. The time and frequency boundaries are calculated using Equations 7.7 and 7.9, and the resulting boundaries are plotted as a rectangle about the appropriate center time, t_{01} , and center frequency ω_0 .

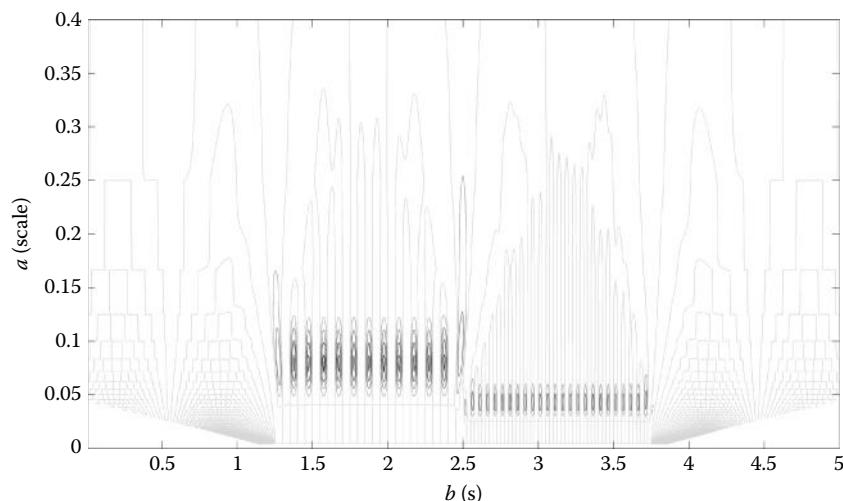


Figure 7.5 Wavelet coefficients produced by applying the Morlet wavelet to a signal consisting of two sequential sine waves of 10 and 20 Hz. Compare this plot with that produced by the STFT in Figure 6.4. Note the sharp time transitions and the respectable frequency resolution.

```
% Example 7.2 Plot of wavelet time-frequency boundaries
%
fs = 200;                                % Sample frequency
N = 1024;                                 % Wavelet half length
a = [.25 .5 1 2];                         % Values of 'a'
wo = pi * sqrt(2/log2(2));                 % Time vector to generate wavelet
t1 = (0:(N-1))/fs;                        % Time vector for wavelet
t2 = [-fliplr(t1) t1(2:end)];             % Frequency vector for wavelet
w = 2*pi*(1:N/2)*fs/N;                    % Frequency vector for wavelet
hold on;
%
for k = 1:length(a)
    t = t1/a(k);                          % Set time vector for Wavelet
    wav = exp(-t.^2).* (1 - 2*t.^2);      % Generate Mexican Hat Wavelet
    psi = [fliplr(wav) wav(2:end)];        % Make symmetrical at t = 0
    psi_sq = abs(psi).^2;                  % Square wavelet
    Psi = abs(fft(psi)/N);                % Get spectrum
    Psi_sq = Psi(1:N/2).^2;                % and square. Use only fs/2 range
    t0 = sum(t2.* psi_sq)/sum(psi_sq);    % t0; Eq. 7.8
    delta_t = sqrt(sum((t2 - t0).^2 .* psi_sq)/sum(psi_sq)); % Δt, Eq. 7.7
    w0 = sum(w.*Psi_sq)/sum(Psi_sq);       % ω0, Eq. 7.10
    delta_w = sqrt(sum((w - w0).^2 .* Psi_sq)/sum(Psi_sq)); % Δω, Eq. 7.9
    % Plot boundaries
    t01 = a(k);                           % Adjust center time
    % Plot time - frequency rectangle
    plot([t01 - delta_t/2 t01 - delta_t/2], [w0 - delta_w/2 w0 + delta_w/2]);
    plot([t01 + delta_t/2 t01 + delta_t/2], [w0 - delta_w/2 w0 + delta_w/2]);
    plot([t01 - delta_t/2 t01 + delta_t/2], [w0 - delta_w/2 w0 - delta_w/2]);
    plot([t01 - delta_t/2 t01 + delta_t/2], [w0 + delta_w/2 w0 + delta_w/2]);
    area(k) = delta_t * delta_w;           % Time - freq. product.
end
.....lables.....
```

Results

The plot generated by this code has already been shown in Figure 7.3. The program also calculates the area of each time–frequency rectangle, which should all be the same based on Equation 7.12. The results are quite similar: 1.0499, 1.0499, 1.0507, and 1.1338 for the four rectangles. Note that these areas are larger than the theoretical minimum of 0.5. Problem 7.2 explores the time–frequency resolution of other wavelets, including the Gaussian function that attains, in theory, the minimum time–frequency product of 0.5.

7.3 Discrete Wavelet Transform

The CWT has one serious problem: it is highly redundant.* The CWT provides an oversampling of the original waveform: many more coefficients are generated than are actually needed to uniquely specify the signal. This redundancy may not be a problem in analysis applications, but will be costly if the application calls for recovery of the original signal. For recovery, all of the coefficients will be required and the computational effort could be excessive. In applications that require bilateral transformations, we prefer a transform that produces the minimum number of coefficients required to recover the original signal. Ideally, the transform should require the same number of points that are in the original signal as is the case for the Fourier transform. The *discrete wavelet transform (DWT)* achieves this parsimony by restricting the variation in translation

* In its continuous form, it is actually infinitely redundant!

Biosignal and Medical Image Processing

and scale, usually to powers of 2. When the scale is changed in powers of 2, the discrete wavelet transform is sometimes termed the *dyadic wavelet transform*, which unfortunately carries the same abbreviation: DWT. The DWT may still require redundancy to produce a bilateral transform unless the wavelet is carefully chosen such that it leads to an orthogonal family (i.e., an orthogonal basis). In this case, the DWT will produce a nonredundant, bilateral transform.

The basic analytical expressions for the DWT are presented here; however, the transform is easier to understand using *filter banks* as described in the next section. Moreover, the implementation of the DWT always uses filter banks. The only purpose of this section is to familiarize you with the equations sometimes found in the literature; they are not used to realize the DWT. The theoretical link between filter banks and the equations is presented just before the MATLAB Implementation section.

The DWT is often introduced in terms of its recovery transform:

$$x(t) = \sum_{k=-\infty}^{\infty} \sum_{\ell=-\infty}^{\infty} d(k, \ell) 2^{-k/2} \Psi(2^{-kt} - \ell) \quad (7.13)$$

where k is related to a as $a = 2^k$; b is related to ℓ as $b = 2^{\ell k}$; $d(k, \ell)$ is a sampling of $W(a, b)$ at discrete points k ; and Ψ is the wavelet function.

In the DWT, a new concept is introduced termed the *scaling function*, a function that facilitates computation of the DWT. To implement the DWT efficiently, the finest resolution is computed first. The computation then proceeds to coarser resolutions but, rather than start over on the original waveform, the computation uses a smoothed version of the fine resolution waveform. This smoothed version is obtained with the help of the scaling function. (In fact, the scaling function is sometimes referred to as the *smoothing function*.) The definition of the scaling function uses *dilation* or a *two-scale difference equation*:

$$\phi(t) = \sum_{n=-\infty}^{\infty} \sqrt{2} c[n] \phi(2t - n) \quad (7.14)$$

where $c[n]$ is a series of scalars that defines the specific scaling function. This equation involves two timescales, t and $2t$, and can be quite difficult to solve.

In the DWT, the wavelet itself can be defined from the scaling function:

$$\Psi(t) = \sum_{n=-\infty}^{\infty} \sqrt{2} d[n] \phi(2t - n) \quad (7.15)$$

where $d[n]$ is a series of scalars that are related to the waveform $x(t)$ (Equation 7.13) and that define the discrete wavelet in terms of the scaling function.

7.3.1 Filter Banks

For most signal- and image-processing applications, DWT-based analysis is best described in terms of filter banks. The use of a group of filters to divide up a signal into various spectral components is termed *subband coding*. The most basic implementation of the DWT uses only two filters as in the filter bank shown in Figure 7.6.

The waveform under analysis is divided into two components, $y_{lp}[n]$ and $y_{hp}[n]$, by the digital filters $H_0(\omega)$ and $H_1(\omega)$. The spectral characteristics of the two filters must be carefully chosen, with $H_0(\omega)$ having a lowpass spectral characteristic and $H_1(\omega)$ having a highpass spectral characteristic. The highpass filter is analogous to the application of the wavelet to the original signal while the lowpass filter is analogous to the application of the scaling or smoothing function. If the filters are *invertible filters*, then it is possible, at least in theory, to construct complementary filters (filters that have a spectrum of the inverse of $H_0(\omega)$ or $H_1(\omega)$) that recover the original

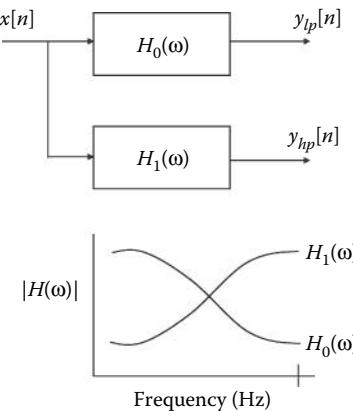


Figure 7.6 A simple filter bank consisting of only two filters applied to the same waveform. The filters have lowpass and highpass spectral characteristics; thus, the filter outputs consist of a lowpass subband, $y_{lp}[n]$, and a highpass subband, $y_{hp}[n]$.

waveform from the subband signals, $y_{lp}[n]$ or $y_{hp}[n]$. Such invertible filters exist, but are not trivial to design. For example, any filter whose gain goes to zero would not be invertible since the frequencies at zero gain would be lost.

Signal recovery is illustrated in Figure 7.6 where a second pair of filters, $G_0(\omega)$ and $G_1(\omega)$, operate on the high- and lowpass subband signals; their sum is used to reconstruct a close approximation of the original signal, $x'(t)$. The filter bank that decomposes the original signal is termed the *analysis filters* while the filter bank that reconstructs the signal is termed the *syntheses filters*. FIR filters are used throughout because they are inherently stable and easier to implement.

Filtering the original signal, $x[n]$, only to recover it with inverse filters would be a pointless operation, although this process may have some instructive value as in Example 7.3. In some analysis applications, only the subband signals are of interest, and reconstruction is not needed, but in other wavelet applications, some operation is performed on the subband signals, $y_{lp}[n]$ and $y_{hp}[n]$ in Figure 7.7, before reconstruction of the output signal. In such cases, the output will no longer be exactly the same as the input. If the output is essentially the same, as occurs in some data compression applications, the process is termed *lossless*; otherwise, it is a *lossy* operation.

There is one major concern with the general approach schematized in Figure 7.7: it requires the generation of, and operation on, twice as many samples as are in the original waveform $x[n]$. This is because each analysis filter produces signals that have the same number of samples as the original signal, $x[n]$. This problem will only get worse if more filters are added to the filter banks. Clearly, there must be redundant information contained in signals $y_{lp}[n]$ and $y_{hp}[n]$, since they are both required to represent $x[n]$, but with twice the number of samples of $x[n]$. However,

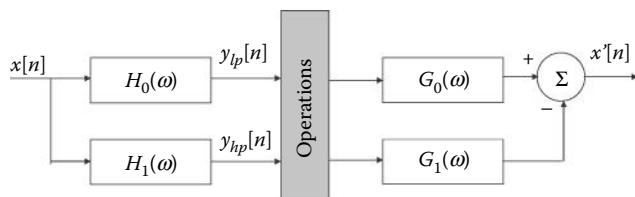


Figure 7.7 A typical wavelet application using filter banks containing only two filters. The input waveform is first decomposed into subbands using what is called the analysis filter bank. Some process is applied to the filtered signals before reconstruction. Reconstruction is performed by the synthesis filter bank.

Biosignal and Medical Image Processing

if the analysis filters are correctly chosen, it is possible to reduce the length of $y_{lp}[n]$ and $y_{hp}[n]$ by one-half and still be able to recover the original waveform. To reduce the signal samples by one-half, and still represent the same overall time period, we eliminate every other point, say every odd sample. This operation is known as *downsampling* and is illustrated schematically by the symbol $\downarrow 2$. The downsampled version of $y_{lp}[n]$ would then include only the samples with even indices $[y_{lp}[2], y_{lp}[4], y_{lp}[6], \dots]$ and have half the number of samples. The same would be true for $y_{hp}[n]$ if it is similarly downsampled.

If downsampling is used, then there must be some method for recovering the missing data samples (those with odd indices) in order to reconstruct the original signal. An operation termed *upsampling* (indicated by the symbol $\uparrow 2$) accomplishes this operation by replacing the missing samples with zeros. The recovered signal ($x'[n]$ in Figure 7.7) will not contain zeros for these data samples as the synthesis filters, $G_0(\omega)$ and $G_1(\omega)$, “fill in the blanks.” Figure 7.8 shows a wavelet application that uses three filter banks and includes the downsampling and upsampling operations. Downsampled amplitudes are sometimes scaled by 2, a normalization that can simplify the filter calculations when matrix methods are used.

Designing the filters to be used in a wavelet filter bank can be quite challenging because the filters must meet a number of criteria. A prime concern is the ability to recover the original signal after passing through the analysis and synthesis filter banks. Accurate recovery is complicated by the downsampling process. Note that downsampling, removing every other point, is equivalent to sampling the original signal at half the sampling frequency. For some signals, this would lead to aliasing, since the highest-frequency component in the signal may no longer be twice the now-reduced sampling frequency. Appropriately chosen filter banks can eliminate the potential for aliasing. If the filter bank contains only two filter types (highpass and lowpass filters) as in Figure 7.7, the criterion for aliasing cancellation is:

$$G_0(z)H_0(-z) + G_1(z)H_1(-z) = 0 \quad (7.16)$$

where $H_0(z)$ is the transfer function of the analysis lowpass filter, $H_1(z)$ is the transfer function of the analysis highpass filter, $G_0(z)$ is the transfer function of the synthesis lowpass filter, and $G_1(z)$ is the transfer function of the synthesis highpass filter.

The ability to recover the original waveform from the subband waveforms places another important requirement on the filters, which is satisfied when

$$G_0(z)H_0(z) + G_1(z)H_1(z) = 2z^{-N} \quad (7.17)$$

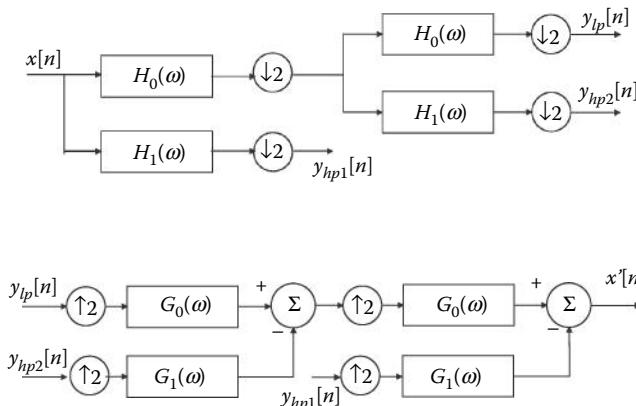


Figure 7.8 Typical wavelet analysis and synthesis filter banks with three filters in each filter bank. The downsampling ($\downarrow 2$) and upsampling ($\uparrow 2$) processes are shown. As in Figure 7.7, some process would be applied to the filter signals, $y_{lp}[n]$, $y_{hp1}[n]$, and $y_{hp2}[n]$, before reconstruction.

7.3 Discrete Wavelet Transform

where the transfer functions are the same as those in Equation 7.16 and N is the number of filter coefficients (i.e., the filter order); z^{-N} is just the delay of the filter. Another desirable feature of wavelet filters is that the subband signals be orthogonal. This means the highpass filter frequency characteristics must have a specific relationship to those of the lowpass filter; specifically

$$H_1(z) = -z^{-N}H_0(-z^{-1}) \quad (7.18)$$

These three equations put constraints on the relationship between the analysis and synthesis filters, between the highpass and lowpass filters, and on the lowpass filter itself. These constraints mean that once a lowpass filter, $H_0(z)$, is chosen, the rest of the filters are prescribed.

Fortunately, quite a few filters have been developed that have most of the desirable properties.* The examples below use filters developed by Daubechies, and are named after her (Figure 7.9). This is a family of popular wavelet filters having four or more coefficients. The coefficients of the lowpass filter, $h_0[n]$, for the four-coefficient Daubechies filter are given as

$$h[n] = \frac{[(1 + \sqrt{3}), (3 + \sqrt{3}), (3 - \sqrt{3}), (1 - \sqrt{3})]}{4\sqrt{2}} \quad (7.19)$$

Other, higher-order, filters in this family are given in the routine daub found in the routines associated with this chapter. It can be shown that orthogonal filters with more than two coefficients must have asymmetrical coefficients.[†] Unfortunately, this precludes these filters from having linear phase characteristics, but this is a compromise that is usually acceptable. More

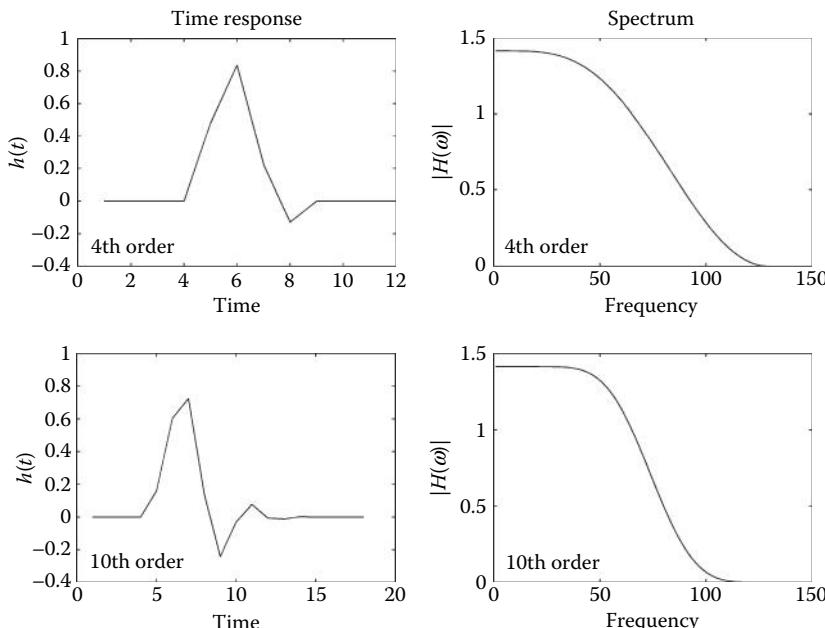


Figure 7.9 The impulse responses and magnitude spectra of 4th order and 10th order Daubechies filters.

* Although no single filter yet exists that has all of the desirable properties.

[†] The two-coefficient, orthogonal filter is $h[n] = [1/2; 1/2]$, and is known as the *Haar* filter. Essentially a two-point moving average, this filter does not have very strong filter characteristics. See Problem 7.7.

Biosignal and Medical Image Processing

complicated *biorthogonal filters* (Strang and Nguyen, 1997) are required to produce minimum phase and orthogonality.

The requirement for orthogonality between lowpass and highpass signals (Equation 7.18) can be implemented by applying the *alternating flip* algorithm to the coefficients of the lowpass filter, $h_0[n]$:

$$h_1[n] = [h_0(N), -h_0(N-1), h_0(N-2), -h_0(N-3), \dots] \quad (7.20)$$

where N is the number of coefficients in $h_0[n]$. Implementation of this alternating flip algorithm is found in the *analyze* program of Example 7.3.

Once the analyzed filters have been constructed, the synthesis filters used for reconstruction are constrained by Equations 7.17 and 7.18. The conditions of Equation 7.17 can be met by making $G_0(z) = H_1(-z)$ and $G_1(z) = -H_0(-z)$. Hence, the synthesis filter transfer functions are related to the analysis transfer functions as

$$G_0(z) = H_0(-z) = z^{-N} H_0(z^{-1}) \quad (7.21)$$

$$G_1(z) = H_1(-z) = z^{-N} H_1(z^{-1}) \quad (7.22)$$

where the second equality in Equations 7.21 and 7.22 comes from the relationship expressed in Equation 7.18. The operations of Equations 7.21 and 7.22 can be implemented in several different ways, but the easiest way in MATLAB is to use the second equality, which can be implemented using the *order flip* algorithm:

$$g_0[n] = [h_0[N], h_0(N-1), h_0(N-2), \dots] \quad (7.23)$$

$$g_1[n] = [h_1(N), h_1(N-1), h_1(N-2), \dots] \quad (7.24)$$

where, again, N is the number of filter coefficients. (It is assumed that all filters have the same order; i.e., they have the same number of coefficients.) These equations demonstrate that all of the filters can be constructed given only the analysis lowpass filter. An example of constructing these filters from the analysis lowpass filter coefficients, $h_0[n]$, is shown in Example 7.3. First, the alternating flip algorithm is used to get the highpass analysis filter coefficients, $h_1[n]$; then, the order flip algorithm is applied as using Equations 7.23 and 7.24 to produce both the synthesis filter coefficients, $g_0[n]$ and $g_1[n]$.

Note that if the filters shown in Figure 7.8 are causal, each would produce a delay that is dependent on the number of filter coefficients. Such delays are expected and natural, and may have to be taken into account in the reconstruction process. However, when the data are stored in the computer, it is possible to implement FIR filters without a delay. Using MATLAB's *conv* with the 'same' option produces an output that is not shifted, as it uses the center points of the convolution as the output. This is sometimes referred to as *periodic convolution*.

7.3.1.1 Relationship between Analytical Expressions and Filter Banks

The filter bank approach described above and the DWT represented by Equations 7.14 and 7.15 were actually developed separately, but have become linked both theoretically and practically. It is possible, at least in theory, to go between the two approaches to develop the wavelet and scaling functions from the filter coefficients and vice versa. In fact, the coefficients $c[n]$ and $d[n]$ in Equations 7.14 and 7.15 are simply scaled versions of the filter coefficients:

$$c[n] = \sqrt{2}h_0[n]; \quad d[n] = \sqrt{2}h_1[n] \quad (7.25)$$

With the substitution of $c[n]$ in Equation 7.14, the equation for the scaling function (the dilation equation) becomes

$$\phi(t) = \sum_{n=-\infty}^{\infty} 2h_0[n]\phi(2t-n) \quad (7.26)$$

Since this is an equation with two timescales (t and $2t$), it is not easy to solve, but a number of approximation approaches have been worked out (Strang and Nguyen, 1997, pp. 186–204). In addition, a number of techniques exist for solving for t in Equation 7.26 given the filter coefficients, $h_1[n]$. Perhaps the most straightforward method of solving for ϕ in Equation 7.26 is to use the frequency-domain representation. Taking the Fourier transform of both sides of Equation 7.26 gives

$$\Phi(\omega) = H_0\left(\frac{\omega}{2}\right)\Phi\left(\frac{\omega}{2}\right) \quad (7.27)$$

Note that $2t$ goes to $\omega/2$ in the frequency domain. The second term in Equation 7.27 can be broken down into $H_0(\omega/4)\Phi(\omega/4)$, so it is possible to rewrite the equation as

$$\begin{aligned} \Phi(\omega) &= H_0\left(\frac{\omega}{2}\right)\left[H_0\left(\frac{\omega}{4}\right)\Phi\left(\frac{\omega}{4}\right)\right] \\ &= H_0\left(\frac{\omega}{2}\right)H_0\left(\frac{\omega}{4}\right)H_0\left(\frac{\omega}{8}\right)\cdots H_0\left(\frac{\omega}{2^N}\right)\Phi\left(\frac{\omega}{2^N}\right) \end{aligned} \quad (7.28)$$

In the limit as $N \rightarrow \infty$, Equation 7.28 becomes

$$\Phi(\omega) = \prod_{j=1}^{\infty} H_0\left(\frac{\omega}{2^j}\right) \quad (7.29)$$

The relationship between $\phi(t)$ and the lowpass filter coefficients can now be obtained by taking the inverse Fourier transform of Equation 7.29. Once the scaling function is determined, the wavelet function can be obtained directly from Equation 7.15 with $2h_1[n]$ substituted for $d[n]$:

$$\psi(t) = \sum_{n=-\infty}^{\infty} 2h_1[n]\phi(2t - n) \quad (7.30)$$

Equation 7.29 also demonstrates another constraint on the lowpass filter coefficients, $h_0[n]$, not mentioned above. In order for the infinite product in Equation 7.29 to converge (or any infinite product for that matter), $H_0(\omega/2)$ must approach 1 as $j \rightarrow \infty$. This implies that $H_0(0) = 1$, a criterion that is easy to meet with a lowpass filter. While Equation 7.30 provides an explicit formula for determining the scaling function from the filter coefficients, an analytical solution is challenging except for very simple filters such as the two-coefficient Haar filter. Solving this equation numerically also has problems due to the short data length: $H_0(\omega)$ would be only 4 points for a four-element filter. Nonetheless, Equation 7.30 provides a theoretical link between the filter bank and analytical DWT methodologies.

These issues described above, along with some applications of wavelet analysis, are presented in the next section on implementation.

7.3.2 MATLAB Implementation

The construction of a filter bank in MATLAB can be achieved using either routines from the MATLAB `filter` routine or simply convolution. All examples here use convolution; specifically, the MATLAB `conv` routine with the 'same' option to generate a noncausal output that is not time-shifted. The next example makes use of three important functions. The routine `daub` is available in the associated files and supplies the coefficients of a Daubechies filter from a simple list of coefficients. In this example, a six-element filter is used, but the routine can also generate coefficients of 4-, 8-, and 10-element Daubechies filters.

EXAMPLE 7.3

Construct a signal consisting of four sine waves having different frequencies and amplitudes along with a small amount of added noise. Deconstruct this signal using an analysis filter bank composed of a six-element Daubechies filter and using $L = 4$ decompositions; that is, a lowpass filter and four highpass filters. Then recover the original signal using an L -level synthesis filter bank. Plot the subbands as well as the filter frequency characteristics.

Solution

This program uses the function `signal` to generate the mixtures of sinusoids. This routine is similar to `sig_noise` except that it generates only mixtures of sine waves without the noise. The first argument specifies the frequency of the sine waves and the third argument specifies the number of samples in the waveform just as in `sig_noise`. The second argument specifies the amplitudes of the sinusoids, not the SNR as in `sig_noise`. The routine assumes $f_s = 1000$ Hz. Noise generated by `randn` is then added to this signal.

This waveform is decomposed into four subbands using the routine `analysis` described below. This routine takes the lowpass filter coefficients as inputs and generates the highpass filter coefficients. The routine uses these filters to construct the subband signals in a single array. The routine also plots the subband signals. Another routine described below, `synthesize`, is used to reconstruct the original signal from the subband array. Since no operation is performed on the subband signals, the reconstructed signal should match the original.

```
% Example 7.3 % Dyadic wavelet Transform Example
%
fs = 1000;                      % Sample frequency
N = 1024;                        % Number of samples
freqsin = [.63 1.1 2.7 5.6];    % Sinusoid frequencies
ampl = [1.2 1 1.2 .75];         % Amplitude of sinusoid
h0 = daub(6);                   % Get filter coeff.
[x t] = signal(freqsin,ampl,N); % Construct signal
x1 = x + (.25 * randn(1,N));   % and add noise
%
an = analyze(x1,h0,4);          % Analytic filters (an contain subbands)
sy = synthesize(an,h0,4);       % Synthesis filters
%
..... plot original and reconstructed signals, offset for
comparison.....
```

The `analysis` function is shown below and implements the analysis filter bank. This routine first generates the highpass filter coefficients, h_1 , from the lowpass filter coefficients, h , using the alternating flip algorithm of Equation 7.20. These FIR filters are then applied using convolution. All of the various subband signals required for reconstruction are placed in a single output array, an . The length of an is the same as the length of the input ($N = 1024$ in this example). The only *lowpass signal* needed for reconstruction is the smoothest lowpass subband (i.e., final lowpass signal in the lowpass chain) and this signal is placed in the first data segment of an taking up the first $N/16$ data points (Figure 7.10). This signal is followed by the last-stage highpass subband, which is of equal length. The next $N/8$ data points contain the second to last highpass subband followed, in turn, by the other subband signals up to the final, highest-resolution highpass subband, which takes up all of the second half of an (Figure 7.10). The remainder of the `analyze` routine calculates and plots the highpass and lowpass filter frequency characteristics.

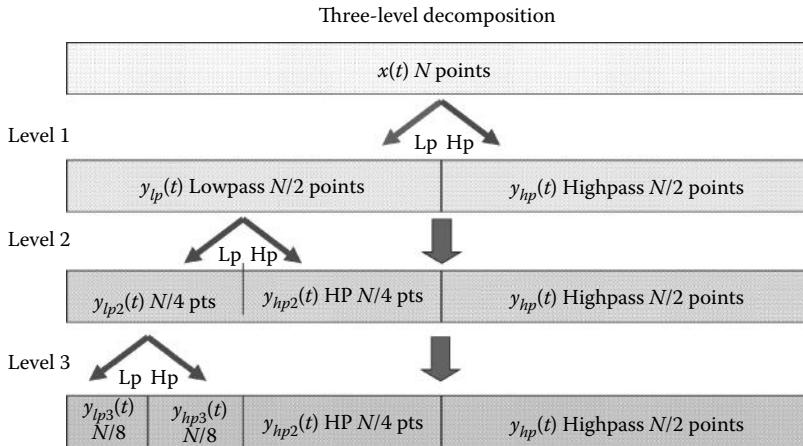


Figure 7.10 The approach employed by the routine `analyze` to decompose an input signal into subbands that ends up using the same amount of memory as the original signal. Each iteration decomposes the lowpass signal into two equal-length subbands half the size. Downsampling occurs between the levels.

```

function an = analyze(x,h0,L)
    .....Comments.....
lf = length(h0);                                % Filter length
lx = length(x);                                % Data length
an = x;                                         % Initialize output
%
% Calculate High pass coefficients from low pass coefficients
for k = 0:(lf - 1)
    h1(k + 1) = (-1)^k * h0(lf - k);        % Uses Eq 7.20
end
%
% Calculate filter outputs for all levels
for k = 1:L
    lpf = conv(an(1:lx),h0,'same');          % Low pass FIR filter
    hpf = conv(an(1:lx),h1,'same');          % High pass FIR filter
    lpf_d = lpf(1:2:end);                   % Downsample
    hpf_d = hpf(1:2:end);
    an(1:lx) = [lpf_d hpf_d];                % Lowpass output at beginning of array.
    lx = lx/2;                             % Occupies half the samples of last pass
.....plot the subbands.....
end
%
.....calculate and plot filter spectra.....
```

The original data are reconstructed from the analyze filter bank signals in the program `synthesize`. This program first constructs the synthesis lowpass filter, g_0 , using order flip applied to the analysis lowpass filter coefficients (Equation 7.23). Although not needed in synthesis, the analysis highpass filter is constructed using the alternating flip algorithm (Equation 7.20). These analysis lowpass and highpass coefficients are then used to construct the synthesis highpass filter coefficients through order flip (Equation 7.24). The synthesis algorithm follows the same pattern as shown in Figure 7.10 except the algorithm starts at the lowest array and works toward the top. The filter loop begins with the coarsest signals first, those in the initial data segments of a with the shortest segment lengths (Figure 7.10, lowest array, left side). The lowpass and highpass signals are upsampled, then filtered using convolution, and the signals added together. This loop

Biosignal and Medical Image Processing

is structured so that on the next pass the recently combined segment is itself combined with the next higher-resolution highpass signal. This iterative process continues until all of the highpass signals are included in the sum.

```
function y = synthesize(a,h0,L)
% Function to calculate synthesize Filter Bank
% y = synthesize(a,h0,L)
% where
%     a = analyze Filter Bank outputs (produced by
%         analyze)
%     h = filter coefficients (low pass)
%     L = decomposition level
%
lf = length(h0);           % Filter length
lx = length(a);           % Data length
lseg = lx/(2^L);          % Length of first low and highpass seg.
y = a;                     % Initialize output
%
g0 = h0(lf:-1:1);        % Order flip
%
Generate analysis, then synthesis highpass coeff.
%
for k = 0:(lf-1)
    h1(k+1) = (-1)^k * h0(lf-k); % Alternating flip, Eq. 7.20
end
g1 = h1(lf:-1:1);        % Order flip, Eq. 7.24
%
% Calculate filter outputs for all levels
for k = 1:L
    lpx = y(1:lseg);          % Get lowpass segment
    hpx = y(lseg + 1:2*lseg); % and highpass segment
    up_lpx = zeros(1,2*lseg); % Initialize vector for upsampling
    up_lpx(1:2:2*lseg) = lpx; % Up sample low pass (every odd point)
    up_hpx = zeros(1,2*lseg); % Repeat for high pass
    up_hpx(1:2:2*lseg) = hpx;
    % Filter using conv and combine.
    y(1:2*lseg) = conv(up_lpx,g0,'same') + conv(up_hpx,g1,'same');
    lseg = lseg * 2;          % Double seg lengths for next pass
end
```

The subband signals are shown in Figure 7.11. Also shown are the frequency characteristics of the Daubechies highpass and lowpass filters. The input and reconstructed output waveforms are shown in Figure 7.12. Note that the reconstructed waveform closely matches the input.

7.3.2.1 Denoising

Example 7.3 is not particularly practical since the reconstructed signal is the same as the original. A more useful application of wavelets is shown in Example 7.4, where some processing is done on the subband signals before reconstruction: in this example, nonlinear filtering. The basic assumption in this application is that the noise is coded into small fluctuations in the higher resolution (i.e., more detailed) highpass subbands. This noise can be selectively reduced by eliminating the sample values that fall below some threshold in the higher-resolution highpass subbands. The idea is that these lower-value samples are purely noise and can safely be eliminated.

7.3 Discrete Wavelet Transform

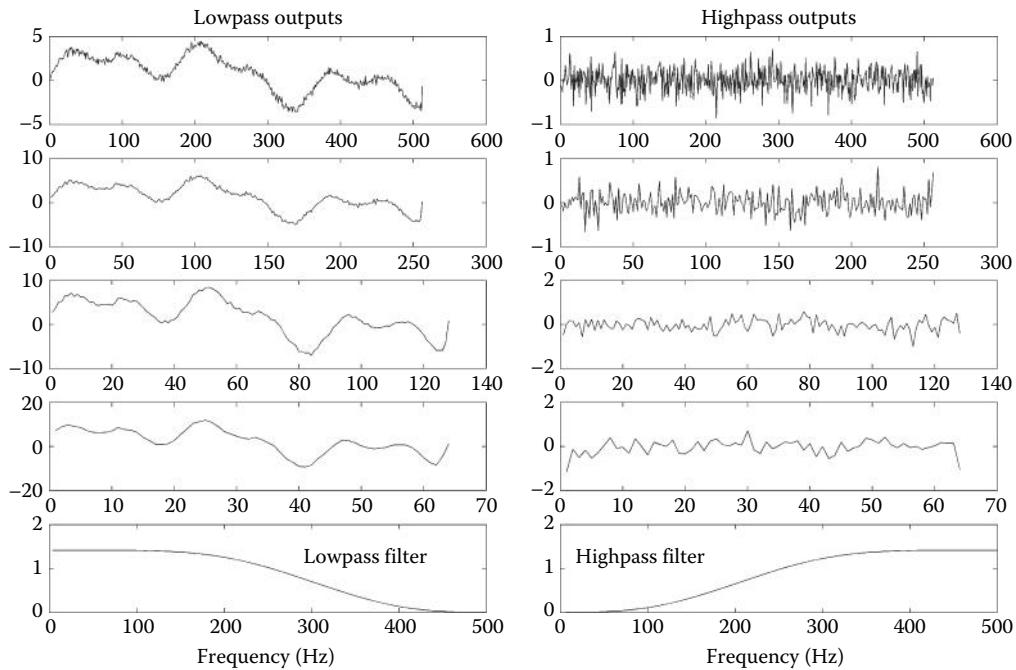


Figure 7.11 Signals generated by the analysis filter bank used in Example 7.3 with the topmost plot showing the outputs of the first set of filters with the finest resolution, the next from the top showing the outputs of the second set of filters, and so on. Only the lowest (i.e., smoothest) low-pass subband signal is included in the output of the analysis routine; the rest are used only in the determination of highpass subbands. The lowest plots show the frequency characteristics of the highpass and lowpass filters.

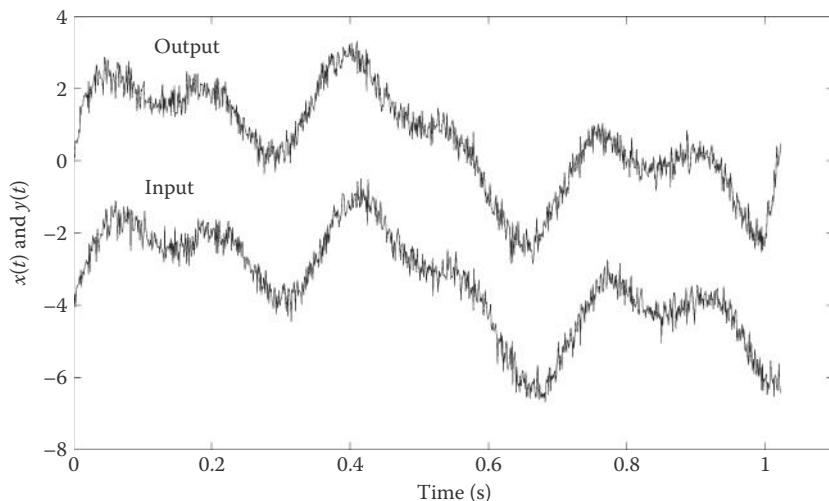


Figure 7.12 Input (lower) waveform to the four-level analysis and synthesis filter banks used in Example 7.3. The upper waveform is the reconstructed output from the synthesis filters. Small artifacts are seen at the beginning and end of the reconstructed signal.

EXAMPLE 7.4

Denoise the signal in Example 7.3 by eliminating samples in the two highest-frequency subbands that are below some threshold value. These samples will be set to zero. The threshold should be set to seven times the variance of the highpass subbands.

Solution

Decompose the signal in Example 7.3 using a four-level filter bank. The filter bank should use a four-element Daubechies filter. Examine the two highest-resolution highpass subbands. These subbands will reside in the last $N/4$ to N samples of the output of the analysis filter bank routine, `analyze`. Set all values in these segments that are *below* seven times the net variance of the subbands to 0.0.

```
% Example 7.4 Application of DWT to nonlinear filtering
%
.....same code as in Example 7.3 to construct signal.....
%
h0 = daub(4); % Daubechies filter
an = analyze(x,h0,4); % Decompose signal, level 4
%
threshold = var(an(N/4:N)); % Determine threshold
for k = (N/4:N) % Examine the two highpass subbands
    if abs(an(k)) < threshold
        an(k) = 0; % Zero samples below threshold
    end
end
y = synthesize(an,h0,4); % Reconstruct original signal
.....plot as in Example 7.3.....
```

Results

The original and reconstructed waveforms are shown in Figure 7.13. The filtering produced by thresholding the highpass subbands is evident. Also, there is no phase shift between the original and reconstructed signals due to the use of periodic convolution, although a small artifact is seen at the beginning and end of the data set. This is because the data set was not really periodic.

7.3.2.2 Discontinuity Detection

Wavelet analysis based on filter bank decomposition is particularly useful for detecting small discontinuities in a waveform. This feature is also useful in image processing. Example 7.5 shows the sensitivity of this method for detecting small changes, even when they are in the higher derivatives.

EXAMPLE 7.5

Load the signal x in file `ex7_5_data.mat`, which consists of two sinusoids to which a small step component (~1% of the amplitude) has been added. (An enlarged version of the step discontinuity is in variable `offset` also found in the data file.) This signal with its small offset was integrated twice to create a waveform with no apparent discontinuity. Plot the signal with the offset and apply a three-level analysis filter bank to the signal. Examine the high-frequency subband for evidence of the discontinuity.

Solution

Load the data file and plot the signal and discontinuity. Apply a three-level wavelet decomposition using the routine `analyze`. Note that this routine plots the subband waveforms.

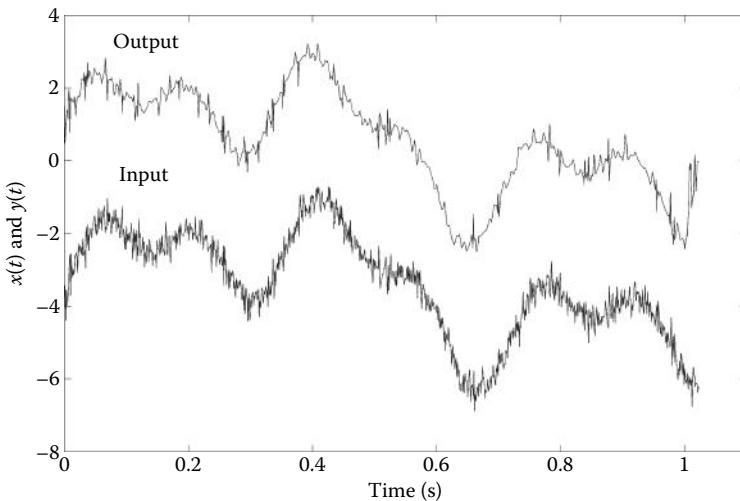


Figure 7.13 Application of the dyadic wavelet transform to nonlinear filtering. After subband decomposition using an analysis filter bank, a threshold process is applied to the two highest-resolution highpass subbands before reconstruction using a synthesis filter bank.

```
% Example 7.5 Discontinuity detection
%
load ex7_5_data; % Get data
plot(t,x,'k',t,offset-2.2,'k'); % Plot signal and offset
..... label and axis.....
%
figure;
h0 = daub(4); % Daubechies 4 - element filter
a = analyze(x,h0,3); % Analytic filter bank, level 3
```

Results

Figure 7.14 shows the waveform with a discontinuity in its second derivative at 0.5 s. The lower trace indicates the actual position of the discontinuity. Note that the discontinuity is not visible in the waveform. The output of the three-level analysis filter bank using the Daubechies four-element filter is shown in Figure 7.15. The position of the discontinuity is clearly visible as a spike in the highpass subbands.

7.4 Feature Detection: Wavelet Packets

The DWT can also be used to construct useful descriptors of a waveform. Since the DWT is a bilateral transform, all of the information in the original waveform must be contained in the subband signals. These subband signals, or some aspect of the subband signals such as their energy over a given time period, could provide a succinct description of some important aspect of the original signal.

In the decompositions described above, only the lowpass filter subband signals were sent on for further decomposition, giving rise to the filter bank structure shown in the upper half of Figure 7.16. This decomposition structure is also known as a *logarithmic tree*. However, other decomposition structures are valid, including the *complete* or *balanced tree* structure shown in the lower half of Figure 7.16. In this decomposition scheme, *both* highpass and lowpass

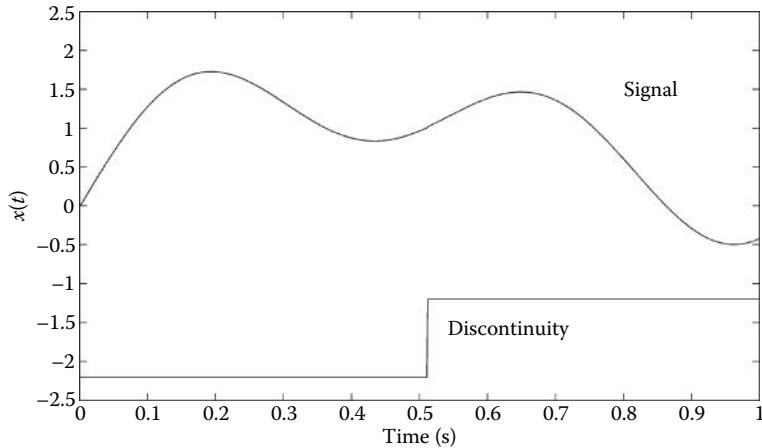


Figure 7.14 A waveform composed of two sine waves with an offset discontinuity in its second derivative at 0.5 s. Note that the discontinuity is not apparent.

subbands are further decomposed into highpass and lowpass subbands up until the terminal signals. Other more flexible tree structures are possible, where a decision on further decomposition (whether or not to split a subband signal) depends on the activity of a given subband. The scaling functions and wavelets associated with such general tree structures are known as *wavelet packets*.

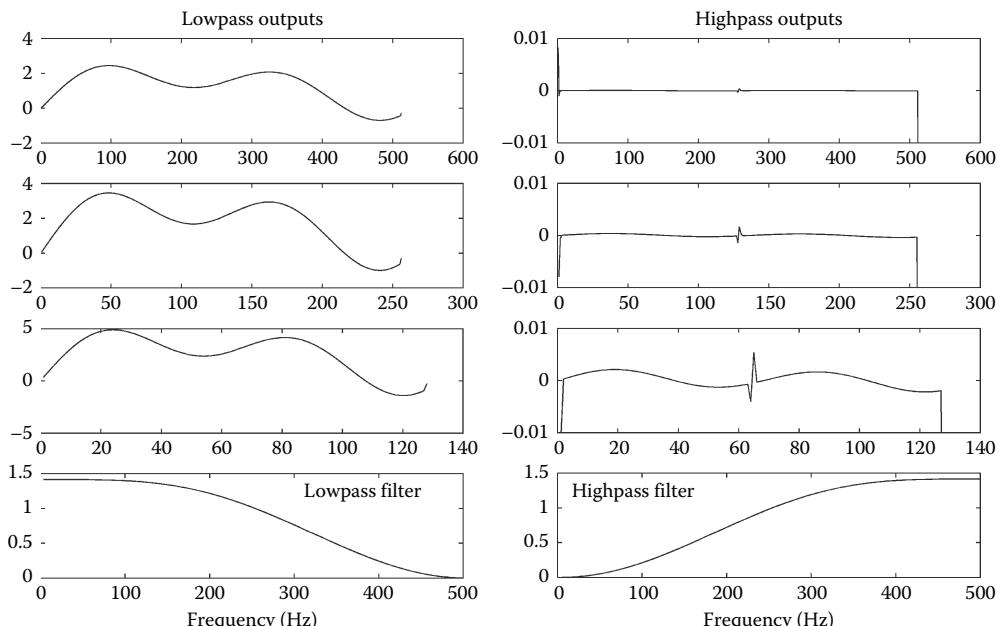


Figure 7.15 Analysis filter bank output of the signal shown in Figure 7.13. Although the discontinuity is not visible in the original signal, its presence and location are clearly identified as a spike in the highpass subbands.

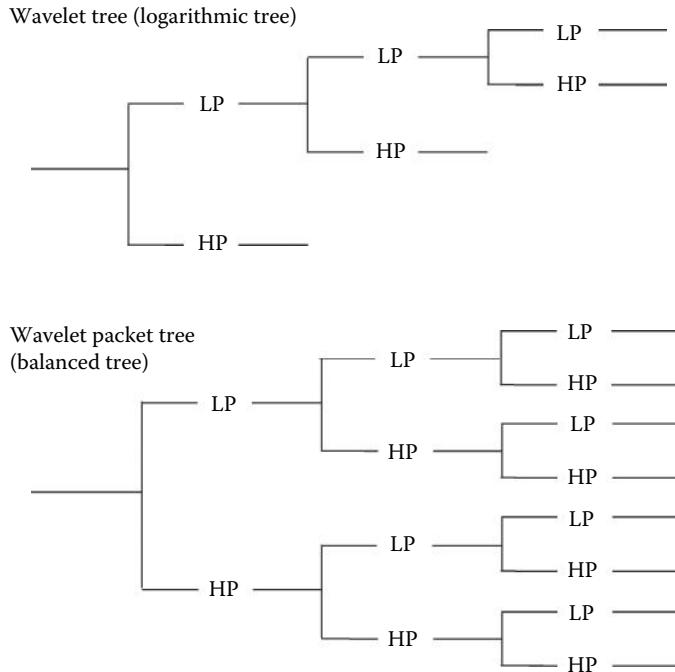


Figure 7.16 Two common tree structures used in wavelet decomposition. In the upper logarithmic tree, only lowpass components are further decomposed. This is the structure implemented in the examples above. In the lower wavelet packet, or balanced tree decomposition, all nonterminal signals are decomposed into highpass and lowpass subbands. This tree structure produces redundant signals and is used primarily for signal analysis.

EXAMPLE 7.6

Apply the wavelet packet decomposition to the waveform consisting of a mixture of four equal-amplitude sinusoids of 1, 12, 44, and 200 Hz. Use a four-element Daubechies filter.

Solution

The main routine in this example is similar to that used in Examples 7.3 and 7.4 except that it calls the balanced tree decomposition routine `w_packet` and plots out the terminal waveforms. The `w_packet` routine is shown below and is used in this example to implement a three-level decomposition, as illustrated in the lower half of Figure 7.16. This will lead to eight output segments that are stored sequentially in the output vector, `an`.

```
% Example 7.6 Example of 'Balance Tree Decomposition'
%
fs = 1000; % Sample frequency
N = 1024; % Number of points in waveform
levels = 3 % Number of decomposition levels
nu_seg = 2^levels; % Number of decomposed segments
freqsin = [1 12 44 200]; % Sinusoid frequencies
ampl = [1 1 1 1]; % Sinusoid amplitudes
[x t] = signal(freqsin, ampl, N); % Construct signal
%
```

Biosignal and Medical Image Processing

```
h0 = daub(10); % Daubechies 10th order
a = w_packet(x,h0,levels); % Wavelet packet analysis
for k = 1:nu_seg % Plot packets
    i_s = 1 + (N/nu_seg) * (k-1); % Location of this subband
    a_p = a(i_s:i_s + (N/nu_seg)-1); % Get subband signal
    subplot(nu_seg/2,2,i); % Plot decompositions
    plot((1:N/nu_seg),a_p,'k');
    .....labels.....
end
```

The balance tree decomposition routine, `w_packet`, operates similarly to the DWT analysis filter banks, except for the overall structure. At each level, signals from the previous level are isolated, filtered (using standard convolution), downsampled, and *both* the highpass and lowpass signals overwrite the single signal from the previous level. At the first level, the input waveform is replaced by the filtered, downsampled highpass and lowpass signals. At the second level, the highpass and lowpass signals are replaced by filtered, downsampled highpass and lowpass signals. After the second level, there are now four sequential signals in the original data array, and after the third level, there are eight.

```
function an = w_packet(x,h0,L)
% Function to generate a ''Balanced Tree'' Filter Bank
.....comments.....
lf = length(h0); % Filter length
lx = length(x); % Data length
an = x; % Initialize output
% Calculate High pass coefficients from low pass coefficients
for i = 0:(lf-1)
    h1(i+1) = (-1)^i * h0(lf-i);
end
%
% Calculate filter outputs for all levels
for k = 1:L
    nu_low = 2^(k-1); % Num. lowpass filters this level
    l_seg = lx/2^(k-1); % Length of each data seg. this level
    for j = 1:nu_low;
        i_start = 1 + l_seg * (j-1); % Location current segment
        a_seg = an(i_start:i_start + l_seg-1);
        lpf = conv(a_seg,h0); % Low pass filter
        hpf = conv(a_seg,h1); % High pass filter
        lpf = lpf(1:2:l_seg); % Downsample
        hpf = hpf(1:2:l_seg);
        an(i_start:i_start + l_seg-1) = [lpf hpf];
    end
end
```

Result

The output produced by this decomposition is shown in Figure 7.17. The filter bank outputs emphasize various components of the three-sine mixture.

The balanced tree decomposition is also useful for defining features of a signal that can later be used with classification techniques (see Chapters 16 and 17) to identify signal characteristics associated with various disease states. Such features can be determined from the wave packet subbands. In the final example, the balanced tree decomposition is applied to two ECG signals and distinguishing features are constructed from the RMS values of the subband signals. This is an example of classification developed in Chapters 16 and 17.

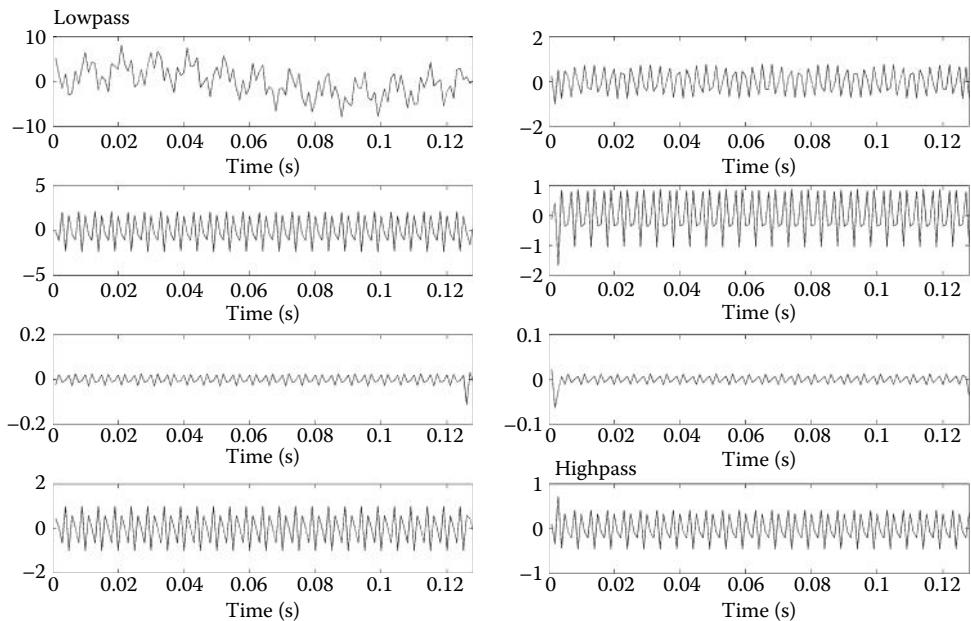


Figure 7.17 Balanced tree or wavelet packet decomposition of a waveform consisting of four equal-amplitude sinusoids at 1, 12, 44, and 200 Hz. The signal from the upper left plot has been lowpass filtered three times and represents the highest terminal in Figure 7.16. The upper right signal has been lowpass filtered twice and highpass filtered once and represents the second highest terminal in Figure 7.16. The rest of the plots follow sequentially with the bottom right plot corresponding to the lowest terminal branch. This signal has been highpass filtered three times.

EXAMPLE 7.7

Apply the balance tree decomposition to two EEG signals taken at different times from the same subject. The file `ecg1.mat` contains two 60-s segments of ECG data in variables `ecg1` and `ecg2` ($f_s = 250$ Hz). Decompose each segment using a three-level balanced tree decomposition. Take as features the RMS value of each decomposed subband. Find two features (i.e., RMS values) that best differentiate between the two signals, and plot the values of these two features against each other on a 2-D plot. Observe the broad separation between the two feature combinations. Use a Daubechies 10-element filter.

Solution

Load the data file `ecg1.mat` that contains the ECG signals and apply three-level balanced tree decomposition using the same code in Example 7.6 to each of the variables. Modify the code to compute and save the RMS values of each of the eight subband outputs. Compare these features against one another and compute the percent difference between sets of features. Find the two features that give the largest difference and plot the feature combination as points on a 2-D plot.

```
Example 7.7 Feature extraction using Balanced Tree Decomposition
%
load ecg1_data.mat; % Get data
levels = 3; % Number of levels for decomposition
nu_seg = 2^levels; % Number of decomposed segments
N = length(ecg1); % Data length
```

Biosignal and Medical Image Processing

```
t = (1:N/nu_seg)/fs; % Time vector for plotting
h0 = daub(10); % Get filter coefficients: Daubechies 10
%
a = w_packet(ecg1,h0,levels); % Decompose signal, Balanced Tree
for i = 1:nu_seg
    i_s = 1 + (N/nu_seg) * (i - 1); % Location for this segment
    a_p = a(i_s:i_s + (N/nu_seg) - 1); % Get subband signal
    feature1(i) = sqrt(mean(a_p.^2)); % RMS value
    .....plot and label subband.....
end
.....Repeat the above using ecg2 to construct feature2.....
% Determine percent difference between features
feature_diff = 100 * abs(feature1 - feature2)./feature2;
disp(['feature1' feature2' feature_diff'])
% Output features and differnce
```

Results

This example displays the following feature values and their percent difference:

Feature Number	Ecg1	Ecg2	Percent Difference (%)
1	306.7	331.3	7.4
2	112.9	127.4	11.4
3	24.1	29.4	18.1
4	52.5	56.6	7.2
5	4.6	4.8	3.7
6	6.0	6.3	4.1
7	16.1	19.7	18.0
8	11.0	11.3	2.8

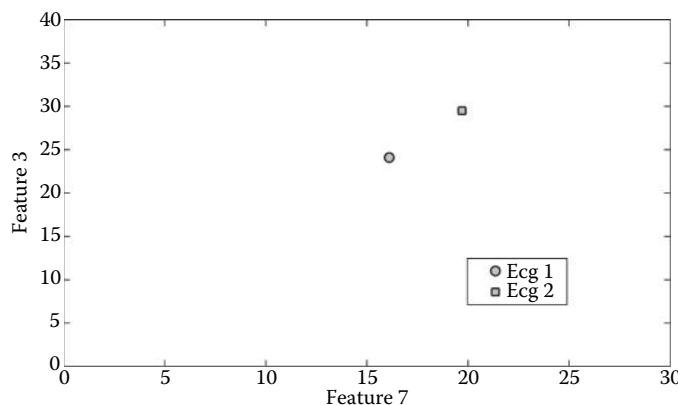


Figure 7.18 A plot of the value of two features extracted from two different ECG signals using balanced tree decomposition. The features are the RMS values of two subbands after three-level decomposition. If these two signals represent different states of the cardiovascular system, such states could be determined from other ECG signals by comparing the value of their features with those plotted here.

The biggest differences are seen in Features 3 and 7. A plot combining these two features is shown in Figure 7.18. Assuming the two signals, `ecg1` and `ecg2`, represent two different states of the cardiovascular system, these features could be used to characterize these states. In a typical classification scenario, other ECG signals would be used to identify the state of the cardiovascular system with respect to these two states by comparing their features with those of Figure 7.18. This is done in one of the problems.

7.5 Summary

Wavelet analysis includes two quite different approaches: the CWT and the DWT. Although it carries the word “continuous,” the CWT is implemented in the discrete domain, just as is the DWT. The transform applies a family of short waveforms called “wavelets” to the signal of interest using cross-correlation, or convolution if the wavelet is symmetrical. The wavelet family is constructed from a base waveform, called the “mother wavelet,” that is then stretched or compressed in time to make up the various family members. As the wavelet is compressed, it probes higher frequencies, but since it is shorter, it also has a higher time resolution. The CWT is not immune to the time–frequency uncertainty principle, but features an automatic adjustment that modifies the time–frequency trade-off depending on the frequency being probed. Common wavelets consist of a periodic waveform such as a sinusoid that is “localized” (i.e., constrained in time) by a Gaussian function. Because the transform produces many redundant signals, it is used only in analysis where reconstruction of the original waveform is not an issue.

The DWT can be employed in a configuration that produces the same number of samples in the transformed data set as in the original signal. In this situation, it is suitable for operations where the original signal is reconstructed from the transformed data set. Although the DWT can be defined and implemented using analytical equations, in the real world, it is implemented using filter banks: groups of FIR filters that receive the same input signal and generate different signal decompositions termed subbands. DWT filter banks consist of paired lowpass and highpass filters. These filter pairs are designed to produce orthogonal subbands and each filter is invertible; that is, it is possible to recover the unfiltered signal using another filter.

The initial filter banks in a DWT system are used to decompose the signal into subbands. Filters that decompose the signal are known as analysis filters. Often, analysis filter pairs are arranged in a logarithmic tree where only the output of the lowpass of the pair is sent on to subsequent filter pairs. After each filter’s output is downsampled by 2, the filter outputs have the same data length as the original signal. Downsampling by 2 has the effect of decreasing the sample frequency by 2, but the filtering prevents aliasing that might otherwise occur with decreasing sample frequency.

After some operation is performed on the decomposed subbands, the signal can be reconstructed by a set of synthesis filter banks. Each subband signal is upsampled by adding zeros between samples and passed to a synthesis filter. These filters smooth the signal, essentially filling in the zeroed samples. Lowpass and highpass filters, again arranged in pairs, combine their outputs in a configuration that is the reverse of the analysis filter’s arrangement. After all the filter outputs have been combined, a reconstructed signal is produced. One of the most popular applications of the logarithmic DWT is in data compression, particularly of images. Since this application is rarely used in biomedical engineering (although there are some applications in the transmission of radiographic images), it is covered in only one of the problems.

Another possible configuration of the analysis filter bank is the balanced or wavelet packet tree. In this configuration, the analysis filters are again grouped into pairs, but the output of *both* the lowpass and highpass filters are sent on to subsequent filter pairs. This produces redundant subband signals, so it is used only for analysis. Often, some measurement is made on the subband signals to produce a set of signal features. These signal features may be useful

Biosignal and Medical Image Processing

in classifying the signal with regard to disease states (see Chapters 16 and 17 on classification techniques).

PROBLEMS

- 7.1 Plot the magnitude spectra of the Mexican Hat and Morlet wavelets. Use $f_s = 100$ Hz and 1024 samples for half the wavelet (so for the full wavelet, $N = 2048$). Scale the horizontal axis to be between 0 and 5 Hz.
- 7.2 Modify Example 7.2 to compare the time–frequency resolution of three different functions: the Morlet wavelet, the Shannon wavelet, and a Gaussian function. Plot wavelet boundaries for the same values of a used in Example 7.2 and determine the product of the two time and frequency resolutions. The equations for the Shannon wavelet and Gaussian function are

$$\Psi_s(t) = 2 \operatorname{sinc}(t/2) \cos(3\pi t/2) \text{ Shannon wavelet}$$

$$\Psi_G(t) = \frac{1}{\sqrt{2\pi}} e^{-0.5t^2} \text{ Gaussian function}$$

Note the difference in the time–frequency product. In theory, the Gaussian function has a time–frequency resolution that equals the minimum value of 0.5.

- 7.3 Apply the CWT used in Example 7.1 to analyze a chirp signal with frequencies between 2 and 30 Hz over a 5-s period. Use the same range of a : from 1 to 1/120. Assume $f_s = 200$ Hz as in Example 7.1. Use the Morlet wavelet and show both contour and 3-D plots. Note from the contour plot the change in resolution as the frequency increases.
- 7.4 Repeat Problem 7.3 using the signal x in file `time_freq5.mat`. This signal combines the chirp signal in Problem 7.3 with noise (SNR = -3 dB). Assume $f_s = 200$ Hz. Use the Morlet wavelet. Note the severe degradation of the resulting time-scale plot.
- 7.5 Load the file `time_freq4.mat` with signal x that contains two sinusoids closely spaced in frequency and in time; specifically, they are 0.1 s and 5 Hz apart ($f_s = 500$ Hz). Apply the CWT using the Morlet wavelet with an a that ranges between 1 and 1/200. Repeat for the Mexican Hat wavelet. Plot only the contour plots. Taking the log of the scale variable, a , before plotting will improve the visibility of the two frequencies. Note that they both give excellent time resolution with a noticeable gap between the two frequencies. Also note that one of the wavelets shows slightly greater differences between the two closely spaced frequencies.
- 7.6 Load the file `time_freq6.mat` containing an unknown signal x ($f_s = 100$ Hz). Apply the CWT using the Mexican Hat wavelet with an a that ranges between 1 and 1/200. Repeat for the Mexican Hat wavelet. Plot only the contour plots. As in Problem 7.5, taking the log of the scale variable, a , before plotting will improve the visibility of the signal. Qualitatively describe the nature of the signal. Which wavelet best shows the signal?
- 7.7 Plot the frequency characteristics (magnitude and phase) of the Haar, Daubechies four-element, and Daubechies 10-element filters. Assume $f_s = 100$ Hz and pad the Fourier transform to an appropriate number of samples. Plot the phase in degrees and use the `unwrap` routine if necessary. Include the filter name with the plots. Note the differences in slope of the three filters.

- 7.8 (a) Generate a Daubechies 10-element filter and plot the magnitude spectrum as in Problem 7.7. Construct the highpass filter using the alternating flip algorithm (Equation 7.20), and plot its magnitude spectrum on the same plot as the lowpass spectrum. Generate the lowpass and highpass *synthesis* filter coefficients using the order flip algorithm (Equation 7.22), and plot their respective magnitude spectra, again on the same plot. Assume $f_s = 100$ Hz and pad the Fourier transform to an appropriate number of samples. (b) Repeat this problem using the Haar filter.
- 7.9 Show that the outputs of analysis lowpass and highpass filters are orthogonal. Generate a Daubechies 10-element lowpass filter using `daub(10)`, then construct the highpass filter using the alternating flip algorithm (Equation 7.20). Pass the same signal through both filters and show that the correlation between the two signals is very low (Equation 2.35). Use MATLAB's `corrcoef` to calculate the correlation and, for a signal, use the signal, \mathbf{x} , in file `sin_sig.mat`, which consists of a mixture of sinusoids.
- 7.10 Construct a waveform of a chirp signal as in Problem 7.2 but add noise. Make the variance of the noise equal to the variance of the chirp signal. Decompose the waveform into four levels, operate on the highest resolution (i.e., the high-resolution highpass signal), then reconstruct. The operation should zero all elements below a given threshold. Find the best threshold. Plot the signal before and after reconstruction. Use Daubechies six-element filter. Would you get the same results if you used only three decomposition levels? Why?
- 7.11 Data compression. Construct a waveform consisting of three equal-amplitude sinusoids of frequencies 20, 100, and 200 Hz. Take the DWT and remove the entire “detail” signal by setting the upper half of the output of `analyze` to zero. This removes half the points required for transmission. Plot the waveform before and after this compression in both the time and frequency domains. Note that while the time plots are similar there are added frequency components as this is not a linear operation.
- 7.12 Discontinuity detection. Load the waveform \mathbf{x} in file `discontinuity_data.mat`, which consists of a signal composed of two sinusoids with a series of diminishing discontinuities in the second derivative. The discontinuities in the second derivative begin at $\sim 0.5\%$ of the sinusoidal amplitude and decrease by a factor of 2 for each pair of discontinuities. Plot the input signal and observe that it contains no obvious discontinuities. Decompose the waveform into three levels and examine and plot only the highest-resolution highpass filter output to detect the discontinuity. [Hint: This filter's output will be located in $N/2$ to N of the analysis output array.] Use a Haar and a Daubechies 10-element filter and compare the difference in detectability. Note that the Haar is a weak filter, so some of the low-frequency components will still be found in its output. You may need to increase the y -axis scale of the Daubechies filter to best visualize the discontinuities.
- 7.13 Apply the balanced tree decomposition to a chirp signal similar to that used in Problem 7.3 except that the chirp frequency should range between 2 and 100 Hz. Decompose the waveform into three levels and plot the outputs at the terminal level as in Example 7.5. Use a Daubechies four-element filter. Note that each output filter responds to different portions of the chirp signal.
- 7.14 Example of balance tree decomposition applied to EEG signals. The file `HR_pre.mat` contains a heart rate signal, $\mathbf{hr_pre}$ taken under normal conditions, while the file `HR_med.mat` contains a heart rate signal $\mathbf{hr_med}$ taken under meditative conditions ($f_s = 250$ Hz). Decompose each signal using a three-level balanced tree composition. As in Example 7.7, take as features the RMS value of each decomposed subband. Following the approach in Example 7.7, find two features (RMS values) that

Biosignal and Medical Image Processing

best differentiate between the two signals and plot the features from each segment on a 2-D plot. Use a Daubechies six-element filter.

After establishing the definitive features, load the file `Hr_ unknown.mat`, which contains a heart rate signal, `hr_ unknown`, taken under an unknown condition. Use the balanced tree to find the two features used above and plot the feature values on the same plot as the features for `hr_ pre` and `hr_ med`. What is the most likely condition under which the heart rate signal `hr_ unknown` was acquired: normal or meditative? This problem is a crude example of classification explored in detail in Chapters 15 and 16. [Note: the signals `hr_ pre`, `hr_ med` and `hr_ unknown` may not be the same length.]

8

Optimal and Adaptive Filters

8.1 Optimal Signal Processing: Wiener Filters

The FIR and IIR filters described in Chapter 4 provide considerable flexibility in altering the frequency content of a signal. Coupled with MATLAB filter design tools, these filters can provide almost any desired frequency characteristic to nearly any degree of accuracy. The actual frequency characteristics attained by the various design routines can be verified through Fourier transform analysis. However, these design routines do not tell the user what frequency characteristics are best for any given situation, that is, what type of filtering will most effectively separate out the signal from noise. That decision is often made based on the user's knowledge of signal or source properties, or by trial and error. Optimal filter theory was developed to provide structure for the process of selecting the most appropriate filter frequency characteristics.

A wide range of different approaches can be used to develop an optimal filter depending on the nature of the problem, specifically, what and how much is known about signal and noise characteristics. If a representation of the desired signal is available, then a well-developed and popular class of filters known as *Wiener filters* can be applied. The basic concept behind Wiener filter theory is to minimize the difference between the filtered output and some desired output. This minimization is based on the least mean square (LMS) approach, which adjusts the filter coefficients to reduce the square of the difference between the desired and actual waveform after filtering. This approach requires an estimate of the desired signal that must somehow be constructed and this estimation is usually the most challenging aspect of the problem.*

The Wiener filter approach is outlined in Figure 8.1. The input waveform containing both signal and noise is operated by a linear process, $H(z)$. The Wiener filter problem is similar to the model-based spectral methods of Chapter 5 in that we are trying to find a linear process whose output matches some desired output. For example, in the AR model method, the desired response, $d[n]$, is the signal to be analyzed and the input to the linear process, $x[n]$, is white noise. After the best match is obtained (i.e., the sum of $e^2[n]$ is minimized), the parameters of the linear process are used as an estimate of the spectrum. This similarity in approach is reflected in the Wiener filter equations shown below.

In practice, the process could be any of the models discussed in Chapter 5: an MA process that is the same as an FIR filter, an AR process, or an ARMA that is the same as an IIR filter. FIR filters

* In principle, only the cross-correlation between the unfiltered and the desired output is necessary for the application of these filters, but that usually means having the desired output available.

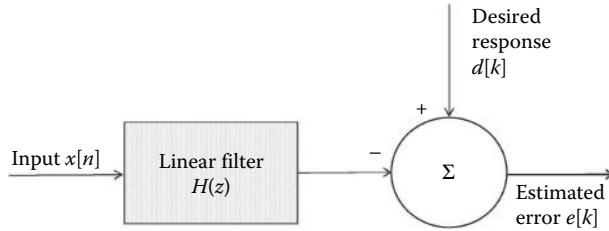


Figure 8.1 Basic arrangement of signals and processes in a Wiener filter.

are generally more popular as they are inherently stable* and our discussion here is limited to the use of FIR filters. FIR filters have only numerator terms in the transfer function (i.e., only zeros) and these filters can be implemented using convolution. Repeating the convolution equation:

$$y[k] = \sum_{n=0}^{L-1} b[n]x[k-n] \quad (8.1)$$

where $b[n]$ is the impulse response of the linear filter and $x[n]$ is the input signal. The output of the filter, $y[k]$, can be thought of as an estimate of the desired signal, $d[k]$ (Figure 8.1). The difference between the estimated and the desired signal can be determined by simple subtraction: $e[k] = d[k] - y[k]$.

As mentioned above, the LMS algorithm is used to minimize the error signal: $e[k] = d[k] - y[k]$. Note that $y[n]$ is the output of the linear filter, $H[z]$. Since we are limiting our analysis to FIR filters, $y[k]$ is the convolution of $b[n]$ with $x[n]$ and the equation for the error, $e[k]^2$, becomes

$$e[k] = d[k] - y[k] = d[k] - \sum_{n=0}^{L-1} b[n]x[k-n] \quad (8.2)$$

where L is the length of the FIR filter. Note that the summation is from 0 to $L-1$, which is correct mathematically, but goes from 1 to L in MATLAB notation. The sum of $e[k]^2$ becomes

$$\sum_{k=0}^{K-1} e^2[k] = \sum_{k=0}^{K-1} \left[d[k] - \sum_{n=0}^{L-1} b[n]x[k-n] \right]^2 \quad (8.3)$$

Squaring and rearranging the double summation that results:

$$\sum_{k=0}^{K-1} e^2[k] = \sum_{k=0}^{K-1} \left[d^2[k] - 2d[k] \sum_{n=0}^{L-1} b[n]x[k-n] + \sum_{n=0}^{L-1} \sum_{\ell=0}^{L-1} b[n]b[\ell]x[k-n]x[k-\ell] \right] \quad (8.4)$$

$$\sum_{k=0}^{K-1} e^2[k] = \sum_{k=0}^{K-1} d^2[k] - 2 \sum_{k=0}^{K-1} \sum_{n=0}^{L-1} b[n]d[k]x[k-n] + \sum_{k=0}^{K-1} \sum_{n=0}^{L-1} \sum_{\ell=0}^{L-1} b[n]b[\ell]x[k-n]x[k-\ell] \quad (8.4)$$

Recall that the definition of cross-correlation is $r_{dx}[n] = \sum_{k=0}^{K-1} d[k]x[k+n]$. Substituting $-n$ for n in the cross-correlation function:

$$r_{dx}[-n] = \sum_{k=0}^{K-1} d[k]x[k-n] \quad (8.5)$$

* IIR filters contain internal feedback paths and can oscillate with certain parameter combinations.

8.1 Optimal Signal Processing

Substituting this into the second term and noting that because of the symmetry property of cross-correlation, $r_{dx}[-n] = r_{xd}[n]$, Equation 8.4 becomes

$$\sum_{k=0}^{K-1} e^2[k] = \sum_{k=0}^{K-1} d^2[k] - 2 \sum_{n=0}^{L-1} b[n] r_{xd}[n] + \sum_{k=0}^{K-1} \sum_{n=0}^{L-1} \sum_{\ell=0}^{L-1} b[n] b[\ell] x[k-n] x[k-\ell] \quad (8.6)$$

In the third term, note that the summation over k of $x[k-n] x[k-\ell]$ can also be written with a change of variable of $m = k - n$ as

$$\sum_{k=0}^{K-1} x[k-n] x[k-\ell] = \sum_{m=-n}^{K-1-n} x[m] x[m+n-\ell] = r_{xx}[n-\ell] \quad (8.7)$$

So, Equation 8.6 becomes

$$\sum_{k=0}^{K-1} e^2[k] = \sum_{k=0}^{K-1} d^2[n] - 2 \sum_{n=0}^{L-1} b[n] r_{xd}[n] + \sum_{n=0}^{L-1} \sum_{\ell=0}^{L-1} b[n] b[\ell] r_{xx}[n-\ell] \quad (8.8)$$

To minimize the squared error with respect to the FIR filter coefficients, take the partial derivative of Equation 8.8 with respect to $b[n]$ and set it to zero:

$$\frac{\partial \sum_{k=1}^K e^2[k]}{\partial b[n]} = 0 \quad (8.9)$$

which leads to

$$-2r_{xd}[n] + 2 \sum_{n=0}^{L-1} b[n] r_{xx}[n-\ell] = 0 \quad (8.10)$$

$$\sum_{n=0}^{L-1} b[n] r_{xx}[n-\ell] = r_{xd}[n] \quad \text{for } 0 \leq \ell \leq L-1 \quad (8.11)$$

Equation 8.11 shows that the optimal filter can be derived by knowing only the autocorrelation function of the input (r_{xx} in Equation 8.11) and the cross-correlation function between the input and desired waveform (r_{xd} in Equation 8.11). In principle, the actual functions are not necessary, only the auto- and cross-correlations; however, in most practical situations, the auto- and cross-correlations are derived from the actual signals, in which case, some representation of the desired signal is required.

To solve for the FIR coefficients in Equation 8.11, we note that this equation actually represents a series of L equations that must be solved simultaneously. Allowing the indexes to range from 1 to L to be compatible with MATLAB, the matrix expression for these simultaneous equations is

$$\begin{bmatrix} r_{xx}[1] & r_{xx}[2] & \dots & r_{xx}[L] \\ r_{xx}[2] & r_{xx}[1] & \dots & r_{xx}[L-1] \\ \vdots & \vdots & \ddots & \vdots \\ r_{xx}[L] & r_{xx}[L-1] & \dots & r_{xx}[1] \end{bmatrix} \begin{bmatrix} b[1] \\ b[2] \\ \vdots \\ b[L] \end{bmatrix} = \begin{bmatrix} r_{dx}[1] \\ r_{dx}[2] \\ \vdots \\ r_{dx}[L] \end{bmatrix} \quad (8.12)$$

Equation 8.12 is commonly known as the *Wiener–Hopf* equation and is a basic component of Wiener filter theory. The equation is similar in the overall structure to the Yule–Walker equation

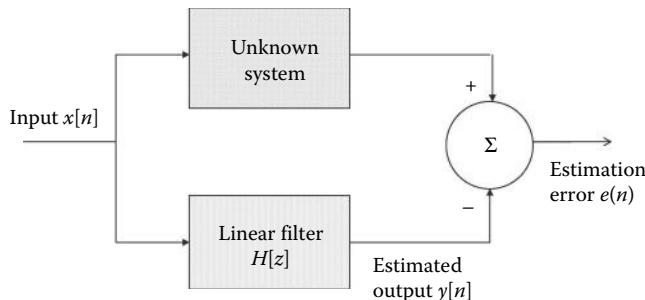


Figure 8.2 Configuration for using optimal filter theory for systems identification.

in Chapter 5 (Equation 5.11) for determining the AR model. The matrix in the equation is the autocorrelation matrix (or just correlation matrix) that is also found in the Yule–Walker equation. The equation can be written more succinctly using standard matrix notation and the FIR coefficients can be obtained by solving the equation through matrix inversion:

$$Rb = r_{xy} \text{ and the solution is } b = R^{-1}r_{xy} \quad (8.13)$$

The MATLAB solution to this equation is very similar to the solution for the Yule–Walker equations; it uses the MATLAB routines `xcorr`, and either `toeplitz` or `corrmtx`. Two examples are presented in Section 8.1.1.

The Wiener–Hopf approach has a number of other applications in addition to standard filtering, including systems identification, interference canceling, and inverse modeling or deconvolution. For system identification, the filter is placed in parallel with the unknown system as shown in Figure 8.2. In this application, the desired output is the output of the unknown system and the filter coefficients are adjusted so that the filter's output best matches that of the unknown system. An example of this application is given in Example 8.2 and in Problems 8.5 and 8.6. In interference canceling, the desired signal contains both signal and noise whereas the filter input is a reference signal that contains only noise or a signal correlated with the noise. This application is explored in Section 8.2 on adaptive filtering since it is more commonly implemented in that context.

8.1.1 MATLAB Implementation

The Wiener–Hopf equation (Equations 8.12 and 8.13) can be solved using MATLAB's matrix left divide operator ('\ \backslash ') as is done in Chapter 5 and in the examples below. In Chapter 2, the use of the `toeplitz` routine in conjunction with `xcorr` to set up the correlation matrix is described, along with the alternative approach using `corrmtx`. For matrices to be left divided, they must be nonsingular, that is, the rows and columns must be independent. Because of the structure of the correlation matrix in Equation 8.12 (termed *positive definite*), it cannot be singular. However, it can be near singular: some rows or columns might be slightly independent. Such an *ill-conditioned* matrix will lead to large errors when it is divided. The MATLAB ' \backslash ' matrix left divide operator provides an error message if the metrics are not well conditioned, but this can be more effectively evaluated using the MATLAB `cond` function:

```
c = cond(X)
```

where X is the matrix under test and c is the ratio of the largest to smallest singular values. A very well-conditioned matrix would have singular values in the same general range; so, the output variable, c , would be close to one. Very large values of c indicate an ill-conditioned matrix. Values $> 10^4$ have been suggested by Stearns and David (1996) as too large to produce reliable

results in the Wiener–Hopf equation. When this occurs, the condition of the matrix can usually be improved by reducing its dimension, that is, reducing the range, L , of the autocorrelation function in Equation 8.12. This will also reduce the number of filter coefficients in the solution.

EXAMPLE 8.1

Given a sinusoidal signal in noise (SNR = –8 dB), design an optimal filter using the Wiener–Hopf equation. Assume that you have a copy of the actual signal available, in other words, a version of the signal without the added noise. In general, this will not be the case: if you had the desired signal, you would not need the filter! In practical situations, you would have to estimate the desired signal or the cross-correlation between the estimated and desired signals. Some other techniques for producing the desired signal are given in the next section.

Solution

After generating the signal using `sig_noise`, the program below uses the routine `wiener_hopf` (shown below) to determine the optimal filter coefficients. These are then applied to the noisy waveform using MATLAB's `filter` routine introduced in Chapter 4, although convolution can also be used.

```
% Example 8.1 Example of Wiener Filter Theory
N = 1024; % Number of points
fs = 1000; % Sample frequency
L = 245; % Filter length
%
% Generate signal and noise data: (SNR = -8 dB)
[xn, t, x] = sig_noise(10,-8,N); % x is noise free (desired) signal
.....plot signal,labels, table, axis.....
%
b= wiener_hopf(xn,x,L); % Apply Wiener-Hopf
y=filter(b,1,xn); % Filter data using optimum filter weights
..... Plot filtered data, label, axis.....
%
h=abs(fft(b,256)).^2 % Calculate filter PS
..... Plot spectrum, label, axis.....
```

The function `wiener_hopf` solves the Wiener–Hopf equations:

```
function b = wiener_hopf(x,y,maxlags)
% Function to compute Wiener-Hopf equations
.....comments.....
%
rxx = xcorr(x,maxlags); % Autocorrelation
rxx = rxx(maxlags + 1:end)'; % Positive lags only
rxy = xcorr(x,y,maxlags); % Crosscorrelation
rxy = rxy(maxlags + 1:end)'; % Positive lags only
rxx_matrix = toeplitz(rxx); % Correlation matrix
b = rxx_matrix\rxy; % Eq. 8.12
% The Levinson recursion could be used here instead of '\'
```

Results

Example 8.1 generates Figure 8.3. The power spectrum of the filter coefficients shows that the optimal filter approach, when applied to a single sinusoid buried in noise, produces a bandpass filter with a peak at the sinusoidal frequency. An equivalent or even more effective filter could have been designed using the tools presented in Chapter 4. Indeed, such a statement could also be made about any of the adaptive filters described below. However, this requires precise *a priori*

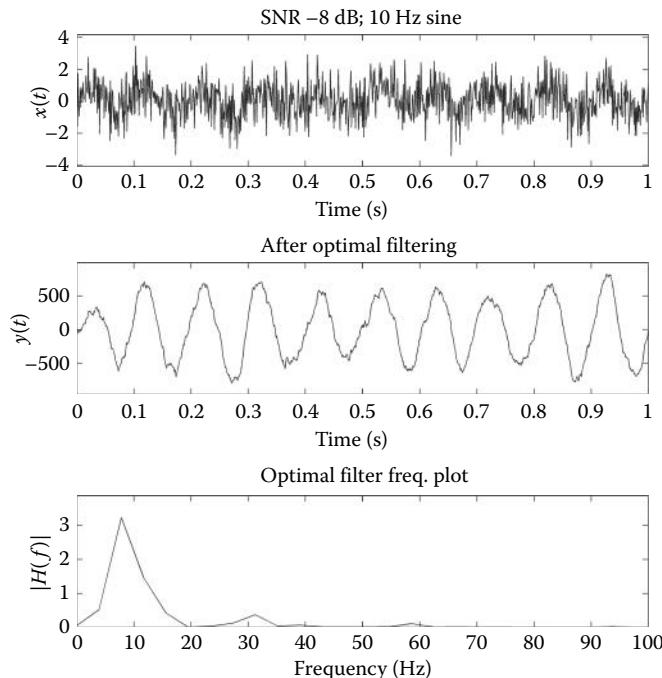


Figure 8.3 The Wiener filter applied to a signal consisting of a 10-Hz sine wave with an SNR of -8 dB. The filter coefficients found by the filter create a bandpass filter centered at 10 Hz.

knowledge of the signal and noise frequency characteristics that may not be available. Moreover, a fixed filter will not be able to behave optimally if the signal and noise characteristics change over time.

EXAMPLE 8.2

Apply the LMS algorithm to a systems identification task. The “unknown” system will be an all-zero linear process with a digital transfer function of

$$H(z) = 0.5 + 0.75z^{-1} + 1.2 z^{-2}$$

Confirm the match by plotting the magnitude of the transfer function for both unknown and matching systems. Since the Wiener filter approach uses an all-zero FIR filter as the matching system that is similar in structure to the unknown system, the match should be quite good. In Problem 8.6, this approach is repeated but for an unknown system that has both poles and zeros. In this case, the FIR (all-zero) filter will need many more coefficients to produce a reasonable match.

Solution

The program below inputs random noise into both the unknown process and the matching filter. In this case, the desired signal is the output of the unknown system that is simulated using convolution. In a practical situation, the unknown process would be a real system and the desired signal would be the output of this system to a random noise input signal. Since the FIR matching filter cannot easily accommodate a pure time delay, care must be taken to compensate for possible time shifts due to the convolution operation. The matching filter coefficients are adjusted using the Wiener–Hopf equation described previously. Frequency characteristics of

8.1 Optimal Signal Processing

both the unknown and matching system are determined by applying the FFT to the coefficients of both processes and the resultant spectra are plotted.

```
% Example 8.2 System Identification
%
fs = 500; % Sampling frequency
N = 1024; % Number of points
L = 8; % Optimal filter order
%
% Generate unknown system and noise input
b_unknown = [.5 .75 1.2]; % Define unknown process
xn = randn(1,N); % Gaussian random numbers
xd = conv(b_unknown,xn); % Simulate unknown output
xd = xd(3:N+2); % Compensate for conv delay.
%
% Apply Wiener filter
b= wiener_hopf(xn,xd,L); % Compute matching coefficients
%
% Calculate power spectra using the FFT
ps_match = (abs(fft(b,N))).^2;
ps_unknown = (abs(fft(b_unknown,N))).^2;
.....Plot frequency characteristics of both systems
.....labels, table, axis.....
```

Results

The output plots from this example are shown in Figure 8.4. Note the close match in spectral characteristics between the “unknown” process and the matching output produced by the Wiener–Hopf algorithm. The transfer functions also closely match, as is seen by the similarity

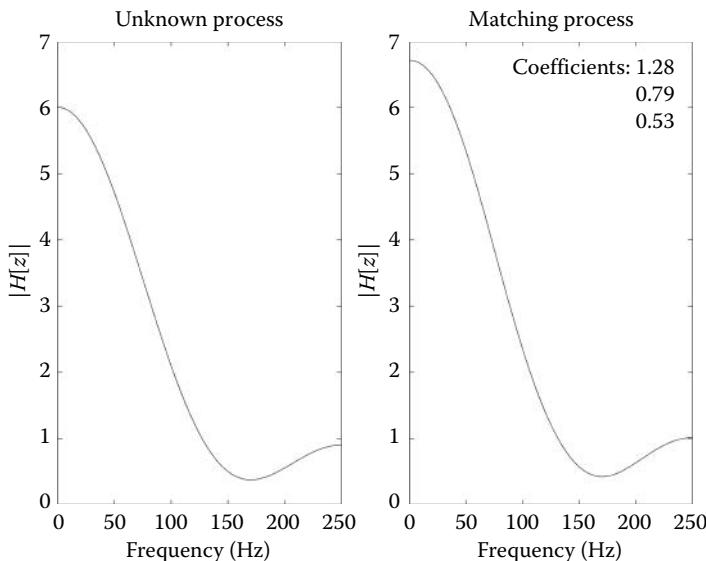


Figure 8.4 Frequency characteristics of an “unknown” process having numerator coefficients of 0.5, 0.75, and 1.2 (an all-zero process). The matching process uses system identification implemented with the Wiener–Hopf adaptive filtering approach. This matching process generates a linear system with a spectrum similar to that of the unknown process. In general, only the spectra would be expected to match, but since the unknown process is also an all-zero system, the transfer function coefficients match as well: $h[n]_{\text{unknown}} = [0.5 \ 0.75 \ 1.2]$, $h[n]_{\text{match}} = [0.53 \ 0.79 \ 1.28]$.

Biosignal and Medical Image Processing

in impulse response as given by the Wiener filter coefficients. The coefficients for the unknown system are 0.5, 0.75, and 1.2 that are closely matched as 0.53, 0.79, and 1.28.

8.2 Adaptive Signal Processing

The area of adaptive signal processing is relatively new; yet, it already has a rich history. As with optimal filtering, only a brief example of the usefulness and broad applicability of adaptive filtering can be covered here. As stated above, the filters described in Chapter 4 were based on an *a priori* design criteria and were fixed throughout their application. Although the Wiener filter described above does not require prior knowledge of the input signal (only the desired outcome), it too is fixed during a given application. As with the classical spectral analysis methods, these filters cannot respond to changes that might occur during the course of the signal. Adaptive filters have the capability of modifying their properties based on selected features of the signal being analyzed.

A typical adaptive filter paradigm is shown in Figure 8.5. In this case, the filter coefficients are modified by a feedback process designed to make the filter's output, $y[n]$, as close to some desired response, $d[n]$, as possible, by reducing the error, $e[n]$, to a minimum. As with optimal filtering, the nature of the desired response depends on the specific problem involved and its formulation may be the most difficult part of the adaptive system specification (Stearns and David, 1996).

The inherent stability of FIR filters makes them attractive in adaptive applications as well as in optimal filtering (Ingle and Proakis, 2000). As with the Wiener filter, the adaptive filter, $H[z]$, can again be represented by a set of FIR filter coefficients, $b[k]$. The FIR filter equation (i.e., convolution) is repeated here, but the role of time variables k and n have been reversed and the filter coefficients are indicated as $b_n[k]$ to show that they vary with time (i.e., the discrete time variable n).

$$y[n] = \sum_{k=0}^{L-1} b_n[k]x[n - k] \quad (8.14)$$

While the Wiener–Hopf equations (Equations 8.12 and 8.13) have been modified for use in an adaptive environment, a simpler and more popular approach is based on gradient optimization. This approach is usually termed the LMS recursive algorithm. As in Wiener filter theory, this algorithm also determines the optimal filter coefficients; it is also based on minimizing the squared error, but it does not require computation of the correlation functions, r_{xx} and r_{xy} . Instead, the LMS algorithm uses a recursive gradient method known as the *steepest-descent* method for finding the filter coefficients that produce the minimum sum of squared error. A similar gradient method is used to train ANN as described in Chapter 17.

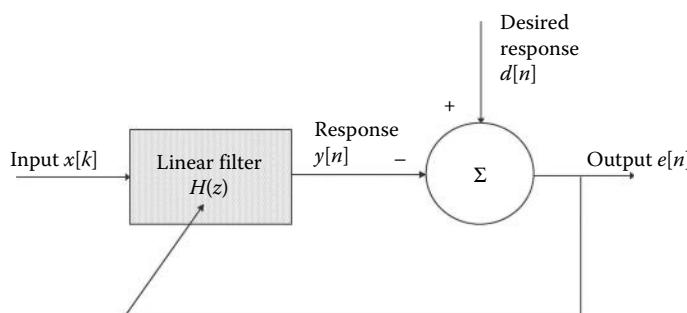


Figure 8.5 Basic configuration of an adaptive filter.

8.2 Adaptive Signal Processing

The adaptive filter operates by modifying the filter coefficients, $b_n[k]$, based on some signal property. The general adaptive filter problem has similarities to the Wiener filter theory problem discussed above in that an error is minimized, usually between the input and some “desired response.” As with optimal filtering, it is the squared error that is minimized and, again, it is necessary to somehow construct a desired signal. In the Wiener approach, the analysis is applied to the entire waveform and the resultant optimal filter coefficients are similarly applied to the entire waveform: the so-called *block approach*. In adaptive filtering, the filter coefficients are adjusted and applied in an ongoing basis as indicated in Figure 8.6. The difference between the filter and desired output (right column, Figure 8.6) is used to modify the filter coefficients so that the error is reduced. Of course, in most real situations, the input and desired signals would be more complicated; so, a careful adjustment strategy is needed.

The examination of Equation 8.3 in the Wiener filter derivation shows that the sum of squared errors is a quadratic function of the FIR filter coefficients, $b[k]$; hence, this function has a single minimum. The goal of the LMS algorithm is to adjust the coefficients so that the sum of squared error moves toward this minimum. Using the method of steepest descent, the LMS algorithm adjusts the filter coefficients based on an estimate of the negative gradient of the error function with respect to a given $b[k]$. This estimate is given by the partial derivative of the squared error, e_n^2 with respect to the coefficients, $b_n[k]$:

$$\nabla_n = \frac{\partial e_n^2}{\partial b_n[k]} = 2e[n] \frac{\partial(d[n] - y[n])}{\partial b[k]} \quad (8.15)$$

Since $d[n]$ is independent of the coefficients $b_n[k]$, its partial derivative with respect to $b_n[k]$ is zero. As $y[n]$ is a function of the input times $b_n[k]$ (Equation 8.14), its partial derivative with

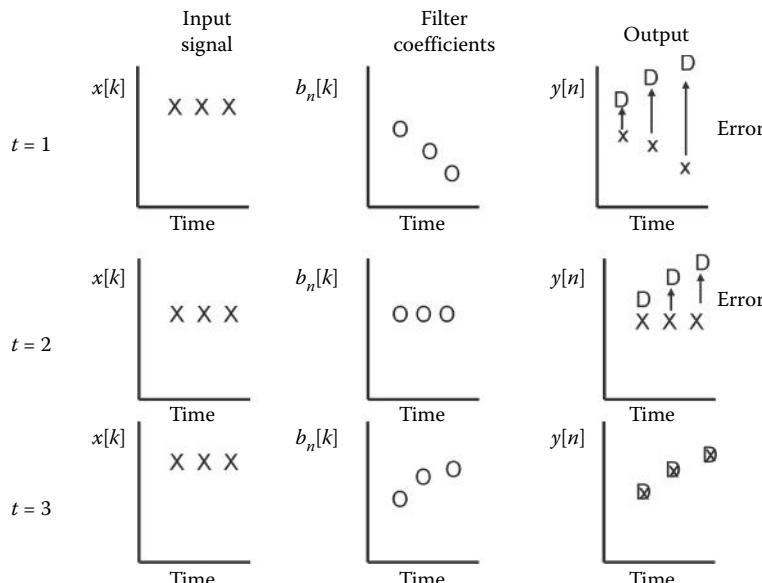


Figure 8.6 Schematic representation of the adaptive adjustment of filter coefficients. The left column shows three samples of the input signal at three subsequent time intervals. For simplicity, the input has constant values. The center column shows three filter coefficients as they are adaptively modified. The right column shows the filter output, “x” points, and the desired signal, “D” points; the latter is shown as a slowly rising signal. In the three time increments shown (top to bottom), the filter output converges toward the desired signal.

Biosignal and Medical Image Processing

respect to $b_n[k]$ is just $x[n-k]$ and Equation 8.15 can be rewritten in terms of the instantaneous product of error and the input:

$$\nabla_n = -2e[n] x[n - k] \quad (8.16)$$

Initially, the filter coefficients are arbitrarily set to some $b_0[k]$: frequently 0.0. With each new input sample, a new error signal, $e[n]$, can be computed (Figure 8.6). On the basis of this new error signal, the new gradient is determined (Equation 8.16) and the filter coefficients are updated:

$$b_n[k] = b_{n-1}[k] + \Delta e[n] x[n - k] \quad (8.17)$$

where Δ is a constant that controls the descent and, hence, the rate of convergence. This parameter must be chosen with some care. A large value of Δ will lead to large modifications of the filter coefficients that will hasten convergence, but they can also lead to instability and oscillations. Conversely, a small value will result in slow convergence of the filter coefficients to their optimal values. A common rule is to select the convergence parameter, Δ , such that it lies in the range

$$0 < \Delta < \frac{1}{10LP_x} \quad (8.18)$$

where L is the length of the FIR filter and P_x is the power in the input signal. P_x can be approximated by

$$P_x \approx \frac{1}{N} \sum_{n=1}^N x^2[n] \quad (8.19)$$

Note that for a waveform of zero mean, P_x equals the RMS value of x . The LMS algorithm given in Equation 8.17 can easily be implemented in MATLAB as shown in the next section.

Adaptive filtering has a number of applications in biosignal processing. It can be used to suppress a narrowband noise source such as 60 Hz that is corrupting a broadband signal. It can also be used in the reverse situation, removing broadband noise from a narrowband signal, a process known as *adaptive line enhancement* (ALE).^{*} It can also be used for some of the same applications as the Wiener filter such as system identification, inverse modeling, and, especially important in biosignal processing, adaptive noise cancellation (ANC). This latter application requires a suitable reference source that is correlated with the noise but not with the signal. Many of these applications are explored in the following sections and in the problems.

8.2.1 ALE and Adaptive Interference Suppression

The configuration for ALE and adaptive interference suppression is shown in Figure 8.7.[†] When this configuration is used in adaptive interference suppression, the input consists of a broadband signal, $x[n]$, and narrowband noise, noise[n], such as 60 Hz. Since the noise is narrowband compared to the relatively broadband signal, the noise portion of sequential samples will remain correlated whereas the broadband signal components will decorrelate after a few samples. If the combined signal and noise are delayed by D samples, the broadband (signal) component of the delayed waveform will no longer be correlated with the broadband component in the upper

^{*} The adaptive line enhancer is so termed because the objective of this filter is to enhance a narrowband signal, one with a spectrum composed of a single point, or in some frequency plots, a single “line.”

[†] Recall that the width of the autocorrelation function is a measure of the range of samples for which the samples are correlated; this width is *inversely* related to the signal bandwidth. Hence, broadband signals remain correlated only for a few samples and vice versa.

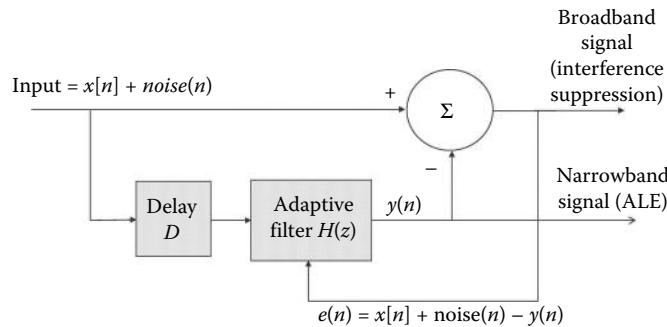


Figure 8.7 Configuration of adaptive filter components for ALE and adaptive interference suppression. In adaptive interference suppression, $x[n]$ is the signal and $\text{noise}[n]$ is the noise, but in ALE, the noise ($\text{noise}[n]$) is what you want. So what was the signal in adaptive interference suppression ($x[n]$) is now noise.

pathway, but if D is carefully selected, the narrowband components may still remain somewhat correlated with the original signal. When the filter's output is subtracted from the input waveform to produce a difference signal, $e[n]$, the adaptive filter will try to adjust its output to minimize this difference signal. But its ability to reduce the energy in the difference signal is limited: the only component that the filter has to play with that has any correlation with the input is the narrowband component, $\text{noise}[n]$. So, the only way this filter can reduce the difference signal is to reduce the narrowband component in the input signal. Hence, to reduce the difference signal, $e[n]$, it must reduce the narrowband component. The difference signal, now containing reduced narrowband noise, constitutes the output in adaptive interference suppression (Figure 8.7) (upper output, $e[n]$).

In ALE, the configuration is the same except the roles of the signal and noise are reversed: the narrowband component, $\text{noise}[n]$, is actually the signal and the broadband component, $x[n]$, is the noise (remember, noise is just a waveform that you do not want). The filter's output will contain as much of the narrowband component as possible since it is trying its best to subtract this component from the combined components in the upper pathway. So, the desired narrowband signal is the ALE filter's output (Figure 8.7) (lower output path, $y[n]$).

Again, it might be possible to construct a filter of equal or better performance using the methods described in Chapter 4; however, the exact frequency or frequencies of the signal would have to be known in advance. Moreover, spectral features would have to be fixed throughout the signal, a situation that is often violated in biological signals. The ALE can be regarded as a *self-tuning, narrowband filter* that tracks changes in signal frequency. An application of ALE is provided in Example 8.3 and an example of adaptive interference suppression is given in the problems.

8.2.2 Adaptive Noise Cancellation

ANC can be thought of as an outgrowth of interference suppression described above, except that a separate channel is used to supply the estimated noise or interference signal. One of the earliest applications of ANC was to eliminate 60-Hz noise from an ECG signal (Widrow et al. 1975). It has also been used to improve measurements of the fetal ECG by reducing interference from the mother's ECG and it has had considerable success in suppressing environmental noise in high-end headphones.

In ANC, a reference channel carries a signal that is correlated with the interference, but not with the signal of interest. The ANC consists of an adaptive filter that operates on this reference signal, $\text{noise}[n]$, to produce an estimate of the interference, $y[n]$ (Figure 8.8). This estimated noise is then subtracted from the signal channel to produce the output. As with ALE and

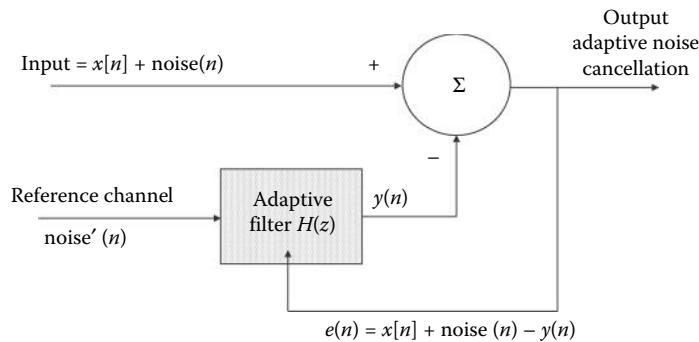


Figure 8.8 Configuration of the adaptive filter component in ANC. A reference channel is required that contains some representation of the noise contained in the signal channel.

interference cancellation, the difference signal is used to adjust the filter coefficients. Again, the strategy is to minimize the difference signal, which in this case is also the output since minimum output signal power corresponds to minimum noise in the output. This is because the only way the filter can reduce the output power is to reduce the noise component. Again, the noise is the only signal component that is available to the filter.

8.2.3 MATLAB Implementation

The implementation of the LMS recursive algorithm (Equation 8.11) in MATLAB is straightforward and is given below. Its application is illustrated through several examples below.

EXAMPLE 8.3

Optimal filtering using the LMS algorithm. Given the same sinusoidal signal in noise as used in Example 8.1, design an adaptive filter to remove the noise. Just as in Example 8.1, assume that you have a copy of the desired signal.

Solution

The program below sets up the problem as in Example 8.1, but uses the LMS algorithm in the routine `lms` instead of the Wiener-Hopf equation.

```
% Example 8.3 Adaptive filter. LMS algorithm applied to the
% data of Example 8.1
%
fs = 1000; % Sampling frequency
N = 1024; % Number of points
L = 256; % Optimal filter order
a = .25; % Convergence gain factor
%
.....Same initial lines as in Example 8-1.....
% Calculate Convergence Parameter
PX = mean(xn.^2); % Approx. power in xn
delta = a * (1/(10*L*PX)); % Calculate Δ
b = lms(xn,x,delta,L); % Apply LMS algorithm
.....Plotting identical to Example 8-1.....
```

Solution of LMS Algorithm

The LMS algorithm is implemented in the function `lms`. This routine outputs the filter coefficients, filtered data (the ALE output), and the difference signal (the interference suppression output).

The routine first initializes the filter coefficients and the filtered output to zero. Since the filter coefficients may not be symmetrical, it is necessary to reverse the input signal (or the filter coefficients) for convolution. Here, the input is reversed and a segment isolated by constructing a reverse-ordered vector that is the length of the number of filter coefficients, L . The value of n is incremented in a loop so that the filter is applied to the entire signal minus the length of the filter. The isolated and reversed signal, x_1 , is multiplied by the filter coefficients, b_n , to get the output of the filter, $y[n]$, in Figure 8.8. The error signal is calculated by subtracting the filter output from the desired signal. The filter coefficients, b_n , are updated using Equation 8.11: multiplying the current coefficients by the error times the convergence constant, δ . This process is repeated for the next data segment, shifted by one sample continuing through the length of the input waveform.

The filter coefficients, filter output, and the subtracted output are all provided as outputs from the routine.

```
function [b,y,e] = lms(x,d,delta,L)
%
% Inputs:    x = input
%             d = desired signal
%             delta = the convergence gain
%             L is the length (order) of the FIR filter.
%
% Outputs:   b = FIR filter coefficients
%             y = ALE output
%             e = residual error
%
M = length(x); % Get data length
b = zeros(1,L); y = zeros(1,M); % Initialize outputs
for n = L:M
    x1 = x(n:-1:n-L+1); % Isolate reversed signal segment
    y(n) = b * x1'; % Convolution
    e(n) = d(n) - y(n); % Calculate error signal
    b = b + delta*e(n)*x1; % Adjust filter coefficients. Eq. 8.17
end
```

While this function operates on the data as a block, it could easily be modified to operate online, that is, as the data are being acquired.

Results

Example 8.3 produces the data in Figure 8.9. As with the Wiener filter, the adaptive process adjusts the FIR filter coefficients to produce a narrowband filter centered about the sinusoidal frequency. The convergence factor, a , is empirically set to give rapid, yet stable, convergence. In fact, close inspection of Figure 8.9 shows a small, slow oscillation in the output amplitude, suggesting marginal stability.

EXAMPLE 8.4

The previous example shows the application of the LMS algorithm to a stationary signal where the desired signal was known as in Example 8.1. Example 8.4 explores the adaptive characteristics of the algorithm in the context of an ALE problem. In this example, a narrowband signal, a single

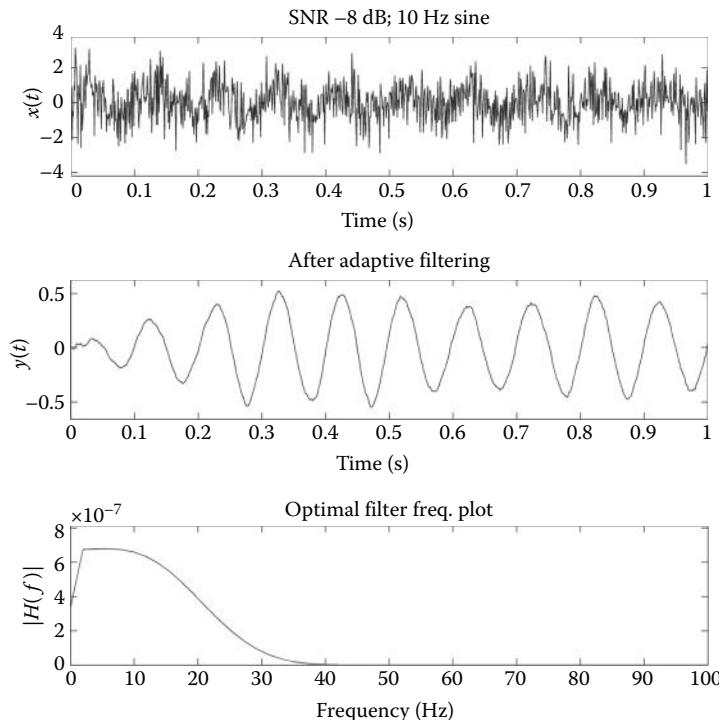


Figure 8.9 Application of an adaptive filter using the LMS recursive algorithm to data containing a single sinusoid (10 Hz) in noise (SNR = -8 dB), the same used in Example 8.1. Note that the filter requires the first 0.4–0.5 s (400–500 points) to adapt and that the frequency characteristics of the coefficients produced after adaptation are those of a bandpass filter with a single peak near 10 Hz. Comparing this figure with Figure 8.3 suggests that the adaptive approach is as effective as the Wiener filter for the same number of filter weights.

sinusoid in noise (SNR = -6 dB), changes abruptly in frequency. An ALE-type filter is used that must filter out the broadband noise, then readjust its coefficients to adapt to the new frequency.

The signal will consist of two sequential sinusoids of 10 and 20 Hz, each lasting 0.6 s. An FIR filter with 256 coefficients will be used. Delay and convergence gain will be set for the best results. (As in many problems, some adjustments must be made on a trial-and-error basis.)

SOLUTION

Use the LMS recursive algorithm to implement the ALE filter. Here, the input signal is the desired signal and the delayed signal is the input to the filter. We use a delay of five samples, a delay that is found by trial and error. The convergence gain of $0.075*1/(10LP_s)$ (Equation 8.12) is found empirically to converge rapidly without oscillation. The influence of this gain on convergence and stability is explored in the problems, as is the effect of the decorrelation delay.

```
% Example 8.4 Adaptive Line Enhancement.
%
fs = 1000; % Sampling frequency
L = 256; % Filter order
N = 2000; % Number of points
```

```

delay = 5; % De-correlation delay
a = .075; % Convergence gain
t = (1:N)/fs; % Time vector for plotting
%
% Generate data: two sequential sinusoids. SNR = 6dB.
x = [sig_noise(10,-6,N/2) sig_noise(20,-6,N/2)];
    .....plot initial data, axis, title..... .
PX = mean(x.^2); % Mean power
delta = (1/(10*L*PX)) * a; % Scale delta by a
xd = [x(delay:N) zeros(1,delay-1)]; % Decorrelation delay
[b,y] = lms(xd,x,delta,L); % Apply LMS
    .....plot filtered data, axis, title..... .

```

Results

The results of this code are shown in Figure 8.10. Since the filter coefficients are initially zero, there is no output for the first 0.3 s, after which the filtered signal develops adaptively. When the signal doubles in frequency at 1.0 s, the filter takes approximately 0.5 s to modify its coefficients for the new narrowband frequency. Of the many delays evaluated, a delay of five samples shows best results.

EXAMPLE 8.5

This is an application of the LMS algorithm to ANC. A sawtooth waveform is the signal and a sinusoid at 2.33 times the sawtooth frequency is the interference signal. The reference signal is a low-pass filtered version of the interference signal, which is phase shifted. This is a crude simulation of what we would expect in practical situations. More realistic examples are given in the problems.

```
% Example 8.5 Adaptive Noise Cancellation
```

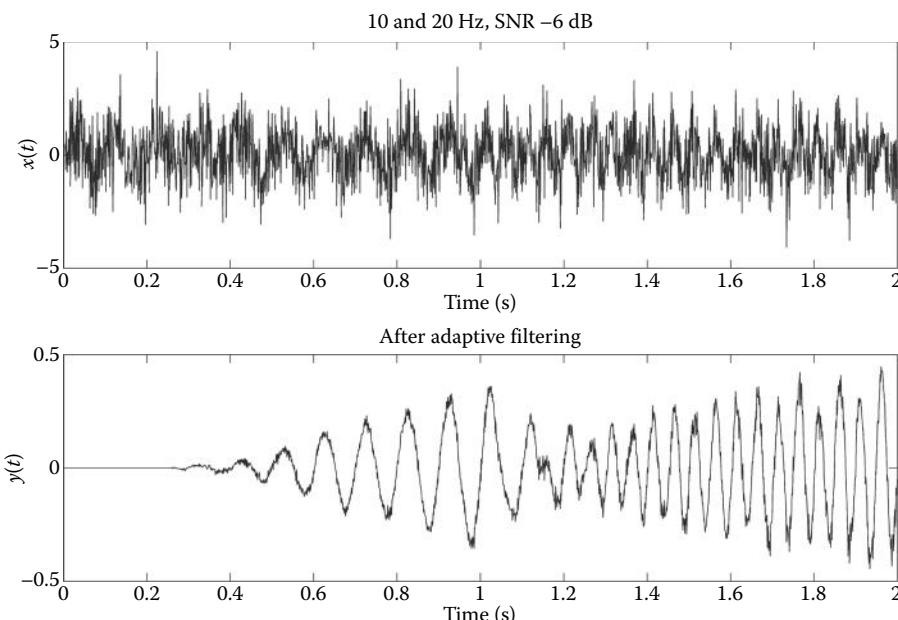


Figure 8.10 ALE applied to a signal consisting of two sequential sinusoids having different frequencies (10 and 20 Hz). The delay of five samples and the convergence gain of 0.075 are determined by trial and error to give the best results with the specified FIR filter length (256).

Biosignal and Medical Image Processing

```

%
fs = 500; % Sampling frequency
L = 256; % Filter order
N = 2000; % Number of points
t = (1:N)/fs; % Time vector for plotting
a = 0.5; % Max, convergence gain
wn = 150/fs; % Filter for creating
[b,a] = butter(4,wn); % reference signal
%
% Generate sawtooth waveform and plot
w = (1:N) * 4 * pi/fs; % Signal frequency vector
x = sawtooth(w,.5); % Sawtooth signal
subplot(3,1,1); plot(t,x,'k'); % Plot original signal, axis, title.....
%
% Add interference signal to sawtooth
intefer = sin(w*2.33); % Interfer freq. = 2.33 signal
x = x + intefer; % Signal plus interference
ref = filter(b,a,intefer); % Reference is filtered
%
% interference signal.
. .... Plot corrupted data, axis, title.....
%
% Solution starts here.
% Apply adaptive filter and plot
Px = mean(x.^2); % Waveform power
delta = (1/(10*L*Px)) * a; % Convergence factor
[b,y,out] = lms(ref,x,delta,L); % Apply LMS
. .... Plot filtered data, axis, title.....

```

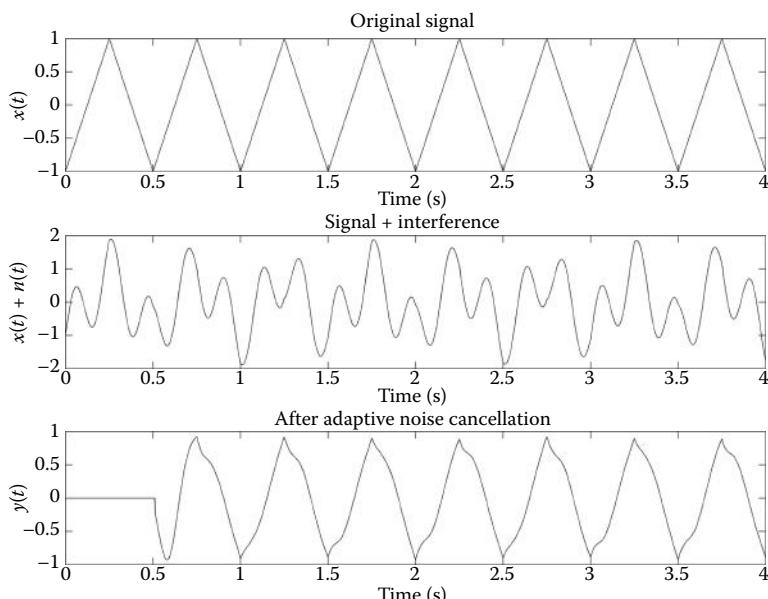


Figure 8.11 An example of ANC. In this example, the reference signal is simulated as a filtered copy of the interference. In practical situations, it may be difficult to find a reference signal that well represents the noise but does not contain signal components. The filter coefficients adapt after a 0.5-s delay and the ANC filter cancels the interference quite well.

Results

Most of the program is devoted to generating the signal and the interference signal. Figure 8.11 shows a good cancellation of the interference signal, but in this example, the reference signal is closely related to the interference signal. In a practical situation, it may be difficult to obtain a reference signal that accurately represents the noise but has little or no signal component. As seen, the adaptation requires approximately 0.5 s to adjust the filter coefficients.

8.3 Phase-Sensitive Detection

Phase-sensitive detection (PSD), also known as *synchronous detection*, is a technique for demodulating *amplitude-modulated* (AM) signals that is also very effective in reducing noise. From a frequency-domain point of view, the effect of amplitude modulation is to shift the signal frequencies to another portion of the spectrum, specifically, to a range on either side of the modulating or *carrier* frequency. Amplitude modulation can be very effective in reducing noise because it shifts signal frequencies to spectral regions where noise is not as prevalent. The application of a narrow-band filter centered about the new frequency range (i.e., the carrier frequency) can then be used to remove the noise including noise that may have been present in the original frequency range.*

PSD is commonly implemented using analog hardware. Prepackaged PSDs that incorporate a wide variety of optional features are commercially available and are sold under the term *lock-in amplifiers*. While lock-in amplifiers tend to be costly, less-sophisticated analog PSDs can be constructed quite inexpensively. The reason PSD is commonly carried out in the analog domain has to do with the limitations on digital storage and ADC. AM signals consist of a *carrier* signal (usually a sine wave) having an amplitude that is modified by the signal of interest. For this to work without loss of information, the frequency of the carrier signal must be much higher than the highest frequency in the signal of interest. (As with sampling, the greater the spectral spread between the highest signal frequency and the carrier frequency, the easier it is to separate the two through demodulation.) Since sampling theory dictates that the sampling frequency must be at least twice the highest frequency in the *input signal*, the sampling frequency of an AM signal must be more than twice the carrier frequency and the carrier frequency must be much higher than the signal frequency. Thus, the sampling frequency needs to be much higher than the highest frequency of interest, which is the modulating frequency and not the carrier frequency. This is also a great deal higher than would be the case if the AM signal was demodulated using an analog PSD before sampling. Hence, digitizing an AM signal before demodulation places a higher burden on memory storage requirements and ADC rates. However, with the reduction in cost of both memory and high-speed ADCs, it is becoming more and more practical to decode AM signals using the software equivalent of PSD. The following analysis applies to both hardware and software PSDs.

8.3.1 AM Modulation

In an AM signal, the amplitude of a sinusoidal carrier signal varies in proportion to changes in the signal of interest. AM signals commonly arise in bioinstrumentation systems when a transducer based on variation in electrical properties is excited by a sinusoidal voltage (i.e., the current through the transducer is sinusoidal). The strain gage is an example of such a transducer where resistance varies in proportion to small changes in length. Assume that two strain gages are differentially configured and connected to a bridge circuit as shown in Figure 1.6. One arm of the bridge circuit contains the transducers, $R + \Delta R$ and $R - \Delta R$, whereas the other arm contains two resistors having a fixed value of R , the nominal (unstretched or uncompressed) strain gage resistance. In this example, we assume that ΔR is a function of time, specifically, a sinusoidal function of time, $\Delta R = k \cos(\omega_s t)$. In other words, the strain applied to the gage varies

* Many biological signals contain frequencies around 60 Hz, a frequency containing serious noise sources.

Biosignal and Medical Image Processing

sinusoidally. In the general case, it would be some time-varying signal, $x(t)$, but then through Fourier analysis, it could be decomposed into a range of sinusoidal frequencies. If the bridge is balanced and $\Delta R \ll R$, then it is easy to show using basic circuit analysis that the bridge output is

$$V_{\text{out}} = \frac{\Delta RV}{2R} \quad (8.20)$$

where V is source voltage of the bridge. If this voltage is also sinusoidal, $V = V_s \cos(\omega_c t)$, where ω_c is the carrier frequency. Then $V_{\text{out}}(t)$ in Equation 8.20 becomes

$$V_{\text{out}} = \frac{\Delta RV_s \cos(\omega_c t)}{2R} \quad (8.21)$$

If the input strain (actual stress) to the strain gages varies sinusoidally, we can write for the strain gage resistance: $\Delta R = k \cos(\omega_s t)$, where ω_s is the signal frequency and is assumed to be $\ll \omega_c$, and k is the strain gage sensitivity. Again, assuming $\Delta R \ll R$ (as must be the case for strain gages, since to produce a large ΔR would require so much stretching that it would break), the equation for $V_{\text{out}}(t)$ becomes

$$V_{\text{out}} = \frac{V_s (\cos(\omega_c t) k \cos(\omega_s t))}{2R} \quad (8.22)$$

Now, applying the trigonometric identity for the product of two cosines:

$$\cos x \cos y = 1/2 \cos(x + y) + 1/2 \cos(x - y) \quad (8.23)$$

The equation for $V_{\text{out}}(t)$ becomes

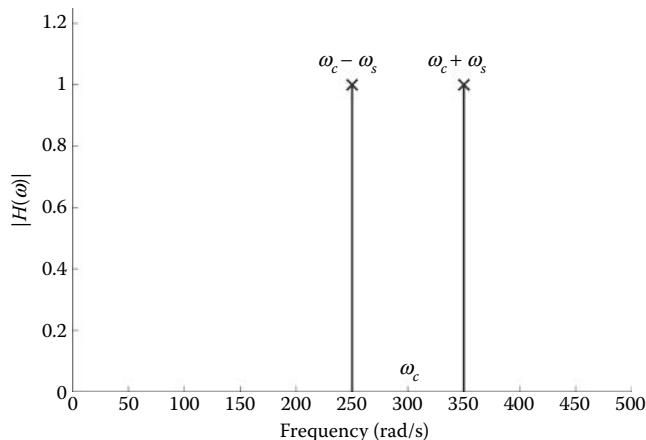


Figure 8.12 Frequency spectrum of the signal created by sinusoidally exciting a variable resistance transducer with a carrier frequency ω_c . The signal appears to be reflected about ω_c . In this figure, the carrier frequency, ω_c , is assumed to be 300 rad/s and the signal frequency, ω_s , is assumed to be 50 rad/s. This type of modulation is termed double sideband-suppressed carrier modulation since the carrier frequency is absent. (Frequency is expressed in radians for variety.)

$$V_{\text{out}}(t) = \frac{V_s k}{4R} [\cos(\omega_c t + \omega_s t) + \cos(\omega_c t - \omega_s t)] \quad (8.24)$$

This signal has the magnitude spectrum given in Figure 8.12. This signal is termed *double sideband-suppressed carrier modulation* since the carrier frequency, ω_c , is missing as seen in Figure 8.12.

Returning to Equation 8.22, $V_{\text{out}}(t)$ can be written as

$$V_{\text{out}}(t) = \frac{V_s k}{2R} (\cos(\omega_c t) \cos(\omega_s t)) = A(t) \cos(\omega_c t) \quad (8.25)$$

where

$$A(t) = \frac{V_s k}{2R} (\cos(\omega_s t)) \quad (8.26)$$

Thus, Equation 8.25 describes a carrier frequency, ω_c , modulated by a signal $A(t)$ that is simply the input signal (in this case, sinusoidal mechanical stress) multiplied by a constant.

8.3.2 Phase-Sensitive Detectors

The basic configuration of a PSD is shown in Figure 8.13.

The first step in PSD is to generate a phase-shifted carrier signal:

$$V_c(t) = \cos(\omega_c t + \theta) \quad (8.27)$$

Assuming the input to the PSD, $V_{\text{in}}(t)$, is the signal from the strain gage ($V_{\text{out}}(t)$ given in Equation 8.25) then the output of the multiplier, $V'(t)$, in Figure 8.13, becomes

$$V'(t) = V_{\text{in}}(t) \cos(\omega_c t + \theta) = A(t) \cos(\omega_c t) \cos(\omega_c t + \theta) \quad (8.28)$$

Then applying the trigonometric identity for the product of two cosines given in Equation 8.23, the output of the multiplier becomes

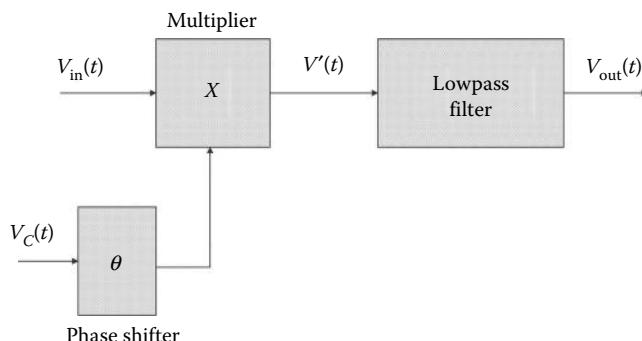


Figure 8.13 Basic configuration of a PSD used to demodulate an AM signal.

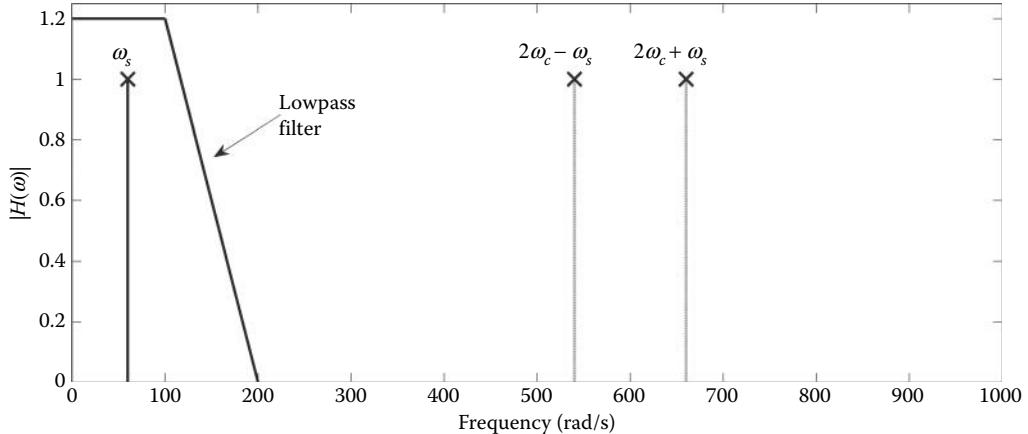


Figure 8.14 Frequency spectrum of the signal created by multiplying $V_{in}(t)$ by the carrier frequency. The signal frequency, ω_s , is reflected about $2\omega_c$. After lowpass filtering, only the original low-frequency signal, ω_s , remains. In this figure, the carrier frequency, ω_c , is assumed to be 300 rad/s and the signal frequency, ω_s , is assumed to be 50 rad/s. In most practical situations, the carrier frequency would be much higher: in the kHz or even in the MHz range.

$$V'(t) = \frac{A(t)}{2} [\cos(2\omega_c t + \theta) + \cos(\theta)] \quad (8.29)$$

To get the full spectrum *before filtering*, substitute Equation 8.26 for $A(t)$ into Equation 8.29:

$$V' = \frac{V_s k}{4R} (\cos(2\omega_c t + \theta) \cos(\omega_s t) + \cos(\omega_s t) \cos(\theta)) \quad (8.30)$$

Again, applying the identity in Equation 8.23 to the first product of cosines in Equation 8.30:

$$V' = \frac{V_s k}{4R} \left[(\cos(2\omega_c t + \theta + \omega_s t) + \cos(2\omega_c t + \theta - \omega_s t))/2 + \cos(\omega_s t) \cos(\theta) \right] \quad (8.31)$$

Note that $\cos(\theta)$ is just a constant since the phase shift, θ , is constant. The spectrum of $V'(t)$ contains three frequencies as shown in Figure 8.14.

After lowpass digital filtering (Figure 8.14), the higher frequency terms, $\omega_c \pm \omega_s$, will be reduced to near zero, so that only the last component in Equation 8.32 remains:

$$V_{out}(t) = \frac{V_s k}{4R} \cos(\omega_s t) \cos(\theta) = \frac{A(t)}{2} \cos(\theta) \quad (8.33)$$

Since $\cos \theta$ is a constant, the output of the PSD is the demodulated signal, $A(t)$, multiplied by $\cos(\theta)/2$. The term “phase-sensitive” is derived from the fact that the constant multiplier is a function of the phase difference between $V_c(t)$ and $V_{in}(t)$. Note that while θ is generally a constant value, any shift in phase between the two signals would induce a change in the output signal level; so, a PSD could also be used to detect phase changes between two signals of constant amplitude.

While the lowpass filter in Figure 8.13 acts on the demodulated signal as shown in Figure 8.14, it is instructive to imagine its influence reflected back on the modulated signal. With regard

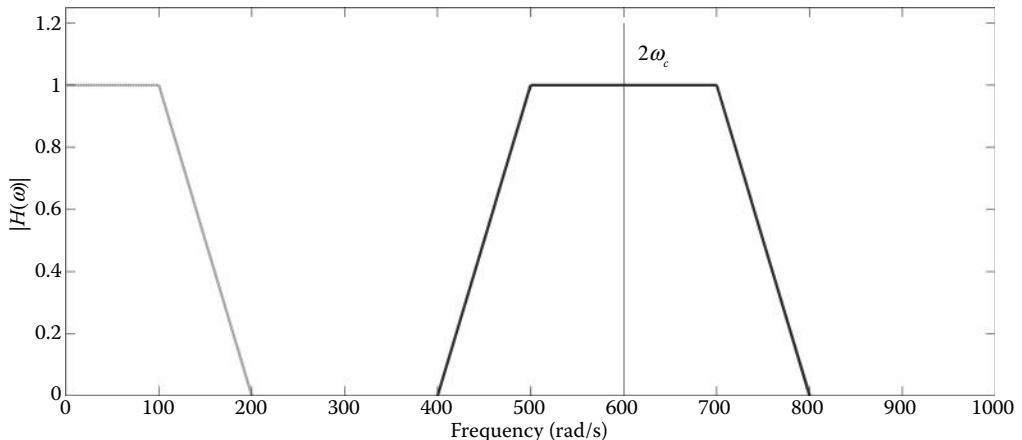


Figure 8.15 The effect of the multiplication process in PSD is to project the lowpass filter on either side of $2\omega_c$, at least with respect to the modulated signal. Although the bandwidth appears quite large in this figure, PSD can effectively produce a very narrowband bandpass filter that automatically tracks any changes in the carrier frequency. In this plot, the carrier frequency is taken to be 300 rad/s and the filter cutoff frequency is taken to be 100 rad/s. Normally, the carrier frequency would be much higher with respect to the filter's cutoff frequency.

to the modulated signal, the lowpass filter is effectively acting to filter the frequencies on either side of $2\omega_c$. Recall that in sampling where a continuous signal is multiplied by a pulse train at the sampling frequency, additional frequencies are generated on either side of the sampling frequency. In PSD, the multiplier effectively shifts the lowpass filter's spectrum to be symmetrical about twice the carrier frequency, giving it the form of a bandpass filter (Figure 8.15). Given that the carrier frequencies are generally quite high and the lowpass filter cutoff frequency can be made very low, it is possible to construct, in effect, an extremely narrowband bandpass filter this way. Consider, for example, a carrier frequency of 1 MHz and a lowpass filter with a cutoff frequency of 1 Hz. The bandpass frequency range would be 1 Hz on either side of a 1-MHz signal. Moreover, because the lowpass filter is reflected around $2\omega_c$, the center frequency of this effective bandpass filter *tracks* any changes in the carrier frequency. It is these two features, the capability for extremely narrowband filtering and tracking the carrier frequency, which give PSD its signal-processing power and the ability to recover signals buried in extensive noise.

The only unmodulated frequencies that are not attenuated by the effective bandpass filter are those around the carrier frequency that also fall within the bandwidth of the lowpass filter. In other words, only the noise that naturally occurs around $2\omega_c$ and that falls within plus or minus of the bandwidth of the lowpass filter will get through to the output. If the lowpass filter bandwidth is small, only a small amount of noise falls within the range of the effective bandpass filter. Of course, the lowpass cutoff frequency limits the bandwidth of the output signal ($V_{\text{out}}(t)$ in Figure 8.13); so, if a large bandwidth is required, the filtering will be less effective, but this is true for any filter in any application.

8.3.3 MATLAB Implementation

PSD is implemented in MATLAB using simple multiplication and filtering. The application of a PSD is given in Example 8.6 below. A carrier sinusoid of 250 Hz is modulated with a sawtooth wave having a frequency of 5 Hz. The AM signal is buried in noise that is 3.16 times the signal (i.e., SNR = -10 dB).

EXAMPLE 8.6

Phase-sensitive detector. Use a PSD to demodulate an AM signal and reduce the noise. The signal consists of a 250-Hz carrier modulated with a 5-Hz sawtooth wave. The AM signal should be buried in broadband noise that is 3.16 times the signal (i.e., SNR = -10 dB).

Solution

After constructing the noisy modulated signal, multiply this signal by the phase-shifted carrier sinusoid. (As is often the case, constructing the signal requires the majority of the code.) Shift the carrier by the number of samples representing 45°. (A shift of 45° is commonly used in PSD.) Make the shift symmetrical so that the end samples are shifted to the beginning. After multiplication (point by point), filter the result with a lowpass filter. Use a second-order Butterworth low-pass filter with a cutoff frequency of 80 Hz. These filter parameters are based on trial and error to produce a smooth demodulated signal that still retained some sharpness in the sawtooth peaks. Since the carrier frequency is 250 Hz, make $f_s = 2 \text{ kHz}$ and $N = 2000$; so, the signal is 1-s long.

```
% Example 8.6 Phase Sensitive Detection
%
fs = 2000; % Sampling frequency
fsig = 5; % Signal frequency
fc = 250; % Carrier frequency
N = 2000; % Use 1 sec of data
t = (1:N)/fs; % Time vector
wn = 80/(2*fs); % PSD lowpass filter
order = 2; % Filter order
[b,a] = butter(order,wn); % Design lowpass filter
%
```

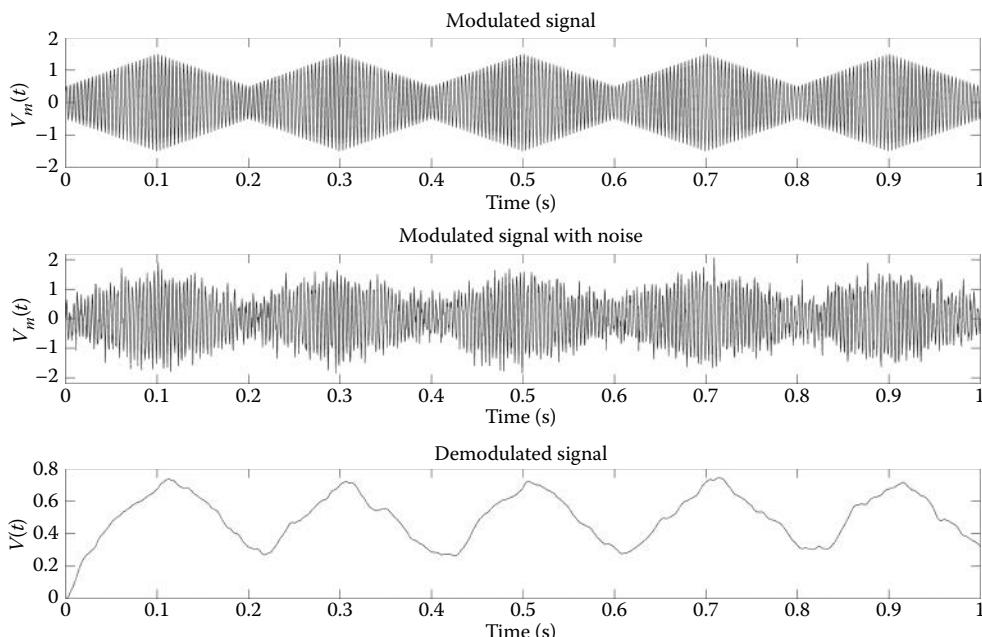


Figure 8.16 Results of applying PSD to a sawtooth waveform corrupted by noise in Example 8.6. The upper plot is the original modulated waveform, the middle plot is the signal after being corrupted by noise, and the lower plot is the demodulated signal recovered by PSD.

```
% Generate AM signal
w = (1:N) * 2*pi*fc/fs; % Carrier frequency = 250 Hz
w1 = (1:N)*2*pi*fsig/fs; % Signal frequency = 5 Hz
vc = sin(w); % Define carrier
vsig = sawtooth(w1,.5); % Define signal
%
% Create modulated signal with a 50% modulation
vm = (1 + .5 * vsig) .* vc;
..... Plot AM Signal, axis, label,title.....
%
% Add noise for SNR = - 10 dB
noise = randn(1,N);
scale = (var(vsig)/var(noise)) * 3.16;
vm = vm + noise * scale; % Add noise to mod. sig.
..... Plot AM Signal, axis, label,title.....
%
% Phase sensitive detection (problem starts here)
ishift = fix((45/360) * fs/fc); % Shift carrier 45 deg
vc = [vc(ishift:N) vc(1:ishift-1)]; % Periodic shift
v1 = vc .* vm; % Multiplier
vout = filter(b,a,v1); % Lowpass filter
.... Plot demodulated signal, axis, label,title.....
```

Results

The results are shown in Figure 8.16 and show reasonable recovery of the demodulated signal from the noise. A major limitation in this example is the characteristics of the lowpass filter: digital filters do not perform well at low frequencies. Better performance would be obtained if the interference signal is narrowband such as 60-Hz interference. An example of using PSD in the presence of a strong 60-Hz signal is given in the problems.

8.4 Summary

If properties of the desired signal are known, the Wiener filter theory can be used to construct an optimal FIR filter. The FIR filter coefficients are adjusted to minimize the mean square error between the output of the filter and the desired signal. These coefficients can be found in a straightforward manner from an equation based on the autocorrelation matrix of the signal and the cross-correlation between the signal and the known, desired signal. If the desired signal is not known, strategies based on delayed versions of the signal are available and presented in Section 8.2. Wiener filters can also be used to identify an unknown system represented as an equivalent FIR filter, which leads to an MA model of the system.

Adaptive filters are a variant of the Wiener filter in which the FIR filter coefficients are continually updated so that they can adjust to changes in the signal. Again, a representation of the desired signal is required. Coefficient adjustments attempt to make the filtered signal the same as the desired signal using data segments the same length as the filter coefficients. Segment position is incremented by one sample and the comparison/adjustment is repeated. This process continues to the end of the data. Changes in filter coefficients at each increment are limited by a convergence factor to ensure that the filters remain stable and converge to an acceptable set of coefficients. Higher convergence factors lead to more rapid convergence, but may also produce instability.

In adaptive interference suppression and ALE, the filter receives a delayed version of the original signal while the desired signal comes from the signal itself. The delayed signal will contain only narrowband information from the original signal since the delay decorrelates broadband information. (Recall broadband signals decorrelate more rapidly than narrowband signals.) The

Biosignal and Medical Image Processing

filter will then try to minimize the difference between the original signal and its narrowband component by subtracting the narrowband component from the original signal, leaving only the broadband component. If the broadband component is desired, the filter is operating to “suppress interference”; the implied assumption is that such interference is narrowband. In ALE, the narrowband component of the original signal is desired and is found enhanced in the output of the filter. Since the filter is adaptive, it will track changes in the narrowband component.

In ANC, the filter’s input is a reference signal that contains some representation of the noise in the signal. The filter again tries to minimize the difference between the reference and the original signal. Since it only has information on the noise component, it can only reduce the difference by minimizing the noise component. The filter will adaptively modify to produce a noise component that best matches that of the signal as this leads to the minimum difference. ANC has been quite successful in canceling the mother’s ECG signal for a recording of fetal ECG and in reducing environmental noise in high-end headphones.

PSD is a method of demodulating an AM signal that can significantly reduce noise. The components of a PSD consist of a phase shifter, multiplier, and lowpass filter. The signal is multiplied by a phase-shifted version of the original signal, then it is lowpass filtered. Accordingly, the carrier signal must be available, but this is usually the case in measurement applications. With regard to the modulated signal, this process acts as if the lowpass filter was a bandpass filter centered around twice the carrier frequency. This allows the effective construction of very narrowband bandpass filters that can lead to substantial reductions in noise. Lock-in amplifiers are PSDs packaged with adjustable phase shifters, lowpass filters, and related electronics. These devices are frequently used in bio-optics to demodulate and filter modulated light signals.

PROBLEMS

- 8.1 Apply the Wiener–Hopf approach to a signal plus noise waveform similar to that used in Example 8.1, except use two sinusoids at 10 and 30 Hz in 8-dB noise. Recall that the function `sig_noise` also provides the noiseless signal as the third output that can be used as the desired signal. Apply this optimal filter for filter lengths of 256 and 512.
- 8.2 Apply the Wiener–Hopf approach to a signal plus noise waveform similar to that used in Problem 8.1, but construct the signal so that the two sinusoids are sequential, specifically, 1 s of a 10-Hz sinusoid and another second of the 30-Hz sinusoid. Again, make the SNR –8 dB and $f_s = 1 \text{ kHz}$. Use `sig_noise` to generate the two sinusoids separately, then concatenate both the noisy signal and the desired signal. Construct a new time vector for the concatenated signal. Apply the Wiener filter using filter lengths of 256 and 512.
- 8.3 Apply the Wiener–Hopf optimal filter approach to the signal `x` in file `bandlimit.mat` ($f_s = 4 \text{ kHz}$, $N = 800$). This file also contains the desired signal, `xd`. The signal is fairly narrowband and the noise is band limited. Apply the Wiener filter using a filter length of 64. Plot the signal before and after filtering. Also plot the magnitude spectrum of the filter. Compare this with the magnitude spectra of the signal and noise given as `spect_sig` and `spect_noise` also found in the file `bandlimit.mat`. Note how well the Wiener filter acts to reduce the noise by maximally filtering the noise and minimally filtering the signal. The filter is particularly effective because, as shown by your spectrum plots, there is little overlap between the signal and noise spectra.

- 8.4 Generate a 100- and 350-Hz sinusoid in noise: SNR = -3 dB, $N = 5000$, and $f_s = 1000$ Hz. Use this as the desired signal while inputting a white-noise signal to an eighth-order Wiener filter. Plot the Fourier transform of filter coefficients. This is equivalent to using an MA model to determine the frequency spectrum. Note that it is difficult for an MA process to produce spectra with sharp peaks unless a high order is used, in which case, the process will generate spurious peaks. Run the problem several times and note the variation in spectra due to the nature of the random input.
- 8.5 Expand Example 8.2 so that the unknown system is a fifth-order, all-zero process having a z -transform transfer function:

$$H[z] = 0.264 + 1.405z^{-1} + 3.331z^{-2} + 3.331z^{-3} + 1.405z^{-4} + 0.264z^{-5}$$

As with Example 8.2, plot the magnitude spectrum of the “unknown” system, $H(z)$ along with the spectrum of the FIR “matching” system. Find the spectrum of the FIR process by taking the squared magnitude of the Fourier transform of $H[z]$ as in Example 8.2. Repeat the problem for filter lengths of 3, 6, and 10. Note that the filter with the largest number of coefficients produces spurious peaks.

- 8.6 Use Wiener filter theory to determine the FIR equivalent to the linear process described by the digital transfer function:

$$H(z) = \frac{0.2 + 0.5z - 1z^{-2}}{1 - 0.2z^{-1} + 0.8z^{-2}}$$

As with Example 8.2, plot the magnitude spectrum of the “unknown” system, $H(z)$, above and of the FIR “matching” system based on the FIR filter coefficients. Find the magnitude spectrum of the IIR process by taking the square of the magnitude of `fft(b,n)./fft(a,n)` (or use `freqz`). Use the MATLAB function `filtfilt` to produce the output of the IIR process to random noise. This routine produces no time delay between the input and filtered output. Determine the approximate minimum number of filter coefficients required to accurately represent the function above, by setting the coefficients to different lengths. (Note: To match an IIR process, using an FIR process will require considerably more weights than used in Example 8.2 or Problem 8.5.)

- 8.7 Repeat Problem 8.3 using the LMS algorithm. Use the same filter length and plot the signal before and after filtering and the magnitude spectrum of the filter. Again, compare the filter’s magnitude spectrum with that of the signal and noise given as `spect_sig` and `spect_noise` also found in the file `bandlimit.mat`. Note that the LMS algorithm acts as the Wiener filter to reduce the noise.
- 8.8 Use the LMS algorithm to find the system given in Problem 8.6. Produce the IIR process output to noise in the same manner using `filtfilt`. Plot the magnitude spectra of both the process, $H(z)$, and the identified system. Again, more filter coefficients will be required to model an IIR process with an FIR filter. Note that the LMS algorithm is as effective as the Wiener filter for this task.
- 8.9 Generate a 20-Hz “interference signal” in noise with an SNR = +8 dB, that is, the interference signal is 8 dB stronger than the noise. (Use ‘`sig_noise`’ with an SNR of +8.) Make $f_s = 1000$ Hz and $N = 2000$. In this problem, the noise is considered as the desired signal. Design an adaptive interference filter using the LMS

Biosignal and Medical Image Processing

algorithm to remove the 20-Hz “noise.” Use an FIR filter with 128 coefficients. Note that the filtered signal should look like pure noise without the 20-Hz sinusoid.

- 8.10 Apply the ALE filter described in Example 8.4 to a signal consisting of two sinusoids of 10 and 30 Hz ($\text{SNR} = -8 \text{ dB}$) that are present simultaneously, rather than sequentially as in Example 8.4. Use FIR filter lengths of 128 and 256 points. Evaluate the influence of modifying the delay between 4 and 18 samples. Plot the filtered output and the spectrum of the filter after adaptation. Note the improvement with filter length.
- 8.11 File `chirp1.mat` contains in variable `x` a chirp signal ranging from 5 to 20 Hz ($f_s = 1000 \text{ kHz}$ and $N = 2000$). The variable `x1` is the noise-free chirp signal. Apply ALE to the noisy signal. Plot the original signal and the ALE output on separate plots and plot the noise-free signal superimposed over the ALE output. Also, calculate the RMS difference between the ALE output and the noise-free signal in `x1`.
Use FIR filter lengths of 64, 128, and 254 points and determine which filter length produces the least RMS error. (Note that since noise is involved, it is best to run the program several times to get a range of errors.) Using the filter length that produces the least RMS error, evaluate the influence of modifying the delay between 2, 4, and 8 samples. Note that the delay has little effect on the RMS error, probably due to the fact that the noise is Gaussian and decorrelates only after one sample.
- 8.12 Repeat Problem 8.3 using ALE. Use a longer filter length of 256 and plot the signal before and after filtering. Note that the ALE algorithm does not produce as good a result as either the LMS algorithm or the Wiener filter because the signal is transient and is not long enough for the filter to adapt. For such signals, the LMS algorithm and Wiener filter are more suitable.
- 8.13 Modify the code in Example 8.5 so that the reference signal is correlated with, but is not the same as, the interference data. This should be done by convolving the reference signal with a lowpass filter consisting of three equal weights, that is,
`b = [.333 .333 .333];`
- 8.14 Load the file `music.mat` that contains in variable `y` a short segment of Handel’s Messiah with background noise added. This file also contains a representation of the noise in variable `noise`. Play the sound in `y` using `sound(y)`, then apply ANC to `y` using `noise` as the reference. Play the result using `sound`, but increase the amplitude by a factor of 10 to account for the attenuation caused by ANC. Since the sample rate is quite high ($f_s = 8129 \text{ Hz}$), use a large number of filter coefficients ($L \approx 2000$). Also, make the convergence gain fairly small ($a < 0.2$) to improve stability.
- 8.15 Redo the PSD in Example 8.6, but replace the white noise with a 60-Hz interference signal. The 60-Hz interference signal should have an amplitude that is 10 times that of the AM signal.

9

Multivariate Analyses

Principal Component Analysis and Independent Component Analysis

9.1 Introduction: Linear Transformations

Principal component analysis (PCA) and *independent component analysis* (ICA) fall within a branch of statistics known as *multivariate analysis*. As the name implies, multivariate analysis is concerned with the analysis of multiple variables (or measurements), but treats them as a single entity, such as variables that are acquired from multiple measurements made on the same process or system. In multivariate analysis, these multiple variables are often represented as a single variable that includes the different variables:

$$x = [x_m(1), x_m(2), \dots, x_m(N)]^T \quad \text{for } 1 \leq m \leq M \quad (9.1)^*$$

In this case, x is composed of M different but related variables, each containing N ($n = 1, \dots, N$) observations. As each row in x is actually a vector variable, x is an $M \times N$ matrix. In signal processing, the observations are time samples, whereas in image processing, they are pixels. Multivariate data can also be considered to reside in M -dimensional space, where each spatial dimension contains one signal (or image).

In general, multivariate analysis seeks to produce results that take into account the relationship between multiple variables and within the variables; so, it uses tools that operate on all the data. For example, the covariance matrix described in Chapter 2 (Equation 2.48) is an example of a multivariate analysis technique, since it includes both information about the individual variables, the *variance*, and information about the relationship between variables, the *covariance*. Since the covariance matrix contains information on both the variance within the variables and the covariance between the variables, it is occasionally referred to as the *variance-covariance* matrix.

A major concern of multivariate analysis is to find transformations of the multivariate data that make the data set smaller or easier to understand. Some typical multivariate questions are: Is it possible that the relevant information contained in a multidimensional variable could be

* The T stands for transpose and represents the matrix operation of switching rows and columns. Normally, all vectors including these multivariate variables are taken as column vectors, but to save space in this chapter, they are often written as row vectors with the transpose symbol to indicate that they are actually column vectors.

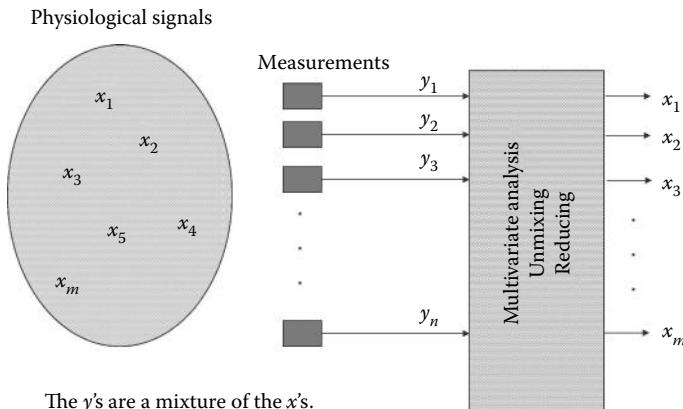


Figure 9.1 In EEG analysis, the electrical activity of a limited number of neural sources, x_m , is monitored by a large number of electrodes, y_n , placed on the scalp. If $m \ll n$ as is usual, the y_n signals contain a lot of redundancy. Some type of multivariate analysis could be used to find the minimum combination of the signals needed to represent the neural sources. Multivariate analysis might also be used to unscramble the mixtures contained in y_n to recover the neural source signals, x_m .

expressed using fewer dimensions (i.e., fewer individual variables); and is there a reorganization of the data set that would produce more meaningful variables? If the latter was true, psychologists would say that the more meaningful variables were hidden or *latent* in the original data. These latent variables represent the underlying processes better than the original data set. A common biomedical example is found in EEG analysis, in which a large number of signals are acquired above the region of the cortex; yet, it is well known that these multiple signals are the result of a smaller number of neural centers (Figure 9.1). It is the behavior of the neural centers, not the EEG signals per se, which are of interest. Some type of multivariate analysis might be able to determine the number of signals required to adequately represent the underlying sources. Since the individual EEG signals are likely to include mixtures of the neural source signals, perhaps, another type of multivariate analysis could unscramble these mixed signals and find the signals from those underlying sources (Figure 9.1).

Linear transformations are frequently applied to multivariate data to produce new data sets that are more meaningful or can be condensed into fewer variables. To reduce the dimensionality of a multivariate data set, the idea is to transform the original data into a new set where some of the new variables have values that are quite small compared to the others. Since these variables have small values, they do not contribute much information to the overall data set and, perhaps, they can be eliminated.* With the appropriate transformation, it is sometimes possible to eliminate a large number of variables that contribute only marginally to the total information.

The data transformation used to produce the new set of variables is often a linear function since linear transformations are easier to compute and their results are easier to interpret than nonlinear transformations. A linear transformation can be represented mathematically as

$$y_i(t) = \sum_{j=1}^M w_{ij} x_j(t), \quad i = 1, \dots, N \quad (9.2)$$

* This assumes that all the original variables have the same scaling. If not, they can be normalized to have the same variance before the transformation.

where w_{ij} are constant coefficients that define the transformation. Since this transformation is a series of equations, it can be equivalently expressed using the notation of linear algebra

$$\begin{bmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_M(t) \end{bmatrix} = W \begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_M(t) \end{bmatrix} \quad (9.3)$$

where W is a matrix of coefficients that define the transformation. As a linear transformation, this operation can be interpreted as a *rotation*, and possibly scaling, of the original data set in M -dimensional space. An example of how a rotation of a data set can produce a new data set with fewer major variables is shown in Figure 9.2 for a 2-D (i.e., two-variable) data set. The original data set is shown as a plot of one variable against the other, the so-called *scatter plot*, in Figure 9.2a. The variance of variable x_1 is 0.34 and the variance of x_2 is 0.20. While the variances are about the same, the two variables are also highly correlated: knowing the value of one of the variables gives you information on the other variable, specifically, a range of possible values.

After the rotation shown in Figure 9.2, the two new rotated variables are now no longer correlated. These new variables, y_1 and y_2 , have variances of 0.53 and 0.005, respectively. This suggests that one variable, y_1 , contains most of the information that was in the original two-variable set. The goal of this approach to data reduction is to find a matrix W that will produce such a transformation, one that will remove correlations in the data or *decorrelate* the data.

The two multivariate techniques discussed below, PCA and ICA, differ in their *goals* and in the criteria applied to the transformation. In PCA, the objective is to transform the data set so as to produce a new set of variables, termed *principal components*, which are *uncorrelated*. The motivation for this decorrelation is to show if, and how much, the dimensionality of the original data can be reduced. It will not necessarily produce variables that are more meaningful. This can be achieved simply by rotating the data in M -dimensional space (where M is the number of

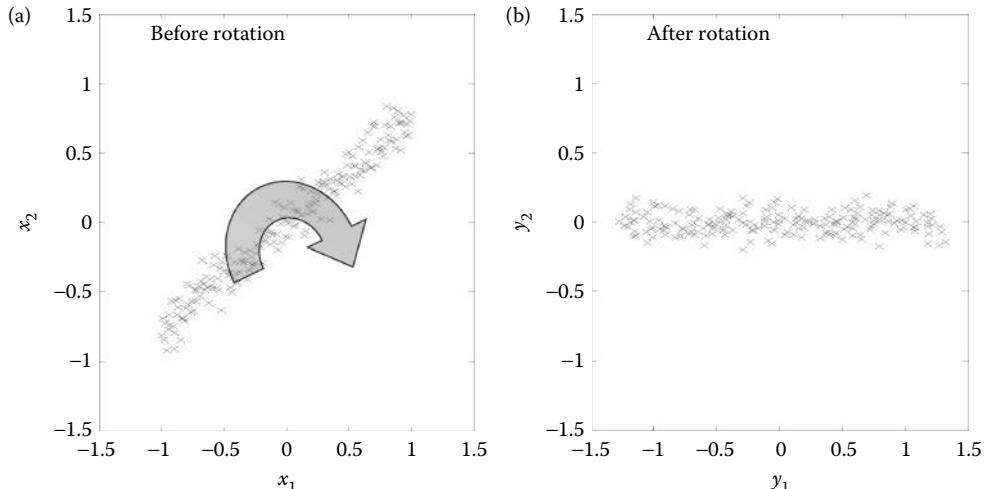


Figure 9.2 (a) Plotting the value of two variables against one another produces what is termed a *scatter plot* and provides an overview of the variables as well as the relationship between the two variables. In this case, the two variables are seen to have about the same variance, but are also highly correlated: knowing the value of one variable provides a restricted range of values for the other variable. (b) After a rotation that is a linear transformation, the correlation between the variables is eliminated. Moreover, most of the information resides in only one variable, y_1 .

different variables) as shown in Figure 9.2 for 2-D data. In ICA, the goal is a bit more ambitious: to find new variables, the *independent components*, which are statistically independent and are therefore likely to be more meaningful.

9.2 Principal Component Analysis

PCA is often referred to as a technique for reducing the number of variables in a data set without loss of information. In a few cases, it may also identify new variables with greater meaning. While PCA can be used to transform one set of variables into another smaller set, the newly created variables are generally not so easy to interpret. PCA has been most successful in applications such as image compression where data reduction, not interpretation, is of primary importance. In many applications, PCA is used only to provide information on the effective dimensionality of a data set. If a data set includes M variables, do we really need all M variables to represent the information or could the variables be recombined into a fewer number of variables that still contain most of the essential information? If so, what is the most appropriate dimension of the reduced data set?

PCA operates by transforming a set of correlated variables into a new set of uncorrelated variables called the principal components. If the variables in a data set are already uncorrelated, PCA is of no value. Since the principal components are uncorrelated, they are also orthogonal. Finally, the principle components are ordered in terms of the variability they represent. That is, the first principle component represents, for a single dimension (i.e., a single variable), the greatest amount of variability in the original data set. Each succeeding orthogonal component accounts for as much of the remaining variability as possible.

The operation performed by PCA can be described in a number of ways, but a geometrical interpretation is the most straightforward. While PCA is applicable to data sets containing any number of variables, it is easier to describe using only two variables since this leads to readily visualized graphs. Figure 9.3a shows two waveforms: a two-variable data set where each variable

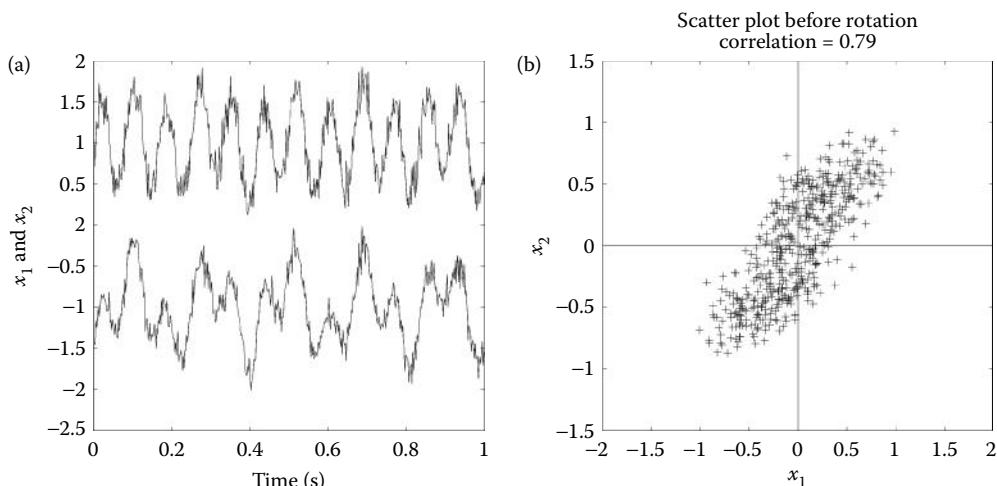


Figure 9.3 (a) Two waveforms made by mixing two sinusoids having different frequencies and amplitudes, then adding noise to the two mixtures. The resultant waveforms are related since they both contain information from the same two sources. (b) The scatter plot of the two variables (or waveforms) obtained by plotting one variable against the other for each point in time (i.e., each data sample). The correlation between the two samples can be seen in the diagonal arrangement of points.

9.2 Principal Component Analysis

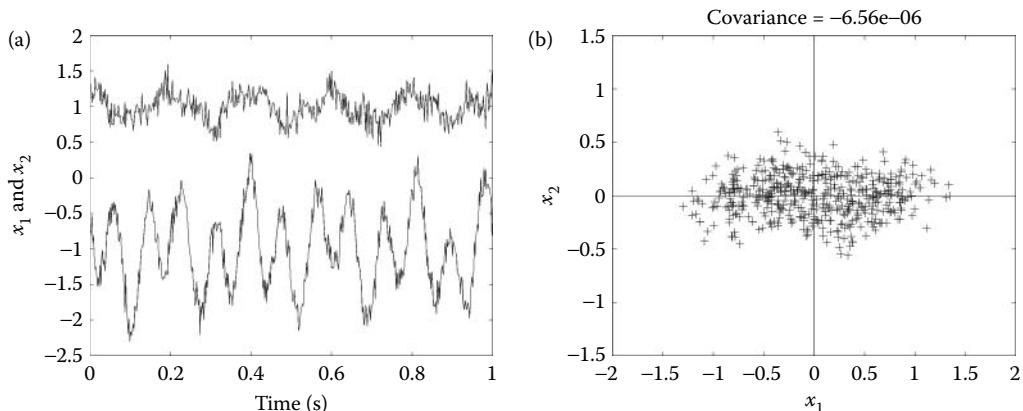


Figure 9.4 (a) Principal components of the two variables shown in Figure 9.3. These were produced by a 2-D rotation of the two variables. (b) The scatter plot of the rotated principal components. The symmetrical shape of the data suggests that the two new components are uncorrelated.

is a different mixture of the same two sinusoids added with different scaling factors. A small amount of noise was also added to each waveform (see Example 9.1). Since the data set was created using two separate sinusoidal sources, it should require two spatial dimensions. However, since each variable is composed of mixtures of the two sources, the variables have a considerable amount of covariance or correlation.* Figure 9.3b is a scatter plot of the two variables, a plot of x_1 against x_2 for each point in time, and shows the correlation between the variables as a diagonal spread of the data points. (The correlation between the two variables is 0.79.) Thus, knowledge of the value of x_1 gives information on the range of possible values of x_2 and vice versa. Note that the x_1 value does not uniquely determine the x_2 value, as the correlation between the two variables is less than 1. If the data were uncorrelated, the x_1 value would provide no information on possible x_2 values and vice versa. A scatter plot produced for such uncorrelated data would be roughly symmetrical with respect to both the horizontal and vertical axes.

For PCA to decorrelate the two variables, it simply needs to rotate the two-variable data set until the data points are distributed symmetrically about the means. Figure 9.4b shows the results of such a rotation, whereas Figure 9.4a plots the time response of the transformed (i.e., rotated) variables. In the decorrelated condition, the variance is maximally distributed along the two orthogonal axes. It is important to note that although the two variables are now decorrelated, the time responses in Figure 9.4a are still mixtures of two signals. Decorrelation does not usually separate mixtures. In the decorrelation operation, it may also be necessary to *center* the data by removing the means before rotation. The original variables plotted in Figure 9.3 had zero means; so, this step was not necessary.

While it is common in everyday language to take the word “uncorrelated” meaning “unrelated” (and hence “independent”), this is not the case in statistical analysis, particularly if the variables are nonlinear. If two (or more) variables are statistically independent, they will also be uncorrelated, but the reverse is not generally true. This is suggested by the plots in Figure 9.4, in which the two signals are decorrelated but still contain mixtures of the same signals so that they cannot be independent. An even more dramatic example is shown in Figure 9.5 for two variables plotted as time and scatter plots in Figure 9.5. Figure 9.5b shows the two signals are uncorrelated, but they are highly related and not independent. In fact, they are both generated using a single equation, the equation for a circle with some noise added. Many other nonlinear relationships

* Recall that covariance and correlation differ only in scaling. The definitions of these terms are given in Chapter 2: the correlation matrix is defined in Equation 2.47 and covariance matrix is defined in Equation 2.48.

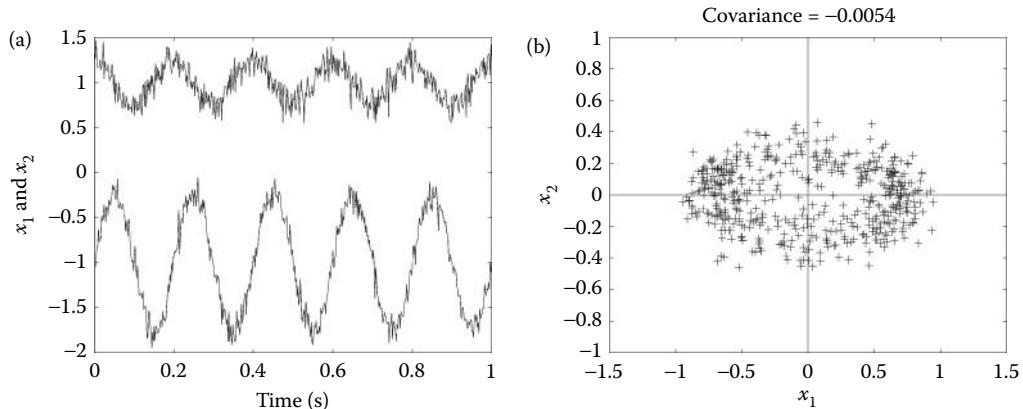


Figure 9.5 Time (a) and scatter plot (b) of two variables that are uncorrelated, but not independent. In fact, the two variables are highly dependent as they were generated by a single equation, that of a circle, with noise added.

(such as the quadratic function) can generate uncorrelated, yet related (i.e., nonindependent) variables. There is one case where decorrelation also means independence: if the variables have a Gaussian distribution (as in the case of most noise). When Gaussianly distributed variables are decorrelated, they are also independent. However, *signals* do not have Gaussian distributions and, therefore, are not likely to be independent after they have been decorrelated. This is one of the reasons why the principal components, the new variables generated by PCA, are not usually meaningful variables: they are still mixtures of the underlying sources as in Figure 9.4. This inability to make two signals independent through decorrelation provides the motivation for the methodology known as *independent component analysis* described later in this chapter.

If only two variables are involved, the rotation performed between Figures 9.3 and 9.4 can be done by trial and error: simply rotate the data until the covariance (or correlation) goes to zero. An example of this approach is given as an exercise in one of the problems. A better way to achieve zero correlation is to use a technique from linear algebra that generates a rotation matrix that reduces the covariance to zero.

9.2.1 Determination of Principal Components Using Singular-Value Decomposition

There are a number of techniques that can be used to find the principal components of a data set, but the most commonly used is based on *singular-value decomposition* (SVD). SVD is a linear algebra operation that reduces a matrix to the product of three matrices: a diagonal matrix that is pre- and postmultiplied with orthonormal matrices (Jackson, 1991). The symbols and the details of SVD vary depending on the author; here, we use the definitions employed by MATLAB

$$X = USV' \quad (9.4)$$

where X is the $m \times n$ data matrix. This matrix is decomposed into U , an $m \times m$ orthonormal matrix; S , an $m \times n$ diagonal matrix; and V , an $n \times n$ matrix that contains the principle components.*

The matrix U provides a rotation of the original data that will produce a new data set that has zero covariance. However, this matrix is not used since the principal components can be obtained

* The dimensions of matrices S and V vary depending on the source; the dimensions given here are the default values produced by MATLAB's svd routine. The default sizes of both S and V are much larger than needed in PCA, but can be reduced using svd options. The sizes of S and V used in PCA are given in Section 9.2.4.

9.2 Principal Component Analysis

directly from V as described below. The matrix S , which is actually the covariance matrix of the new data set, will have off-diagonal values of zero; hence, it is a diagonal matrix. The diagonal elements of S are the *singular values* or variance of the principle components, which are square roots of the *eigenvalues* and are denoted as $\lambda_1, \lambda_2, \dots, \lambda_n$. These eigenvalues are ordered by value with $\lambda_1 > \lambda_2 > \dots, \lambda_n$. The columns of V are the characteristic vectors or *eigenvectors* u_1, u_2, \dots, u_n . The eigenvectors have the same order as the eigenvalues and should be scaled by their respective singular values (the square root of the eigenvalues) to become the principle components.

9.2.2 Order Selection: The Scree Plot

Since the eigenvalues give the variances (when scaled by $1/N$, the length of the data vectors) of the principle components, they determine what percentage of the total variance is represented by each principal component. (Note that the total variance of the data set is equal to the sum of all eigenvalues.) This is a measure of the associated principal component's importance, at least with regard to how much of the total information it represents. Since the eigenvalues are in order of magnitude, the first principal component accounts for the maximum variance possible, the second component accounts for the maximum of the remaining variance, and so on, with the last principal component accounting for the smallest amount of variance.

The eigenvalues can be very helpful in determining how many of the principal components are really significant and how much the new data set can now be reduced. For example, if several eigenvalues are zero or close to zero, then the associated principal components contribute little to the data and can be eliminated with little loss of information. This also tells us the effective dimension of the data set. Of course, if the eigenvalues are identically zero, then the associated principal component should clearly be eliminated; but where do you make the cut when the eigenvalues are small but nonzero?

There are two popular methods for determining eigenvalue thresholds. (1) Take the sum of all eigenvectors (that must account for all the variances), then delete those eigenvalues that fall below some percentage of that sum. For example, if you want the remaining variables to account for 90% of the variance, then choose a cutoff eigenvalue where the sum of all lower eigenvalues is <10% of the total eigenvalue sum. (2) Plot the eigenvalues in order and look for break points on the slope of this curve. Eigenvalues representing noise should not change much in value and will plot as a flatter slope when plotted sequentially (recall the eigenvalues are in order of large to small). Such a curve is discussed in Chapter 5 and is known as the scree plot (see Figure 5.8). These approaches are explored in Example 9.2.

9.2.3 MATLAB Implementation

9.2.3.1 Data Rotation

Many multivariate techniques rotate the data set as part of their operation. Imaging also uses data rotation to change the orientation of an object or image. From basic trigonometry, it is easy to show that, in two dimensions, rotation of a data point (x_1, x_2) can be achieved by multiplying the data points by the sines and cosines of the rotation angle

$$\begin{aligned}y_1 &= x_1 \cos(\theta) + x_2 \sin(\theta) \\y_2 &= -x_1 \sin(\theta) + x_2 \cos(\theta)\end{aligned}\tag{9.5}$$

where θ is the angle through which the data set is rotated in radians. Using matrix notation, this operation can be done by multiplying the data matrix by a “rotation” matrix consisting of

$$W = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}\tag{9.6}$$

Biosignal and Medical Image Processing

This is the strategy used by the routine `rotation` given in Example 9.1 below. The generalization of this approach to three or more dimensions is straightforward. In PCA, the rotation is done by the algorithm as described below; so, explicit rotation is not required. (Nonetheless, it is required for two of the problems at the end of this chapter and later in image processing.) An example of the application of rotation in two dimensions is given in Example 9.1.

EXAMPLE 9.1

Generate two cycles of a sine wave and rotate the wave by 45°.

Solution

The routine below uses the function `rotation` to perform the rotation. This function operates only in 2-D data. In addition to multiplying the data set by the matrix in Equation 9.6, the function checks the input matrix and ensures that it is in the right orientation for rotation with the variables as columns of the data matrix.

```
% Example 9.1 Example of data rotation
%
N = 100; % Data length
% Create a two variable data set
x(1,:) = (0:N-1)/N; % x1 linear
x(2,:) = 0.1 * sin(x(1,:)*4*pi); % x2 = sin(x1) - two periods
..... plot and label.....
y=... phi = 45*(2*pi/360); % Rotation angle equals 45 deg
y=rotation(x,phi); % Rotate data
..... plot, axis, and label rotated data.....
```

The rotation is performed by the function `rotation` following Equation 9.5.

```
function y = rotation(x,phi)
    .....informative comments.....
%
% Assumes 2-D data and performs 2-D rotation.
[r c] = size(x);
transpose_flag = 'n';
if r < c % Make row vector if needed
    x = x';
    transpose_flag = 'y';
end
W = [cos(phi) sin(phi); -sin(phi) cos(phi)]; % Set up rotation matrix
y = x * W; % Rotate input
```

Results

The results of this example of rotation are shown in Figure 9.6.

9.2.4 PCA in MATLAB

PCA can be implemented using SVD. In addition, the MATLAB Statistics Toolbox has a special program, `princomp`, but this just implements the SVD algorithm. To apply SVD to a data array, X , where the signals are arranged in rows:

```
[U,S,V] = svd(X,'econ');
```

where S is a diagonal matrix containing the singular values and V contains the unscaled principal components in columns. The use of the '`econ`' option reduces the size of S and V matrices considerably. If X is an $m \times n$ matrix where $m < n$, S is an $m \times m$ matrix and V is an $n \times m$

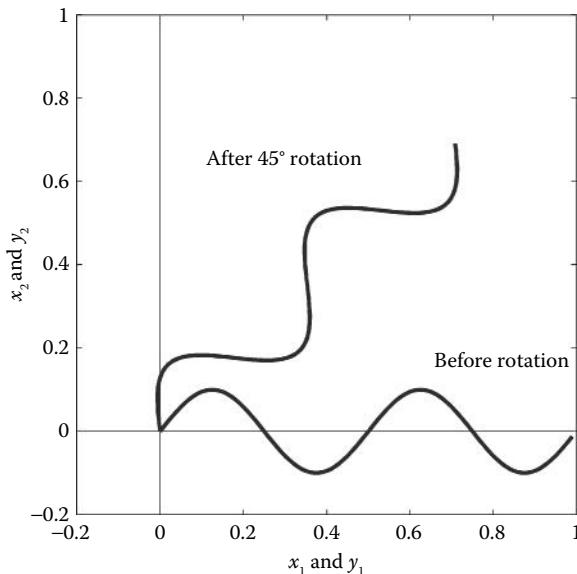


Figure 9.6 A two-cycle sine wave is rotated 45° in Example 9.1 using the function `rotation` that implements Equation 9.6.

matrix. Without this option, the size of S and V is $m \times n$ and $n \times n$, respectively. If X contains signals, the signal length, n , will probably be much greater than m , the number of signals.

The eigenvalues can be obtained from S using the `diag` command:

```
eigen = diag(S) .^2;
```

It is common to normalize the principal components by the variances so that different components can be compared. While a number of different normalizing schemes exist, in the examples here, we multiply the eigenvector by the square root of the associated eigenvalue (i.e., the variances) since this gives rise to principal components that have the same value as a manually rotated data array (see Problem 9.1).

EXAMPLE 9.2

Generate a data set with five variables, but from only two sources and noise. Compute the principal components and associated eigenvalues using SVD. Compute the eigenvalue ratios and generate the Scree plot. Plot the significant principal components.

```
% Example 9.2 Example of PCA
%
N = 1000; % Number points (2 sec of data)
fs = 500; % Sample frequency
t = (1:N)/fs; % Time vector
% Generate data
x = .75 *sin(10*pi*t); % One component a sine
y = sawtooth(7*pi*t,0.5); % One component a sawtooth
%
% Combine x and y in different proportions
D(1,:) = .5*y + .5*x + .1*rand(1,N);
D(2,:) = .2*y + .75*x + .15*rand(1,N);
```

Biosignal and Medical Image Processing

```
D(3,:) = .7*y + .25*x + .1*rand(1,N);
D(4,:) = -.5*y + .4*x + .2*rand(1,N);
D(5,:) = .6* rand(1,N); % Noise only
%
% Center data - subtract mean. There is a more
% efficient way to do this in MATLAB
for i = 1:5
    D(i,:) = D(i,:) - mean(D(i,:));
end
%
% Find Principal Components
[U,S,pc] = svd(D,'econ'); % Singular value decomposition
eigen = diag(S).^2; % Calculate eigenvalues
for i = 1:5 % Scale principal components
    pc(:,i) = pc(:,i) * sqrt(eigen(i));
end
plot(eigen); % Plot scree plot
.....labels and title.....
%
% Calculate Eigenvalue ratio
total_eigen = sum(eigen);
for i = 1:5
    pct(i) = sum(eigen(i:5))/total_eigen;
end
disp(pct*100) % Eigenvalue ratios in %
%
% Output covariance Matrix of principal components
S = cov(pc)
%
.....Plot principal components and original data
```

Results

The five variables are plotted in Figure 9.7. The strong dependence between the variables may seem apparent, but they are not so obvious if you do not already know that they are the product of only two different sources.

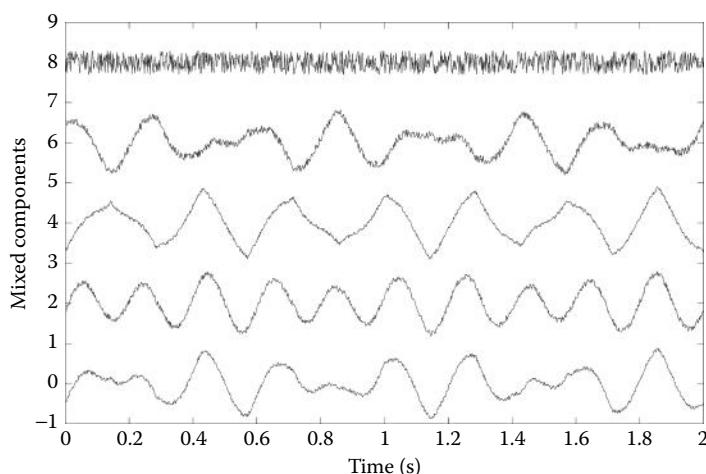


Figure 9.7 Plot of the five variables used in Example 9.2. They were all produced from only two sources (shown in Figure 9.9) and noise.

9.2 Principal Component Analysis

The new covariance matrix taken from the principal components after rotation shows that all five components are uncorrelated and also gives the variance of the five principal components:

0.4366	0.0000	0.0000	0.0000	0.0000
0.0000	0.2016	0.0000	-0.0000	0.0000
0.0000	0.0000	0.0300	-0.0000	-0.0000
0.0000	-0.0000	-0.0000	0.0022	-0.0000
0.0000	0.0000	-0.0000	-0.0000	0.0009

The percentage of variance accounted by the sums of the various eigenvalues is given by the program as

CP 1–5	CP 2–5	CP 3–5	CP 4–5	CP 5
100%	35%	4.9%	0.44%	0.14%

Note that the last three components account for <5% of the variance of the data. This suggests that the actual dimension of the data is closer to two than to five. The Scree plot, the plot of unscaled eigenvalue versus component number introduced in Chapter 5, provides another method for checking data dimensionality. As shown in Figure 9.8, there is a break in the slope at three, again suggesting that the actual dimension of the data set is two since the eigenvalues from three to five do not change much in their value. (Which we know is correct since the data set was created using only two independent sources and noise.)

The first two principal components are shown in Figure 9.9 along with the waveforms of the original sources. While the principal components are uncorrelated, as shown by the covariance matrix above, they do not reflect the two independent data sources. The two principal components account for most of the information in the original data set but they are still mixtures of the two sources. They are uncorrelated, but not independent. This occurs because the sinusoidally based variables do not have a Gaussian distribution, so that decorrelation does not imply independence. Another technique described in the next section can be used to make the variables independent, in which case, the original sources can be recovered.

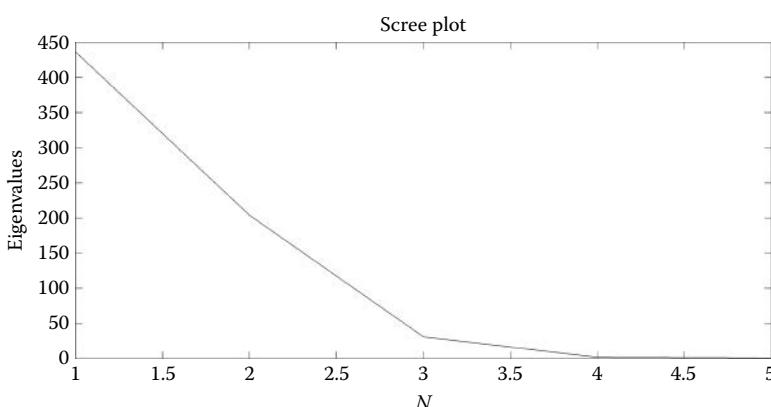


Figure 9.8 Plot of an eigenvalue (unscaled) against the component number, the Scree plot. Since the eigenvalue represents the variance of a given component, it can be used as a measure of the amount of information the associated component represents. A break at three is seen because the last three components have similar, small variances. This suggests that only the first two principal components are necessary to describe most of the information in the data.

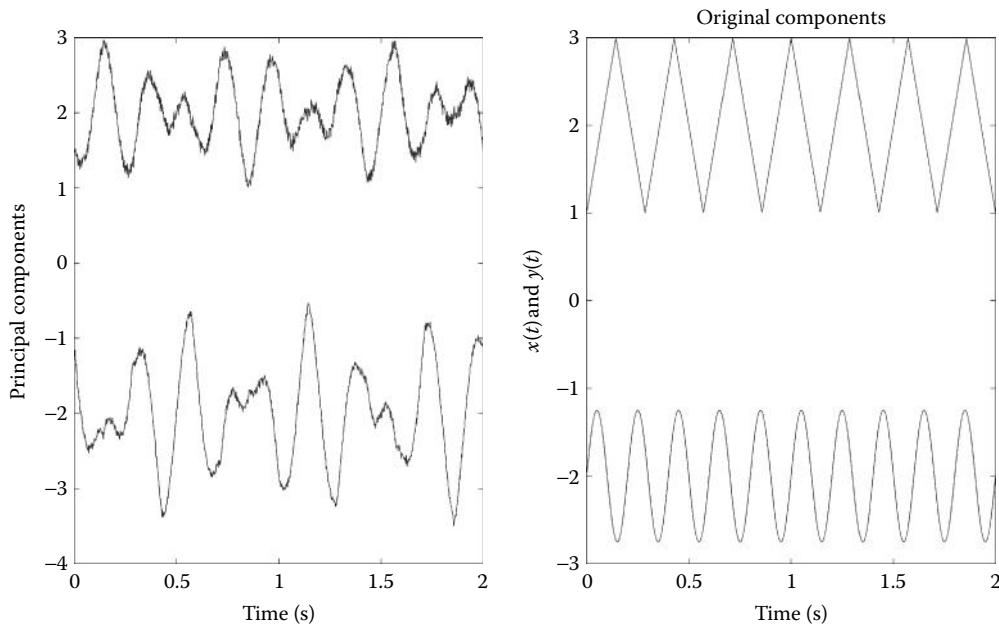


Figure 9.9 Plot of the first two principal components and the original two sources. Note that the components are not the same as the original sources. Even though they are uncorrelated (see covariance matrix above) and they account for most of the information in the two original variables, they are not independent and are still mixtures of these variables.

9.3 Independent Component Analysis

The application of PCA described in Example 9.2 shows that decorrelating the data is not sufficient to produce independence between the variables, at least when the variables have non-Gaussian distributions. ICA seeks to transform the original data set into a number of *independent* variables. The motivation for this transformation is to uncover more meaningful variables, not to reduce the dimension of the data set. When data set reduction is also desired, it is accomplished by *preprocessing* the data set using PCA and then using the dominant principle components in ICA.

One of the most dramatic illustrations of ICA is found in the “cocktail party problem.” In this situation, multiple people are speaking simultaneously within the same room. We assume that their voices are recorded from a number of microphones placed around the room, where the number of microphones is greater than, or equal to, the number of speakers. Figure 9.10 shows this situation for two microphones and two speakers. Each microphone will pick up some mixture of all speakers in the room. Since presumably the speakers are generating signals that are independent (as would be the case in a real cocktail party), the successful application of ICA to a data set consisting of microphone signals should recover the signals produced by the different speakers. In fact, ICA has been quite successful in this problem. Again, the goal is not to reduce the number of signals, but to produce signals that are more meaningful: in the case of the cocktail party, the separated speech of the individual speakers. This problem is found in EEG analysis since all the signals recorded from external electrodes represent combinations of the underlying neural sources.

The most significant computational difference between ICA and PCA is that PCA uses only second-order statistics (such as the variance that is a function of the data squared) whereas ICA uses higher-order statistics (such as functions of the data raised to the fourth power). Variables with a Gaussian distribution have zero-statistical moments above second order, but most *signals*

9.3 Independent Component Analysis

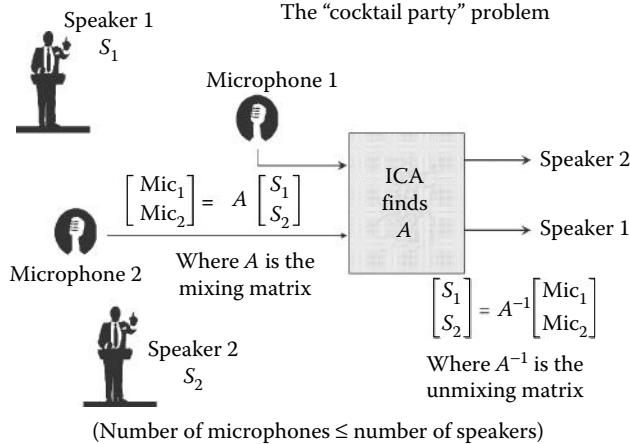


Figure 9.10 Schematic of the “cocktail party problem” where two speakers are talking simultaneously and their voices are recorded by two microphones. Each microphone detects the voice of both speakers. The problem is to unscramble or unmix the two signals from the combinations in the microphone signals. No information is presumed available about the content of the speeches or the placement of the microphones and speakers.

do not have a Gaussian distribution and do have higher-order moments. These higher-order statistical properties are put to good use in ICA.

The basis of most ICA approaches is a generative model, that is, a model that describes how the mixed signals are produced. The model assumes that the mixed signals are the product of instantaneous *linear combinations* of the independent sources. Such a model can be stated mathematically as

$$x_i(t) = a_{i1}s_1(t) + a_{i2}s_2(t) + \dots + a_{iN}s_N(t) \quad (9.7)$$

where $x(t)$ are the signals obtained from the sources, $s(t)$. You have the former and want the latter. Note that this is a series of equations for the N different signal variables, $x_1(t)-x_N(t)$, and in this equation, it is assumed that there are the same number of signals as sources, specifically, N . If there are more signals than sources, PCA can and should be used to reduce the number of signals since the signals contain redundant information. In the discussions on the ICA model equation, it is common to drop the time function. Indeed, most ICA approaches do not take into account the ordering of variable elements; hence, the fact that s and x are time functions is irrelevant.

In matrix form, Equation 9.7 becomes

$$\begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_N(t) \end{bmatrix} = A \begin{bmatrix} s_1(t) \\ s_2(t) \\ \vdots \\ s_N(t) \end{bmatrix} \quad (9.8)$$

where A is a matrix containing the weight constants, a_{ij} . Equation 9.8 can be written succinctly as

$$x = As \quad (9.9)$$

where s is a vector composed of all the source signals, A is the mixing matrix composed of the constant elements a_{ij} , and x is a vector of the measured signals. The model described by

Biosignal and Medical Image Processing

Equations 9.8 and 9.9 is also known as a *latent variables* model since the source variables, s , cannot be observed directly; they are hidden or “latent” in x . Of course, the principle components in PCA are also latent variables; however, since they are not independent, they are usually difficult to interpret. Note that noise is not explicitly stated in the model, although ICA methods will usually work in the presence of moderate noise as shown in Example 9.4. The next example generates a mixture of three signals using a mixing matrix. The resultant data set is used in Example 9.4 in a search for the independent components.

EXAMPLE 9.3

This is an example of signal mixing based on Equation 9.9. Construct a data set, X , consisting of five observed signals that are linear combinations of three different waveforms. Assume $f_s = 500$ Hz and use $N = 1000$ that will generate 2-s signals. The three source signals should consist of a double sine wave, sawtooth wave, and a half-rectified sine wave (positive portion only), all with added noise. Plot the original source signals and the mixture. Save the mixed signals for use in the next example.

Solution

Generate the three signals using MATLAB’s `sin` and `sawtooth` routines. Use the MATLAB `sign` function to produce the half-rectified sine wave. Construct a 5×3 mixing matrix, A , consisting of more-or-less randomly chosen fractions that indicate how much each signal contributes to the five mixtures. Arrange the three signals as rows of a signal matrix. Then apply Equation 9.9 and multiply (using matrix multiplication) the signal matrix with the mixing matrix to produce a data matrix of mixed signals. Center the data before plotting and save the data set and the mixing matrix in the file `mix_sig.mat`.

```
% Example 9.3 Generation of a set of 3 signals mixed 5 ways
%
clear all; close all;
N = 1000; % Number points (2 sec of data)
fs = 500; % Sample frequency
w = (1:N) * 2*pi/fs; % Normalized frequency vector
t = (1:N)/fs; % Time vector
f1 = 8; % Freq sine wave 1
f2 = 26; % Freq sine wave 2
fs = 14; % Sawtooth frequency
%
% Generate three source signals
s1 = 0.5*sin(2*pi*f1*t) + 0.5*sin(2*pi*f2*t) + .07*randn(1,N);
s2 = sawtooth(14*pi*t,.5)+ .05*randn(1,N);
s3 = sign(sin(2*pi*t)).*sin(2*pi*t) + .07*randn(1,N);
%
% Plot original signals
plot(t,s1-2.5,'k',t,s2,'k',t,s3+1.8,'k'); % Plot source signals displaced
% .....labels and title
%
% Combine signals. Define mixing matrix
A = [.5 .5 .5; .2 .7 .7; .7 .4 .2; -.5 .2 -.6; .7 -.5 -.4];
s = [s1; s2; s3]; % Signal matrix
X = A * s; % Eq. 9.9
save mix_sig.mat X A; % Save waveform and matrix
figure; hold on;
% Center data and plot mixed signals
for i = 1:5
    X(i,:) = X(i,:)-mean(X(i,:)); % Remove means
```

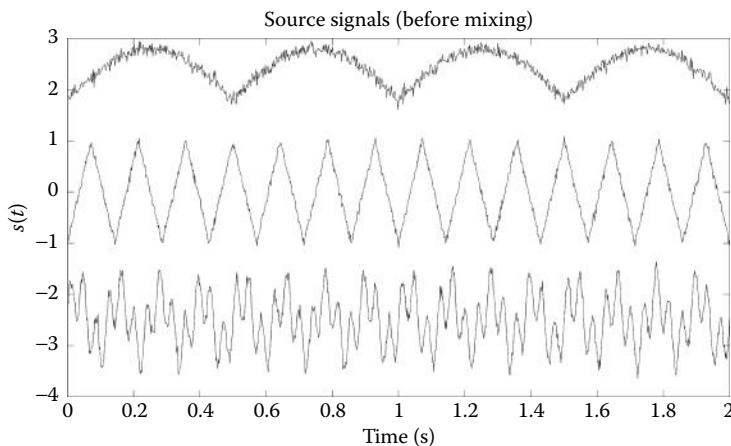


Figure 9.11 Three source signals used to create the mixture seen in Figure 9.12 and are used in Example 9.3.

```
plot(t,X(i,:)+2*(i-1), 'k'); % Plot mixture. Offset signals
end
.....labels, titles.....
```

Results

The original source signals are shown in Figure 9.11.

The source signals are mixed in different proportions using the mixing matrix A to produce the five signals shown in Figure 9.12. The original signals are not discernible in the mixture, nor is it obvious that there are only three source signals in this mixture.

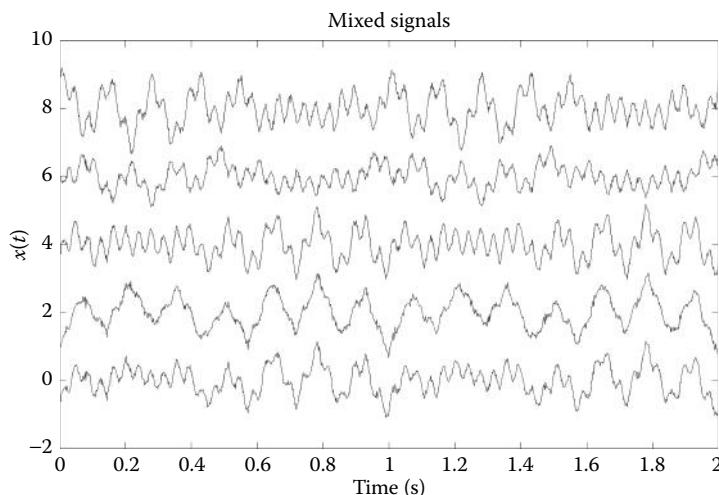


Figure 9.12 Five signals created by mixing three different noisy waveforms. In Example 9.4, ICA is applied to this data set to recover the original signals.

Biosignal and Medical Image Processing

ICA techniques are used to solve the mixing matrix, A , from which the independent components, s , can be obtained through matrix inversion:

$$s = A^{-1}x \quad (9.10)$$

In Figure 9.10, if the exact placement of the microphones and speaker is known, then it might be possible to determine the mixing matrix A from the geometry of the microphone and source positions. Then the underlying sources can be found simply by solving Equation 9.10. However, ICA is used in the more general situation where the physical situation that produced the mixed signals is unknown; so, the mixing matrix is also unknown. For this reason, ICA is one approach to a basic problem known as *blind source separation*. The basic idea is that if the measured signals, x , are related to the underlying source signals, s , by a linear transformation (i.e., a rotation and scaling operation captured by the mixing matrix, A), then some inverse transformation (specifically that of the unmixing matrix, A^{-1}) can be found that recovers the original signals. To estimate the mixing matrix, ICA needs to make only two assumptions: that the source variables, s , are truly independent,* and that they are non-Gaussian. Both conditions are usually met when the sources are real signals. A third restriction is that the mixing matrix must be square; in other words, the number of sources must equal the number of measured signals. This is not really a restriction since, as long as there are at least as many signals as sources, PCA can be applied to reduce the dimension of the signal data set, x , to equal that of the source data set, s .

The requirement that the underlying signals must be non-Gaussian stems from the fact that ICA relies on higher-order statistics (i.e., higher than second order) to separate the variables. As noted above, the higher-order statistics (i.e., moments and related measures) of Gaussian signals are zero. ICA does not require that the distribution of the source variables be known; it only requires that they must not be Gaussian. Note that if the measured variables are already independent, ICA has nothing to contribute, just as PCA is of no use if the variables are already uncorrelated.

The only information available to ICA is the measured variables; it has no information on either the mixing matrix, A , or the underlying source variables, s . Hence, there are some limits to what ICA can do: there are some unresolvable ambiguities in the components estimated by ICA. First, ICA cannot determine the variances, hence the energies or amplitudes of the actual sources. This is understandable if one considers the cocktail party problem. The sounds from a loudmouth at the party could be offset by the positions and gains of the various microphones, making it impossible to definitively identify this excessive volume. Similarly, a soft-spoken party goer could be closer to a number of microphones and appear unduly loud in the recorded signals. Unless something is known about the mixing matrix (in this case the position and gains of the microphones with respect to the various speakers), this ambiguity cannot be resolved. Since the amplitude of the sources cannot be resolved, it is usual to fix the amplitudes so that a signal's variance is one. It is also impossible, for the same reasons, to determine the sign of the source signal, although this is not usually of much concern in most applications.

A second restriction is that, unlike PCA, the order of the components cannot be established. This follows from the arguments above: establishing the order of a given signal requires some information about the mixing matrix, which, by definition, is unknown. Again, in most applications, this is not a serious shortcoming.

The determination of the independent components begins by removing the mean values of the variables, also termed *centering* the data, as in PCA. The next step is to *whiten* the data, also known as *sphering* the data when combined with centering. Data that have been whitened are

* In fact, the requirement for strict independence can be somewhat relaxed in many situations.

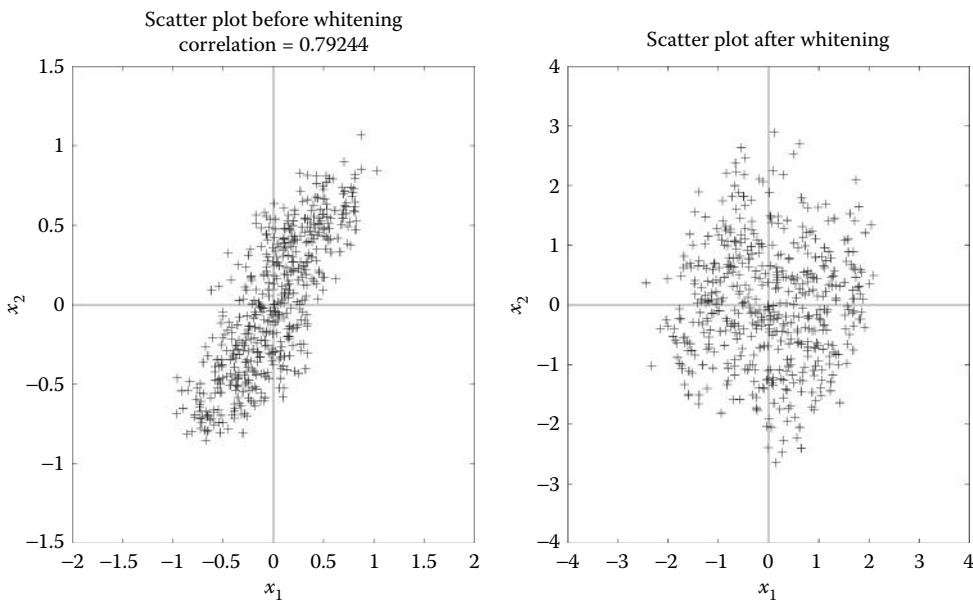


Figure 9.13 Two-variable multivariate data before (left) and after (right) whitening. Whitened data have been decorrelated and the resultant variables have been scaled so that their variance is one. The whitened data generally have a circular shape. A whitened three-variable data set has a spherical shape, hence the term *sphericalizing* the data.

uncorrelated (as are the principal components) and, in addition, all the variables have variances of one. PCA can be used for both these operations since it decorrelates the data and provides information on the variance of the decorrelated data in the form of eigenvectors. Figure 9.13 shows the scatter plot of the data used in Figure 9.2 before and after whitening using PCA to decorrelate the data and then scaling the components to have unit variances.

The unmixing matrix, A^{-1} , is determined by applying a linear transformation to the whitened data. There are quite a number of different approaches for estimating A^{-1} , but they all make use of an *objective function* that relates to variable independence. This function is maximized (or minimized) by an optimization algorithm. The various approaches differ in two ways: technically, in the optimization method used; and more definitively, in the specific metric used for measuring independence, the objective function.

One of the most intuitive approaches uses an objective function that is related to the non-Gaussianity of the data set. This approach takes advantage of the fact that mixtures tend to be more Gaussian than the distribution of independent sources. This is a direct result of the central limit theorem, which states that the sum of k independent, identically distributed random variables converges to a Gaussian distribution as k becomes large regardless of the distribution of the individual variables. Hence, mixtures of non-Gaussian sources will be more Gaussian than the unmixed sources. This was demonstrated in Figure 2.3 using averages of uniformly distributed random data. Here, we demonstrate the action of the central limit theorem using a deterministic function: a sine wave. In Figure 9.14a, a Gaussian distribution is estimated using the histogram of a 10,000-point sequence of Gaussian noise as produced by the MATLAB function `randn`. A distribution that is closely aligned with an actual Gaussian distribution (dotted line) is seen. A similarly estimated distribution of a single sine wave is shown in Figure 9.14b along with the Gaussian distribution. The sine wave distribution (solid line) is very different from a Gaussian distribution (dashed line). However, a mixture of only two independent sinusoids

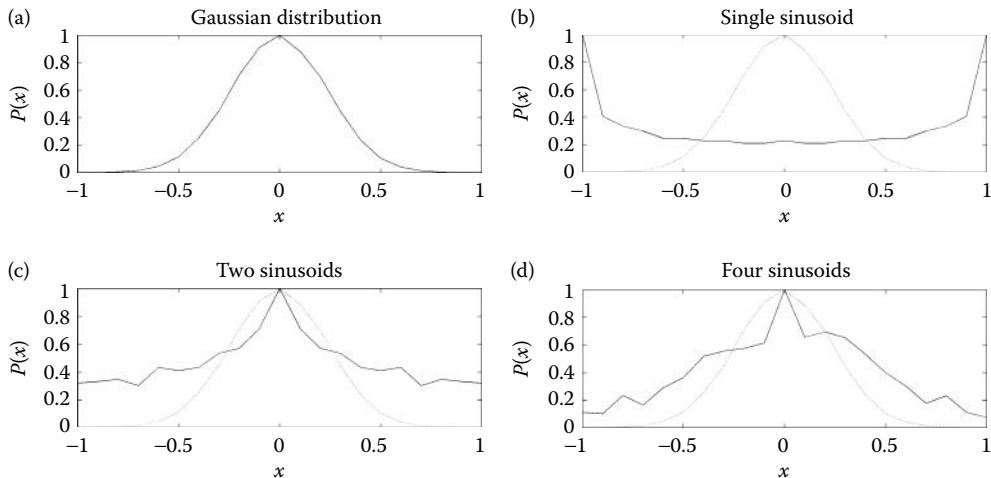


Figure 9.14 Approximate distributions for four variables determined from histograms of 40,000-point waveforms. (a) Gaussian noise (from MATLAB `randn` function). (b) Single sinusoid at 100 Hz. (c) Two sinusoids mixed together (100 and 30 Hz). (d) Four sinusoids mixed together (100, 70, 30, and 25 Hz). The more sine waves that are mixed together, the more the distributions approach a Gaussian distribution.

(having different frequencies) is seen to be much closer to the Gaussian distribution (Figure 9.14c). The similarity improves as more independent sinusoids are mixed together. As shown in Figure 9.14d, the distribution becomes even more Gaussian when four sinusoids (nonharmonically related) are added together.

To take advantage of the relationship between non-Gaussianity and component independence requires a method to measure Gaussianity or lack of Gaussianity. With such a measure, it is possible to find A^{-1} by rotating the original data set until the non-Gaussianity of the transformed data set is maximized. One approach to quantifying non-Gaussianity is to use *kurtosis*, the fourth-order cumulant of a variable. This metric is zero for Gaussian data and nonzero otherwise. For data having zero mean (i.e., centered), the formal definition of kurtosis is

$$\text{Kurt}(x) = E\{x^4\} - 3[E\{x^2\}]^2 \quad (9.11)$$

where E represents the expectation operator, but for real data, the mean is used. Note that for real data that have zero mean, $E\{x^2\}$ is just the variance, σ^2 . If the data are whitened, then $\sigma^2 = 1.0$ and $[E\{x^2\}]^2 = 1.0$; so, in practical situations for spherred data (whitened and centered), Equation 9.11 becomes

$$\text{Kurt}(x) = \text{mean}(x^4) - 3 \quad (9.12)$$

Although the MATLAB Statistical Toolbox has a routine for calculating kurtosis, it is easy to calculate using standard MATLAB. Kurtosis can be either positive or negative depending on the shape of the distribution. The distributions associated with negative kurtosis are called *sub-Gaussian* and have distributions that are broad and flat, whereas those associated with positive kurtosis are called *super-Gaussian* and have spiky distributions.

Kurtosis has a linear additive property that simplifies the calculation of the overall kurtosis for multiple variables. Specifically, if x_1 and x_2 are independent variables

$$\text{Kurt}(x_1 + x_2) = \text{kurt}(x_1) + \text{kurt}(x_2) \quad (9.13)$$

Using a metric that involves the fourth power of data does present particular problems as the influences of outliers can be greatly enhanced. Accordingly, the ICA algorithms that use kurtosis may compress the data using a nonlinear function to restrain outliers before taking the fourth power. An example of using kurtosis to separate two mixed signals manually is given in the problem set.

Other popular metrics that have been used to measure independence in ICA algorithms include the information-theoretic processes of negentropy and mutual information. The latter is covered in Chapter 10, whereas entropy measurements are described in Chapter 11. An excellent treatment of the various approaches and their strengths and weaknesses can be found in Hyvärinen et al. (2001).

9.3.1 MATLAB Implementation

The development of an ICA algorithm is a nontrivial task; however, a number of excellent algorithms can be downloaded from the Internet in the form of MATLAB *m*-files. Two particularly useful algorithms are the “FastICA” algorithm developed by the ICA group at the Helsinki University:

<http://www.cis.hut.fi/projects/ica/fastica/fp.html>

and the “Jade” algorithm for real-valued signals developed by J-F Cardoso:

<http://sig.enst.fr/~cardoso/stuff.html>.

The Jade algorithm is used in the example below, although the FastICA algorithm allows greater flexibility including an interactive mode. The Jade algorithm is provided in the accompanying data files and is called using

```
W = jade(X, nu_ica); % Application of ICA
```

where *X* is the mixed signal matrix with the signals in rows, *nu_ica* is the number of desired independent components, and *W* is an $m \times n$ unmixing matrix where *m* is the number of independent components and *n* is the number of rows in *X*. To get the original source signals, simply multiply the unmixing matrix by the data matrix:

```
S = W*X; % Unmix the signals
```

In the next example, the mixed signals created with a mixing matrix in Example 9.3 are used to evaluate the Jade ICA algorithm.

EXAMPLE 9.4

Use the mixed signal matrix, *x*, constructed in Example 9.3 and found in the file *mixed_sig.mat* in an evaluation of the Jade ICA algorithm. In this example, we pretend that we do not know the number of source signals present, as would be the case in a real-world problem.

Solution

After loading the data, we apply PCA to the data set and plot the Scree plot as well as the percent variances as in Example 9.2. After examining these two items, we input the desired number of independent components to the Jade algorithm. After applying the Jade algorithm as described above, we plot the unmixed signals.

```
% Example 9.4 Apply the Jade ICA algorithm to the matrix of mixed signals,
% X, in file mix_sig.mat.
```

Biosignal and Medical Image Processing

```
%  
load mix_sig.mat; % Get data  
% Assign constants  
N = 1000; % Number of points  
fs = 500; % Sample frequency  
t = (1:N)/fs; % Time vector for plotting  
  
% Do PCA and plot Eigenvalues  
[U,S,pc] = svd(X,'econ');  
eigen = diag(S).^2;  
plot(eigen,'k');  
.....labels and title.....  
  
%  
total_eigen = sum(eigen); % Find percent variance  
for i = 1:5  
    pct(i) = 100 * sum(eigen(i:5))/total_eigen;  
end  
disp(pct) % Display percent variances  
% Ask for number of independent components  
nu_ICA = input('Enter the number of independent components');  
  
% Compute ICA  
W = jadeR(X,nu_ICA); % Determine unmixing matrix  
ic = W*X; % Find independent components  
figure hold on; % Plot independent components  
for k = 1:nu_ICA  
    plot(t,ic(k,:)+4*(k-1),'k');  
end  
.....labels and title.....
```

Results

The Scree plot of the eigenvalues obtained from the five-variable data set does show a break at two and three and goes to zero at four, suggesting that there are two or three separate components (Figure 9.15). The percent variances indicate that none of the variance is represented by the last two

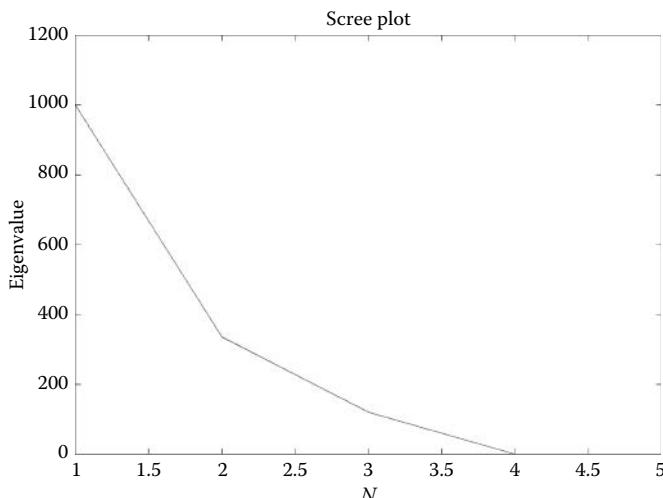


Figure 9.15 Scree plot of eigenvalues from the data set of Figure 9.12. There is a break at $N=2$ and at $N=3$ and the plot goes to zero at $N=4$, suggesting that there are three, but possibly two, independent variables in the data set of five waveforms.

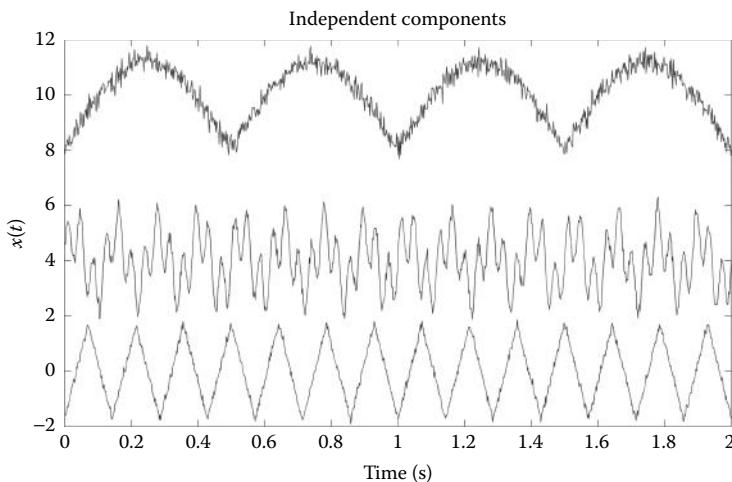


Figure 9.16 Three independent components found by ICA in Example 9.3. Note that these are nearly identical to the original, unmixed components. The presence of a small amount of noise does not appear to hinder the algorithm.

components, but 8.25% is contained in the last three components, but really just the third component, suggesting that three components are present. Hence, the ICA algorithm will be requested to search for three components.

CP 1–5	CP 2–5	CP 3–5	CP 4–5	CP 5
100%	31.3%	8.25%	0.0%	0.0%

Applying ICA to the five-variable mixture in Figure 9.12 recovers the original source signals as shown in Figure 9.16. Comparing this figure with the original signals in Figure 9.11 shows them to be the same, except for a slight change in signal amplitude that is expected since, as mentioned above, ICA cannot recover the original energy levels. This figure dramatically demonstrates the ability of this approach to recover the original signals even in the presence of modest noise. ICA has been applied to biosignals to estimate the underlying sources in EEG signals, improve the detection of active neural areas in functional magnetic resonance imaging (fMRI), and to uncover the underlying neural control components in an eye-movement motor-control system. Given the power of the approach, many other applications are sure to follow.

9.4 Summary

The analyses presented in this chapter use linear transformations applied to multivariate data: multiple signals or data that relate to each other. The two major analysis techniques discussed in this chapter apply only to signals that can be considered as different observations of the same events. The common examples of multidimensional data in biomedical engineering include EEG recordings where multiple electrodes record brain activity and image analyses where multiple images are taken of the same area as in fMRI (see Chapter 15).

Multivariate signals are envisioned as occupying a multidimensional space where the spatial dimension equals the number of signals (or separate observations for other data). Scatter plots show signals plotted against one another on a point-by-point basis and provide an overview of the signal space. The goal of PCA is to determine the true dimension of a data set and

Biosignal and Medical Image Processing

reduce the dimensionality if possible. In the geometric interpretation of PCA, the data set is rotated through multiple dimensions until the signal in each dimension is uncorrelated with (i.e., orthogonal to) the signals (or data) in all other dimensions. It is assumed that the signals have zero mean and are similarly scaled (i.e., they all use the same or equivalent measurement systems). The decorrelated signals are termed the principal components and they are ranked according to how much their variance accounts for the total variance in the data set. If that percentage is very small, we might consider eliminating these signals as it contributes little information to the overall data set.

While the principal components provide an insight into how much data we need to retain in a multivariable data set, they usually do not have greater meaning than the original data. That is the task of ICA: to find the more meaningful source signals that may be mixed together in the data set. ICA operates to unmix the signals without knowing anything about the mixing process. This approach makes two assumptions: that the mixing was linear and that the original signals were truly independent. As with PCA, ICA works using a linear transformation that can be visualized as data rotation through multidimensional space. There are a number of different approaches to this task, but they are all based on a trial-and-error search for the rotation that produces the most independent signals within the various dimensions. The methods differ in two ways: the technique used to implement the optimization, and the metric used to measure independence. As mixtures of signals have distributions that are more Gaussian than those of the unmixed signals, methods based on producing signals that are the least Gaussian are popular. One method of measuring how much a signal deviates from Gaussian is kurtosis, which is based on the fourth moment of the data and is zero for Gaussian signals. Other measures of independence that are used include negentropy and mutual information. Many MATLAB-based ICA algorithms are available on the web.

PROBLEMS

- 9.1 This is an example of manual PCA. Load the two-variable data set, X , contained in the file `p9_1_data`. Assume $f_s = 500$ Hz. Although you are not given the dimensions *or orientation* of the data set, you can assume that the number of time samples is much greater than the number of measured signals. (a) Rotate these data by an angle entered through the keyboard and output the covariance (from the covariance matrix) after each rotation. Use the function `rotation` to do the rotation. See comments in the help file or function itself for details. Continue to rotate the data set “manually” until the covariances are very small ($<10^{-4}$). Plot the rotated and unrotated variables as a scatter plot and output their variances (also from covariance matrix). The rotated data will be the principal components. (b) Now, apply PCA using the approach given in Example 9.2 and compare the scatter plots with the manually rotated data. Compare the variances of the principal components from PCA (that can be obtained from the covariance matrix) with the variances obtained by manual rotation in Part (a) above. Note that after your manual rotation, your principal components may not be in rank order in which case the scatter plots will be perpendicular to one another.
- 9.2 Load the multivariable data file, `prob9_2_data.mat` with a matrix variable X that consists of five signals ($f_s = 500$ Hz). Apply PCA to this data set to determine the actual dimension of the data. Use a plot of the eigenvalues (Scree plot) to estimate the dimensionality of the data. Also estimate the dimensionality of the data set using the percentage of variance accounted by the sums of the various eigenvalues as in Example 9.2. Note that the Scree plot provides a less-definitive cutoff than the variance percentages.

- 9.3 Repeat Problem 9.2 using the data set X in the file, `prob9_3_data.mat`. This data set also consists of five signals ($f_s = 500$ Hz). Apply PCA and use both the Scree plot and percentage of variance to estimate the dimensionality of the data. Note that unlike Problem 9.2, the two methods clearly lead to different results. This shows that measurement to assess data dimensionality cannot always be trusted.
- 9.4 This is an example of manual ICA. Load the data file `p9_1_data.mat` that contains a 2-D variable in X . This problem is the ICA equivalent to Problem 9.1. You are to find the independent components using a manual approach. Determine the kurtosis for each of the two signals in the file and sum. Rotate the data set using rotation to find the maximum (or minimum if the kurtosis sum is negative) value. Apply the Jade ICA algorithm and compare the results with the manual separation.
- 9.5 Load the multivariable data set, X , contained in the file '`p9_2_data`'. Make the same assumptions with regard to sampling frequency and data set size as in Problem 9.1 above. (a) Determine the actual dimension of the data using PCA and the Scree plot. (b) Perform an ICA analysis using either the "Jade" or "FastICA" algorithm, limiting the number of components determined from the Scree plot. Plot the independent components.
- 9.6 Load the file `music_mix.mat` that contains three variables: $x1$, $x2$, and fs . The variables $x1$ and $x2$ contain different mixtures of two pieces of music. Play each piece using `sound(x1)` and `sound(x2)`. (The two mixtures are sampled at the default for the `sound` routine; so, a second argument is not required.) Arrange $x1$ and $x2$ in a matrix and apply ICA (note transformations may be required). Play the resulting independent components using `sound`. Note that since ICA normalizes the independent components, they will sound better if you divide the components by 10 before playing them using `sound`. In this problem, ICA is able to achieve near-perfect separation.
- 9.7 This is an example of the limitations of ICA. Load file `mix_sig3.mat` that contains the data set in matrix X ($f_s = 500$ Hz). Note that the orientation of X may need adjustment. Use PCA to estimate the number of independent components, then apply ICA using the Jade algorithm to separate these components. For this data set, ICA is unable to completely separate all the components. Compare the components found by ICA with the actual source components. (The actual source signals can be found in matrix s , also in the data file.) You should be able to explain why ICA failed with those components it did not correctly separate.

10

Chaos and Nonlinear Dynamics

10.1 Nonlinear Systems

So far, we have only considered the analysis of signals produced by linear systems, but many important biological systems are nonlinear. In nonlinear systems, the superposition principle does not hold as it would in a linear system. In a linear system, the principle of superposition holds: the output to a linear combination of signals is equal to a linear addition of outputs produced by the individual signals. For a nonlinear system, this is generally not true.

A simple example of a nonlinear system is the mathematical system:

$$y(t) = x^2(t) \quad (10.1)$$

If the input to this system is a value A , then the output is A^2 , whereas the output for an input of value B is B^2 . If this system was linear, the output to the combined input of $A + B$ would be $A^2 + B^2$, but in this system, it is $(A + B)^2 = A^2 + 2AB + B^2$. Hence, in the simple nonlinear system represented by Equation 10.1, superposition does not hold.

Of special interest in this chapter is how nonlinearities in a system influence the signals produced by these systems. These properties are discussed throughout the chapter but simply put, signals from a nonlinear system contain irregularities that appear more complicated than signals from linear systems. Complex signals are not always the product of nonlinear systems, but if a measured signal appears complicated, there is a chance that it comes from a nonlinear system. Depending on the conditions, signals from nonlinear systems may exhibit only weak nonlinear properties; however, we refer to any signal produced by a nonlinear system as a nonlinear signal even if the signal contains only weak nonlinearities. We will see that nonlinear characteristics can make signal analysis challenging and require new techniques.

While linear methods, such as spectral analysis, can be applied to signals from nonlinear systems, these signals have features that cannot be revealed by linear signal processing. For this reason, it is important to have some idea of whether or not your signal comes from a nonlinear system. If you know that your signal is not the product of a nonlinear system or contains no nonlinear properties, then nonlinear analyses are pointless. Additionally, some of the methods discussed in this section lack specificity for nonlinear behavior and can give misleading results for a complicated, yet linear, signal. If you incorrectly assume that your signal is nonlinear, some nonlinear analyses may support that assumption, but you would be wrong.

Fortunately, there are reasonable methods to determine if your signal contains nonlinearity. The best assessment would be based on *a priori* knowledge that the system producing your signal is nonlinear. If you know that the system contains nonlinear properties, then the assumption

that you need nonlinear analyses is reasonable. In many cases, however, the nature of the system that produces your signal is unknown, and we need to apply a formal method of testing for nonlinearity known as *surrogate data analysis*. This analysis uses artificial reconstructions of the signal to probe for nonlinear properties. Since understanding the nature of your signal is of primary importance, it is logical that we discuss surrogate data testing first. However, surrogate data testing relies on nonlinear analysis methods; so, while these tests may be the first to be performed on an unknown signal, their presentation must wait until the end of this chapter.

10.1.1 Chaotic Systems

A nonlinear system is one in which the governing equations contain a nonlinear term, such as $\sin(x)$ or x^2 as in Equation 10.1. The presence of nonlinear terms introduces complicated features into the output of the system. These signals can become so complex that they have no apparent long-term pattern and seem unpredictable over long timescales. These systems may also be extremely sensitive to the initial system state. Signals that exhibit both properties are termed chaotic systems. *Chaos* is deterministic behavior, but may appear to contain little structure or may even appear to be random. However, chaotic signals do contain structure that can be revealed through appropriate analyses.

An example of a nonlinear system whose outputs may contain both linear and nonlinear behavior is the damped, driven pendulum. This is analogous to pushing a child on a swing. A damped, driven pendulum has force applied periodically and is damped by friction with air. The push tends to increase the velocity of the pendulum whereas the damping decreases the velocity.

The equations of motion for the pendulum are given as a system of first-order differential equations:

$$\dot{\theta} = \omega \quad (10.2)$$

$$\dot{\omega} = -\frac{g}{l}\sin(\theta) + k\sin(t) - b\omega \quad (10.3)$$

where k is the magnitude of the driving force, θ is the angle of the pendulum, ω is angular velocity, g is the gravitational constant, l is the length of the pendulum, and b is the damping factor. Although this system can be described by a single equivalent second-order equation, the first-order, dual-equation format used here is required by MATLAB's differential equation solver.

Equation 10.3 is nonlinear because of the $\sin(\theta)$ and $\sin(t)$ terms, but if θ is small, the solution is approximately linear because for small angles, $\sin(\theta) \approx \theta$ and the sine function approaches linearity. If there is no driving force (i.e., $k = 0$), the $\sin(t)$ term is eliminated. With these two modifications, Equations 10.2 and 10.3 then describe a linear, damped second-order system similar to a mass–spring system.

As a real pendulum is free to swing at any angle, the angle can become large enough so that the small angle approximation no longer holds. The behavior of the pendulum then becomes more complicated and can include inversion (the weight goes around completely) and/or large angle deflections. The solutions to the differential equation are also more complicated and cannot be described by simple harmonic motion. The introduction of a driving force allows for even more complicated motion, including chaotic solutions, as described in the following example.

EXAMPLE 10.1

Solve the differential equations, Equations 10.2 and 10.3, using parameters and initial conditions that give linear, nonlinear, and chaotic solutions. For all three situations, assume friction is zero, that is, $b = 0$. For the linear case, remove the driving force ($k = 0$) and make the initial displacement small: $\theta_0 = \pi/100$ rad (1.8°). Since no energy is being added to the system, the angle

of the pendulum will never reach a value greater than its initial displacement. Thus, the angle is always small and within the linear region of the sine function. This system behaves in a linear manner. For the nonlinear case, we simply increase the initial angle to $\theta_0 = \pi/1.1$ rad (164°). To create chaotic behavior, we return to the small initial displacement but add a driving force, $k = 0.75$. The driving force adds energy to the system and the angular displacement can become greater than the linear region of the sine function, leading to nonlinear solutions. This force is with respect to the gravitational constant that is normalized to 1.0. Additionally, to simplify the differential equations, the ratio of the gravitational constant to the length of the pendulum (g/l in Equation 10.3) is set to 1.0.

Solution

Use the MATLAB differential equation solver `ode45` to solve for the linear, nonlinear, and chaotic solutions to the motion of the pendulum. The different solutions are obtained by varying the parameters of Equation 10.3 as described above. The resulting time functions, $\theta(t)$, are plotted.

The MATLAB function `ode45` uses the Runge–Kutta algorithm to solve differential equations numerically. Using `ode45` is a multistep process, a short guide to the application of this function to solve differential equations is given in Appendix A. Here, we only discuss using `ode45` in the context of this specific example. The function call needed in this example is

```
[t,sol] = ode45(@pend,tspan,initial_values,[],b,k);
```

The outputs are t , a vector of time samples, and sol , a matrix containing solutions to the differential equations. In our example, the differential equation system defines $\dot{\theta}$ and $\dot{\omega}$; therefore, the first and second columns of sol are θ and its first derivative, ω . The input `pend` is the filename of a function that defines the differential equation system in Equations 10.2 and 10.3. The construction of `pend` is specific for MATLAB differential equation solvers and is discussed in more detail in Appendix A, where the function can be found; for now, it is sufficient to understand that this function specifies the relevant equations. The second argument, `tspan`, defines the beginning and ending times as a two-element vector: here, we use 0.0 for the start time and 100 as the end time. The third input, `initial_values`, contains the initial conditions for θ and ω . The initial condition for ω will always be 0.0, but the initial condition for θ will depend on the desired behavior: small ($\pi/100$ rad) for a linear response and large ($\pi/1.1$ rad) for a nonlinear response. For the chaotic response, the initial condition for θ will again be small, but a nonzero driving force will be used. The brackets, `[]`, indicate that default accuracy tolerances are to be used. The last two input arguments are the parameters passed to the `pend` routine: b is damping factor b and k is forcing factor k .

```
% Example 10.1
% Solve the pendulum equations for three initial conditions
%
% Linear condition
initial_values = [pi/100,0] % Small initial θ, 0 initial ω
k=0; % Driving force is 0
b=0; % Damping factor is 0
tspan=[0,100]; % Start and end times
%
[t,sol] = ode45(@pend,tspan,initial_values,[],b,k); % Eq. solver
theta=sol(:,1); % Solution for θ and ω
.....label and plot.....
%
% Nonlinear condition
initial_values = [pi/1.1,0]; % Large initial angle
```

```

%
[t,sol] = ode45(@pend,tspan, initial_values,[],b,k); % Eq. solver
.....label and plot.....
%
% Chaotic condition
theta_0 = [pi/100,0]; % Small initial angle.
k = .75; % Add driving force
tspan = [0,1000]; % Extend time frame
%
[t,sol] = ode45(@pend,tspan,initial_values,[],b,k); % Eq. solver
theta = sol(:,1);
theta = mod(pi+theta,2*pi)-pi; % Circular boundary conditions
.....label and plot.....

```

Analysis

Note that for the solution in which the driving force is large, for example, in the third case, in which the driving force variable, k , equals 0.75, it is possible for the pendulum to invert: the pendulum may be driven hard enough to go over the top. This produces a rotation greater than π rad, which is technically correct but not appropriate for plotting where we want to describe the angle of the pendulum weight relative to 0 rad (the straight down position). Therefore, we use the MATLAB modulus function, `mod`, to reduce the angle to its equivalent value within $-\pi < \theta \leq \pi$.

Results

The output from this example is shown in Figure 10.1. There are stark differences in the behavior of the system when conditions are changed. The first trace in Figure 10.1a shows typical linear

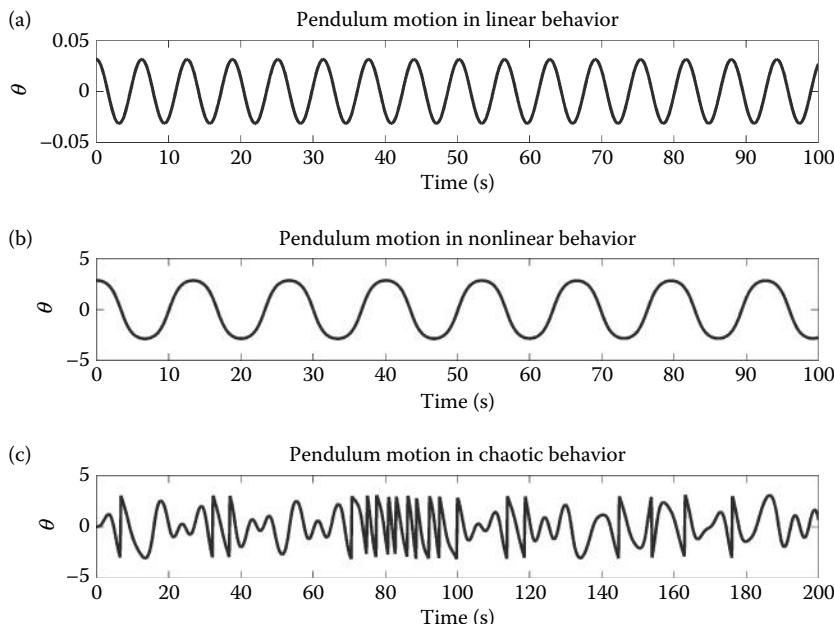


Figure 10.1 The pendulum equations (Equations 10.2 and 10.3) are solved for three conditions: (a) No driving force and a small initial angle giving rise to linear behavior; (b) no driving force, but a large initial angle leading to nonlinear behavior; and (c) small initial angle, but with a driving force leading to chaotic behavior.

Table 10.1 Behavior of a Forced Pendulum for Various Conditions and Parameters

Initial $\theta >$ (rad)	Force Parameter k	Behavior
$\pi/100$	0	Linear
$\pi/1.1$	0	Nonlinear (but still regular)
$\pi/100$	0.75	Chaotic

behavior, a sine wave. In this condition, the pendulum is not given an initial velocity or a driving force and the initial deflection angle of the pendulum is small enough to be in the linear region. For a larger deflection, the pendulum is no longer in the linear region; so, while the middle trace demonstrates regular periodic behavior, it is no longer sinusoidal (Figure 10.1b): the troughs and peaks are too wide. In the lower plot (Figure 10.1c), we see an example of chaotic behavior. This motion is not regular or periodic and it is difficult to predict the behavior. Linear analyses such as spectral analysis can no longer fully describe the signal, but we develop tools to quantify this type of behavior throughout this chapter.

Table 10.1 summarizes the behavior of the pendulum as a result of different initial conditions. This example is not an exhaustive exploration of the behavior of a pendulum, but it does show that even a simple mechanical object may exhibit complex, chaotic behavior. The source of the chaotic behavior is the driving force that induces two effects. First, it can drive the pendulum beyond the limit of the small angle approximation, introducing nonlinearities. The second effect is more complicated, a phenomenon known as a *bifurcation*. By allowing the pendulum to invert, we set up a situation where a small difference in the initial velocity or driving force can induce a large change in the output. If the pendulum does not invert, it will show nonlinear but regular periodic behavior. However, a slightly larger velocity can produce a radically different behavior, for if it inverts, it will swing in the *opposite* direction. Thus, only a slight change in the pendulum's initial conditions can lead to a dramatic change in the evolution of velocity and position profiles after a period of time. This is an example of sensitivity to the initial conditions and is a hallmark of chaos. Later, in this chapter, we discuss how this sensitivity can be quantified.

10.1.2 Types of Systems

The properties of the signals produced by any system depend on the behavior of a system's components and its input (if it has an input). Linear systems produce linear outputs, but if a system is not time invariant, these signals can be nonstationary or *stochastic* (i.e., having some random variation) resulting from time-varying or randomly varying components. Even deterministic, linear time-invariant systems can produce nonstationary or stochastic output signals if they have nonstationary or stochastic input signals. Nonlinear systems also generate signals that reflect both their input signals and component behavior. Given nonstationary or stochastic inputs and/or components, nonlinear systems produce signals with those properties. However, deterministic, time-invariant nonlinear systems with deterministic inputs can produce chaotic output signals under certain circumstances as seen in Figure 2.1.

Deterministic systems can be described by a system of differential equations, at least in principle. These equations may be very complicated and they may not always be known. A deterministic signal has no randomness in its behavior and it is possible to predict the future behavior of the signal from past values. A noise-free sine wave is an example of a deterministic signal; it can be produced by a linear system as demonstrated in Example 10.1 (Figure 10.2, upper curve). We can predict with perfect accuracy the value of a sine wave at any time knowing only the frequency, amplitude, and phase of the sine wave.

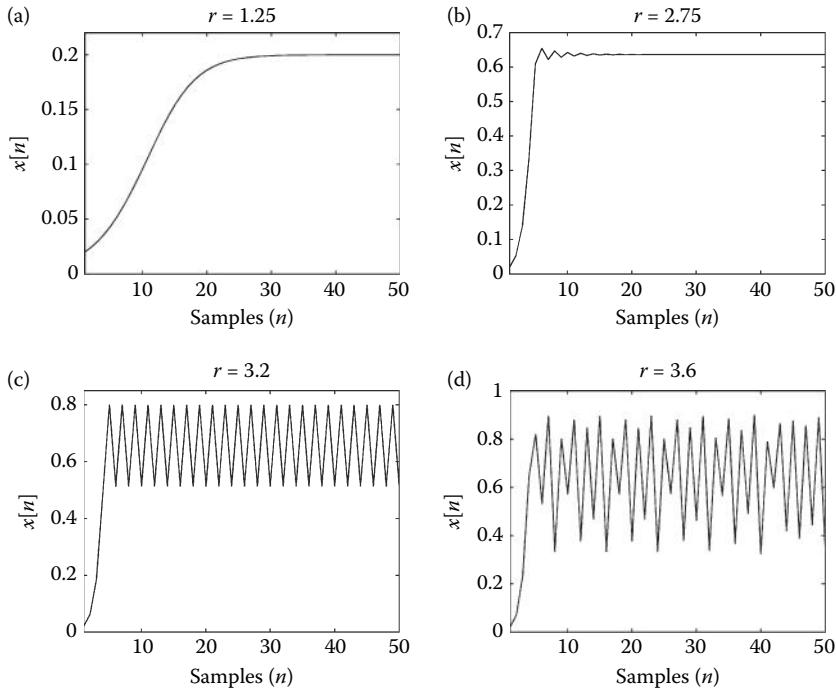


Figure 10.2 The response of a deterministic system that describes population growth in various species and is defined in Equation 10.4. As the value of the driving factor, r , increases from 1.5 (a) to 2.75 (b), the species reaches a higher steady population level in fewer generations. However, when r exceeds 3.0, the population oscillates between two levels (c). This behavior results from a small change in a system parameter and is known as a *bifurcation*. When r exceeds 3.57 (d), the population appears to vary randomly between generations although it is deterministic as specified by Equation 10.4. This is another example of a chaotic system.

In practice, even signals produced by deterministic systems are corrupted by some noise so that their future values may not be exactly predictable. In fact, all measurements in the real world, whether they are from linear or nonlinear systems, contain noise and this noise adds a random element to the measured signal. For either system, if the SNR is high, we do not consider the signal as stochastic, but if the SNR is very low, the signal is stochastic (and often of little value).

10.1.3 Types of Noise

Signals may contain different levels of noise and the noise itself can have different properties. As shown in Chapter 2, the autocorrelation function of a *completely* stochastic signal drops to near zero for all nonzero lags (see Figure 2.18d). It is as if each sample in this signal is randomly selected from a Gaussian distribution: there is no correlation between adjacent points. This decorrelation between adjacent samples implies that the values of subsequent samples cannot be determined by knowledge from prior samples, even by using an infinite number of previous samples. A truly random signal can only be described by its statistical properties such as mean and variance. The measurements of random signals are not of much interest since there is little we can say about them. However, there are situations in which the Gaussian noise can be a useful *input* to a system such as in bandwidth testing (since the Gaussian noise is broadband) and

in the control of chaotic systems (control of chaotic systems is a fascinating topic, but is outside the scope of this chapter).*

Not all noise is Gaussian. The so-called “ $1/f$ noise,” also known as *fractal noise*, is associated with nonlinearity. This type of noise is termed $1/f$ because of its spectral characteristics: on a log–log plot spectral power decreases proportionally as a function of frequency. As of yet, there is no mathematical model that can fully account for all the properties of $1/f$ noise. A system whose output shows $1/f$ behavior may be nonlinear due to the presence of a $1/f$ noise element, it may have a $1/f$ noise input, or it may have white noise that has been filtered to have a $1/f$ spectral characteristic.

Noise with $1/f$ properties is *self-similar* in the frequency domain. This means that the signal appears the same no matter what frequency scale is used to describe it. A self-similar signal requires infinite precision to be fully characterized because it has information (or at least variation) on any observable frequency scale; there is no lower limit to the resolution needed to completely define it. There is no way to measure a signal containing $1/f$ noise in a way that accounts for all the characteristics of the signal because infinite precision is required, at least in theory.

Another type of noise is integrated noise, which is the Gaussian noise that has been integrated, producing a random walk. This type of noise may produce signals similar to that of $1/f$ noise, specifically $1/f^2$. Integrated noise can be produced by a linear process, yet may give the appearance of $1/f$ signals. Thus, the appearance of $1/f$ noise is not a definitive test of nonlinearity. Since integrated noise is easy to produce, it is useful as a test signal of a linear nonstationary process and this type of noise is used in several examples in this and in the next chapter.

10.1.4 Chaotic Systems and Signals

The signal from a chaotic system appears to combine properties from both random and deterministic signals but, as shown in Example 10.1, chaotic processes are deterministic, not stochastic. There are three criteria used to define a chaotic system: (1) steady-state values are not fixed and do not repeat: they are aperiodic, (2) system behavior is very sensitive to the initial conditions, and (3) the system is not just the response to a random input. There are methods to test if an apparently random signal is actually chaotic, many based on the first two criteria. Unfortunately, these tests are often less than definitive, especially if noise is intermixed with the chaotic signal.

An example of a chaotic system is presented in Example 10.1, featuring a driven pendulum. Another popular example of a chaotic system comes from the study of the variability in species populations. An equation that predicts animal populations fairly well is a simple quadratic equation called the *logistic difference equation* or *logistic map*, since it maps population values at generation n to values at the next generation, $n + 1$. In other words, the equation relates the population $x[n + 1]$, to the previous generation’s population, $x[n]$. This type of function is also known as an *iterated map*. Iterated maps are discussed more fully in Section 10.2.1. Here, we only introduce the logistic map as an example of a simple system that demonstrates chaotic solutions. The logistic map is given as

$$x[n] = rx[n](1 - x[n]) \quad (10.4)$$

where $x[n]$ is the population at generation n (normalized to 1.0), $x[n + 1]$ is the population of the next generation, and r is the so-called *driving parameter* that is related to growth. For small values of r , < 3.0 , Equation 10.4 produces smooth monotonic changes from one generation to the next, which eventually converge to a stable value. Above 3.0, the population alternates between two levels and as r increases further, it alternates between 4, 8, and then 16 different population values: a behavior known as *period doubling*, which continues to double until r is about 3.57.

* A technical aside: Many of the examples in this chapter make use of MATLAB’s random number generator; so, when you run an example on your computer, you will not get exactly the same figures as in this chapter. However, the results should be reasonably similar.

Biosignal and Medical Image Processing

Above a value of 3.57, the generation-to-generation fluctuation becomes apparently random. Although this fluctuation appears to be random, it is not since it is still completely determined by Equation 10.4. This behavior is explored in the next example.

EXAMPLE 10.2

Plot the first 50 generations predicted by Equation 10.4 using four values of r : 1.25 2.75, 3.2, and 3.6. Start with an initial population, x_0 , of 0.02. (As shown in Problem 10.1, the initial value affects the speed of convergence, but not the final convergence value that is dependent only on r .)

Solution

Set the first value of a 50-point array to the initial population value of 0.02 and then implement the equation using a loop. Use an outer loop to run the simulation four times at different values of r .

```
% Example 10.2 Example of logistic equation (Eq. 10.4)
%
x(1) = 0.02; % Initial population value
r1=[1.25 2.75 3.2 3.6]; % Values of driving parameter
for k=1:length(r1)
    r=r1(k); % Set driving parameter
    for n=1:50
        x(n+1) = r*x(n)*(1-x(n)); % Logistic eq. (Eq. 10.4)
    end
    .....set subplot, plot x, label, and title.....
end
```

Results

As shown in Figure 10.2, when r equals 1.5 or 2.75, the population reaches a constant value, but for the larger value of r , the equilibrium is reached sooner and at a higher level. However, when r increases to 3.2, the population alternates between two values and when r is 3.6, the population value varies from generation to generation, seemingly randomly. In fact, this variation is not random since it is completely determined by Equation 10.4. The behavior of this equation over a wide range of r -values is shown in Figure 10.16a and is explored in Exercise 10.5.

Figure 10.3 compares the three types of signals described in this section: a stochastic signal, chaotic signal, and a deterministic biological signal. The bottom plot shows an example of a deterministic signal, a typical ECG signal. Although it is deterministic, studies have shown that the timing of this signal is nonlinear. The stochastic signal is constructed by randomly sampling a Gaussian distribution using MATLAB's `randn`. (Actually, this is a pseudorandom series and will repeat if the sequence is long enough.) The chaotic signal comes from the logistic map investigated in Example 10.2, using an r value of 3.9999. Note the similarity between the deterministic chaotic signal and the random signal. Neither appears to be predictable; yet, if we knew the initial conditions, system parameter (i.e., r), and generation (i.e., n), we could accurately predict future populations using Equation 10.4. Usually, these parameters and the governing equation(s) are not known; so, in practice, we are unable to predict the behavior of a chaotic system over a sustained time period.

10.2 Phase Space

The study of chaotic and other nonlinear signals often makes use of *phase space* to provide a descriptive analysis. Phase space represents all possible internal states of a system and is visualized

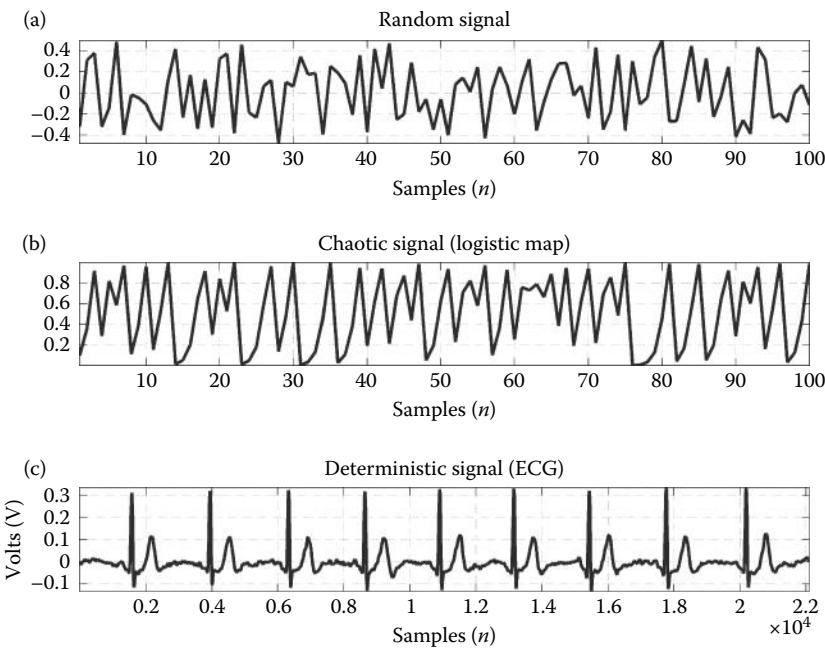


Figure 10.3 A random signal (a), deterministic chaotic signal (b), and a deterministic biological signal, the ECG (c). Both the chaotic and random signals appear to be composed of uncorrelated noise, but the chaotic signal is deterministic. The biological signal is an ECG that shows signs of determinism as the signal is regular and contains a repeated structure. Nonetheless, the timing of this signal, the so-called *R–R* interval, may be nonlinear.

as a plot of the system variables or *state variables* against one another. Since phase space plots the state variables, it is also known as the *state-space* plot or *vector-space* plot. The path that the variables trace throughout the phase space is known as the *phase trajectory* or just the *trajectory*. Each point on the trajectory completely describes the state of the system at a given instance in time. The trajectory can be embedded in any number of dimensional spaces, but to describe a system completely, the phase trajectory must occupy a phase space that includes at least as many dimensions as the free parameters in the system. If the system is continuous, then the trajectory will be continuous and smooth, whereas for a discrete function, the trajectories may or may not be continuous. In a second-order mechanical system, there are two free parameters and only two variables, position, and velocity are needed to completely describe the state of the system. So, the phase space consists of the state variables position and velocity and can be plotted in two dimensions. Traditionally, velocity is plotted on the vertical axis and positioned on the horizontal axis. Such a 2-D phase space is also known as the *phase plane*. If a system contains more than two variables, a phase plane can be constructed by projecting any pair of its variables in 2-D space. A phase space plot of the pendulum operating in the linear range is provided in the next example. As a second-order system, the pendulum has two state variables, angular position, and angular velocity; so, it can be completely described in a 2-D phase plane plot.

EXAMPLE 10.3

Solve the differential equations for the pendulum given in Equations 10.2 and 10.3 and plot the phase plane. Solve for two conditions: no damping, $b = 0$, and a damping factor of $b = 0.2$. Plot the state variables, angle, and angular velocity, as time functions and plot the phase plane for the two conditions.

Solution

This problem is the same as Example 10.1 except for parameter values and plotting. Use the same code and differential equation solver, `ode45`, and solve the equations twice for the two parameter values. To define Equations 10.2 and 10.3, the function `pend` from Example 10.1 can be used.

```
% Example 10.3 Generate the phase plane for damped and undamped pendulum.  
%  
% Set up time span and initial values for ode45  
theta_0 = [pi/100,0] % Initial θ is small, initial ω=0  
k = 0; % Driving force is 0  
b = 0.2 % Damping factor is 0.2  
%  
tspan = [0,100]; % Start and stop time  
%  
[t,sol] = ode45(@pend,tspan,theta_0,[],b,k); % Eq. solver  
theta = sol(:,1); % Get solutions  
omega = sol(:,2);  
plot(theta,omega); % Plot phase plane  
..... subplots, plot time plots, label.....  
%  
% Repeat for the undamped case.  
b = 0; % Damping is 0.0  
[t,y] = ode45(@pend,tspan,theta_0,[],b,k); % Eq. solver  
plot(theta,omega); % Plot phase plane  
..... subplots, plot time plots, label.....
```

Results

The results of this code are shown in Figures 10.4 and 10.5. For the undamped case, the phase plane plots as a circle, whereas for the damped case, the phase trajectory spirals in as both position and velocity decrease over time. Note that time is not explicitly shown on the phase plane plots and that a fixed position, such as the final resting position of the system (Figure 10.5) plots as a single point on the horizontal axis.

The steady-state behavior of a trajectory in phase space is determined by an *attractor*. An attractor is a region of phase space toward which a phase trajectory tends to move over time. Ordinary attractors are well-defined geometrical regions of phase space: points, lines, curves, planes, and manifolds. If the system reaches a constant steady-state value as in Figure 10.5, this attractor is just a fixed point in phase space. If, as in Figure 10.4, the steady state is an oscillation that does not decay, also known as a *limit cycle*, the attractor has the shape of a loop. If the solutions grow continuously, for example, exponential growth with the trajectory continuing to infinity, then the phase space contains a *repeller*, not an attractor. It is worth noting that we cannot actually see the attractor, only the *effect* of an attractor on a particular trajectory. However, following a trajectory for long periods of time reveals the nature of the attractor even if we cannot see the attractor itself.

Unlike linear systems, chaotic systems are aperiodic: typically, there is no stable value or sequence of values, even as time approaches infinity. Hence, phase trajectories of chaotic systems never converge to a point (as in Figure 10.5) or a fixed shape (as in Figure 10.4). Moreover, chaotic trajectories never overlap, but most of the trajectories do reside in a limited region of phase space, forming irregular shapes. The attractors for chaotic systems typically have a fractional dimension in phase space and are known as *strange attractors*, a phrase coined by Ruelle and Takens (1971). Strange attractors are fractal objects and display self-similarity as discussed in Section 10.5.

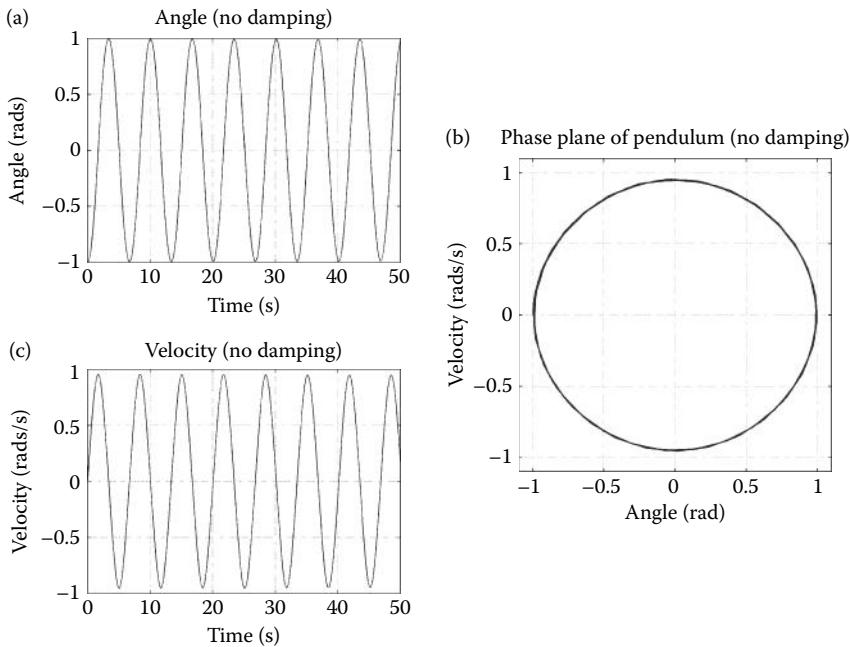


Figure 10.4 In the linear region, the undamped pendulum swings sinusoidally with position (a), velocity (c), and 90° out of phase; so, the phase plane (b) plots as a circle.

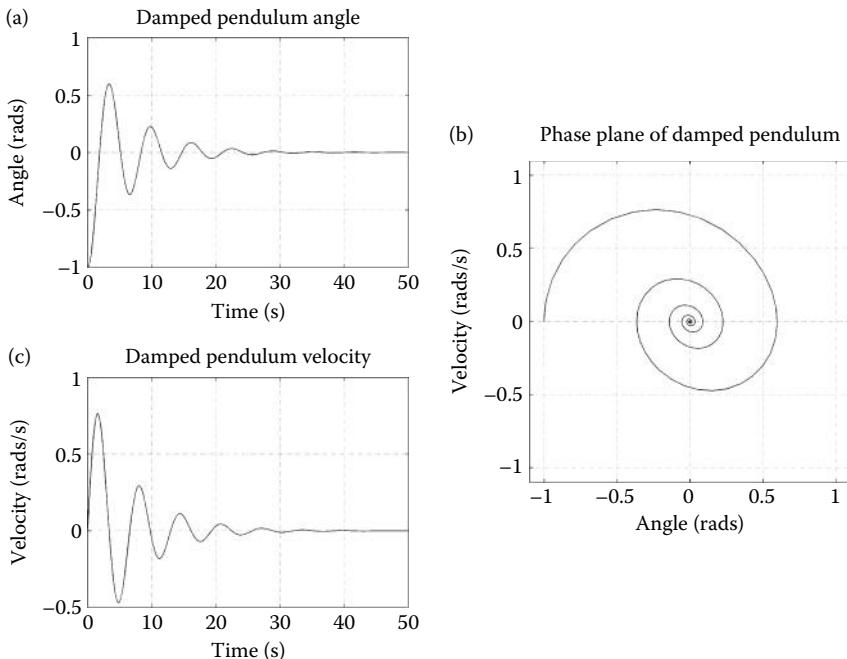


Figure 10.5 The damped pendulum swings with an angle amplitude that decreases over time. Both the angular displacement (a) and velocity (c) decrease in time, but are 90° out of phase leading to a decreasing spiral in phase space (b). The center point in the phase space is referred to as a stable node since the system is at steady state at this point.

Biosignal and Medical Image Processing

Many signal-processing methods in this chapter are concerned with assessing the properties of the attractor of the system that generates the signal. Typically, we are interested in either characterizing the system's dynamics, that is, the time behavior of the trajectories shaped by the attractor or determining the geometric properties of the attractor itself. A prime example of the former is a time-dependent analysis technique known as "Lyapunov exponent analysis" that is described in Section 10.4. A good example of the latter is a geometric method known as "correlation dimension analysis" that is presented in Section 10.5.

10.2.1 Iterated Maps

Here, we briefly describe *iterated maps*, since we use several iterated maps as examples in this chapter. We have already seen one iterated map (also known as a *mapping function*), the logistic map, defined in Equation 10.4 and analyzed in Example 10.2. Mapping functions relate the value of a system at a future time index to the current time index. A subset of mapping functions, termed *difference equations*, are discretized differential equations derived from continuous differential equations such as Equations 10.2 and 10.3. An iterated map is more general than difference equations because difference equations are derived from continuous equations and have limitations inherent to the continuous domain. Since iterated maps are not based on continuous equations, their trajectories can be quite complex, including discontinuous jumps in the phase space. These discontinuities necessitate an alternative method for displaying an iterated map trajectory that would be used for a continuous differential equation system such as the pendulum system. Typically, we plot the trajectory of a noncontinuous map using discrete points without connection lines. (This can be accomplished by using MATLAB's dot-plotting option, '.', when plotting). A discontinuous mapping function would generate a very confusing plot if the discontinuities were connected. As with continuous systems, a mapping function from both continuous and discontinuous systems can exist in one dimension or as a system of equations in several dimensions.

10.2.2 The Hénon Map

One of the most well-known mapping functions is the Hénon map (Hénon, 1976). This map was developed by Michel Hénon as a way to better understand what happens to a set of initial condition points in the phase space of a chaotic system. It has properties similar to the Lorenz system, a 3-D system that describes a simple convective flow that is more fully described in Section 10.2.3. The Hénon map was developed to simulate the stretching and folding behavior that occurs in the Lorenz attractor. Hénon wanted to write a set of equations that perform similar folding and stretching operations. Because of the stretching and folding mechanism, points in the phase space that are close together can, after only a few iterations, become far apart. This rapid divergence in phase space indicates *sensitivity to initial conditions*, an important hallmark of a chaotic system. The Hénon map is actually a simplified version of a mapping function that describes a 2-D slice through the 3-D trajectory of the Lorenz system. As the Lorenz system is chaotic, a trajectory made of a 2-D slice should also have chaotic properties. In fact, the Hénon map is chaotic for certain parameter values.

The mapping function for the Hénon map is given by equations of two variables, $x[n]$ and $y[n]$:

$$x[n + 1] = 1 + y[n] - ax[n]^2 \quad (10.5)$$

$$y[n + 1] = bx[n] \quad (10.6)$$

where a and b are constants. The Hénon map is chaotic for $a = 1.4$ and $b = 0.3$. For these parameter values, the y coordinates of points in the map become stretched into a parabolic shape, whereas the x coordinates are shrunk by a factor of b . This stretching and shrinking are shown

in Figure 10.6. Figure 10.6a shows the individual points of a large number of initial values for x and y . These initial values are chosen to form a rectangle so that we may more easily follow the changes in shape produced by subsequent iterations. Applying the mapping function defined by Equations 10.5 and 10.6 to each of these initial conditions causes the rectangular shape to change with each iteration, n . Figure 10.6b through d shows how each of the points is arranged after one, two, and three iterations as they are remapped by the mapping function. We can see how, only in a few iterations, the various trajectories alter the simple rectangular pattern. By choosing initial values in the shape of a rectangle, the folding and stretching becomes apparent.

Folding and stretching can be thought of in terms of applying the Hénon mapping function to a physical square made of a very pliable material and noting how the square sheet deforms with each iteration as the map is applied to every point on the sheet. As the trajectories undergo their first evolution (Figure 10.6b), we see that there is shrinking in the y direction and the formerly straight vertical edges of the box have become parabolic. This procedure rather quickly transforms the volume of the sheet into a shape similar to the attractor of the Hénon map. After only three iterations (Figure 10.6d), the sheet now has a shape very near to that of the attractor itself. If enough iterations are carried out, this figure would resemble the shape in Figure 10.7a. Our original square sheet of pliable material would be twisted into the shape of the Hénon attractor.

The property of shrinking a volume onto an attractor is known as dissipation: the map is said to be dissipative, another property of chaotic systems. Unlike for the Lorenz system, the Hénon map is not *bounded*. With the Lorenz attractor, any initial condition pair in phase space produces a trajectory mapped to the Lorenz attractor, but not all initial condition pairs map to

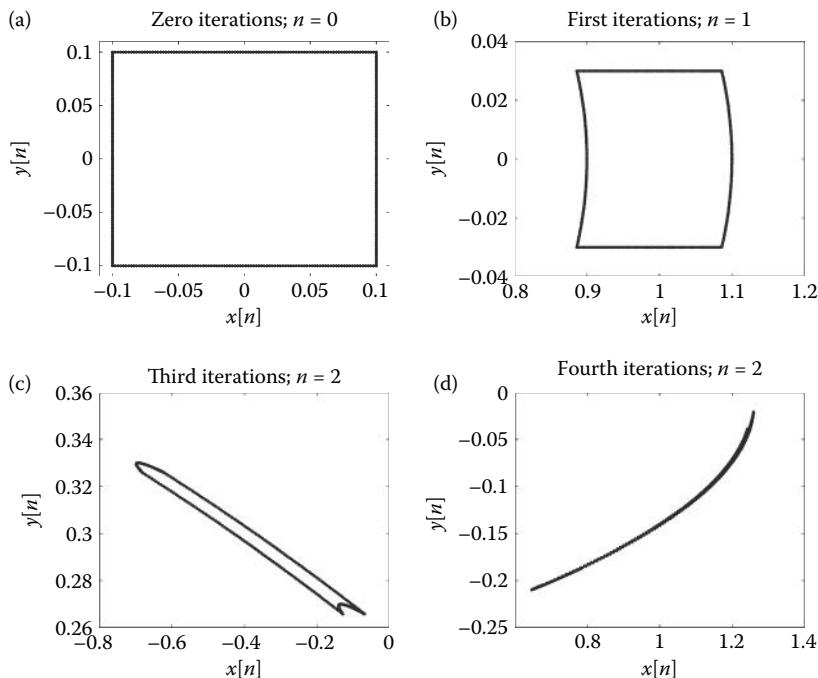


Figure 10.6 Initial values arranged to form a rectangular pattern that will be stretched and folded by the Hénon map. (a) A large number of initial conditions for x and y are arranged in a rectangular pattern before applying the map. (b) The rectangle is deformed after a single iteration of the Hénon map. (c) The deformation becomes severe and the area of the pattern has been sharply reduced after only two iterations. (d) After three iterations, the area is further reduced and the points now lie close to the attractor of the Hénon map.

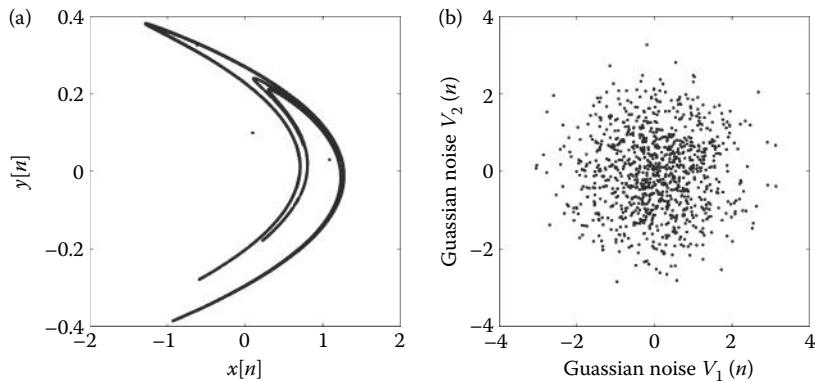


Figure 10.7 (a) The phase plane of the Hénon map with parameters that lead to a chaotic solution. The Hénon map takes on a characteristic bent-horseshoe shape. (b) The phase plane of the Gaussian noise demonstrates that decorrelated noise fills the phase plane evenly.

the Hénon map. However, a subset of initial conditions that are within a trapezoid surrounding the attractor will map to the attractor.

In Example 10.4, we calculate 10,000 values of the Hénon map and show the trajectories in a 2-D phase plane.

EXAMPLE 10.4

Plot the phase plane trajectory of the Hénon map defined by Equations 10.5 and 10.6. Also, plot the trajectory of random noise for comparison. Use the starting points of $x[1], y[1] = [0.5, 0.5]$ and to ensure a chaotic solution, use $a = 1.4$ and $b = 0.3$. Plot 10,000 individual points of the mapping function (i.e., not connected). For the plot of random noise, use only 1000 samples, otherwise the density of points in the phase space will be too great to visualize the characteristics of the plot.

Solution

The Hénon map is described in two dimensions, x and y ; so, we can plot the phase plane by simply plotting these variables against each other. In MATLAB, we determine values of x and y by iterating the equations given in Equations 10.5 and 10.6 10,000 times. To compare the results, we also plot vectors of $V_1[n]$ and $V_2[n]$ of Gaussian random noise as generated with `randn`.

```
% Example 10.4
N=10000; % Get 10000 values of the Henon map
x=0.1; y=0.1; % Set up initial conditions
a=1.4; b=0.3; % Set up constants
%
for k=1:N % Step through N iterations
    x(k+1) = 1-a*x(k).^2 + y(k); % Eq. 10.5
    y(k+1) = b*x(k); % Eq. 10.6
end
plot(x,y,'.k'); % Plot as points
%
N=1000; % Number of values for random vectors
V1=randn(1,N); % Create two vectors
V2=randn(1,N); % of Gaussian noise
.....plot as above, labels.....
```

Results

The results of this code are shown in Figure 10.7. The phase plane for the Hénon map is horse-shoe shaped whereas for random noise, there is no distinguishing characteristic shape to the phase space. This reflects the fact that there is no “order” to random noise: each sample is independent and uncorrelated with all other samples. The plot for the Gaussian distribution constitutes only an approximate phase plane. Since there is no dependence of one sample to the next, points in the phase space can appear anywhere in any dimension of a multidimensional space. The noise tends to fill the phase space in any dimension it is viewed in (assuming we use enough points). We justify using only a 2-D phase plane by noting that the vectors used (V_1 and V_2) are linearly independent.

The methods to quantify the features of the phase plane are discussed later in this chapter, but first, we need to address an important point. The systems used in Examples 10.3 and 10.4, the pendulum and the Hénon map, are 2-D. We know this because we know their state equations. In many practical situations, we measure only one signal and the underlying state equations are unknown. We usually have no way of knowing the number of variables in the measured system and even if we did, it is likely that the number of system variables exceeds the number of variables we can measure. This leads to a critical question: Can we recreate the multidimensional trajectory from a single signal? A related question is: Can we estimate the number of system variables, that is, the number of phase space dimensions required to accurately represent the underlying system from a single signal? These important questions are addressed in the next section.

10.2.3 Delay Space Embedding

To construct the phase space given only one signal, we need a way to recover the multidimensional information contained internally within the system using only our 1-D measured signal. Logically, this seems like it should be possible provided that the variables governing our system are coupled to each other: if the variable values from one dimension affect all other dimensions. In this case, all the internal variables make some contribution to our measured variable. Therefore, we need a way to create new vectors representing the other, higher dimensions from our single measured vector (i.e., the signal). Fortunately, a technique for doing just that exists, a technique known as *delayed space embedding*. In this method, delayed versions of the measured signal are used as substitutes for the hidden internal signals. The phase plane is then constructed by plotting the delayed vectors against the original.

The embedding equation is given as

$$y[n_d, k] = x[n + (k - 1)\tau, k] \cdots x[N - (m - 1)\tau, m] \quad (10.7)$$

where N is the length of the original signal, n_d is the time index of the delayed vector and has a range between 1 and $N-(m-1)$, n is the time index of the original time series, k is a dimension ranging between 1 and m , and $y[n, k]$ is the reconstructed multidimensional signals embedded in m dimensions. Hence, $y[n, k]$ is an $m \times N-(m-1)\tau$ matrix where the reconstructed signals are in rows. As seen in Equation 10.7, the reconstructed signals, $y[n_d, k]$, are composed of segments of the original 1-D data vector, $x[n]$, delayed by τ . For a single-dimensional mapping function such as the logistic map, this method is obvious since the independent variable will be $x[n]$ and the dependent variable will be $x[n + 1]$. However, it is not obvious that this approach works for higher-dimensional systems.

The included routine `delay_emb` is provided to perform the operation of delay embedding. The function has a format:

```
y = delay_emb(x, m, tau); % Delay embedding
```

Biosignal and Medical Image Processing

where x is the original 1-D time series, m is the embedding dimension of the new m -dimensional vector y , and τ is the delay (Equation 10.7). As long as enough samples have been taken, any number of dimensions can be reconstructed. This function is extensively used in the examples and methods described in the rest of this chapter.

Delay embedding gives us a signal with multiple dimensions, but how do we know if these new higher-dimensional signals actually reconstruct the trajectory correctly? The problem is that we typically have access to only a 1-D signal even though we suspect that the data come from a multidimensional system. If the 1-D measured time series does come from a multidimensional system, then it is likely that information from the additional variables is contained, but hidden, within the output signal. We would like to reconstruct the information from these hidden variables and produce an accurate representation of the system's phase trajectory. Accuracy in this context means that the reconstructed trajectory has the same properties as the "actual" trajectory, the one that we would find if we have access to all the internal system variables. The reconstructed trajectory need not appear identical to the real attractor, just having similar properties. If this is the case, we have reconstructed the trajectory correctly.

Takens (1981) showed that the important characteristics of a D -dimensional phase space *can be reconstructed* from 1-D information, where D is the number of actual dimensions in the system we are trying to analyze. Specifically, Takens' theorem imposes the criterion that the number of embedding vectors should be $2D + 1$, but it has been shown that fewer dimensions suffice for measured data; so, this criterion is more of a guideline. In fact, since the number of samples needed to accurately reconstruct a trajectory grows exponentially with dimension, it is preferable to use as few dimensions as possible. We shall see in Example 10.5 that using fewer than $2D + 1$ vectors is sufficient for reconstructing a well-known 3-D system. Unfortunately, there may be no way to determine D without *a priori* knowledge of the system being measured. We first discuss the methods of determining the delay τ followed by methods to determine an appropriate embedding dimension m .

The combination of the Takens' theorem and delay embedding gives us tools to estimate a multidimensional trajectory from a single time series, but we need to find the best value for delay, τ , as well as for the embedding dimension, m . Again, the idea is to choose values for both τ and m that produce signals that are faithful to the system attractor. These values for τ and m must be estimated from the data.

Takens showed that the value of τ is theoretically arbitrary, but in practice, τ has an optimum range. If τ is too short, then the embedding vectors are too similar to each other and will not be linearly independent. If τ is too long, delaying for many samples, there may be no correlation between the vectors. This is because biological signals typically have a relevant time scale. For example, consider an ECG signal sampled at 100 Hz where the heart rate is 60 beats/min. Each cardiac cycle would be represented by 100 samples. If we make the assumption that each cardiac cycle is a discrete event and independent of other cycles, then only samples that are contained within one heartbeat will be correlated.* Therefore, for this signal, values of $\tau > 100$ are inappropriate. The best value of τ also depends on the frequency content of the signal and the relative timescales of the events being probed. Since τ is specified in terms of the number of samples, it also depends on the sampling frequency. If the signal is heavily oversampled, sample values remain correlated over many samples and a larger value of τ is needed.

One method for estimating τ is to use the autocorrelation. The lag at which the autocorrelation reaches the first zero can be used as an estimate for τ . The mutual information function may be preferable to the autocorrelation because it is sensitive to nonlinear self-correlations, whereas the autocorrelation is a linear function that is only sensitive to linear correlations. (The mutual

* In Example 2.11, we find that there is some correlation in the beat-to-beat interval as determined by taking the autocovariance of the instantaneous heart rate. However, this example only examines one feature of the cardiac cycle, the heart rate. It is likely that many other features of the cardiac cycle are independent, even in neighboring beats.

information is a nonlinear analog of autocorrelation and is covered in Chapter 11.) Often, trial and error is used to determine values for τ , with the goal to optimize some desired characteristic of the resulting phase trajectory. The examples below and in Exercise 10.7 explore the influence of τ on the reconstructed trajectory.

Determining the best embedding parameters for a time series and reconstructing the attractor is an involved, multistep procedure. To demonstrate these steps one by one, we introduce a well-known chaotic system, the *Lorenz system*. This system has specific parameters that allow us to apply and evaluate the various methods for delay embedding in a way that is less subjective than for a measured signal, as such signals usually contain noise and uncertainty. After reconstructing and analyzing the reconstructed trajectory of the Lorenz system, we perform the same reconstruction and analysis on an ECG signal. We find that decisions become more subjective when applied to real signals.

10.2.4 The Lorenz Attractor

To further illustrate the properties of a chaotic system, we use perhaps the most famous chaotic system, the *Lorenz attractor*. The Lorenz attractor was named after Edward Lorenz, the MIT climatologist who discovered it. Lorenz was interested in the evolution of weather patterns and he developed a simplified weather system to study them better. He discovered that the solutions to his experimental system demonstrated what we now term “sensitivity to initial conditions.” The Lorenz system was the first chaotic system to be described; it was quite shocking at the time to learn that chaos could evolve from such a simple set of equations.

The trajectory produced by the Lorenz attractor can be visualized in MATLAB by entering `lorenz`. (This is an animated MATLAB demo and has no inputs or outputs.) While running the program `lorenz`, you will notice that the trajectories are constrained to the shape of a butterfly, but if you look closely, you will see that they never cross each other or repeat themselves exactly. This is a hallmark of a chaotic/strange attractor and indicates that the associated system generates an aperiodic signal: trajectories never contain repeated points and never reach the steady state.

Despite the complicated behavior, the Lorenz system, such as the logistic map, is deterministic and can be defined by a set of differential equations: Equations 10.8 through 10.10. These equations represent a simplification of the equations for the convective flow of a fluid, which Lorenz used to model the convective flow of warm and cold air currents:

$$\frac{dx}{dt} = \sigma(y - x) \quad (10.8)$$

$$\frac{dy}{dt} = x(\rho - z) - y \quad (10.9)$$

$$\frac{dz}{dt} = xy - z\beta \quad (10.10)$$

where x , y , and z are spatial dimensions (i.e., system variables) and σ , β , and ρ are constants. The three constants are σ , the Prandtl number, the ratio of the viscosity of the fluid to the thermal conductivity; β , the Rayleigh number, a dimensionless number that represents a ratio of buoyant forces to viscous forces within the flow of a fluid; and ρ , the density of the fluid.

The next example compares the attractor derived from the analytical solution of the Lorenz equation with an attractor constructed from a single signal using delay embedding (Equation 10.7). Each of the three state variables is used to reconstruct the attractor since, presumably, any of the three could be an output from the system. The results show differences in the resulting attractor depending on which variable is used as the output signal.

EXAMPLE 10.5

Solve the Lorenz equations, Equations 10.8 through 10.10, for x , y , and z using `ode45`. Plot the phase trajectory of the three variables as a 3-D plot. Use initial values for y_0 of [8, 9, 25] and a 1000-sample time vector `tspan` from 0.1 to 100. Then use delay embedding (Equation 10.7) to construct the embedded vectors and plot the phase trajectory based on these vectors. Use each of the three variables, x , y , or z , as the single measured signal for delay embedding. Compare the results with the actual phase trajectory obtained by numerical solution of the defining equations.

Solution

We use `ode45` to solve the equation for x , y , and z . After defining the initial conditions of these variables and the time frame, MATLAB's numerical integrator, `ode45`, is called with appropriate input arguments. The defining function, `lorenzeq`, is given in Appendix A and sets the constants to values that ensure a chaotic solution. The phase trajectory is developed by plotting the solutions for the three variables against each other as a 3-D plot using `plot3`.

Next, we reconstruct the phase trajectories using each of the three variables as the output signal. For each reconstruction, the `delay_emb` routine is used with $m = 3$ and $\tau = 6$. In this example, the selection of 3 for m is obvious since we know that the Lorenz system is 3-D. The selection of 6 for τ comes from trial and error.

```
% Example 10.5 Reconstruction of the phase trajectory.
%
y0 = [8,9,25]'; % Initial x, y, and z
tspan = linspace(.01,100,10000); % Solution time
[t,sol] = ode45(@lorenzeq,tspan,y0); % Solve Lorenz Eqs.
x = sol(:,1); % Get x,y and z
y = sol(:,2); % solutions from
z = sol(:,3); % output vector sol
figure;
subplot(2,2,1)
plot3(x,y,z); % Plot trajectory
..... title, labels, grid .....
%
m = 3 % Embedding dimension
for i = 1:3 % Solve using 3 variables
    tau = 6;
    y = delay_emb(sol(:,i),m,tau); % Delay embed x, y, or z
    subplot(2,2,i+1);
    plot3(y(:,1),y(:,2),y(:,3)); % Plot trajectory
.....titles and labels .....
end
```

Results

The results of this example are shown in Figure 10.8, which displays the Lorenz attractor as determined from a numerical solution of Equations 10.8 through 10.10 in the upper left plot and reconstructions using delay embedding in the other three plots. Each reconstruction is developed from only one of the three variables: x , y , or z . Comparing the reconstructions with the actual trajectory (Figure 10.8a), we see that either the x or y variables produce similar features to the numerical solution. These reconstructions are not exact, but we show that subsequent analysis of the reconstructed trajectory demonstrates similar properties to the original attractor (see Examples 10.7 and 10.12). When the z variable is taken as the output signal and used for reconstruction, the results do not show the full details of the attractor (Figure 10.8d). The z variable reconstruction appears to contain only one loop instead of the two intertwined loops that are

10.3 Estimating the Embedding Parameters

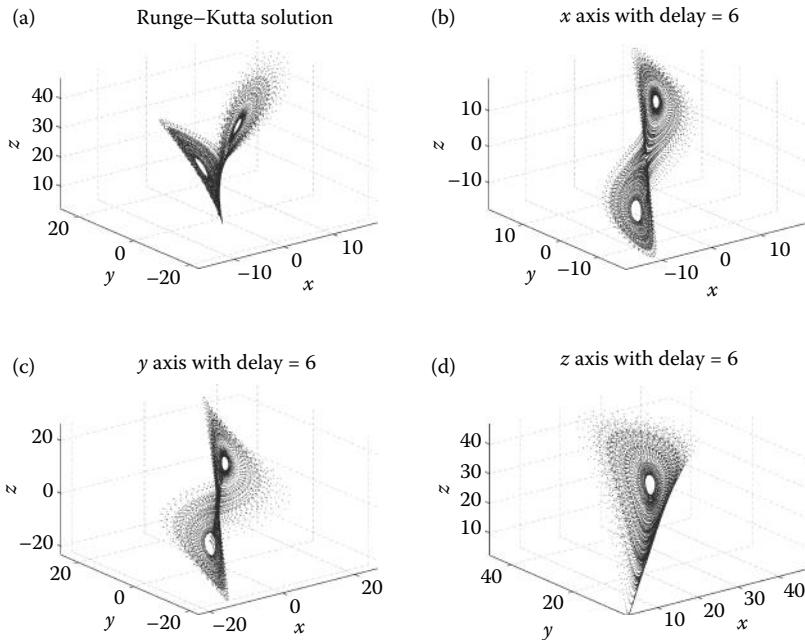


Figure 10.8 (a) The Lorenz attractor found using Runge–Kutta integration. (b) The phase trajectory reflecting the reconstructed trajectories found through delay embedding when the x variable is used as the sole signal. (c) The reconstructed trajectories found through delay embedding when the y variable is used as the sole signal. The reconstructions in (b) and (c) show trajectories qualitatively similar to those of the Lorenz system. (d) The reconstructed trajectories found through delay embedding when the z variable is used as the sole signal. The loops of the Lorenz attractor are folded over each other, leading to the appearance of only one loop. This reconstruction does not reflect the actual trajectory.

characteristic of this system’s trajectory. This is because the Lorenz attractor is symmetric about the z -axis. Therefore, the z -component reconstruction does not reflect the differences between positive and negative values of x and y . This causes the two loops that are characteristic of the Lorenz attractor to fold on top of each other giving the appearance of only one loop. This demonstrates the limitations of the delay embedding approach to reconstructing multidimensional trajectories from a single signal.

10.3 Estimating the Embedding Parameters

The Lorenz system is known to be 3-D, but typically, we cannot say *a priori* how many dimensions are needed when dealing with measured data. Several methods are covered in this section, including false nearest neighbors, SVD, and dimensional invariance. It is generally a good idea to use several methods to validate any results.

Dimensional invariance is the most conceptually simple method. Here, we examine the data for a particular property and find a range of dimensions for which it is invariant. The property should be invariant for any dimension greater than the minimum embedding dimension. The particular property could be any of the parameters discussed in the later sections of this chapter, including the ratio of false nearest neighbors, the Lyapunov exponent, and the correlation dimension.

Biosignal and Medical Image Processing

10.3.1 Estimation of the Embedding Dimension Using Nearest Neighbors

A neighbor is simply a point in phase space that is nearby to the point of interest. Here, we measure distances between points in phase space using the Euclidean distance function

$$d = \sqrt{(\vec{x}_a - \vec{x}_b)^2} \quad (10.11)$$

where d is the distance between two vectors \vec{x}_a and \vec{x}_b . This equation is also known as the *norm* operation indicated by the $\|$ symbol. Hence, Equation 10.11 can also be written as

$$d = \|\vec{x}_a - \vec{x}_b\| \quad (10.12)$$

Using this definition of distance, points are considered nearest neighbors if d is less than some threshold that we define. The idea of determining the nearest neighbors of a point in phase space is critical for several analysis methods discussed in this chapter. The distance equation, Equation 10.11, is also used in some of the classification methods presented in Chapter 16. In the next example, we use the Hénon map to demonstrate locating the nearest neighbors and how the neighbor distance threshold affects the determination of nearest neighbors.

EXAMPLE 10.6

Using the Hénon map, find all the nearest neighbors that are less than a specific threshold distance away from an arbitrary point on the trajectory. For this example, use the 50th point as a test point and threshold distances of 0.03 and 0.05.

Solution

First, we calculate the x and y trajectories from Equations 10.5 and 10.6 using the code in Example 10.4. Rather than using the two variables to generate the 2-D phase trajectory, we apply delay embedding to the y variable signal to generate the attractor. We then pick a point at random along the trajectory: in this case, $n = 50$. To implement Equation 10.11 using vector operations, we employ the handy MATLAB function `repmat`. This function duplicates a matrix. We use it to construct a matrix in which every row is our test point vector and there are as many rows as there are points in our trajectory. The calling format for `repmat`:

```
K = repmat(A,m,n); % Build matrix
```

where K is the new matrix that formed from A concatenated with itself m times along the rows and n times along the columns. This way, we use a matrix operation to very quickly determine the distance of the test point to all other test points in the trajectory based on Equation 10.11. This distance vector is searched for distances check that falls within two different thresholds: $r1$ and $r2$. The points associated with these distances are then displayed as explained below.

```
% Example 10.6 Determination of distances from a Henon map.  
%  
n = 50; % Arbitrary point  
r1 = .03; r2 = .05; % Distance thresholds  
  
%  
.....x and y Henon trajectories determined as in Example 10.4.....  
tau = 1; % Embedding delay (1)  
Hen = delay_emb(y,2,tau); % Take y vector as only output signal  
%  
testp = Hen(n,:); % Get 50th point on trajectory  
testp = repmat(testp,N+1-tau,1); % Repeat over rows
```

10.3 Estimating the Embedding Parameters

```

dist = sqrt(sum((testp - Hen).^2,2));           % Eq. 10.11
%
n1 = dist(dist < r1);                         % Find points where distance < r1
n1(n1 == 0) = [];                               % Remove self matches
n2 = dist(dist < r2);                         % Find points where distance < r2
n2(n2 == 0) = [];                               % Remove self matches
%
numnear1 = length(n1);                         % Number of nearest neighbors for r1
numnear2 = length(n2);                         % Number of nearest neighbors for r2
%
% Use plotting function 'viscircles' to indicate threshold distances
plot(Hen(:,1),Hen(:,2));                      % Plot nearest neighbors
hold on;
plot(Hen(n,1),Hen(p,2));                      % Plot test point and
viscircles([Hen(n,:)]);                        % thresholds radii
viscircles([Hen(n,:)]);

```

Analysis

When determining the nearest-neighbor plot in 2-D, we use circles to indicate the two threshold distances (Figure 10.10). These are plotted using the MATLAB plotting routine `viscircles`. The trajectory is sparsely populated for the purpose of this example. Typically, more samples would be needed to adequately visualize the attractor, but with a sparsely populated attractor, there are fewer neighbors for each point and the points are easier to see on the trajectory.

Results

Figure 10.9 shows the nearest neighbors of the 50th point in the Hénon map for the distance thresholds used in Example 10.6. There are six neighbors that fall within a distance threshold of 0.05 and two neighbors fall for a threshold of 0.03.

Now that we have shown how to find the nearest neighbors to any given point on a trajectory, we can investigate how the nearest neighbors change with the embedding dimension. The changes in the number of nearest neighbors help to determine an embedding dimension. The idea is that points that are close neighbors in an insufficiently dimensioned embedding space are *no longer* neighbors if the embedding space dimension is adequate. Such neighbors

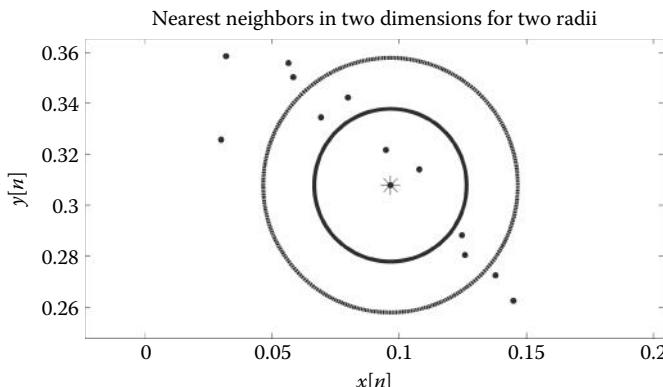


Figure 10.9 The nearest neighbors to a random point (starred point) on a trajectory reconstructed from the y output of the Hénon map. These results show six neighbors within a distance threshold of 0.05 and two neighbors within a threshold of 0.03.

Biosignal and Medical Image Processing

are considered *false nearest neighbors* because true neighbors would remain close no matter how large the embedding dimension. By examining the number of false nearest neighbors of points as a function of increasing the embedding dimension, an estimate of the proper embedding dimension can be made.

To compute the plot of average false nearest neighbors as a function of increasing the embedding dimension, the routine `fnumnear` is written based on the code in Example 10.6. The routine implements an algorithm described by Kantz and Schreiber (2004). Unlike in Example 10.6, here, we test *all* points on the trajectory for nearest neighbors. This way, a very large number of points are used; so, any variations in point densities average out. This routine takes an input signal and embeds it in both m and $m + 1$ dimensions. For each point along the trajectory, the routine finds the nearest neighbors based on a user-specified distance threshold, then determines the number of false nearest neighbors by comparing the numbers at m and $m + 1$ dimension. This continues up to some maximum dimension specified by the user. The routine outputs the average fraction of neighbors in dimension m that are determined to be false. The calling syntax is

```
Numnear = fnumnear(x, tau, em, r);
```

where x is the data to be analyzed, τ is the delay, em is the maximum number of dimensions to test, and r is the distance that defines the nearest neighbors. In practice, the value of r is not critical, provided that it is large enough for at least several points in the phase space to be the nearest neighbors. A reasonable choice is 0.1 times the standard deviation of the signal. Kantz and Schreiber (2004) suggest testing multiple values of r and τ . They also suggest a repeat of the analysis using surrogate data. The generation and application of surrogate data is covered in Section 10.6.

The next example uses routine `fnumnear` to estimate the number of embedding dimensions required for the Lorenz system.

EXAMPLE 10.7

Use the method of false nearest neighbors to determine an appropriate embedding dimension for the Lorenz system. Use a maximum embedding dimension of 6, an embedding delay of 6, and a distance threshold of 0.1 times the standard deviation of the signal.

Solution

First, we need to generate our data set from the Lorenz equations, Equations 10.8 through 10.10. We can use the same code in Example 10.5, but since we need to evaluate more dimensions than actually required for embedding, we generate a larger data set: $N = 100,000$. The Lorenz equation and `ode45` parameters are set as in Example 10.5. Then we use `fnumnear` to determine the average number of false nearest neighbors for all points on the phase trajectory for embedding dimensions 1 through 6.

```
% Example 10.7
% Estimating embedding dimension using nearest neighbors.
.....Solve Lorenz equation as in Example 10.5.....
x=sol(:,1); % Get x solutions from solution matrix
m=6; % Largest embedding dimension
tau=6; % Delay
r=.1*std(x); % Threshold radius
numnear=fnumnear(x,tau,m,r); % Find nearest neighbors
.....plot numnear, labels.....
```

Results

The results of this code produce vector `numnear`, which is a vector of the mean % false nearest neighbors for each tested dimension. A plot of this vector as a function of the embedding dimension (Figure 10.10) shows that there is a large drop after an embedding dimension of 1 and a small drop after an embedding dimension of 2. After three embedding dimensions, no further drop is seen. This makes sense given that we know that three embedding dimensions are appropriate for the Lorenz system.

The reconstructed trajectory of the Lorenz system can be shown to have a dimensionality a little greater than 2 using a technique that we explain later. The meaning of fractional dimensionality is discussed in Section 10.5.1, but this “slightly greater than 2” dimensionality is suggested in the 3-D phase space plot of the Lorenz system. The trajectories primarily appear as 2-D loops with a slight bend in the middle (Figure 10.8a). This slight bend effectively increases the dimensionality of the trajectory past 2-D toward a third dimension. Therefore, the largest drop in the average number of false nearest neighbors occurs when comparing the first and second embedded dimensions whereas a much smaller drop occurs when comparing the second and third embedded dimensions (Figure 10.10). Note that this behavior is exactly what we would predict about the Lorenz system, but it is revealed using only a 1-D output from the Lorenz system.

The Lorenz system provides useful test signals for nonlinear analysis methods because of its dimensionality and chaotic properties, but performing nonlinear analysis on a biological system, which may contain noise and other artifacts, is not straightforward. In the next example, we demonstrate this by repeating the nearest-neighbor analysis using a 10-s ECG segment as our test signal (Figure 10.11a).

The first step is to downsample the signal to a manageable size, if needed (depending on the sampling frequency of the original signal). This is an important step because if you inspect the nearest-neighbor algorithm and many of the other algorithms presented in this chapter, you will

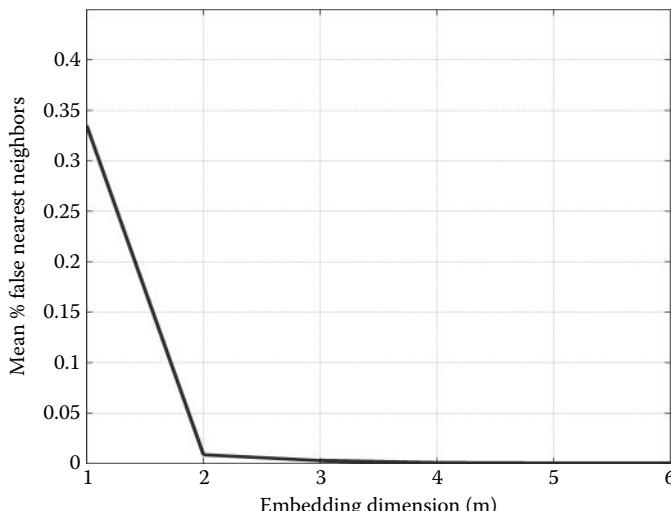


Figure 10.10 Mean % false nearest neighbors for the Lorenz system as determined in Example 10.7. The small difference in false nearest neighbors when comparing the second and third embedding dimension as compared to the first and second dimensions is related to the fact that the Lorenz system has a dimensionality slightly greater than 2. The concept of an attractor having fractional dimensions is presented in Section 10.5.1.

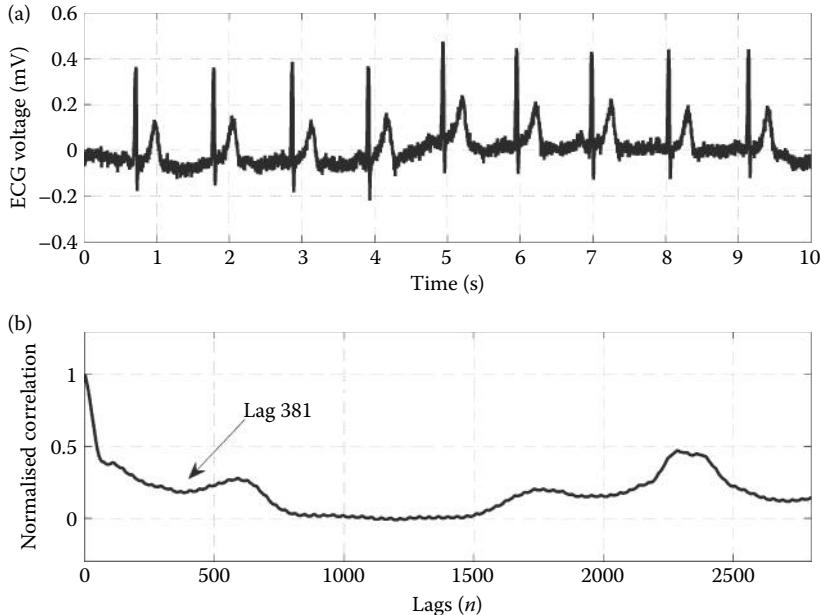


Figure 10.11 (a) The ECG signal evaluated in Example 10.8. (b) The first local minimum of the autocorrelation of the ECG signal is used to motivate an embedding delay of 381 samples. Different values of the embedding delay emphasize different scales of the ECG waveform.

see that they all operate sample by sample in a loop. Some components of this process are vectorized to take advantage of MATLAB's efficient linear algebra routines, but for the most part, these methods are highly demanding computationally. Reducing the number of data points is necessary to ensure that the computations run in a reasonable amount of time. Keep in mind, however, that there are limits to how small a data set we can use. The lower boundary is traditionally about 1 or 2×10^m , where m is the embedding dimension. Since the nearest-neighbor method requires testing in many embedding dimensions, a rather large number of points are needed. In Example 10.8, we use approximately 250,000 samples that give us enough samples to test up to six dimensions.

EXAMPLE 10.8

Plot the phase trajectory derived for the ECG signal shown in Figure 10.11a. Use a signal length of 250,000 samples ($f_s = 2210$ Hz). Use the autocorrelation function to determine an appropriate delay. Use the nearest-neighbor analysis to select the embedding dimension.

Solution, Embedding Delay

Since this example has several steps, we tackle it in pieces. These steps are typical for the analysis of a signal believed to contain nonlinearity.

The autocorrelation function for the ECG signal is determined using `xcorr` with the '`coeff`' option; so, the function is normalized to 1.0 and zero lag (see Section 2.3.3). This function is then plotted against lag.

```
% Example 10.8 nearest neighbor evaluation of embedding dimension
% using a real signal.
%
```

```

x = load('ECGtest.csv')          % Load the ECG signal
fs = 2210;                      % Sampling frequency
[x_xcorr,lags]= xcorr(x,'coeff'); % Get autocorrelation

```

Results, Embedding Delay

The autocorrelation function of the ECG is shown in Figure 10.11b. Note that there are several minima and maxima. Since the autocorrelation curve does not have a zero crossing, we choose the lag where it reaches its first local minima. This occurs at approximately 381 samples.

Solution, Embedding Delay

In the next part, this example uses the nearest-neighbor approach to determine an embedding dimension. The radius r is found by trial and error. If the value of r is too large, then the ratio of nearest neighbors in increasing dimensions is always 1 or nearly 1, as the whole trajectory is counted as the nearest neighbors. If the value of r is too small, there are no nearest neighbors and the ratio is mathematically undefined because of a divide-by-zero error. Therefore, values of r were chosen such that the ratio shows a monotonic decrease with increasing dimension. Three values of r are used.

```

% Evaluate false nearest neighbors,
r = [1.25,1.5,1.75];           % Distance thresholds evaluated
for kk = 1:3;                  % Nearest neighbor ratio cutoff
    em = 7;                     % Max tested embedding dimension
    tau = 381;                  % Embedding Delay
    [numnear(1:7,kk)] = fnumnear2(x,tau, em,r(kk));   % False nearest
                                                % neighbor test
end
.....plot false nearest neighbors.....

```

Results, False Nearest-Neighbor Analysis

Figure 10.12 shows the average fraction of false nearest neighbors in the m dimension compared to the $m + 1$ dimension, as a function of m . Figure 10.12 is constructed using the code above. Unfortunately, this is not particularly informative as there is no clear break in the plot of mean fraction of false nearest neighbors versus dimension. This is likely due to noise.* For this signal, we should try an alternative method to estimate the appropriate embedding dimension. However, for the sake of this example, we plot the reconstructed ECG attractor using three dimensions, as that is the highest number of dimensions that can be easily visualized. We will show in the next part of the example, using PCA to determine the number of dimensions, that three dimensions are in fact a reasonable embedding.

Solution, Phase Trajectory

Once the appropriate values of the embedding dimension, m , and delay, τ , have been determined (or guessed at), this phase trajectory is determined using Equation 10.7 and plotted. In this case, a 3-D plot captures the important parts of the trajectory.

```

x=delay_emb(p,3,381);           % Delay space embedding
plot3(x(:,1),x(:,2),x(:,3));    % Plot the trajectory in 3D

```

* Filtering would help this signal and this is explored in a problem at the end of this chapter. In general, the effect of noise and filtering on nonlinear signals is not fully understood; so, it is best avoided when possible. For a filtering scheme specifically designed for nonlinear signals, see Kantz and Schreiber (2004).

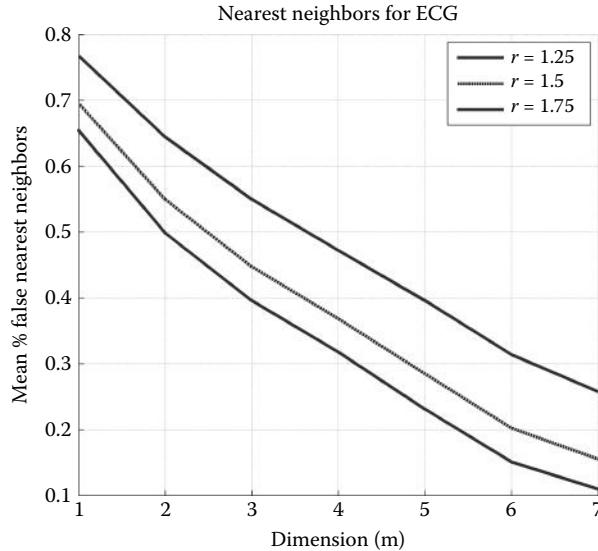


Figure 10.12 Mean % false nearest neighbors as a function of dimension. Unfortunately, the number of false nearest neighbors does not show a definitive embedding value for any of the three distance thresholds.

Results, Phase Trajectory

The reconstructed ECG attractor is shown in Figure 10.13. A delay, τ , of 381 samples and an embedding dimension, m , of three samples was used. These parameters result in a shape that contains many points clustered about the origin with three arms projected into the x_1 , x_2 , and x_3 axes. The main portion of the trajectory lies in one dimension along the x_3 axis. The two arms in the x_1 - x_2 plane are due to the slower P and T waves of the ECG. Unlike for an artificial signal such as the logistic map, Figure 10.13 shows that the reconstructed trajectory for the ECG contains some noise and uncertainty typical of a biological signal. Therefore, the optimal values for embedding can only be estimated within an appropriate range.

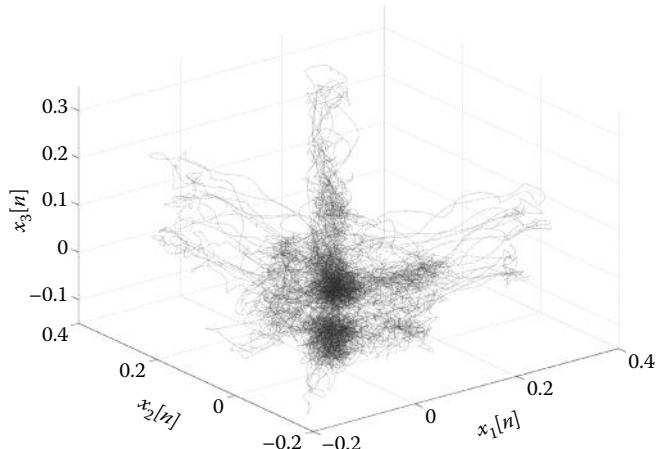


Figure 10.13 A reconstruction of the attractor of the ECG signal. The arms extending from the center in the x_1 and x_2 direction are due to the slower P and T wave in the ECG.

Analysis

Using a delay based on the autocorrelation function requires some judgment. Choosing lags corresponding to different minima and maxima emphasizes different properties of the ECG signal. For this reason, visual inspection of the signal is a crucial element for nonlinear analysis. The methods described in this chapter for determining the embedding parameters are not absolute and may not always give the best answer. One should qualitatively evaluate the signal to verify that the parameters suggested by parameter estimation techniques make sense. For example, in the ECG signal, we know *a priori* that peaks in the autocorrelation function correspond to different events in the ECG signal such as the *P*-wave or the QRS complex. The large peak in the autocorrelation at about 2300 lags corresponds to the QRS complex (since the sampling frequency is about 2100 Hz and the heart rate in this example is about 56 beats/min; one cardiac cycle covers 2300 samples). Similarly, the peak at about 600 lags corresponds to the QT (QT refers to the interval of an ECG cycle that occurs between the 2nd peak and the third peak of the cycle) interval of approximately 0.27 s. If there is known information concerning the important timescales of a signal, it should be used as a starting point when testing for embedding parameters. Likewise, *a priori* knowledge regarding the number of dimensions of a system can be used as a guideline for testing.

Ultimately, the best test to determine if an embedding dimension, m , is appropriate is to evaluate different dimensions using multiple testing procedures including false nearest-neighbor analysis and SVD (Section 10.3.2); adequate surrogate data testing; leveraging of *a priori* knowledge; and testing for invariance of parameters such as the maximal Lyapunov exponent (Section 10.4). Determining the best value of embedding delay, τ , follows a similar line of logic, but a reasonable rule of thumb for ensuring that the delay is not too large or too small is to observe the embedding in two dimensions: if the delay is too small, the trajectory will resemble a straight line whereas if it is too high, the embedded trajectory will look like that of uncorrelated noise.

10.3.2 Embedding Dimension: SVD

Another method for determining the best value for the embedding dimension, m , is to use SVD for PCA as described in Chapter 9. In Chapter 9, SVD is used to estimate the number of independent signals contained in an ensemble of related signals. (ICA is then applied to actually find these components.) SVD can also be used to estimate the most significant contributors of a delay-embedded time series constructed from a signal. The number of significant contributors can be used as an estimate for the dimensional values. This approach is demonstrated in the following example:

EXAMPLE 10.9

Load the file `ECGtest.csv` that contains an ECG signal into variable `x` ($f_s = 2100$ Hz). After embedding this signal into six dimensions, use SVD to find the appropriate number of dimensions.

Solution

After loading the data, we embed this signal in six dimensions using `delay_emb`. We apply SVD to these embedded signals and determine the eigenvalues as in Chapter 9. We then plot the six eigenvalues as a function of component number (i.e., the “Scree” plot, see Section 9.2.2) and search for a break point.

```
Example 10.9 Determination of embedding dimension using PCA
%
x = load('ECGtest.csv'); % Load the ECG file.
fs = 2100; % Sampling frequency
m = 6; % Embedding dimension
%
```

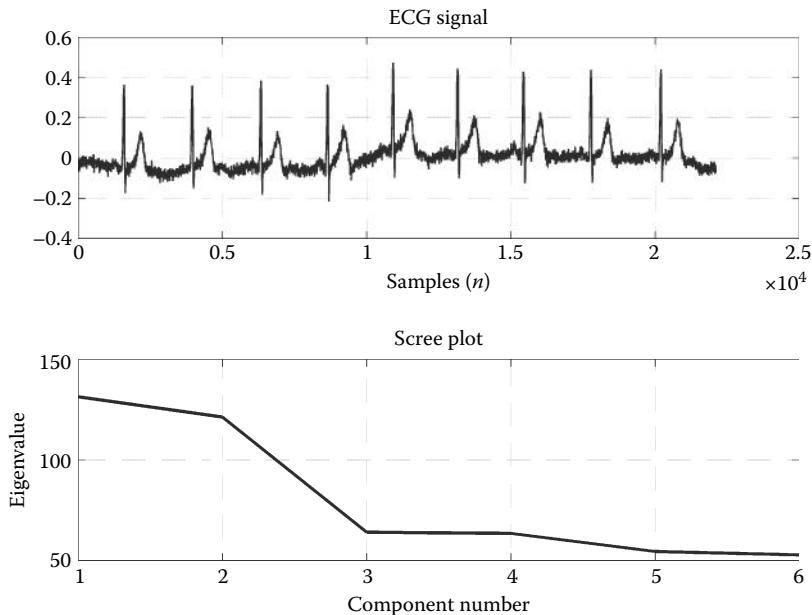


Figure 10.14 (Upper plot) ECG signal used in Example 10.9. (Lower plot) The Scree plot for the ECG as determined in Example 10.9. The dominant break point of the curve appears to be at the third eigenvalue, indicating that 3 is a reasonable embedding dimension.

```
X = delay_emb(x,m,fs); % Generate delayed signals
[U,S,pc] = svd(X', 'econ'); % Perform the decomposition
Eigen = diag(S).^.5; % Get the eigenvalues
.....label and plot.....
```

Results

The PCA analysis applied to a section of the ECG signal produces the Scree plot shown in Figure 10.14 (lower plot). The Scree plot shows a substantial break point at 3, suggesting that 3 is an appropriate embedding dimension. Note that we are not using SVD to reduce the dimension of multivariate data as in Chapter 9. PCA is a linear technique and if applied to nonlinear data might destroy the nonlinear features in the signal we wish to analyze. Rather, we are using only SVD to estimate the best embedding dimension.

10.4 Quantifying Trajectories in Phase Space: The Lyapunov Exponent

The properties of the attractor of a system can be used to infer the properties of the system itself. The first step in analyzing the attractor using time series data is to reconstruct an estimate of the trajectory in phase space. The estimated embedding dimension and delay themselves could be viewed as a first-order description of system properties. Now, we introduce more sophisticated approaches to quantify the properties of the attractor. The first parameter to be presented is the *Lyapunov exponent* that characterizes how trajectories along the attractor change in time. We see that the Lyapunov exponent is actually a fundamental characteristic of the time series and its attractor, particularly for a chaotic time series.

At its most basic level, the Lyapunov exponent describes the time interval over which the system's trajectory remains predictable. Linear deterministic systems are predictable over long

10.4 Quantifying Trajectories in Phase Space

periods of time because their trajectories can only take the form of a limited number of stereotyped behaviors. Chaotic deterministic systems have trajectories that exhibit quite complicated behavior that is predictable only over short periods of time. By characterizing the time frame of predictability, Lyapunov exponent analysis gives us an important insight into the characteristics of the system.

Lyapunov exponent analysis works by examining how multiple trajectories with similar starting conditions evolve along the attractor with time. For a chaotic system, trajectories that begin with similar initial conditions exponentially diverge over time, whereas for a linear system, they remain close or may converge. We define divergence as an increase in distance between trajectories that start from similar conditions: the trajectories that diverge become increasingly decorrelated. This divergence happens because chaotic systems magnify small changes in a trajectory that grow over time. Because of this property, if we compare the two trajectories of a chaotic system that start at similar initial conditions, their time histories quickly diverge; in fact, they diverge exponentially. The Lyapunov exponent is a measurement of this divergence.

We have been referring to the Lyapunov exponent in the singular and for our purposes, this is appropriate, but this does not tell the whole story. A chaotic system does not have a single Lyapunov exponent. This is because the exponential divergence may not occur at the same rate in all phase space dimensions. A series of Lyapunov exponents across all the dimensions of phase space is known as the *Lyapunov spectrum*. However, we are typically interested in the *maximum* Lyapunov exponent or sometimes the sums of the Lyapunov exponents, as these give the most information concerning the tendency of a system's trajectories to exponentially diverge. A chaotic system is highly likely to diverge in a manner described by the largest Lyapunov exponent, but why is this true?

To answer this question, recall that for a chaotic attractor, there is no steady-state value. Rather, trajectories continue forever in a bounded geometric region in which the attractor lies. This implies, and has subsequently been proven, that trajectories on a chaotic attractor visit all the regions of the attractor if enough time has passed. Some regions of the chaotic attractor may actually have properties of convergence or a limit cycle in the short term, but over the long term, the trajectories eventually reach a region of the attractor where a large Lyapunov exponent dominates. This is true even if some of the Lyapunov exponents within the Lyapunov spectrum of the system are negative or zero. Therefore, if you measure a chaotic system over a long enough time period, over many points along the attractor, and examine the divergence behavior, the overall divergence (and thus the long-term behavior of the system) is best described by the largest Lyapunov exponent. Throughout the rest of this chapter, when we refer to the Lyapunov exponent or its symbol λ , we are referring not to the Lyapunov spectrum, but the single largest Lyapunov exponent.

To bring mathematical formality to our description of the Lyapunov exponent, we first define two time series, $x[n]$ and $y[n]$, which originate from the same system and have similar initial conditions. We can define a distance vector: $\text{dist}[n] = \|x[n] - y[n]\|$. (This is the same as Equations 10.11 and 10.12 for a 1-D signal.) The Lyapunov exponent, λ , can be shown to be

$$\lambda = \frac{1}{n} \log \frac{\text{dist}[n]}{\text{dist}[0]} \quad (10.13)$$

where n is the sample number and $\text{dist}[0]$ is the distance between the initial sample points on the two trajectories. In Equation 10.13, we see that if the divergence is exponential, then the Lyapunov exponent will be some positive constant; if there is exponential convergence, then the Lyapunov exponent will be a negative constant. Note that in theory, λ is a constant and should not vary with n ; however, estimates of λ can vary with n due to variations in divergence for different dimensions, saturation when n reaches the size of the trajectory, and noise. If there is no divergence or convergence (the trajectories reach a steady state), then the Lyapunov exponent is zero. Nonexponential divergence or convergence is discussed later in this section.

Biosignal and Medical Image Processing

In practice, it is often easier to implement Equation 10.13 in an equivalent form given by Equation 10.14, which is the approach used here. If the slope of $\log(\text{dist}[n])$ with respect to n has a linear region, we can use this slope as an estimate of the Lyapunov exponent, λ . Using the slope to estimate the exponent:

$$\lambda = \frac{d(\log(\text{dist}[n]))}{dn} \quad (10.14)$$

where n ranges from the start to the end of the *linear region* of the derivative. Note that as written, this is appropriate for iterated functions in which the independent variable is n . If the signal of interest is a time series, one takes the derivative with respect to the time-sampled vector $t[n]$ rather than just n .

First, we must determine if the plot of the natural log of the divergence is linear. If this plot does not have a linear region, then the divergence is not exponential and the system is not chaotic, or perhaps, the signal is just too noisy to make that determination. If the plot of natural log divergence appears linear, then the second determination is the sign and magnitude of the slope of the linear region. The linear slope becomes our estimate of the Lyapunov exponent, λ . If λ is positive, then the system has exponential divergence and is chaotic. If λ is negative, there is exponential convergence; if the slope is zero, then there is no divergence or convergence and the system is stable.

Since it is vital that we determine whether or not Equation 10.14 has a linear region, we would like a method to make this determination objectively. We can use the R^2 error or some other measurement of goodness of fit to decide if the curve has a linear region. If there is a good fit to a straight line, then we know that the divergence is exponential (positive or negative). If there is a poor fit, the divergence is not exponential.

10.4.1 Goodness of Fit of a Linear Curve

The estimation of the Lyapunov exponent requires an estimate of the linear fit to a constructed curve. In addition, another nonlinear analysis method, the correlation dimension introduced in Section 10.5, also requires this assessment. The same curve fitting and error analysis applies to both methods. In this section, we review the concept of the R^2 error for a linear fit and we use that as an estimate of the goodness of fit.

To analyze the linearity of a function such as λ , we measure the goodness of fit between the function, a segment of the function, and the best linear fit. To quantify goodness of fit, we use the R^2 error, also known as the coefficient of determination. The R^2 error is a ratio of the total variation present due to the independent variable, with respect to the total variation of the dependent variable that is the measured signal. (In other words, it is the ratio of the variation due to error in time sampling versus measurement error due to noise.) If the ratio is 1, then for a linear fit, all the variance of our data can be explained by the fit. The independent variable is typically a series of time samples, although for an iterated map, this is simply n . R^2 is computed as 1 minus the ratio of the variation due to the fit and the total variation. The R^2 is also equivalent to the square of the normalized correlation coefficient between the fit and the measured data. The R^2 values are normalized to range from 0 to 1. One indicates that the linear fit is a perfect model for the data and 0 indicates that the linear fit is a poor model. In the general case, the R^2 values are determined for a series of samples, $y[n]$, as

$$R^2 = 1 - \frac{\sum_n (y[n] - \bar{y})^2}{\sum_n (y[n] - \bar{y})^2} \quad (10.15)$$

where \bar{y} is the mean of $y[n]$ (the measured values) and $\text{fit}[n]$ are the fitted values, that is, the values of a straight line that gives the best linear fit. In the case of the Lyapunov exponent, $y[n]$ is the $\log(\text{dist}[n])$ term in Equation 10.14 and is computed as

$$R^2 = 1 - \frac{\sum_n (\log(dist[n]) - fit[n])^2}{\sum_n (\log(dist[n]) - \overline{\log(dist[n])})^2} \quad (10.16)$$

This equation is implemented with the routine `rsquared*` whose calling structure is

```
rsquare = rsquared(log_dist, fit);
```

where `log_dist` is the log of the distance measurements of Equation 10.14 and `fit` is the time-series fit. Note that, here, we have used variable `log_dist`, to make clear what time series we are using to compute λ ; however, the R^2 coefficient is used in more general cases as well.

```
function rsquare = rsquared(log_dist, fit)
%
a = sum((log_dist-fit).^2); % Numerator
b = sum((log_dist-mean(log_dist)).^2); % Denominator
rsquare = 1-a/b; % Eq 10.16
```

For both the correlation dimension and the Lyapunov exponent, it is vital to show that some part of the curve of interest is linear. For the Lyapunov exponent, the curve of interest is $\log(\text{dist}[n])$. If the R^2 value for a linear fit in some region is close to 1, then we have objectively demonstrated that this is the case.

It is important to note that R^2 values are reduced by noise. With noisy data, it is possible that a linear fit would be appropriate even if R^2 values are relatively low. In such cases, the only course of action is to try to reduce the amount of noise as much as possible.

10.4.2 Methods of Determining the Lyapunov Exponent

This section describes a practical method to determine λ for a time series. We can use that information along with Equations 10.13 and 10.14 to estimate λ for a time series. We perform this estimation for the logistic map in Example 10.10.

EXAMPLE 10.10

Use Equation 10.14 to estimate the Lyapunov exponent for the logistic map using a driving force, r , of 4 to put the system into the chaotic range. Create the two trajectories, $x[n]$ and $y[n]$, using the logistic equation, but beginning with two slightly different initial conditions: 0.1 and 0.1001. (To facilitate the code, put both trajectories in a matrix variable `xmat` with the two trajectories in rows.) Then plot the natural log of $\text{dist}[n]$ and find the slope of a linearly increasing region to estimate λ .

Solution

First, we determine the divergence function by calculating the first 25 values of the logistic map. Values of the map are computed twice, once using 0.1 as the initial value and once using 0.1001 as an initial value. The divergence is found by comparing how these trajectories evolve from their respective starting positions using $\text{dist}[n]$ (Equation 10.12).

We estimate λ using the slope of the natural log of $\text{dist}[n]$ (Equation 10.14). We only need the first 14 samples because $\text{dist}[n]$ reaches a saturation point after these many samples. The

* R^2 can also be determined with the MATLAB function `regress` in the Statistics Toolbox.

Biosignal and Medical Image Processing

appropriate region for determining the estimate of λ (samples 1 through 14) is determined by visual inspection of the data in Figure 10.15.

```
% Ex 10.10 Determine the Lyapunov exponent of the logistic equation.
r=4; % Driving force for chaotic solution
N=25; % Number of samples (not many needed)
%
xint = [.1,.1001]; % Initial conditions.
% Generate data
for j=1:2 % Create two trajectories
    clear x;
    x(1)=xint(j); % with different init. conditions
    for n=1:(N-1);
        x(n+1)=r*x(n)*(1-x(n)); % Construct logistic map
    end
    xmat(j,1:N)=x;
end
%
d=sqrt(xmat(1,2)-b*xmat(2,:).^2); % Distance between trajectory points.
.....plot lambda and inspect.....
d=log(d); % Log divergence, Eq. 10.14
.....plot lambda and inspect.....
x=1:14 % Range of linear fit
[p]=polyfit(x,d(x),1) % Linear fit on first 14 samples
lambda_s=p(1); % Lyapunov exponent is the slope
r2=rsquared(d(x),p(1)*x+p(2)) % Determine the R^2 value of the fit
.....Label and plot.....
```

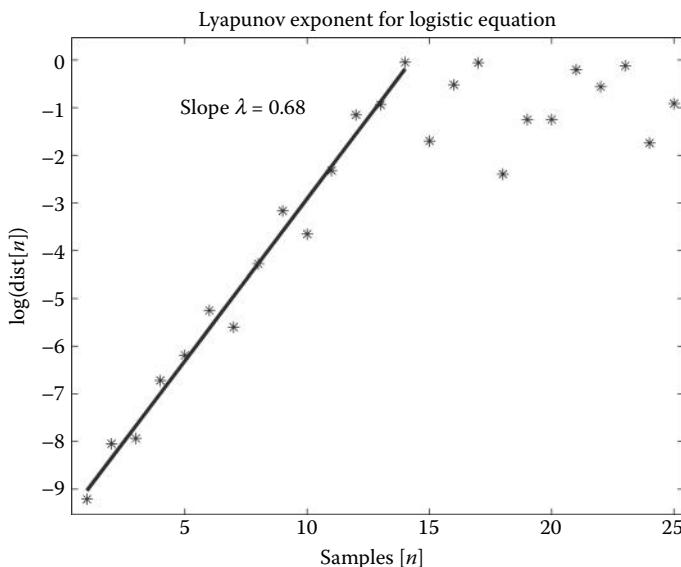


Figure 10.15 An estimate of the Lyapunov exponent based on the slope of the log of the divergence. Equation 10.14 gives a value of 0.68. The nominal value can be shown using an analytic method to be equal to $\log(2) \approx 0.693$; so, this value is very close to the nominal value. The R^2 value (Equation 10.16) is approximately 0.98, indicating that the curve can be considered very close to a straight line.

Result

The result of this code is presented in Figure 10.15. The Lyapunov exponent for the logistic map in the chaotic growth region ($r = 4$) can be analytically shown to be equal to the natural log of 2 ($=0.693$).^{*} The values determined in Figure 10.15, $\lambda = 0.68$, compare quite favorably with this theoretical result.

Equation 10.14 is also used to determine λ by estimating the slope of the natural log of distance change (Figure 10.15). Typically, we attempt to determine the linear portion of the divergence curve and estimate the slope based on Equation 10.14. In the determination of the slope, the R^2 value (Equation 10.15) is 0.98, indicating that the slope of $\log(\text{dist}[n])$ is very close to a straight line.

In this example, we make only a single estimate of λ . In Example 10.10, we use an algorithm for estimating λ that makes multiple estimates using the same set of data. Multiple estimates of λ allow for a more thorough estimate through averaging.

The slope of the natural log divergence curve will have some small dependence on the initial values chosen. Here, we compare an initial value of 0.1–0.1001, but choosing different values leads to a slightly altered slope. The solution to this issue is to assume that the differences in measured slope have a Gaussian distribution and that the ideal slope is recovered by taking the average over many test points. The next section introduces a method that uses this averaging technique.

When measuring the Lyapunov exponent of a known system, we may be able to determine the largest Lyapunov exponent exactly if the state equations are known. Alternatively, the state equations can be used to construct a series of trajectories that have similar initial conditions. From these, we could determine the divergence as in the last example. Identifying trajectories that have a well-defined set of similar initial conditions may also be possible for signals that are the responses to a stimulus-driven system such as evoked potentials. During an evoked potential test, we assume that all stimulus-driven recordings have similar initial conditions. However, for most real signals, we do not have multiple responses that start from similar initial conditions. Instead, we must arbitrarily choose a test point in the phase space along the reconstructed trajectory and then find a nearest neighbor to that point. Since this neighbor has values that are similar to our test point, we can use this neighbor as though it was the beginning of a “new” time series that has similar initial conditions. Of course, the time series starting from the neighbor is not really a newly measured time series; it is simply a different segment of the same signal.

Once the two trajectories are selected, the time history following the test point is compared against the time history following the neighbor. Their divergence is measured as though the two trajectories are the beginning of two different time series generated from the same system. Figure 10.16 illustrates the use of nearest neighbors in the phase space as stand-ins for repeated signals having similar initial conditions. We can see from this figure that the nearest neighbors in the phase space may not be close together in time. In fact, it is preferable that phase space neighbors are distant in time, or the estimation of the Lyapunov exponent will be biased. For this figure, the logistic map is used to generate the time series. This function has a parabolic trajectory in phase space (Figure 10.16). To generate this illustration, we only use 150 samples; the threshold used to define closeness is 0.2 times the standard deviation. Both these values are insufficient for accurate Lyapunov exponent determination; they are only used for illustrative purposes. With a larger number of samples, it would be difficult to resolve individual points and for smaller nearest-neighbor thresholds, the nearest neighbors would be too close together to tell

* The analytical method for determining λ is not suitable for measured data and thus is not relevant to real-world signal-processing operations.

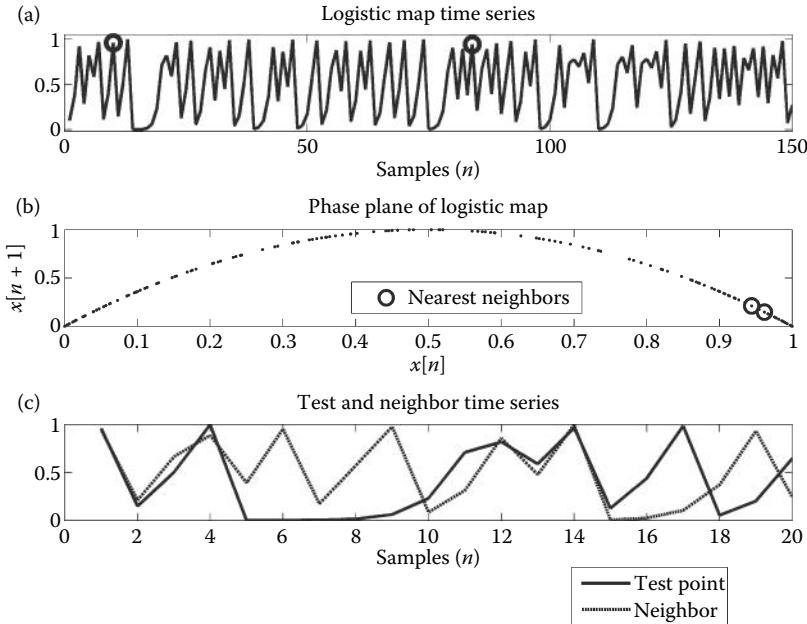


Figure 10.16 The use of nearest neighbors as substitutes for similar initial conditions. (a) The values used as nearest neighbors in the time domain (circled in black) are not that close in time. (b) In phase space, the location of these nearest neighbors (also circled in black) is close. (c) The two time signals with similar phase space values are shown superimposed in an expanded time frame. The time evolution shows that the signals remain similar for four samples before diverging.

apart. Even with this limitation, trajectories remain similar for four samples before diverging (Figure 10.16c).

10.4.3 Estimating the Lyapunov Exponent Using Multiple Trajectories

In Figure 10.16, we see that it is possible for points close together in the phase space to be separated in time. In Figure 10.16c, the two time segments are superimposed and we see that although the two time series begin with the same initial conditions, they quickly diverge. If we were to plot the natural log of this divergence, we would see that on average, this divergence is exponential. In practice, we need to use many such test points with similar initial conditions to estimate the average largest Lyapunov exponent.

A method described by Kantz and Schreiber (2004) takes advantage of the fact that any point along the trajectory may have nearest neighbors. If we treat these nearest neighbors as if they are the starting points of trajectories of similar initial conditions, we can use them to measure trajectory divergence. We can then find the average divergence of a number of similar trajectory pairs and determine the Lyapunov exponent from the average. By averaging the divergence, the effect of noise is diminished. One might think that taking the average over many divergence plots will result in the exponent converging to zero, but this will not happen if the signal is truly chaotic. In a chaotic system, the trajectories along the attractor always approach the region of maximum Lyapunov exponent and show exponential divergence over long enough time periods. Some regions may produce measurements with negative or low divergence, particularly if the trajectory is not examined over a long enough time history, but these contributions should be washed out in the average.

10.4 Quantifying Trajectories in Phase Space

The determination of the Lyapunov exponent using the multiple divergence-averaging method described in the previous paragraph is formalized below in Equations 10.17 through 10.19. The idea is that we are repeating measurements of the type performed in Figure 10.17. The equations are

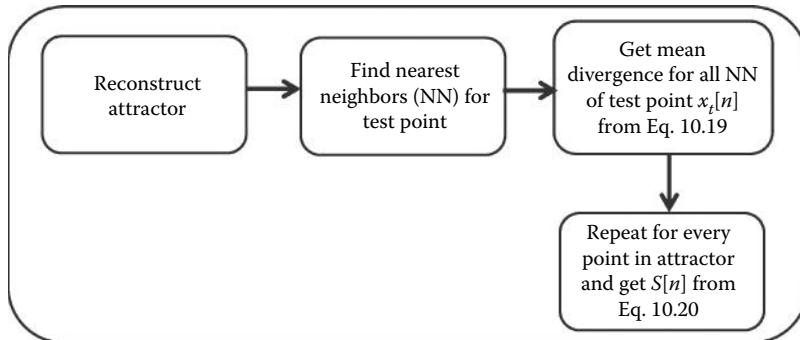
$$\text{Dist}_j[n] = \frac{1}{N} \sum_{k=1}^N \|x_T^j[n] - x_k[n]\| \quad (10.17)$$

$$S[n] = \frac{1}{M} \sum_{j=1}^M \log(\text{dist}_j[n]) \quad (10.18)$$

$$\lambda = \frac{dS}{dt} \quad (10.19)$$

where $x_T^j[n]$ is the current test point, $\|x_T^j[n] - x_k[n]\|$ is the norm of the divergence of the trajectory of the j th test point in the phase space having index n , $x_k[n]$ is the k th nearest neighbor of $x_T^j[n]$ and $\|$ indicates the norm operator. Kantz and Schreiber suggest that using the maximum norm rather than the Euclidean norm is sufficient because the loss of precision is justified by the

(a) Part one of `max_1yp` reconstructs the attractor and finds the nearest neighbors



(b) Part two of `max_1yp` determines λ from $S[n]$

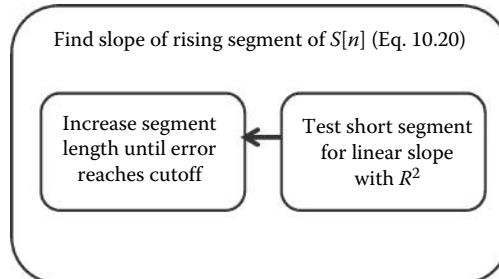


Figure 10.17 A flowchart for the first part of function `max_1yp`. This section performs the attractor reconstruction using delay embedding, finds the nearest neighbors for points, and gets the mean $\text{dist}[n]$ for each set of points and the respective nearest neighbors. (a) Flowchart of initial section of function `max_1yp`. (b) Flowchart of final section of `max_1yp`.

Biosignal and Medical Image Processing

simplified computation, but we suggest using the Euclidean norm anyway because the computational cost is insignificant on modern hardware. The average divergence for N -nearest neighbors of the j th point is $\text{dist}_j[n]$ and $S[n]$ is the average natural log divergence for all M -tested points in the phase space. As in Example 10.10, λ is determined as the slope of the rising part of $S[n]$ in Equation 10.20, provided that the slope is linear. If the slope is not linear or the slope is negative, then the system is not chaotic.

Because of the sensitivity to initial conditions, chaotic measurements are generally very sensitive to noise. Since the multiple trajectory-averaging method involves averaging over many determinations of the divergence, it is somewhat more robust to noise. When using this method, the number of samples in your signal and the sampling frequency can be critical since the number of calculations can quickly grow to become quite large. A high sampling rate will not only increase the number of samples in the signal, but will also increase the number of nearest neighbors (since the points in phase space will be closer together). While we need not use every point on the trajectory to estimate the Lyapunov exponent, a trade-off exists between accuracy and the number of calculations (and therefore computation time).

Function `max_1yp` is used to obtain an estimate of the largest Lyapunov exponent by estimating the mean divergence and calculating the slope of the rising region. The function uses Equations 10.19 through 10.21 to make the Lyapunov estimation. The syntax is

```
[lambda, S_mean, linear_end] = max_1yp(x,m,tau,fs,radius);
```

where x is the time series to be analyzed, the embedding parameters m and τ have their usual meaning, fs is the sampling frequency of the signal, and $radius$ is the threshold used to determine the nearest neighbors. For a time signal, fs has the usual meaning, but for a mapping function such as the logistic equation, fs should be the number of samples per iteration, usually 1. The output variable $lambda$ is the estimate of λ and $linear_end$ is the time index at which the estimated slope ends.

The code has two main parts, the execution of the algorithm to find S_n ($S[n]$ in Equation 10.18) and a second section that determines the slope of the transient portion of curve S_n . The program itself is rather long and is included in Appendix A, but here, we summarize the steps in the flowchart shown in Figure 10.18. There are two major components to the determination of S_n . The first is the attractor that is reconstructed using delay embedding. The second is a

Part one of `max_1yp` reconstructs the attractor and finds the nearest neighbors

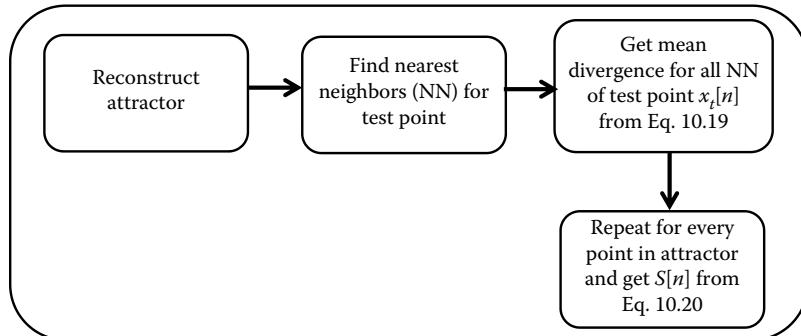


Figure 10.18 A flowchart for the second part of function `max_1yp`. This section determines the mean log divergence for the entire attractor, $S[n]$ (Equation 10.18) and then finds λ by determining the slope of the initial linear segment of the curve (Equation 10.19).

series of nested loops that search for the nearest neighbors and use them to determine the $\text{dist}[n]$ (Equation 10.4) for each pair of test points. The outermost loop finds the nearest neighbors for a given test point. The inner loop is used to determine the divergence for each nearest neighbor and the mean is taken (Equation 10.17). The outer loop cycles through each test point whereas the inner loop cycles through each nearest neighbor. The mean value of the divergences from the outer test loop is $S[n]$ in Equation 10.18.

In the second major part of the function `max_1yp`, the slope of the rising part of S_n is determined. Here, we take a small segment of the divergence curve from the beginning, extended through a few samples. The R^2 value of the linear fit to this segment is measured. If the R^2 value is below a certain threshold, the segment is elongated; this process proceeds iteratively until the R^2 value threshold is reached. The slope at the R^2 value threshold is taken as λ . An example of a Lyapunov exponent calculation that uses `max_1yp` is given in Example 10.11 where λ for the logistic map is estimated as a function of parameter r .

EXAMPLE 10.11

Determine the Lyapunov exponent for the logistic map over a range of driving force values, r . Plot both the population values from the logistic equation ($x[n + 1]$ in Equation 10.4) and also λ as a function of the driving parameter r . Determine the Lyapunov exponent estimate for the logistic map when $r = 4$.

Solution

Find the equilibrium population values of the logistic map after a number of generations. In this case, use 200 generations. The plot showing the equilibrium state for a number of generations as a function of the driving parameter, r , is known as the *bifurcation plot* because it shows how these equilibrium values behave and how these values will divide, or bifurcate, for certain ranges of r . The value of r should range between 3 and 4 since these values produce the most interesting behaviors. For each value of r , use the function `max_1yp` to determine the Lyapunov exponent. For `max_1yp`, use a dimension of 2, a delay of 1, and a cutoff value of 0.1 times the standard deviation of the time series.

We use Equation 10.4 to determine 200 iterations of the logistic equation (although we only plot the last 20 iterations) as a function of r for 10,000 different values between 3 and 4.

```
% Example 10.11 Lyapunov exponent for the logistic map
%
R=linspace(3,4,10000); % Create vector r values to test
N=200; % Length of logistic map function
m=2; tau=1; % Embedding parameters
fs=1; % Sampling frequency

for k=1:length(r)
    x(1)=.1 % Set initial value to 0.1 to insure
              % full bifurcation is represented
    for n=1:(N-1); % Iterate logistic map to get values
        x(n+1)=r(i)*x(n)*(1-x(n));
    end
    x_end(k,1:20)=x(181:200); % Save end values for bifurcation plot
cutoff=0.1*std(x); % Get cutoff value for the nearest neighbors
Lyapunov(k)=max_1yp(x,m,tau,fs,cutoff) % Get max Lyapunov exp.
end
[L_r4]=Lyapunov(k); % Get the Lyapunov exponent for r=4
.....Label and plot.....
```

Results

Figure 10.19a shows a bifurcation plot for the logistic map. This plot is constructed as in Example 10.2 using Equation 10.4 to calculate the value of the system (i.e., $x[n + 1]$) after enough generations are determined for the equilibrium behavior to develop. In general, a bifurcation plot shows the output of a function with respect to some critical parameter. The bifurcation plot shown in Figure 10.19a plots the last 20 iteration values of the logistic map at every value of r . Since the number of equilibrium states doubles at every bifurcation, many points are needed to show the range of values as the number of states increases. For each r value, an estimate of the maximum Lyapunov exponent is also shown in Figure 10.19b. Since negative exponents are indicative of nonchaotic behavior, positive values are expected when $r > 3.59$. Within this chaotic region, the exponents are nearly all positive, except around 3.825, the so-called “period-3” window. This region is a small window of nonchaotic behavior and we see values of λ close to zero. Finally, the estimate of the Lyapunov exponent $r = 4$ is 0.66.

Care must be taken when using the included routine `max_1yp`. This program gives an estimate of the slope of the rising portion of the natural log divergence curve, regardless of whether or not that portion of the curve is truly a straight line. As we have noted, an increasing divergence curve only means that trajectories on the attractor diverge but do not, by themselves, indicate that a system is nonlinear. Therefore, the actual divergence curve should be inspected to make sure that the program has not yielded a misleading result. The presence of chaos is only confirmed if the slope is increasing and has an approximate linear, straight line fit. If the slope does not have a straight slope, the signal is from a linear system or is too noisy to be assessed using the Lyapunov exponent. In addition, since this determination relies on determining the nearest neighbors, all the caveats discussed in the section on nearest-neighbor analysis apply here. Care is required in the selection of embedding dimension and delay as well as for the threshold used to define the nearest neighbors. The routine `max_1yp` plots the divergence curve over both long

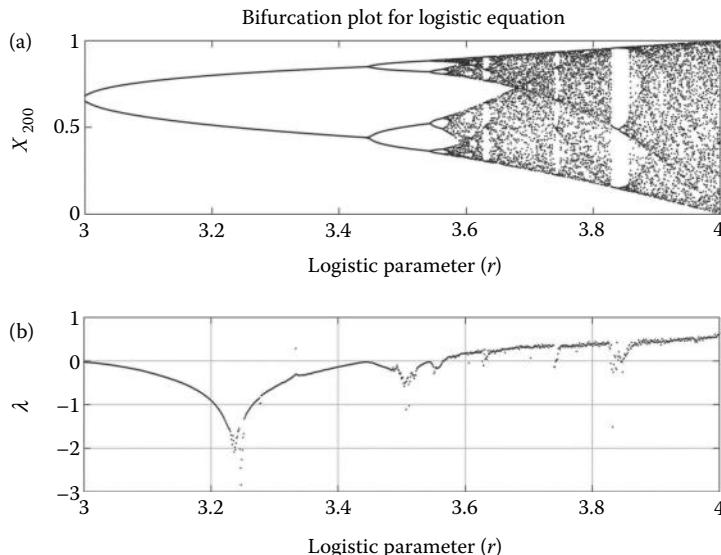


Figure 10.19 (a) The bifurcation diagram of the logistic map for r values between 3 and 4. The bifurcation plot of the logistic map shows *period doubling* between 3 and 3.57 and chaotic behavior for values > 3.57 . (b) The maximal Lyapunov exponent is shown to be negative in the periodic region and positive in the chaotic region.

and short scales to show the region of a straight line fit. It is advisable to use multiple values of the nearest-neighbor cutoff threshold (input **radius**) to validate the results.

We should mention a few things to keep in mind while testing for λ , as there are a number of additional potential pitfalls in the process. For example, in a noisy or highly stochastic linear system, trajectories that start at similar initial conditions diverge nearly instantaneously, after only a few samples,* although they do not grow exponentially. Therefore, the presence of noise can make a linear system appear to have some divergence from initial conditions and this divergence may be mistaken for exponential divergence if not examined over a long enough time frame. Additionally, when we use delay embedding to reconstruct an attractor, the reconstruction may be inaccurate due to inappropriate values of either τ or m . In such a situation, the trajectories do not accurately represent the system and its attractor. Alternatively, we might reconstruct the attractor using a time frame that is too short to show the true nature of the system. Finally, we might not have measured enough regions along the attractor to get a good assessment of the overall divergence.

The errors described above, alone or in combination, may result in a divergence estimate greater or less than the actual trajectory divergence. It is important to get a sense of error for your measurement. One simple way to do this is to simply evaluate many signals from the same system so that you can get a sense of the mean and standard deviation of your nonlinear metrics. Keeping all these issues in mind during the analysis should alleviate many signal-processing headaches.

In the next section, we examine a geometric property of the attractor rather than a dynamic property. This evaluation is called the *correlation dimension*.

10.5 Nonlinear Analysis: The Correlation Dimension

The Lyapunov exponent examines the tendency of similar trajectories in chaotic systems to exponentially diverge. Here, we discuss a different phase space characterization, the *correlation dimension*. The correlation dimension measures a more fundamental property of the system, the shape of the attractor in phase space. In particular, the correlation dimension is a measurement of the *space-filling* property of the trajectory. We have discussed the concept of the minimum embedding dimension. This can be considered the dimension in which the attractor is fully unfolded, that is, when trajectories do not overlap. Since the paths of chaotic trajectories do not repeat and similar points display exponential divergence, correctly unfolded trajectories must be very close, yet must never cross. It is possible for an attractor to be contained within a dimension that is less than the minimum embedding dimension, but is still higher than the highest insufficient embedding dimension. This can happen when the dimension of the attractor is fractal (has a fractional dimension and self-similar properties) so that it falls *between* the highest insufficient dimension and the minimum embedding dimension. Correlation dimension provides an estimate of the dimension of the attractor and if it is a fractional dimension, this may indicate a strange attractor associated with a chaotic system.

10.5.1 Fractal Objects

Unlike typical geometric objects that come in integer dimensions (i.e., 1-D, 2-D, and 3-D), a fractal object has a fractional dimension (e.g., 1.24-D, 2.34-D, etc.). The dimensionality of fractal objects is somewhere between the integer values. It may be difficult to comprehend what it

* In fact, if the signal is randomly Gaussian, it decorrelates only after one sample and no further divergence can possibly occur.

Biosignal and Medical Image Processing

means to have a fractional dimension, but some idea can be gained by imagining crumpling a piece of paper, clearly a 2-D object. However, as the paper is crumpled into a ball, it begins to take the form of a 3-D sphere. The tighter it is crumpled, the closer it comes to a solid sphere, but is still just a highly folded 2-D paper. This highly folded shape can be thought of as “almost” 3-D: more than two dimensions but less than three, a noninteger. The correlation dimension is a measurement of this kind of dimensionality.

Fractal objects can be found in nature, such as coastlines. Coastlines are considered *self-similar*, meaning their basic shape remains similar when observed at different length scales: a coastline viewed from space has a ragged quality, just as it does if viewed from a plane, at ground level, or even microscopically (a little stretch of the imagination is needed here). Fractal objects generated mathematically (as opposed to objects found in nature) are exactly self-similar: the shape is identical at any length scale. An example of a mathematical fractal object is the Koch curve. This is a fractal object that has a dimension of >1 but <2 . It is constructed by first drawing a straight line. The next step is to divide the line into thirds. Next, the middle third is replaced by an equilateral triangle, each side of which is a third the length of the middle segment. Finally, the base of the triangle is removed (Figure 10.20, upper frame). This procedure can be performed iteratively, treating every newly created line segment the same as the initial line. Since fractal objects are self-similar, the object will appear identical at any scale.

With a computer program to automate the procedure, the fractal object can be constructed for any number of iterates. A program that generates the Koch curve is provided in the associated

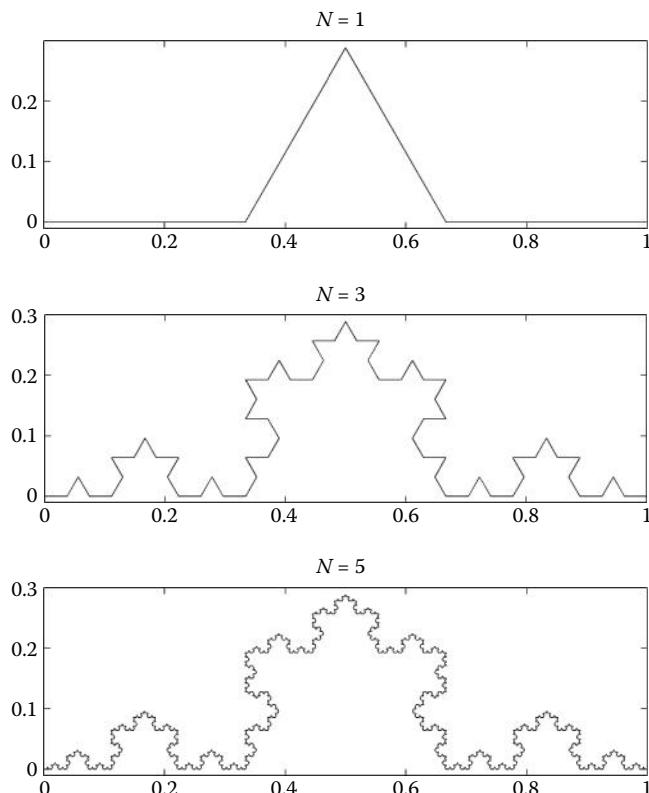


Figure 10.20 The Koch curve shows self-similarity, that is, it appears similar at different scales. Shown here are constructions based on one, three, and five iterations. Even at only five iterations, the shape appears highly complex.

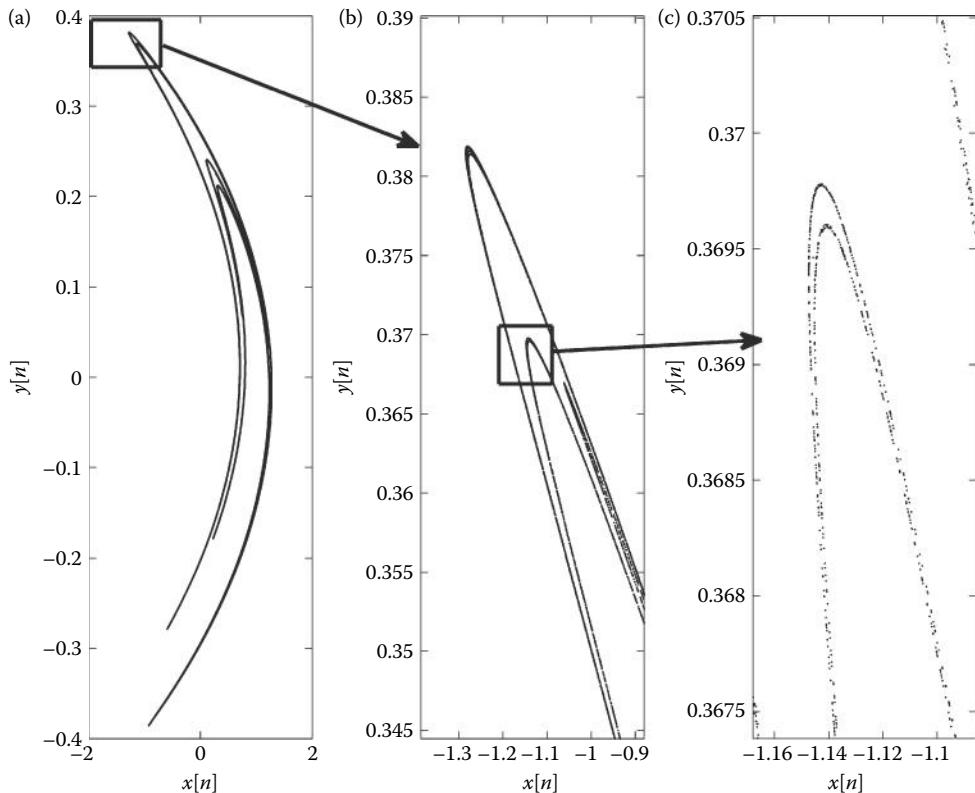


Figure 10.21 Three views of the Hénon map attractor showing the boxed region in (a) at two magnifications or scales (note vertical axis scale). Owing to the fractal nature of the attractor, the plots in (a), (b), and (c) resemble each other despite the difference in scale.

routines and is given in Appendix A. The output of the program is shown in Figure 10.20 after one, three, and five iterations. The curve shows self-similarity to the N th degree (where N is the number of iterations) and therefore is not fractal in an infinite sense. That is, since we cannot iterate this function an infinite number of times, there is only a finite number of the repeated base patterns within any interval.

Just as the Koch curve demonstrates fractal self-similarity, the trajectories of a chaotic attractor are typically fractal. For example, Figure 10.21 shows a section of the attractor of the Hénon map, a chaotic system with a 2-D attractor. The segment of the attractor shown in Figure 10.21a is enlarged and shown in Figure 10.21b that, in turn, is enlarged in Figure 10.21c. The plots in Figure 10.21a through c appears highly similar despite the difference in the scale (note axis values) of the image: in each plot, we see a convex curve within a convex curve. If we were to examine even finer scales, similar curves would be found. Since the attractor is fractal, similar curves would be found at any scale down to infinitely small scales. This generates a filling of phase space to the extent that 1-D space is insufficient to contain the attractor; in the limit, two dimensions would be needed. The correlation dimension allows us to examine and quantify this space-filling property.

10.5.2 The Correlation Sum

The correlation dimension can be estimated using the *Grassberger-Procaccia algorithm* (GPA). This algorithm uses a *correlation integral* (*correlation sum* for discrete data) to examine how an

Biosignal and Medical Image Processing

attractor fills the phase space. As in the Lyapunov exponent calculation, a key calculation of the correlation integral is the determination of nearest neighbors. Rather than investigating how points that are nearest neighbors evolve over time, the GPA measures the change in the number of nearest neighbors as a function of the radius around the point. The rate of increase of the average number of nearest neighbors as the radius is allowed to grow is related to the density of points along the attractor in the phase space. The summation of the nearest neighbors as a function of increasing radius value is known as the correlation sum.

The *correlation sum* shown in Equation 10.20 was introduced as a numerical approximation to the correlation integral. The equation is

$$C_d(R) = \frac{1}{N(N-1)} \sum_{i=1}^N \sum_{j=1, j \neq i}^N \Theta(R - \|x[i] - x[j]\|) \quad (10.20)$$

where the correlation sum, $C_d(R)$, represents the total number of points contained within a hypersphere of radius R as a function of R normalized to the number of points squared. The symbol Θ represents the Heaviside operator. The Heaviside operator, $\Theta(x)$, is 1 for $x > 0$ and 0 for all other values. To test if points are neighbors, we measure the distance between them using the Euclidean norm (Equation 10.11).

The correlation dimension is determined from a *scaling region* found in a plot of the correlation sum as a function of R . Specifically, the scaling region is a monotonically increasing linear region within a plot of $\log C_d(R)$ versus $\log R$. If a scaling region exists, we expect to see a section of the curve that increases linearly before it eventually saturates. The mechanics here is similar to that of finding the Lyapunov exponent from the slope of the distance versus samples curve (Equation 10.14) in Example 10.10 (Figure 10.15). Once a scaling region is observed, the slope of the scaling region is taken as the estimate of the correlation dimension, D^c .

Alternatively, the derivative of $\log C_d(R)$ with respect to $\log R$ can be used to find this region:

$$D^c = \frac{d \log C_d(R)}{d \log(R)} \quad (10.21)$$

In this case, a plot of this derivative curve is examined for a region that is linear and flat. The magnitude of the curve in this region becomes the estimate of D^c . As found with the Lyapunov exponent, it is typically easier to examine the curve given by Equation 10.20 for a linearly increasing monotonic region than to examine the derivative curve for a scaling region.

As previously described, a chaotic attractor is generally a fractal object, that is, the shape is self-similar and repeats at multiple scales. Therefore, a point in the phase space will have many neighbors. It should be apparent that since we are examining the point density of fractal objects, a good many points are needed for an accurate measurement. An example of the correlation dimension calculation using the GPA is given below.

EXAMPLE 10.12

Use the GPA (Equation 10.20) to estimate the correlation dimension of the Lorenz system, using the x dimension of the output of the numerically solved system. Determine the slope of the linear region of a plot of $\log C_d(R)$ versus $\log R$. Use the parameters from Example 10.5 to compute the Lorenz outputs. Use the embedding parameters of $m = 3$ and $\tau = 6$, a sampling frequency of 100 Hz, and 100 s of data. Use function `cordim_original` to compute the correlation dimension from numerical solutions of the x , y , and z system responses. Get the correlation dimension estimate using both Equations 10.20 and 10.21. For Equation 10.21, plot $C_d(r)$ on a double log plot and find the slope of the linear region, and plot the derivative of $\log C_d(r)$ with respect to $\log(R)$ (Equation 10.21) and find the region of 0 slope.

Solution

Use the approach in Example 10.5 including routine `lorenzeq` to generate the x output of the Lorenz system. Write a separate routine, `cordim`, to determine the correlation sum, $C_d(R)$. The output of this routine, CR, will be the correlation sum from Equation 10.20. Then plot the correlation sum on a double log plot and after determining the linear region, use the MATLAB routine `polyfit` to get the slope of this region. Determine the derivative of $\log C_d(r)$ using MATLAB's `diff` function and plot the derivative curve to look for a scaling region, the region of flat slope. This is another estimate of the correlation dimension.

```
% Ex 10.12 GPA to determine the correlation dimension of the
% Lorenz system.
%
.....Solve the Lorenz eq. as in Example 10.5.....
.....Solution is in variable sol.....
R = exp(-3:-0.05:-1);
    % Setup R values
[CR, CR_stat] = cordim(x, tau, m, R);      % Compute Cd(R)
%
x1 = 19;           % Start and stop points of observed
xf = 28;           % scaling (linear) region

method = 1; %choose Equation 10.22 for D_c computation
subplot(2,1,1);
Cd = cor_dim_plot(R, CR, x1, xf, method); % Compute Dc and plot
subplot(2,1,2);
method = 2; % Choose Equation 10.23 for C_d computation
Cd = cor_dim_plot(R, CR, x1, xf, method); % Compute Dc and plot
```

Solution

routine `cordim`. This routine is used to calculate the correlation sum (Equation 10.20):

```
CR = cordim(x, tau, m, R);      % Compute correlation sum
```

where x is the time series to be analyzed, τ and m are the embedding parameters, and R is the vector of radius values used to perform nearest-neighbor testing. In this example, the test points include 41 points that range logarithmically between e^{-3} and e^{-1} .

As in the function `max_1yp`, the first step is to use `delay_emb` to perform embedding. The function then uses two nested loops. The first loop cycles through each radius value in R and the second loop cycles through each point along the reconstructed trajectory. For each point, all the nearest neighbors that fall within the current radius R are counted and saved in `rcount`. We then use Equation 10.20 to get `CR` that is a count of the total number of neighbors along the trajectory as determined for every point on the trajectory. This count is saved in `CR` as a function of the radius tested.

```
function CR= cordim(x, tau, m, R)
    % Descriptive comments
%
y=delay_emb(x, m, tau);      % Construct the embedded vector
N=length(y(:,1));            % Embedded vector length
% Nearest neighbors are counted as in the Lyapunov exponent
CR=zeros(1,length(R));      % CR will be correlation sum
for jj=1:length(R)
    numc=0;                  % Test point loop
    rcount=0;                 % Initialize count to 0
    % Initialize rcount to 0
```

Biosignal and Medical Image Processing

```

for k=1:length(y(:,1)); % Get nearest neighbors
    vec=repmat(y(k,:),N,1);
    dist=sqrt(sum((y-vec).^2,2));
    dist(1:k)=[];
    numc=dist < R(jj);

    %count nearest neighbors
    rcount = sum(numc) + rcount; % Compute number of nearest neighbors
    rcount_stat(k) = sum(numc);
    dist = []; % Clear dist
end
CR(jj)=rcount; % Save counts
CR_stat(:,jj)=rcount_stat;
end
CR(CR==0)=1; % Set 0 values to 1
CR=2*CR/(N*(N-1)); % Average count using Eq 10.21

```

Analysis

Since plotting the results and determining the linear fit of the scaling region is a common repetitive part of using the correlation sum, we have put the appropriate plotting code in a separate function, `cor_dim_plot`:

```
Cd=cor_dim_plot(R,CR, x1,xf,method);
```

Here, R is the same as the input for `cor_dim`. CR is the output of `cor_dim` and $x1$, xf are the indices of the beginning and end of the scaling region. Values for $x1$ and xf are determined by observation from the plot of $\log C_d(R)$ versus $\log R$. If `method` is equal to 1, then the correlation dimension is estimated using a plot of the $\log C(R)$. If `method` is equal to 2, then the correlation dimension is estimated using $d \log C(R)/d \log (R)$. Output Cd is the estimated value for D^c .

The code contained within `cor_dim_plot` is straightforward and primarily consists of plotting commands, but we highlight several of the more important lines within it. To find the correlation dimension based on the log of the correlation sum, we use `method = 1`, and perform the following calculations:

```

logR = logR(x1:xf); % Get x and y values for curve fitting
logCR = logCR(x1:xf);
P = polyfit(logR,logCR,1); % Use polyfit to get curve fit for C^d;
Cd = P(1); % Slope is the Correlation dimension

```

Variables $\log R$ and $\log CR$ are segments from the $\log (R)$ and $\log C_d(R)$ series, respectively, which we believe to be the scaling region. The function `polyfit` performs a linear polynomial fit (i.e., order 1) as is done in the Lyapunov exponent function `max_1typ`. If we want to use $d \log C_d(R)/d \log (R)$, we use `method = 2`. The key components of the resulting analysis are

```

dlogR = diff(logR); % Get values for R axis, plot
% results, and curve fit
dlogCR = diff(logCR)./dlogR; % Get log C^d(R)
logR = logR(x1:xf); % x and y values for curve fitting
logCR = dlogCR(x1:xf);
P = polyfit(logR,logCR,1); % Use polyfit to get fit for C^d;

```

To compare the results using a reconstruction of the Lorenz system trajectory to the actual trajectory (one using all the three variables from the solution), we repeat the above calculations using a slightly modified form of `cordim`. This is function `cordim_original`. It is almost

identical to `cordim`, except that the portion of the code that performs delay embedding is removed.

```
% Repeat using all three dimensions of the numerical solution
Figure;
CR cordim_original(sol,R); % Use cordim to compute C^d(R)
```

Note that since we do not need to use delay embedding, the embedding parameters are not inputs to `cordim _ original`. The plotting is similar to that of the correlation dimension estimate for the estimated trajectory.

Results

The correlation dimension, D^c , of the Lorenz system is determined for a 10,000-sample signal taken from the x -axis solution of the Lorenz system. Figure 10.22 shows the correlation of 100 radius points plotted against the radius, R , as a log-log plot determined from the estimated trajectory. On the basis of the slope of the linear region indicated by a parallel dashed line in Figure 10.22a, the correlation dimension, D^c , for this system is 2.07. This value is in agreement with the estimate originally found by Grassberger et al. (1983) using the same algorithm as used here. With the derivative curve based on Equation 10.21, the value for D^c is found to be 2.126 but, as noted in Chapter 4, the derivative can introduce noise that may cause errors. If we want a more accurate measurement of the correlation, it would be prudent to try the determination again using a smaller range of R values to more accurately pinpoint the scaling region

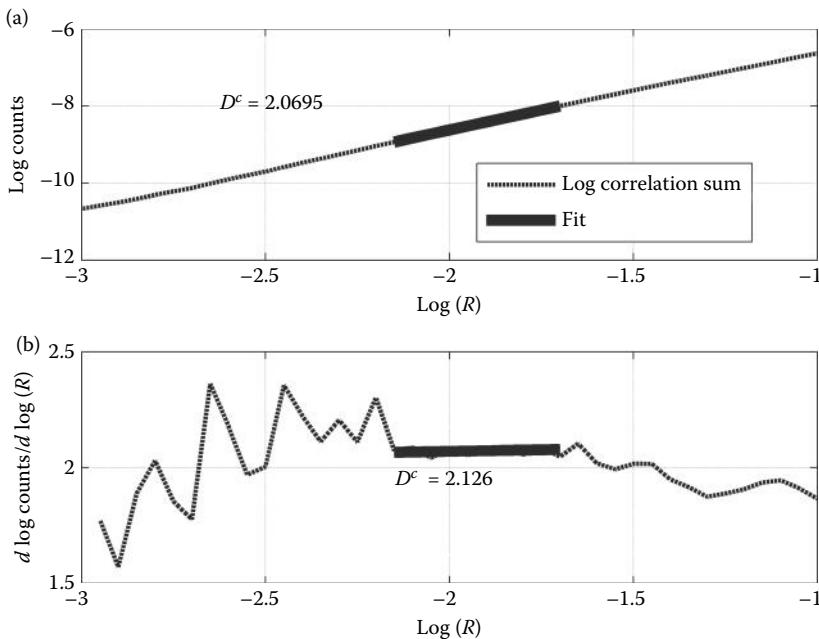


Figure 10.22 Estimating the correlation dimension of the reconstructed attractor of the Lorenz system. (a) The log correlation sum as a function of radius using the GPA. The correlation dimension is the slope of the scaling region, indicated by the thick black line. This flat portion of the curve indicates a correlation dimension of about 2.07. (b) Using the derivative of the curve in (a), we look for a flat region and estimate the value to arrive at the correlation dimension. Here, the estimate is about 2.126. Since the differentiation performed with `diff` can be noisy, determining the scaling region this way can be problematic.

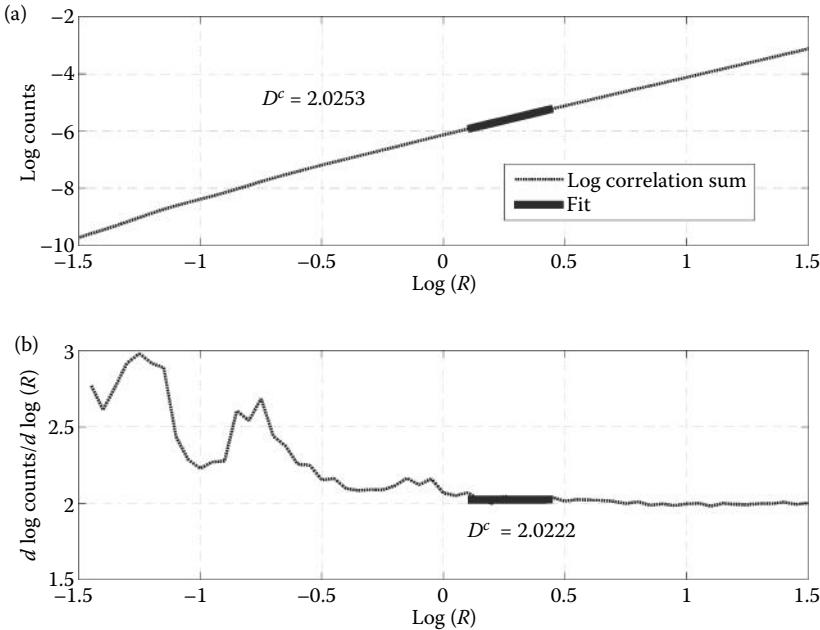


Figure 10.23 The derivate curve using the trajectory from the solution to the numerical solution of the differential equation. (a) The correlation dimension estimated using $C_d(R)$ gives a value of about 2.03. (b) Using the derivative of $\ln C_d(R)$ gives an estimate of about 2.02.

or to solve the Lorenz system for more points. Here, we intentionally broadened the range of R to show typical features such as the high-end saturation of nearest neighbors and the low-end determination of zero nearest neighbors. It is generally a good idea to start with a broad range of R values and modify as needed.

A second trial is performed using the numerically obtained x , y , and z outputs of the Lorenz system rather than delay vectors of the x component. The obtained correlation dimension is approximately 2.02, very close to the value obtained using the embedded vectors (Figure 10.23).

The scaling region is found by searching for the region in which there is the least error between the curve and a linear fit. The flat regions at the beginning and end of the curve are excluded because these regions correspond to the region where the radius is either too small or too large to give meaningful results. When the radius is too small, no points are counted; when it is too large, all points are counted.

A major limitation of the GPA is that it requires an accurate representation of the attractor. To reconstruct a fractal attractor accurately requires many noise-free data points: at least 10^m where m is the embedding dimension. The choice of radius values to test is also important. If a small radius is used, there may be no neighbors within the radius; so, the number of nearest neighbors found will be zero.* If the radius is too large, then every single point in the phase space will be contained within it. Finally, if too few points are tested, it will be difficult to determine the location of the scaling region. Since it is often difficult to determine the best range *a priori*, it is common to use several ranges for the radius, R , until a suitable range is found.

* In theory, if the number of samples in phase space is infinitely large, the number of nearest neighbors will not be zero for any nonzero radius. However, for finite samples, there is a radius below which no neighbors will be found.

As a final and arguably the most important point, the GPA has been shown to give low-dimensional fractal correlation dimensions for systems that are neither nonlinear nor chaotic. A deterministic signal will have a strange attractor only if the system is chaotic, but stochastic signals may produce attractors with fractal properties that can fool the GPA. One such signal is correlated or integrated noise. This is a noise signal that is integrated one or more times, which imposes correlation onto otherwise uncorrelated points. These systems do not contain nonlinear dynamics, but they may appear complex. Many biological systems may produce signals similar to correlated noise. When the GPA was first introduced, many signal processors were eager to apply it to biological signals in search for nonlinearity, especially in EEG analysis. In the late 1980s to the early 1990s, researchers used the GPA to show evidence of chaos in EEG waves. However, new methods for determining the presence of nonlinearity within signals, combined with the revelation from Osborne (1989) and Provenzale (1992) that correlated noise can fool the GPA, have made it clear that normal resting EEG waves contain little-to-no nonlinearity. This is explored in Example 10.13. Such errors illustrate the need for validating the presence of signal nonlinearity by using methods described in Section 10.6.

EXAMPLE 10.13

Construct a correlated noise signal using the MATLAB cumulative summation routine `cumsum`. For a vector input, this routine produces a vector output that is the cumulative sum of input elements, thus performing a simplified numerical integration. Apply this routine to a random vector of 10,000 samples uniformly distributed between -0.5 and 0.5 . Plot the autocorrelation to show that this signal does indeed contain correlated samples. Use the GPA to estimate the D^c of the correlated noise signal. For delay embedding, use three dimensions and a delay of 3000 samples. This delay value is determined from the autocorrelation function and is rather large because the integration produces long-term memory (i.e., long-term correlation). Test logarithmically spaced radius values between 10^{-3} and 10^3 when computing the correlation sum. Estimate the correlation dimension from a plot of the log of the correlation sum versus log radius by observing a scaling region and taking the slope.

Solution

The signal is created by applying `cumsum` to uniformly distributed noise from `rand`. The autocorrelation function is determined using `xcorr` with the '`'coeff'` option and plotted. The `delay_emb` routine is used to embed the signal in three dimensions and the resulting trajectory is plotted. The correlation sum is then determined using the correlation integral through the routine `cor_dim`. The slope of the log correlation sum is then found using a linear fit, which is taken to be the estimate of the correlation dimension. This analysis is the same as in Example 10.12; the only differences are the input signal and the estimate of the linear region of the log correlation sum. Routine `cor_dim_plot` is used for plotting and identifying the linear fit.

```
% Ex. 10.13 Correlation dimension applied to a structure noise signal
%
m = 3; % Embedding dimension
tau = 3000; % Embedding delay
%
sol = cumsum(rand(1,10000) - .5); % Integral of uniform noise
%
[x_cor, lags] = xcorr(sol, 'coeff'); % Autocorrelation function
plot(lags, x_cor) % Plot autocorrelation
%
```

Biosignal and Medical Image Processing

```

x_emb = delay_emb(sol,m,tau); % Delay embedding
plot3(x_emb(:,1),x_emb(:,2),x_emb(:,3)) % Plot trajectory,
R = 10.*linspace(-3,3,200); % Setup R values to test using GPA
CR = cordim(sol,tau,m,R); % Use cordim to compute C(R)
logR = log(R); % Get values for R axis, plot results,
% and perform curve fit
[var_b,var_m] = fit_error(logR(x1:xf),log(CR_stat(:,x1:xf)));
std_Cd = sqrt(var_m);
x1 = 53; % Start and stop points
xf = 76;
method = 1; % of observed scaling region
Cd = cor_dim_plot(R,CR,,x1,xf,method); % Compute Dc and plot

```

Results

Figure 10.24a shows the autocorrelation function that shows correlation between samples for well over 1000 lags. This extensive correlation is due to the integration process. Figure 10.23b shows the phase space trajectory produced by this signal in 3-D space. Note that the trajectory extends into all the directions of the phase space, a characteristic of noise. This is unlike a dissipative chaotic system in which trajectories are bounded. Since the trajectory is of a noisy signal, it technically could be embedded in any dimension and still tends to fill the phase space, but as we see, we can still infer a measurable correlation dimension from the 3-D trajectory.

A log-log plot of the correlation sum, D^c , versus the radius, R , is shown in Figure 10.25. The correlation dimension is taken as the best fit of the rising segment that appears the most linear. This is empirically determined to be between R -indices 53–86. The R^2 value of a linear fit in this region is 0.9995; so, the scaling region we have identified is indeed quite linear. The correlation dimension is estimated from the linear slope to be 2.84. The correlated noise signal has a fractional correlation dimension indicating chaos, but we know the signal is linear. This shows that you cannot solely rely on a fractional correlation dimension as a conclusive evidence of chaos.

The misleading results found in Example 10.13 clearly illustrate the importance of testing data for nonlinearity before testing for chaos. Chaotic systems are a subset of nonlinear systems and a lack of nonlinearity implies a lack of chaotic behavior. Evidence for nonlinearity should be present to proceed with a determination of the Lyapunov exponents or the correlation

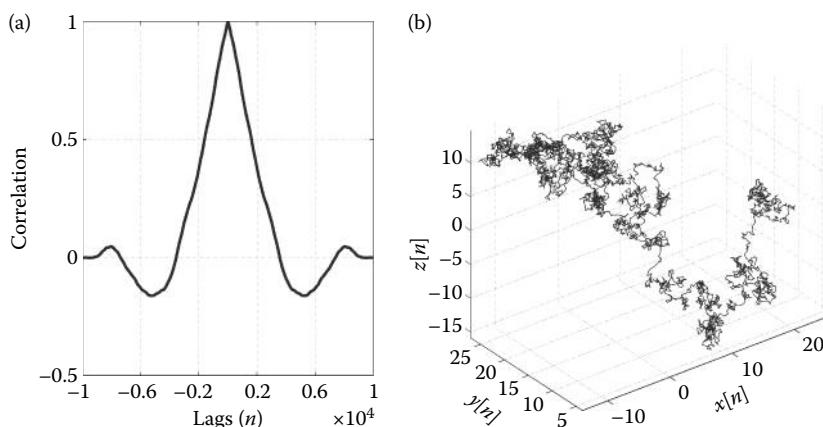


Figure 10.24 (a) The autocorrelation of correlated noise shows long-term correlation over many lags, with the zero crossing occurring at approximately 3000 lags. (b) A plot of the trajectory embedded in 3-D space shows a complicated-looking trajectory behavior that extends in three dimensions. Given enough points, this trajectory would tend to fill the phase space.

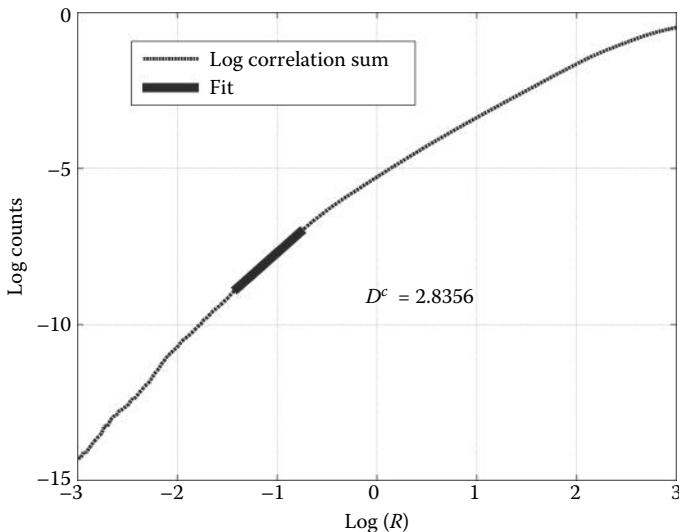


Figure 10.25 The slope of the log correlation sum as a function of log radius gives a fractal dimension of about 2.8356 for this signal; however, this signal is neither deterministic nor chaotic.

dimension. Tests for chaos should not be carried out blindly, but should only be used when there is reasonable suspicion that there are nonlinear signatures in the data. The next section describes a method to test for the presence of nonlinearity.

10.6 Tests for Nonlinearity: Surrogate Data Analysis

A simple way to test for nonlinearities is known as *surrogate* data analysis. In this method, an artificial data set is constructed that has linear properties similar to those of the original signal, but with potential nonlinearities removed. When testing for nonlinearities, this artificial signal is used as a surrogate or “stand-in,” for the original data, hence the name. By comparing the test results of the original signal with those of the surrogate data, we can check the validity of test results that indicate nonlinearity. The test results from the linear surrogate data should be conclusively different from those of the original signal; otherwise, the original signal does not contain nonlinearities.

Several methods for constructing surrogate data exist, but here, we cover only one simple, but powerful and popular, method. Each of the methods for constructing surrogate data tests a unique aspect of the signal and multiple methods should be used for a thorough evaluation of the presence of nonlinearity. The method presented here is a good starting point and can be used to definitively demonstrate if a data set does *not* come from a nonlinear system. (Again, chaotic systems are a subset of nonlinear systems and disproving nonlinearity is sufficient for disproving chaos.)

Surrogate representations of our signal are constructed by modifying the original signal in the frequency domain. In this approach, a surrogate signal is constructed that contains the same magnitude spectrum as the original signal, but none of the phase information. In destroying the phase information, we eliminate any nonlinearity contained in the original signal. The surrogate signal then contains only the linear information in the original signal. By destroying the phase information while preserving the magnitude spectrum, the process produces a signal that can be represented as the output of an ARMA process driven by a stochastic input (see Section 5.1.1).

Biosignal and Medical Image Processing

Given a sufficiently high order, an ARMA process can produce quite complex behavior, but since an ARMA process is linear, its output signal cannot contain nonlinear dynamics.

After constructing the surrogate data, we apply one of our nonlinear evaluation analyses and compare the results with those from the original signal. The differences between the surrogate analysis results and those of the original signal would indicate that the original data do *not* come from an ARMA process with stochastic input and may be generated by a nonlinear system. Such a mismatch is called “passing the surrogate data test.” While passing this test is not a definitive proof of nonlinear behavior, failing the surrogate data test indicates that all observable properties of the signal can be explained by linear features.

In constructing a surrogate data set, the Fourier transform of the original signal is taken, leading to a magnitude signal, M , and phase signal, θ (Figure 10.26). The first step in generating a new phase component, θ' , is to remove all the phase information for frequencies greater than $f_s/2$. Recall, these higher-frequency phase components are just the inverse mirror image of the lower-frequency components. The remaining lower-frequency phase values are then randomized. Next, these randomized values are concatenated with their inverse mirror image to restore the odd symmetry of the phase signal. (This is required to make the inverse Fourier transform real.) The surrogate signal is constructed by taking the inverse Fourier transform of the new phase component, θ' , and original magnitude component, M . This method can be used to generate any number of surrogates from the same original time series just by using a different random phase reshuffling. These steps are summarized in the flowchart in Figure 10.26.

The surrogate data construction method of Figure 10.26 is performed in the routine `gauss_surrogate`. This function is shown below and follows the procedure as outlined in the flowchart with one minor exception: the phase is padded with a zero because of the way MATLAB handles mathematical indexes. Creating multiple surrogate data vectors from one time series is easy: merely save the output of `gauss_surrogate` to a new variable. Since `gauss_surrogate` reshuffles the phase randomly every time, it outputs a different surrogate time series each time. Surrogates made from the same input time series have power spectra identical to the input signal but very different time domain behavior due to the phase randomization.

In routine `gauss_surrogate`, first the Fourier transform is taken and the magnitude and phase components are extracted from the complex transform. The phase is randomized using the MATLAB routine `randperm` that generates a random permutation of integers. These random indexes are used to reorder the original phase vector. After tacking on the mirror-image phase component, a new complex Fourier transform is constructed. Note that the complex Fourier transform can be written as $Me^{j\theta}$, where M is the magnitude and θ is the phase. The function then outputs the inverse Fourier transform of this new transform.

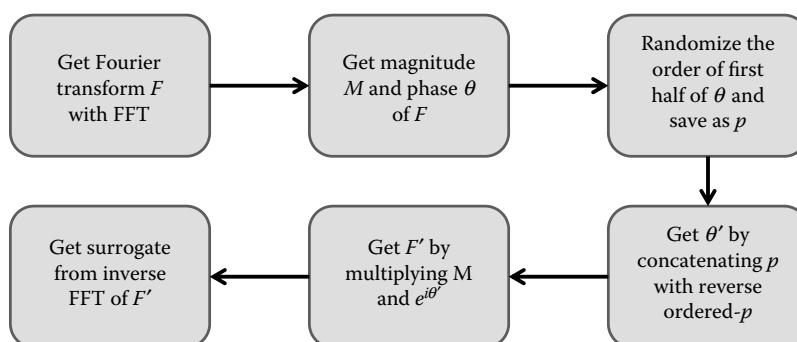


Figure 10.26 A flowchart outlining the procedure to produce surrogate data from a time series. These steps are performed in the included routine `gauss_surrogate`.

```

function y=gauss_surrogate(x)
    .....comments.....
%
x=x(:);
x=x';                                % Enforce row vector
N=length(x);                          % Get data length
if mod(N,2) ~=0;
    x(end) = [];
N=N - 1;
end
fftx=fft(x);                         % Fourier transform
mag=abs(fftx);                        % Magnitude component
phase=unwrap(angle(fftx));            % Phase component (unwrapped)
phase=phase(1:(N/2))-pi;              % Normalize first half of phase by pi
phase2=phase(randperm(N/2));          % Random phase vector
phase2=[0,phase2,-phase2(end-1:-1:1)]; % Make phase vector odd
fftn=mag.*exp(j*(phase2));           % Construct the new FT
y=ifft(fftn);                        % Inverse FT to get Surrogate

```

One thing to keep in mind when using this method is that the power spectra of the original and surrogate series will only be equivalent if the power spectra are computed using the same number of frequency components as were used when the surrogate series was developed. In `gauss_surrogate`, the number of frequency components used is equal to the number of samples in the signal.

The next example explores the use of surrogate data to analyze a time series. Here, the series comes from the Lorenz system; so, we know that it contains nonlinearities. Therefore, we expect to get different metrics from the surrogate data than those from the original time series.

EXAMPLE 10.14

Use an output of the Lorenz system, a signal known to contain nonlinearities, in an evaluation of the surrogate data test. Take the x output of the Lorenz system as a signal and apply correlation dimension analysis. Construct a surrogate of this signal and again apply correlation dimension analysis. Compare the two results. Show that the power spectrum of the original and surrogate series is the same despite having different time behavior.

Solution

Find the output of the Lorenz system using `ode45` with a time span of 100 s and 100-Hz sampling frequency. Take the x variable of the Lorenz system as a signal known to contain nonlinear properties. Determine the correlation dimension as in Example 10.12. Then using the function `gauss_surrogate`, construct a surrogate signal and determine the correlation dimension of that signal. Compute the magnitude spectra by taking the magnitude of the Fourier transform.

```

% Ex. 10.14 Use the surrogate data technique to create and test
% a surrogate of the Lorenz system.
%
% Set up Lorenz system initial conditions as in Example 10.12
[t,sol]=ode45(@lorenzq,tspan,y0); % Solve using ode45
x=sol(:,1)                         % Get output (X axis) of Lorenz
%
sur=gauss_surrogate(x);             % Construct surrogate

```

Biosignal and Medical Image Processing

```

fftx=20*log10(abs(fft(x))) ; % Magnitude spectra of surrogate
fftsur =20*log10(abs(fft(sur))) ; % and original signal.
freq=(1:10000)/10000*fs; % Generate frequency vector
.....label, plot.....
% The correlation dimension of the surrogate time
% series is computed as in Example 10.12.
.....label and plot.....

```

Results

Figure 10.27 shows the Fourier transform and time-series behavior of the Lorenz series and its corresponding surrogate. The two time series are quite different, with the sharp transient characteristic of the Lorenz system very apparent in Figure 10.27a, but not in the surrogate data (Figure 10.27b). However, the magnitude spectra (Figure 10.27c and d) appear identical. Any analysis method that characterizes the system based on spectral properties or statistical moments alone will not find a difference between the two systems.

The correlation dimension of the surrogate data is found from the plot of $\log C_d(R)$ (Figure 10.27). A comparison of the results of this example with those of Example 10.12 shows that the correlation dimension is different for the original data and their surrogate: 2.02 for the original and 2.83 for the surrogate. This would suggest that the signal does contain nonlinearities, but a comparison with only one surrogate is not enough to make any claims about nonlinearity. The surrogate signal is generated by scrambling the phase randomly so that this signal is inherently random. We need to use a large set of surrogate signals and compare the distribution of the analysis metric (correlation dimension in this example) from this set with the value obtained from the original data. If the original data fall within this distribution, then the signal probably does not contain nonlinear properties. Conversely, if the analysis metric from the original signal falls outside the

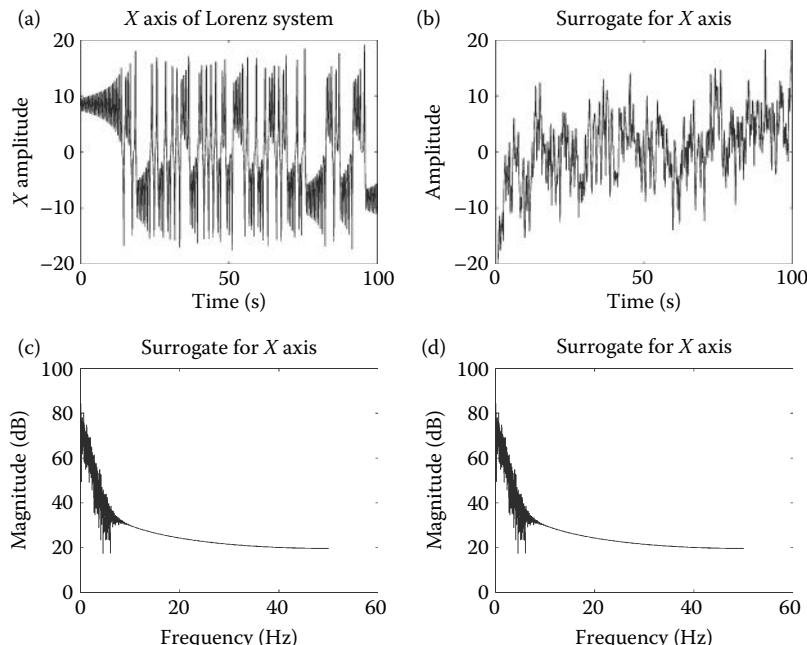


Figure 10.27 The time behavior of the Lorenz series (a) and a surrogate signal (b) show quite different dynamic behavior. The magnitude spectrum of the Lorenz series (c) is very similar to that of the surrogate signal (d).

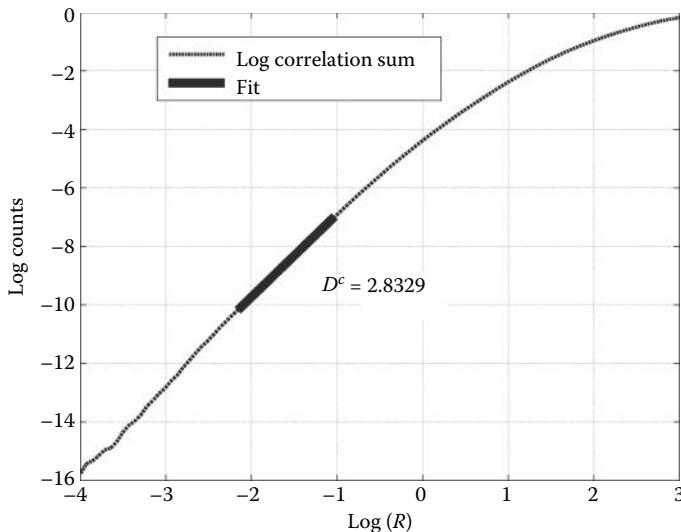


Figure 10.28 The correlation dimension of the surrogate series in Figure 10.25b is shown. This metric gives a D^c of 2.83. This is quite different from the value of 2.02 found for the original signal in Example 10.12.

surrogate signal distribution, preferably far outside, then this is evidence that the signal contains nonlinearity. Figure 10.28 shows the correlation dimension estimate for a surrogate of the Lorenz equation solution. The estimate of the surrogate is about 2.83, compared with an estimate of the Lorenz equation of about 2.02. If this trend held for several surrogates, we would be reasonably certain that the Lorenz equation solution contained nonlinearity. Indeed, we know that it does.

Although correlation dimension was used here, this approach applies to a variety of analysis measures. Since we cannot assume that the distribution of whatever metric we use is Gaussian, it is not correct to express the difference between the original and surrogate values in terms of standard deviations from a mean value. Rather, we use the rank order of the metric with respect to its distribution. Since rank order analysis does not require the metric to have a specific distribution, it is known as a *nonparametric* statistical test. In this method, shown in the next example, we sort the metric values from least to greatest using values obtained from both the original and surrogate data. We determine if the original data metric is inside or outside the range of the surrogate metrics. If the measured data metric is well outside the range of the surrogates, that is considered a passing grade for nonlinear behavior. In this example, we demonstrate a signal that does not pass surrogate testing.

EXAMPLE 10.15

Use `sig_noise2` to create a noisy 10-Hz sine wave with 10-dB SNR ($N = 2000$ and $f_s = 100$ Hz). This generates a deterministic signal with some stochastic properties added as is often found in biological signals. Determine if this signal has nonlinear behavior using the correlation dimension and compare the dimension value from the original signal with those determined from a data set consisting of 10 surrogate signals.

Solution

Compute the correlation dimension using the GPA using test radii of 100 evenly spaced values from e^{-4} to e^3 . For embedding parameters, use an m of 2 and a delay τ of 3 (this is approximately

Biosignal and Medical Image Processing

a quarter of the sine wave wavelength).^{*} First, we determine the correlation dimension of the noisy signal. In a loop, we generate 10 surrogates using the routine gauss _ surrogate and evaluate the correlation dimension. Finally, we compare the results of the original noisy sine wave and its surrogates using rank analysis.

```
% Ex 10.15 Determine if a signal contains nonlinear properties.  
%  
N=2000; % Number of points  
fs=100; % Sampling frequency  
SNR=10; % Sine wave SNR  
f=10; % Sine wave frequency  
m=2; % Embedding dimension  
tau=round(fs/10/4); % Embedding delay (1/4 wavelength)  
R=exp(linspace(-4,3,100)); % R values to test using GPA  
%  
x=sig_noise(f,SNR,N,fs); % Generate noisy sine wave  
method=1;  
[CR,CR_stat]=cordim(x,tau,m,R); % Use cordim to compute Cd(R)  
x1=36; xf=56; % Indices for scaling region  
[Cd, std_Cd]=cor_dim_plot(R,CR,CR_stat,x1,xf) % Get Dc and plot  
%  
for k=2:11  
    surrogate=gauss_surrogate(x);  
    CR=cordim(surrogate,tau,m,R); % Compute Cld(R)  
    Cd(k)=cor_dim_plot(R,CR,x1,xf,method); % Get Dlc and plot  
end  
%  
[ranked,ord]=sort(Cd) % Order Cd low to high  
original_rank=find(ord == 1); % Find rank of sine wave data  
%  
bar(ranked,'k'); hold on; % Plot surrogate Cd  
bar(original_rank,temp(original_rank),'w'); % Plot signal Cd  
.....labels.....
```

Results

Figure 10.29 shows the rank order for the noisy sine wave (white bar) and the surrogates (gray bars). The correlation dimension measured is about 1.97, similar to what we would expect for a sine wave, which should yield a correlation dimension of 2. The correlation dimension measured for the 10 surrogates ranges from about 1.95 to 1.98; so, our sine wave estimate falls within the range of the surrogates. This indicates that the properties of the noisy sine wave can be reproduced by an ARMA process and that the signal contains no nonlinear signatures. A more definitive answer would require a larger number of surrogates (20, or even 100, would be ideal), but only 10 are used here so that the computation time is reasonable for reproduction on a PC.

To rule out the hypothesis that our data could have been generated by an ARMA process, we want the value from the original signal to rank, not just first or last, but to be widely separated from the surrogate signal values. An example of rank order of the Lyapunov exponent for 20 surrogates is shown in Figure 10.30. The procedure is similar to that of Example 10.15 except that the test signal comes from the logistic map, the test metric is the Lyapunov exponent, and the 20 surrogates are used.

* This delay is approximately a quarter wavelength; so, the delayed signal is 90° out of phase and should produce a circle in a 2-D embedding. Without noise, the phase trajectory will be a perfect circle; so, the correlation dimension is clearly 2.0.

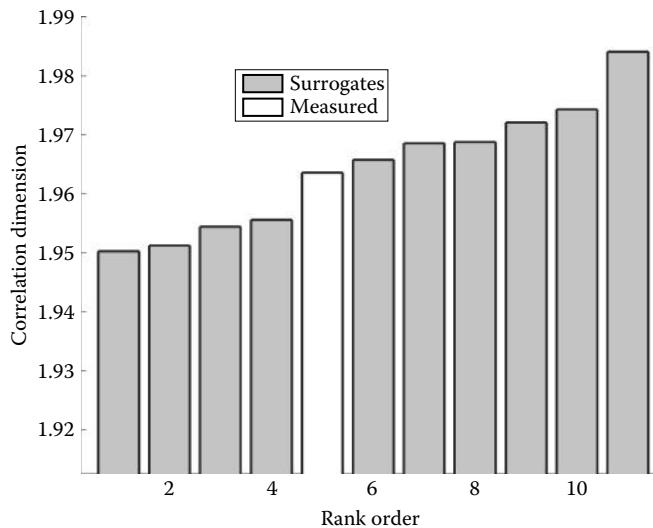


Figure 10.29 The rank of the correlation dimension of a signal, in this case, a noisy sine wave, compared to 10 surrogate signals. The correlation dimension of the signal falls within the range of the surrogates and therefore fits the hypothesis that it could be generated by an ARMA process and is not nonlinear. Note the scale offset of the vertical axis.

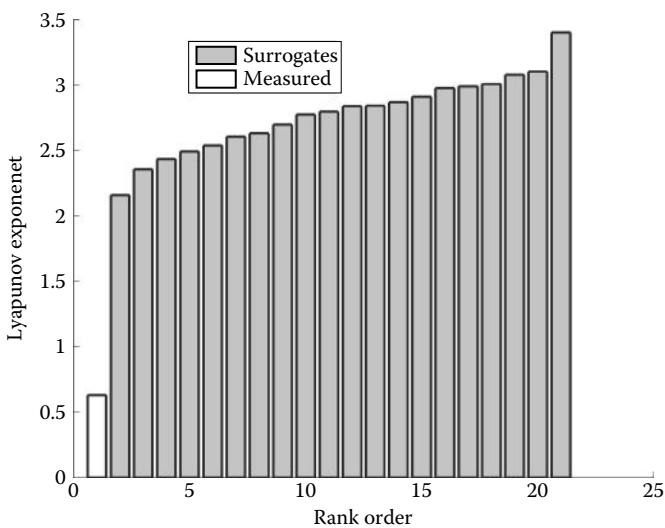


Figure 10.30 The Lyapunov exponent of the logistic map (white bar) has the lowest rank order of the set of Lyapunov exponents determined for original and surrogate data. The value of the logistic map exponent differs by a factor of 4 from the mean value of the surrogates. Since a large number of surrogates were used, this is a strong evidence (but not definitive proof) that the logistic map is not an ARMA process and contains nonlinearities.

EXAMPLE 10.16

Determine if a signal based on the logistic map is nonlinear by comparing the Lyapunov exponent from this signal with that of 20 surrogates. For the logistic map, make $r = 4$ so that this resulting signal is chaotic and could not be generated by an ARMA process. Use 1000 samples, $m = 2$ and $\tau = 1.0$. Set the threshold radius for identifying the nearest neighbors to 0.01 times the standard deviation of the associated signals.

Solution

Generate the logistic map values and determine the Lyapunov exponent using `max_1yp`. Then use `gauss_surrogate` to generate 20 surrogates and determine their Lyapunov exponents. Use a rank order approach to compare the Lyapunov exponents of the original and surrogate signals.

```
% Ex10.16 Evaluate the logistic map for nonlinear behavior
%
r = 4; % logistic map driving parameter
N = 1000; % Number of generations (samples)
x(1) = 0.1; % Initial condition
m = 2; % Embedding dimension
tau = 1; % Embedding delay
fs = 1; % Sampling frequency
%
for n = 1:(N-1); % Generate data from the logistic map
    x(n + 1) = r*x(n) * (1-x(n));
end
cutoff = .01*std(x); % Nearest neighbor threshold
Lyapunov(1) = max_1yp(x,m,tau,fs,cutoff); % Lyapunov exponent
%
for k = 2:21 % Surrogate Lyapunov exponent
    surrogate = gauss_surrogate(x);
    Lyapunov(k) = max_1yp(surrogate,m,tau,fs,cutoff); % Surrogate exps.
end
.....plotting similar to Example 10.16.....
```

Results

Figure 10.30 shows, in rank order, the Lyapunov exponents found for both the logistic map (white bar) and the surrogates (gray bars). Unlike for the sine wave function, the logistic map has a Lyapunov exponent that ranks much lower than all the surrogates. The separation is more than four times the mean of the surrogates and since we use 20 surrogate values, there is at least a 95% probability that this difference is not due to chance. If the distribution of the surrogate measurements appears to be Gaussian, the mean and variance can be used to determine statistical significance; however, a Gaussian distribution cannot be assumed.

You might question why the estimated Lyapunov exponent of the surrogates is positive even though the surrogates are linear and not chaotic. This is because the broadband characteristic of the logistic mapping function leads to surrogate data that appear very noise like. The Lyapunov exponent of noise should be infinite and if we were to examine the $S[n]$ curves of the surrogates, we would see that they saturate very quickly. However, since the data are discrete, the $S[n]$ curves require at least one sample to saturate. This leads to positive and finite Lyapunov exponents. This also shows that an estimated positive Lyapunov exponent is not always associated with chaos and that it does not, in itself, provide definitive proof that a signal is chaotic.

10.7 Summary

Nonlinear analysis has not reached the level of rigor that linear analysis has achieved, but in this chapter, we have tried to present a logical approach for how to apply nonlinear analyses to a signal. This approach roughly correlates with the sections of this chapter. The three main steps of the process are to first decide if nonlinear analysis is appropriate, then choose appropriate nonlinear measurement techniques, and finally verify the results using some sort of objective test. At the end of these steps, you should have a good idea if your signal contains nonlinearity and if further testing with more advanced methods should be performed.

There are several reasons to choose to perform nonlinear analysis. These include *a priori* knowledge of the system being measured (the system is known to contain nonlinear elements, such as the complex systems mediating heart rate) and a complex appearance of the measured signal. A weaker reason to attempt nonlinear analysis is if linear analysis has not produced sufficient results. This motivation dictates especially careful verification of the presence of nonlinearity.

After deciding that your signal may contain evidence of nonlinearity, the next step is to attempt to reconstruct the attractor of the system. The method of delay embedding allows for the reconstruction of a signal's phase trajectory in any number of dimensions, provided the signal is long enough. To estimate proper embedding dimensions, the method of false nearest neighbors and SVD may be used to estimate a sufficient embedding dimension. One can also test for the value of an invariant measure as a function of the embedding dimension. In this test, the optimal embedding dimension is the minimum dimension for which the invariant measure reaches a consistent value. All these methods should be tried to provide some verification. Finally, any knowledge about the system being tested should also be used to determine the embedding dimension. Determining the embedding delay is a little more straightforward. The popular methods include the autocorrelation of the signal, rate of automutual information decay (covered in the next chapter), and leveraging knowledge of relevant timescales of the signal. Again, all available methods should be used to validate the results.

After reconstruction of the attractor, the next step is to choose a nonlinear analysis method and apply it to the results. In this chapter, we present two nonlinear metrics. The first, the maximum Lyapunov exponent, is a parameter that characterizes how trajectories in phase space diverge over time after starting from similar initial conditions. The Lyapunov exponent is positive for chaotic systems and negative or zero for stable nonchaotic systems. The Lyapunov exponent is also positive for noisy systems. The second metric, the correlation dimension, characterizes the geometry of the attractor in the phase space by estimating the dimensionality of the attractor. Correlation dimensions may be fractional, implying an attractor that fills a dimension that is between two integer dimensions. A positive Lyapunov exponent and a fractional correlation dimension imply that a system contains nonlinear signatures; however, these results are not sufficient and surrogate data testing is required.

Surrogate data testing is a method for verifying the presence of nonlinearity in a signal. The concept is to create surrogate signals that contain the same linear properties as your signal but none of the potential nonlinear properties. A pool of surrogate signals is developed and tested with a nonlinear metric. If the results are similar to the original data set, then the data are unlikely to contain nonlinearity, but if the results are very different, then the signal may truly contain nonlinearity. A simple way to develop surrogate data is to create a signal with the same magnitude spectrum as your measured signal but with a randomized phase. This signal will have the same linear properties as the measured signal but none of the nonlinear properties. After performing surrogate data analysis, you can be reasonably certain about the presence of nonlinearity in your signal.

EXERCISES

- 10.1 Using a pendulum model and ODE45 (as in Example 10.1), simulate a damped chaotic pendulum and plot the results. Is there a damping factor value that returns the

Biosignal and Medical Image Processing

chaotic solution to a linear solution? Why do you think this is the case? Use a driving force of $k = 0.75$.

- 10.2 Linearize the function in pend using the small angle approximation and name it pendL. Use this function with pend and ODE45 to show that the nonlinear pendulum under small initial deflection gives the same output as a linear pendulum. This demonstrates what is known as “small signal linearity.”
- 10.3 Use pend to construct the phase space plot for a linear, nonlinear, and chaotic solution of the system. Use the same parameters as in Exercise 10.1.
- 10.4 The Van der Pol oscillator is a nonlinear oscillator similar to the linear oscillator, except it has a nonlinear damping term that is positive for large oscillations but negative (adds energy) for small oscillations. The Van der Pol oscillator develops stable but nonlinear oscillations and can be described as a first-order system with the equations:

$$\begin{aligned}\dot{x} &= y \\ \dot{y} &= u(1 - x^2)y - x\end{aligned}$$

Using the function vanderpol, solve the Van der Pol equations using a time span from 0 to 100, and initial conditions of $x_0 = 0.01$, $y_0 = 0$. Solve the equations for $u = 0$, 0.1, 1.0, and 5.0. For each value of u , plot the trajectories in phase space as well as $x(t)$ versus t .

- 10.5 On the same set of axes, plot the first 100 values of the logistic map using initial values of 0.1 and 0.101 and an r value of 2. Repeat this for values of 3 and 3.8. How do the trajectories starting from 0.1 to 0.101 compare for each value of r ?
- 10.6 Plot a phase trajectory for the logistic equation using the derivative on the horizontal axis (i.e., $dx[n]/dt$) and $x[n]$ on the vertical and compare it to the trajectory constructed using a delay of $n = 1$, that is, $x[n]$ against $x[n + 1]$. Are they the same? As with the pendulum system, the phase planes of most mechanical and other real-world systems plot the derivative of the output against the output (i.e., velocity against position).
- 10.7 Create a phase plane plot for the logistic map for $r = 1, 2, 3$, and 3.6. How does the parameter affect the phase plane?
- 10.8 Compare the phase plane formed when plotting $x[n + 1]$ against $x[n]$ for uniformly distributed noise (plot $x[n + 1]$ while holding $x[n]$ constant) and for Gaussian’s distributed noise (use MATLAB function rand and randn.) Use an N of 10,000. Plot a vector of the points in the phase plane for Gaussian’s distributed noise. Does this look like what you would expect based on what you know about the Gaussian distribution? (What does a Gaussian distribution look like in 1-D?)
- 10.9 The correct value for the delay is important for proper phase space reconstruction. Using a sine wave of 10 Hz and a sampling frequency of 5000 Hz, write a program that lets you increase the delay until the phase space appears correctly as a circle. Make sure to use square axes when plotting. How does the delay value relate to the sine wave?
- 10.10 Using the solution $x(t)$ of the Van der Pol equation from Problem 10.4 with $u = 5$ and the autocorrelation, determine an appropriate embedding delay for reconstructing the trajectory in a 2-D phase space. Compare this reconstruction with the trajectory determined when plotting $x(t)$ versus $y(t)$.

- 10.11 Using the delay from Problem 10.10, use `fnumnear` to confirm that 2 is a reasonable embedding dimension for the Van der Pol system. Search for an appropriate r for `fnumnear` value between 1 and 3.
- 10.12 Repeat Example 10.8 but apply a 4th order bandpass filter from 10 to 100 Hz to the ECG signal before using `fnumnear`. How does the filtering affect the results? Why is this the case?
- 10.13 From inspection of the Hénon map equations, determine the best values for the embedding dimension and delay.
- 10.14 Using autocorrelation, determine the best value for the delay of the Hénon map.
- 10.15 Using the technique of false nearest neighbors and a delay value of 1, show that the optimal embedding dimension of the Hénon map is 2. Use r values between 2 and 5 until there is a clear break in the `numnear` plot. Plot the phase space first using embedding vectors determined from the x -axis and then the y -axis. How do these compare?
- 10.16 Plot the phase space in three dimensions of the ECG using delays correlating to 1.25, 12.5, 25, and 125 ms. How does the change in delay affect the phase space? Why does this happen?
- 10.17 How would the presence of broadband noise in a signal of interest affect your estimation of the optimal delay? Would you expect the same behavior if the signal is linear compared to nonlinear? What about a comparison of stochastic versus deterministic systems?
- 10.18 How will the nearest neighbor estimation be affected if an estimated attractor is sparse (i.e., there are few points in the phase space relative to the dimension)? What if there are many more samples than needed? How would the choice of radius be affected by the number of points available?
- 10.19 Using a Scree plot and the Hénon map (see Example 10.9), determine the best embedding dimension. Do this using $x[n]$ and then repeat using $y[n]$. For both, use at least four delayed versions of the test signal to build your matrix for SVD.
- 10.20 You have just measured a time series for a system that you are reasonably certain is nonlinear and chaotic, but you have incorrectly embedded the time series in an embedding dimension that is too low. How does this affect your choice for choosing the nearest neighbors to test for exponential divergence? What effect does this have on the divergence measurement?
- 10.21 Prove that the estimated Lyapunov exponent for the logistic map with ($r = 4$) is a function of the initial test point by testing multiple points. Test at least five points. Then modify the code in Example 10.10 to make a better estimate by computing the Lyapunov exponent for many initial values and taking the average over the estimated values. Remember, every time you change the initial value for your test point, you need a new nearest neighbor as your second test point. (I.e., if your new initial value is 0.4, you need a new neighbor test value such as 0.4001.)
- 10.22 What is the theoretical Lyapunov exponent of a periodic function such as a sine wave and why? Prove your case by using `max_lyp` to estimate the Lyapunov exponent of a 100-Hz sine wave with a 1000-Hz sampling frequency.
- 10.23 Examine the data provided in file `Lyapunov_test_data.mat`. This contains variable data: each row is a measurement from an “unknown” source that may be

Biosignal and Medical Image Processing

chaotic or random. Assume each vector contains an independent measurement of the system but that both systems start from similar initial conditions. Use the methods from Example 10.10 to determine if the signal is noise or is from a chaotic system.

- 10.24 Use the delay found in Exercise 10.10 and an embedding dimension of 2, 5; use `max_1yp` to find the Lyapunov exponent of the Van der Pol equation for $u = 5.0$. Then compare this to the estimate of the Lyapunov exponent using Equation 10.13 (as in Example 10.10).
- 10.25 As with the harmonic oscillator, the Van der Pol system gives chaotic solutions if a forcing function is added. Create a new function called `vanderpol_chaos.m` that includes a forcing function. This is akin to adding a $k \cdot \sin(t)$ term, where k is the strength of the forcing function (as in Equation 10.3). Keeping $u = 5.0$, vary k until chaotic solutions are found. Prove that they are chaotic by finding the Lyapunov exponent using `max_1yp`.
- 10.26 If we apply the rules of the Koch curve to an equilateral triangle rather than a straight line, we arrive at a shape known as the Koch snowflake. Write a MATLAB program to draw a Koch snowflake by modifying the code in function `make_koch`.
- 10.27 Load the data in file `surrogate_data.mat`. This contains a signal in vector `x`. First determine an optimal embedding delay using the autocorrelation. Then create ten surrogate signals using `gauss_surrogate` and determine the correlation dimension of `x` and its ten surrogates. Use an embedding dimension of 2 for your analysis. Use a radius range as in Example 10.6. Then perform a rank analysis to determine if the correlated noise fits the hypothesis of being generated from a stochastic process.
- 10.28 Will the autocorrelation function of a measured time series and its surrogate be the same or different?
- 10.29 Modify Example 10.15 to use 100 surrogates and perform the Lyapunov exponent estimation for r values of 2, 3.5, 3.8, and 4 and an N of 1000 (note that for the surrogate data, you need to use 0.1 times the standard deviation of the radius parameter). Can the logistic map be approximated with an ARMA process for each of these values of r ?
- 10.30 Repeat Problem 10.28 using the Van der Pol system with $u = 5$. Do you expect the system to “pass” surrogate data testing? Why?

Nonlinearity Detection

Information-Based Methods

11.1 Information and Regularity

In Chapter 10, we discussed signal-processing methods that quantify a system's sensitivity to initial conditions and the geometric properties of the system attractor. In this chapter, we examine a different aspect of a nonlinear system: the information carried in its signals. The idea is that, at any given moment, a signal contains information, and this information can be sustained or lost over time. Analysis of information and the related concept of entropy are found in many disciplines; the definitions for these terms depend somewhat on the context. The common thread in the definition for entropy is that it represents some measurement of disorder. In dynamical systems analysis, entropy refers to the rate at which a signal loses or gains information, while in information theory, it is a measurement of the information in a signal. The information theory sense of entropy and information is more appropriate for stochastic signals, as it directly relates to probabilities, while the dynamical analysis sense is more appropriate for deterministic linear and nonlinear signals. To avoid ambiguities, we refer to entropy as a measurement of signal information and entropy rate as a measure of the rate of change of the entropy.

When we estimate the entropy or entropy rate of a signal, what we are attempting to do is make a statement about the regularity of the signal. Signals that are very regular (by that we mean they have some cyclic or asymptotic behavior) do not contain large amounts of entropy because the number of states these signals can have is limited. For example, a steady-state sine wave of constant amplitude and frequency contains no information. Nonlinear systems allow for more complicated dynamics, and their signals have many possible states and higher entropy. In linear systems, the entropy rate typically falls to zero because their signals reach equilibrium values or limit cycles. However, nonlinear systems can produce complicated dynamics, including chaos, so they can produce signals with nonzero entropies and entropy rates for long periods of time. We start the discussion of information analysis of signals by first describing what we mean by information and entropy.

The idea of describing a signal by its information content was first proposed by Claude Shannon in 1948. Shannon wanted to describe the average amount of information contained within a transmitted signal (or rather, the average amount of uncertainty within the signal). Since the amount of information contained within a signal is inversely proportional to the predictability of the signal, measurements of uncertainty or complexity in a signal seemed promising. Shannon borrowed terminology from the physical sciences and called signal uncertainty

Biosignal and Medical Image Processing

entropy. As in the physical sciences, entropy is a measure of the available states of a system. Since Shannon was interested in transmission of electronic signals, it was a natural choice to use the bit as the unit of an uncertainty measure, where the uncertainty is related to bits being switched on or off.

The end goal of information-based analysis is to be able to make some statement about the complexity of the signal. Signal complexity can be counterintuitive. For example, it can be shown with the methods described in this chapter that the complexity of Beethoven's 5th Symphony is far less than the static heard on a poorly tuned radio. This is because entropy measurements do not take meaning into account, merely unexpectedness.

11.1.1 Shannon's Entropy Formulation

Shannon reasoned that, on average, the entropy of a signal is related to the probability spectrum of the potential states of a system's output signal. As an example system, consider a coin biased 100% toward heads where the input is a coin flip. Here, the output signal could be a series of recordings of the result of the n th flip. The probability of landing on heads is 1.0 and the probability of landing on tails is 0.0. In this case, very few bits are needed to represent the signal. All you need to know to fully represent the signal is that the coin is biased completely toward heads. For a signal of any length or any point in time, the signal will always be heads. Conversely, more information is needed if the coin is a fair coin. We cannot say *a priori* on which side the coin will fall. One reason we might expect that the entropy of a completely biased coin is low is because there is no uncertainty in the system. Conversely, the entropy for fair coin is high as there is much uncertainty regarding subsequent states.

Shannon developed a mathematical formulation for determining the entropy of a system similar to the coin flip just discussed, which is given in Equation 11.1

$$H_x = -\sum_m p(x) \log_2 p(x) \quad (11.1)$$

where x is the signal, $p(x)$ is the probability distribution function of all possible states of the signal, and H_x is the entropy of the signal. Thus, the entropy of the system is a function of the probability of the states that the system can contain.

EXAMPLE 11.1

What is the entropy of a fair coin toss (i.e., there is equal probability of the coin landing on either side) and the entropy of a completely biased coin?

Solution

For the fair coin, the probability distribution is $p(x) = [0.5 \ 0.5]$, as there are two states, each with a 50% probability of occurrence. Using these values in Equation 11.1 gives entropy of 1.0 bit:

$$H_{\text{fair coin}} = -(0.5 \cdot \log_2(0.5) + 0.5 \cdot \log_2(0.5)) = 1 \text{ bit}$$

To determine the entropy of a completely biased coin: $p(x) = [1, 0]$:

$$H_{\text{biased coin}} = -(1.0 \cdot \log_2(1.0) + 0.0 \cdot \log_2(0.0)) = 0 \text{ bits}$$

Note that a completely biased coin yields zero entropy. This agrees with the intuitive arguments above. This also agrees with the typical usage of the word "bit" to quantitatively define one of two possible states.

While Equation 11.1 has a straightforward application to processes that have easily defined probability spectra such as stochastic processes (the coin flip is an example of a stochastic

process), it does not appear to be applicable to deterministic signals that cannot be fully characterized by a probability spectrum. In fact, each of the entropy analysis methods described in this chapter tries to resolve this issue in its own way.

The simplest way to determine a representative probability distribution for a time series is to use the values of the signal itself and estimate their frequency. This can be done by creating a histogram of the values of the system and normalizing by the total number of samples. Doing so destroys timing and long-term correlations, but is a perfectly valid operation. This method is straightforward to program and works reasonably well for many signals. As appropriate, this method can be modified by quantizing the data at a new bit rate (reducing the number of potential states). Depending on the number of discrete values a signal may have (i.e., the *bit depth*) and the quantization size, there can be a very large number of potential states (see Equation 1.6 in Chapter 1). Therefore, one typically will use a large bin size or reduce the bit depth of the signal when determining the frequency distribution. Constructing the probability distribution from a histogram works best if we assume that data samples are uncorrelated, that is that samples at time n are unrelated to previous samples. Later in the chapter, we describe other entropy analysis methods that do take correlation into account.

11.2 Mutual Information Function

In Example 11.1, we looked at a signal that was composed of a single measurement (a single coin flip). What if we assume that our information is carried over more than one signal? That is, rather than looking at singular measurements in time, we observe pairs of simultaneous measurements. A general example might be flipping a coin and rolling a die. For such situations, it is a natural question to ask if these events are correlated. One way to formulate this inquiry is to ask how much information from one event, the coin flip, could be ascertained from knowing the outcome of the second event, the die roll. This quantity is known as *mutual information*.

Before we introduce the relevant equations, we use two examples to help understand exactly how mutual information can be determined. Both examples suppose we are picking colored balls from a box. In the first example, the box contains only two balls (one red and one blue) and you are instructed to pick exactly two, removing a ball each time it is picked. After picking the first ball, there is only one ball left in the box and there is no choice but to pick that one. So while there is a 50% probability of picking either red or blue as the first ball, the probability of picking a specific second ball given the first is 100%. Here, the mutual information is equal to the information contained in the first choice of balls. In the second example, however, we return the first ball after it is picked. Since there are still two balls to choose from after the first pick, the first choice has no effect on the second choice. Information on the color of the first pick is of no help in predicting the color of the second pick, so the mutual information is 0. From these two examples, it is clear that there is high mutual information in joint events that are highly correlated, but no mutual information if there is zero correlation. What these examples tell us is that the mutual information between two events depends on their level of correlation and the information of the individual events themselves.

The quantities needed to determine the mutual information of two events include their joint probability distribution and the probability distribution of the individual events. Determining these properties is trivial for a simple stochastic system like a coin flip, but is more involved (though still relatively straightforward) for a measured time series. We shift our terminology from pairs of events to pairs of signals, keeping in mind that signals can be considered a series of events recorded in sequence. Let us consider a signal $x[n]$ of length N . Assume that this signal consists of stochastic, linearly independent events with m potential states, such that for any sample n , the probability of obtaining any particular value is $p(x_m)$, simplified to $p(x)$. This is the probability distribution function for any of the m potential states. Now, suppose we also have a

Biosignal and Medical Image Processing

similar signal $y[n]$, with k possible states with probability distribution function $p(y_k)$, again simplified to $p(y)$. Assume that the *joint distribution function* between the events is $p(x,y)$. This is the probability of events in $x[n]$ and $y[n]$ occurring simultaneously. The expression for information contained in the joint event is

$$H_{xy} = -\sum_{mk} p(x,y) \log_2 p(x,y) \quad (11.2)$$

Using these quantities, we can arrive at a description of the information contained within $x[n]$, if $y[n]$ is known. This is known as the mutual information, and is defined as the difference between the sum of information that can be estimated from both signals and the information contained within their joint occurrence. The mutual information of the two signals is

$$MI_{xy} = H_x + H_y - H_{xy} = \sum_{mk} p(x,y) \log_2 \frac{p(x,y)}{p(x)p(y)} \quad (11.3)$$

Here, we can see that the mutual information is a measure of the amount of information removed from the total information in either signal by the amount of information contained in their joint probability distribution.

Going back to the red and blue ball example, we can use Equation 11.3 to determine the mutual information in each case.

EXAMPLE 11.2

What is the mutual information for the system described above where there are two different colored balls in a box and they are picked at random without and with replacement?

Solution

For the first case, the information in the first ball retrieval is the same as that for the coin flip. The probability distribution functions, taking x to be the first retrieval and y to be the second retrieval, is

$$p(x) = [0.5, 0.5], \quad p(y) = [0.5, 0.5], \quad p(x,y) = [0.5, 0.5]$$

From Equation 11.3, we get

$$H_x = -0.5 * \log_2(0.5) - 0.5 * \log_2(0.5) = 1$$

$$H_y = -0.5 * \log_2(0.5) - 0.5 * \log_2(0.5) = 1$$

$$H_{xy} = -0.5 * \log_2(0.5) - 0.5 * \log_2(0.5) = 1$$

$$\text{So, } M_{xy} = H_x + H_y - H_{xy} = 1$$

This is in line with the reasoning above.

For the second case, where the first ball is returned after picking it, the first choice has no effect on the second and the events are independent. The probability distributions are:

$$p(x) = [0.5, 0.5]; \quad p(y) = [0.5, 0.5]; \quad p(x,y) = [0.25, 0.25, 0.25, 0.25]$$

The value for joint probability distribution arises because there are four equally likely outcome pairs: red-red, red-blue, blue-blue, and blue-red.

Again from Equation 11.3

$$H_x = -\log_2(0.5) - \log_2(0.5) = 1$$

$$H_y = -\log_2(0.5) - \log_2(0.5) = 1$$

$$H_{xy} = 4(-0.25 * \log_2(0.25)) = 2$$

$$\text{So, } M_{xy} = H_x + H_y - H_{xy} = 0$$

This is again congruent to what we reasoned.

As with the determination of entropy, to apply Equation 11.3 to a time series, we need to develop a probability distribution for the events in our system. However, once we have done this, we may apply Equation 11.3 to multiple-related events and quickly determine their mutual information. The key step in approximating the entropy and mutual information from a time series is to estimate a probability distribution that describes the time series in some way. The major difference in methods for estimating both entropy and entropy rate is the way in which this probability distribution is estimated. We have briefly mentioned in Section 11.1 that a histogram of data values can be used to develop a descriptive probability distribution. For example, if the signal is a coin flip, after 100 flips, we can make a histogram of two bins representing heads and tails. The count of each should be roughly 50–50; dividing by the total number of counts (i.e., 100), let us empirically determine the probability of flipping heads or tails to be roughly 0.5. However, observing Equation 11.3, we see that in order to determine the mutual information, we also need a method to estimate a probability distribution function for the joint probability of x and y values. To do this, we need to develop a 2-D histogram of the $\{x[n], y[n]\}$ outcome pairs; instead of counting the frequency of individual values from x or y , we count pairs of values of x and y . To reduce the effects of noise and to make counting more manageable, we quantize the values into bins that are often smaller than the signal quantization value. Since these bins are now 2-D and have x and y components, we refer to them as cells. A count in the cell means that a pair of x and y values both contain values that fall into the range of that cell. Note that the range of the cell values can be different for x than for y , although typically it would be similar assuming similar signal amplitudes. In Example 11.3, we use this method to determine the joint distribution for two signals: uniform and Gaussian distributed noise.

You may have also noticed that to determine the mutual information using Equation 11.3, we also need H_y and H_x . However, these probabilities can be obtained directly from the joint probability histogram, H_{xy} , by summing over the rows and the columns. The resulting individual signal distributions are termed the *marginal distributions* for x and y .

EXAMPLE 11.3

Use routine `hist2` (described below) to find the joint distribution of two noise signals. Both signals contain 10,000 samples. The first signal is Gaussian noise and the second signal is evenly distributed noise from 0 to 1. The histogram should range between the minimum and maximum value of the signals and contain 50 bins. Determine the marginal distribution of each signal by taking the sum of the joint histogram across its rows and columns. Plot these histograms using bar plots.

Solution

Use MATLAB's `rand` and `randn` functions to create vectors of evenly distributed and Gaussianly distributed noise. Use the routine `hist2` with inputs x , y and 50 for the number of bins to get

Biosignal and Medical Image Processing

the joint histogram function of x and y . Note that to convert from the joint histogram to the joint probability distribution, one simply scales the histogram values by the total number of $[x,y]$ pairs. In this example, that would be equivalent to dividing by 10,000. You can verify this amount by summing across the rows and columns of `hist` and observing this value to be 10,000.

The routines used to compute the histogram estimations and the associated mutual information are `hist2` and `inform2`. Function `hist2` has the form

```
hist = hist2(x,y,nu_bin);
```

Inputs x and y are the time series to be analyzed and must be of the same length. Input `nu_bin` is the number of bins in the 2-D histogram used to estimate for H_{xy} (Equation 11.2). If no value for `nu_bin` is given, the default value used is the total number of unique values of the set made of x and y .

The included function `hist2` as shown below:

```
function hist = hist2(x,y,nu_bin)

if nargin < 3
    nu_bin = length(unique([x,y]));           % Default bins is number
end                                         % of unique elements (IE no quantizing)

ln = length(x);
hist = zeros(nu_bin,nu_bin); % Zero array
%
x_range = max(x) - min(x);                 % x signal amplitude
y_range = max(y) - min(y);                 % y signal amplitude
% Quantize data
quant_x = round((nu_bin-1)*(x-min(x))/x_range)+1; % Quantize x
quant_y = round((nu_bin-1)*(y-min(y))/y_range)+1; % Quantize y
% Bin data
for n=1:ln
    hist(quant_x(n),quant_y(n)) = hist(quant_x(n),quant_y(n))+1;
end
```

Use the MATLAB function `pcolor` in conjunction with `shading('interp')` to view the joint histogram of x and y . To get the marginal distribution of x , sum along the rows. To get the marginal distribution of y , sum along the columns. Use MATLAB's `bar` to plot the marginal histograms.

```
% Example 11.3
% Find the joint distribution of Gaussian and evenly distributed
% noise signal
%
x=randn(1,10000);    % Create signals, Gaussian
y=rand(1,10000);      % and uniform noise
%
H=hist2(x,y,50]); % Use hist2 to get 2D histogram
%
colormap gray         % Set the colormap to grayscale
subplot(1,2,1)
    pcolor(-H);          % Use negative H so that
    shading('interp');    % higher values are darker in plot
    xlabel('Joint X Y Histogram')
subplot(2,2,2)
    bar(sum(H));          %
    xlabel('Histogram of Y')
```

```

subplot(2,2,4)
bar(sum(H,2))
xlabel('Histogram of X')

```

Analysis

We use the commands `colormap` to set the image to black and white, and `pcolor` to view the matrix of histogram values as an image. The latter is a MATLAB routine that displays matrix data as a 2-D image with values coded in color or, as in this case, grayscale intensity. We negate the values of histogram H so that the image yields darker colors for higher intensity values. This is just a visualization preference.

Results

The resulting histograms are shown in Figure 11.1. The joint distribution shows an even intensity level horizontally, across the columns, but a Gaussian distribution vertically, across the rows (Figure 11.1a). This is reflected in the marginal distributions, which show the expected flat and bell-shaped curves of uniform and Gaussian noise in Figures 11.1b and 11.1c, respectively.

The included routine `mutual` can be used to determine the mutual information between two signals.

```

function H_xy=mutual(hist_xy)
% Function to compute mutual information from 2-D histogram
%
% Normalize histogram and make sure no zeros (i.e.,add eps)
p_xy=(hist_xy)/sum(sum(hist_xy))+eps;
p_y=sum(p_xy);           % Find marginal distribution of y
p_x=sum(p_xy,2);         % Find marginal distribution of x
H_xy=sum(sum(p_xy.*log2(p_xy./(p_x*p_y))));    % Eq. 11.3

```

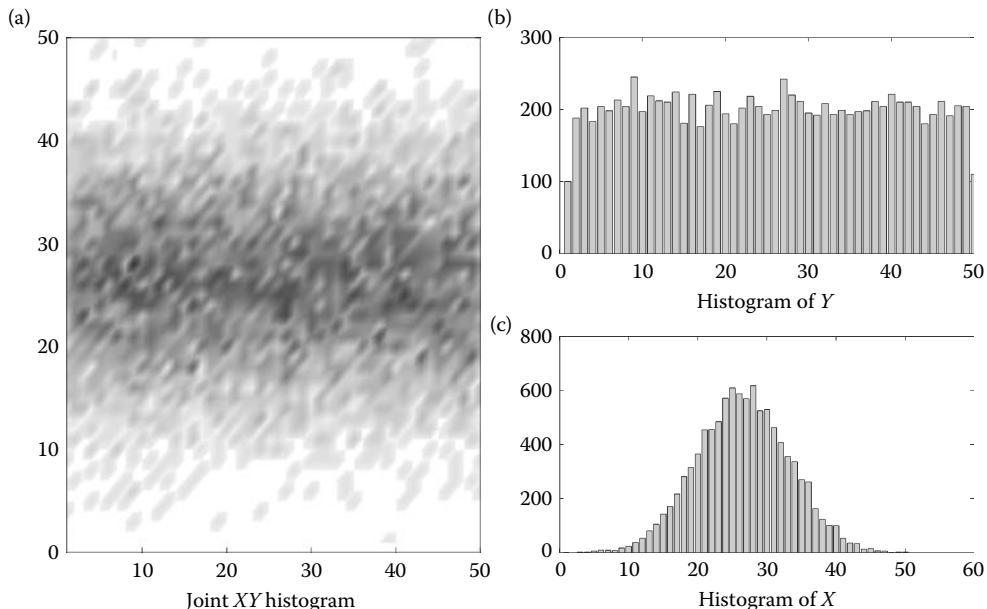


Figure 11.1 (a) The 2-D joint histogram of Gaussian and uniformly distributed noise. (b) The marginal histogram of the uniformly distributed noise. (c) The marginal histogram of the Gaussian noise. These histogram sums and the joint distribution can be used to estimate the mutual information between x and y as given by Equation 11.3.

Biosignal and Medical Image Processing

The routine is a straightforward implementation of Equation 11.3. The input `hist_xy` is the histogram values of signals x and y . The marginal distributions of x and y are found by summing across the rows and columns as in Example 11.3. One small caveat is the need to add the smallest floating point value represented by a MATLAB double precision variable (i.e., `eps`) to the histogram to eliminate any zeros that would produce errors when taking the log.

Using `mutual`, we can determine the mutual information between the signals of Example 11.3. In Example 11.4, we also find the mutual information between a 50-Hz sine wave and the same sine wave embedded in 10-dB noise.

EXAMPLE 11.4

Use function `mutual` with the signals from Example 11.3 to determine the mutual information between a Gaussian and a uniform random noise signal. Also compare the mutual information between a 50-Hz sine wave with the same sine wave embedded in a 10-dB noise.

Solution

Generate the noise signals using `rand` and `randn` and the noisy sinusoid using `sig_noise`. Use `hist2` to get the 2-D histogram of the signal pairs. Then use function `mutual` to get the mutual information between the two.

```
% Ex 11.4 Determine the mutual information of two random vectors.  
%  
x = randn(1,10000); % Gaussian and IID noise  
y = rand(1,10000);  
%  
hist_xy = hist2(x,y); % Use hist2 to get the 2D histogram  
H_xy = mutual(hist_xy); % Use mutual to get mutual information  
clear x y  
[x, y] = sig_noise(50, 10, 1000); % Generate sinusoids  
hist_xy = hist2(x,y); % Use hist2 to get the 2D histogram  
H_xy = mutual(hist_xy); % Use mutual to get mutual information
```

Results

This code reports the mutual information between the Gaussian and uniform noise signals to be ~6 bits. Between the noisy and clean sine waves, the mutual information is about 3 bits (of course, due to randomness, your values when running the example will vary). The results of this example appear counterintuitive: why is the mutual information between the clean sine wave and the sine wave with noise added less than the mutual information between the Gaussian and uniform noise signals? This is because sine waves in general contain very little entropy. For the case of the sine wave with and without noise, the values H_x , H_y , and H_{xy} are all small, so mutual information ($H_x + H_y - H_{xy}$) is also small. In the case of Gaussian and uniformly distributed noise, H_x and H_y are both very large relative to H_{xy} .

11.2.1 Automutual Information Function

Finding mutual information is a useful method for evaluating similarity between two signals, but we are not limited to separate signals; we can also investigate the relationship between a signal and its delayed version. Intuitively, we would expect that mutual information estimates will decrease as delay increases. As with correlation, increasing the distance between segments decreases the association between them. When the delay is short, the signals are close together in time, and thus features with short time scales dominate. For very long delays, features with long time scale information can be probed. Estimating mutual information over a range of delay

11.2 Mutual Information Function

values is termed the *automutual information function*, or *AMIF*. The AMIF is a nonlinear analog to autocorrelation and we adopt the autocorrelation vocabulary referring to delays as lags. As with autocorrelation, the maximum mutual information occurs when the lag is zero since the signal exactly predicts itself.

The main feature we probe through the AMIF is the information loss rate; that is, over what time frame (i.e., how many lags) the signal can predict the information in its future values. A signal's ability to predict future values of itself is indicative of its long-term memory. The AR model that would be required to represent a signal with long-term memory would need many coefficients to accurately represent the signal's extensive memory: its self-correlations that extend over many samples. Because mutual information is related to predictability, mutual information decreases as a function of scale for chaotic systems and noisy signals, but remains constant for cyclic signals.

The AMIF can be characterized by several parameters. One such parameter is the lag at which the function reaches a value of $1/e$, a parameter referred to as lag_τ . (This value assumes the mutual information function, like autocorrelation, is normalized to have a value of 1.0 at zero lag.) Two additional parameters are the lag and value of the first local maximum, referred to as lag_{PD} and PD , respectively. These relate to the rate of information loss and describe the time frame over which the signal remains similar: the time frame and degree to which the signal can predict its future self. The lags at which other peaks occur tell us something about the relevant length scales within the signal. The AMIF can be helpful in determining values of delay used for delay embedding (Chapter 10).

Here, we develop a routine to derive the AMIF and the parameters described above. This function is named `amif` and is called as

```
[AMIF_curve, lag_tau, PD, lag_PD] = amif(x,max_lags,nu_bin);
```

where `x` is the signal to be analyzed, `max_lags` is the maximum number of lags for analysis (default = half the length of the signal), and `nu_bin` is the number of bins to be used to develop the joint and marginal probability distributions needed to compute mutual information. The outputs are the `AMIF_curve`, the AMIF curve from lags 0 to `max_lags`; `lag_tau`, the lag at which the AMIF curve reaches a value close to $1/e$; `PD`, the value at the first local maximum; and `lag_PD`, the lag of the first local maximum.

This function effectively has two sections. In the first section, the routine `inform2` is used to determine the mutual information (MI) between the signal and its delayed version for every lag between 0 and `max_lags`. In the second section, the AMIF is analyzed to determine the parameters `lag_tau`, `PD`, and `lag_PD`. When determining the lag of the local maximum, we assume that the AMIF for zero lag has the highest possible MI value. This comes from definition of mutual information; at zero lag, the two signals are identical and one signal completely describes the other. In searching for the local maximum, we employ a four-sample local average as this reduces spurious effects due to noise in the signal.

```
function [AMIF_curve,lag_tau, PD, lag_PD] = amif(x,max_lags,nu_bin)
% Routine to calculate the auto-mutual information function
% and determine descriptive parameters
%
N=length(x); % Get length of vectors
if nargin<2
    max_lags=floor(N/2); % Default number of lags is half
end
%
x=x(:); % Make a column vector
x=x'; % Enforce row vector
```

Biosignal and Medical Image Processing

```
% Construct AMIF curve
AMIF_curve = zeros(1,max_lags); % Initialize AMIF curve
x1 = x; % Initial and delayed x
del_x = x;
AMIF_curve(1) = inform2(x,del_x,nu_bin); % Mutual info at lag 0
for k = 2:(max_lags+1)
    x1(end) = [];
    del_x(1) = [];
    AMIF_curve(k) = inform2(x1,del_x,nu_bin); % Get mutual information
                                                % between x1 and del_x
end
AMIF_curve = AMIF_curve - min(AMIF_curve);
AMIF_curve = AMIF_curve/(AMIF_curve(1)); % Normalize to 1.0
% Now find peaks
lag_tau = find(AMIF_curve < 0.37,1)-1; % Find the lag at which AIF
                                            % is less than 0.37. Adjust
% for nonzero indexing
ii = 0; % Initialize ii to 0;
while ii < (length(AMIF_curve)-10);
    ii = ii + 1;
    % Find first local minimum. Use 4 point average
    if sum(AMIF_curve(ii:(ii+5))) < sum(AMIF_curve((ii+6):(ii+10)))
        break;
    end
end
%
[PD,i_PD] = max(AMIF_curve(ii:end)); % Find first local max
lag_PD = i_PD + ii - 2; % Subtract two since indices
                         % start at 1 and lag starts at 0
```

In the next example, we determine the AMIF of correlated noise and Gaussian noise. For noise that has some correlation, we expect that the MI between the series and its delayed version will have some maximum value and then drop over the next few lags. However, we expect the mutual information for Gaussian noise to drop quickly and remain low for all nonzero lags.

EXAMPLE 11.5

Create a correlated noise vector of 1000 samples by taking the cumulative sum of a Gaussianly distributed random noise signal. Use this signal with function AMIF to estimate the automutual function curve and the properties of lag , lag_{PD} , and PD . Use a maximum lag of 300 samples and 10 bins for the histogram. Repeat with a 1000-sample vector of Gaussian noise. Plot the AMIF over the 300 lags. Also compare the numerical sum of the AMIF for each noise type.

Solution

Create a 1000-sample noise signal using `randn` and then use the MATLAB function `cumsum` to construct a correlated noise signal. Use 10 bins to construct the histogram. Ten bins were chosen because correlated noise is nonstationary and can have a wide range of values. Using a larger number of bins might result in empty or sparsely populated bins; we want to avoid sparsely populated bins as these can produce numerical errors. Use `amif` to generate the AMIF curve and determine parameters lag , lag_{PD} , and PD . Compare the AMIF sum for each noise type by taking the sum over `amif_curve`.

```
% Ex 11.5 Evaluation of AMIF in correlated and uncorrelated noise.
%
rng(2); % Set the random number generator so everyone gets the same values
x = cumsum(randn(1,1000)); % Create a correlated noise vector
max_lags = 300; % Set the maximum number of lags
nu_bin = 10; % and histogram bin size
%
% Perform the AMIF calculation
[AMIF_curve,lag_tau, PD, lag_PD] = amif(x,max_lags,nu_bin);
AMIF_sum_cornoise = sum(AMIF_curve); % Get the sum
%
% Repeat for a Gaussian random signal
x = (randn(1,1000)); % Create a uncorrelated noise vector
%
% Perform the AMIF calculation
[AMIF_curve,lag_tau, PD, lag_PD] = amif(x,max_lags,nu_bin);
AMIF_sum_gauss = sum(AMIF_curve); % Get the sum
%.....label and Plot...
```

RESULT

Figure 11.2 shows the automutual function for Gaussian and correlated noise. The curves are similar to what you would get from the autocorrelation, in fact the automutual information can be thought of as the nonlinear analog to the autocorrelation. As we would expect, correlated noise has a much longer roll-off period compared to Gaussian noise. Mutual information for Gaussian noise falls essentially to zero after 1 lag. Recall that in Chapter 2, we saw that the autocorrelation of Gaussian noise also falls very quickly, but not instantly with nonzero lags. Results of the parameters for the AMIF curve for correlated noise and Gaussian noise are given in Table 11.1.

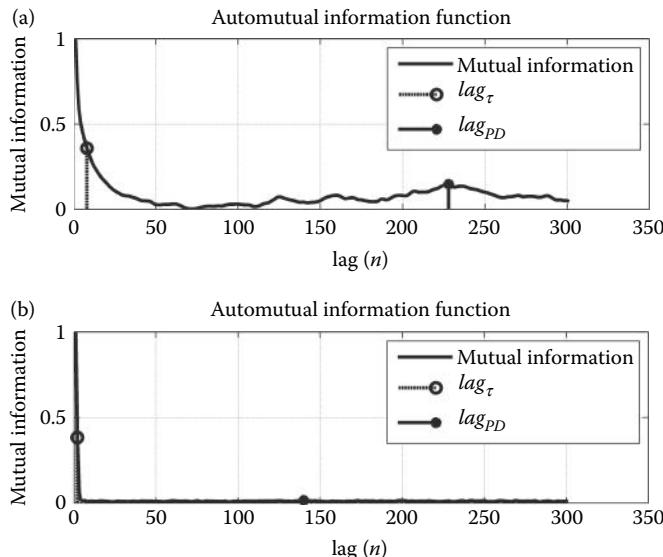


Figure 11.2 An example of automutual information curve for (a) integrated noise and (b) Gaussian noise. Correlated noise has a lag_{τ} of 12 while uncorrelated noise has a lag_{PD} of 239 and a PD of 0.142. Gaussian noise has a lag_{τ} of 2 but no meaningful values for lag_{PD} or PD, because the AMIF is near zero for all lags greater than 2.

Table 11.1 AMIF Parameters for Gaussian and Correlated Noise

Noise Type	lag_{τ}	lag_{PD}	PD	AMIF Sum (Bits)
Correlated	12	239	0.142	4
Gaussian	2	299 ^a	0.023 ^a	37

^a Inaccurate values because the AMIF curve of the Gaussian noise has no peaks for nonzero lags.

Since correlated noise by definition contains correlations over many samples, we expect the values of lag_{τ} and lag_{PD} for correlated noise to be greater than their respective values for Gaussian noise. This is true for lag_{τ} but not for lag_{PD} . This is because for Gaussian signals, the values determined for peaks at nonzero lags are not meaningful since the AMIF has no peaks for these lags.

As stated above, AMIF is the nonlinear analog to the autocorrelation. In the next example, we investigate the difference between these two curves using one of our favorite nonlinear signals, the Lorenz attractor.

EXAMPLE 11.6

Perform an automutual information analysis on the x -dimension of the Lorenz attractor and compare it to the autocorrelation function. For the AMIF, use a bin size of 10. For both mutual information and autocorrelation, use 300 lags.

Solution

The x -dimension of the Lorenz attractor is determined using `ode45` as in Example 10.5. The automutual information is determined using function `amif`. To facilitate the comparison between the correlation and the mutual information function, parameters used in describing the AMIFs are determined for the autocorrelation function using the routine `xcorr_amif` described below. This function call is

```
[xcorr_curve, lag_tau, lag_PD, PD] = xcorr_amif (x,max_lags);
```

where `xcorr_curve` is the autocorrelation function of x , and the other input and outputs are as defined for the function `amif`. The procedures for finding `lag_tau`, `lag_PD`, and `PD` are identical to that used in `amif`, except that instead of searching for the local maximum using a four-sample average, we use a two-sample average. Finding a local maximum in the presence of noise can be problematic, but the procedure described in the previous two examples works well on most occasions.

```
function [xcorr_curve,lag_tau,lag_PD,PD]=xcorr_amif(x,max_lags)
% Routine to calculate the autocorrelation function
% and determine descriptive parameters
%
% Get the normalized auto correlation
xcorr_curve=xcorr(x,max_lags,'coeff');
half=floor(length(xcorr_curve)/2);
xcorr_curve=xcorr_curve(half:end); % Remove negative lags
%
% Find the lag at which AIF is less than 0.37.
lag_tau=find(xcorr_curve<0.37,1)-1; % Adjust for index
%
ii=0; % Initialize ii to 0;
while ii<(length(xcorr_curve)-10);
    ii=ii+1;
```

```
% Find first local minimum
if sum(xcorr_curve(ii:(ii+1))) < sum(xcorr_curve((ii+2):(ii+3)))
    break;
end
[PD, i_PD] = max(xcorr_curve(ii:end)); % Find first local maximum
lag_PD = i_PD + ii - 2; % Adjust for index
```

The execution and plotting of the analysis is identical to that of Example 11.5 except for the development of the test signal. The modified code is

```
x=sol(:,1); % Get x solution from Lorenz system
% Set the maximum number of lags and histogram size bin
max_lags=300;
nu_bin=10;
% Perform the AIF calculation
[AMIF_curve,lag_tau, PD, lag_PD]=amif(x,max_lags,nu_bin);
.....labels and plotting.....
%
[xcorr_curve,lag_tau,lag_PD,PD]= xcorr_amif(x,max_lags); % Get xcorr
.....label and plot.....
```

Results

The results of Example 11.6 are shown in Figure 11.3. The AMIF and autocorrelation functions are similar, but the AMIF function shows more minima and maxima than the autocorrelation, which only shows two. The greater detail in the AMIF curve is because the automutual

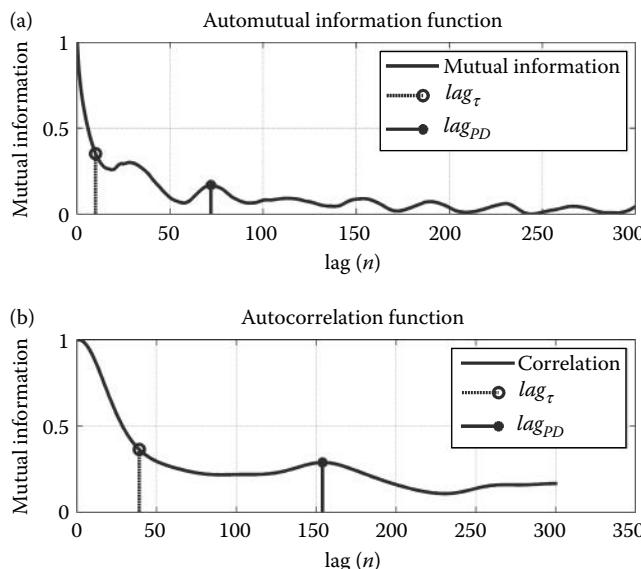


Figure 11.3 The automutual and autocorrelation functions for the x dimension of the Lorenz system. The automutual information function (a) shows many undulating minima and maxima, while the autocorrelation (b) shows only two. This is because the automutual information is sensitive to nonlinear behavior and reflects the information that is present at many different time scales. The automutual information has a lag_{PD} of 72 compared with 154 for the cross-correlation. The lag_τ for the automutual information and autocorrelation functions was 10 and 39, respectively.

information is sensitive to nonlinear behavior, which in this case is the result of information being present over many relevant scales. Each maximum represents a scale at which the signal has correlations with itself. The AMIF shows a number of the long- and short-term correlations in the Lorenz system not seen in the autocorrelation function. This is evident from the lag_{PD} measurements. The automutual information has a lag_{PD} of 72 compared with 154 for the autocorrelation. Interestingly, autocorrelation has a slower roll-off as shown by comparing lag_τ for the two functions. It is shorter for the automutual information, 10, than for the autocorrelation, which has a value of 39 lags. We mentioned in Chapter 10 that the automutual information is sometimes used to estimate the best value for delay parameter τ . A delay of 10 is close to the delay of 6 used in Example 10.7. In that example, the delay was found by trial and error, but here we show a more objective estimation using the mutual information.

11.3 Spectral Entropy

Spectral entropy is the frequency-domain analog of the Shannon time-domain equation. The Shannon equation (Equation 11.1) uses the probability distribution of the time function, which usually must be determined empirically from a histogram of the time function. As noted at the end of Section 11.1.1, the histogram is a good estimate of the probability distribution if time samples are independent, which is not always the case for biological signals. With spectral entropy, the frequency-domain representation of a signal is used instead of the time-domain histogram; specifically, the power spectrum replaces the probability distribution function in the Shannon equation. Because harmonic decomposition employs sinusoids that are orthogonal (see Section 2.3.1.2), the frequency components in the power spectrum are linearly independent. This is true even if the spectrum is derived from signals with nonlinear correlations. It is therefore more suitable for estimating the entropy of deterministic signals, but it comes at a cost. Because the power spectrum is determined using a linear property, and phase information is discarded, this method is a linear technique and is not sensitive to signal nonlinearities.

Spectral entropy is a measurement of how broadly the power in the signal is distributed across the available frequencies (frequencies less than the Nyquist limit). A broadband signal has more energy distributed throughout its power spectrum and thus has more entropy than a narrowband signal. Therefore, a signal composed of many narrowband frequencies, or containing broadband noise, is assessed as containing more spectral entropy than a signal with only a few narrowband frequencies. Since efficient methods for computing the power spectra exist, spectral entropy can be determined with excellent computational efficiency and speed.

An algorithm for estimating spectral entropy is a modification of the Shannon equation (Equation 11.1) using the normalized power spectrum in place of the probability distribution function. The normalized power spectrum is defined as

$$Q(f) = \frac{PS(f)}{\sum PS(f)} \quad (11.4)$$

and spectral entropy is determined using

$$H(f) = \frac{Q(f)\log\left(\frac{1}{Q(f)}\right)}{\log(N)} \quad (11.5)$$

$$E_m = \sum_f H(f) \quad (11.6)$$

where entropy E_m is the spectral entropy and is estimated in several straightforward steps. First, the power spectrum (PS) is estimated using standard methods described in Chapter 3. Next, the PS is normalized by the sum of the spectral components, yielding $Q(f)$ (Equation 11.4). $Q(f)$ is then modified with the Shannon transformation and divided by the log of the number of frequency components, N , yielding $H(f)$ (Equation 11.5). In the included function specen, we use $N/2$ instead of N to eliminate redundant frequencies. Comparing Equation 11.5 to the traditional Shannon equation (Equation 11.1) shows that spectral entropy uses the normalized frequency distribution instead of the probability distribution. The entropy is taken as the sum (integration) over $H(f)$ (Equation 11.6). Each Shannon-transformed spectral component contributes to entropy. As shown in the next example, the application of Equations 11.4 through 11.6 to find the spectral entropy of a sine wave results in a value of 0. The spectral entropy of a noisy sine wave is also determined.

EXAMPLE 11.7

Apply Equations 11.4 through 11.6 to find the spectral entropy of a sine wave and a noisy sine wave. The sine wave should have a frequency of 100 Hz ($f_s = 10$ kHz, $N = 1000$). The noisy sine wave should be created by adding correlated noise to the sine wave. Then elucidate the spectral entropy algorithm step by step, for each signal plot: the original signal, the normalized power spectrum (Equation 11.4), and the power spectrum after the Shannon transformation (i.e., the output of Equation 11.5 as a function of Q). Then apply the summation (Equation 11.6) to get the entropy.

Solution

The sine wave is generated and correlated noise added to construct the two signals. The analysis is a direct application of Equations 11.4 through 11.6 applied to each of the two signals using a loop.

```
% Example 11.7 Application of spectral entropy to sinusoid
%
f1=100; % Signal frequency
fs=10000; % Sample frequency
N=1000; % Number of points
t=(0:N-1)/fs; % Make time vector
f=(0:N/2-1)/N*fs; % Generate frequency vector
%
% Generate two test signals
x(1,:)=sin(2*pi*f1*t); % Generate sine wave and
x(2,:)=x(1,:)+cumsum(randn(1,N)); % add correlated noise
for k=1:2 % Repeat analysis for each signal
%
% Compute normalized power spectrum
Psx=abs(fft(x(k,:))).^2; % Get power spectrum
%
Psx=psx(1:(N/2)); % Valid frequencies
Q=psx/sum(psx); % Normalize as in Eq. 11.4
%
frq=frq(1:(N/2)); % Frequency vector for plotting
%
H=Q.*log2(1./(Q))/log2(N/2); % Apply Eq. 11.5
H(isnan(H))=0; % Remove potential Nans caused
% by the log operation
entropy(k)=sum(H); % Eq. 11.6
....labels and plot.....
end
```

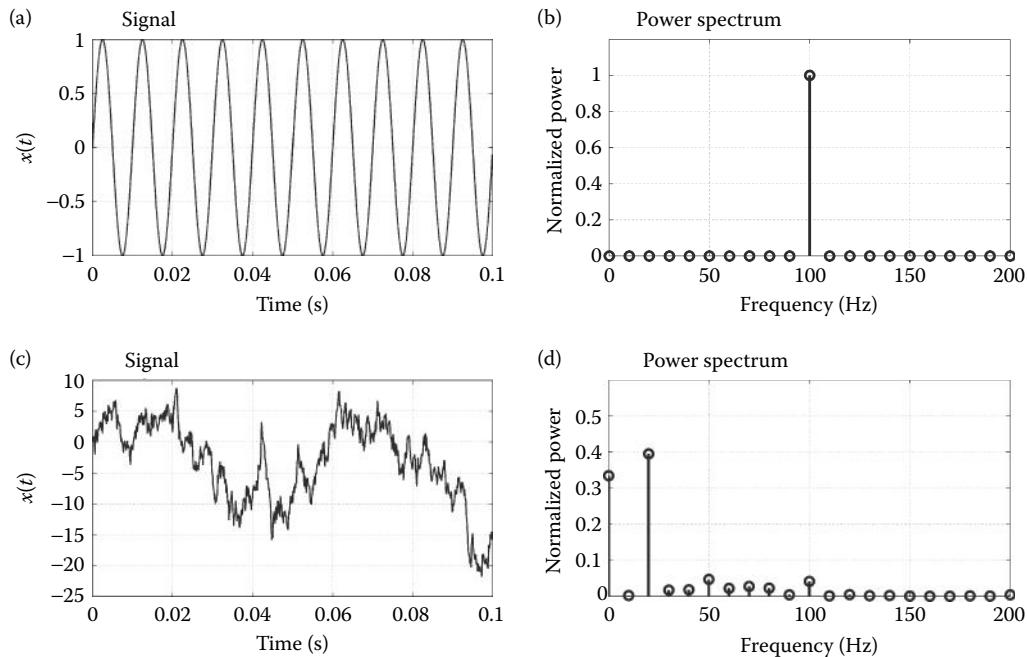


Figure 11.4 (a) 100-Hz sine wave. (b) Normalized PS showing a single component at 100 Hz, as in Equation 11.4. (c) The same sine wave with correlated noise. (d) Normalized PS showing energy at multiple frequencies.

Analysis

Results of Equation 11.4 on a noisy and a noise-free sine wave are shown in Figure 11.4. The sine wave has the expected PS single component at 100 Hz normalized to 1.0 (Figure 11.4a). The Shannon transformation is shown in Figure 11.5. Transforming these with the Shannon transformation (Equation 11.5) and summing gives an experimentally derived value of $E_m = 6.02 \times 10^{-4}$, very close to the expected value of zero. For a noisy sine wave, the process is similar, but there are multiple components in the PS (Figure 11.4c). Here, components of the PS are not transformed to zero and a larger value for entropy is measured. Note that while the transformed components are small, they sum to a value of $E_m = 0.32$.

The code given in Example 11.7 has been encapsulated into a function `specen` that can be used to determine the spectral entropy of a signal.

```
entropy = specen(y,NF); % Calculate spectral entropy
```

where `y` is the signal, `NF` is the number of frequency components used in the FFT (default is `length(y)`), and `entropy` is the estimated spectral entropy. Spectral entropy is a fast and efficient method that is suitable for many signals but, again, is not sensitive to the nonlinear properties of a signal. A simple way to demonstrate this is through the use of surrogate data as described in Section 10.6.

EXAMPLE 11.8

Create a nonlinear signal from the Lorenz system and determine the spectral entropy using `specen`. Compare this to the spectral entropy of a surrogate of the same signal, which will not have nonlinear properties as demonstrated in Section 10.6.

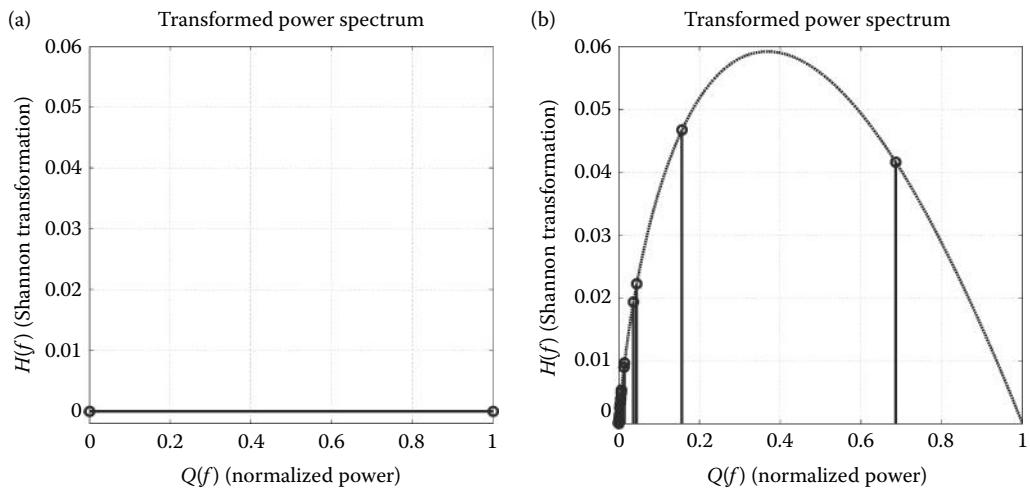


Figure 11.5 (a) For the sine wave without noise, the Shannon transform maps both values of $Q(f)$, 1 and 0, to zero, which sums to an entropy value of zero. (b) For the noisy signal, several components map to nonzero values. Summing over $H(f)$ (Equation 11.6) leads to nonzero entropy of 0.32.

Solution

The spectral entropy of the x output of the nonlinear Lorenz system is determined using specen. A surrogate of this signal is constructed using the approach and code developed in Section 10.6. The spectral entropy of the surrogate is constructed again using specen and the two entropies are compared.

```
% Example 11.8 Comparison of spectral entropy of a nonlinear
% signal and one of its surrogates
%
% .....calculate solution to the Lorenz system.....
x=sol(:,1); % The signal is the x dimension of the Lorenz system.
%
% Use 1024 frequency components, this is sufficient for the FFT
Lorenz_entropy=specen(x); % Calculate spectral entropy
%
% Repeat with surrogate
surrogate=gauss_surrogate(x); % From Section 10.6
surrogate_entropy=specen(surrogate); % Spectral entropy
```

Results

The spectral entropy for both is the same, $E_m = 0.534$, even though one signal is linear and the other is nonlinear. Note that gauss _ surrogate uses the length of the signal as the number of FFT components to create the surrogate data, and we need to use the same number of components again for the FFT when we compute the spectral entropy. Otherwise the power spectra of the surrogate and the original will not be equal.

Spectral entropy, being based on the efficient FFT, is ideal for situations in which speedy processing is a necessity. It is used commonly for real-time EEG monitoring for a quick estimate of depth of consciousness during anesthesia. The concept is that reductions in EEG entropy indicate a deeper state of unconsciousness. This is a situation in which quick and robust results are a priority. However, the results from Example 11.8 show that spectral entropy cannot be used

to detect nonlinearity. For this, we need to look at more sophisticated measurements of entropy. Two such examples are approximate entropy and its successor sample entropy.

11.4 Phase-Space-Based Entropy Methods

11.4.1 Approximate Entropy

The equations for spectral entropy and the Shannon entropy are easy to apply, but do not take into account the dynamics of a system, a critical component of nonlinear analysis. An early, popular method for determining entropy rate of a time series that does take system dynamics into account was developed by Grassberger and Procaccia (1983b). This method was based on the Sinai–Kolmogorov entropy (KSE) and uses the correlation integral to formulate an entropy calculation (see Equation 10.21 and Hilborn [2000] for more on KSE). This approach probes for repeated patterns within a time series. However, since these calculations are based on the correlation sum, they require a large segment of noise-free data, which often cannot be obtained from biological systems. KSE is also difficult to estimate in real-world signals because KSE involves the application of a triple limit that can only be approximated in measured data. Therefore, modifications to KSE were needed to develop a method more suitable for real-world signals. These modifications required relaxing the need for strict limits in the triple integral. Before we explain the method, we first review the triple integral and the limits in question.

The formulation for KS entropy is as follows. First, define the average log correlation sum as

$$\Phi^m(R) = \frac{1}{N - m + 1} \sum_{n=1}^{N-m+1} \log C(R) \quad (11.7)$$

where C , N , m , R , and n are as defined as in Equation 10.21. Specifically, $C(R)$ is the correlation sum as a function of radius R (R is a range of many radius values r), N is the total number of points n along the attractor, and m is the dimension. Equation 11.7 therefore describes the average log correlation sum for all points along the attractor of a system as a function of R . Then KSE is defined as

$$\text{KSE} = \lim_{R \rightarrow 0} \lim_{m \rightarrow \infty} \lim_{N \rightarrow \infty} [\Phi^m(R) - \Phi^{m+1}(R)] \quad (11.8)$$

Taking these limits gives us an expression for how often the trajectory reaches a point along the attractor in an infinitely small space. How does this relate to entropy? If the attractor is a limit cycle or a stable node, then trajectories visit the same space many times. But we have already shown that such signals have very low entropy (e.g., the sine wave in Example 11.6). More complicated chaotic signals do not visit areas of the attractor with the same regularity. (Chaotic trajectories do visit the same regions of space but not exactly as with a limit cycle and, unlike with a limit cycle, chaotic trajectories diverge after a period of time.)

Because real data are finite and bounded, the limits described in Equation 11.8 cannot be evaluated without approximation. The limit of N and m cannot approach infinity because of the limited number of sample points (infinite samples require a large hard drive) and R cannot go to 0 because of the presence of noise and the limited number of points. *Approximate entropy (ApEn)*, introduced by Pincus (1994), relaxes the requirements of these limits by only examining samples at fixed values for N , R , and m . Such approximations result in a trade-off. The relaxed limit integral can be applied to time-series data that may contain noise or be of short length, but the approximate approach can no longer be used to determine if a system contains signatures of deterministic chaos. Therefore, if you are interested in determining

the presence of chaos, you should use approaches described in Chapter 10 such as surrogate data analysis.

Approximate entropy quantifies the tendency of signal trajectories to repeat. If there are many repeating segments, the signal is more predictable and has less entropy, while if the signal has few repeating segments, it is less predictable and contains more entropy. Remember that entropy is not a measure of the information in the signal; it is a measure of the uncertainty, a subtle but important difference.

Before developing ApEn, we first note that currently ApEn is considered out of date. Its usefulness is as an introduction to an updated version known as *sample entropy* (*SampEn*). The process of determining ApEn can be understood in several equivalent ways. If we consider the phase-space, then ApEn can be determined by comparing the frequency of finding nearest neighbors in an m th dimension and $m + 1$ dimension. In fact, the procedure is very similar to the approach used by the false nearest-neighbor technique of Section 10.3.1. However, there is a simpler way to approach the ApEn method based on the time signal in which we compare a short segment of a signal with a time-shifted version of the signal. We discuss this method, as opposed to the phase-space interpretation, in detail.

The idea behind ApEn is that the signal regularity can be measured by determining how often a short series of a signal is repeated. To estimate this probability, we choose a sequence of samples from the signal and use that as our test series.

This amounts to a routine in which a test signal is composed, a comparison segment is chosen, a mathematical operation is performed as a similarity test, and then a new comparison segment is chosen by shifting the signal. The length of the test segment is given by m . For example, if $m = 2$, we begin the procedure by first determining the series that is to be the basis of comparison. The first such series is samples $\{x[1], x[2]\}$. We refer to the series that serves as the basis of comparison as the template. We choose a test series to compare with the template coming from $x[n]$. We refer to the test series as the window. If $m = 2$, the window is two samples long and is given by $\{x[n], x[n + 1]\}$, where n ranges from $n = 1$ to $n = m - \tau + 1$. We determine the Euclidean norm between the template and window, and if this value is less than r , we count it as a match. We then acquire a new window by shifting the window by a sample. The norm is again taken and a potential match is counted. This shifting and matching procedure continues until we have shifted through the entire length of the signal. At this point, a new template is chosen by shifting the template by one sample of $x[n]$ and the procedure is repeated again. This entire sequence of template matching, window shifting, and template shifting is then repeated until the template has shifted through the entire length of the signal. The count of all the matches for every n th template is recorded as B_n . To get A_n , we increase the length of the template by one, so that the template length is now $m + 1$, and the procedure is repeated. B_n is more formally defined as

$$B_n = \sum_{k=1}^{N-m} \Theta(\|x_n^m - x_k^m\| < r) \quad (11.9)$$

where x_i^m is an m length template composed of $\{[x[n] \ x[n + 1], \dots, x[m]]\}$, r is a radius for comparison of points, N is the length of x , Θ is the Heaviside operator, and $\|\cdot\|$ is the Euclidean norm (see Equations 10.11 and 10.12). Here, a template match is not an exact match, but one for which the distance between the template and its match is within radius r . The value of r is usually taken as 0.15 times the standard deviation of the signal. A_n (Equation 11.10) is similarly defined as

$$A_n = \sum_{k=1}^{N-m+1} \Theta(\|x_n^{m+1} - x_k^{m+1}\| < r) \quad (11.10)$$

where Θ, x, m, n, k, N , and r are as given in Equation 11.9.

Biosignal and Medical Image Processing

ApEn is then calculated as the sum of the natural log of the ratio of A_n to B_n and normalized by the total number of points, or more conveniently as given in Equation 11.11:

$$\text{ApEn}(x, m, r, N) = - \left(\frac{\sum_{n=1}^{N-m} -\log A_n}{N-m} - \frac{\sum_{n=1}^{N-m+1} -\log B_n}{N-m+1} \right) \quad (11.11)$$

where all variables are as described for Equation 11.10.

Figure 11.6 illustrates the template-matching process using correlated noise as a test signal. Here, m is equal to 1 in the upper image (Figure 11.6a) and 2 in the lower image (Figure 11.6b). In both examples, we use the second sample as the beginning of the template, so $n = 2$. For $m = 1$, the template is just a single point in the time series and highlighted as the sample at $x[2]$, with a black box as a marker (Figure 11.6a). Template matches are simply all the other samples within the range $x[n] - r$ and $x[n] + r$, where $r = 1$. Since $x[2]$ is 2.039, the range for a match is 1.039 through 3.039. This region is shaded in gray. There are six points in our sample signal that fall in that range, and they are marked with diamonds and labeled 1 through 6. Therefore, for the sample at $n = 2$, the value of B_2 is 7 (6 points plus the original point).

For A_2 (Equation 11.11), we increase m by 1, as shown in the lower plot. Now the template is not a single sample, but a sequence of two samples, $\{x[2], x[3]\}$. These samples are shown connected

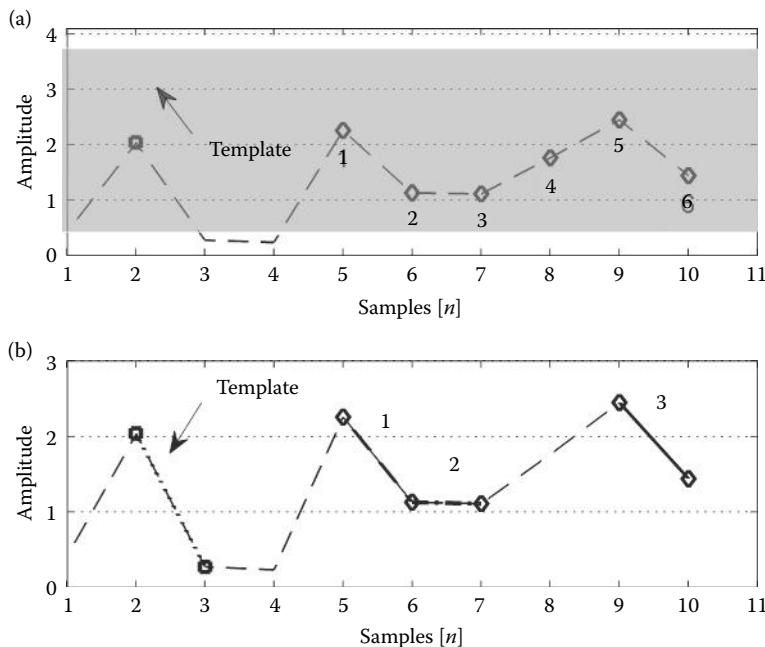


Figure 11.6 Template matching for B_2 and A_2 (Equations 11.9 and 11.10), where $\tau = 1$ and $n = 2$. Here, we show template matching in two dimensions with $r = 1$. (a) For $m = 1$, this is a scalar comparison, and we find six template matches; that is, there are six samples whose value is the range $x[2] - r < x[n] < x[2] + r$. Each template match is labeled 1 through 6 and lies in the shaded region. (b) In this case, $m = 2$ and two-sample template matches are indicated, where we define a match using the Euclidean norm. Specifically, a match occurs between two-sample pairs when $\| (x[1], x[2]) - (x[n], x[n+1]) \| \leq 1$. Here, there are three such matches. Template matching during computation of ApEn for one $m = 1$ is shown in (a) and for $m = 2$ is shown in (b).

by a dotted line and with black squares. We count the number of two-sample windows that are template matches. A two-sample window is considered a match if the Euclidean norm between the series values of the template and the window is less than r , where $r = 1$. Here, the template is $\{2.03, 0.27\}$. There are three template matches. Each two-sample window that shows a match is indicated using diamonds and labeled from 1 to 3. The first match is the window $\{x[5], x[6]\}$. The second match is $\{x[6], x[7]\}$, and the last is $\{x[9], x[10]\}$. These three matches are connected by a dash-line, a dash-dot line, and a dark black line. A_2 is 4 since there are three template matches plus the original template. For the full calculation of entropy, A_n and B_n are determined for the entire length of the signal followed by the entropy calculation using Equation 11.12.

The ApEn algorithm is implemented in the function `apen`, and has a calling structure

```
Ap = apen(x, tau, m, r); % Calculate approximate entropy
```

where x is the signal to be evaluated, τ and m are the typical delay parameters, and r is the threshold value for nearest neighbors. The output, Ap , is the estimated approximate entropy.

```
function Ap = apen(x, tau, m, ry1 = delay_emb(x, m, tau));
% Function to determine approximate entropy
%
y2 = delay_emb(x, m + 1, tau); % Delay in dimension m + 1
N = length(y2(:, 1)); % Get length of embedded signal
%
for ii = 1:N
    % Get the number of nearest neighbors (same as in
    % GPA, Lyapunov from Chapter 10)
    vec1 = repmat(y1(ii, :), length(y1(:, 1)), 1); % Subtraction array
    dist1 = sqrt(sum((y1 - vec1).2, 2)); % Euclidean distance
    numc1 = r > dist1; % Distance test
    Bn(ii) = sum(numc1); % Count neighbors: Eq. 11.10
    %
    % Repeat for m=m+1;
    vec2 = repmat(y2(ii, :), length(y2(:, 1)), 1);
    dist2 = sqrt(sum((y2 - vec2).2, 2));
    numc2 = r > dist2;
    An(ii) = sum(numc2); % Count neighbors Eq. 11.11
end
%
% Get one more sample for rcount1
ii = ii + 1;
vec1 = repmat(y1(ii, :), length(y1(:, 1)), 1);
dist1 = sqrt(sum((y1 - vec1).2, 2));
numc1 = r > dist1;
Bn(ii) = sum(numc1);
Ap = - (sum(log(An)) / (N - m) - sum(log(Bn)) / (N - m + 1)); % Determine entropy
% via Eq 11.12
```

This function is used in the next example to determine the entropy of several test signals.

EXAMPLE 11.9

Create three signals, including Gaussianly distributed noise; a sine wave ($f = 50$ Hz, $f_s = 1$ kHz); and samples of the Hénon map. Estimate the approximate entropy using `apen`. For all three

Biosignal and Medical Image Processing

signals, $N = 1000$. For apen parameters, use $m = 2$ and $\tau = 1$. For the Hénon map, use parameters $a = 1.4$ and $b = 0.3$, with initial values of 0.1 for x and y .

Solution

Use the function `randn` to generate the random signal, and `henon` to get the samples of the Hénon map. Apply `apen` to the three signals using a loop.

```
% Example 11.9 Example to calculate approximate entropy
% to 3 test signals
%
f=50;                      % Sine wave frequency
fs=1000;                    % Sine wave sampling frequency
N=1000;                     % Number of samples
xint=0.01;                  % Initial values for the Henon map
yint=0.01;
a=1.4;                      % a and b parameters for
b=0.3;                      % chaotic Henon map
m=2;                        % Embedding dimension
tau=1;                      % Embedding delay
%
% Generate the 3 test signals
[y,x_hennon]=henon(xint,yint,N-1,a,b);      % Henon map
x_rand=randn(1,N);                         % Random signal
x_sine=sin(2*pi*f*(1:N)/fs);                % Sine wave
test=[x_sine; x_hennon; x_rand];             % Matrix of test signals
%
for k=1:3
    x=test(k,:);
    r=0.15*std(x);                         % Distance threshold
    Ap(k)=apen(x,tau,m,r);                 % Get apen
end
```

Results

The values determined in this example are given in Table 11.2. As we would expect, the sine wave has the lowest entropy and the random signal has the highest.

In the next section, we describe sample entropy, a modification of ApEn that adjusts for a flaw in the way ApEn is determined.

11.4.2 Sample Entropy

Although approximate entropy works well for many signals, it is a biased measurement that always underestimates the entropy. This is because during the calculation of ApEn, the template match between the current template and itself is always counted. This match does not represent a real repeat within the structure of the data being measured and is just an artifact.* To address this issue, SampEn was introduced by Richman and Moorman (2000) as an improvement.

Table 11.2 Approximate Entropy for the Signals Used in Example 11.9

Signal	Sine Wave	Hénon Map	Random
Approximate entropy	0.140	0.560	1.469

* It should be noted that for data sets with a large number of samples (>1000), the bias caused by the self-match in ApEn is not significant.

11.4 Phase-Space-Based Entropy Methods

Ideally, the self-match would just be discarded, but this is not possible in the ApEn formulation. Without the self-match, it is possible for A_n or B_n to be zero for some templates, which gives an undefined value due to the log operation in Equation 11.11. By redefining A_n and B_n such that the logarithm step occurs last, A_n and B_n always have values greater than zero as long as there is one template match.

Templates and template matches are defined for SampEn as they are for ApEn. The difference between their calculations is that for SampEn, template matches for templates of length m (Equation 11.12) and length $m + 1$ (Equation 11.13) are counted for every template in the trajectory before taking logarithms. The equations for SampEn are given in Equations 11.12 through 11.14

$$B = \frac{\sum_{n=1}^{N-m} B_n}{N - m} \quad (11.12)$$

$$A = \frac{\sum_{n=1}^{N-m} A_n}{N - m - 1} \quad (11.13)$$

$$\text{SampEn} = -\log \frac{A}{B} \quad (11.14)$$

where B_n is the number of points close to point x_n of the data in the embedded time series with one point subtracted to account for the self-match. A_n is calculated similarly at a dimension $m + 1$.

As with approximate entropy, sample entropy is appropriate for use with a biological time series and is robust in the presence of noise. It shares the drawback that it cannot be used as a marker of deterministic chaos. However, because of the removal of the biased term, sample entropy is more robust than approximate entropy for shorter sequences of time series.

The function `sampen` computes sample entropy:

```
samp = sampen(x, tau, m, r),
```

where `samp` is the computed sample entropy and all other variables are as in `apen`. The function is very similar to `apen`, so not every line is shown here. The main body of the function is

```
%...x,m,y1,y2,N1,N2, and N are defined as in apen
%
for n=1:N-m
    vec=repmat(y1(n,:),N1,1);
    dist=sqrt(sum((y1-vec).T2,2)); % A vector of the template so that
                                              % Perform the norm operation
                                              % as array function without a loop
    numc=r>dist;                      % Perform template match test
    Bn(n)=sum(numc)-1; % Get number of neighbors. Subtract for self-match
    % Repeat for m+1, otherwise the same
    vec2=repmat(y2(n,:),N2,1);
    dist2=sqrt(sum((y2-vec2).T2,2)); % As above for m=m+1
    numc2=r>dist2
    An(n)=sum(numc2)-1;
end
% Get one more sample for Bn.
```

Biosignal and Medical Image Processing

```
% Repeat the process from the loop once more
n=n+1;
vec=repmat(y1(n,:),N1,1);
dist=sqrt(sum((y1-vec).2,2));
numc=r>dist;
Bn(n)=(sum(numc))-1;
%
B=sum(Bn)/(N-m+1); % Eq. 11.12
A=sum(An)/(N-m); % Eq. 11.13
Samp=-log(A/B); % Eq. 11.14
```

In Example 11.10, we compare sample entropy and approximate entropy so that the effect of signal length can be made clear.

EXAMPLE 11.10

Using functions `apen` and `sampen`, estimate the entropy of a signal created using `rand` for 13 length sizes. Use lengths in powers of 2 from 2^2 to 2^{14} , $m = 2$, and a cutoff radius 0.15 times the signal's standard deviation and $\tau_{au} = 1$.

Solution

Use a loop to generate the random stimulus, and `sampen` and `apen` to solve for sample and approximate entropy, respectively.

```
m=2; % Setup embedding parameters
tau=1;
% Get the apen and the sampen for each signal length
for N=2:14
    x=randn(1,2*N); % Generate random signal
    r=.15*std(x); % Radius threshold value
    Sp(N-1)=sampen(x,tau,m,r); % Determine Sampen
    Ap(N-1)=apen(x,tau,m,r); % Determine ApEn
end
.....lables and plotting..... % Resample by factor of L + 1
```

Results

The SampEn and ApEn as functions of number of samples are shown in Figure 11.7. Both the SampEn and ApEn converge to similar entropy levels (about 2.75), but the sample entropy converges faster; it is accurate for fewer samples than ApEn. While approximate entropy gives results for short segments (less than 256 samples), the entropy levels are incorrect. This is because for such short data segments, the self-matches in ApEn have a strong influence mathematically, but are not meaningful for the determination of entropy. There is no reason to use ApEn in place of SampEn, although for long signals, the difference is minimal. There is a straightforward reason why SampEn is undefined on the plot for lengths below 100 samples, but this is left as a chapter problem.

A drawback of all the entropy methods discussed so far is that they can only probe for behavior at one scale. However, many systems contain information at multiple scale lengths. Consider the heart-beat electrocardiogram signal. The interbeat period between successive heart beats (known as the *R-R* interval) can have features at time scales on the order of fractions of a second, to hours, to days. A measurement that reveals details of a system at multiple scales can give information that is not readily apparent at a single-scale measurement. This is particularly useful for diagnosis of clinical disease. For example, it is known that the *R-R* interval is affected by both sympathetic and parasympathetic stimulation, but that parasympathetic regulation affects

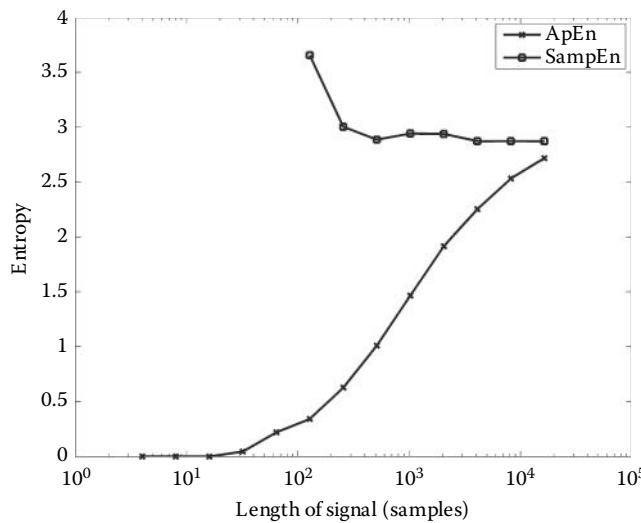


Figure 11.7 The effects of signal length on ApEn and SampEn. SampEn (squares) gives a reasonable estimate of the entropy for a signal with about 256 samples, but the ApEn (x markers) is not accurate until the signal has more than 2000 samples.

the *R-R* interval at shorter time scales than sympathetic regulation. Fortunately, a relatively simple way to evaluate a signal at multiple scales was introduced by Costa (2002). This method is a combination of the coarse graining technique and sample entropy and is called *multiscale entropy* (MSE).

11.4.3 Coarse Graining

Chapter 10 introduces the concept of a fractal object. Fractal objects have information at every scale. For a 2-D fractal object, the fractal nature is evident in the way that the object appears similar at any level of magnification (see, e.g., the Koch curve in Figure 10.20). In this text, however, we are mainly concerned with signals as a function of time. In one dimension, the fractal nature is evident by the behavior of the signal as a function of sampling frequency. A signal that is fractal in time appears the same when sampled at any frequency. This implies that a simple way to examine a signal for scaling behavior is to alter the sampling frequency and examine the signal properties at each scale. This is precisely what we do, though typically a lowpass filter is added to prevent aliasing. This procedure is known as *coarse graining*.

Coarse graining is a resampling technique that reduces the time scale of a signal. With the scale of a signal reduced, one can then perform a signal measurement to determine how the new scale effects the measurement. If the measurement is scale-invariant, it will not change for different time scales. Costa introduced a method for reducing the scale of the series in which samples were grouped into nonoverlapping segments and then each segment was averaged. The problem with this method is that the simple average is a weak lowpass filter, and if there is high-frequency content within the signal, this method can introduce aliasing. Valencia proposed an improvement to this method by using a lowpass Butterworth filter instead of a multipoint average. The Butterworth filter (see Section 4.5.2) reduces the bandwidth of the signal before samples are removed to reduce the scale. When this procedure is combined with a determination of sample entropy after each rescale step, it is known as MSE. This also carries one additional refinement. In sample entropy, the threshold for determining what is considered a template match is typically defined as 0.15 times the standard deviation of the signal. However, the standard deviation decreases after each rescaling; therefore, it is appropriate to recalculate the template-match

Biosignal and Medical Image Processing

threshold with every new scale. Calculating sample entropy (or other metric) at each scale leads to an MSE curve: sample entropy as a function of scale. To reduce this plot to a single number, it is common to integrate the curve (i.e., the summation) or to determine its slope around specific scale values.

The routine `coarsening` performs coarse graining by resampling after the application of a Butterworth filter. The function is called as

```
resamp_x = coarsening(x,L); % Resample x
```

where x is the signal, L is the number of sampling levels, and resamp_x is the resampled data. The body of the function is straightforward, but we take advantage of a subtle algebra trick. The second input of MATLAB function `butter` is the cutoff frequency of the filter. However, `butter` does not take as the input the frequency itself, but the ratio of the cutoff frequency to half the sampling frequency (see Chapter 4). Since we really only care about this ratio, we can simply set the ratio in terms of L . As written, if L is 1, the cutoff ratio is $1/2$, which is to say that the filter removes half the frequency content. For $L = 2$, the cutoff ratio is $1/3$, and the upper $2/3$ rd of the frequencies are removed. So we define a cutoff frequency as $1/(1+L)$, which becomes the cutoff frequency normalized by half the sampling frequency. The reason we like to define the cutoff this way is that L can also be used to perform the downsampling, since we are removing all the frequencies by a factor of $1+L$, and we can reduce the number of points by a factor of $1+L$ as well.

```
function resamp_x = coarsening2(x,L)
% Function to resample signal in x
cutoff = 1/(1+L); % Butter defines the cutoff
% frequency in terms of fs/2
[B,A] = butter(6,cutoff); % Define filter
resamp_x = filtfilt(B,A,x); % Filter
resamp_x = resamp_x(1:(L+1):end); % Resample by factor of L+1
```

For a 10,000-sample sine wave ($f_s = 10 \text{ kHz}$, $f = 100 \text{ Hz}$), if $L = 1$, the output will be a 5000-sample wave of ($f_s = 5 \text{ kHz}$, $f = 100 \text{ Hz}$). The signal will have been filtered at a cutoff frequency of $(1/(L+1))f_s/2 = 2.5 \text{ kHz}$.

An example of correlated noise signal after 2, 3, and 6 scaling factors is shown in Figure 11.8. Correlated noise has features that exist on many scales, but at each level of filtering, some features are removed. At six scaling steps, we see only broad features. Note that this is different from just filtering the signal at progressively lower cutoff frequencies. Such filtering does not remove information at the filtered frequencies, it only reduces their power.

In the next example, we determine the MSE of a signal that we design specifically to have information at multiple scales.

EXAMPLE 11.11

Create a test signal composed of a 100-Hz sine wave ($f_s = 2 \text{ kHz}$) and correlated noise. Use the output of `randn` divided by 10 with `cumsum` to make the correlated noise vector. Measure the entropy of the signal for 30 scale lengths and plot the entropy as a function of scale. Use sample entropy as the entropy measurement, with a template length of 2 and a delay of 1.

Solution

Add the sine wave to the correlated noise signal to create the test signal. Use a loop to coarse grain the test signal for scale lengths from 1 to 30. After each coarse graining step, calculate a new r value and then use `sampen` to get the SampEn, with $\tau_{\text{au}} = 1$ and $m = 2$.

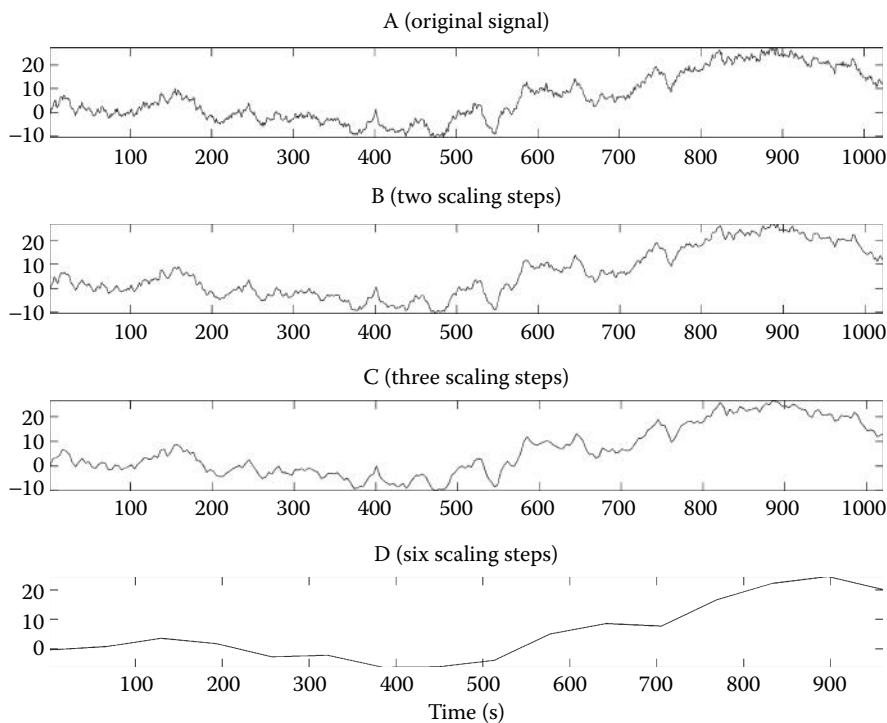


Figure 11.8 A correlated noise signal coarse grained at multiple scale lengths. (A) The original signal. (B) The signal scaled down by half. (C) The signal scaled down three times. (D) The signal scaled down six times.

```
%Example 11.11
% Create a test signal of a sine wave
% and correlated noise and perform MSE
%
fs = 2000;          % Sine wave sample frequency
f = 100;            % Sine wave frequency
N = 10000;          % Number of samples
L = 30;             % Number of coarse graining steps
m = 2;              % Embedding dimension (or template length)
tau = 1;             % Delay
t = linspace(1/fs,N/fs,N);           % Time vector
%
% Generate test signal
xnoise = cumsum(randn(1,N)/10);      % Correlated noise
y = sin(2*pi*f*t);                  % Sine wave
x = y + xnoise;                    % Test signal
%
r = 0.15*std(x);                  % Threshold
MSE(1) = sampen(x,tau,m,r);        % Sampen for original signal
for k = 1:L
    resamp = coarsening(x,k);       % Perform coarse graining
    r = 0.15*std(resamp);          % Recalculate threshold
    MSE(1+k) = sampen(resamp,tau,m,r); % Sample entropy
end
.....plotting and labels.....
```

Analysis

Using a low-level correlated noise signal in our test signal means that the signal has information at many time scales, especially longer time scales, while using the 100-Hz sine wave gives us information at a shorter time scale.

Results

SampEn as a function of scale factor is shown in Figure 11.9. We see that there is initially very little entropy, as we would expect from a signal dominated by a sine wave. As the scale factor increases, the sampling frequency decreases and the entropy increases until it reaches a maximum at scale factor 5. At a scale factor of 5, the effective sampling frequency is 333 Hz (2000 Hz/6). Half this sampling frequency (i.e., $f_s/2$) is still greater than the frequency of the 100-Hz sine wave, but at this sampling frequency, the sine wave is not well sampled, so it loses some of its periodicity and the entropy increases. At subsequent scale factors, the entropy decreases as the sine wave is filtered out. The entropy then slowly increases after a scale factor of 10, as the effects of the correlated noise come to dominate for longer length scales.

The key feature of the MSE determination is the plot of the entropy versus scale that is produced, but there are several different metrics that are used to describe this plot. The most straightforward metric is the sum of the entropy over all scales. One can also quantify the rate at which entropy increases or decreases as a function of scale, or really any metric that describes

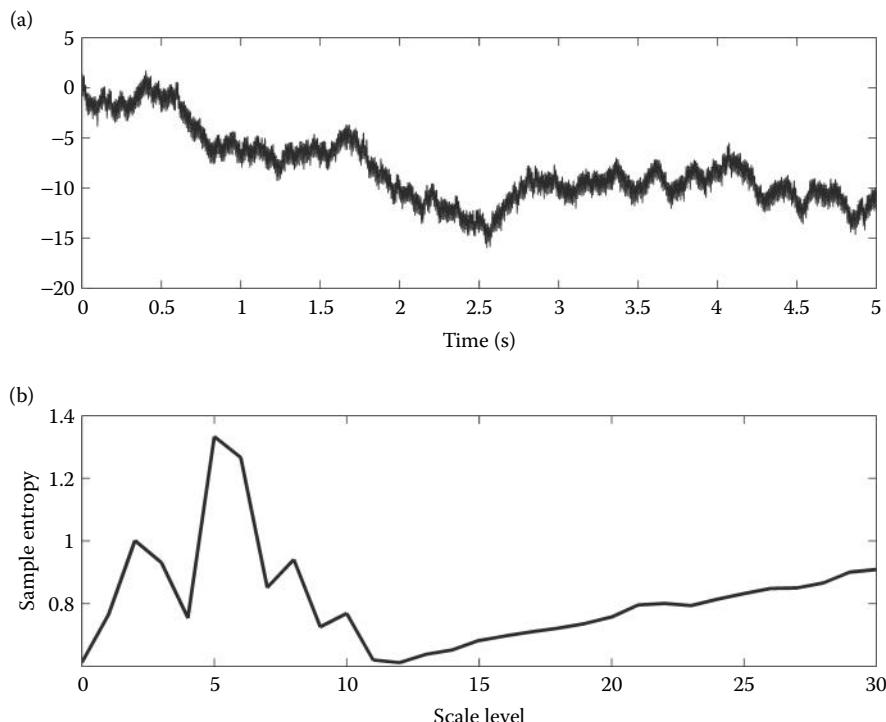


Figure 11.9 A signal and its multiscale entropy. (a) The test signal is correlated noise added to a 100-Hz sine wave. (b) The multiscale entropy shows an initial increase and decrease in entropy when the sampling frequency approaches and passes the Nyquist frequency of the 100-Hz sine wave. For longer time scales, a steady increase in entropy is seen as the correlated noise dominates the signal.

some property of a 1-D curve. The best method depends on the application. Lastly, even though we have used SampEn in this analysis, other methods can be used in combination with scaling to perform a multiscale analysis. Again, the preferred method may only be determined through trial and error.

11.5 Detrended Fluctuation Analysis

The final nonlinear method of analysis to be discussed here is called *detrended fluctuation analysis (DFA)*. As with MSE analysis, the goal of DFA is to measure the properties of a signal at multiple scales. However, with DFA, the process of assessing the multiscale behavior is performed by removing trends rather than filtering (though the trend removal could be regarded as a nonlinear filter). This technique was introduced by Peng (1994) and has been shown to elicit differences in scaling behavior in *R-R* interval records for normal and diseased subjects.

The main assumption of DFA is that the signal is influenced both by short-term transient effects and by long-term effects that act slowly and do not die out quickly. For example, when analyzing the *R-R* interval for clinical significance, we are typically not interested in interbeat changes that result from short-term stimuli. Instead, we are interested in long-term behavior that occurs over an extended period of time. It is reasonable to expect that the behavior of a system in response to a short-term external perturbation will not affect long-term correlations, whereas internal dynamics may display long-term correlations. DFA is a method that recursively evaluates a signal at multiple length scales. By examining trends at multiple length scales, we can sort the effects of short-term perturbations from the effects of the internal dynamics (i.e., the body's many internal feedback systems), which presumably act at a longer scale.

The first step in the algorithm is to integrate the signal:

$$y_{int}[n] = \sum_{t=1}^N y[n] \quad (11.15)$$

where n is the time index and N is the number of samples in the signal.

The signal y_{int} is then divided into K segments of length m and, for each segment, a polynomial fit, y_{lf} , is removed. Usually, the polynomial is first-order, so the fit is a linear fit. After removing the linear trend from each segment, the RMS value of the entire signal length is calculated:

$$F(m) = \sqrt{\frac{1}{N} \sum_{k=1}^{K} [y_{int}(k) - y_{lf}(k)]^2} \quad (11.16)$$

where m is the segment length and $y_{int}(k)$ is the k th segment from which a linear fit, $y_{lf}(k)$, has been subtracted. This is performed using increasing segment lengths, m , and results in a decreasing number of segments, K . To quantify the fluctuation behavior, $F(m)$ is plotted versus m on a double log plot and either the integration of this curve or the linear fit to its slope can be used as a measure of the fluctuation.

To illustrate the procedure, we show the results of DFA applied to a short segment of integrated Gaussian noise. Specifically, we show DFA applied to nine segments (Figure 11.10). The trend lines (i.e., linear fit) for nine segments, each 10 samples long, are shown in Figure 11.10a. The linear trends are shown as straight lines that overlay each of the nine segments (segments are identified by the light vertical lines, Figure 11.10a). The trend line or linear fit may, or may not, accurately represent the signal for a given segment depending on the fluctuations. As described in Equation 11.17, these trend lines are subtracted from the signal and the RMS value of the residual is taken. The result of that procedure carried out over 10 length scales is shown

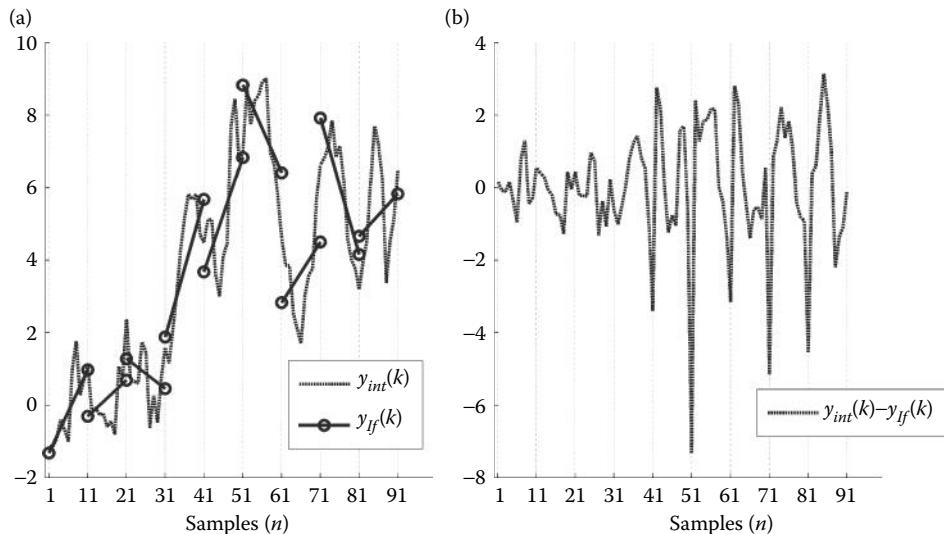


Figure 11.10 (a) Correlated noise and its trend lines for segment lengths of 10 samples. The trends show a good fit in some segments but not others. (b) The correlated noise signal from (a) with the linear trends removed. The RMS of this signal would give $F(10)$ as indicated in Equation 11.16. (b) shows the results of detrending using just one length scale, but if we combine many length scales using Equation 11.16, we can develop the $F(m)$ curve. This curve is displayed on a log-log plot in Figure 11.11.

in Figure 11.10b, which shows the curve of Figure 11.10a with the linear trends removed. Figure 11.11 shows the completed log(RMS) curve on a log-scaled plot, where the log(RMS) of the residuals is plotted against the log of segment length. We can see that the correlated noise signal contains information at many scales, as expected from this signal. In fact, there is increasing RMS value with increasing length scale; this is typical of correlated noise.

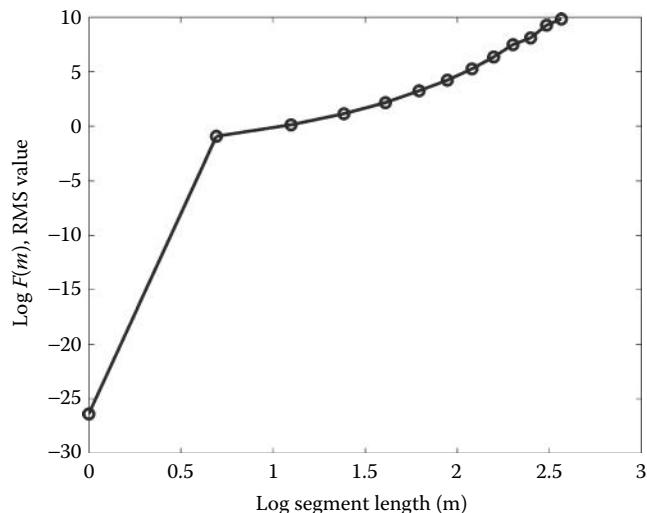


Figure 11.11 The log RMS versus log length scale shows an increase with scale, typical of data with long-term correlations.

11.5 Detrended Fluctuation Analysis

The function DFA performs detrending and RMS determination for one segment length

```
F = DFA(x,win_length,order);
```

where x is the signal to be analyzed, win_length is the segmentation length of the signal, and order is the fit order. Output F is the RMS value of the detrended signal for a given win_length . To develop the RMS curve as in Figure 11.11, this function is used in a loop, in which win_length increases from 1 to whatever upper limit is chosen. As in MSE, the metric taken as the DFA parameter is some measurement of the resulting curve shown as a function of scale. Typically, the curve is plotted on a log-log scale and the slope of linear portion of the curve is measured. The main body of the code that performs the DFA calculations of Equations 11.15 and 11.16 is

```
.....Segment of code that computes the DFA.....
x(:);
x=x'; % Enforce row vector
N=length(x); % Get length of data
N=floor(N/win_length); % n=number of segments
N1=n*win_length; % Length truncated data
y=zeros(N1,1); % Initialize y and Yn
Yn=zeros(N1,1);
Fitcoef=zeros(n,order+1); % Variable to hold fit coef
mean1=mean(x(1:N1)); % Mean of truncated data
y=cumsum(x(1:N1)- mean1); % cumulative sum of data

for j=1:n % Find the fit for each segment
    fitcoef(j,:)=polyfit(1:win_length, ...
    y(((j-1)*win_length+1):j*win_length),order);
end
%
% Get the fitted values used in subtraction
for j=1:n
    Yn(((j-1)*win_length+1):j*win_length)=...
    polyval(fitcoef(j,:),1:win_length);
end
%
F=sum((y'-Yn).2)/N1; % Remove the trends
F=sqrt(F); % and get RMS (Eq. 11.16)
```

As a final example for this chapter, we perform a DFA analysis using the ECG segment data used in Chapter 10.

EXAMPLE 11.12

Load in samples of ECG data and use DFA to perform the nonlinear analysis. Use window lengths of 1 through 30 and a linear fit order. Measure the RMS as function of window length by taking the slope.

Solution

Load the signal ECG.dat and then use a loop to get the detrended RMS, F , for each iteration using routine **DFA**. Plot $\log(F(m))$ versus $\log(m)$, then use **polyfit** to get the slope of this curve.

Biosignal and Medical Image Processing

```
% Ex 11.12 Example of DFA applied to ECG data
%
signal=load('ECGtest.csv');           % Load in the sample signal
order=1;                             % Set order of trends (linear
m=2.^{1:14};                         % Segment lengths: 14 powers of 2
for ii=1:length(m);
F_m(ii)=DFA(signal,m(ii),order);    % Get DFA
end
fit=polyfit(log(m),log(F_m),1);      % Find linear slope
.....plotting and labels.....
```

Analysis

Since segment lengths are plotted and analyzed following a log operation, we choose an exponential range for m from 2^2 up to 2^{14} , which is the largest power of 2 that fits within the length of our data.

Results

The log-scaled plot of $F(m)$ versus m is given in Figure 11.12. The curve is close to linear. This indicates that the signal contains information at many scales and that there is no process at any particular scale that dominates the signal.

In some cases, there can be several linear portions of the log-log-scaled curve of $F(m)$, and in these cases, one can take the slope of each of these regions as a metric. However, because this signal contains information evenly distributed across all scales, only one linear region appears to be present.

DFA is a technique that is relatively simple but powerful if you believe that your data may contain signatures of long-term processes but is dominated by unwanted or uninteresting events at the shorter terms.

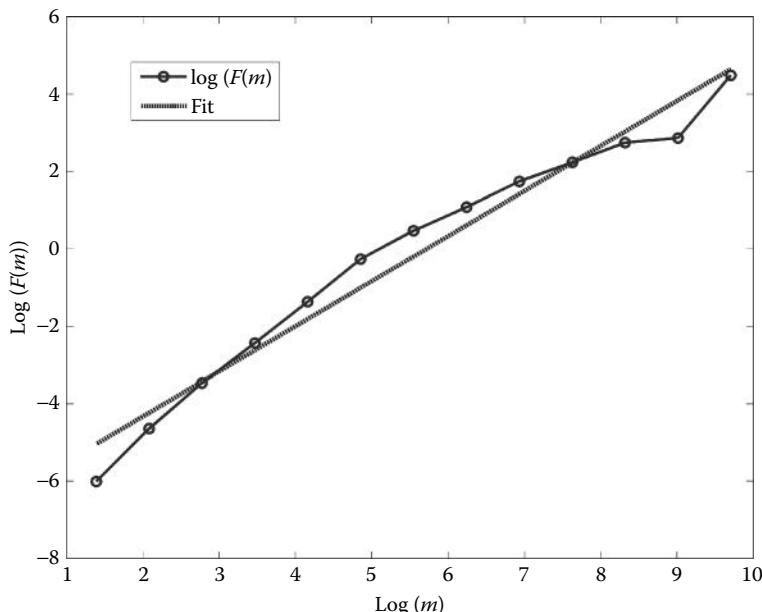


Figure 11.12 The slope of the RMS curve for the ECG signal versus the scale factor, m , shows a nearly linear slope. The slope of the fit is 1.164. This indicates that the signal contains information at many scales and that there is no process that dominates the signal.

11.6 Summary

Unlike the methods discussed in Chapter 10, entropy and information-based measures do not test for any particular parameter of a system. Rather, they give a sense of the complexity of the signal, which is a measure of the signal dynamics. These methods do not give a conclusive statement about the signal, but are useful when a general statement about signal regularity is needed. They can be quite useful for distinguishing between noisy signals and complicated, but deterministic, signals. They are therefore more appropriate for general analysis or for systems that may contain nonlinearity or chaos. This type of analysis is more useful when the goal is merely to try to differentiate between data from two (or more) classes rather than to make a definitive statement about the data themselves.

The idea of applying concepts of entropy to describe measured signals was first introduced by Shannon primarily for stochastic signals that involve bit transmission. The relation of entropy to dynamical systems is not immediately obvious, but it becomes more so when one considers that the entropy of a signal can describe its regularity, and the regularity of a signal depends strongly on system properties. In general (though not always), nonlinear or chaotic systems can produce signals that are highly irregular, yet are fully deterministic and contain long-term correlations. Therefore, it is of interest to determine not only the entropy of a signal but also its entropy rate: the rate at which information in the signal is gained or lost.

One way to estimate the rate at which entropy is lost, as well as to get a sense of how the system properties change at differing scales, is to use the mutual information function. Mutual information is the nonlinear analog to autocorrelation. Autocorrelation indicates over how many future samples a signal is able to predict itself while the AMIF indicates over how many future samples a signal is able to predict *information* about itself. The AMIF is useful for exploring the properties of signals that may have hidden scaling information.

Spectral entropy's strength, and weakness, is that it is based on the power spectrum of the signal. Because the power spectrum can be computed using computationally efficient FFT algorithms, spectral entropy is very useful as a real-time analysis method, especially in EEG analysis. However, because phase information is discarded, it cannot be used to probe for or establish nonlinearity within a signal, even with surrogate data testing. That requires more sophisticated entropy techniques, such as SampEn.

SampEn is an improvement over ApEn and can be seen as a multisample extension to a histogram-based entropy assessment. Because SampEn can take more than one sample of data into account at a time, it is sensitive to data that have correlations. SampEn can be used in conjunction with surrogate data testing to detect nonlinearity as well as provide a measurement of signal entropy. SampEn is also accurate for signals less than 1000 samples long, though the accuracy also depends on the signal itself. Signals with as many samples as possible should be used.

Though SampEn can be used to analyze signals with correlated samples, it is not suitable for establishing long-term correlations and scaling. To analyze long-term correlations in a signal, we need to perform coarse graining, which reduces the scale of a signal. We can then observe how the entropy changes as a function of scale. This procedure is known as MSE and is useful for analyzing signals, such as the heart rate, in which there is reason to believe that the signal is influenced by physiological systems that act over many different time scales. The coarse graining technique can also be applied in conjunction with other signal analysis methods, and the influence of scale can be studied using techniques other than entropy.

DFA is not an entropy-based method, but it shares with MSE the feature of describing how a signal changes as a function of scale. Here, a signal has trends removed over different segment lengths (corresponding to different scales) and the RMS value of the residuals at each scale is observed. This technique is useful for signals that are under the influence of hidden long-term and/or short-term features.

The topics presented here provide a good starting point for entropy-based analyses of signals. For a more detailed discussion of the methods described here, the reader is encouraged to refer to the references in the Bibliography.

PROBLEMS

- 11.1 Assume that a loaded die has a probability distribution function such that $p[x] = 1/n^2$, where n is the n th side of the die. What is the amount of information in the die? What if the die is fair (each side is equally likely)?
- 11.2 Sarah likes both chocolate and vanilla ice cream, but she eats chocolate ice cream 60% of the time and vanilla 40% of the time. If she chooses chocolate ice cream, she prefers cherry toppings 75% of the time, but if she chooses vanilla ice cream, she chooses strawberry toppings 50% of the time. If she is just picking fruit, she likes strawberries 75% of the time compared to cherries. What is the mutual information between Sarah's ice cream and topping choices?
- 11.3 Write a MATLAB script that uses the function `hist` and Equation 11.1 to estimate the entropy of a signal.
- 11.4 Compare the mutual information for a signal with itself as a function of signal length. Use a sine wave of frequency 10 Hz with $f_s = 1000$ Hz and $N = 1000$. Use `mutual` to get the mutual information of the signal, and then repeat the determination 500 more times, each time removing the last sample, such that the signal becomes shorter after each iteration. Repeat this procedure with a Gaussian noise signal.
- 11.5 Compare the AMIF for random noise, the logistic map, and a sine wave. Use signals of 1000 samples long, with `max_lags = 150` and `nu_bin = 10`. For the logistic map use an initial value of 0.1 and an r of 4.0. Use a sine wave as in Problem 11.4.
- 11.6 Use the AMIF to estimate a value for the delay τ to be used in delay embedding of the data in file `ECGtest.csv`. How does this compare to the estimate for τ determined in Chapter 10?
- 11.7 The AMIF provided in this chapter has a flaw: the number of cells in the 2-D histogram used to compute the mutual information is fixed; however, for large lags, this might mean that some of the histogram cells are sparse or empty. Write a new `amif_modified` script by modifying the program `amif`. The new program should change the number of cells based on the length of the signals being used by routine `inform2` inside the function `amif`. Specifically, the number of cells could be 1/10 of the total length of the shortened signal at each lag. Test it on correlated noise as in Example 11.7.
- 11.8 Using the logistic equation introduced in Chapter 10, plot the sample entropy as a function of the forcing parameter of the logistic equation as the parameter varies between 3 and 4. That is, develop the logistic equation for 1000 iterations for each of 100 values using a forcing parameter, r , between 3 and 4. For the sample entropy, use a template length (m) of 2 and a delay of 1.
- 11.9 Modify Example 11.10 to show the results of varying template length (m) instead of signal length. Vary m from 1 to 10. Use a signal length of 10,000 points.
- 11.10 Modify Example 11.10 to explore the effects of changing the cutoff radius. Use a signal length of 10,000 and a template length of 2, but vary the cutoff radius between 0.01 and 0.4.
- 11.11 Determine the spectral, sample, and approximate entropy of the harmonic oscillator under the three conditions used in Example 10.1 (i.e., linear oscillations, nonlinear oscillations, and chaos).
- 11.12 Using `sig_noise`, determine whether `sampen` or `specen` is more robust to noise by determining the SNR at which both measures give poor results. Get a

baseline measurement by first measuring the entropy at a high SNR of +60 dB, then repeat the measurements for decrements of 10 dB until there is a large decrease in baseline. Use a 100-Hz signal.

- 11.13 Examine the effect of filtering on Sampen by creating a noise signal using `randn` of 10,000 and measuring the sample entropy after filtering with a 4th order Butterworth lowpass filter using a cutoff of 1000, 500, 250, and 100 Hz. Assume a sampling frequency of 2 kHz.
- 11.14 Modify `apen` and `sampen` to give the output of A_n and B_n . For a 10,000-sample random noise signal (use `randn`), compare the outputs of A_n and B_n for `apen` and `sampen`. Repeat for a 1000-sample random signal. Try again with a 100-sample signal. How does this explain why the Sampen may be undefined for very short signals?
- 11.15 Determine the spectral entropy, approximate entropy, and sample entropy of the Van der Pol equations (given in Problem 10.4) using a time span from 0 to 100, and initial conditions of $x_0 = 0.01$, $y_0 = 0$. Solve the equations for $u = 0, 0.1, 1.0$, and 5.0.
- 11.16 Repeat Example 11.8 using sample entropy instead of spectral entropy.
- 11.17 To observe the difference between coarse graining with resampling and simple filtering of the signal, construct a plot similar to that of Figure 11.8 in which a signal is repeatedly filtered at decreasing filter cutoffs frequencies. Use only filtering; do not resample the signal. Compare this to the plot in Figure 11.8. [Hint: create a new function using a modified version of `coarsening`.]
- 11.18 Redo Example 11.11 but this time with just the sine wave. Then repeat with just the noise signal. How do the results change?
- 11.19 Develop a MATLAB function to compute the MSE of a signal, but using spectral entropy instead of sample entropy. Test your function by repeating Example 11.11 using your new function.
- 11.20 What kind of signal do you think will produce a parabolic MSE curve? Test your guess by computing the MSE of the proposed signal. If your guess fails, keep trying until you get it.
- 11.21 Determine $F(m)$ for a test signal consisting of the sum of a 200 Hz sine wave and a random signal. The sine wave should have a sampling frequency of 2000 Hz. How many linear regions does the $F(m)$ curve have on a log–log plot? What is the slope of these regions? Use a linear trend for the DFA algorithm.
- 11.22 Repeat Exercise 11.21, but use a second-order trend and then a third-order trend.

12

Fundamentals of Image Processing

The MATLAB Image Processing Toolbox

12.1 Image-Processing Basics: MATLAB Image Formats

Images can be treated as 2-D data and many of the signal-processing approaches presented in the previous chapters are equally applicable to images: some can be directly applied to image data while others require some modification to account for 2-D variables. For example, both PCA and ICA have been applied to image data treating the 2-D image as if it were a single extended “waveform.” Other signal-processing methods can be applied to images using 2-D extensions, including the Fourier transformation, convolution, and digital filtering. 2-D images are usually represented by 2-D data arrays, and MATLAB follows this tradition; however, MATLAB offers a variety of data formats in addition to the standard format used by most MATLAB operations. 3-D images can be constructed using multiple 3-D representations, but these multiple arrays can also be treated as a single-volume image.

12.1.1 General Image Formats: Image Array Indexing

Irrespective of the image format or encoding scheme, the basic image is always represented as a 2-D array, $I(m, n)$. Other dimensions may be added for color images* or multiframe images as described later. Each element of the variable, I , represents a single picture element, or *pixel*. If the image is being treated as a volume, then the element represents an elemental volume and is termed a *voxel*. The most commonly used indexing protocol follows the traditional matrix notation, with the horizontal pixel locations indexed left to right by the second index, n , and the vertical locations indexed top to bottom by the first index, m (Figure 12.1). This indexing protocol is termed *pixel coordinates* by MATLAB. A possible source of confusion with this protocol is that the vertical axis positions increase from top to bottom and also that the second integer references the horizontal axis, the opposite of conventional graph notation.

In some special operations such as spatial transformation, MATLAB uses a different indexing protocol that accepts noninteger indexes. In this protocol, termed *spatial coordinates*, the pixel is considered to be a square patch, the center of which has an integer value. In the default coordinate system, the *center* of the upper left-hand pixel still has a reference of (1,1), but the

* Actually, MATLAB considers all image arrays to be three-dimensional as described later in the text.

Pixel coordinate system				
$I(1,1)$	$I(1,2)$	$I(1,3)$	$I(1,n)$
$I(2,1)$	$I(2,2)$	$I(2,3)$	$I(2,n)$
⋮			↗	
$I(m,1)$	$I(m,2)$	$I(m,3)$	$I(m,n)$

Figure 12.1 Indexing format for MATLAB images using the pixel coordinate system. This indexing protocol follows the standard matrix notation.

upper left-hand corner of this pixel has coordinates of (0.5,0.5) (Figure 12.2). In this spatial coordinate system, the locations of image coordinates are positions on a (discrete) plane and are described by general variables x and y .

There are two sources of potential confusion with the spatial coordinate system. As with the pixel coordinate system, the vertical axis increases downward. However, the positions of the vertical and horizontal indexes (now better thought of as coordinates) are reversed in this system: the horizontal index is first followed by the vertical coordinate. This more closely follows the conventional x, y coordinate references, but again y coordinates increase downward. In the default spatial coordinate system, integer coordinates correspond with their pixel coordinates, remembering the position swap, so that $I(5,4)$ in pixel coordinates references the same pixel as $I(4.0, 5.0)$ in spatial coordinates. Most routines use the pixel coordinate system

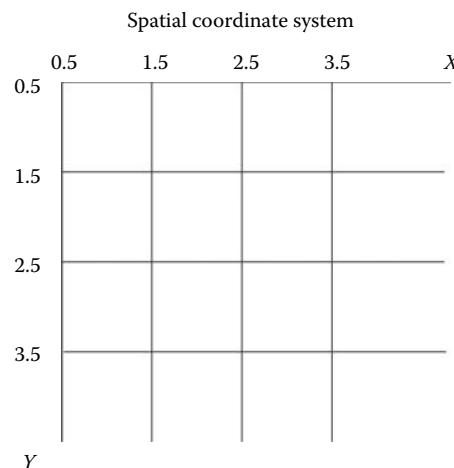


Figure 12.2 The spatial coordinate system for referencing pixel location. This scheme is used by spatial transformation routines and allows for fractional pixel indices.

and produce outputs in that system. Examples of spatial coordinates are found primarily in the spatial transformation routines described in Chapter 13.

It is possible to change the baseline reference in the spatial coordinate system, as certain commands allow you to redefine the coordinates of the reference corner. This option is described in context with related spatial transformation routines.

12.1.2 Image Classes: Intensity Coding Schemes

There are four different data *classes*, or encoding schemes, used by MATLAB for image representation. Moreover, each of these data classes can store the data in three different *formats*. The four different image classes are *indexed* images, *RGB* images, *intensity* images, and *binary* images. The three different formats are *double*, *uint8*, and *uint16*. This variety reflects the variety in image types (color, grayscale, and black and white), and the desire to represent images as efficiently as possible in terms of memory storage. The efficient use of memory storage is motivated by the fact that images often require a large number of array locations: an image of 400 by 600 pixels will require 240,000 data points, each of which will need one or more bytes depending on the data format. The classes are summarized in Table 12.1, which shows the formats each class supports and the convention for naming an image variable belonging to that class. Image classes are discussed below and data formats in the next section.

The indexed and RGB classes are used to store color images. With indexed images, the pixel values do not represent a color or grayscale value, but are pointers to a table that *maps* the pixel value to a color value. While this is an efficient way to store color images, the data sets do not necessarily lend themselves to arithmetic operations (and, hence, most image-processing operations) since the results do not always produce meaningful images. Often, indexed images are used to convert a grayscale image (which is amenable to arithmetic operations) to a color-coded image for enhanced visual effect.

Indexed images need an associated matrix variable that contains the *colormap*, and this map variable needs to accompany the image variable in some operations, specifically those that involve image display. Colormaps are N by 3 matrices that function as *lookup tables*. A specific pixel value points to a row in the map, and the three columns associated with that row contain the intensity of the colors red, green, and blue, respectively. The values of the three columns range between 0 and 1, where 0 is the absence of the related color and 1 is the strongest intensity of that color. MATLAB has a suggested convention for keeping track of the various image types, and recommends that indexed arrays use variable names beginning with `X-` and that the name used for the colormap variable is `map--`. While indexed variables are not useful in image-processing operations, they provide a compact method of storing color images, and provide an effective, flexible, method for colorizing grayscale data to produce *pseudocolor* images of otherwise boring gray-level images.

The MATLAB Image Processing Toolbox provides 13 prepackaged colormaps, several with evocative names such as `winter` or `hot`. However, it is easy, and sometimes fun, to make up your own. (A few of the problems play with colormaps.) These colormaps can be implemented with any number of rows, but the default is 64 rows. Hence, if any of the standard colormaps

Table 12.1 Summary of Image Classes

Class	Formats Supported	Preferred Name
Indexed	All	<code>X-</code>
RGB	All	<code>RGB--</code>
Intensity	All	<code>I-</code>
Binary	<code>uint8</code> , <code>double</code>	<code>BW-</code>

Biosignal and Medical Image Processing

are used with the default value, the indexed image should have pixel values scaled to between 1 and 64 to use the full range of the map without saturation. An example of the application of a MATLAB colormap is given in Example 12.3. An extension of that example demonstrates methods for colorizing grayscale data using colormaps.

The other method for coding color images is the RGB (red, green, blue) coding scheme in which three different but associated arrays are used to indicate the intensity of the three color components of the image: red, green, or blue. This coding scheme produces what is known as a *truecolor* image. The larger the pixel value, the brighter the respective color. In this coding scheme, each of the color components can be operated on separately. Obviously, this color coding scheme will use more memory than indexed images, but this may be unavoidable if extensive processing is to be done on a color image. By MATLAB convention, the variable name `RGB`-- is used for variables of this data class. These variables are actually 3-D arrays having dimensions M by N by 3. While we have not encountered such 3-D arrays thus far, they are fully supported by MATLAB (as are 4-D arrays, which are used later in this chapter). These arrays are indexed as `RGB(m,n,i)`, where $i = 1, 2, 3$. In fact, all image variables are conceptualized in MATLAB as 3-D arrays, except that for non-RGB images, the third dimension is simply 1. This fact becomes important when we construct groups of images using 4-D arrays; we must account for the third array dimension even if it is only 1.

Grayscale images are stored as intensity class images where the pixel value represents the brightness or *grayscale* value of the image at that point. MATLAB convention suggests variable names beginning with `I` for variables in class intensity.

If an image is only black and white (not intermediate grays), then a binary coding scheme can be used where the representative array is a “logical” array containing either 0 s or 1 s. MATLAB convention calls for beginning black-and-white image variable names with `BW`--. A common problem working with binary images is the failure to remember and/or define the array as logical, in which case the image variable is misinterpreted by the display routine. Binary class variables can be specified as logical (set the logical flag associated with the array) using the command: `BW = logical(A)`, assuming `A` consists of only 0 s and 1 s. Usually, a binary array is produced by some MATLAB image-processing routine, which automatically sets the logical flag. A logical array can be converted to a standard array using the “unary plus” operator: `A = +BW`. It is possible to check if a variable is logical using the routine: `isa(I,'logical')`; which will return a 1 if true and 0 otherwise. Binary images can be operated using logical operations such as AND (`&`), OR (`||`), or NOT (`~`) just as other logical variables.

12.1.3 Data Formats

In an effort to further reduce image storage requirements, MATLAB provides three different data formats for most of the classes mentioned above (Table 12.2). The `uint8` and `uint16` data formats require 1 and 2 bytes, respectively, for each array element. Binary images do not support the `uint16` format. The third data format, the `double` format, is the same as that used in standard MATLAB operations and, hence, is the easiest to use. Image arrays that use the `double` format can be treated as regular MATLAB matrix variables subject to all the power of

Table 12.2 Summary of Data Formats

Format	Bytes/Pixel	Range	
		Black	White
<code>uint8</code>	1	0.0	255
<code>uint16</code>	2	0.0	65,535
<code>double</code>	8	0.0	1.0

MATLAB and its many operations and functions. In fact, most of the topics, examples, and problems in this section on image processing use the double format. The problem is that this format uses 8 bytes for each array element (i.e., pixel), which can lead to very large data storage requirements.

In all three data formats, a zero corresponds to the lowest intensity value, that is, black (Table 12.2). For the uint8 and uint16 formats, the brightest intensity value (i.e., white, or the brightest color) is taken as the largest possible number for that coding scheme: for uint8, $2^8 - 1$ or 255; and for uint16, $2^{16} - 1$ or 65,535. For the double format, the brightest value corresponds to 1.0.

The `isa` routine can also be used to test the format of an image. The routine, `isa(I,'type')` will return a 1 if `I` is encoded in the format '`'type'`', and a 0 otherwise. The variable '`'type'`' can be '`'unit8'`', '`'unit16'`', or '`'double'`'. There are a number of other assessments that can be made with the `isa` routine that are described in the associated help file.

Multiple images can be grouped together as one variable by adding another dimension to the array. Since image arrays are already considered 3-D, the additional images are added as the fourth dimension. Multi-image variables are termed *mulfirame* variables, and a single image in a multiframe variable is termed a *frame*. Multiframe variables can be generated within MATLAB by incrementing along the fourth index as shown in Example 12.2, or by concatenating several images together using the `cat` function:

```
MFW = cat(4, I1, I2, I3, ...);?
```

The first argument, 4, indicates that the images are to be concatenated along the fourth dimension, and the other arguments are the variable names of the images. All images in the list must be the same type and size. It is common to name multiframe variables beginning with MFW.

12.1.4 Data Conversions

The variety of coding schemes and data formats complicates even the simplest of operations (but is justified in the cause of efficient memory use). Certain operations require a given data format and/or class. For example, standard MATLAB operations require the data be in double format. Many MATLAB image-processing functions also expect a specific format and/or coding scheme, and generate an output usually in the same format as the input. Since there are so many combinations of coding and data type, there are several routines for converting between different types. For converting format types, the most straightforward procedure is to use the `im2xxx` routines given below:

```
I_uint8 = im2uint8(I);           % Convert to uint8 format
I_uint16 = im2uint16(I);        % Convert to uint16 format
I_double = im2double(I);        % Convert to double
```

These routines accept any data class as input, including the *same class* as the output in case you get confused or are unsure of the data class of the original. (So just convert the image to the format you *want* as it does not hurt if it is already in that format.) If the class is indexed, the input argument, `I`, must be followed by the term '`'indexed'`'. These routines also handle the necessary rescaling except for indexed images. When converting indexed variables, range can be a concern: for example, to convert an indexed variable to uint8, the variable range must be between 0 and 255 and the colormap should also match this range.

Converting between different image encoding schemes can sometimes be done by scaling. To convert a grayscale image in uint8 or uint16 format to an indexed image, select an appropriate grayscale colormap from the MATLAB's established colormaps (or make up your own), then scale the image variable so the values lie within the range of colormap rows; that is, the

Biosignal and Medical Image Processing

data range should fall between 0 and M , where M is the number of rows in the colormap. MATLAB's colormaps have a default depth of 64, but this is easily modified. This approach is demonstrated in Example 12.3. However, an easier solution is simply to use MATLAB's `gray2ind` function listed below. This function, as with all the conversion functions, scales the input data appropriately, and also supplies an appropriate grayscale colormap, although alternate colormaps of the same depth can also be substituted. The routines that convert to indexed data are

```
[x, map] = gray2ind(I, M); % From grayscale to indexed  
[x, map] = rgb2ind(RGB, M or map); % From truecolor to indexed
```

where I is the input image and M is the length of the desired colormap. Both these routines accept data in any format, including logical, and produce an output of type `uint8` if the associated map length, M , is less than or equal to 64 or `uint16` if greater than 64 (M must range from 1 to 65,536). For `gray2ind`, the colormap is `gray`* with a depth of M , or the default value of 64 if M is omitted. For RGB conversion using `rgb2ind`, a colormap of M levels is generated to best match the RGB data. Alternatively, if a colormap is provided as the second argument, `rgb2ind` works the other way to generate an indexed array with values that best match the colors given in `map`. Since conversion from truecolor to indexed will always involve some compromise, the `rgb2ind` function has a number of options that alter the image conversion, options that allow trade-offs between color accuracy and image resolution. (See the associated help file.)

An alternative method for converting a grayscale image to indexed values that can be used to generate interesting pseudocolor images is the routine `grayslice` that converts using thresholding:

```
x = grayslice(I, M or V); % Grayscale to indexed  
% using thresholding
```

where any input format is acceptable. This function *slices* the image into M levels using an equal-step thresholding process. Each slice is then assigned a specific level on whatever colormap is selected. This process allows quite flexible color representations of grayscale images as demonstrated in Example 12.4. If the second argument is a vector, V , then it contains the threshold levels (which can now be unequal), and the number of slices corresponds to the length of this vector. The output format is either `uint8` or `uint16` depending on the number of slices, similar to the two conversion routines above.

Two conversion routines convert from indexed images to other encoding schemes:

```
I = ind2gray(x, map); % Convert to intensity  
RGB = ind2rgb(x, map); % Convert to RGB
```

Both functions accept any data format: `ind2gray` produces outputs in the same format, while `ind2rgb` produces outputs formatted as double. Function `ind2gray` removes the hue and saturation information while retaining the luminance, while function `ind2rgb` produces a truecolor RGB variable.

To convert an image to binary coding use

```
BW = im2bw(I, Level); % convert to Binary logical
```

* The `gray` colormap is not your most exciting colormap if you are going to display the image in color, but you can replace it with the map of your choice at any time. Try `hot` or `jet` for more action.

Table 12.3 Summary of Conversion Routines

Class	Conversion Routines
Indexed	gray2ind ¹ , grayslice ¹ , rgb2ind ¹
Intensity	ind2gray ² , mat2gray ^{2,3} , rgb2gray ³
RGB	ind2rgb ²
Binary	im2bw ⁴ , dither ¹

where `Level` specifies the threshold that is used to determine if a pixel is coded as white (1) or black (0). The input image, `I`, can be intensity, RGB, or indexed,* and in any format (`uint8`, `uint16`, or `double`). If a format other than intensity is used, the image is first converted into grayscale. Thus, `Level` should always between 0 and 1 regardless of the range of the input image. While most functions output binary images in `uint8` format, `im2bw` outputs the image in logical format. In this format, the image values are either 0 or 1, but each element is the same size as the double format (8 bytes). This format can be used in standard MATLAB operations, but does use a great deal of memory for just a black-and-white image.

A final conversion routine does not really change the data class, but does scale the data and is very useful. This routine converts class `double` data to intensity data; that is, the variable's range is now scaled between 0 and 1:

```
I = mat2gray (A, [Amin Amax]); % Scale double to 0 to 1.0
```

where `A` is a matrix and the optional second term specifies the values of `A` to be scaled to zero (i.e., `Amin` is scaled to black) or 1 (i.e., `Amax` is scaled to white). Since a matrix is already in `double` format, this routine provides only scaling. Normally, we use this routine without the second argument, so the largest number in the matrix is scaled to 1.0 and the smallest (or most negative) to 0.0. While using this default scaling is common, it can be a problem if the image contains a few irrelevant pixels having large (or small) values. This can occur after certain image-processing operations due to border or edge effects. In such cases, other scaling may be imposed, usually determined empirically, to achieve a suitable range of image intensities.

The various conversion routines and related data formats are summarized in Table 12.3.

The superscripts shown in Table 12.3 indicate the *output format*: (1) `uint8` or `uint16` depending on the number of levels requested, (2) `double`, (3) no format change (output format equals input format), and (4) logical (size `double`).

12.2 Image Display

There are several options for displaying an image, but the most useful and easiest to use is the `imshow` function. The basic calling format of this routine is

```
imshow(I,arg); % Display image I
```

where `I` is the image array and `arg` is an argument that is optional, except in the case of indexed data where it specifies the colormap. This holds for all display functions when indexed data are involved. For intensity class image variables, `arg` can be a scalar, in which case it specifies the

* As with all conversion routines, and many other routines, when the input image is in indexed format, it must be followed by the colormap variable.

Biosignal and Medical Image Processing

number of levels to use in rendering the image or, if `arg` is a vector such as `[low high]`, its values are used to readjust the black-and-white range limits of a specific data format.* If the empty matrix, `[]`, is given as `arg`, the maximum and minimum values in array `I` are taken as the `low` and `high` values (in other words, an operation similar to that of `mat2gray` is performed prior to `display`). The `imshow` function has a number of other options that make it quite powerful. These options can be found with the `help` command.

There are two functions designed to display multiframe images. The function `montage` displays a set of images in a grid-like pattern as shown in Example 12.2.

```
montage(MFW); % Display image montage
```

The `montage` function can accept inputs in any format as long as they are multiframe variables. Multiframe variables can be displayed in sequence as a movie using the `imovie` and `movie` commands:

```
mov = imovie(MFW, map); % Generate movie variable  
movie(mov, n, fps); % Display movie n times
```

or

```
implay(mov); % Display movie using GUI
```

The `imovie` command requires the input variable be either an RGB or indexed variable. In the latter case, the variable must be followed by a colormap variable. Movies are displayed using `movie` or `implay`. The latter calls a GUI, which can play a variety of formats. The second argument of `movie` is optional and specifies the number of times to repeat the movie (the default is 1). The third argument, also optional, specifies the frames per second (`fps`) for display. The default is 12 fps. The default for `implay` is 20 fps; this can be changed by an optional argument or in the GUI. The `movie` or `implay` functions cannot be displayed in a textbook, but they are used in some problems in this chapter, and in several amusing examples at the end of Chapter 13. In some early versions of MATLAB, indexed images must be converted to `uint8` format before being used in `imovie` (i.e., `mov = im2uint8(movie)`).

Some basic features of the MATLAB Imaging Processing Toolbox are illustrated in Example 12.1.

EXAMPLE 12.1

Generate an image of a *sine wave grating* having a *spatial frequency* of 2 cycles/inch. A sine wave grating is a pattern that is constant in the vertical direction, but varies sinusoidally in the horizontal direction. It is used as a visual stimulus in experiments dealing with visual perception. Assume the figure will be 4 inches²; hence, the overall pattern will contain 4 cycles (i.e., 2 cycles/inch). Place the figure in a 400 by 400 pixels array (i.e., 100 pixels/inch) using `uint8` format.

Solution

As most reproductions have limited grayscale resolution, the `uint8` data format called for in this example is adequate. However, we need to use a standard MATLAB variable in double format to calculate the sine wave. To save on memory, we first generate a 400 by 1 image in double format,

* Recall the default minimum and maximum values for the three nonindexed classes were [0, 256] for `uint8`; [0, 65535] for `uint16`; and [0, 1] for double arrays.

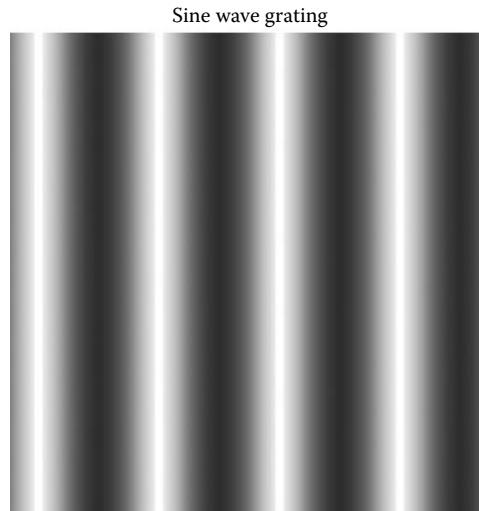


Figure 12.3 Sine wave grating produce by Example 12.1. Such images are used in research on the visual system. If the size of this image was 2 by 2 inches, the spatial frequency of the grating would be 2 cycles/inch.

then convert it to uint8 format using the conversion routine `im2uint8`. The `uint8` image can then be extended vertically to 400 pixels using a loop, but the MATLAB routine `repmat` is more efficient.

```
% Example 12.1 Generate a sine wave grating
%
N = 400; % Vertical and horizontal size
Nu_cyc = 4; % Produce 4 cycle grating
x = (1:N) *Ny_cyc/N; % Spatial vector
%
% Generate a single horizontal line of the image in
% a vector of 400 points
%
I_sin(1,:) = .5 * sin(2*pi*x) + .5; % Generate sin(0 to 1)
I_8 = im2uint8(I_sin); % Convert to a uint8
I = repmat(I_8,400,1); % Add 400 rows
%
imshow(I); % Display image
.....title.....
```

The output of this example is shown as Figure 12.3. As with all images shown in this text, there is a loss in both detail (resolution) and grayscale variation from reproduction. To get the best images, all figures can be reconstructed on screen using the code from the different examples provided in the accompanying downloadable files.

EXAMPLE 12.2

Generate a multiframe variable consisting of a series of sine wave gratings having different phases between 0.0 and 2π . Display these images as a montage. Border the images with black for separation on the montage plot. Generate 12 frames, but reduce the image to 100 by 100 to save memory.

Solution

The sine wave can be created as in Example 12.1, but with the addition of a loop to generate the 12 images and a phase term that varies between 0 and 2π . To set up the multiframe image, the fourth index must be varied between 1 and 12 and the third index set to 1.* Another complication is the need to add a black border around each image to improve the display (otherwise, the 12 images would run together). This can be done by first adding zeros before and after the sine wave then, after duplicating the row to fill in the vertical component, adding rows of zeros before the first and after the last row in the image.

```
% Example 12.2 Generate a multiframe array consisting
% of sine wave
%
N = 100; % Vertical and horizontal size
Nu_cyc = 2; % Produce 2 cycle grating
M = 12; % Produce 12 images
x = (1:N)*Nu_cyc/N; % Generate spatial vector
%
for j = 1:M % Generate M (12) images
    phase = 2*pi*(j-1)/(M-1); % Shift phase through 360 deg
    I_sin = .5 * sin(2*pi*x + phase) + .5; % Generate sine (0, 1)
    % Add some black space at left and right borders
    I_sin = [zeros(1,10) I_sin(1,:) zeros(1,10)];
    I_8 = im2uint8(I_sin); % Convert to a uint8 vector
    I_temp = repmat(I_8,100,1); % Add 100 rows
    % Add some black space at top and bottom borders
    I(:,:,:1,j) = [zeros(10,120); I_temp; zeros(10,120)];
end
montage(MFW); % Display image as montage
.....title.....
```

Result

The montage created by this example is shown in Figure 12.4. The multiframe data set, MFW, was constructed one frame at a time and each frame was placed in MFW using the frame index, the fourth index ($MFW(:,:,1,j)$). The zeros inserted at the beginning and end of the sine wave and at the top and bottom of the image provide dark bands between the images. Finally, the sine wave is phase-shifted through 360 degrees over the 12 frames.

EXAMPLE 12.3

Construct a multiframe variable with sine wave grating images as in Example 12.2, but display these images as a movie.

Solution

This example is the same as Example 12.2 except for a slight modification in image generation and the method of display. Since the images will play as a movie, we do not need the borders, so we can eliminate that code. Also, to produce a smoothly moving movie, we increase the number of images to 20 and modify the phase shift to avoid repeating the image at 2π . Since the `imovie` function requires the multiframe image variable to be in either RGB or

* Recall, the third index is reserved for referencing the color plane. For non-RGB variables, this index will always be 1. For images in RGB format, the third index will vary between 1 and 3.

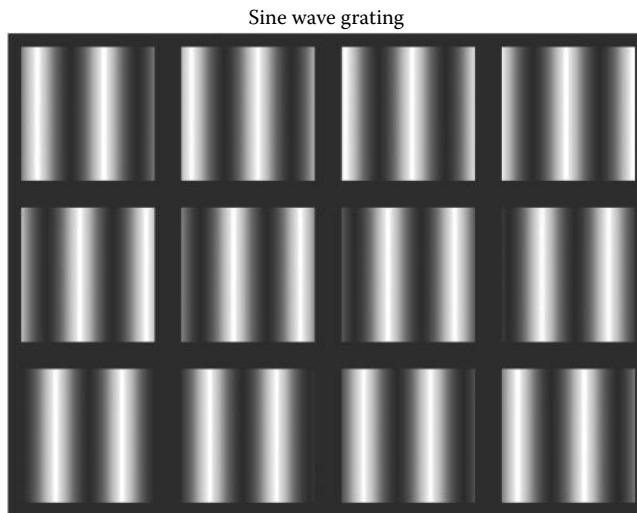


Figure 12.4 Montage of sine wave gratings produced by Example 12.2. Each image includes a border on all four sides to provide visual separation of the individual gratings.

indexed format, convert the uint8 data to indexed format. This can be done by the gray2ind function and the default range of 64 should be adequate. MATLAB colormaps can also be specified to be of any depth, but as with gray2ind, the default level is 64. We will display the movie using immmovie since this allows us to run the movie multiple times. We will substitute the colormap generated by gray2ind with one on MATLAB's more colorful maps, jet. (If you run this example on your computer, try some different colormaps such as winter, summer, hot, or hsv.)

```
% Example 12.3 Generate a movie of a multiframe array.
%
N = 100; % Vertical and horizontal points
Nu_cyc = 2; % Produce 4 cycle grating
M = 20; % Produce 20 images
x = (1:N)*Nu_cyc/N; % Generate spatial vector
for j = 1:M % Generate M (20) image
    % Generate sine; scale between 0 and 1
    phase = 2*pi*j/M; % Shift phase
    I_sin(1,:) = .5 * sin(2*pi*x + phase) + .5;
    I_8 = im2uint8(I_sin); % Convert to a uint8 vector
    Mf (:,:,1,j) = repmat(I_8,100,1); % Add 100 rows;
end
%
[Mf map] = gray2ind(Mf); % Convert to indexed
mov = immmovie(Mf,jet); % Make movie using jet colormap
movie(mov,4); % Play movie 4 times
```

Results

To appreciate this example, you need to run this program under MATLAB. The 20 frames are created using code similar to that in Example 12.3, except for modification noted above. Note that the size of the multiframe array, Mf, is $(100, 100, 1, 20)$ or $20 \times 10^4 \times 1$ bytes = 200,000 bytes.

12.3 Image Storage and Retrieval

Images may be stored on disk using the `imwrite` command:

```
imwrite(I, filename.ext, arg1, arg2,...); % Generate image file
```

where `I` is the array to be written into file `filename`. There are many file formats for storing image data, and MATLAB supports the most common formats. The file format is indicated by the filename's extension, `.ext`, which may be any number of popular image formats, including `.bmp` (Microsoft bitmap), `.gif` (graphic interchange format), `.jpeg` (joint photographic experts group), `.pcx` (Paintbrush), `.png` (portable network graphics), and `.tif` (tagged image file format). The arguments are optional and may be used to specify image compression or resolution, or other format-dependent information. The specifics can be found in the `imwrite` help file. The `imwrite` routine can be used to store any of the data formats or data classes mentioned above; however, if the image array, `I`, is an indexed array, then it must be followed by the colormap variable, `map`. Most image formats actually store `uint8`-formatted data, but the necessary conversions are done by the `imwrite`.

The `imread` function is used to retrieve images from the disk. It has the calling structure:

```
[I map] = imread('filename.ext', fmt or frame); % Read image file
```

where `filename` is the name of the image file and `.ext` is any of the extensions listed above. The optional second argument, `fmt`, only needs to be specified if the file format is not evident from the filename. The alternative optional argument `frame` is used to specify which frame of a multiframe image is to be read into `I`. The use of `imread` to read multiframe data is found in Example 12.4. As most file formats store images in `uint8` format, `I` will often be in that format. File formats `.tif` and `.png` support `uint16` format, so `imread` may generate data arrays in `uint16` format from these file types. The output class depends on the manner in which the data are stored in the file. If the file contains grayscale image data, then the output is encoded as an intensity image; if truecolor, then as RGB. For both these cases, the variable `map` will be empty, which can be checked with the `isempty(map)` command (see Example 12.4). If the file contains indexed data, then both `I` and `map` will contain data.

The type of data format used by a file can also be obtained by querying a graphics file using the function `inffinfo`.

```
information = inffinfo(filename.ext)
```

where `information` contains text providing the essential information about the file, including the `ColorType`, `FileSize`, and `BitDepth`. Alternatively, the image data and `map` can be loaded using `imread` and the image data determined from the MATLAB `whos` command. The `whos` command will also give the structure of the data variable (`uint8`, `uint16`, or `double`). With regard to data type, we use three approaches in the image-processing chapters: (1) assume a file contains a specific, known data type; (2) check the type of data contained in the image variable after the file is read and convert if needed; or (3) simply convert the data to the desired format using the appropriate routine from Table 12.3 knowing there is no problem even if the image is already in that format.

12.4 Basic Arithmetic Operations

If the image data are stored in the `double` format, then *all* MATLAB standard mathematical and operational procedures can be applied directly to the image variables. Most of the image-processing examples use `double` format variables and standard MATLAB commands. This is an easy

approach; however, the double format does require four times as much memory as the uint16 format and eight times as much memory as the uint8 format. To reduce the reliance on the double format, MATLAB supplies functions to carry out some basic mathematics on uint8 and uint16 format arrays. They also run faster than the standard MATLAB equivalents. These routines work on either format; they actually carry out the operations in double precision on an element-by-element basis then convert back to the input format. This reduces round-off and overflow errors. All of these commands work on a pixel-by-pixel basis over all the pixels in the image.

```
I_diff = imabsdiff(I, J);           % Absolute difference between J and I
I_comp = imcomplement(I);          % Complements image I
I_add = imadd(I, J);              % Adds image I and J
I_sub = imsubtract(I, J);          % Subtracts J from image I
I_divide = imdivide(I, J);         % Divides image I by J
I_multiply = immultiply(I, J);      % Multiply image I by J
```

In these commands J can also be a scalar double.

For the last four routines, J can be either another image variable or a constant in double format. Several arithmetical operations can be combined using the `imlincomb` function. The function essentially calculates a weighted sum of images. For example, to add 0.5 to the values in image I_1 , 0.3 to the values in image I_2 , and 0.75 to the values in image I_3 , use

```
I_combined = imlincomb(.5, I1, .3, I2, .75, I3);
% Linear combination of images
```

The arithmetic operations of multiplication and addition by constants are easy methods for increasing the contrast or brightness of an image. Some of these arithmetic operations are illustrated in Example 12.4.

EXAMPLE 12.4

Select an image from a multiframe MRI data set of brain images. Perform five different image-processing operations on the selected image: (1) increase the image contrast; (2) invert the image gray levels; (3) slice the image into five levels and display in different colors; (4) convert the image to BW using thresholding and invert (make black become white and white become black); and (5) lighten the center but not the sides. Also plot the original image.

Solution

First we read in the images in `mri.tif` using `imread`. The routine `imfinfo` shows there are 27 frames in this file, so we use a loop to read in all 27 frames and display them using `montage`. Since we do not know if the images are indexed, we read and display assuming that a map variable exists. We next ask for operator input to select a particular image slice. Since the image is stored in tif format, it could be in either uint8 or unit16 format. In fact, the source data format does not matter since we will convert to double using `im2double`. If we find it is an indexed image (i.e., map is not empty), we first convert it to grayscale using `ind2gray`. We then convert the image to double using `im2double`. (Since we want the image to be in double format, we apply `im2double` without regard to whether or not it is already in double format.)

The five operations are carried out using the image in double format. To increase the contrast, we multiply the image by 1.2 using `immultiply`. Since the image is in double format, we could just use the standard multiply operator “ $*$,” but `immultiply` is faster. Similarly, we invert the grayscale values using `imcomplement`. The `grayslice` routine divides the gray-scale levels into five values: 1 through 5. We can then treat this image as an indexed image with

Biosignal and Medical Image Processing

an associated five-level map. We use MATLAB's 'hot' map implemented with five rows; that is, `map = hot(5);`. For the threshold image, we use `im2bw` with a threshold of 0.75, then complement this image using the logical NOT: “~.” Finally, for the center highlighted image, we multiply the image in both the horizontal and vertical direction by a Hanning window. This is not a standard image-processing operation, but it does make for a highlighted center. A sine function would have worked just as well, but the Hamming window is convenient. After multiplying by the Hanning window, we use the `mat2gray` routine to ensure that the image intensities are properly scaled. Recall that if `mat2gray` is called without any arguments, it scales the array to range over the full intensity range (i.e., 0 to 1).

```
% Example 12.4 Example of image read and basic image processing
%
N_slice = 5;                                % Number of slices for sliced image
Level = .75;                                 % Threshold for binary image
%
for frame = 1:27                             % Read all frames into mri
    [mri(:,:,:,:frame), map] = imread('mri.tif', frame);
end
montage(mri, map);                          % Display images as a montage
                                              % Include map in case indexed image
frame_select = input('Select frame for processing: ');
I = mri(:,:,:,:frame_select);               % Select specific frame
if ~isempty(map)                            % Check if indexed image
    I = ind2gray(I, map);                  % If so, convert to intensity image
end
I1= im2double(I);                           % Convert to Class double
%
% Begin image processing
I_bright = immultiply(I1,1.2);             % Increase the contrast (#1)
I_invert = imcomplement(I1);                % Complement image (#2)
x_slice = grayslice(I1,N_slice);           % Slice in 5 levels (#3)
map1 = hot(N_slice);                       % Colorize sliced image
BW = im2bw(I1,Level);                     % Convert to binary (#4)
[M N] = size(I1);                         % Get image size
for i = 1:M                                % Multiple horizontally (#5)
    I_wind(i,:) = I1(i,:).*hanning(N);    % by Hanning window
end
for i = 1:N                                  % Multiply vertically
    I_wind(:,i) = I_wind(:,i).*hanning(M); % by Hanning window
end
I_window = mat2gray(I_window);              % Scale windowed image
subplot(3,2,1);                            % Display all images in one plot
imshow(I1);                                % Display original image
.....display other images with titles.....
```

Results

Using the `montage` function places all 27 frames in one figure (Figure 12.5). After the image is selected (in this case, frame 17), the various image-processing operations are performed and the results shown in Figure 12.6.

Many images that are grayscale can benefit from color coding. The next two examples work with pseudocolor images. In the next example, a grayscale image from the MRI image set is sliced into five levels using `grayslice`; each level is coded to a specific color using a user-developed colormap. The following example works with the RGB format to highlight specific features of a grayscale image by placing them in a specific color plane.

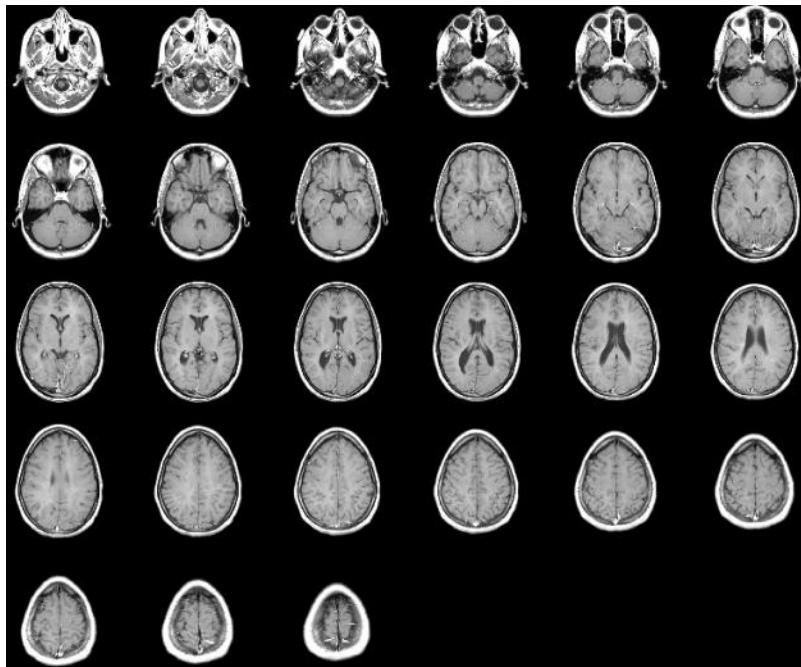


Figure 12.5 Montage display of the 27 frames in the image file `mri.tif`. One of these images (frame 17) was selected for further processing as shown in Figure 12.6.

EXAMPLE 12.5

Load MRI image frame number 16. Slice this image into five levels and plot the levels as black, blue, red, yellow, and white.

Solution

The image can be loaded as in Example 12.4, but only loading frame 16. The image is sliced into five levels using `grayslice`. However, to get the desired colors, it is necessary to design our own colormap. For this colormap, we need the colors black (0 0 0), blue (0 0 1), red (1 0 0), yellow (1 1 0), and white (1 1 1). This is easily achieved by defining a colormap as

```
map = [0 0 0; 0 0 1; 1 0 0; 1 1 0; 1 1 1];
```

The image is then displayed with `imshow` using this colormap.

```
% Example 12.5 Example of pseudocolor
%
[I(:,:,1),map] = imread('mri.tif', 16); % Read frame
if isempty(map) == 0 % If indexed data convert
    I = ind2gray(I,map); % to intensity image
end
I1= im2double(I); % Convert to double
%
x_slice = grayslice(I1,5); % Slice image: 5 levels
% Construct desired colormap
map = [0 0 0; 0 0 1; 1 0 0; 1 1 0; 1 1 1];
%
imshow(I1);
```

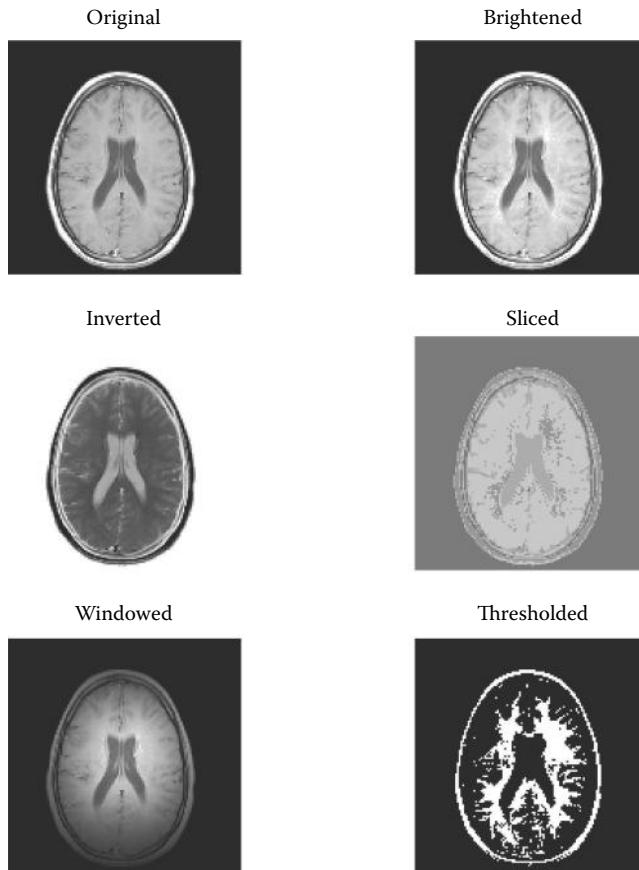


Figure 12.6 Images produced by Example 12.4. The sliced image is actually in color, but you need to run this example on your own computer to see this.

```
title('Original');
figure;
imshow(x_slice, map); % Display color slices
title('Sliced');
```

Results

Figure 12.7 shows the original and pseudocolor image. Unfortunately, to appreciate the colors in this image, it is necessary to run this example on your own computer, but the blues and reds complement each other nicely.

RGB images can be used to provide greater colorization control of a pseudocolor image. The next example expands different grayscale ranges into three different colors.

EXAMPLE 12.6

Generate an RGB image from an MR image using pseudocolor. Apply the pseudocolor scheme to frame 16 of the MRI images where grayscale values above 0.75 are coded in shades of yellow, values between 0.40 and 0.75 are coded in red, and values below 0.40 are coded in green. Each grayscale range should be expanded to cover the full *range of color intensity*. Finally, convert the black background to blue. To construct this image, use *masking* techniques and show the masks used.

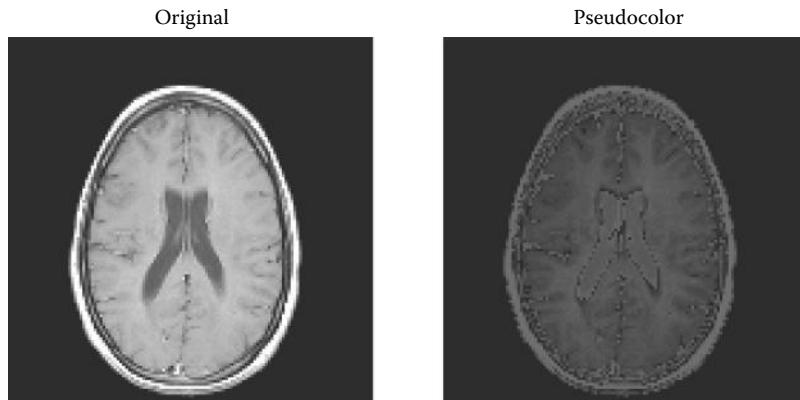


Figure 12.7 Pseudocolor image produced in Example 12.5 by slicing the image and applying a user-designed colormap. In color this make a dramatic image.

Solution

The image is loaded as in Example 12.5 and the same conversions are applied. There are a variety of approaches to achieving the desired image, but here we use *masks* as they are fundamental to image segmentation, an important tool in biomedical imaging discussed in Chapter 14. A mask is simply a BW image that is 1 in the area of interest and 0 everywhere else. If we multiply an image by a mask, only the area of interest remains, the rest of the image is blacked out. Here, we multiply our image by different masks to get the three color planes of an RGB image.

We use four masks in the problem. A “yellow” mask is generated by applying `im2bw` to the original image with a threshold of 0.75. This is just a BW image with 1 s where the pixel values of the original image exceeded 0.75 and 0 s everywhere else. For the “green” mask, we apply `im2bw` with a threshold of 0.4, then invert the mask (using the NOT operator) so the pixels below 0.4 in the image are 1 in the mask. The “red” mask will be constructed by inverting the “yellow” mask and combining it, using the AND operator, with the inverted “green” mask. The “blue” mask will identify 0.0 values in the original image by applying a very low threshold (0.0001) to the image, then inverting the mask.

Yellow is a combination of red and green, so the red plane will consist of the image multiplied by an OR combination of the red and yellow masks while the green plane will be an OR combination of the “yellow” and “green” masks. After multiplication, we use `mat2gray` to get the full range of each colorization as requested in the problem. The blue plane is just the blue mask since we just want a solid blue background.

Example 12.6 Using masks to construct a pseudocolor image
%

```

.....get frame 17 and convert as in Example 12.5
[M N] = size(I); % Size of image
RGB = zeros(M,N,3); % Initialize RGB array
BW_Y = im2bw(I,0.75); % Yellow mask (> 0.75)
BW_G = ~im2bw(I,.4); % Green mask (> 0.40 inverted)
BW_R = ~BW_G & ~BW_Y; % Red mask (> 0.4 and < 0.75)
BW_B = ~im2bw(I,0.0001); % Blue mask (> 0.0001 inverted) %
RGB(:,:,1) = mat2gray(I.*BW_R|BW_Y); % Red plane
RGB(:,:,2) = mat2gray(I.*BW_G|BW_Y); % Green plane
RGB(:,:,3) = BW_B; % Blue plane
.....display all masks, original, and
.....pseudocolor images, titles.....

```

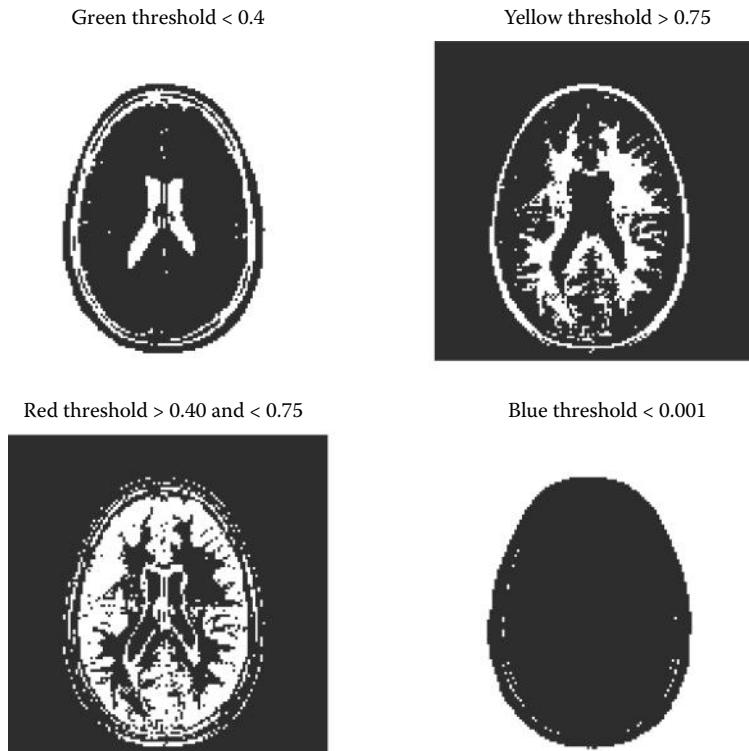


Figure 12.8 Masks used to generate pseudocolor image in Example 12.6. Each mask is white in the region that it identifies.

Results

The four masks are shown in Figure 12.8. Note that each mask is white in the area that it represents. For example, the “yellow” mask is white in the lightest regions of the original image: those above 0.75. The pseudocolor image produced by this code is shown in Figure 12.9. While using masks may not be the easiest way to construct this image, they are essential in image segmentation and other image-processing operations where we want to isolate specific regions of an image.

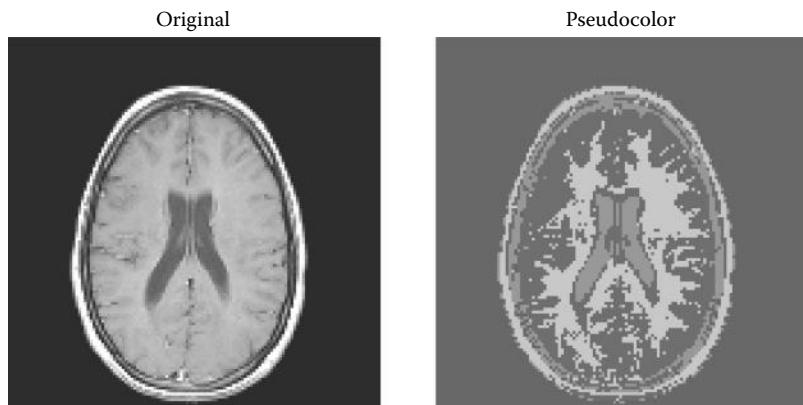


Figure 12.9 Pseudocolor image of the brain produced by Example 12.6. Again you need to run the example on your own computer to see the colors. The blue background is pleasing.

12.5 Block-Processing Operations

Many of the signal-processing techniques presented in the previous chapters operate on small, localized groups of data. For example, both FIR and adaptive filters use data samples within the same general neighborhood. Some image-processing techniques also operate on neighboring data elements, except now the neighborhood extends to two dimensions, horizontally and vertically. Many image-processing operations can be considered 2-D extensions of signal-processing operations we already know. For example, the next chapter discusses 2-D filters that are implemented with 2-D convolution and analyzed with the 2-D Fourier transform. While many image-processing operations are conceptually the same as those used in signal processing, the implementation is somewhat more involved due to the additional bookkeeping required to operate on data in two dimensions. The MATLAB Image Processing Toolbox simplifies much of the tedium of working in two dimensions by introducing functions that facilitate 2-D block, or neighborhood, operations.

The MATLAB Imaging Processing Toolbox facilitates local operations known as *block-processing operations*, which are divided into two categories: *sliding neighborhood* operations and *distinct block* operations. Most block-processing operations are sliding neighborhood operations where a block slides across the image as in convolution; however, the block must slide in both horizontal and vertical directions. 2-D convolution and the related 2-D correlation are examples of two very useful sliding neighborhood operations. In the less used distinct block operations, the image area is divided into a number of fixed groups of pixels, although these groups may overlap. This is analogous to the overlapping segments used in the Welch approach to the Fourier transform described in Chapter 3.

12.5.1 Sliding Neighborhood Operations

The sliding neighborhood operation alters one pixel at a time based on some operation performed on the surrounding pixels, pixels that lie within the neighborhood defined by the block. The block is placed as symmetrically as possible around the pixel being altered called the “center pixel,” although in some cases, this pixel will be slightly off center (Figure 12.10). The center pixel will only be in the center if the number of pixels in the block is odd in both dimensions; otherwise, the so-called center pixel position favors the left and upper sides of the block (Figure 12.10).*

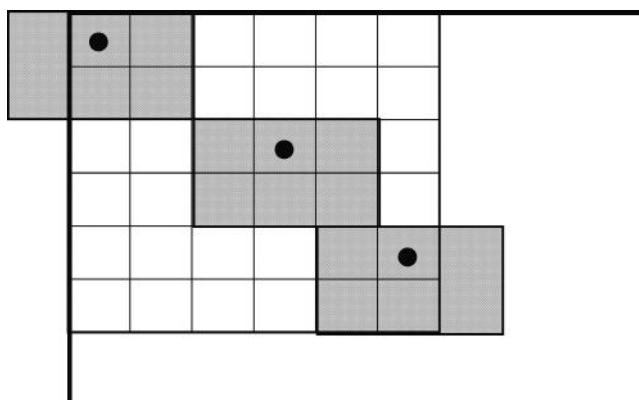


Figure 12.10 A 2 by 3 sliding neighborhood block ($M = 2$, $N = 3$). The block (shaded area) is shown in three different positions. Note that the block will sometimes fall off the picture and padding (usually zero padding) is required. In actual use, the block slides one element at a time over the entire image. The active or center pixel is indicated by the circle.

* In MATLAB notation, the center pixel of an M by N block is located at `floor(([M N] + 1)/2)`.

Biosignal and Medical Image Processing

Just as in signal processing, there is a problem that occurs at the edge of the image when a portion of the block extends beyond the image as in the upper left block of Figure 12.10. In this case, most MATLAB sliding block functions automatically perform zero padding for these pixels. Alternatively, the image filtering routine, `imfilter`, described in the next chapter offers some of the options, such as repeating the end point, or constant padding as discussed in Chapter 1.

The MATLAB routines `conv2` and `filter2` are both sliding neighborhood operators that are directly analogous to the 1-D convolution routine, `conv` and `filter`. These functions are described in the next chapter on image filtering. Other 2-D functions that are directly analogous to their 1-D counterparts include `mean2`, `std2`, `corr2`, and `fft2`. In this section, a general-purpose sliding neighborhood routine is described that can be used to implement a wide variety of image-processing operations. Since these operations can be, but are not necessarily, nonlinear, the function is named `nlfilter`, presumably standing for “nonlinear filter.” The calling structure is

```
I1 = nlfilter(I, [M N], func, P1, P2, ...);
```

where `I` is the input image array, `M` and `N` are the dimensions of the neighborhood block (vertical and horizontal, Figure 12.9), and `func` specifies the function that will operate over the block. The optional parameters `P1`, `P2`, ..., will be passed to the function if it requires input parameters. The function should take an `M` by `N` input and must produce a *scalar output* that will be used as the value of the center pixel. The input can be of any class or data format supported by the function, and the output image array, `I1`, will depend on the format provided by the routine’s output.

The function may be specified in one of three ways: as a string containing the desired operation, as a *function handle* to an M-file, or as a function established by the routine `inline`.^{*} The first approach is straightforward: simply embed the function operation, which can be any appropriate MATLAB statement(s), within single quotes. For example

```
I1 = nlfilter(I, [3 3], 'mean2');
```

This command will slide a 3 by 3 moving average across the image. Each pixel in the new image is the average of the 3 by 3 neighborhood in the original image. This produces a lowpass filtered version of the original image analogous to a 1-D FIR filter of $[1/3 \ 1/3 \ 1/3]$. This linear operation could be more effectively implemented using the filter routines described in the next chapter, but more complicated, perhaps nonlinear, operations could be called within the quotes. For example, if the quotes include the operation `std2`, a nonlinear operation is performed that takes the standard deviation of neighboring points. Of course, the quotes can also specify a user-defined routine as long as that routine has an appropriately sized matrix as input and produced a scalar output.

The use of a function handle is shown in the code

```
I1 = nlfilter(I, [3 3], @my_function);
```

where `@my_function` is the name of an M-file function. The function handle `@my_function` contains all the information required by MATLAB to execute the function. Again, this file should have a single scalar output produced by some operation on the `M` by `N` input array. The function can also take input arguments that follow `@my_function` as input arguments.

The `inline` routine has the ability to take string text and convert it into a function for use in `nlfilter` or any of the other MATLAB routines that have a function as an input argument.

```
F = inline('2*x(2,2) -sum( x(1:3,1))/3...
           -sum(x(1:3,3))/3 -x(1,2) -x(3,2)');
I1 = nlfilter(I, [3 3], F);
```

* A number of other MATLAB routines require functions as inputs and they are handled in a similar manner. For example, `quad`, `fminsearch`, and `fminbnd` are routines that require functions as inputs.

Function `inline` assumes that the input variable is `x`, but it also can find other variables based on the context, and it allows for additional arguments, `P1`, `P2`,... (see the associated help file). The particular function shown above takes the difference between the center point and its eight surrounding neighbors, performing a differentiator-like operation. There are better ways to perform spatial differentiation described in the next chapter, but this form is demonstrated as one of the operations in Example 12.7 below.

EXAMPLE 12.7

Load the image of blood cells in `blood.tif` found in MATLAB's image files. Convert the image to intensity class and double format. Perform the following sliding neighborhood operations: averaging over a 5 by 5 sliding block; differencing (spatial differentiation) using the function, `F`, above; and detecting a vertical boundary using a 2 by 3 vertical difference operator. This difference operator subtracts a vertical set of three left-hand pixels from the three adjacent right-hand pixels. The result is a brightening of vertical boundaries that go from dark to light and a darkening of vertical boundaries that go from light to dark. Display all the images in the same figure, including the original. Also include binary images of the vertical boundary image thresholded at two different levels to emphasize the left and right boundaries.

Solution

Load the image and convert as in Example 12.5. Apply the averaging operation using `nlfILTER` and calling MATLAB function `mean2`. For the differencing operation, use `nlfILTER` in conjunction with `inline` and the differencing function `F` given above. To detect vertical boundaries, again use `inline` to define a function: '`sum(x(1:3,2)) - sum(x(1:3,1))'` which takes the sum of three vertical pixels and subtracts the sum of the three vertical pixels immediately to the left. This makes left blood cell boundaries that go from light on the left to dark on the right darker, and right boundaries that go from dark/left to light/right brighter. We then apply `im2bw` to this boundary-enhanced image and empirically determine thresholds that best isolate the two boundaries. Finally, we invert the left (dark) boundary `BW` image so that both left and right cell boundaries appear white against a black background.

```
% Example 12.7 Demonstration of Sliding Neighborhood operations
%
[I map] = imread('blood1.tif');      % Input image
....convert image to intensity class as in Example 12.5..... %
% Perform the various sliding neighborhood operations.
%
I_avg = nlfILTER(I, [5 5], 'mean2'); % Running average
%
F = inline('x(2,2) - sum(x(1:3,1))..
            /3 - sum(x(1:3,3))/3 - x(1,2) - x(3,2)');
I_diff = nlfILTER(I, [3 3], F);      % Differencing
%
F1 = inline ('sum(x(1:3,2)) - sum(x(1:3,1))');
I_vertical = nlfILTER(I, [3 2], F1); % Vertical boundary detection
%
% Rescale all arrays
I_avg = mat2gray(I_avg);
I_diff = mat2gray(I_diff);
I_vertical = mat2gray(I_vertical);
%
BW_left = ~im2bw(I_vertical,.68);    % Identify-invert left boundaries
BW_right = im2bw(I_vertical,.8);     % Identify right boundaries
.....display and title images.....
```

Result

The code in Example 12.7 produces the images in Figure 12.11. These operations are quite time-consuming: Example 12.7 took around 45 s to run on a current laptop computer. Techniques for increasing the speed of sliding operations can be found in the help file for `colfilt`, but it is always faster to use linear filter routines such as `conv2` and `imfilter` when possible.

The averaging has improved contrast, but the resolution is reduced so that edges are no longer distinct. The “differentiated” image has improved cell boundaries, but some of the gray-level detail is lost. The vertical boundaries produced by the 3 by 2 sliding block are not very apparent in the intensity image (vertical boundaries), but become quite evident in the thresholded binary images. The intensity image, `I_vertical`, is thresholded at two different levels to produce the left and right boundaries. In addition, the thresholded left image has been inverted using the `~` operator before display to produce a white boundary.

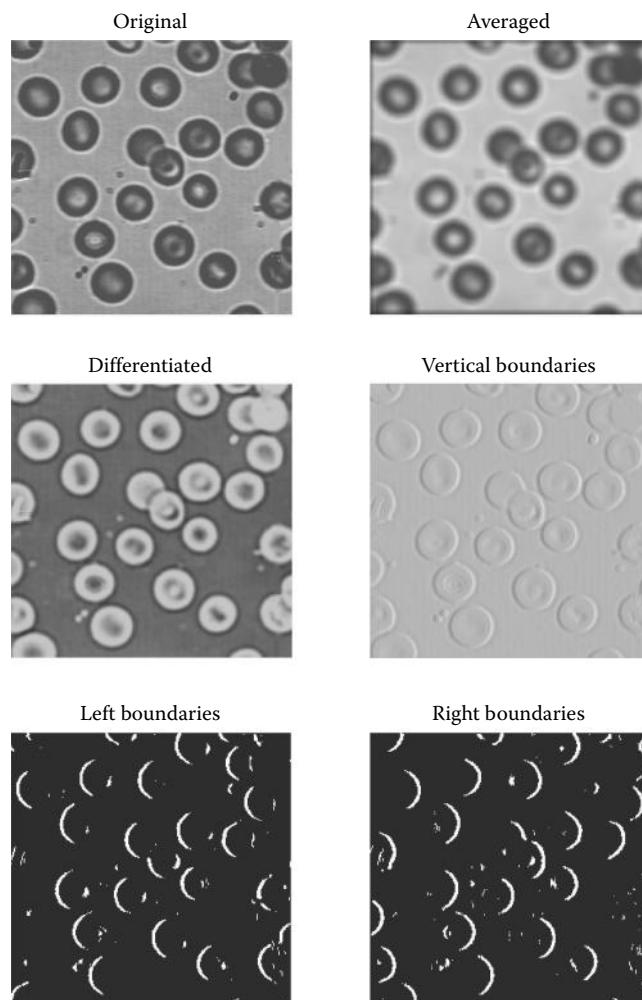


Figure 12.11 A variety of sliding neighborhood operations carried out on an image of blood cells. (MATLAB blood.tif, used with permission.) Note the improved definition of the cells with the differentiated image.

12.5.2 Distinct Block Operations

All of the sliding neighborhood options can also be implemented using configurations of fixed blocks (Figure 12.12). These blocks do not slide, but are fixed with respect to the image (although they may overlap), so they produce very different results. The MATLAB function for implementing *distinct block* operations is similar in format to the sliding neighborhood function:

```
I1 = blkproc(I, [M N], [Vo, Ho], func);
```

Where M and N specify the vertical and horizontal size of the block, Vo and Ho are optional arguments that specify the vertical and horizontal overlap of the block, func is the function that operates on the block, I is the input array, and I1 is the output array.* As with nlfilter, the data format of the output depends on the output of the function. The function is specified in the same manner as in nlfilter; however, the function output will be different.

Function outputs for sliding neighborhood operations have to be scalars, which became the value of the center pixel. In distinct block operations, the block does not move, so the function output normally produces values for every pixel in the block. If the function is an operation that normally produces only a single value, its output can be expanded by multiplying it with an array of ones that is the same size as the block. This will place the single output value in every pixel in the block. For example

```
I1 = blkproc(I [4 5], 'std2 * ones(4,5)');
```

places the output of the MATLAB function std2 into a 4 by 5 array, which then becomes the output of the function. It is also possible to use the inline function to describe the function:

```
F = inline('std2(x) * ones(size(x))');
I1 = blkproc(I, [4 5], F);
```

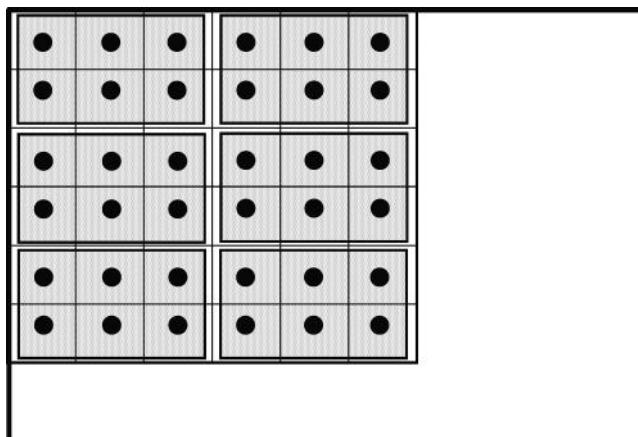


Figure 12.12 A 2 by 3 pixel distinct block. As with the sliding neighborhood block, these fixed blocks can fall off the picture and require padding (usually zero padding). The center pixel has no special significance in this approach.

* MATLAB now recommends using a similar routine, blockproc instead of blkproc. The calling arguments are the same except that Vo and Ho are not used. The difference is in how func is specified. In blockproc, func is a block struct and the data and block size (M,N) is passed to the function as elements of a structure. This means the user function, func, does not need to specify a fixed block size. The routine also features a number of useful options. See the associated help file for details.

Biosignal and Medical Image Processing

Of course, it is possible that certain operations could produce a different output for each pixel in the block. An example of block processing is given in Example 12.8.

EXAMPLE 12.8

Load the blood cell image used in Example 12.7 and perform the following distinct block-processing operations: (1) display the average for a block size of 8 by 8; (2) for a 3 by 3 block, perform the differentiator operation used in Example 12.7; and (3) apply the vertical boundary detector from Example 12.7 to a 3 by 3 block. Display all the images, including the original, in a single figure.

Solution

Load and convert the image as in previous examples. Then use the same code to generate the desired functions, but use `blkproc` instead of `nlfILTER` to create the various images.

```
% Example 12.8 Distinct Block operations
..... Image load, same as in Example 12.7.....
%
% Perform the various Distinct Block operations.
% Average of the image
I_avg = blkproc(I, [10 10], 'mean2 * ones(10,10)');
%
% Differentiation - place result in all blocks
F = inline('(x(2,2) - sum(x(1:3,1))/3 - ...
    sum(x(1:3,3))/3 - x(1,2) - x(3,2)) * ones(size(x))');
I_diff = blkproc(I, [3 3], F);
%
% Vertical edge detector -
F1 = inline ('(sum(x(1:3,2)) - sum(x(1:3,1)))...
    * ones(size(x))');
I_vertical = blkproc(I, [3,2], F1);
.....Rescale and plotting as in Example 12.7.....
```

Figure 12.13 shows the images produced by Example 12.8. The “differentiator” and edge detection operators look similar to those produced by the sliding neighborhood operation because they operate on fairly small block sizes. The averaging operator shows images that appear to have large pixels since the neighborhood average is placed in block of 8 by 8 pixels and the output of `blkproc` is the same for all eight pixels.

12.6 Summary

This chapter presents the basics of MATLAB’s Image Processing Toolbox, but many of the concepts would apply to any image-processing system. MATLAB uses two different coordinate systems to reference pixel locations. The most common is the pixel coordinate system, which indexes image pixels in the same manner as matrix elements: M by N , where M , the first index, specifies vertical location from top to bottom and N , the second index, specifies horizontal location from left to right. In the spatial coordinate system, the first index, x , specifies horizontal location left to right and the second index, y , specifies vertical location, again top to bottom. Moreover, spatial coordinate indexes can be fractional, so a pixel has a range between 0.5 and 1.5 times the actual pixel index.

Images can be coded into four different classes: grayscale, black and white (BW), indexed, and RGB (red, green, blue). The latter two are used to encode color. The RGB format is straightforward; the color image is stored as three grayscale images representing the red, green, or blue

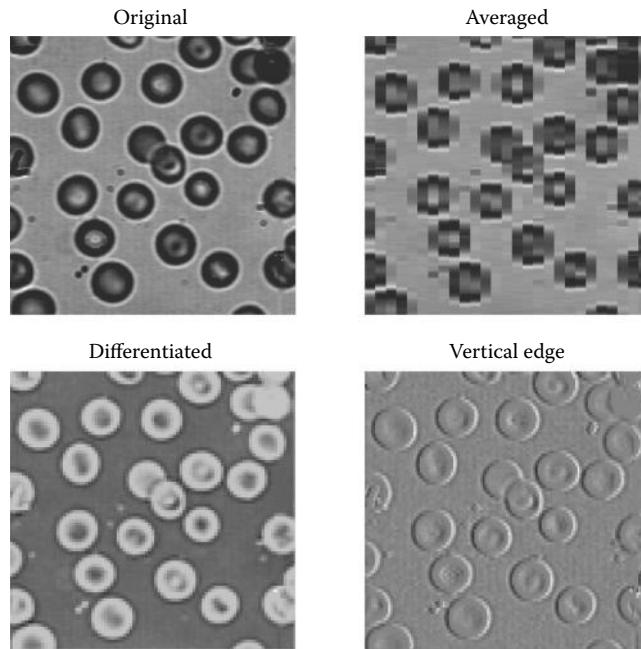


Figure 12.13 The blood cell image of Example 12.8 processed using three distinct block operations: block averaging, block differentiation, and block vertical edge detection.

content of the image. The three “color planes” are identified as the third index of a 3-D variable. Hence, an RGB image requires three times the number of pixels as other images, but allows for individual manipulation of each color plane. In indexed imaging, the color image is stored as a 2-D array, where each element points to a position in a colormap. The color map encodes the percentage of red, green, and blue associated with a given image value. Indexed imaging is useful for colorizing grayscale images. The color scheme used to encode these pseudocolor images can be easily modified by changing the associated colormap. Grayscale images consist of pixels whose values define image intensity with higher values corresponding to lighter pixels. The final image class, black and white, has only two grayscale values: black or white (as the image class name implies). These images are actually logical arrays and can be combined using standard logical operation such as AND ($\&$), OR ($|$), or NOT (\sim). They are particularly useful as masks. When an image is multiplied by a mask, only the regions corresponding to 1 in the mask will remain, the rest of the image will be blank (i.e., set to black). This is an effective way to isolate features in an image such as potential lesions or tumors.

Because of the large storage requirements of images, more concise data storage formats must be considered. In fact, it is common to store images using 1 byte per pixel using the uint8 format. Alternatively, the uint16 format requires 2 bytes per pixel. Although a few basic routines exist for operating on data in uint8 or uint16 format, conversion to double format is common so that all the standard MATLAB operations can be applied to the image. Unfortunately, this format requires 8 bytes per pixel. Routines for converting between the various data formats are available.

When images fall into groups, such as cross-sectional MRI slices or frames of a movie, MATLAB provides a multiframe format that allows these images to be referenced as a single variable. A specific image or frame is specified as the fourth index of the variable. (Note that if the image is not RGB, the third index will always be 1, but it must still be specified.) Multiframe images can be displayed as montages or as a movie.

Biosignal and Medical Image Processing

Routines to read and write images to data files are available and these support all the image classes and all data formats. A range of popular image storage formats can be read or written, including jpg, tif, png, gif, bmp, and pcx. Some restrictions apply regarding image class and storage format, which are described in the help files of `imwrite` and `imread`.

Two image-processing operations were introduced in this chapter, which facilitate neighborhood operations in which a new image is created based on calculations using neighboring pixels. A very general filter can be used to alter the sharpness or reduce noise in an image. More efficient linear filters are described in the next chapter, but the approach described here can be used to implement nonlinear neighborhood operations. As we will see in Chapter 14, nonlinear operations can help to identify textural patterns in an image.

In subsequent chapters, the fundamentals introduced in this chapter will be used to develop a variety of useful image-processing techniques such as filtering, Fourier and other transformations, segmentation (isolating regions of interest), and image registration (alignment of multiple images).

PROBLEMS

- 12.1 (a) Following the approach used in Example 12.1, generate an image that is a sinusoidal grating in both horizontal and vertical directions (it will look somewhat like a checkerboard). [Hint: This can be done with *only one* additional instruction.] (b) Combine this image with its inverse as a multiframe image and show it as a movie. Use multiple repetitions. The movie should look like a flickering checkerboard. (You must convert the images to indexed and be sure to pass the map to `imovie` that was generated by `gray2ind` or a map of your choosing.) Submit the two static images.
- 12.2 Load the x-ray image of the spine (`spine.tif`) from the MATLAB Image Processing Toolbox. Slice the image into four different levels, then plot in pseudocolor using red, green, blue, and yellow for each slice. The 0-level slice should be blue, the next highest green, the next red, and the highest level slice should be yellow. Use `grayscale` and *construct your own colormap*. Plot the original and sliced image in the same figure. (If the “original” image also displays in pseudocolor, it is because the computer display is using the same four-level colormap for both images. In this case, you should convert the sliced image to RGB before displaying.)
- 12.3 Remake the movie in Example 12.3, but make the bars move from top to bottom. Also make the movie more colorful by using a different colormap such as `spring`, or `summer` in the second argument of `imovie`.
- 12.4 Load frame 20 from the MRI image (`mri.tif`) and code it in pseudocolor using an RGB image. Put the unaltered image in the green color plane and the inverted image in the blue color plane. Then make a mask of all values over 0.9 and apply it to the original image using the AND operator, and place the masked image in the red color plane.
- 12.5 Load the image of a cancer cell (from rat prostate, courtesy of Alan W. Partin, MD, Johns Hopkins University School of Medicine) `cell.tif` and apply a sine correction to the intensity values of the image. Specifically, modify each pixel in the image by a function that is a quarter-wave sine wave (see Figure P12.5). That is, the corrected pixels are the output of the sine function of the input pixels: $I_{out} = \sin(I_{in} \pi/2)$. Remember that pixel values must range between 0 and 1.
- 12.6 Load the mask in file `b_mask.tif` and the image `blood1.tif` used in Examples 12.7 and 12.8. The mask contains a binary image where the ones indicate the

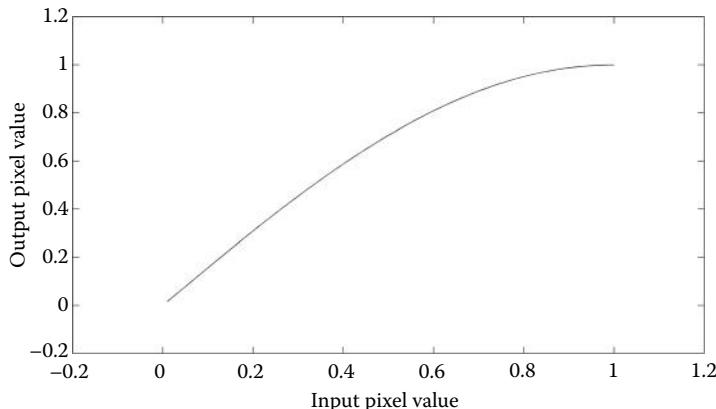


Figure P12.5 A plot of the gamma correction used in Problem 12.5.

placement and shape of the blood cells in `blood.tif` and the zeros represent the background. This mask was developed from a segmentation operation on the cell image as covered in Chapter 14. Use the mask to isolate the cells and, using an RGB image, plot them in shades of yellow (the intensity values inside the cells encoded as shades of yellow), with the background as a solid blue.

- 12.7 Load the RGB image in file `peppers.png`. Generate a new image where the red peppers are green and the green peppers are red. Display the original and transformed images side by side.
- 12.8 Load the blood cell image in `blood1.tif`. Write a sliding neighborhood function to enhance *horizontal* boundaries that go from dark to light. Write a second function that enhances boundaries that go from light to dark. Threshold both images so as to enhance the boundaries. Use a 3 by 2 sliding block. [Hint: This program may require several minutes to run. You do *not* need to rerun the program each time to adjust the threshold for the two binary images. This can be done outside the program from the MATLAB command line.]
- 12.9 Load the file `blood2.tif`, which contains an image of the blood cells used in Examples 12.6 and 12.7, but contaminated by *salt and pepper* noise. Apply a 4 by 4 pixel moving average filter using `nlfilter` with the `mean2` operator. Note how effectively this simple averaging filter reduces the noise in the image. Display the original and filtered image on the same plot.
- 12.10 Load the file `bacteria.tif` and apply two sliding neighborhood operations followed by a threshold operation. The first filter should take the standard deviation of a 3 by 3 area of surrounding pixels using `nlfilter` with the `std2` operation. This gives an image that highlights variations between the pixels. The next filter should apply the 3 by 3 differentiator used in Example 12.7 to the standard deviation image. This will produce ghostly outlines of the bacteria. These outlines can be thresholded with `im2bw` to generate an image that outlines the cells. Again the threshold values can be determined from the filtered images without rerunning the program. Plot the original, the two filtered, and the thresholded images on one page.
- 12.11 Load the file `testpat3.tif` that contains a test pattern image of 256 by 256 pixels. Use a mask to isolate the center 128 by 128 pixels as well as the surrounding pixels.

Biosignal and Medical Image Processing

Apply the differentiator filter used in Example 12.7 to the surround section and the averaging filter, also from Example 12.7, to the center, combine the two images, and display along with the original image. [Note that the derivative operation will cause the black center to become white, so before combining the images, it is necessary to blacken out the center again using the mask (or its inverse, depending on the logic you used to create the mask).]

- 12.12 Load the blood cells in `blood1.tif`. Apply a distinct block function that replaces all of the values within a block by the maximum value in that block. Use a 4 by 4 block size. Repeat the operation using a function that replaces all the values by the minimum value in the block. (This is one of those problems for illustrative purposes only. Admittedly, the value of the transformed images is not obvious.)

13

Image Processing *Filters, Transformations, and Registration*

13.1 Two-Dimensional Fourier Transform

The Fourier transform and the efficient algorithm for computing it, the fast Fourier transform, extend in a straightforward manner to two (or more) dimensions. The 2-D version of the Fourier transform can be applied to images providing a spectral analysis of the image content. Of course, the resulting spectrum will be in two dimensions and it is more difficult to interpret than a 1-D spectrum. Nonetheless, it can be a useful analysis tool, both for describing the contents of an image and as an aid in the construction of imaging filters as described in the next section.

When applied to images, the spatial directions are equivalent to the time variable in the 1-D Fourier transform, and the analogous spatial frequency is given in terms of cycles/unit length (i.e., cycles/cm or cycles/inch) or normalized to cycles per sample. Many of the concerns raised with sampled time data apply to sampled spatial data. For example, undersampling an image will lead to aliasing. In such cases, the spatial frequency content of the original image is greater than $f_s/2$, where f_s now is 1/(pixel size). Figure 13.1 shows an example of aliasing in the spatial domain. The upper left-hand image contains a chirp signal of increasing in spatial frequency from left to right. The high-frequency elements on the right side of this image are adequately sampled in the left-hand image. The same pattern is shown in the upper right-hand image except that the sampling frequency has been reduced by a factor of 6. The low-frequency variations in intensity are unchanged, but the high-frequency variations have additional frequencies mixed in as the aliasing folds in the frequencies above $f_s/2$ (see Section 1.6.2). The lower figures show the influence of aliasing on a diagonal pattern. The jagged diagonals are characteristic of aliasing as are moiré patterns seen in other images. The problem of determining an appropriate sampling size is even more acute in image acquisition since oversampling can quickly lead to excessive memory storage requirements.

The 2-D Fourier transform in continuous form is a direct extension of the equation given in Chapter 3:

$$F(\omega_1 \omega_2) = \int_{m=-\infty}^{\infty} \int_{n=-\infty}^{\infty} f(x, y) e^{-j\omega_1 m} e^{-j\omega_2 n} dx dy \quad (13.1)$$

The variables ω_1 and ω_2 are still frequency variables, although their units are in radians per sample. As with the time-domain spectrum, $F(\omega_1, \omega_2)$ is a complex-valued function that is

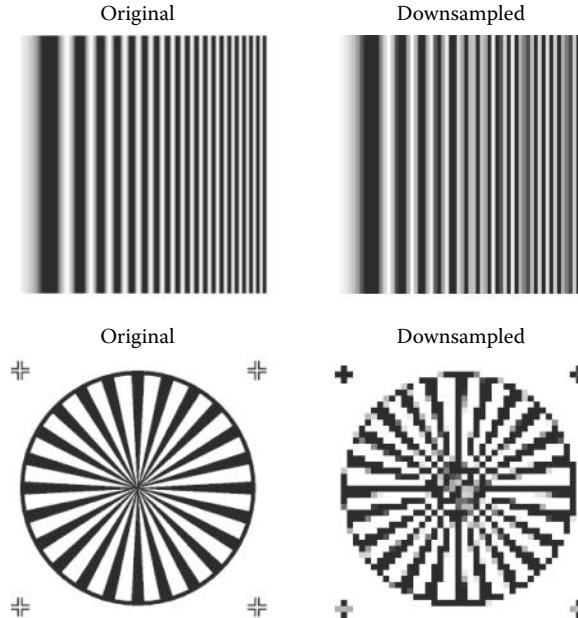


Figure 13.1 The influence of aliasing due to undersampling on two images with high spatial frequencies. Aliasing in these images manifests as additional sinusoidal frequencies in the upper right image and jagged diagonals in the lower right image.

periodic in both ω_1 and ω_2 . Usually, only a single period of the spectral function is displayed as was the case with the time-domain analog.

The inverse 2-D Fourier transform is defined as

$$f(x,y) = \frac{1}{4p^2} \int_{\omega_1=-p}^p \int_{\omega_2=-p}^p F(\omega_1\omega_2) e^{-j\omega_1 x} e^{-j\omega_2 y} d\omega_1 d\omega_2 \quad (13.2)$$

As with the time-domain equivalent, this statement is a reflection of the fact that any 2-D function can be represented by a series (possibly infinite) of sinusoids, but now the sinusoids extend over the two dimensions.

The discrete form of Equations 13.1 and 13.2 is again similar to their time-domain analogs. For an image size of M by N , the discrete Fourier transform becomes

$$F(p,q) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) e^{-j(2pm/M)} e^{-j(2qn/N)} \quad (13.3)$$

$$p = 0, 1, \dots, M-1; \quad q = 0, 1, \dots, N-1$$

The values $F(p,q)$ are the Fourier transform coefficients of $f(m,n)$. The discrete form of the inverse Fourier transform becomes

$$f(m,n) = \frac{1}{MN} \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} F(p,q) e^{-j(2pm/M)} e^{-j(2qn/N)} \quad (13.4)$$

$$m = 0, 1, \dots, M-1; \quad n = 0, 1, \dots, N-1$$

13.1.1 MATLAB Implementation

Both the Fourier transform and the inverse Fourier transform are supported in two dimensions by MATLAB functions. The 2-D Fourier transform is evoked as

```
F = fft2(x,M,N); % Two dimensional Fourier transform
```

where F is the output matrix and x is the input matrix. M and N are optional arguments that specify padding for the vertical and horizontal dimensions, respectively, just as in the 1-D `fft`. In the time domain, the frequency spectrum of simple waveforms can often be anticipated and the spectra of even relatively complicated waveforms can usually be understood. With two dimensions, it becomes more difficult to visualize the expected Fourier transform even of fairly simple images. In Example 13.1, a simple thin rectangular bar is constructed and the Fourier transform of the object is determined. The resultant spatial frequency function is plotted both as a 3-D function and as an intensity image.

EXAMPLE 13.1

Determine and display the 2-D Fourier transform of a thin rectangular object. The image should be 22 by 30 pixels in size and consist of a 3 by 11 pixel solid white bar against a black background. Display the Fourier transform both as a function (i.e., as a mesh plot) and as an image plot.

```
% Example 13.1 Two-dimensional Fourier transform of a
% simple object.
%
close all; clear all;
% Construct the rectangular object
I = zeros(22,30); % Original figure is small
I(10:12,10:20) = 1; % but will be padded
F = fft2(I,128,128); % Take FT; pad to 128 by 128
F = abs(fftshift(F)); % Shift center; get magnitude
%
imshow(I); % Plot object
.....labels.....
figure;
mesh(F); % Plot Fourier transform
.....labels.....
figure;
F = log(F); % Take log function
I = mat2gray(F); % Scale as intensity image
imshow(I); % Plot FT as image
```

Results

Note that in the above program, the image size was kept small (22 by 30) since the image will be padded with zeros by `fft2`. The `fft2` routine places the DC component in the upper left corner. The `fftshift` routine is used to shift this component to the center of the image as the resulting plot is easier to interpret. The log of the function was taken before plotting as an image to improve the grayscale quality in the figure. Figures 13.2 and 13.3 show the results of this example.

The horizontal chirp image shown in Figure 13.1 also produces an easily interpretable Fourier transform as shown in Figure 13.4. The fact that this image changes in only one direction, the horizontal direction, is reflected in the Fourier transform. The linear increase in spatial frequency in the horizontal direction produces an approximately constant spectral curve in that direction.

The 2-D Fourier transform is also useful in the construction and evaluation of linear filters as described in the following section.

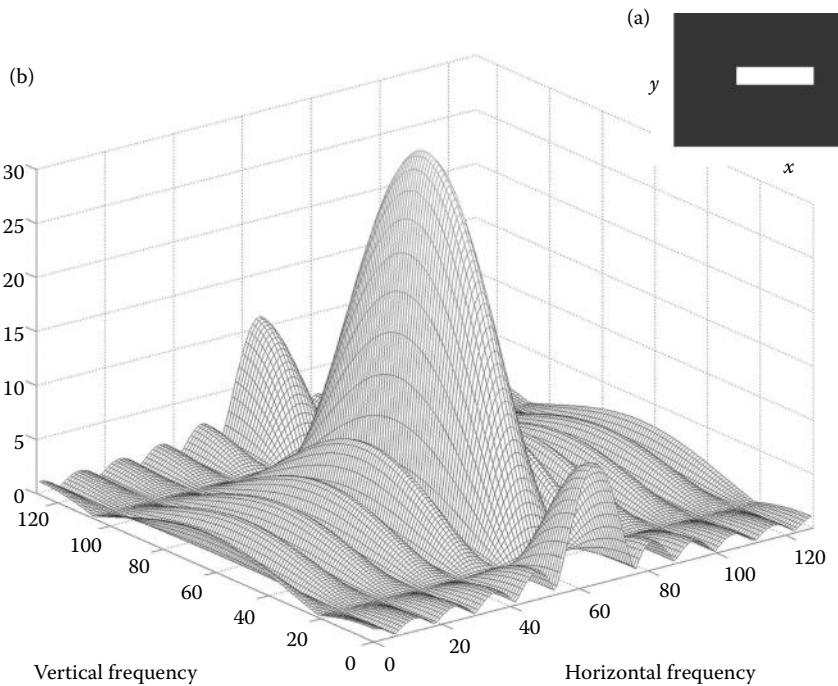


Figure 13.2 (a) The rectangular object (3 pixels by 11 pixels) used in Example 13.1. (b) The magnitude Fourier transform of this image is shown here and in Figure 13.3. More energy is seen, particularly at higher frequencies, along the vertical axis because the object's vertical cross section appears as a narrow pulse. The border horizontal cross section produces frequency characteristics that fall off rapidly at higher frequencies.

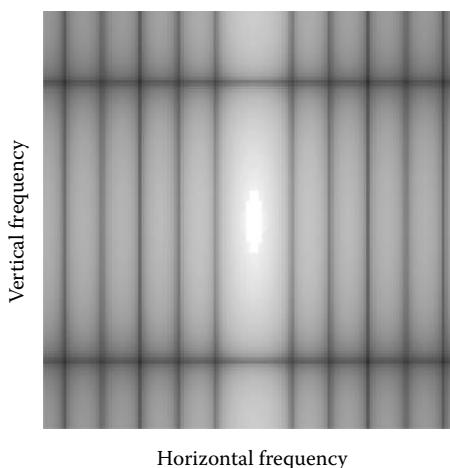


Figure 13.3 The magnitude Fourier transform of the image in Figure 13.2a shown as a grayscale image. The log of the function was taken before plotting to improve the details. Again, more high-frequency energy is seen in the vertical direction as indicated by the dark vertical band.

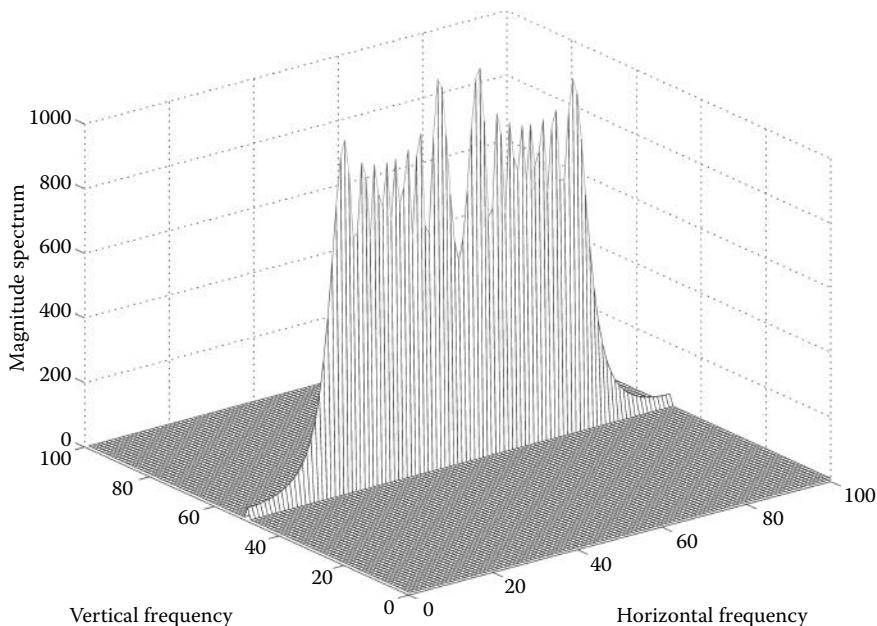


Figure 13.4 Magnitude spectrum of the horizontal chirp signal shown in Figure 13.1. The spatial frequency characteristics of this image are zero in the vertical direction since the image is constant in this direction. The linear increase in spatial frequency in the horizontal direction is reflected in the more-or-less constant amplitude of the spectrum in this direction.

13.2 Linear Filtering

The techniques of linear filtering described in Chapter 4 can be directly extended to two dimensions and applied to images. In image processing, FIR filters are usually used because of their linear phase characteristics. Filtering an image is a local or neighborhood operation just as it was in signal filtering, although in this case the neighborhood extends in two directions around a given pixel. In image filtering, the value of a filtered pixel is determined from a linear combination of surrounding pixels. For the FIR filters described in Chapter 4, the linear combination for a given FIR filter was specified by the impulse response function, the filter coefficients, $b[k]$. In image filtering, the filter function exists in two dimensions, $b[m,n]$. These 2-D filter weights are applied to the image using 2-D convolution (or correlation) in an approach analogous to 1-D filtering.

The equation for 2-D convolution is a straightforward extension of the 1-D form (Equation 2.55):

$$y[m,n] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} x[k_1,k_2] b[m - k_1, n - k_2] \quad (13.5)$$

While this equation would not be difficult to implement using MATLAB statements, MATLAB has a function that implements 2-D convolution directly.

Using convolution to perform image filtering parallels its use in signal processing: the image array is convolved with a set of filter coefficients. However, in image analysis, the filter coefficients are defined in two dimensions, $b[m,n]$. A classic example of a digital image filter is the *Sobel filter*, a set of coefficients that perform a horizontal spatial derivative operation

Biosignal and Medical Image Processing

for enhancement of horizontal edges (or vertical edges if the coefficients are rotated using transposition):

$$b[m,n]_{\text{Sobel}} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (13.6)$$

These 2-D filter coefficients are sometimes referred to as the *convolution kernel*. The application of a Sobel filter to an image is seen in Example 13.2.

When convolution is used to apply a series of weights to either image or signal data, the weights represent a 2-D impulse response and, as with a 1-D impulse response, the weights are applied to the data in *reverse order* as indicated by the negative sign in the 1-D and 2-D convolution equations (Equation 2.55 in Chapter 2 and Equation 13.5).^{*} This can be a mild source of confusion in 1-D applications and is even more confusing in 2-D applications. It becomes quite difficult to conceptualize how a 2-D filter, such as given in Equation 13.6, will affect an image. Image filtering is easier to visualize if the weights are applied directly to the image data in the same orientation. This is possible if the digital filtering is implemented using 2-D *correlation* rather than convolution. Image filtering using correlation is still a sliding neighborhood operation, where the value of the center pixel is just the weighted sum of neighboring pixels with the weighting given by the filter coefficients. When correlation is used, the set of weighting coefficients is termed the *correlation kernel* to distinguish it from the standard filter coefficients. In fact, the operations of correlation and convolution both involve weighted sums of neighboring pixels, and the only difference between correlation kernels and convolution kernels is a 180° rotation of the coefficient matrix. MATLAB filter routines use correlation kernels because their operation is easier to conceptualize.

13.2.1 MATLAB Implementation

2-D convolution is implemented using the routine `conv2`:

```
I2 = conv2(I1, b, shape);
```

where `I1` and `b` are image and filter coefficients (or two images, or simply two matrices) to be convolved and `shape` is an optional argument that controls the size of the output image. If `shape` is '`full`', the default, then the size of the output matrix follows the same rules as in 1-D convolution: each dimension of the output is the sum of the two matrix lengths along that dimension minus one. Hence, if the two matrices have sizes `I1(M1, N1)` and `b(M2, N2)`, the output size is `I2(M1 + M2 - 1, N2 + N2 - 1)`. If `shape` is '`valid`', then every pixel evaluation that requires image padding is ignored and the size of the output image is `I2(M1 - M2 + 1, N1 - N2 + 1)`. Finally, if `shape` is '`same`', the size of the output matrix is the same size as `I1`; that is, `I2(M1, N1)` just as in 1-D convolution using that option. These options allow a great deal of flexibility and can simplify the use of 2-D convolution; for example, the '`same`' option can eliminate the need for dealing with the additional points generated by convolution.

2-D correlation is implemented with the routine `imfilter`, which provides even greater flexibility and convenience in dealing with size and boundary effects. The calling structure of this routine is given below.

```
I2 = imfilter(I1, b, options);
```

* In one dimension, this is equivalent to applying the weights in reverse order. In two dimensions, this is equivalent to rotating the filter matrix by 180° before multiplying corresponding pixels and coefficients.

where again $I1$ and h are the input matrices and options can include up to three separate control options. One option controls the size of the output array using the same terms as in `conv2` above: '`'same'`' and '`'full'`' ('`'valid'`' is not valid in this routine!). With `imfilter`, the default output size is '`'same'`' (not '`'full'`), which is the more likely option in image analysis. A second possible option controls how the edges are treated. If a constant is given, then the edges are padded with the value of that constant. The default is to use a constant of zero, that is, standard zero padding. The boundary option '`'symmetric'`' uses a mirror reflection of the end points as shown in Figure 1.21. Similarly, the option '`'circular'`' uses the periodic extension also shown in Figure 1.21. The last boundary control option is '`'replicate'`', which pads using the nearest edge pixel. When the image is large, the influence of the various border control options is subtle, as shown in Example 13.4. However, edge effects could influence subsequent image analysis routines by producing edge pixels with much higher or lower values than those found in the original image. The `imfilter` routine will accept all of the data formats and image classes defined in the previous chapter and produces an output in the same format; however, filtering may not be appropriate for indexed images. In the case of RGB images, `imfilter` operates on all three image planes.

13.2.2 Filter Design

The MATLAB Image Processing Toolbox provides considerable support for generating the filter coefficients.* A number of useful image-processing filters can be easily generated using MATLAB's special routine:

```
b = fspecial(type, parameters);
```

where `type` specifies a specific filter and the optional parameters are related to the filter selected. Filter type options include '`'gaussian'`', '`'disk'`', '`'sobel'`', '`'prewitt'`', '`'laplacian'`', '`'log'`', '`'average'`', and '`'unsharp'`'.

The '`'gaussian'`' option produces a Gaussian lowpass filter. The equation for a Gaussian filter is similar to the equation for the Gaussian distribution:

$$b[m,n] = e^{-(d^2)/2} \quad \text{where } d = \sqrt{(m^2 + n^2)} \quad (13.7)$$

This filter has particularly desirable properties when applied to an image: it provides an optimal compromise between smoothness and filter sharpness. For this reason, it is popular as a general-purpose image lowpass filter. The MATLAB routine for this filter accepts two parameters: the first specifies the dimension of the square filter coefficient array (the default is 3) and the second the value of sigma. Changing the dimension of the coefficient array alters the number of pixels over which the filter operates. As seen from Equation 13.7, the value of sigma adjusts the attenuation slope. The default is 0.5, which produces a modest slope while larger values provide steeper slopes. As a general rule, the coefficient array size should be three to five times the value of sigma.

Both the '`'sobel'`' and '`'prewitt'`' options produce a 3 by 3 filter that enhances horizontal edges (or vertical if transposed). The '`'unsharp'`' filter produces a contrast enhancement filter. The unsharp filter gets its name from an abbreviation of the term "unsharp masking", a double negative that indicates that the unsharp or low spatial frequencies are suppressed (i.e., masked). In fact, it is a special form of highpass filter. This filter has a parameter that specifies the shape of the highpass characteristic. The '`'average'`' filter simply produces a constant set of weights each of which equals $1/N$, where N is the number of elements in the filter (the default size of this

* Since MATLAB's preferred implementation of image filters is through correlation, not convolution, MATLAB's filter design routines generate correlation kernels. The term "filter coefficients" is used here for either kernel format.

Biosignal and Medical Image Processing

filter is 3 by 3 in which case the weights are all 1/9). The filter coefficients for a 3 by 3 Gaussian lowpass filter ($\sigma = 0.5$) and the unsharp filter ($\alpha = .2$) are shown below:

$$b[m,n]_{\text{Unsharp}} = \begin{bmatrix} -0.1667 & -0.6667 & -0.1667 \\ -0.6667 & 4.3333 & -0.6667 \\ -0.1667 & -0.6667 & -0.1667 \end{bmatrix} \quad b[m,n]_{\text{Gauss}} = \begin{bmatrix} 0.0133 & 0.0838 & 0.0133 \\ 0.0838 & 0.6193 & 0.0838 \\ 0.0133 & 0.0838 & 0.0133 \end{bmatrix} \quad (13.8)$$

Note that in the unsharp filter all the coefficients are negative except the center coefficient while in the Gaussian filter all the coefficients are positive.

The 'laplacian' filter is used to take the second derivative of an image: $\partial^2/\partial x^2$. The 'log' filter is actually a *laplacian of gaussian* filter and is used to take the first derivative, $\partial/\partial x$, of an image.

MATLAB also provides a routine to transform 1-D FIR filters, such as those described in Chapter 4, into 2-D filters. This approach is termed the *frequency transform* method and preserves most of the characteristics of the 1-D filter including the transition bandwidth and ripple features. The frequency transformation method is implemented using

```
b2 = ftrans2(b); % Convert to 2-D filter
```

where b2 contains the output filter coefficients (given in *correlation kernel* format), and b contains the 1-D filter coefficients. The latter could be produced by any of the FIR routines described in Chapter 4 (i.e., fir1 or fir2). The function ftrans2 can take an optional second argument that specifies the transformation matrix, the matrix that converts the 1-D coefficients to two dimensions. The default transformation is the *McClellan transformation* that produces a nearly circular pattern of filter coefficients. This approach brings a great deal of power and flexibility to image filter design since it couples all of the FIR filter design approaches described in Chapter 4 to image filtering.

The 2-D Fourier transform described above can be used to evaluate the frequency characteristics of a given filter. In addition, MATLAB supplies a 2-D version of freqz, termed freqz2, that is slightly more convenient to use since it also handles the plotting. The basic call is

```
[H fx fy] = freqz2(b, Ny, Nx); % Filter spectrum
```

where b contains the 2-D filter coefficients and Nx and Ny specify the size of the desired frequency plot, that is, the padded size. The output argument, H, contains the 2-D frequency spectra and fx and fy are plotting vectors; however, if freqz2 is called with no output arguments, then it generates the frequency plot directly. The examples presented below do not take advantage of this function, but simply use the 2-D Fourier transform for filter evaluation.

EXAMPLE 13.2

This is an example of linear filtering using two of the filters in fspecial. Load one frame of the MRI image set (mri.tif) and apply the sharpening filter, b_{unsharp} , described above. Apply a horizontal Sobel filter, b_{Sobel} (also shown above), to detect horizontal edges. Then apply the Sobel filter to detect the vertical edges and combine the two edge detectors. Plot both the horizontal and combined edge detectors.

Solution

To generate the vertical Sobel edge detector, simply transpose the horizontal Sobel filter. While the two Sobel images could be added together using imadd, the program below first converts both images to binary then combines them using a logical OR. This produces a more dramatic black-and-white image of the boundaries. The conversion to black and white images will be done using im2bw with the default threshold of 0.5. More sophisticated methods for adjusting threshold level are discussed in Chapter 14.

```
% Example 13.2 Example of linear filtering using
% selected filters from the MATLAB's fspecial.
%
....load a check image as in Chapter 10.....
%
b_unsharp = fspecial('unsharp', .5); % Unsharp
I_unsharp = imfilter(I,b_unsharp); % filter
%
b_s = fspecial('sobel'); % Sobel filter.
I_sobel_h = imfilter(I,b_s); % horizontal
I_sobel_v = imfilter(I,b_s'); % and vertical
%
% Combine by converting to binary and or-ing together
I_sobel_comb = im2bw(I_sobel_h) | im2bw(I_sobel_v);
%
.....plot images .....
% Plot filter frequency characteristics
F= fftshift(abs(fft2(b_unsharp,32,32)));
mesh (-16:15,-16:15,F);
.....colormap, titles and labels.....
%
F = fftshift(abs(fft2(b_s,32,32)));
mesh (-16:15,-16:15,F);
.....colormap, titles and labels.....
```

Results

The images produced by this example program are shown below along with the frequency characteristics associated with the 'unsharp' and 'sobel' filter in Figures 13.5 and 13.6. Note that the 'unsharp' filter has the general frequency characteristics of a highpass filter, that is, a positive slope with increasing spatial frequencies (Figure 13.6, left plot). The double peaks of the Sobel filter that produce edge enhancement are evident in Figure 13.6, right plot. Since this is a magnitude plot, both peaks appear as positive (Figure 13.6, right plot).

In the next example, routine `ftrans2` is used to construct 2-D filters from 1-D FIR filters. Lowpass and highpass filters are constructed using the filter design routine `fir1` from Chapter 4. This routine generates filter coefficients based on the ideal rectangular window approach described in that chapter. Example 13.3 also illustrates the use of an alternate padding technique to reduce the edge effects caused by zero padding. Specifically, the 'replicate' option of `imfilter` is used to pad by repeating the last (i.e., image boundary) pixel value. This eliminates the dark border produced by zero padding, but the effect is subtle.

EXAMPLE 13.3

Example of the application of standard 1-D FIR filters extended to two dimensions. Load the blood cell image (`blood1.tif`) and filter using lowpass and highpass filters developed from 1-D filters. The filters should be based on rectangular window filters (Equation 4.19) and should have cutoff frequencies of $0.125f_s/2$. Apply the highpass filter using `imfilter` with and without the 'replicate' option. Show the original and highpass filtered images. Also threshold the highpass filtered images with `im2bw` using a threshold that best identifies the cell boundaries. Finally, compare the lowpass filter with a 10×10 lowpass Gaussian filter with a sigma of 2.0 and plot the spectra of these two filters.

Solution

Generate the 1-D lowpass and highpass filters using `fir1`. After generating these filters, extend them into two dimensions, 32×32 , using `ftrans2` and apply to the original image. After

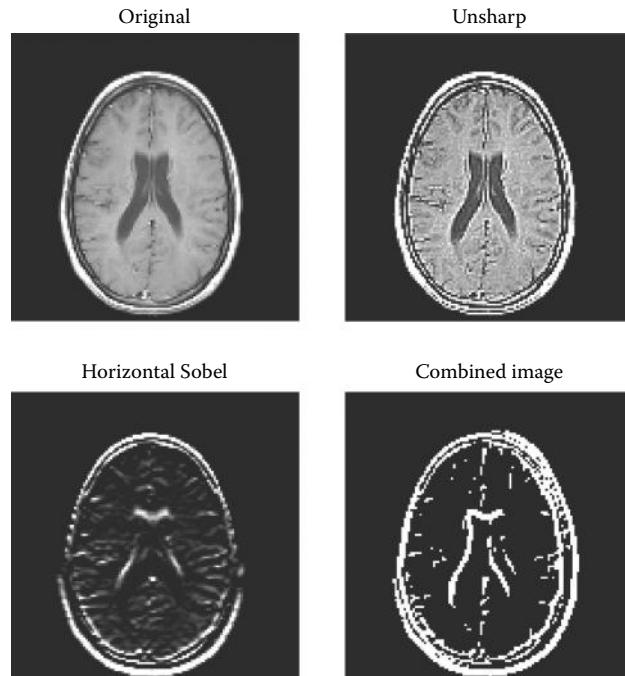


Figure 13.5 Filtering applied to an MRI image of the brain. The unsharp filter shows enhancement of high-frequency spatial details. The horizontal Sobel after thresholding emphasizes horizontal boundaries where the top is brighter than the bottom. The combined image also emphasizes vertical boundaries where the left side is brighter than the right side. (Image courtesy of MathWorks.)

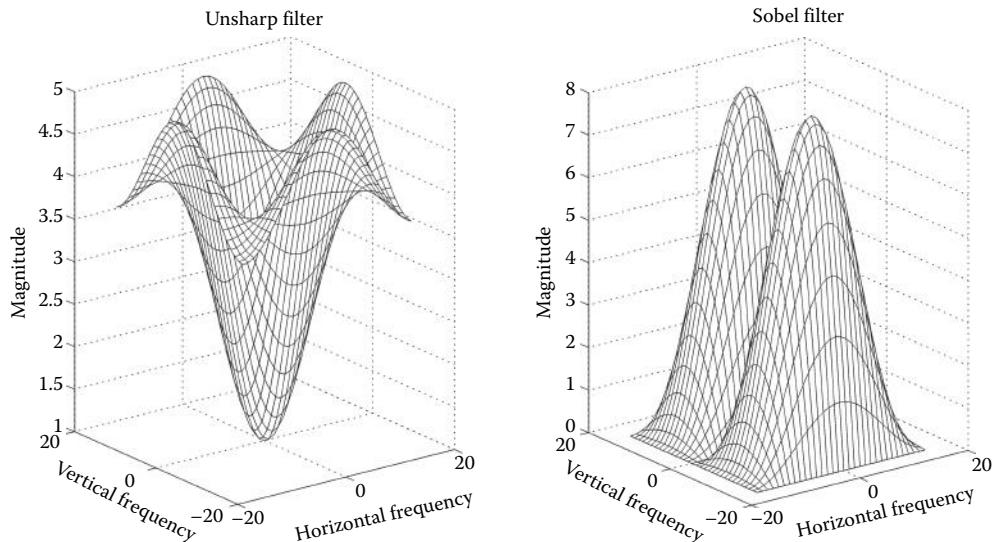


Figure 13.6 Magnitude spectra of the unsharp and Sobel filters. The unsharp filter shows high-pass characteristics (recall the low frequencies are in the center of this plot). The Sobel magnitude spectrum shows both peaks as positive, but one of them is actually negative if the phase is taken into consideration.

applying the highpass filter use `mat2gray` to readjust the intensity range of the output. Adjust the threshold of `im2bw` interactively until the cell boundaries are highlighted with the least amount of background noise. Generate the Gaussian lowpass filter with `fspecial`, then use `fft2` in conjunction with `fftshift` to determine the lowpass filters' spectra and plot using `mesh`.

```
% Example 13.3 Linear filtering example
% Load the image and filter
%
N = 32; % Filter order
w_lp = .125; % Lowpass cutoff frequency
w_hp = .125; % Highpass cutoff frequency
.....load image blood1.tif; convert...
%
b = fir1(N,w_lp); % Generate the lowpass filter
h_lp = ftrans2(b); % Convert to 2-dimensions
I_lowpass = imfilter(I,h_lp); % and apply
%
b = fir1(N,w_hp,'high'); % Repeat for highpass
h_hp = ftrans2(b);
I_highpass = imfilter(I, h_hp); % Apply highpass filter
I_highpass = mat2gray(I_highpass); % w/wo replicate
I_highpass_rep = imfilter(I,h_hp,'replicate');
I_highpass_rep = mat2gray(I_highpass_rep);
BW_high = im2bw(I_highpass, 0.54); % Threshold image
BW_high_rep = im2bw(I_highpass_rep, 0.54); % Threshold image
%
b_g = fspecial('gaussian',8,2); % Gaussian lowpass filter
I_lowgauss = imfilter(I,b_g); % Apply Gaussian lowpass filter
.....plot images and filter spectra.....
% Now plot the highpass and lowpass frequency characteristics
N = 128;
F= fftshift(abs(fft2(b2_lp,N,N))); % Calculate magnitude spectrum
f = N/2:(N/2)-1; % Frequency vector for plotting
mesh(f,f,F); % and plot
.....subplot, labels, and repeat for highpass filter.....
```

Results

Figure 13.7 shows the original figure, the highpass image, and the thresholded images. The highpass filtered image shows a derivative-like characteristic that enhances edges. The thresholds of the black-and-white images were adjusted to produce outlines of the cells, which could be useful in isolating these cells. A threshold value of 0.54 was found to produce well-defined cell borders without including too many other features. Isolating features of interest within an image is known as image *segmentation* and is covered in the next chapter. The only difference in the two thresholded images is at the edges, but it is not possible to detect a difference from the reproduction.

The two lowpass images are shown in Figure 13.8 along with the magnitude spectrum of the filters. The lowpass filter derived from the 1-D filter has a slightly sharper cutoff and steeper slope than the Gaussian filter and this is reflected in the amount of blurriness seen in the two images. The sharpness of the Gaussian filter slope could be increased by increasing the size of the filter coefficient matrix. The magnitude spectrum of the highpass filter is not shown, but is simply the inverse of the lowpass filter in Figure 13.8, lower left.

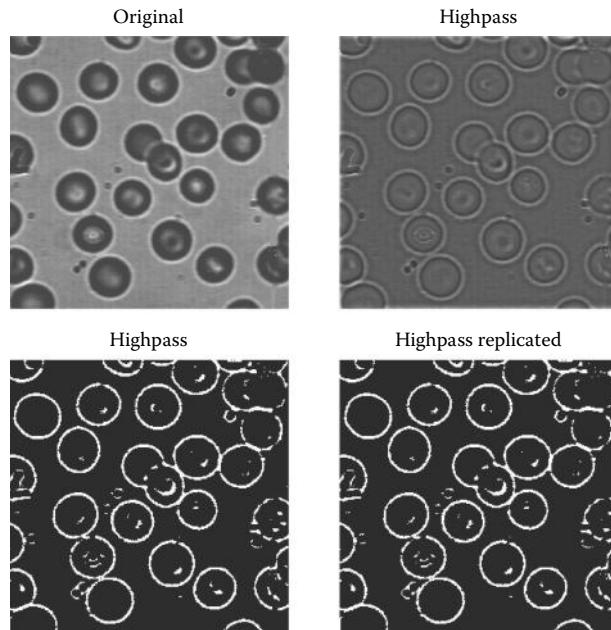


Figure 13.7 Blood cells images (`blood.tif`) before and after highpass filtering and after thresholding the filter image. In the right-hand threshold image, the 'replicate' option is used to pad the edges; this produces a smoother edge (not visible here). (Image courtesy of MATLAB.)

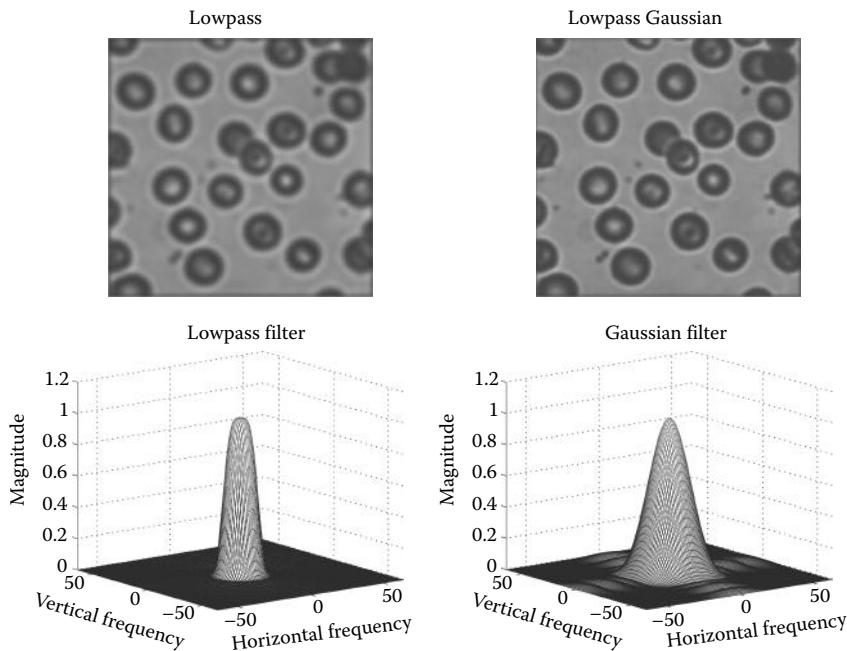


Figure 13.8 Images produced by applying two lowpass filters to the blood cell images. The left-hand image and spectrum were produced by a lowpass filter developed from a 1-D rectangular window filter. The right-hand image and spectrum were produced by a Gaussian filter generated by the `fpcpecial` routine. The left-hand filter has a sharper cutoff and produces an image with more blur.

13.3 Spatial Transformations

Several useful transformations take place entirely in the spatial domain. Such transformations include image resizing, rotation, cropping, stretching, shearing, and image projections. Spatial transformations perform a remapping of pixels and often require some form of interpolation in addition to possible antialiasing. The primary approach to antialiasing is lowpass filtering. For interpolation, there are three methods popularly used in image processing, and MATLAB supports all three. All three interpolation strategies use the same basic approach: the interpolated pixel in the output image is the weighted sum of pixels in the vicinity of the original pixel after transformation. The methods differ primarily in how many neighbors are considered.

As mentioned above, spatial transforms involve a remapping of one set of pixels (i.e., image) to another. In this regard, the original image can be considered as the input to the remapping process and the transformed image is the output of this process. If images were continuous, then remapping would not require interpolation, but the discrete nature of pixels usually necessitates interpolation.* The simplest interpolation method is the *nearest neighbor* method in which the output pixel is assigned the value of the closest pixel in the transformed image (Figure 13.9). If the transformed image is larger than the original and involves more pixels, then a remapped input pixel may fall into two or more output pixels. In the *bilinear* interpolation method, the output pixel is the weighted average of transformed pixels in the nearest 2 by 2 neighborhood, and in *bicubic* interpolation, the weighted average is taken over a 4 by 4 neighborhood.

Both computational complexity and accuracy increase with the number of pixels that are considered in the interpolation, so there is a trade-off between quality and computational time. In MATLAB, the functions that require interpolation have an optional argument that specifies the method. For most functions, the default method is the nearest neighbor. This method produces acceptable results on all image classes and is the only method appropriate for indexed images. The method is also the most appropriate for binary images. For RGB and intensity image classes, the bilinear or bicubic interpolation method is recommended since it leads to better results.

MATLAB provides several routines that can be used to generate a variety of complex spatial transformations such as image projections or specialized distortions. These transformations can be particularly useful when you are trying to overlay or *register* images of the same structure taken at different times or with different modalities (e.g., PET scans and MRI images). While MATLAB's spatial transformations routines allow for any imaginable transformation, only two types of transformations will be discussed here: *affine* transformations and *projective* transformations. Affine transformations are defined as transformations in which straight lines remain straight and parallel lines remain parallel, but rectangles may become

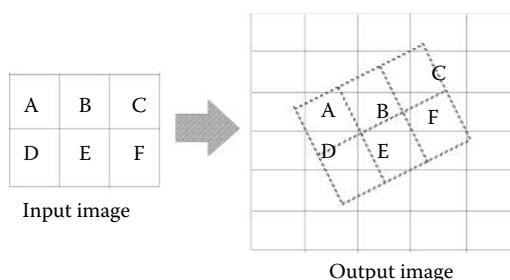


Figure 13.9 A rotation transform using the nearest neighbor interpolation method. Pixel values in the output image (solid background grid) are assigned values from the nearest pixel in the transformed input image (dashed grid).

* A few transformations may not require interpolation such as rotation by 90° or 180°.

Biosignal and Medical Image Processing

parallelograms. These transformations include rotation, scaling, stretching, and shearing. In projective translations, straight lines still remain straight, but parallel lines often converge toward *vanishing points*.

13.3.1 Affine Transformations

MATLAB provides a procedure described below for implementing any affine transformation; however, a subset of these transformations are so popular they are supported by separate routines. These include image resizing, cropping, and rotation. Both image resizing and cropping are techniques to change the dimensions of an image: the latter is interactive using the mouse and display while the former is under program control. To change the size of an image, MATLAB provides the `imresize` command:

```
I_resize = imresize(I, arg or [M N], 'method');
```

where I is the original image and I_resize is the resized image. If the second argument is a scalar arg , then it gives a magnification factor, and if it is a two-element vector, $[M\ N]$, it indicates the desired new dimensions in vertical and horizontal pixels, M, N . If $arg > 1$, then the image is increased (magnified) in size proportionally and if $arg < 1$, it is reduced in size (minified). This will change image size proportionally. If the vector $[M\ N]$ is used to specify the output size, image proportions can be modified: the image can be stretched or compressed along a given dimension. The argument '`method`' specifies the type of interpolation to be used and can be either '`nearest`', '`bilinear`', or '`bicubic`', referring to the three interpolation methods described above. The nearest neighbor method is the default. If image size is reduced, then `imresize` automatically applies an antialiasing, lowpass filter unless the interpolation method is '`nearest`'; that is, the default. The logic of this is that the nearest neighbor interpolation method would usually only be used with indexed images, and lowpass filtering is not really appropriate for these images.

Image cropping is an interactive command:

```
I_resize = imcrop;
```

The `imcrop` routine waits for the operator to draw an onscreen cropping rectangle using the mouse. The current image is resized to include only the image within the rectangle.

Image rotation is straightforward using the `imrotate` command:

```
I_rotate = imrotate(I, deg, 'method', 'bbox');
```

where I is the input image, I_rotate is the rotated image, deg is the degrees of rotation (counterclockwise if positive, and clockwise if negative), and '`method`' describes the interpolation method as in `imresize`. Again, the nearest neighbor method is the default even though the other methods are preferred except for indexed images. After rotation, the image will not, in general, fit into the same rectangular boundary as the original image. In this situation, the rotated image can be cropped to fit within the original boundaries or the image size can be increased to fit the rotated image. Specifying the '`bbox`' argument as '`crop`' will produce a cropped image having the dimensions of the original image, while setting '`bbox`' to '`loose`' will produce a larger image that contains the entire original, unrotated, image. The '`loose`' option is the default. In either case, additional pixels will be required to fit the rotated image into a rectangular space (except for some orthogonal rotations) and `imrotate` pads these with zeros, producing a black background to the rotated image (see Figure 13.10).

Application of the `imresize` and `imrotate` routines is shown in Example 13.4. The application of `imcrop` is presented in one of the problems at the end of this chapter.

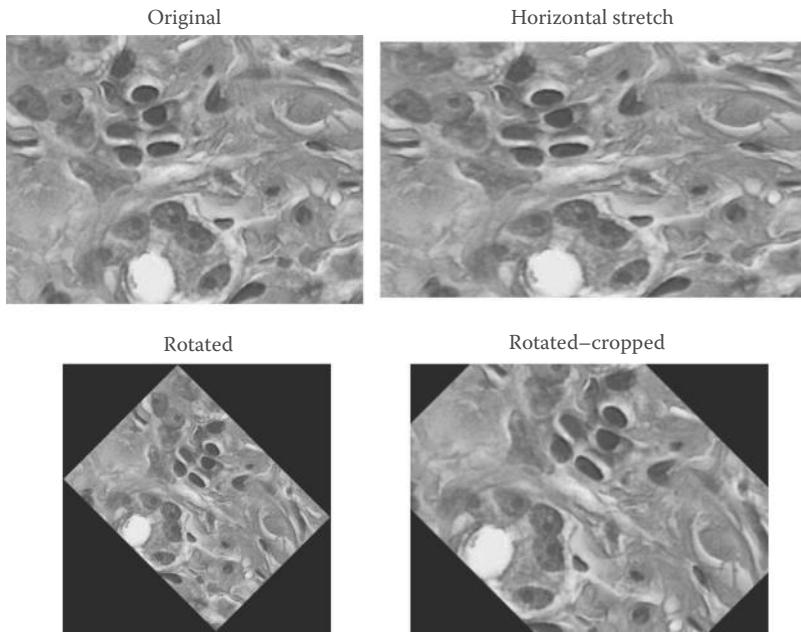


Figure 13.10 Two common affine transformations supported by special MATLAB routines. The original image is stretched 25% (upper right) and rotated 45° (lower images). The rotation is done to fit the original image dimensions and, in the cropped image, to keep the original image size the same. (Stained image courtesy of Alan W. Partin, MD, PhD, Johns Hopkins University School of Medicine.)

EXAMPLE 13.4

Demonstrate resizing and rotation spatial transformations. Load the image of stained tissue (`hestain.png`) and transform it so that the horizontal dimension is 25% longer than in the original while keeping the vertical dimension unchanged. Rotate the original image 45° clockwise, with and without cropping. Display the original and transformed images in a single figure.

Solution

Use `imresize` to stretch the image. To stretch the horizontal dimension by 25%, first get the image dimension using `size` and multiply the column number by 1.25 in the call to `imresize`. For rotation, simply call `imrotate` with and without the crop option. Use the '`bilinear`' option for both affine operations.

```
% Example 13.4 Example of various affine transformations
%
.....read image and convert if necessary .....
%
% Stretch by 25% horizontally.
[M N] = size(I);
I_stretch = imresize(I,[M N*1.25], 'bilinear');
%
% Rotate image with and without cropping
I_rotate = imrotate(I,-45,'bilinear');
I_rotate_crop = imrotate(I, -45, 'bilinear', 'crop');
..... display the images .....
```

Results

The images produced by this code are shown in Figure 13.10.

13.3.1.1 General Affine Transformations

In the MATLAB Image Processing Toolbox, both affine and projective spatial transformations are defined by a `Tform` structure, which can be constructed using one of two routines: the routine `maketform` uses parameters supplied by the user to construct the transformation while `cp2tform` uses image reference points, or *landmarks*, interactively placed on different images to generate the transformation. Both routines are very flexible and powerful, but that also means they are quite involved. This section describes aspects of the `maketform` routine while the `cp2tform` routine will be presented in context with image registration.

Irrespective of the way in which the desired transformation is specified, it is implemented using the routine `imtransform`. This routine is only slightly less complicated than the transformation specification routines, and only some of its features are discussed here. (The associated help file should be consulted for more detail.) The basic calling structure used to implement the spatial transformation is

```
B = imtransform(A,Tform,'interp','Parm1',value1,'Para2',value2,...);
```

where `A` and `B` are the input and output arrays, respectively, and `Tform` provides the transformation specifications as generated by `maketform` or `cp2tform`. The additional arguments are optional, including `interp`, which can be '`nearest`', '`bilinear`' (the default), or '`bicubic`'. The other optional parameters are specified as pairs of arguments: a string containing the name of the optional parameter (i.e., '`Parm1`') followed by the value.* These parameters can specify the pixels to be used from the input image (the default is the entire image); permit a change in pixel size; specify how to fill any extra background pixels generated by the transformation; and specify the size and range of the output array. Only the parameters that specify output range will be discussed here, as they can be used to override the automatic rescaling of image size performed by `imtransform`. To specify output image range and size, parameters '`XData`' and '`YData`' are followed by a two-variable vector that gives the *x* or *y* coordinates of the first and last elements of the output array, `B`. If these parameters are not specified, `imtransform` will shift the image to keep the entire transform image in the image frame. To keep the size and range in the output image the same as the input image, simply specify the horizontal and vertical size of the input array, that is

```
[M N] = size(A);  
...  
B = imtransform(A,Tform,'interp','Xdata',[1 N],Ydata',[1 M]);
```

As with the transform specification routines, `imtransform` uses the *spatial coordinate system* described in Section 12.1.1. In this system, the first dimension is the *x* coordinate while the second is the *y*, the reverse of the matrix subscripting convention used by MATLAB. (However, the *y* coordinate still *increases in the downward direction*.) In addition, noninteger values for *x* and *y* indexes are allowed.

The routine `maketform` can be used to generate the spatial transformation descriptor, `Tform`. There are two alternative approaches to specifying the transformation, but the most

* This is a common approach used in many MATLAB routines when a large number of arguments are possible, especially when many of these arguments are optional. It allows the arguments to be specified in any order and easily omitted.

straightforward uses simple geometrical objects to define the transformation. The calling structure under this approach is

```
Tform = maketform('type', U, X);
```

where 'type' defines the type of transformation and U and X are vectors that define the specific transformation by defining the input (U) and output (X) geometries. While `maketform` supports a variety of transformation types, including custom, user-defined types, only the affine and projective transformations are discussed here. These are specified by 'type' parameters '`'affine'`' and '`'projective'`'.

Only *three points* are required to define an affine transformation, so, for this transformation type, U and X define corresponding vertexes of input and output triangles. Specifically, U and X are 3 by 2 matrices where each row defines a corresponding vertex that maps input to output geometry. For example, to stretch an image vertically, define an output triangle that is taller than the input triangle. Assuming an input image of size M by N (i.e., $[M, N] = \text{size}(x)$), to increase the vertical dimension by 50% define input (U) and output (X) triangles as

```
U = [1,1; 1,M; N,M]; X = [1,1-0.5*M; 1,M; N,M];
```

In this example, the input triangle, U, is simply the upper left, lower left, and lower right corners of the image. The output triangle, X, has its top, left vertex increased by 50%. (Recall the coordinate pairs are given as x, y the reverse of M, N, but that positive y still increases downward. Also note that negative coordinates are acceptable).

As another example, to increase the vertical dimension symmetrically, change X to

```
X = [1,1-0.25*M; 1,1.25*M; N,1.25*M];
```

In this case, the upper vertex is increased by only 25% and the two lower vertexes are lowered in the y direction by increasing the y coordinate value by 25%. This transformation could be done with `imresize`, but this would also change the dimension of the output image. When this transformation is implemented with `imtransform`, it is possible to control output size as described above.* Hence this approach, although more complicated, allows greater control of the transformation. Of course, if output image size is kept the same, the contents of the original image, when stretched, may exceed the boundaries of the image and will be lost.

The `maketform` routine can be used to implement other affine transformations such as *shearing*. For example, to shear an image to the left, define an output triangle that is skewed by the desired amount with respect to the input triangle. In Figure 13.11, the input triangle is specified as $U = [N/2, 1; 1, M; N, M]$ (solid lines) and the output triangle as $X = [1, 1; 1, M; N, M]$ (solid line). This shearing transform is implemented in Example 13.5.

13.3.2 Projective Transformations

In projective transformations, straight lines remain straight but parallel lines may converge. Projective transformations can be used to give objects perspective. Projective transformations require *four points* for definition; hence, the defining geometrical objects are quadrilaterals. Figure 13.12 shows a projective transformation in which the original image appears to be tilted back. In this transformation, vertical lines in the original image converge in the transformed image. In addition to adding perspective, these transformations are of value in correcting for

* If `imtransform` is called without specifying the `Ydata` and `Ydata` parameters, the use of `imtransform` and `imresize` lead to identical images since `imtransform` will automatically shift the output image coordinates to keep the image within the frame.

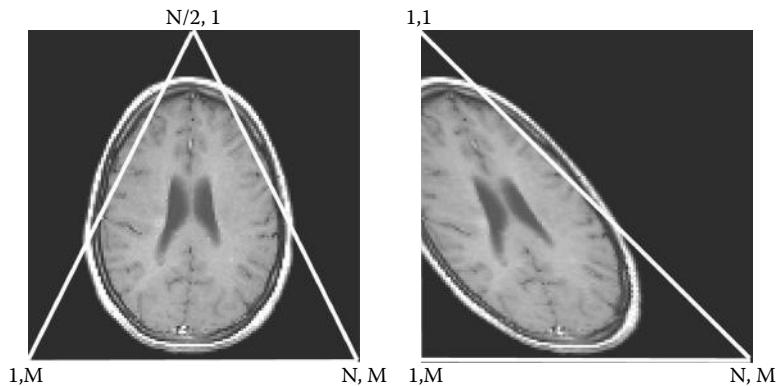


Figure 13.11 An affine transformation can be defined by three points. The transformation shown here is defined by an input (left) and output (right) triangle and produces a sheared image. M and N are rows and columns, respectively, but are specified in the transformation algorithm in reverse order following the spatial coordinate indexing system.

relative tilts between image planes during image registration. *Image registration*, the alignment of similar images, is the primary biomedical application of these transformations and they are revisited in the next section. Example 13.5 illustrates the use of these general image transformations for affine and projective transformations.

EXAMPLE 13.5

General spatial transformations. Apply the affine and projective spatial transformation to one frame of the MRI image in `mri.tif`. The affine transformation should skew the top of the image to the left, just as shown in Figure 13.11. The projective transformation should tilt the image back as shown in Figure 13.12. A second projective transformation should tilt the image forward.

Solution

After the image is loaded, the affine input triangle is defined as an equilateral triangle inscribed within the full image. The output triangle is defined by shifting the top point to the left side, so the output triangle is now a right triangle as in Figure 13.11. In both projective transformations, the input quadrilateral is a rectangle the same size as the input image. In the first projective transformation, the output quadrilateral is generated by moving the upper points inward and

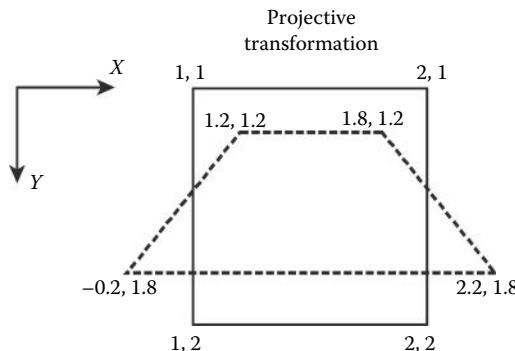


Figure 13.12 Specification of a projective transformation by defining two quadrilaterals. The solid lines define the input quadrilateral and the dashed lines define the desired output quadrilateral.

down by an equal amount while the lower points are moved outward and up, also by an equal amount following the geometry shown in Figure 13.12. The second projective transformation is achieved by reversing the operations performed on the top and bottom corners.

```
% Example 13.5 General Spatial Transformations
%
% .....load frame 18 .....
% Define affine transformation
U1 = [N/2 1; 1 M; N M]; % Input triangle
X1 = [1 1; 1 M; N M]; % Output triangle
Tform1 = maketform('affine', U1, X1); % Generate transform
I_affine = imtransform(I,Tform1, 'Xdata',[1 N],...
    'Ydata',[1 M]); % Apply transform
%
% Define projective transformation vectors
offset = .25*N;
U = [1 1; 1 M; N M; N 1]; % Input quadrilateral
X = [1-offset 1+ offset; 1+ offset M-offset;...
    N-offset M-offset; N+ offset 1+ offset]; % Output
%
Tform2 = maketform('projective', U, X); % Generate transform
I_proj1 = imtransform(I,Tform2,'Xdata',[1 N],...
    'Ydata',[1 M]); % Apply transform
%
% Second projective transformation.
X = [1+ offset 1+ offset; 1- offset M-offset;...
    N+ offset M-offset; N- offset 1+ offset]; % Output
Tform3 = maketform('projective', U, X); % Generate transform
I_proj2 = imtransform(I,Tform3,'Xdata',[1 N],...
    'Ydata',[1 M]); % Apply transform
..... display images .....
```

Results

The original image, affine transformation, and the two projective transformations are shown in Figure 13.13. In the image on the lower left, the brain appears to tilt forward, while in the lower right-hand image, the brain appears to tilt backward.

Of course, a great many other transformations can be constructed by redefining the output (or input) triangles or quadrilaterals. Some of these alternative transformations are explored in the problems.

All these transformations can be applied to produce a series of images having slightly different projections. When these multiple images are shown as a movie, they will give an object the appearance of moving through space, perhaps in three dimensions. Several of the problems at the end of this chapter explore these features. The following example demonstrates the construction of such a movie.

EXAMPLE 13.6

Construct a series of projective transformations that when shown as a movie give the appearance of the image tilting backward in space. Use one of the frames of the MRI image.

Solution

The code below uses the projective transformation to generate a series of images that appear to tilt back because of the geometry used. The approach uses the second projective transformation in Example 13.5, but adjusts the transformation to produce a slightly larger apparent tilt in each

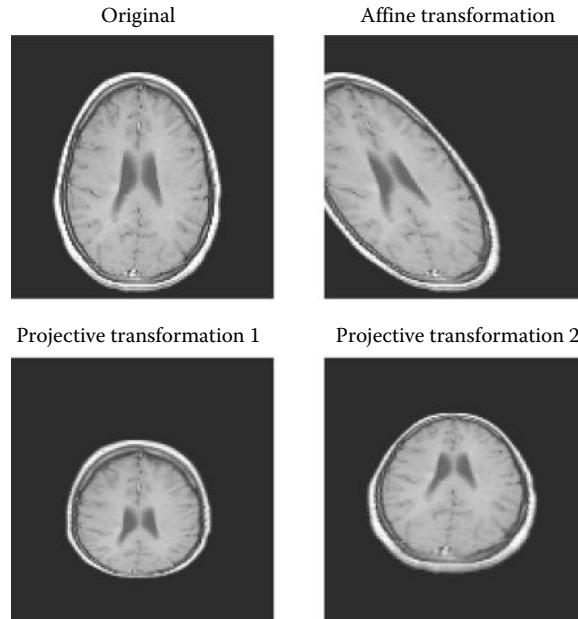


Figure 13.13 One affine and two projective spatial transformations produced in Example 13.5. Upper left: Original image. Upper right: An affine transformation with the image skewed to the left. Lower left: A projective transform in which the image is made to appear tilted forward. Lower right: A projective transformation in which the image is made to appear tilted backward.

frame. The program fills 24 frames in such a way that the first 12 have increasing angles of tilt and the last 12 have decreasing tilt. When shown as a movie, the image will appear to rock backward and return. This same approach is also used in Problem 13.11. Note that as the images are being generated by `imtransform`, they are converted to indexed images using `gray2ind` since this is the format required by `immovie`. The grayscale map generated by `gray2ind` is used (at the default level of 64), but any other map could be substituted in `immovie` to produce a pseudocolor image.

```
% Example 13.6 Spatial Transformation movie
%
Nu_frame = 12;           % Number of frames each direction
Max_tilt = .5;            % Maximum tilt
.....load MRI frame 12 as in previous examples.....
%
U = [1 1; 1 M; N 1];    % Input quadrilateral
for i = 1:Nu_frame        % Nu_frame*2 frames
    % Define transformation that varies to Max_tilt
    offset = Max_tilt*N*i/Nu_frame;
    X = [1 + offset 1 + offset; 1 - offset M - offset; ...
          N + offset M - offset; N - offset, 1 + offset];
    Tform2 = maketform('projective', U, X);
    [I_proj(:,:,:,i), map] = gray2ind(imtransform...
        (I,Tform2,'Xdata',[1 N], 'Ydata',[1 M]));
    % Fill remaining frames in reverse order to make
    % make image rock back and forth
    I_proj(:,:,:1,2*Nu_frame + 1 - i) = I_proj(:,:,:1,i)
end
```

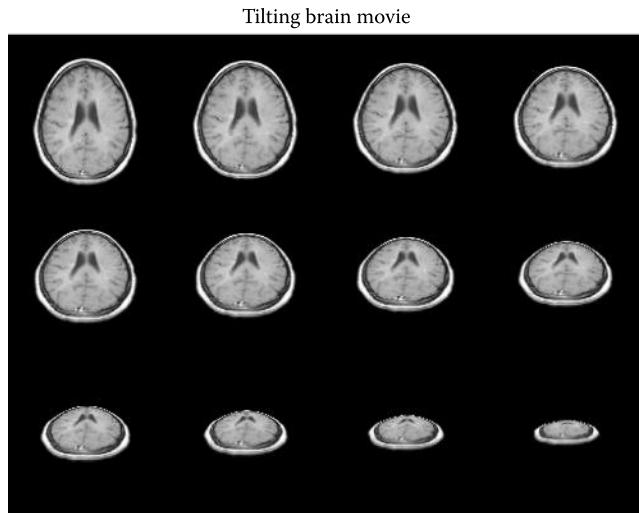


Figure 13.14 Montage display of the movie produced by the code in Example 13.6. The various projections give the appearance of the brain slice tilting and moving back in space. Only half the 24 frames are shown here as the rest are the same, just presented in reverse order to give the appearance of the brain rocking back and forth.

```
%  
montage(I_proj(:,:,1:12),map); % Display as a montage  
mov = immmovie(I_proj,map); % Display as movie  
movie(mov,5);
```

Results

While it is not possible to show the movie that is produced by this code, the various frames are shown as a montage in Figure 13.14. Several problems explore the use of spatial transformations to make some amusing movies.

13.4 Image Registration

Image registration is the alignment of two or more images so they are best superimposed. This task has become increasingly important in medical imaging as it is used for merging images acquired from different modalities, such as from MRI and PET. Registration is also useful for comparing images taken of the same structure at different points in time. In functional magnetic resonance imaging (fMRI), image alignment is needed for images taken sequentially in time as well as between images that may have different resolutions. To achieve the best alignment, it may be necessary to transform the images using any or all of the transformations described previously. Image registration can be quite challenging even when the images are identical or very similar (as will be the case in the examples and problems given here). Frequently, the images to be aligned are not that similar, perhaps because they have been acquired using different modalities. The difficulty in accurately aligning images that are only moderately similar presents a significant challenge to image registration algorithms, so the task is often aided by a human intervention or the use of embedded markers for reference.

Approaches to image registration can be divided into two broad categories: unassisted image registration where the algorithm generates the alignment without human intervention; and

interactive registration where a human operator guides or aids the registration process. The former approach usually relies on some optimization technique to maximize the correlation between the images. In the latter, a human operator may aid the alignment process by selecting corresponding reference points in the images to be aligned: corresponding features are identified by the operator and tagged using some interactive graphics procedure. Both approaches are supported in MATLAB and are demonstrated in the examples and problems.

13.4.1 Unaided Image Registration

Unaided image registration usually involves the application of an optimization algorithm. The value that is optimized, also referred to as the *cost function*, is the correlation, or other measure of similarity, between the images. In this strategy, the appropriate transformation is applied to one of the images, termed the *input image*, and the comparison is made between this transformed image and the *reference image* (also termed the *base image*). In some applications, there may be a number of input images. The optimization routine seeks to vary the transformation in some manner until the comparison is the best possible. The problem with this approach is the same as with all optimization techniques: the optimization process may converge on a suboptimal solution, a *local maximum*, not the optimal solution, the *global maximum*. Often, the solution depends on the starting values of the transformation variables. An example of convergence to a suboptimal solution and dependency on initial variables is found in Problem 13.15.

Example 13.7 uses the optimization routine that is part of the basic MATLAB package, `fminsearch`. This routine is based on the simplex (direct search) method, and will adjust any number of “adjustable parameters” to minimize a function specified in a user routine. Since this routine seeks a minimum, the *negative* of the correlation between the two images is used as the cost function. The routine `fminsearch` will automatically adjust the transformation variables to achieve a minimum.*

The routine `fminsearch` is called as

```
x = fminsearch('fun',x0,options,param1,param2,...);
```

where '`fun`' is a MATLAB routine that defines the operation being optimized, `x0`, is a vector of initial values for the adjustable parameters, `options` is a vector that controls the optimization process, and `param1`, `param2`, and so on are potential additional input parameters to the routine '`fun`'. The initial values in `x0` are usually set randomly or set to all ones or zeros. The `options` argument controls such aspects of the optimization as when to stop (the minimum error or change in error) and usually the default values can be used. The output, `x`, is a vector containing the final value of the adjustable parameters.

The MATLAB routine `fun` takes as input the adjustable parameters (in this case, the transformation parameters) and outputs the parameter being minimized (in this case negative correlation). This user-generated routine applies the current value of the transformation parameters to obtain a trial transformation, compares the trial image and the reference images, and outputs the negative correlation.[†] In addition to the transformation parameters, this routine requires both the input and reference images, which are passed as additional parameters. Through this routine, the programmer specifies the structure of the transformation; the optimization routine (`fminsearch`) works blindly and simply seeks the set of adjustable parameters that result in a minimum output. After `fminsearch` finishes its

* Remember, this minimum is not necessarily the optimal, global minimum. As the number of parameters being adjusted increases, the chances of getting stuck at a local minimum increase.

[†] From a mathematical standpoint, this routine defines the function being optimized.

adjustment of the transformation parameters, the final values of these adjustable parameters can be used to produce the aligned image.

The transformation selected should be based on the possible mechanisms for misalignment: translations, size changes, rotations, skewness, projective misalignment, or other more complex distortions. The transformation selected should be one that minimizes the number of transformation parameters.* Reducing the number of variables increases the likelihood of optimal convergence and substantially reduces computation time. An alignment requiring three transformation parameters is shown in the next example.

EXAMPLE 13.7

This is an example of unaided image registration requiring an affine transformation. The input image (the image to be aligned) is a distorted version of the reference image. Specifically, it has been stretched horizontally, compressed vertically, and tilted using an affine transformation. The optimization routine is called upon to find a transformation that realigns this image with the reference image.

Solution

After loading the image, an affine transformation is applied to construct a distorted version of the image that is used as the input image. Then, MATLAB's optimization routine `fminsearch` is used to determine an optimal transformation that will restore the original image. The `fminsearch` routine calls the user-defined routine `realign` to perform the transformation and calculate the correlation between the two images. The `realign` routine assumes that an affine transformation is required and that only the horizontal, vertical, and tilt dimensions need to be adjusted. It does not, for example, take into account possible translations between the two images, although this would not be too difficult to incorporate.

The `fminsearch` routine requires, as input arguments, the name of the routine whose output is to be minimized (in this example, `realign`) and the initial values of the adjustable parameters (in this example, all ones). The routine uses the size of the initial parameter vector to determine how many variables it needs to adjust (in this case, three variables). The additional input arguments follow the `options` parameter and include the input and reference images. The default values are used for the `options` vector (i.e., `[]`). The optimization routine (`fminsearch`) continues to call `realign` automatically until it finds an acceptable minimum for the error, or until some maximum number of iterations is reached (see the help file).

```
% Example 13.7 Unaided image registration
%
H_scale = .25; % Define distorting param
V_scale = .2; % Horizontal, vertical, and
tilt = .2; % tilt in percent/100
..... load mri.tif, frame 18.....
[M N] = size(I);
H_scale = H_scale * N/2; % Convert to pixels
V_scale = V_scale * M;
tilt = tilt * N
%
% First construct distorted image.
U = [1 1; 1 M; N M]; % Input triangle
```

* The number of defining variables depends on the transformation. For example, rotation alone only requires one variable, linear displacements require two variables, affine transformations require three variables, and projective transformations require four variables.

Biosignal and Medical Image Processing

```
X = [1-H_scale + tilt 1+V_scale; 1-H_scale M; N+H_scale M];
Tform = maketform('affine', U, X);
I_trans = imtransform(I,Tform,'bicubic',...
    'Xdata',[1 N],'Ydata',[1 M]))*.8;
%
% Solution starts here. Find transformation to realign image
init = [1 1 1];           % Set initial values
[scale,Fval] = fminsearch('realign',init,[ ],I,I_trans); % Optimize
disp(Fval)                 % Display final correlation
%
% Realign image using optimized transform
X = [1 + scale(1) + scale(3) 1 + scale(2);....
    1 + scale(1) M; N - scale(1) M];
Tform = maketform('affine', U, X);
I_aligned = imtransform(I_trans,Tform,'bicubic',...
    'Xdata',[1 N],'Ydata',[1 M]);
.....display original and transformed images.....
```

The `realign` routine is used by `fminsearch`. This routine takes in the transformation variables supplied by `fminsearch`, performs a trial transformation, and compares the trial image with the reference image. The routine then returns the error to be minimized, calculated as the negative of the correlation between the two images.

```
function err = realign(scale,I,I_trans);
% Function used by 'fminsearch' to rescale an image
% Performs trial transformation and computes
% correlation between images.
% Inputs:
%     scale - Current transform parameters
%     I - reference image
%     I_trans - input image
% Outputs:
%     Negative correlation between images.
%
[M N] = size(I);          % Get image size
U = [1 1; 1 M; N M];      % Input triangle
%
% Perform trial transformation
X = [1 + scale(1) + scale(3), 1 + scale(2);...
    1 + scale(1) ,M; N - scale(1) ,M];
Tform = maketform('affine',U,X);
I_aligned = imtransform(I_trans,Tform,',...
    'Xdata',[1 N],'Ydata',[1 M]);
%
% Calculate negative correlation
err = -abs(corr2(I_aligned,I));
```

Results

The results achieved by this registration routine are shown in Figure 13.15. The reference image is shown on the left and the input image is in the center. This image has been distorted by three affine transformations (horizontal scratching, vertical compression, and a tilt). The aligned image achieved by the optimization is shown on the right. This image is very similar to the reference image. This optimization is fairly robust: it converges to a correlation of 0.99 from both positive and negative initial values. However, in many cases, convergence can depend on the initial values. In addition, the realigned image is not quite as sharp as the original, but then neither

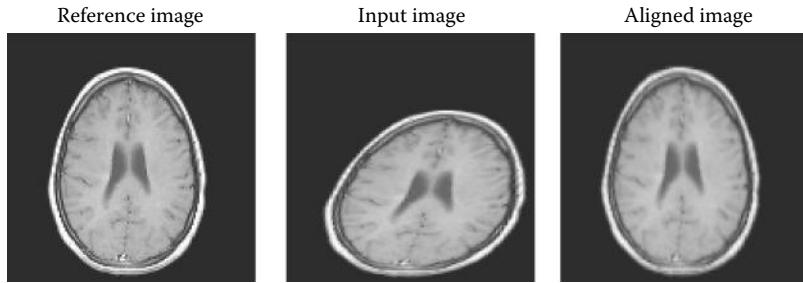


Figure 13.15 Unaided registration using optimization. The left-hand image is the original unaided image and the center image is the distorted image used as the input image. The realigned image on the right is quite similar to the reference image on the left although not as sharp.

is the distorted image. Image sharpness can be improved by using the 'bicubic' option in the two transformations.

13.4.2 Interactive Image Registration

Several strategies may be used to guide the registration process. In the example used here, registration depends on reference marks provided by a human operator. Reference points are also termed *fiducial points*. Interactive image registration is well supported by the MATLAB Image Processing Toolbox and includes a graphically based program, `cpselect`, that automates the process of establishing corresponding reference marks. Under this procedure, the user interactively identifies a number of corresponding features in the reference and input image, and a transformation is constructed from these pairs of reference points. The program must specify the type of transformation to be performed ('linear', 'affine', or 'projective') and the minimum number of reference pairs required will depend on the type of transformation. The number of reference pairs required is the same as the number of variables needed to define a transformation: an affine transformation will require a minimum of three reference points while a projective transformation requires four variables. Linear transformations require only two pairs while other, more complex, transformations may require six or more point pairs. In most cases, the alignment is improved if *more* than the minimal number of point pairs is given.

In Example 13.8, an alignment requiring a projective transformation is presented. This example uses the routine `cp2tform` to produce a transformation in `Tform` format, based on point pairs obtained interactively. The `cp2tform` routine has a large number of options, but the basic calling structure is

```
Tform = cp2tform(input_points, base_points, 'type');
```

where `input_points` is an m -by-2 matrix consisting of x , y coordinates of the reference points in the input image and `base_points` is a matrix containing the same information for the reference image. This routine assumes that the points are entered in the same order; that is, that corresponding rows in the two vectors describe corresponding points. The 'type' variable is the same as in `maketform` and specifies the type of transformation ('affine', 'projective', etc.). The use of this routine is demonstrated in Example 13.8.

EXAMPLE 13.8

An example of interactive image registration. In this example, an input image is generated by transforming the reference image with a projective transformation, including vertical and horizontal translations. The program then realigns the image using interactive graphics.

Biosignal and Medical Image Processing

Solution

The program opens two windows displaying the reference and input images. Then four reference points are acquired for each image from the operator using MATLAB's `ginput`. As each point is taken, it is displayed as an "x" overlaid on the image. Once all eight points have been acquired (four from each image), a transformation is constructed using `cp2tform`. This transformation is then applied to the input image using `imtransform`. The reference, input, and realigned images are displayed.

```
% Example 13.8 Interactive Image Registration
%
nu_points = 4; % Number of reference points
.....Load mri.tif, frame 18 .....
[M N] = size(I);
%
% Construct transformed input image.
U = [1 1; 1 M; N M; N 1];
offset = .15*N; % Projection offset
H = .2 * N; % Horizontal translation
V = .15 * M; % Vertical translation
X = [1-offset + H 1 + offset-V; 1 + offset + H M-offset-V;...
      N-offset + H M-offset-V; N + offset + H 1 + offset-V];
Tform1 = maketform('projective', U, X);
I_input = imtransform(I_ref, Tform1, 'Xdata',...
    [1 N], 'Ydata', [1 M]);
%
% Acquire reference point. Display both images.
fig(1) = figure;
imshow(I_ref);
fig(2) = figure;
imshow(I_input);
%
%
for i = 1:2 % Get reference points
    figure(fig(i)); % Open window i
    hold on;
    title('Enter four reference points');
    for j = 1:nu_points
        [x(j,i), y(j,i)] = ginput(1); % Get ref. point
        plot(x(j,i), y(j,i), 'x'); % Mark with x
    end
end
%
% Construct transformation structure with cp2tform
% and implement with imtransform
%
[Tform2, inpts, base_pts] = cp2tform([x(:,2)...
    y(:,2)], [x(:,1) y(:,1)], 'projective');
I_aligned = imtransform(I_input, Tform2, ...
    'Xdata', [1 N], 'Ydata', [1 M]);
%
.....display transformed images.....
```

Results

The reference and input windows are shown along with the reference points selected in Figure 13.16. The minimal four reference points were used, although more could have been used, probably leading to an improved result. The influence of the number of reference points used is

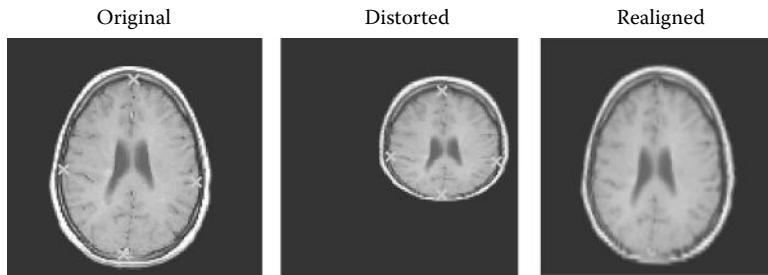


Figure 13.16 Interactive image registration from Example 13.8. The image on the left is the reference image and has been marked with four reference points. The corresponding points on the distorted input (center) image were then indicated interactively by a human operator. The resultant transformation of the distorted center image is shown on the right and closely matches the original (correlation 95%). Some reduction in sharpness is seen in the realigned image as a result of information lost in the distortion process.

explored in Problem 13.16. The result of the transformation is shown in Figure 13.16. Considering the distortion of the input image (middle image), the alignment is good and the correlation after alignment is 0.95. The primary advantage of this method is that it couples into the extraordinary abilities of human visual identification and, hence, can be applied to images that are only vaguely similar when unaided correlation-based methods would surely fail.

13.5 Summary

Many 1-D signal-processing routines extend directly to two dimensions. In this chapter, we explored the 2-D Fourier transform and 2-D filtering, both supported by MATLAB routines. While the 2-D Fourier transform of most images is difficult to interpret, it and its inverse are particular useful in medical image processing. In Chapter 15, we will find that the inverse Fourier transform is a critical operation in magnetic resonance imaging. 2-D filtering is a straightforward extension of the 1-D filters and can be implemented using the 2-D version of convolution. However, MATLAB implements image filtering using correlation and applies a correlation kernel, which is a rotated version of the traditional filter coefficients. These eliminate the “coefficient reversal” inherent in convolution and make it easier to conceptualize the effect of a given 2-D filter. Of course, if the filter coefficients are symmetric, it does not matter if correlation or convolution is used, as shown in one of the problems.

Spatial transformations are important operations in medical image registration. Image registration is needed to compare cross-modality images (e.g., fMRI and CT), to access disease progression from images taken at different times, and to align sequential images obtained in fMRI. Biomedical image transformations generally fall into two categories, both supported by MATLAB: affine transformations, where straight parallel lines remain straight and parallel, and projective translations, where straight parallel lines remain straight but converge. MATLAB uses the spatial coordinate system described in Section 12.1.1 to define transformation parameters. Transformations usually require interpolation of the transformed image over a rectangular pixel grid.

Image registration is carried out either unaided, that is, automatically by the computer program, or interactively with a human operator. Unaided image registration relies on optimizing a cost function, a parameter that describes the correctness of the alignment. Usually, the correlation between a reference image and the realigned image is used. In aided registration, an operator selects corresponding reference points known as fiducial points. Sometimes, these

Biosignal and Medical Image Processing

fiduciary points can be incorporated into an image by using special markers placed in the tissue or material being imaged.

PROBLEMS

- 13.1 Load the MATLAB test pattern image, I , found in `testpat1.png`. Plot the image along with a plot of the Fourier transform of this image. Then remove the mean of the image, take the Fourier transform again, and plot only the 50 points. Note that more detail is visible after the image mean has been removed. Use `mesh` to plot the Fourier transform and shift the transform before plotting using `fftshift`.
- 13.2 Load the image in `double_chirp.tif`, which has a chirp going both horizontally and vertically. Take the Fourier transform and plot as in Problem 13.1. Rather than subtract the mean of the image, simply remove the first point before applying `fftshift`. This is a more effective way of removing the “DC” component of the image. Show the original image and use `mesh` to plot the magnitude of the Fourier transform.
- 13.3 Load the horizontal chirp pattern shown in Figure 13.13 (found as `'im_chirp.tif'`) and take the Fourier transform as in the above problem. Then, line by line, multiply the Fourier transform in the horizontal direction (i.e., row by row) by a half-wave sine function of the same length. Now take the inverse Fourier transform of this windowed function and plot it alongside the original image. Also apply the window in the vertical direction (i.e., column by column), take the inverse Fourier transform, and plot the resulting image. Do not apply `fftshift` to the Fourier transform, since the inverse Fourier transform routine, `ifft2`, expects the DC component to be in the upper left corner as `fft2` initially presents it. Also, taking the inverse Fourier transform results in a small imaginary component, so take the real part and use `mat2gray` to normalize the image intensities. (The chirp image is square, so you do not have to recompute the half-wave sine function; however, you may want to plot the sine wave to verify that you have a correct half-wave sine function.) You should be able to explain the resulting images. [Hint: Recall the frequency spectrum of the two-point central difference algorithm used for taking the derivative. Also note the orientation of the Fourier transform of the chirp image in Figure 13.4.]
- 13.4 Load the blood cell image (`blood1.tif`). Design and implement your own 3-by-3 filter that enhances vertical edges that go from dark to light. Repeat for a filter that enhances horizontal edges that go from light to dark. Plot the two images along with the original. Convert the first image (vertical edge enhancement) to a binary image and adjust the threshold to emphasize the edges. Plot this image with the others in the same figure (i.e., use `subplot`). Plot the 3-D magnitude spectra of the two filters together in another figure.
- 13.5 Load the chirp image, `imchirp.tif`. Design a 1-D 24th-order narrowband bandpass filter with cutoff frequencies of 0.1 and 0.125 Hz and apply it to the chirp image. Plot the modified image with the original. Repeat for a 64th-order filter and plot the result with the others. In another figure, plot the 3-D frequency representation of 64th-order filter.
- 13.6 Load the slightly blurry image of the brain in `blur_brain.tif`. Apply an “unsharp” filter generated using `fspecial`. To improve the filtering, apply the filter twice. Plot the two images side by side and note the modest improvement in image sharpness along with the introduction of some artifact. Do not apply

`mat2gray` to the filtered image as it will reduce the contrast since the filtered image has a few pixels greater than 1.0. Also plot the magnitude of the Fourier transform of the unsharp filter and note that it has the features of a highpass filter. Deblurring an image is an art, and like so many signal-processing activities, it will benefit from the knowledge of the process that blurred the original image.

- 13.7 Load the image of the brain in `noise_brain.tif` that has “salt and pepper” noise. Apply a Gaussian lowpass filter generated using `fspecial`. Adjust the parameters of the filter to eliminate the noise without unduly reducing the sharpness of the image. Plot the original and filtered images side by side. As in Problem 13.6, you do not need to use `mat2gray` on the filtered image. Also plot the magnitude of the Fourier transform of the Gaussian filter and note that it has the features of a lowpass filter.
- 13.8 Load the image of the brain in `noise_brain2.tif` that has Gaussian noise. As in Problem 13.7, apply a Gaussian filter and adjust the parameters of the filter to eliminate the noise without unduly reducing the sharpness of the image. Plot the original and filtered images side by side. Again, you do not need to use `mat2gray` on the filtered image.
- 13.9 Produce a movie of the rotating brain. Load frame 16 of the MRI image (`mri.tif`). Make a multiframe image of the basic image by rotating that image through 360°. Use 36 (10° per rotation) frames to cover the complete 360°. (If your resources permit, you could use 64 frames with 5° per rotation.) Submit a montage plot of those frames that cover the first 90° of rotation; that is, the first eight images (or 16, if you use 64 frames).
- 13.10 “The expanding brain.” Back in the 1960s, people were into “expanding their minds” through meditation, drugs, rock and roll, or other such experiences. In this problem, you will expand the brain in a movie using an affine transformation. (Note that `imresize` will not work because it changes the number of pixels in the image and `immovie` requires that all images have the same dimensions.) Load frame 18 of the MRI image (`mri.tif`). Make a movie where the brain stretches in and out horizontally from 75% to 150% of the normal size. The image will probably exceed the frame size during its larger excursions, but this is acceptable. The image should grow symmetrically about the center (i.e., in both directions). Use around 24 frames with the latter half of the frames being the reverse of the first as in Example 13.6, so the brain appears to grow and then shrink. Note: Use some care in getting the range of image sizes to be between 75% and 150%. [Hint: To simplify the computation of the output triangle, it is best to define the input triangle at three of the image corners. All three triangle vertices will have to be modified to stretch the image in both directions, symmetrically about the center.] Submit a montage of the first 12 images.
- 13.11 “The tilting brain.” Produce a spatial transformation movie using a projective transformation. Load a frame of the MRI image (`mri.tif`, your choice of frame). Use the projective transformation to make a movie of the image as it tilts vertically. Use 24 frames as in Example 13.6: the first 12 will tilt the image back while the rest tilt the image back to its original position. You can use any reasonable transformation that gives a vertical tilt or rotation. Submit a montage of the first 12 images.
- 13.12 “The squishing brain.” Load frame 12 of `mri.tif`. Use a transformation to make a movie of the brain sliding into the left side of the picture and squishing against the left side. In other words, after it touches the left edge, it should decrease horizontally

Biosignal and Medical Image Processing

and increase vertically. As it decreases horizontally, the left side of the brain should remain against the wall as the brain flattens. Use about 12 frames to translate the brain to the left edge and another 18 frames to squish the brain (using an affine transformation) against the edge. This problem is easier to solve using two separate transformation loops.

- 13.13 Load frame 12 of `mri.tif` and use `imrotate` to rotate the image by 15° clockwise. (Use the bicubic interpolation method.) Use MATLAB's basic optimization program `fminsearch` to align the image that has been rotated. (You will need to write a function similar to `realign` in Example 13.7 that rotates the image based on the first input parameter, and then computes the negative correlation between the rotated image and the original image.)
- 13.14 Image registration after affine spatial transformation. Load the file `transform1.mat`, which contains a reference image `I` and a transformed image `I_input`. The transformed image has been shifted in both the `x` and `y` directions and scaled both vertically and horizontally. Modify Example 13.7 to use `fminsearch` to align `I_input` to `I`. You will need four adjustable parameters to accomplish this: `X_shift`, `Y_shift`, `H_scale`, and `Y_scale`. Output the correlation of the aligned and reference images. Correlation should be over 95% for this problem.
- 13.15 Image registration after projective spatial transformation. Load the file `transform2.mat`, which contains a reference image `I` and a transformed image `I_input`. The transformed image has been tilted. Unlike Example 13.6 where the tilt was just backward and only one parameter was used to define tilt, here the tilt could also be along the vertical axis. Moreover the tilt forward and backward is not necessarily the same. Therefore, you will need to adjust all four corners independently and will require four adjustable parameters. Note that this transformation is challenging and will not result in as high a correlation as in Problem 13.14. Changing the `fminsearch` stopping criteria using `optimset` could improve the final correlation.
- 13.16 Load file `transform3.mat`, which contains a reference image `I` and a transformed image `I_input`. Use interactive registration and the MATLAB function `cp2tform` to transform the image. Use (a) the minimum number of points and (b) twice the minimum number of points. Compare the correlation between the original and the realigned images using the different number of reference points. [Hint: Enlarge the images on the screen to improve placement of the markers.]

14

Image Segmentation

14.1 Introduction

Image segmentation is the identification and isolation of an image into regions that, one hopes, correspond to structural units. It is an especially important operation in biomedical image processing since it is used to isolate physiological and biological structures of interest. The problems associated with segmentation have been well studied and a large number of approaches have been developed, many specific to particular image features. The general approaches to segmentation can be grouped into four classes: pixel-based, regional or continuity-based, edge-based, and morphological methods. Pixel-based methods are the easiest to understand and to implement, but are also the least powerful and, since they operate on one element at a time, are particularly susceptible to noise. Continuity-based and edge-based methods approach the segmentation problem from opposing sides: edge-based methods search for differences while continuity-based methods search for similarities. Morphological methods use information on shape to constrain or define the segmented image. Recently, information concerning the mechanics and dynamics of the imaged tissue has been adapted in advanced approaches to segmentation. In biomedical imaging, segmentation is often very challenging, and multiple approaches are used.

14.2 Pixel-Based Methods

The most straightforward and common of the pixel-based segmentation methods is *thresholding* in which all pixels having intensity values above or below some level are classified as part of the segment. Thresholding is an integral part of converting an intensity image into a binary image as demonstrated in Example 14.2. Even when preceded by other approaches, thresholding is usually required to produce a segmentation *mask*, a black-and-white image of the feature of interest.

Thresholding is usually quite fast and can be done in real time, allowing for interactive adjustment of the threshold. The basic concept of thresholding can be extended to include both upper and lower boundaries, an operation termed *slicing* since it isolates a specific range of pixel intensities. Slicing can be generalized to include a number of different upper and lower boundaries, each encoded into a different number. An example of multiple slicing was presented in Chapter 12 using MATLAB's `gray2slice` routine. Finally, when an RGB image is involved, thresholding can be applied to each color plane separately. The resulting image can be either a thresholded RGB image or a single image composed of a logical combination (AND or OR) of the three image planes after thresholding. An example of this approach is seen in the problems.

Biosignal and Medical Image Processing

A technique that can aid in all image analysis approaches, but is particularly useful in pixel-based methods, is intensity remapping. In this global procedure, pixel values are rescaled so as to extend over different maximum and minimum values. Usually, the rescaling is linear; so, each point is adjusted proportionally with a possible offset. MATLAB supports rescaling with the routine `imadjust` described below, which also provides a few common nonlinear rescaling options. Of course, any rescaling operation is possible using MATLAB code if the intensity images are in double format or the image arithmetic routines described in Chapter 12 are used.

14.2.1 Threshold Level Adjustment

A major concern in pixel-based methods is setting the threshold or slicing level(s) appropriately. Usually, these levels are set by the program, although in some situations, they are set interactively by the user. Finding an appropriate threshold level can be aided by a plot of the distributions of pixel intensity over the image, regardless of whether you adjust pixel level interactively or automatically. Such a plot is termed the *intensity histogram* and is supported by the MATLAB routine `imhist` described below. Figure 14.1 shows an x-ray image of the spine with its associated intensity histogram. Figure 14.1 also shows the binary image obtained by applying a threshold at a specific point on the histogram indicated by the vertical line. When RGB color images are being analyzed, intensity histograms can be obtained from all three color planes and different thresholds can be established for each color plane with the aid of the corresponding histogram.

Intensity histograms can be very helpful in selecting threshold levels, not only for the original image, but also for images produced by various segmentation algorithms described later. Intensity histograms can also be useful in evaluating the efficacy of different processing schemes: as the separation between structures improves, histogram peaks should become more distinctive. This relationship between separation and histogram shape is demonstrated in Figure 14.2 and, more dramatically, in Figures 14.7 and 14.11.

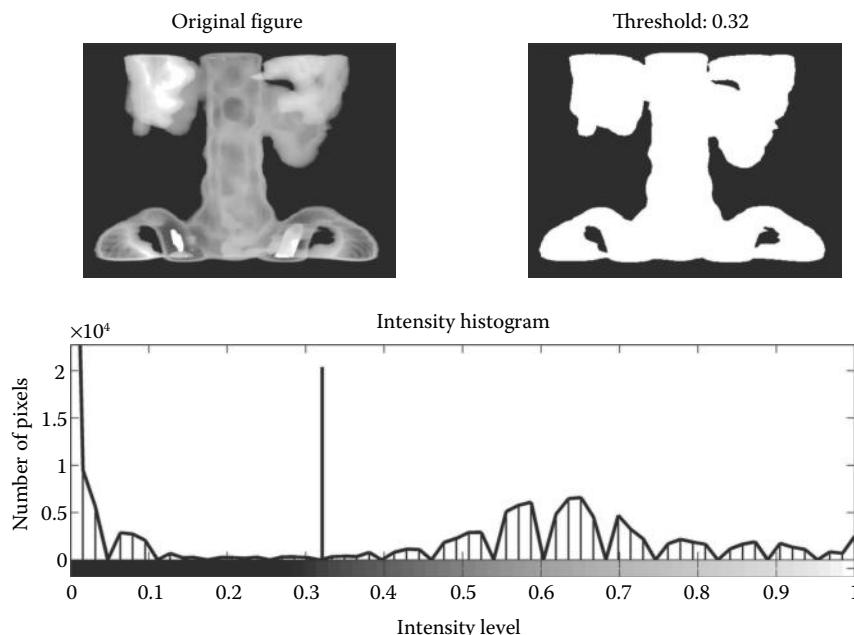


Figure 14.1 X-ray image of the spine, upper left, and its associated intensity histogram, lower plot. The upper right image is obtained by thresholding the original image at a value corresponding to the vertical line on the histogram plot. (Image courtesy of MATLAB.)

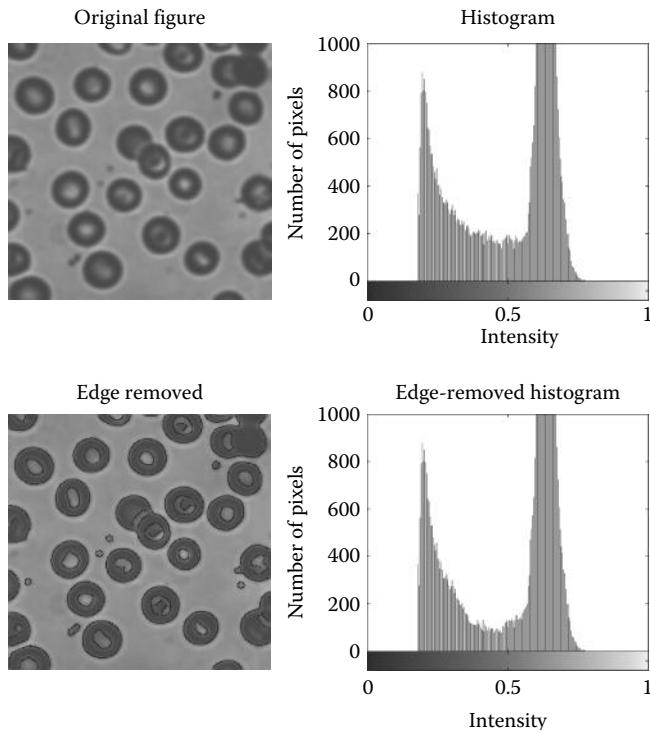


Figure 14.2 Image of blood cells with (upper) and without (lower) intermediate boundaries removed. The associated histograms (right side) show improved separability when the boundaries are eliminated. The code that generated these images is given in Example 14.1.

Although intensity histograms contain no information on position, they can still be useful for segmentation, particularly for estimating threshold(s) from the histogram (Sonka et al. 1993). If the intensity histogram is, or is assumed to be, bimodal (or multimodal), a common strategy is to search for low points or minima in the histogram. This is the strategy used in Figure 14.1 where the threshold is set at 0.32, the intensity value where the histogram shows a minimum. Such points represent the fewest number of pixels, but often, the histogram minima are difficult to determine visually.

An approach to improve the determination of histogram minima is based on the observation that many boundary points carry values intermediate to those on either side of the boundary. These intermediate values lie between the actual boundary values and may mask the optimal threshold value. However, these intermediate points also have the highest gradient, and it should be possible to identify them using a gradient-sensitive filter such as the Sobel or Canny filter. After these boundary points are identified, they can be eliminated from the image and a new histogram can be computed that has a more definitive distribution. This strategy is used in Example 14.1, and Figure 14.2 shows the images and associated histograms before and after the removal of boundary points identified using the Canny filtering. A slight reduction in the number of intermediate points can be seen in the middle of the histogram (around 0.45). As shown in Figure 14.3, this leads to somewhat better segmentation of the blood cells when the threshold is based on the histogram as explained in Example 14.1.

Another histogram-based strategy that can be used if the distribution is bimodal is to assume that each mode is the result of a unimodal, Gaussian distribution. An estimate is then made of the underlying distributions, and the point at which the two estimated distributions intersect

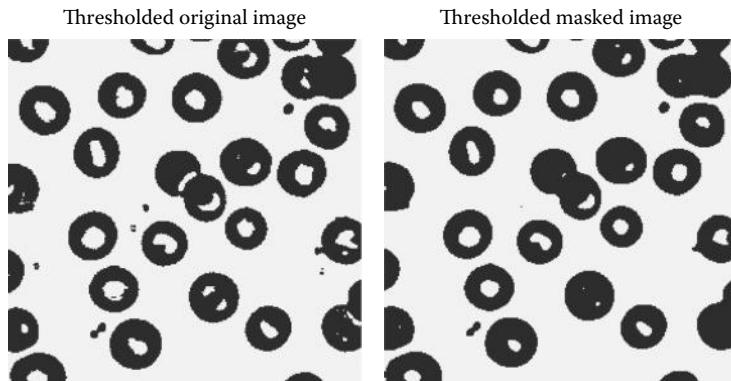


Figure 14.3 Thresholded blood cell images. Thresholds based solely on the histogram (as explained below) were applied to the blood cell images in Figure 14.2 with (left) and without (right) boundary pixels removed (i.e., set to zero). Somewhat fewer inappropriate pixels are seen in the right image.

should provide the optimal threshold. The principal problem with this approach is that the distributions are unlikely to be Gaussian.

A threshold strategy that does not use the histogram directly is based on the concept of minimizing the variance between presumed foreground and background elements. Although the method assumes two different gray levels, it works well even when the distribution is not bimodal (Sonka et al. 1993). The approach termed *Outso's method* uses an iterative process to find a threshold that minimizes the *variance* between the intensity values on either side of the threshold level. This approach is implemented using the MATLAB routine `graythresh` described below and used in Example 14.1.

A pixel-based technique that provides a segment boundary directly is *contour mapping*. Contours are lines of equal intensity and, in a continuous grayscale image, they are necessarily continuous: they cannot end within the image, although they can branch or loop back on themselves. In digital images, these same properties exist, but the value of any given contour line generally requires interpolation between adjacent pixels. To use contour mapping to identify image structures requires accurate setting of the contour levels, and this carries the same burdens as thresholding. Nonetheless, contour maps do provide boundaries directly and, if “subpixel” interpolation is used in establishing the contour position, they may be more spatially accurate. Contour maps are easy to implement in MATLAB, as shown in the next section. Figure 14.4 shows

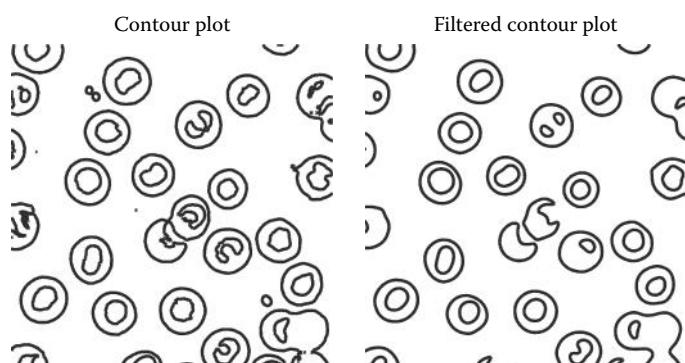


Figure 14.4 Contour maps drawn from the blood cell image of Figures 14.2 and 14.3. The right image was prefiltered with a Gaussian lowpass filter ($\alpha = 3$) before the contour lines were drawn. The contour values were set manually to provide good outlines.

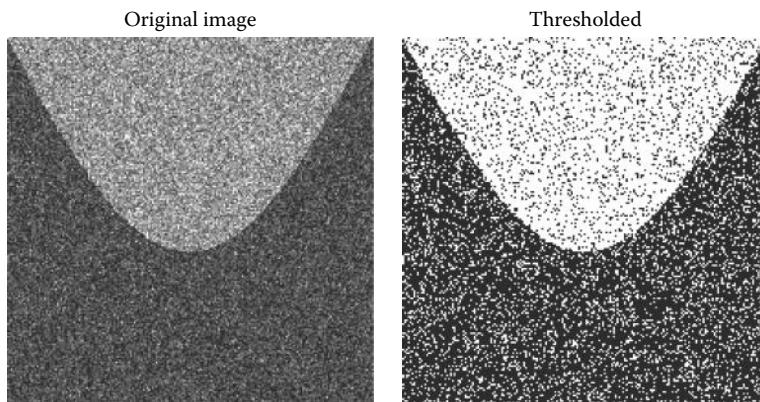


Figure 14.5 Image with two regions that have different average gray levels. The two regions are clearly distinguishable, but it is not possible to accurately segment the two regions using thresholding alone because of noise.

contour maps for the blood cell images shown in Figure 14.2. The right image was prefiltered with a Gaussian lowpass filter that reduces noise slightly and improves the resultant contour image.

Pixel-based approaches can lead to serious errors, even when the average intensities of the various segments are clearly different, due to noise-induced intensity variation within the structure. Such variation could be acquired during image acquisition, but could also be inherent in the structure itself. Figure 14.5 shows two regions with quite different average intensities. Even with optimal threshold selection, many inappropriate pixels are found in both segments due to intensity variations within the segments. The techniques for improving separation in such images are explored in Section 14.3.

14.2.2 MATLAB Implementation

Some of the routines for implementing pixel-based operations such as `im2bw` and `grayslice` have been described in the preceding chapters. The image intensity histogram routine is produced by `imhist` without the output arguments:

```
[counts, x] = imhist(I, N);
```

where `counts` is the histogram value at a given `x`, `I` is the image, and `N` is an optional argument specifying the number of histogram bins (the default is 255). As mentioned above, `imhist` is usually invoked without the output arguments to produce a plot directly. Several pixel-based techniques are presented in Example 14.1.

An automated thresholding operation is performed by the MATLAB routine `graythresh`. This program determines a threshold based on the intensity histogram using Outso's method. As mentioned above, this method iteratively adjusts the threshold to minimize the variance on either side. The routine is called as

```
thresh = graythresh(I); % Determine threshold using Outso's method
```

The output threshold can be used directly in `im2bw` to construct a thresholded image (in fact, the routine can be embedded in the call to `im2bw`).

EXAMPLE 14.1

An example of segmentation using pixel-based methods. Load the image of blood cells and display this image along with the intensity histogram. Remove the edge pixels from the image and

Biosignal and Medical Image Processing

display the histogram of this modified image. Determine thresholds using the minimal variance iterative technique (Outso's method) and apply this approach to threshold both the original and modified images. Display the resultant thresholded images.

Solution

To remove the edge boundaries, first identify these boundaries using an edge detection scheme. While any of the edge detection filters described previously can be used, here, we use the Canny filter as it is the most robust to noise. This filter is described in Section 14.6 and is implemented as an option of MATLAB's edge routine, which produces a binary image of the boundaries. This binary image is converted into a boundary *mask* by inverting the image using the NOT operator. After inversion, the edge pixels will be zero while all other pixels will be one. Multiplying the original image by the boundary mask produces an image in which the boundary points are removed (i.e., set to zero or black). Then graythresh is used to find the minimum variance thresholds that are used in im2bw to obtain thresholded images. All the images involved in this process, including the original image, are then displayed.

```
% Example 14.1 Pixel-based segmentation
%
.....input image and convert to double.....
%
h = fspecial('gaussian',14,2);           % Gaussian filter
I_f = imfilter(I,h,'replicate');         % Filter image
%
I_edge = edge(I_f,'canny',0.3);          % Find edge points
% Complement edge points and mask using multiplication
I_rem = I_f .* ~ I_edge
      ..... Display images and histograms, new figure.....
%
% Get thresholds
t1 = graythresh(I);                     % Use minimum variance
t2 = graythresh(I_f);                   % (Outso's) method
BW1 = im2bw(I,t1);                    % Threshold images
BW2 = im2bw(I_f,t2));
.....display thresholded images.....
```

Results

The results have been shown previously in Figures 14.2 and 14.3 and the improvement in the histogram and threshold separation has been mentioned. The edge image and its complement used as a mask are shown in Figure 14.6. While the change in the histogram is fairly small (Figure 14.2), it does lead to a reduction in artifacts in the thresholded image as shown in Figure 14.3. This small improvement can be quite significant in some applications. The methods for removing the small artifacts remaining will be described in Section 14.5.

14.3 Continuity-Based Methods

Continuity-based approaches look for similarity or consistency in the search for feature elements. These approaches can be very effective in segmentation tasks, but they all tend to reduce edge definition. This is because they are based on neighborhood operations that operate on a local area and blur the distinction between edge and feature regions. The larger the neighborhood used, the more poorly edges will be defined. Increasing neighborhood size usually improves the power of any given continuity-based operation, setting up a compromise between identification ability and edge definition.

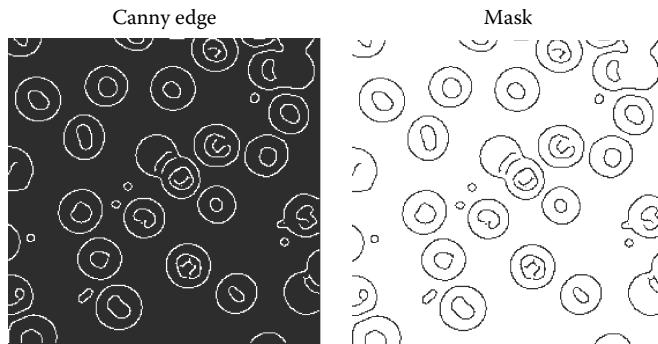


Figure 14.6 The BW image of edges produced by the MATLAB edge routine using the Canny filter (left). The inverted image, right, is used to mask (i.e., remove) boundaries in the cell image as seen in Figure 14.2. Eliminating these intermediate boundary values slightly improves threshold-based segmentation (Figure 14.3).

One simple continuity-based technique is lowpass filtering. Since a lowpass filter is a sliding neighborhood operation that takes a weighted average over a region, it enhances consistent features. Figure 14.7 shows histograms of the image in Figure 14.5 before and after filtering with a 10×10 Gaussian lowpass filter ($\alpha = 1.5$).

After lowpass filtering, the two regions are evident in the histogram (Figure 14.7) and the boundary found by minimum variance results in perfectly isolated segments as shown in Figure 14.8. The thresholding uses the same minimum variance technique in both Figures 14.5 and 14.8 and the improvement brought about by simple lowpass filtering is remarkable.

Image features related to *texture* can be particularly useful in segmentation. Figure 14.9 shows three regions that have approximately the same average intensity values, but are readily distinguished visually because of differences in texture. Several neighborhood-based operations can be used to distinguish textures: the small-segment Fourier transform, local variance (or standard deviation), the *Laplacian* operator, the *range* operator (the difference between maximum and minimum pixel values in the neighborhood), the *Hurst* operator (maximum difference as a function of pixel separation), and the *Haralick* operator (a measure of distance moment). Many of these approaches are directly supported in MATLAB or can be implemented using the `nlfilter` routine described in Chapter 12. Example 14.2 attempts to separate the three regions shown in Figure 14.9 by applying one of these operators to convert the texture pattern into a difference in intensity that can then be separated using thresholding.

14.3.1 MATLAB Implementation

EXAMPLE 14.2

Separate out the three segments in Figure 14.9 that differ only in texture. Use one of the texture operators described above and demonstrate the improvement in separability through histogram plots. Determine the appropriate threshold levels for the three segments from the histogram plot.

Solution

Use the nonlinear range operator to convert the textural patterns into differences in intensity. The *range operator* is a sliding neighborhood procedure that sets the center pixel to the difference between the maximum and minimum pixel value with the neighborhood. Implement this operation using MATLAB's `nlfilter` routine with a 7×7 neighborhood. This neighborhood size was empirically found to produce good results. The three regions are then thresholded using

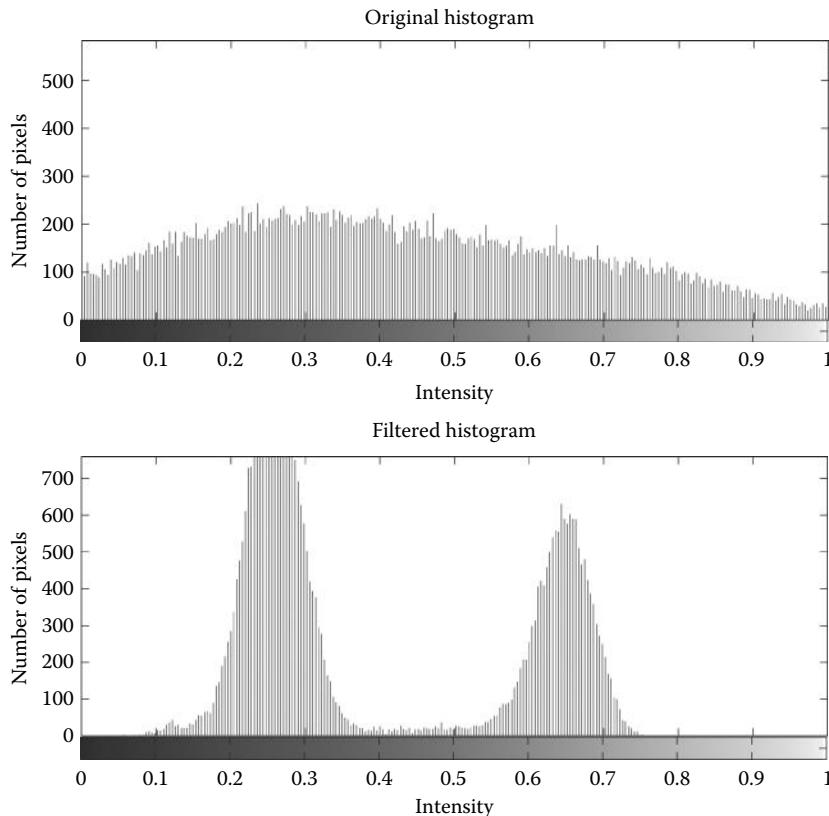


Figure 14.7 Histogram of the image shown in Figure 14.5 before (upper) and after (lower) lowpass filtering. Before filtering, the two regions overlap to such an extent that they cannot be identified in the histogram. After lowpass filtering, the two regions are evident and the boundary found by minimum variance is 0.45. The application of this boundary to the filtered image results in perfect separation as shown in Figure 14.8.

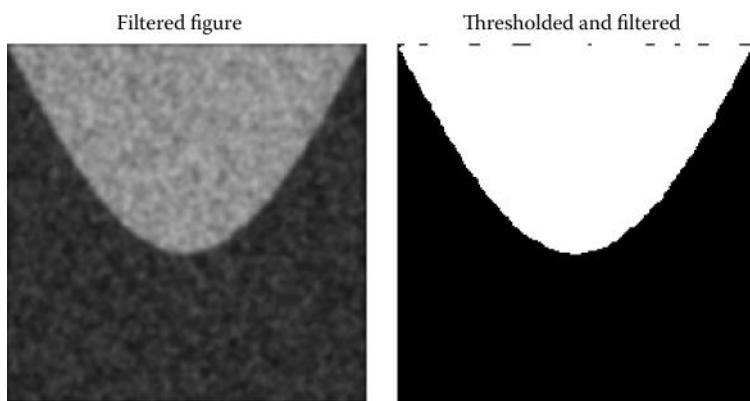


Figure 14.8 The same image as in Figure 14.5 after lowpass filtering. The two features can now be separated perfectly by thresholding as demonstrated in the right-hand image.

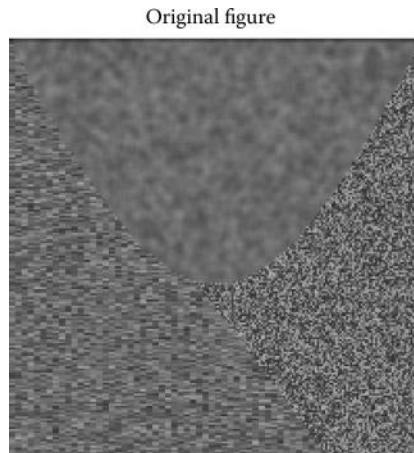


Figure 14.9 Image containing three regions having approximately the same intensity, but different textures. While these areas can be distinguished visually, separation based on intensity or edges will surely fail.

an empirically determined threshold of 0.22 and 0.55. The upper texture is segmented by inverting the BW image formed by the lower threshold and the right side texture is segmented by the upper threshold. The remaining texture can be found using a logical combination (the AND operation) applied to the two isolated images after inverting.

```
% Example 14.2 Segmentation of textures using the range operator
%
I = imread('texture3.tif'); % Load image and
I = im2double(I); % Convert to double
%
range = inline('max(max(x)) - min(min(x))'); % Range operator
I_f = nlfilter(I,[7 7], range); % Range operator
I_f = mat2gray(I_f); % Rescale intensities
%
.....Display range operation image and histograms.....
% Threshold and display segments, invert as needed, subplot as needed
BW1 = ~im2bw(I_f,.21); % Isolate upper texture
BW2 = im2bw(I_f,.55); % Isolate right-side texture
BW3 = ~BW1 & ~BW2; % Isolate remaining texture
```

Results

The image produced by the range filter is shown in Figure 14.10 and a clear distinction in intensity level can now be seen between the three regions. This is also demonstrated in the histogram plots of Figure 14.11. The histogram of the original figure (upper plot) shows a single Gaussian-like distribution with no evidence of the three patterns.* After filtering, the three patterns emerge as three distinct distributions. Using this distribution, two thresholds were chosen at a minima between the distributions (at 0.21 and 0.55: the solid vertical lines in Figure 14.11) and the three segments isolated based on these thresholds. The upper and right side patterns are isolated using `im2bw` and the third pattern is found for a logical combination of the two.

* In fact, the distribution is Gaussian since the image patterns were generated by filtering an array filled with Gaussianly distributed numbers generated by `randn`.

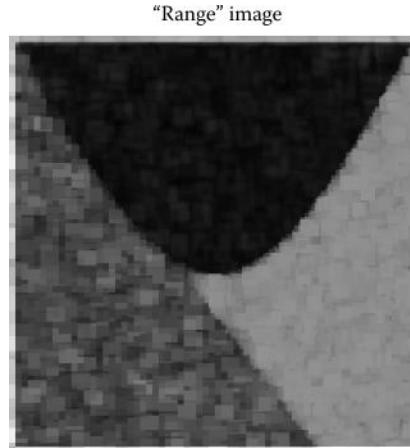


Figure 14.10 Texture pattern shown in Figure 14.9 after application of the nonlinear range operation. This operator converts the textural properties in the original figure into a difference in intensities. The three regions are now clearly visible as intensity differences and can be isolated using thresholding.

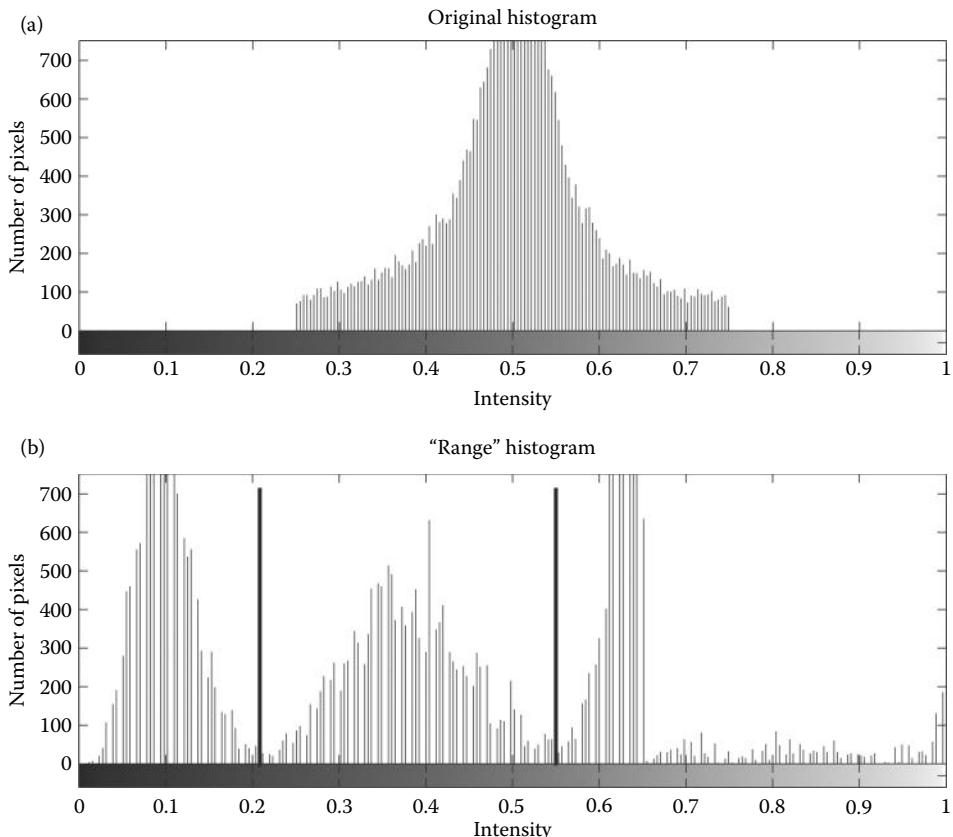


Figure 14.11 Histogram of the original texture pattern before (a) and after nonlinear filtering using the *range* operator (b). After filtering, the three intensity regions are clearly seen. The thresholds used to isolate the three segments are indicated.



Figure 14.12 Isolated regions of the texture pattern in Figure 14.9 produced by Example 14.2. Although there are some artifacts, the segmentation is quite good considering the original image. The methods for improving the segmented images are demonstrated in Section 14.5 and in the problems.

The three fairly well-separated regions are shown in Figure 14.12. A few artifacts remain in the isolated images and some of the morphological methods described in Section 14.5 could be used to eliminate or reduce these erroneous pixels. The separation can also be improved by applying lowpass filtering to the range image as demonstrated in one of the problems.

Occasionally, segments have similar intensities and textural properties, except that the texture differs in orientation. Such patterns can be distinguished using a variety of nonlinear operators that have orientation-specific properties. The local Fourier transform can also be used to distinguish orientation. Figure 14.13 shows a pattern with texture regions that are different only in terms of their orientation. To segment the three texture-specific features in this image, Example 14.3 uses a direction-specific operator followed by a lowpass filter that improves separation.

EXAMPLE 14.3

Isolate segments from a texture pattern that includes two patterns with the same textural statistical properties except for orientation. Also plot histograms of the original image and the image after application of the nonlinear filter.

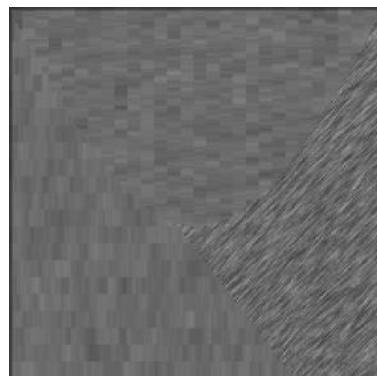


Figure 14.13 Textural pattern used in Example 14.3. The upper and left diagonal features have similar statistical properties and would not be separable with a standard range operator. However, they do have different orientations. As in the previous example, all three features have the same average intensity.

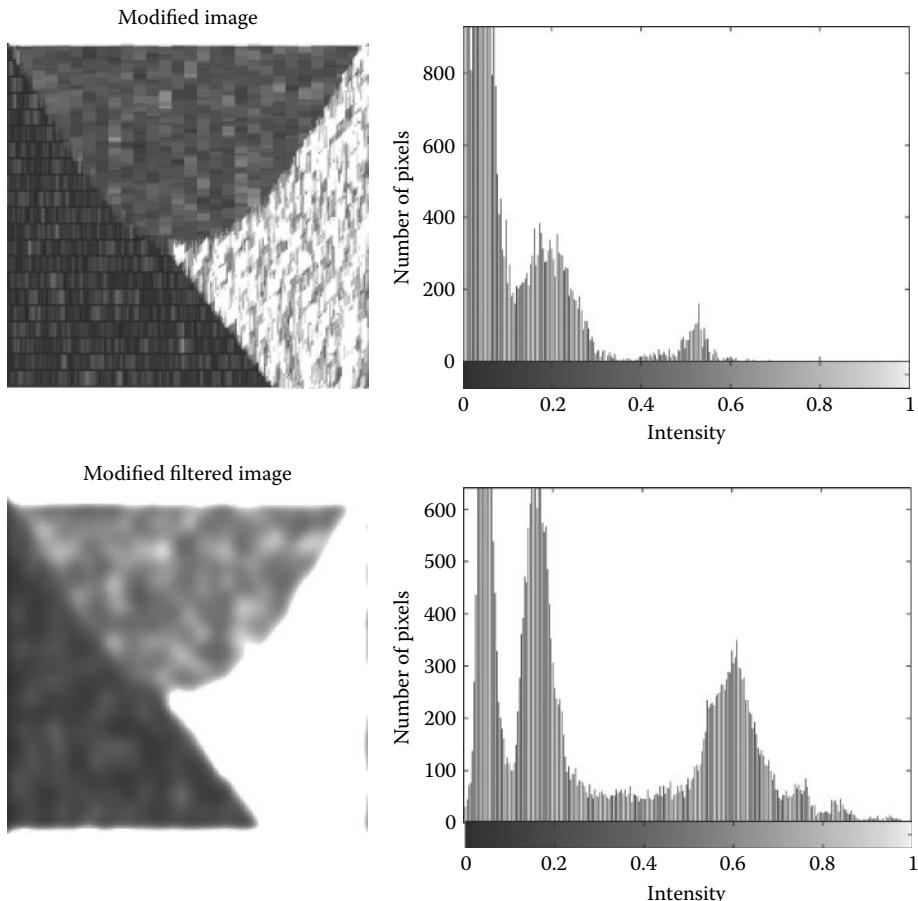


Figure 14.14 Images produced by the application of a directional range operator applied to the image in Figure 14.13 before (upper) and after (lower) lowpass filtering. The histograms demonstrate the improved separability of the filter image showing deeper minima in the filtered histogram.

Solution

Because of the similarity in the statistical properties of the vertical and horizontal patterns, the standard range operator used in Example 14.2 will not provide good separation. Instead, we apply a filter that has directional sensitivity. A Sobel or Prewitt filter can be used, followed by the range or similar operator, or the operations can be done in a single step by using a directional range operator. The choice made in this example is to use a horizontal range operator implemented with `n1filter`. This is followed by a strong lowpass filter (20×20 Gaussian, $\alpha = 4$) to improve separation by smoothing over intensity variations. Two segments are then isolated using standard thresholding. As in Example 14.2, the third segment is constructed by applying logical operations to the other two segments.

```
% Example 14.3 Analysis of texture pattern having similar textural
% characteristics but with different orientations..
%
I = imread('texture4.tif'); % Load texture
I = im2double(I); % Convert to double
%
```



Figure 14.15 Isolated segments produced by thresholding the lowpass filtered image in Figure 14.14. The rightmost segment was found by applying logical operations to the other two images.

```
% Define filters and functions
range = inline('max(x) - min(x)'); % Range function
b_lp = fspecial('gaussian', 20, 4); % Gaussian filter
I_nl = nlfilt(I, [9 1], range); % Directional filter
I_h = imfilter(I_nl*2, b_lp); % Scale image and filter
.....Display images and histograms.....
%
BW1 = ~im2bw(I_f,.1); % Isolate upper texture
BW2 = im2bw(I_f,.29); % Isolate right-side texture
BW3 = ~ BW1 & ~BW2; % Isolate remaining texture
.....Display image segments.....
```

Results

The image and histogram produced by the horizontal range operator with, and without, low-pass filtering are shown in Figure 14.14. The range operation produces a very dark image and it is multiplied by a scaling factor for better viewing; however, the BW masks were obtained by applying `im2bw` to the unscaled images. Note the improvement in separation produced by the lowpass filtering as indicated by the better defined peaks in the histogram. The thresholded images are shown in Figure 14.15. As in Example 14.3, the separation is not perfect, but it is quite good considering the challenges posed by the original image. Again, the separation could be further improved by morphological operations.

14.4 Multithresholding

The results of several different segmentation approaches can be combined either by adding the images together or, more commonly, by first thresholding the images into separate binary images and then combining them using logical operations. The AND, OR, or NOT (`~`) operator is used depending on the characteristics of each segmentation procedure. If each procedure identifies all the segments and also includes nondesired areas, the AND operator is used to reduce artifacts. An example of the use of the AND and NOT operations is given in Examples 14.2 and 14.3 where one segment is found using the inverse of a logical AND of the other two segments. Alternatively, if each procedure identifies some portion of the segment, then the OR operator is used to combine the various portions. This approach is illustrated in Example 14.4 where the first two, then three, thresholded images are combined to improve segment identification. The structure of interest is a cell that is shown on a gray background. Threshold levels above and below the gray background are combined (after one is inverted) to provide improved isolation. Including a third binary image obtained by thresholding a textured image further improves the identification.

EXAMPLE 14.4

Isolate the cell structures from the image of a cell shown in Figure 14.16.

Solution

Since the cell is projected against a gray background, it is possible to isolate some portions of the cell by thresholding above and below the background level. After inversion of the lower thresholded image (the one that is below the background level), the images are combined using a logical OR. Since the cell also shows some textural features, a textured image is constructed by taking the regional standard deviation. This textured image is then filtered with a lowpass filter (20×20 Gaussian, alpha = 2) and is shown in Figure 14.16, right hand image. After thresholding, this texture-based image is also combined with the other two images.

```
% Example 14.4 Analysis of the image of a cell using combined
% texture and intensity information
%
I = imread('cell.tif'); % Load cell image
I = im2double(I); % Convert to double
%
b = fspecial('gaussian', 20, 2); % Gaussian filter
%
I_std = nlfilter(I,[3 3], 'std2')*10; % Std operation
I_lp = imfilter(I_std, b); % Apply filter
%
.....Display original and filtered images, new figure.....
%
BW_th = im2bw(I,.5); % Threshold image
BW_thc = ~im2bw(I,.42); % and its complement
BW_std = im2bw(I_std,.2); % Thresh. texture image
BW1 = BW_th | BW_thc; % Combine two images
BW2 = BW_std | BW_th | BW_thc; % Combine all images
.....Display BW images and title.....
```

The original and texture/filtered images are shown in Figure 14.16. Note that the texture image has been scaled up by a factor of 10 to bring it within a nominal image range. The intensity

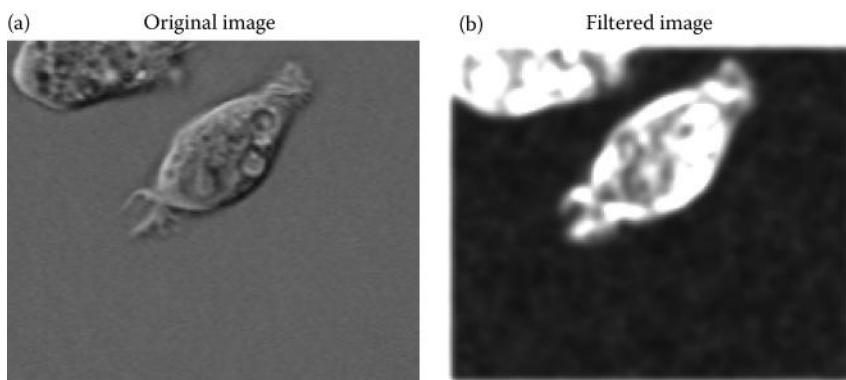


Figure 14.16 Image of cells (a) on a gray background. The textural image (b) was created based on local variance (standard deviation) followed by lowpass filtering and shows somewhat more definition. (Cancer cell from rat prostate, courtesy of Alan W. Partin, Johns Hopkins University School of Medicine.)

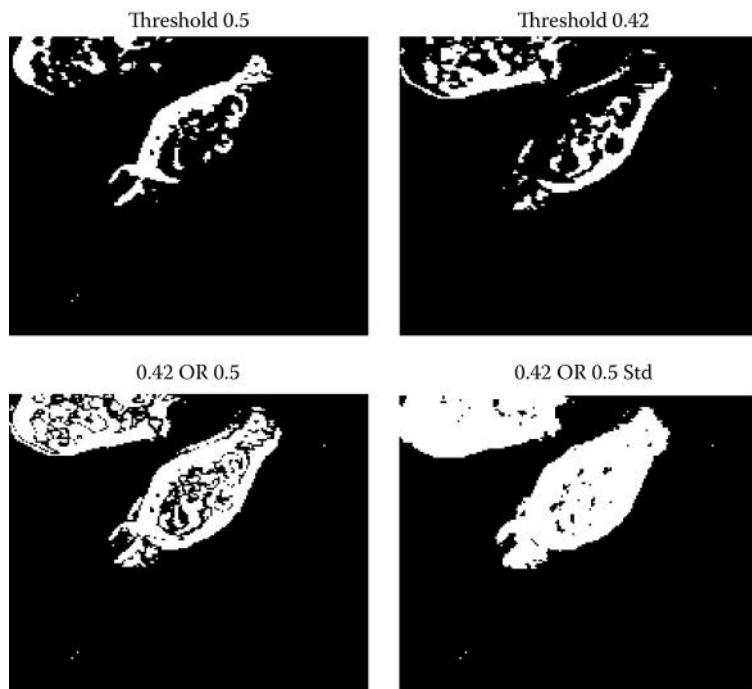


Figure 14.17 Isolated portions of the cells shown in Figure 14.16 (left image). The upper images were created by thresholding the intensity at the levels indicated. The lower left image is a combination (logical OR) of three images: the upper two BW images and a thresholded texture-based image.

thresholded images are shown in Figure 14.17 (upper images; the upper right image has been inverted) and the thresholds are shown. These images are ORed in the lower left image. The lower right image shows the OR of both thresholded images with a thresholded texture image. This method of combining images can be extended to any number of different segmentation approaches.

14.5 Morphological Operations

Morphological operations have to do with processing *shapes*. In this sense, they are both continuity based and may also operate on edges. In fact, morphological operations have many image-processing applications in addition to segmentation and they are well represented and supported in the MATLAB Image Processing Toolbox.

The two basic morphological operations are *dilation* and *erosion*. In dilation, the rich get richer, and in erosion, the poor get poorer. Specifically, in dilation, the center or active pixel is set to the maximum of its neighbors, and in erosion, it is set to the minimum of its neighbors. Since these operations are often performed on binary images, dilation tends to expand edges, borders, or regions, while erosion tends to decrease or even eliminate small regions. Obviously, the size and shape of the neighborhood used will have a very strong influence on the effect produced by either operation.

The two processes can be done in tandem over the same area. Since both erosion and dilation are nonlinear operations, they are not invertible transformations, that is, one followed by the other will not generally result in the original image. If erosion is followed by dilation, the operation is termed *opening*. If the image is binary, this combined operation will tend to remove small

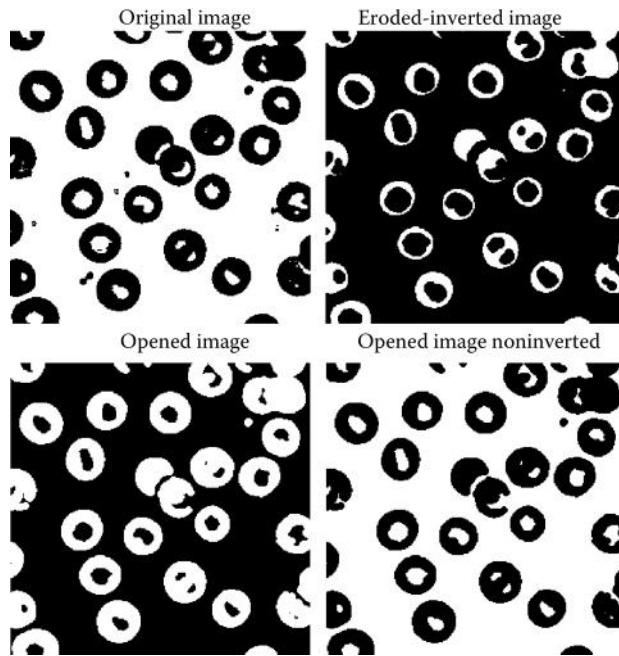


Figure 14.18 Example of the *opening* operation to remove small artifacts. Note that the final image has fewer background artifacts, but now one of the cells has a gap in the wall.

objects without changing the shape and size of larger objects. Basically, the initial erosion tends to reduce all objects, but some of the smaller objects will disappear altogether. The subsequent dilation will restore those objects that were not completely eliminated by erosion. If the order is reversed and dilation is performed first followed by erosion, the combined process is called *closing*. Closing connects objects that are close to each other, tends to fill up small holes, and smooths an object's outline by filling small gaps. As with the more fundamental operations of dilation and erosion, the size of objects removed by opening or filled by closing depends on the size and shape of the neighborhood that is selected.

An example of the opening operation is shown in Figure 14.18, including the erosion step. This is applied to the blood cell image after thresholding, the same image originally shown in Figure 14.3 (left side). Since we wish to eliminate black artifacts in the background, we first invert the image and erode as shown in Figure 14.18, upper right. As can be seen in the final, opened image, there is a reduction in the number of artifacts seen in the background, but now, there is also a gap created in one of the cell walls. The opening operation would be more effective on the image in which intermediate values were masked out (Figure 14.3, right side).

Figure 14.19 shows an example of closing applied to the same blood cell image. Again, the operation is performed on the inverted image. This operation tends to fill the gaps in the center of the cells, but it also fills in gaps between the cells. A much more effective approach to filling holes is to use the `imfill` routine described in Section 14.5.1.

Other MATLAB morphological routines provide local maxima and minima, and allow for manipulating the image's maxima and minima, which implement various fill-in effects.

14.5.1 MATLAB Implementation

The erosion and dilation could be implemented using the nonlinear filter routine `nlfilt`, although this routine limits the shape of the neighborhood to a rectangle. The MATLAB

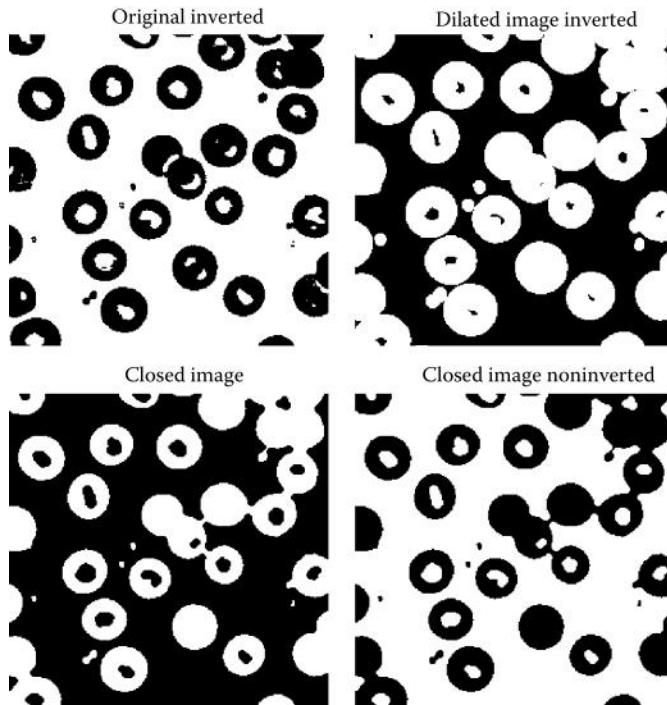


Figure 14.19 Example of *closing* to fill gaps. In the closed image, some of the cells are now filled, but some of the gaps between cells are also filled.

routines `imdilate` and `imerode` provide for a variety of neighborhood shapes and are much faster than `nlfilter`. As mentioned above, opening consists of erosion followed by dilation; closing is the reverse. MATLAB also provides routines for implementing these two operations in one statement.

To specify the neighborhood used by all these routines, MATLAB uses a *structuring element*.^{*} A structuring element is defined by a binary array, where the 1s represent the neighborhood and the 0s are irrelevant. This allows for easy specification of neighborhoods that are nonrectangular; indeed, that can have any arbitrary shape. In addition, MATLAB makes a number of popular shapes directly available.

The routine to specify the structuring element is `strel` and is called as

```
structure = strel(shape, NH, arg);
```

where `shape` is the type of shape desired, `NH` usually specifies the size of the neighborhood, and `arg` is an argument that applies to `shape`. If `shape` is '`'arbitrary'`', or simply omitted, then `NH` is an array that specifies the neighborhood in terms of the neighborhoods as described above. The prepackaged shapes include

<code>'disk'</code>	a circle of radius <code>NH</code> (in pixels)
<code>'line'</code>	a line of length <code>NH</code> and angle <code>arg</code> in degrees

* Not to be confused with a similar term, *structural unit* is used in the beginning of this chapter. This refers to an object of interest in the image.

Biosignal and Medical Image Processing

```
'rectangle'      a rectangle where NH is a two-element vector specifying
                  rows and columns
'diamond'        a diamond where NH is the distance from the center to each
                  corner
'square'         a square with linear dimensions NH
```

For many of these shapes, the routine `strel` produces a “decomposed” structure that runs significantly faster.

Given a structure designed by `strel`, the statements for dilation, erosion, opening, and closing are

```
I1 = imdilate(I, structure);
I1 = imerode(I, structure);
I1 = imopen(I, structure);
I1 = imclose(I, structure);
```

where `I1` is the output image, `I` is the input image, and `structure` is the neighborhood specification given by `strel` as described above. In all cases, `structure` can be replaced by an array specifying the neighborhood as ones, bypassing the `strel` routine. In addition, '`imdilate`' and '`imerode`' have optional arguments that provide packing and unpacking of the binary input or output images.

EXAMPLE 14.5

Apply opening and closing to the thresholded blood cell images of Figure 14.3 in an effort to remove small background artifacts and to fill holes. Use a circular structure with a diameter of four pixels.

Solution

With the use of MATLAB routines, these morphological operations are straightforward. Since opening and closing are performed on BW images, we begin by using thresholding of the original image using the minimum variance method. This is followed by erosion and dilation to perform the opening operation. After the opened and intermediate image (i.e., the eroded image) are displayed, the closing operation is applied to the thresholded image by first dilating, then eroding the image, and the resulting images are displayed. Of course, both operations could have been performed with a single MATLAB command, but then the intermediate images would not have been available for display.

```
% Example 14.5 Demonstration of morphological opening to eliminate
% small artifacts and of morphological closing to fill gaps
%
I = imread('blood1.tif');      % Get image and threshold
I = im2double(I);
BW = ~im2bw(I,graythresh(I)); % Threshold image (Otsu's method)
%
SE = strel('disk',4);          % Define structure
BW1= imerode(BW,SE);          % Opening operation: erode
BW2 = imdilate(BW1,SE);        % then dilate
%
.....display images.....
%
BW3= imdilate(BW,SE);          % Closing operation:
BW4 = imerode(BW3,SE);          % dilate, then erode
.....display images.....
```

Results

This example produced the images shown in Figures 14.18 and 14.19. The next example shows how these morphological operations can be used to improve the segmentation produced by other methods. This example applies opening to the cell images in Figure 14.17 and to one of the textured images in Figure 14.14. The problems also address improving the segmentation of the textured images shown in Figure 14.12.

EXAMPLE 14.6

Apply an opening operation to remove the dark patches seen in the thresholded cell image of Figure 14.17. Also remove *both* the black and white specks from the texture-segmented image of Figure 14.12 (right side).

Solution

The opening operation is used in both figures since it tends to eliminate small objects. The opening operation acts on activated (i.e., white) pixels; so, it is necessary to invert the image using the NOT operation before opening to remove the black artifacts. After the opening operation, the image is inverted again (reinverted) to produce an image similar to the original.

In the textured image of Figure 14.14, both the light and dark specks are to be removed. The latter can be done by performing the opening operation on the inverted image as above, then, after reinverting, performing a second opening operation to eliminate the white specks.

```
% Example 14.6 Use opening to remove the dark patches in the
% thresholded cell image of Figure 14.17 and to
% remove both white and black specks from the
% texture image on the right side of Figure 14.14
%
load Ex14_4_data;           % Get cell image(BW2)
SE = strel('square',5);     % Define structure:
BW1= ~imopen(~BW2,SE);    % Opening/inverted operation
.....Display images, new figure.....
%
load Ex14_2_data;           % Repeat for texture BW image
BW = ~imopen(~BW3,SE);     % Opening/inverted operation
BWA = imopen(BW,SE);       % Opening operation
.....Display images.....
```

Results

Applying the opening operation to the inverted cell image using a 5×5 square structural element results in the elimination of all the dark patches within the cells as seen in Figure 14.20 (right). The size (and shape) of the structural element controls the size of the artifact removed and no attempt is made to optimize its shape. The size is set here as the minimum that would still remove all the dark patches.

As shown in Figure 14.21 (center), the opening operation applied to the texture-segmented image eliminates the dark specks from the white background,* but not the light specks in the isolated feature. A second opening applied to the center image removes most of these white specks (right-hand image). To use that as a segmentation mask to isolate the right-hand feature, the image would be inverted; so, the right-hand feature is white while the rest of the image is black. The original image would then be multiplied by this BW image.

* A dark speck in the corner remains left over from the border artifact. A larger structure element would be required to remove this artifact.

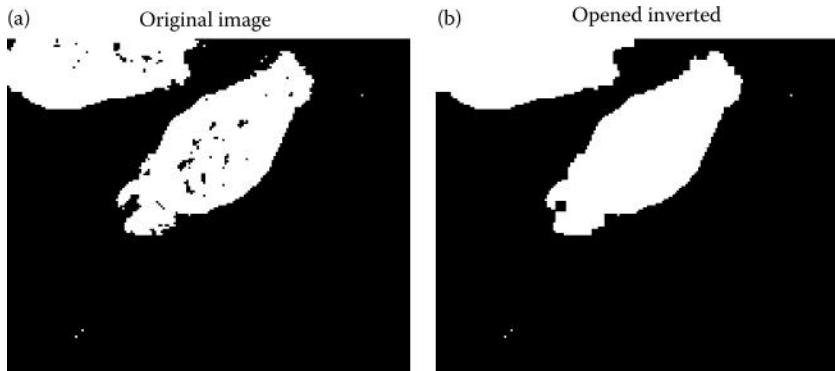


Figure 14.20 The opening operation performed on the thresholded cell image (a) eliminates the dark artifacts from the interior of the cells. (b) A 5×5 rectangular structure is used.

MATLAB morphology routines also allow for manipulation of maxima and minima in an image. This is useful for identifying objects and for filling. Of the many other morphological operations supported by MATLAB, only the `imfill` operation is described here. This operation begins at a designated pixel and changes the connected background pixels (0 s) to foreground pixels (1 s), stopping only when a boundary is reached. For grayscale images, `imfill` brings the intensity levels of the dark areas that are surrounded by lighter areas up to the same intensity level as the surrounding pixels. (In effect, `imfill` removes regional minima that are not connected to the image border.) The initial pixel can be supplied to the routine or can be obtained interactively. Connectivity can be defined as either *four connected* or *eight connected*. In four connectivity, only the four pixels bordering the four edges of the pixel are considered, while in eight connectivity, all pixels that touch the pixel are considered, including those that touch only at the corners.

The basic `imfill` statement is

```
I_out = imfill(I, [r c], con);
```

where I is the input image, I_out is the output image, $[r\ c]$ is a two-element vector specifying the beginning point, and con is an optional argument that is set to 8 for eight connectivity

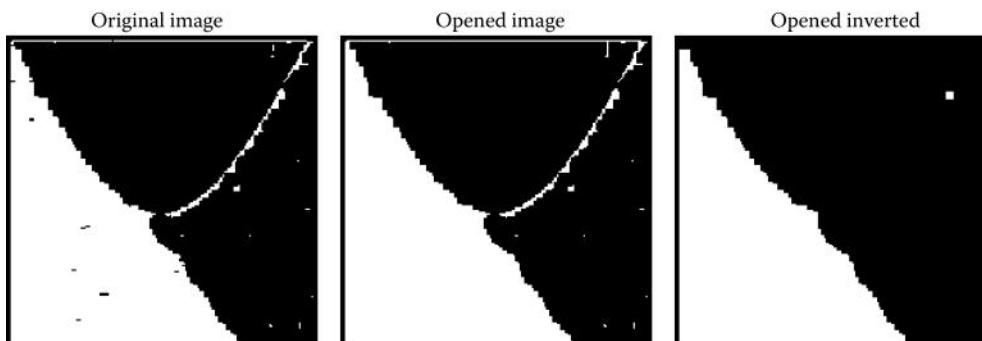


Figure 14.21 Applying opening to an inverted original image then inverting produces the center image which is free of dark speck in the white section of this image; however white artifacts remain in the dark section. Reapplying opening to the center image remove all but one white speck as shown in the right hand image.

(four connectivity is the default). (See the help file to use `imfill` interactively.) A special option of `imfill` is available specifically for filling holes. If the image is binary, a hole is a set of background pixels that cannot be reached by filling in the background from the edge of the image. If the image is an intensity image, a hole is an area of dark pixels surrounded by lighter pixels. To invoke this option, the argument following the input image should be 'holes'. Example 14.7 shows the operation performed on the blood cell image by `imfill` using the 'holes' option. This operation is followed by thresholding and opening to produce a good segmentation of the blood cell images.

EXAMPLE 14.7

Load the image of blood cells (`blood.tif`) and apply `imfill` to darken the center of the cells. Then apply thresholding to produce a mask of the cells followed by opening to remove any artifacts.

```
% Example 14.7 Demonstration of imfill with option 'holes' on a
% grayscale image followed by thresholding and opening to
% produce a segmented image of the blood cells.
%
I = imread('blood1.tif');           % Get image and
I = im2double(I);                  % Convert to double
I1 = imcomplement(I);              % Invert original image
%
I2 = imfill(I1,'holes');
% Threshold and open the filled image
BW = im2bw(I2,.5);
SE = strel('disk',5);    % Define structure:
BW1 = imopen(BW,SE);
```

Results

The image produced by `imfill` is shown in Figure 14.22. This routine is applied to the grayscale image, and portions of the image that are enclosed by the circular cell walls become black. This image is then thresholded at 0.5 (Figure 14.23, left image). Opening is then used to remove the small white artifacts (Figure 14.23, right image), resulting in a well-segmented image of the cells.

The image in Figure 14.23 can be used as a mask to isolate only the cells. This is done simply by multiplying (point by point) the mask image in Figure 14.23 by the original blood cell image. The result is shown in Figure 14.24. This image has been scaled by 2 to enhance the cell interiors.

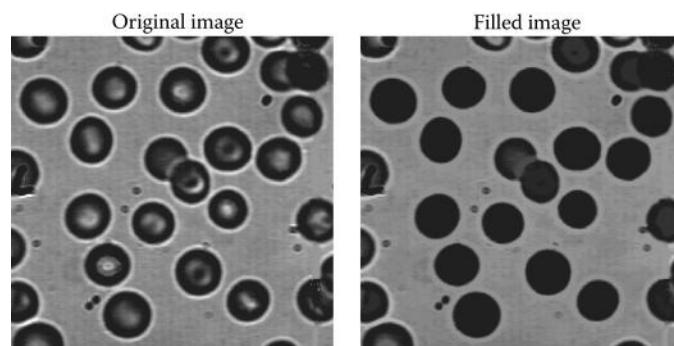


Figure 14.22 Hole-filling operation produced by `imfill`. Note that the edge cells and the overlapped cell in the center are not filled since they are not actually holes.

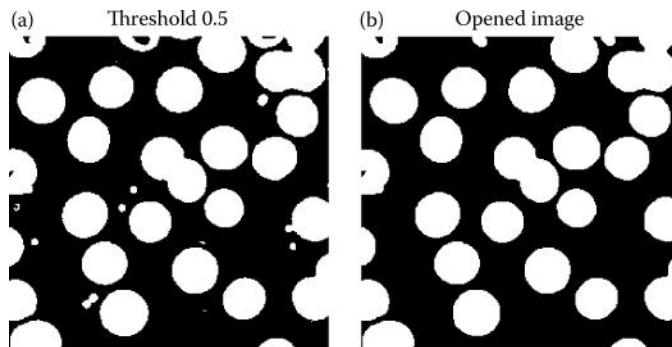


Figure 14.23 Filled image in Figure 14.22 after thresholding (a) and opening (b) to remove the artifacts.

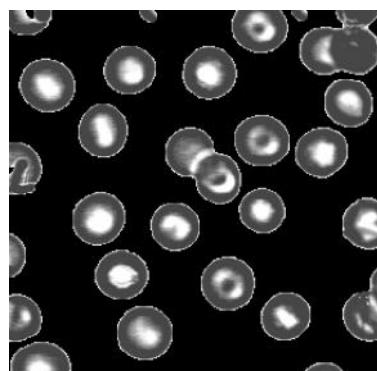


Figure 14.24 Image of the blood cells after the background is removed by masking (i.e., multiplying) the original image using the black-and-white image in Figure 14.23 (right). The image has been scaled (multiplied) by 2.0 to improve the visual appearance of the cell interiors.

14.6 Edge-Based Segmentation

Historically, edge-based methods were the first set of tools developed for segmentation. To move from edges to segments, it is necessary to group edges into chains that correspond to the structural boundaries, that is, the sides of structural units. The approaches vary in how much prior information they use, that is, how much is used of what is known about the possible shape. False edges and missed edges are two of the more obvious and more common problems associated with this approach.

The first step in edge-based methods is to identify edges that then become candidates for boundaries. Some of the filters presented in Chapter 13 perform edge enhancement, including the Sobel, Prewitt, Roberts, Log, and Canny filters. These edge detection filters can be divided into two classes: filters that use the gradient of the first derivative, and filters that use zero crossings of the second derivative. The Sobel, Roberts, and Prewitt filters are all examples of the derivative-based filter and differ in how they estimate the first derivative. The Log (that actually stands for “Laplacian of Gaussian”) filter takes the spatial second derivative and is used in conjunction with zero crossings to detect edges. These filters are used with MATLAB’s edge routine described below that sets thresholds that can be automatically adjusted.

The Canny filter, the most advanced edge detector filter supported by MATLAB’s edge routine, is a type of zero-crossing filter. The approach finds edges by looking for local maxima of the

gradient of the image. The gradient is calculated using the derivative of a Gaussian filter, similar to that used in the Log filter. The method uses two thresholds to detect strong and weak edges and includes the weak edges in the output only if they are connected to strong edges. Therefore, this method is less likely than the others to be fooled by noise and is more likely to detect true weak edges. All these filters produce a binary output that can be a problem if a subsequent operation requires a graded edge image. These filter operations are explored in Example 14.8 and in the problems.

Edge relaxation is one approach used to build chains from edge candidate pixels. This approach takes into account the local neighborhood: weak edges positioned between strong edges are probably part of the edge, while strong edges in isolation are likely to be spurious. The Canny filter incorporates a type of edge relaxation. Various formal schemes have been devised under this category. A useful method, described in Sonka et al. (1993), establishes edges between pixels (the so-called crack edges) based on the pixels located at the end points of the edge.

Another more advanced method for extending edges into chains is termed *graph searching*. In this approach, the endpoints (that can both be the same point in a closed boundary) are specified and the edge is determined based on minimizing some cost function. The possible pathways between the endpoints are selected from candidate pixels, those that exceed some threshold. The actual path is selected using an optimization technique to minimize the cost function. The cost function can include features such as the strength of an edge pixel and total length, curvature, and proximity of the edge to other candidate borders. This approach allows for a great deal of flexibility.

The methods briefly described above use local information to build up the boundaries of the structural elements. The details of these methods can be found in Sonka et al. (1993). MATLAB supports a few of such local operations as described in Section 14.6.2.

Model-based edge detection methods can be used to exploit prior knowledge of the structural unit. These are some of the most powerful segmentation tools and are the subject of much of the current research. If the shape and size of the image are known, then a simple matching approach based on correlation can be used (matched filtering). When the general shape is known, but not the size, a model-based method described in the next section can be used. Another approach is to use the underlying physical properties of the structure of interest to set constraints on the shape of the boundaries. For example, possible deformations of the heart are limited by the mechanical properties of the cardiac tissue. Using a model of a structure's mechanical properties to guide the edge detection task is another area of research interest.

14.6.1 Hough Transform

The *Hough transform* approach was originally designed for identifying straight lines and curves, but can be expanded to other shapes provided the shape can be described analytically. The basic idea behind the Hough transform is to transform the image into a parameter space that analytically defines the desired shape. Maxima in this parameter space then correspond to the presence of the desired image in image space.

For example, if the desired object is a straight line (the original application of the Hough transform), one analytic representation for this shape is $y = mx + b$,^{*} and such shapes can be completely defined by a 2-D parameter space of m and b parameters. All straight lines in image space map to points in parameter space (also known as the *accumulator array* for reasons that will become obvious). Operating on a binary image of edge pixels, all possible lines through a given pixel are transformed into m, b combinations, which then increment the accumulator array. Hence, the accumulator array accumulates the number of *potential* lines that could exist in the image. Any active pixel will give rise to a large number of possible line slopes, m , but only a limited number of m, b combinations. If the image actually contains a line, then the accumulator element that

* This representation of a line will not be able to represent vertical lines since $m \rightarrow \infty$ for a vertical line. However, lines can also be represented in two dimensions using cylindrical coordinates, r and θ : $r = a \sin \theta / \sin(\theta - \varphi)$.

Biosignal and Medical Image Processing

corresponds to that particular line's m, b parameters will have accumulated a large number of "hits," at least relative to all the other accumulator elements. The accumulator array is searched for maxima, or a number of suprathreshold locations, and these locations identify a line, or lines, in the image.

This concept can be generalized to any shape that can be described analytically, although the parameter space (i.e., the accumulator) may have to include several dimensions. For example, to search for circles, a circle can be defined in terms of three parameters, a, b , and r :

$$(y - a)^2 + (x - b)^2 = r^2 \quad (14.1)$$

where a and b define the center point of the circle and r is the radius. Hence, the accumulator space must be 3-D to represent a, b , and r .

14.6.2 MATLAB Implementation

The edge detection filters described above can be implemented using `fspecial` and `imfilter`, but MATLAB offers an easier implementation using the routine `edge`:

```
BW = edge(I, 'method', threshold, options);
```

where `BW` is the binary output image, `I` is the input image, and '`method`' specifies the type of filter. The method can be the name (no caps) of any of the filters mentioned above. The `threshold` argument is optional and if it is not specified (or empty), Outso's method is used to automatically determine the threshold. The Canny method calls for two thresholds, both of which can be determined automatically or can be entered as a vector. There are a few other options and they will be described when used. The behavior of these filters is compared on a single image in the next example.

EXAMPLE 14.8

Load the blood cell image and detect the cell boundaries using the three derivative-based filters, `sobel`, `prewitt`, and `roberts`. Also compare the two zero-crossing filters `log` and `canny`. Select the threshold empirically to be the highest possible and still provide solid boundaries.

```
% Example 14.8 and Figure 14.25 and 14.26
% Apply various edge detection schemes to the blood
% cell image
%
clear all; close all;
.....Load and convert 'blood1.tif'.....
imshow(I);figure;
% Apply the 5 filters to the cell image
[BW1,thresh1] = edge(I,'sobel',.13,'nothinning');
[BW2,thresh2] = edge(I,'roberts',.09,'nothinning');
[BW5,thresh3] = edge(I,'prewitt',.13,'nothinning');
[BW3,thresh4] = edge(I,'log',.004);
[BW4,thresh5] = edge(I,'canny',[.04 .08]);
.....Display and label images.....
```

Results

The original image and the results of the derivative-based filters are shown in Figure 14.25. All these filters used the option '`nothinning`' to make the lines thicker to improve the display. The filter thresholds were adjusted to give approximately the same level of boundary

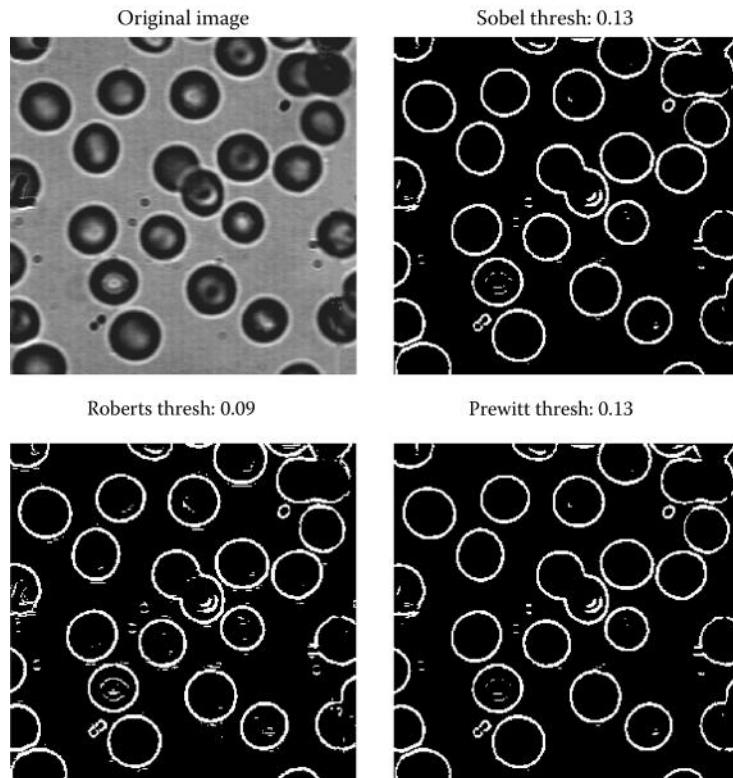


Figure 14.25 Blood cell edges determined from three derivative-based filters. Filter thresholds are indicated.

integrity. The performance of the three filters is comparable, although the Prewitt filter has slightly fewer or smaller artifacts.

The output of the two zero-crossing filters is shown in Figure 14.26. Again, filter thresholds were adjusted to give reasonably intact boundaries with minimal artifacts. These filters show two edges around each cell as they detect both the light-to-dark and dark-to-light transitions. This could be an advantage in other applications, but is not as useful if the goal is to segment the

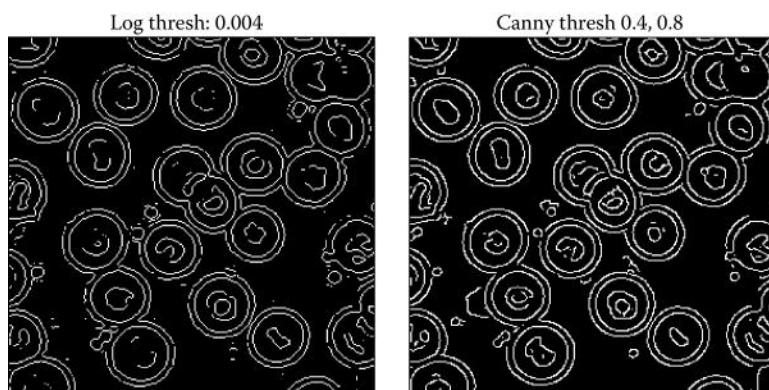


Figure 14.26 Blood cell edges determined by two filters based on zero crossing. The more advanced Canny filter that used two threshold criteria shows more complete boundaries.

Biosignal and Medical Image Processing

blood cells. The more advanced Canny filter shows better edge identification and fewer artifacts than the Laplacian of Gaussian (Log) filter. Other operations on binary images are explored in the problems.

The Hough transform is supported by MATLAB image-processing routines, but only for straight lines. It is supported as the *Radon transform* that computes projections of the image along a straight line, but this projection can be done at any angle.* This transform produces a projection matrix that is the same as the accumulator array for a straight line Hough transform when expressed in cylindrical coordinates.

The Radon transform is implemented by the statement

```
[R, xp] = radon(BW, theta);
```

where *BW* is a binary input image and *theta* is the projection angle in degrees, usually a vector of angles. If not specified, *theta* defaults to (1:179). The output *R* is the projection array where each column is the projection at a specific angle. The second output, *xp*, is a vector that gives the radial value of each row of *R*. Hence, maxima in *R* correspond to the positions (encoded as an angle and distance) in the image. An example of the use of *radon* to perform the Hough transformation is given in Example 14.9.

EXAMPLE 14.9

Find the strongest line in the image of the Saturn in *saturn1.tif*. Plot that line superimposed on the image.

Solution

First convert the image into an edge array using MATLAB's edge routine with the Canny filter. Apply the Hough transform (implemented for straight lines using *radon*) to the edge image to build an accumulator array. Find the maximum point in that array (using *max*), which will give *theta*, the angle perpendicular to the line, and the distance along that perpendicular line of the intersection. Convert that line into rectangular coordinates, then plot the line superimposed on the image. Finding the maximum point in the transform array and converting it into rectangular coordinates are the most difficult aspects of the problem.

```
% Example 14.9 The Hough Transform implemented using radon
% to identify the strongest line in an image.
%
radians = 2*pi/360; % Degrees to radians
I = imread('saturn1.tif'); % Get image of Saturn
BW = edge(I, 'canny', .05); % Threshold image
[R, xp] = radon(BW, theta); % Hough transform
%
.....Display original and thresholded images.....
[~, c] = max(max(R)); % Find maximum element
[~, r] = max(R(:,c));
[ri ci] = size(I); % Size of image array
[ra ca] = size(R); % and accumulator array
% Convert to rectangular coordinates
A = xp(r); % Get line displacement
phi = theta(c); % and angle
m = tan((90-phi)*radians); % Calculate slope and
b = A/cos((90-phi)*radians); % intercept from
```

* The Radon transform is an important concept in CT as described in Chapter 15.

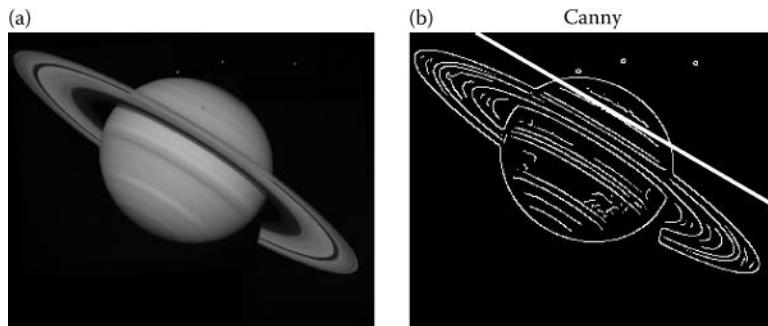


Figure 14.27 Normal (a) and thresholded (b) image of the Saturn with the dominant line found by the Hough transform. (The original image is a public domain image courtesy of NASA, Voyager 2 image, 08-24-1981.)

```

x = (0:ci); % basic trig.
y = mx + b; % Construct line
figure(fig1); subplot(1,2,2); hold on;
plot(x,-y,'w'); % Plot line on Saturn
figure(fig2); hold on;
plot(phi,A,'*b'); % Plot max accumulator point
.....Labels.....

```

Results

This example produces the images shown in Figure 14.27. The broad white line superimposed on the edge image is the line found as the most dominant using the Hough transform. The location of the maximum coordinate is in the accumulator array (i.e., the parameter space) is indicated by a black * in the accumulator array (Figure 14.28). The location of the maximum point must be converted into a degree-and-offset position and then must be converted into rectangular coordinates to plot the line. Other points nearly as strong (i.e., bright) can be seen in the parameter array representing other lines in the image. Of course, it is possible to identify these lines as well by searching for maxima other than the global maximum. This is done in Problem 14.12.

14.7 Summary

Segmentation, the isolation of regions of interest, is often a critical first step in biomedical image analysis. In segmentation problems, the goal is to develop a “mask,” a BW image that is 1.0 over the region of interest and 0.0 elsewhere. When this mask is multiplied by the original image, all features except for the region of interest are set to black (0.0). Sometimes, the task is so difficult that human intervention is required, but new techniques permit increasingly automated segmentation operations. A large number of basic and advanced segmentation operations exist. These approaches broadly fall into four categories: pixel-based, regional or continuity-based, edge-based, and morphological methods.

Pixel-based methods primarily involve separation by grayscale intensity and are the fastest and easiest to implement, but are also the least powerful. They operate on one element at a time and do not consider the larger aspects of the image; therefore, they are particularly susceptible to noise. Finding the best threshold to isolate a region is often aided by a plot of the number of pixels at a given intensity versus intensity, the intensity histogram. MATLAB routines that support threshold operations include an operation that automatically adjusts the threshold to minimize the total variance in intensity level on either side of the threshold boundary (Otsu’s method).

Biosignal and Medical Image Processing

Regional methods identify similarities in the region of interest such as textural properties while edge-based methods search for differences that indicate boundaries. Filters, including nonlinear filters, are a mainstay of regional methods as they act on a group of pixels. Averaging or lowpass operations are the commonly used linear filters while nonlinear operations include the range, Laplacian, Hurst (maximum difference as a function of pixel separation), and Haralick operators.

Edge-based operations search for differences in image features to identify boundaries. These approaches also rely on filtering, but use derivative or derivative-like filters that enhance boundaries. More advanced methods involving multiple thresholds can be used to fill in gaps in the boundary. MATLAB's Canny filter is an example of such an advanced filter. Another advanced method uses an analytical description of the boundary shape and keeps track of all the shapes that meet the analytical criteria. MATLAB supports this procedure for straight line boundaries in the form of the Hough transform, developed for the analysis of CT imaging.

Morphological methods use information on shape to constrain or define the segmented image. Advanced morphological methods may include information on the biomechanics and biodynamics of the tissue of interest. Morphological operations supported by MATLAB include dilation, erosion, opening, and closing. These operations are applied to BW images. In dilation, the rich get richer so that areas surrounded by active pixels become active (i.e., set to 1.0). In erosion, the poor get poorer and active pixels are set to 0.0 if they have few active neighbors. Control of the size and shape of the region considered is a major design issue in the implementation of these operations. The two operations can be applied sequentially and, since they are nonlinear operations, the results are highly dependent on the sequence. In opening, erosion is followed by dilation; during the former, some points disappear altogether and thus are not restored by dilation. Opening is useful for removing small, isolated artifacts often caused by noise. In closing, dilation is followed by erosion: gaps that are filled by dilation remain that way after the subsequent erosion. Closing connects objects that are close to each other, fills small holes, and smooths boundaries.

Many challenging segmentation problems require the application of multiple approaches. In such cases, partial masks are created and combined using the logical AND/OR operations. Segmentation has received considerable attention by biomedical imagers and continues to progress with the development of new algorithms and more powerful, faster, hardware.

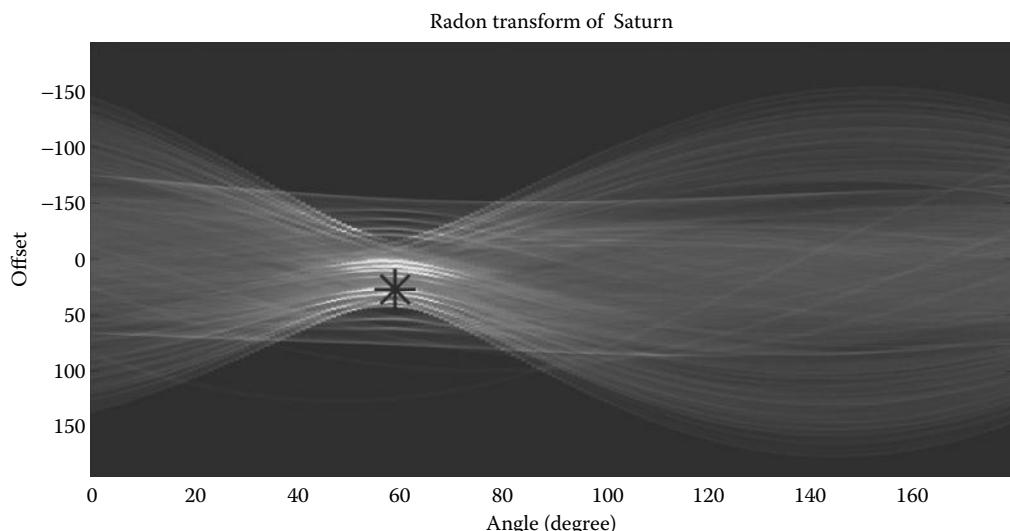


Figure 14.28 Accumulator array found from the Hough transform of the image of the Saturn in Example 14.9. The maximum point in this array provides the location of the strongest line and is indicated by an “*.”

PROBLEMS

- 14.1 Load the bone marrow image in `bonemarr.tif`. Convert the image into a BW mask using `graythresh` to determine the threshold. Then construct a second mask with fewer artifacts by finding a threshold manually. You will find that the artifacts in the image found using `graythresh` can only be improved slightly before the smaller white segments are eliminated. Plot the original image and the two threshold images side by side.
- 14.2 Load the blood cell image in `blood1.tif`. Filter the image with two lowpass filters, both Gaussian 20×20 , one having a low cutoff (an alpha of 0.5) and the other having a high cutoff ($\text{alpha} \geq 4$). Threshold the two filtered images using the minimum variance routine `graythresh`. Display the original and filtered images along with their histograms. Also display the two thresholded images. Note the improvement in artifact elimination with the strong filter. Regarding the histograms, also note the reduction in low-intensity pixels after strong lowpass filtering.
- 14.3 Repeat Problem 14.2 for the original image shown in Figure 14.5 and found in file `Fig14_5.tif`. As in Figure 12.8, there is a substantial improvement in the separation of peaks in the histogram with the stronger filter and a radical improvement in segmentation. The downside is that the border is less precisely defined in the heavily filtered image.
- 14.4 The Laplacian filter that calculates the second derivative can also be used to find edges. In this case, edges will be located where the second derivative is near zero. Load the image of the spine (`spine.tif`) and filter using the Laplacian filter obtained from the `fspecial` routine (use the default constant). Then, threshold this image. You will need to take a fairly low threshold (<0.02) since you are interested in the values near zero. Note the extensive tracing of subtle boundaries that this filter produces.
- 14.5 Load the image `texture3.tif` that contains three regions having the same average intensities but different textural patterns. Before applying the nonlinear range operator used in Example 14.2, preprocess with a Laplacian filter ($\text{alpha} = 0.5$). Apply the range operator as in Example 14.2 using `nlfiltex`. Plot the original and “range” images along with their histograms. Threshold the range image to isolate the segments and compare with the figures in this book. [Hint: You may have to adjust the thresholds slightly, but you do not have to rerun the time-consuming range operator to adjust these thresholds. You can do that from the MATLAB command line.] You should observe a modest improvement over Example 14.2: one of the segments can now be perfectly separated.
- 14.6 Load the texture orientation image `texture4.tif`. Separate the segments by first preprocessing the image with a Sobel filter. Then apply a nonlinear filter using a standard deviation operation determined over a 2×9 pixel grid. Finally, postprocess the output image from the nonlinear filter using a strong Gaussian lowpass filter. (Note: You will have to multiply the output of the nonlinear filter by around 3.5 to get it into an appropriate range.) Plot the output image of the nonlinear filter and that of the lowpass filtered image and the two corresponding histograms. Separate the lowpass filtered image into three segments as in Examples 14.2 and 14.3. Use the histogram of the lowpass filtered image to determine the best boundaries for separating the three segments. Display the three segments as white objects (i.e., as segment masks).
- 14.7 Load the thresholded images of Figure 14.12 found as `BW1`, `BW2`, and `BW3` in file `Ex14_2_data.mat`. Invert the image in `BW1` and apply opening to the inverted

Biosignal and Medical Image Processing

image to eliminate as many points as possible in the upper field without affecting the lower field. Then use closing on the inverted image to try to blacken as many points as possible in the lower field without affecting the upper field. You should be able to whiten the upper field completely with opening and blacken the lower field completely with closing. You should pick the best structural element to accomplish the task. Plot the output of the two operations inverted so that the segmented section is white. [Hint: The spots you are trying to eliminate are small and round. You may need different structural elements for the upper and lower fields.]

- 14.8 As in Problem 14.7, load the thresholded images found as BW1, BW2, and BW3 in file Ex14_2_data.mat. Apply opening to an inverted BW3 to eliminate as many points as possible in the lower field without affecting the lower field. Then eliminate the artifacts in the upper field of the opened image in two different ways. Apply closing to the opened image and opening to an inverted opened image. You may need different structural elements for these different operations, but both approaches should completely eliminate the upper and lower artifacts from the image. Show the original image, the initial opened image, and the two artifact-free images. Invert the images as needed so that the lower diagonal field is white.
- 14.9 As in Problem 14.7, load the images from Figure 14.12 found in file Ex14_2_data.mat. Clear the artifacts in both portions of BW2 in two ways: using opening applied twice (as in Problem 14.8) and closing applied twice. Invert the images as required to eliminate the artifacts and display with the right segment as white. For each approach, display the original and final images along with the intermediate image.
- 14.10 Load the image of the bacteria found in bacteria.tif. The objective of this problem is to segment the bacterial cell using a multistep process. First, apply the Canny edge filter with a primary threshold of 0.2. Then add this edge image to the grayscale image of the bacteria. Next, apply imfill to this combined image that will make the interior of all but one cell white. Finally, threshold this image at a very high level (0.99) to get the bacteria mask. One cell is not captured by this method because its border is not continuous. To capture this cell, erode the thresholded image using a fairly small structure. All the cells should now be captured by this method.
- 14.11 Load the image of the brain in brain1.tif. Convert that into double and use mat2gray to ensure proper intensity scaling. The objective is to segment the ventricles, the darker areas in the center of the brain. This is a multistep process somewhat similar to that used in Problem 14.10. Ultimately, the segmentation will rely on imfill and a careful separation based on a small intensity range; however, an important first step is to eliminate the bright areas around the outer edges of the brain.

To accomplish this, identify those bright regions using edge and a Canny filter with a fairly high threshold so that you detect only the outer edges of the brain. Then increase the size of those edges using dilation and subtract these highlighted images from the main image. (You could also invert the BW edge image and multiply the brain image by this mask.) For dilation, you should use a structural element that is just large enough to remove all the outer features from the subtracted (or masked) image.

If you then apply imfill to this modified image, it will outline the ventricles using a constant intensity level that you can isolate using two BW images generated with carefully selected thresholds. When combined appropriately, the ventricles will be highlighted in white. There will also be additional speckles in this image that can be removed using erosion. The same structural element used in dilation above

- seems to work well. The result will be the segmented ventricles. Plot the final thresholded image (before erosion), the final segmented image, and the original image together. Also plot the dilated edge image, the original image after removing the outer features, and the filled image.
- 14.12 Modify Example 14.9 to plot the *fourth* strongest line in the Saturn image. Plot the original line, then plot the fourth strongest line on the same plot using a dashed line and/or a different color. Also plot the fourth largest point in the accumulator array using a different marker type and/or color. [Hint: Use a loop to set the accumulator entry corresponding to the first three maximum values to zero, then find the maximum value of the modified accumulator array. Also use the “hot” colormap to plot the accumulator array.]

15

Image Acquisition and Reconstruction

15.1 Imaging Modalities

Medical imaging utilizes several different physical principles or imaging *modalities*. The common modalities used clinically include x-ray, *computed tomography* (CT), *positron emission tomography* (PET), *single photon emission computed tomography* (SPECT), *magnetic resonance imaging* (MRI), and ultrasound. Other approaches under study, or development, include optical imaging* and impedance tomography. Except for simple x-ray images, which provide a shadow of intervening structures, some type of image processing is required to produce a useful image. The algorithms used for image reconstruction depend on the modality. In MRI, reconstruction techniques involve a 2-D inverse Fourier transform as described later in this chapter. PET and CT use *projections* from *collimated beams* and the reconstruction algorithm is critical. The quality of the image can be strongly dependent on the image reconstruction algorithm.[†]

15.2 CT, PET, and SPECT

Reconstructed images from PET, SPECT, and CT all use collimated beams directed through the target, but they vary in the mechanism used to produce these beams. CT is based on x-ray beams produced by an external source that are collimated by the detector: the detector includes a *collimator*, usually a long thin tube that absorbs diagonal or off-axis photons. In SPECT, the collimated photons are produced by the decay of a radioactive isotope within the patient. Because of the nature of the source, the beams are not well collimated in SPECT and this leads to an unavoidable reduction in image resolution. Although PET is also based on photons emitted from a radioactive isotope, the underlying physics provides an opportunity to improve beam collimation through

* Of course, optical imaging is used in microscopy, but because of scattering, optical imaging is limited to the transparent tissue (as in the eye), superficial tissue, or thin slices of tissue. A number of advanced image-processing methods are under development to reduce the problems due to scattering and to provide useful images of the deeper tissue using either coherent or noncoherent light.

[†] CT may be the first instance where the analysis software is an essential component of medical diagnosis and comes between the physician and patient: the physician has no recourse but to trust the software.

Biosignal and Medical Image Processing

the so-called *electronic collimation*. PET isotopes emit positrons that are short lived because after traveling only a short distance, they interact with an electron. During this interaction, the positron and electron masses annihilate and two photons are generated traveling in opposite directions: 180° from one another. Photon detectors are arranged in a ring around the patient and if two detectors are activated at essentially the same time, then it is likely that a positron annihilation occurred somewhere along a line connecting these two detectors. This *coincident detection* provides an electronic mechanism for establishing a path that intersects with the original positron emission. Note that this establishes the path of the annihilation, not the location of the original positron emission. Since the original positron does not decay immediately and may travel several centimeters in any direction before annihilation, there is an inherent limitation on resolution.

In all three modalities, the basic data consist of measurements of the absorption of x-rays (CT) or concentrations of radioactive material (PET and SPECT) along a known or computed beam path. From this basic information, the reconstruction algorithm must generate an image of either the tissue absorption characteristics or isotope concentrations. The mathematics is fairly similar for both absorption and emission processes and is described here in terms of absorption processes, that is, CT (see Kak and Slaney, 1988 for a mathematical description of emission processes.)

15.2.1 Radon Transform

In CT, the intensity of an x-ray beam is dependent on the intensity of the source, I_o , the absorption coefficient, μ , and length, ℓ , of the intervening tissue:

$$I(x, y) = I_o e^{-\mu \ell} \quad (15.1)$$

where $I(x, y)$ is the beam intensity (proportional to the number of photons) at position x, y . If the beam passes through tissue components having different absorption coefficients, then, assuming the tissue is divided into equal segments, $\Delta\ell$, Equation 15.1 becomes

$$I(x, y) = I_o \exp\left(-\sum_i \mu(x, y) \Delta\ell\right) \quad (15.2)$$

The projection $p(x, y)$ is the natural log of this intensity after normalizing the initial intensity, I_o , and inverting to remove the negative sign in the exponential:

$$p(x, y) = \ln\left[\frac{I_o}{I(x, y)}\right] = \sum_i \mu(x, y) \Delta\ell \quad (15.3)$$

Equation 15.3 can also be expressed as a continuous equation where it becomes the line integral of the attenuation coefficients from the source to the detector:

$$p(x, y) = \int_{\text{Source}}^{\text{Detector}} \mu(x, y) d\ell \quad (15.4)$$

Figure 15.1a shows a series of collimated parallel beams traveling through the tissue.* All these beams are at the same angle, θ , with respect to the reference axis. The output of each beam is just the projection of absorption characteristics of the intervening tissue as defined in Equation 15.4. The projections of all the individual parallel beams constitute a *projection profile* of the intervening tissue absorption coefficients.

* In modern CT scanners, the beams are not parallel, but are dispersed in a spreading pattern from a single source to an array of detectors, the so-called *fan beam* pattern. To simplify the analysis presented here, we assume parallel beam geometry. Kak and Slaney (1988) also cover the derivation of reconstruction algorithms for fan beam geometry.

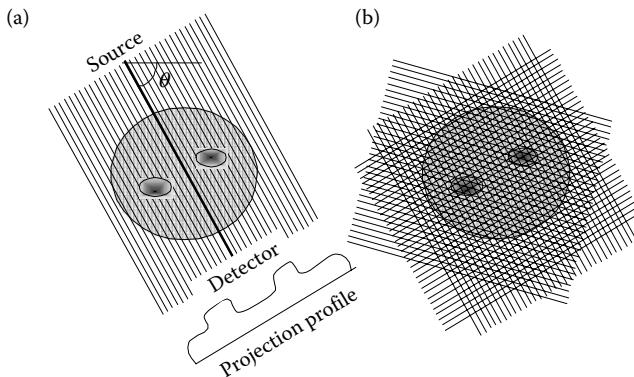


Figure 15.1 (a) A series of parallel beam paths at a given angle, θ , is projected through the biological tissue. The net absorption of each beam can be plotted as a projection profile. (b) A large number of such parallel paths, each and at different angles, is required to obtain enough information to reconstruct the image.

With only one projection profile, it is not possible to determine how the tissue absorptions are distributed along the paths. However, if a large number of projections are taken at different angles through the tissue (Figure 15.1b), it ought to be possible, at least theoretically, to determine the distribution of absorption coefficients by appropriately analyzing the projections. This analysis is the challenge given to the CT reconstruction algorithm.

If the problem was reversed, that is, if the distribution of tissue absorption coefficients was known, determining the projection profile produced by a set of parallel beams would be straightforward. As stated in Equation 15.4, the output of each beam is the line integral over the beam path through the tissue. If the beam is at an angle, θ , as shown in Figure 15.2, then the equation for a line passing through the origin at angle θ is

$$x \cos \theta + y \sin \theta = 0 \quad (15.5)$$

And the projection, p_θ , for that single line at a fixed angle, θ , becomes

$$p_\theta = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(x, y)(x \cos \theta + y \sin \theta) dx dy \quad (15.6)$$

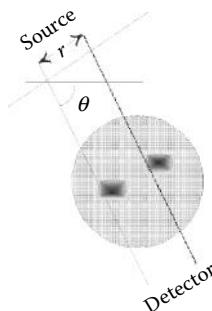


Figure 15.2 A single beam path passing through the origin at an angle θ is defined mathematically by Equation 15.5. If the beam is displaced at a distance r from the origin, then it is defined by Equation 15.7.

Biosignal and Medical Image Processing

where $I(x,y)$ is the distribution of absorption coefficients in Equation 15.2. If the beam is displaced at a distance, r , from the axis in a direction perpendicular to θ (Figure 15.2), the equation for that path is

$$x \cos \theta + y \sin \theta - r = 0 \quad (15.7)$$

The whole family of parallel paths can be mathematically defined using Equations 15.6 and 15.7 combined with the *Dirac delta distribution*, δ to represent the discrete set of parallel beams. The equation describing the entire projection profile, $p_\theta(r)$, becomes

$$p_{\theta}(r) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(x,y) \delta(x \cos \theta + y \sin \theta - r) dx dy \quad (15.8)$$

This equation is known as the *Radon transform*, \mathfrak{R} . It is the same as the Hough transform (Chapter 14) for the case of straight lines. The expression for $p_\theta(r)$ can be written succinctly as

$$p_\theta(r) = \mathfrak{R}[I(x,y)] \quad (15.9)$$

The forward Radon transform can be used to generate raw CT data from image data, useful in problems, examples, and simulations. This is the approach that is used in some of the examples given in Section 15.2.4 and also to generate the CT data used in the problems.

The Radon transform is helpful in understanding the problem, but does not help in the actual reconstruction. Reconstructing the image from the projection profiles is a classic *inverse problem*. You know what comes out, the projection profiles, but you want to know the image that produced these profiles. From the definition of the Radon transform in Equation 15.9, the image should result from the application of an inverse Radon transform, \mathfrak{R}^{-1} , to the projection profiles, $p_\theta(r)$:

$$I(x,y) = \mathfrak{R}^{-1}[p_\theta(r)] \quad (15.10)$$

While the Radon transform (Equations 15.8 and 15.9) and inverse Radon transform (Equation 15.10) are expressed in terms of continuous variables, in imaging systems, the absorption coefficients are given in terms of discrete pixels, $I(n,m)$, and the integrals in the above equations become summations. In the discrete situation, the absorption of each pixel is an unknown and each beam path provides a single projection ratio that can be used as the solution to a multivariable equation. If the image contains $N \times M$ pixels and there are $N \times M$ different projections (i.e., beam paths) available, then the system is adequately determined and the reconstruction problem is simply a matter of solving a large number of simultaneous equations. Unfortunately, the number of simultaneous equations that must be solved is generally so large that a direct solution becomes unworkable.

Early attempts at CT reconstruction used an iterative approach called the *algebraic reconstruction algorithm* (ART). In this algorithm, each pixel is updated based on errors between projections that are obtained from the current pixel values (as determined from the easily solvable forward Radon transform) and the actual projections. When many pixels are involved, convergence is slow and the algorithm becomes computationally intensive and time consuming. The current approaches can be classified as either transform methods or series expansion methods. The *filtered back projection* method described below falls into the first category and is one of the most popular of CT reconstruction approaches.

15.2.2 Filtered Back Projection

Filtered back projection can be described in either the spatial or spatial frequency domain. While often implemented in the latter, the former is more intuitive. In back projection, each pixel absorption coefficient is set to the sum (or average) of the values of all projections that traverse the pixel.

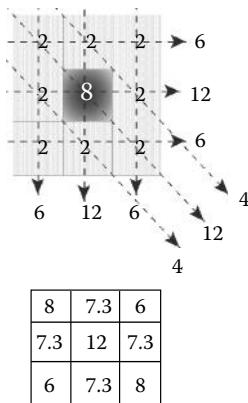


Figure 15.3 Example of back projection on a simple 3×3 pixel grid. The upper grid represents the original image that contains a dark (absorption 8) center pixel surrounded by lighter (absorption 2) pixels. The projections are taken as the linear addition of all intervening pixels. In the lower reconstructed image, each pixel is set to the average of all projections that cross that pixel. (Normally, the sum would be taken over a much larger set of pixels.) The center pixel is still higher in absorption, but the background no longer has the same values. This represents a smearing of the original image.

In other words, each projection that traverses a pixel contributes its full value to the pixel and the contributions from all the beam paths that traverse that pixel are simply added or averaged. Figure 15.3 shows a simple 3×3 pixel grid with a highly absorbing center pixel (absorption coefficient of 8) against a background of lesser absorbing pixels. Three projection profiles are shown traversing the grid horizontally, vertically, and diagonally. The lower grid shows the image that would be reconstructed using back projection alone. Each grid contains the average of the projections through that pixel. This reconstructed image resembles the original with a large central value surrounded by smaller values, but the background is no longer constant. This background variation is the result of blurring or smearing the central image over the background.

To correct the blurring or smoothing associated with the back projection method, a spatial filter can be used. Since the distortion is in the form of a blurring or smoothing, spatial differentiation might improve the image. The most common filter is a pure derivative up to some maximum spatial frequency. In the frequency domain, this filter, termed the *Ram-Lak* filter, is a ramp up to some maximum cutoff frequency. As with all derivative filters, high-frequency noise will be increased; so, this filter is often modified by the addition of a lowpass filter. Lowpass filters that can be used include the Hamming window, the Hanning window, a cosine window, and a sinc function window (the *Shepp-Logan* filter).

An example of a filtered and unfiltered back projection image is shown in Figure 15.4. The original image is a simple light square surrounded by a dark background. The projection profiles produced by the image are also shown (upper right). The back projection reconstruction of this image (lower left) shows a blurred version of the basic square shape with indistinct borders. The application of a highpass filter sharpens the image (Figure 15.4, lower right). The MATLAB implementation of the inverse Radon transform is `iradon` and is described in the next section. This routine uses the filtered back projection method and also provides for all the filter options mentioned above.

Filtered back projection is easiest to implement in the frequency domain. The *Fourier slice theorem* states that the 1-D Fourier transform of a projection profile forms a single, radial line in the 2-D Fourier transform of the image. This radial line will have the same angle in the spatial frequency domain as the projection angle (Figure 15.5). Once the 2-D Fourier transform space is filled from the individual 1-D Fourier transforms of the projection profiles, the image can

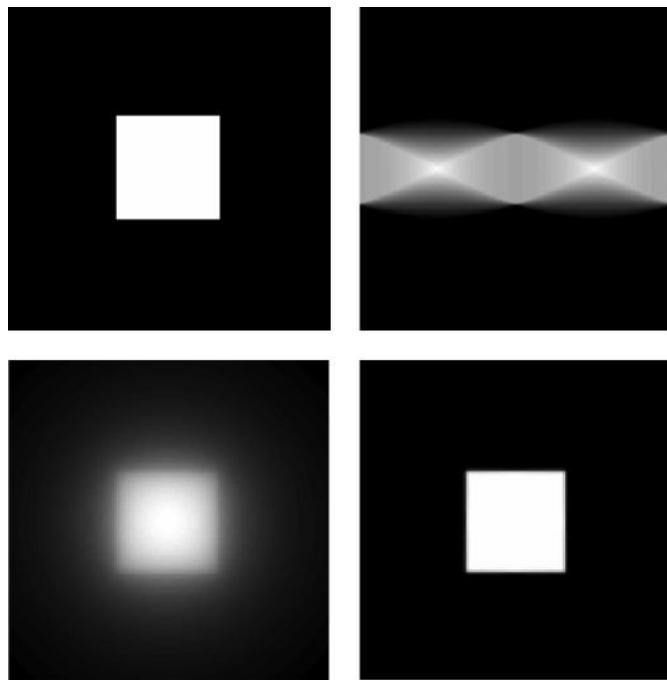


Figure 15.4 Image reconstruction of a white square against a black background. Back projection alone produces a smeared image (lower left) that can be corrected with a spatial derivative filter as shown in the lower right image. These images were generated using the code given in Example 15.1. The upper left image is the Radon transform of this original image.

be constructed by applying the inverse 2-D Fourier transform to this space. Before the inverse transform is done, the appropriate filter can be applied directly in the frequency domain using multiplication.

As with other images, reconstructed CT images can suffer from aliasing if they are undersampled. Undersampling can be the result of an insufficient number of parallel beams in the

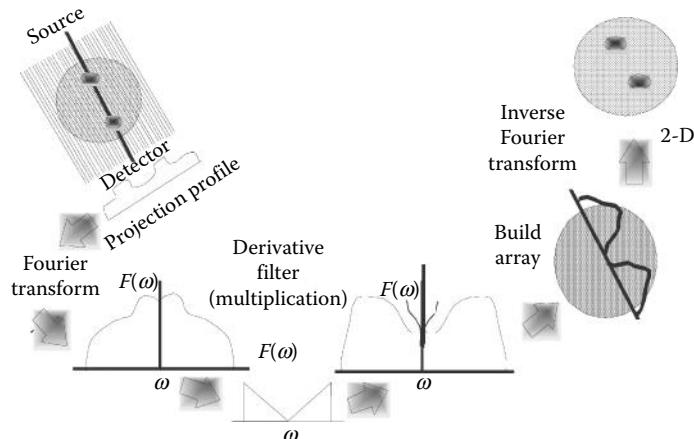


Figure 15.5 Schematic representation of the steps in filtered back projection using frequency-domain techniques. The steps prior to the inverse Fourier transform are for a single projection profile and would be repeated for each projection angle.

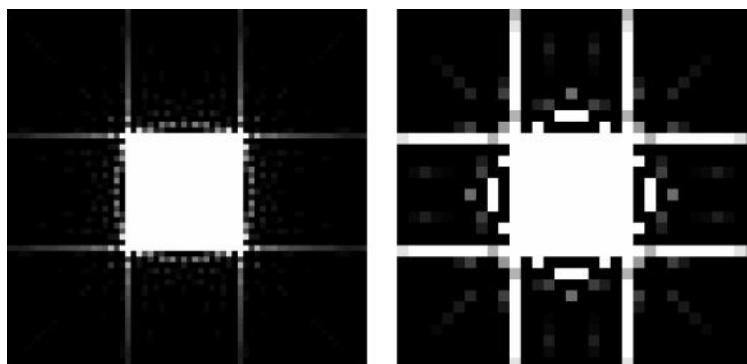


Figure 15.6 Image reconstructions of the same simple pattern shown in Figure 15.4, but undersampled by a factor of 2 (left image) or 4 (right image). The contrast has been increased by a factor of 10 to enhance the relatively low-intensity aliasing patterns.

projection profile or too few rotation angles. The former is explored in Figure 15.6, which shows the square pattern of Figure 15.4 sampled with one-half (left-hand image) and one-quarter (right-hand image) the number of parallel beams used in Figure 15.4. The images have been multiplied by a factor of 10 to enhance the faint aliasing artifacts. One of the problems at the end of this chapter explores the influence of undersampling by reducing the number of angular rotations as well as reducing the number of parallel beams.

15.2.3 Fan Beam Geometry

For practical reasons, modern CT scanners use fan beam geometry. This geometry usually involves a single source and a ring of detectors. The source rotates around the patient while the multiple detectors in the beam path acquire the data. This allows very-high-speed image acquisition, less than half a second. The source fan beam is shaped so that the beam hits a number of detections simultaneously (Figure 15.7). MATLAB provides several routines that provide the Radon and inverse Radon transform for fan beam geometry.

15.2.4 MATLAB Implementation of the Forward and Inverse Radon Transforms: Parallel Beam Geometry

The MATLAB Image Processing Toolbox contains routines that perform both the Radon and inverse Radon transforms in both parallel and fan beam configurations. The Radon transform routine has already been introduced as an implementation of the Hough transform for straight-line objects. The procedure here is essentially the same, except that an intensity image is used as the input instead of the binary image used in the Hough transform.

```
[p, xp] = radon(I, theta); % Forward Radon Transform
```

where I is the image of interest and θ is the projection angle in degrees, usually a vector of angles. If not specified, θ defaults to 1:179. The output parameter p is the projection array, where each column is the projection profile at a specific angle. The optional output parameter, xp , gives the radial coordinates for each row of p and can be used in displaying the projection data.

MATLAB's inverse Radon transform is based on filtered back projection and uses the frequency-domain approach illustrated in Figure 15.5. Several filtering options are available and are implemented directly in the frequency domain.

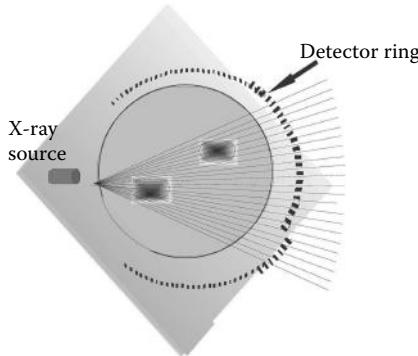


Figure 15.7 A series of beams is projected from a single source in a fan-like pattern. The beams fall upon a number of detectors arranged in a ring around the patient. Fan beams typically range between 30° and 60°. In the most recent CT scanners (the so-called “fourth-generation” machines), the detectors completely encircle the patient and the source can rotate continuously.

The calling structure of the inverse Radon transform is

```
[I,f] = iradon(p,theta,interp,filter,d,n); % Inverse Radon Transform
```

where *p* is the only required input argument and is a matrix where each column contains one projection profile. The angle of the projection profiles is specified by *theta* in one of two ways: if *theta* is a scalar, it specifies the angular spacing (in degrees) between projection profiles (with an assumed range of 0 to the number of columns -1); if *theta* is a vector, it specifies the angles themselves, which must be evenly spaced. The default *theta* is 180° divided by the number of columns. During reconstruction, *iradon* assumes that the center of rotation is half the number of rows (i.e., the midpoint of the projection profile determined using *ceil(size(p,1)/2)*).

Although there are a number of optional parameters, their operations are straightforward. The optional argument *interp* is a string specifying the back projection interpolation method: 'nearest', 'linear' (the default), and 'spline'. The *filter* option is also specified as a string. The 'Ram-Lak' option is the default and consists of a ramp in frequency (i.e., an ideal derivative) up to some maximum frequency. Since this filter is prone to high-frequency noise, other options multiply the ramp function by a lowpass filter. These lowpass filters are the same as described above: the Hamming window ('Hamming'), Hanning window ('Hann'), cosine ('cosine'), and sinc ('Shepp-Logan') function. The filter's frequency characteristics can be modified by the optional parameter, *d*, which scales the frequency axis: if *d* is <1 (the default value is 1), then the filter spectral values are compressed to range between 0 and *d* in normalized frequency. Values above *d* are set to 0. Decreasing *d* decreases the effective lowpass cutoff frequency of the various filters. The optional input argument, *n*, can be used to rescale the image. In addition, it is possible to eliminate all filtering, including the derivative, and get the image only due to back projection by using the filter option 'none'. These filter options are explored in several of the problems.

The image is contained in the output matrix, *I* (class double), and the optional output vector, *h*, contains the filter's frequency response. (This output vector is used in a problem to generate the frequency spectrum of several filters.) An application of the inverse Radon transform is given in Example 15.1.

EXAMPLE 15.1

Example of the use of back projection and filtered back projection. Generate a simple 128 × 128 pixel image of a white square against a black background. The white square should be centered

and should be 20×20 pixels. Construct the CT projections using the forward Radon transform. The original image will be reconstructed from these projections using both the filtered and unfiltered back projection algorithm.

Solution

The black background is generated by initializing an appropriately sized matrix to 0.0. The 20 central pixels are set to 1.0 to generate the white square. MATLAB's `radon` routine is used to generate an array of projection profiles from the image. Projection angles range from 1° to 180° .

The MATLAB `iradon` routine is used with the default to get the filtered image and with the option 'none' to get the unfiltered back projection. When used with this option, the resulting image needs to be rescaled using `mat2gray`. The original image, the projections, and the two reconstructed images are displayed in Figure 15.4.

```
% Example 15.1.
% Image Reconstruction using filtered and unfiltered back projection.
%
I=zeros(128,128); % Construct image: black
I(44:84,44:84)=1; % with a central white square
%
% Generate the projections using 'radon'
theta=(0:179); % Projection angles
[P,xp]=radon(I, theta); % Radon transform to get projections
%
% Reconstruct the images
I_back=iradon(p,delta_theta,'none'); % No filter
I_back=mat2gray(I_back); % Required when no filter
I_filter_back=iradon(p,delta_theta); % Filtered
. .... Display images. ....
```

Results

The images generated by this code are given in Figure 15.4. The blur produced by unfiltered back projection is evident as is the sharpening produced by the default filter (Ram–Lak). Example 15.2 explores the effect of filtering on the reconstructed images.

EXAMPLE 15.2

The inverse Radon transform. Generate CT data by applying the Radon transform to an MRI image of the brain (an unusual example of mixed modalities!). Reconstruct the image using the inverse Radon transform with only the Ram–Lak filter (the default) and with the additional cosine filter with or without reducing the filter's frequency spectrum by half. Display the original and reconstructed images.

Solution

Read in the image and convert that into double format using the approach shown in Example 12.4. Use `radon` with projection angles from 1° to 180° to generate an array of projection profiles. To recreate the images, use `iradon` with no additional filtering (the default) and with the cosine filter option with the frequency-scaling parameter, `d`, set to the default (1.0) and to 0.5. Display the images using `subplot` and `imshow`.

Biosignal and Medical Image Processing

```
% Ex. 15.2 Image Reconstruction using filtered back projection
%
% Initial code is the same as in Example 12.5
frame=18; % Use MR image slice 18
[I(:,:,:,1), map] = imread('mri.tif', frame);
if isempty(map) == 0 % Check to see if Indexed data
    I = ind2gray(I, map); % If so, convert to Intensity image
end
I = im2double(I); % Convert to double and scale
%
% Construct projections of MR image
delta_theta = (0:179);
[P, xp] = radon(I, delta_theta) % Construct projections
%
% Reconstruct image
I1 = iradon(p, delta_theta); % No additional filter
I2 = iradon(p, delta_theta, 'cosine'); % Standard Cosine filter
I3 = iradon(p, delta_theta, 'cosine', 0.5); % Stronger cosine filter
%
..... Display images using subplot and imshow, titles.....
```

Results

Figure 15.8 shows the original and two restricted images. Neither of the reconstructions is as sharp as the original but the reconstruction with no additional filter is the sharpest. The reconstruction with the cosine filter with the reduced spectral range provides the most lowpass filtering and is the least sharp. While the additional filtering is detrimental in this case, there could be noisy images that would benefit.

15.2.5 MATLAB Implementation of the Forward and Inverse Radon Transforms: Fan Beam Geometry

The MATLAB routines for performing the Radon and inverse Radon transform using fan beam geometry are termed `fanbeam` and `ifanbeam`, respectively, and have the form

```
fan = fanbeam(I, D)
```

where `I` is the input image and `D` is a scalar that specifies the distance between the beam vertex and the center of rotation of the beams. The output, `fan`, is a matrix containing the fan beam projection profiles, where each column contains the sensor samples at one rotation angle. It is assumed that the sensors have a 1.0° spacing and the rotation angles are spaced equally from 0° to 359°. A number of optional input variables specify different geometries, sensor spacing, and rotation increments.

The inverse Radon transform for fan beam projections is specified as

```
I = ifanbeam(fan, D)
```

where `fan` is the matrix of projections and `D` is the distance between beam vertex and the center of rotation. The output, `I`, is the reconstructed image. Again, there are a number of optional input arguments specifying the same type of information as in `fanbeam`. This routine first converts the fan beam geometry into a parallel geometry, then applies filtered back projection as in `iradon`. During the filtered back projection stage, it is possible to specify filter options as in `iradon`. To specify, the string '`Filter`' should precede the filter name ('Hamming','Hann','cosine', etc.).

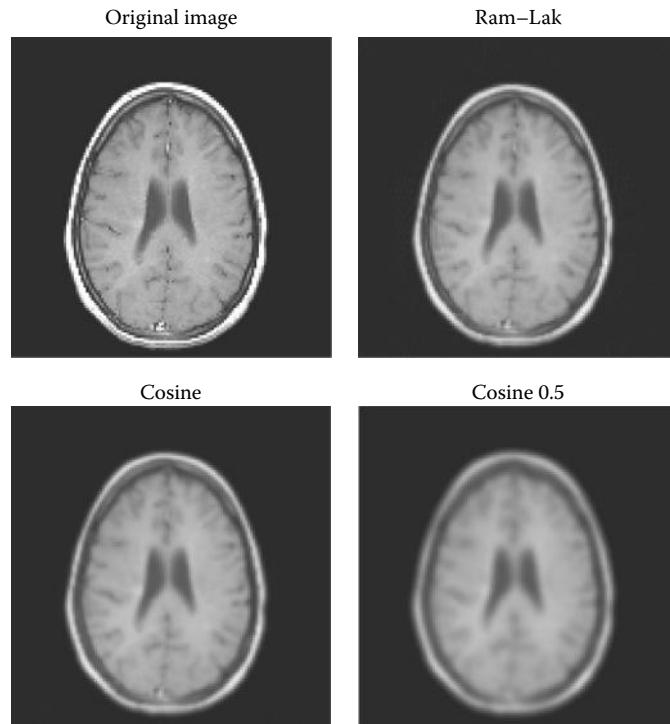


Figure 15.8 Original image (left) and three reconstructions using `iradon`. The two lower images were reconstructed using the cosine filter, but the reconstruction on the right has the filter's frequency spectrum reduced by 0.5 causing it to act more like a lowpass filter. While not advantageous for this image, reducing the filter spectrum could be helpful with a noisy image. (Original image is from the MATLAB Image Processing Toolbox. The Math Works, Inc. Used with permission.)

EXAMPLE 15.3

Fan beam geometry. Apply the fan beam and parallel beam Radon transform to a 128×128 pixel image of four squares of varying grayscale levels (0.25, 0.5, 0.75, and 1.0). Reconstruct the image using the inverse Radon transform with the Shepp–Logan filter for both geometries.

Solution

Generate the image using a black (0.0) 128×128 matrix and set appropriate pixels to the intensity values requested. Construct the projection profiles for the two geometries using `radon` and `fanbeam`. Reconstruct the images using `iradon` and `ifanbeam` with the Shepp–Logan filter option. For the fan beam geometry, use a `FanSensorSpacing` of 0.5 for both the Radon and inverse Radon transform.

```
% Example 15.3 Reconstruction example.
%
D=150; % Beam vertex to center of rotation
theta=(0:179); % Parallel projection angles
%
I=zeros(128,128); % Generate image
I(22:54,22:52)=.25; % Four squares of different
I(76:106,22:52)=.5; % shades against a black
I(22:52,76:106)=.75; % background
```

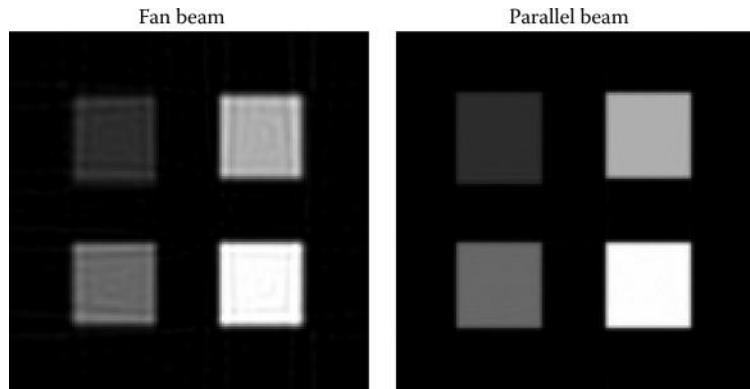


Figure 15.9 Reconstruction of an image of four squares having different intensities using parallel beam and fan beam geometry. Some artifact is seen in the fan beam geometry due to the closeness between the beam source and object (see Problems).

```
I(76:106,76:106) = 1;
%
% Construct projections
[F,Floc,Fangles] = fanbeam(I,D,'FanSensorSpacing',.5);
[R,xp] = radon(I,theta);
%
% Reconstruct images. Use Shepp-Logan filter
I_rfb = ifanbeam(F,D,'FanSensorSpacing',.5,'Filter','Shepp-Logan');
I_rpl = iradon(R,theta,'Shepp-Logan');
..... Display and title images.....
```

Results

The reconstructed images generated by this example are shown in Figure 15.9. For the fan beam geometry, there are small artifacts due to the distance between the beam source and the center of rotation. The influence of this distance on image quality is explored in Problem 15.7.

15.3 Magnetic Resonance Imaging

MRI images can be acquired in a number of ways using different image acquisition protocols. One of the most common protocols, the *spin echo pulse sequence*, is described with the understanding that a number of alternatives are commonly used. In this sequence, the image is constructed on a slice-by-slice basis, although the data are obtained on a line-by-line basis. For each slice, the raw MRI data encode the image as a variation in signal frequency in 1-D and as a signal phase in the other. To reconstruct the image only requires the application of a 2-D inverse Fourier transform to this frequency/phase-encoded data. If desired, spatial filtering can be implemented in the frequency domain before applying the inverse Fourier transform.

The physics underlying MRI is involved and requires quantum mechanics for a complete description. However, most descriptions are approximations that use classical mechanics. The description provided here is even more abbreviated than most. (For a detailed classical description of the MRI physics, see Wright's chapter in Enderle et al. 2000.) Nuclear magnetism occurs in nuclei with an odd number of nucleons (protons and/or neutrons). In the presence of a magnetic field, such nuclei process a magnetic dipole due to a quantum mechanical property

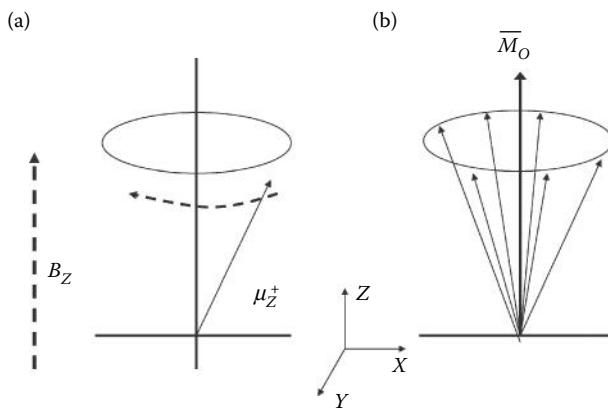


Figure 15.10 (a) A single proton has a magnetic moment that rotates in the presence of an applied magnetic field, B_z . This dipole moment could be up or down with a slight favoritism toward up, as shown. (b) A group of upward dipoles create a net moment in the same direction as the magnetic field, but any horizontal moments (x or y) tend to cancel. All these dipole vectors should be rotating, but for obvious reasons, they are shown as stationary: you must imagine that they rotate or, more rigorously, that the coordinate system is rotating.

known as spin.* In MRI lingo, the nucleus and/or the associated magnetic dipoles are termed *spins*. For clinical imaging, the hydrogen proton is used because it occurs in large numbers in biological tissue. There are a large number of hydrogen protons, or spins, in the biological tissue (1 mm³ of water contains 6.7×10^{19} protons). Unfortunately, even if they are all aligned, the net magnetic moment that can be produced is small due to the near balance between spin-up (1/2) and spin-down (-1/2) states. When they are placed in a magnetic field, the magnetic dipoles are not static, but rotate around the axis of the applied magnetic field such as spinning tops (Figure 15.10a). (Hence, the spins themselves spin!) A group of these spins produces a net moment (M_0 in Figure 15.10b) in the direction of the magnetic field, z , but since they are not in phase, any horizontal moment in the x - or y -direction tends to cancel.

While the various spins do not have the same relative phase, they do all rotate at the same frequency, a frequency given by the *Larmor equation*:

$$\omega_o = \gamma H \quad (15.11)$$

where ω_o is the frequency in radians, H is the magnitude of the magnetic field, and γ is a constant termed the *gyromagnetic constant*. Although γ is primarily a function of the type of nucleus, it also depends slightly on the local chemical environment. As shown below, this equation contains the key to spatial localization in MRI: variations in a local magnetic field encode as variations in rotational frequency of the protons.

If these rotating spins are exposed to electromagnetic energy at the rotational or *Larmor frequency* specified in Equation 15.11, they absorb this energy and rotate further and further from their equilibrium position near the z -axis: they are “tipped” away from the z -axis (Figure 15.11a). They will also be synchronized by this energy, so that they now have a net horizontal moment. For protons, the Larmor frequency is in the radio-frequency (rf) range; so, an rf pulse of the appropriate frequency in the xy -plane will tip the spins away from the z -axis, an amount that depends on the length of the pulse:

* Nuclear spin is not really a spin, but one of those mysterious quantum mechanical properties. Nuclear spin can take on values of $\pm 1/2$, with $+1/2$ slightly favored in a magnetic field.

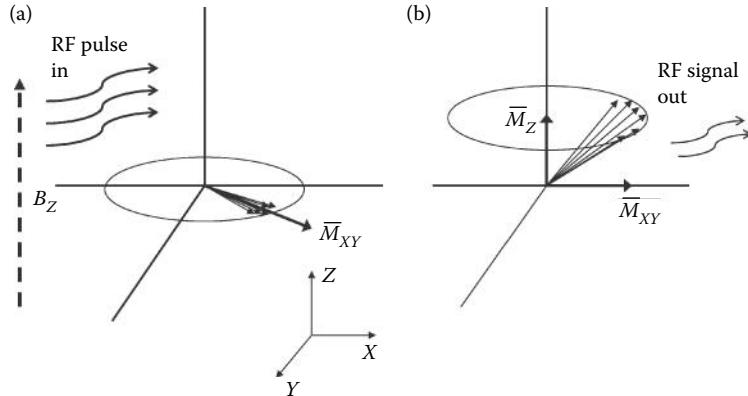


Figure 15.11 (a) After an external rf pulse tips the spins 90° , the net magnetic moment looks like a vector, \bar{M}_{xy} , rotating in the xy -plane. The net vector in the z -direction is zero. (b) After the rf energy is removed, all the spins begin to relax back to their equilibrium position, increasing the z -component, \bar{M}_z , and decreasing the xy -component, \bar{M}_{xy} . The xy -component decreases even more rapidly as the spins desynchronize and begin to partially cancel.

$$\theta = \gamma HT_p \quad (15.12)$$

where θ is the *tip angle* and T_p is the pulse time. Usually, T_p is adjusted to tip the angle either at 90° or 180° . As described subsequently, a 90° tip is used to generate the strongest possible signal and a 180° tip, which reverses the sign of the moment, is used to generate an *echo* signal as described later. Only those spins that are exposed to the combination of local magnetic field strength, H , and rf frequency, ω , given in Equation 15.11 will flip.

When all the spins in a region are tipped at 90° and are synchronized, there will be a net magnetic moment rotating in the xy -plane, but the component of the moment in the z -direction will be zero (Figure 15.11 (left side)). When the rf pulse ends, the rotating magnetic field will generate its own rf signal, also at the Larmor frequency. This signal is known as the *free induction decay* (FID) signal. This signal can induce a small (very small) voltage in a receiver coil and it is this signal that is used to construct the MR image. Hence, in MR imaging, tissue protons become tiny radio transmitters. Immediately after the pulse ends, the signal generated is given by

$$S(t) = \rho \sin\theta(\cos(\omega_0 t)) \quad (15.13)$$

where ω_0 is the Larmor frequency, θ is the tip angle, and ρ is the density of spins. Note that a tip angle of 90° produces the strongest signal.

Over time, the spins will tend to relax toward the equilibrium position (Figure 15.11 (right side)). This relaxation is known as the *longitudinal* or *spin lattice* relaxation time and is approximately exponential with a time constant denoted as T_1 . As seen in Figure 15.11 (right side), it has the effect of increasing the horizontal moment, M_z , and decreasing the xy -moment, M_{xy} . The xy -moment is decreased even further, and much faster, by a loss of synchronization of the collective spins, since they are all exposed to a slightly different magnetic environment from neighboring atoms. This so-called *transverse* or *spin-spin* relaxation time is also exponential and decays with a time constant called T_2 . The spin–spin relaxation time is always less than the spin lattice relaxation time, so that by the time the net moment returns to equilibrium position along the z -axis, the individual spins are completely dephased. Local inhomogeneities in the applied magnetic field can increase the speed at which spins dephase. When the dephasing time constant is modified to include this effect, it is termed T_2^* (pronounced “tee two star”). This time constant also

includes the T_2 influences. When these relaxation processes are included, the equation for the FID signal becomes

$$S(t) = \rho \cos(\omega_0 t) e^{-t/T_2^*} e^{-t/T_1} \quad (15.14)$$

While frequency dependence (i.e., the Larmor equation) is used to achieve localization, the three variables in Equation 15.14 are used to achieve contrast: the two relaxation times, T_1 and T_2^* , and the proton density, ρ .

Proton density, ρ , for any given collection of spins is a relatively straightforward measurement: it is proportional to FID signal amplitude as shown in Equation 15.14. Measuring the local T_1 and T_2 (or T_2^*) relaxation times is more complicated and is done through clever manipulations of the rf pulse and local magnetic field gradients, as briefly described in the next section.

15.3.1 Magnetic Gradients

As mentioned above, the Larmor equation (Equation 15.11) is the key to localization. If each position in the sample is subjected to a different magnetic field strength, then the locations are tagged by their resonant frequencies. Magnetic field strength is locally varied by *gradient fields* applied using electromagnets, the so-called *gradient coils*, in the three dimensions. The gradient fields provide a linear change in magnetic field strength over a limited area within the MR imager. For example, the gradient field in the z -direction, G_z (along the bore of the imager), sets up a linear variation of magnetic field strength in the z -direction. This variation can be used to isolate a specific xy -slice in the object since, if magnetic field strength varies in the z -direction, only one xy -plane will have the appropriate field strength to satisfy the Larmor equation (Equation 15.11): only the spins in one xy -plane will have a resonant frequency that matches the rf pulse frequency.* This process is known as slice selection.

In MRI, since the rf pulse has a finite duration (i.e., it cannot be infinite in duration), it cannot consist of a single frequency,[†] but rather has a range of frequencies. The thickness of the slice, that is, the region in the z -direction over which the spins are excited, depends on the steepness of the gradient field and the narrowness of the rf pulse's bandwidth:

$$\Delta z \propto \gamma G_z z (\Delta\omega) \quad (15.15)$$

Very thin slices, Δz , require a very narrowband pulse, $\Delta\omega$, (i.e., a long pulse) and/or a steep gradient field, G_z . Gradient field strength is limited by concerns for patient safety.

15.3.2 Data Acquisition: Pulse Sequences

A combination of rf pulse frequency, magnetic gradient delays, and data acquisition delay is termed a *pulse sequence*. Two approaches could be used to identify the signal from a particular region. Use an rf pulse with only one frequency component, and if each location has a unique magnetic field strength, then only the spins in one region will be excited, those whose magnetic field correlates with the rf frequency as given by the Larmor equation. Alternatively, excite a broader region, then vary the magnetic field strength so that different regions are given different resonant frequencies. In clinical MRI, both approaches are used.

One of the first pulse sequences to be developed was the *spin echo pulse sequence*. This sequence has become a model for many other pulse sequences. The spin echo pulse sequence

* Selected slices can be in any plane by appropriate activation of the gradients during the rf pulse. For simplicity, this discussion assumes that the slice is selected by the z -gradient so that spins in the specific xy -planes are excited.

[†] Recall that in the discussion on time–frequency constraints in Chapter 6, there is a limit on the product of bandwidth and time duration (Equation 6.3): the shorter the pulse, the wider the bandwidth of the pulse. In MRI, rf pulses are fairly short; so, the bandwidth cannot be zero and the pulse must contain a range of frequencies.

Biosignal and Medical Image Processing

involves an echo technique, a trick for eliminating the dephasing caused by local magnetic field inhomogeneities and related artifacts (i.e., the T_2^* decay). One possible way to eliminate the influence of the T_2^* decay is to sample immediately after the rf pulse, but this is not practical. The alternative is to sample a realigned echo. If, after the spins have begun to spread out, their direction is suddenly reversed, they will come together again after a fixed delay. The classic example is that of a group of runners who are told to reverse direction at the same time, say 1 min after the start. In principle, they all should get back to the start line at the same time (1 min after reversing) since the fastest runners will have the farthest to go at the time of reversal, and vice versa. In MRI, the reversal is accomplished by a phase-reversing 180° rf pulse. The realignment will occur with the same timing as the misalignment and will be determined by T_2^* . This echo approach will only cancel the dephasing due to magnetic inhomogeneities, not the variations due to the sample itself, that is, those that produce the T_2 relaxation. That is actually desirable because the sample variations that cause T_2 relaxation are often of interest.

If all three gradients, G_x , G_y , and G_z , were activated prior to the rf pulse, then only the spins in one unique volume would be excited. However, only one voxel would be acquired for each pulse repetition and to acquire a large volume would be quite time consuming. Other strategies allow the acquisition of the entire lines, planes, or even volumes with one pulse excitation. The spin echo pulse sequence acquires one line of data in a spatial frequency domain. The sequence begins with a shaped rf pulse in conjunction with a G_z pulse that provides slice selection (Figure 15.12). The G_z pulse includes a reversal at the end to cancel a z -dependent phase shift. Next, a y -gradient pulse of a given amplitude is used to phase encode the data. This essentially selects a line, although it is a line in the frequency-domain space. This is followed by a second rf- G_z combination to produce the echo. As the echo regroups the spins, an x -gradient pulse frequency encodes the signal and the data are digitized with an ADC. Since the x -gradient pulse is on when the data are acquired, it encodes the position as a slight variation in frequency. Taking the Fourier transform of the acquired data gives a signal related to position. Because of the previous

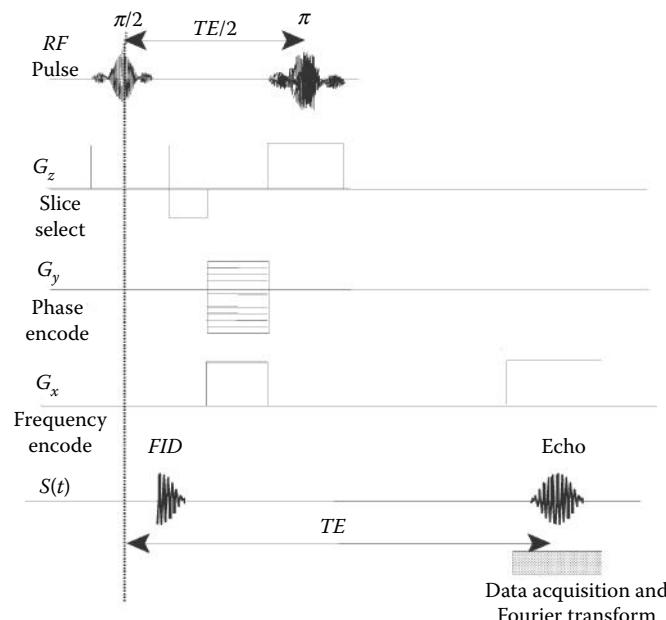


Figure 15.12 The spin echo pulse sequence. The events are timed with respect to the initial rf pulse. See text for explanation.

phase-encoding step, the data acquired consist of one line of frequency-encoded data, but again, this is one line in the frequency domain (termed “*k*-space” in MRI). Since the echo signal duration is several hundred microseconds, high-speed data acquisition is necessary to sample up to 256 points from 5- to 150-ms data acquisition period.

Two important time variables in the spin echo sequence are the pulse repetition time, *TR*, and the echo time, *TE*, as shown in Figure 15.12. These variables directly influence the signal that is detected in this sequence. In the spin echo pulse sequence, the signal equation is a variation of Equation 15.14:

$$S = k\rho(1 - e^{-TR/T_1})e^{-TE/T_2} \quad (15.16)$$

The image acquisition parameters, *TR* and *TE*, can be varied to emphasize one of the signal parameters in Equation 15.16. For example, to emphasize proton density, ρ , and to produce the so-called *PD weighted* image, a short *TE* (20 ms) is used in combination with a long *TR* (>1600 ms). To produce a T_1 *weighted* image, a short *TE* (10–20 ms) is combined with a short *TR* (300–600 ms), while a T_2 *weighted* image is produced using a long *TE* (>60 ms) and a long *TR* (>1600 ms). These different weightings lead to images that enhance certain types of tissue (water, fat, etc.). The spin echo provides high-resolution images, particularly with T_1 weighting, but has the serious disadvantage of requiring considerable time to build the image. Numerous modifications of the spin echo sequence have been developed to reduce the required imaging time.

As with slice thickness, the ultimate pixel size depends on the strength of the magnetic gradients. Pixel size is directly related to the number of pixels in the reconstructed image and the actual size of the imaged area, the so-called *field of view* (FOV). Modern imagers are capable of at least a 2-cm FOV with samples up to 256×256 pixels, giving a pixel size of 0.078 mm. In practice, image resolution is usually limited by SNR considerations, since as the pixel (or voxel) area decreases, the number of spins available to generate the signal diminishes. For imaging some areas of the body, special receiver coils are available that increase the SNR by positioning the receiver coil closer to the body part. These coils can lead to substantial improvement in image quality and/or resolution.

As with other forms of signal processing, MR image noise can be improved by averaging. Figure 15.13 shows an image of the *Shepp-Logan phantom* and the same image is acquired with different levels of detector noise.* Figure 15.13, lower right, shows the noise reduction resulting from averaging four of the images taken under the same noise conditions in Figure 15.13, lower left. Unfortunately, this strategy increases scan time in direct proportion to the number of images averaged.

15.4 Functional MRI

Once the MR image is acquired, additional processing is the same as used on other images. In fact, MR images have been used in a number of examples and problems in the previous chapters. One application of MRI does have some unique processing requirements: the area of fMRI. In this approach, neural areas that are active in specific tasks are identified by increases in local blood flow. As blood is paramagnetic, MRI is sensitive to changes in cerebral blood, a phenomenon known as BOLD, which stands for “blood oxygenation level dependent.” Special pulse sequences have been developed that can acquire images sensitive to the BOLD phenomenon very quickly. However, the effect is very small: changes in signal level are only a few percent or less.

During a typical fMRI experiment, the subject is given a task that is physical (such as finger tapping), purely sensory (such as being shown a flashing visual stimulus), purely mental (such

* The Shepp-Logan phantom was developed to demonstrate the difficulty of identifying a tumor in a medical image.

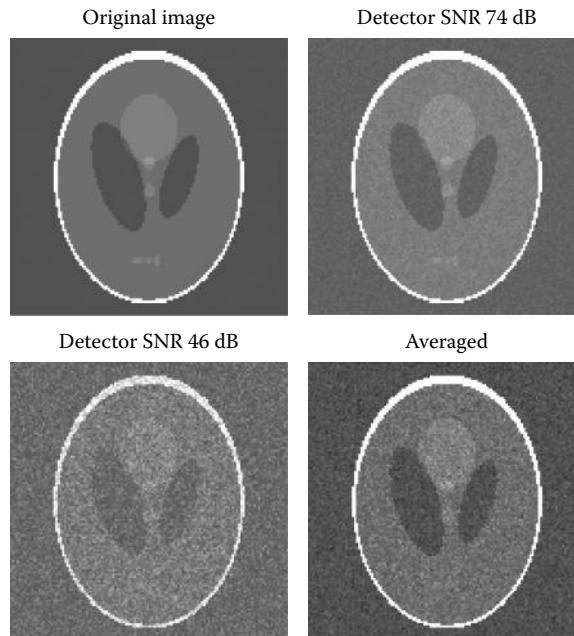


Figure 15.13 An image of the Shepp–Logan phantom (upper left) and the same image with noise added in the frequency domain (upper right and lower left). Averaging four of the noisier images provides the expected improvement (lower right). (Original image is from the MATLAB Image Processing Toolbox. The Math Works, Inc. Used with permission.)

as performing mathematical calculations or thinking about music), or involving sensorimotor activity (such as pushing a button whenever a given image appears). In single-task protocols, the task alternates with a nontask or baseline activity period. Task periods are usually 20–30 s long, but can be shorter and can even be single events under certain protocols. Protocols involving more than one task are possible and are increasingly popular. During each task, a number of MR images are acquired. The primary role of the analysis software is to identify pixels that have some relationship to the task/nontask activity.

There are a number of software packages available that perform fMRI analysis, some written in MATLAB such as “statistical parametric mapping” (SPM), others written in c-language such as “analysis of functional neuroimages” (AFNI). Most packages can be obtained at no charge off the internet. In addition to identifying the active pixels, these packages perform various preprocessing functions such as aligning the sequential images and reshaping the images to conform to standard models of the brain.

Preprocessing alignment is usually done using correlation and is similar to the examples in Chapter 13. Following preprocessing, there are a number of different approaches to identifying regions where local blood flow correlates with the task/nontask timing. One approach is simply to use correlation, that is, correlate the change in signal level, on a pixel-by-pixel basis, with a task-related function. Typically, such functions use a 1 to indicate the task is active and a 0 to indicate the nontask period producing a square wave-like function. More complicated task functions account for the dynamics of the BOLD process that has a 4- to 6-s time constant. Finally, some approaches based on ICA (Chapter 9) can be used to extract the task function from the data itself. The use of correlation and ICA analysis is explored in the next section and in the problems. Other univariate statistical techniques are common such as *t*-tests and *f*-tests, particularly in multitask protocols.

15.4.1 fMRI Implementation in MATLAB

A basic demonstration of fMRI analysis can be provided using standard MATLAB routines. The identification of active pixels using correlation with a task protocol function is shown in Example 15.4. Several files have been created that simulate regions of brain activity. The variations in pixel intensity are small and noise and other artifacts have been added to the image data, as would be the case with real data. The analysis presented here is done on each pixel independently, but in most fMRI analyses, the identification procedure requires activity in a number of adjoining pixels for identification. Lowpass filtering can also be used to smooth the image.

EXAMPLE 15.4

This example uses correlation to identify potentially active areas from MRI images of the brain. The task follows the block protocol described above with a nontask period followed by a task period. This sequence is repeated once. In this experiment, 24 frames were taken (typical fMRI experiments would contain at least twice that number). The first six frames were acquired during baseline activity (nontask period) and the next six were acquired during the task. This off-on cycle was repeated for the next 12 frames. The frames are stored in file `fmri1.mat` that contains all 24 frames.

Solution

Load the image in MATLAB file `fmri1.mat`. Generate a square wave function that represents the off-on task protocol and correlate this function with each pixel's variation over the 24 frames. To perform the correlation, use MATLAB's `corrcoef` so that the correlation is between ± 1.0 . Identify pixels that have correlation above a given threshold and mark the image where these pixels occur. Usually, this would be done in color with higher correlations given a brighter color, but here, we use grayscale. Use a correlation threshold of 0.5 and 0.7. Finally, display the time sequence of one of the active pixels. (Most programs provide time displays of pixels or regions that can be selected interactively.)

```
% Example 15.4 Identification of active area using correlation.
%
thresh = .7; % Correlation threshold
load fmri1; % Get data
i_stim2 = ones(24,1); % Construct task profile
i_stim2(1:6) = 0; % First 6 frames are no-task
i_stim2(13:18) = 0; % Frames 13 through 18 are no-task
%
% Do correlation: pixel by pixel over the 24 frames
I_fmri_marked = I_fmri;
active = []; % Matrix for active pixel locations
for iv=1:128 % Search vertically
    for ih=1:128 % Search horizontally
        for k=1:24
            temp(k) = I_fmri(iv,ih,1,k); % Get data across all frames
        end
        cor_temp = corrcoef([temp' i_stim2]); % Correlation
        corr(iv,ih) = cor_temp(2,1); % Correlation value
        if corr(iv,ih) > thresh % Apply threshold
            I_fmri_marked(iv,ih,:,:1) = I_fmri(iv,ih,:,:1) + corr(iv,ih);
            active = [active; iv,ih]; % Save supra-threshold locations
        end
    end
end
% ....display marked image, title.......
```

Biosignal and Medical Image Processing

```
imshow(I_fmri_marked(:,:,:,:,1)); title('fMRI Image');
```

To plot one of the active pixels, we select (arbitrarily) the second entry in the list of active pixels. We then collect the index values (i.e., m,n) of this pixel over the 24 frames and plot.

```
for i = 1:24 % Plot one of the active areas  
    active_neuron(i) = I_fmri(active(2,1),active(2,2),:,:,i);  
end  
plot(active_neuron); title('Active neuron #2');
```

Results

The marked image produced with a correlation threshold of 0.5 is shown in Figure 15.14. The actual active area is the rectangular area on the right side of the brain slightly above the center-line, and this threshold does detect all the pixels correctly. However, a number of error pixels can be observed (including some in the background) due to noise that induces these pixels to have a sufficiently high correlation with the task profile (0.5 in this case). In Figure 15.15, the

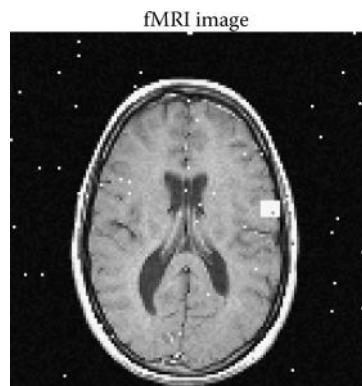


Figure 15.14 White pixels are identified as active based on correlation with the task profile. The actual active area is a rectangle on the right side slightly above the center line. Owing to inherent noise, false pixels are also identified, some even outside the brain. The correlation threshold is set at 0.5 for this image. (Original image is from the MATLAB Image Processing Toolbox. The Math Works, Inc. Used with permission.)

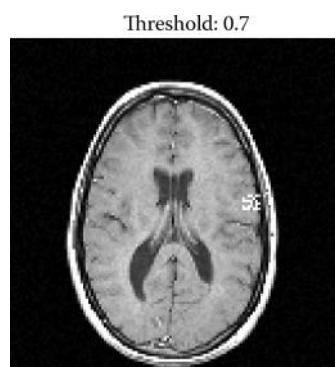


Figure 15.15 The same image as in Figure 15.14 analyzed using a higher correlation threshold. Fewer errors are seen, but the active area is only partially identified.

correlation threshold has been increased to 0.7 and most of the error pixels have been eliminated, but now, the active region is only partially identified. An intermediate threshold might result in a better compromise and this is explored in one of the problems.

Functional MRI software packages allow isolation of specific *regions of interest* (ROI), usually through interactive graphics. Pixel values in these ROI can be plotted over time and subsequent processing can be done in the isolated region. Figure 15.16 shows the variation over time (actually, over the number of frames) of one of the active pixels. Note the approximate correlation with the square wave-like task profile. The reduced correlation is due to noise and other artifacts, and is fairly typical of fMRI data. Identifying the very small signal within the background noise is one of the major challenges for fMRI image-processing algorithms. Note that in this example, the fMRI data came from a high-resolution 128×128 pixel image. Real fMRI images have fewer pixels and reduced resolution so that they can be acquired quickly. Active pixels from these low-resolution images are often displayed superimposed over a high-resolution image from an “anatomical scan” acquired separately, an average of the fMRI images, or a standardized brain image.

15.4.2 Principal Component and ICA

In the above analysis, active pixels were identified by correlation with the task profile. However, the neuronal response would not be expected to follow the task temporal pattern exactly because of the dynamics of the blood flow response (i.e., blood flow *hemodynamics*) that requires around 4–6 s to reach the final values. In addition, there may be other processes at work that systematically affect either neural activity or pixel intensity. For example, respiration can alter pixel intensity in a consistent manner. Identifying the actual dynamics of the fMRI process and any consistent artifacts may be possible by a direct analysis of the data. One approach is to search for components related to blood flow dynamics or artifacts from the pixel variation itself using either PCA or ICA.

Candidate active pixels in the ROI are first identified using either standard correlation or other statistical methods. A low threshold is used to ensure that all potentially active pixels are identified. Then the candidate active pixels from each frame are reformatted so that they form a 1-D vector. Vectors from all candidate pixels are arranged, usually as rows, in a matrix.

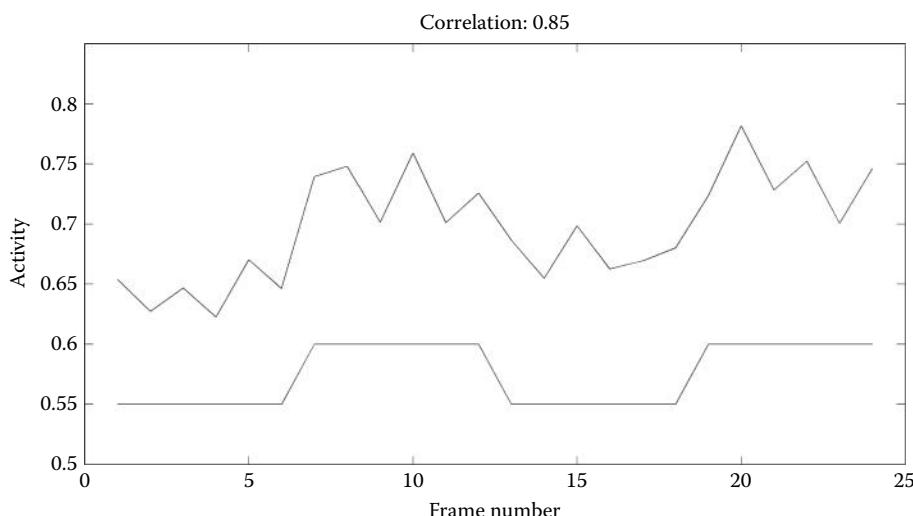


Figure 15.16 Variation in intensity of a single pixel within the active area of Figures 15.14 and 15.15. Some correlation with the task profile is seen, but noise is present.

Biosignal and Medical Image Processing

The data from each frame are now arranged rows of a matrix. ICA or PCA is applied to the ensemble composed of rows from all the frames so that each pixel is treated as a different sample and each frame as a different observation. Pixels whose intensity varies in a nonrandom manner should produce one or more components in the analyses. The component that is mostly like the task profile can then be used as a more accurate estimate of blood flow hemodynamics in the correlation analysis: the isolated component is used instead of the original task profile. An example of this approach is given in Example 15.5.

EXAMPLE 15.5

File `roi2.mat` contains a multidimensional data set, ROI, of pixels extracted from a region of interest of over 24 frames. These ROI data include the image pixels that are considered potentially active and were identified using correlation and a low threshold. This region was selected by the authors, but normally, the region would be selected interactively by an operator. The ROI data were obtained from the fMRI images used in Example 15.4 and include the block of active pixels at the mid-right-hand boundary (Figure 15.14). Determine the correlation function that best matches the dynamic characteristics of the BOLD response using PCA and ICA.

Solution

Reformat the ROI data so that pixels from each frame are strung together as a single row vector, then place that vector as one column of a matrix. Hence, each row represents the response over time of one pixel. Perform both an ICA and PCA analysis on the matrix treating the active pixels as separate observations of pixel variation over time. Plot the resulting components.

```
% Example 15.5 Use of PCA and ICA to identify signal
%      and artifact components in a region of interest
%
nu_comp = 2;                      % Number of independent components
load roi1;                         % Get ROI data
[r c dum frames] = size(ROI);      % Find size and number of frames
%
% Convert each image frame to a vector and construct ensemble where
% each row is a different frame.
data = [] ;                         % Data matrix
for i=1:frames
    col = [] ;                      % Vector to store active pixels
    for j=1:r
        col = [col ROI(j,:,:,:i)]; % String pixels into vector
    end
    data = [data col'];             % Combine each frame as a column in a matrix
end
%
% Now apply PCA analysis
[U,S,pc]= svd(data,0);            % Use singular value decomposition
eigen=diag(S).^-2;                 % Get eigenvalues
for i=1:length(eigen)
    pc(:,i)=pc(:,i) * sqrt(eigen(i)); % Take principle components
end
%
% Determine the Independent Components
w=jadeR(data, nu_comp);           %ICA
ica=(w * data);
..... Plot components.....
```

Results

The principal components produced by this analysis are shown in Figure 15.17. A waveform similar to the task profile is seen in the second plot below. Since this waveform is derived from the data, it should more closely represent the actual blood flow hemodynamics. The third waveform shows a regular pattern, possibly due to respiration artifact. The other two components may also contain some of that artifact, but do not show any other obvious pattern.

The two patterns in the data are separated better by ICA. Figure 15.18 shows the two independent components, and both the blood flow hemodynamics and the artifact are clearly shown. The former can be used instead of the square wave-like task profile in the correlation analysis. The results of using the profile obtained through ICA in the correlation instead of the square wave function are shown for a different set of images in Figures 15.19 and 15.20. Both activity maps are obtained from the same data set using the same correlation threshold (0.7). In Figure 15.19, the task profile function is used, while in Figure 15.20, the hemodynamic profile (the function in the upper plot of Figure 15.18) is used in the correlation. The improvement in identification is apparent. When the task function is used, very few of the areas actually active are identified and a few error pixels are identified. Figure 15.20 contains about the same number of errors, but all the active areas are identified. Many of the errors seen in both Figures 15.19 and 15.20 can be eliminated by requiring a minimum number of adjacent pixels to be active as is done in most fMRI analysis software.

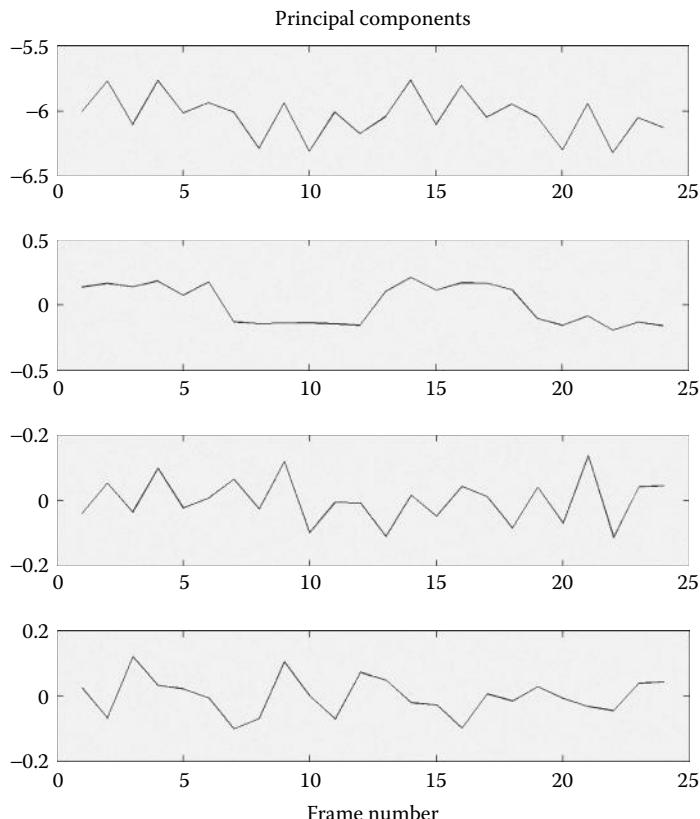


Figure 15.17 First four components from PCA applied to an ROI in Figure 15.14 that includes the active area. A function similar to the task is seen in the second component. The third component also has a possible repetitive structure that could be related to respiration.

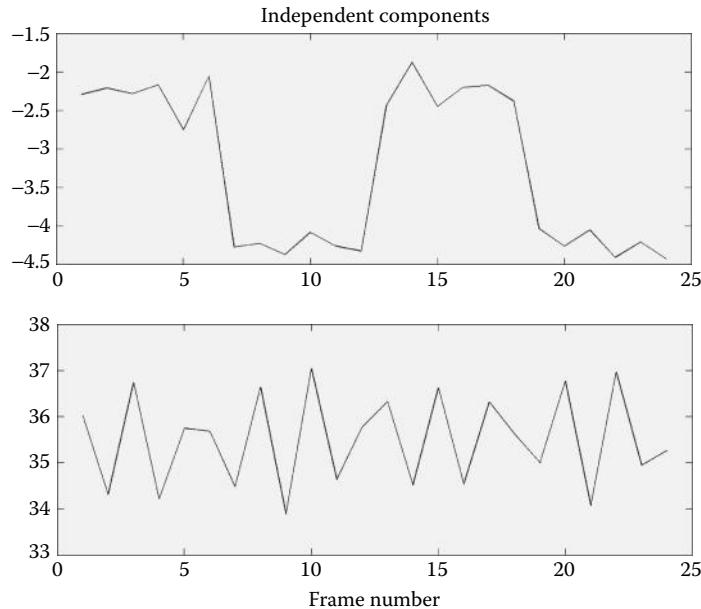


Figure 15.18 Two components found by ICA. The task-related function and the respiration artifact are now clearly identified.

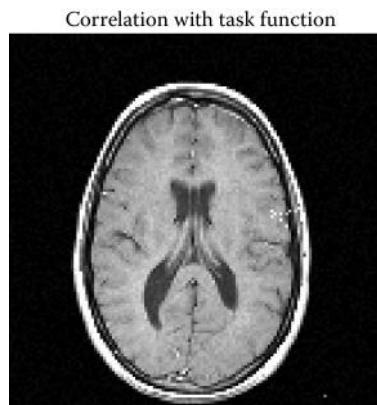


Figure 15.19 Activity map obtained by correlating pixels with the square wave task function. The correlation threshold was 0.7.

15.5 Summary

Different imaging modalities are based on different physical principles. Common clinical modalities include x-ray, computed tomography (CT), positron emission tomography (PET), single photon emission computed tomography (SPECT), magnetic resonance imaging (MRI), and ultrasound. All of these approaches, except for simple x-ray images, require some type of image processing to produce a useful image.

In CT, a collimated x-ray beam is used to scan across the tissue of interest in a grid or fanbeam pattern. The data consists of the intensity of these x-ray beams after absorption by the intervening tissues. The relationship between tissue absorption and the resultant beam intensity data is known

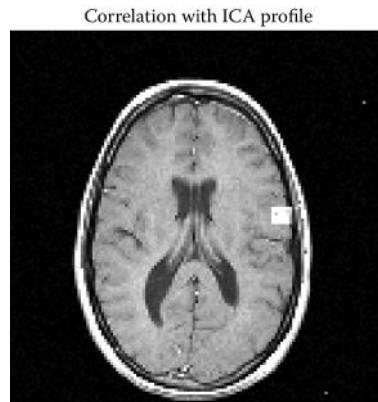


Figure 15.20 Activity map obtained by correlating pixels with the estimated hemodynamic profile obtained from ICA in Example 15.5. The correlation threshold was 0.7, the same as that used to obtain the image in Figure 15.19, but the function that was correlated with the pixels was derived using ICA decomposition of all pixel activities within the `roi`. Although approximately the same number of errors occur, many more active pixels are correctly identified.

as the radon transform. Analysis software must solve an inverse problem: reconstruction of an image of the tissue absorptions characteristics based on the beam absorption pattern, the inverse radon transform. While theoretically it ought to be possible to reconstruct the image by solving a large number of simultaneous equations, this approach is impractical. Instead, several approximate methods have been developed including the popular filtered backprojection. In this method, each pixel in the approximation is given the value (sum of average) of all the beams that transect that pixel. This approximate image is blurred, so a derivative like filter is used to sharpen the image. Filtered backprojection can be efficiently implemented in the frequency domain, including filtering through multiplication, and the image reconstructed using the 2-D inverse Fourier transform.

In MRI, tissue protons have their spins (a quantum mechanical property) aligned by a very strong magnetic field. These “spins” do not actually align with the magnetic field, but precess around the field direction at a velocity related to the strength of the field. In clinical imagers, the precession velocity or resonance frequency is around 64 MHz or higher. If these spins are exposed to a radio-frequency (rf) electromagnetic field at the resonance frequency, they will tip away from alignment with an angle proportional to rf pulse timing. When the rf pulse ends, the spins return to their original position and radiate a small rf signal in the process. This signal diminishes as spins realign, but also as they dephase and begin to cancel. The time course of dephasing and realignment depends on the environment and can be used to develop image contrast. Image location is determined by giving the spins slight differences in resonance phase and frequency though manipulation of additional magnetic fields called gradient fields. The combination of rf pulse timing and gradient field manipulations is termed as pulse sequence. Many sequences have been developed that essentially trade off image resolution for acquisition time.

A special application of MRI is used to image brain function. In functional MRI (fMRI), pixel intensity is correlated with a task profile: usually an on-off pattern of motor, sensory, or other brain-driven activity. The changes in neuronal activity produce changes in local blood flow that are sensed by MR imaging. By correlating the changes in pixel intensity over sequential images (hence over time) with the task profile, areas related to the activity can be identified. The intensity changes are very small and a number of different techniques have been developed to circumvent problems related to noise, including the use of PCA and ICA to develop improved task profiles.

PROBLEMS

- 15.1 The file `proj_prof1.mat` contains a projection profile p that was acquired at 1.0° angles from 1° to 180° . Apply the unfiltered back projection using routine `iradon` with the filter option '`'none'`'. This produces a blurred image with no derivative filter. This produces an blurred image with no derivative filter. Filter this image using the unsharp filter from Chapter 13. This filter is not ideal for filtering raw back-projected images; so, apply the filter two times in succession to get enough filtering to remove most of the blur. Plot both the unfiltered and double-filtered images. Note: You need to apply the image normalization routine `mat2gray` to both the raw back-projected image and the twice-filtered image.
- 15.2 Load slice 13 of the MR image used in Example 15.3 (`mri.tif`). Construct parallel beam projections of this image using the Radon transform with angular spacings of 5° and 10° between rotations. In addition, reduce the spacing of the 5° data by a factor of 2. Reconstruct the three images (5° unreduced, 5° reduced, and 10°) and display along with the original image. Multiply the images by a factor of 10 to enhance artifacts in the background.
- 15.3 The data file `proj_prof3.mat` contains projections of the Shepp–Logan phantom, an image featuring several different densities and originally developed to show the difficulty of detection of tumors in a medical image. Noise has been added to this image. Reconstruct the image using the inverse Radon transform with four filter options: no lowpass filter (i.e., only the derivative Ram–Lak filter, the default), the Shepp–Logan filter, the Hamming filter, and the cosine filter. Note that the four options produce about the same image and do not strongly filter the image.
- 15.4 Repeat Problem 15.3, but in addition to Ram–Lak or the default option (i.e., no lowpass filter), create three images filtered with the cosine lowpass filter. For the three cosine filters, modify the effective cutoff frequencies by setting the frequency scaling to 1.0 (no change), 0.5 ($f_c/2$), and 0.25 ($f_c/4$). Note the improvement in noise reduction with the lower cutoff frequencies, but at the expense of sharpness.
- 15.5 Load the data file `proj_prof3.mat` containing projections of the Shepp–Logan phantom. Use `iradon` with the default Ram–Lak option to generate an image without lowpass filtering. Then lowpass filter this image using a 7×7 Gaussian filter (as described in Chapter 13) with a sigma of 2.0. Compare the image produced by this filtering with that produced by `iradon` using the Shepp–Logan filter with a frequency scale of 0.3.
- 15.6 The second output argument from `iradon` is a vector containing the frequency characteristics of the filter used in the back projection algorithm. Apply `iradon` to a dummy projection profile to obtain and plot the magnitude frequency spectra of the Ram–Lak, Hamming, cosine, and Shepp–Logan filters. To create a dummy projection profile, you can apply `radon` to any image such as the one in Example 15.1.
- 15.7 Load the image `squares.tif`. Use `fanbeam` to construct fan beam projections and `ifanbeam` to produce the reconstructed image. Repeat for two different beam distances: 100 and 300 (pixels). Plot the reconstructed images. Use a `FanSensorSpacing` of 1.
- 15.8 The rf pulse used in MRI is a shaped pulse consisting of a sinusoid at the base frequency that is amplitude-modulated by some pulse-shaping waveform. The sinc waveform, $\sin(x)/x$, is commonly used to modulate the sinusoid. Construct a

simulated rf pulse and find its frequency characteristics using the Fourier transform. Note the narrow frequency-domain energy centered around the base frequency.

To construct a shaped pulse, generate a baseline signal consisting of $\cos(2\pi f_1 t)$ where f_1 is 64 MHz. To simplify the numbers, assume everything is scaled to 10^6 ; so, $f_1 = 64$. Make $f_s = 1000$ and construct a time vector from -0.5 to 0.5 , which due to the scaling will be in microseconds. Use a time vector of ± 0.5 because the sinc function is symmetrical about $t = 0$. [Hint: Construct the time vector as $-0.5:1/f_s:0.5$. Since $f_s = 1000$ and is assumed to be in MHz, the sampling interval, T_s , will represent 1.0 ns.] Next, construct the shaping sinc waveform using the MATLAB sinc function. Make the sinc function frequency much less than f_1 , the sinusoidal frequency; specifically, $f_2 = f_1/50$. Note that since the sinc function is already normalized by π , the input vector to this function needs to be multiplied by 2.0 and not by π (i.e., $\text{sinc}(2*f2*t)$). To get the shaped pulse, multiply, point by point, the sinc and cosine signals. Plot this shaped rf signal, take the Fourier transform as requested, and plot. Note: Plot only the first 150 points (up to 150 MHz). Repeat the plots for $f_1 = 32$ MHz.

- 15.9 Repeat the methods used in Problem 15.8 to construct an rf pulse with a center frequency of $f_1 = 64$ MHz (again scaled to 10^6) modulated by a sinc signal with a frequency of $f_2 = f_1/50$. The width of the rf pulse can be effectively decreased by increasing the frequency of the sinc signal. Shorten the rf pulse by increasing the sinc frequency to $f_1/10$ (a fivefold increase in the sinc frequency) and also to a further increase of $f_1/5$. Take the Fourier transform of all three rf pulses and plot along with the three rf time-domain signals. Note the increase in effective bandwidth. This increase in bandwidth is predicted by time-frequency limitation first described in Chapter 6 and quantified by Equation 6.3.
- 15.10 Repeat the approach used in Problem 15.9, but use only the two longer sinc signals, that is, $f_{\text{sinc}} = f_1/10$ and $f_1/5$ where $f_1 = 64$ MHz (scaled as in the previous problems). Use MATLAB's find routine to estimate the bandwidth of the two spectral plots. Find the first and last points where the spectrum is greater than 0.707 of its maximum value. (Search only the frequencies $< f_s/2$ to avoid the reflected frequencies.) Display the bandwidth on the spectral plots and confirm that doubling the sinc frequency, which halves the effective rf pulse duration, increases the pulse bandwidth by a factor of 2.
- 15.11 This problem further shows the benefits of image averaging. The file mri_noise.mat contains a multidimensional data set, I , consisting of eight MR images with added noise. Each image is in a different frame of the data set. Show the image in the first frame as an example of an unaveraged image. Then show the first two frames averaged, the first four, and then all eight images averaged. Note the noticeable improvement in image quality.
- 15.12 Example of the identification of an active area using correlation. Load file fmri3.mat that contains the multidimensional variable, I_fmri , which consists of 24 frames of an MR image. Construct a stimulus profile assuming the same task profile as in Example 15.4: the first six frames taken during no-task conditions, the next six frames during the task condition, and then the cycle is repeated. Rearrange the code in Example 15.4 so that the correlation coefficients are computed first, then the thresholds are applied (so that each new threshold value does not require another calculation of correlation coefficients). Search for the threshold that produces no (or very few) error pixels. Also, search for the threshold that captures the entire active area (which is square in shape). Finally, find a threshold you consider the best.

Biosignal and Medical Image Processing

These images contain more noise than those used in Example 15.4; so, even the best threshold will contain error pixels.

- 15.13 Example of identification of an active area using correlation. Repeat Problem 15.12 except filter the individual images before applying the analysis. Use a 4×4 averaging filter (`fspecial` can be helpful in constructing the filter). Find only the optimal threshold. Note the considerable reduction in noise pixels.
- 15.14 Example of identification of an active area using correlation. Repeat Problem 15.12 except filter the matrix containing the *pixel correlations* before applying the threshold. Use a 4×4 averaging filter. (Again, `fspecial` can be used.) Note the need to reduce the threshold to capture all the active pixels. Nonetheless, the filtering does produce an overall reduction in noise. This approach emphasizes pixels that have high correlation and are contiguous.
- 15.15 Example of using PCI and ICI to identify the signal and artifact. Load the ROI file `roi4.mat` that contains the variable `ROI`. This variable contains 24 frames in a small region around the active area of `fmri3.mat`. Reformat to a matrix as in Example 15.5 and apply PCA and ICA analysis. Plot the first four principal components and the first two independent components. Note the very slow time constant of the blood flow hemodynamics.

16

Classification I

Linear Discriminant Analysis and Support Vector Machines

16.1 Introduction

In many circumstances, the output of a signal-processing or image-processing algorithm can be sent directly to a human operator through some sort of graphics or image display. For example, a reconstructed MR image can be used directly by a radiologist to make a medical diagnosis; no further computer analysis is necessary. However, sometimes it is appropriate for the computer to use the fruits of signal or image analysis to suggest diagnoses indicated by the data, that is, to associate patterns of measurements with a particular disease state or condition. There are two common situations where computers are called upon to evaluate a medical condition: real-time monitoring of a patient's condition and situations where the available data are just too vast or too complex for easy interpretation by a doctor or medical technician. A simple example of the first situation is found in heart rate monitoring where an alarm is sounded when the heart rate becomes abnormal. The patient's condition is monitored by applying upper and lower thresholds to the heart rate, a measurement variable usually extracted through signal processing applied to the patient's ECG signal. An example of the second category is found in the evaluation of data from mass screenings where there is a large amount of data and analysis by human operators would be costly.

Determining a disease or condition from a range of data is an application of *classification*. A *classifier* makes an association between a pattern of data and two or more *classes* or conditions. A classifier may be viewed as just another input-output device where the input is a set of measurement data and the output is the most likely class associated with that data set. Often, only two classes are used in medical applications representing normal or abnormal conditions. Medical applications that involve multiple categories can also occur, for example, the identification of cell types from microscopy images or the classification of ectopic heart beats from the ECG signal.

Figure 16.1 shows a block diagram of a typical classification problem. Here, the classifier receives a number of variables often derived from signal- or image-processing algorithms and, based on an analysis of these combined parameters, estimates the state of the system. The inputs can take on any value, but the output is a discrete variable that specifies the class or condition. When only two classes are involved, the output values are either 0 or 1, to identify the

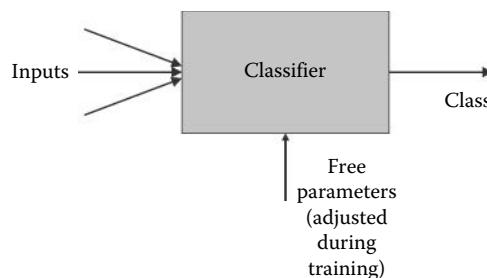


Figure 16.1 A classifier is an input–output device that takes in a number of inputs and, based on the pattern of these inputs, determines the most likely condition associated with that pattern. The inputs can have any value, but the output is a discrete value that uniquely identifies a class. All classifiers have a number of free parameters and these parameters are adjusted, often adaptively, to maximize classifier performance.

two classes.* In addition to determining a current medical condition, classifiers are sometimes called upon to make predictions: to distinguish between a normal healthy patient and one who is a candidate for some particular disease or disorder.

The earliest classifiers were based on Bayesian analysis, an approach termed *maximum likelihood*. This technique essentially finds the optimum separation boundary between the probability distributions of the various classes. While this technique guarantees optimal classification, it requires that the distributions of the classes be known ahead of time, a condition that is never met in practice. It also requires that the distributions be Gaussian, another condition that does not usually apply to real-world situations. Because of these serious limitations, a wide variety of other methods have been developed.

Classification is one component of a more general analysis that tries to associate a pattern of input variables with either a specific class or another variable. If the output is a variable, the analysis is referred to as “*regression*,” while if the output is a discrete number identifying a specific class, the analysis is referred to as “*classification*.” Here, we explore only classification algorithms that produce discrete outputs since they are more frequently used in medical applications. Since classifiers establish a relationship between the pattern of inputs and a discrete output, they can be viewed as mathematical functions and the development of a classifier can be thought of as *function approximation*. Other terms for classification include *identification*, *estimation*, or *pattern recognition*.

In principle, only a single measurement variable is required for classification. A heart rate monitor that sets off an alarm if the heart rate is too low or too high is an example. Similarly, to determine if a patient is either normal or has a fever, you only need a measurement of body core temperature. In both cases, the classifier is based on simple thresholding. In the case of heart rate, two thresholds classify heart rate into three classes: alarmingly low, alarmingly high, and normal. In the case of body temperature, a single threshold (98.6°F) is used to determine if the individual is normal or feverish. You could add more classes; so, for example, someone with a temperature above 102°F was classified as “very feverish,” but again, the classification would be based on a simple threshold applied to a single variable. Most classifiers do contain threshold elements, but when two or more variables are involved, a wide range of algorithms exist to try to make the best classification from the pattern presented by all the variables. Again, the role of the classifier is to determine the most likely class associated with a given input pattern. As the term “most likely” suggests, the classifier may not always come up with the correct classification (i.e., diagnosis), but then even highly trained humans make mistakes.

* Some classifiers output -1 and $+1$ to represent the two states. The mathematics of some classifier algorithms is simplified by using one of the two coding schemes.

Many algorithms have been developed to classify input patterns of two or more variables. As with many of the topics presented here, classification justifies an entire textbook and several such books are listed in the bibliography of this book. In this and the next chapter, only a couple of the most common methods of classification are discussed. These methods have been chosen because they are popular and/or effective in a large number of situations. The goal here is to understand the basic problem, the most common options that are available to solve the problem, and the strengths and limitations of these approaches.

Computer-based classification is done using two basic strategies: *supervised* and *unsupervised* learning. Both these strategies fall under the general heading of *machine learning*. In unsupervised learning, the classifier attempts to find patterns within the input data itself: the classifier has no *a priori* knowledge of the data patterns and perhaps not even the number of classes that exist. In supervised learning, the classifier is first *trained* using data for which the correct classes are known. The training data are known as the *training set* and include the correct answer(s), that is, the correct classifications.

During the training period, the free parameters associated with the classifier are adjusted adaptively to minimize classification errors. Classifier performance can be evaluated *during* training using a *validation set* for which the correct classifications are also known but are not used to modify classifier parameters. When training is complete, the classifier is put to the test using a data *test set* where it performs its designed function to determine the most likely condition based on a given data pattern. It is only the classification error that occurs during the testing phase that really matters; hence, a good classifier should be able to perform with minimum error on data that it has never seen. A common failure of classifiers is that they perform well on the training set (after training of course), but then they do poorly on the test data, their real-world application. Such classifiers are said to *generalize poorly* and/or be *overtrained*. The problem of classifier generalization is discussed in Sections 16.1.1 and 16.6 below.

A geometrical approach can be used to study classifiers and the classification problem. Although classifiers frequently deal with a number of input variables, it is easiest to visualize the data set and the classification operation when only two input variables are involved; so, most of the examples here use only two input variables. This is not really a limitation since the classification methods described extend to multiple inputs with minimal modification as demonstrated in the examples and problems.

Several two-variable input data patterns are shown in Figure 16.2. The input variables are plotted against one another in a scattergram as in Figure 9.1. In Figure 16.2a, the two classes are identified using different symbols (circles and squares in this case), and are easily separated by a straight line as shown. This line is known as a *decision boundary*. The classifier only needs to specify input combinations below and to the left of the line as class 0, and combinations that fall above and to the right of the line as class 1 (or alternatively as class -1 and class 1). Such data are called *linearly separable*. In Figure 16.2b, the data cannot be separated by a straight line, but can be separated by a curved line; hence, they are *nonlinearly separable*. In Figure 16.2c, only a very complicated boundary can separate the two classes. Nonetheless, even in this partially overlapping data set, a boundary can be found that separates the two classes. Usually, such complicated boundaries, tuned as they are to fit the training set, do not generalize well.

From the three data sets and their respective decision boundaries, you might think that the best way to classify data is by drawing lines between them and constructing an algorithm based on these lines. The problem with this approach is that it is hard to do when more than two parameters are involved, and such an approach is unlikely to generalize well. Consider the two data sets in Figure 16.3. The left-hand set is just a repetition of the training set data in Figure 16.2c with its complicated boundary. The right-hand data constitute a test set that has the same statistical properties as the left-hand set; specifically, the two classes are both the Gaussian distributions having the same mean and standard deviation. The complicated boundary that worked well on the training set now makes three errors on the test set. You might imagine that a well-placed straight line would do a better job, and indeed with only two errors this is the case (Figure 16.3c).

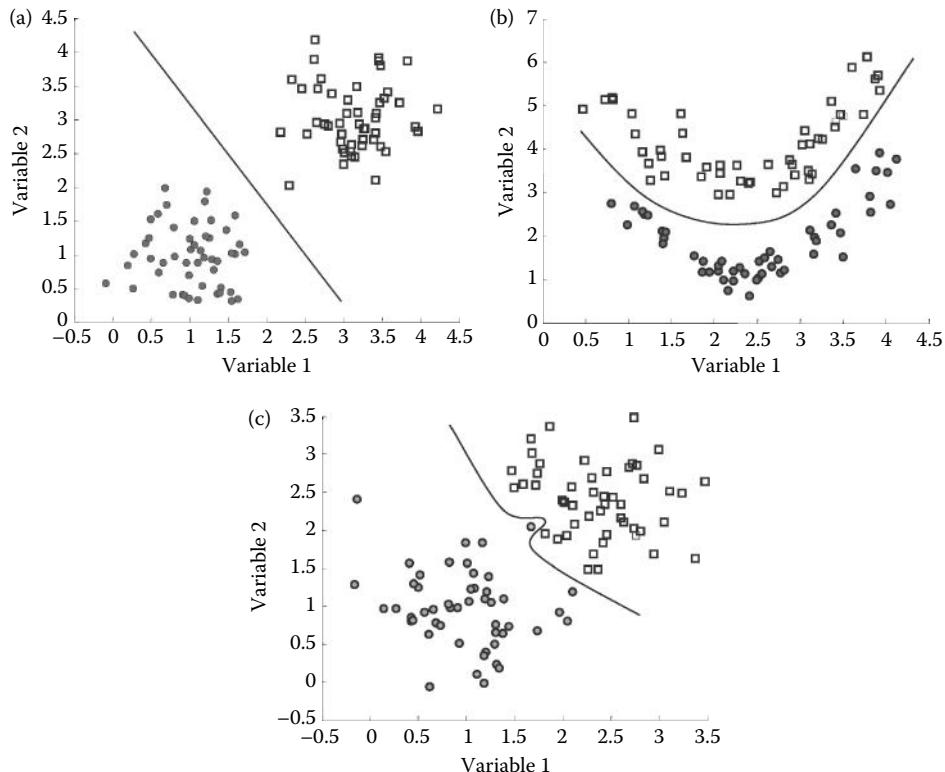


Figure 16.2 Input data patterns displayed as a scattergram. (a) The two classes (square and circle) are apparent and can be easily classified by using a straight line as a boundary between them. This separating line is termed the decision boundary. (b) The two classes are still distinguishable but a curved boundary is needed to separate them. (c) The two classes overlap and a very complicated boundary is required to fully separate them.

This straight line is derived from the *same training set data* of Figure 16.3a. This demonstrates that a straight line generalizes better than the complicated curve for these data. Put another way, the complicated curve is a result of *overtraining* on the training set. In fact, it can be shown that a simple straight line provides optimal separation for Gaussianly distributed data.

16.1.1 Classifier Design: Machine Capacity

Many different classifier algorithms are available on the web as downloadable MATLAB files. This relieves us of the burden of developing our own software code. Yet, we still need to make important decisions. In addition to selecting an approach, the most important choice is how complicated to make the decision boundaries. The test set data in Figure 16.3 show that a complicated boundary is not always the best even if it fits the training set better than a simple boundary.

Classification is a form of *machine learning* and the complexity of a classification algorithm is often referred to as *machine capacity*. Increasing machine capacity leads to a more complex decision boundary; so, machine capacity is a major factor in classifier design. Machine capacity should match in some way the requirements of the data. If the classifier has more capacity than appropriate for the data, it will *overtrain* on the data, performing well on the training set but not generalizing well on the test set data. A machine with too little capacity will show excessive errors in training as well as subpar performance in classifying the test set data. Machine capacity is closely related to the complexity of the classification algorithm, specifically, the number of

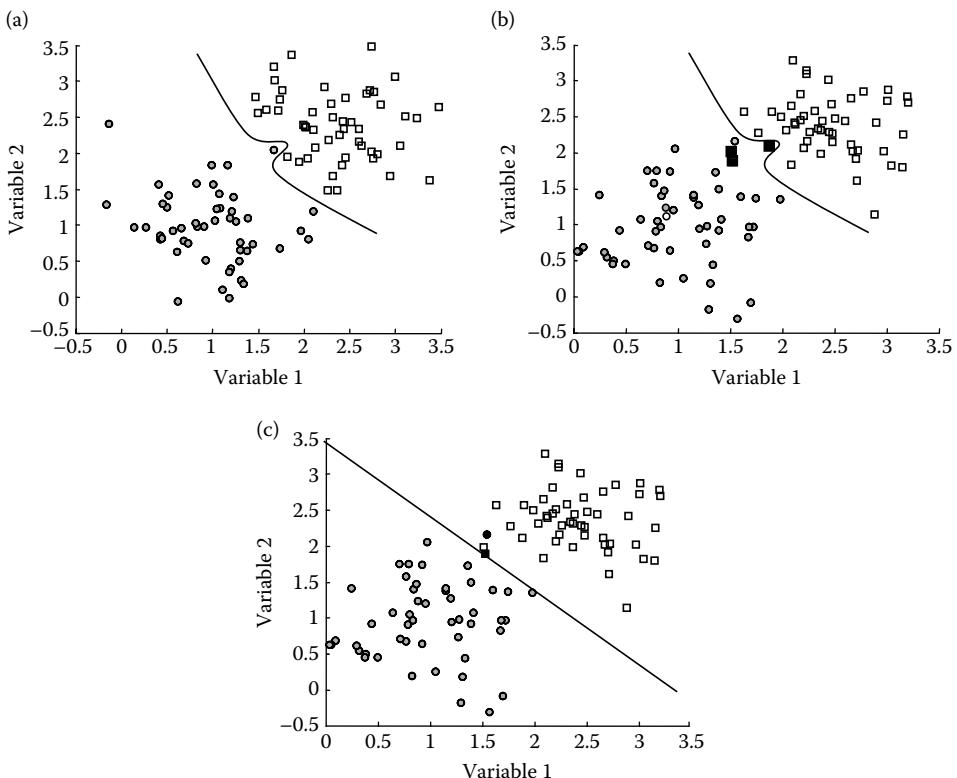


Figure 16.3 Input data patterns as in Figure 16.2c. (a) The overlapping data set from Figure 16.2c and its boundary. (b) A *test set* having the same statistical properties as in (a), showing three classification errors that arise from the training set boundaries (three filled squares). (c) A simple straight-line boundary produces only two errors on the test set (filled circle and square). The complicated boundary is a classic example of overfitting or overtraining and is discussed in a later section of this chapter.

free parameters. The problems associated with machine capacity and overtraining are examined in the context of the specific classification techniques described here.

16.2 Linear Discriminators

As the name implies, linear classifiers use decision boundaries that are linear: straight lines for two variables, planes for three variables, and *hyperplanes* for four or more variables. These classifiers produce only a single boundary; so, they can separate only two classes at a time. However, they can be applied to subsets of the data to identify more than two classes as shown in a later example.

In a linear discriminator, class is specified through the output of a linear equation:

$$y = \sum_{i=1}^M x_i w_i + b \quad (16.1)$$

where M is the number of input variables (i.e., the number of different types of measurements) and x_i are the input variables: one variable for each type of measurement. (It is assumed that each variable contains the same number of measurements and that there are no “missing variables,” i.e., that there are values for each type of measurement in the data set.) The classifier’s free parameters include w_i , the weights (one for each type of measurement) and b , the *bias* or

Biosignal and Medical Image Processing

offset. The output y indicates the class depending on whether it is greater or less than 0.5. In other words, if $y > 0.5$, then the classifier is predicting that the input data belong to class 1 and if $y \leq 0.5$, the classifier is predicting that the input data belong to class 0. Note that “class 0” and “class 1” are arbitrary names for the two classes and can stand for any two categories such as “normal” versus “diseased,” “malignant” versus “benign,” or other dichotomous conditions.

Since most of the examples presented here involve only two variables ($M = 2$), x_i consists of two vectors, x_1 and x_2 . However, all the classifiers described here extend to any number of input variables. In fact, with the exception of plotting routines, most of the software used in the examples can be applied without modification to data sets of three or more variables.

In addition to the input data x_i , a training set must also include the answers: the correct classifications to each input pattern. These correct answers are given by vector d having the same length as the number of input patterns or measurements (i.e., the length of the variable vectors). The correct class is defined as

$$d = \begin{cases} 0 & \text{or} & -1 & \text{class 0} \\ 1 & & & \text{class 1} \end{cases} \quad (16.2)$$

In some cases, d will specify the two classes as -1 and $+1$ and at other times as 0 and 1 depending on the classifier. In the case where there are more than two classes, d is usually a vector of 0s and 1s where the *location* of the 1 indicates the class:

$$d = \begin{cases} 1000 & \text{class 0} \\ 0100 & \text{class 1} \\ 0010 & \text{class 3} \\ 0001 & \text{class 4} \end{cases} \quad (16.3)$$

This coding scheme simplifies the code for training classifiers that treat multiple classes as will be shown in Example 16.8. Alternatively, consecutive numbers may be used to identify the classes (i.e., $0, 1, 2, \dots, N$).

The output of the linear classifier to any set of input variables is defined in Equation 16.1 and is wholly determined by the weights and bias. These constitute the free parameters of this classification method. To construct a linear classifier, it is only necessary to find values for w_i and b that best separate the data. Equation 16.1 can be simplified by lumping the bias, b , with the other weights, w_i , if the last input variable is always 1.0. This is done in the linear classifier and in some other classifiers as it simplifies the calculations. In this case, the bias is taken as the last weight variable, w_{M+1} , and a dummy measurement of 1.0 is added to each variable in the input data set. The input data set then becomes a set of vectors, $x_i = [x_1, x_2, x_3, \dots, x_M, 1s]$, where 1s is a vector filled with 1s the same length as the variable vectors. For inputs consisting of two variables, the input becomes $x_i = [x_1, x_2, 1s 1]$. When this is done, Equation 16.1 simplifies to

$$y = \sum_{i=1}^{M+1} x_i w_i = \mathbf{X} \mathbf{w} \quad (16.4)$$

where \mathbf{w} is the combined weight–bias vector and \mathbf{X} is the matrix of input variables (including the column of 1s).

Sometimes, linear classifiers are optimal, for example, a linear boundary is the best way to separate two Gaussian distributions. Linear classifiers can be trained very quickly and are easy to implement. One popular method for setting the weights is to apply the least squares approach to the training set. In this approach, the weights are selected to minimize the sum of the squared error between the output of the classifier, y , and the known class. The sum of the squared error between the output of the linear classifier and the correct class is given by

$$\varepsilon^2(w) = \sum_{i=1}^{N+1} (d_i - x_i^T w)^2 \quad (16.5)$$

In matrix notation:

$$\varepsilon^2(w) = (\mathbf{d} - \mathbf{Xw})^T (\mathbf{d} - \mathbf{Xw}) \quad (16.6)$$

Minimizing $\varepsilon^2(w)$ in Equation 16.6 is done in the usual manner: $\varepsilon^2(w)$ is differentiated with respect to w , set to zero, and the resultant equation is solved for w . If the matrix $\mathbf{X}^T \mathbf{X}$ is nonsingular, then a unique solution can be obtained as

$$w = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{d} \quad (16.7)$$

The implementation of this equation in MATLAB is straightforward as shown in the first example.

EXAMPLE 16.1

Generate a test set consisting of two Gaussian distributions with centers 3.0 standard deviations apart. The test set should include the correct classification in vector \mathbf{d} . Apply the least squares method to classify these two data sets and then plot the results. Plot the two classes as circles and squares as in Figure 16.2 and plot any misclassified data points in black. Finally, plot the decision boundary produced by the linear classifier.

Solution

This example uses the routine `gen_data2` found in the associated files to create the training set data. The format of this routine is

```
[X, d] = gen_data2(dist, angle, 'type', [class_range], npts);
```

where X is the training set data and d is the actual class, the output we desire from the classifier. The only required input variable is `dist`, the geometrical distance between the means of the two distributions (with respect to standard deviation of the data sets). Both distributions have a standard deviation of 1.0. The `angle` defines the angle between distributions in degrees with a default of 30°, which is used here. The optional variable `'type'` indicates the type of distribution, in this case, '1' for linear Gaussian (the default). Other options are discussed when they are used. The optional variable `class_range` is a two-variable vector that specifies the value to be given to each class, in this case, 0 and +1 (vs. -1 and +1). Since 0 and +1 are the defaults, they do not have to be specified here. The final optional input variable is `npts`, the number of patterns in the data set. In this example, the default value of 100 input patterns is used.

After generating the training set data, the weights are trained using Equation 16.7. These weights along with the original data, X , and correct classifications, d , are passed to the routine `linear_eval` that plots the data identifying misclassifications and also plots the decision boundary.

```
% Example 16.1 Linear classification using least squares.
%
distance = 6; % Distance between distribution centers
[X, d] = gen_data2(distance); % Generate data, use defaults
[r, c] = size(X);
X = [X, ones(r, 1)]; % To account for bias
w = inv(X'*X) * (X'*d'); % Train the classifier
linear_eval(X, d, w); % Evaluate the classifier
```

Analysis

The support routine `linear_eval` is found in the associated files and is presented below. It has as inputs the original data, X , the correct classifications, d , and the weights, w , which define the linear classifier. The correct classifications are assumed to be coded as in Equation 16.2. The routine uses the weights established during training to classify the data set using the matrix form of Equation 16.1: $y = Xw$. Then the data are plotted. The plotting section checks to see if d and y match using 0.5 as the decision threshold between the two classes, that is, the data are correctly classified if $d = 1$ and $y > 0.5$, or $d = 0$ and $y \leq 0.5$. When the input pattern is incorrectly classified (i.e., $d = 1$ and $y \leq 0.5$, or $d = 0$ and $y > 0.5$), then the circle or square is filled in black. The misclassifications are counted separately and are used to construct performance measures of sensitivity and specificity as described later.

When you generate the decision boundary, recall that this decision boundary occurs at $y = 0.5$. Input patterns that produce classifier outputs >0.5 are classified as belonging to class 1, while patterns that produce classifier values less than or equal to 0.5 are associated with class 0. Going back to Equation 16.4 and substituting 0.5 for the output, y gives the equation for the decision boundary:

$$y = \sum_{i=1}^{N+1} x_i w_i = Xw = 0.5 \quad (16.8)$$

For two inputs, this equation becomes

$$w_1 x_1 + w_2 x_2 + w_3 = 0.5 \quad (16.9)$$

Solving for x_2 in terms of x_1 :

$$x_2 = \left(\frac{-w_1}{w_2} \right) x_1 - \frac{0.5 - w_3}{w_2} \quad (16.10)$$

which is a straight line with a slope of $-w_1/w_2$ and a bias of $-(0.5-w_3)/w_2$. These equations are implemented in routine `linear_eval` shown below. This routine includes an optional input that can be used to modify the threshold value to be something other than 0.5. The routine also keeps track of the errors and their type under the assumption that 1.0 represents a “negative” class and 0.0 represents a “positive” class. This terminology will become clear when we discuss classifier performance in Section 16.3.

```
function [sensitivity, specificity] = linear_eval(X, d, w, threshold)
%
% .... extensive comments describing input and output variables.....
%
if nargin < 4
    threshold = .5; % The default decision threshold
end
%
% .... zero counters that count misclassifications.....
%
[r, c] = size(X); % Determine data set size
y = X * w; % Evaluate the output
%
% Plot the results
clf; hold on;
```

```

% Assumes classes are 0 and 1
% with 0 as the positive class and 1 as the negative class
% Evaluates each point for all four possibilities:
% 2 types of correct responses and two types of errors
for i=1:r
    if d(i) > threshold & y(i) > threshold
        % True negative evaluation: plot in gray
        plot(X(i,1),X(i,2),'sqk','MarkerFaceColor',...
              [.8 .8 .8],'LineWidth',1);
        tn=tn+1;                                % True negative
    elseif d(i) > threshold & y(i) <= threshold
        % False positive evaluation: plot in black
        plot(X(i,1),X(i,2),'sqk','MarkerFaceColor','k');
        fp=fp+1;                                % False positive
    elseif d(i) <= threshold & y(i) <= threshold
        % True positive evaluation: plot in light gray
        plot(X(i,1),X(i,2),'ok','MarkerFaceColor','c');
        tp=tp+1;                                % True positive
    elseif d(i) <= threshold & y(i) > threshold
        % False negative evaluation: plot in black
        plot(X(i,1),X(i,2),'ok','MarkerFaceColor','k');
        fn=fn+1;                                % False negative
    end
end
%
V=axis                                % Used to reset axis
%
% Plot decision boundary
x1=[min(X(:,1)),max(X(:,1))];          % Construct x1 to span x1
x2=-w(1)*x1/w(2)+(-w(3)+.5)/w(2);    % Calculate x2 using Eq.16.16
plot(x1,x2,'k','LineWidth',2);          % Plot boundary line
axis(V);                                % Restore original axis
%
% Evaluate sensitivity and specificity
specificity=(tn/(tn+fp))*100;           % Calculate specificity
sensitivity=(tp/(tp+fn))*100;            % Calculate sensitivity

```

Results

The results for the data set generated in Example 16.1 are shown in Figure 16.4a. The least squares algorithm produces the boundary shown in Figure 16.4a (solid line). The boundary separates the two classes without error. Yet, when we examine the region between the two clusters, other boundaries can be drawn that perfectly separate the two clusters and look just as good as, or better than, the one found by least squares (Figure 16.4b, dashed lines). The boundary determined by the least squares linear classifier is based on all the data points in both clusters. If the two data clusters are Gaussianly distributed and are adequately represented by the training set data, then the boundary found by this method will be optimal, at least with respect to the entire data set. However, even if the first assumption is true, the training set usually does not provide adequate information about the cluster centers, particularly if the training set is small. The poor generalization of the least squares method when a small training set is used is illustrated in the next example.

EXAMPLE 16.2

Train the least squares classifier of Example 16.1 on two training sets having the same centers and spread. One training set should be large ($N = 200$) and the other should be small ($N = 20$). Evaluate the results of each training condition with a test set of 500 points also having the same centers and distributions.

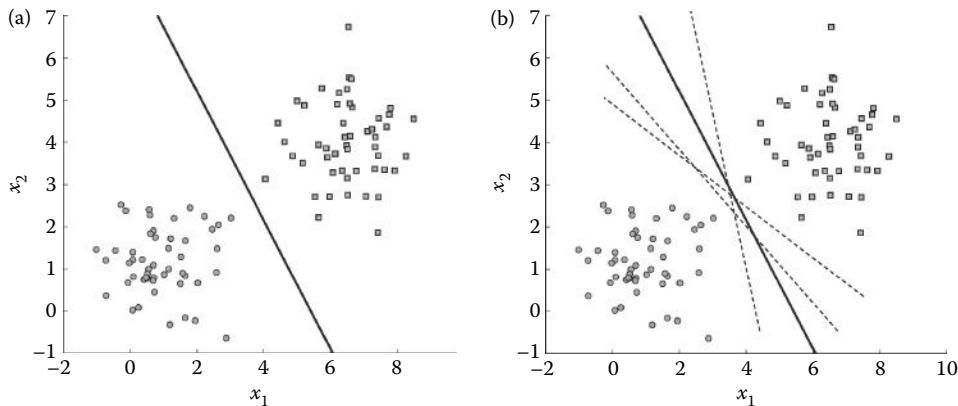


Figure 16.4 (a) Two data classes (circles and squares) classified by a least squares linear classifier. The decision boundary produced by Equation 16.7 (solid line) shows perfect separation between the two classes. (b) Many other decision boundaries can also perfectly separate the two classes (dashed lines). Actually, an infinite number of boundaries can be found. Some of the alternative boundaries look better than those found by the least squares method.

Solution

Modify Example 16.1 by adding a section that uses `gen_data2` to generate a second data set of 500 points having the same properties as the training set. Do not train on this second set, but simply evaluate the classifier's performance by using `linear_eval` applied to the new set of training data. The added code is

```
% Example 16.2 Evaluation of a linear classifier
%
.....same as in Example 16.1 but repeated for two
    different training set sizes.....
[X,d] = gen_data2(distance,[],[],[],500); % Test set
[r,c] = size(X);
X = [X,ones(r,1)];      % To account for bias
linear_eval(X,d,w);    % Evaluate classifier
```

Results

The least squares method of selecting the weights works well if the training set is large and closely reflects the test set data. However, if the training set is small, as is often the case, this rule may not generalize well. Figure 16.5 shows the results from Example 16.2 using a small ($N = 20$) and a large ($N = 200$) training set. The decision boundary found from the small training set performs well on the training set (Figure 16.5a) with one error, but performs poorly on the large test set. Figure 16.5b shows six errors: five class 1 points (squares) misclassified as class 0 (circles) and one class 0 point classified as class 1. If the least squares method is trained on the larger training set, but is evaluated on the same test set, only three classification errors occur: one class 0 (circle) and two class 1 (squares) types are misclassified (Figure 16.5c). This demonstrates the limitation of the least squares method with regard to generalization when a small training set is used.

16.3 Evaluating Classifier Performance

Even if the classifier is optimal, classification errors can occur if the classes overlap. In most medical problems, perfect classification never occurs. There are several methods of quantifying

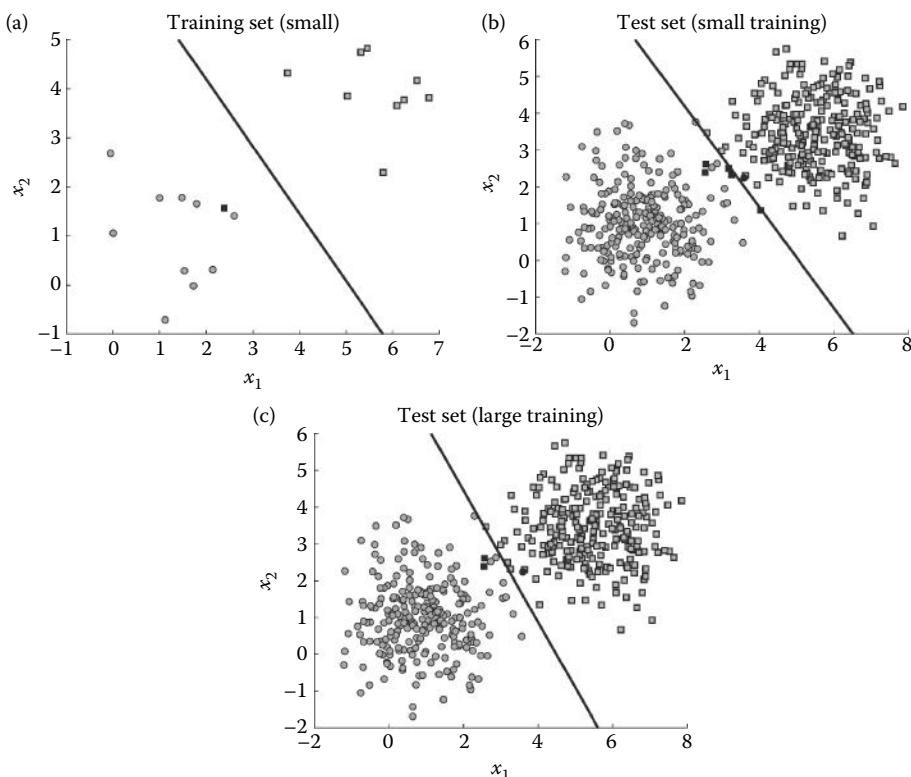


Figure 16.5 The least squares classifier does not generalize well when the training set is small. (a) Training data with the decision boundary ($N=20$) shows good separation with one error. (b) Applying this decision boundary to a test set of 500 points shows six errors. (c) If the classifier is trained on a larger training set ($N=200$), only three errors are made on the same test set. These errors occur due to overlapping points that no linear boundary could separate.

classification errors. The two most popular methods are the *confusion matrix* and measurements of *sensitivity* and *specificity*.

The confusion matrix is a table of correct and incorrect classification, usually given in percentages. Table 16.1 gives an example of the confusion matrix for a three-class problem, that is, one where there are three different categories to be classified.

As seen in Table 16.1, the diagonal indicates the percentage of correctly classified data for each class while the off-diagonals indicate the various combinations of misclassifications. This is useful because some misclassifications may have more serious consequences than others. For example, misclassifying a diseased patient as normal is usually a more serious error than

Table 16.1 Confusion Matrix

True Class	Predicted Class		
	Class 0	Class 1	Class 2
Class 0	% Correct	% Error	% Error
Class 1	% Error	% Correct	% Error
Class 2	% Error	% Error	% Correct

Biosignal and Medical Image Processing

misclassifying a normal patient as diseased. This table can be extended to include situations with any number of classes.

Measures of sensitivity and specificity are used only when two classes are involved; these are very popular in medical classification.* In applying these measures, one class is taken as the unusual class and the other is taken as the nominal class. In medicine, the unusual class would represent disease or other abnormality while the nominal class would represent the disease-free or normal condition. In such circumstances, the classification problem becomes a detection problem with the unusual class termed *positives* and the normal class *negatives*.† Correct detection (i.e., classification) of an abnormal condition is known as a *true positive* while correct classification of a normal condition is termed a *true negative*. Classification errors are either *false positives*, incorrectly classifying normal as abnormal, or *false negatives* where abnormal is classified as normal. Using these definitions, sensitivity and specificity can be defined as

$$\text{Sensitivity} = 100 \frac{\text{True positives}}{\text{True positives} + \text{false negatives}} = 100 \frac{\text{True positives}}{\text{Total positives}} \quad (16.11)$$

$$\text{Specificity} = 100 \frac{\text{True negatives}}{\text{True negatives} + \text{false positives}} = 100 \frac{\text{True negatives}}{\text{Total negatives}} \quad (16.12)$$

As an example, assume that the circles in Figure 16.5 represent patients with a disease and the squares are normals. Given that there are 250 patients in each class, the sensitivity and specificity of the classifier (detector) in Figure 16.5b is

$$\text{Sensitivity} = 100 \frac{\text{True positives}}{\text{True positives} + \text{false negatives}} = 100 \frac{(250 - 1)}{250} = 99.6\%$$

$$\text{Specificity} = 100 \frac{\text{True negatives}}{\text{True negatives} + \text{false positives}} = 100 \frac{(250 - 5)}{250} = 98\%$$

Often, it is possible to vary the decision boundary to increase or decrease the detection of abnormalities. This produces an adjustable trade-off where increasing the detection of true positives increases the number of false positives. The trade-off is also between sensitivity and specificity where increasing sensitivity will decrease specificity. If we assume that in Example 16.2, class 0 represents an abnormal or diseased condition, then having more normals misclassified as diseased is, likely a more desirable situation than the reverse when more diseased patients would be missed. If the threshold was raised, it would eventually be possible to detect all the diseased patients, but more normals would be classified as diseased. By changing the threshold, it is always possible to trade-off between sensitivity and specificity. A curve showing the trade-off between sensitivity and specificity can be helpful in determining where to set the detection threshold, that is, the cutoff for classifying a patient as diseased. The plot of sensitivity versus specificity is called the ROC curve. ROC stands for *receiver operator characteristic*, a term derived from its early application to the detection of transmitted signals (by a receiver such as a radio).

A schematic representation of several ROC curves is shown in Figure 16.6. It is common to plot sensitivity on the vertical axis and 100% specificity on the horizontal axis in reverse order.

* If more than two classes exist, these measures can still be used, but they are applied to pairs of classes.

† In `linear_eval`, class 0 is the positive class and class 1 is the negative class, but these definitions can be easily reversed with no loss of generality.

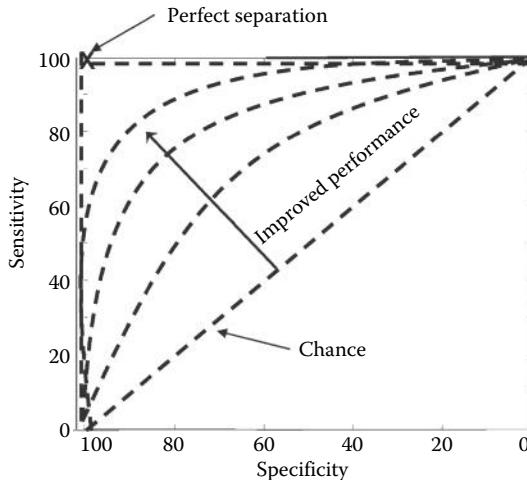


Figure 16.6 Example of several ROC curves. The 1-to-1 line is the sensitivity–specificity trade-off obtained by pure chance: simply by randomly predicting a patient’s condition. The curve of a perfect classifier falls along the vertical and horizontal axes passing through the upper left corner at 100% sensitivity and 100% specificity. Several other intermediate curves are shown.

The ideal curve ascends directly along the vertical axis to pass through the 100% sensitivity and 100% specificity point at the upper left-hand corner. Conversely, the diagonal 1-to-1 line describes the behavior of a classifier based on chance, for example, predicting a patient’s condition based on the throw of a dice. Curves falling near this line show performance just above chance level while curves of better-performing classifiers more closely approach the upper left corner. Curves that fall below the diagonal indicate that the criterion for detection should be reversed: patients below the threshold should be classified as diseased.

An ROC curve can be obtained by varying the threshold that is used to generate the decision boundary and calculating the sensitivity and specificity at each level. This can be done by altering the threshold value input to `linear_eval`. The next example develops an ROC curve for the least squares classifier when a number of points in the data set overlap.

EXAMPLE 16.3

Construct an ROC curve for a least squares linear classifier applied to a data set where the two classes show significant overlap. Set the distance between the centers of the two classes at two standard deviations to ensure a fair number of misclassifications.

Solution

Use a loop to vary the threshold value of `linear_eval` to accept values that vary between 0.05 and 0.9 in increments of 0.05. Save the values of sensitivity and specificity for each threshold. Use the sensitivity and specificity calculated by `linear_eval` and plot sensitivity against 100-specificity (i.e., specificity reversed on the horizontal axis) to generate the ROC curve. Again, `linear_eval` assumes that class 0 is the positive or abnormal class. Raising the threshold will increase the number of diseased patients detected but will also increase the number of normals misclassified as diseased.

```
% Example 16.3 Construction of an ROC curve using the
% least squares linear classifier.
%
```

Biosignal and Medical Image Processing

```

.....same as Example 16.1 except distance=2.....
%
% Evaluate the classifier over a number of different thresholds
for i=1:18
    threshold(i) = .05*i; % Set threshold between .05 and .9
    figure; % Draw new figures
    [sensitivity(i), specificity(i)] = linear_eval(X,d,w,threshold(i));
    title(['Threshold: ',num2str(threshold(i),2)]);
end
figure;
plot(100-specificity, sensitivity); % Plot ROC curve
for k=1:2:18 % Put threshold values on ROC curve
    text(160-specificity(k),sensitivity(k),num2str(threshold(k),2));
end
.....label and axes.....

```

Figure 16.7 shows the data plots at four different threshold values. As the two data clusters are quite close to one another, there are many overlapping data points, making classification a challenge; however, it is not uncommon for medical data to have this kind of overlap. Assuming that circles represent diseased ($N = 50$) and squares normals ($N = 50$), at a low threshold of 0.3, all but one of the normals are correctly diagnosed, but 22 diseased (black circles) are missed. As the threshold is increased to 0.5, the number of missed diseased falls to seven, but now five

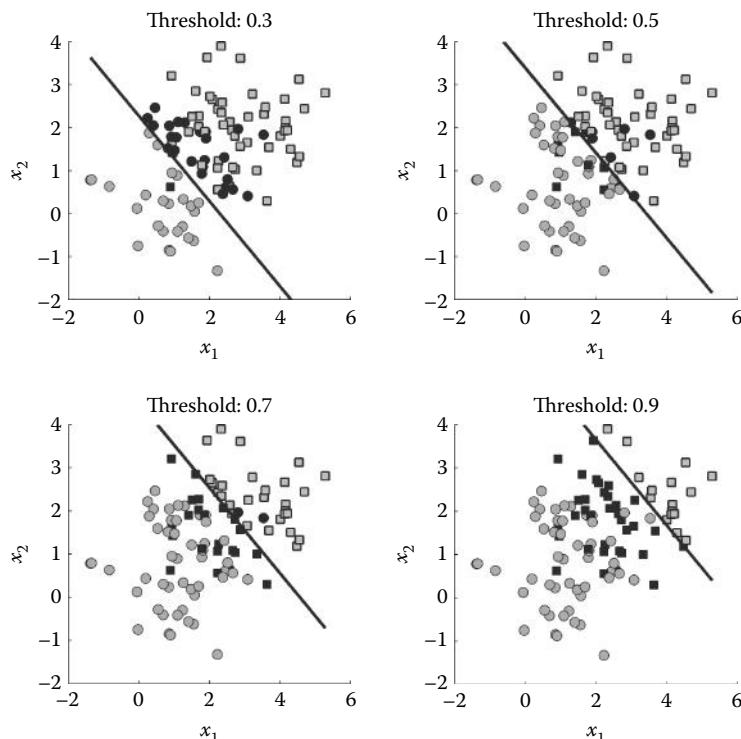


Figure 16.7 Performance of a linear classifier at four threshold levels. Assuming that the circles are diseased and squares are normals, as the threshold increases, the number of incorrectly diagnosed diseased (black circles—false negatives) decreases, but the number of misdiagnosed normals (black squares—false positives) increases.

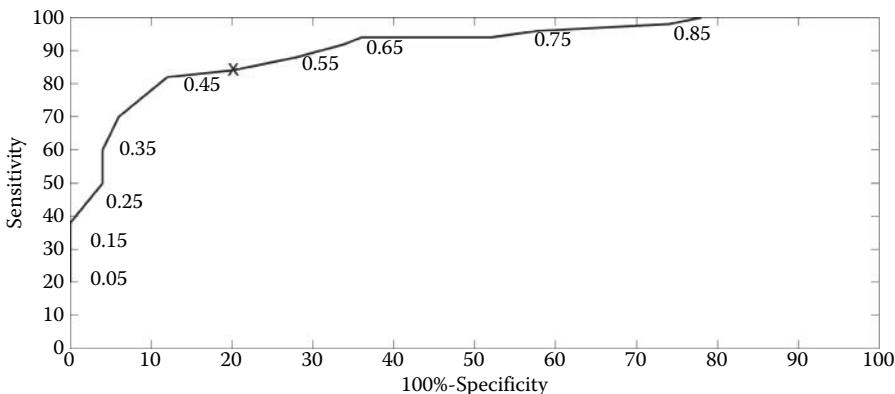


Figure 16.8 ROC curve obtained for a linear classifier by varying the threshold. The original data consist of two closely spaced Gaussian distributions shown in Figure 16.7. The associated thresholds are shown plotted adjacent to the curve. Note: MATLAB cannot plot in reverse order, so specificity is plotted as 100%-specificity.

normal patients are diagnosed as diseased (black squares). Increasing the threshold still further to 0.7 decreases the missed diseased to only two and increases the misdiagnosed normals to 18. At a threshold of 0.9, all diseased are correctly classified, but now 37 normal patients are misdiagnosed as diseased.

Figure 16.8 shows the ROC curve generated from 18 different thresholds used in Example 16.3. Some of the threshold values are shown adjacent to the curve. The threshold value that should be chosen depends on the nature of the diagnosis. One candidate threshold might be 0.5 (the “x” point), which leads to a sensitivity of approximately 84% with a specificity of approximately 80% (100–20%). Relating this performance to a typical medical test, the sensitivity of 84% is reasonable as is the specificity of 80%. An alternative might be to increase the threshold to 0.65, which gives a very good sensitivity of 94%, but the specificity of approximately 70% would lead to many false positives and is likely to be unacceptably high. This is largely due to the fact that the data show a lot of overlap (Figure 16.6), indicating that the measurements that produced the variables x_1 and x_2 were either not very definitive or contained a lot of noise.

16.4 Higher Dimensions: Kernel Machines

Linear classifiers are limited to decision boundaries that are straight lines. Many classification problems involve data that are separable, but not by a straight line (or plane or hyperplane). Figure 16.9a shows two classes (again circles and squares), each of which consists of two clusters diagonally across from one another. They are easily separable, but not by a single straight line. In more complex data sets such as this, it is still possible to separate the classes using a linear boundary if the data are transformed into a higher-dimensional space. The input data that start out, logically, in *input space* are mapped into a higher-dimensional *feature space*. In fact, if the number of dimensions is high enough, you can *always* find a linear boundary (hyperplane) that will separate the data without error.* As an extreme example, if you had one dimension for each data point pattern, you would be able to perfectly separate any data set no matter how complicated or intermixed the classes.

* The fact that there is some dimension where any set of points becomes linearly separable is known as *Cover's theorem*.

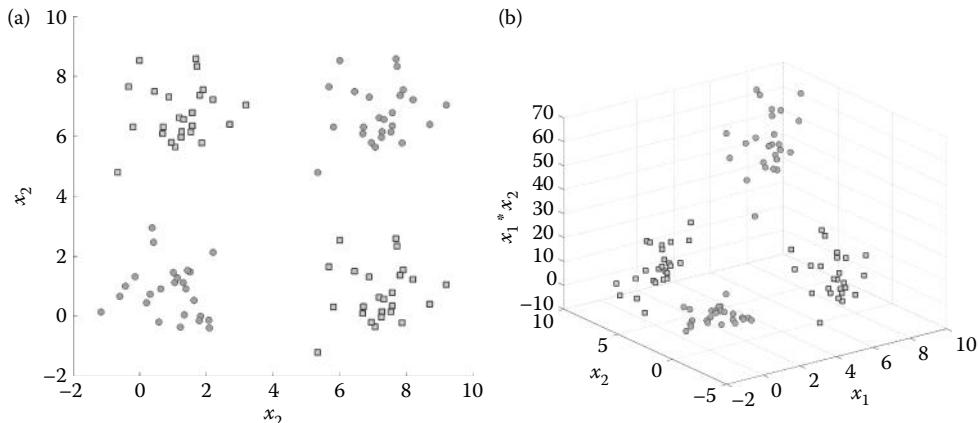


Figure 16.9 (a) A two-class data set that is clearly separable, but not by a single linear boundary. These data will be accurately separated in Example 16.4 using a higher-dimensional feature space. (b) The data in (a) have been transformed into a 3-D feature space by adding a cross-product term. The variable x_3 is just $x_1 * x_2$.

The major drawback of this use of higher dimensions is that it does not generalize well unless you have a large number of data points in the training set. When the training set is limited, as is often the case, the higher-dimensional space becomes very sparse and the optimal boundary becomes hard to define. In addition, training time goes up considerably. The problems associated with using a high-dimensional feature space have been given the colorful term “the curse of dimensionality.” On the other hand, such a mapping means the number of free parameters (the weights and bias) are no longer bound by the number of input variables (i.e., the dimension of the input space), but can be extended to any number of parameters. This is referred to as “decoupling” machine capacity (related to the number of free parameters) from the input space dimension.

Higher-dimensional spaces are created using a kernel function $k(x_i)$ that performs some nonlinear transformation on the original data, x_i , to create a set of new variables. Usually, this new set includes the original variables plus others generated by the function. Popular kernel functions include polynomials, Gaussians, and trigonometric polynomials. The quadratic kernel takes the square of all the original variables and also includes their cross-products. So, for a two-variable data set consisting of x_1 and x_2 , the new data set would consist of x_1 , x_2 , x_1x_2 , x_1^2 , and x_2^2 . The original 2-D input space has been transformed into a five-dimensional feature space. An application of a higher-dimensional feature space to separate the data in Figure 16.9 is shown in the next example.

EXAMPLE 16.4

Classify the data in Figure 16.9a using the least squares linear classifier operating in a higher-dimensional space.

Solution

The quadratic kernel can be used to separate data such as in Figure 16.9a. Since the data are so widely spaced, only the cross-product term, x_1x_2 , is needed to obtain perfect separation. This allows us to implement the classifier in three dimensions, which is easier to demonstrate. The evaluation program, `linear_eval3D`, has been modified to plot 3-D data.

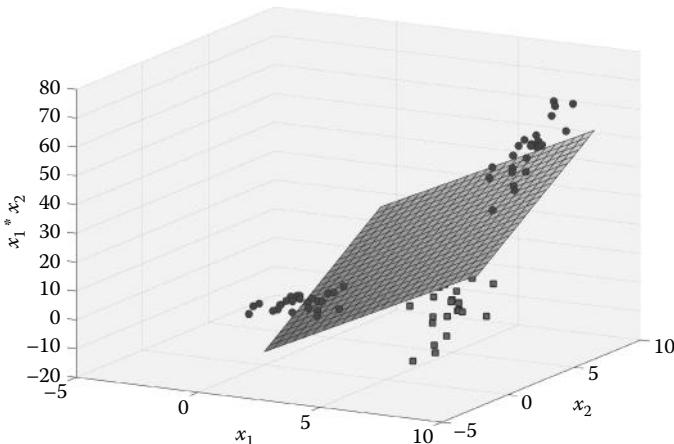


Figure 16.10 A plane gives perfect separation of the transformed data set. The original 2-D data are shown in Figure 16.9a. Note that some of the square data points are hidden behind the decision plane.

```
% Example 16.4 and Figure 16.9
% Program to use of higher dimensions to separate the data set
%     shown in Figure 16.9A
close all; clear all;
distance=6;
[X d]=gen_data2(distance, [], 'd'); % Generate data (2-D)
[r c]=size(X);
X=[X X(:,1).*X(:,2) ones(r,1)]; % Add the cross-product term
.....plot the new data in 3D.....
w=inv(X'*X)*(X'*d'); % Train the classifier
[sensitivity, specificity]=linear_eval3D(X,d,w); % Evaluate
```

Results

Note that the implementation of the equation used to train the weights (Equation 16.7) has the exact same MATLAB code in three dimensions as in the previous 2-D examples, showing that there is no difficulty in extending this approach to more variables. The cross-products are formed simply by adding a column of $X(:,1).*X(:,2)$ in the same line of code that adds 1s to the data matrix. The transformed data are shown in Figure 16.9b. After training, the data in Figure 16.9 can be perfectly separated by a plane as shown in Figure 16.10.

16.5 Support Vector Machines

As mentioned above, the linear methods described above work fairly well if the training sets are large and closely reflect the characteristics of the test data. However, small training sets can lead to errors. Example 16.1 shows that the decision boundary found by the least squares method is not necessarily the best (i.e., not the optimal), and Example 16.2 shows that when the training set is small, performance is also degraded.

The problem with the least squares method and a number of similar methods is that they base the construction of the decision boundary on *all* the data points, which puts too much emphasis on data points that are not critical. For example, in the data shown in Figure 16.11, the points that are closest to the other class are the most important in separating the two classes. An approach that maximizes the distance between these critical data points would likely produce

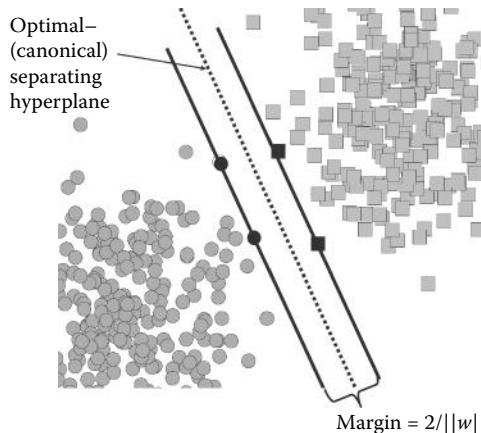


Figure 16.11 Points closest to another class are the most critical in separating the classes and are called the support vectors (solid black points). The derivation of the margin width is given below.

better separation of the test set data. The points closest to the boundary are called the *support vectors* and are shown as black in Figure 16.11. A *support vector classifier* determines the boundary that maximizes the distance between the critical support vectors, the distance labeled “Margin” in Figure 16.11. Since support vector classifiers maximize the *margin*, they are also known as *maximum margin classifiers*. Support vector machine (SVM) classifiers have become quite popular because they produce excellent results in practical problems.

Finding the boundaries that maximize the margins is one key to the SVM classifier and involves a classic optimization process. When the data are linearly separable as in Figure 16.11, the goal is to find the hyperplane that maximizes M , the margin, subject to the constraint that all the data points are on the appropriate side of the boundary.* In SVM analyses, the classes are assumed to be identified as ± 1 since this simplifies the mathematics. The decision boundary is then at $y = 0$; so, using Equation 16.1:

$$y = \sum_{i=1}^N w_i x_i + b = \mathbf{x}_i \mathbf{w} + b = 0 \quad (16.13)$$

where x_i are the input patterns, w is the weight vector, and b is the offset or bias. Since the two classes are defined by $y = \pm 1$, the value of y must be ± 1 at the closest points (i.e., the support vectors) (Figure 16.12). So, the equations for the lines that go through these support vector points must be

$$\mathbf{x}_i \mathbf{w} + b \geq 1 \quad \text{when } y = +1 \quad (16.14)$$

$$\mathbf{x}_i \mathbf{w} + b \leq -1 \quad \text{when } y = -1 \quad (16.15)$$

This can be combined into a single equation:

$$y_i(\mathbf{x}_i \mathbf{w} + b) \geq 1 \quad (16.16)$$

* The hyperplane that best separates the support vectors with the greatest margin is sometimes termed the *optimal conical separating hyperplane* (OCSH) or just the optimal separating hyperplane.

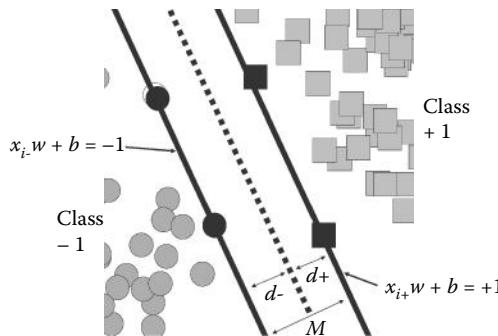


Figure 16.12 Two lines passing through the support vectors mark the boundary for $y \geq \pm 1$ and this determines the equation for these lines.

This equation simply states that \mathbf{w} and b should be such that the two classes fall on the appropriate side of the support vector lines.

To determine the equation for the margin, M , note that the distance of any hyperplane, $\mathbf{x}_i \mathbf{w} + b = 0$, to the origin is $-b/\|\mathbf{w}\|$ where $\|\mathbf{w}\|$ is the norm of \mathbf{w} and is equal to $\sqrt{w_1^2 + w_2^2 + w_3^2 + \dots + w_n^2}$ or, in matrix notation, $\sqrt{\mathbf{w}^T \mathbf{w}}$. If the hyperplane is equal to ± 1 , then the distance to the origin is just $(\pm 1 - b)/\|\mathbf{w}\|$. This can be used to find the equation for the distance between the two lines defined in Equations 16.16 and 16.15, which is equal to the size of the margin. For the line separating class 1 (i.e., $y_i \geq 1$), the distance to the origin is

$$d_o = \frac{(1 - b)}{\|\mathbf{w}\|} \quad (16.17)$$

And for the line separating class -1 (i.e., $y_i \leq -1$), the distance to the origin is

$$d_o = \frac{(-1 - b)}{\|\mathbf{w}\|} \quad (16.18)$$

The difference between the two lines is obtained by subtracting Equation 16.17 from Equation 16.18:

$$M = \frac{(1 - b)}{\|\mathbf{w}\|} - \frac{(-1 - b)}{\|\mathbf{w}\|} = \frac{2}{\|\mathbf{w}\|} \quad (16.19)$$

So, the maximum margin is obtained by minimizing $\|\mathbf{w}\|$ that equals $\sqrt{\mathbf{w}^T \mathbf{w}}$. The margins can also be maximized by minimizing $\|\mathbf{w}\|^2$ that is equal to $\mathbf{w}^T \mathbf{w}$ and is somewhat easier to perform. The minimization must be done subject to the constraint imposed by Equation 16.16, which ensures that the boundaries are on the correct side. This type of minimization problem is known as a *quadratic programming* (QP) optimization problem and there are a number of routines available to solve this. The associated files contain one such optimization routine developed by Alex Smola and is used in conjunction with a set of SVM MATLAB routines developed by Steve Gunn at the University of Southampton, United Kingdom. These routines are described in the next section and provide graphical display as well as SVM classification. They are used in the problems and examples given here.

If the data are not linearly separable and the points overlap, the optimization process still maximizes M , but the constraint is relaxed so that some of the points can be on the wrong side of the boundary. This should still lead to a better boundary than found using other linear classifiers.

Biosignal and Medical Image Processing

The technique described so far can only be used to produce linear boundaries and is referred to as *linear support vector machines* (LSVM). This is effective if the data are linearly separable. If not, there are two alternatives: use a linear boundary and simply accept some error, or transform the data into higher dimensions as described previously. The SVM approach can then be applied to the transformed data to generate a hyperplane boundary that effectively provides a nonlinear boundary as shown in Example 16.4.

In more general SVMs, the support vector classifier is combined with the use of higher dimensions so that complex, nonlinear boundaries can be obtained. Various kernels are used to transform the input space into a feature space that is symmetric, which greatly simplifies the mathematics. (In fact, the boundary can actually be evaluated in the lower-dimensional input space). Because of this simplification and the improved generality of a maximum margin classifier, the enlarged space can get very large while avoiding the typical problems associated with the curse of dimensionality. Nonetheless, transforming the data into higher dimensions can still result in overfitting or overtraining, leading to poor generalization as well as extensive computational time. Again, it is a question of matching machine capacity, in this case, reflected by the feature space dimension, with the requirements of the data set.

16.5.1 MATLAB Implementation

The routines used to implement the SVM classifier are found in the associated files and have been developed by Steve Gunn at the University of Southampton, United Kingdom. This package uses an OP optimization routine developed by Alex Smola. The basic routine for determining training an SVM is

```
[nsv alpha bias] = svc(X,d,ker,C)
```

where X is the training set input pattern and d is the correct classification. The optional argument ker is the kernel used to transform the data into higher dimensions. The default value is 'linear' in which case the routine applies the support vector approach to the untransformed data. Other kernels include 'poly', which transforms the data using a polynomial, 'spline', which applies a spline function, and 'bspline', which implements a higher-order spline function. For a complete list of possible kernels, see the help for svkernal.

All the kernels except 'linear' and 'spline' take one or two additional arguments. For example, 'poly' requires an argument to specify the order of the polynomial. The additional arguments are passed through two global parameters, p1 and p2. The p1 parameter is the only parameter needed for most of the kernels and is used to specify the order of the polynomial for 'poly' and the degree of the spline for 'bspline'. The last input argument, C, is optional and is used to indicate how the classifier is to behave in situations where the data overlap. In general, the default value is appropriate.

The outputs alpha and bias are the classifier's free parameters and as such, define the classifier. The first output argument, nsv, indicates the number of support vectors that are found and used to determine the boundaries. The bias variable is simply the bias of the hyperplane, and alpha defines the weights in terms of La Grange multipliers. These arguments are used by other routines to classify the test set data and plot the result.

The routine `svmplot` is used to plot the training set data. The calling format showing only the necessary arguments is

```
svcplot(X,d,ker,alpha,bias)
```

where the input arguments are the same as described above and include the output of the `svc` routine. This routine has a number of additional arguments that control the plot format and these are described in the related help. Example 16.5 applies these two routines to a training set of linearly separable data.

16.5 Support Vector Machines

Once the classifiers free parameters, alpha and bias, are determined, the classifier can be applied to the test set data using

```
y = svcoutput(Xt, dt', X, ker, alpha, b); % Apply classifier
```

where the input parameters are as defined above: Xt is the training set and X is the test set, dt defines the training set classes, and ker is the kernel specifying the type of nonlinearity, which along with alpha and b defines the classifier. The output, y, is the classifier's finding and is compared with a threshold to determine the class.

The data can be plotted and the sensitivity and specificity can be determined using the routine plot _ results. This is a modification of the program linear _ eval:

```
[Sensitivity, Specificity] = plot_results(X, d, y, thresh);
```

where the inputs specify the test set, correct test set classes, classifier output, and the classifier threshold. For SVM classifiers, this should be 0. The outputs are sensitivity and specificity as defined previously.

Finally, classifier boundaries can be drawn using

```
svcbound(Xt, dt', ker, alpha, b); % Plot boundaries
```

where Xt and dt are the training set data and class definitions, and again the last three inputs specify the classifier using free parameters alpha and b, and ker. Hence, working with the SVM classifier is merely a matter of stringing the appropriate routines together.

EXAMPLE 16.5

Use the linear SVM to classify a linearly separable data set. Use gen _ data2 to produce a training set ($N = 100$) having two classes identified as class -1 and class +1, which are Gaussianly distributed and separated by 5 standard deviations.

```
%Example 16.5 Use the SVM routines to classify a  
% linearly separable training set  
%  
[X, d] = gen_data2(5, 45, [], [-1 1]); % Generate data  
[nsv, L_alpha, b] = svc(X, d', 'linear'); % Train LSVM  
svcplot(X, d', 'linear', alpha, b); % Plot results  
.....labels.....
```

Results

The results from Example 16.5 are shown in Figure 16.13 below. The dashed lines are defined by Equations 16.16 and 16.15. Note that the decision boundary (solid line) appears well placed with respect to points closest to the other class, the support vectors (filled black points).

The next example applies the SVM to a training set, then evaluates classifier performance using a test set having the same statistical properties as the training set. In these data sets, the centers of the two Gaussian distributions are close enough so that some of the data overlap.

EXAMPLE 16.6

Use the LSVM software package to classify data that have some overlap. Train this classifier on a training set of 100 points and evaluate on a test set of 400 points.

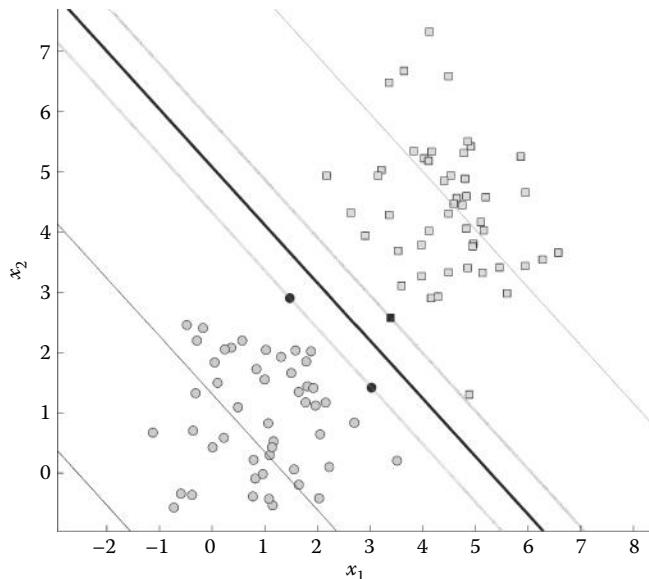


Figure 16.13 The SVM classifier applied to a training set of linearly separable data. The support vectors are shown as filled black points.

Solution

Use `gen_data2` to produce a training set having two classes ($N = 100$) identified as class -1 and class $+1$, which are Gaussianly distributed and separated by 3 standard deviations. Also generate a test set ($N = 400$) having the same properties. Use `svc` with the '`linear`' option to construct the LSVM classifier and `svcpplot` to plot the results. Apply the classifier to the test data using `svcoutput`, plot the data using `plot_results`, and plot the boundaries using `svcbound`.

```
% Example 16.6 Example of using the Linear Support Vector Machine
% on a data set that is not completely separable
%
clear all; close all;
[Xt,dt] = gen_data2(3,45,[],[-1 1]); % Generate training set
[X,d] = gen_data2(3,45,[],[-1 1],400); % Generate test data
[nsv,alpha,b] = svc(Xt,dt','linear'); % Train LSVM
svcpplot(Xt,dt','linear',alpha,b); % Plot test results
.....labels and new figure.....
% Plot test set results
y=svcoutput(Xt,dt',X,'linear',alpha,b); % Apply classifier
[Sensitivity, Specificity]=plot_results(X,d,y,0); % Plot data
svcbound(Xt,dt','linear',alpha,b); % Plot boundaries
```

Results

The training set data and boundaries found by `svc` are plotted by `svcpplot` as shown in Figure 16.14. The classification of the test set is shown in Figure 16.15. The sensitivity is 91% and the specificity is 96%.

Note that while the classification has errors, the boundaries shown in Figure 16.15 appear to be reasonable and probably are the best that can be achieved given the overlapping data points. The boundary is also constrained to be linear but this is really not a limitation given that the data are Gaussianly distributed. With such distributions, a linear boundary is optimal.

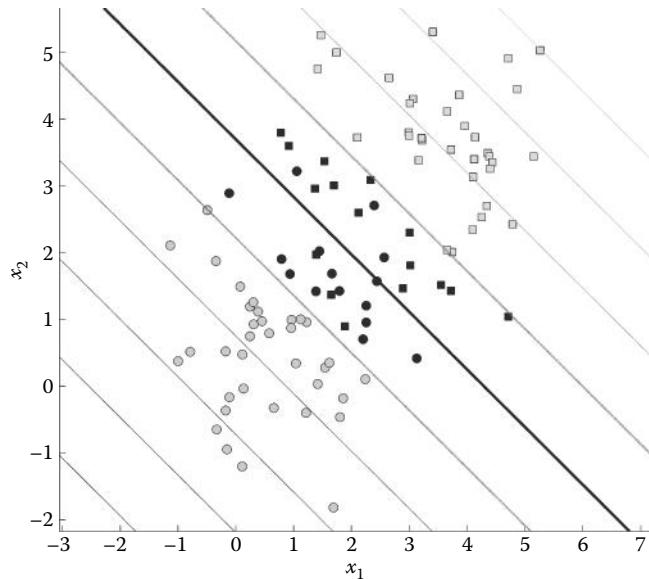


Figure 16.14 Classification of the training set used in Example 16.6. This data set is not completely separable. The classification errors and support vectors are shown as solid black.

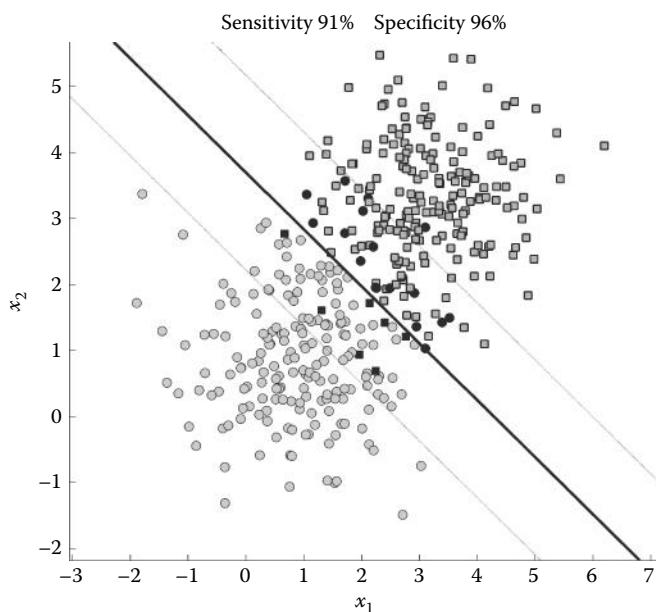


Figure 16.15 Classification of the test set ($N=400$) in Example 16.6. Classification errors are shown in black. The sensitivity and specificity for this data set are 91% and 96%, respectively.

Biosignal and Medical Image Processing

The following example shows the SVM classifier applied to a nonlinear data set. In this example, one class is surrounded on three sides by the other class (the 'c' option in gen_data2).

EXAMPLE 16.7

Apply the SVM classifier to a nonlinear data set that requires a nonlinear boundary for the best separation. Use a training set of 100 points, then evaluate the classifier on a test set of 400 points.

Solution

The distribution of the two classes where one class is surrounded on three sides by the other class suggests that a polynomial boundary would be appropriate. To avoid overfitting, it is better to keep the boundary as simple as possible; so, a second-order boundary is used. Other than the addition of a global parameter ($p1 = 2$) and the options used for gen_data2, svc, svcplot, and svm_boundaries, the code is the same as that used in the last example.

```
% Example 16.7 Example using the Support Vector Machine
% on a data set that is nonlinearly separable.
% A 2th order polynomial kernel is used.
%
global p1; % Kernel parameter
p1=2; % Kernel order
ker='poly'; % Kernel type
[Xt,dt]=gen_data2(4,45,'c',[-1 1]); % Generate training set
[X,d]=gen_data2(4,45,'c',[-1 1],400); % Generate test set
[nsv,alpha,b]=svc(Xt,dt',ker); % Train SVM
svcplot(Xt,dt','poly',alpha,b); % Plot training results
.....labels and new figure.....
%
% Evaluate and plot test set results
y=svcoutput(Xt,dt',X,ker,alpha,b); % Apply classifier
[Sensitivity, Specificity]=plot_results(X,d,y,0); % Plot data
svcbound(Xt,dt',ker,alpha,b); % Plot boundaries
```

Results

Figure 16.16 shows that the boundary produced during training is able to separate the two classes of the training set although there are a number of errors due to the overlap of points. When applied to the test set ($N = 400$) (Figure 16.17), this classifier produces a fair number of errors; however, the performance is still reasonable with a sensitivity of 97% and a specificity of 96%. Most medical tests have lower sensitivities and/or specificities.

16.6 Machine Capacity: Overfitting or “Less Is More”

With the availability of several MATLAB-based classifiers on the web, the user is freed from the task of developing the necessary software. MATLAB routines can be found for SVM classifiers (such as the one used here), ANN classifiers, cluster analysis classifiers, and a number of other approaches. (ANNs are described in the next chapter and cluster analysis is discussed at the end of this chapter.) The problem for the user is to determine the classifier complexity required for a specific classification problem. In many cases, a linear classifier will perform better on the test set than a more complicated classifier, as shown in Figure 16.3. In classifier parlance, the problem is to fit the machine capacity to the complexity of the data set. Recall that machine capacity is related to the complexity of the boundary. For example, an SVM that uses a polynomial boundary would have greater machine complexity than one with a linear

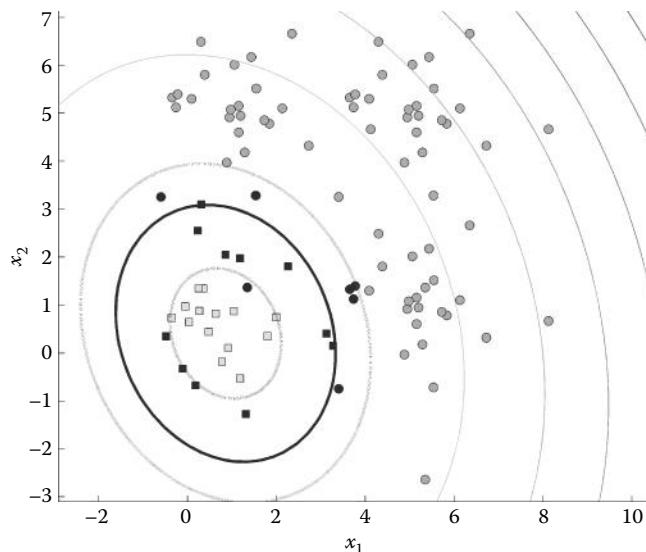


Figure 16.16 Applying the SVM classifier with a fourth-order polynomial kernel to a test set where one class is surrounded on three sides by the other class. This classifier provides good separation on the training set ($N = 100$) although a number of errors are seen (black squares or circles).

boundary. Moreover, the higher the order of the polynomial, the greater the machine capacity and boundary complexity.

An example of overfitting can be seen by rerunning Example 16.7 with a higher-order polynomial. Figure 16.18 shows the test set results when the polynomial order is increased to 6. Since machine capacity is higher, the decision boundary is more complicated. The specificity

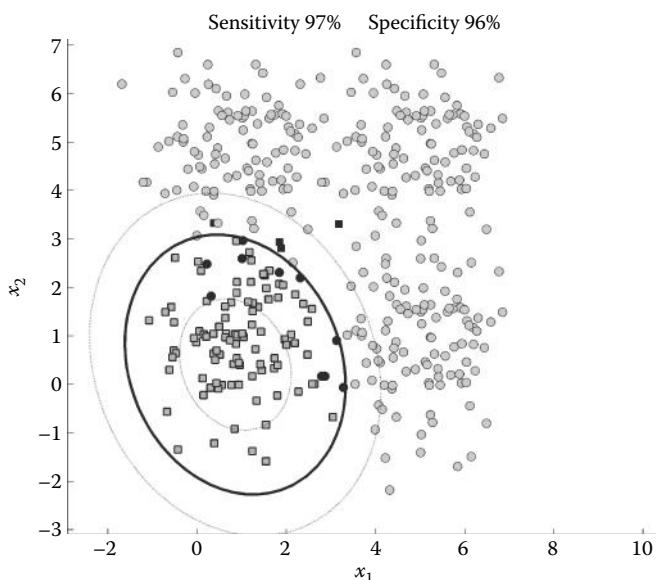


Figure 16.17 Applying the SVM classifier to a larger test set ($N=400$) shows a fair number of errors due to overlapping points, but the overall sensitivity and specificity are reasonable.

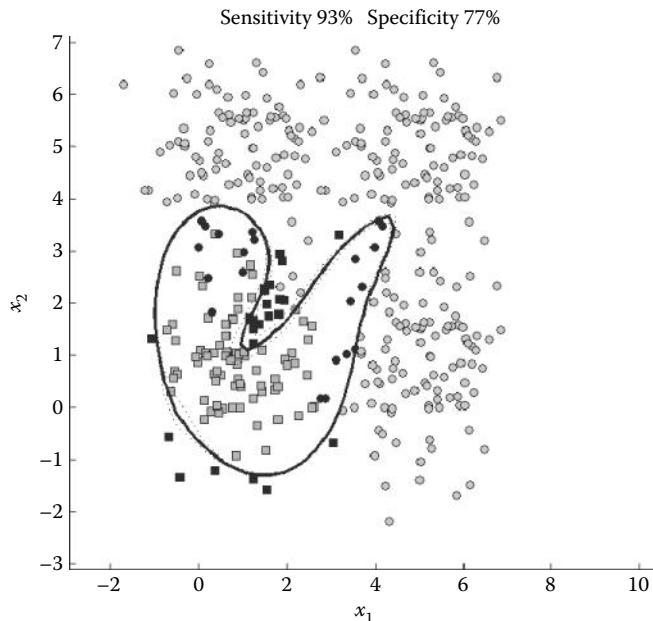


Figure 16.18 The SVM classifier used in Example 16.7 evaluated on a test set having the same statistical properties. This classifier has a greater machine capacity achieved by using a higher-order polynomial. The sensitivity and specificity of this more complex classifier has dropped from 97% and 96% to 93% and 77%.

dropped from 96% to 77% while the sensitivity of the more complicated classifier experienced a lesser drop of 97% to 93%. In this classifier, the complexity has been used to overtrain on a few points in the training set that were not representative of the overall data set. Because of the increased complexity, this SVM does not generalize as well as the one used in Example 16.7 with less machine capacity. This issue of overfitting is further explored in the problems.

16.7 Extending the Number of Variables and Classes

Although the examples have been limited to two input variables to simplify the plots, these approaches are easily extended to more variables. In fact, most of the routines can be used with as many input variables as desired without modification. Only the plotting functions need to be changed, and some of those do handle three-variable inputs. For example, `net_eval` will plot three-variable data including the boundary if it is linear. Classification problems with three variables are given in the problems.

All the examples include only two classes; however, they can all be extended to more than two classes using the same classifier algorithms and training approaches. To train, we just compare each class against all other classes. Training is used to construct a unique boundary for each class. Training multiclass classifiers is simplified by identifying class membership using the scheme given in Equation 16.3. Each column specifies the identity for one class; all the other classes are given zeros. Hence, if the appropriate column is selected, a given class will be trained to generate a boundary that includes only its members and excludes all other classes. The next example applies the SVM classifier to separate four classes.

EXAMPLE 16.8

Use the SVM classifier to separate a two-variable data set consisting of four classes.

Solution

To generate the data set, use `gen_data4`, which is similar to `gen_data2` but generates four classes. The correct classes are indicated in the matrix `D` using the format described in Equation 16.3. To separate the four classes, just apply the `svc` routine three times, selecting the correct answer vector `d` for each class. The fourth class is assumed to be those points not in the other three classes. Since the classes are unlikely to be linearly separable, we try SVM with a second-order polynomial kernel.

```
% Example 16.8 Example of using the Support Vector Machine software
% on a data set with four classes.
%
clear all; close all;
global p1 p2;
p1=2;                                % Kernel order 2
ker='poly';                            % Kernel type
[Xt,Dt]=gen_data4(2.75,200,[-1 1]);    % Training set
%
for i=1:3
    d=Dt(:,i);                      % Select appropriate correct class
    [nsv,alpha,b]=svc(Xt,d,ker);      % Generate classifier
    svcplot4(Xt,d,ker,alpha,b,i);     % Evaluate classifier and plot
end
for k=1:200                           % Plot remaining data points
    if Dt(k,4) == 1
        plot(Xt(k,1),Xt(k,2),'kv');   % Options not shown
    end
end
.....labels.....
```

The results produced by this example are shown in Figure 16.19. The plot routine `svcplot4` produces an output similar to the other plotting routine `svcplot`, except that errors are not plotted in black. In addition, `svcplot4` does not clear the figure prior to plotting to allow overplotting the results from the other three SVMs. Finally, the margins are not plotted to simplify the plot, and the different classes and boundaries are plotted using different shapes. Since the classes are widely separated, the SVM classifier with a second-order polynomial kernel has no difficulty classifying the training set. Evaluation of the test set is performed in the problems. Note that with this approach, some of the boundaries overlap. In cases where test set points fall within these overlapping areas, an additional criterion may be required. The possible criteria include distance to the center of mass of the competing classes or class of the nearest neighbor. Alternatively, such points could be perceived as in a unique class.

Most biomedical engineering problems are concerned with only two classes: normal and diseased (or abnormal). When multiple classes are involved, the cluster analysis techniques described in the next section are ideal as they lend themselves naturally to any number of classes.

16.8 Cluster Analysis

Cluster analysis is particularly popular for unsupervised classification, especially when the number of classes is unknown. Here, only two *supervised* versions of cluster analysis are described: *k-nearest neighbor classifiers* and *k-means clustering*.

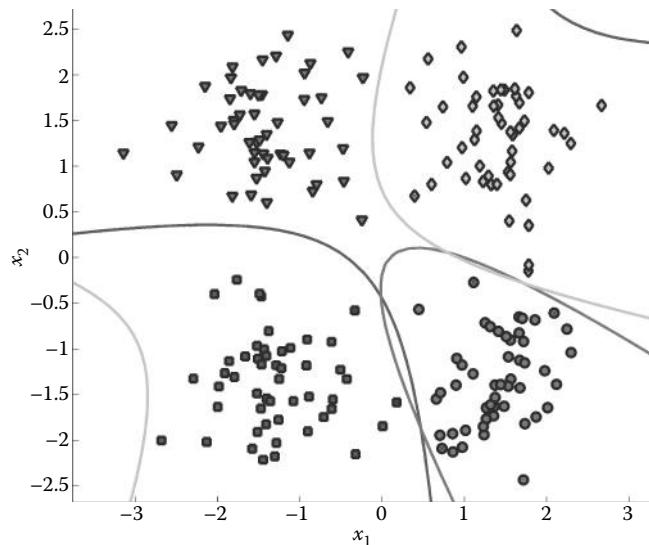


Figure 16.19 Separation of a four-class training set produced by applying an SVM classifier with a second-order kernel. As the four classes are widely separated, the SVM classifier can separate the classes without error.

16.8.1 *k*-Nearest Neighbor Classifier

The *k*-nearest neighbor classifier is very simple, yet has been quite successful in many real-world problems such as identifying ECG patterns and handwritten numbers. This classifier operates to classify test set data directly off the training set and does not need prior training. For this classifier, if the training set classes are identified as 0 and 1, or higher if there are more than two classes. The classifier simply takes each test set point and determines the distance to the *k*-nearest training set points, where *k* is a constant. It then takes the average of the class values of these nearest points and assigns the test set point to the majority class. In other words, the average class value of the *k*-nearest points is rounded to an integer to get the class value of the test set point. For example, if *k* = 5 and the five training set points that are closest to the test set point have class values of 0, 1, 1, 0, and 0, then the average is 2/5 and, since this is <0.5, class 0 is assigned to that test point. If the five nearest training set points have values of 0, 1, 1, 1, and 0, then the average is 3/5 and class 1 is assigned. This approach can be used for any number of classes and any number of input variables.

Distances between points can be measured using a variety of metrics but the most common and straightforward is the Euclidean distance. The Euclidean distance between two points is

$$dist = \sqrt{\mathbf{x}_2 - \mathbf{x}_1} = \|\mathbf{x}_2 - \mathbf{x}_1\| \quad (16.20)$$

where \mathbf{x}_1 and \mathbf{x}_2 are vectors and $\|\mathbf{x}_2 - \mathbf{x}_1\|$ is the norm of the vector that results after subtraction. It is also common to normalize each variable so that it has a variance between ± 1 . This way, the distance measurements are not skewed by a variable that extends over a much wider range than the other variables. An example of the *k*-nearest neighbor approach is shown in Example 16.9.

EXAMPLE 16.9

Use the *k*-nearest neighbor approach to classify a two-class test set of 500 points using a training set of 100 points. Use a *k* of 5 and also of 15. The training and test set are in file Ex16_9.mat and

are nonlinearly separable with some overlap. Both data sets have been normalized by setting their variances equal to ± 1 . Calculate the confusion matrix and plot the boundaries.

Solution

For each test set input pattern, calculate the distances to all the training points using the MATLAB norm routine. Store these distances, along with their associated class values, in a matrix. Then sort the matrix for increasing distance values using the MATLAB sortrows routine. Take the mean of the first k -class values and round to an integer. This integer is the class assigned to the test data point.

```
% Example 16.9 Example of k-nearest neighbors classification
% Two nonlinearly separable classes. k=5 and 15
%
clear all; close all;
K=5;                                % Number of nearest points
nu_classes=2;                          % Number of classes
load Ex16_9.mat;                      % Load data
[r,c]=size(X);                        % Test set size
[rt,c]=size(Xt);                      % Training set size
%
% Find nearest neighbors
class=zeros(r,1);                    % Class assignment vector
for j=1:r
    for i=1:rt                         % Find all distances
        Distance(i,1)=norm(X(j,:)-Xt(i,:));
        Distance(i,2)=dt(i);             % Save class value
    end
    Distance=sortrows(Distance,1);      % First row ascending
    majority=mean(Distance(1:K,2));     % Distance to K closest pts
    class(j)=round(majority);           % Assign class
end
%
% Evaluation begins here
% Construct confusion matrix and plot
hold on;
confusion=zeros(nu_classes);
correct=0; incorrect=0;
for i=1:r                             % Build confusion matrix
    if class(i) == d(i)                % and plot
        plot(X(i,1),X(i,2));          % Options not shown
        correct=correct+1;
        confusion(d(i)+1,d(i)+1)=confusion(d(i)+1,d(i)+1)+1;
    else
        plot(X(i,1),X(i,2));          % Options not shown
        incorrect=incorrect+1;
        confusion(d(i)+1,class(i)+1)=...;
        confusion(d(i)+1,class(i)+1)+1;
    end
end
k_nearest_boundaries(X,K,Xt,dt);       % Plot boundaries
.....label and axis.....
disp(confusion);                       % Confusion matrix
```

The boundaries are plotted using the routine `k_nearest_boundaries` that works the same as the other boundary programs. Test points are evaluated over a grid (100×100 points) that spans the data range, and the class values obtained for this grid are plotted with MATLAB's

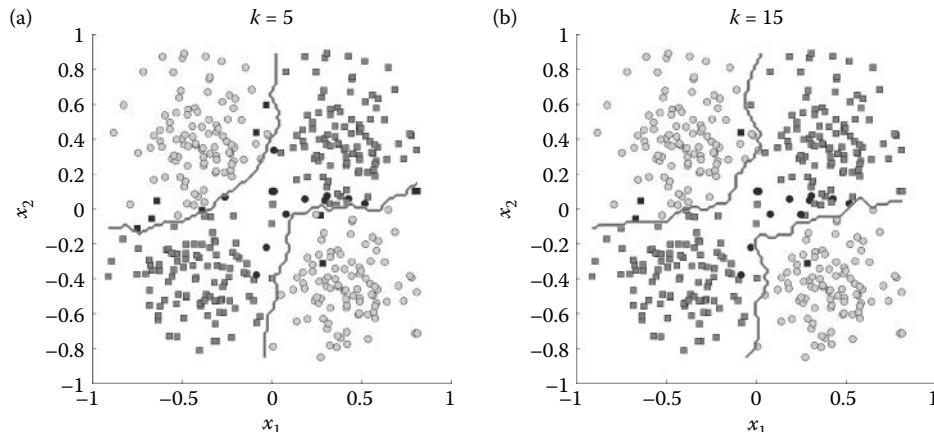


Figure 16.20 Boundaries obtained using the k -nearest neighbor classifier applied to a nonlinear, overlapping data set. (a) $k = 5$. (b) $k = 15$. Note that the higher k value produces a more complicated boundary.

contour routine. This approach can be used with any number of classes. The results for the two values of k are shown in Figure 16.20.

The results from the confusion matrix are given in Table 16.2. As can be seen, the performance is slightly better when $k = 15$. A larger value of k means that more of the data are being considered, which improves generality, but if k is too large, boundary points can be misclassified if they are close to a large group of the other class. On the other hand, smaller values of k can lead to misclassification due to a small number of outliers in the other class. The behavior of this classifier over a range of k values is explored in the problems.

16.8.2 k -Means Clustering Classifier

The k -means clustering is a related classification method that represents the training data with a number of data centers known as *prototypes*. In this approach, the k stands for something quite different: the number of prototype centers. Once these prototype centers are established, the test data are assigned to the class of the closest prototype. Thus, the position of the prototypes determines the boundary, and the number of prototypes chosen to represent each class determines the complexity of the boundary: the larger the value of k , the more complicated the boundary. The value of k is directly related to machine capacity. The number of prototypes is selected by the user and the prototype centers are positioned during a training period. There are several different methods for training the prototypes to find the best location to represent the data. The method described here is known as the *learning vector quantization* (LVQ) method and is straightforward and fast.

Table 16.2 Confusion Matrix from Example 16.9

$k = 5$			$k = 15$		
True Class	Predicted Class		True Class	Predicted Class	
	Class 0	Class 1		Class 0	Class 1
Class 0	190	10	Class 0	196	4
Class 1	12	188	Class 1	11	189

In the LVQ method, the initial prototypes are placed randomly within each class. During training, these prototypes are moved to more ideal locations. A random training set point is selected and the closest prototype is found. If that prototype is of the same class as the training point, the prototype is moved toward the training point. If not, the prototype is moved away from the training point. The amount of movement is proportional to the distance and the proportional constant is a *learning rate* constant. Once all the training points have been used, the procedure begins again with a smaller learning rate constant. This continues until the learning rate constant is zero. The application of the k -means clustering method using the LVQ training method is given in Example 16.10.

EXAMPLE 16.10

Use the k -means clustering approach to classify the test set used in Example 16.9 and found in Ex16_9.mat. Use eight prototype centers to represent each of the two classes, that is, $k = 5$. After training, apply the classifier to the test set and again calculate the confusion matrix and plot the boundaries.

Solution

The eight initial prototype locations (per class) are selected randomly. Since the positions of the data points are random and they are in random order, the initial clusters can be chosen using the first 16 training data points.* Since gen_data2 alternates the classes, eight points are selected in each class. The locations are then moved to more representative locations in routine cluster_learn, which is shown below. The data points are plotted and the confusion matrix is constructed in routine cluster_eval. The boundaries are drawn in routine cluster_boundaries.

```
% Example 16.10 Example of classification using k-means
% cluster analysis. Uses the same data as in Example 16.9.
%
clear all; close all;
K=8;                                % Clusters per class
nu_classes=2;                          % Number of classes
alpha=0.25;                            % Learning constant
nu_training=1000;                      % Number of training cycles
load Ex16_9.mat;                       % Load data
[r,c]=size(Xt);
%
% Initialize cluster centers. Pick location of first prototypes
for i=1:nu_classes*K
    Proto(i,:)= [Xt(i,:),dt(i)];      % Include prototype class
end
%
% Train by moving prototype centers
for i=1:nu_training
    [Xt,dt]=mix_data(Xt,dt);        % Randomize input sequence
    Proto=cluster_learn(Xt,dt,Proto,alpha);    % Train clusters
    alpha=alpha*(1 - i/nu_training);   % Reduce learning constant
end
%
% Now evaluate on test set of 400 points
figure;
confusion=cluster_eval(X,d,Proto);     % Classify and plot
```

* This is the number of prototypes per class times the number of classes, that is, 8×2 .

Biosignal and Medical Image Processing

```
cluster_boundaries(X,d,Proto); % Draw cluster boundaries
disp(confusion)
```

The routine `cluster_learn` uses the strategy outlined previously to move the prototypes to better positions. The training points are selected in sequence, but they are randomized before each pass by the routine `mix_data`, which randomly rearranges the training matrix while keeping the correct associated classifications.

```
function Proto=cluster_learn(X,d,Proto,alpha)
% Function to use Learning Vector Quantization to find
% the best cluster centers
%
[r,c]=size(X); % Testing set size
[r1,c1]=size(Proto); % Number prototype centers
% Train centers. Pick data training points sequentially
for i1=1:r % Use all the training data points
    for j1=1:r1 % Distances to prototypes centers
        distance(j1)=norm(Proto(j1,1:2)-X(i1,:));
    end
    [dum,i_close]=min(distance); % Closest prototype
    if d(i1) == Proto(i_close,3) % Test class
        % Move prototype center closer to this training point
        Proto(i_close,1:2)=Proto(i_close,1:2)...
            +alpha*(X(i1,:)-Proto(i_close,1:2));
    else
        % Move prototype center farther away from this point
        Proto(i_close,1:2)=Proto(i_close,1:2)...
            -alpha*(X(i1,:)-Proto(i_close,1:2));
    end
end
```

The evaluation routine `cluster_eval` classifies the data by searching for the closest prototype to each of the test data points. The test set data are then classified as belonging to the same class as the prototype. This routine also plots the test set data, flagging the errors by filling the plotted points in black. Finally, the routine plots the final location of the prototypes as larger symbols marked with an “x.”

```
function [confusion]=cluster_eval(X,d,Proto);
% Evaluates clusters defined by Proto
[r,c]=size(X); % Test set size
[r1,c1]=size(Proto); % Number prototypes
nu_classes=max(d)+1; % Number of classes
%
% Classify test set data
for i=1:r
    for i1=1:r1 % Find closest prototype
        distance(i1)=norm(Proto(i1,1:2)-X(i,:));
    end
    [dum,i_close]=min(distance);
    y(i)=Proto(i_close,3); % Predicted class
end
% Plots for results
clf; hold on;
confusion=zeros(nu_classes);
% Plot and build confusion and plot
for i=1:r
    if y(i) == d(i)
```

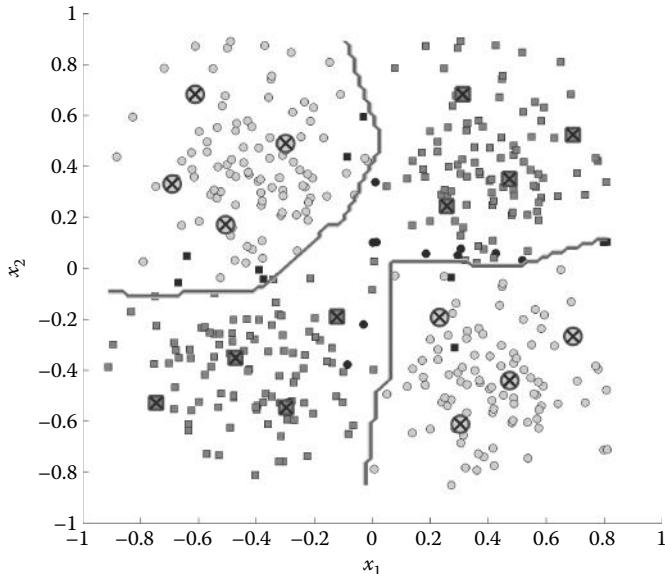


Figure 16.21 Boundaries determined by the k -means clustering classifier using the same test set as shown in Figure 16.20. The large symbols with the “x” are the prototypes. Since $k = 8$, there are eight prototypes in each class.

```

plot(X(i,1),X(i,2)); % Plot options omitted
confusion(d(i) + 1,d(i) + 1) = confusion(d(i) + 1,d(i) + 1) + 1;
else
    plot(X(i,1),X(i,2)); % Plot filled (options not shown)
    confusion(d(i) + 1,y(i) + 1) = confusion(d(i) + 1,y(i) + 1) + 1;
end
end
..... plot prototype centers oversize.....

```

The results from this example are shown in Figure 16.21, and the confusion matrix is given in Table 16.3. The confusion matrix shows that this approach is slightly less accurate in classifying the test data than the better of the two k -nearest neighbor classifiers, at least with the parameters used here. Some improvement can be made by modifying k as shown in one of the problems.

Both cluster analyses can be applied to data having multiple classes. In fact, the code in both examples can be used directly; only the `nu_classes` variable needs to be changed. Figure 16.22 shows the results of classifying a four-class data using the k -means classifier in Example 10.10. The confusion matrix resulting from this four-class classification example is shown in Table 16.4. The test set data contains 400 points and is normalized as described previously. The applications of these cluster analyses to multiclass data sets are found in the problems.

Table 16.3 Confusion Matrix from Example 16.10

True Class	Predicted Class	
	Class 0	Class 1
Class 0	188	12
Class 1	9	191

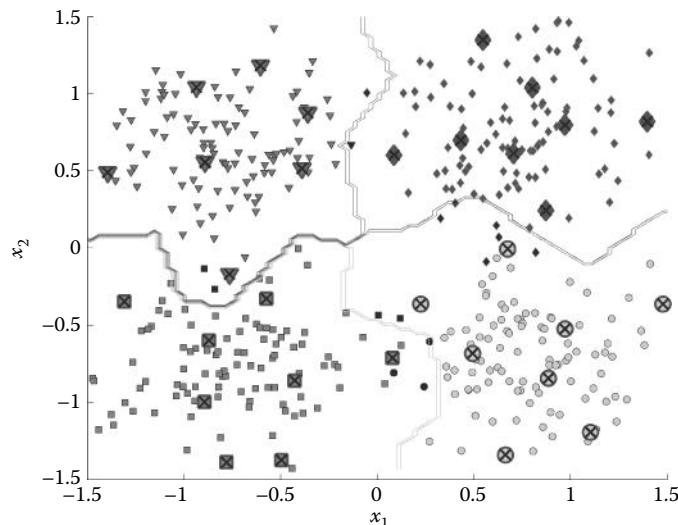


Figure 16.22 Classification of a four-class, two-variable test set using k -means clustering. Eight prototypes are used to represent each class ($k = 8$) and their positions after training are shown as large symbols with an “X.” The errors are summarized in the confusion matrix given in Table 16.4.

Table 16.4 Confusion Matrix from Data in Figure 16.22

True Class	Predicted Class			
	Class 0	Class 1	Class 2	Class 3
Class 0	96	1	0	2
Class 1	0	97	1	0
Class 2	0	4	93	0
Class 3	1	0	0	99

16.9 Summary

Making decisions from data is an important component of biomedical engineering. Most of the time, such data-driven decisions are made by medical staff or researchers, but sometimes, it is desirable to automate the decision-making process because of time, economic, or other considerations. Classification searches for patterns in data sets, sometimes very large multivariable data sets, with the goal of associating these patterns with a limited set of classes. In medical applications, classes often relate to specific medical conditions, but in biological research, classification may describe different experimental outcomes. Occasionally, classification is used in a desperate attempt to bring some utility to a disappointing data set.

Classifiers may be supervised or unsupervised. In the former, a classifier adapts its response based on a trial data set known as the training set. After training, classifier performance can be evaluated using an evaluation data set* before performing its assigned task on the test set data. Unsupervised classifiers search the data for intrinsic patterns and receive no *a priori*

* An evaluation data set is used to evaluate the performance of the classifier, but is not used to train the classifier. In that sense, all of the so-called “test sets” used in the various examples are really evaluation data sets.

information on what the data set may hold. Such classifiers are sometimes used in “data mining” approaches, which search large data sets for information that may be hidden or unknown. Only supervised classifiers are covered in this text as they are most commonly used in medical instrumentation and medical analysis.

Linear classifiers attempt to construct boundaries that consist of straight lines, planes, or hyperplanes depending on the dimension of the data sets, that is, the number of measurement variables being considered. Linear classifiers can be quickly and easily trained using an approach that seeks to minimize classifier error over the entire data set. Unfortunately, least squared error training may not lead to boundaries that generalize well so that classifier performance degrades when applied to a much larger test set. Such training emphasizes the entire training set, but data closer to the boundaries are more critical in terms of error. The LSVM (linear support vector machine) constructs a linear boundary that emphasizes data that are close to the boundary and are more indicative of potential errors.

Often, a linear boundary is optimal as when the data are Gaussianly distributed. If a nonlinear boundary is deemed appropriate, linear classifiers can still be used, but the input data are embedded in a higher-dimensional feature space. Almost any nonlinear boundary can be achieved from a linear classifier by transforming the data to a higher dimension. However, the “curse of dimensionality” warns us that high-dimensional spaces will be sparsely populated unless the training set is very large; so, the boundaries found are not likely to generalize well. The SVM approach uses embedding in higher-dimensional space to produce nonlinear boundaries. The fact that SVM emphasizes data points close to the boundary tends to circumvent the scarcity of data points in higher-dimensional embedding and is probably a major factor in its real-world success.

The two performance measures commonly used to evaluate measurement systems and classifiers are sensitivity and specificity. These measures are used when the outcome of the class is dichotomous. If one class is labeled positive and the other class is labeled negative (where the class you mostly want to detect is positive), then sensitivity is the percentage of true positives (correctly identified positives) out of the total positives. Specificity is the percentage of true negatives out of the total negatives. A perfect classifier or test would have 100% sensitivity and 100% specificity. In many systems and classifiers, it is possible to trade-off between sensitivity and specificity. The receiver operating curve (ROC) describes this trade-off for any given classifier or measurement system and can be used to determine an acceptable compromise. When more than two classes are involved, classifier performance can be described using a confusion matrix that is a table containing the percentage of correct classifications for each class and the percentages of error for the various combinations.

PROBLEMS

- 16.1 Load the data file `prob16_1.mat` that contains both training set variables, `Xt` and `dt` ($N = 100$), and test set variables, `X` and `d` ($N = 400$). Both sets have the same statistical distribution: two classes that are Gaussianly distributed with a center separation of four standard deviations. This spacing between the classes is such that some overlap will occur. Apply the linear least squares classifier used in Example 16.1 to train on the training set (`Xt`, `dt`), then apply the classifier to the test set. The routine `linear_eval` can be used to plot the data and determine the sensitivity–specificity of both the training and test sets. [Hint: Do not forget to add the additional column of 1s to *both* the training and test set data to account for the bias term as was done in Example 16.1.] In this and the other problems in this chapter, it is convenient to put the sensitivity and specificity values in the figure titles.
- 16.2 Repeat Problem 16.1 with data file `prob16_2.mat`. This is similar to the data set used in Problem 16.1 except that the training set consists of only 26 points. Compare the test set sensitivity–specificity obtained from this more limited training set with

Biosignal and Medical Image Processing

that of the test set and with the difference in training and test set performance obtained in Problem 16.1. In this and the other problems in this chapter, it is convenient to put the sensitivity and specificity values in the figure titles.

- 16.3 Load the data file `prob16_3.mat` that contains only a training set data `Xt` and `dt` where each input pattern consists of *four different variables* ($N = 400$). This data set cannot be plotted or evaluated using the support routines provided here; so, you will have to write your own routine. The training is exactly the same as in the linear examples (i.e., Equation 16.7 applies), but the evaluation needs special software. After training the linear classifier using the least squares technique, determine and output the true positives, true negatives, false positives, false negatives, sensitivity, and specificity. Of course, you will not be able to plot this 4-D data set. Again, write your own evaluation software to determine these measurements. [Hint: Use Equation 16.7 to train the classifier and an extension of Equation 16.4 to evaluate the results.]
- 16.4 Load the data file `prob16_4.mat` that contains a training set data `Xt` and `dt` ($N = 100$) and a test set `X` and `d` ($N = 400$) where each input pattern consists of four different variables. Train the four-variable classifier using the approach in Problem 16.3, but evaluate the data using the separate test set. The evaluation can be done using the same software developed in Problem 16.3. Determine and output the true positives, true negatives, false positives, false negatives, sensitivity, and specificity for both the training and the test set. As expected, the sensitivity and specificity are lower for the test set data than for the training set.
- 16.5 Load the data set `problem16_5.mat` that contains a training set `Xt` and `Dt` ($N = 200$) consisting of two variables, but four different classes. Apply linear analysis. Use an approach similar to that of Example 16.8 with a loop that trains each class separately. (Note: The statement that adds the column of 1s to the data set should be outside and should precede the loop). Use `linear_eval` in a loop to plot the boundaries for each class. Use `subplot` to plot the results of each class separately. Put the sensitivity and specificity obtained from each class in the title of the subplot.
- 16.6 Extend Problem 16.5 to include an analysis of test data. Load the data set `problem16_5.mat` that contains a training set `Xt` and `Dt` ($N = 200$) and test set `X` and `D`. These sets consist of two variables, but four different classes. Apply linear analysis. Use `linear_eval` in the loop to plot the data, boundaries, and sensitivity/specificity of both the training and the test evaluation. Note that this approach does not take into account the possibility that some points could be classified into two different classes. Also, the specificity is based on the data in all other classes.
- 16.7 Load the data file `prob16_7.mat` that contains training (`Xt`, $N = 50$) and test set data ($N = 400$, `X`) and two sets of correct classifications: `dt` and `d` for the training and test set data where the classes are indicated by ± 1 , and `dt1` and `d1` for the same training and test set data where the classes are indicated by 0 and 1. These different correct classification vector will allow you to compare both a linear least squares and a linear SVM on the training and test data sets. Use `linear_eval` to plot the least squares training and test set results and use `svcpplot` and `plot_results` to plot the LSVM training and test results. Also, use `svcbound` to plot the boundaries on the test set data. [Note: The routine `svcpplot` requires its own figure window.] Note that while the sensitivity and specificity obtained by the two linear methods are different, they just represent different trade-offs and the general performance is about the same.
- 16.8 Repeat Problem 16.7 with the training and test sets in file `prob16_8.mat`. In this file, the training set is only 10 points while the test set is 200 points. Correct

classification is specified by d and dt as ± 1 and by $dt1$ and $d1$ as 0 and 1. Again, apply both a linear least squares and a linear SVM to these data sets. Plot only the test set results side by side. Unlike in Problem 16.7, you should see somewhat better performance from the LSVM classifier. The ability to find boundaries using small training sets (or training sets with a number of dimensions) that still generalize well is a particular strength of the SVM approach.

- 16.9 Load the data file `prob16_9.mat` that contains two classes, one of which is surrounded on three sides by the other class as in Figure 16.16.
- Use the least squares classifier in conjunction with a cross-product kernel as in Example 16.4 to classify this training set data. Use `linear_eval3D` to evaluate the classification results.
 - Use the least squares classification applied directly to the data (i.e., without the kernel) and compare the performance with regard to sensitivity–specificity.
(Note that while the classification is quite fast, the rendering performed by `linear_eval3D` can take some computational time.)
- 16.10 Load the file `prob16_10.mat` that contains the same training set as used in Problem 16.9 except that the classes are specified as ± 1 instead of 0 and 1. Apply the SVM algorithm using a second-order polynomial to the training set. After the classifier is specified using SVC and the data plotted, you can determine the sensitivity and specificity by applying the classifier using the `svcoutput` to the training data:

```
y = svcoutput(Xt, dt', Xt, ker, alpha, b);
```

Then use `plot_results` to get the sensitivity and specificity just as you would on test set data. Note the improvement in sensitivity and specificity over the approach used in Problem 16.9. This is probably because the polynomial boundary is more appropriate than the effective boundary developed in Problem 16.9.

- 16.11 Repeat Example 16.7 using the same training and test set data found in file `Ex17_7.mat`, but use a polynomial order of 6. Compare the resulting sensitivity and specificity with the second-order polynomial used in Example 16.7 and the fourth-order polynomial used in Figure 16.18. Note the increasing degradation in performance emphasizing the need for appropriate machine capacity.
- 16.12 Load the data file `prob16_12.mat` that contains the training and test data of two classes arranged diagonally as in Figure 16.9. Train on the training set data ($N = 50$) and evaluate on the test set data as in Example 16.7. Use an SVM with a polynomial kernel. Evaluate with polynomial orders of 2, 4, and 6. Note that the higher-order polynomials are not as effective as the second-order polynomial.
- 16.13 Repeat Problem 16.12 using the training and test set data in file `prob16_13.mat`. This set has the same statistical characteristics as the data set in Problem 16.12, but with only 20 points in the training set. Note that the degradation with increased machine complexity is greater in this case when the training set is smaller.
- 16.14 Load the data file `prob16_14.mat` that contains a four-class training and test set with similar statistical properties as the data in Example 16.8 except that the classes are more closely spaced. Follow the approach used in Example 16.8 to determine and plot the boundaries found for these training data. Each of the four classes contains 50 points. You need to determine the sensitivity and specificity obtained for each class by

Biosignal and Medical Image Processing

- identifying and counting the errors in the plot. [Note: This problem will take several minutes to run, but the SVM classifier converges to an optimal solution for each class.]
- 16.15 Apply the k -nearest neighbors classification method to the training and test set data from Problem 16.1 in file `prob16_1.mat`. Use $k = 9$. (Note that `d` and `dt` are already in the necessary format.) Determine the sensitivity and specificity from the confusion matrix data supplied by `cluster_eval`. (In the cluster analysis programs, the diseased patients are class 1 and the normals are class 0.) Note that although the border is no longer a straight line, the performance is about the same as the linear classifier in Problem 16.1. Now, repeat the analysis using the data in `prob16_2.mat` where the training set is much smaller ($N = 26$ versus 100). Note that the reduction in sensitivity is even greater than with the linear classifier in Problem 16.2. Recall that when the data are Gaussianly distributed, as they are here, a straight line is optimal.
- 16.16 Apply the k -nearest neighbors classification method to the training and test data in file `prob16_16.mat` containing variables `Xt`, `dt`, `X`, and `d`. The test and training set data have been whitened and the correct classification vectors are in the appropriate format (numbers 0–3). These are four-class data sets; so, modify Example 16.9 accordingly. Use k values of 8, 10, and 16 and output the confusion matrix. Take the average sensitivities (the average of the diagonals of the confusion matrix) as a measure of the overall performance. Plot the boundary graph of the best result. How does performance vary with k ?
- 16.17 Apply k -means classifier to the data in the file used in Problem 16.15 (`prob16_16.mat`). Again, this has training and test sets consisting of four classes. Classify the test set using k values of 8, 10, and 16 and output the confusion matrix. Take the average sensitivities (the average of the diagonals of the confusion matrix) as a measure of the overall performance. Plot the boundary graph of the best result. How does performance vary with k ? Overall, how does the k -means classifier compare with the k -nearest neighbor classifier?

17

Classification II

Adaptive Neural Nets

17.1 Introduction

Adaptive neural nets (ANNs) constitute a popular method for classification that can generate very complex decision boundaries yet generalize well. ANNs are based on elements that are an outgrowth of early attempts to model the human nervous system and the name carries a sense of mystery that is largely undeserved. They are easy to implement either through software packages such as MATLAB's Adaptive Neural Net Toolbox or standard MATLAB. Here, we explore ANNs using standard MATLAB code as is done with classification methods in Chapter 16. While the toolboxes provide more flexibility, implementing ANNs directly in MATLAB code provides more insight into their construction and operation.

A typical three-layer neural net is shown in Figure 17.1. In this configuration, the upper layer of artificial neurons receives the inputs and is called the *input layer*. The bottom layer of neurons provides the output(s) and is called the *output layer*. The middle layer is called the *hidden layer* because the outputs of these neurons are not readily available. The outputs of the input (top) layer are also unavailable; so, this could also be considered a hidden layer. In Figure 17.1, both input signals are connected to each input neuron and each neuron's output is connected to all the neurons in the next layer. This is termed a *fully connected* net and is commonly used when there is little *a priori* information about the nature of the input signals. Special configurations are sometimes used for specific input signal patterns.

17.1.1 Neuron Models

The first neuron model was developed by McCullough and Pitts in the 1940s and established the guidelines for future artificial neurons. In the McCullough–Pitts neuron, the inputs are scaled by multiplier weights and these weighted inputs are then summed (Figure 17.2 and Equation 17.1).

$$a = \sum_{i=1}^N x_i w_i \quad (17.1)$$

where x is the input signal, w is the multiplier weights, i is the input channel, and N is the total number of inputs. This equation is identical to the equation of a linear classifier in Chapter 16

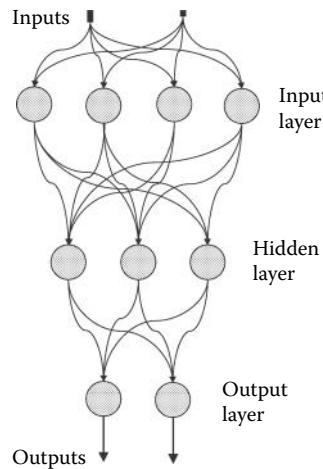


Figure 17.1 A fully connected three-layer neural net with two input signals and two output signals. The top layer is the input layer, the bottom layer is the output layer, and the middle layer is the hidden layer.

(Equation 16.4). As with the linear discriminator, the output or activation, a , is sent to a threshold element that produces an output that is either 1 or -1 depending on the sign of the activation signal (Figure 17.2). Specifically, the output is 1 if the activation a is ≥ 0 and is -1 if $a < 0$. Just as in linear discrimination, the neuron can also have a constant bias input, b , or the bias parameter can be represented by adding a constant input signal of $+1$, and the weight for this input is the bias.

Since the McCullough–Pitts neuron has only two output levels, it can only identify two classes. However, multiple classes or patterns can be classified using multiple neurons in parallel. Each parallel element is connected to all the input signals. Alternatively, preprocessing elements common to the input signals can feed the neuron. Such a collection of elements was developed by Rosenblatt in 1967 and, in conjunction with a set of specialized preprocessing elements, was termed the *perceptron* because of its ability to sense patterns (Figure 17.3). In

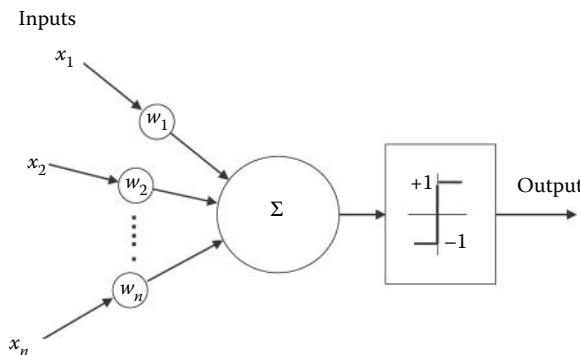


Figure 17.2 The McCullough–Pitts neuron. The input signals, x_i , are summed to generate an activation signal. The activation signal is fed to a threshold function that produces either a $+1$ or a -1 output signal depending on the sign of the activation signal. In some versions of this neuron, the output is 0 and 1 rather than ± 1 . These neurons are also sometimes referred to as *threshold logic units* (TLUs), reflecting their function. The equation for this neuron is the same as for the linear discriminator described in Chapter 16.

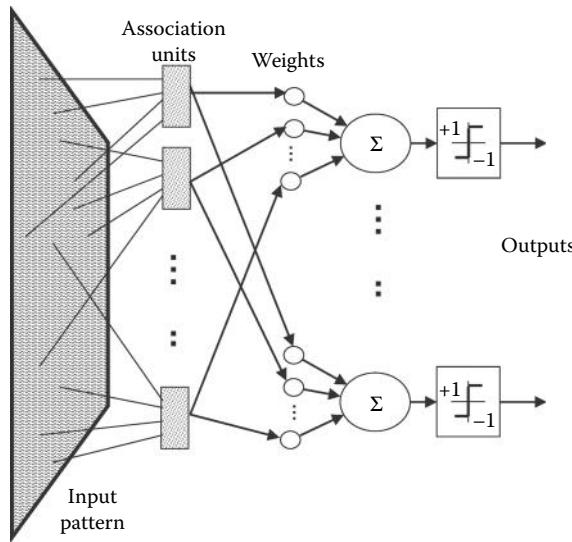


Figure 17.3 The perceptron: a set of parallel McCullough–Pitts neurons with the addition of pre-processing elements that modify the incoming signals in a specialized way. These elements can be used to detect any number of classes or features.

classifiers based on artificial neurons such as the McCullough–Pitts neuron, the weights, w , and biases, b , constitute the free parameters and, as in linear discriminators, are adjusted during the training period to give the best classification.

A number of variations of the basic McCullough–Pitts designs have been developed mainly by modifying the threshold operator that follows the weighted summation. For example, in the linear neuron (Figure 17.4a), the activation signal feeds a linear function: effectively, the output is simply the activation signal, a , or a scaled version of this signal. Unlike linear classifications, the output of this type of neuron can take on any value. Another set of artificial neurons retains the nonlinearity of the McCullough–Pitts neurons, but reduces the sharp transition produced by the threshold element by using a smooth nonlinear function such as a hyperbolic tangent function (Equation 17.2 and Figure 17.4b) or a sigmoid (or logistic) function (Equation 17.2 and Figure 17.4c). The equations for the sigmoid and hyperbolic functions are

$$\begin{aligned} y &= \frac{1}{1 + e^{-a}} && \text{Sigmoid} \\ y &= \frac{e^a - e^{-a}}{e^a + e^{-a}} && \text{Hyperbolic Tangent} \end{aligned} \quad (17.2)$$

where a is the input and y is the output of the nonlinear element. In these modified McCullough–Pitts neurons, the output can take on a range of values, but this range must be between 0 and +1 for the sigmoid function and ±1 for the hyperbolic function. The derivatives of these functions shown as dashed lines in Figure 17.4 are used in training as described later.

The neuron types described above form the basic elements in most ANNs. Neural nets could be constructed using specially designed hardware components, but it is easier to simulate them on a computer. Computer simulation of neural nets is used for both training and testing the net. As with linear classifiers, training is done by adaptively modifying the neuron's free parameters, in this case its weights and bias, using a training set in which the correct classes are known. The basic concern in the training of ANNs is the same as for the classifiers described in Chapter 14: minimizing classification errors while retaining generality. The speed of convergence during

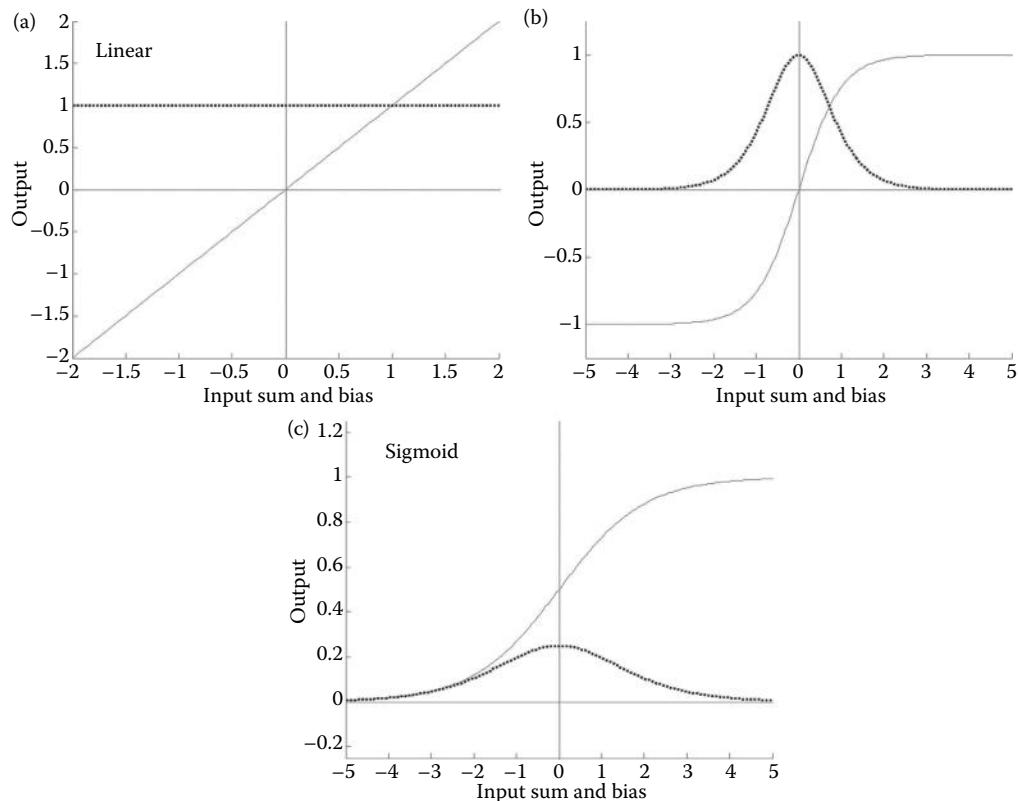


Figure 17.4 Functions commonly used in artificial neurons to convert the activation signal (a in Equation 17.1) into an output signal. (a) A linear function. (b) A hyperbolic tangent function. (c) A sigmoid function. The darker line is the derivative of the function and its importance is explained later.

training can also be a concern, but since in most applications, the net has to be trained only once (or occasionally retrained as more training data become available), this is usually not a major factor. It is important to remember that the performance of any classifier is based on its ability to correctly classify the test set and as with other classifiers; it is all too easy to generate ANNs that perform superbly on test sets, but end up as poor classifiers in practice.

17.2 Training the McCullough–Pitts Neuron

The McCullough–Pitts neuron (or perceptron) can be trained using a very simple learning rule and this rule can be used to classify any linearly separable training set. The McCullough–Pitts neuron generates a linear boundary between input patterns. Since the boundary occurs at $a = 0$, the boundary equation is a slight modification of the linear discriminator boundary given in Equation 16.10. Rewriting Equation 17.2 to explicitly include a bias term, we get

$$a = \sum_{i=1}^N w_i x_i + b \quad (17.3)$$

where again i is the input channel and N is the number of input channels.

17.2 Training the McCullough–Pitts Neuron

The threshold occurs at $a = 0$; so, the decision boundary can be found by setting the left-hand side of Equation 17.3 to 0:

$$w_i x_i + b = 0; \quad w_i x_i = -b \quad (17.4)$$

Equation 17.4 can be solved for any number of input signals, x_i . If we set the number of inputs to 2 (as we do in most examples to allow easy plotting), Equation 17.4 becomes

$$w_1 x_1 + w_2 x_2 = -b$$

And solving for x_2 in terms of x_1 :

$$x_2 = \left(\frac{-w_1}{w_2} \right) x_1 - \frac{b}{w_2} \quad (17.5)$$

which is the equation of a straight line having a slope of $-w_1/w_2$ and an intercept of $-b/w_2$. This equation is the same as for a linear discriminator (Equation 16.10) except for the inclusion of the bias term, b , and the absence of the 0.5 term since the threshold of this neuron is 0 instead of 0.5. With the proper adjustment of the weights and bias, this neuron should be able to distinguish between two classes as long as the input pattern is linearly separable. If there are three input channels (i.e., three variables), Equation 17.4 still holds except the boundary becomes a plane in 3-D space (x_1, x_2, x_3) instead of the line in Equation 17.5. Example 17.8 shows the classification of a three-input pattern where the decision pattern is a plane (Figure 17.18). If there are more than three input channels, the decision boundary is a hyperplane in N -dimensional space, where N is the number of input channels.

The McCullough–Pitts neuron is trained using the *perceptron rule*, so-called because it was first applied to train perceptrons. For each input, the output of the neuron is calculated and compared with the desired output. If the error is zero, the pattern is correctly classified and the weights are not changed. However, if there is an error, the weights are changed in a manner that reduces the error. For example, if the inputs are positive, a negative error suggests that the weights are too low and should be increased, whereas a positive error indicates that the weights should be reduced. If all the inputs are negative, then the reverse is true. Thus, the weights are modified by the error, taking into account the sign of the input on a given weight. In fact, it makes sense to modify each weight by the product of the error times the weight's input signal (magnitude and sign), since a larger input signal indicates that the associated weight has contributed more heavily to the error. For example, if a given input channel has an input that is zero or very small, that channel and weight has had little to do with the error. Conversely, if the signal is large, the weight's contribution to the error is large and a greater weight change is in order. This concept of “responsibility” toward the error is carried over to more complicated training algorithms in a later section of this chapter. The bias is updated simply by multiplying it by the error when an error exists (since it can be considered as a weight on a channel with an input of +1). This strategy translates to an equation for updating the weights and biases when an error, e , occurs:

$$\begin{aligned} w_i(n+1) &= w_i(n) + e x_i(n) && \text{where: } e = d - y \\ b_i(n+1) &= b_i(n) + e \end{aligned} \quad (17.6)$$

where the updated weights and biases are on the left side of the equations and the original weights and biases are on the right side. The error, e , equals $d-y$ where d is the desired output and y is the actual output.

Biosignal and Medical Image Processing

Equation 17.6 is based on only one input pattern (or at least one at a time), so that it could overcorrect with regard to the entire data set. For that reason, the error correction factor, $ex_i(n)$, is sometimes scaled down by a multiplying constant, $\alpha < 1$. The updated equation for the weights under this condition would be

$$\begin{aligned} w_i(n+1) &= w_i(n) + \alpha ex_i(n) \\ b_i(n+1) &= b_i(n) + \alpha e \end{aligned} \quad (17.7)$$

The constant α determines how fast the weights converge to a stable solution and must be determined empirically as in adaptive filtering. If α is too large, the weights will oscillate around the correct solution and if it is too small, convergence will be very slow taking a large number of passes through the data set. (Each pass through the data set is termed an *epoch*.) Usually, a scaling constant of 1.0 leads to convergence; so, in most versions of the perceptron, training rule α is omitted, as in Example 17.1 below.

The perceptron training rule is very simple and it is guaranteed to converge to a solution. Unfortunately, there are shortcomings to this approach that become evident in the first example. Example 17.1 applies the perceptron training rule to a McCullough–Pitts neuron to classify a linearly separable data set.

EXAMPLE 17.1

Classify a two-variable input pattern consisting of two classes. Each class consists of 50 patterns having a Gaussian distribution over both variables. The centers of the two classes are far enough apart so that the classes are *linearly separable*.

Solution

Use a McCullough–Pitts neuron in Equation 17.1 and the perceptron training rule given by Equation 17.6. Generate the training set using the `gen_data2` routine described in the previous chapter. To train the neuron, repeat the training over multiple *epochs*. Report the result of training every epoch and stop when the error becomes zero.

```
% Example 17.1 Single McCullough-Pitts neuron example
% Uses the perceptron learning rule
%
% Generate training set: two Gaussian distributions
angle=45;           % Angle between distributions in deg.
distance=6;         % Separation of the two clusters in STD
[X, d]=gen_data2(distance, angle,'l',[ -1 1]); % Generate data
%
% Initialize weights and bias
[~,c]=size(X);          % Deterimine nu. inputs (N)
w=(rand(c,1) - .5)/4;    % Initialize weights and bias to small
b=(rand - .5)/4;          % random numbers between -/+ 0.25.
%
% Train using training set
for k=1:10              % Multiple epochs
    [w,b]=percpt_learn(X,d,w,b); % Evaluate training
    [sensitivity, specificity]=linear_eval([X,ones(r,1)],d,[w;b],0);
    if sensitivity == 100 && specificity == 100
        disp('Convergence')
        break;
    end
end
```

Analysis

After generating the data set X in `gen_data2`, the program initializes the two weights to a small random number between ± 0.25 . The weights are stored in vector w and the length of this vector is equal to the number of input signals as determined from the number of columns in the training set. This keeps the program general so that it can handle any number of inputs, but in this example, there are only two inputs; so, X contains two columns.

Training the net is accomplished by the routine `percept_learn` shown below that has as inputs: the signals, X , the desired response, d , and the weights and bias, w and b . As in the previous chapter, the input signals or *input patterns* are stored in X . In this example, they are stored in a 100×2 matrix where the columns contain the two input signals. The outputs for `percept_learn` are the updated weights and biases, that is, w and b . This training routine operates on the entire training set, that is, one epoch. Following training, the performance of the perceptron is evaluated using `linear_eval`, described in Chapter 16. This routine plots the data and determines sensitivity and specificity (see Section 16.3). The training and evaluation routines are placed in a loop so that multiple epochs can be employed to train the neuron, but often, the net converges to zero error in one epoch. The number of epochs required to produce zero error depends on the difficulty of the training set: specifically, how close the two classes are overlapping. A problem at the end of this chapter explores this issue. The training routine is shown below; see Chapter 16 for the evaluation routine code.

```
function [w,b] = percept_learn(X,d,w,b)
% Simple learning rule for perceptrons
% Inputs X, d are training set data
%
% Inputs: assumed a matrix where the number of rows are different
%         samples and the number of columns are the input patterns)
%
% X Input patterns (a matrix)
% d Correct outputs; i.e., targets (assumed a vector)
% w Current weights
% b Current bias
%
% Outputs
% X Updated weights
% b Updated bias
%
[r, ~] = size(X); % Number of input patterns
for i=1:r % Train over epoch
    y=MCP_neuron(X(i,:),w,b); % Apply MCP neuron
    e=d(i) - y; % Compute the error
    w=w+e*X(i,:>'); % Update weights
    b=b+e; % Update bias
end
```

This routine implements the perceptron learning rule of Equation 17.6 directly for $\alpha = 1$. Note that if the error, e , is zero, no change in the weights or bias occurs as the rule dictates. This routine uses the routine `MCP_neuron` to implement the McCullough–Pitts neuron:

```
function y=MCP_neuron(x,w,b)
% Function to simulate a McCullough-Pitts neuron
% Inputs
% x inputs to neuron (vector)
% w weights (vector)
% b bias (scalar)
```

Biosignal and Medical Image Processing

```
% Outputs
% y output (-1 or 1)
%
a = sum(x.*w') + b;      % Get sum of weighted inputs
if a >= 0                  % Set output threshold
    y = 1;                  % Determine output
else
    y = -1;
end
```

This routine first determines the total activation by multiplying the input vector, x , by the weight vector, w , and summing (Equation 17.3). The output is set to -1 if the sum is negative or $+1$ if the sum is positive or zero. (The MATLAB `sign` function could be used here except it returns a zero if the sum is zero.)

Results

After each epoch, the error is evaluated by routine `linear_eval` described in Chapter 16. The sensitivity and specificity are evaluated and the training ends when they both reach 100. In this example, the training usually required only one epoch and the plot generated by `linear_eval` is shown in Figure 17.5.

Figure 17.5 illustrates the major problem with the perceptron training rule. As long as the data are linearly separable, the rule always results in a solution, but it is not necessarily the best solution. The decision boundary seen in Figure 17.5 is clearly not optimal and while it works well for the training set, we would expect it to perform poorly on a test set. (Comparisons of this sort are presented in several problems at the end of this chapter.) Clearly, the boundary would be placed better further to the right. Moreover, if the data were not linearly separable, the training algorithm will not terminate since the stopping criterion, zero error, can never be met. Other stopping rules could be developed, but again, there is no guarantee that these

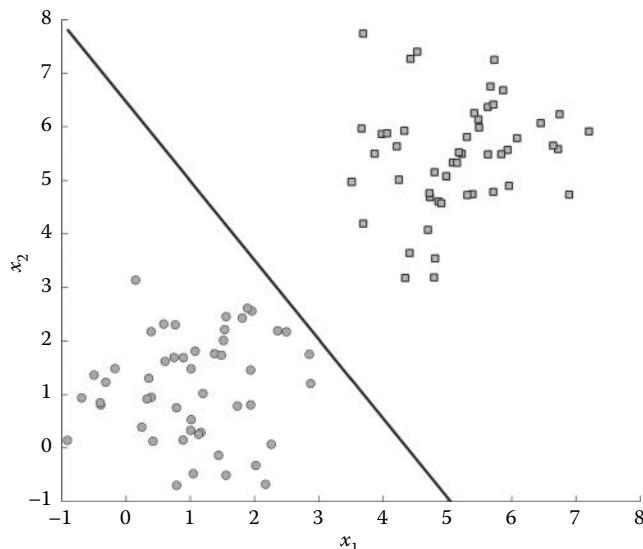


Figure 17.5 Classification using a single McCullough–Pitts neuron. The two classes (squares and circles) are Gaussianly distributed about centers separated by six standard deviations and are linearly separable. The solid line shows the decision boundary found by the perceptron training rule. As the two classes are perfectly separated, both the sensitivity and specificity are 100%.

17.3 The Gradient Decent Method or Delta Rule

would result in the best solution. The next section presents a training approach that will find the optimum solution even when the data are not linearly separable and this approach is easily extended to nets with multiple layers. The approach does require many more training epochs to reach the optimum solution, but since a net does not require frequent training, this is not usually a problem.

17.3 The Gradient Decent Method or Delta Rule

The gradient decent method is very similar to the LMS algorithm used in adaptive filtering and even employs the same approximation developed by Widrow–Hoff. Since it is based on descending the error gradient, it is often referred to as the *delta rule* where “delta” refers to *change in error*. As with the least square method, the delta rule minimizes an error measure, usually the squared error or mean squared error (MSE). The neuron’s weights and bias are adjusted so that the error proceeds toward a minimum along its steepest gradient. To determine the true error gradient, the MSE should be averaged over a training epoch, but this is computationally intensive as it requires evaluating the error and its gradient over what could be a large number of input patterns. One shortcut is to use the error determined for each input pattern as in the perceptron training rule (Equation 17.7). An even better way to reduce the computational burden is to approximate the error gradient using the approach by Widrow and Hoff. Specifically, the error gradient is approximated by the input times the error:

$$\frac{\partial e_i}{\partial w_i} \approx e_i x_i \quad (17.8)$$

The gradient method cannot be applied to nets containing McCullough–Pitts neurons because the error gradient cannot be computed given the discontinuous behavior of that neuron. However, neurons based on any of the continuous functions shown in Figure 17.4 lead to meaningful error gradients. The nonlinear functions, the sigmoid, and the hyperbolic tangent are particularly popular as they compress the output and that tends to give their nets greater stability. In the following examples, the sigmoid function is used, but the hyperbolic tangent gives equally good results as illustrated in Problem 17.2.

If the neuron is nonlinear, the gradient decent approximation method should also include the derivative of the nonlinearity in the weight-update equation (Equation 17.8). Since the updating is done on each input pattern, a moderating constant that performs the same function as α in Equation 17.7 is needed to ensure stability. In the gradient method, this constant is termed the *learning constant*. Combining these features leads to the gradient equation for updating the weights and bias:

$$w_i(n+1) = w_i(n) + \alpha \left(\frac{\partial f(a)_{Ni}}{\partial a} \right) e x_i(n) = w_i + \alpha f'(a)_{Ni} e x_i = w_i(n) + \delta e x_i \quad (17.9)$$

$$b_i(n+1) = b_i(n) + \alpha \left(\frac{\partial f(a)_N}{\partial a} \right) e = b_i(n) + \alpha f'(a)_{Ni} e = b_i(n) + \delta e \quad (17.10)$$

where $e = d - y$ as in Equation 17.6 and $\partial f(a)/\partial a \equiv f'(a)$ is the derivative of the neuron nonlinearity under activation a . Often, the learning constant, α , and the neuron derivative, $f'(a)$, are combined into a single term, δ , that represents the net scaling of weights and bias.

Except for the derivative term, this equation is identical to the perceptron training rule (Equation 16.7). However, there is really no direct link between the two as they were developed

Biosignal and Medical Image Processing

using fundamentally different theoretical concepts: the perceptron rule was based on adjustments of the decision boundary whereas the delta rule is a descendent of the Widrow–Hoff approach to approximating error gradients.

As with the perceptron rule, the delta rule is guaranteed to converge provided α is small enough. The application of the delta rule to a data set similar to that of Example 17.1 is given in Example 17.2.

EXAMPLE 17.2

Classify a two-variable input pattern consisting of two classes. Each class consists of 50 patterns having a Gaussian distribution over both variables and the classes are linearly separable.

Solution

Use a sigmoid neuron and the delta training rule given by Equation 17.10. To train the neuron using the delta rule, a large number of epochs is required. Evaluate the result of training every 20 epochs. Adjust the learning constant empirically to provide rapid but stable convergence. Stop when the change in error between reporting periods becomes very small, <0.0001, or after 2000 epochs. The main program is given below.

```
% Example 17.2. Single neuron training
% Uses the delta algorithm
%
alpha = .5;          % Learning rate (set empirically)
.....Generate data and initialize weights as in Example 17.1.....
last_mse = 2;        % Set initial mse error high
for k = 1:100
    for k1 = 1:20
        [w,b] = lms_learn(X,d,alpha,w,b); % Multiple epochs
    end
    %
    % Evaluate neuron performance every 20 epochs
    [mse, sensitivity, specificity] = net_eval(X,d,w,b);
    if abs(last_mse - mse) < .00001 % Stopping criteria
        disp('Converged')
        break;
    end
    last_mse = mse;      % Update last mse
    disp([mse sensitivity specificity])
end
% Plot final result
[mse, sensitivity, specificity] = net_eval(X,d,w,b);
.....title and display total epochs.....
```

Analysis

The learning routine, `lms_learn`, is based on Equations 17.9 and 17.10. The inputs include the training data and the learning constant, `alpha`, and the initial weights and bias (`w` and `b`). The outputs are the updated weights. Note the similarity of the code between `lms_learn` and `lms` from Chapter 8.

```
function [w,b] = lms_learn(X,d,alpha,w,b)
%
[r, ~] = size(X);
for i = 1:r
    [y,der] = neuron(X(i,:),w,b,'s');
    e = d(i) - y;                      % Compute the error
```

17.3 The Gradient Decent Method or Delta Rule

```
w=w+alpha*e*der*X(i,:); % Update weights  
b=b+alpha*e*der;  
end
```

The routine to simulate the neuron (`neuron`) is now more general and includes the capability of representing McCullough–Pitts, linear, sigmoid, and hyperbolic tangent neurons. The activation is determined as the weighted sum of the inputs and the bias just as in routine `MCP_neuron`. The function call to routine `neuron` is

```
[y, der] = neuron(x,w,b,'type')
```

where `y` is the neuron output and `der` is the local derivative at that output. The input `x` is the vector of inputs and `w` and `b` are the neuron weights and biases. The argument '`type`' specifies the neuron function: '`m`' McCullough–Pitts, '`l`' linear, '`h`' hyperbolic tangent, and '`s`' sigmoid (the default). The code for the sigmoid option is shown below. The code for the complete routine can be found with the accompanying files.

```
function [y, der] = neuron(x,w,b,type)  
%  
delta=0.01; % Increment for derivative calc.  
a=(sum(x .* w'))+b; % Sum of weighted inputs  
if type == 's' % Sigmoid operator  
    y=1 / (1+exp(-a)); % Current output  
    y2=1/ (1+exp(-a-delta)); % Approximate next output  
    der=(y2-y)/delta; % to estimate derivative  
elseif .....other operators follow.....
```

The evaluation routine, `net_eval`, performs the same function as `linear_eval` in Example 17.1. However, `net_eval` has been extended to evaluate one-, two-, and three-layer nets composed of neurons having either linear, hyperbolic, or sigmoid (the default) functions. This routine uses `plot_results` from Chapter 16 to plot the data and calculate sensitivity and specificity. This routine adjusts the threshold to be appropriate for the neuron type: 0.0 for linear and hyperbolic neurons, and 0.5 for sigmoid neurons. Recall that for purposes of sensitivity and specificity, `plot_results` assumes that class 1 is negative and class 0 is positive. In addition to sensitivity and specificity, `net_eval` also calculates and outputs the MSE:

```
mse=sqrt(mean((d - y).^2));
```

where `d` is the correct classification and `y` is the neuron's output. Both are vectors taken over the entire data set.

Results

The classification produced by the single neuron applied to linearly separable data in Example 17.2 is shown in Figure 17.6. Note that the boundary appears more centrally placed between the two data sets in Figure 17.5. However, the delta approach requires training many more epochs than the perceptron training rule. In this case, 2000 training epochs occurred before the limit was reached and the change in RMS error, evaluated every 20 epochs, was still greater than the threshold of 0.0001.

Despite the large number of training epochs required, the training took <35 s on a modern PC computer and, since training is done infrequently, the training time is not a major factor. Generalization is a major factor and the boundary shown in Figure 17.6 is much more likely to generalize other data than the boundary in Figure 17.7. Moreover, the continuous nonlinearities

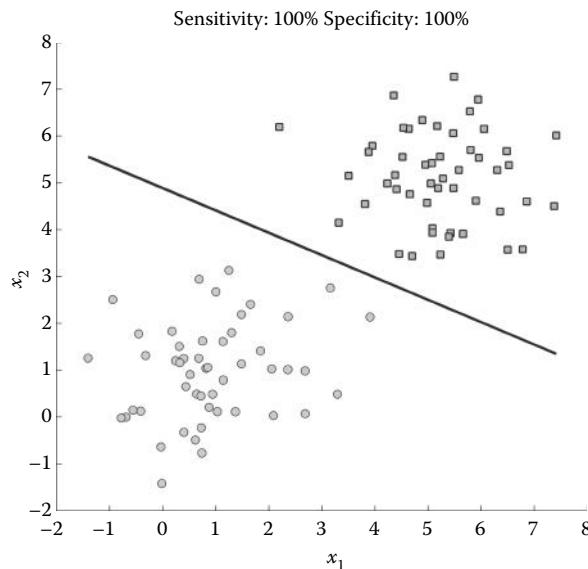


Figure 17.6 Classification using a single sigmoid neuron trained using the Widrow–Hoff delta training rule given in Equation 17.10. The two classes are shown as in Figure 17.5 along with the decision boundary. A comparison with Figure 17.5 shows that the decision boundary resulting from this rule is placed better, although the training took much longer. As the two classes are perfectly separated, both the sensitivity and specificity are 100%.

such as the sigmoid and the hyperbolic tangent work well in multilayer nets as demonstrated in the subsequent sections. For these reasons, the McCullough–Pitts neuron is rarely used in contemporary neural nets.

The sigmoid neuron does not really have the sharp boundary that is implied by the line in Figure 17.6. Since the neuron's output is a continuous function, the boundary has

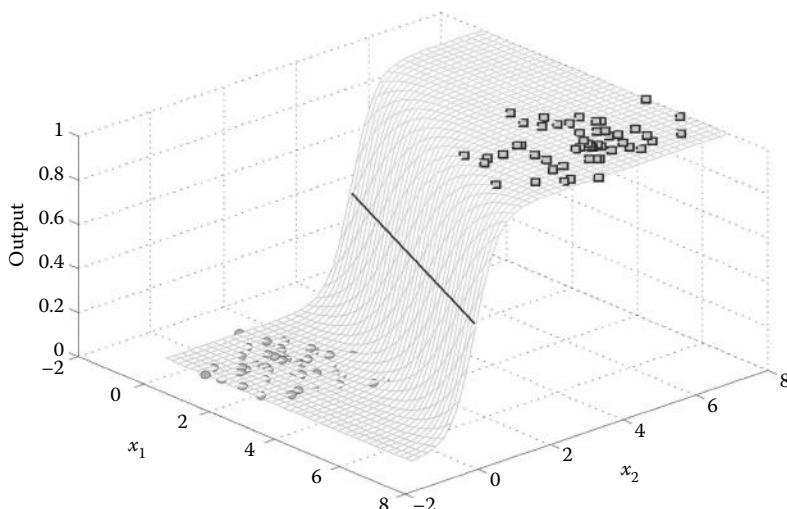


Figure 17.7 The actual output produced by the sigmoid neuron. The decision boundary occurs where the output is 0.5 as shown by the solid line here and in Figure 17.6.

a steep but smooth shape. The actual neuron output is shown as a continuous transition in Figure 17.7. The boundary shown in Figure 17.6 is where the neuron output is 0.5, half its maximum value.

A single neuron can only produce a single linear decision boundary. Some data sets, although separable, require a nonlinear boundary such as illustrated in Figure 16.2b. Neural nets with more than one layer can produce decision boundaries that are nonlinear. The next section covers the construction and training of two-layer nets. The training approach used for nets with more than one layer is known as *back projection*. This approach can be easily extended to three-layer nets as shown in Section 17.5.

17.4 Two-Layer Nets: Back Projection

A simple two-layer net is shown in Figure 17.8. This net has two input-layer neurons and one output neuron. It is easy to simulate the response of this net to any input pattern by having the outputs of the two input neurons become the inputs to the output neuron. It is also easy to train the output neuron using the delta algorithm since all the elements of Equations 17.9 and 17.10 are known. For example, the inputs to the output neuron, N_0 , are the outputs from the input-layer neurons (N_1 and N_2) that can be determined from the inputs to the net. The error, e , is found from the output of N_0 . Training the two input neurons, N_1 and N_2 , is more complicated. Their inputs are known, but their contribution to the error in the output must also be evaluated.

The most common technique for training multilayer nets is to project the error for the output layer back onto the preceding layers, an approach logically known as *back projection*. (Back projection as applied to neural nets is distantly related to the back-projection method used in image analysis as described in Chapter 15. Both methods project error back through the process to adjust parameters, but they greatly differ in details.) An estimation of the neuron's contribution to the output error is substituted for e in Equations 17.9 and 17.10. Consider the error contribution from neuron N_1 in Figure 17.8. The N_1 error component contributes to the output error via the left pathway after being scaled by weight, w_1 . Thus, the error component of N_1 can be approximated by projecting the output error back to N_1 and that means scaling the actual error e by weight w_1 . If the output neuron is nonlinear, the nonlinearity should be taken into account

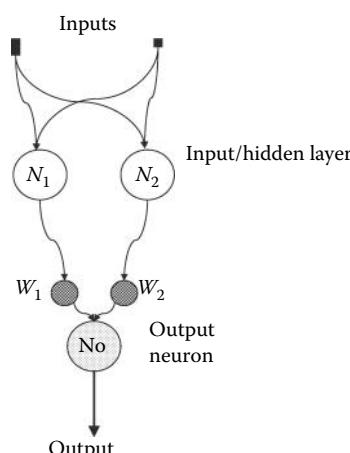


Figure 17.8 A simple two-layer neural net. Input neurons weights are not shown.

Biosignal and Medical Image Processing

by scaling the back-projected error by the derivative of this nonlinearity. The back-projected error is then estimated as

$$e_{back} = w_1 \left(\frac{\partial f(a)_{N_0}}{\partial a} \right) e = w_1 f'(a)_{N_0} e \quad (17.11)$$

where $\partial a / \partial a_{N_0} \equiv f'(a)_{N_0}$ is the derivative of the output neuron nonlinearity and e is the actual output error. This is the back-projected error. Substituting this new error for e into Equations 17.9 and 17.10 gives the updating equation for N_1 .

$$w_i(n+1) = w_i(n) + \alpha \left(\frac{\partial f(a)_{N_1}}{\partial a} \right) w_1 \left(\frac{\partial f(a)_{N_0}}{\partial a} \right) ex_i = w_i(n) + \alpha w_1 f'(a)_{N_1} f'(a)_{N_0} ex_i = w_i(n) + \delta ex_i \quad (17.12)$$

$$b_i(n+1) = b_i(n) + \alpha \left(\frac{\partial f(a)_{N_1}}{\partial a} \right) w_1 \left(\frac{\partial f(a)_{N_0}}{\partial a} \right) e = b_i(n) + \alpha w_1 f'(a)_{N_1} f'(a)_{N_0} e = b_i(n) + \delta e \quad (17.13)$$

where $\delta = \alpha w_i(a)_{N_1} f'(a)_{N_0}$, w_i and b_i are the weights and bias of N_1 , and x_i is the input to N_1 . Here, δ is more complicated than in Equations 17.9 and 17.10 because it combines the contribution from the output neuron, N_0 , with that of the neuron being trained.

The equation for updating the weights of N_2 is the same with appropriate changes in the various terms. Another way to look at these equations is that they assign the “blame” (or more positively, the “responsibility”) that neuron N_1 has for the overall error. Its ability to change the error at the output depends on the weighting applied to its output signal by the output neuron (w_1) and the derivative of the output neuron. If the output signal of N_1 is scaled down by a small weight value or if the output neuron has a small derivative (because it is at the extremes of the function), N_1 did not contribute much to the error and there is little it can do to change the error. On the other hand, if these two values are large, then N_1 is both responsible for, and has a strong ability to change, the output error. The δ component of Equations 17.12 and 17.13 is sometimes referred to as the *credit assignment*.

Equations 17.12 and 17.13 assume a configuration similar to that of Figure 17.8 where only one neuron is connected to N_1 . If the input-layer neuron connects to more than one output neuron or, in the case of three-layer nets, more than one neuron in the next layer, it receives a back-projected error from each neuron that is connected to its output. In this case, Equations 17.12 and 17.13 must be modified to include a summation of all the back-projected error.

$$w_i(n+1) = w_i(n) + \alpha f'(a)_{N_1} w_1 x_i \sum_{k=1}^K (w_k f'(a)_k) e \quad (17.14)$$

$$b_i(n+1) = b_i(n) + \alpha f'(a)_{N_1} w_1 \sum_{k=1}^K (w_k f'(a)_k) e \quad (17.15)$$

where $f'(a)_k$ is the derivative of the k th neuron in the next layer down that is connected to N_1 and K is the total number of neurons in the next layer down that connects to N_1 . This equation holds for any neuron in the hidden layer: only the inputs, weights and derivatives, and possibly K will be different.

While there are several terms in Equations 17.13 and 17.14, they are all *local to the neuron* being trained, that is, they can all be obtained from the neuron itself and neurons connected to its output. The weights, biases, or signals of all other neurons in the net are not relevant

or needed for training this neuron. This makes coding in MATLAB straightforward and also facilitates parallel-processing methods. This is shown in the next example, which implements a two-layer net to classify data that are separable but require a nonlinear decision boundary.

EXAMPLE 17.3

Use a two-layer net to classify a training set in which one class is surrounded on three sides by the other class. Construct a net with four input neurons that feed a single output neuron.

Solution

Implement the net and solve using back projection. Since the input neurons feed only a single output neuron, Equations 17.12 and 17.13 can be used to train the net. The main program follows the same structure as the previous examples except that more weights and biases need to be defined and initialized and the training and evaluation programs are modified.

```
% Two-layer net example. Shows the ability of a two-layer net to
% perform a classification of various Gaussian distributions.
% Uses the delta learning algorithm with back-projection
%
clear all; close all;
alpha=.1;                                % Learning constant
h1=4;                                     % Number hidden layer neurons
[X, d]=gen_data2(6,[],'s');    % Generate data, separable
[r,nu_inputs]=size(X);
%
% Generate initial weights for both layers
W1=(rand(nu_inputs,h1).1)*.25;
b1=(rand(1,h1)-1)*.25;
W2=(rand(h1,1)-1)*.25;        % Output layer with h1 inputs
b2=(rand(1,1)-1)*.25;        % and h1 weights, one bias
last_mse=2;                                % Initial mse, set large
%
for k3=1:400
    for k1=1:20
        [X,d]= mix_data(X,d);          % Training
        [W1,b1,W2,b2]=net_learn_2(X,d,alpha,W1,b1,W2,b2);
    end
    [mse, sen, sp]=net_eval(X,d,W1,b1,W2,b2);      % Evaluate net
    if abs(rms_error . last_rms_error)<.00001
        disp('Delta error minimum')    % Stopping criterion
        break;
    end
    last_mse=mse;                      % Update last_mse
    disp(ms_error)
end
net_boundaries(X,d,W1,b1,W2,b2);        % Plot boundaries
disp([sen sp])           % Display sensitivity and specificity
```

Analysis Support Routines

Example 17.3 uses one new routine and one routine that is a modification of the one used in the last example. The training routine, `net_learn_2`, shown below uses Equations 17.12 and 17.13 to train the hidden layer and Equations 17.10 and 17.11 to train the output neuron. Before each training epoch, the routine `mix_data` scrambles the order of the training data, which may improve learning. The routine `net_boundaries` is used to evaluate the boundary and plot Figure 17.9. This routine, found in the accompanying files, determines the output of the

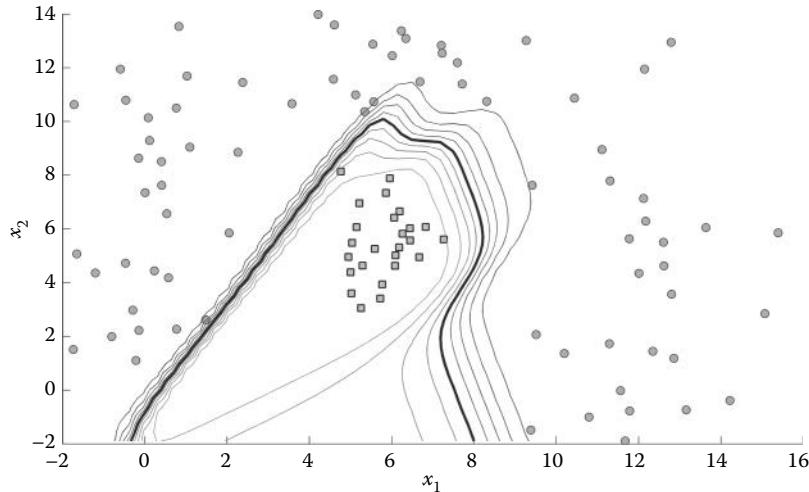


Figure 17.9 Boundaries found by the two-layer net in Example 17.3. The nonlinear boundary perfectly separates the two classes, but requires 1340 training epochs.

net over the range of input patterns by incrementing, in turn, each of the two input signals and evaluating the net's output for each input pair. The net response to the range of input patterns is then plotted using MATLAB's mesh and contour routines along with the training data set. The routine `net_eval`, when called with six input arguments, assumes the net is a two-layer net and plots the data points and determines sensitivity, specificity, and MSE that can be used to stop training. Both `net_boundaries` and `net_eval` are written to work with two-layer nets and with the three-layer nets introduced in the next section.

```

function [W1,b1,W2,b2] = net_learn_2(X,d,alpha,W1,b1,W2,b2)
% Train a two-layer network using the delta algorithm and
% back projection. Assumes only one output neuron
% Inputs
% X input variables a matrix
% d class associated with each variable (i.e. desired output)
% alpha leaning constant (scalar)
% W1 weights of hidden layer: The weights are a matrix where the
% column indicates the neuron number and row is
% input channel.
% b1 bias of layer 1: a vector of biases for the neurons
% in the hidden layer
% W2 weights of output layer neuron (a vector)
% b2 bias of layer 2: scalar of bias for the neuron in the
% output layer.
%
[r, ~] = size(X); % Nu. input patterns
[~, nu_hl] = size(W1); % Nu. weights in first layer
%
% Train net
for i=1:r
    [y_hl, der_hl] = net_layer(X(i,:),W1,b1,'s'); % First layer
    [y, der.ol] = neuron(y_hl,W2,b2,'s'); % Output layer
    e=d(i) - y; % Error
    %Compute local errors
    err.ol=der.ol * e; % Output neuron error

```

```

for k=1:nu_hl          % Now back-project
    % Calculate back-projected error and update weights in layer 1
    err_hl=der_hl(k) * err_ol * W2(k,1);
    W1(:,k)=W1(:,k)+alpha * err_hl * X(i,:)';
    b1(k)=b1(k)+alpha * err_hl;
end
W2=W2+alpha * err_ol*y_hl';      % Update weights &
b2=b2+alpha * err_ol;           % biases of
                                % output layer
end

```

Analysis

The first layer, consisting of four neurons, is implemented in the routine `net_layer.m` that sets up multiple calls to `neuron.m` and presents the output as a vector (including a vector of the derivatives to be used in training). These outputs (`y_hl`) become the inputs to the output neuron, again implemented using `neuron.m`. The output neuron produces the output, `y`.

The training program first determines the output of the hidden layer neurons, `y_hl`. This output is compared with the desired output, `d`, and the error, `e`, is determined. The error is reflected back through the output neuron by multiplying by the neuron's derivative, `der_ol`. This modified error, `err_ol`, is then used along with the output neuron weights, `w2`, to back project onto the four neurons in the hidden layer. After the hidden layer weights and biases have been updated, the output neuron's weights and biases are updated. This is done last, since it is the original weights (not the newly updated weights) that should be used in the back-projection algorithm. The results of training the two-layer net in Example 17.3 are shown in Figure 17.9. The training requires 1340 epochs before the error criterion is met: the difference in error between successive epochs should be < 0.00001 . The nonlinear boundary produced by the net results in perfect separation.

17.5 Three-Layer Nets

The evaluation and training concepts described above can be easily extended to nets of three or more layers, although considerably, more time is required to train these larger nets. In fact, there is no reason to go beyond three layers because as long as they have a sufficient number of neurons in each layer, they can generate boundaries of any desired complexity. As discussed at length in Chapter 16, it is easy to construct classifiers that are overtrained and so specific to the training data that they do not generalize well.

To implement a three-layer net, the output of the first hidden layer becomes the input to the second layer and the second layer neurons feed the output neuron(s). For training, Equations 17.12 and 17.13 now apply to both hidden layers whereas Equations 17.10 and 17.11 still apply to the output neuron. These approaches are illustrated in the three-layer training algorithm, `net_learn_3`, shown below.

```

function [W1,b1,W2,b2,W3,b3]=...
    net_learn_3(X,d,alpha,W1,b1,W2,b2,W3,b3,'type')
% Trains a three-layer network using the delta algorithm and
%       back projection.
% Inputs
%   X   input variables (matrix)
%   d   class associated with each variable (vector)
%   alpha leaning constant (scalar)
%   W1  weights of hidden layer 1: Same format as Example 17.3.
%   b1  bias of layer 1: Format as in Example 17.3

```

Biosignal and Medical Image Processing

```
% W2 weights of hidden layer 2
% b2 bias of layer 2:
% W3 weights of output layer 3 (single neuron assumed)
% b3 bias of layer 3: (scalar)
%      'type' neuron type (default = 's')
%
if nargin < 10
    type = 's';
end
[r,nu_inputs] = size(X);           % Number of input patterns
[dum,nu_hl1] = size(W1);          % Number first layer neurons
dum,nu_hl2] = size(W2);          % Number second layer neurons
% Train net
for i = 1: r
    [y_hl1, der_hl1] = net_layer(X(i,:),W1,b1,'type');    % 1st layer
    [y_hl2, der_hl2] = net_layer(y_hl1,W2,b2,'type');    % 2nd layer
    [y, der_ol] = neuron(y_hl2,W3,b3,'type');           % 3rd layer
    ei = d(i) - y;                                     % Output error
    %Compute local errors
    err_ol = der_ol * e(i);                           % Output neuron errors
    W2_old = W2;                                      % Save unmodified weights
    for k = 1:nu_hl2                                    % Backproject to layer 2
        err_hl2(k) = der_hl2(k) * err_ol * W3(k,1);
        W2(:,k) = W2(:,k) + alpha * err_hl2(k) * y_hl1';
        b2(k) = b2(k) + alpha * err_hl2(k);
    end
    for k = 1:nu_hl1                                % Now backproject to layer 1 neurons
        for m = 1: nu_hl2      % Responsibility, layer 1 neurons
            % Sum over all connections to layer 2 neurons
            credit(m) = err_hl2(m)*W2_old(k,m);
        end
        err_hl1 = der_hl1(k) * sum(credit); % Backprojected error
        % Update layer 1 neurons
        W1(:,k) = W1(:,k) + alpha * err_hl1 * X(i,:)';
        b1(k) = b1(k) + alpha * err_hl1;
    end
    W3 = W3 + alpha * err_ol*y_hl2';    % Update output layer
    b3 = b3 + alpha * err_ol;          % weights and bias
end
```

Analysis

Evaluation of a three-layer net requires only one additional call to the routine `net_layer` to evaluate the extra layer. Implementing the back-projection algorithm is a little more complicated than for a two-layer net. Updating the middle layer is the same as for the two-layer net in Example 17.3. However, since the input layer connects to all the neurons in the middle layer, the “responsibility” of this layer must be determined by summing over the product of the middle layer’s back-projected error times the middle neuron’s connecting weight (variable `credit` in `net_learn_3`). Again, the original (i.e., unmodified) weights are used in the back projection.

A three-layer net is implemented and trained in Example 17.4 using the training algorithm described above.

EXAMPLE 17.4

Train a three-layer net to classify a training set that consists of two classes arranged diagonally across from one another. Use a 100-point training set and a net with six neurons in the input and hidden layers. Evaluate the trained net on a test set of 400 points.

Solution

Set up a multiepoch training process using the delta learning algorithm as in Example 17.3, but use the back-projection training algorithm in routine `net_learn_3` as shown below. After training, use `net_eval` and `net_boundaries` to apply to the test data to evaluate the net.

```
% Example 17.4 Three-layer net example including evaluation
% using a test set.
%
alpha = .2;           % Learning constant
h11 = 6;              % Number of neurons in input layer
h12 = 6;              % Number of neurons in hidden layer
[Xt,dt] = gen_data2(4,[],'d');    % Generate training data (N = 100)
[X, d] = gen_data2(4,[],'d',400); % Generate test data (N=400)
% Generate initial weights for all layers
[r,nu_inputs] = size(X);
W1 = (rand(nu_inputs,h11)-1)*.25; % Initialize weights
b1 = (rand(1,h11)-1)*.25;        % and biases, input layer
W2 = (rand(h11,h12)-1)*.25;      % Initialize hidden layer
b2 = (rand(1,h12)-1)*.25;
W3 = (rand(h12,1)-1)*.25;         % Initialize output neuron
b3 = (rand(1,1)-1)*.25;
last_mse = 2;                   % Set initial mse large
%
for k3 = 1:200
    for k1 = 1:20
        [Xt,dt] = mix_data(Xt,dt);
        [W1,b1,W2,b2,W3,b3] = ...
            net_learn_3(Xt,dt,alpha,W1,b1,W2,b2,W3,b3);
    end
    [mse,sens,sp] = net_eval(X,d,W1,b1,W2,b2,W3,b3); % Evaluate net
    if abs(mse - last_mse) < 0.00001
        disp('Delta error minimum')
        break;
    end
    last_mse = mse;          % Update last mse
    disp([mse, sens, sp])
end
net_boundaries(Xt,dt,W1,b1,W2,b2,W3,b3);
[~, sensitivity, specificity] = net_eval(Xt,dt,W1,b1,W2,b2,W3,b3);
% Training data
.....display epochs and title for training data.....
net_boundaries(X,d,W1,b1,W2,b2,W3,b3);                                % Test data
[~, sensitivity, specificity] = net_eval(X,d,W1,b1,W2,b2,W3,b3); % Test data
.....display epoch and title for test data.....
```

Results

The main routine follows the format of the previous examples, generating the data, setting the initial weights and biases randomly, training the net, evaluating the net, and exiting when the change in error becomes very small. The results of Figure 17.10a show training data and the boundaries with a sensitivity of 96% (two class 0 points were misclassified) and

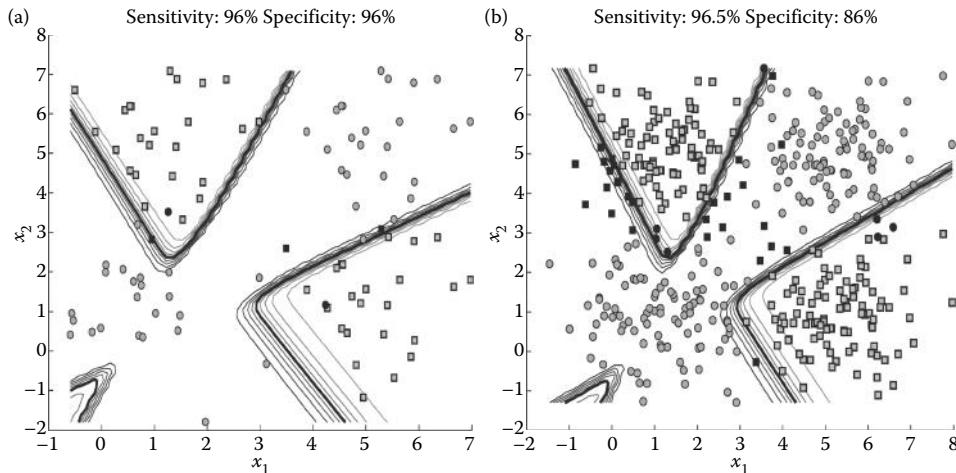


Figure 17.10 (a) Training set points and boundaries found by the three-layer net in Example 17.4. (b) Test set points found by the three-layer net in Example 17.4.

a specificity of 96% (two class 1 points were misclassified). The training period consisted of 2000 epochs when the epoch limit was reached. When evaluated on the test set, this classifier had an even higher sensitivity of 96.5% but a lower specificity of 86%.

17.6 Training Strategies

This section describes some of the strategies that are used during training to speed up the training process or to provide better generalization.

17.6.1 Stopping Criteria: Cross-Validation

The gradient method requires some sort of stopping criteria. Since we want to minimize the MSE, we might consider stopping when the MSE is minimum, but we do not know in advance what the minimum MSE will be. In the examples above, the criteria are based on the *change* in MSE error although there is also a maximum number of iterations imposed by the `for` loops that enclose the training algorithm. In these examples, this strategy works fairly well, but it has the potential to stop the training prematurely if an unusually flat region in the error surface is encountered. Remember that the goal is to train the net to be as general as possible, that is, to do as well as possible on the test data set, not just to fit the training data. It is also possible to overtrain a net so that it continues to improve on the training data, but actually performs worse on the test data set.

Cross-validation is a criterion that attempts to stop the training at a point where the net is maximally generalized. The idea is to set aside a small portion of the training data set to be used as a mini test set. This small data set, traditionally 10% of the total training data, is termed the *validation* data set and is not used in training the net; rather, it is used for an ongoing evaluation of the generality of the net during training. If the improvement during training is general, then *both* the training set and validation set error will decrease. At some point, the validation error may increase even though the training error continues to decrease and that is the point at which training should be stopped. Example 17.5 uses cross-validation to train a three-layer net for the best generalization.

EXAMPLE 17.5

Train a three-layer net for maximum generalization by using a validation data set. The data set should have a few overlapping data points so that it is not entirely separable.

Solution

Modify Example 17.4 to include cross-validation and use this as a stopping criterion. Use a training set consisting of 100 input patterns and a validation set consisting of 24 input patterns. Use `net_eval` to evaluate the validation data set as well as the training data set every 20 epochs and plot both errors as a function of epoch. Stop when the validation MSE begins to increase. To show the increase in validation set error, it will be necessary to continue the iterations beyond the validation error minimum.

```
Example 17.5 Cross-validation example. Trains a three-layer neural
%
.....Initializaton as in Ex 17.4.....
%
% Generate training data with some overlapping points
[Xt dt] = gen_data2(4.75, [], 'd'); % Test data N=100
[Xv dv] = gen_data2(4.75, [], 'd', [], 24); % Validation data N=24
% Generate initial weights for both layers
..... Same code as in Example 17.4 ......

last_mse_validation = 2; %Initial validation MSE
for k3 = 1: 110
    for k1 = 1:20
        [W1,b1,W2,b2,W3,b3] = ...
            net_learn_3(Xt,dt,alpha,W1,b1,W2,b2,W3,b3);
    end
    % Save errors generated by the evaluation routines
    mse_training(k3) = net_eval(Xt,dt,W1,b1,W2,b2,W3,b3);
    mse_validation(k3) = net_eval(Xv,dv,W1,b1,W2,b2,W3,b3);
    if mse_validation(k3) > last_mse_validation
        disp('Validation error increase')
        % break; % Take additional records
    end
    last_mse_validation = mse_validation(k3);
end
.....plot boundaries and title.....
figure; % Plot errors (learning curve)
x_plot = (1:length(mse_training))*20; % Plotting vector
plot(x_plot,mse_training,:'k', x_plot,mse_validation,'k');
```

The plots of the two error curves are shown in Figure 17.11. After about 320 epochs, the validation set error curve (solid line) starts to increase whereas the training set error (dashed line) continues to decrease. Cross-validation suggests that training should be stopped at around 360 iterations (Figure 17.11).

The performance on the training data is shown in Figure 17.12. The training set is correctly classified by the nonlinear boundaries in this example (Figure 17.12).

17.6.2 Momentum

As you will find when working some of the problems, it can take many iterations and a fair amount of computational time to train a three-layer net. One technique for speeding up the learning process is called *momentum learning*. The idea is that if the weights are rapidly changing in a consistent direction, they are on a steep downward slope and should be encouraged to move even faster. Conversely, if the weights are near the true (i.e., global) minimum, you want the weights to change slowly so as to find the absolute minimum. This is illustrated graphically in Figure 17.13.

In the left-hand figure, the weight values (condensed to a single vertical dimension) are on a steep gradient and should move quickly toward a minimum, but if their step sizes increase,

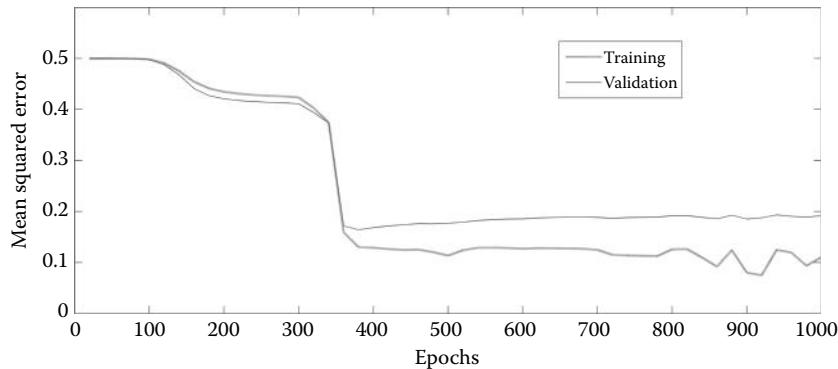


Figure 17.11 Learning curves for both training and test sets of Example 11.5. The validation set (solid lower curve) reaches the minimum error after approximately 360 iterations although the training set error (dashed upper curve) continues to decrease.

they may have enough “momentum” to move beyond any small local minimum. On the other hand, if weight values are near a (hopefully global) minimum, the changes should be small; so, the weights do not continuously bounce around the minimum but find the optimal set of values.

The momentum concept is easy to implement: simply alter Equations 17.13 and 17.14 to include an additional term that depends on the signed difference of the last two weight values.

$$w_i(n+1) = w_i(n) + \delta e_i(n)x_i + \eta(w_i(n) - w_i(n-1)) \quad (17.16)$$

$$b_i(n+1) = b_i(n) + \delta e + \eta(b_i(n) - b_i(n-1)) \quad (17.17)$$

where η is a constant chosen empirically. The momentum approach is illustrated in Example 17.6.

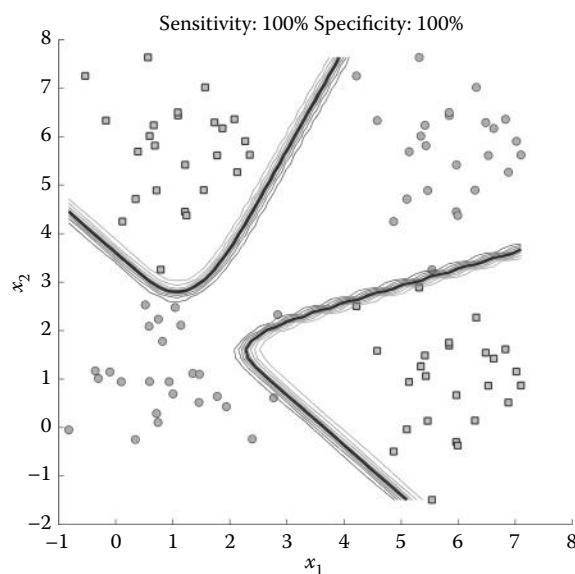


Figure 17.12 Boundaries and performance of the three-layer net in Example 17.5 on the training data ($N = 100$). All the points in the training set are correctly classified.

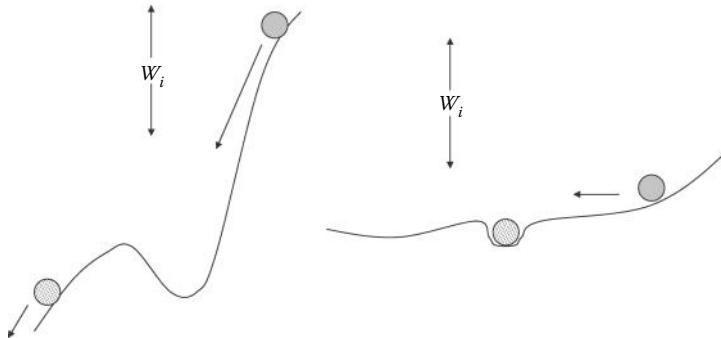


Figure 17.13 Graphical illustration of the *momentum* technique for adjusting weights during neural net training. Weights that change rapidly should be given an additional push so that they converge faster and may even bypass local minima. Weights changing slowly are likely to be near the minimum and should slowdown to find the optimal values.

EXAMPLE 17.6

Train the three-layer net used in Examples 17.4 and 17.5 to a diagonal pattern with some possible overlap. Use the standard back-projection methods from the last two examples: with and without momentum. Compare the difference in output and in the number of epochs.

Solution

Modify the training program `net_learn_3` to include momentum. This requires storing the weights of past cycles to determine the weight slope.

```
% Ex. 17.6 Momentum example. Trains a three-layer neural net
% with and without momentum.
%
alpha=.1           % Learning constant
.....Initialize weights. Save copy for training without momentum.....
% Train with momentum
nu=.25;           % Momentum constant
last_mse=2;
for k3=1:400
    for k1=1:20
        [X,d]= mix_data(X,d);
        [W1,b1,W2,b2,W3,b3] = net_learn_3_mom(X,d,alpha,nu,... % Training
            W1,b1,W2,b2,W3,b3);
    end
    [mse, sen, sp] = net_eval(X,d,W1,b1,W2,b2,W3,b3);
    mse_savel(k3)=mse;           % Save mse for plotting
    if abs(mse-last_mse) < .00001 && mse < .12 % Stop criterion
        disp('Delta error minimum')
        break;
    end
    last_mse=mse;
    disp(mse)
end
net_boundaries(X,d,W1,b1,W2,b2,W3,b3);
%
.....Restore original weights and train without momentum.....
nu=0;             % Momentum constant zero
for k3=1:400
```

Biosignal and Medical Image Processing

```

.....Same as above loop.....
end
net_boundaries(X,d,W1,b1,W2,b2,W3,b3);
%
% Plot training curves.
x1=(1:length(mse_save1))*20;      % With momentum
plot(x1,mse_save1,'k');
x2=(1:length(mse_save2))*20;      % Without momentum
plot(x2,mse_save2,:k')
.....labels.....

```

The training routine `net_learn_3_mom` has been modified to store the weights (and biases) before updating and to use these stored values along with the current weights in the next update, as indicated by Equations 17.15 and 17.16.

```

function [W1,b1,W2,b2,w3,b3]=net_learn_3_mom(X,d,alpha,nu, ...
                                                W1,b1,W2,b2,w3,b3)
% Modification of net_learn_3 that uses momentum training.
% The arguments are the same as in net_learn_3 except for the
% addition of the momentum constant nu
% Preserve old weights and biases
persistent W1_old b1_old W2_old b2_old w3_old b3_old;
if isempty(W1_old)          % Check if first time around
    W1_old=W1;             % Initialize save weights and biases
    b1_old=b1;              % if first time around
    W2_old=W2;
    b2_old=b2;
    w3_old=W3;
    b3_old=b3;
end
%
[r,nu_inputs]=size(X);           % Determine nu. input patterns
[~,nu_hl1]=size(W1);            % Determine nu. 1st layer neurons
[~,nu_hl2]=size(W2);            % Determine nu. 2nd layer neurons
% Train net
for i=1:r                         % Calculate net output
    [y_hl1, der_hl1]=net_layer(X(i,:),W1,b1,'s');
    [y_hl2, der_hl2]=net_layer(y_hl1,W2,b2,'s');
    [y, der_ol]=neuron(y_hl2,w3,b3,'s');
    e(i)=d(i)-y;                  % Calculate the error
    %Compute local errors
    err_ol=der_ol * e(i);         % Output neuron local errors
    for k=1:nu_hl2                 % Now back-project to layer 2
        err_hl2(k)=der_hl2(k) * err_ol * W3(k,1); % Local error
        % Calculate weight  $\delta$  including momentum
        delta=alpha * err_hl2(k) * y_hl1' + ...
               nu*(W2(:,k) - W2_old(:,k)); % Add momentum
        W2_old(:,k)=W2(:,k);          % Save weights before update
        W2(:,k)=W2(:,k)+delta;       % Update weights
        delta=alpha * err_hl2(k) + nu*(b2(k) - b2_old(k));
        b2_old(k)=b2(k);             % Save bias before update
        b2(k)=b2(k)+delta;
    end
    for k=1:nu_hl1                 % Back-project to layer 1
        for m=1: nu_hl2            % Get responsibility layer 1 neurons
            % Sum over all layer 2 neurons, m
            credit(m)=err_hl2(m)*W2_old(k,m);
        end
    end

```

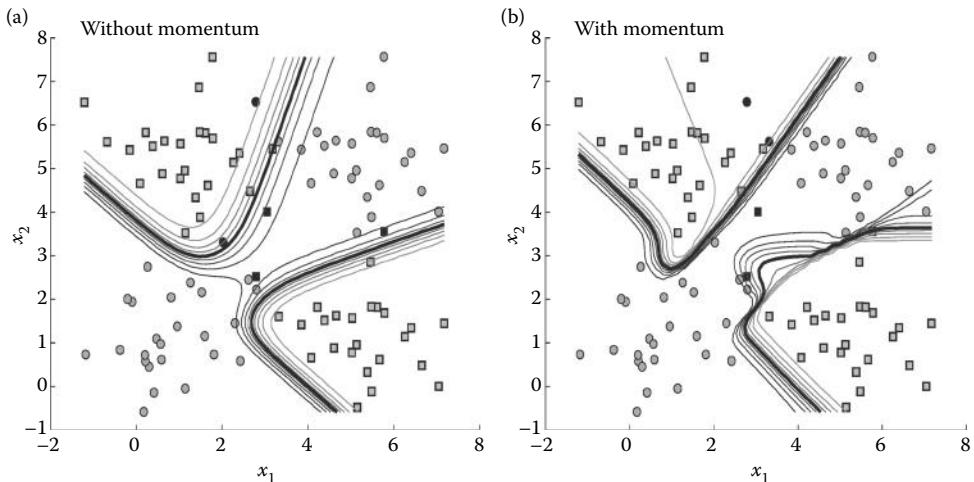


Figure 17.14 Boundaries found by a three-layer net using back projection with and without momentum. The boundaries are different but both lead to the same classification errors (filled symbols).

```

err_hl1=der_hl1(k) * sum(credit); % Local error
    % Calculate weight δ including momentum
delta=alpha * err_hl1 * X(i,:)'+nu*(W1(:,k)...
    - W1_old(:,k));
W1_old(:,k)=W1(:,k);           % Save weights before update
W1(:,k)=W1(:,k)+delta;         % Update weights
delta=alpha * err_hl1+nu*(b1(k) - b1_old(k));
b1_old(k)=b1(k);              % Save bias before update
b1(k)=b1(k)+delta;            % Update bias
end
delta=alpha * err_ol*y_hl2' +nu*(W3 - W3_old); % Output layer
W3_old=W3;
W3=W3+delta; % Update weights and biases in output layer
delta=alpha * err_ol+nu*(b3 - b3_old);
b3_old=b3;
b3=b3+delta;
end

```

Results

The final decision boundaries found with and without momentum show one less error for the former (Figure 17.14) and the learning curve shows a substantial reduction in the number of epochs required to reach the minimum error when momentum is used to train the weights (Figure 17.15). The advantage of momentum is further explored in the problems.

17.7 Multiple Classifications

Neural nets can be extended to handle multiple classes much the same way as the discriminant and SVM methods of Chapter 16. Multiple subnets are placed in parallel, each with its own output neuron. This output neuron is specific to one class. To define the correct class in the training data, the variable that specifies the correct class becomes a matrix following the format of Equation 16.3: the associated class is indicated by 1 while the other classes have 0. An example of a two-layer net that classifies four classes is shown in Figure 17.16. In this net, each output neuron receives two inputs from the input-layer neurons.

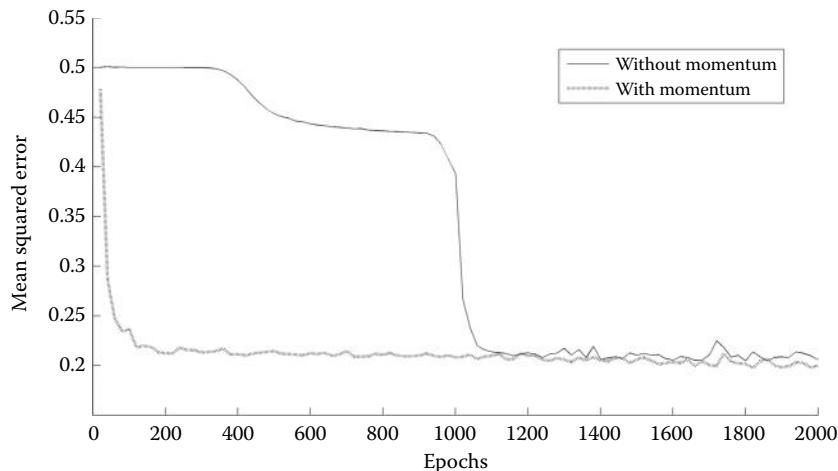


Figure 17.15 The *learning curve* (MSE versus number of epochs) obtained while training a three-layer net without momentum (solid line) and with momentum (heavy dashed line). The use of momentum decreases the number of training epochs required to reach the minimum error, although in this case, they both reach approximately the same level of error.

EXAMPLE 17.7

Identify four different classes using a two-layer net. The classes should be close, but should not be overlapping. This example uses the routine `gen_dat4` introduced in Chapter 16 to produce the four-class training data. This routine produces a matrix that indicates the correct class following the protocol given in Equation 16.3.

Solution

Modify the example of a two-layer, two-class net used in Example 17.3 so that four nets are arranged in parallel. Each of the four “sub-nets” consists of two input-layer neurons and one output neuron as shown in Figure 17.16. All input neurons receive the two input signals. Each output neuron is responsible for classifying one of the four classes. The same training routine

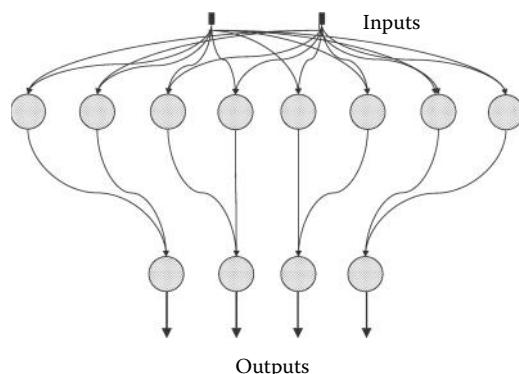


Figure 17.16 Four two-layer nets are arranged in parallel to identify four different classes. Each subnet consists of two neurons in the input layer and a single output neuron. Each subnet is trained to identify one of the classes given the two inputs and the appropriate desired output.

used in Example 17.3 (`net_learn_2`) can be used in a loop to train each of the subnets. The boundary display program, `net_boundary_4`, is a slight modification of `net_boundaries` that plots the multiple classes and boundaries in different symbols and grayscales.

```
% Ex. 7.7 Four-class identification using two-layer nets.
% Trains a two-layer neural net to identify four classes
% Similar to Example 17.3
%
alpha=.05; % Learning constant
h1l=8; % Number of neurons in layer 1
ol=4; % Number of neurons in output layer
[X, D]=gen_data4(2.75); % Generate four-class data
.....Generate initial weights and biases as in Ex. 11.3.....
last_mse=4; %Initial mean squared error
for k3=1:50
    for k1=1:20
        for k2=1:4 % Uses 4 two-layer nets in parallel
            j=(k2-1)*2+1;
            [W1(:,j:j+1),b1(j:j+1),W2(:,k2),b2(k2)]=...
                net_learn_2(X,D(:,k2),alpha, ...
                W1(:,j:j+1),b1(j:j+1),W2(:,k2),b2(k2));
        end
    end
    mse=net_eval_4(X,D,W1,b1,W2,b2); % Evaluate net
    if abs(sum(mse)-last_mse)<.0001
        disp('Delta error minimum') % Exit loop if change
        break; % in mse < .0001
    end
    last_mse=sum(mse);
    disp(mse)
end
net_boundaries_4(X,D,W1,b1,W2,b2); % Display boundaries
```

Results

The program uses `net_learn_2` in a loop to train the four subnets. During each pass through the loop, both the weights and biases and the desired response must be selected for the appropriate subnet. These parallel nets learn fairly quickly and, in combination, can accurately separate four closely spaced training sets as shown in Figure 17.17.

17.8 Multiple Input Variables

As with the classifiers in Chapter 16, ANN classifiers easily extend to more than two variables. Most of the routines can be used with as many inputs as desired without modification. Only the plotting functions need to be changed. If a single neuron net is used with three variables, the routine `net_eval` will plot the boundary. If more than one neuron is involved (i.e., more than two classes), `net_eval` can still determine the sensitivity and specificity, but does not plot the boundary.

The next example classifies three-variable data using a single neuron. Since the net consists of only one neuron, the decision boundary is a plane, but recall that this boundary is adequate if the data are linearly separable. The example uses `lms_learn` to train the net and is very similar to Example 17.2 except for the routine used to generate the data and the use of `net_eval` to plot the results and find the sensitivity and specificity. One of the problems applies a two-layer net (`lms_learn2`) to a three-variable data set and uses `net_eval` to determine the sensitivity and specificity of both the training and test set data.

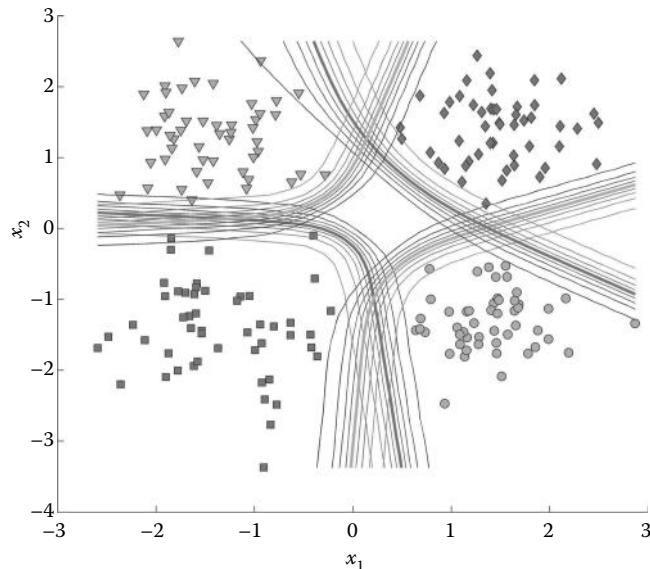


Figure 17.17 Four different classes represented by squares, circles, diamonds, and triangles are accurately identified by the four output neurons in Figure 17.16. The net _ boundary routine was modified to plot the decision boundaries established by each of the output neurons. These decision boundaries are shown as heavier lines.

EXAMPLE 17.8

Generate a three-variable training set ($N = 100$) and use it to train a single neuron. Use a large separation so that the classes are linearly separable.

Solution

Generate the data using a modification of gen _ data2 that produces three-variable data sets. Use lms _ learn to train a single neuron and net _ eval to plot the data. This routine will also plot the decision plane and determine the sensitivity and specificity.

```
% Example 17.8. Example of 3-variable training
%
alpha = .05; % Learning constant
[X, d] = gen_data3D(6, 45); % Generate 3-variable data
[r, c] = size(X);
w = (rand(c, 1) - .5)/4; % Initialize weights and bias
b = (rand - .5)/4;
last_mse = 2;
for k3 = 1:300
    for k1 = 1:10
        [w, b] = lms_learn(X, d, alpha, w, b); % Train single neuron
    end
    mse = net_eval(X, d, w, b); % Evaluate neuron
    if abs(last_mse - mse) < .00001
        disp('Delta error minimum')
        break;
    end
end
```

```

last_mse = mse;
disp(mse)
end
[mse, Sensitivity, Specificity] = net_eval(X,d,w,b) % Evaluate

```

Results

The results from this example are shown in Figure 17.18. Since the two classes are linearly separable, the single neuron net provides a sensitivity and specificity of 100% for this three-variable data set.

17.9 Summary

ANNs evolved from attempts by biomedical engineering pioneers to model biological neurons. The McCullough–Pitts neuron performs a weighted sum on its inputs just as the cell body is thought to do on dendritic inputs. As in an actual neuron, this sum is presented to a threshold element that determines if an output (i.e., an action potential) is generated. An early pattern recognition scheme, the perceptron, used a set of parallel McCullough–Pitts neurons in conjunction with specialized preprocessing elements to identify complex patterns. More complicated neural nets were developed that featured more neurons and up to three layers. The number of neurons combined with the number of layers determines the complexity of the net and corresponds to “machine capacity” as described in Chapter 16. Three-layer nets are capable of very complex decision boundaries and three layers are sufficient to generate a decision boundary of arbitrary complexity given enough neurons in each layer. The final, or output layer, has one neuron for each class being identified and becomes active when that class is sensed.

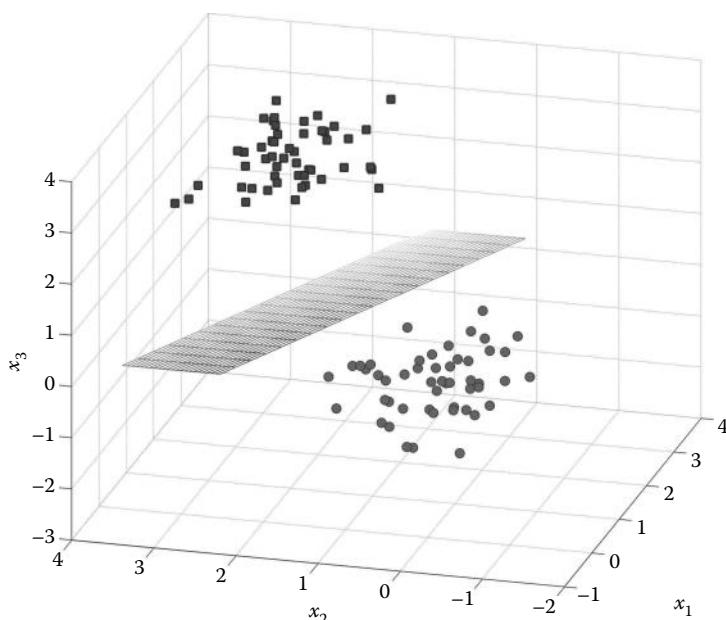


Figure 17.18 Plot showing the decision plane generated by a single neuron net applied to a three-variable training set. For these training data, the single neuron net provides perfect classification.

Biosignal and Medical Image Processing

One of the major breakthroughs in ANNs was the training strategy known as back projection. In this strategy, classification errors propagate backwards through the net to adjust the weights of neurons in each layer. Weights update take into account the error and the degree to which a neuron contributed to the output. Neurons that were in pathways that contributed more to the output experience larger weight changes. Improvements in training strategies include the use of a validation set and the concept of momentum.

PROBLEMS

Many of these problems require a considerable amount of computational time to converge or reach the epoch limit, especially problems that involve two- or three-layer nets. Patience is required! In some problems, files are provided that contain suggested initial conditions, since not all initial conditions produce the same speed of convergence. When available, these files are identified in the problems. For all problems, you can use a conservative learning constant, alpha, of 0.1, but in many problems, you will get much faster convergence with learning constants as high as 1.0.

- 17.1 Load the data file `prob17_1.mat` that contains the training set data, `Xt` and `dt`, and a larger test set, `X` and `d`. These data sets were generated from two Gaussian distributions and have some overlap (i.e., they are not linearly separable). Use the training set data to train a single neuron with a sigmoid nonlinearity. Train using `lms_learn` with an `alpha = 0.1` and train until the change in error is <0.0001 or 2000 epochs, whichever occurs first. After training, use `net_eval` again with the test data set to evaluate the net. Show the final boundary and the sensitivity and specificity of the test set (it is convenient to put the sensitivity and specificity in the title of the test set data plot). Repeat using a minimum error change of 0.00001. Does further training improve the classifier's performance?
- 17.2 Repeat Problem 17.1 using the data file `prob17_2.mat` and the hyperbolic tangent function. This file contains the same training and test set as `prob17_1.mat`, except the classes are identified as -1 and +1 to be compatible with the hyperbolic tangent function. Use only one stop criteria, the one that used in Problem 17.1 (an error <0.0001 or 2000 epochs). After training, use `net_eval` again with the test data set to evaluate the net. Show the final boundary and the sensitivity and specificity on the test set plot. Does the hyperbolic tangent function change the sensitivity and/or specificity of the classifier when applied to the test set?
- 17.3
 - a. Repeat Problem 17.1 using the data in `prob17_1.mat` except train and evaluate using a two-layer net containing a hidden layer of four neurons. Again, use only one stop criterion: an error <0.0001 or 2000 epochs. Compare the sensitivity and specificity with that obtained in Problem 17.1. If desired, the initial weights can be found in `initial_weights_3A.mat`.
 - b. Repeat (a) using a two-layer net containing a hidden layer of eight neurons and compare the sensitivity and specificity of the net with the less-complex less-capacity net in (a). If desired, the initial weights can be found in `initial_weights_3A.mat`. What do you conclude from this comparison and the results form Problem 17.1 or 17.2?
- 17.4 Repeat Problem 17.1 using the data `prob17_1.mat`, except train and evaluate using a three-layer net containing four neurons in each of the two hidden layers. Continue the training until the change in error is <0.0001 or 2000 epochs, whichever comes first. Plot the boundaries using `net_boundaries`. If desired, the

- initial weights can be found in `initial_weights_4.mat`. Compare the sensitivity and specificity with that obtained in Problems 17.1 and 17.3(a) and (b).
- 17.5 Train and evaluate the data `prob17_5.mat` using both a single neuron and a three-layer net as in Problem 17.4. The training and test data set are in the same format as in the preceding problems. For the three-layer net, the initial weights can be found in `initial_weights_5B.mat`. Compare the performance as indicated by sensitivity and specificity and determine if the single neuron net has sufficient machine capacity for this data set.
- 17.6 a. Train and evaluate the data `prob17_6.mat` using a single neuron and a three-layer net as in Problem 17.4. The training and test set data consist of one class surrounded on three sides by the other class and are in the same format as in the preceding problems. Train all three sections using 2000 epochs.
- b. Train and evaluate on a two-layer net having four input layer neurons using the same data set. For the two-layer net, the initial weights can be found in `initial_weights_6B.mat`.
- c. Train and evaluate on a three-layer net having four neurons in both the hidden and input layers. Compare the performance of the three nets as indicated by sensitivity and specificity and determine which is the most appropriate (i.e., has the appropriate machine capacity) for the data set. (Note that this problem can take 5–10 min or more to run.) For the three-layer net, the initial weights can be found in `initial_weights_6C.mat`.
- 17.7 Use the training set in `prob17_7.mat` to compare two configurations that have the same number of the total weights: a two-layer net with eight weights in one hidden layer and a three-layer net with four weights in each of the two hidden layers. Compare the sensitivity and specificity in response to the training session as well as the test session. The training data set is in variables X_t and d_t and the test data set is in variables X and d . In addition to sensitivity and specificity, plot the MSE as a function of epoch number (i.e., the learning curve) and note the difference in rates of convergence. Comment on the behavior of the learning curve for the three-layer net for these data. (Suggested initial conditions: For the two-layer net: `initial_weights_7A.mat` and for the three-layer net: `initial_weights_7B.mat`.)
- 17.8 The file `prob17_8.mat` contains a training set ($X_t, d_t, N = 100$), a validation set ($X_v, d_v, N = 50$), and a test set ($X, d, N = 500$). Use cross-validation to train a two-layer net with eight input neurons. Run the training initially for 2000 epochs and note the sensitivity and specificity. Plot the learning curve for both the training and validation data. Then rerun the training starting at the same initial weights and biases for the number of epochs that occurred before the training and validation set diverged. Compare the sensitivities and specificities for the two training lengths. (Suggested initial conditions: `initial_weights_8.mat`.)
- 17.9 Use the data in `prob17_7.mat` to train a three-layer net having six neurons in the input layer and four neurons in the hidden layers. Train the net with and without momentum. Train for 2000 epochs. Plot the curve of MSE versus epoch (i.e., the learning curve) as well as compare the performance (sensitivity/specificity) between the two training schemes. (Suggested initial conditions: `initial_weights_9.mat`.)
- 17.10 Use the data in `prob17_8.mat` to train a three-layer net having four neurons in the input and hidden layers. Train the net with momentum using an η of 0.5 and

Biosignal and Medical Image Processing

- 0.05. Train for 2000 epochs. Plot the boundaries produced by each training session and the curve of MSE versus epoch. Compare the performance with regard to sensitivity as well. (Suggested initial conditions: `initial_weights_10.mat`.)
- 17.11 Use the data in `prob17_8.mat` to train two three-layer nets: one having four neurons in each layer and the other with eight neurons in each layer. Train for 2000 epochs using momentum. Plot the boundaries produced by each training session and the curve of MSE versus epoch. Compare the performance with regard to convergence and sensitivity and specificity. Does the smaller net have sufficient machine capacity to represent the data? (Suggested initial conditions: `initial_weights_11A.mat` for the first net and `initial_weights_11B.mat` for the second net.)
- 17.12 Train a two-layer net on the data set in `prob17_12.mat` that has three classes. These training data ($N = 180$) are in matrix `Xt` and the classes are specified in matrix `Dt` using the same format as in Example 17.7. Evaluate after training using the training set in `X` and `D` ($N = 600$). Use `net_boundaries_m` to plot the boundaries after training. Given that there are 200 data points in each of the three classes, calculate the sensitivity and specificity of each class using the errors shown in the boundary plot. (You will have to calculate sensitivity and specificity manually. The errors are plotted in red.)
- 17.13 Load the file `prob17_13.mat` that contains three-variable training ($N = 100$) and test ($N = 400$) data sets. Train on a two-layer net for up to 2000 epochs or until the change in error is < 0.00001 . Use six neurons in the input layer. There is no program that plots the *boundaries* of three-variable data sets for multilayer nets, but you can plot the points and evaluate the sensitivity and specificity of both the training and test set data using `net_eval`. (Suggested initial conditions: `initial_weights_13.mat`.)

Appendix A: Numerical Integration in MATLAB

The MATLAB function `ode45` is a first-order differential equation solver that can be used to solve first-order ordinary differential equations (ODE) or a system of first-order differential equations. The function uses the Runge–Kutta algorithm, a fourth-order numerical integration technique that is highly accurate, to solve differential equations numerically. Solver `ode45` is one of several first-order differential equation solvers in MATLAB, but is the most general and works well on a wide variety of equations. Throughout this tutorial, we will be addressing differential equations that are a function of time and thus our example variables refer to time; however, the procedure is easily altered for differential equations that are functions of some other independent variable, so long as they are a system of first-order ODE.

MATLAB uses a generic structure for its numerical differential equation solvers. The same calling structure is used for several different functions, including the one discussed here, `ode45`. Each solver is optimized for different types of differential equations, but since the calling format is similar for all of them, generally all you need to do to change the solver is to replace the function name with a different solver. The function call is structured as:

```
[t,sol] = ode45(@func,tspan,initial_values,options,variable_arguments);
```

The outputs are: `t`, a vector of time samples, and `sol`, a matrix containing solutions to the differential equations. Specifically, `sol` is a matrix where each column corresponds to the solution for a system of first-order ODE (by solution we mean each column contains the variable that is being differentiated in the differential equation system). Output `sol` has the same number of rows as time output `t`. So, for example, if a system of three differential equations is solved for 10 time steps, output `sol` will be a 10×3 array and output `t` will be a 1-column 10-row array. In addition to these two outputs, `ode45` has other optional outputs that are more advanced and not typically needed. The reader should consult the MATLAB documentation for more information on these outputs. Here, we will move on to the inputs.

The inputs to `ode45` are a little tricky and can involve many advanced options. We will describe here the basic inputs needed for most typical differential equations. For more advanced uses of `ode45`, consult the MATLAB documentation. The input `func` is the filename of a function that contains the differential equation system to be solved. Prefacing `func` with the `@` symbol is necessary for `ode45` to work properly. The structure of `func` is nuanced and will be discussed separately. Input `tspan` is an input that can take one of two forms. It can either be a length-2 vector that contains the begin and end times, or it can take the form of a complete time vector. In either case, the elements of `tspan` must be in increasing order and may not contain repeated values. If `tspan` has only two elements, MATLAB will solve the differential equation using the default number of time steps. If `tspan` is larger than three elements, the outputs of `ode45` (`sol` and `t`) will correspond to the values of `tspan` (`tspan` will actually be identical to `t` in this case). The next input, `initial_values`, is an array that contains the initial values of the system (the first row of output `sol` will be identical to the input `initial_values`). This must have the same number of elements as the number of equations in the first-order system is to be solved, and they must also be ordered in the same arrangement as the system of differential equations is to be solved. The rest of the inputs to `ode45` are optional; however, we need some of them in our examples throughout Chapters 10 and 11. Input options allow for setting of several parameters for the equation solver, mostly related to error tolerance. Typically

Appendix A

the default solver options work well and in this chapter we leave this input blank by setting it to the empty set `[]`. Finally, `ode45` allows equation parameters and constants to be passed to the equation defined in routine `func`. There can be any number of parameters passed along, but they must occur in at least the 6th argument of `ode45`. In Example 10.1, we use these arguments to specify the force and damping factors (b and k) of our differential equation.

The function `func` is a user-defined function that describes the system of first-order ODE to be solved (of course, `func` can also describe a single ODE that is not a system). As noted above, this defining function is referenced as one of the inputs to `ode45`.^{*} In Example 10.1, this function is named `pend` and it contains the system of first-order ODE that describes the motion of a pendulum. The differential equation function header must conform to several specific requirements. The function header follows the format:

```
function d_var = func(t, diff_variable, variable_arguments) .
```

This function defines a series of differential equations of the form: $\dot{y} = f(t, y)$. The construction of this function may appear confusing, because even though we need to compose function `func` to define the differential equation to be solved, the user does not ever call this function directly and does not need to define all the inputs.

The function output, `d_var`, is a column vector that defines the differential series of equations, $\dot{y} = f(t, y)$. Each row of `d_var` is an equation in the system. There is no particular order needed for `d_var`; however, the order of each equation needs to be consistent with the order of the initial conditions in the variable `initial_value` and input `diff_variable`. It is required by MATLAB that the output variable `d_var` is a column vector.

The first two inputs to `func` are variables that are created by `ode45`. The user can specify the names of either of these variables, but they must be used to represent the independent variable and the state variables of the differential equation. The first variable that is defined by `ode45` and used as an input to `func` is `t`, in our example it is the current time of the simulation. This time is distinct from the time variable `tspan` used as input to `ode45`. (MATLAB uses optimized values of `t` to solve the differential equation even though the outputs will correspond to `tspan`.) When we write the differential equation in the body of `func`, we will use `t` anywhere that the differential equations contain the time variable `t`. The second variable that is defined by `ode45` is `diff_variable`. The use of `diff_variable` is not particularly intuitive and needs to be used carefully.

The second input, `diff_variable`, is a vector of variables (the y variables) for which every element represents one of the state variables of the corresponding differential equation of the system. In our example, the equation variables are θ and ω ; however, `ode45` does not require the elements contained within `diff_variable` to be related, they must merely be the differentiated value in each equation within the system of equations to be solved. Keep in mind that `ode45` can only solve equations differentiated with respect to one value (the definition of a first-order ODE) only even though the system itself may contain equations of different variables.

The function used to define the differential equation for Example 10.1 is `pend`. This defines Equations 10.2 and 10.3 in MATLAB format.

```
function d_theta_omega = pend(t, theta_omega, b, k)
%
d_theta_omega = [theta_omega(2);
- sin(theta_omega(1)) + k*sin(t) + b*theta_omega(2)];
```

* We will see a similar strategy used in MATLAB's Image Processing Toolbox to define nonlinear filters applied to images.

In our example function, pend, diff _ variable is named theta _ omega because those are the names of the variables of the differential system that theta _ omega represent. We have named the output variable, the one that contains the differential equation system, d _ var. Because the first equation (and row of d _ var) in the system defines $\dot{\theta}$ and theta _ omega(1) represents the angle θ . Likewise, the second equation (and row of d _ var) defines $\dot{\omega}$ and theta _ omega(2) represents the angular velocity ω . [J1] When we translate the differential equations in Equations 10.2 and 10.3 into a form that MATLAB can interpret, we need to use theta _ omega(1) in place of θ and omega(2) in place ω .

The third input argument(s), variable _ arguments, are optional parameters carried through from the ode45 function call. In our example case, they are the constants b and k , the damping and forcing factors. If we intend to use b and k in our example, they either need to be defined explicitly in the script or used as optional arguments.

To solve the differential equation system represented by Equations 10.2 and 10.3, we call ode45 with the command,

```
[t,sol] = ode45(@pend,tspan,initial_values,[],b,k);
```

This calls ode45 and uses it to solve the differential equation described in pend for a range of values described in vector tspan and with initial conditions described by initial _ values. The empty brackets [] tell MATLAB to use the default ode45 options and the optional parameters b and k are constant values needed for function pend.

In general, using ode45 with the default parameters gives a good result for many differential equations. If the result is unsatisfactory (the solutions give unexpected or impossible results), a different solver such as ode15 or ode15s can be used. Since these have similar calling structures to ode45, using them instead of ode45 is a simple matter of changing the solver name. The same logic applies to the default parameters of ode45, such as the relative and absolute tolerances. The default tolerances generally work well and do not need to be changed unless there is a specific reason.

Appendix B: Useful MATLAB Functions

These functions can also be found in the accompanying MATLAB files. These functions are used in Chapter 10. Function descriptions are found in the associated comments.

Function lorenzeq:

```
function yprime = lorenzeq(t,y)
% Function containing the Lorenz equations. This is used with ODE45
% The rows of yprime correspond to Equations 10.8 10.9 and 10.10
% y(1) is x, y(2) is y, and y(3) is z.

% The Lorenz equation gives chaotic solutions for these parameter values
sigma=10.;
R=28.;
Beta=8./3.;

yprime=[ sigma.* (y(2)-y(1)); % dx/dt
y(1).*(R-y(3))-y(2); % dy/dt
y(1).*y(2)-Beta*(y(3))]; % dz/dt
```

Function max_lyp:

```
function [lambda,S_mean,kk]=max_lyp(fx,m,tau,fs,radius)
% Function to compute the maximum lyapunov exponent using
% the method given by Kantz et al.
% fx is the data to be analyzed, m is the embedding dimension,
% tau is the embedding delay, fs is the sample rate, and radius is an
% optional argument to determine the nearest neighbor search radius

count=0; % Initial count set to zero

Mx=delay_emb(fx,m,tau); % Delay embedding
[r, ~ ]=size(Mx); % Rows and columns of Mx
time=linspace(0,r/fs,r); % Create time vector

test_length=floor(.5*r);

if nargin<5
    radius =0.1*std(fx); % Default Radius used
end
% Otherwise use the radius that is user inputted

% First loop, find the nearest neighbors of each point
% since we need at least test_length numbers of points
% only loop from 1 to (r-test_length)
N=r-test_length; % Number of phase space points
```

Appendix B

```

S_n=zeros(test_length+1,1); % Initialize S_n for speed
S_mean=zeros(test_length+1,N);
countjj=0; % Initialize counter

for k=1:N
    x_t=Mx(k,:); % Get test point
    x_tvec=repmat(x_t,r,1); % Use rep mat so test point can
    % be performed as a vector
    distance=sqrt(sum((x_tvec-Mx).2,2)); % Get distances

    nearest_i=find(distance<radius); % Get index of
    % nearest neighbors
    nearest_i(nearest_i==k) = []; % Removes test point
    nearest_i=nearest_i(nearest_i<N); % Remove points in num near
    % longer than test length
    num_near=length(nearest_i);
    trajectory1=Mx(k:(test_length+k),:); % Trajectory of test point

    % Second for loop: compute the divergence
    diverge=zeros(test_length+1,1); % Initialize divergence
    if num_near~=0; % Calculation if neighbor exists
        count=count+1; % Number of nearest neigh.
        countjj=0;
        for jj=1:num_near % For 1 to number of nearest neigh.
            jj_ind=nearest_i(jj); % Starting point of NN trajectory
            countjj= countjj+1; % Count neighbors
            trajectory2=Mx(jj_ind:(test_length+jj_ind),:); % Get neighbor trajectory
            dive=sqrt(sum( (trajectory1 - trajectory2).2,2)); % Get divergence
            diverge=diverge+dive+eps; % Running sum of divergences
        end %jj
        diverge=log(diverge/countjj); % Average divergence
    end %end iff

    S_n=S_n+diverge; % Sum for computing second average
    S_mean(:,k)=diverge;
end %% end ii

S_mean=S_mean';
S_mean(S_mean(:,1)==0,:)=[];

S_n=S_n/count; % Final divergence average
S_n=S_n-S_n(1); % Divergence starts at 0, not needed
% but makes plot easier to read
%%
% Find the flat region and determine the slope
% Works by finding the region where the slope changes
% from its initial value

```

```

if fs <= 1                                % Length of testing region
    lima=2*fs;
else
    lima=fs;
end

y=S_n(1:lima);                            % Time vectors for fitting
x=time(1:lima)';
m1=polyfit(x,y,1);                        % Use 1st order polyfit for linear fit
ms1=m1(1);                                % Get slope
er=1;                                      % Initiate error at 0, to be used for finding
kk=lima;                                    % the saturation value

whilecount = 0;                            % Initiate counter
while er>.999                               % Loop iterates 'til r12 goes below 0.9
    x=time(1:kk)';                          % New vectors for fitting
    y=S_n(1:kk);
    [m2]=polyfit(x,y,1);
    f=polyval(m2,x);
    er=rsquared(y,f);                      % Percent change of slope
    kk=kk+1;                                % Iterate i by test length
    if er <.65 && whilecount == 0 % Break if no slope shift
        m2=m1;
        break
    end
    whilecount=whilecount+1;                  % Break if slope shift does not
    if kk>.2*length(S_n);                  % occur after a certain
        break;                                % number of points
    end
end
lambda=m2(1); % The Lyapunov exponent is the last fit.

% Plotting
subplot(2,1,1)
plot(x,y,x*x*m2(1)+m2(2))
text((max(x)-min(x))*.1+min(x),(max(y)-min(y))*.95+min(y),['Lyapunov
Exponent= ',num2str(m2(1))])

subplot(2,1,2)
plot((1:length(S_n))/fs,S_n)

```

Function make _ koch:

This function works by taking advantage of the fractal nature of the Koch curve. The curve for N iterations is made of scaled and angled versions of the first iteration of the curve. We can use some algebra and trigonometry to compute the correct placement, size, and angle of the first iteration shape to form higher-order iterations.

```

function make_koch(N)
% Function to draw Koch Curve for N iterates
% The method is to plot rescaled versions of the N=1 iteration
% at the correct angle. At every iteration, each straight line

```

Appendix B

```
% is replaced the N=1 iterate.  
%  
xs = [0,1]; % Initialize segment for N=1  
ys = [0,0];  
xk = [0, 1/3,.5,(2/3),1];  
yk = [0, 0,sin(pi/3)*(1/3),0,0]; % This follows from the geometry  
temp = [xs',ys']; % Create N=1 vector of (x,y)  
kochs = zeros(4\N+1,2); % Init. N length vector of (x,y)  
%  
for ii=1:N % For the number of iterations  
    numseg=4\l(ii-1); % Total number of segments in curve  
    length=(1/3)\l(ii-1); % Length of each segment  
    b=0; % Init. x value of segment start  
    for jj=1:numseg % Segments in current iteration  
        x=[temp(jj,1),temp(jj+1,1)]; % New straight segment  
        y=[temp(jj,2),temp(jj+1,2)];  
        t=atan2(y(2)-y(1),x(2)-x(1)); % Angle of segment  
        R=[cos(t),-sin(t);sin(t) cos(t)]; % Rotation of N =1 segment  
        for i=1:5  
            coord(i,:)=length *R*[xk(i);yk(i)]; % Scaling  
        end  
        coord(:,1)=coord(:,1)+x(1); % Put segment at correct spot  
        coord(:,2)=coord(:,2)+y(1);  
        r2=5;  
        if jj ~= numseg  
            coord(5,:)=[]; % If not the last segment  
            r2=4; % delete last point  
        end  
        a=b+1; % Update x values of the segment  
        b=a+r2 - 1; % Update x values  
        kochs(a:b,:)= [coord]; % Updates kochs sample value  
    end  
    temp(1:(4\l ii+1),:)=kochs(1:(4\l ii+1),:); % Update tem  
end  
plot(temp(:,1),temp(:,2)) % Plot the values after N iterations  
axis equal;
```

Bibliography

The following is a very selective list of books or articles that will be of value in providing greater depth and mathematical rigor to the material presented in this book. Comments regarding the particular strengths of the reference are included.

- Adomian, G. *Stochastic Systems: Mathematics in Science and Engineering*. Academic Press, New York, NY, 1983. The first chapter provides a very readable introduction to stochastic systems.
- Akansu, A.N. and Haddad, R.A. *Multiresolution Signal Decomposition: Transforms, Subbands, Wavelets*. Academic Press, San Diego, CA, 1992. A modern classic that presents, among other things, some of the underlying theoretical aspects of wavelet analysis.
- Aldroubi, A. and Unser, M. (eds) *Wavelets in Medicine and Biology*. CRC Press, Boca Raton, FL, 1996. Presents a variety of applications of wavelet analysis to biomedical engineering.
- Boashash, B. *Time-Frequency Signal Analysis*. Longman Cheshire Pty. Ltd., 1992. Early chapters provide a very useful introduction to time-frequency analysis followed by a number of medical applications.
- Boashash, B. and Black, P.J. An efficient real-time implementation of the Wigner–Ville distribution. *IEEE Trans. Acoust. Speech Sig. Proc. ASSP-35*: 1611–1618, 1987. Practical information on calculating the Wigner–Ville distribution.
- Boashash, B. and Reilly, A. Algorithms for time-frequency signal analysis. In: *Time-Frequency Signal Analysis*. Longman Cheshire, Pty. Ltd. Melbourne, Australia, 1992. Good coverage of analytic signal and code fragments for distribution kernels used in Chapter 6.
- Boudreault-Bartels, G.F. and Murry, R. Time-frequency signal representations for biomedical signals. In: *The Biomedical Engineering Handbook*. J. Bronzino (ed.) CRC Press, Boca Raton, Florida and IEEE Press, Piscataway, NJ. This article presents an exhaustive, or very nearly so, compilation of Cohen’s class of time-frequency distributions.
- Bruce, E.N. *Biomedical Signal Processing and Signal Modeling*, John Wiley and Sons, New York, NY, 2001. Rigorous treatment with more of an emphasis on linear systems than signal processing. Introduces nonlinear concepts such as chaos.
- Cichocki, A. and Amari, S. *Adaptive Blind Signal and Image Processing: Learning Algorithms and Applications*. John Wiley and Sons, Inc. New York, NY, 2002. Rigorous, somewhat dense, treatment of a wide range of principal component and independent component approaches. Includes disk.
- Cohen, L. Time-frequency distributions—A review. *Proc. IEEE* 77: 941–981, 1989. Classic review article on the various time-frequency methods in Cohen’s class of time-frequency distributions.
- Cohen, L. Introduction: A primer on time-frequency analysis. In: *Time-Frequency Signal Analysis*. Longman Cheshire, Pty. Ltd. Melbourne, Australia, 1992. Excellent introduction to time-frequency analysis.
- Costa, M., Goldberger, A.L. and Peng, C.K. Multiscale entropy analysis of complex physiologic time series. *Phys. Rev. Lett.* 89: 068102-1–4, 2002. The introductory paper for the multi-scale entropy analysis method. This paper has a more thorough explanation as well as examples on the application of the method.
- Ferrara, E. and Widrow, B. Fetal electrocardiogram enhancement by time-sequenced adaptive filtering. *IEEE Trans. Biomed. Engr. BME-29*: 458–459, 1982. Early application of adaptive noise cancellation to a biomedical engineering problem by one of the founders of the field. See also Widrow below.
- Friston, K. Statistical Parametric Mapping Online at: <http://www.fil.ion.ucl.ac.uk/spm/> Through discussion of practical aspects of fMRI analysis including pre-processing, statistical methods, and experimental design. Based around SPM analysis software capabilities.
- Grassberger, P. and Procaccia, I. Characterization of strange attractors. *Phys. Rev. Lett.* 50(5): 346–349, 1983. The introductory paper that describes the correlation sum and Grassberger–Procaccia method. The GPA is applied to several common nonlinear systems.
- Grassberger, P. and Procaccia, I. Estimation of the Kolmogorov entropy from a chaotic signal. *Phy. Rev. A Am. Phys. Soc.* 28: 2591–2593, 1983. A thorough explanation of how to use the Grassberger–Procaccia algorithm to estimate the Sinai–Kolmogorov entropy.
- Hasti, T., Tibshirani, R., and Friedman, J. *The Elements of Statistical Learning*. Springer-Verlag, New York, NY, 2001. Comprehensive treatment of classification and regression. Good introductory material. Includes topics on cluster analysis, k -nearest neighbor, and k -means approaches.

Bibliography

- Haykin, S. *Adaptive Filter Theory* (2nd ed.). Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991. The definitive text on adaptive filters including Weiner filters and gradient-based algorithms.
- Hénon, M. A two-dimensional mapping with a strange attractor. *Communications in Mathematical Physics* 50: 69–77, 1976. The introductory paper for the Hénon map. Provides a thorough description of how the Lorenz equations are related to the Hénon map than what could be covered in Chapter 10.
- Hilborn, R.C. *Chaos and Nonlinear Dynamics: An Introduction for Scientists and Engineers*. Oxford University Press, New York, NY, 2000. An excellent resource for a more theoretical background of the concepts of nonlinear dynamics, including nonlinear attractor, the Lyapunov exponent, and the correlation dimension.
- Hoyer, D. Mutual information and phase dependencies: Measures of reduced nonlinear cardiorespiratory interactions after myocardial infarction. *Med. Eng. Phys.* 24: 33–43, 2002. A discussion of the mutual information and automutual information and its application to the nonlinear heart-rate variability analysis.
- Hubbard, B.B. *The World According to Wavelets* (2nd ed.). A.K. Peters, Ltd. Natick, MA, 1998. Very readable introductory book on waveforms including an excellent section on the Fourier transformed. Can be read by a non-signal processing friend.
- Hyvärinen, A., Karhunen, J., and Oja, E. *Independent Component Analysis*. John Wiley and Sons, Inc., New York, NY, 2001. Fundamental, comprehensive, yet readable book on independent component analysis. Also provides a good review of principal component analysis.
- Ingle, V.K. and Proakis, J.G. *Digital Signal Processing with MATLAB*. Brooks/Cole, Inc., Pacific Grove, CA, 2000. Excellent treatment of classical signal processing methods including the Fourier transform and both FIR and IIR digital filters. Brief, but informative section on adaptive filtering.
- Jackson, J.E. *A User's Guide to Principal Components*. John Wiley and Sons, New York, NY, 1991. Classic book providing everything you ever want to know about principal component analysis. Also covers linear modeling and introduces factor analysis.
- Johnson, D.D. *Applied Multivariate Methods for Data Analysis*. Brooks/Cole, Pacific Grove, CA, 1998. Careful, detailed coverage of multivariate methods including principal components analysis. Good coverage of discriminant analysis techniques.
- Kak, A.C. and Slaney, M. *Principles of Computerized Tomographic Imaging*. IEEE Press, New York, NY, 1988. Thorough, understandable treatment of algorithms for reconstruction of tomographic images including both parallel and fan-beam geometry. Also includes techniques used in reflection tomography as occurs in ultrasound imaging.
- Kantz, H. and Schreiber, T. *Nonlinear Time Series Analysis*. Cambridge University Press, Cambridge, UK, 2004. A practical look at nonlinear dynamic systems analysis. This is an excellent text that covers several of the nonlinear analysis methods described in Chapters 10 and 11 as well as more advanced methods. This book suitable for the signal processor who is interested in a deeper evaluation of nonlinear systems and signals.
- Kaplan, J.L. and Yorke J.A. Chaotic behavior of multidimensional difference equations. In: H.-O. Peitgen, and H.-O. Walther (eds.), *Functional Differential Equations and Approximation of Fixed Points*, Vol. 730 of Lecture Notes in Mathematics, pp. 204–227. Springer, Berlin Heidelberg, 1979. This source provides a thorough demonstration of the chaotic behavior of the logistic map and an analysis of the logistic map system, as well as other iterated maps.
- Lorenz, E.N. *The Essence of Chaos*. Jessie and John Danz Lectures. University of Washington Press, Seattle, WA, 1995. This book is a collection of essays describing how the chaotic Lorenz system was discovered and how the discovery of chaos changed the world of mathematics and the natural sciences.
- Marple, S.L. *Digital Spectral Analysis with Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1987. Classic text on modern spectral analysis methods. In-depth, rigorous treatment of Fourier transform, parametric modeling methods (including AR and ARMA), and eigenanalysis-based techniques.
- Najarian, K. and Splinter, R. *Biomedical Signal and Image Processing*, CRC Press, Taylor and Francis, Boca Raton, FL. 2006. Many of the same topics covered here including the Fourier transform, wavelet analysis, digital filtering, and classification and clustering. This textbook also includes topics on the physiological origin of signals and image instrumentation. Some MATLAB problems.
- Osborne, A. and Provenzale, P. Finite correlation dimension for stochastic systems with power-law spectra. *Phys. D* 35(3): 357–381, 1989. The first of a pair of papers showing that a fractal correlation dimension is not specific for chaotic behavior.

Bibliography

- Peng, C.K. et al. Mosaic organization of DNA nucleotides. *Phys. Rev. E* 49: 1685–1689, 1994. The introductory to the detrended fluctuation analysis method. Here DFA is used to analyze DNA nucleotide patterns.
- Pincus, S., Gladstone, I., and Ehrenkranz, R. A regularity statistic for medical data analysis. *J. Clin. Mon. Compu.* Springer Netherlands, 7: 335–345, 1991. An introductory paper for approximate entropy.
- Poulikas, A.D. and Ramadn, Z. M. *Adaptive Filtering Primer with MATLAB*. CRC Press, Taylor and Francis, Boca Raton, FL. 2006. A very detailed description of Wiener filters and the LMS algorithm with a few MATLAB examples.
- Principe, J., Euliano, M.R., and Lefebvre, W.C. *Neural and Adaptive Systems*. John Wiley and Sons, Inc., New York, NY, 1999. A textbook with detailed coverage of selected classifiers with emphasis on adaptive neural nets. Also covers general signal processing concepts. Comes with a disk that has a complete software system to implement adaptive neural nets.
- Provenzale, A. et al. Distinguishing between low-dimensional dynamics and randomness in measured time series. *Physica D: Nonlinear Phenomena* 58(1–4): 31–49, 1992. The second of a pair of papers demonstrating that the GPA may not be reliable for distinguishing nonlinear chaos if the signal of interest contains correlated noise.
- Rao, R.M. and Bopardikar, A.S. *Wavelet Transforms: Introduction to Theory and Applications*. Addison-Wesley, Inc., Reading, MA, 1998. Good development of wavelet analysis including both the continuous and discreet wavelet transforms.
- Richman, J.S. and Moorman, J.R. Physiological time-series analysis using approximate entropy and sample entropy. *Am. J. Physiol. Heart Circulat. Physiol.* 278: H2039–H2049, 2000. An introductory paper for sample entropy that does a thorough comparison between approximate entropy and sample entropy. The methods are compared against artificial and cardiovascular data.
- Ruelle, D. and Takens, F. On the nature of turbulence. *Commun. Math. Phys.* 23: 343–344, 1971. The seminal paper that introduced the term strange attractor to the world.
- Shannon, C.E. A mathematical theory of communication. *Bell Syst. Tech. J.* 27: 379–423, 1948. The seminal paper that introduces the concepts of information and entropy for a signal. This paper is thorough but also well written and easy to understand.
- Shiavi, R. *Introduction to Applied Statistical Signal Analysis* (2nd ed.). Academic Press, San Diego, CA, 1999. Emphasizes spectral analysis of signals buried in noise. Excellent coverage of Fourier analysis, and autoregressive methods. Good introduction to statistical signal processing concepts.
- Smith, S.W. *The Scientists and Engineer's Guide to Digital Signal Processing*. California Technical Publishing. 1997. A very clear and detailed description of basic digital signal processing concepts using a minimum of mathematics. Excellent coverage of the basics: Fourier transform, convolution basic filters. Available through the website: DSPGuide.com
- Sonka, M., Hlavac, V., and Boyle, R. *Image Processing, Analysis, and Machine Vision*. Chapman & Hall Computing, London, 1993. A good description of edge-based and other segmentation methods.
- Stam, C.J. Nonlinear dynamical analysis of EEG and MEG: Review of an emerging field. Clinical neurophysiology: Official. *J. Int. Fed. of Clin. Neurophysiol.* 116(10): 2266–2301, 2005. A thorough history of the use of nonlinear analysis for EEG signals. It covers the major rise and fall of the GPA as well as other nonlinear methods.
- Stearns, S.D. and David, R.A. *Signal Processing Algorithms in MATLAB*. Prentice-Hall, Upper Saddle River, NJ, 1996. Good treatment of the classical Fourier transform and digital filters. Also covers the LMS adaptive filter algorithm. Disk enclosed.
- Strang, G. and Nguyen, T. *Wavelets and Filter Banks*. Wellesley-Cambridge Press, Wellesley, MA, 1997. Thorough coverage of Wavelet Filter Banks including extensive mathematical background.
- Strogatz, S. *Nonlinear Dynamics and Chaos*. Perseus Books, Reading, MA. 1994. An excellent and highly readable text discussing nonlinear systems and analysis. A good resource for those not as mathematically inclined.
- Takens, F. *Dynamical Systems and Turbulence. Lecture Notes in Mathematics*, Vol. 898, pp. 366–381. Springer-Verlag, Berlin Heidelberg, Germany, 1981. While we have asked you to take on faith that the delay embedding procedure is mathematically valid, this paper by Takens contains the mathematical proof.
- Tan, L. *Digital Signal Processing*. Elsevier Press, Burlington, MA, 2008. Detailed coverage of many of the topics in this book. Some MATLAB problems and examples.
- Viertio-Oja, H., Maja, V., Sarkela, M., Talja, P., Tenkanen, N., Tolvanen-Laakso, H., Paloheimo, M., Vakkuri, A., Yli-Hankala, A., and Merilainen, P. Description of the entropy algorithm as applied in the datex-ohmeda s/5 entropy module. *Acta Anaesthesiologica Scandinavica*, 48(2): 154–161, 2004. A detailed

Bibliography

- description and analysis of the spectral entropy method discussed in Chapter 11 in the context of a system for measuring the depth of anesthesia during surgery.
- West, B.J., Chen, D., and Mackey, H.J. Methods for distinguishing chaos from colored noise. In: *Patterns, Information and Chaos in Neuronal Systems*. B.J. West (ed.) Chapter 1, pp. 1–41, 1993. World Scientific. Pub. Ltd. Singapore. A book chapter that discusses the problem of distinguishing noise from chaos that covers more advanced materials than we have in this text.
- Wickerhauser, M.V. *Adapted Wavelet Analysis from Theory to Software*. A.K. Peters, Ltd and IEEE Press. Wellesley, MA, 1994. Rigorous, extensive treatment of wavelet analysis.
- Widrow, B., Glover, J., McCool, J. et al. Adaptive noise cancelling: Principles and applications. *Proc. IEEE* 63, 1692–1716, 1975. Classic original article on adaptive noise cancellation.
- Williams, S. and Jeong, J. Reduced interference time-frequency distributions. In: *Time-Frequency Signal Analysis*. Longman Cheshire, Pty. Ltd. Melbourne, Australia, 1992. Detailed description of time-frequency kernels that reduce cross-products.
- Wright, S. Nuclear magnetic resonance and magnetic resonance imaging. In: *Introduction to Biomedical Engineering*. Enderle, J., Blanchard, S.M., and Bronzino, J. (eds). Academic Press, San Diego, CA, 2000. Good mathematical development of the physics of MRI using classical concepts.

BIOSIGNAL and MEDICAL IMAGE PROCESSING

Third Edition

JOHN L. SEMMLOW
BENJAMIN GRIFFEL

See What's New in the Third Edition:

- Two new chapters on nonlinear methods for describing and classifying signals.
- Additional examples with biological data such as EEG, ECG, respiration and heart rate variability
- Nearly double the number of end-of-chapter problems
- MATLAB® incorporated throughout the text
- Data “cleaning” methods commonly used in such areas as heart rate variability studies

Written specifically for biomedical engineers, **Biosignal and Medical Image Processing, Third Edition** provides a complete set of signal and image processing tools, including diagnostic decision-making tools, and classification methods. Thoroughly revised and updated, it supplies important new material on nonlinear methods for describing and classifying signals, including entropy-based methods and scaling methods. A full set of PowerPoint slides covering the material in each chapter and problem solutions is available to instructors for download.



CRC Press
Taylor & Francis Group
an informa business

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
711 Third Avenue
New York, NY 10017
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

“...An excellent review of the actual trendiest techniques in signal processing with a very clear (and simplified) description of their capabilities in signal and image analysis. MATLAB examples are an excellent addition to provide students with capabilities to understand better how the techniques work...”

—**Enrique Nava Baro, PhD,**

University of Málaga, Spain

“The book is a welcome addition to the teaching literature for biomedical engineering, building on the previous edition’s friendly approach to introducing the material. This makes it particularly suitable for biomedical engineering, a field in which students come from a variety of backgrounds, and where familiarity of the fundamentals of electrical engineering cannot be assumed.”

—**David A. Clifton,**

University of Oxford, UK

K16329

ISBN: 978-1-4665-6736-8
9 0000



9 781466 567368
www.crcpress.com