Real-Time Digital Signal Processing

Fundamentals, Implementations and Applications

THIRD EDITION

SEN M. KUO | BOB H. LEE | WENSHUN TIAN

WILEY

REAL-TIME DIGITAL SIGNAL PROCESSING

REAL-TIME DIGITAL SIGNAL PROCESSING

FUNDAMENTALS, IMPLEMENTATIONS AND APPLICATIONS

Third Edition

Sen M. Kuo
Northern Illinois University, USA

Bob H. Lee
Ittiam Systems, Inc., USA

Wenshun Tian
Sonus Networks, Inc., USA

WILEY

This edition first published in 2013
© 2013 John Wiley & Sons, Ltd

Registered office
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.

The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The publisher is not associated with any product or vendor mentioned in this book

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. It is sold on the understanding that the publisher is not engaged in rendering professional services and neither the publisher nor the author shall be liable for damages arising herefrom. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

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.

Library of Congress Cataloging-in-Publication

Kuo, Sen M. (Sen-Maw)
Real-time digital signal processing : implementations and applications / Sen M. Kuo, Bob H. Lee, Wenshun Tian. – Third edition.
pages cm
Includes bibliographical references and index.
ISBN 978-1-118-41432-3 (hardback)
1. Signal processing–Digital techniques. 2. Texas Instruments TMS320 series microprocessors. I. Lee, Bob H. II. Tian, Wenshun. III. Title.
TK5102.9.K86 2013
621.382′2–dc23

2013018929

A catalogue record for this book is available from the British Library.

ISBN: 978-1-118-41432-3

Set in 10/12 pt, Times Roman by Thomson Digital, Noida, India

1: 2013

Contents

Preface **xv**

Acknowledgments **xix**

1 Introduction to Real-Time Digital Signal Processing **1**
1.1 Basic Elements of Real-Time DSP Systems 2
1.2 Analog Interface 3
1.2.1 Sampling 3
1.2.2 Quantization and Encoding 7
1.2.3 Smoothing Filters 8
1.2.4 Data Converters 9
1.3 DSP Hardware 10
1.3.1 DSP Hardware Options 11
1.3.2 Digital Signal Processors 13
1.3.3 Fixed- and Floating-Point Processors 14
1.3.4 Real-Time Constraints 15
1.4 DSP System Design 16
1.4.1 Algorithm Development 16
1.4.2 Selection of DSP Hardware 18
1.4.3 Software Development 19
1.4.4 Software Development Tools 20
1.5 Experiments and Program Examples 21
1.5.1 Get Started with CCS and eZdsp 22
1.5.2 C File I/O Functions 26
1.5.3 User Interface for eZdsp 30
1.5.4 Audio Playback Using eZdsp 35
1.5.5 Audio Loopback Using eZdsp 38
Exercises 42
References 43

2 DSP Fundamentals and Implementation Considerations **44**
2.1 Digital Signals and Systems 44
2.1.1 Elementary Digital Signals 44
2.1.2 Block Diagram Representation of Digital Systems 47

2.2 System Concepts 48
2.2.1 LTI Systems 48
2.2.2 The z-transform 52
2.2.3 Transfer Functions 54
2.2.4 Poles and Zeros 58
2.2.5 Frequency Responses 61
2.2.6 Discrete Fourier Transform 65
2.3 Introduction to Random Variables 66
2.3.1 Review of Random Variables 67
2.3.2 Operations of Random Variables 68
2.4 Fixed-Point Representations and Quantization Effects 72
2.4.1 Fixed-Point Formats 72
2.4.2 Quantization Errors 75
2.4.3 Signal Quantization 75
2.4.4 Coefficient Quantization 78
2.4.5 Roundoff Noise 78
2.4.6 Fixed-Point Toolbox 79
2.5 Overflow and Solutions 81
2.5.1 Saturation Arithmetic 81
2.5.2 Overflow Handling 82
2.5.3 Scaling of Signals 82
2.5.4 Guard Bits 83
2.6 Experiments and Program Examples 83
2.6.1 Overflow and Saturation Arithmetic 83
2.6.2 Function Approximations 86
2.6.3 Real-Time Signal Generation Using eZdsp 94
Exercises 99
References 101

3 Design and Implementation of FIR Filters 102
3.1 Introduction to FIR Filters 102
3.1.1 Filter Characteristics 102
3.1.2 Filter Types 104
3.1.3 Filter Specifications 106
3.1.4 Linear Phase FIR Filters 108
3.1.5 Realization of FIR Filters 110
3.2 Design of FIR Filters 114
3.2.1 Fourier Series Method 114
3.2.2 Gibbs Phenomenon 116
3.2.3 Window Functions 118
3.2.4 Design of FIR Filters Using MATLAB® 120
3.2.5 Design of FIR Filters Using the FDATool 122
3.3 Implementation Considerations 125
3.3.1 Quantization Effects in FIR Filters 125
3.3.2 MATLAB® Implementations 127
3.3.3 Floating-Point C Implementations 128
3.3.4 Fixed-Point C Implementations 129

3.4 Applications: Interpolation and Decimation Filters 130
3.4.1 Interpolation 130
3.4.2 Decimation 131
3.4.3 Sampling Rate Conversion 133
3.4.4 MATLAB® Implementations 134
3.5 Experiments and Program Examples 135
3.5.1 FIR Filtering Using Fixed-Point C 135
3.5.2 FIR Filtering Using C55xx Assembly Program 136
3.5.3 Symmetric FIR Filtering Using C55xx Assembly Program 137
3.5.4 Optimization Using Dual-MAC Architecture 138
3.5.5 Real-Time FIR Filtering 140
3.5.6 Decimation Using C and Assembly Programs 141
3.5.7 Interpolation Using Fixed-Point C 142
3.5.8 Sampling Rate Conversion 142
3.5.9 Real-Time Sampling Rate Conversion 143
Exercises 144
References 147

4 Design and Implementation of IIR Filters 148
4.1 Introduction 148
4.1.1 Analog Systems 148
4.1.2 Mapping Properties 150
4.1.3 Characteristics of Analog Filters 151
4.1.4 Frequency Transforms 153
4.2 Design of IIR Filters 154
4.2.1 Bilinear Transform 155
4.2.2 Filter Design Using the Bilinear Transform 156
4.3 Realization of IIR Filters 158
4.3.1 Direct Forms 158
4.3.2 Cascade Realizations 160
4.3.3 Parallel Realizations 161
4.3.4 Realization of IIR Filters Using MATLAB® 162
4.4 Design of IIR Filters Using MATLAB® 164
4.4.1 Filter Design Using MATLAB® 164
4.4.2 Frequency Transforms Using MATLAB® 166
4.4.3 Filter Design and Realization Using the FDATool 166
4.5 Implementation Considerations 168
4.5.1 Stability 168
4.5.2 Finite-Precision Effects and Solutions 170
4.5.3 MATLAB® Implementations of IIR Filters 172
4.6 Practical Applications 174
4.6.1 Recursive Resonators 174
4.6.2 Recursive Quadrature Oscillators 177
4.6.3 Parametric Equalizers 179
4.7 Experiments and Program Examples 179
4.7.1 Direct-Form I IIR Filter Using Floating-Point C 179

4.7.2	*Direct-Form I IIR Filter Using Fixed-Point C*	181
4.7.3	*Cascade IIR Filter Using Fixed-Point C*	182
4.7.4	*Cascade IIR Filter Using Intrinsics*	185
4.7.5	*Cascade IIR Filter Using Assembly Program*	188
4.7.6	*Real-Time IIR Filtering*	189
4.7.7	*Parametric Equalizer Using Fixed-Point C*	190
4.7.8	*Real-Time Parametric Equalizer*	190
Exercises		191
References		194
5	**Frequency Analysis and the Discrete Fourier Transform**	**195**
5.1	Fourier Series and Fourier Transform	195
5.1.1	*Fourier Series*	195
5.1.2	*Fourier Transform*	197
5.2	Discrete Fourier Transform	198
5.2.1	*Discrete-Time Fourier Transform*	198
5.2.2	*Discrete Fourier Transform*	200
5.2.3	*Important Properties*	202
5.3	Fast Fourier Transforms	205
5.3.1	*Decimation-in-Time*	206
5.3.2	*Decimation-in-Frequency*	208
5.3.3	*Inverse Fast Fourier Transform*	209
5.4	Implementation Considerations	210
5.4.1	*Computational Issues*	210
5.4.2	*Finite-Precision Effects*	210
5.4.3	*MATLAB® Implementations*	211
5.4.4	*Fixed-Point Implementation Using MATLAB®*	212
5.5	Practical Applications	214
5.5.1	*Spectral Analysis*	214
5.5.2	*Spectral Leakage and Resolution*	215
5.5.3	*Power Spectral Density*	219
5.5.4	*Convolution*	222
5.6	Experiments and Program Examples	224
5.6.1	*DFT Using Floating-Point C*	224
5.6.2	*DFT Using the C55xx Assembly Program*	226
5.6.3	*FFT Using Floating-Point C*	227
5.6.4	*FFT Using Fixed-Point C with Intrinsics*	227
5.6.5	*Experiment with the FFT and IFFT*	231
5.6.6	*FFT Using the C55xx Hardware Accelerator*	231
5.6.7	*Real-Time FFT Using the C55xx Hardware Accelerator*	233
5.6.8	*Fast Convolution Using the Overlap–Add Technique*	234
5.6.9	*Real-Time Fast Convolution*	235
Exercises		236
References		238

6 Adaptive Filtering **239**
6.1 Introduction to Random Processes 239
6.2 Adaptive Filters 243
6.2.1 Introduction to Adaptive Filtering 243
6.2.2 Performance Function 244
6.2.3 Method of Steepest Descent 248
6.2.4 LMS Algorithm 249
6.2.5 Modified LMS Algorithms 251
6.3 Performance Analysis 252
6.3.1 Stability Constraint 252
6.3.2 Convergence Speed 253
6.3.3 Excess Mean-Square Error 254
6.3.4 Normalized LMS Algorithm 254
6.4 Implementation Considerations 255
6.4.1 Computational Issues 255
6.4.2 Finite-Precision Effects 256
6.4.3 MATLAB® Implementations 257
6.5 Practical Applications 259
6.5.1 Adaptive System Identification 259
6.5.2 Adaptive Prediction 262
6.5.3 Adaptive Noise Cancellation 264
6.5.4 Adaptive Inverse Modeling 267
6.6 Experiments and Program Examples 268
6.6.1 LMS Algorithm Using Floating-Point C 268
6.6.2 Leaky LMS Algorithm Using Fixed-Point C 270
6.6.3 Normalized LMS Algorithm Using Fixed-Point C and Intrinsics 270
6.6.4 Delayed LMS Algorithm Using Assembly Program 274
6.6.5 Experiment of Adaptive System Identification 275
6.6.6 Experiment of Adaptive Predictor 276
6.6.7 Experiment of Adaptive Channel Equalizer 277
6.6.8 Real-Time Adaptive Prediction Using eZdsp 279
Exercises 280
References 282

7 Digital Signal Generation and Detection **283**
7.1 Sine Wave Generators 283
7.1.1 Lookup Table Method 283
7.1.2 Linear Chirp Signal 286
7.2 Noise Generators 288
7.2.1 Linear Congruential Sequence Generator 288
7.2.2 Pseudo-random Binary Sequence Generator 289
7.2.3 White, Color, and Gaussian Noise 290
7.3 DTMF Generation and Detection 291
7.3.1 DTMF Generator 291
7.3.2 DTMF Detection 292

7.4 Experiments and Program Examples 298
7.4.1 Sine Wave Generator Using Table Lookup 298
7.4.2 Siren Generator Using Table Lookup 299
7.4.3 DTMF Generator 299
7.4.4 DTMF Detection Using Fixed-Point C 300
7.4.5 DTMF Detection Using Assembly Program 301
Exercises 302
References 302

8 Adaptive Echo Cancellation 304
8.1 Introduction to Line Echoes 304
8.2 Adaptive Line Echo Canceler 306
8.2.1 Principles of Adaptive Echo Cancellation 306
8.2.2 Performance Evaluation 308
8.3 Practical Considerations 309
8.3.1 Pre-whitening of Signals 309
8.3.2 Delay Estimation 309
8.4 Double-Talk Effects and Solutions 312
8.5 Nonlinear Processor 314
8.5.1 Center Clipper 314
8.5.2 Comfort Noise 315
8.6 Adaptive Acoustic Echo Cancellation 315
8.6.1 Acoustic Echoes 316
8.6.2 Acoustic Echo Canceler 317
8.6.3 Subband Implementations 318
8.6.4 Delay-Free Structures 321
8.6.5 Integration of Acoustic Echo Cancellation with Noise Reduction 321
8.6.6 Implementation Considerations 322
8.7 Experiments and Program Examples 323
8.7.1 Acoustic Echo Canceler Using Floating-Point C 323
8.7.2 Acoustic Echo Canceler Using Fixed-Point C with Intrinsics 325
8.7.3 Integration of AEC and Noise Reduction 326
Exercises 328
References 329

9 Speech Signal Processing 330
9.1 Speech Coding Techniques 330
9.1.1 Speech Production Model Using LPC 331
9.1.2 CELP Coding 332
9.1.3 Synthesis Filter 334
9.1.4 Excitation Signals 337
9.1.5 Perceptual Based Minimization Procedure 340
9.1.6 Voice Activity Detection 341
9.1.7 ACELP Codecs 343
9.2 Speech Enhancement 350
9.2.1 Noise Reduction Techniques 350

9.2.2 Short-Time Spectrum Estimation 351
9.2.3 Magnitude Spectrum Subtraction 353
9.3 VoIP Applications 355
9.3.1 Overview of VoIP 355
9.3.2 Discontinuous Transmission 357
9.3.3 Packet Loss Concealment 358
9.3.4 Quality Factors of Media Stream 359
9.4 Experiments and Program Examples 360
9.4.1 LPC Filter Using Fixed-Point C with Intrinsics 360
9.4.2 Perceptual Weighting Filter Using Fixed-Point C with Intrinsics 364
9.4.3 VAD Using Floating-Point C 365
9.4.4 VAD Using Fixed-Point C 367
9.4.5 Speech Encoder with Discontinuous Transmission 368
9.4.6 Speech Decoder with CNG 369
9.4.7 Spectral Subtraction Using Floating-Point C 370
9.4.8 G.722.2 Using Fixed-Point C 372
9.4.9 G.711 Companding Using Fixed-Point C 373
9.4.10 Real-Time G.711 Audio Loopback 373
Exercises 374
References 375

10 Audio Signal Processing 377
10.1 Introduction 377
10.2 Audio Coding 378
10.2.1 Basic Principles 378
10.2.2 Frequency-Domain Coding 383
10.2.3 Lossless Audio Coding 386
10.2.4 Overview of MP3 387
10.3 Audio Equalizers 389
10.3.1 Graphic Equalizers 389
10.3.2 Parametric Equalizers 391
10.4 Audio Effects 397
10.4.1 Sound Reverberation 398
10.4.2 Time Stretch and Pitch Shift 399
10.4.3 Modulated and Mixed Sounds 401
10.4.4 Spatial Sounds 409
10.5 Experiments and Program Examples 411
10.5.1 MDCT Using Floating-Point C 411
10.5.2 MDCT Using Fixed-Point C and Intrinsics 415
10.5.3 Pre-echo Effects 416
10.5.4 MP3 Decoding Using Floating-Point C 419
10.5.5 Real-Time Parametric Equalizer Using eZdsp 421
10.5.6 Flanger Effects 422
10.5.7 Real-Time Flanger Effects Using eZdsp 423
10.5.8 Tremolo Effects 424
10.5.9 Real-Time Tremolo Effects Using eZdsp 425

10.5.10 Spatial Sound Effects 425
10.5.11 Real-Time Spatial Effects Using eZdsp 426
Exercises 427
References 428

11 Introduction to Digital Image Processing **430**
11.1 Digital Images and Systems 430
11.1.1 Digital Images 430
11.1.2 Digital Image Systems 431
11.2 Color Spaces 432
11.3 YC_bC_r Sub-sampled Color Space 433
11.4 Color Balance and Correction 434
11.4.1 Color Balance 434
11.4.2 Color Correction 435
11.4.3 Gamma Correction 436
11.5 Histogram Equalization 437
11.6 Image Filtering 440
11.7 Fast Convolution 448
11.8 Practical Applications 452
11.8.1 DCT and JPEG 452
11.8.2 Two-Dimensional DCT 452
11.8.3 Fingerprint 455
11.8.4 Discrete Wavelet Transform 456
11.9 Experiments and Program Examples 461
11.9.1 YC_bC_r to RGB Conversion 462
11.9.2 White Balance 464
11.9.3 Gamma Correction and Contrast Adjustment 465
11.9.4 Image Filtering 467
11.9.5 DCT and IDCT 468
11.9.6 Image Processing for Fingerprints 469
11.9.7 The 2-D Wavelet Transform 470
Exercises 474
References 475

Appendix A Some Useful Formulas and Definitions **477**
A.1 Trigonometric Identities 477
A.2 Geometric Series 478
A.3 Complex Variables 479
A.4 Units of Power 480
References 483

Appendix B Software Organization and List of Experiments **484**

Appendix C Introduction to the TMS320C55xx Digital Signal Processor **490**
C.1 Introduction 490
C.2 TMS320C55xx Architecture 490

C.2.1 Architecture Overview 490
C.2.2 On-Chip Memories 494
C.2.3 Memory-Mapped Registers 495
C.2.4 Interrupts and Interrupt Vector 498
C.3 TMS320C55xx Addressing Modes 498
C.3.1 Direct Addressing Modes 501
C.3.2 Indirect Addressing Modes 502
C.3.3 Absolute Addressing Modes 505
C.3.4 MMR Addressing Mode 505
C.3.5 Register Bits Addressing Mode 506
C.3.6 Circular Addressing Mode 507
C.4 TMS320C55xx Assembly Language Programming 508
C.4.1 Arithmetic Instructions 508
C.4.2 Logic and Bit Manipulation Instructions 509
C.4.3 Move Instruction 509
C.4.4 Program Flow Control Instructions 510
C.4.5 Parallel Execution 514
C.4.6 Assembly Directives 516
C.4.7 Assembly Statement Syntax 518
C.5 C Programming for TMS320C55xx 520
C.5.1 Data Types 520
C.5.2 Assembly Code Generation by C Compiler 520
C.5.3 Compiler Keywords and Pragma Directives 522
C.6 Mixed C and Assembly Programming 525
C.7 Experiments and Program Examples 529
C.7.1 Examples 529
C.7.2 Assembly Program 530
C.7.3 Multiplication 530
C.7.4 Loops 531
C.7.5 Modulo Operator 532
C.7.6 Use Mixed C and Assembly Programs 533
C.7.7 Working with AIC3204 533
C.7.8 Analog Input and Output 534
References 535

Index **537**

Preface

In recent years, real-time digital signal processing (DSP) using general-purpose DSP processors has provided an effective way to design and implement DSP systems for practical applications. Many companies are actively engaged in real-time DSP research for developing new applications. The study of real-time DSP applications has been and will continue to be a challenging field for students, engineers, and researchers. It is important to master not only the theory, but also the skill of system design and implementation techniques.

Since the publication of the first edition of the book entitled *Real-Time Digital Signal Processing* in 2001 and the second edition in 2006, the use of digital signal processors has penetrated into a much wider range of applications. This has led to curriculum changes in many universities to offer new real-time DSP courses that focus on implementations and applications, as well as enhancing the traditional theoretical lectures with hands-on real-time experiments. In the meantime, advances in new processors and development tools constantly demand up-to-date books in order to keep up with the rapid evolution of DSP developments, applications, and software updates. We intend with the third edition of this book to integrate the theory, design, applications, and implementations using hands-on experiments for the effective learning of real-time DSP technologies.

This book presents fundamental DSP principles along with many MATLAB® examples and emphasizes real-time applications using hands-on experiments. The book is designed to be used as a textbook for senior undergraduate/graduate students. The prerequisites for this book are concepts of signals and systems, basic microprocessor architecture, and MATLAB® and C programming. These topics are usually covered at the sophomore and junior levels in electrical and computer engineering, computer science, and other related science and engineering fields. This book can also serve as a reference for engineers, algorithm developers, and embedded system designers and programmers to learn DSP principles and implementation techniques for developing practical applications. We use a hands-on approach by conducting experiments and evaluating the results in order to help readers to understand the principles behind complicated theories. A list of textbooks and technical papers with mathematical proofs are provided as references at the end of each chapter for those who are interested in going beyond the coverage of this book.

The major aims and changes for this third edition are summarized as follows:

1. Focus on practical applications and provide step-by-step hands-on experiments for the complete design cycle starting from the evaluation of algorithms using MATLAB® to the implementation using floating-point C programming, and updated to fixed-point C

programming, and software optimization using mixed C and assembly programming with C intrinsics and assembly routines for fixed-point digital signal processors. This methodology enables readers to concentrate on learning DSP fundamentals and innovative applications by relaxing the intensive programming efforts, especially the time-consuming assembly programming.

2. Enhance many examples and hands-on experiments to make the DSP principles more interesting and interactive with real-world applications. All the C and assembly programs are carefully updated using the most recent versions of development tools, the Code Composer Studio and the low-cost TMS320C5505 (a member of C55xx family) eZdsp USB stick, for real-time experiments. Due to its affordable cost and portability, the eZdsp enables students, engineers, professionals, and hobbyists to conduct DSP experiments and projects at more convenient locations instead of in traditional laboratories. This new hardware tool is widely used by universities and industrial organizations to replace the older and more expensive development tools.
3. Add attractive and challenging DSP applications such as speech coding techniques for next generation networks and cell (mobile) phones; audio coding for portable players; several audio effects including spatial sounds, graphic and parametric audio equalizers for music, and audio post-recording effects; two-dimensional discrete wavelet transform for JPEG2000; image filtering for special effects; and fingerprint image processing. Also develop real-time experiments with modular designs and flexible interfaces such that the software may serve as prototyping programs to create other related applications.
4. Organize chapters in a more flexible and logical manner. Some related applications are grouped together. We also removed some topics, such as channel coding techniques, that may not be suitable for a semester-long course. The hardware-dependent topics in the second edition have been greatly simplified and presented here as an appendix for readers who are required or interested to learn about TMS320C55xx architecture and assembly programming. All of these changes are made intentionally for the purpose of focusing on the fundamental DSP principles with enhanced hands-on experiments for practical applications.

Many DSP algorithms and applications are available in MATLAB® and floating-point C programs. This book provides a systematic software development process for converting these programs to fixed-point C and optimizing them for implementation on fixed-point processors. To effectively illustrate DSP principles and applications, MATLAB® is used for the demonstration, design, and analysis of algorithms. This development stage is followed by floating-point and fixed-point C programming for implementing DSP algorithms. Finally, we integrate the CCS with the C5505 eZdsp for hands-on experiments. To utilize the advanced architecture and instruction set for efficient software development and maintenance, we emphasize using mixed C and assembly programs for real-time applications.

This book is organized into two parts: principles and applications. The first part (Chapters 1–6) introduces DSP principles, algorithms, analysis methods, and implementation considerations. Chapter 1 reviews the fundamentals of real-time DSP functional blocks, DSP hardware options, fixed- and floating-point DSP devices, real-time constraints, and algorithm and software development processes. Chapter 2 presents fundamental DSP concepts and practical considerations for the implementation of DSP algorithms. The theory, design, analysis, implementation, and application of finite impulse response and infinite impulse response filters are presented in Chapters 3 and 4, respectively.

The concepts and considerations of using the discrete Fourier transform for frequency analysis, and the implementation and application of the fast Fourier transform, are introduced in Chapter 5. Basic principles of adaptive signal processing with many practical considerations for applications are presented in Chapter 6.

The second part (Chapters 7–11) introduces several important DSP applications that have played important roles in the realization of modern real-world systems and devices. These selected applications include digital signal generation and dual-tone multi-frequency (DTMF) detection in Chapter 7; adaptive echo cancellation especially for VoIP and hands-free phone applications in Chapter 8; speech processing algorithms including speech enhancement and coding techniques for mobile phones in Chapter 9; audio signal processing including audio effects, equalizers, and coding methods for portable players in Chapter 10; and image processing fundamentals for applications including JPEG2000 and fingerprints in Chapter 11. Finally, Appendix A summarizes some useful formulas used for the derivation of equations and solving exercise problems in the book, and Appendix C introduces the architecture and assembly programming of the TMS320C55xx for readers who are interested in these topics.

As with any book attempting to capture the state of the art at a given time, there will certainly be updates that are necessitated by the rapidly evolving developments in this dynamic field. We hope that this book can serve as a guide for what has already come and as an inspiration for what will follow.

Software Availability

This book utilizes various MATLAB®, floating-point and fixed-point C, and TMS320C55xx assembly programs in examples, experiments, and applications. These programs along with many data files are available in the companion software package from the Wiley website (http://www.wiley.com/go/kuo_dsp). The directory and the subdirectory structure and names of these software programs and data files are explained and listed in Appendix B. The software is required for conducting the experiments presented in the last section of each chapter and Appendix C, and can enhance the understanding of DSP principles. The software can also be modified to serve as prototypes for speeding up other practical uses.

Acknowledgments

We are grateful to Cathy Wicks and Gene Frantz of Texas Instruments, and to Naomi Fernandes and Courtney Esposito of MathWorks, for providing us with the support needed to write this book. We would like to thank several individuals at John Wiley & Sons, Ltd for their support of this project: Alexandra King, Commissioning Editor; Liz Wingett, Project Editor; and Richard Davies, Senior Project Editor. We would also like to thank the staff at John Wiley & Sons, Ltd for the final preparation of this book. Our thanks also go to Hui Tian for creating the special audio clips used in the examples and experiments. Finally, we thank our families for their endless love, encouragement, patience, and understanding they have shown since we began our work on the first edition in the late 1990s.

Sen M. Kuo, Bob H. Lee, and Wenshun Tian

1

Introduction to Real-Time Digital Signal Processing

Signals can be classified into three categories: continuous-time (analog) signals, discrete-time signals, and digital signals. The signals that we encounter daily are mostly analog signals. These signals are defined continuously in time, have infinite resolution of amplitude values, and can be processed using analog electronics containing both active and passive circuit elements. Discrete-time signals are defined only at a particular set of time instances, thus they can be represented as a sequence of numbers that have a continuous range of values. Digital signals have discrete values in both time and amplitude, thus they can be stored and processed by computers or digital hardware. In this book, we focus on the design, implementation, and applications of digital systems for processing digital signals [1–6]. However, the theoretical analysis usually uses discrete-time signals and systems for mathematical convenience. Therefore, we use the terms "discrete-time" and "digital" interchangeably.

Digital signal processing (DSP) is concerned with the digital representation of signals and the use of digital systems to analyze, modify, store, transmit, or extract information from these signals. In recent years, the rapid advancement in digital technologies has enabled the implementation of sophisticated DSP algorithms for real-time applications. DSP is now used not only in areas where analog methods were used previously, but also in areas where analog techniques are very difficult or impossible to apply.

There are many advantages in using digital techniques for signal processing rather than analog devices such as amplifiers, modulators, and filters. Some of the advantages of DSP systems over analog circuitry are summarized as follows:

1. *Flexibility.* Functions of a DSP system can be easily modified and upgraded with software that implements the specific operations. One can design a DSP system to perform a wide variety of tasks by executing different software modules. A digital device can be easily upgraded in the field through the on-board memory (e.g., flash memory) to meet new requirements, add new features, or enhance its performance.

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

2. *Reproducibility.* The functions of a DSP system can be repeated precisely from one unit to another. In addition, by using DSP techniques, digital signals can be stored, transferred, or reproduced many times without degrading the quality. By contrast, analog circuits will not have the same characteristics even if they are built following identical specifications, due to analog component tolerances.
3. *Reliability.* The memory and logic of DSP hardware do not deteriorate with age. Therefore, the performance of DSP systems will not drift with changing environmental conditions or aged electronic components as their analog counterparts do.
4. *Complexity.* DSP allows sophisticated applications such as speech recognition to be implemented using low-power and lightweight portable devices. Furthermore, there are some important signal processing algorithms such as image compression and recognition, data transmission and storage, and audio compression, which can only be performed using DSP systems.

With the rapid evolution in semiconductor technologies, DSP systems have lower overall cost compared to analog systems for most applications. DSP algorithms can be developed, analyzed, and simulated using high-level language software such as C and MATLAB®. The performance of the algorithms can be verified using low-cost, general-purpose computers. Therefore, DSP systems are relatively easy to design, develop, analyze, simulate, test, and maintain.

There are some limitations associated with DSP. For example, the bandwidth of a DSP system is limited by the sampling rate. Also, most of the DSP algorithms are implemented using a fixed number of bits with limited precision and dynamic range, resulting in undesired quantization and arithmetic errors.

1.1 Basic Elements of Real-Time DSP Systems

There are two types of DSP applications: non-real-time and real-time. Non-real-time signal processing involves manipulating signals that have already been stored in digital form. This may or may not represent a current action, and the processing result is not a function of real time. Real-time signal processing places stringent demands on DSP hardware and software design to complete predefined tasks within a given timeframe. This section reviews the fundamental functional blocks of real-time DSP systems.

The basic functional blocks of DSP systems are illustrated in Figure 1.1, where a real-world analog signal is converted to a digital signal, processed by DSP hardware, and converted back

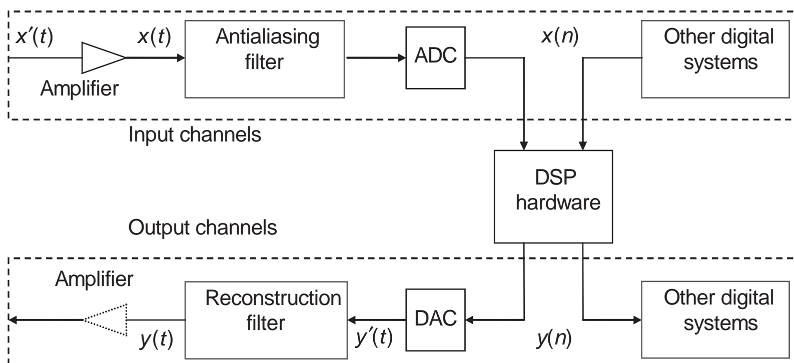


Figure 1.1 Basic functional block diagram of a real-time DSP system

to an analog signal. For some applications, the input signal may be already in digital form and/or the output data may not need to be converted to an analog signal, for example, the processed digital information may be stored in memory for later use. In other applications, DSP systems may be required to generate signals digitally, such as speech synthesis and signal generators.

1.2 Analog Interface

In this book, a time-domain signal is denoted with a lowercase letter. For example, $x(t)$ in Figure 1.1 is used to name an analog signal of x which is a function of time t. The time variable t and the amplitude of $x(t)$ take on a continuum of values between $-\infty$ and ∞. For this reason we say $x(t)$ and $y(t)$ are continuous-time (or analog) signals. The signals $x(n)$ and $y(n)$ in Figure 1.1 depict digital signals which have values only at time instant (or index) n. In this section, we first discuss how to convert analog signals into digital signals. The process of converting an analog signal to a digital signal is called the analog-to-digital (A/D) conversion, usually performed by an A/D converter (ADC).

The purpose of A/D conversion is to convert the analog signal to digital form for processing by digital hardware. As shown in Figure 1.1, the analog signal $x'(t)$ is picked up by an appropriate electronic sensor that converts pressure, temperature, or sound into electrical signals. For example, a microphone can be used to pick up speech signals. The sensor output signal $x'(t)$ is amplified by an amplifier with a gain of value g to produce the amplified signal

$$x(t) = gx'(t). \tag{1.1}$$

The gain value g is determined such that $x(t)$ has a dynamic range that matches the ADC used by the system. If the peak-to-peak voltage range of the ADC is ± 2 volts (V), then g may be set so that the amplitude of signal $x(t)$ to the ADC is within ± 2 V. In practice, it is very difficult to set an appropriate fixed gain because the level of $x'(t)$ may be unknown and changing with time, especially for signals with larger dynamic ranges such as human speech. Therefore, many practical systems use digital automatic gain control algorithms to determine and update the gain value g based on the statistics of the input signal $x'(t)$.

Once the digital signal has been processed by the DSP hardware, the result $y(n)$ is still in digital form. In many DSP applications, we have to convert the digital signal $y(n)$ back to the analog signal $y(t)$ before it can be applied to appropriate analog devices. This process is called the digital-to-analog (D/A) conversion, typically performed by a D/A converter (DAC). One example is a digital audio player, in which the audio music signals are stored in digital format. An audio player reads the encoded digital audio data from the memory and reconstructs the corresponding analog waveform for playback.

The system shown in Figure 1.1 is a real-time system if the signal to the ADC is continuously sampled and processed by the DSP hardware at the same rate. In order to maintain real-time processing, the DSP hardware must perform all required operations within the fixed time, and present the output sample to the DAC before the arrival of the next sample from the ADC.

1.2.1 Sampling

As shown in Figure 1.1, the ADC converts the analog signal $x(t)$ into the digital signal $x(n)$. The A/D conversion, commonly referred to as digitization, consists of the sampling (digitization in time) and quantization (digitization in amplitude) processes as illustrated in Figure 1.2. The basic sampling function can be carried out with an ideal "sample-and-hold"

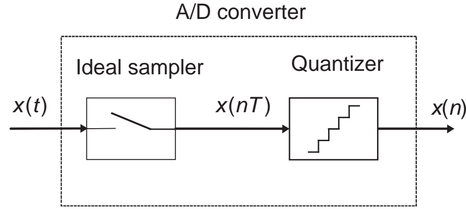


Figure 1.2 Block diagram of an ADC

circuit, which maintains the sampled signal level until the next sample is taken. The quantization process approximates the waveform by assigning a number to represent its value for each sample. Therefore, the A/D conversion performs the following steps:

1. The signal $x(t)$ is sampled at uniformly spaced time instants nT, where n is a positive integer and T is the sampling period in seconds. This sampling process converts an analog signal into a discrete-time signal $x(nT)$ with continuous amplitude value.
2. The amplitude of each discrete-time sample $x(nT)$ is quantized into one of 2^B levels, where B is the number of bits used to represent each sample. The discrete amplitude levels are represented (or encoded) into binary words $x(n)$ with the fixed wordlength B.

The reason for making this distinction is that these two processes introduce different distortions. The sampling process causes aliasing or folding distortion, while the encoding process results in quantization noise. As shown in Figure 1.2, the sampler and quantizer are integrated on the same chip. However, a high-speed ADC typically requires an external sample-and-hold device.

An ideal sampler can be considered as a switch that periodically opens and closes every T seconds. The sampling period is defined as

$$T = \frac{1}{f_s}, \tag{1.2}$$

where f_s is the sampling frequency in hertz (Hz) or sampling rate in samples per second. The intermediate signal $x(nT)$ is a discrete-time signal with continuous value (a number with infinite precision) at discrete time nT, $n = 0, 1, \ldots, \infty$, as illustrated in Figure 1.3. The

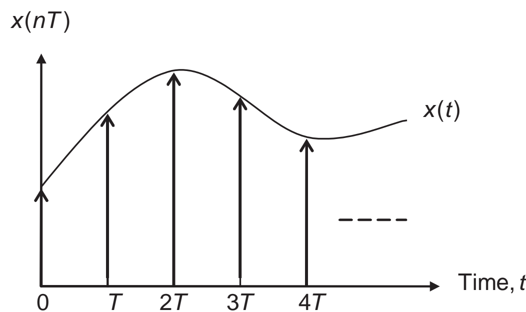


Figure 1.3 Sampling of analog signal $x(t)$ and the corresponding discrete-time signal $x(nT)$

analog signal $x(t)$ is continuous in both time and amplitude. The sampled discrete-time signal $x(nT)$ is continuous in amplitude, but defined only at discrete sampling instants $t = nT$.

In order to represent the analog signal $x(t)$ accurately by the discrete-time signal $x(nT)$, the sampling frequency f_s must be at least twice the maximum frequency component f_M in the analog signal $x(t)$. That is,

$$f_s \geq 2f_M, \tag{1.3}$$

where f_M is also called the bandwidth of the bandlimited signal $x(t)$. This is Shannon's sampling theorem, which states that when the sampling frequency is greater than or equal to twice the highest frequency component contained in the analog signal, the original analog signal $x(t)$ can be perfectly reconstructed from the uniformly sampled discrete-time signal $x(nT)$.

The minimum sampling rate, $f_s = 2f_M$, is called the Nyquist rate. The frequency, $f_N = f_s/2$, is called the Nyquist frequency or folding frequency. The frequency interval, $[-f_s/2, f_s/2]$, is called the Nyquist interval. When the analog signal is sampled at f_s, frequency components higher than $f_s/2$ will fold back into the frequency range $[0, f_s/2]$. The folded back frequency components overlap with the original frequency components in the same frequency range, resulting in the corrupted signal. Therefore, the original analog signal cannot be recovered from the folded digital samples. This undesired effect is known as aliasing.

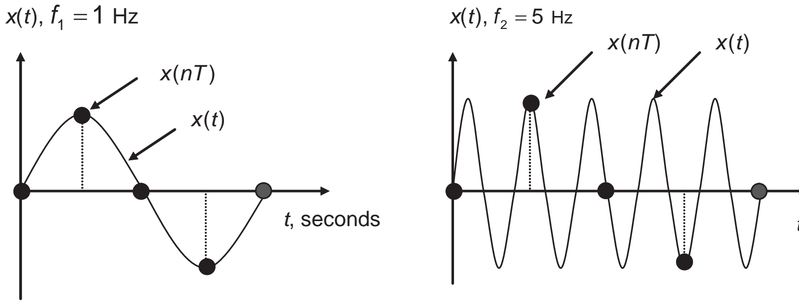
Example 1.1

Consider two sine waves of frequencies $f_1 = 1$ Hz and $f_2 = 5$ Hz that are sampled at $f_s = 4$ Hz, rather than at least 10 Hz according to the sampling theorem. The analog waveforms and the digital samples are illustrated in Figure 1.4(a), while their digital samples and reconstructed waveforms are illustrated in Figure 1.4(b). As shown in the figures, we can reconstruct the original waveform from the digital samples for the sine wave of frequency $f_1 = 1$ Hz. However, for the original sine wave of frequency $f_2 = 5$ Hz, the resulting digital samples are the same as $f_1 = 1$ Hz, thus the reconstructed signal is identical to the sine wave of frequency 1 Hz. Therefore, f_1 and f_2 are said to be aliased to one another, that is, they cannot be distinguished by their discrete-time samples.

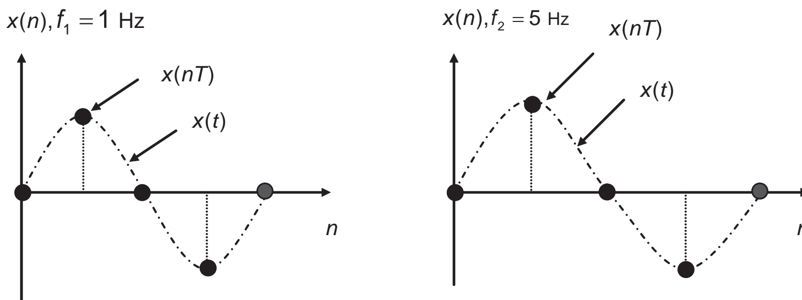
Note that the sampling theorem assumes the signal is bandlimited by f_M. For many practical applications, the analog signal $x(t)$ may have significant frequency components outside the highest frequency of interest, or may contain noise with a wider bandwidth. In some applications, the sampling rate is predetermined by given specifications. For example, most voice communication systems define the sampling rate of 8 kHz (kilohertz). Unfortunately, the frequency components in typical speech can be much higher than 4 kHz. To guarantee that the sampling theorem is satisfied, we must eliminate the frequency components above the Nyquist frequency. This can be done by using an antialiasing filter which is an analog lowpass filter with the cutoff frequency bounded by

$$f_c \leq \frac{f_s}{2}. \tag{1.4}$$

Ideally, an antialiasing filter should remove all frequency components above the Nyquist frequency. In many practical systems, a bandpass filter is preferred to remove frequency



(a) Original analog waveforms and digital samples for $f_1 = 1$ Hz and $f_2 = 5$ Hz.



(b) Digital samples of $f_1 = 1$ Hz and $f_2 = 5$ Hz and the reconstructed waveforms.

Figure 1.4 Example of the aliasing phenomenon

components above the Nyquist frequency, as well as to eliminate undesired DC offset, 60 Hz hum, or other low-frequency noises. For example, a bandpass filter with a passband from 300 to 3400 Hz is widely used in telecommunication systems to attenuate the signals whose frequencies lie outside this passband.

Example 1.2

The frequency range of signals is large, from approximately gigahertz (GHz) in radar down to fractions of hertz in instrumentation. For a specific application with given sampling rate, the sampling period can be determined by (1.2). For example, some real-world applications use the following sampling frequencies and periods:

1. In International Telecommunication Union (ITU) speech coding/decoding standards ITU-T G.729 [7] and G.723.1 [8], the sampling rate is $f_s = 8$ kHz, thus the sampling period $T = 1/8000$ seconds $= 125\ \mu$s (microseconds). Note that $1\ \mu\text{s} = 10^{-6}$ seconds.
2. Wideband telecommunication speech coding standards, such as ITU-T G.722 [9] and G.722.2 [10], use the sampling rate of $f_s = 16$ kHz, thus $T = 1/16\,000$ seconds $= 62.5\ \mu$s.
3. High-fidelity audio compression standards, such as MPEG-2 (Moving Picture Experts Group) [11], AAC (Advanced Audio Coding), MP3 (MPEG-1 layer 3) [12] audio, and

Dolby AC-3, support the sampling rate of $f_s = 48$ kHz, thus $T = 1/48\,000$ seconds = 20.833 μs. The sampling rate for MPEG-2 AAC can be as high as 96 kHz.

The speech coding algorithms will be discussed in Chapter 9 and the audio coding techniques will be introduced in Chapter 10.

1.2.2 Quantization and Encoding

In previous sections, we assumed that the sample values $x(nT)$ are represented exactly using an infinite number of bits (i.e., $B \rightarrow \infty$). We now discuss the quantization and encoding processes for representing the sampled discrete-time signal $x(nT)$ by a binary number with a finite number of bits. If the wordlength of an ADC is B bits, there are 2^B different values (levels) that can be used to represent a digital sample $x(n)$. If $x(nT)$ lies in between two quantization levels, it will either be rounded or truncated to produce $x(n)$. Rounding assigns to $x(nT)$ the value of the nearest quantization level while truncation replaces $x(nT)$ by the value of the level below it. Since rounding produces a less biased representation of the true value, it is widely used by ADCs. Therefore, quantization is a process that represents a continuous-valued sample $x(nT)$ with its nearest level that corresponds to the digital signal $x(n)$.

For example, 2 bits define four equally spaced levels (00, 01, 10, and 11), which can be used to classify the signal into the four subranges illustrated in Figure 1.5. In this figure, the open circles represent the discrete-time signal $x(nT)$, and the solid circles the digital signal $x(n)$. The spacing between two consecutive quantization levels is called the quantization width, step, or resolution. A uniform quantizer has the same spacing between these levels. For uniform quantization, the resolution is determined by dividing the full-scale range by the total number of quantization levels, 2^B.

In Figure 1.5, the difference between the quantized number and the original value is defined as the quantization error, which appears as noise in the output of the converter. Thus, the quantization error is also called the quantization noise, which is assumed to be a random noise. If a B-bit quantizer is used, the signal-to-quantization-noise ratio (SQNR) is approximated by the following equation (to be derived in Chapter 2):

$$\text{SQNR} \approx 6B \text{ dB}. \tag{1.5}$$

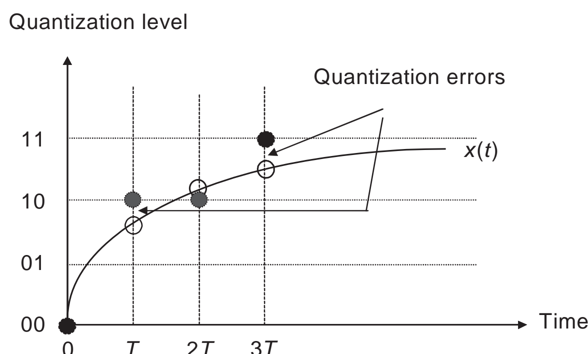


Figure 1.5 Digital samples using 2-bit quantizer

In practice, the achievable SQNR will be less than this theoretical value due to imperfections in the fabrication of converters. Nevertheless, Equation (1.5) provides a simple guideline to determine the required bits for a given application. For each additional bit, a digital signal will have about 6 dB gain in SQNR. The problems of quantization noise and their solutions will be further discussed in Chapter 2.

Example 1.3

If the analog signal varies between 0 and 5 V, we have the resolutions and SQNRs for the following commonly used ADCs:

1. An 8-bit ADC with 256 (2^8) levels can only provide 19.5 mV resolution and 48 dB SQNR.
2. A 12-bit ADC has 4096 (2^{12}) levels of 1.22 mV resolution, and provides 72 dB SQNR.
3. A 16-bit ADC has 65536 (2^{16}) levels, and thus provides 76.294 μV resolution with 96 dB SQNR.

Obviously, using more bits results in more quantization levels (or finer resolution) and higher SQNR.

The dynamic range of speech signals is usually very large. If the uniform quantization scheme is adjusted for loud sounds, most of the softer sounds may be pressed into the same small values. This means that soft sounds may not be distinguishable. To solve this problem, we can use a quantizer with quantization level varying according to the signal amplitude. For example, if the signal has been compressed by a logarithm function, we can use a uniform level quantizer to perform non-uniform quantization by quantizing the logarithm-scaled signal. The compressed signal can be reconstructed by expanding it. The process of compression and expansion is called companding (compressing and expanding). The ITU-T G.711 μ-law (used in North America and parts of Northeast Asia) and A-law (used in Europe and most of the rest of the world) schemes [13] are examples of using companding technology, which will be further discussed in Chapter 9.

As shown in Figure 1.1, the input signal to DSP hardware may be digital signals from other digital systems that use different sampling rates. The signal processing techniques called interpolation or decimation can be used to increase or decrease the sampling rates of the existing digital signals. Sampling rate changes may be required in many multi-rate DSP systems, for example, between the narrowband voice sampled at 8 kHz and wideband voice sampled at 16 kHz. The interpolation and decimation processes will be introduced in Chapter 3.

1.2.3 Smoothing Filters

Most commercial DACs are zero-order-hold devices, meaning they convert the input binary number to the corresponding voltage level and then hold that level for T seconds. Therefore, the DAC produces the staircase-shaped analog waveform $y'(t)$ as shown by the solid line in Figure 1.6, which is a rectangular waveform with amplitude corresponding to the signal value with a duration of T seconds. Obviously, this staircase output signal contains many high-frequency components due to an abrupt change in signal levels. The reconstruction or

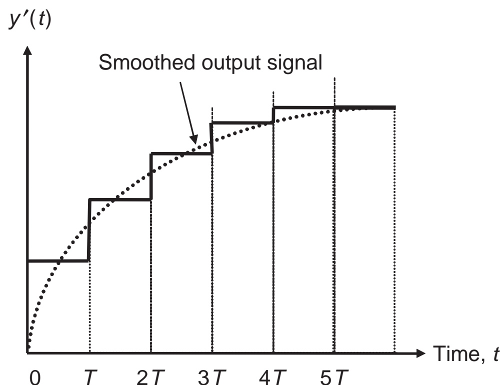


Figure 1.6 Staircase waveform generated by DAC and the smoothed signal

smoothing filter shown in Figure 1.1 smoothes the staircase-like analog signal generated by the DAC. This lowpass filtering has the effect of rounding off the corners (high-frequency components) of the staircase signal and making it smoother, which is shown as the dotted line in Figure 1.6. This analog lowpass filter may have the same specifications as the antialiasing filter with cutoff frequency $f_c \leq f_s/2$. Some high-quality DSP applications, such as professional digital audio, require the use of reconstruction filters with very stringent specifications. To reduce the cost of using high-quality analog filters, the oversampling technique can be adopted to allow the use of low-cost filters with slower roll-off.

1.2.4 Data Converters

There are two methods of connecting an ADC and DAC to a digital signal processor: serial and parallel. A parallel converter receives or transmits all *B* bits in one pass, while a serial converter receives or transmits *B* bits in a serial of bit stream, 1 bit at a time. Parallel converters are attached to the digital signal processor's external address and data buses, which are also attached to many different types of devices. Serial converters can be connected directly to the built-in serial ports of digital signal processors. Since serial converters require a few signals (pins) to connect with digital signal processors, many practical DSP systems use serial ADCs and DACs.

Many applications use a single-chip device called an analog interface chip (AIC) or coder/decoder (CODEC or codec), which integrates an antialiasing filter, ADC, DAC, and reconstruction filter on a single chip. In this book, we will use Texas Instruments AIC3204 on the TMS320C5505 eZdsp USB (universal serial bus) stick for real-time experiments. Typical applications using a CODEC include speech systems, audio systems, and industrial controllers. Many standards that specify the nature of the CODEC have evolved for the purposes of switching and transmission. Some CODECs use a logarithmic quantizer, that is, *A*-law or μ-law, which must be converted into linear format for processing. Digital signal processors implement the required format conversion (compression or expansion) either by hardware or software.

The most popular commercially available ADCs are successive approximation, dual-slope, flash, and sigma–delta. The successive-approximation type of ADC is generally accurate and

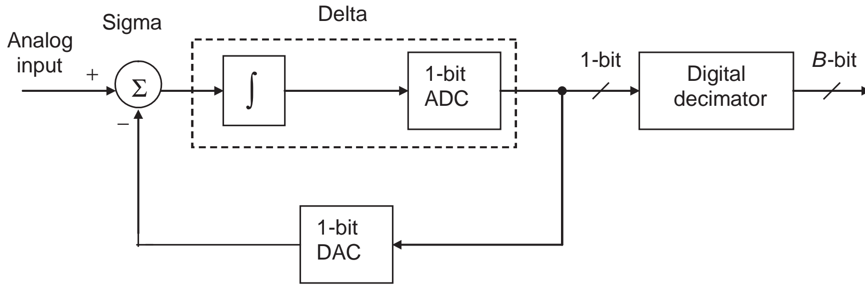


Figure 1.7 A conceptual sigma–delta ADC block diagram

fast at a relatively low cost. However, its ability to follow changes in the input signal is limited by its internal clock rate, so it may be slow to respond to sudden changes in the input signal. The dual-slope ADC is very precise and can produce ADCs with high resolution. However, they are very slow and generally cost more than successive-approximation ADCs. The major advantage of a flash ADC is its speed of conversion; unfortunately, a B-bit ADC requires $(2^B - 1)$ expensive comparators and laser-trimmed resistors. Therefore, commercially available flash ADCs usually have lower bits.

Sigma–delta ADCs use oversampling and quantization noise shaping to trade the quantizer resolution with sampling rate. A block diagram of a sigma–delta ADC is illustrated in Figure 1.7, which uses a 1-bit quantizer with a very high sampling rate. Thus, the requirements for an antialiasing filter are significantly relaxed (i.e., a lower roll-off rate). A low-order antialiasing filter requires simple low-cost analog circuitry and is much easier to build and maintain. In the process of quantization, the resulting noise power is spread evenly over the entire spectrum. The quantization noise beyond the required spectrum range can be attenuated using a digital lowpass filter. As a result, the noise power within the frequency band of interest is lower. In order to match the sampling frequency with the system and increase its resolution, a decimator is used to reduce the sampling rate. The advantages of sigma–delta ADCs are high resolution and good noise characteristics at a competitive price using digital decimation filters.

In this book, as mentioned above, we use the AIC3204 stereo CODEC on the TMS320C5505 eZdsp for real-time experiments. The ADCs and DACs within the AIC3204 use a sigma–delta technology with integrated digital lowpass filters. It supports a data wordlength of 16, 20, 24, and 32 bits, with sampling rates from 8 to 192 kHz. Integrated analog features consist of stereo-line input amplifiers with programmable analog gains and stereo headphone amplifiers with programmable analog volume control.

1.3 DSP Hardware

Most DSP systems are required to perform intensive arithmetic operations such as repeated multiplications and additions. These operations may be implemented on digital hardware such as microprocessors, microcontrollers, digital signal processors, or custom integrated circuits. The selection of appropriate hardware can be determined by the given application based on the performance, cost, and/or power consumption. In this section, we will introduce several different digital hardware options for DSP applications.

1.3.1 DSP Hardware Options

As shown in Figure 1.1, the processing of digital signal $x(n)$ is performed using the DSP hardware. Although it is possible to implement DSP algorithms on different digital hardware, the given application determines the optimum hardware platform. The following hardware options are widely used for DSP systems:

1. Special-purpose (custom) chips such as application-specific integrated circuit (ASICs).
2. Field-programmable gate arrays (FPGAs).
3. General-purpose microprocessors or microcontrollers (μP/μC).
4. General-purpose digital signal processors.
5. Digital signal processors with application-specific hardware (HW) accelerators.

The characteristics of these hardware options are summarized in Table 1.1.

ASIC devices are usually designed for specific tasks that require intensive computation such as digital subscriber loop (DSL) modems, or high-volume products that use mature algorithms such as fast Fourier transforms. These devices perform the required functions much faster because their dedicated architecture is optimized for the required operations, but they lack flexibility to modify the specific algorithms and functions for new applications. They are suitable for implementing well-defined and popular DSP algorithms for high-volume products, or applications demanding extremely high speeds that can only be achieved by ASICs. Recently, the availability of core modules for some common DSP functions can simplify ASIC design tasks, but the cost of prototyping ASIC devices, the longer design cycle, and the lack of standard development tool support and reprogramming flexibility sometimes outweigh their benefits.

FPGAs have been used in DSP systems for years as glue logics, bus bridges, and peripherals for reducing system costs and affording higher levels of system integration. Recently, FPGAs have been gaining considerable attention in high-performance DSP applications, and are emerging as coprocessors [14] for standard digital signal processors that need specific accelerators. In these cases, FPGAs work in conjunction with digital signal processors for integrating pre- and post-processing functions. These devices are hardware reconfigurable, and thus allow system designers to optimize the hardware architecture for implementing algorithms that require higher performance and lower production cost. In addition, designers can implement high-performance complex DSP functions using a fraction of the device, and use the rest of the device to implement system logic or interface functions, resulting in both lower costs and higher system integration.

Table 1.1 Summary of DSP hardware implementations

	ASIC	FPGA	μP/μC	Digital signal processor	Digital signal processors with HW accelerators
Flexibility	None	Limited	High	High	Medium
Design time	Long	Medium	Short	Short	Short
Power consumption	Low	Low–medium	Medium–high	Low–medium	Low–medium
Performance	High	High	Low–medium	Medium–high	High
Development cost	High	Medium	Low	Low	Low
Production cost	Low	Low–medium	Medium–high	Low–medium	Medium

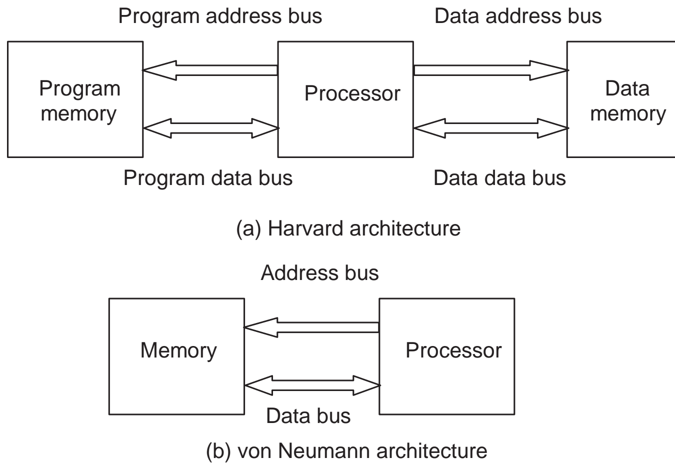


Figure 1.8 Different memory architectures

General-purpose μP/μC become faster and increasingly capable of handling some DSP applications. Many electronics products such as automotive controllers use microcontrollers for engine, brake, and suspension control and are often designed using these processors. If new DSP functions are needed for an existing product based on μP/μC, it is preferable to implement these functions in software than modify existing hardware.

General μP/μC architectures fall into two categories: Harvard architecture and von Neumann architecture. As illustrated in Figure 1.8(a), Harvard architecture has separate memory spaces for the program and the data, thus both memories can be accessed simultaneously. The von Neumann architecture assumes that the program and data are stored in the same memory as illustrated in Figure 1.8(b). Operations such as add, move, and subtract are easy to perform on μP/μC. However, complex instructions such as multiplication and division are slow since they need a series of conditional shift, addition, or subtraction operations. These devices do not have the architecture or on-chip facilities required for efficient DSP operations, and they are not cost effective or power efficient for many DSP applications. It is important to note that some modern microprocessors, specifically for mobile and portable devices, can run at high speed, consume low power, provide single-cycle multiplication and arithmetic operations, have good memory bandwidth, and have many supporting tools and software available for ease of development.

A digital signal processor is basically a microprocessor with architecture and instruction set designed specifically for DSP applications [15–17]. The rapid growth and exploitation of digital signal processor technology is not a surprise, considering the commercial advantages in terms of the fast, flexible, low-power consumption, and potentially low-cost design capabilities offered by these devices. In comparison to ASIC and FPGA solutions, digital signal processors have advantages in ease of development and being reprogrammable in the field to upgrade product features or fix bugs. They are often more cost effective than custom hardware such as ASIC and FPGA, especially for low-volume applications. In comparison to general-purpose μP/μC, digital signal processors have better speed, better energy efficiency or power consumption, and lower cost for many DSP applications.

Today, digital signal processors have become the foundation of many new markets beyond the traditional signal processing areas for technologies and innovations in motor and motion control, automotive systems, home appliances, consumer electronics, medical and healthcare devices, and a vast range of communication and broadcasting equipment and systems. These general-purpose programmable digital signal processors are supported by integrated software development tools including C compilers, assemblers, optimizers, linkers, debuggers, simulators, and emulators. In this book, we use the TMS320C55xx for hands-on experiments.

1.3.2 Digital Signal Processors

In 1979, Intel introduced the 2920, a 25-bit integer processor with a 400 ns instruction cycle and a 25-bit arithmetic logic unit (ALU) for DSP applications. In 1982, Texas Instruments introduced the TMS32010, a 16-bit fixed-point processor with a 16×16 hardware multiplier and a 32-bit ALU and accumulator. This first commercially successful digital signal processor was followed by the development of faster products and floating-point processors. Their performance and price range vary widely.

Conventional digital signal processors include hardware multipliers and shifters, execute one instruction per clock cycle, and use the complex instructions that perform multiple operations such as multiply, accumulate, and update address pointers. They provide good performance with modest power consumption and memory usage, and thus are widely used in automotive applications, appliances, hard disk drives, and consumer electronics. For example, the TMS320C2000 family is optimized for control applications, such as motor and automobile control, by integrating many microcontroller features and peripherals on the chip.

The midrange processors achieve higher performance through the combination of increased clock rates and more advanced architectures. These processors often include deeper pipelines, instruction caches, complex instructions, multiple data buses (to access several data words per clock cycle), additional hardware accelerators, and parallel execution units to allow more operations to be executed in parallel. For example, the TMS320C55xx has two multiply and accumulate (MAC) units. These midrange processors provide better performance with lower power consumption, thus are typically found in portable applications such as medical and healthcare devices like digital hearing aids.

These conventional and enhanced digital signal processors have the following features for common DSP algorithms:

- *Fast MAC units.* The multiply–add or multiply–accumulate operation is required in most DSP functions including filtering, fast Fourier transform, and correlation. To perform the MAC operation efficiently, digital signal processors integrate the multiplier and accumulator into the same data path to complete the MAC operation in a single instruction cycle.
- *Multiple memory accesses.* Most DSP processors adopted modified Harvard architectures that keep the program memory and data memory separate to allow simultaneous fetch of instruction and data. In order to support simultaneous access of multiple data words, digital signal processors provide multiple on-chip buses, independent memory banks, and on-chip dual-access data memory.
- *Special addressing modes.* digital signal processors often incorporate dedicated data address generation units for generating data addresses in parallel with the execution of

instructions. These units usually support circular addressing and bit-reversed addressing needed for some commonly used DSP algorithms.

- *Special program control.* Most digital signal processors provide zero-overhead looping, which allows the implementation of loops and repeat operations without extra clock cycles for updating and testing loop counters, or branching back to the top of the loop.
- *Optimized instruction set.* Digital signal processors provide special instructions that support computationally intensive DSP algorithms.
- *Effective peripheral interface.* Digital signal processors usually incorporate high-performance serial and parallel input/output (I/O) interfaces to other devices such as ADCs and DACs. They provide streamlined I/O handling mechanisms such as buffered serial ports, direct memory access (DMA) controllers, and low-overhead interrupt to transfer data with little or no intervention from the processor's computational units.

These digital signal processors use specialized hardware and complex instructions to allow more operations to be executed in a single instruction cycle. However, they are difficult to program using assembly language and it is difficult to design efficient C compilers in terms of speed and memory usage for supporting these complex instruction architectures.

With the goals of achieving high performance and creating architectures that support efficient C compilers, some digital signal processors use very simple instructions. These processors achieve a high level of parallelism by issuing and executing multiple simple instructions in parallel at higher clock rates. For example, the TMS320C6000 uses the very long instruction word (VLIW) architecture that provides eight execution units to execute four to eight instructions per clock cycle. These instructions have few restrictions on register usage and addressing modes, thus improving the efficiency of C compilers. However, the disadvantage of using simple instructions is that the VLIW processors need more instructions to complete a given task, and thus require relatively high program memory space. These high-performance digital signal processors are typically used in high-end video and radar systems, communication infrastructures, wireless base stations, and high-quality real-time video encoding systems.

1.3.3 Fixed- and Floating-Point Processors

A basic distinction between digital signal processors is the arithmetic format: fixed-point or floating-point. This is the most important factor for system designers to determine the suitability of the processor for the given application. The fixed-point representation of signals and arithmetic will be discussed in Chapter 2. Fixed-point digital signal processors are either 16-bit or 24-bit devices, while floating-point processors are usually 32-bit devices. A typical 16-bit fixed-point processor, such as the TMS320C55xx, stores numbers as 16-bit integers. Although coefficients and signals are stored only with 16-bit precision, intermediate values (products) may be kept at 32-bit precision within the internal 40-bit accumulators in order to reduce cumulative rounding errors. Fixed-point DSP devices are usually cheaper and faster than their floating-point counterparts because they use less silicon, have lower power consumption, and require fewer external pins. Most high-volume, low-cost embedded applications such as appliances, hard disk drives, audio players and digital cameras use fixed-point processors.

Floating-point arithmetic greatly expands the dynamic range of numbers. A typical 32-bit floating-point digital signal processor, such as the TMS320C67xx, represents numbers with a 24-bit mantissa and 8-bit exponent. The mantissa represents a fraction in the range -1.0 to $+1.0$, while the exponent is an integer that represents the number of binary points that must be shifted left or right in order to obtain the true value. A 32-bit floating-point format covers a large dynamic range, thus the data dynamic range restrictions may be virtually ignored in the design using floating-point processors. This is in contrast to the design of fixed-point systems, where the designer has to apply scaling factors and other techniques to prevent arithmetic overflows, which are very difficult and time-consuming processes. Therefore, floating-point digital signal processors are generally easy to program and use with higher performance, but are usually more expensive and have higher power consumption.

Example 1.4

The precision and dynamic ranges of 16-bit fixed-point processors are summarized in the following table:

	Precision	Dynamic range
Unsigned integer	1	$0 \le x \le 65\,535$
Signed integer	1	$-32\,768 \le x \le 32\,767$
Unsigned fraction	2^{-16}	$0 \le x \le (1 - 2^{-16})$
Signed fraction	2^{-15}	$-1 \le x \le (1 - 2^{-15})$

The precision of 32-bit floating-point processors is 2^{-23} since there are 24 mantissa bits. The dynamic range is $1.18 \times 10^{-38} \le x \le 3.4 \times 10^{38}$.

System designers have to determine the dynamic range and precision needed for the applications. Floating-point processors may be needed in applications where coefficients vary in time and the signals and coefficients require large dynamic ranges and high precision. Floating-point processors also support the efficient use of high-level C compilers, thus reducing the cost of development and maintenance. The faster development cycle for floating-point processors may easily outweigh the extra cost of the processor itself for low-quantity products. Therefore, floating-point processors also can be justified for applications where development costs are high and/or production volumes are low.

1.3.4 Real-Time Constraints

A major limitation of DSP systems for real-time applications is the bandwidth of the system. The processing speed determines the maximum rate at which the analog signal can be sampled. For example, with sample-by-sample processing, one output sample is generated before the new input sample is presented to the system. Therefore, the time delay between the input and output for sample-by-sample processing must be less than one sampling interval (T seconds). A real-time DSP system demands that the signal processing time, t_p, must be less

than the sampling period, T, in order to complete the processing before the new sample comes in. That is,

$$t_p + t_o < T, \tag{1.6}$$

where t_o is the overhead of I/O operations.

This hard real-time constraint limits the highest frequency signal that can be processed by DSP systems using sample-by-sample processing approach. This limit on real-time bandwidth f_M is given as

$$f_M \leq \frac{f_s}{2} < \frac{1}{2(t_p + t_o)}. \tag{1.7}$$

It is clear that the longer the processing time t_p, the lower the signal bandwidth that can be handled by the system.

Although new and faster digital signal processors have been continuously introduced, there is still a limit to the processing that can be done in real time. This limit becomes even more apparent when system cost is taken into consideration. Generally, the real-time bandwidth can be increased by using faster digital signal processors, simplified DSP algorithms, optimized DSP programs, and multiple processors or multi-core processors, and so on. However, there is still a trade-off between system cost and performance.

Equation (1.7) also shows that the real-time bandwidth can be increased by reducing the overhead of I/O operations. This can be achieved by using a block-by-block processing approach. With block processing methods, the I/O operations are usually handled by DMA controllers, which place data samples in memory buffers. The DMA controller interrupts the processor when the input buffer is full and the block of signal samples are available for processing. For example, for real-time N-point fast Fourier transforms (to be discussed in Chapter 5), the N input samples have to be buffered by the DMA controller. The block computation must be completed before the next block of N samples arrives. Therefore, the time delay between input and output in block processing is dependent on the block size N, and this may cause a problem for some applications.

1.4 DSP System Design

A generalized DSP system design process is illustrated in Figure 1.9. For a given application, signal analysis, resource analysis, and configuration analysis are first performed to define the system specifications.

1.4.1 Algorithm Development

A DSP system is often characterized by the embedded algorithm, which specifies the arithmetic operations to be performed. The algorithm for a given application is initially described using difference equations and/or signal-flow block diagrams with symbolic names for the inputs and outputs. The next stage of the development process is to provide more details on the sequence of operations that must be performed in order to derive the output. There are two methods of characterizing the sequence of operations in a program: flowcharts or structured descriptions.

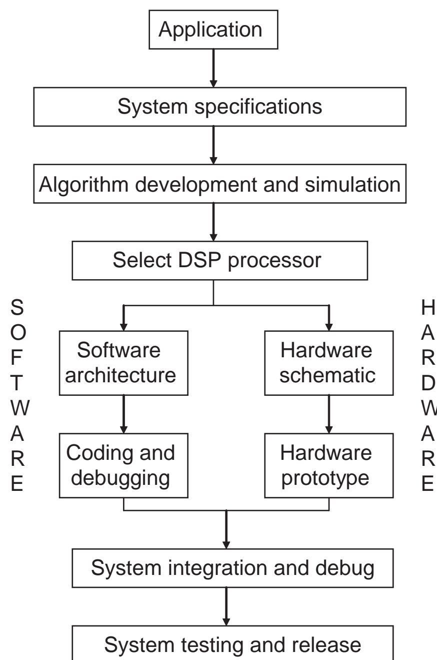


Figure 1.9 Simplified DSP system design flow

At the algorithm development stage, it is easier to use high-level language tools (such as MATLAB® or C/C++) for the algorithmic-level simulations. A DSP algorithm can be simulated using a general-purpose computer to test and analyze its performance. A block diagram of software development using a general-purpose computer is illustrated in Figure 1.10. The testing signals may be internally generated by signal generators, digitized from the experimental setup or real environment based on the given application, or received from other computers via the networks. The simulation program uses the signal samples stored in data file(s) as input(s) to produce output signal(s) that will be saved as data file(s) for further analysis.

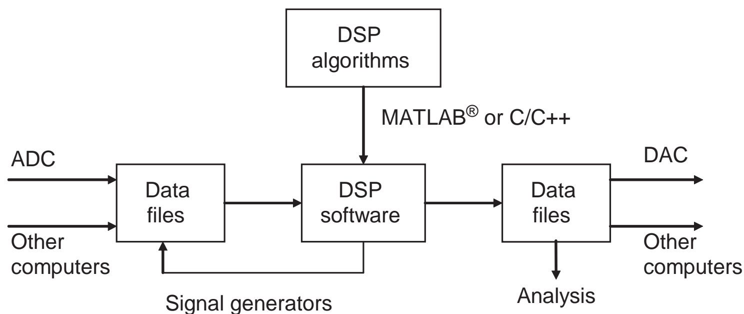


Figure 1.10 DSP software developments using a general-purpose computer

The advantages of developing DSP algorithms using a general-purpose computer are:

1. Using high-level languages such as MATLAB®, C/C++, or other DSP software packages on computers can significantly save algorithm development time. In addition, the prototype C programs used for algorithm evaluation can be ported to different DSP hardware platforms.
2. It is easier to debug and modify high-level language programs on computers using integrated software development tools.
3. I/O operations based on disk files are easy to implement and the behaviors of the system are easy to analyze.
4. Floating-point data formats and arithmetic can be used for computer simulations, thus ease of development.
5. Bit-true simulations of the developed algorithms can be performed using MATLAB® or C/C++ for fixed-point DSP implementation.

1.4.2 Selection of DSP Hardware

As discussed earlier, digital signal processors are used in a wide range of applications from high-performance radar systems to low-cost consumer electronics. DSP system designers require a full understanding of the application requirements in order to select the right DSP hardware for the given application. The objective is to choose the processor that meets the project's requirements with the most cost-effective solution [18]. Some decisions can be made at an early stage based on arithmetic format, performance, price, power consumption, ease of development and integration, and so on. For real-time DSP applications, the efficiency of data flow into and out of the processor is also critical.

Example 1.5

There are a number of ways to measure a processor's execution speed, as follows:

1. MIPS – Millions of instructions per second.
2. MOPS – Millions of operations per second.
3. MFLOPS – Millions of floating-point operations per second.
4. MHz – clock rate in mega hertz.
5. MMACS – Millions of multiply–accumulate operations.

In addition, there are other metrics to be considered such as milliwatts (mW) for measuring power consumption, MIPS per mW, or MIPS per dollar. These numbers provide a simple indication of performance, power, and price for the given application.

As discussed earlier, hardware cost and product manufacture integration are important factors for high-volume applications. For portable, battery-powered products, power consumption is more critical. For low- to medium-volume applications, there will be trade-offs among development time, cost of development tools, and the cost of the hardware itself. The likelihood of having higher performance processors with upwards-compatible software is also an important factor. For high-performance, low-volume applications such as communication

infrastructures and wireless base stations, the performance, ease of development, and multiprocessor configurations are paramount.

Example 1.6

A number of DSP applications along with the relative importance for performance, price, and power consumption are listed in Table 1.2. This table shows, for handheld devices, that the primary concern is power efficiency; however, the main criterion for the communication infrastructures is performance.

When processing speed is at a premium, the only valid comparison between processors is on an algorithm implementation basis. Optimum code must be written for all candidates and then the execution time must be compared. Other important factors are memory usage and on-chip peripheral devices, such as on-chip converters and I/O interfaces. In addition, a full set of development tools and support listed as follows are important for digital signal processor selection:

1. Software development tools such as C compilers, assemblers, linkers, debuggers, and simulators.
2. Commercially available DSP boards for software development and testing before the target DSP hardware is available.
3. Hardware testing tools such as in-circuit emulators and logic analyzers.
4. Development assistance such as application notes, DSP function libraries, application libraries, data books, low-cost prototyping, and so on.

1.4.3 Software Development

There are four common measures of good DSP software: reliability, maintainability, extensibility, and efficiency. A reliable program is one that seldom (or never) fails. Since most programs will occasionally fail, a maintainable program is one that is easy to correct. A truly maintainable program is one that can be fixed by someone other than the original programmers. An extensible program is one that can be easily modified when the requirements change. A good DSP program often contains many small functions with only one purpose, which can be easily reused by other programs for different purposes.

Table 1.2 Some DSP applications with the relative importance rating (adapted from [19])

Application	Performance	Price	Power consumption
Audio receiver	1	2	3
DSP hearing aid	2	3	1
MP3 player	3	1	2
Portable video recorder	2	1	3
Desktop computer	1	2	3
Notebook computer	3	2	1
Cell phone handset	3	1	2
Cellular base station	1	2	3

Rating: 1 to 3 with 1 being the most important

As shown in Figure 1.9, hardware and software design can be conducted at the same time for a given DSP application. Since there are many interdependent factors between hardware and software, an ideal DSP designer will be a true "system" engineer, capable of understanding issues with both hardware and software. The cost of hardware has gone down dramatically in recent years, thus the major cost of DSP solutions now resides in software development.

The software life cycle involves the completion of the software project: namely, project definition, detailed specifications, coding and modular testing, system integration and testing, and product software maintenance. Software maintenance is a significant part of the cost for DSP systems. Maintenance includes enhancing the software functions, fixing errors identified as the software is used, and modifying the software to work with new hardware and software. It is important to use meaningful variable names in source code, and to document programs thoroughly with titles and comment statements because this greatly simplifies the task of software maintenance. Programming tricks should be avoided at all costs, as they will not be reliable and will be difficult for someone else to understand even with lots of comments.

As discussed earlier, good programming techniques play an essential role in successful DSP applications. A structured and well-documented approach to programming should be initiated from the beginning. It is important to develop overall specifications for signal processing tasks prior to writing any program. The specifications include the basic algorithm and task description, memory requirements, constraints on the program size, execution time, and so on. The thoroughly reviewed specifications can catch mistakes even before the code has been written and prevent potential code changes at system integration stage. A flow diagram would be a very helpful design tool to adopt at this stage.

Writing and testing DSP code is a highly interactive process. With the use of integrated software development tools that include simulators or evaluation boards, code may be tested regularly as it is written. Writing code in modules or sections can help this process, as each module can be tested individually, thus increasing the chance of the entire system working at system integration stage.

There are two commonly used programming languages in developing DSP software: assembly language and C. Assembly language is similar to the machine code actually used by the processor. Programming in assembly language gives engineers full control of processor functions and resources, thus resulting in the most efficient program for mapping the algorithm by hand. However, this is a very time-consuming and laborious task, especially for today's highly parallel processor architectures and complicated DSP algorithms. C language, on the other hand, is easier for software development, upgrading, and maintenance. However, the machine code generated by the C compiler is often inefficient in both processing speed and memory usage.

Often the best solution is to use a mixture of C and assembly code. The overall program is written using C, but the runtime critical inner loops and modules are replaced by assembly code. In a mixed programming environment, an assembly routine may be called as a function or intrinsics, or in-line coded into the C program. A library of hand-optimized functions may be built up and brought into the code when required.

1.4.4 Software Development Tools

Most DSP operations can be categorized as being either signal analysis or filtering. Signal analysis deals with the measurement of signal properties. MATLAB® is a powerful tool for signal analysis and visualization, which are critical components in understanding and

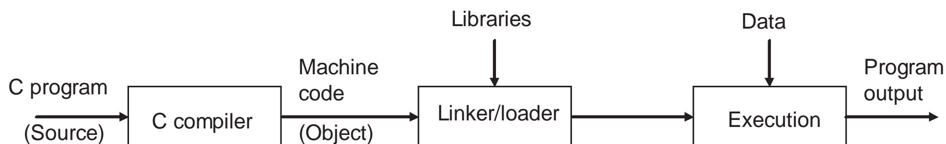


Figure 1.11 Program compilation, linking, and execution flow

developing DSP systems. C is an efficient tool for performing signal processing and is portable over different DSP platforms.

MATLAB® is an interactive, technical computing environment for scientific and engineering numerical analysis, computation, and visualization. Its strength lies in the fact that complex numerical problems can be solved easily in a fraction of the time required by programming languages such as C/C++. By using its relatively simple programming capability, MATLAB® can be easily extended to create new functions, and is further enhanced by numerous toolboxes. In addition, MATLAB® provides many graphical user interface (GUI) tools such as the Signal Processing Tool (SPTool) and Filter Design and Analysis Tool (FDATool).

The purpose of programming languages is to solve problems involving the manipulation of information. The purpose of DSP programs is to manipulate signals to solve specific signal processing problems. High-level languages such as C/C++ are usually portable, so they can be recompiled and run on many different computer platforms. Although C/C++ is categorized as a high-level language, it can also be used for low-level device drivers. In addition, C compilers are available for most modern digital signal processors. Thus, C programming is the most commonly used high-level language for DSP applications.

C has become the language of choice for many DSP software development engineers, not only because it has powerful commands and data structures, but also because it can easily be ported to different digital signal processors and platforms. C compilers are available for a wide range of computers and processors, thus making the C program the most portable software for DSP applications. The processes of compilation, linking/loading, and execution are outlined in Figure 1.11. The C programming environment includes the GUI debugger, which is useful in identifying errors in source programs. The debugger can display values stored in variables at different points in the program, and step through the program line by line.

1.5 Experiments and Program Examples

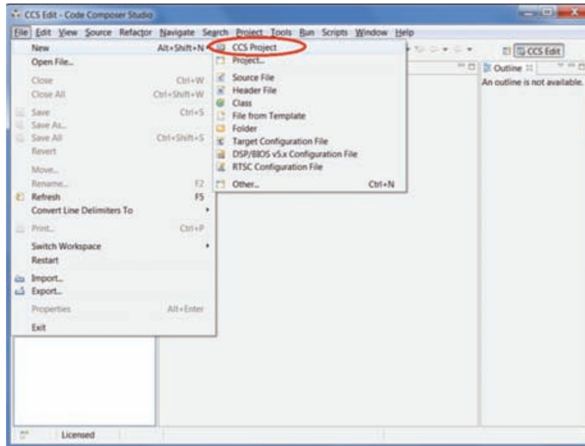
The Code Composer Studio (CCS) is an integrated development environment for DSP applications. CCS has several built-in tools for software development including project build environment, source code editor, C/C++ compiler, debugger, profiler, simulator, and real-time operating system. CCS allows users to create, edit, build, debug, and analyze software programs. It also provides a project manager to handle multiple programming projects for building large applications. For software debugging, CCS supports breakpoints, watch windows for monitoring variables, memory, and registers, graphical display and analysis, program execution profiling, and display assembly and C instructions for single-step instruction traces.

This section uses experiments to introduce several key CCS features including basic editing, memory configuration, and compiler and linker settings for building programs. We will demonstrate DSP software development and debugging processes using CCS with the low-cost TMS320C5505 eZdsp USB stick. Finally, we will present real-time audio experiments using eZdsp, which will be used as prototypes for building real-time experiments

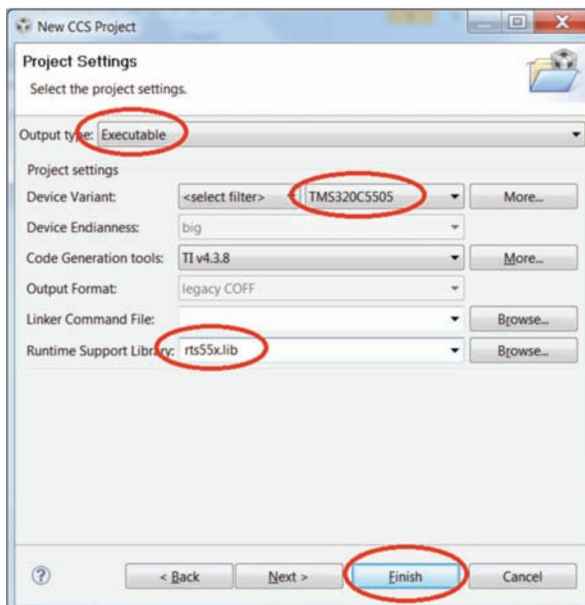
throughout the book. To conduct these real-time experiments, we have to connect the eZdsp to a USB port of a computer with CCS installed.

1.5.1 Get Started with CCS and eZdsp

In this book, we use the C5505 eZdsp with CCS version 5.x for all experiments. To learn some basic features of CCS, perform the following steps to complete the experiment Exp1.1.



(a) Create the CCS project, File→New→CCS Project.



(b) Select executable and `rts55x.lib` library.

Figure 1.12 Create a CCS project

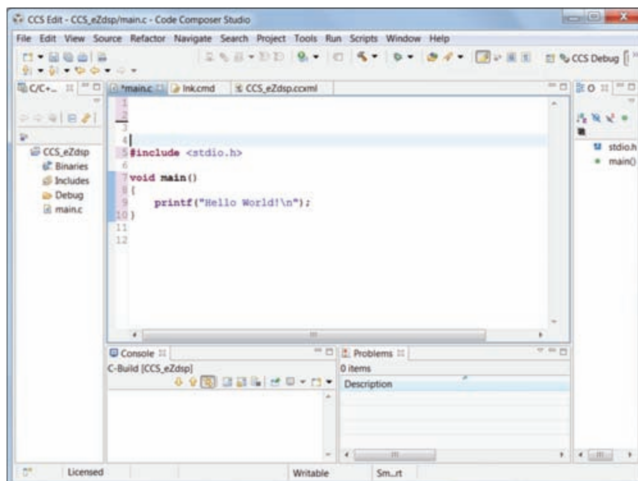


Figure 1.13 C program for the experiment Exp1.1

Step 1, start CCS from the host computer and create the C5505 CCS project as shown in Figure 1.12. In this experiment, we use `CCS_eZdsp` as the project name, select executable output type, and use `rts55x` runtime support library.

Step 2, create a C program under the CCS project and name the C file as `main.c` via **File→New→Source File**. Then, use the CCS text editor to write the C program to display "Hello World" as shown in Figure 1.13.

Step 3, create the target configuration file for the C5505 eZdsp. Start from **File→New→Target Configuration File**, select `XDS100v2 USB Emulator` and `USBSTK5505` as the target configuration and save the changes, see Figure 1.14.

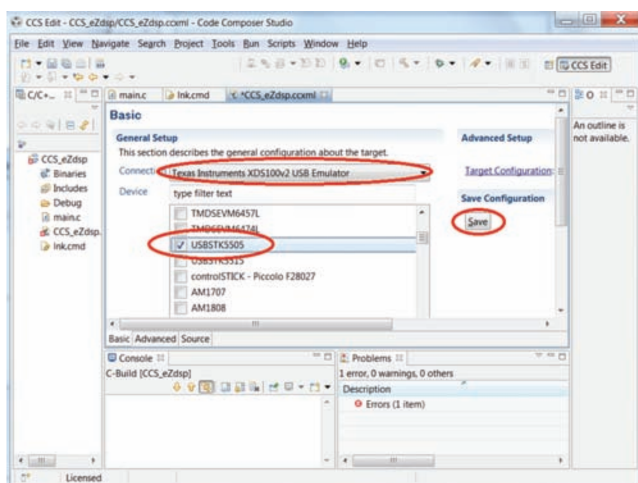


Figure 1.14 Create the target configuration

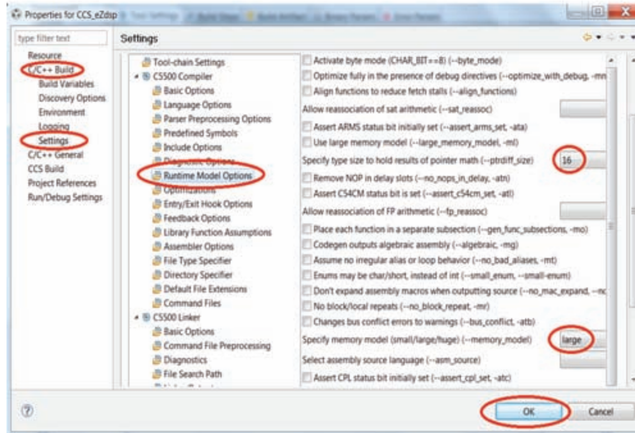


Figure 1.15 Setting the CCS project runtime environment

Step 4, set up the CCS environment. Open the property of the C5505 project we have created by right-clicking on the project, **CCS_eZdsp→Properties**, under the **Resource** window, select and open the **C/C++ Build→Settings→Runtime Model Options**, then set up for large memory model and 16-bit pointer math as shown in Figure 1.15.

Step 5, add the C5505 linker command file. Use the text editor to create the linker command file `c5505.cmd` as listed in Table 1.3, which will be discussed later. This file is available in the companion software package.

Step 6, connect the CCS to the target device. Go to **View→Target Configurations** to open the **Target Configuration** window, select the experiment target, right-click on it to launch the

Table 1.3 Linker command file, `c5505.cmd`

```
-stack    0x2000        /* Primary stack size   */
-sysstack 0x1000        /* Secondary stack size */
-heap     0x2000        /* Heap area size       */
-c                      /* Use C linking conventions: auto-init vars at
                           runtime */
-u _Reset               /* Force load of reset interrupt handler */

MEMORY
{
   MMR     (RW) : origin = 0000000h length = 0000c0h /* MMRs */
   DARAM   (RW) : origin = 00000c0h length = 00ff40h /* On-chip DARAM */
   SARAM   (RW) : origin = 0030000h length = 01e000h /* On-chip SARAM */

   SAROM_0 (RX) : origin = 0fe0000h length = 008000h /* On-chip ROM 0 */
   SAROM_1 (RX) : origin = 0fe8000h length = 008000h /* On-chip ROM 1 */
   SAROM_2 (RX) : origin = 0ff0000h length = 008000h /* On-chip ROM 2 */
   SAROM_3 (RX) : origin = 0ff8000h length = 008000h /* On-chip ROM 3 */
}
```

Table 1.3 (*Continued*)

```
SECTIONS
{
   vectors (NOLOAD)
   .bss           : > DARAM     /* Fill = 0 */
   vector         : > DARAM       ALIGN = 256
   .stack         : > DARAM
   .sysstack      : > DARAM
   .sysmem        : > DARAM
   .text          : > SARAM
   .data          : > DARAM
   .cinit         : > DARAM
   .const         : > DARAM
   .cio           : > DARAM
   .usect         : > DARAM
   .switch        : > DARAM
}
```

target configuration, then right-click on the USB Emulator 0/C55xx and select **Connect Target**, see Figure 1.16.

Step 7, build, load, and run the experiment. From **Project→Build All**, after the build is completed without error, load the executable program from **Run→Load→Load Program**, see Figure 1.17. When CCS prompts for the program to be loaded, navigate to the project folder and load the C5505 executable file (e.g., `CCS_eZdsp.out`) from the Debug folder.

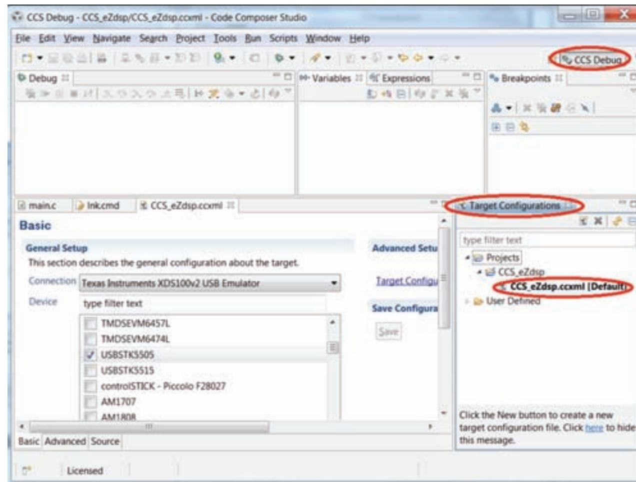
As shown in Table 1.3, the linker command file, `c5505.cmd`, defines the C55xx system memory for the target device and specifies the locations of program memory, data memory, and I/O memory. The linker command file also describes the starting locations of memory blocks and the length of each block. More information on the hardware specific linker command file can be found in the C5505 data sheet [20]. Table 1.4 lists the files used for the experiment Exp1.1.

Procedures of the experiment are listed as follows:

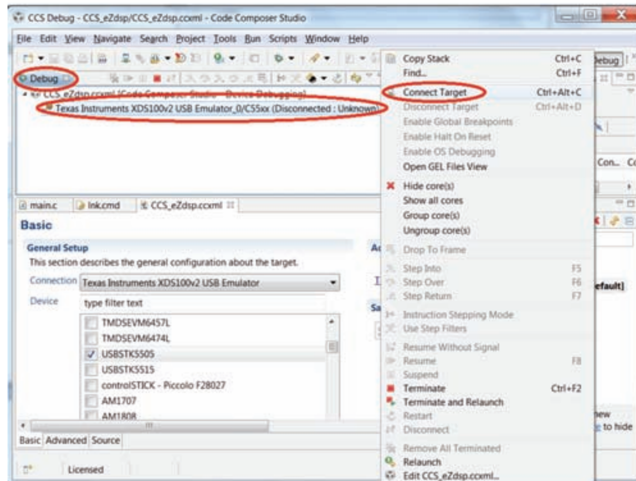
1. Follow the experiment steps presented in this section to create a CCS workspace for Exp1.1.
2. Remove the linker command file `c5505.com` from the project. Rebuild the experiment. There will be warning messages displayed. CCS generates these warnings because it uses default settings to map the program and data to the processor's memory spaces when the linker command file is missing.
3. Load `CCS_eZdsp.out`, and use **Step Over** (F6) through the program. Then, use CCS **Reload Program** to load the program again. Where is the program counter (cursor) location?

Table 1.4 File listing for the experiment Exp1.1

Files	Description
`main.c`	C source file for experiment
`c5505.cmd`	Linker command file



(a) Open target configuration window.



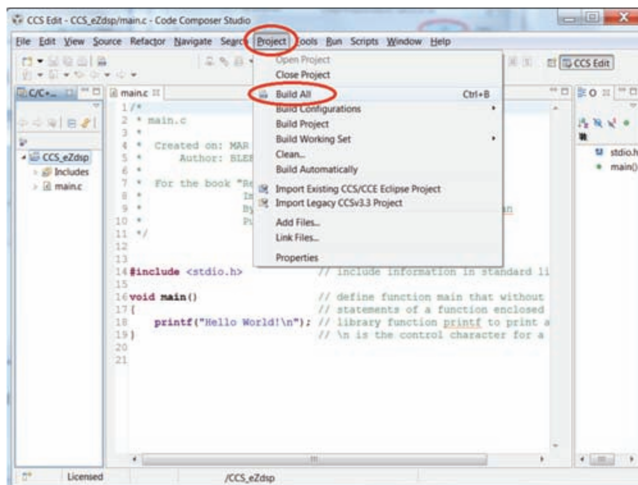
(b) Connect the CCS to target device.

Figure 1.16 Connect CCS with the target device, C5505 eZdsp

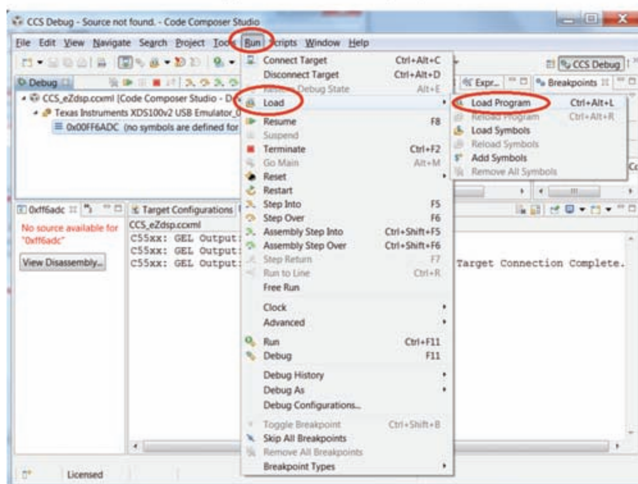
4. Use **Resume** (F8) instead of **Step Over** (F6) to run the program again. What will be showing on the console display window? Observe the differences from step 3.
5. After running the program, use **Restart** and **Resume** (F8) to run the program again. What will be showing on the console display window?

1.5.2 C File I/O Functions

We can use C file I/O functions to access the ASCII formatted or binary formatted data files that contain input signals to simulate DSP applications. The binary data file is more efficient for storage and access, while the ASCII data format is easy for the user to read and check. In



(a) Build the CCS project.



(b) Load the executable file to eZdsp.

Figure 1.17 Build, load, and run the experiment using CCS

practical applications, digitized data files are often stored in binary format to reduce memory requirements. In this section, we will introduce the C file I/O functions provided by CCS libraries.

CCS supports standard C library functions such as `fopen`, `fclose`, `fread`, `fwrite` for file I/O operations. These C file I/O functions are portable to other development environments.

The C language supports different data types. To improve program portability, we use the unique type definition header file, `tistdtypes.h`, to specify the data types to avoid any ambiguity.

Table 1.5 lists the C program that uses `fopen`, `fclose`, `fread`, and `fwrite` functions. The input data is a linear PCM (Pulse Code Modulation) audio signal stored in a binary file. Since

Table 1.5 C program using file I/O functions, `fielIO.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "tistdtypes.h"
Uint8 waveHeader[44]={                         /* 44 bytes for WAV file
                                                  header */
      0x52,  0x49,  0x46,  0x46,  0x00,  0x00,  0x00,  0x00,
      0x57,  0x41,  0x56,  0x45,  0x66,  0x6D,  0x74,  0x20,
      0x10,  0x00,  0x00,  0x00,  0x01,  0x00,  0x01,  0x00,
      0x40,  0x1F,  0x00,  0x00,  0x80,  0x3E,  0x00,  0x00,
      0x02,  0x00,  0x10,  0x00,  0x64,  0x61,  0x74,  0x61,
      0x00,  0x00,  0x00,  0x00};
#define SIZE 1024
Uint8 ch[SIZE];                                /* Declare a char[1024]
                                                  array for experiment */
void main()
{
   FILE *fp1,*fp2;                             /* File pointers */
   Uint32 i;                                   /* Unsigned long integer
                                                  used as a counter */
   printf("Exp. 1.2 --- file IO\n");
   fp1 = fopen("..\\data\\C55DSPUSBStickAudioTest.pcm", "rb");
                                               /* Open input file */
   fp2 = fopen("..\\data\\C55DSPUSBStickAudioTest.wav", "wb");
                                               /* Open output file */
   if (fp1 == NULL)                            /* Check if the input file
                                                  exists */
   {
      printf("Failed to open input file 'C55DSPUSBStickAudioTest.
pcm'\n");
      exit(0);
   }
   fseek(fp2, 44, 0);                          /* Advance output file
                                                  point 44 bytes */
   i=0;
   while (fread(ch, sizeof(Uint8), SIZE, fp1) == SIZE)
                                               /* Read in SIZE of input
                                                  data bytes */
   {
      fwrite(ch, sizeof(Uint8), SIZE, fp2); /* Write SIZE of data bytes
                                                  to output file */
      i += SIZE;
      printf("%ld bytes processed\n", i);   /* Show the number of data
                                                  is processed */
   }
   waveHeader[40] = (Uint8)(i&0xff);          /* Update the size
                                                  parameter into WAV
                                                  header */
```

Table 1.5 (*Continued*)

```
    waveHeader[41] = (Uint8)(i>>8)&0xff;
    waveHeader[42] = (Uint8)(i>>16)&0xff;
    waveHeader[43] = (Uint8)(i>>24)&0xff;
    waveHeader[4] = waveHeader[40];
    waveHeader[5] = waveHeader[41];
    waveHeader[6] = waveHeader[42];
    waveHeader[7] = waveHeader[43];

    rewind(fp2);                                /* Adjust output file
                                                   point to beginning */
    fwrite(waveHeader, sizeof(Uint8), 44, fp2); /* Write 44 bytes of WAV
                                                   header to output
                                                   file */

    fclose(fp1);                                /* Close input file */
    fclose(fp2);                                /* Close output file */

    printf("\nExp. completed\n");
}
```

the C55xx CCS file I/O libraries support only byte formatted binary data (`char`, 8-bit), the 16-bit PCM data file can be read using `sizeof(char)`, and the output wave (WAV) data file can be written by CCS in byte format [21–23]. For 16-bit short-integer data types, each data read or data write requires two memory accesses. As shown in Table 1.5, the program reads and writes 16-bit binary data in byte units. To run this program on a computer, the data access can be changed to its native data type `sizeof(short)`. The output file of this experiment is a WAV file that can be played by many audio players. Note that the WAV file format supports several different file types and sampling rates. The files used for the experiment are listed in Table 1.6.

Procedures of the experiment are listed as follows:

1. Create CCS workspace for the experiment Exp1.2.
2. Create a C5505 project using `fileIO` as the project name.
3. Copy `fileIOTest.c`, `tistdtypes.h`, and `c5505.cmd` from the companion software package to the experiment folder.
4. Create a data folder under the experiment folder and place the input file `C55DSPUSB-StickAudioTest.pcm` into the data folder.

Table 1.6 File listing for the experiment Exp1.2

Files	Description
`fileIOTest.c`	Program file for testing C file I/O
`tistdtypes.h`	Data type definition header file
`c5505.cmd`	Linker command file
`C55DSPUSBStickAudioTest.pcm`	Audio data file for experiment

5. Set up the CCS project build and debug environment using the 16-bit data format and large runtime support library `rts55x.lib`.
6. Set up the target configuration file `fileIO.ccxml` for using the eZdsp.
7. Build and load the experiment executable file. Run the experiment to generate the output audio file, `C55DSPUSBStickAudioTest.wav`, saved in the data folder. Listen to the audio file using an audio player.
8. Modify the experiment such that it can achieve the following tasks:
   (a) Read the input data file `C55DSPUSBStickAudioTest.pcm` and write an output file in ASCII integer format in `C55DSPUSBStickAudioTest.xls` (or another file format instead of `.xls`). (Hint: replace the `fwrite` function with `fprintf`.)
   (b) Use Microsoft Excel (or other software such as MATLAB®) to open the file `C55DSPUSBStickAudioTest.xls`, select the data column, and plot the waveform of the audio.
9. Modify the experiment to read "`C55DSPUSBStickAudioTest.xls`" created in the previous step as the input file and write it out in a WAV file. Listen to the WAV file to verify it is correct.

1.5.3 User Interface for eZdsp

An interactive user interface is very useful for developing real-time DSP applications. It provides the flexibility to change runtime parameters without the need to stop execution, modify, recompile, and rerun the program. This feature becomes more important for large-scale projects that consist of many C programs and prebuilt libraries. In this experiment, we use the `scanf` function to get interactive input parameters through the CCS console window. We also introduce some commonly used CCS debugging methods including software breakpoints, viewing processor's memory and program variables, and graphical plots.

Table 1.7 lists the C program that uses `fscan` function to read user parameters via the CCS console window. This program reads the parameters and verifies their values. The program will replace any invalid value with the default value. This experiment has three user-defined parameters: gain `g`, sampling frequency `sf`, and playtime duration `p`. The files used for the experiment are listed in Table 1.8.

Table 1.7 C program with interactive user interface, `UITest.c`

```
#include <stdio.h>
#include "tistdtypes.h"

#define SIZE 48
Int16 dataTable[SIZE];

void main()
{
    /* Pre-generated sine wave data, 16-bit signed samples */
    Int16 table[SIZE] = {
      0x0000, 0x10b4, 0x2120, 0x30fb, 0x3fff, 0x4dea, 0x5a81, 0x658b,
      0x6ed8, 0x763f, 0x7ba1, 0x7ee5, 0x7ffd, 0x7ee5, 0x7ba1, 0x76ef,
      0x6ed8, 0x658b, 0x5a81, 0x4dea, 0x3fff, 0x30fb, 0x2120, 0x10b4,
```

Table 1.7 (*Continued*)

```
    0x0000, 0xef4c, 0xdee0, 0xcf06, 0xc002, 0xb216, 0xa57f, 0x9a75,
    0x9128, 0x89c1, 0x845f, 0x811b, 0x8002, 0x811b, 0x845f, 0x89c1,
    0x9128, 0x9a76, 0xa57f, 0xb216, 0xc002, 0xcf06, 0xdee0, 0xef4c
  };

  Int16 g,p,i,j,k,n,m;
  Uint32 sf;

  printf("Exp. 1.3 --- UI\n");

  printf("Enter an integer number for gain between (-6 and 29)\n");
  scanf ("%d", &g);

  printf("Enter the sampling frequency, select one: 8000, 12000,
  16000, 24000 or 48000\n");
  scanf ("%ld", &sf);

  printf("Enter the playing time duration (5 to 60)\n");
  scanf ("%i", &p);

  if ((g < -6)||(g > 29))
  {
     printf("You have entered an invalid gain\n");
     printf("Use default gain = 0dB\n");
     g = 0;
  }
  else
  {
     printf("Gain is set to %ddB\n", g);
  }
  if ( (sf == 8000)||(sf == 12000)||(sf == 16000)||(sf == 24000)||(sf ==
  48000))
  {
     printf("Sampling frequency is set to %ldHz\n", sf);
  }
  else
  {
     printf("You have entered an invalid sampling frequency\n");
     printf("Use default sampling frequency = 48000 Hz\n");
     sf = 48000;
  }
  if ((p < 5)||(p > 60))
  {
     printf("You have entered an invalid playing time\n");
     printf("Use default duration = 10s\n");
     p = 10;
  }
```

(*continued*)

Table 1.7 (*Continued*)

```
    else
    {
       printf("Playing time is set to %ds\n", p);

    }
    for (i=0; i<SIZE; i++)
       dataTable[i] = 0;

    switch (sf)
    {
       case 8000:
          m = 6;
          break;
       case 12000:
          m = 4;
          break;
       case 16000:
          m = 3;
          break;
       case 24000:
          m = 2;
          break;
       case 48000:
       default:
          m = 1;
          break;
    }

    for (n=k=0, i=0; i<m; i++)      // Fill in the data table
    {
       for (j=k; j<SIZE; j+=m)
       {
          dataTable[n++] = table[j];
       }
       k++;
    }

    printf("\nExp. completed\n");
}
```

Table 1.8 File listing for the experiment Exp1.3

Files	Description
`UITest.c`	Program file for testing user interface
`tistdtypes.h`	Data type definition header file
`c5505.cmd`	Linker command file

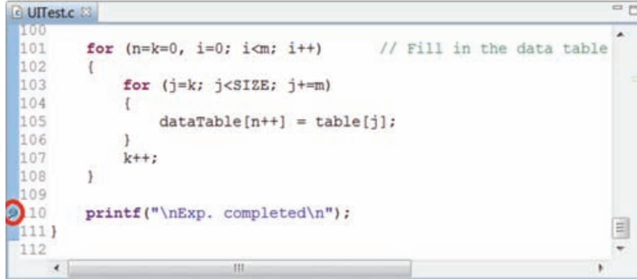


Figure 1.18 Setting a software breakpoint

CCS has many built-in tools including software breakpoints, watch windows, and graphic plots for debugging, testing, and evaluating programs. When the program reaches a software breakpoint, it will halt program execution at that location. CCS will preserve the processor's register and system memory values at that instant for users to validate the results. The software breakpoint can be set by double-clicking on the left sidebar of an instruction. Figure 1.18 shows the breakpoint is set on line number 110. Once the program runs to the breakpoint, it stops, then the user can step over the program statements, or step into the function if this line contains another function. The breakpoint can be removed by double-clicking on the breakpoint itself.

Once the program hits the breakpoint, we can use watch windows to examine registers and data variables. For example, we can use the CCS viewing feature to display data variables such as `g` and `sf` from the CCS menu **View→Variables**, see Figure 1.19.

We can also view data values stored in memory. To view memory, we open the memory watch window from **View→Memory Browser**. Figure 1.20 shows the CCS memory watch window that contains the data values stored in `dataTable[SIZE]` at data memory address 0x2E9D.

We can also use the CCS graphical tools to plot the data for visual examination. For this experiment, we activate the plot tools from **Tool→Graph→Single Time**. Open the graph properties setting dialog window, set **Acquisition Buffer** to 48 (table size), select **Data Type** as 16-bit signed integer (based on the data type), set the data **Start Address** to `dataTable` (data memory address), and finally set **Display Data Size** to 48. Figure 1.21 displays the graph of the sinusoidal data stored in the 16-bit integer array `dataTable[SIZE]`.

Variables

Name	Type	Value
g	short	0
i	short	4
j	short	51
k	short	4
m	short	4
n	short	48
p	short	5
sf	unsigned long	12000

Figure 1.19 Variable watch window

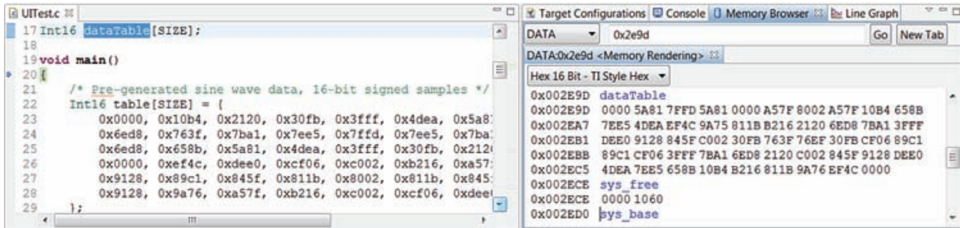


Figure 1.20 Memory watch window

Procedures of the experiment are as follows:

1. Create the CCS project, set up the project build and debug environment, and use `UI` as the project name.
2. Create the C source file `UITest.c` as listed in Table 1.7, which is available in the companion software package. Add linker command file `c5505.cmd` and header file `tistdtypes.h`.
3. Connect the eZdsp to the computer, and set up the proper target configuration file for the eZdsp. Build and load the executable program for experiment.
4. Perform single-step operation to check the program.
5. Set some software breakpoints in `UITest.c`, and step through the program to observe the variable values of `g`, `sf`, and `p`.
6. Rerun the experiment and use the CCS graphical tool to plot the data stored in the array `dataTable[]` for different sampling frequencies at 8000, 12 000, 16 000, 32 000, and 48 000 Hz. Show these plots and compare their differences.
7. Set up the CCS variable watch window and examine what other data types can be supported by the watch window. Change the data type and observe how the watch window displays different data types.
8. How does one find a variable memory's address for setting up the watch windows? Set up both data and variable watch windows, single step through the program, and watch how the variables are updated and displayed on the watch windows.
9. Data values in the memory can be modified via CCS by directly editing the memory locations. Try to change the variable values and rerun the program.
10. CCS can plot different kinds of graphs. Select the graph parameters to plot the array `data` in the same window with the following settings:

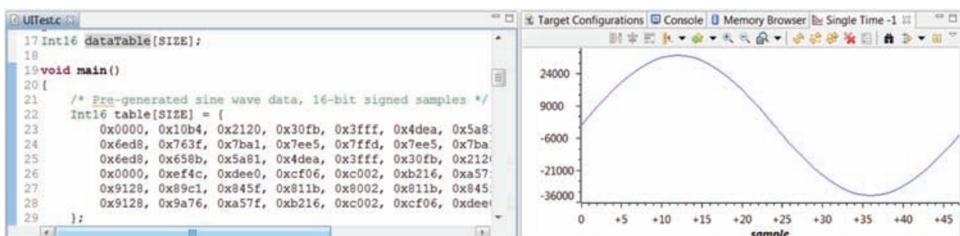


Figure 1.21 Use graphical tool to plot data samples

(a) Add x–y axis labels to the plot.
(b) Add a grid to the graph plot.
(c) Change the display from line to large square.

1.5.4 Audio Playback Using eZdsp

The C programs presented in previous experiments can be executed on the C55xx simulator which comes with the CCS or hardware devices such as the C5505 eZdsp. In this experiment, we use the eZdsp for real-time experiments. The C5505 eZdsp has a standard USB interface to connect with the host computer for program development, debugging, algorithm evaluation and analysis, and real-time demonstration.

CCS comes with many useful supporting functions and programs including device drivers specifically written for C55xx processors and the eZdsp. The latter comes with an installation CD which includes the `C55xx_csl` folder containing files for the C55xx chip support libraries and the `USBSTK_bsl` folder containing files for the eZdsp board support libraries. The experiments given in Appendix C provide detailed descriptions of the chip support libraries and board support libraries. In this experiment, we modify the user interface experiment presented in Exp1.3 to control audio playback using the C55xx board support library and chip support library. This experiment plays audible tones using the eZdsp, and allows the user to control the output volume, sampling frequency, and play time from the CCS console window. The user interface C program, `playToneTest.c`, asks the user to enter three parameters for the experiment: gain `(gain)`, sampling frequency (`sf`), and time duration (`playtime`). After receiving the user parameters, the program generates a tone table, and calls the function `tone` to play the tone in real time using the eZdsp.

After the user parameters are passed to the function `tone` listed in Table 1.9, it calls the function `USBSTK5505_init` to initialize the eZdsp and the function `AIC3204` to initialize the analog interface chip AIC3204. Once enabled, AIC3204 will operate at the user-specified

Table 1.9 C program for real-time tone playback, `tone.c`

```
#define AIC3204_I2C_ADDR 0x18
#include "usbstk5505.h"
#include "usbstk5505_gpio.h"
#include "usbstk5505_i2c.h"
#include <stdio.h>

extern void aic3204_tone_headphone();
extern void tone(Uint32, Int16, Int16, Uint16, Int16*);
extern void Init_AIC3204(Uint32 sf, Int16 gDAC, Uint16 gADC);

void tone(Uint32 sf, Int16 playtime, Int16 gDAC, Uint16 gADC, Int16
*sinetable)
{
    Int16 sec, msec;
    Int16 sample, len;
```

Table 1.9 (*Continued*)

```
/* Initialize BSL */
USBSTK5505_init();

/* Set A20_MODE for GPIO mode */
CSL_FINST(CSL_SYSCTRL_REGS->EBSR, SYS_EBSR_A20_MODE, MODE1);

/* Use GPIO to enable AIC3204 chip */
USBSTK5505_GPIO_init();
USBSTK5505_GPIO_setDirection(GPIO26, GPIO_OUT);
USBSTK5505_GPIO_setOutput( GPIO26, 1);    /*Take AIC3204 chip out
                                            of reset */

/* Initialize I2C */
USBSTK5505_I2C_init();

/* Initialized AIC3204 */
Init_AIC3204(sf, gDAC, gADC);

/* Initialize I2S */
USBSTK5505_I2S_init();

switch (sf)
{
   case 8000:
      len = 8;
      break;
   case 12000:
      len = 12;
      break;
   case 16000:
      len = 16;
      break;
   case 24000:
      len = 24;
      break;
   case 48000:
   default:
      len = 48;
      break;
}
/* Play tone */
for ( sec = 0; sec < playtime; sec++)
{
   for ( msec = 0; msec < 1000; msec++)
   {
      for ( sample = 0; sample < len; sample++)
      {
```

Table 1.9 (*Continued*)

```
                /* Write 16-bit left channel data */
                USBSTK5505_I2S_writeLeft( sinetable[sample]);

              /* Write 16-bit right channel data */
              USBSTK5505_I2S_writeRight(sinetable[sample]);
          }
      }
  }

    USBSTK5505_I2S_close();   // Disable I2S
    AIC3204_rset( 1, 1);      // Reset codec

   USBSTK5505_GPIO_setOutput( GPIO26, 0); // Disable AIC3204
}
```

sampling frequency to convert the digital signal to analog form and send out the stereo tone to the connected headphone or loudspeaker at the user-specified output volume. Table 1.10 lists the files used for the experiment.

The eZdsp uses the TLV320AIC3204 analog interface chip for the A/D and D/A conversions. This experiment uses the function `initAIC3204()` to set up AIC3204 registers for the sampling frequency and gain values entered by the user. The sampling frequency is calculated by

$$f_s = \frac{MCLK \times JD \times R}{P \times NDAC \times MDAC \times DOSR}, \tag{1.8}$$

where the master clock $MCLK = 12$ MHz and $JD = 7.168$ (J and D are two registers of the AIC3204, we set register J with integer 7 and register D with fraction number 168). If we preset the rest of the registers as $R = 1$, $NDAC = 2$, $MDAC = 7$, and $DOSR = 128$, we can change the value of P to set different sampling rates. For example, by changing P from 1, 2, 3, 4, and 6, we can configure AIC3204 to operate at different sampling frequencies at 48, 24, 16, 12, and 8 kHz, respectively. The TLV320AIC3204 data sheet [24] provides some examples for setting these parameters for different sampling rates.

Table 1.10 File listing for the experiment Exp1.4

Files	Description
`playToneTest.c`	Program file for testing eZdsp real-time tone generation
`tone.c`	C source file for tone generation
`initAIC3204.c`	C source file to initialize AIC3204
`tistdtypes.h`	Data type definition header file
`c5505.cmd`	Linker command file
`C55xx_csl.lib`	C55xx chip support library
`USBSTK_bsl.lib`	eZdsp board support library

The experiment procedures are listed as follows:

1. Create the experiment folder, `Exp1.4`, and copy the experiment software to the working directory including all the files in the folder `playTone`, and subfolders `src`, `lib`, `C55x_csl`, and `USBSTK_bsl`.
2. Start CCS and import the existing project workspace for the experiment to CCS.
3. Open the property of the `playTone` project and check **C/C++ Build Settings**. The **Include Options** should include the paths for `C55x_csl ..\C55xx_csl\inc` and `USBSTK_bsl ..\USBSTK_bsl\inc`, and the **Runtime Model Options** should be set for the 16-bit and large-memory model.
4. Connect the eZdsp to the host computer and connect a loudspeaker or headphone to the eZdsp.
5. Use the **Build All** command to rebuild the program, load the program, and run the experiment with user parameters: gain, sampling frequency, and tone playtime duration. Redo the experiment using different values of these three parameters and observe the differences.

1.5.5 Audio Loopback Using eZdsp

The previous audio tone playback experiment is written for sample-by-sample processing, which processes digital signals one sample at a time. On the other hand, data samples can be processed in groups using block-by-block processing. When a processor processes data using the sample-by-sample scheme, the processor may often be in an idle state waiting for the next available sample. That is, after processing one sample, the processor must wait for the next input sample. The idle time depends upon the sampling frequency and the time needed to process each sample. The advantage of sample-by-sample processing is its short processing delay. However, sample-by-sample processing is not very efficient in terms of data I/O overhead due to the waiting time for input samples. In contrast, block-by-block signal processing uses direct memory access (DMA) for data transfer that is performed in parallel with signal processing operations. Such a system can greatly reduce the I/O overhead to achieve the maximum processing efficiency. The trade-off between sample-by-sample processing and block-by-block processing is the minimized processing delay vs. the maximized processing efficiency. Many DSP systems use multithread operating systems so the applications are often programmed using block processing.

This experiment uses the eZdsp for real-time audio playback using block-by-block processing. The signal buffer size is `XMIT_BUFF_SIZE` and this can be adjusted to different sizes for different applications. The audio source can be a microphone or an audio player. The audio source is connected to the eZdsp's audio input STEREO IN jack (J2) using a stereo cable. The processed audio samples are played via a headphone or loudspeaker connected to the eZdsp's audio output HP OUT jack (J3). To use block-by-block processing, we use DMA to transfer input and output audio data samples. The sampling rate is set using the AIC3204. The C5505 DMA manages the data transfer between the C5505 and AIC3204.

Table 1.11 lists the main program for the experiment. It begins by setting the DMA and AIC3204, and then starts looping audio input to the output. The audio path is set for stereo with left and right audio channels. The program uses flags to identify which channel (left or right) of signal is coming from the AIC3204. Each channel uses two DMA data buffers of

Table 1.11 Real-time audio loopback program, `audioLoopTest.c`

```
#include <stdio.h>
#include "tistdtypes.h"
#include "i2s.h"
#include "dma.h"
#include "dmaBuff.h"

#define XMIT_BUFF_SIZE          256
Int16 XmitL1[XMIT_BUFF_SIZE]; /* DMA uses the same buffer names, do not
                                 rename */
Int16 XmitR1[XMIT_BUFF_SIZE];
Int16 XmitL2[XMIT_BUFF_SIZE];
Int16 XmitR2[XMIT_BUFF_SIZE];

Int16 RcvL1[XMIT_BUFF_SIZE];
Int16 RcvR1[XMIT_BUFF_SIZE];
Int16 RcvL2[XMIT_BUFF_SIZE];
Int16 RcvR2[XMIT_BUFF_SIZE];

Int16 dsp_process(Int16 *input, Int16 *output, Int16 size);

extern void AIC3204_init(Uint32, Int16, Int16);

#define IER0 *     (volatile unsigned *)0x0000

#define SF48KHz     48000
#define SF24KHz     24000
#define SF16KHz     16000
#define SF12KHz     12000
#define SF8KHz      8000

#define DAC_GAIN    3    // 3 dB range: -6 dB to 29 dB
#define ADC_GAIN    0    // 0 dB range: 0 dB to 46 dB

void main(void)
{
   Int16 status, i;

   // Clean output buffers before running the experiment
   for (i=0; i<XMIT_BUFF_SIZE; i++)
   {

       XmitL1[i]=XmitL2[i]=XmitR1[i]=XmitR2[i]=0;
       RcvL1[i]=RcvL2[i]=RcvR1[i]=RcvR2[i]=0;
   }

   setDMA_address();    // DMA address setup for each buffer
```

Table 1.11 (*Continued*)

```
    asm(" BCLR ST1_INTM");  // Disable all interrupts
    IER0 = 0x0110;          // Enable DMA interrupt

    set_i2s0_slave();       // Set I2S
    AIC3204_init(SF48KHz, DAC_GAIN, (Uint16)ADC_GAIN); // Set AIC3204
    enable_i2s0();

    enable_dma_int();       // Set up and enable DMA
    set_dma0_ch0_i2s0_Lout(XMIT_BUFF_SIZE);
    set_dma0_ch1_i2s0_Rout(XMIT_BUFF_SIZE);
    set_dma0_ch2_i2s0_Lin(XMIT_BUFF_SIZE);
    set_dma0_ch3_i2s0_Rin(XMIT_BUFF_SIZE);

    status = 1;
    while (status)          // Forever loop for the demo if status is set
    {
      if((leftChannel == 1)||(rightChannel == 1))
      {
        leftChannel = 0;
        rightChannel= 0;
        if ((CurrentRxL_DMAChannel == 2)||(CurrentRxR_DMAChannel == 2))
        {
          status = dsp_process(RcvL1, XmitL1, XMIT_BUFF_SIZE);
          status = dsp_process(RcvR1, XmitR1, XMIT_BUFF_SIZE);
        }
        else
        {
          status = dsp_process(RcvL2, XmitL2, XMIT_BUFF_SIZE);
          status = dsp_process(RcvR2, XmitR2, XMIT_BUFF_SIZE);
        }
      }
    }
}

// Simulated a DSP function
Int16 dsp_process(Int16 *input, Int16 *output, Int16 size)
{
    Int16 i;

    for(i=0; i<size; i++)
    {
      *(output + i) = *(input + i);
    }
    return 1;
}
```

Table 1.12 File listing for the experiment Exp1.5

Files	Description
`audioLoopTest.c`	Program file for testing real-time audio loopback
`vector.asm`	Assembly source file for interrupt vectors
`c5505.cmd`	Linker command file
`tistdtypes.h`	Data type definition header file
`dma.h`	C header file for DMA function and variable definition
`dmaBuff.h`	C header file for DMA buffer definition
`i2s.h`	C header file for I2S function and variable definition
`lpva200.inc`	C55xx assembly include file
`myC55xUtil.lib`	Experiment support library: DMA and I2S functions

equal length. This double-buffer method is often used for block signal processing. While the AIC3204 is filling one of the DMA data buffers, the C5505 process the data available in the other buffer. Once the process is complete, the DMA controller will switch the buffers for the next DMA transfer. The DMA channel identifier is used to manage which DMA buffer will be used. This ping-pong buffering scheme can avoid memory read and write collisions. The ping-pong buffer mechanism will introduce a certain buffering delay. The delay time equals the number of data samples in the buffer multiplied by the sampling period. If the data buffer contains 48 samples, and the sampling frequency is 48 000 Hz, the buffer introduces a time delay of 0.01 seconds.

In this experiment, we include the function `dsp_process` which simply copies the data from the input buffer to the output buffer. In subsequent experiments, we will replace this function by other DSP functions such as digital filters for real-time experiments. The assembly program `vector.asm` handles real-time interrupts for the C5505 system. The TMS320C5505 architecture and assembly language programming are introduced in Appendix C. The files used for the experiment are listed in Table 1.12.

The experiment procedures are listed as follows:

1. Copy the experiment software from the companion software package to the working directory and import the existing project.
2. Connect the eZdsp to the host computer. Connect a loudspeaker or headphone to the eZdsp HP OUT jack. Connect an audio source such as an MP3 player to the eZdsp STEREO IN jack.
3. Use CCS to build the project, load the executable program, and run the experiment.
4. Modify the experiment such that the left audio output channel will output the sum of input signals from both left and right channels, while the right audio output channel will be output the difference of input signals from the left and right channels. (Hint: modify the function `dsp_process`.)
5. Modify the audio loopback experiment such that it runs at 8000 Hz or other sampling frequencies.
6. Write a new function to generate a 1000 Hz tone. Modify the experiment such that it will loop the input audio on the left output channel and output the 1000 Hz tone on the right output channel.

Exercises

1.1. Given an analog audio signal that is bandlimited by 10 kHz:
(a) What is the minimum sampling frequency that allows a perfect reconstruction of the analog signal from its discrete-time samples?
(b) What will happen if a sampling frequency of 8 kHz is used?
(c) What will happen if the sampling frequency is 50 kHz?
(d) When the sampling rate is 50 kHz, and taking only every other sample (this is decimation by 2), what is the sampling frequency of the new signal? Is this causing aliasing?

1.2. Refer to Example 1.1. Assuming that we have to store 50 ms (milliseconds, 1 ms = 10^{-3} s) of digitized signals, how many samples are needed for (a) narrowband telecommunication systems with $f_s = 8$ kHz, (b) wideband telecommunication systems with $f_s = 16$ kHz, (c) audio CDs with $f_s = 44.1$ kHz, and (d) professional audio systems with $f_s = 48$ kHz?

1.3. Assume the dynamic range of the human ear is about 100 dB, and the highest frequency a human can hear is 20 kHz. For a high-end digital audio system designer, what size of converters and sampling rate are needed? When the design uses a 16-bit converter at 44.1 kHz sampling rate, how many bits are needed to store one minute of music?

1.4. A speech file (`timit_1.asc`) was digitized using a 16-bit ADC with one of the following sampling rates: 8, 12, 16, 24, or 32 kHz. Use MATLAB® to play it and find the correct sampling rate. This can be done by running the MATLAB® program `exercise_4.m` under the Exercises directory. This script plays the file at 8, 12, 16, 24, and 32 kHz. Press the Enter key to continue after the program is paused. What is the correct sampling rate?

1.5. Aliasing is caused by using an incorrect sampling rate that violates the sampling theorem. The MATLAB® script below generates a chirp signal, where `fl` and `fh` are the lower and upper frequencies of the chirp signal respectively, and the sampling frequency `fs` is 800 Hz. Edit and run the MATLAB® script, and listen and plot the signal. If we change `fs` to 200 Hz, what will happen and why?

```
fl = 0;              % Low frequency
fh = 200;            % High frequency
fs = 800;            % Sampling frequency
n = 0:1/fs:1;        % 1 second of data
phi = 2*pi*(fl*n + (fh-fl)*n.*n/2);
y = 0.5*sin(phi);
sound(y, fs);
plot(y)
```

References

1. Oppenheim, A.V. and Schafer, R.W. (1989) *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
2. Orfanidis, S.J. (1996) *Introduction to Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
3. Proakis, J.G. and Manolakis, D.G. (1996) *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd edn, Prentice Hall, Englewood Cliffs, NJ.
4. Bateman, A. and Yates, W. (1989) *Digital Signal Processing Design*, Computer Science Press, New York.
5. Kuo, S.M. and Morgan, D.R. (1996) *Active Noise Control Systems – Algorithms and DSP Implementations*, John Wiley & Sons, New York.
6. McClellan, J.H., Schafer, R.W., and Yoder, M.A. (1998) *DSP First: A Multimedia Approach*, 2nd edn, Prentice Hall, Englewood Cliffs, NJ.
7. ITU Recommendation (2012) G.729, Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP), June.
8. ITU Recommendation (2006) G.723.1, Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 kbit/s, May.
9. ITU Recommendation (1988) G.722, 7kHz Audio-Coding Within 64 kbit/s, November.
10. 3GPP TS (2002) 26.190, AMR Wideband Speech Codec: Transcoding Functions, 3GPP Technical Specification, March.
11. ISO/IEC (2006) 13818-7, MPEG-2 Generic Coding of Moving Pictures and Associated Audio Information, January.
12. ISO/IEC 11172-3 (1993) Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s – Part 3: Audio, August.
13. ITU Recommendation (1988) G.711, Pulse Code Modulation (PCM) of Voice Frequencies, November.
14. Zack, S. and Dhanani, S. (2004) DSP Co-processing in FPGA: Embedding High-performance, Low-cost DSP Functions, Xilinx White Paper, WP212.
15. Kuo, S.M. and Gan, W.S. (2005) *Digital Signal Processors – Architectures, Implementations, and Applications*, Prentice Hall, Upper Saddle River, NJ.
16. Lapsley, P., Bier, J., Shoham, A., and Lee, E.A. (1997) *DSP Processor Fundamentals: Architectures and Features*, IEEE Press, Piscataway, NJ.
17. Berkeley Design Technology, Inc. (2000) The Evolution of DSP Processor, BDTi White Paper.
18. Berkeley Design Technology, Inc. (2000) Choosing a DSP Processor, White Paper.
19. Frantz, G. and Adams, L. (2004) The three Ps of value in selecting DSPs. *Embedded System Programming*, October. Available at: http://staging.embedded.com/design/configurable-systems/4006435/The-three-Ps-of-value-in-selecting-DSPs (accessed May 9, 2013).
20. Texas Instruments, Inc. (2012) TMS320C5505 Fixed-Point Digital Signal Processor Data Sheet, SPRS660E, January.
21. IBM and Microsoft (1991) Multimedia Programming Interface and Data Specification 1.0, August.
22. Microsoft (1994) New Multimedia Data Types and Data Techniques, Rev. 1.3, August.
23. Microsoft (2002) Multiple Channel Audio Data and WAVE Files, November.
24. Texas Instruments, Inc. (2008) TLV320AIC3204 Data Sheet, SLOS602A, October.

2

DSP Fundamentals and Implementation Considerations

This chapter introduces fundamental DSP principles and practical implementation considerations for the digital filters and algorithms [1–4]. DSP implementations, especially using fixed-point processors, require special consideration due to quantization and arithmetic errors.

2.1 Digital Signals and Systems

This section briefly introduces some basic terminologies of digital signals and systems that will be used in the book.

2.1.1 Elementary Digital Signals

A digital signal is a sequence of numbers $x(n)$, $-\infty < n < \infty$, where integer n is the time index. Signals can be classified as deterministic or random. Deterministic signals such as sinusoidal signals can be expressed mathematically. Random signals such as speech and noise cannot be described exactly by equations. Some basic deterministic signals will be introduced in this section along with the frequency concepts, while random signals will be discussed in Section 2.3.

The unit-impulse signal is defined as

$$\delta(n) = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0, \end{cases} \tag{2.1}$$

where $\delta(n)$ is also called the Kronecker delta function. This unit-impulse signal is very useful for measuring, modeling, and analyzing the characteristics of DSP systems.

The unit-step signal is defined as

$$u(n) = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0. \end{cases} \tag{2.2}$$

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

This function is very convenient for describing causal signals, which are the most commonly encountered signals in real-time DSP systems. For example, $x(n)u(n)$ clearly defines a causal signal $x(n)$ such that $x(n) = 0$ for $n < 0$.

Sinusoidal signals (also called sinusoids) can be expressed in simple mathematical formulas. For example, an analog sine wave can be expressed as

$$x(t) = A\sin(\Omega t + \phi) = A\sin(2\pi f t + \phi), \tag{2.3}$$

where A is the amplitude of the sine wave, f is the frequency in cycles per second (hertz or Hz),

$$\Omega = 2\pi f \tag{2.4}$$

is the frequency in radians per second (rad/s), and ϕ is the initial phase in radians.

Sampling the analog sinusoidal signal defined in (2.3) results in a digital sine wave expressed as

$$x(n) = A\sin(\Omega nT + \phi) = A\sin(2\pi f nT + \phi), \tag{2.5}$$

where $T = 1/f_s$ is the sampling period in seconds and f_s is the sampling rate (or sampling frequency) in Hz. This digital signal can also be expressed as

$$x(n) = A\sin(\omega n + \phi) = A\sin(F\pi n + \phi), \tag{2.6}$$

where

$$\omega = \Omega T = \frac{2\pi f}{f_s} \tag{2.7}$$

is the digital frequency in radians per sample, and

$$F = \frac{\omega}{\pi} = \frac{f}{(f_s/2)} \tag{2.8}$$

is the normalized digital frequency in cycles per sample.

The units, relationships, and ranges of these analog and digital frequency variables are summarized in Table 2.1. Sampling of analog signals results in the mapping of analog frequency variable f (or Ω) into a finite range of digital frequency variable ω (or F). The

Table 2.1 Units, relationships, and ranges of four frequency variables

Variables	Units	Relationships	Ranges
Ω	Radians per second	$\Omega = 2\pi f$	$-\infty < \Omega < \infty$
f	Cycles per second (Hz)	$f = \frac{\Omega}{2\pi} = \frac{\omega f_s}{2\pi}$	$-\infty < f < \infty$
ω	Radians per sample	$\omega = \Omega T = \frac{2\pi f}{f_s}$	$-\pi \le \omega \le \pi$
F	Cycles per sample	$F = \frac{f}{(f_s/2)} = \frac{\omega}{\pi}$	$-1 \le F \le 1$

highest frequency in a digital signal is $f = f_s/2, \omega = \pi$, or $F = 1$ based on the sampling theorem defined in (1.3). Therefore, the frequency contents of digital signals are restricted to the limited range as shown in Table 2.1.

Example 2.1

Generate 32 sine wave samples with $A = 2$, $f = 1000$ Hz, and $f_s = 8000$ Hz using MATLAB® [5–7].

From Table 2.1, we have $\omega = 2\pi f/f_s = 0.25\pi$. From (2.6), we can express the sine wave as $x(n) = 2\sin(0.25\pi n)$, $n = 0, 1, \ldots, 31$. The generated sine wave samples are plotted (as shown in Figure 2.1) and saved in a data file using ASCII format by the following MATLAB® script (`example2_1.m`):

```
n = [0:31];                   % Time index n=0,1, . . . ,31
omega = 0.25*pi;              % Digital frequency
xn = 2*sin(omega*n);          % Sine wave generation
plot(n, xn, '-o');            % Samples are marked by 'o'
xlabel('Time index, n');
ylabel('Amplitude');
axis([0 31 -2 2]);            % Define ranges of plot
save sine.dat xn -ascii;      % Save in ASCII data file
```

In the code, the MATLAB® function `save` is used to store the variable `xn` using eight-digit ASCII format in the data file `sine.dat`. This ASCII file can be read by other MATLAB® scripts using the function `load`. The default format for the `save` and `load` functions is a binary MAT-file with the extension `.mat`.

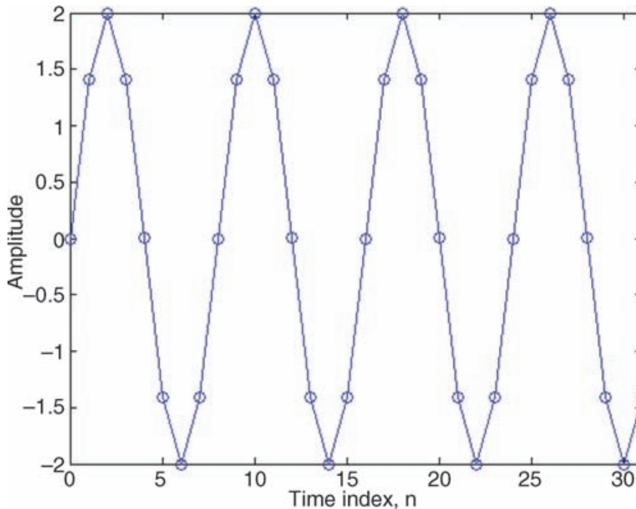


Figure 2.1 An example of a sine wave with $A = 2$ and $\omega = 0.25\pi$

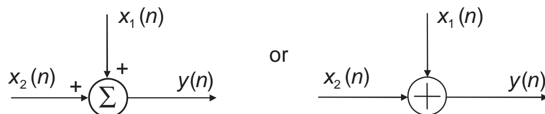


Figure 2.2 Block diagram of an adder

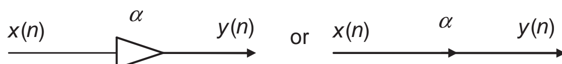


Figure 2.3 Block diagram of a multiplier

2.1.2 Block Diagram Representation of Digital Systems

A DSP system performs prescribed operations on signals. The processing of digital signals can be described as a combination of three basic operations: addition (or subtraction), multiplication, and time shift (or delay). Thus, a DSP system consists of the interconnection of three basic elements: adders, multipliers, and delay units.

Two signals, $x_1(n)$ and $x_2(n)$, can be added as illustrated in Figure 2.2, where the adder output is expressed as

$$y(n) = x_1(n) + x_2(n). \tag{2.9}$$

The adder could be drawn as a multi-input adder with more than two inputs, but the additions are typically performed with two signals at a time in practical DSP systems.

A given signal can be multiplied by a scalar, α, as illustrated in Figure 2.3, where $x(n)$ is the multiplier input and the multiplier's output is

$$y(n) = \alpha x(n). \tag{2.10}$$

Multiplication of a sequence by a gain factor, α, results in all samples in the sequence being scaled by α. The output signal is amplified if $|\alpha| > 1$, or attenuated if $|\alpha| < 1$.

The signal $x(n)$ can be delayed in time by one sampling period, T, as illustrated in Figure 2.4, where the box labeled z^{-1} represents the unit delay, $x(n)$ is the input signal, and the output signal

$$y(n) = x(n-1). \tag{2.11}$$

In fact, the signal $x(n-1)$ is actually the previous signal sample stored in memory before the current time n. Therefore, the delay unit is very easy to realize in a digital system with memory, but is difficult to implement in an analog system. A delay by more than one unit

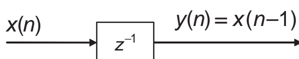


Figure 2.4 Block diagram of a unit delay

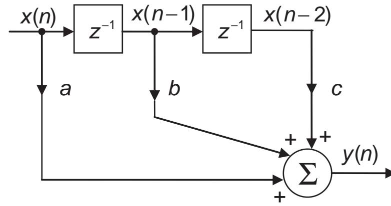


Figure 2.5 Signal-flow diagram of the DSP system described by (2.12)

can be implemented by cascading several delay units in a row. Therefore, an L-unit delay requires $L+1$ memory locations configured as a first-in, first-out buffer (tapped delay line) in memory.

Example 2.2

Consider a simple DSP system described by the difference equation

$$y(n) = ax(n) + bx(n-1) + cx(n-2), \tag{2.12}$$

where a, b, and c are real numbers. The signal-flow diagram of the system using three basic building blocks is sketched in Figure 2.5, which shows that the output signal $y(n)$ is computed using three multiplications and two additions. The difference equation given in (2.12) defines the I/O relationship of the system, and thus is also called the I/O equation.

2.2 System Concepts

This section introduces basic concepts and techniques to describe and analyze linear time-invariant (LTI) digital systems.

2.2.1 LTI Systems

If the input signal to an LTI system is the unit-impulse signal $\delta(n)$ defined in (2.1), then the output signal $y(n)$ is the impulse response of the system, $h(n)$.

Example 2.3

Consider a digital system defined by the I/O equation

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2). \tag{2.13}$$

Applying the unit-impulse signal $\delta(n)$ to the input of the system, the output $y(n)$ is the system impulse response $h(n)$ that can be computed as follows:

$$h(0) = y(0) = b_0 \cdot 1 + b_1 \cdot 0 + b_2 \cdot 0 = b_0$$
$$h(1) = y(1) = b_0 \cdot 0 + b_1 \cdot 1 + b_2 \cdot 0 = b_1$$
$$h(2) = y(2) = b_0 \cdot 0 + b_1 \cdot 0 + b_2 \cdot 1 = b_2$$
$$h(3) = y(3) = b_0 \cdot 0 + b_1 \cdot 0 + b_2 \cdot 0 = 0$$

Therefore, the impulse response of the system defined in (2.13) is $\{b_0, b_1, b_2, 0, 0, \ldots\}$.

The I/O equation given in (2.13) can be generalized with L coefficients as

$$\begin{aligned} y(n) &= b_0 x(n) + b_1 x(n-1) + \cdots + b_{L-1} x(n-L+1) \\ &= \sum_{l=0}^{L-1} b_l x(n-l). \end{aligned} \tag{2.14}$$

Substituting $x(n) = \delta(n)$ into (2.14), the output is the impulse response expressed as

$$\begin{aligned} h(n) &= \sum_{l=0}^{L-1} b_l \delta(n-l) \\ &= \begin{cases} b_n, & n = 0, 1, \ldots, L-1 \\ 0, & \text{otherwise.} \end{cases} \end{aligned} \tag{2.15}$$

Therefore, the length of the impulse response is L for the system defined in (2.14). This system is called a finite impulse response (FIR) filter. The parameters, b_l, $l = 0, 1, \ldots, L-1$, are filter coefficients (also referred as filter weights or taps). For FIR filters, the filter coefficients are identical to the non-zero samples of impulse response.

The signal-flow diagram of the system described by the I/O equation (2.14) is illustrated in Figure 2.6. The string of z^{-1} units is the tapped-delay line. The parameter, L, is the length of the FIR filter. Note that the order of the filter is $L-1$ for the FIR filter with length L since there are $L-1$ zeros. The design and implementation of FIR filters will be further discussed in Chapter 3.

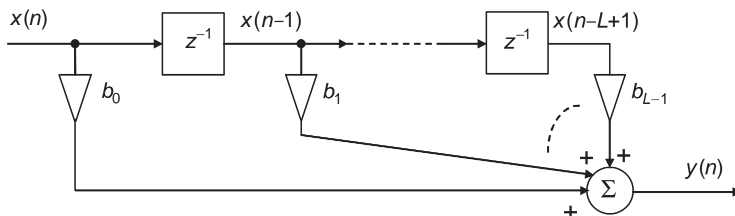


Figure 2.6 Detailed signal-flow diagram of an FIR filter

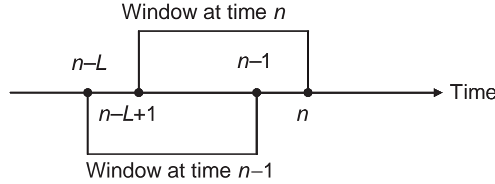


Figure 2.7 Time windows at current time n and previous time $n-1$

The moving (running) average filter is a simple example of the FIR filter. Consider the L-point moving-average filter defined as

$$\begin{aligned} y(n) &= \frac{1}{L}[x(n) + x(n-1) + \cdots + x(n-L+1)] \\ &= \frac{1}{L}\sum_{l=0}^{L-1} x(n-l), \end{aligned} \tag{2.16}$$

where each output signal sample is the average of L consecutive samples of current and passed input signal. Implementation of (2.16) requires $L-1$ additions and L memory locations for storing signal samples $x(n), x(n-1), \ldots, x(n-L+1)$ in the memory buffer. Because most digital signal processors have a hardware multiplier, the division by a constant L can be implemented by multiplication of a constant α, where $\alpha = 1/L$.

The concept of a moving window is illustrated in Figure 2.7, where L signal samples inside the rectangular window at time n are used to compute the current output signal $y(n)$. Comparing the windows at time n and $n-1$, the oldest sample $x(n-L)$ for the window at time $n-1$ is replaced by the newest sample $x(n)$ for the window at time n, and the remaining $L-1$ samples are the same as those samples used by the previous window at time $n-1$ to compute $y(n-1)$. Therefore, the averaged signal, $y(n)$, can be computed recursively as

$$y(n) = y(n-1) + \frac{1}{L}[x(n) - x(n-L)]. \tag{2.17}$$

This recursive equation can be realized by using only two additions. Comparing it to the direct implementation of (2.16) that needs $L-1$ additions, the recursive equation (2.17) shows that with careful design (or optimization), the complexity of a system (or algorithm) can be reduced. However, we still need $L+1$ memory locations for keeping $L+1$ signal samples $\{x(n)\ x(n-1)\ \ldots\ x(n-L)\}$.

Consider the LTI system illustrated in Figure 2.8, where the output signal can be expressed as

$$\begin{aligned} y(n) &= x(n) * h(n) = h(n) * x(n) \\ &= \sum_{l=-\infty}^{\infty} x(l)h(n-l) = \sum_{l=-\infty}^{\infty} h(l)x(n-l), \end{aligned} \tag{2.18}$$

where $*$ denotes the linear convolution operation. The exact internal structure of the system is either unknown or ignored. The only way to interact with the system is by using its input and

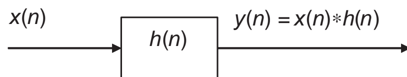


Figure 2.8 An LTI system expressed in the time domain

output terminals as shown in Figure 2.8. This "black box" representation is a very effective way to depict complicated DSP systems.

A digital system is a causal system if and only if

$$h(n) = 0, \quad n < 0. \tag{2.19}$$

A causal system does not provide a zero-state response prior to input application. That is, the output depends only on the current and past samples of the input. This is an obvious property for real-time DSP systems. However, if the data is a previous digitized signal saved in a file, the algorithm operating on the data set does not need to be causal. For a causal system, the lower bounds of the summation in (2.18) can be modified to reflect this restriction as

$$y(n) = \sum_{l=0}^{\infty} h(l)x(n-l). \tag{2.20}$$

Example 2.4

Consider the I/O equation of a digital system expressed as

$$y(n) = bx(n) - ay(n-1), \tag{2.21}$$

where each output signal $y(n)$ is dependent on the current input signal $x(n)$ and the past output signal $y(n-1)$. Assume that the system is causal, that is, $y(n) = 0$ for $n < 0$, and the input is a unit-impulse signal, that is, $x(n) = \delta(n)$. The output signal (impulse response) samples are computed as

$$\begin{aligned}
y(0) &= bx(0) - ay(-1) = b \\
y(1) &= bx(1) - ay(0) = -ay(0) = -ab \\
y(2) &= bx(2) - ay(1) = -ay(1) = a^2b.
\end{aligned}$$

In general, we have $h(n) = y(n) = (-1)^n a^n b, n = 0, 1, 2, \ldots, \infty$. This system has infinite impulse response $h(n)$ if the coefficients a and b are non-zero, and thus is called an infinite impulse response (IIR) system.

A digital filter can be classified as either an FIR filter or IIR filter, depending on whether the impulse response of the filter has finite or infinite duration. The system defined in (2.21) is an IIR filter since it has infinite impulse response as shown by Example 2.4. The I/O equation of an IIR system can be generalized as

$$\begin{aligned}
y(n) &= b_0x(n) + b_1x(n-1) + \cdots + b_{L-1}x(n-L+1) - a_1y(n-1) - \cdots - a_My(n-M) \\
&= \sum_{l=0}^{L-1} b_l x(n-l) - \sum_{m=1}^{M} a_m y(n-m),
\end{aligned} \tag{2.22}$$

where M is the order of the system. This IIR system is defined by a set of feedforward coefficients $\{b_l, l = 0, 1, \ldots, L-1\}$ and feedback coefficients $\{a_m, m = 1, 2, \ldots, M\}$. Since the weighted output samples (by a_m) are fed back and combined with the weighted input samples (by b_l), IIR systems are feedback systems. Note that if all the a_m are zero, Equation (2.22) is identical to (2.14) which defines an FIR filter. Therefore, an FIR filter is a special case of an IIR filter when all feedback coefficients a_m are zero. The design and implementation of IIR filters will be further discussed in Chapter 4.

Example 2.5

The IIR filters given in (2.22) can be implemented using the MATLAB® function `filter` as

```
yn = filter(b, a, xn);
```

The vector `b` contains feedforward coefficients $\{b_l, l = 0, 1, \ldots, L-1\}$ and the vector `a` contains feedback coefficients $\{a_m, m = 0, 1, 2, \ldots, M$, where $a_0 = 1\}$. The signal vectors `xn` and `yn` are the input and output buffers of the system, respectively. The FIR filter defined in (2.14) can be implemented as

```
yn = filter(b, 1, xn);
```

This is because all a_m are zero except $a_0 = 1$ for the FIR filter.

Example 2.6

Assume the window length L is large so that the oldest sample $x(n-L)$ can be approximated by its average $y(n-1)$; then the moving-average filter defined in (2.17) can be approximated as

$$\begin{aligned} y(n) &\cong \left(1 - \frac{1}{L}\right) y(n-1) + \frac{1}{L} x(n) \\ &= (1-\alpha) y(n-1) + \alpha x(n) \end{aligned} \tag{2.23}$$

where $\alpha = 1/L$. This is a simple first-order IIR filter. Comparing (2.23) to (2.17), we need two multiplications instead of one, but only two memory locations instead of $L+1$. Thus, the recursive equation (2.23) is the most efficient technique for approximating a moving-average filter.

2.2.2 The z-transform

Continuous-time systems are commonly designed and analyzed using the Laplace transform, which will be briefly introduced in Chapter 4. For discrete-time systems, the transform

corresponding to the Laplace transform is the z-transform. The z-transform of a digital signal $x(n)$ is defined as

$$X(z) = \sum_{n=-\infty}^{\infty} x(n) z^{-n}, \tag{2.24}$$

where $X(z)$ represents the z-transform of $x(n)$. The variable z is a complex number and can be expressed in polar form as

$$z = r\mathrm{e}^{\mathrm{j}\theta}, \tag{2.25}$$

where r is the magnitude (radius) of z and θ is the angle of z. When $r = 1$, $|z| = 1$ is called the unit circle on the z-plane. Since the z-transform involves an infinite power series, it exists only for those values of z where the power series defined in (2.24) converges. The region on the complex z-plane in which the power series converges is called the region of convergence.

For causal signals, the two-sided z-transform defined in (2.24) becomes the one-sided z-transform expressed as

$$X(z) = \sum_{n=0}^{\infty} x(n) z^{-n}. \tag{2.26}$$

Example 2.7

Consider the exponential function

$$x(n) = a^n u(n).$$

The z-transform can be computed as

$$X(z) = \sum_{n=-\infty}^{\infty} a^n z^{-n} u(n) = \sum_{n=0}^{\infty} \left(az^{-1}\right)^n.$$

Using the infinite geometric series given in Appendix A, we have

$$X(z) = \frac{1}{1 - az^{-1}} = \frac{z}{z - a} \quad \text{if } \left|az^{-1}\right| < 1$$

Thus the equivalent condition for convergence (or region of convergence) is

$$|z| > |a|,$$

which is the region outside the circle with radius a.

Some z-transform properties are useful for the analysis of discrete-time LTI systems. These properties are summarized as follows:

1. *Linearity* (superposition). The z-transform (ZT) of the sum of two sequences is the sum of the z-transforms of the individual sequences. That is,

$$\begin{aligned} \mathrm{ZT}[a_1 x_1(n) + a_2 x_2(n)] &= a_1 \mathrm{ZT}[x_1(n)] + a_2 \mathrm{ZT}[x_2(n)] \\ &= a_1 X_1(z) + a_2 X_2(z), \end{aligned} \tag{2.27}$$

   where a_1 and a_2 are constants.

2. *Time shifting* (delay). The z-transform of the shifted (delayed) signal $y(n) = x(n-k)$ is

$$Y(z) = \mathrm{ZT}[x(n-k)] = z^{-k} X(z). \tag{2.28}$$

   For example, $\mathrm{ZT}[x(n-1)] = z^{-1} X(z)$. The unit delay z^{-1} corresponds to a time shift to the right of one sample.

3. *Convolution*. Considering the signal $x(n)$ as a linear convolution of two sequences

$$x(n) = x_1(n) * x_2(n), \tag{2.29}$$

   we have

$$X(z) = X_1(z) X_2(z). \tag{2.30}$$

   Thus, the z-transform converts the linear convolution in the time domain to multiplication in the z-domain.

2.2.3 Transfer Functions

Consider the LTI system illustrated in Figure 2.8. Using the convolution property, we have

$$Y(z) = X(z) H(z), \tag{2.31}$$

where $X(z) = \mathrm{ZT}[x(n)]$, $Y(z) = \mathrm{ZT}[y(n)]$, and $H(z) = \mathrm{ZT}[h(n)]$. The combination of time- and z-domain representations of the LTI system is illustrated in Figure 2.9, where ZT^{-1} denotes the inverse z-transform. This diagram shows that we can compute the output of a linear system by replacing the time-domain convolution with the z-domain multiplication.

The transfer function of an LTI system is defined in terms of the system's input and output. From (2.31), the transfer function is defined as

$$H(z) = \frac{Y(z)}{X(z)}. \tag{2.32}$$

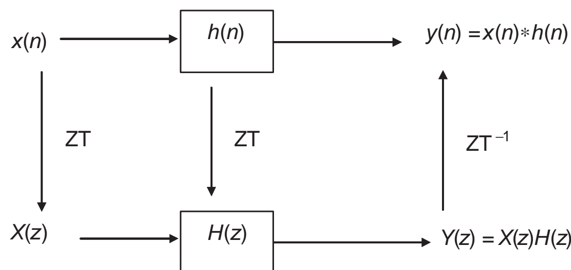


Figure 2.9 A block diagram of the LTI system in both the time domain and z domain

The transfer function can be used to create alternative filters that have exactly the same I/O behavior. An important example is the cascade or parallel connection of two or more low-order systems to realize a high-order system, as illustrated in Figure 2.10. In the cascade (series) interconnection shown in Figure 2.10(a), we have

$$Y_1(z) = X(z)H_1(z) \quad \text{and} \quad Y(z) = Y_1(z)H_2(z).$$

Thus,

$$Y(z) = X(z)H_1(z)H_2(z).$$

Therefore, the overall transfer function of the cascade of two systems is

$$H(z) = H_1(z)H_2(z) = H_2(z)H_1(z). \tag{2.33}$$

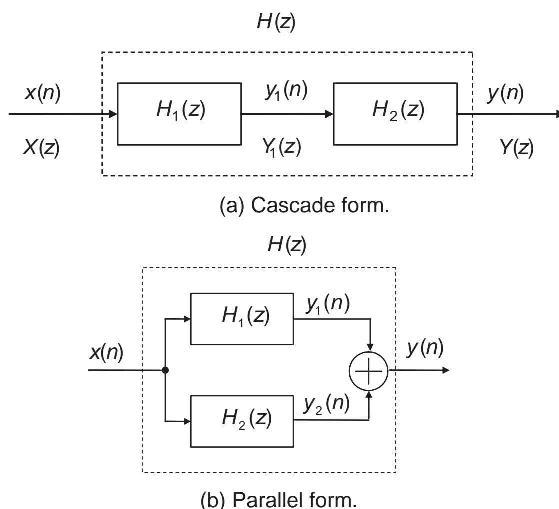


Figure 2.10 Interconnect of digital systems

Since multiplication is commutative, these two systems can be cascaded in either order to obtain the same overall system. Realization of IIR filters using a cascade structure for practical applications will be discussed in Chapter 4.

Similarly, the overall transfer function of the parallel connection of two LTI systems shown in Figure 2.10(b) is given by

$$H(z) = H_1(z) + H_2(z). \tag{2.34}$$

In practical IIR filtering applications, we factor polynomials to break down a high-order filter $H(z)$ into small sections, such as second-order or first-order filters, and use the cascade form. The concepts of parallel and cascade realizations of IIR filters will be further discussed in Chapter 4.

Example 2.8

The LTI system with transfer function

$$H(z) = \frac{1}{1 - 2z^{-1} + z^{-3}}$$

can be factored as

$$H(z) = \left(\frac{1}{1 - z^{-1}}\right)\left(\frac{1}{1 - z^{-1} - z^{-2}}\right) = H_1(z)H_2(z).$$

Thus, the overall system $H(z)$ can be realized as the cascade of the first-order system $H_1(z) = 1/(1 - z^{-1})$ and the second-order system $H_2(z) = 1/(1 - z^{-1} - z^{-2})$.

The I/O equation of an FIR filter is given in (2.14). Taking the z-transform of both sides using the delay property given in (2.28), we have

$$\begin{aligned} Y(z) &= b_0X(z) + b_1z^{-1}X(z) + \cdots + b_{L-1}z^{-(L-1)}X(z) \\ &= \left(b_0 + b_1z^{-1} + \cdots + b_{L-1}z^{-(L-1)}\right)X(z). \end{aligned} \tag{2.35}$$

Therefore, the transfer function of the FIR filter is expressed as

$$H(z) = b_0 + b_1z^{-1} + \cdots + b_{L-1}z^{-(L-1)} = \sum_{l=0}^{L-1} b_l z^{-l}. \tag{2.36}$$

Similarly, taking the z-transform of both sides of the IIR filter defined in (2.22) yields

$$\begin{aligned} Y(z) &= b_0X(z) + b_1z^{-1}X(z) + \cdots + b_{L-1}z^{-L+1}X(z) - a_1z^{-1}Y(z) - \cdots - a_Mz^{-M}Y(z) \\ &= \left(\sum_{l=0}^{L-1} b_l z^{-l}\right)X(z) - \left(\sum_{m=1}^{M} a_m z^{-m}\right)Y(z). \end{aligned} \tag{2.37}$$

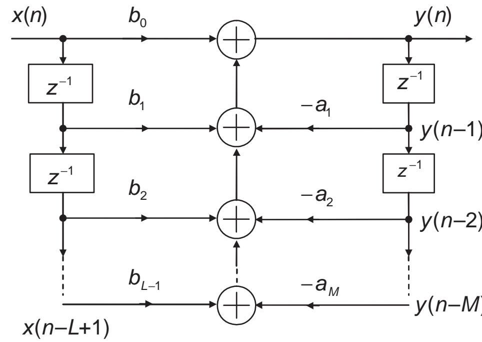


Figure 2.11 Detailed signal-flow diagram of IIR filter

By rearranging these terms, we can derive the transfer function of the IIR filter as

$$H(z) = \frac{\sum_{l=0}^{L-1} b_l z^{-l}}{1 + \sum_{m=1}^{M} a_m z^{-m}} = \frac{\sum_{l=0}^{L-1} b_l z^{-l}}{\sum_{m=0}^{M} a_m z^{-m}}, \tag{2.38}$$

where $a_0 = 1$. A detailed signal-flow diagram of the IIR filter is illustrated in Figure 2.11 for $M = L - 1$.

Example 2.9

Consider the moving-average filter given in (2.16). Taking the z-transform of both sides, we have

$$Y(z) = \frac{1}{L}\sum_{l=0}^{L-1} z^{-l} X(z).$$

Using the geometric series defined in Appendix A, the transfer function of the filter can be expressed as

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1}{L}\sum_{l=0}^{L-1} z^{-l} = \frac{1}{L}\left[\frac{1 - z^{-L}}{1 - z^{-1}}\right]. \tag{2.39}$$

This equation can be rearranged as

$$Y(z) = z^{-1}Y(z) + \frac{1}{L}\left[X(z) - z^{-L}X(z)\right].$$

Taking the inverse z-transform of both sides, we obtain

$$y(n) = y(n-1) + \frac{1}{L}[x(n) - x(n-L)]. \tag{2.40}$$

This is a formal way of deriving (2.17) from (2.16).

2.2.4 Poles and Zeros

Factoring the numerator and denominator polynomials of the rational function $H(z)$, (2.38) can be expressed as

$$H(z) = b_0 \frac{\prod_{l=1}^{L-1}(z - z_l)}{\prod_{m=1}^{M}(z - p_m)} = \frac{b_0(z-z_1)(z-z_2)\ldots(z-z_{L-1})}{(z-p_1)(z-p_2)\ldots(z-p_M)}. \tag{2.41}$$

The roots of the numerator polynomial are the zeros of the transfer function $H(z)$ since they are the values of z for which $H(z) = 0$. Thus, $H(z)$ given in (2.41) has $(L-1)$ zeros at $z = z_1, z_2, \ldots, z_{L-1}$. The roots of the denominator polynomial are the poles since they are the values of z such that $H(z) = \infty$, and there are M poles at $z = p_1, p_2, \ldots, p_M$. The LTI system defined in (2.41) is a pole–zero system of order M, while the system described in (2.36) is an all-zero system of order $L-1$.

Example 2.10

The roots of the numerator polynomial defined in (2.39) determine the zeros of $H(z)$, that is, $z^L - 1 = 0$. Using the complex arithmetic given in Appendix A, we have

$$z_l = e^{j(2\pi/L)l}, \quad l = 0, 1, \ldots, L-1. \tag{2.42}$$

Therefore, there are L zeros equally spaced on the unit circle $|z| = 1$.

Similarly, the poles of $H(z)$ are determined by the roots of the denominator $z^{L-1}(z-1)$. Thus, there are $L-1$ poles at the origin $z = 0$ and one pole at $z = 1$. A pole–zero diagram of $H(z)$ for $L = 8$ on the complex z-plane is illustrated in Figure 2.12. Note that the pole at $z = 1$ is canceled by the zero at $z = 1$. Therefore, the moving-average filter defined by (2.39) is an all-zero (or FIR) filter.

The pole–zero diagram provides important insight into the properties of LTI systems. To find the poles and zeros of a rational function $H(z)$, we can use the MATLAB® function `roots` on both the numerator and denominator polynomials. Another useful MATLAB® function for analyzing transfer functions is `zplane(b,a)`, which displays the pole–zero diagram of $H(z)$.

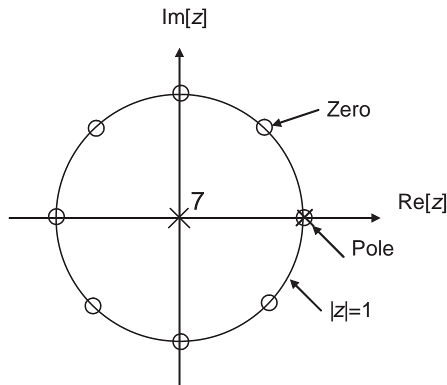


Figure 2.12 Pole–zero diagram of the moving-averaging filter, $L = 8$

Example 2.11

Consider the IIR filter with the transfer function

$$H(z) = \frac{1}{1 - z^{-1} + 0.9z^{-2}}.$$

We can plot the pole–zero diagram (Figure 2.13) using the following MATLAB® script (`example2_11a.m`):

```
b=[1];              % Define numerator
a=[1, −1, 0.9];     % Define denominator
zplane(b,a);        % Pole-zero plot
```

Similarly, we can plot the pole–zero diagram of a moving-average filter using the following MATLAB® script (`example2_11b`) for $L = 8$:

```
b=[1, 0, 0, 0, 0, 0, 0, 0, −1];
a=[1,  −1];
zplane(b,a);
```

Example 2.12

Consider the recursive approximation of the moving-average filter given in (2.23). Taking the z-transform of both sides and rearranging terms, we obtain the transfer function

$$H(z) = \frac{\alpha}{1 - (1 - \alpha)z^{-1}}. \tag{2.43}$$

This is the simple first-order IIR filter with a zero at the origin and a pole at $z = 1 - \alpha$. Note that $\alpha = 1/L$ (where L is equivalent to the length of the window) results in

$1-\alpha=(L-1)/L$, which is slightly less than one. Thus for a long window, L is large, the value of $1-\alpha$ is close to one, and the pole is close to the unit circle.

An LTI system $H(z)$ is stable if and only if the poles

$$|p_m|<1, \quad \text{for all } m. \tag{2.44}$$

That is, all poles are inside the unit circle. In this case, $\lim_{n\to\infty}h(n)=0$, that is, the impulse response will converge to zero. A system is unstable if $H(z)$ has any pole outside the unit circle or any multiple-order pole on the unit circle. For example, if $H(z)=z/(z-1)^2$, $h(n)=n$, this system is unstable. A system is marginally stable, or oscillatory bounded, if $H(z)$ has a first-order pole that lies on the unit circle and the rest of the poles are inside the unit circle. For example, if $H(z)=z/(z+1)$, $h(n)=(-1)^n$, $n\geq 0$, this system is marginally stable.

Example 2.13

Consider the LTI system with transfer function

$$H(z)=\frac{z}{z-a}.$$

There is a zero at the origin $z=0$ and a pole at $z=a$. From Example 2.7, we have

$$h(n)=a^n, \quad n\geq 0. \tag{2.45}$$

When $|a|>1$, the pole at $z=a$ is outside the unit circle. Also, from (2.45) we have $\lim_{n\to\infty} h(n)\to\infty$, thus the system is unstable. However, when $|a|<1$, the pole is inside the unit circle and we have $\lim_{n\to\infty} h(n)\to 0$, which is a stable system.

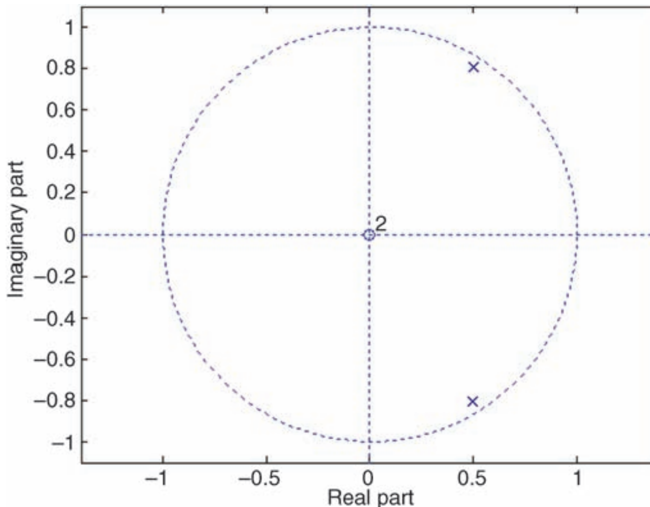


Figure 2.13 A pole–zero diagram generated by MATLAB®

2.2.5 Frequency Responses

The frequency response of a digital system $H(\omega)$ can be obtained from its transfer function $H(z)$ by setting $z = \mathrm{e}^{\mathrm{j}\omega}$. This is equivalent to computing the discrete-time Fourier transform (to be discussed in Chapter 5) of the infinite length impulse response $h(n)$ expressed as

$$H(\omega) = H(z)|_{z=\mathrm{e}^{\mathrm{j}\omega}} = \sum_{n=-\infty}^{\infty} h(n)\mathrm{e}^{-\mathrm{j}\omega n}. \tag{2.46}$$

Therefore, the frequency response $H(\omega)$ is obtained by evaluating the transfer function on the unit circle $|z| = \left|\mathrm{e}^{\mathrm{j}\omega}\right| = 1$. As listed in Table 2.1, the digital frequency ω defined in (2.7) is in the range of $-\pi \le \omega \le \pi$.

The characteristics of the systems can be analyzed using the frequency response. In general, $H(\omega)$ is a complex-valued function expressed in polar form as

$$H(\omega) = \mathrm{Re}\,[H(\omega)] + \mathrm{j}\,\mathrm{Im}[H(\omega)] = |H(\omega)|\,\mathrm{e}^{\mathrm{j}\phi(\omega)}, \tag{2.47}$$

where

$$|H(\omega)| = \sqrt{\{\mathrm{Re}[H(\omega)]\}^2 + \{\mathrm{Im}\,[H(\omega)]\}^2} \tag{2.48}$$

is the magnitude (or amplitude) response and

$$\phi(\omega) = \begin{cases} \tan^{-1}\left\{\dfrac{\mathrm{Im}[H(\omega)]}{\mathrm{Re}[H(\omega)]}\right\}, & \text{if } \mathrm{Re}[H(\omega)] \ge 0 \\ \pi + \tan^{-1}\left\{\dfrac{\mathrm{Im}[H(\omega)]}{\mathrm{Re}[H(\omega)]}\right\}, & \text{if } \mathrm{Re}[H(\omega)] < 0 \end{cases} \tag{2.49}$$

is the phase response (or angle) of the system $H(z)$. The magnitude response $|H(\omega)|$ is an even function of ω, that is, $|H(-\omega)| = |H(\omega)|$; and the phase response $\phi(\omega)$ is an odd function, that is, $\phi(-\omega) = -\phi(\omega)$. Thus, we only need to evaluate these functions in the frequency region $0 \le \omega \le \pi$ since $|H(\omega)|$ is symmetric (or a mirror image) about $\omega = 0$. Note that $|H(\omega_0)|$ is the gain and $\phi(\omega_0)$is the phase shift of the system at frequency ω_0.

Example 2.14

The two-point moving-average filter can be expressed as

$$y(n) = \frac{1}{2}[x(n) + x(n-1)], \quad n \ge 0.$$

Taking the z-transform of both sides and rearranging the terms, we obtain

$$H(z) = \frac{1}{2}\left(1 + z^{-1}\right).$$

This is the simple first-order FIR filter. From (2.46), we have

$$\begin{aligned}
H(\omega) &= \frac{1}{2}\left(1 + e^{-j\omega}\right) = \frac{1}{2}(1 + \cos\omega - j\sin\omega), \\
|H(\omega)| &= \sqrt{\{\mathrm{Re}[H(\omega)]\}^2 + \{\mathrm{Im}[H(\omega)]\}^2} = \sqrt{\frac{1}{2}(1 + \cos\omega)}, \\
\phi(\omega) &= \tan^{-1}\left\{\frac{\mathrm{Im}[H(\omega)]}{\mathrm{Re}[H(\omega)]}\right\} = \tan^{-1}\left(\frac{-\sin\omega}{1 + \cos\omega}\right).
\end{aligned}$$

From Appendix A,

$$\sin\omega = 2\sin\left(\frac{\omega}{2}\right)\cos\left(\frac{\omega}{2}\right) \quad \text{and} \quad \cos\omega = 2\cos^2\left(\frac{\omega}{2}\right) - 1.$$

Therefore, the phase response is

$$\phi(\omega) = \tan^{-1}\left[-\tan\left(\frac{\omega}{2}\right)\right] = -\frac{\omega}{2}.$$

For the transfer function $H(z)$ expressed in (2.38) where the numerator and denominator coefficients are given in the vectors `b` and `a`, respectively, the frequency response can be analyzed using the MATLAB® function

```
[H,w]=freqz(b,a)
```

which returns the complex frequency response vector `H` and the frequency vector `w`. Note that the function `freqz(b,a)` will plot both the magnitude response and the phase response of $H(z)$.

Example 2.15

Consider the IIR filter defined as

$$y(n) = x(n) + y(n-1) - 0.9y(n-2).$$

The transfer function is

$$H(z) = \frac{1}{1 - z^{-1} + 0.9z^{-2}}.$$

The MATLAB® script (`example2_15a.m`) for analyzing the magnitude and phase responses of this IIR filter is listed as follows:

```
b=[1]; a=[1, -1, 0.9];    % Define numerator and denominator
freqz(b,a);               % Plot magnitude and phase responses
```

Similarly, we can plot the magnitude and phase responses (shown in Figure 2.14) of the moving-average filter for $L = 8$ using the following script (`example2_15b.m`):

```
b=[1, 0, 0, 0, 0, 0, 0, 0, -1];  a=[1, -1];
freqz(b,a);
```

A useful method for obtaining the brief frequency response of LTI systems is based on the geometric evaluation of the poles and zeros. For example, consider the second-order IIR filter expressed as

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}, \tag{2.50}$$

where the filter coefficients are real valued. The roots of the characteristic equation

$$z^2 + a_1 z + a_2 = 0 \tag{2.51}$$

are the two poles of the filter, which may be either both real or complex-conjugate poles. Complex-conjugate poles can be expressed as

$$p_1 = r\,\mathrm{e}^{\mathrm{j}\theta} \quad \text{and} \quad p_2 = r\,\mathrm{e}^{-\mathrm{j}\theta}, \tag{2.52}$$

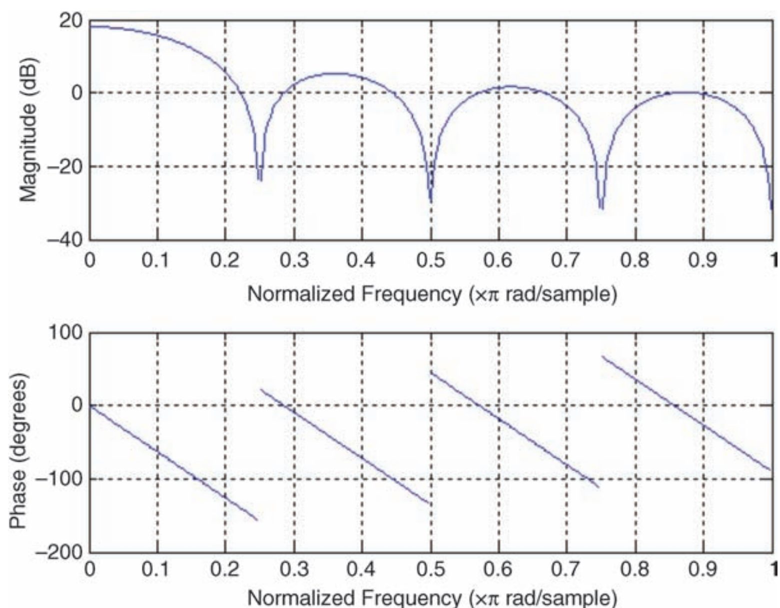


Figure 2.14 Magnitude (top) and phase responses of a moving-average filter, $L = 8$

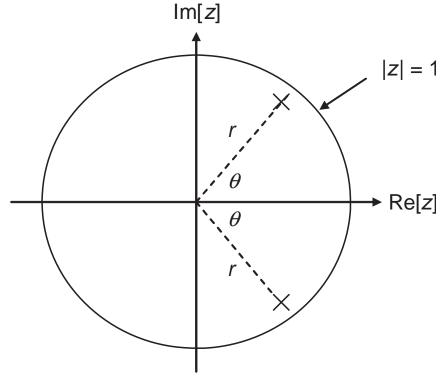


Figure 2.15 A second-order IIR filter with complex-conjugate poles

where r is the radius of the pole and θ is the angle of the pole. Therefore, (2.51) becomes

$$\left(z - r\,\mathrm{e}^{\mathrm{j}\theta}\right)\left(z - r\,\mathrm{e}^{-\mathrm{j}\theta}\right) = z^2 - 2r\cos(\theta)z + r^2 = 0. \tag{2.53}$$

Comparing this equation to (2.52), we obtain

$$r = \sqrt{a_2} \quad \text{and} \quad \theta = \cos^{-1}\left(\frac{-a_1}{2r}\right). \tag{2.54}$$

The system with the pair of complex-conjugate poles given in (2.52) is illustrated in Figure 2.15. This filter behaves as a digital resonator for r close to unity. The digital resonator is a bandpass filter with its passband centered at the resonant frequency θ. This issue will be further discussed in Chapter 4.

Similarly, we can obtain two zeros, z_1 and z_2, by evaluating $b_0z^2 + b_1z + b_2 = 0$. Thus, the transfer function defined in (2.50) can be expressed as

$$H(z) = \frac{b_0(z - z_1)(z - z_2)}{(z - p_1)(z - p_2)}. \tag{2.55}$$

In this case, the frequency response is given by

$$H(\omega) = \frac{b_0\left(\mathrm{e}^{\mathrm{j}\omega} - z_1\right)\left(\mathrm{e}^{\mathrm{j}\omega} - z_2\right)}{(\mathrm{e}^{\mathrm{j}\omega} - p_1)(\mathrm{e}^{\mathrm{j}\omega} - p_2)}. \tag{2.56}$$

The magnitude response can be obtained by evaluating $|H(\omega)|$ as the point z moves on the unit circle in a counterclockwise direction from $z = 1$ ($\theta = 0$) to $z = -1$ ($\theta = \pi$). As the point z moves closer to the pole p_1, the magnitude response increases. The closer r is to unity, the sharper and higher the peak. On the other hand, as the point z moves closer to the zero z_1, the magnitude response decreases. The magnitude response exhibits a peak at the pole angle (or frequency), whereas the magnitude response falls to a valley at an angle of zero. For example,

Figure 2.12 shows four zeros at $\theta = \pi/4$, $\pi/2$, $3\pi/4$, and π corresponding to four valleys shown in Figure 2.14.

2.2.6 Discrete Fourier Transform

To perform frequency analysis of $x(n)$, we can convert the time-domain signal into the frequency domain using the z-transform defined in (2.26), and substitute $z = e^{j\omega}$ in $X(z)$ as shown in (2.46). However, $X(\omega)$ is a continuous function of continuous frequency ω, and it also requires an infinite number of $x(n)$ samples for calculation. Therefore, it is difficult to compute $X(\omega)$ using digital hardware.

The discrete Fourier transform (DFT) of N-point signals $\{x(0), x(1), x(2), \ldots, x(N-1)\}$ can be obtained by sampling $X(\omega)$ on the unit circle at N equally spaced frequencies $\omega_k = 2\pi k/N$, $k = 0, 1, \ldots, N-1$. From (2.46), we have

$$X(k) = X(\omega)|_{\omega=2\pi k/N} = \sum_{n=0}^{N-1} x(n)e^{-j(2\pi k/N)n}, \quad k = 0, 1, \ldots, N-1, \tag{2.57}$$

where n is the time index, k is the frequency index, and $X(k)$ is the kth DFT coefficient. The computation of the DFT can be manipulated to obtain very efficient algorithms called fast Fourier transforms (FFTs). The derivation, implementation, and application of DFTs and FFTs will be further discussed in Chapter 5.

MATLAB® provides the function `fft(x)` to compute the DFT of the signal vector `x`. The function `fft(x,N)` performs N-point DFT. If the length of `x` is less than `N`, then `x` is padded with zeros at the end to create an N-point sequence. If the length of `x` is greater than `N`, the function `fft(x,N)` truncates the sequence `x` and performs the DFT of the first `N` samples only.

The DFT generates N coefficients $X(k)$ equally spaced over the frequency range from 0 to 2π. Therefore, the frequency resolution of the N-point DFT is

$$\Delta_\omega = \frac{2\pi}{N} \tag{2.58}$$

or

$$\Delta_f = \frac{f_s}{N}. \tag{2.59}$$

The analog frequency f_k (in Hz) corresponding to the index k can be expressed as

$$f_k = k\Delta_f = \frac{kf_s}{N}, \quad k = 0, 1, \ldots, N-1. \tag{2.60}$$

Note that the Nyquist frequency $(f_s/2)$ corresponds to the frequency index $k = N/2$. Since the magnitude $|X(k)|$ is an even function of k, we only need to display the magnitude spectrum for $0 \le k \le N/2$ (or $0 \le \omega_k \le \pi$).

Example 2.16

Similar to Example 2.1, we can generate 100 sine wave samples with $A = 1, f = 1000$ Hz, and a sampling rate of 10 000 Hz. The magnitude spectrum of the signal can be computed and plotted (as in Figure 2.16) using the following MATLAB® script (`example2_16.m`):

```
N=100; f = 1000; fs = 10000;        % Define parameter values
n=[0:N-1]; k=[0:N-1];               % Define time and frequency indices
omega=2*pi*f/fs;                    % Frequency of sine wave
xn=sin(omega*n);                    % Generate sine wave
Xk=fft(xn,N);                       % Perform DFT
magXk=20*log10(abs(Xk));            % Compute magnitude spectrum
plot(k, magXk);                     % Plot magnitude spectrum
axis([0, N/2, -inf, inf]);          % Plot from 0 to pi
xlabel('Frequency index, k');
ylabel('Magnitude in dB');
```

From (2.59), the frequency resolution is 100 Hz. The peak of the spectrum shown in Figure 2.16 is located at the frequency index $k = 10$, which corresponds to 1000 Hz as indicated by (2.60). The magnitude spectrum is usually represented by a dB (decibel) scale, which is computed using `20*log10(abs(Xk))`, where the function `abs` calculates absolute value.

2.3 Introduction to Random Variables

The signals encountered in practice are often random signals such as speech, music, and ambient noise. In this section, we will give a brief introduction to the basic concepts of random variables that are useful for understanding quantization effects presented in this book [8].

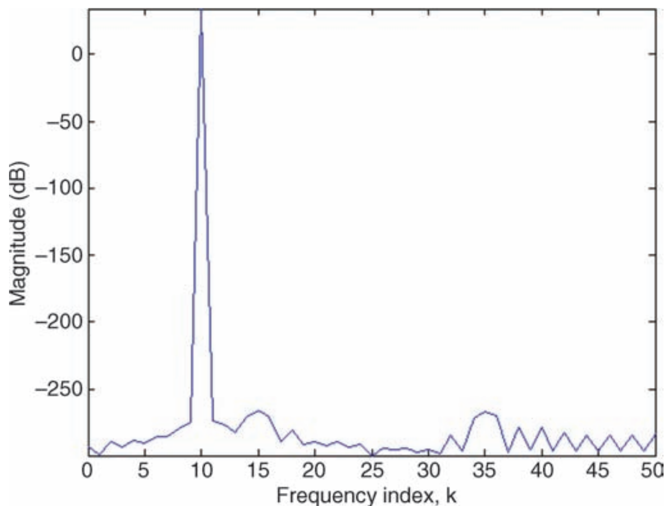


Figure 2.16 Magnitude spectrum of sine wave

Additional principles of random variables will be further discussed in Chapter 6 for introducing adaptive filters.

2.3.1 Review of Random Variables

An experiment that has at least two possible outcomes is fundamental to the concept of probability. The set of all possible outcomes in any given experiment is called the sample space S. A random variable, x, is defined as a function that maps all elements from the sample space S into points on the real line. For example, considering the outcomes of rolling of a fair die, we obtain a discrete random variable that can be any one of the discrete values from 1 through 6.

The cumulative probability distribution function of a random variable x is defined as

$$F(X) = P(x \leq X), \tag{2.61}$$

where X is a real number, and $P(x \leq X)$ is the probability of $\{x \leq X\}$. The probability density function of a random variable x is defined as

$$f(X) = \frac{\mathrm{d}F(X)}{\mathrm{d}X} \tag{2.62}$$

if the derivative exists. Two important properties of the probability density function $f(X)$ are summarized as follows:

$$\int_{-\infty}^{\infty} f(X)\,\mathrm{d}X = 1 \tag{2.63}$$

$$P(X_1 < x \leq X_2) = F(X_2) - F(X_1) = \int_{X_1}^{X_2} f(X)\,\mathrm{d}X. \tag{2.64}$$

If x is a discrete random variable that can be any one of the discrete values $X_i, i = 1, 2, \ldots,$ as the result of an experiment, we define the discrete probability function as

$$p_i = P(x = X_i). \tag{2.65}$$

Example 2.17

Consider a random variable x that has the following probability density function:

$$f(X) = \begin{cases} 0, & x < X_1 \quad \text{or} \quad x > X_2 \\ a, & X_1 \leq x \leq X_2, \end{cases}$$

which is uniformly distributed between X_1 and X_2. The constant value a can be computed using (2.63) as

$$\int_{-\infty}^{\infty} f(X)\mathrm{d}X = \int_{X_1}^{X_2} a \cdot \mathrm{d}X = a[X_2 - X_1] = 1.$$

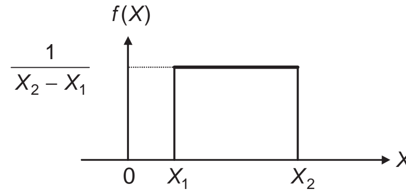


Figure 2.17 The uniform density function

Thus, we obtain

$$a = \frac{1}{X_2 - X_1}.$$

If a random variable x is equally likely to be any value between two limits X_1 and X_2, and cannot assume any value outside that range, it is uniformly distributed in the range $[X_1, X_2]$. As shown in Figure 2.17, the uniform density function is defined as

$$f(X) = \begin{cases} \dfrac{1}{X_2 - X_1}, & X_1 \leq x \leq X_2 \\ 0, & \text{otherwise.} \end{cases} \tag{2.66}$$

2.3.2 Operations of Random Variables

The statistics associated with random variables provide more meaningful information than the probability density function. The mean (expected value) of a random variable x is defined as

$$\begin{aligned} m_x = E[x] &= \int_{-\infty}^{\infty} Xf(X)\mathrm{d}X, \quad \text{continuous-time case} \\ &= \textstyle\sum_i X_i p_i, \quad \text{discrete-time case,} \end{aligned} \tag{2.67}$$

where $E[\cdot]$ denotes the expectation operation (or ensemble averaging). The mean m_x defines the level about which the random process x fluctuates.

The expectation is a linear operation. Two useful properties of the expectation operation are $E[\alpha] = \alpha$ and $E[\alpha x] = \alpha E[x]$, where α is a constant. If $E[x] = 0$, x is the zero-mean random variable. The MATLAB® function `mean` calculates the mean value. For example, the statement `mx = mean(x)` computes the mean `mx` of all the elements in the vector `x`.

Example 2.18

Consider the rolling of a fair die N times ($N \to \infty$), the probability of outcomes is listed as follows:

X_i	1	2	3	4	5	6
p_i	1/6	1/6	1/6	1/6	1/6	1/6

The mean of the outcomes can be computed as

$$m_x = \sum_{i=1}^{6} p_i X_i = \frac{1}{6}(1+2+3+4+5+6) = 3.5.$$

The variance is a measure of the spread about the mean, and is defined as

$$\begin{aligned} \sigma_x^2 &= E\left[(x-m_x)^2\right] \\ &= \int_{-\infty}^{\infty} (X-m_x)^2 f(X)\mathrm{d}X, \quad \text{continuous-time case} \\ &= \sum_i p_i (X_i - m_x)^2, \quad \text{discrete-time case,} \end{aligned} \tag{2.68}$$

where $(x - m_x)$ is the deviation of x from the mean value m_x. The positive square root of the variance is called the standard deviation σ_x. The MATLAB® function `std` calculates the standard deviation of the elements in the vector.

The variance defined in (2.68) can be expressed as

$$\begin{aligned} \sigma_x^2 &= E\left[(x-m_x)^2\right] = E\left[x^2 - 2xm_x + m_x^2\right] = E[x^2] - 2m_x E[x] + m_x^2 \\ &= E[x^2] - m_x^2. \end{aligned} \tag{2.69}$$

We call $E[x^2]$ the mean-square value of x. Thus, the variance is the difference between the mean-square value and the square of the mean value.

If the mean value is equal to zero, then the variance is equal to the mean-square value. For the zero-mean random variable x, that is, $m_x = 0$, we have

$$\sigma_x^2 = E\left[x^2\right] = P_x, \tag{2.70}$$

which is the power of x.

Consider the uniform density function defined in (2.66). The mean of the function can be computed by

$$\begin{aligned} m_x &= E[x] = \int_{-\infty}^{\infty} Xf(X)\mathrm{d}X = \frac{1}{X_2 - X_1}\int_{X_1}^{X_2} X\,\mathrm{d}X \\ &= \frac{X_2 - X_1}{2}. \end{aligned} \tag{2.71}$$

The variance of the function is

$$\begin{aligned} \sigma_x^2 &= E[x^2] - m_x^2 = \int_{-\infty}^{\infty} X^2 f(X)\mathrm{d}X - m_x^2 \\ &= \frac{1}{X_2 - X_1}\int_{X_1}^{X_2} X^2\,\mathrm{d}X - m_x^2 = \frac{1}{X_2 - X_1}\left(\frac{X_2^3 - X_1^3}{3}\right) - m_x^2 \\ &= \frac{(X_2 - X_1)^2}{12}. \end{aligned} \tag{2.72}$$

In general, if x is the random variable uniformly distributed in the interval $(-\Delta, \Delta)$, we have

$$m_x = 0 \quad \text{and} \quad \sigma_x^2 = \Delta^2/3. \tag{2.73}$$

Example 2.19

The MATLAB® function `rand` generates pseudo-random numbers uniformly distributed in the interval [0, 1]. From (2.71), the mean of the generated pseudo-random numbers is 0.5. From (2.72), the variance is 1/12.

To generate zero-mean random numbers, we subtract 0.5 from every generated random number. The numbers are now distributed in the interval [−0.5, 0.5]. To make these pseudo-random numbers have unit variance, that is, $\sigma_x^2 = \Delta^2/3 = 1$, the generated numbers must be equally distributed in the interval $[-\sqrt{3}, \sqrt{3}]$. This can be done by multiplying by $2\sqrt{3}$ every generated number from which 0.5 was subtracted.

The following MATLAB® statement can be used to generate the uniformly distributed random numbers with mean 0 and variance 1:

```
xn = 2*sqrt(3)*(rand-0.5);
```

The MATLAB® code of generating zero-mean, unit-variance ($\sigma_x^2 = 1$) white noise is given in `example2_19.m`.

Note that in earlier versions of MATLAB®, the integer seed `sd` is used in the syntax `rand('seed',sd)` to reproduce exactly the same random numbers each time. This syntax is no longer recommended in the newer versions of MATLAB®. In order to reproduce exactly the same (or repeatable) random numbers each time when we restart MATLAB® or rerun the program, we can reset the random number generator to its default startup settings by either using

```
rng('default')
```

or using the integer seed `sd` (a good choice of value is 12 357) as

```
rng(sd)
```

The principles of random number generators will be introduced in Chapter 7.

A sine wave $s(n)$ corrupted by white noise $v(n)$ can be expressed as

$$x(n) = A \sin(\omega n) + v(n). \tag{2.74}$$

When the signal $s(n)$ with power P_s is corrupted by the noise $v(n)$ with power P_v, the signal-to-noise ratio (SNR) in dB is defined as

$$\text{SNR} = 10 \log_{10}\left(\frac{P_s}{P_v}\right) \text{dB}. \tag{2.75}$$

From (2.70), the power of the sine wave defined in (2.6) can be computed as

$$P_s = E\left[A^2 \sin^2(\omega n)\right] = A^2/2. \tag{2.76}$$

Example 2.20

This example generates the signal $x(n)$ expressed in (2.74), where $v(n)$ is the zero-mean, unit-variance white noise. As shown in (2.76), when the sine wave amplitude $A = \sqrt{2}$, the power is equal to one. From (2.75), the SNR is 0 dB.

We can generate a sine wave corrupted by the zero-mean, unit-variance white noise with SNR = 0 dB using MATLAB® script `example2_20.m`.

Example 2.21

We can compute the N-point DFT of the signal $x(n)$ to obtain $X(k)$. The magnitude spectrum in the decibel scale can be calculated as $20\log_{10}|X(k)|$ for $k = 0, 1, \ldots, N/2$. Using the signal $x(n)$ generated by Example 2.20, the magnitude spectrum is computed and displayed in Figure 2.18 using the MATLAB® script `example2_21.m`. This magnitude spectrum shows that the power of white noise is uniformly distributed at frequency indices $k = 0, \ldots, 128$ (0 to π), while the power of the sine wave is concentrated at the frequency index $k = 26$ (0.2π).

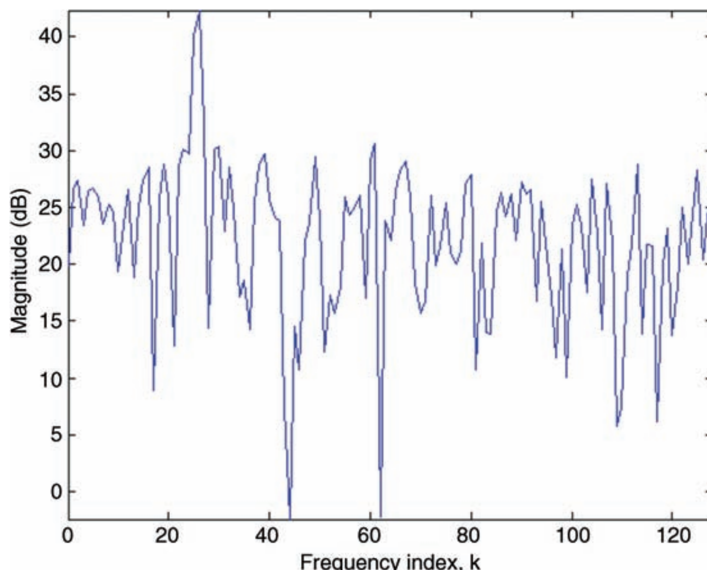


Figure 2.18 Spectrum of sine wave corrupted by white noise, SNR = 0 dB

2.4 Fixed-Point Representations and Quantization Effects

The basic element in digital hardware is the binary device that contains 1 bit of information. A register (or memory unit) containing B bits of information is called the B-bit word. There are several different methods for representing numbers and carrying out arithmetic operations. In this book, we focus on widely used fixed-point implementations [9–14].

2.4.1 Fixed-Point Formats

The most commonly used fixed-point representation of a fractional number x is illustrated in Figure 2.19. The wordlength is $B\ (=M+1)$ bits, that is, M magnitude bits and one sign bit. The most significant bit (MSB) is the sign bit, which represents the sign of the number as follows:

$$b_0 = \begin{cases} 0, & x \geq 0 \text{ (positive number)} \\ 1, & x < 0 \text{ (negative number)}. \end{cases} \tag{2.77}$$

The remaining M bits represent the magnitude of the number. The rightmost bit, b_M, is called the least significant bit (LSB), which represents the precision of the number.

As shown in Figure 2.19, the decimal value of a positive ($b_0 = 0$) binary fractional number x can be expressed as

$$\begin{aligned} (x)_{10} &= b_1 2^{-1} + b_2 2^{-2} + \cdots + b_M 2^{-M} \\ &= \sum_{m=1}^{M} b_m 2^{-m}. \end{aligned} \tag{2.78}$$

Example 2.22

The largest (positive) 16-bit fractional number in binary format is $x = 0111\ 1111\ 1111\ 1111$b (the letter "b" denotes the binary representation of the number). The decimal value of this number can be computed as

$$\begin{aligned} (x)_{10} &= \sum_{m=1}^{15} 2^{-m} = 2^{-1} + 2^{-2} + \cdots + 2^{-15} \\ &= 1 - 2^{-15} \approx 0.999\,969. \end{aligned}$$

The smallest non-zero positive number is $x = 0000\ 0000\ 0000\ 0001$b. The decimal value of this number is

$$(x)_{10} = 2^{-15} = 0.000\,030\,518.$$

Negative numbers ($b_0 = 1$) can be represented using three different formats: the sign magnitude, 1's complement, and 2's complement. Fixed-point digital signal processors usually use the 2's complement format to represent negative numbers because it allows the processor to perform addition and subtraction using the same hardware. With the 2's complement format, a negative number is obtained by complementing all the bits of the positive binary number and then adding one to the LSB.

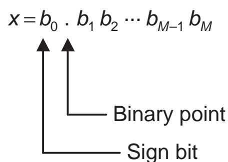


Figure 2.19 Fixed-point representation of binary fractional numbers

In general, the decimal value of a B-bit binary fractional number can be calculated as

$$(x)_{10} = -b_0 + \sum_{m=1}^{15} b_m 2^{-m}. \tag{2.79}$$

For example, the smallest (negative) 16-bit fractional number in binary format is $x = 1000$ 0000 0000 0000b. From (2.79), its decimal value is -1. Therefore, the range of fractional binary numbers is

$$-1 \leq x \leq \left(1 - 2^{-M}\right). \tag{2.80}$$

For a 16-bit fractional number x, the decimal value range is $-1 \leq x \leq 1 - 2^{-15}$ with a resolution of 2^{-15}.

Example 2.23

Table 2.2 lists 4-bit binary numbers representing both integers and fractional numbers (decimal values) using the 2's complement format.

Example 2.24

Sixteen-bit data x with the decimal value 0.625 can be initialized using the binary form $x = 0101$ 0000 0000 0000b, the hexadecimal form $x = $ 0x5000, or the decimal integer $x = 2^{14} + 2^{12} = 20\,480$.

As shown in Figure 2.19, the easiest way to convert the normalized 16-bit fractional number into the integer that can be used by the C55xx assembler is to move the binary point to the right by 15 bits (at the right of b_M). Since shifting the binary point right by 1 bit is equivalent to multiplying the fractional number by 2, this can be done by multiplying the decimal value by $2^{15} = 32\,768$. For example, $0.625*32\,768 = 20\,480$.

It is important to note that an implied binary point is used to represent the binary fractional number. The position of the binary point will affect the accuracy (dynamic range and precision) of the number. The binary point is a programmer's convention and the programmer has to keep track of the binary point when manipulating fractional numbers in assembly language programming.

Table 2.2 Four-bit binary numbers in the 2's complement format and their corresponding decimal values

Binary numbers	Integers (*sxxx.*)	Fractions (*s.xxx*)
0000	0	0.000
0001	1	0.125
0010	2	0.250
0011	3	0.375
0100	4	0.500
0101	5	0.675
0110	6	0.750
0111	7	0.875
1000	−8	−1.000
1001	−7	−0.875
1010	−6	−0.750
1011	−5	−0.675
1100	−4	−0.500
1101	−3	−0.375
1110	−2	−0.250
1111	−1	−0.125

Different notations can be used to represent different fractional formats. Similar to Figure 2.19, the more general fractional format Q*n.m* is illustrated in Figure 2.20, where $n+m=M=B-1$. There are n bits to the left of the binary point representing the integer portion, and m bits to the right representing fractional values. The most popular used fractional number representation shown in Figure 2.19 is called the Q0.15 format ($n=0$ and $m=15$), which is also simply called the Q15 format since there are 15 fractional bits. Note that the Q*n.m* format is represented in MATLAB® as [*B m*]. For example, the Q15 format is represented as [16 15].

Example 2.25

The decimal value of the 16-bit binary number $x=0100\ 1000\ 0001\ 1000$b depends on which Q format is used by the programmer. Using a larger n increases the dynamic range of the number with the cost of decreasing the precision, and vice versa. Three examples representing the 16-bit binary number $x=0100\ 1000\ 0001\ 1000$b are given as follows:

$$\begin{aligned} \text{Q0.15}, x &= 2^{-1}+2^{-4}+2^{-11}+2^{-12}=0.56323 \\ \text{Q2.13}, x &= 2^{1}+2^{-2}+2^{-9}+2^{-10}=2.25293 \\ \text{Q5.10}, x &= 2^{4}+2^{1}+2^{-6}+2^{-7}=18.02344. \end{aligned}$$

Fixed-point arithmetic is often used by DSP hardware for real-time processing because it offers fast operation and relatively economical implementation. Its drawbacks include small dynamic range and low resolution. These problems will be discussed in the following sections.

Figure 2.20 A general binary fractional number

2.4.2 *Quantization Errors*

As discussed in Section 2.4.1, numbers used by digital devices are represented by a finite number of bits. The errors caused by the difference between the desired and actual values are called the finite-wordlength (finite-precision, numerical, or quantization) effects. In general, finite-precision effects can be broadly categorized into the following classes:

1. Quantization errors:
   (a) Signal quantization.
   (b) Coefficient quantization.
2. Arithmetic errors:
   (a) Roundoff (or truncation).
   (b) Overflow.

2.4.3 *Signal Quantization*

As discussed in Chapter 1, the ADC converts an analog signal $x(t)$ into a digital signal $x(n)$. The input signal is first sampled to obtain the discrete-time signal $x(nT)$ with infinite precision. Each $x(nT)$ value is then encoded (quantized) using B-bit wordlength to obtain the digital signal $x(n)$. We assume that the signal $x(n)$ is interpreted as the Q15 fractional number shown in Figure 2.19 such that $-1 \leq x(n) < 1$. Thus, the dynamic range of fractional numbers is 2. Since the quantizer employs B bits, the number of quantization levels available is 2^B. The spacing between two successive quantization levels is

$$\Delta = \frac{2}{2^B} = 2^{-B+1} = 2^{-M}, \tag{2.81}$$

which is called the quantization step (interval, width, or resolution). For example, the output of a 4-bit converter with quantization interval $\Delta = 2^{-3} = 0.125$ is summarized in Table 2.2.

This book uses rounding (instead of truncation) for quantization. The input value $x(nT)$ is rounded to the nearest level as illustrated in Figure 2.21 for a 3-bit ADC. Assuming there is a line equally between two adjacent quantization levels, the signal value above this line will be assigned to the higher quantization level, while the signal value below this line is assigned to the lower level. For example, the discrete-time signal $x(T)$ in Figure 2.21 is rounded to 010b, since the real value is below the middle line between 010b and 011b, while $x(2T)$ is rounded to 011b since the value is above the middle line.

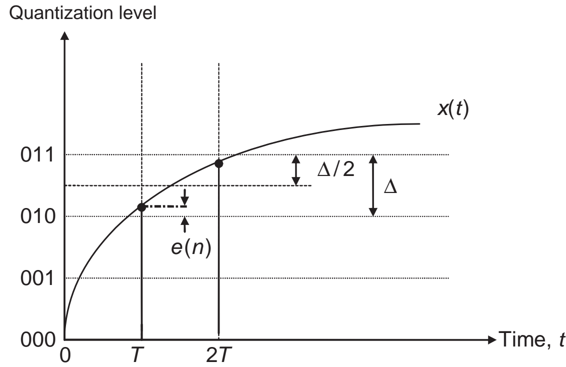


Figure 2.21 Quantization process related to a 3-bit ADC

The quantization error (or noise), $e(n)$, is the difference between the discrete-time signal $x(nT)$ and the quantized digital signal $x(n)$, expressed as

$$e(n) = x(n) - x(nT). \tag{2.82}$$

Figure 2.21 clearly shows that

$$|e(n)| \leq \frac{\Delta}{2} = 2^{-B}. \tag{2.83}$$

Thus, the quantization noise generated by an ADC depends on the quantization step determined by the wordlength B. The use of more bits results in a smaller quantization step (or finer resolution), thus yielding lower quantization noise.

From (2.82), we can express the ADC output as the sum of the quantizer input $x(nT)$ and the error $e(n)$. That is,

$$x(n) = Q[x(nT)] = x(nT) + e(n), \tag{2.84}$$

where $Q[\cdot]$ denotes the quantization operation. Therefore, the nonlinear operation of the quantizer is modeled as the linear process that introduces an additive noise $e(n)$ into the digital signal $x(n)$.

For an arbitrary signal with fine quantization (B is large), the quantization error $e(n)$ is assumed to be uncorrelated with $x(n)$, and is a random noise that is uniformly distributed in the interval $[-\Delta/2, \Delta/2]$. From (2.71), we have

$$E[e(n)] = \frac{-\Delta/2 + \Delta/2}{2} = 0. \tag{2.85}$$

Thus, the quantization noise $e(n)$ has zero mean. From (2.73), the variance is

$$\sigma_e^2 = \frac{\Delta^2}{12} = \frac{2^{-2B}}{3}. \tag{2.86}$$

Therefore, larger wordlength results in smaller input quantization error.

The signal-to-quantization-noise ratio (SQNR) can be expressed as

$$\mathrm{SQNR} = \frac{\sigma_x^2}{\sigma_e^2} = 3 \cdot 2^{2B} \sigma_x^2, \tag{2.87}$$

where σ_x^2 denotes the variance of the signal $x(n)$. Usually, the SQNR is expressed in dB as

$$\begin{aligned} \mathrm{SQNR} &= 10 \log_{10}\left(\sigma_x^2/\sigma_e^2\right) = 10 \log_{10}\left(3 \cdot 2^{2B}\sigma_x^2\right) \\ &= 10 \log_{10} 3 + 20B \log_{10} 2 + 10 \log_{10} \sigma_x^2 \\ &= 4.77 + 6.02B + 10 \log_{10} \sigma_x^2. \end{aligned} \tag{2.88}$$

This equation indicates that for each additional bit used in the ADC, the converter provides about 6 dB gain. When using a 16-bit ADC ($B = 16$), the maximum SQNR is about 98.1 dB if the input signal is a sine wave. This is because the sine wave has maximum amplitude 1.0, so $10 \log_{10}(\sigma_x^2) = 10 \log_{10}(1/2) = -3$, and (2.88) becomes $4.77 + 6.02 \times 16 - 3.0 = 98.09$. Another important feature of (2.88) is that the SQNR is proportional to the variance of the signal σ_x^2. Therefore, we want to keep the power of the signal as large as possible. This is an important consideration when we discuss scaling issues in Section 2.5.

Example 2.26

Signal quantization effects may be subjectively evaluated by observing and listening to quantized speech. The sampling rate of speech file `timit1.asc` is $f_s = 8$ kHz with $B = 16$. This speech file can be viewed and played using the MATLAB® script (`example2_26.m`):

```
load timit1.asc;
plot(timit1);
soundsc(timit1, 8000, 16);
```

where the MATLAB® function `soundsc` automatically scales and plays the vector as sound.

We can simulate the quantization of data with 8-bit wordlength by

```
qx = round(timit1/256);
```

where the function `round` rounds the real number to the nearest integer. We then evaluate the quantization effects by

```
plot(qx);
soundsc(qx, 8000, 16);
```

By comparing the graph and sound of `timit1` and `qx`, the signal quantization effects may be understood.

2.4.4 Coefficient Quantization

The digital filter coefficients, b_l and a_m, computed by a filter design package such as MATLAB®, are usually represented using the floating-point format. When implementing a digital filter, these filter coefficients must be quantized for the given fixed-point processor. Therefore, the performance of the fixed-point digital filter will be different from its original design specifications.

The coefficient quantization effects become more significant when tighter specifications are used, especially for IIR filters. Coefficient quantization can cause serious problems if the poles of the IIR filters are too close to the unit circle. This is because these poles may move outside the unit circle due to coefficient quantization, resulting in an unstable filter. Such undesirable effects are far more pronounced in high-order systems.

Coefficient quantization is also affected by the digital filter structure. For example, the direct-form implementation of IIR filters is more sensitive to coefficient quantization than the cascade structure (to be introduced in Chapter 4) which consists of multiple sections of second- (or one first-)order IIR filters.

2.4.5 Roundoff Noise

Consider the example of computing the product $y(n) = \alpha x(n)$ in DSP systems. Assume α and $x(n)$ are B-bit numbers, and multiplication yields $2B$-bit product $y(n)$. In most applications, this product may have to be stored in memory or output as a B-bit word. The $2B$-bit product can be either truncated or rounded to B bits. Since truncation causes an undesired bias effect, we should restrict our attention to the rounding.

Example 2.27

In C programming, rounding a real number to an integer number can be implemented by adding 0.5 to the real number and then truncating the fractional part. The following C statement

```
y = (short)(x + 0.5);
```

rounds the real number `x` to the nearest integer `y`. As shown in Example 2.26, MATLAB® provides the function `round` for rounding a real number.

The process of rounding a $2B$-bit product to B bits is similar to that of quantizing a discrete-time signal using a B-bit quantizer. Similar to (2.84), the nonlinear roundoff operation can be modeled as a linear process expressed as

$$y(n) = Q[\alpha x(n)] = \alpha x(n) + e(n), \tag{2.89}$$

where $\alpha x(n)$ is the $2B$-bit product and $e(n)$ is the roundoff noise due to rounding the $2B$-bit product to B bits. The roundoff noise is a uniformly distributed random variable as defined in (2.83); thus, it has zero mean, and its power is defined in (2.86).

It is important to note that most commercially available fixed-point digital signal processors such as the TMS320C55xx have double-precision accumulator(s). As long as

the program is carefully written, it is possible to ensure that rounding occurs only at the final stage of the calculation. For example, consider the computation of FIR filtering given in (2.14). We can keep the sum of all temporary products, $b_l x(n-l)$, in the double-precision accumulator. Rounding is only performed when the final sum is saved to memory with B-bit wordlength.

2.4.6 Fixed-Point Toolbox

The MATLAB® *Fixed-Point Toolbox* [15] provides fixed-point data types and arithmetic for developing fixed-point DSP algorithms. The toolbox provides the `quantizer` function for constructing a quantizer object. For example, we can use the syntax

```
q = quantizer
```

to create the quantizer object `q` with properties set to the following default values:

```
mode = 'fixed';
roundmode = 'floor';
overflowmode = 'saturate';
format = [16 15];
```

Note that `[16 15]` is equivalent to the Q15 format.

After we have constructed the quantizer object, we can apply it to data using the `quantize` function with the following syntax:

```
y = quantize(q, x)
```

The command `y = quantize(q, x)` uses the quantizer object `q` to quantize `x`. When `x` is a numeric array, each element of the `x` will be quantized.

Example 2.28

Similar to Example 2.19, we generate a zero-mean white noise using MATLAB® function `rand`, which uses the double-precision, floating-point format. We then construct two quantizer objects and quantize the white noise to Q15 (16-bit) and Q3 (4-bit) formats. We plot the quantized noise in the Q15 and Q3 formats and the difference between these two formats in Figure 2.22 using the following MATLAB® script (`example2_28.m`):

```
N = 16;
n = [0:N-1];
xn = sqrt(3)*(rand(1,N)-0.5);  % Generate zero-mean white noise
q15 = quantizer('fixed', 'convergent', 'wrap', [16 15]); % Q15
q3 = quantizer('fixed', 'convergent', 'wrap', [4 3]);    % Q3
```

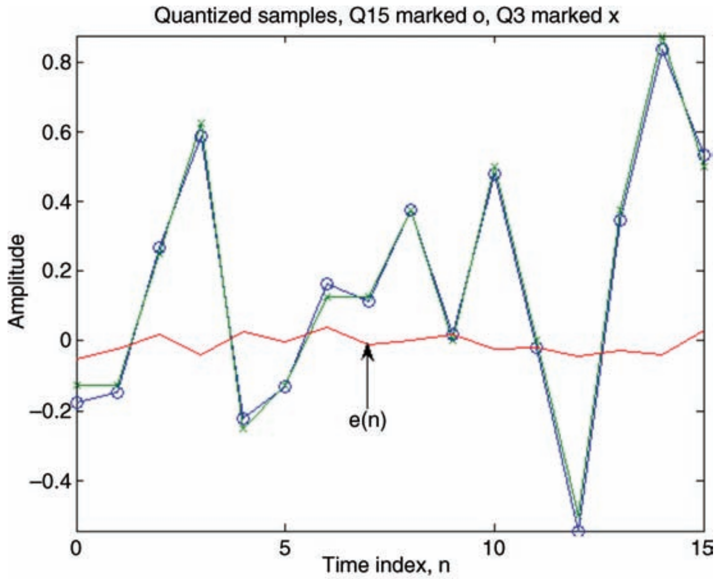


Figure 2.22 Quantization using Q15 and Q3 formats and the difference $e(n)$

```
y15 = quantize(q15,xn);        % Quantization using Q15 format
y3 = quantize(q3,xn);          % Quantization using Q3 format
en = y15-y3,                   % Difference between Q15 and Q3
plot(n,y15,'-o',n,y3,'-x',n,en);
```

The MATLAB® *Fixed-Point Toolbox* also provides several radix conversion functions, which are summarized in Table 2.3. For example,

```
y = num2int(q,x)
```

uses `q.format` to convert a number `x` to an integer `y`.

Example 2.29

In order to test some DSP algorithms using fixed-point C programs, we may need to generate specific data files for simulations. As shown in Example 2.28, we can use MATLAB® to generate a testing signal and construct a quantizer object. In order to save the Q15 data in integer format, we use the function `num2int` in the following MATLAB® script (`example2_29.m`):

```
N=16; n=[0:N-1];
xn = sqrt(3)*(rand(1,N)-0.5); % Generate zero-mean white noise
q15 = quantizer('fixed', 'convergent', 'wrap', [16 15]); % Q15
Q15int = num2int(q15,xn);
```

Table 2.3 List of radix conversion functions using quantizer object

Function	Description
`bin2num`	Convert 2's complement binary string to a number
`hex2num`	Convert hexadecimal string to a number
`num2bin`	Convert a number to binary string
`num2hex`	Convert a number to its hexadecimal equivalent
`num2int`	Convert a number to a signed integer

2.5 Overflow and Solutions

Assuming that the signals and filter coefficients have been properly normalized in the range of -1 and 1 for fixed-point arithmetic, the sum of two B-bit numbers may fall outside the range of -1 and 1. The term overflow means that the result of the arithmetic operation exceeds the capacity of the register used to hold the result. When using a fixed-point processor, the range of numbers must be carefully examined and adjusted in order to avoid overflow. This may be achieved by using different Q$n.m$ formats with desired dynamic ranges. For example, a larger dynamic range can be obtained by using a larger n, with the cost of reducing m (decreasing resolution).

Example 2.30

Assume 4-bit fixed-point hardware uses the fractional 2's complement format (see Table 2.2). If $x_1 = 0.875$ (0111b) and $x_2 = 0.125$ (0001b), the binary sum of $x_1 + x_2$ is 1000b. The decimal value of this signed binary number is -1, not the correct answer $+1$. That is, when the addition result exceeds the dynamic range of the register, overflow occurs and an unacceptable error is produced.

Similarly, if $x_3 = -0.5$ (1100b) and $x_4 = 0.625$(0101b), $x_3 - x_4 = 0110$b, which is $+0.875$, not the correct answer -1.125. Therefore, subtraction may also result in underflow.

For the FIR filtering defined in (2.14), overflow will result in severe distortion of the output $y(n)$. For the IIR filter defined in (2.22), the overflow effect is much more serious because the errors will be fed back. The problem of overflow may be eliminated using saturation arithmetic and proper scaling (or constraining) signals at each node within the filter to maintain the magnitude of the signal. The MATLAB® *DSP System Toolbox* provides the function `scale(hd)` to scale the second-order IIR filter `hd` to reduce possible overflows when the filter is operated using fixed-point arithmetic.

2.5.1 *Saturation Arithmetic*

Most digital signal processors have mechanisms to protect against overflow and automatically indicate the overflow if it occurs. For example, saturation arithmetic prevents overflow by clipping the results to a maximum value. Saturation logic is illustrated in Figure 2.23 and can

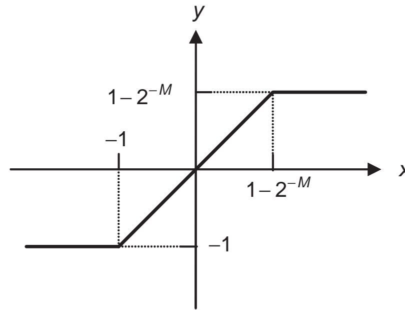


Figure 2.23 Characteristics of saturation arithmetic

be expressed as

$$y = \begin{cases} 1-2^{-M}, & \text{if } x \geq 1-2^{-M} \\ x, & \text{if } -1 \leq x < 1 \\ -1, & \text{if } x < -1, \end{cases} \tag{2.90}$$

where x is the original addition result and y is the saturated adder output. If the adder is in saturation mode, the undesired overflow can be avoided since the 32-bit accumulator fills to its maximum (or minimum) value, but does not roll over. Similar to Example 2.28, when 4-bit hardware with saturation arithmetic is used, the addition result of $x_1 + x_2$ is 0111b, or 0.875 in decimal value. Compared to the correct answer 1, there is an error of 0.125. This result is much better than the hardware without saturation arithmetic.

Saturation arithmetic has a similar effect of "clipping" the desired waveform. This is the nonlinear operation that will introduce undesired nonlinear components into the saturated signal. Therefore, saturation arithmetic can be used to guarantee that overflow will not occur. Nevertheless, it should not be the only solution for solving overflow problems.

2.5.2 Overflow Handling

As mentioned earlier, the C55xx supports saturation logic to prevent overflow. The logic is enabled when the overflow mode bit (SATD) in status register ST1 is set (SATD = 1). The C55xx also provides overflow flags that indicate whether or not the arithmetic operation has overflowed. A flag will remain set until a reset is performed or when the status bit clear instruction is executed. If a conditional instruction (such as a branch, return, call, or conditional execution) that tests overflow status is executed, the overflow flag will be cleared.

2.5.3 Scaling of Signals

The most effective technique in preventing overflow is by scaling down the signal. For example, consider the simple FIR filter illustrated in Figure 2.24 without the scaling factor β (or $\beta = 1$). Let $x(n) = 0.8$ and $x(n-1) = 0.6$; then the filter output $y(n) = 1.2$. When this filter is implemented on a fixed-point processor using the Q15 format without saturation arithmetic, undesired overflow occurs. As illustrated in Figure 2.24, the scaling factor, $\beta < 1$, can be used to scale down the input signal. For example, when $\beta = 0.5$, we have $x(n) = 0.4$ and $x(n-1) = 0.3$, and the result $y(n) = 0.6$ without overflow.

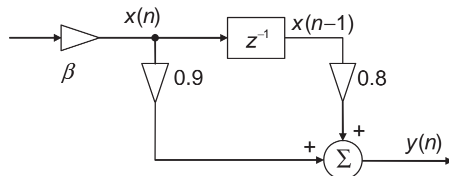


Figure 2.24 Block diagram of simple FIR filter with scaling factor β

If the signal $x(n)$ is scaled by β, the resulting signal variance becomes $\beta^2\sigma_x^2$. Thus, the SQNR in dB given in (2.88) changes to

$$\begin{aligned}\text{SQNR} &= 10\log_{10}\left(\beta^2\sigma_x^2/\sigma_e^2\right) \\ &= 4.77 + 6.02B + 10\log_{10}\sigma_x^2 + 20\log_{10}\beta.\end{aligned} \tag{2.91}$$

Since we perform fractional arithmetic, $\beta < 1$ is used to scale down the input signal. The last term $20\log_{10}\beta$ has a negative value. Thus, scaling down the signal reduces the SQNR. For example, when $\beta = 0.5$, $20\log_{10}\beta = -6.02$ dB, thus reducing the SQNR of the input signal by about 6 dB. This is equivalent to losing 1 bit in representing the signal.

2.5.4 Guard Bits

The C55xx provides four 40-bit accumulators, each consisting of a 32-bit accumulator with additional eight guard bits. The guard bits are used to prevent overflow from iterative computations such as the FIR filtering defined in (2.14).

Because of the potential overflow problem from fixed-point implementation, special care must be taken to ensure the proper dynamic ranges are maintained throughout the implementation. This usually demands greater coding and testing efforts. In general, the optimum solution is a combination of scaling factors, guard bits, and saturation arithmetic. In order to maintain high SQNR, the scaling factors (less than one) are set as large as possible such that there may only be few occasional overflows which can be avoided by using guard bits and saturation arithmetic.

2.6 Experiments and Program Examples

This section presents hands-on experiments to demonstrate DSP programming using the CCS and C5505 eZdsp.

2.6.1 Overflow and Saturation Arithmetic

As discussed in the previous section, overflow may occur when processors perform fixed-point accumulation such as FIR filtering. Overflow can also occur when storing data from the accumulator to memory because C55xx accumulators (AC0–AC3) have 40 bits while the data memory is usually defined as a 16-bit word. In this experiment, we use the assembly routine `ovf_sat.asm` to demonstrate the program execution results with and without overflow protection. A concise introduction to the C55xx architecture and its assembly programming is given in Appendix C. Table 2.4 lists a portion of the assembly code used for the experiment.

Table 2.4 Program for the experiment of overflow and saturation

```
    .def  _ovftest
    .def  _buff,_buff1

    .bss  _buff,(0x100)
    .bss  _buff1,(0x100)
;
;   Code start
;
_ovftest
    bclr  SATD                ; Clear saturation bit if set
    xcc   start,T0! = #0      ; If T0! = 0, set saturation bit
    bset  SATD

start
    mov   #0,AC0
    amov  #_buff,XAR2         ; Set buffer pointer
    rpt   #0x100-1            ; Clear buffer
    mov   AC0,*AR2+

    amov  #_buff1,XAR2        ; Set buffer pointer
    rpt   #0x100-1            ; Clear buffer1
    mov   AC0,*AR2+

    mov   #0x80-1,BRC0        ; Initialize loop counts for addition
    amov  #_buff + 0x80,XAR2  ; Initialize buffer pointer

    rptblocal add_loop_end-1
    add   #0x140<<#16,AC0     ; Use upper AC0 as a ramp up counter
    mov   hi(AC0),*AR2+       ; Save the counter to buffer
add_loop_end

    mov   #0x80-1,BRC0        ; Initialize loop counts for subtraction
    mov   #0,AC0
    amov  #_buff + 0x7f,XAR2  ; Initialize buffer pointer

    rptblocal sub_loop_end-1
    sub   #0x140<<#16,AC0     ; Use upper AC0 as a ramp down counter
    mov   hi(AC0),*AR2-       ; Save the counter to buffer
sub_loop_end

    mov   #0x100-1,BRC0       ; Initialize loop counts for sinewave
    amov  #_buff1,XAR2        ; Initialize buffer pointer
    mov   mmap(@AR0),BSA01    ; Initialize base register
    mov   #40,BK03            ; Set buffer to size 40
    mov   #20,AR0             ; Start with an offset of 20 samples
    bset  AR0LC               ; Active circular buffer

    rptblocal sine_loop_end-1
    mov   *ar0 + <<#16,AC0    ; Get sine value into high AC0
```

Table 2.4 (*Continued*)

```
    sfts  AC0,#9            ; Scale the sine value
    mov   hi(AC0),*AR2+     ; Save scaled value
sine_loop_end

    mov   #0,T0             ; Return 0 if no overflow
    xcc   set_ovf_flag,overflow(AC0)
    mov   #1,T0             ; Return 1 if overflow detected
set_ovf_flag

    bclr  AR0LC             ; Reset circular buffer bit
    bclr  SATD              ; Reset saturation bit
    ret
    .end
```

In the assembly program, the following code repeatedly adds a constant of value 0x140 to the accumulator AC0:

```
   rptblocal add_loop_end-1
   add       #0x140<<#16,AC0
   mov       hi(AC0),*AR2+
add_loop_end
```

The updated value is stored in the memory location pointed at by AR2. The contents of AC0 will grow larger and larger, and eventually the accumulator AC0 will overflow. Without protection, the positive number in AC0 suddenly becomes negative when the overflow occurs. However, if the C55xx saturation mode is set, the overflowed positive number will be limited to 0x7FFFFFFF. The second half of the code stores the left-shifted sine wave values to data memory locations. Without saturation protection, this shift will cause some of the large values to overflow after the shift.

In the program, the following segment of code sets up and uses the circular addressing mode (see Appendix C for details):

```
mov    #sineTable,BSA01     ; Initialize base register
mov    #40,BK03             ; Set buffer size to 40
mov    #20,AR0              ; Start with offset of 20 samples
bset   AR0LC                ; Activate circular buffer
```

The first instruction sets up the circular buffer base register BSA01 because AR0 is used as the circular buffer pointer. The second instruction initializes the size of the circular buffer. The third instruction initializes the offset from the base for use as the starting point of the circular buffer. In this case, the offset is set to 20 from the base of `sineTable[]`. The last instruction enables AR0 as the circular pointer. Table 2.5 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Copy the entire project from the companion software package to the working folder, import the CCS project, and build and load the program to the C5505 eZdsp.

Table 2.5 File listing for the experiment Exp2.1

Files	Description
`overflowTest.c`	Program for showing overflow experiment
`ovf_sat.asm`	Assembly function showing overflow
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file

2. Run the program, use the graphic tool in conjunction with breakpoints to plot and observe the results in `buff` and `buff1` while running the program.
3. Turn off overflow protection and repeat the experiment to observe overflow without the protection.

2.6.2 Function Approximations

This experiment uses polynomial approximation of sinusoidal functions to show a typical DSP algorithm design and implementation process. The DSP algorithm development usually starts with using MATLAB® or floating-point C for computer simulation, converts to fixed-point C, optimizes the code to improve its efficiency, and uses assembly functions if necessary.

The cosine and sine functions can be expanded as infinite power (Taylor) series as follows:

$$\cos(\theta) = 1 - \frac{1}{2!}\theta^2 + \frac{1}{4!}\theta^4 - \frac{1}{6!}\theta^6 + \ldots, \tag{2.92a}$$

$$\sin(\theta) = \theta - \frac{1}{3!}\theta^3 + \frac{1}{5!}\theta^5 - \frac{1}{7!}\theta^7 + \ldots, \tag{2.92b}$$

where θ is the angle in radians and "!" represents the factorial operation. The accuracy of the approximation depends on the number of terms used in the series. Usually more terms are needed for larger values of θ. However, only a limited number of terms can be used in real-time DSP applications.

A. Implementation Using Floating-Point C

In this experiment, we implement the cosine function approximation in (2.92a) using the C program listed in Table 2.6. In the function `fCos1()`, 12 multiplications are required. The C55xx compiler has a built-in runtime support library for floating-point arithmetic operations. These floating-point functions are inefficient for real-time applications. For example, the program `fCos1()` requires several thousand clock cycles to compute one sine value.

We can improve the computational efficiency by reducing the multiplications from 12 to 4. The modified program is listed in Table 2.7. This improved program reduces the clock cycles by about half. To further improve the efficiency, we can use the fixed-point C and assembly programs. The files used for the experiment are listed in Table 2.8.

Procedures of the experiment are listed as follows:

1. Copy the entire project from the companion software package to the working folder, import the CCS floating-point project from the folder `..\Exp2.2\funcAppro`, and build and load the program to the C5505 eZdsp.
2. Run the program and verify the results.

Table 2.6 Floating-point C program for cosine approximation

```
// Coefficients for cosine function approximation
double fcosCoef[4] = {
     1.0, -(1.0/2.0), (1.0/(2.0*3.0*4.0)), -(1.0/
(2.0*3.0*4.0*5.0*6.0))
};

// Direct implementation of function approximation
double fCos1(double x)
{
    double cosine;

    cosine  = fcosCoef[0];
    cosine += fcosCoef[1]*x*x;
    cosine += fcosCoef[2]*x*x*x*x;
    cosine += fcosCoef[3]*x*x*x*x*x*x;
    return(cosine);
}
```

Table 2.7 Efficient floating-point C program for cosine approximation

```
// More efficient implementation of function approximation
double fCos2(double x)
{
   double cosine,x2;

    x2 = x * x;
    cosine = fcosCoef[3] * x2;
    cosine = (cosine + fcosCoef[2]) * x2;
    cosine = (cosine + fcosCoef[1]) * x2;
    cosine = cosine + fcosCoef[0];
    return(cosine);
}
```

Table 2.8 File listing for the experiment Exp2.2A

Files	Description
`fcosTest.c`	Floating-point C program testing function approximation
`c5505.cmd`	Linker command file

3. Use the CCS Clock Tool (under **Run→Clock**) to measure the cycles needed for the floating-point C implementation of the functions `fCos1()` and `fCos2()`.

B. Implementation Using Fixed-Point C
The fixed-point C implementation of the cosine function approximation using the Q15 format is listed in Table 2.9. This fixed-point C program significantly improves the runtime efficiency. Table 2.10 lists the files used for the experiment.

Table 2.9 Fixed-point C program for function approximation

```
#define UNITQ15 0x7FFF

// Coefficients for cosine function approximation
short icosCoef[4] = {
  (short)(UNITQ15),
  (short)(-(UNITQ15/2.0)),
  (short)(UNITQ15/(2.0*3.0*4.0)),
  (short)(-(UNITQ15/(2.0*3.0*4.0*5.0*6.0)))
}

// Fixed-point implementation of function approximation
short iCos1(short x)
{
   long cosine,z;
   short x2;
   z = (long)x * x;
   x2 = (short)(z>>15);                    // x2 has x(Q14)*x(Q14)
   cosine = (long)icosCoef[3] * x2;
   cosine = cosine >> 13;                  // Scale back to Q15
   cosine = (cosine + (long)icosCoef[2]) * x2;
   cosine = cosine >> 13;                  // Scale back to Q15
   cosine = (cosine + (long)icosCoef[1]) * x2;
   cosine = cosine >> 13;                  // Scale back to Q15
   cosine = cosine + icosCoef[0];
   return((short)cosine);
}
```

Procedures of the experiment are listed as follows:

1. Import the CCS fixed-point project from the folder `..\Exp2.2\funcAppro`, and build and load the program to the C5505 eZdsp.
2. Run the program and verify the results.
3. Use the Clock Tool to profile the cycles needed for the fixed-point C `iCos1()` implementation. Compare the result to the floating-point C functions `fCos1()` and `fCos2()` given in previous experiments.

C. Implementation Using C55xx Assembly Program
   In many real-world applications, the DSP algorithms are implemented using assembly language. The assembly program can be verified by comparing its output against the

Table 2.10 File listing for the experiment Exp2.2B

Files	Description
icosTest.c	Fixed-point C program testing function approximation
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file

Table 2.11 C55xx assembly program for cosine function approximation

```
   .data
_icosCoef   ; [1 (-1/2!) (1/4!) (-1/6!)]
   .word    32767,-16383,1365,-45

   .sect ".text"
   .def   _cosine

_cosine:
    amov #(_icosCoef + 3),XAR3 ; ptr = &icosCoef[3]
    amov #AR1,AR2              ; AR1 is used as temp. register
||  mov  T0,HI(AC0)
    sqr  AC0                   ; AC0 = (long)T0 * T0
    sfts AC0,#-15              ; T0  = (short)(AC0>>15)
    mov  AC0,T0
    mpym *AR3-,T0,AC0          ; AC0 = (long)T0 * *ptr--
    sfts AC0,#-13              ; AC0 = AC0 >> 13
    add  *AR3-,AC0,AR1         ; AC0 = (short)(AC0 + *ptr--) * (long)T0
    mpym *AR2,T0,AC0
    sfts AC0,#-13              ; AC0 = AC0 >> 13
    add  *AR3-,AC0,AR1         ; AC0 = (short)(AC0 + *ptr--) * (long)T0
    mpym *AR2,T0,AC0
    sfts AC0,#-13              ; AC0 = AC0 >> 13
||  mov  *AR3,T0
    add  AC0,T0                ; AC0 = AC0 + *ptr
    ret                        ; Return((short)AC0)
    .end
```

output of the fixed-point C code. This experiment uses the assembly program shown in Table 2.11 for computing the cosine function. The files used for the experiment are listed in Table 2.12.

Procedures of the experiment are listed as follows:

1. Import the assembly project from the folder `..\Exp2.2\funcAppro`, and build and load the program to the C5505 eZdsp.
2. Run the program and verify the results by comparing to the results obtained in the previous fixed-point C experiment.

Table 2.12 File listing for the experiment Exp2.2C

Files	Description
`c55xxASMTest.c`	Program for testing function approximation
`cos.asm`	Assembly routine for cosine approximation
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file

3. Profile the cycles needed for the assembly implementation of the cosine approximating function `cosine()`. Record the measured cycles from the floating-point C, fixed-point C, and assembly programs and fill in the following table for comparison:

Arithmetic	Function	Implementation details	Profile (cycles/call)
Floating-point C	`fCos1()`	Direct implementation, 12 multiplications	
	`fCos2()`	Reduced multiplications, 4 multiplications	
Fixed-point C	`iCos1()`	Using fixed-point arithmetic	
	`iCos()`	Simulate assembly instruction	
Assembly language	`cosine()`	Hand-code assembly routine	

D. Practical Applications

Since the input arguments to the cosine function are in the range of $-\pi$ to π, we must map the data values from the range of $-\pi$ and π to the linear 16-bit data variables as shown in Figure 2.25. Using 16-bit wordlength, we map 0 to 0x0000, π to 0x7FFF, and $-\pi$ to 0x8000 to represent the radius arguments. Therefore, the function approximation given in (2.92) is no longer the best choice, and different function approximations should be considered for practical applications.

Using the Chebyshev approximation, $\cos(\theta)$ and $\sin(\theta)$ can be computed as

$$\cos(\theta) = 1 - 0.001922\theta - 4.90014746\theta^2 - 0.264892\theta^3 + 5.04541\theta^4 + 1.800293\theta^5, \quad (2.93a)$$

$$\sin(\theta) = 3.140625\theta + 0.020\,26367\theta^2 - 5.325196\theta^3 + 0.54467888\theta^4 + 1.800293\theta^5, \quad (2.93b)$$

where the value of θ is defined in the first quadrant, $0 \le \theta < \pi/2$. For other quadrants, the following properties can be used to transfer it from the first quadrant:

$$\sin(180^\circ - \theta) = \sin(\theta), \quad \cos(180^\circ - \theta) = -\cos(\theta) \quad (2.94)$$

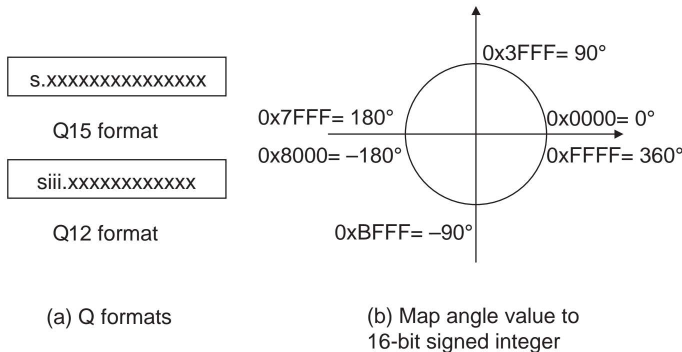


Figure 2.25 Scaled fixed-point number representations

$$\sin(-180^\circ + \theta) = -\sin(\theta), \quad \cos(-180^\circ + \theta) = -\cos(\theta) \tag{2.95}$$

and

$$\sin(-\theta) = -\sin(\theta), \quad \cos(-\theta) = \cos(\theta). \tag{2.96}$$

The C55xx assembly routine (listed in Table 2.13) synthesizes the sine and cosine functions, which can be used to calculate the angle θ from -180° to 180°.

Since the absolute value of the largest coefficient given in this experiment is 5.325 196, we must scale the coefficients or use a different Q format as shown in Figure 2.20. This can be achieved by using the Q3.12 format, which has one sign bit, three integer bits, and twelve fraction bits to cover the range $(-8, 8)$, as illustrated in Figure 2.25(a). In the example, we use the Q3.12 format for all coefficients, and map the angles in the range of $-\pi \leq \theta \leq \pi$ to signed 16-bit numbers ($0x8000 \leq x \leq 0x7FFF$) as shown in Figure 2.25(b).

When the assembly subroutine `sine_cos` is called, the 16-bit mapped angle (function argument) is passed to the assembly routine using register T0. The quadrant information is tested and stored in TC1 and TC2. If TC1 (bit 14) is set, the angle is located in either quadrant II or IV. The program uses 2's complement to convert the angle to the first or third quadrant. The program also masks out the sign bit to calculate the third quadrant angle in the first quadrant, and the negation changes the fourth quadrant angle to the first quadrant. Therefore, the angle to be calculated is always located in the first quadrant. Because the program uses the Q3.12 format for coefficients, the computed result needs to be left-shifted 3 bits to scale back to the Q15 format. The files used for the experiment are listed in Table 2.14.

Procedures of the experiment are listed as follows:

1. Import the CCS project, and rebuild and load the program to the eZdsp.
2. Calculate the angles in the following table, and run the experiment to obtain the approximation results and compare the differences:

θ	30°	45°	60°	90°	120°	135°	150°	180°
$\cos(\theta)$								
$\sin(\theta)$								
θ	-150°	-135°	-120°	-90°	-60°	-45°	-30°	0°
$\cos(\theta)$								
$\sin(\theta)$								

3. Modify the experiment to implement the following square root approximation equation:

$$\sqrt{x} = 0.2075806 + 1.454895x - 1.34491x^2 + 1.106812x^3 - 0.536499x^4 + 0.1121216x^5.$$

This equation approximates the input variable within the range of $0.5 \leq x \leq 1$. Based on the x values listed in the following table, calculate $\sqrt{x}$:

x	0.5	0.6	0.7	0.8	0.9
$\sqrt{x}$					

Table 2.13 C55xx assembly program for approximation of sine and cosine functions

```
 .def _sine_cos
;
; Approximation coefficients in Q12 format
;
 .data
coeff ; Sine approximation coefficients
 .word 0x3240 ; c1 =  3.140625
 .word 0x0053 ; c2 =  0.02026367
 .word 0xaacc ; c3 = -5.325196
 .word 0x08b7 ; c4 =  0.54467780
 .word 0x1cce ; c5 =  1.80029300
 ; Cosine approximation coefficients
 .word 0x1000 ; d0 =  1.0000
 .word 0xfff8 ; d1 = -0.001922133
 .word 0xb199 ; d2 = -4.90014738
 .word 0xfbc3 ; d3 = -0.2648921
 .word 0x50ba ; d4 =  5.0454103
 .word 0xe332 ; d5 = -1.800293
;
; Function starts
;
 .text
_sine_cos
 amov  #14,AR2
 btstp AR2,T0           ; Test bit 15 and 14
 nop
;
; Start cos(x)
;
  amov  #coeff+10,XAR2 ; Pointer to the end of coefficients
  xcc   _neg_x,TC1
  neg   T0              ; Negate if bit 14 is set
_neg_x
   and   #0x7fff,T0     ; Mask out sign bit
   mov   *AR2-<<#16,AC0; AC0 = d5
||  bset SATD           ; Set saturation bit
   mov   *AR2-<<#16,AC1; AC1 = d4
||  bset FRCT           ; Set up fractional bit
   mac   AC0,T0,AC1     ; AC1 = (d5*x + d4)
||  mov  *AR2-<<#16,AC0; AC0 = d3
   mac   AC1,T0,AC0     ; AC0 = (d5*x^2 + d4*x + d3)
||  mov  *AR2-<<#16,AC1; AC1 = d2
   mac   AC0,T0,AC1     ; AC1 = (d5*x^3 + d4*x^2 + d3*x + d2)
||  mov  *AR2-<<#16,AC0; AC0 = d1
   mac   AC1,T0,AC0     ; AC0 = (d5*x^4 + d4*x^3 + d3*x^2 + d2*x + d1)
||  mov  *AR2-<<#16,AC1; AC1 = d0
   macr  AC0,T0,AC1     ; AC1 = (d5*x^4 + d4*x^3 + d3*x^2 + d2*x + d1)
                        ; *x + d0
```

Table 2.13 (*Continued*)

```
|| xcc  _neg_result1,TC2
   neg  AC1

_neg_result1
    mov  *AR2-<<#16,AC0; AC0 = c5
||  xcc  _neg_result2,TC1
    neg  AC1
_neg_result2
    mov  hi(saturate(AC1<<#3)),*AR0+ ; Return cos(x) in Q15
;
; Start sin(x) computation
;
   mov  *AR2-<<#16,AC1; AC1 = c4
   mac  AC0,T0,AC1     ; AC1 = (c5*x + c4)
|| mov  *AR2-<<#16,AC0; AC0 = c3
   mac  AC1,T0,AC0     ; AC0 = (c5*x^2 + c4*x + c3)
|| mov  *AR2-<<#16,AC1; AC1 = c2
   mac  AC0,T0,AC1     ; AC1 = (c5*x^3 + c4*x^2 + c3*x + c2)
|| mov  *AR2-<<#16,AC0; AC0 = c1
   mac  AC1,T0,AC0     ; AC0 = (c5*x^4 + c4*x^3 + c3*x^2 + c2*x + c1)
   mpyr T0,AC0,AC1     ; AC1 = (c5*x^4 + c4*x^3 + c3*x^2 + c2*x + c1)*x
|| xcc  _neg_result3,TC2
   neg  AC1
_neg_result3
   mov  hi(saturate(AC1<<#3)),*AR0- ; Return sin(x) in Q15
|| bclr FRCT           ; Reset fractional bit
   bclr SATD           ; Reset saturation bit
   ret
   .end
```

4. Write a function to implement the inverse square root approximation equation as follows:

$$1/\sqrt{x} = 1.84293985 - 2.57658958x + 2.11866164x^2 - 0.67824984x^3.$$

This equation approximates the input variable in the range of $0.5 \le x \le 1$. Use this approximation equation to compute $1/\sqrt{x}$ in the following table:

x	0.5	0.6	0.7	0.8	0.9
$1/\sqrt{x}$					

Table 2.14 File listing for the experiment Exp2.2D

Files	Description
`sineCosineTest.c`	Program for testing function approximation
`sine_cos.asm`	Assembly routine for sine and cosine approximation
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file

Note that $1/\sqrt{x}$ will result in numbers greater than 1.0. Use the Q1.14 format to increase the dynamic range.

5. Implement the arctangent function expressed as follows:

$$\tan^{-1}(x) = 0.318253x + 0.003314x^2 - 0.130908x^3 + 0.068542x^4 - 0.009195x^5.$$

This equation approximates the input variable in the range of $x < 1$. Use this approximation equation to compute $\tan^{-1}(x)$ for the x values listed in the following table:

x	0.1	0.3	0.5	0.7	0.9
$\tan^{-1}(x)$					

2.6.3 Real-Time Signal Generation Using eZdsp

This section uses the C5505 eZdsp to generate tones and random numbers. The generated signals will be played by the eZdsp in real time using the AIC3204 chip.

A. Noisy Tone Generation Using Floating-Point C

This experiment generates and plays a tone embedded in random noise using the C5505 eZdsp. Table 2.15 lists the functions used to generate the tone and random noise. Table 2.16 lists the files used for the experiment.

Table 2.15 Floating-point C program for tone and noise generation

```
#define UINTQ14    0x3FFF
#define PI         3.1415926

// Variable definition
static unsigned short n;
static float twoPI_f_Fs;

void initFTone(unsigned short f, unsigned short Fs)
{
    n = 0;
    twoPI_f_Fs = 2.0*PI*(float)f/(float)Fs; // Define frequency
}

short fTone(unsigned short Fs) // Cosine generation
{
    n++;
    if (n >= Fs)
        n = 0;
    return( (short)(cos(twoPI_f_Fs*(float)n)*UINTQ14));
}
void initRand(unsigned short seed) // Random number initialization
{
    srand(seed);
}
short randNoise(void) // Random number generation
{
    return((rand()-RAND_MAX/2)>>1);
}
```

Table 2.16 File listing for the experiment Exp2.3A

Files	Description
`floatPointTest.c`	Program for testing experiment
`ftone.c`	Floating-point C function for tone generation
`randNoise.c`	C function for generating random numbers
`vector.asm`	Assembly program contains interrupt vector
`dma.h`	Header file for DMA functions
`dmaBuff.h`	Header file for DMA data buffer
`i2s.h`	i2s header file for i2s functions
`Ipva200.inc`	C5505 processor include file
`tistdtypes.h`	Standard type define header file
`myC55xUtil.lib`	BIOS audio library
`c5505.cmd`	Linker command file

Procedures of the experiment are listed as follows:

1. Copy the entire project from the companion software package to the working folder, import the CCS floating-point project from the folder `..\Exp2.3\signalGen`, and build and load the program to the C5505 eZdsp.
2. Connect a headphone to the output port of the C5505 eZdsp, run the program, and listen to the audio output.
3. Use an oscilloscope to examine the generated waveform from the audio output jack. Identify the generated tone frequency.
4. Redo the experiment using different tone frequencies and SNRs.

B. Tone Generation Using Fixed-Point C

The experiment given in Section 2.6.2C uses the cosine function written in C55xx assembly language. This experiment mixes the same assembly routine with fixed-point C programs. Table 2.17 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the fixed-point `toneGen` project, and build and load the program to the C5505 eZdsp.

Table 2.17 File listing for the experiment Exp2.3B

Files	Description
`toneGenTest.c`	Program for testing experiment
`tone.c`	C function controls tone generation
`cos.asm`	Assembly routine computes cosine values
`vector.asm`	Assembly program contains interrupt vector
`dma.h`	Header file for DMA functions
`dmaBuff.h`	Header file for DMA data buffer
`i2s.h`	i2s header file for i2s functions
`Ipva200.inc`	C5505 processor include file
`tistdtypes.h`	Standard type define header file
`myC55xUtil.lib`	BIOS audio library
`c5505.cmd`	Linker command file

2. Connect a headphone to the C5505 eZdsp, run the program, and listen to the audio output.
3. Use an oscilloscope connected to the eZdsp audio output jack to verify the generated waveform.
4. Redo the experiment using different tone frequencies.

C. Random Number Generation Using Fixed-Point C

The linear congruential sequence method is widely used because of its simplicity. The random number generation can be expressed as

$$x(n) = [ax(n-1) + b]_{\text{mod } M}, \tag{2.97}$$

where the modulo operation (mod) returns the remainder after division by M. For this experiment, we select $M = 2^{20} = 0\text{x}100000$, $a = 2045$, $b = 0$, and $x(0) = 12\,357$. The C program for the random number generation is listed in Table 2.18, where `seed=x(0)=12357`.

Floating-point multiplication and division are very slow on fixed-point digital signal processors such as the C5505. We can use a mask instead of the modulo operation for a power-of-2 number. The runtime efficiency can be improved by the program listed in Table 2.19. The files used for the experiment are listed in Table 2.20.

Procedures of the experiment are listed as follows:

1. Import the fixed-point `randGenC` project, and build and load the program to the C5505 eZdsp.
2. Connect a headphone to the C5505 eZdsp and run the program.
3. Listen to the audio output from both random number generators `randNumber1()` (in Table 2.18) and `randNumber2()` (in Table 2.19).

Table 2.18 C program for random number generation

```
// Variable definition
static volatile long n;
static short a;

void initRand(long seed)
{
   n = (long)seed;
   a = 2045;
}
short randNumber1(void)
{
   short ran;
   n = a*n + 1;
   n = n - (long)((float)(n*0x100000)/(float)0x100000);
   ran = (n + 1)/0x100001;
   return (ran);
}
```

Table 2.19 C program using mask for modulo operation

```
short randNumber2(void)
{
  short ran;
  n = a*n;
  n = n&0xFFFFF000;
  ran = (short)(n>>20);
  return (ran);
}
```

4. Redo the experiment using $M = 2^{31}$, $a = 69\,069$, $b = 0$, and $x(0) = 1$ in (2.97) and verify the result.
5. Redo the experiment using $M = 2^{31} - 1$, $a = 16\,807$, $b = 0$, and $x(0) = 1$ in (2.97) and verify the result.
6. Optimize the programs used for step 4 and step 5. Using CCS to measure their runtime clock cycles, which random number generator is more efficient, and why?

D. Random Number Generation Using C55xx Assembly Program

To further improve the efficiency, we use an assembly program for random number generation. The assembly routine listed in Table 2.21 reduces the runtime clock cycles. Table 2.22 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the `randGen` project, and build and load the assembly random number generator program to the C5505 eZdsp.
2. Connect a headphone to the C5505 eZdsp and run the program.
3. Listen to the output to examine the assembly implementation of the random number generator. Measure the clock cycles needed for the assembly program.
4. For 16-bit processors, a random number generator can be obtained using (2.97) with $M = 2^{16}$, $a = 25\,173$, $b = 0$, and $x(0) = 13\,849$; redo the experiment to simulate 16-bit processors using the C or assembly program. C5505 temporary registers (T0–T3) and auxiliary registers (AR0–AR7) are all 16-bit registers but C5505 accumulators

Table 2.20 File listing for the experiment Exp2.3C

Files	Description
`randGenCTest.c`	Program for testing experiment
`rand.c`	C function generates random numbers
`vector.asm`	Assembly program contains interrupt vector
`dma.h`	Header file for DMA functions
`dmaBuff.h`	Header file for DMA data buffer
`i2s.h`	i2s header file for i2s functions
`Ipva200.inc`	C5505 processor include file
`tistdtypes.h`	Standard type define header file
`myC55xUtil.lib`	BIOS audio library
`c5505.cmd`	Linker command file

Table 2.21 C55xx assembly program of random number generator

```
    .bss    _n,2,0,2            ; long n
    .bss    _a,1,0,0            ; short a

    .def    _initRand
    .def    _randNumber

    .sect   ".text"
_initRand:
     mov    AC0,dbl(*(#_n))     ; n = (long)seed
     mov    #2045,*(#_a)        ; a = 2045
     ret

_randNumber:
     amov   #_n,XAR0
     mov    *(#_a),T0
     mpym   *AR0+,T0,AC0        ; n = a*n
     mpymu  *AR0-,T0,AC1        ; This is 32x16 integer multiply
     sfts   AC0,#16
     add    AC1,AC0
||   mov    #0xFFFF<<#16,AC2 ; n = n&0xFFFFF000
     or     #0xF000,AC2
     and    AC0,AC2
     mov    AC2,dbl(*AR0)
||   sfts   AC2,#-20,AC0        ; ran = (short)(n>>20)
     mov    AC0,T0              ; Return (ran)
     ret
     .end
```

(AC0–AC3) are 32-bit accumulators (40-bit with the guard bits). For this experiment, avoid using the accumulators since 32-bit (40-bit) accumulators are not available for 16-bit processors. If this experiment is written in C, use the CCS disassembly window to verify that the program does not use any accumulator.

Table 2.22 File listing for the experiment Exp2.3D

Files	Description
randGenATest.c	Program for testing experiment
rand.asm	Assembly routine generates random numbers
vector.asm	Assembly program contains interrupt vector
dma.h	Header file for DMA functions
dmaBuff.h	Header file for DMA data buffer
i2s.h	i2s header file for i2s functions
Ipva200.inc	C5505 processor include file
tistdtypes.h	Standard type define header file
myC55xUtil.lib	BIOS audio library
c5505.cmd	Linker command file

Table 2.23 File listing for the experiment Exp2.3E

Files	Description
`signalGenTest.c`	Program for testing experiment
`tone.c`	C function controls tone generation
`cos.asm`	Assembly routine computes cosine values
`rand.asm`	Assembly routine generates random numbers
`vector.asm`	Assembly program contains interrupt vector
`dma.h`	Header file for DMA functions
`dmaBuff.h`	Header file for DMA data buffer
`i2s.h`	i2s header file for i2s functions
`Ipva200.inc`	C5505 processor include file
`tistdtypes.h`	Standard type define header file
`myC55xUtil.lib`	BIOS audio library
`c5505.cmd`	Linker command file

E. Signal Generation Using C55xx Assembly Program

This experiment combines the tone and random number generators for generating random noise, tone, and tone with additive random noise. The files used for the experiment are listed in Table 2.23.

Procedures of the experiment are listed as follows:

1. Import the `signalGen` project, and build and load the signal generator program to the C5505 eZdsp.
2. Connect a headphone to the C5505 eZdsp and run the program.
3. Listen to the signal generated by the C5505 eZdsp. It will generate three different signals in real time: random number, tone, and tone with random noise.
4. Modify the experiment so it is able to generate the following signals:
   (a) Tones at different frequencies and sampling rates.
   (b) Tones embedded in noise with different SNRs.
   (c) Tones at different frequencies and sampling rates embedded in noise with different SNRs.

Run the experiment to verify it works properly for these different settings.

Exercises

2.1. The all-digital touch-tone phones use the summation of two sine waves for signaling. Frequencies of these sine waves are defined as 697, 770, 852, 941, 1209, 1336, 1477, and 1633 Hz (see details in Chapter 7). Since the sampling rate used by the telecommunications is 8000 Hz, convert those eight analog frequencies into terms of radians per sample and cycles per sample. Also, generate 40 ms of these sine waves using MATLAB®.

2.2. Compute the impulse response $h(n)$ for $n = 0, 1, 2, 3, 4$ of the digital systems defined by the following I/O equations:
(a) $y(n) = x(n) + 0.75y(n-1)$.
(b) $y(n) - 0.3y(n-1) - 0.4y(n-2) = x(n) - 2x(n-1)$.
(c) $y(n) = 2x(n) - 2x(n-1) + 0.5x(n-2)$.

2.3. Construct detailed signal-flow diagrams for the digital systems defined in Problem 2.2.

2.4. Similar to the signal-flow diagram for the IIR filter shown in Figure 2.11, construct a general signal-flow diagram for the IIR filter defined in (2.38) for $M \neq L-1$.

2.5. Find the transfer functions of the three digital systems defined in Problem 2.2.

2.6. Find the zero(s) and/or pole(s) of the digital systems given in Problem 2.2. Discuss the stability of these systems.

2.7. For the second-order IIR filter defined in (2.38) with two complex-conjugate poles defined in (2.52), if the radius $r = 0.9$ and the angle $\theta = \pm 0.25\pi$, find the transfer function and I/O equation of this filter.

2.8. A 2000 Hz analog sine wave is sampled with 10 000 Hz sampling rate. What is the sampling period? What is the digital frequency in terms of ω and F? If there are 100 samples, how many sine wave cycles are covered? Use MATLAB® to plot these 100 sine wave samples.

2.9. For the digital sine wave given in Problem 8, if we compute the DFT with $N = 100$, what is the frequency resolution? If we display the magnitude spectrum as shown in Figure 2.16, what is the value of k that corresponds to the peak spectrum? What happens if the frequency of the sine wave is 1550 Hz and how can the problem be solved?

2.10. Similar to Table 2.2, construct a new table for 5-bit binary numbers.

2.11. Find the fixed-point 2's complement binary representation with $B = 6$ for the decimal numbers 0.570 312 5 and −0.640 625. Also, find the hexadecimal representation of these two numbers. Round the binary numbers to 6 bits and compute the corresponding roundoff errors.

2.12. Similar to Example 2.22, represent the two fractional numbers in Problem 2.11 in integer format for the C55xx assembly programs.

2.13. Represent the 16-bit number given in Example 2.25 in Q1.14, Q3.12, and Q15.0 formats.

2.14. If the quantization process uses truncation instead of rounding, show that the truncation error, $e(n) = x(n) - x(nT)$, will be in the interval $-\Delta < e(n) < 0$. Assuming that the truncation error is uniformly distributed in the interval $(-\Delta, 0)$, compute the mean and the variance of $e(n)$.

2.15. Generate and plot (40 samples) the sinusoidal signals in (a), (b), and (d) using MATLAB®:
(a) $A = 1$, $f = 100$ Hz, and $f_s = 1000$ Hz.
(b) $A = 1$, $f = 400$ Hz, and $f_s = 1000$ Hz.
(c) Discuss the difference of results between (a) and (b).
(d) $A = 1$, $f = 600$ Hz, and $f_s = 1000$ Hz.
(e) Compare and explain the results obtained from (b) and (d).

2.16. Using MATLAB®, draw a pole–zero diagram of the three digital systems given in Problem 2.2.

2.17. Use MATLAB® to display the magnitude and phase responses of the three digital systems given in Problem 2.2.

2.18. Generate 1024 samples of pseudo-random numbers with zero mean and unit variance using the MATLAB® function `rand`. Then use MATLAB® functions `mean`, `std`, and `hist` to verify the results.

2.19. Generate 1024 samples of a sinusoidal signal at a frequency of 1000 Hz, amplitude equal to unity, and sampling rate 8000 Hz. Mix the generated sine wave with the zero-mean pseudo-random number of variance 0.2. What is the SNR (see Appendix A for the definition of SNR in dB)? Calculate and display and the magnitude spectrum using MATLAB®.

2.20. Write a MATLAB® or C program to implement the moving-average filter defined in (2.17). Test the filter using the corrupted sine wave generated in Problem 2.19 as input for different values of L. Plot both the input and output waveforms and magnitude spectra. Discuss the results related to the filter length L.

2.21. Given the difference equations in Problem 2.2, calculate and plot the impulse response $h(n)$, $n = 0, 1, \ldots, 127$ using MATLAB®.

2.22. Similar to Example 2.28, use MATLAB® functions `quantizer` and `quantize` to convert the speech file `timit1.asc` given in Example 2.26 to 4-, 8-, and 12-bit data, and use `soundsc` to play back the quantized signals.

2.23. Select the proper radix conversion functions listed in Table 2.3 to convert the white noise generated in Example 2.29 to hexadecimal format.

References

1. Ahmed, N. and Natarajan, T. (1983) *Discrete-Time Signals and Systems*, Prentice Hall, Englewood Cliffs, NJ.
2. Oppenheim, A.V. and Schafer, R.W. (1989) *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
3. Orfanidis, S.J. (1996) *Introduction to Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
4. Proakis, J.G. and Manolakis, D.G. (1996) *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd edn, Prentice Hall, Englewood Cliffs, NJ.
5. The MathWorks, Inc. (2000) Using MATLAB, Version 6.
6. The MathWorks, Inc. (2004) Signal Processing Toolbox User's Guide, Version 6.
7. The MathWorks, Inc. (2004) Filter Design Toolbox User's Guide, Version 3.
8. Peebles, P. (1980) *Probability, Random Variables, and Random Signal Principles*, McGraw-Hill, New York.
9. Bateman, A. and Yates, W. (1989) *Digital Signal Processing Design*, Computer Science Press, New York.
10. Kuo, S.M. and Morgan, D.R. (1996) *Active Noise Control Systems – Algorithms and DSP Implementations*, John Wiley & Sons, Inc., New York.
11. Marven, C. and Ewers, G. (1996) *A Simple Approach to Digital Signal Processing*, John Wiley & Sons, Inc., New York.
12. McClellan, J.H., Schafer, R.W., and Yoder, M.A. (1998) *DSP First: A Multimedia Approach*, 2nd edn, Prentice Hall, Englewood Cliffs, NJ.
13. Grover, D. and Deller, J.R. (1999) *Digital Signal Processing and the Microcontroller*, Prentice Hall, Upper Saddle River, NJ.
14. Kuo, S.M. and Gan, W.S. (2005) *Digital Signal Processors – Architectures, Implementations, and Applications*, Prentice Hall, Upper Saddle River, NJ.
15. The MathWorks, Inc. (2004) Fixed-Point Toolbox User's Guide, Version 1.

3

Design and Implementation of FIR Filters

As discussed in Chapter 2, digital filters include FIR and IIR filters. This chapter introduces digital FIR filters with a focus on the design, implementation, and application issues [1–10].

3.1 Introduction to FIR Filters

Some advantages of using FIR filters are summarized as follows:

1. FIR filters are always stable.
2. Design of linear phase filters can be guaranteed.
3. Finite-precision errors are less severe in FIR filters.
4. FIR filters can be efficiently implemented on most digital signal processors with optimized hardware and special instructions for FIR filtering.

The process of deriving filter coefficients that satisfies a given set of specifications is called filter design. Even though a number of computer-aided tools such as MATLAB® with related toolboxes such as the *Signal Processing Toolbox* [11] and *DSP System Toolbox* (*Filter Desing Toolbox* in older version [12]) are available for designing digital filters, we still need to understand the basic characteristics of filters and become familiar with the techniques used for implementing digital filters.

3.1.1 Filter Characteristics

Linear time-invariant filters are characterized by magnitude response, phase response, stability, rising time, settling time, and overshoot. Magnitude and phase responses determine the steady-state response of the filter, while the rising time, settling time, and overshoot specify the transient response. For an instantaneous input change, the rising time specifies its output-changing rate. The settling time describes how long it takes for the output to settle down to a stable value, and the overshoot shows if the output exceeds the desired value.

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

As illustrated in Figure 2.9 and defined in (2.31), the magnitude and phase responses of the input signal, filter, and output signal can be expressed as

$$|Y(\omega)| = |X(\omega)||H(\omega)| \tag{3.1}$$

and

$$\phi_Y(\omega) = \phi_X(\omega) + \phi_H(\omega), \tag{3.2}$$

where $\phi_Y(\omega)$, $\phi_X(\omega)$, and $\phi_H(\omega)$ denote the phase responses of the output, input, and filter, respectively. These equations show that the magnitude and phase spectra of the input signal are modified by the filter. The magnitude response $|H(\omega)|$ specifies the gain and the phase response $\phi_H(\omega)$ affects the phase shift (or time delay) of the filter at a given frequency.

A linear phase filter has a phase response that satisfies

$$\phi_H(\omega) = -\alpha\omega \quad \text{or} \quad \phi_H(\omega) = \pi - \alpha\omega. \tag{3.3}$$

The group delay function of the filter is defined as

$$T_{\rm d}(\omega) = \frac{-{\rm d}\phi_H(\omega)}{{\rm d}\omega}. \tag{3.4}$$

Therefore, for the linear phase filter defined in (3.3), the group delay $T_{\rm d}(\omega)$ is a constant α for all frequencies. This filter avoids phase distortion because all frequency components in the input signal are delayed by the same amount of time. Linear phase is important in many real-world applications where the temporal relationships between different frequency components are critical.

Example 3.1

Consider the simple two-point moving-average filter given in Example 2.14. The magnitude response is

$$|H(\omega)| = \sqrt{\frac{1}{2}[1 + \cos(\omega)]}.$$

Since the magnitude response falls off monotonically to zero at $\omega = \pi$, this is a lowpass filter with the phase response

$$\phi_H(\omega) = \frac{-\omega}{2},$$

which is linear phase as shown in (3.3). Therefore, this filter has constant time delay

$$T_{\rm d}(\omega) = \frac{-{\rm d}\phi_H(\omega)}{{\rm d}\omega} = 0.5.$$

These characteristics can be verified using the MATLAB® script `example3_1.m`. The magnitude and phase responses using `freqz(b,a)` are shown in Figure 3.1. The group delay is computed and displayed using `grpdelay(b,a)`, which shows a constant delay of 0.5 for all frequencies.

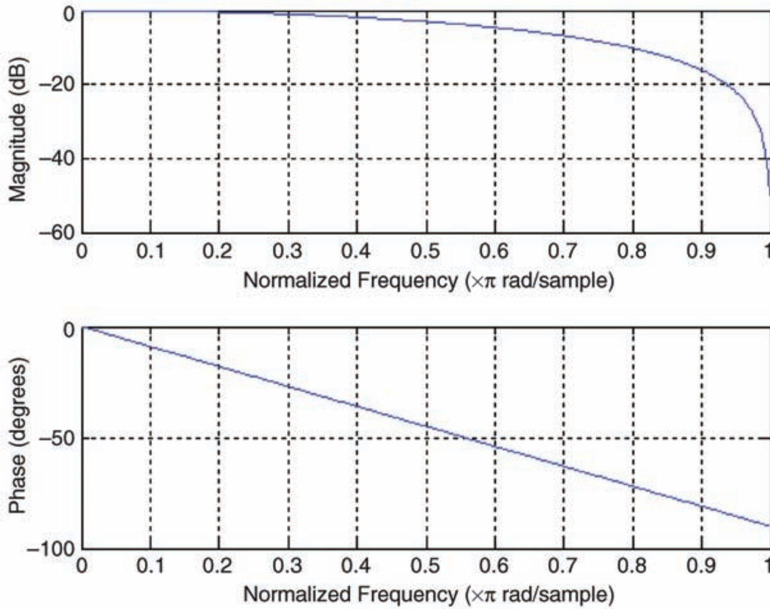


Figure 3.1 Magnitude and phase responses of two-point moving-average filter

3.1.2 Filter Types

A filter is usually defined in terms of its magnitude response, and there are four different types of frequency-selective filters: lowpass, highpass, bandpass, and bandstop filters. Because the magnitude response of a digital filter with real coefficients is an even function of ω, the filter specifications are usually defined in the frequency range of $0 \leq \omega \leq \pi$.

The magnitude response of an ideal lowpass filter is illustrated in Figure 3.2(a). The regions $0 \leq \omega \leq \omega_c$ and $\omega > \omega_c$ are referred to as the passband and stopband, respectively, and the frequency ω_c is called the cutoff frequency. An ideal lowpass filter has magnitude response $|H(\omega)| = 1$ in the passband $0 \leq \omega \leq \omega_c$, and $|H(\omega)| = 0$ for the stopband $\omega > \omega_c$. Thus, the ideal lowpass filter passes low-frequency components below the cutoff frequency and attenuates high-frequency components above ω_c.

The magnitude response of an ideal highpass filter is illustrated in Figure 3.2(b). A highpass filter passes high-frequency components above the cutoff frequency ω_c and attenuates low-frequency components below ω_c. In practice, highpass filters can be used to eliminate low-frequency noise.

The magnitude response of an ideal bandpass filter is illustrated in Figure 3.2(c). The frequencies ω_a and ω_b are called the lower and upper cutoff frequencies, respectively. The ideal bandpass filter passes frequency components between the two cutoff frequencies ω_a and ω_b, and attenuates frequency components below the frequency ω_a and above the frequency ω_b.

The magnitude response of an ideal bandstop (or band-reject) filter is illustrated in Figure 3.2(d). A filter with a very narrow stopband is also called a notch filter. For example, a power line generates a 60 Hz sinusoidal noise called power line interference or 60 Hz hum, which can be removed by the notch filter with center frequency at 60 Hz.

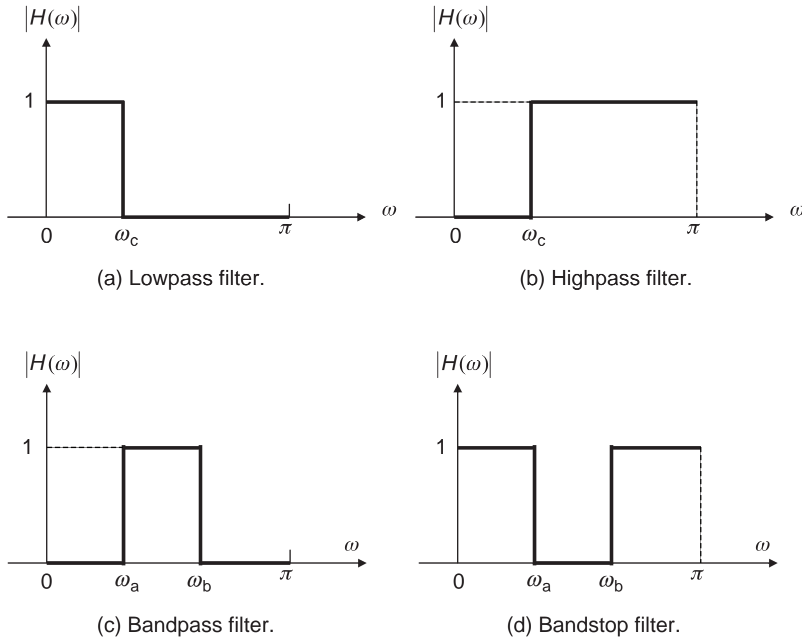


Figure 3.2 Magnitude responses of four different ideal filters

In addition to these frequency-selective filters, an allpass filter provides frequency response $|H(\omega)| = 1$ for all ω. From (3.2), allpass filters can be designed to correct the phase distortion introduced by physical systems without changing the amplitudes of frequency components. A very special case of the allpass filter is the ideal Hilbert transformer, which produces a 90° phase shift of the input signal.

A multiband filter has more than one passband or stopband. A special case of the multiband filter is the comb filter. The comb filter has equally spaced zeros, with the shape of the magnitude response resembling a comb. The difference equation of the comb filter is given as

$$y(n) = x(n) - x(n-L), \tag{3.5}$$

where L is a positive integer. The transfer function of this FIR filter is

$$H(z) = 1 - z^{-L} = \frac{z^L - 1}{z^L}. \tag{3.6}$$

Thus, the comb filter has L zeros equally spaced on the unit circle at the locations

$$z_l = \mathrm{e}^{\mathrm{j}(2\pi/L)l}, \quad l = 0, 1, \ldots, L-1. \tag{3.7}$$

Example 3.2

A comb filter with $L = 8$ has eight zeros at

$$z_l = 1, \mathrm{e}^{\pi/4}, \mathrm{e}^{\pi/2}, \mathrm{e}^{3\pi/4}, \mathrm{e}^{\pi}(-1), \mathrm{e}^{5\pi/4}, \mathrm{e}^{3\pi/2}, \mathrm{e}^{7\pi/4}$$

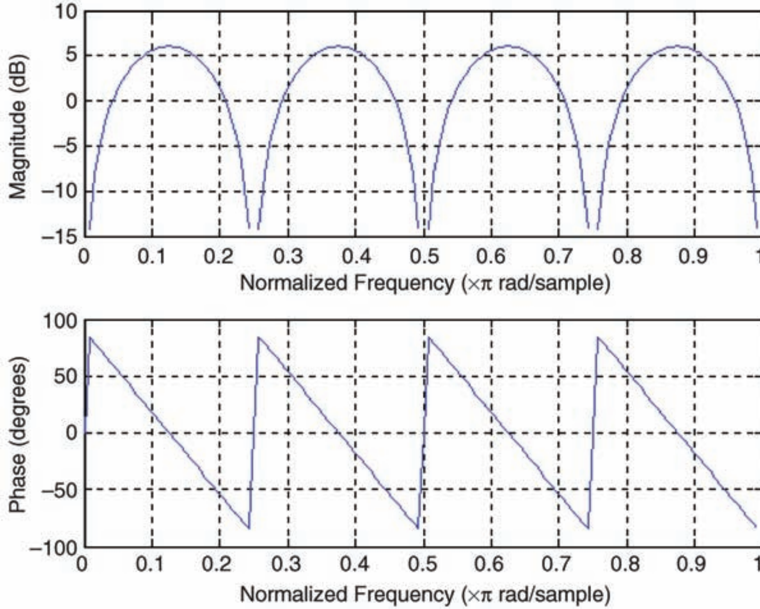


Figure 3.3 Magnitude and phase responses of comb filter with $L = 8$

for $l = 0, 1, \ldots, 7$, respectively. The frequency response of this comb filter is plotted in Figure 3.3 using the MATLAB® script `example3_2.m` for $L = 8$.

Figure 3.3 shows that the comb filter can be used as a multiple bandstop filter to remove narrowband noise at frequencies

$$\omega_l = 2\pi l/L, \quad l = 0, 1, \ldots, L/2 - 1. \tag{3.8}$$

The centers of the passbands lie halfway between the adjacent zeros of the frequency response, that is, at frequencies $(2l + 1)\pi/L$, $l = 0, 1, \ldots, L/2 - 1$.

Comb filters are useful for passing or eliminating specific frequencies and their harmonics. Using comb filters for attenuating periodic signals with harmonic-related components is more efficient than designing individual filters for each harmonic. For example, the humming sound produced by large transformers located in electric utility substations comprises even-numbered harmonics (120, 240, 360 Hz, etc.) of the 60 Hz power line frequency. When a signal is corrupted by the transformer noise, the comb filter with notches at multiples of 120 Hz can be used to eliminate those undesired harmonic components.

3.1.3 Filter Specifications

The characteristics of digital filters are often specified in the frequency domain, thus the design is usually based on magnitude response specifications. In practice, we cannot achieve the infinitely sharp cutoff as in the ideal filters illustrated in Figure 3.2. A practical filter has a

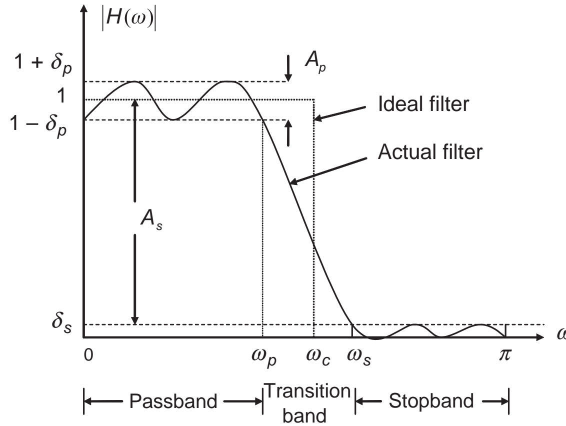


Figure 3.4 Magnitude response and performance measurement of lowpass filter

gradual roll-off transition band from the passband to the stopband. The specifications are often given in the form of tolerance (or ripple) schemes, and the transition band is specified to permit the smooth magnitude roll-off.

The typical magnitude response of a lowpass filter is illustrated in Figure 3.4, where the dashed horizontal lines in the figure indicate the tolerance limits. The magnitude response has the peak deviation δ_p in the passband, and the maximum deviation δ_s in the stopband. The frequencies ω_p and ω_s are the passband edge (cutoff) frequency and the stopband edge frequency, respectively.

As shown in Figure 3.4, the magnitude of the passband ($0 \le \omega \le \omega_p$) is approximately unity with an error of $\pm\delta_p$. That is,

$$1 - \delta_p \le |H(\omega)| \le 1 + \delta_p, \quad 0 \le \omega \le \omega_p. \tag{3.9}$$

The passband ripple δ_p is the allowed variation in magnitude response in the passband. Note that the gain of the magnitude response is normalized to one (0 dB).

In the stopband, the magnitude response is approximately zero with an error of δ_s. That is,

$$|H(\omega)| \le \delta_s, \quad \omega_s \le \omega \le \pi. \tag{3.10}$$

The stopband ripple (or attenuation) δ_s describes the attenuation for signal components above ω_s.

Passband and stopband deviations are usually expressed in decibels. The passband ripple and the stopband attenuation are defined as

$$A_p = 20\log_{10}\left(\frac{1+\delta_p}{1-\delta_p}\right) \text{dB} \tag{3.11}$$

and

$$A_s = -20\log_{10}\delta_s \text{ dB}. \tag{3.12}$$

Example 3.3

Consider a filter that has passband ripples within ±0.01, that is, $\delta_p = 0.01$. From (3.11), we have

$$A_p = 20\log_{10}\left(\frac{1.01}{0.99}\right) = 0.1737\ \text{dB}.$$

When the stopband attenuation is given as $\delta_s = 0.01$, we have

$$A_s = -20\log_{10}(0.01) = 40\ \text{dB}.$$

The transition band is the frequency region between the passband edge frequency ω_p and the stopband edge frequency ω_s. The magnitude response decreases monotonically from the passband to the stopband in this region. The width of the transition band determines how sharp the filter is. Generally, a higher order filter is required for realizing smaller δ_p and δ_s and narrower transition bands.

3.1.4 Linear Phase FIR Filters

The signal-flow diagram of the FIR filter is shown in Figure 2.6, and the I/O equation is defined in (2.14). If L is an odd number, we define $M = (L-1)/2$. Equation (2.14) can be written as

$$\begin{aligned} B(z) &= \sum_{l=0}^{2M} b_l z^{-l} = \sum_{l=-M}^{M} b_{l+M} z^{-(l+M)} = z^{-M}\left[\sum_{l=-M}^{M} h_l z^{-l}\right] \\ &= z^{-M} H(z), \end{aligned} \tag{3.13}$$

where

$$H(z) = \sum_{l=-M}^{M} h_l z^{-l}. \tag{3.14}$$

Let h_l have the symmetry property

$$h_l = h_{-l}, \qquad l = 0, 1, \ldots, M. \tag{3.15}$$

From (3.13), the frequency response $B(\omega)$ can be written as

$$\begin{aligned} B(\omega) &= B(z)\big|_{z=e^{j\omega}} = \mathrm{e}^{-\mathrm{j}\omega M} H(\omega) \\ &= \mathrm{e}^{-\mathrm{j}\omega M}\left[\sum_{l=-M}^{M} h_l\, \mathrm{e}^{-\mathrm{j}\omega l}\right] = \mathrm{e}^{-\mathrm{j}\omega M}\left[h_0 + \sum_{l=1}^{M} h_l\left(\mathrm{e}^{\mathrm{j}\omega l} + \mathrm{e}^{-\mathrm{j}\omega l}\right)\right] \\ &= \mathrm{e}^{-\mathrm{j}\omega M}\left[h_0 + 2\sum_{l=1}^{M} h_l \cos(\omega l)\right]. \end{aligned} \tag{3.16}$$

If L is an even integer and $M = L/2$, the derivation of (3.16) has to be modified slightly.

Equation (3.16) shows that if h_l is real valued,

$$H(\omega) = h_0 + 2\sum_{l=1}^{M} h_l \cos(\omega l)$$

is a real function of ω. Thus, the phase and group delay of $B(\omega)$ are

$$\phi_B(\omega) = \begin{cases} -M\omega, & H(\omega) \geq 0 \\ \pi - M\omega, & H(\omega) < 0 \end{cases} \tag{3.17a}$$

and

$$T_\mathrm{d}(\omega) = \frac{-\mathrm{d}\phi_H(\omega)}{\mathrm{d}\omega} = M = \begin{cases} L/2, & L \text{ is even} \\ (L-1)/2, & L \text{ is odd.} \end{cases} \tag{3.17b}$$

These equations show that the phase of the FIR filter is a linear function of ω and the group delay measured in samples is a constant M for all frequencies. However, there are sign changes in $H(\omega)$ corresponding to 180° phase shifts in $B(\omega)$, and $B(\omega)$ is only piecewise linear in the passbands as shown in Figure 3.3.

If h_l has the antisymmetry (negative symmetry) property

$$h_l = -h_{-l}, \quad l = 0, 1, \ldots, M, \tag{3.18}$$

this implies $h(0) = 0$. Following the similar derivation of (3.16), we also can show that the phase of $B(z)$ is a linear function of ω.

In conclusion, an FIR filter has linear phase if its coefficients satisfy the positive symmetric condition

$$b_l = b_{L-1-l}, \quad l = 0, 1, \ldots, L-1, \tag{3.19}$$

or the antisymmetric condition

$$b_l = -b_{L-1-l}, \quad l = 0, 1, \ldots, L-1. \tag{3.20}$$

The group delay of a symmetric (or antisymmetric) FIR filter is $T_\mathrm{d}(\omega) = (L-1)/2$, which corresponds to the midpoint of the FIR filter. Depending on whether L is even or odd and whether b_l has positive or negative symmetry, there are four types of linear phase FIR filters.

The symmetry (or antisymmetry) property of a linear phase FIR filter can be exploited to reduce the total number of multiplications required by filtering. Consider the FIR filter with even number length L and positive symmetry as defined in (3.19). Equation (2.36) can be modified as

$$H(z) = b_0\left(1 + z^{-L+1}\right) + b_1\left(z^{-1} + z^{-L+2}\right) + \cdots + b_{L/2-1}\left(z^{-L/2+1} + z^{-L/2}\right). \tag{3.21}$$

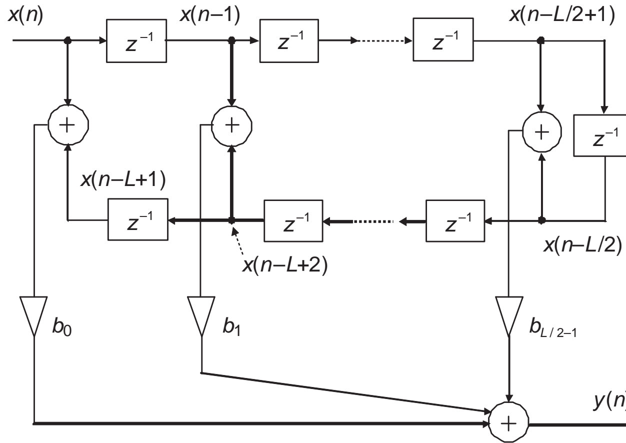


Figure 3.5 Signal-flow diagram of symmetric FIR filter; L is an even number

The realization of $H(z)$ defined in (3.21) is illustrated in Figure 3.5 with the I/O equation expressed as

$$\begin{aligned} y(n) &= b_0[x(n) + x(n-L+1)] + b_1[x(n-1) + x(n-L+2)] + \cdots \\ &\quad + b_{L/2-1}[x(n-L/2+1) + x(n-L/2)] \\ &= \sum_{l=0}^{L/2-1} b_l[x(n-l) + x(n-L+1+l)]. \end{aligned} \tag{3.22}$$

For the antisymmetric FIR filter, the addition of two signals is replaced by subtraction. That is,

$$y(n) = \sum_{l=0}^{L/2-1} b_l[x(n-l) - x(n-L+1+l)]. \tag{3.23}$$

As shown in (3.22) and Figure 3.5, the number of multiplications is reduced to half by first adding the pair of samples, then multiplying the sum by the corresponding coefficient. The trade-off is that two address pointers are needed to point at both $x(n-l)$ and $x(n-L+1+l)$ instead of accessing data linearly through the same buffer with a single pointer. The TMS320C55xx provides two special assembly instructions, `FIRSADD` and `FIRSSUB`, for implementing the symmetric and antisymmetric FIR filters, respectively. An experiment using `FIRSADD` will be presented in Section 3.5.3.

3.1.5 Realization of FIR Filters

An FIR filter can be operated either on a block-by-block basis or a sample-by-sample basis. In block processing, the input samples are segmented into multiple data blocks. The filtering operation is performed one block at a time, and the resulting output blocks are recombined to form the overall output. The filtering process for each data block can be implemented using

linear convolution, or by fast convolution, which will be introduced in Chapter 5. In sample-by-sample processing, the input sample is processed in every sampling period after the current input $x(n)$ becomes available.

As discussed in Section 2.2.1, the output of a linear time-invariant system is the input samples convoluted with the impulse response of the system. Assuming that the filter is casual, the output at time n is computed as

$$y(n) = \sum_{l=0}^{\infty} h(l)x(n-l). \tag{3.24}$$

The process of computing the linear convolution involves the following four steps:

1. Folding – fold $x(l)$ about $l = 0$ to obtain $x(-l)$.
2. Shifting – shift $x(-l)$ by n samples to the right to obtain $x(n-l)$.
3. Multiplication – multiply the overlapped $h(l)$ and $x(n-l)$ to obtain the products of $h(l)x(n-l)$ for all l.
4. Summation – sum all the products to obtain the output $y(n)$ at time n.

Repeat steps 2 through 4 in computing the output of the system at other time instants n. Note that convolution of the input signal of length M with an impulse response of length L results in the output signal of length $L+M-1$.

Example 3.4

Consider the FIR filter consisting of four coefficients b_0, b_1, b_2, and b_3. Then we have

$$y(n) = \sum_{l=0}^{3} b_l x(n-l), \quad n \geq 0.$$

Linear convolution yields

$$\begin{aligned} n &= 0, y(0) = b_0x(0) \\ n &= 1, y(1) = b_0x(1) + b_1x(0) \\ n &= 2, y(2) = b_0x(2) + b_1x(1) + b_2x(0) \\ n &= 3, y(3) = b_0x(3) + b_1x(2) + b_2x(1) + b_3x(0). \end{aligned}$$

In general, we have

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) + b_3x(n-3), \quad n \geq 3.$$

The graphical interpretation is illustrated in Figure 3.6.

As shown in Figure 3.6, the input sequence is flipped around (folding) and then shifted to the right one sample at a time to overlap the filter coefficients. At each time instant, the output value is the sum of products of the overlapped coefficients with the corresponding input data aligned below it. This flip-and-slide form of linear convolution can be illustrated as in Figure 3.7. Note that shifting $x(-l)$ to the right is equivalent to shifting b_l to the left by one unit at each sampling period.

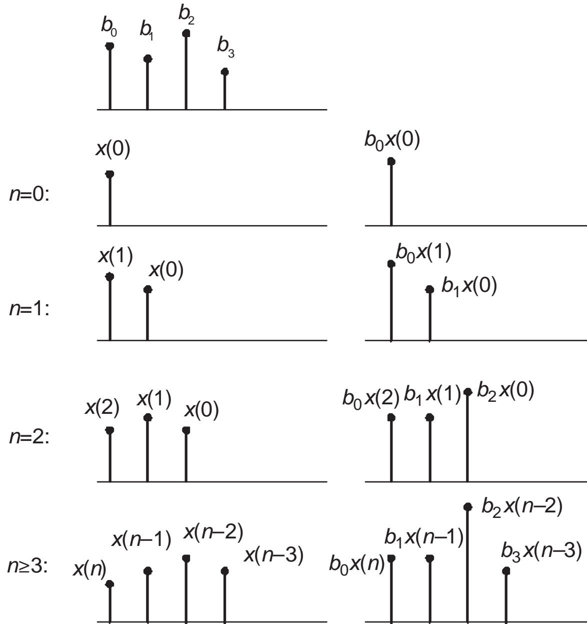


Figure 3.6 Graphical interpretation of linear convolution, $L=4$

At time $n=0$, the input sequence is extended by padding $L-1$ zeros to its right. The only non-zero product comes from b_0 multiplied by $x(0)$, which are time aligned. It takes the filter $L-1$ iterations before it is completely overlapped by the input sequence. Therefore, the first $L-1$ outputs correspond to the transient state of the FIR filtering. After $n \geq L-1$, the signal buffer of the FIR filter is full and the filter is in the steady state.

In FIR filtering, the coefficients are constants, but the data in the signal buffer (or tapped delay line) changes every sampling period, T. The signal buffer is refreshed in the fashion illustrated in Figure 3.8, where the oldest sample $x(n-L+1)$ is discarded and the rest of the samples are shifted one location to the right in the buffer. A new sample (from an ADC in real-time applications) is inserted into the memory location labeled as $x(n)$. This $x(n)$, coming in at time n, will become $x(n-1)$ in the next sampling period, then $x(n-2)$, and so on, until it

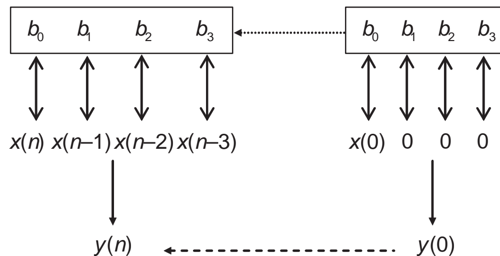


Figure 3.7 Flip-and-slide process of linear convolution

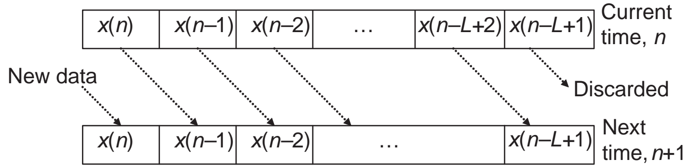


Figure 3.8 Signal buffer refreshing for FIR filtering

simply drops off the end of the delay chain. Refreshing the signal buffer (as shown in Figure 3.8) requires intensive processing time if the data-move operations are not implemented by hardware.

The most efficient method for refreshing a signal buffer is to arrange the signal samples in a circular fashion as illustrated in Figure 3.9(a). Instead of shifting the data samples forward while holding the fixed starting address of buffer as shown in Figure 3.8, the data samples in the circular buffer do not move, but the buffer starting address is updated backwards (counterclockwise). The current signal sample, $x(n)$, is pointed by the start-address pointer and the previous samples are already loaded sequentially in a clockwise direction. As a new sample is received, it is placed at the position $x(n)$ and the filtering operation is performed. After calculating the output $y(n)$, the start pointer is moved counterclockwise one position to $x(n-L+1)$ and waits for the next input signal to arrive. The next input at time $n+1$ will be written to the $x(n-L+1)$ position, and it is referred as $x(n)$ for the next iteration. The circular buffer is very efficient because the update is carried out by adjusting the start-address (pointer) without physically shifting any data samples in memory.

Figure 3.9(b) shows the circular buffer for FIR filter coefficients, which allows the coefficient pointer to wrap around when it reaches the end of the coefficient buffer. That

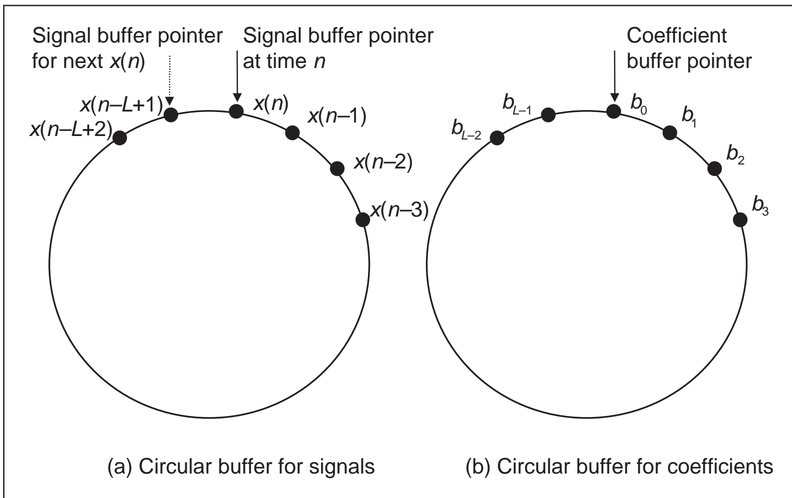


Figure 3.9 Circular buffers for FIR filter. (a) Circular buffer for holding the signals. The start pointer to $x(n)$ is updated in the counterclockwise direction. (b) Circular buffer for FIR filter coefficients, the pointer always pointing to b_0 at the beginning of filtering

is, the pointer moves from b_{L-1} to b_0 such that the FIR filtering will always start at the first coefficient.

3.2 Design of FIR Filters

The objective of designing FIR filters is to determine a set of filter coefficients that satisfies the given specifications. A variety of techniques have been developed for designing FIR filters. The Fourier series method offers a simple way of computing FIR filter coefficients, and thus is used to explain the principles of FIR filter design.

3.2.1 Fourier Series Method

The Fourier series method designs an FIR filter by calculating the impulse response of the filter that approximates the desired frequency response. The frequency response of a filter is defined using the discrete-time Fourier transform as

$$H(\omega) = \sum_{n=-\infty}^{\infty} h(n)\mathrm{e}^{-\mathrm{j}\omega n}, \tag{3.25}$$

where the impulse response of the filter can be obtained using the inverse discrete-time Fourier transform as

$$h(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H(\omega)\mathrm{e}^{\mathrm{j}\omega n}\,\mathrm{d}\omega, \quad -\infty \leq n \leq \infty. \tag{3.26}$$

This equation shows that the impulse response $h(n)$ is double-sided and has infinite length.

For the desired FIR filter with frequency response $H(\omega)$, the corresponding impulse response $h(n)$ (same as the filter coefficients) can be calculated by evaluating the integral defined in (3.26). A finite-duration impulse response $\{h'(n)\}$ can be obtained by truncating the ideal infinite-length impulse response defined in (3.26). That is,

$$h'(n) = \begin{cases} h(n), & -M \leq n \leq M \\ 0, & \text{otherwise.} \end{cases} \tag{3.27}$$

A causal FIR filter can be derived by shifting the $h'(n)$ sequence to the right by M samples and re-indexing the coefficients as

$$b'_l = h'(l-M), \quad l = 0, 1, \ldots, 2M. \tag{3.28}$$

Assuming L is an odd number ($L = 2M+1$), this FIR filter has L coefficients $b'_l, l = 0, 1, \ldots, L-1$. The impulse response is symmetric about b'_M due to the fact that $h(-n) = h(n)$ given in (3.26). Therefore, the transfer function $B'(z)$ with coefficients defined in (3.28) has linear phase and a constant group delay.

Example 3.5

The ideal lowpass filter of Figure 3.2(a) has frequency response

$$H(\omega) = \begin{cases} 1, & |\omega| \leq \omega_c \\ 0, & \text{otherwise.} \end{cases} \tag{3.29}$$

The corresponding impulse response can be computed using (3.26) as

$$\begin{aligned} h(n) &= \frac{1}{2\pi}\int_{-\pi}^{\pi} H(\omega)e^{j\omega n}\,d\omega = \frac{1}{2\pi}\int_{-\omega_c}^{\omega_c} e^{j\omega n}\,d\omega \\ &= \frac{1}{2\pi}\left[\frac{e^{j\omega n}}{jn}\right]_{-\omega_c}^{\omega_c} = \frac{1}{2\pi}\left[\frac{e^{j\omega_c n} - e^{-j\omega_c n}}{jn}\right] \\ &= \frac{\sin(\omega_c n)}{\pi n} = \frac{\omega_c}{\pi}\,\text{sinc}\left(\frac{\omega_c n}{\pi}\right), \end{aligned} \tag{3.30}$$

where the sinc function is defined as

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}.$$

By setting all impulse response coefficients outside the range $-M \leq n \leq M$ to zero, and shifting M units to the right, we obtain the causal FIR filter of finite length L with coefficients

$$b'_l = \begin{cases} \dfrac{\omega_c}{\pi}\,\text{sinc}\left[\dfrac{\omega_c(l - M)}{\pi}\right], & 0 \leq l \leq L-1 \\ 0, & \text{otherwise.} \end{cases} \tag{3.31}$$

Example 3.6

Design a lowpass FIR filter with the following frequency response:

$$H(f) = \begin{cases} 1, & 0 \leq f \leq 1\,\text{kHz} \\ 0, & 1\,\text{kHz} < f \leq 4\,\text{kHz}, \end{cases}$$

where the sampling rate is 8 kHz. The impulse response is limited to 2.5 ms.

Since $2MT = 0.0025$ s and $T = 0.000\,125$ s, we need $M = 10$. Thus, the actual filter has 21 ($L = 2M + 1$) coefficients, and 1 kHz corresponds to $\omega_c = 0.25\pi$. From (3.31), we have

$$b'_l = 0.25\text{sinc}\,[0.25(l - 10)], \quad l = 0, 1, \ldots, 20.$$

Example 3.7

Design a lowpass filter of cutoff frequency $\omega_c = 0.4\pi$ with filter length $L = 61$.

When $L = 61$, $M = (L - 1)/2 = 30$. From (3.31), the designed filter coefficients are

$$b'_l = 0.4\text{sinc}[0.4(l - 30)], \quad l = 0, 1, \ldots, 60.$$

These coefficients are computed and the magnitude response of the designed filter is plotted in Figure 3.10 using the MATLAB® script `example3_7.m`.

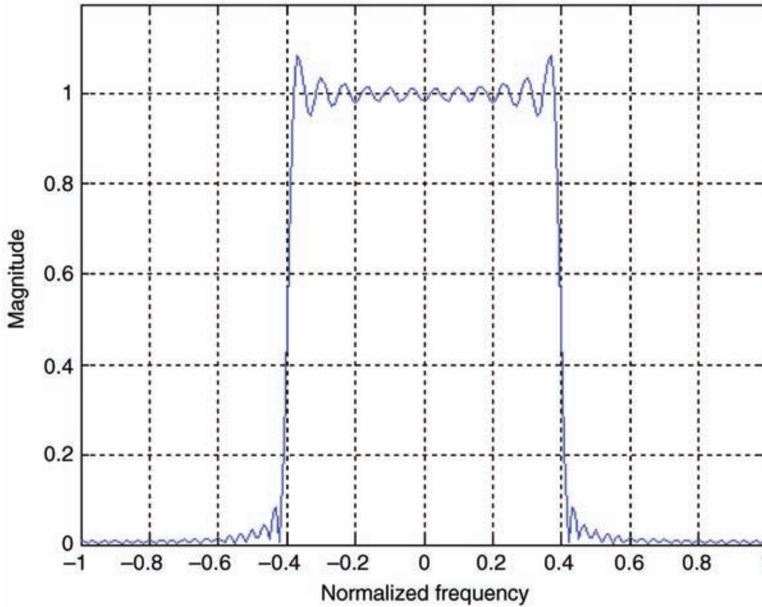


Figure 3.10 Magnitude response of lowpass filter designed by Fourier series method

3.2.2 Gibbs Phenomenon

As shown in Figure 3.10, the FIR filter obtained by simply truncating the impulse response of the desired filter defined in (3.27) exhibits an oscillatory behavior (or ripples) in its magnitude response. As the length of the filter increases, the number of ripples in both the passband and stopband increases, and the width of the ripple decreases. The larger ripples occur near the transition edges and their amplitudes are independent of L.

The truncation operation described in (3.27) can be considered as multiplication of the infinite-length sequence $h(n)$ by the finite-length rectangular window $w(n)$. That is,

$$h'(n) = h(n)w(n), \quad -\infty \leq n \leq \infty, \tag{3.32}$$

where the rectangular window $w(n)$ is defined as

$$w(n) = \begin{cases} 1, & -M \leq n \leq M \\ 0, & \text{otherwise.} \end{cases} \tag{3.33}$$

Example 3.8

The oscillatory behavior of the truncated Fourier series representation of FIR filter coefficients, observed in Figure 3.10, can be explained by the frequency response of the rectangular window defined in (3.33). The frequency response can be expressed as

$$W(\omega) = \sum_{n=-M}^{M} \mathrm{e}^{-\mathrm{j}\omega n} = \frac{\sin[(2M+1)\omega/2]}{\sin(\omega/2)}. \tag{3.34}$$

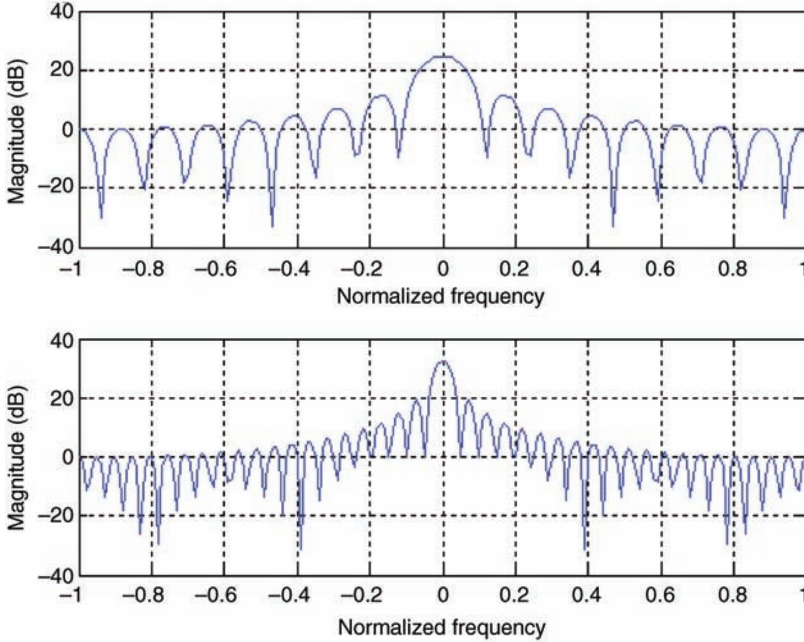


Figure 3.11 Magnitude responses of the rectangular windows for $M = 8$ (top) and $M = 20$ (bottom).

Magnitude responses of $W(\omega)$ for $M = 8$ and 20 are generated using MATLAB® script `example3_8.m`. As illustrated in Figure 3.11, the magnitude response has a mainlobe centered at $\omega = 0$. All the other ripples are called sidelobes. The magnitude response has the first zero at $\omega = 2\pi/(2M + 1)$. Therefore, the width of the mainlobe is $4\pi/(2M + 1)$. From (3.34), it is easy to show that the magnitude of the mainlobe is $|W(0)| = 2M + 1$. The first sidelobe is approximately located at frequency $\omega_1 = 3\pi/(2M + 1)$ with a magnitude of $|W(\omega_1)| \approx 2(2M + 1)/3\pi$ for $M \gg 1$. The ratio of the mainlobe magnitude to the first sidelobe magnitude is

$$\left|\frac{W(0)}{W(\omega_1)}\right| \approx \frac{3\pi}{2} = 13.5\ \text{dB}.$$

As ω increases from 0 to π, the denominator becomes larger. This results in the damped function shown in Figure 3.11. As M increases, the width of the mainlobe decreases.

The rectangular window has an abrupt transition to zero outside the range $-M \le n \le M$, which causes the Gibbs phenomenon in the magnitude response. The Gibbs phenomenon can be reduced either by using a window that tapers smoothly to zero at each end, or by providing a smooth transition from the passband to the stopband. A tapered window will reduce the height of the sidelobes and increase the width of the mainlobe, resulting in a wider transition at the discontinuity. This phenomenon is often referred to as leakage or smearing, which will be discussed in Chapter 5.

3.2.3 Window Functions

A large number of tapered windows have been developed and optimized for different applications. In this section, we restrict our discussion to the commonly used Hamming window of length $L = 2M + 1$. That is, $w(n)$, $n = 0, 1, \ldots, L-1$, and is symmetric about its middle, $n = M$. Two factors that determine the performance of windows in FIR filter design are the window's mainlobe width and the relative sidelobe level. To ensure a fast transition from the passband to the stopband, the window should have a small mainlobe width. On the other hand, to reduce the magnitudes of passband and stopband ripples, the area under the sidelobes should be small. Unfortunately, there is a trade-off between these two requirements for choosing a suitable window.

The Hamming window function is defined as

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{L-1}\right), \quad n = 0, 1, \ldots, L-1, \tag{3.35}$$

which also corresponds to a raised cosine, but with different weights for the constant and cosine terms. As shown in (3.35), the Hamming function tapers the end values to 0.08. MATLAB® provides the Hamming window function

```
w = hamming(L);
```

The Hamming window function and its magnitude response generated by MATLAB® script `hamWindow.m` are shown in Figure 3.12. The mainlobe width of the Hamming window

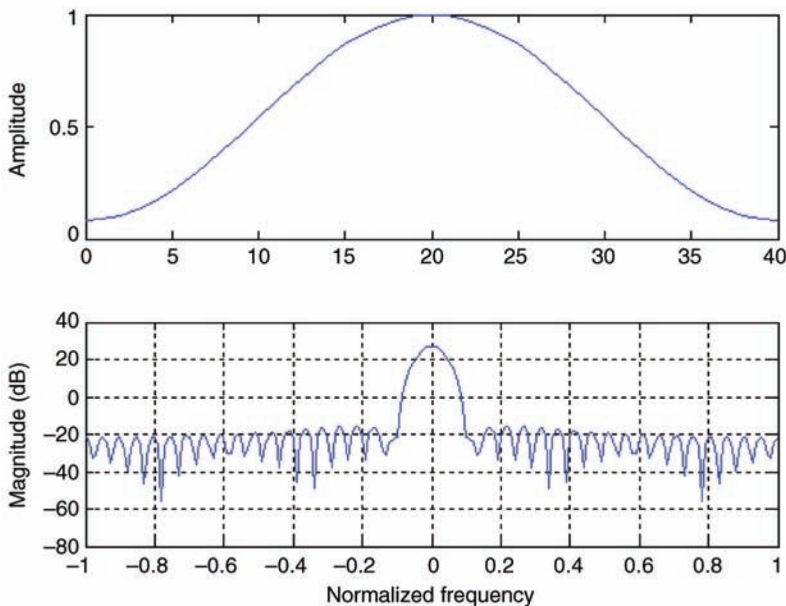


Figure 3.12 Hamming window function (top) and its magnitude response (bottom), $L = 41$

is about the same as the Hanning window, but the latter has an additional 10 dB of stopband attenuation (total of 41 dB). The Hamming window gives low ripple over the passband and good stopband attenuation, and it is usually more appropriate for lowpass filter design.

Example 3.9

Design a lowpass FIR filter of cutoff frequency $\omega_c = 0.4\pi$ and order $L = 61$ using the Hamming window.

Using the MATLAB® script (`example3_9.m`) similar to the one used in Example 3.7, we plot the magnitude responses of designed filters in Figure 3.13 using both the rectangular and Hamming windows. We observe that the ripples produced by the rectangular window design are almost completely eliminated by the Hamming window design. The trade-off for reducing the ripples is increasing transition width.

The procedure for designing FIR filters using Fourier series and windows is summarized as follows:

1. Determine the window type that will satisfy the stopband attenuation requirements.
2. Determine the window size L based on the given transition width.
3. Calculate the window coefficients $w(l)$, $l = 0, 1, \ldots, L-1$.
4. Generate the ideal impulse response $h(n)$ using (3.26) for the desired filter.

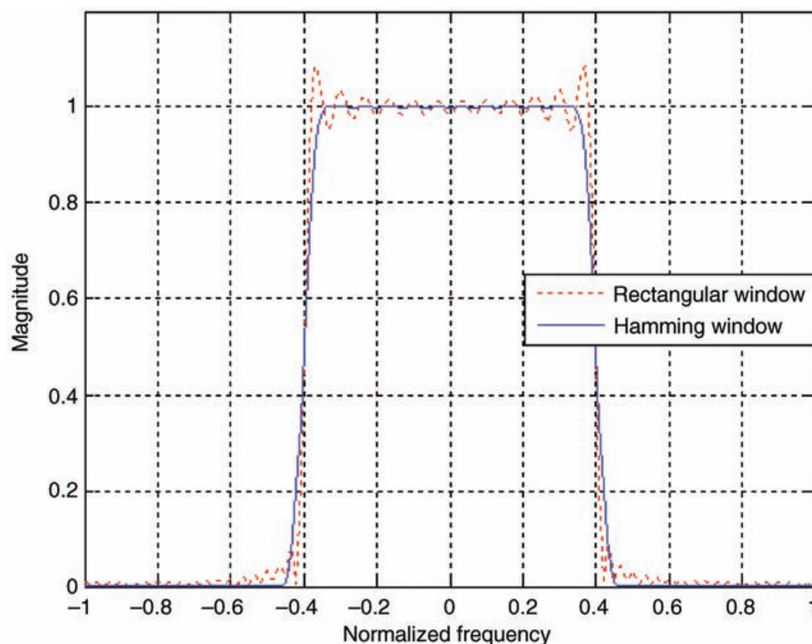


Figure 3.13 Magnitude responses of designed lowpass filter using rectangular and Hamming windows, $L = 61$

5. Truncate the ideal impulse response of infinite length using (3.27) to obtain $h'(n), -M \leq n \leq M$.
6. Make the filter causal by shifting M units to the right using (3.28) to obtain $b'_l, l = 0, 1, \ldots, L-1$.
7. Multiply the window coefficients obtained in step 3 and the impulse response obtained in step 6 to obtain the following filter coefficients:

$$b_l = b'_l w(l), \quad l = 0, 1, \ldots, L-1. \tag{3.36}$$

Applying a window to an FIR filter's impulse response has the effect of smoothing the resulting filter's magnitude response. A symmetric window will preserve a symmetric FIR filter's linear phase response.

MATLAB® provides a GUI tool called the Window Design & Analysis Tool (WinTool) that allows users to design and analyze windows. It can be activated by entering the following command in the MATLAB® command window:

```
wintool
```

It opens with a default 64-point Hamming window. With this tool, we can evaluate different windows such as Blackman, Chebyshev, and Kaiser windows. Another GUI tool provided by MATLAB® for analyzing windows is `wvtool`, the window visualization tool.

3.2.4 Design of FIR Filters Using MATLAB®

FIR filter design algorithms use iterative optimization techniques to minimize the error between the desired and actual frequency responses. The most widely used algorithm is the Parks–McClellan algorithm for designing optimum linear phase FIR filters. This algorithm spreads out the error to produce equal-magnitude ripples. In this section, we consider only the design methods and filter functions available in the MATLAB® *Signal Processing Toolbox*, which are summarized in Table 3.1. The MATLAB® *DSP System Toolbox* (*Filter Design Toolbox* in older versions) provides more advanced FIR filter design methods.

For example, the `fir1` and `fir2` functions design FIR filters using the windowed Fourier series method. The function `fir1` designs FIR filters using the Hamming window as

```
b=fir1(L, Wn);
```

Table 3.1 List of FIR filter design methods and functions available in MATLAB®

Design methods	Filter functions	Description
Windowing	`fir1, fir2, kaiserord`	Truncated Fourier series with windowing methods
Multiband with transition bands	`firls, firpm, firpmord`	Equiripple or least squares approach
Constrained least squares	`fircls, fircls1`	Minimize squared integral error over entire frequency range
Arbitrary response	`cfirpm`	Arbitrary responses

where Wn is the normalized cutoff frequency between 0 and 1 ($\omega = \pi$). The function fir2 designs FIR filters with arbitrarily shaped magnitude response as

```
b=fir2(L, f, m);
```

where the frequency response is specified by the vectors f and m that contain the frequency and magnitude, respectively. The frequencies in f must be within $0 < \mathtt{f} < 1$ in increasing order.

A more efficient algorithm designs optimum linear phase FIR filters based on the Parks–McClellan algorithm, which uses the Remez exchange algorithm and Chebyshev approximation theory to design filters with an optimal fit between the desired and actual frequency responses. The firpm function (remez in older versions) has the following syntax:

```
b = firpm(L, f, m);
```

This function returns a linear phase FIR filter with length $\mathtt{L}+1$ which has the best approximation to the desired frequency response described by f (the vector of frequency band edges in pairs, in ascending order between 0 and 1) and m (the real vector with the same size as f which specifies the desired amplitude of the magnitude response of the resultant filter b). This function will be used to design an FIR filter for experiment in Section 3.5.1.

Example 3.10

Design a linear phase FIR bandpass filter of length 18 with the passband from normalized frequency 0.4 to 0.6.

This filter can be designed using MATLAB® script example3_10.m. The desired and obtained actual magnitude responses are shown in Figure 3.14.

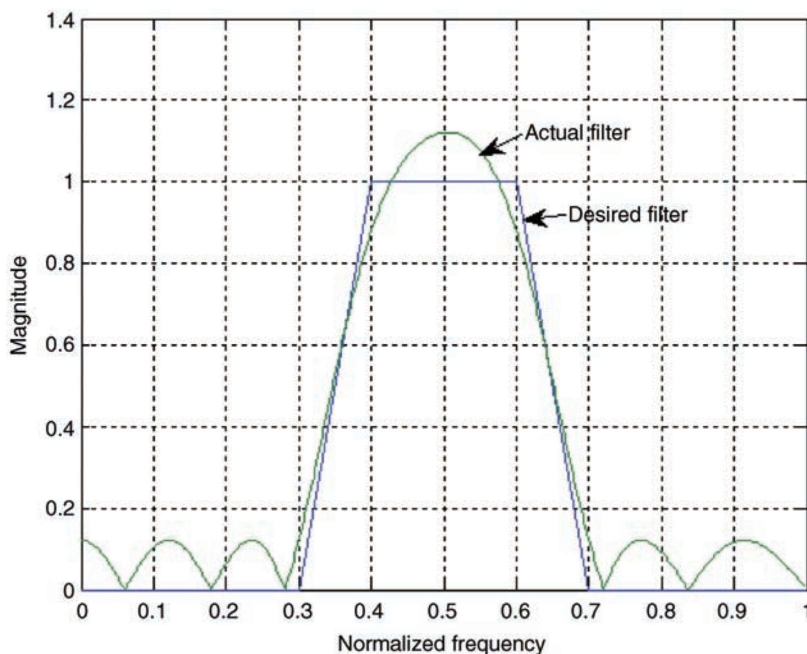


Figure 3.14 Magnitude responses of the desired and actual FIR filters

3.2.5 Design of FIR Filters Using the FDATool

The Filter Design and Analysis Tool (FDATool) is a GUI for designing, quantizing, and analyzing digital filters. It includes a number of advanced filter design techniques and supports all the filter design methods in the *Signal Processing Toolbox*. This tool has the following uses:

1. Design filters by setting filter specifications.
2. Analyze designed filters.
3. Convert filters to different structures.
4. Quantize and analyze quantized filters.

In this section, we briefly introduce the FDATool for designing and quantizing FIR filters.
We can open the FDATool by typing

```
fdatool
```

in the MATLAB® command window. The **Filter Design & Analysis Tool** window is shown in Figure 3.15. We can choose from several response types: **Lowpass**, **Highpass**, **Bandpass**,

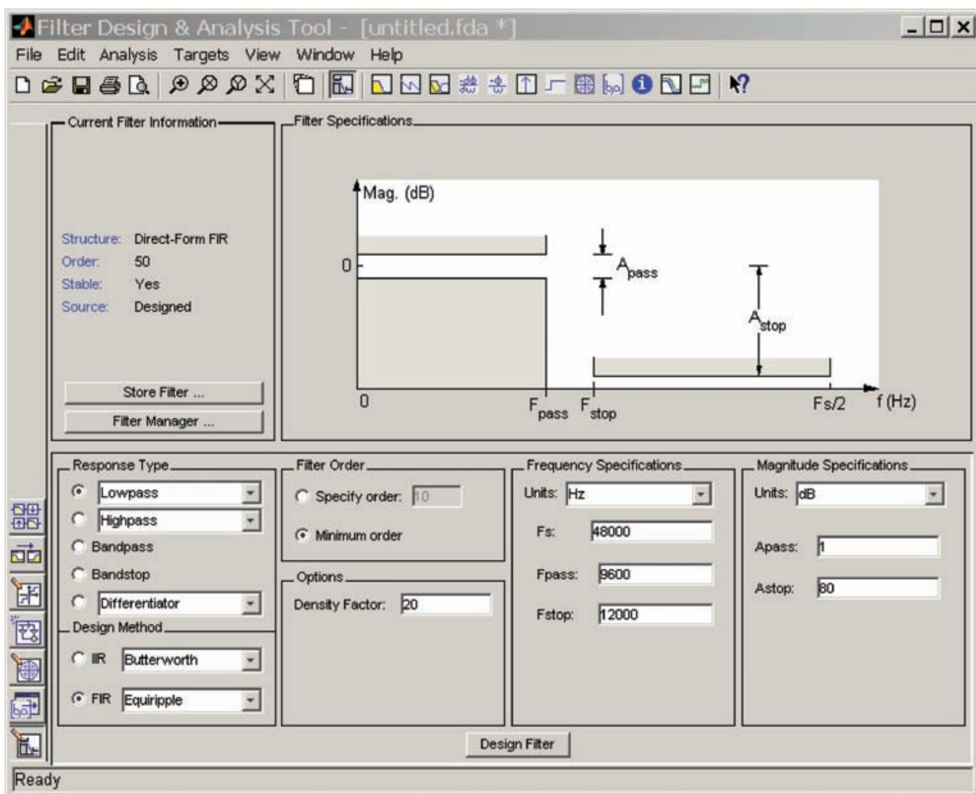


Figure 3.15 FDATool window

Bandstop, and **Differentiator**. For example, to design a bandpass filter, select the radio button next to **Bandpass** in the **Response Type** region on the GUI. It has multiple options for **Lowpass**, **Highpass**, and **Differentiator** types.

It is important to compare the **Filter Specifications** region in Figure 3.15 to Figure 3.4. The parameters $\mathbf{F_{pass}}$, $\mathbf{F_{stop}}$, $\mathbf{A_{pass}}$, and $\mathbf{A_{stop}}$ correspond to ω_p, ω_s, A_p, and A_s, respectively. These parameters can be entered in the **Frequency Specifications** and **Magnitude Specifications** regions. The frequency units are **Hz** (default), **kHz**, **MHz**, or **GHz** and the magnitude options are **dB** (default) or **Linear**.

Example 3.11

Design a lowpass FIR filter with the following specifications:

Sampling frequency $\mathrm{F_s} = 8\,\mathrm{kHz}$,
Passband cutoff frequency $\mathrm{F_{pass}} = 2\,\mathrm{kHz}$,
Stopband cutoff frequency $\mathrm{F_{stop}} = 2.5\,\mathrm{kHz}$,
Passband ripple $\mathrm{A_{pass}} = 1\,\mathrm{dB}$,
Stopband attenuation $\mathrm{A_{stop}} = 60\,\mathrm{dB}$.

We can easily design this filter by entering parameters in the **Frequency Specifications** and **Magnitude Specifications** regions as shown in Figure 3.16. Press the **Design Filter** button to compute the filter coefficients. The **Filter Specifications** region will show the **Magnitude Response (dB)** of the designed filter. We can analyze different characteristics of the designed filter by clicking on the **Analysis** menu. For example, selecting the **Impulse Response** available in the menu opens the new **Impulse Response** window to display the designed FIR filter coefficients.

We have two options for determining the filter order: we can either specify the filter order by **Specify order**, or use the default **Minimum order**. In Example 3.11, we use the default minimum order, and the order (31) is shown in the **Current Filter Information** region. Note that order $= 31$ means the length of the FIR filter is $L = 32$.

Once the filter has been designed (using 64-bit double-precision floating-point arithmetic and representation) and verified, we can turn on the quantization mode by clicking the **Set**

Figure 3.16 Frequency and magnitude specifications for a lowpass filter

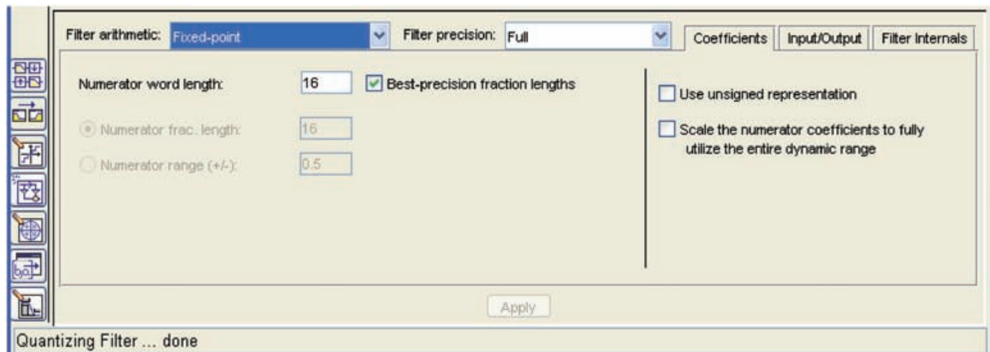
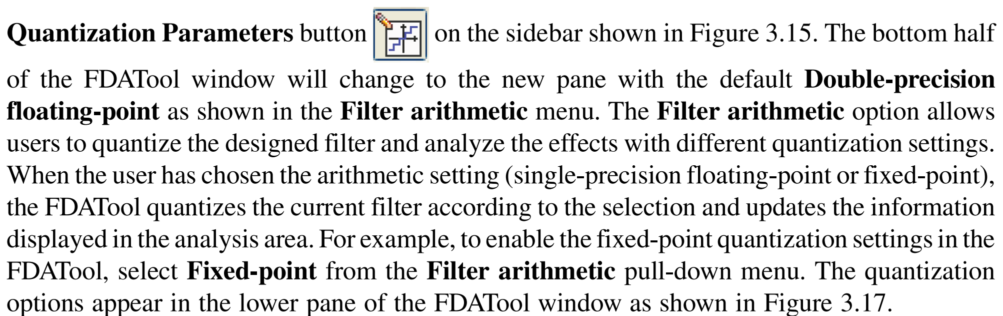


Figure 3.17 Setting fixed-point quantization parameters in the FDATool



Quantization Parameters button on the sidebar shown in Figure 3.15. The bottom half of the FDATool window will change to the new pane with the default **Double-precision floating-point** as shown in the **Filter arithmetic** menu. The **Filter arithmetic** option allows users to quantize the designed filter and analyze the effects with different quantization settings. When the user has chosen the arithmetic setting (single-precision floating-point or fixed-point), the FDATool quantizes the current filter according to the selection and updates the information displayed in the analysis area. For example, to enable the fixed-point quantization settings in the FDATool, select **Fixed-point** from the **Filter arithmetic** pull-down menu. The quantization options appear in the lower pane of the FDATool window as shown in Figure 3.17.

In this figure, there are three tabs in the dialog window for users to select quantization tasks from the FDATool:

1. The **Coefficients** tab defines the coefficient quantization.
2. The **Input/Output** tab quantizes the input and output signals.
3. The **Filter Internals** tab sets a variety of options for the arithmetic.

After setting the proper options for the desired filter, click on **Apply** to start the quantization processes.

The **Coefficients** tab is the default active pane. The filter type and structure determine the available options. **Numerator word length** sets the wordlength used to represent the coefficients of FIR filters. Note that the **Best-precision fraction lengths** box is also checked and the **Numerator word length** box is set to 16 by default. We can uncheck the **Best-precision fraction lengths** box to specify **Numerator frac. length** or **Numerator range (+/−)**.

The filter coefficients can be exported to the MATLAB® workspace as a coefficient file or MAT-file. To save the quantized filter coefficients as a text file, select **Export** from the **File** menu on the toolbar. When the **Export** dialog box appears, select **Coefficient File (ASCII)** from the **Export to** menu and choose **Decimal**, **Hexadecimal**, or **Binary** from the **Format** options. After clicking on the **OK** button, the **Export Filter Coefficients to .FCF file** dialog box will appear. Enter a filename and click on the **Save** button.

To create a C header file containing filter coefficients, select **Generate C header** from the **Targets** menu. For the FIR filters, variables used in the C header file are for numerator name

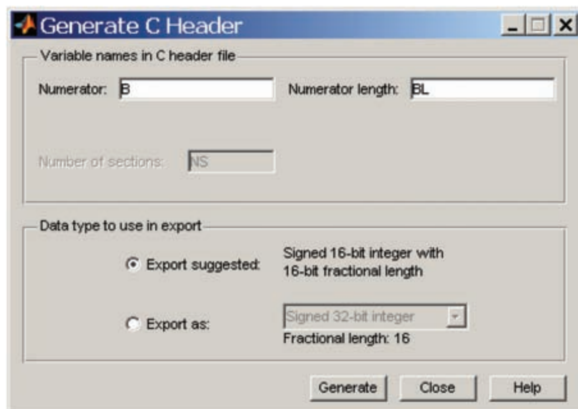


Figure 3.18 Generate C header dialog box

and length. We can use the default variable names `B` and `BL` as shown in Figure 3.18 in C, or change them according to the variable names defined in C that will include this header file. As shown in the figure, we can use the default **Signed 16-bit integer with 16-bit fractional length**, or select **Export as** and choose the desired data type. Clicking on the **Generate** button opens the **Generate C Header** dialog box. Enter the filename and click on **Save** to save the file.

3.3 Implementation Considerations

This section discusses finite-wordlength effects of digital FIR filters, and uses MATLAB® and C to illustrate some important software implementation issues.

3.3.1 Quantization Effects in FIR Filters

Consider the FIR filter defined in (2.36). The filter coefficients, b_l, are obtained from a filter design package such as MATLAB®. These coefficients are usually designed and represented by double-precision floating-point numbers and have to be quantized for implementation on fixed-point processors. The filter coefficients are quantized and analyzed during the design process. If the quantized filter no longer meets the given specifications, we will optimize, redesign, restructure, and/or use more bits to satisfy the specifications.

Let b'_l denote the quantized values corresponding to b_l. As discussed in Chapter 2, the nonlinear quantization can be modeled as the linear operation expressed as

$$b'_l = Q[b_l] = b_l + e(l), \tag{3.37}$$

where $e(l)$ is the quantization error and can be assumed to be a uniformly distributed random noise of zero mean.

The frequency response of the actual FIR filter with quantized coefficients b'_l can be expressed as

$$B'(\omega) = B(\omega) + E(\omega), \tag{3.38}$$

where

$$E(\omega) = \sum_{l=0}^{L-1} e(l)\mathrm{e}^{-\mathrm{j}\omega l} \tag{3.39}$$

represents the error in the desired frequency response $B(\omega)$. The error spectrum is bounded by

$$|E(\omega)| = \left|\sum_{l=0}^{L-1} e(l)\mathrm{e}^{-\mathrm{j}\omega l}\right| \leq \sum_{l=0}^{L-1} |e(l)|\left|\mathrm{e}^{-\mathrm{j}\omega l}\right| \leq \sum_{l=0}^{L-1} |e(l)|. \tag{3.40}$$

As shown in (2.83),

$$|e(l)| \leq \frac{\Delta}{2} = 2^{-B}. \tag{3.41}$$

Thus, (3.40) becomes

$$|E(\omega)| \leq 2^{-B}L. \tag{3.42}$$

This bound is too conservative because it can only be reached if all errors $e(l)$ are of the same sign and have the maximum value in the range. A more realistic bound can be derived assuming $e(l)$ are statistically independent random variables.

Example 3.12

This example first uses the least-square method to design the FIR filter. To convert it to the fixed-point FIR filter, we use the filter construction function `dfilt` and change the arithmetic setting for the filter to fixed-point arithmetic as follows:

```
hd = dfilt.dffir(b); % Create the direct-form FIR filter
set(hd,'Arithmetic','fixed');
```

The first function returns the digital filter object, `hd`, of type `dffir` (direct-form FIR filter). We can use the MATLAB® FVTool (Filter Visualization Tool) to plot the magnitude responses of both the quantized filter and the corresponding reference filter.

The fixed-point filter object `hd` uses 16 bits to represent the filter coefficients. We can make several copies of the filter using different wordlengths. For example, we can use 12 bits to represent the filter coefficients as follows:

```
h1 = copy(hd);                       % Copy hd to h1
set(h1,'CoeffWordLength',12); % Use 12 bits for coefficients
```

The MATLAB® script is given in `example3_12.m`.

3.3.2 MATLAB® Implementations

For evaluation purposes, it is convenient to use a powerful software package such as MATLAB® for software implementation and testing of digital filters. MATLAB® provides the function `filter` (or `filtfilt`) for FIR and IIR filtering. The basic form of this function is

```
y = filter(b, a, x);
```

For FIR filtering, the vector `a` = 1 and filter coefficients b_l are contained in the vector `b`. The input vector is `x` while the filter output vector is `y`.

Example 3.13

A 1.5 kHz sine wave with sampling rate 8 kHz is corrupted by white noise. This noisy signal can be generated, saved in file `xn_int.dat`, and plotted by MATLAB® script `example3_13.m`. In the program, the signal samples are normalized from the original floating-point numbers and saved in the Q15 integer format using the following MATLAB® commands:

```
xn_int = round(32767*in./max(abs(in))); % Normalize to 16-bit integer
fid = fopen('xn_int.dat','w');          % Save signal to xn_int.dat
fprintf(fid,'%4.0f\n',xn_int);          % Save in integer format
```

FIR filter coefficients can be exported to the current MATLAB® workspace by selecting **File** → **Export**. From the pop-up dialog box **Export**, type `b` in the **Numerator** box, and click on **OK**. This saves the filter coefficients in vector `b`, which is available for use in the current MATLAB® directory. Now, we can perform FIR filtering using the MATLAB® function `filter` by the command:

```
y = filter(b, 1, xn_int);
```

The filter output is saved as vector `y` in the workspace, which can be plotted to compare to the input waveform.

Example 3.14

This example evaluates the accuracy of the 16-bit fixed-point filter as compared to the original double-precision floating-point (64-bit) version using random data as the input signal. A quantizer is used to generate uniformly distributed white-noise data using the Q15 format as follows:

```
rand('state',0);         % Initializing the random number generator
q = quantizer([16,15],'RoundMode','round');
xq = randquant(q,256,1); % 256 samples in the range [-1,1)
xin = fi(xq,true,16,15);
```

Now `xin` is the array of integers with 256 members, represented as a fixed-point object (a `fi` object). Now we perform the actual fixed-point filtering as follows:

```
y = filter(hd,xin);
```

The complete MATLAB® program is given in `example3_14.m`.

3.3.3 Floating-Point C Implementations

The FIR filter implementation usually begins with floating-point C, migrates to fixed-point C, and then to the assembly program if needed.

Example 3.15

The input data is denoted as x and the filter output is y. The filter coefficients are stored in the coefficient array h[]. The filter tapped-delay line (signal vector) w[] keeps the past data samples. The sample-by-sample floating-point C program is listed as follows:

```
void floatPointFir(float *x, float *h, short order, float *y, float *w)
{
    short i;
    float sum;

    w[0] = *x++;                       // Get current data to delay line
    for (sum=0, i=0; i<order; i++)     // FIR filtering loop
    {
       sum += h[i] * w[i];             // Sum of products
    }
    *y++ = sum;                        // Save filter output

    for (i=order-1; i>0; i-)           // Update signal buffer
    {
       w[i] = w[i-1];
    }
}
```

The signal buffer w[] is updated every sampling period as shown in Figure 3.8. For each update process, the oldest sample at the end of the signal buffer is discarded and the remaining samples are shifted one location down in the buffer. The most recent data sample $x(n)$ is inserted into the top location at w[0]. With the circular addressing mode available on most digital signal processors such as the C55xx, shifting the data in the signal buffer can be replaced with circular buffers to improve efficiency.

It is more efficient to implement DSP algorithms using a block processing technique. For many practical applications such as wireless communications, speech processing, and audio compression, the signal samples are often grouped into blocks or frames. An FIR filter that processes data by frames instead of sample by sample is called a block FIR filter.

Example 3.16

The block FIR filtering function processes one block of data samples for each function call. The input samples are stored in the array x[] and the filtered output is stored in the array y[]. In the following C program, the block size is denoted as blkSize:

```
void floatPointBlockFir(float *x, short blkSize, float *h, short order,
                        float *y, float *w, short *index)
```

```
{
  short i,j,k;
  float sum;
  float *c;

  k = *index;
  for (j=0; j<blkSize; j++)        // Block processing
  {
    w[k] = *x++;                   // Get current data to delay line
    c = h;
    for (sum=0, i=0; i<order; i++) // FIR filter processing
    {
      sum += *c++ * w[k++];
      k %= order;                  // Simulate circular buffer
    }
    *y++ = sum;                    // Save filter output
    k = (order + k - 1)%order;     // Update index for next time
  }
  *index = k;                      // Update circular buffer index
}
```

3.3.4 Fixed-Point C Implementations

The fixed-point implementation using fractional representation was introduced in Chapter 2. The Q15 format is commonly used by fixed-point digital signal processors. In this section, we use an example to introduce fixed-point C implementations of FIR filtering.

Example 3.17

For fixed-point implementation, we use the Q15 format to represent data samples in the range of -1 to $1–2^{-15}$. The ANSI C compiler requires the data type defined as `long` to ensure that the product can be saved as left-aligned 32-bit data. When saving the filter output, the 32-bit temporary variable `sum` is shifted 15 bits to the right to simulate the conversion from 32-bit to 16-bit Q15 data after multiplication. The fixed-point C code is listed as follows:

```
void fixedPointBlockFir(short *x, short blkSize, short *h, short order,
                        short *y, short *w, short *index)
{
  short i,j,k;
  long sum;
  short *c;

  k = *index;
  for (j=0; j<blkSize; j++)        // Block processing
  {
    w[k] = *x++;                   // Get current data to delay line
    c = h;
    for (sum=0, i=0; i<order; i++) // FIR filter processing
    {
```

```
        sum += *c++ * (long)w[k++];
        if (k == order)                  // Simulate circular buffer
            k = 0;
      }
      *y++ = (short)(sum>>15);           // Save filter output
      if (k-- <=0)                       // Update index for next time
      k = order-1;
    }
    *index = k;                          // Update circular buffer index
  }
```

3.4 Applications: Interpolation and Decimation Filters

In multi-rate signal processing applications, such as interconnecting DSP systems operated at different sampling rates, changing sampling frequency is necessary. The process of converting digital signals from the original sampling rate to different sampling rates is called sampling rate conversion. The key operation for sampling rate conversion is lowpass FIR filtering.

Increasing the sampling rate by an integer factor U is called interpolation (or up-sampling), while decreasing by an integer factor D is called decimation (or down-sampling). The combination of interpolation and decimation with proper U and D factors allows the digital system to change the sampling rate to any ratio. For example, in an audio system that uses oversampling and decimation, the analog input can be first filtered by a low-cost analog antialiasing filter and then sampled at a higher rate. The decimation process is then used to reduce the sampling rate of the oversampled digital signal. In this application, the digital decimation filter provides high-quality lowpass filtering and reduces the cost of using expensive analog filters to reduce the overall system cost.

3.4.1 Interpolation

Interpolation can be done by inserting additional zero samples between successive samples of the original low-rate signal and then applying a lowpass (interpolation) filter on the interpolated samples. For the interpolation of ratio 1:U, $(U-1)$ zeros are inserted in between the successive samples of the original signal $x(n)$ of sampling rate f_s, thus the sampling rate is increased to Uf_s, or the original sampling period T is reduced to T/U. This intermediate signal, $x(n')$, is then filtered by a lowpass filter to produce the final interpolated signal $y(n')$ at the sampling rate of Uf_s.

The most popular lowpass filter used for interpolation is a linear phase FIR filter. The FDATool introduced in Section 3.2.5 can be used to design interpolation filters. The interpolation filter $B(z)$ operates at the high sampling rate $f'_s = Uf_s$ with the ideal frequency response

$$B(\omega) = \begin{cases} U, & 0 \le \omega \le \omega_c \\ 0, & \omega_c < \omega \le \pi, \end{cases} \tag{3.43}$$

where the cutoff frequency is determined as

$$\omega_c = \frac{\pi}{U} \quad \text{or} \quad f_c = f'_s/2U = f_s/2. \tag{3.44}$$

Since the insertion of $(U-1)$ zeros spreads the energy of each signal sample over U output samples, the gain U compensates for the energy loss of the up-sampling process. The interpolation increases the sampling rate; however, the effective bandwidth of the interpolated signal is still the same as the original signal $(f_s/2)$, as shown in (3.44).

Assume the interpolation filter is an FIR filter of length L where L is a multiple of U. Because the interpolation introduces $(U-1)$ zeros between successive samples of the input signal, the required filtering operations may be rearranged to multiply only the non-zero samples. Suppose these non-zero samples are multiplied by the corresponding FIR filter coefficients $b_0, b_U, b_{2U}, \ldots, b_{L-U}$ at time n. At the following time $n(+1$, the non-zero samples are multiplied by filter coefficients $b_1, b_{U+1}, b_{2U+1}, \ldots, b_{L-U+1}$. This alternate process can be accomplished by replacing the high-rate FIR filter $B(z)$ of length L with U filters, $B_m(z)$, $m=0, 1, \ldots, U-1$, of length L/U operated at the low (original) rate f_s. The computational efficiency of the filter structure comes from dividing the single L-point FIR filter into U smaller filters of length L/U, and each filter operates at the lower sampling rate f_s. Furthermore, these U filters share a single signal buffer of size L/U to further reduce memory requirements.

Example 3.18

Given the signal file `wn8kHz.dat`, which is sampled at 8 kHz, MATLAB® script `example3_18.m` can be used to interpolate it to 48 kHz. Figure 3.19 shows the spectra of (a) the original signal, (b) the interpolated signal (by an interpolation factor of 6) before lowpass filtering, and (c) after lowpass filtering. This example clearly demonstrates that the lowpass filtering removes all folded image spectra as shown in Figure 3.19(b). Comparing the spectra of the original signal (a) and the final interpolated signal (c), interpolation increases the sampling rate without increasing the bandwidth (frequency contents) of the signal. Some useful MATLAB® functions used in this example will be presented in Section 3.4.4.

3.4.2 Decimation

Decimation of a signal of sampling rate f_s' by a factor D results in the lower sampling rate $f_s'' = f_s'/D$. The down-sampling by an integer factor of D may be simply done by discarding $(D-1)$ samples for every D samples. However, decreasing the sampling rate by the factor D reduces the bandwidth by the same factor D. Thus, if the original high-rate signal has frequency components outside the new reduced bandwidth, aliasing will occur. This aliasing problem can be solved by lowpass filtering the original signal $x(n')$ prior to the decimation process. The cutoff frequency of the lowpass filter is given as

$$f_c = f_s''/2 = f_s'/2D. \tag{3.45}$$

This lowpass filter is called the decimation filter. The decimation filter output $y(n')$ is down-sampled to obtain the desired low-rate decimated signal $y(n'')$.

The decimation filter operates at the high rate f_s'. However, because only every Dth output sample of the filter is needed, it is unnecessary to compute other $(D-1)$ output samples that will be discarded. Therefore, the overall computation can be reduced by the factor of D.

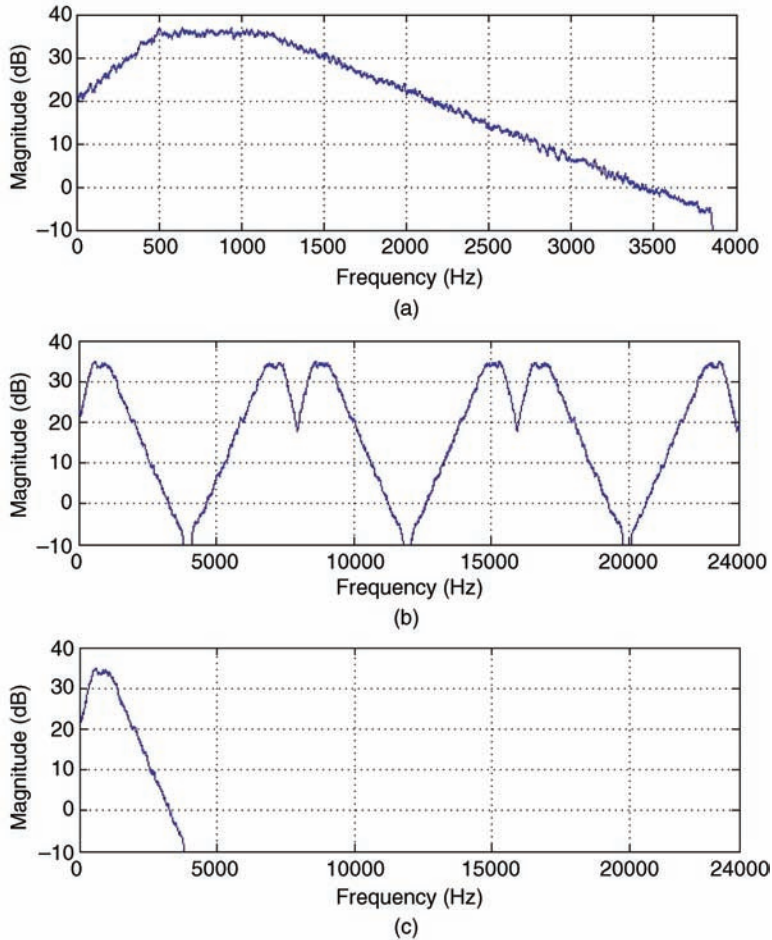


Figure 3.19 Spectra of interpolation signals: (a) original signal; (b) interpolated (by 6) signal before lowpass filtering; and (c) after lowpass filtering

Example 3.19

The signal file `wn48kHz.dat` is sampled at 48 kHz. MATLAB® script `example3_19.m` can be used to decimate it to 8 kHz. Figure 3.20 shows the spectra of (a) the original signal, (b) decimated by 6 without lowpass filtering, and (c) with lowpass filtering before the decimation process. The spectrum shown in Figure 3.20(b) is distorted especially in the low-magnitude segments. Figure 3.20(c) represents the spectrum from 0 to 4 kHz of Figure 3.20(a) because the high-frequency contents are eliminated by the lowpass filter. This example shows the lowpass filtering before discarding the $(D-1)$ samples to prevent aliasing, thus the decimation process reduces the signal bandwidth by the factor D as shown in (3.45).

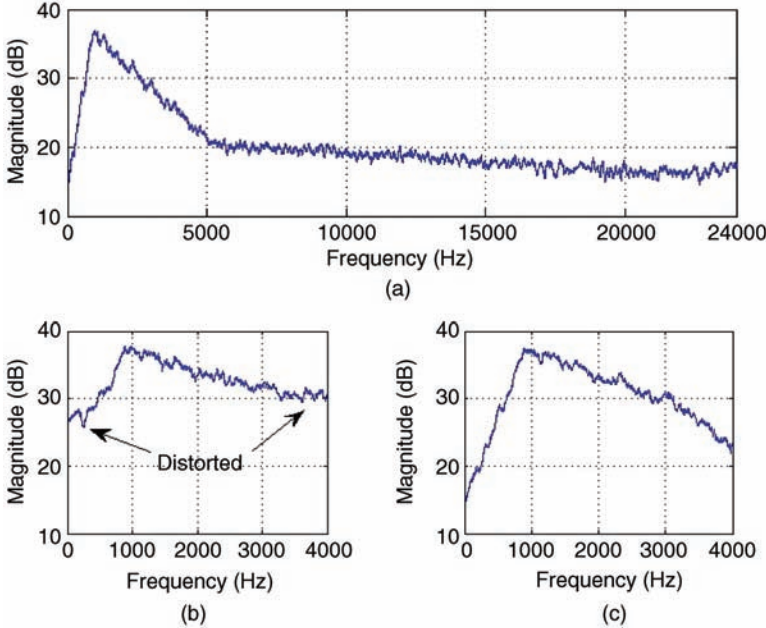


Figure 3.20 Spectra of signals in decimation process: (a) original signal; (b) without lowpass filtering before discarding samples; and (c) with lowpass filtering

3.4.3 Sampling Rate Conversion

The sampling rate conversion by a rational factor U/D can be done entirely in the digital domain with proper interpolation and decimation factors. To minimize the reduction of signal bandwidth, we must do this by first performing interpolation of the factor U, and then decimating the interpolated signal by the factor D. For example, we can convert the broadcast (32 kHz) digital audio signals to professional audio (48 kHz) using a factor of $U/D = 3/2$. That is, we first interpolate the 32 kHz signal (16 kHz bandwidth) with $U = 3$, and then decimate the resulting 96 kHz signal with $D = 2$ to obtain the desired 48 kHz (with 16 kHz bandwidth). It is very important to note that we have to perform interpolation before decimation in order to preserve the desired spectral characteristics. Otherwise, decimation may remove part of the high-frequency components that cannot be recovered by interpolation. For example, if we first decimate the 32 kHz signal (16 kHz bandwidth) with $D = 2$ and then interpolate by $U = 3$, the final bandwidth will be reduced to 8 kHz by the decimation with $D = 2$.

The interpolation filter has the cutoff frequency given in (3.44), and the cutoff frequency of the decimation filter is given in (3.45). Because the interpolation lowpass filter is applied to the signal samples after inserting zeros and the decimation lowpass filter is applied before discarding the signal samples, these two filters can be combined into a single lowpass filter. Thus the frequency response of the lowpass filter for sampling rate conversion should ideally have the cutoff frequency

$$f_c = \frac{1}{2} \min \left(f_s, f_s''\right). \tag{3.46}$$

Example 3.20

Change the sampling rate of a sine wave from 48 kHz to 44.1 kHz using the MATLAB® script `example3_20.m` (adapted from the MATLAB® **Help** menu for `upfirdn`).

For sampling rate conversion, the MATLAB® function `gcd` can be used to find the conversion factor *U*/*D*. For example, to find the factors *U* and *D* for converting the audio signal from a CD (44.1 kHz) for transmission using telecommunication channels (8 kHz), the following commands can be used:

```
g = gcd(8000, 44100);   % Find the greatest common divisor
U = 8000/g;             % Up-sampling factor
D = 44100/g;            % Down-sampling factor
```

In this example, we obtain $U = 80$ and $D = 441$ since `g` = 100.

3.4.4 MATLAB® Implementations

The interpolation process introduced in Section 3.4.3 can be implemented by the MATLAB® function `interp` with the following syntax:

```
y = interp(x, U);
```

The interpolated signal vector `y` is `U` times longer than the original input vector `x`.

The decimation for decreasing the sampling rate of a given signal can be implemented using the MATLAB® function `decimate` with the following syntax:

```
y = decimate(x, D);
```

This function uses an eighth-order lowpass Chebyshev type I IIR filter by default. We can employ the FIR filter by using the following syntax:

```
y = decimate(x, D, 'fir');
```

This command uses the 30-order FIR filter generated by `fir1(30, 1/D)` for lowpass filtering of the data. We can also specify the FIR filter of order *L* by using `y = decimate (x, D, L, 'fir')`.

Example 3.21

Given the speech file `timit_4.asc`, which is digitized by a 16-bit ADC with a sampling rate of 16 kHz, the following MATLAB® script (`example3_21.m`) can be used to decimate it by a factor of 8 to 2 kHz:

```
load timit_4.asc -ascii;                    % Load speech file
soundsc(timit_4, 16000);                    % Play at 16 kHz

timit2 = decimate(timit_4,8,60,'fir');      % Decimation by 8
soundsc(timit2, 2000);                      % Play the decimated speech
```

We can tell the sound-quality (bandwidth) difference by listening to the speech file `timit_4` with 8 kHz bandwidth and the file `timit2` with 1 kHz bandwidth. In the MATLAB® script, we also interpolate the original 16 kHz signal to 48 kHz and play the high-rate speech. Note that the sound quality of the interpolated 48 kHz signal is the same as

the original 16 kHz signal since they both have the same bandwidth of 8 kHz. Note also that, in the MATLAB® script, we use the function `soundsc`, which performs an auto-scaling of the signal to the maximum level without clipping.

The sampling rate conversion algorithm is supported by the MATLAB® function `upfirdn`. This function implements the efficient polyphase filtering technique. For example, the following command can be used for sampling rate conversion:

```
y = upfindn(x, b, U, D);
```

This function first interpolates the signal in the vector `x` with the factor `U`, filters the intermediate resulting signal by the FIR filter given in the coefficient vector `b`, and finally decimates the intermediate result using the factor `D` to obtain the final output vector `y`. The quality of the sampling rate conversion result will depend on the quality of the FIR filter.

Another function that performs sampling rate conversion is `resample`. For example,

```
y = resample(x, U, D);
```

This function converts the sequence in the vector `x` to the sequence in the vector `y` with the conversion ratio *U/D*. It implements the FIR lowpass filter using `firls` with a Kaiser window.

MATLAB® also provides the function `intfilt` for designing interpolation (and decimation) FIR filters. For example,

```
b = intfilt(U, L, alpha);
```

designs a linear phase lowpass FIR filter with an interpolation ratio of 1:U, and saves the filter coefficients in the vector `b`. The length of `b` is 2`UL` − 1, and the bandwidth of the filter is `alpha` times the Nyquist frequency, where `alpha` = 1 results in the full Nyquist interval.

3.5 Experiments and Program Examples

In this section, the C5505 eZdsp is used for FIR filtering experiments including real-time demonstrations using C and C55xx assembly programs.

3.5.1 FIR Filtering Using Fixed-Point C

This experiment uses the fixed-point C program to implement the block FIR filter presented in Section 3.3.4. The input signal sampled at 8000 Hz is saved in 16-bit Q15 format, which consists of three sinusoidal components at frequencies of 800, 1800, and 3300 Hz. This experiment uses a 48-tap bandpass FIR filter designed using the following MATLAB® script:

```
f=[0 0.3 0.4 0.5 0.6 1]; % Define frequency band edges
m=[0 0 1 1 0 0];         % Define desired magnitude response
b=firpm(47, f, m);       % Design FIR filter with 48 coefficients
```

The MATLAB® function `firpm` designs an equiripple FIR filter with passband from 1600 to 2000 Hz. This bandpass filter will attenuate the 800 and 3300 Hz sinusoids. The input signal file and the 48-tap bandpass filter will be used in experiments Exp3.1 through Exp3.4 to compare different FIR filter implementation techniques. The files used for the experiments are listed in Table 3.2.

Table 3.2 File listing for the experiment Exp3.1

Files	Description
fixedPointBlockFirTest.c	Program for testing block FIR filter
fixedPointBlockFir.c	C function for fixed-point block FIR filter
fixedPointFir.h	C header file
firCoef.h	FIR filter coefficients file
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
input.pcm	Input data file

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the program using the input data file `input.pcm` provided in the data folder.
3. Use MATLAB® to plot the magnitude response of the bandpass FIR filter with filter coefficients in the array `firCoefFixedPoint[]` listed in the header file `firCoef.h`.
4. Compare the filter output to the input by listening to the audio files. Compare the filter performance by plotting the magnitude spectra of both input and output signals. What are the differences between both signals, and why? Does the filter output signal's characteristic agree with the magnitude response of the filter shown in step 3?
5. Profile the FIR filtering program to evaluate its efficiency and identify the most time-consuming operations.
6. Use the FDATool to design an equiripple FIR bandstop filter with the following specifications:

   Sampling frequency $F_s = 8000$ Hz,
   Passband cutoff frequency $F_{pass1} = 1400$ Hz,
   Stopband cutoff frequency $F_{stop1} = 1700$ Hz,
   Stopband cutoff frequency $F_{stop2} = 1900$ Hz,
   Passband cutoff frequency $F_{pass2} = 2200$ Hz,
   Passband ripple $A_{pass1} = 1$ dB,
   Stopband attenuation $A_{stop} = 50$ dB,
   Passband ripple $A_{pass2} = 1$ dB.

   Redo step 4 of the experiment using this filter.

3.5.2 FIR Filtering Using C55xx Assembly Program

As presented in Appendix C, the C55xx processor has MAC (multiply–accumulate) instructions, circular addressing modes, and zero-overhead nested loops to efficiently implement FIR filtering. This experiment implements the FIR filter with the C55xx assembly program using the circular coefficient and signal buffers. The same bandpass filter and input signal file from the previous experiment are used.

This experiment shows that the implementation of an FIR filter using the C55xx assembly program needs order + 4 clock cycles to process each input sample. Thus, the 48-tap filter requires 52 cycles (excluding the overhead) for generating one output sample. Table 3.3 lists the files used for the experiment.

Table 3.3 File listing for the experiment Exp3.2

Files	Description
`blockFirTest.c`	Program for testing block FIR filter
`blockFir.asm`	Assembly program of block FIR filter
`blockFir.h`	C header file
`blockFirCoef.h`	FIR filter coefficients file
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`input.pcm`	Input data file

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the program using the input data file `input.pcm` provided in the data folder.
3. Validate the filtering result using CCS to show that the 800 and 3300 Hz sinusoidal components have been attenuated. Also, verify the experiment by comparing the output file to the result obtained in Exp3.1.
4. Profile the assembly implementation of the FIR filter and compare it to the required clock cycles for the fixed-point C experiment presented in Exp3.1.
5. Repeat the experiment using the filter designed in step 6 of Exp3.1.

3.5.3 Symmetric FIR Filtering Using C55xx Assembly Program

As discussed in Section 3.1.4, the linear phase FIR filter has symmetric or antisymmetric coefficients. The C55xx provides two assembly instructions, `FIRSADD` and `FIRSSUB`, for efficient implementation of the symmetric and antisymmetric FIR filters, respectively. This experiment uses `FIRSADD` to implement a linear phase FIR filter with symmetric coefficients. Since the bandpass FIR filter used for experiments Exp3.1 and Exp3.2 is a symmetric filter, the same FIR filter and input signal file are used for this experiment. Note that only half of the FIR filter coefficients are needed (e.g., 24 coefficients) to implement a symmetric FIR filter.

Two implementation issues should be carefully considered. First, as described in (3.22), the instruction `FIRSADD` adds two corresponding signal samples, which may cause an undesired overflow. Second, the `FIRSADD` instruction needs three reads (one coefficient and two signal samples) in the same cycle, which may cause data bus contention. The first problem can be solved by scaling down the input signal using the Q14 format and scaling up the filter output signal to the Q15 format. The second problem can be resolved by placing the coefficient buffer and the signal buffer in different memory blocks. We can use the `pragma` directive to place program code and data variables into the desired memory locations. In this experiment, the C55xx assembly implementation of the symmetric FIR filter takes (order/2) + 5 clock cycles to process each signal sample. Thus, the 48-tap filter needs 29 cycles to generate one output sample excluding the overhead. Table 3.4 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the program using the input data file `input.pcm` provided in the data folder.

Table 3.4 File listing for the experiment Exp3.3

Files	Description
symFirTest.c	Program for testing symmetric FIR filter
symFir.asm	Assembly routine of symmetric FIR filter
symFir.h	C header file
symFirCoef.h	FIR filter coefficients file
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
input.pcm	Input data file

3. Compare the output signal file to the output file obtained from Exp3.2 by computing the difference between these two results. Is there any difference resulting from using the Q14 data operation required by implementation of the symmetric FIR filter?
4. Use MATLAB® or CCS to plot both the input and output signals in the time domain (e.g., waveform) and frequency domain (e.g., magnitude spectrum) to examine the experiment result.
5. Profile the clock cycles needed for the symmetric FIR filter and compare its efficiency to the previous experiments.
6. Design a 49-tap FIR bandpass filter using the following MATLAB® script:

```
f=[0 0.3 0.4 0.5 0.6 1]; % Define frequency band edges
m=[0 0 1 1 0 0];         % Define desired magnitude response
b=firpm(48,f,m);         % Design FIR filter with 49 coefficients
```

   (a) Modify the symmetric block FIR filter assembly program to implement this symmetric filter with odd-numbered order.
   (b) Write a fixed-point C program to implement this block FIR filter using only the first 25 coefficients.

3.5.4 Optimization Using Dual-MAC Architecture

The efficiency of FIR filtering can be improved by using the dual-MAC architecture of the C55xx. Two output samples $y(n)$ and $y(n+1)$ can be computed in one clock cycle by performing two MAC operations in parallel, see Appendix C for details.

There are three issues to be considered when applying dual-MAC architecture. First, the length of the signal buffer must be increased by one to accommodate an extra memory location required for computing two output signals in parallel. Second, the implementation of the FIR filter using dual MAC requires three memory reads (two signal samples and one filter coefficient) simultaneously. To avoid memory bus contention, the signal buffer and the coefficient buffer must be placed in different memory blocks. Finally, since the dual-MAC computation results are kept in two accumulators, two memory stores are needed to save both output samples. Dual-memory store instructions can be used to save both samples from accumulators to the data memory in one cycle. However, the dual-memory store instruction requires the data to be aligned on an even-word (32-bit) boundary. This alignment can be set

Table 3.5 File listing for the experiment Exp3.4

Files	Description
`dualMacFirTest.c`	Program for testing dual-MAC FIR filter
`dualMacFir.asm`	Assembly routine of dual-MAC FIR filter
`dualMacFir.h`	C header file
`dualMacFirCoef.h`	FIR filter coefficients file
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`input.pcm`	Input data file

using the `DATA_SECTION` pragma and `DATA_ALIGN` pragma directives to tell the linker where to place the output data samples.

The C55xx assembly implementation of the FIR filter using dual MAC needs (order/2) + 5 clock cycles to process each input signal. Thus, the 48-tap FIR filter needs 29 cycles to generate one output sample excluding the overhead. This experiment uses the same bandpass FIR filter and input signal file as the previous experiments. The files used for the experiment are listed in Table 3.5.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the program using the input data file `input.pcm` provided in the data folder.
3. Use MATLAB® or CCS to validate the experiment result by showing the 800 and 3300 Hz sinusoidal components are attenuated. Compare the experiment output signal to the results obtained from previous experiments. Is there any difference resulting from the dual-MAC operations?
4. Profile the dual-MAC FIR filter implementation and compare its efficiency to the results from fixed-point C implementation, assembly implementation, and symmetric FIR filter implementation in the previous experiments.
5. Design a lowpass FIR filter with 24 coefficients using the FDATool. The filter specifications are:

   Sampling frequency $F_s = 8000\,\text{Hz}$,
   Passband cutoff frequency $F_{pass} = 1000\,\text{Hz}$,
   Stopband cutoff frequency $F_{stop} = 1200\,\text{Hz}$,
   Passband ripple $A_{pass} = 1\,\text{dB}$,
   Stopband attenuation $A_{stop} = 50\,\text{dB}$.

   Use this new lowpass filter for the FIR filtering experiment. Compare the output and input signals in both the time domain and frequency domain. Does the filter result meet the filter specifications?
6. Use the FDATool to redesign the FIR lowpass filter that meets the filter specifications given in step 5. What will be the filter order? Redo the experiment and compare the result to step 5.

Table 3.6 File listing for the experiment Exp3.5

Files	Description
`realtimeFIRTest.c`	Program for real-time FIR filtering
`dualMacFir.asm`	Assembly routine for dual-MAC implementation of FIR filter
`firFilter.c`	FIR filter initialization and control
`vector.asm`	Vector table for real-time experiment
`dualMacFir.h`	C header file
`dualMacFirCoef.h`	FIR filter coefficients file
`tistdtypes.h`	Standard type define header file
`dma.h`	Header file for DMA functions
`dmaBuff.h`	Header file for DMA data buffer
`i2s.h`	i2s header file for i2s functions
`Ipva200.inc`	C5505 processor include file
`myC55xUtil.lib`	BIOS audio library
`c5505.cmd`	Linker command file

3.5.5 Real-Time FIR Filtering

In this experiment, the audio loopback experiment presented in Chapter 1 is modified for real-time FIR filtering. Instead of passing the input signal directly to the output for audio loopback, this experiment applies an FIR filter to the input signal and outputs the filtered signal. The FIR filter used for the experiment is a lowpass filer with the following specifications:

Sampling frequency $F_s = 48$ kHz,
Passband cutoff frequency $F_{pass} = 2$ kHz,
Stopband cutoff frequency $F_{stop} = 4$ kHz,
Passband ripple $A_{pass} = 1$ dB,
Stopband attenuation $A_{stop} = 60$ dB.

The sampling rate of the C5505 eZdsp is set at 48 000 Hz. The files used for the experiment are listed in Table 3.6.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Connect a headphone to the eZdsp audio output jack, and connect an audio source to the eZdsp audio input jack. Load and run the program.
3. Listen to the audio playback to verify that the frequency components higher than 2000 Hz are attenuated by the lowpass filter with a cutoff frequency of 2000 Hz.
4. Use MATLAB® to plot the magnitude response of the lowpass FIR filter with the filter coefficients `dualMacFirCoef[]` in the header file `dualMacFirCoef.h`. Compare the obtained passband ripple and stopband attenuation to the filter specifications.
5. Design an antisymmetric bandpass filter using the following MATLAB® script:

```
f=[0 0.03 0.1 0.9 0.97 1]; % Define frequency band edges
m=[0 0 1 1 0 0];           % Define desired magnitude response
b=firpm(47, f, m, 'h');    % 48-tap antisymmetric FIR filter
```

(a) Examine the filter characteristics including the bandwidth, edge frequencies, stopband attenuation, and the passband ripples.
(b) Either modify the symmetric block FIR assembly program for the antisymmetric FIR filter implementation.
(c) Or write a fixed-point C program to implement this antisymmetric FIR filter. Use only 24 filter coefficients for the antisymmetric FIR filter.

Set the sampling frequency of eZdsp to 8000 Hz and redo the experiment using this filter. Pay attention to the overflow problem. Evaluate the experimental results by listening to the original and filtered signals.

3.5.6 Decimation Using C and Assembly Programs

This experiment implements a two-stage decimator with 2:1 and 3:1 decimation ratios using two FIR filters. This decimation experiment uses the input, output, and temporary buffers. The input buffer size equals the output frame size multiplied by the decimation factor. For example, when the frame size is 96, the 48 000 to 8000 Hz decimation requires an input buffer with a size of 576 (96∗6) where 6 is the decimation factor. The temporary buffer size is the input buffer size divided by the first decimation factor (which is 2 in this experiment). Table 3.7 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the program to generate the decimation output using the input data file `tone1k_48000.pcm` provided in the data folder.
3. Use MATLAB® to plot the magnitude spectrum of decimation output to verify that it is a 1000 Hz tone at 8000 Hz sampling frequency. Play the decimation output signal and listen to it.
4. Profile the runtime efficiency of the assembly function for the decimation filter. Rewrite this function using the fixed-point C program. Repeat the experiment and compare its efficiency to the assembly implementation.
5. Design a 3:1 decimator. Modify the experiment to decimate the tone from the sampling rate of 48 000 Hz to 16 000 Hz and verify the decimation result.
6. Can the filter coefficients given in `coef24to8.h` be used to decimate the tone sampled at 48 000 to 16 000 Hz when the sampling rate of the eZdsp is set at 48 000 Hz? Why?

Table 3.7 File listing for the experiment Exp3.6

Files	Description
`decimationTest.c`	Program for testing decimation experiment
`decimate.asm`	Assembly routine of decimation filter
`decimation.h`	C header file
`coef48to24.h`	FIR filter coefficients for 2:1 decimation
`coef24to8.h`	FIR filter coefficients for 3:1 decimation
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`tone1k_48000.pcm`	Data file – 1 kHz tone at 48 kHz sampling rate

Table 3.8 File listing for the experiment Exp3.7

Files	Description
`interpolateTest.c`	Program for testing interpolation experiment
`interpolate.c`	C function for interpolation filter
`interpolation.h`	C header file
`coef8to16.h`	FIR filter coefficients for 1:2 interpolation
`coef16to48.h`	FIR filter coefficients for 1:3 interpolation
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`tone1k_8000.pcm`	Data file – 1 kHz tone at 8 kHz sampling rate

3.5.7 Interpolation Using Fixed-Point C

This experiment implements a two-stage interpolator with interpolation factors 2 and 3 to change the sampling rate of the input signal from 8000 to 48 000 Hz. Two interpolation filters are implemented using the fixed-point C program that mimics the circular addressing mode. The circular buffer index is `index`. The coefficient array is `h[]`, and the signal buffer is `w[]`. Since it is not necessary to filter the inserted zero data samples, the coefficient array pointer uses an index offset equal to the interpolation factor. Table 3.8 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the program to generate the interpolation output using the input data file `tone1k_8000.pcm` provided in the data folder.
3. Verify the output signal is a 1000 Hz tone sampled at 48 000 Hz by counting the number of samples in each cycle. (Hint: a 1000 Hz sine wave at 8000 Hz sampling rate contains eight samples per cycle.)
4. Design a 1:4 interpolator using MATLAB®. Modify the experiment to interpolate the tone from the sampling rate of 8000 Hz to 32 000 Hz. Verify the interpolation result by plotting the magnitude spectrum of the output signal using MATLAB® or CCS.
5. Use an assembly routine to implement the FIR filter for interpolation. (Hint: the key difference between the assembly block FIR filter function `blockFir.asm` in Exp3.2 and the interpolation FIR filter `interpolate.c` is the indexing of the interpolator filter coefficient.) Redo the experiment using the assembly filter function. Verify the result using MATLAB. Profile and compare the efficiency improvement achieved by using the assembly function over the C function.

3.5.8 Sampling Rate Conversion

In this experiment, the sampling rate is converted from 48 000 to 32 000 Hz. To achieve this goal, the signal is first interpolated from 48 000 to 96 000 Hz using a 1:2 interpolator, and then the intermediate result is decimated to 32 000 Hz using a 3:1 decimator. Figure 3.21 illustrates the procedures of sampling rate conversion from 48 000 to 32 000 Hz. The input signal file is dual-tone multi-frequency (DTMF) tones representing “digit 5,” which contains two sinusoids at frequencies 770 and 1336 Hz. The files used for the experiment are listed in Table 3.9.

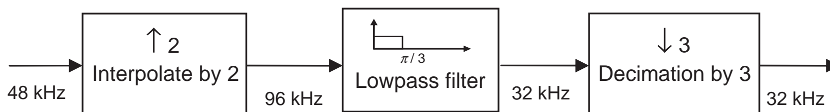


Figure 3.21 Flow diagram of sampling rate conversion

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the program to generate the output data file using the input data file `DTMF5s_48KHz.pcm` provided in the data folder.
3. Use MATLAB® or CCS to plot the magnitude spectrum of the output signal and verify that the sample rate conversion results in the right frequencies of two tones.
4. Use MATLAB® for sampling rate conversion to convert a CD music file from 44 100 Hz sampling rate to 48 000 Hz (Hint: refer to Example 3.20.) Verify the design and convert this MATLAB® experiment to a C5505 eZdsp experiment.

3.5.9 Real-Time Sampling Rate Conversion

This experiment uses the C5505 eZdsp to digitize an analog signal for real-time sampling rate conversion. The conversion is only applied to the left channel of the signal while passing the right-channel signal without processing for comparison purposes. The two-stage interpolation for up-sampling from 8000 to 48 000 Hz and the two-stage decimation down-sampling from 48 000 to 8000 Hz are used in the experiment. The files used for the experiment are listed in Table 3.10.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Connect a headphone to the eZdsp audio output jack and an audio source to the eZdsp audio input jack. Load and run the program.
3. Listen to the eZdsp audio output to evaluate the sampling rate conversion result by comparing the left- and right-channel output audio signals.

Table 3.9 File listing for the experiment Exp3.8

Files	Description
`srcTest.c`	Program for testing sampling rate conversion
`interpolate.c`	C function for interpolation filter
`decimate.asm`	Assembly routine for decimation filter
`interpolation.h`	C header file for interpolation
`decimation.h`	C header file for decimation
`coef48to96.h`	FIR filter coefficients for 1:2 interpolation
`coef96to32.h`	FIR filter coefficients for 3:1 decimation
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`DTMF5s_48KHz.pcm`	Data file – DTMF tone “digit 5” at 48 kHz

Table 3.10 File listing for the experiment Exp3.9

Files	Description
`realtimeSRCTest.c`	Program for testing sampling rate conversion
`rtSRC.c`	Sampling rate conversion function
`interpolate.c`	C function for interpolation filter
`decimate.asm`	Assembly routine for decimation filter
`vector.asm`	Vector table for real-time experiment
`interpolation.h`	C header file for interpolation
`decimation.h`	C header file for decimation
`coef8to16.h`	FIR filter coefficients for 1:2 interpolation
`coef16to48.h`	FIR filter coefficients for 1:3 interpolation
`coef48to24.h`	FIR filter coefficients for 2:1 decimation
`coef24to8.h`	FIR filter coefficients for 3:1 decimation
`tistdtypes.h`	Standard type define header file
`dma.h`	Header file for DMA functions
`dmaBuff.h`	Header file for DMA data buffer
`i2s.h`	i2s header file for i2s functions
`Ipva200.inc`	C5505 processor include file
`myC55xUtil.lib`	BIOS audio library
`c5505.cmd`	Linker command file

4. Design a decimation filter for 48 000 Hz to 24 000 Hz conversion and an interpolation filter for up-sampling 24 000 Hz to 48 000 Hz. Modify the program to use these new filters for real-time sampling rate conversion based on the same mechanism, that is, the left channel for the sample rate conversion signal and the right channel for the pass-through signal. The sampling rate of the eZdsp is set at 48 000 Hz. Compare the experimental results. Is there any noticeable artifact?

Exercises

3.1. Consider the two-point moving-average filter given in Example 3.1. Compute the 3 dB bandwidth of this filter for a sampling rate of 8 kHz.

3.2. Consider the FIR filter with the impulse response $h(n) = \{1, 1, 1\}$. Calculate the magnitude and phase responses, and verify that the filter has linear phase.

3.3. Consider the comb filter given in Example 3.2 with a sampling rate of 8 kHz. A periodic signal with fundamental frequency 500 Hz and harmonics at 1, 1.5, . . . , 3.5 kHz is filtered by this comb filter. Which harmonics will be attenuated? Why?

3.4. Use the graphical interpretation of linear convolution given in Figure 3.6 to compute the linear convolution of $h(n) = \{1, 2, 1\}$ and $x(n)$ defined as follows:
(a) $x(n) = \{1, -1, 2, 1\}$.
(b) $x(n) = \{1, 2, -2, -2, 2\}$.
(c) $x(n) = \{1, 3, 1\}$.
Verify the results using MATLAB®.

3.5. The comb filter can also be described as

$$y(n) = x(n) + x(n-L).$$

Find the transfer function and zeros. Also, compute the magnitude response of this filter using MATLAB® and compare the results to Figure 3.3 (assume $L = 8$).

3.6. Assuming $h(n)$ has the symmetry property $h(n) = h(-n)$ for $n = 0, 1, \ldots, M$, verify that $H(\omega)$ can be expressed as

$$H(\omega) = h(0) + \sum_{n=1}^{M} 2h(n)\cos(\omega n).$$

3.7. The simplest digital approximation of a continuous-time differentiator is the first-order operation defined as

$$y(n) = \frac{1}{T}[x(n) - x(n-1)].$$

Find the transfer function $H(z)$, the magnitude response, and the phase response of the differentiator. Verify the computed results using the frequency response plots in MATLAB®.

3.8. Redraw the signal-flow diagram of the symmetric FIR filter shown in Figure 3.5 and modify (3.22) and (3.23) for the case when L is an odd number.

3.9. Consider FIR filters with the following impulse responses:
(a) $h(n) = \{-4, 1, -1, -2, 5, 0, -5, 2, 1, -1, 4\}$.
(b) $h(n) = \{-4, 1, -1, -2, 5, 6, 5, -2, -1, 1, -4\}$.
Use MATLAB® to plot the magnitude responses, phase responses, and locations of zeros for both filters.

3.10. Sketch the pole–zero diagram and the magnitude response of the lowpass filter

$$H(z) = \frac{1}{L}\left(\frac{1 - z^{-L}}{1 - z^{-1}}\right)$$

for $L = 8$, verify the result using MATLAB®, and compare the result to Figure 3.3.

3.11. Use Examples 3.5 and 3.6 to design and plot the magnitude response of a linear phase FIR highpass filter with cutoff frequency $\omega_c = 0.6\pi$ by truncating the impulse response of the ideal highpass filter to length $L = 2M = 1$ for $M = 32$ and 64. Verify the results using MATLAB®.

3.12. Repeat Problem 3.11 using Hamming and Blackman window functions. Show that the undesired oscillatory behavior can be reduced using the windowed Fourier series method.

3.13. Design a bandpass filter

$$H(f) = \begin{cases} 1, & 1.6\,\text{kHz} \le f \le 2\,\text{kHz} \\ 0, & \text{otherwise} \end{cases}$$

with a sampling rate of 8 kHz and a duration of impulse response of 50 ms using the Fourier series method with different window functions, and verify the result using the MATLAB® function `fir1`. Plot the magnitude and phase responses.

3.14. Repeat Problem 3.13 using the FDATool with different design methods, and compare the results to Problem 3.13.

3.15. Redo Example 3.13, quantize the designed filter coefficients using the fixed-point Q15 format, and save the coefficients in a C header file. Write a floating-point C program to implement this FIR filter and test the result by comparing both input and output signals in terms of time-domain waveforms and frequency-domain spectra. Also, implement this filter using MATLAB®, and compare the output signal to the floating-point C program output.

3.16. Redo Problem 3.15 using the fixed-point (16-bit) C program and plot the error signal (difference) by subtracting the fixed-point C output from the previous floating-point C program to evaluate numerical errors.

3.17. Redo Example 3.13 with different edge frequencies and ripples using the FDATool, and summarize their relationship with the required filter orders. Also, quantize the designed filters using fixed-point arithmetic with coefficient wordlength of 8, 12, and 16 bits.

3.18. List the window functions supported by the MATLAB® Wintool. Also, use this tool to study the Kaiser window with different values of L and β and compare the results to the popular Hamming window using the same L.

3.19. Write MATLAB® and C programs to implements a comb filter with $L = 8$. The program must have the input/output capability. Also, generate a sinusoidal signal that contains frequencies $\omega_1 = \pi/4$ and $\omega_2 = 3\pi/8$. Test the filter using this sinusoidal signal as the input signal. Explain the results based on the distribution of the zeros of the filter and the associate magnitude response.

3.20. Given the speech file `TIMIT_4.ASC` (sampling rate 16 kHz) in the companion software package, decimate it to a sampling rate of 8, 4, 2, and 1 kHz. Use the MATLAB® function `soundsc` to examine different signal bandwidths by listening to the decimated signals.

3.21. Use the sampling rate conversion techniques presented in this chapter to convert the sampling rate of speech file `TIMIT_4.ASC` to 12, 5, and 3 kHz.

3.22. Given the speech file `TIMIT_4.ASC`, decimate it to a sampling rate of 1 kHz, and interpolate it back to 16 kHz. Listen to the output speech and compare the quality to the original signal. Observe, and explain, the differences.

3.23. Given the speech file `TIMIT_4.ASC`, interpolate it to a sampling rate of 48 kHz, and decimate it back to 16 kHz. Listen to the output speech and compare the quality to the output signal obtained by Problem 3.22. Observe, and explain, the differences.

References

1. Ahmed, N. and Natarajan, T. (1983) *Discrete-Time Signals and Systems*, Prentice Hall, Englewood Cliffs, NJ.
2. Ingle, V.K. and Proakis, J.G. (1997) *Digital Signal Processing Using MATLAB V.4*, PWS Publishing, Boston, MA.
3. Kuo, S.M. and Gan, W.S. (2005) *Digital Signal Processors*, Prentice Hall, Upper Saddle River, NJ.
4. Oppenheim, A.V. and Schafer, R.W. (1989) *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
5. Orfanidis, S.J. (1996) *Introduction to Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
6. Proakis, J.G. and Manolakis, D.G. (1996) *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd edn, Prentice Hall, Englewood Cliffs, NJ.
7. Mitra, S.K. (1998) *Digital Signal Processing: A Computer-Based Approach*, 2nd edn, McGraw-Hill, New York.
8. Grover, D. and Deller, J.R. (1999) *Digital Signal Processing and the Microcontroller*, Prentice Hall, Englewood Cliffs, NJ.
9. Taylor, F. and Mellott, J. (1998) *Hands-On Digital Signal Processing*, McGraw-Hill, New York.
10. Stearns, S.D. and Hush, D.R. (1990) *Digital Signal Analysis*, 2nd edn, Prentice Hall, Englewood Cliffs, NJ.
11. The MathWorks, Inc. (2004) Signal Processing Toolbox User's Guide, Version 6, June.
12. The MathWorks, Inc. (2004) Filter Design Toolbox User's Guide, Version 3, October.

4

Design and Implementation of IIR Filters

This chapter presents the theory, design, analysis, realization, and application of digital IIR filters [1–12]. We use experiments to demonstrate the implementation and applications of IIR filters in different forms using fixed-point processors.

4.1 Introduction

Design of a digital IIR filter usually begins with designing an analog filter and applying a mapping technique to transform the analog filter from the s-plane into the z-plane. Therefore, this section briefly reviews the related principles of Laplace transforms, analog filters, mapping properties, and frequency transformation.

4.1.1 Analog Systems

Given a causal continuous-time function, that is, $x(t) = 0$ for $t < 0$, the Laplace transform is defined as

$$X(s) = \int_{-\infty}^{\infty} x(t)e^{-st}\,dt, \tag{4.1}$$

where s is a complex variable defined as

$$s = \sigma + j\Omega, \tag{4.2}$$

and σ is a real number. The inverse Laplace transform is expressed as

$$x(t) = \frac{1}{2\pi j}\int_{\sigma - j\infty}^{\sigma + j\infty} X(s)e^{st}\,ds. \tag{4.3}$$

The integral is evaluated along the straight line $\sigma + j\Omega$ on the complex plane from $\Omega = -\infty$ to $\Omega = \infty$.

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

Equation (4.2) defines the complex s-plane with real axis σ and imaginary axis jΩ. When $\sigma = 0$, the values of s are on the jΩ axis, that is, $s = \mathrm{j}\Omega$, and (4.1) becomes

$$X(s)|_{s=\mathrm{j}\Omega} = \int_{-\infty}^{\infty} x(t)\mathrm{e}^{-\mathrm{j}\Omega t}\,\mathrm{d}t = X(\Omega), \tag{4.4}$$

which is the Fourier transform of the signal $x(t)$. Therefore, given a function $X(s)$, we can evaluate its frequency characteristics by substituting $s = \mathrm{j}\Omega$ if the region of convergence of the Laplace transform includes the imaginary axis jΩ.

If $x(t)$ is the input signal to the linear time-invariant (LTI) system with impulse response $h(t)$, the output signal $y(t)$ can be computed as

$$\begin{aligned} y(t) &= x(t) * h(t) = h(t) * x(t) \\ &= \int_{-\infty}^{\infty} x(\tau)h(t-\tau)\mathrm{d}\tau = \int_{-\infty}^{\infty} h(\tau)x(t-\tau)\mathrm{d}\tau, \end{aligned} \tag{4.5}$$

where $*$ denotes linear convolution. Linear convolution in the time domain is equivalent to multiplication in the Laplace (or frequency) domain, expressed as

$$Y(s) = H(s)X(s), \tag{4.6}$$

where $Y(s)$, $X(s)$, and $H(s)$ are the Laplace transforms of $y(t)$, $x(t)$, and $h(t)$, respectively.

The transfer function of an LTI system is defined as

$$H(s) = \frac{Y(s)}{X(s)} = \int_{-\infty}^{\infty} h(t)\mathrm{e}^{-st}\,\mathrm{d}t. \tag{4.7}$$

The general form of the system transfer function can be expressed by the following rational function:

$$H(s) = \frac{b_0 + b_1 s + \cdots + b_{L-1}s^{L-1}}{a_0 + a_1 s + \cdots + a_M s^M} = \frac{\sum\limits_{l=0}^{L-1} b_l z^{-l}}{\sum\limits_{m=0}^{M} a_m z^{-m}} = \frac{N(s)}{D(s)}. \tag{4.8}$$

The roots of numerator $N(s)$ are the zeros of $H(s)$, the roots of denominator $D(s)$ are the poles of the system, and the order of the system is M. MATLAB® provides the function `freqs` to compute the frequency response of analog system $H(s)$.

Example 4.1

The causal signal $x(t) = \mathrm{e}^{-2t}u(t)$ is applied to an LTI system, and the output of the system is $y(t) = (\mathrm{e}^{-t} + \mathrm{e}^{-2t} - \mathrm{e}^{-3t})u(t)$. Compute the system transfer function $H(s)$ and the impulse response $h(t)$.

From (4.1), we have

$$X(s) = \int_{0}^{\infty} \mathrm{e}^{-2t}\mathrm{e}^{-st}\,\mathrm{d}t = \int_{0}^{\infty} \mathrm{e}^{-(s+2)t}\,\mathrm{d}t = \frac{1}{s+2}.$$

Similarly, we have

$$Y(s) = \frac{1}{s+1} + \frac{1}{s+2} - \frac{1}{s+3} = \frac{s^2+6s+7}{(s+1)(s+2)(s+3)}.$$

From (4.7), we obtain the transfer function

$$H(s) = \frac{Y(s)}{X(s)} = \frac{s^2+6s+7}{(s+1)(s+3)} = 1 + \frac{1}{s+1} + \frac{1}{s+3}.$$

Taking the inverse Laplace transform of the above $H(s)$ term by term, we get the impulse response as

$$h(t) = \delta(t) + \left(\mathrm{e}^{-t} + \mathrm{e}^{-3t}\right)u(t).$$

The stability condition of an analog system can be analyzed by its impulse response $h(t)$ or its transfer function $H(s)$. A system is stable if

$$\lim_{t\to\infty} h(t) = 0. \tag{4.9}$$

This condition requires that all the poles must lie in the left half of the s-plane, that is, $\sigma < 0$. If $\lim_{t\to\infty} h(t) \to \infty$, the system is unstable. This is equivalent to the system having one or more poles located in the right half of the s-plane, or having multiple-order (repeated) pole(s) on the $\mathrm{j}\Omega$ axis. Therefore, we can evaluate the stability of an analog system from its impulse response $h(t)$, or, more efficiently, from the pole locations of transfer function $H(s)$.

4.1.2 Mapping Properties

The z-transform can be obtained from the Laplace transform by the change of variable

$$z = \mathrm{e}^{sT}. \tag{4.10}$$

This relationship represents the mapping of a region in the s-plane to the z-plane because both s and z are complex variables. Since $s = \sigma + \mathrm{j}\Omega$, we have

$$z = \mathrm{e}^{\sigma T}\,\mathrm{e}^{\mathrm{j}\Omega T} = |z|\mathrm{e}^{\mathrm{j}\omega}, \tag{4.11}$$

where the magnitude is

$$|z| = \mathrm{e}^{\sigma T} \tag{4.12}$$

and the angle is

$$\omega = \Omega T. \tag{4.13}$$

When $\sigma = 0$ (the $\mathrm{j}\Omega$ axis on the s-plane), the amplitude given in (4.12) is $|z| = 1$ (the unit circle on the z-plane), and (4.11) is simplified to $z = \mathrm{e}^{\mathrm{j}\Omega T}$. It is apparent that the portion of the

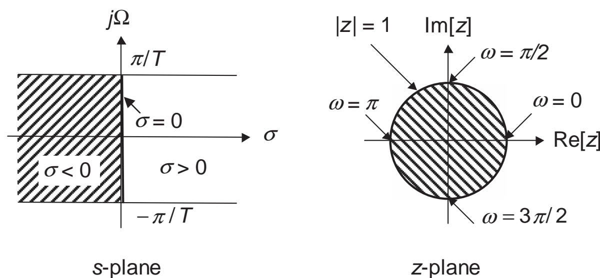


Figure 4.1 Mapping properties between the *s*-plane and the *z*-plane

jΩ axis between $\Omega = -\pi/T$ and $\Omega = \pi/T$ on the *s*-plane is mapped onto the unit circle on the *z*-plane from $-\pi$ to π as illustrated in Figure 4.1. As Ω increases from π/T to $3\pi/T$, this results in another counterclockwise encirclement of the unit circle. Thus, as Ω varies from 0 to ∞, there are infinite numbers of encirclements of the unit circle on the *z*-plane in the counterclockwise direction. Similarly, there are infinite numbers of encirclements of the unit circle in the clockwise direction as Ω varies from 0 to $-\infty$.

From (4.12), $|z| < 1$ when $\sigma < 0$. Thus, each strip of width $2\pi/T$ in the left half of the *s*-plane is mapped inside the unit circle. This mapping occurs in the form of concentric circles in the *z*-plane as σ varies from 0 to $-\infty$. Equation (4.12) also implies that $|z| > 1$ if $\sigma > 0$. Thus, each strip of width $2\pi/T$ in the right half of the *s*-plane is mapped outside the unit circle. This mapping also occurs in concentric circles in the *z*-plane as σ varies from 0 to ∞.

In conclusion, the mapping from the *s*-plane to the *z*-plane is not a one-to-one function since infinite points on the *s*-plane are mapped to a single point on the *z*-plane. This issue will be discussed later in Section 4.2 when we design a digital IIR filter $H(z)$ by transforming from an analog filter $H(s)$.

4.1.3 Characteristics of Analog Filters

The ideal lowpass filter prototype is obtained by finding a polynomial approximation to the squared magnitude $|H(\Omega)|^2$, and then converting this polynomial into a rational function. The approximations of the ideal prototype will be discussed briefly, based on Butterworth filters, Chebyshev type I and type II filters, elliptic filters, and Bessel filters.

The Butterworth lowpass filter is an all-pole approximation to the ideal filter, which is characterized by the squared magnitude response

$$|H(\Omega)|^2 = \frac{1}{1 + \left(\Omega/\Omega_p\right)^{2L}}, \tag{4.14}$$

where L is the order of the filter, which determines how closely the Butterworth filter approximates the ideal filter. Equation (4.14) shows that $|H(0)| = 1$ and $|H(\Omega_p)| = 1/\sqrt{2}$ (or $20\log_{10}|H(\Omega_p)| = -3$ dB) for all values of L. Thus, Ω_p is called the 3 dB cutoff frequency. The magnitude response of a typical Butterworth lowpass filter decreases monotonically in both the passband and stopband as illustrated in Figure 4.2. The Butterworth filter has a flat magnitude response over the passband and stopband, hence is often referred to as the

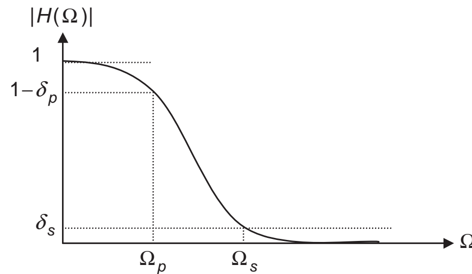


Figure 4.2 Magnitude response of Butterworth lowpass filter

"maximally flat" filter. This flat passband is achieved at the expense of slow roll-off in the transition band from Ω_p to Ω_s.

Although the Butterworth filter is easy to design, the rate of its magnitude response decreases is slow in the frequency range $\Omega \geq \Omega_p$ for small L. We can improve the speed of roll-off by increasing the filter order L. Therefore, for achieving a desired transition band, the Butterworth filter order is often higher than the order of other types of filters.

The Chebyshev filter has a steeper roll-off near the cutoff frequency than the Butterworth filter by allowing a certain number of ripples in either the passband or stopband. There are two types of Chebyshev filters. Type I Chebyshev filters are all-pole filters that exhibit equiripple behavior in the passband and monotonic characteristic in the stopband, see the top plot of Figure 4.3. Type II Chebyshev filters contain both poles and zeros, and exhibit monotonic behavior in the passband and equiripple behavior in the stopband, as shown in the bottom plot of Figure 4.3. In general, a Chebyshev filter can meet the given specifications with fewer poles than the corresponding Butterworth filter and can improve the roll-off rate; however, it has a poorer phase response.

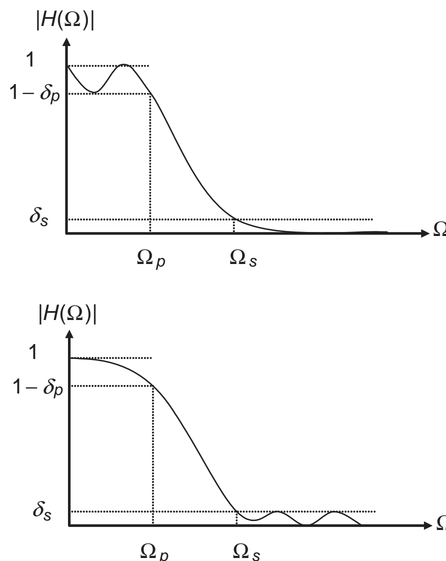


Figure 4.3 Magnitude responses of Chebyshev type I (top) and type II lowpass filters

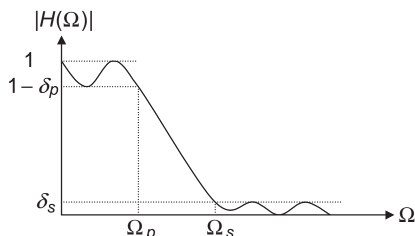


Figure 4.4 Magnitude response of elliptic lowpass filter

The sharpest transition from passband to stopband for the given δ_p, δ_s, and L can be achieved using the elliptic filter design. As shown in Figure 4.4, elliptic filters exhibit equiripple behavior in both the passband and stopband. However, the phase response of an elliptic filter is extremely nonlinear in the passband, especially near the cutoff frequency. Therefore, elliptic filters are only used for applications in which the phase is not an important design parameter.

In summary, the Butterworth filter has monotonic magnitude response at both the passband and stopband with slow roll-off. By allowing ripples in the passband for the type I and in the stopband for the type II, a Chebyshev filter can achieve sharper roll-off with the same number of poles. The elliptic filter has an even sharper roll-off than a Chebyshev filter of the same order, but it comes with passband and stopband ripples. The design of these filters strives to achieve the desired magnitude response with trade-offs in phase response and roll-off rate. In addition, the Bessel filter is an all-pole filter that approximate linear phase in the sense of maximally flat group delay in the passband. However, the steepness of roll-off must be sacrificed in the transition band.

4.1.4 Frequency Transforms

We have discussed the design of prototype lowpass filters with cutoff frequency Ω_p. Although the same process can be applied to design highpass, bandpass, and bandstop filters, it is easier to obtain these filters by applying a frequency transformation method to the lowpass filters. In addition, most classical filter design techniques generate lowpass filters only.

A highpass filter $H_\mathrm{hp}(s)$ can be obtained from the lowpass filter $H(s)$ by

$$H_\mathrm{hp}(s) = H(s)|_{s=1/s} = H\left(\frac{1}{s}\right). \tag{4.15}$$

For example, we have the Butterworth lowpass filter $H(s) = 1/(s+1)$ for $L = 1$. From (4.15), we obtain the highpass filter as

$$H_\mathrm{hp}(s) = \left.\frac{1}{s+1}\right|_{s=1/s} = \frac{s}{s+1}. \tag{4.16}$$

This example shows that the highpass filter $H_\mathrm{hp}(s)$ has an identical pole to the lowpass prototype, but with an additional zero at the origin.

A bandpass filter can be obtained from the lowpass prototype by replacing s with $\left(s^2 + \Omega_m^2\right)/\mathrm{BW}$. That is,

$$H_{\mathrm{bp}}(s) = H(s)\big|_{s=(s^2+\Omega_m^2)/\mathrm{BW}}, \tag{4.17}$$

where Ω_m is the center frequency of the bandpass filter defined as

$$\Omega_m = \sqrt{\Omega_a \Omega_b}, \tag{4.18}$$

where Ω_a and Ω_b are the lower and upper cutoff frequencies, respectively. The filter bandwidth BW is defined as

$$\mathrm{BW} = \Omega_b - \Omega_a. \tag{4.19}$$

For example, based on a Butterworth lowpass filter with $L = 1$, we have

$$H_{\mathrm{bp}}(s) = \left.\frac{1}{s+1}\right|_{s=(s^2+\Omega_m^2)/\mathrm{BW}} = \frac{\mathrm{BW}s}{s^2 + \mathrm{BW}s + \Omega_m^2}. \tag{4.20}$$

Thus, we can obtain a bandpass filter of order $2L$ from a lowpass filter of order L.

Finally, a bandstop filter can be obtained from the corresponding highpass filter using the following transformation:

$$H_{\mathrm{bs}}(s) = H_{\mathrm{hp}}(s)\big|_{s=(s^2+\Omega_m^2)/\mathrm{BW}s}. \tag{4.21}$$

4.2 Design of IIR Filters

As defined in (2.38), the transfer function of the digital IIR filter is

$$H(z) = \frac{\sum_{l=0}^{L-1} b_l z^{-l}}{1 + \sum_{m=1}^{M} a_m z^{-m}}. \tag{4.22}$$

This IIR filter can be realized by the following difference equation:

$$y(n) = \sum_{l=0}^{L-1} b_l x(n-l) - \sum_{m=1}^{M} a_m y(n-m). \tag{4.23}$$

The digital filter design problem is to determine the coefficients b_l and a_m so that $H(z)$ satisfies the given specifications.

The purpose of designing a digital IIR filter is to determine the transfer function $H(z)$ which approximates the prototype analog filter $H(s)$. Two methods can be used to map the analog

filter to the equivalent digital filter: the impulse-invariant method and the bilinear transform. The impulse-invariant method preserves the impulse response of the original analog filter by sampling its impulse response, but has inherent aliasing problems as discussed in Chapter 2. The bilinear transform can preserve the magnitude response characteristics of the analog filters, and thus is better for designing frequency-selective IIR filters. Detailed descriptions of using the bilinear transform method for digital IIR filter design are presented in the following sections.

4.2.1 Bilinear Transform

The procedure for designing digital filters using a bilinear transform is illustrated in Figure 4.5. This method first maps the digital filter specifications to the equivalent analog filter specifications. Then, the analog filter is designed according to the analog filter specifications. Finally, the desired digital filter is obtained by mapping the analog filter using the bilinear transform.

The bilinear transform is defined as

$$s = \frac{2}{T}\left(\frac{z-1}{z+1}\right) = \frac{2}{T}\left(\frac{1-z^{-1}}{1+z^{-1}}\right). \tag{4.24}$$

Because both the numerator and denominator are linear functions of z, (4.24) is known as the bilinear transform. Substituting $s = \mathrm{j}\Omega$ and $z = \mathrm{e}^{\mathrm{j}\omega}$ into (4.24), we obtain

$$\mathrm{j}\Omega = \frac{2}{T}\left(\frac{\mathrm{e}^{\mathrm{j}\omega}-1}{\mathrm{e}^{\mathrm{j}\omega}+1}\right). \tag{4.25}$$

The corresponding mapping of frequencies is obtained as

$$\Omega = \frac{2}{T}\tan\left(\frac{\omega}{2}\right), \tag{4.26}$$

or equivalently,

$$\omega = 2\tan^{-1}\left(\frac{\Omega T}{2}\right). \tag{4.27}$$

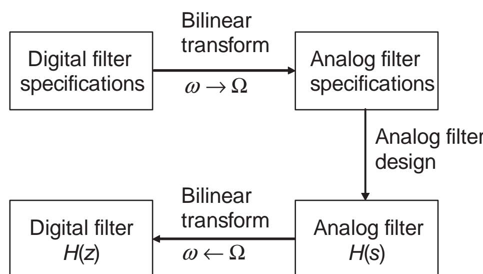


Figure 4.5 Digital IIR filter design using the bilinear transform

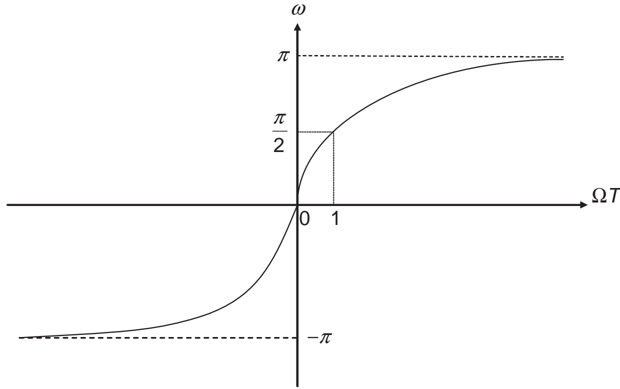


Figure 4.6 Frequency warping of bilinear transform defined by (4.27)

These equations show that the entire jΩ axis is compressed into the interval $[-\pi/T, \pi/T]$ for ω in a one-to-one manner. The portion of $0 \rightarrow \infty$ in the s-plane is mapped onto the $0 \rightarrow \pi$ portion of the unit circle, while the $0 \rightarrow -\infty$ portion in the s-plane is mapped onto the $0 \rightarrow -\pi$ portion of the unit circle. Each point in the s-plane is uniquely mapped onto the z-plane.

The relationship between the frequency variables Ω and ω is illustrated in Figure 4.6. The bilinear transform provides a one-to-one mapping of the points along the jΩ axis onto the unit circle, or onto the Nyquist band $|\omega| \leq \pi$. However, the mapping is highly nonlinear. The point $\Omega = 0$ is mapped to $\omega = 0$ (or $z = 1$), and the point $\Omega = \infty$ is mapped to $\omega = \pi$ (or $z = -1$). The entire band of $\Omega T \geq 1$ is compressed into the region $\pi/2 \leq \omega \leq \pi$. This frequency compression effect is known as frequency warping, and must be taken into consideration for digital filter design using the bilinear transform. The effective solution is to pre-warp the critical frequencies according to (4.26).

4.2.2 Filter Design Using the Bilinear Transform

The bilinear transform of an analog filter $H(s)$ can be completed by replacing s with z using (4.24). The filter specifications are given in terms of the critical frequencies of the digital filter. For example, the critical frequency ω for a lowpass filter is the bandwidth of the filter. The three steps for IIR filter design using the bilinear transform are summarized as follows:

1. Pre-warp the critical frequency ω_c of the digital filter using (4.26) to obtain the corresponding analog filter's frequency Ω_c.
2. Scale the analog filter $H(s)$ with Ω_c to obtain the scaled transfer function as follows:

$$\hat{H}(s) = H(s)|_{s=s/\Omega_c} = H\left(\frac{s}{\Omega_c}\right). \tag{4.28}$$

3. Replace s using (4.24) to obtain desired digital filter $H(z)$. That is,

$$H(z) = \hat{H}(s)|_{s=2(z-1)/(z+1)T}. \tag{4.29}$$

Example 4.2

Use the simple analog lowpass filter $H(s) = 1/(s+1)$ and the bilinear transform method to design a digital lowpass filter with bandwidth 1000 Hz and sampling frequency 8000 Hz.

The critical frequency of the lowpass filter is the bandwidth, which can be computed as $\omega_c = 2\pi(1000/8000) = 0.25\pi$, and the sampling period is $T = 1/8000$ seconds.

Step 1: Pre-warp the critical frequency from digital to analog as

$$\Omega_c = \frac{2}{T}\tan\left(\frac{\omega_c}{2}\right) = \frac{2}{T}\tan(0.125\pi) = \frac{0.8284}{T}.$$

Step 2: Use frequency scaling to obtain

$$\hat{H}(s) = H(s)|_{s=s/(0.8284/T)} = \frac{0.8284}{sT + 0.8284}.$$

Step 3: Use the bilinear transform defined in (4.29) to obtain the desired transfer function

$$H(z) = \hat{H}(s)|_{s=2(z-1)/(z+1)T} = 0.2929\frac{1+z^{-1}}{1-0.4142z^{-1}}.$$

MATLAB® provides the `impinvar` and `bilinear` functions [13,14] to support the impulse-invariant and bilinear transform methods, respectively. For example, we can perform the bilinear transform for the given numerator and denominator polynomials as follows:

```
[NUMd,DENd]=bilinear(NUM,DEN,Fs,Fp);
```

where `NUMd` and `DENd` are the numerator and denominator coefficient vectors, respectively, of the digital filter obtained from the bilinear transform. `NUM` and `DEN` are row vectors containing numerator and denominator coefficients in descending powers of s, respectively, `Fs` is the sampling frequency in Hz, and `Fp` is the pre-warped frequency.

Example 4.3

In order to design a digital IIR filter using the bilinear transform, we first design the analog prototype filter. The numerator and denominator polynomials of the prototype filter are then mapped to the polynomials of the digital filter using the bilinear transform. The following MATLAB® script (`example4_3.m`) designs a Butterworth lowpass filter:

```
Fs = 2000;                          % Sampling frequency
Wn = 300;                           % Edge frequency
n = 4;                              % Order of analog filter
[b, a] = butter(n, Wn, 's');        % Design analog filter
[bz, az] = bilinear(b, a, Fs, Wn);  % Determine digital filter
freqz(bz, az, 512, Fs);             % Display magnitude & phase
```

4.3 Realization of IIR Filters

An IIR filter can be realized in different forms or structures. In this section, we discuss direct-form I, direct-form II, cascade, and parallel IIR filter structures. These realizations are equivalent mathematically, but they may have different behaviors in practical implementations due to the finite-wordlength effects.

4.3.1 Direct Forms

The direct-form I realization is defined by the input/output equation given in (4.23). This filter has $(L+M)$ coefficients and needs $(L+M+1)$ memory locations to store $\{x(n-l), l=0, 1, \ldots, L-1\}$ and $\{y(n-m), m=0, 1, \ldots, M\}$. It also requires $(L+M)$ multiplications and $(L+M-1)$ additions. The detailed signal-flow diagram for the case of $L=M+1$ is illustrated in Figure 2.11.

Example 4.4

Consider the second-order IIR filter

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}. \tag{4.30}$$

The input/output equation for the direct-form I realization is

$$y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) - a_1 y(n-1) - a_2 y(n-2). \tag{4.31}$$

The signal-flow diagram is illustrated in Figure 4.7.

As shown in Figure 4.7, the IIR filter $H(z)$ can be interpreted as a cascade of two transfer functions $H_1(z)$ and $H_2(z)$. That is,

$$H(z) = H_1(z)H_2(z), \tag{4.32}$$

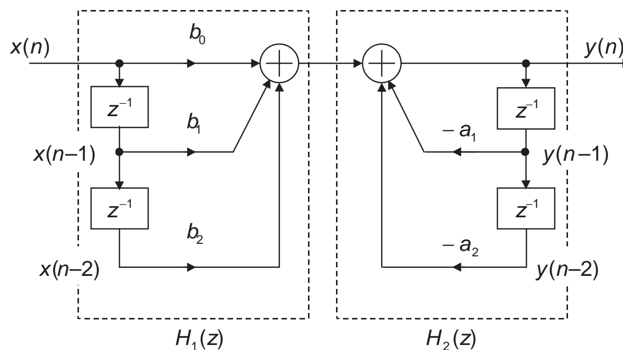


Figure 4.7 Direct-form I realization of second-order IIR filter

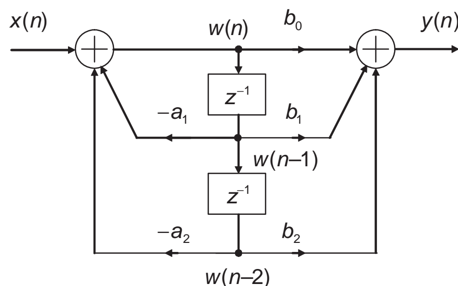


Figure 4.8 Direct-form II realization of second-order IIR filter

where $H_1(z) = b_0 + b_1z^{-1} + b_2z^{-2}$ and $H_2(z) = 1/(1 + a_1z^{-1} + a_2z^{-2})$. Since multiplication is commutative, (4.32) can be rewritten as $H(z) = H_2(z)H_1(z)$. Therefore, Figure 4.7 can be redrawn by exchanging the order of $H_1(z)$ and $H_2(z)$, and combining two signal buffers into one (since both buffers have the same signal samples) as illustrated in Figure 4.8. This more efficient realization of a second-order IIR filter is called the direct-form II (or biquad), which requires three memory locations as opposed to the six required by the direct-form I shown in Figure 4.7. Therefore, the direct-form II is also called the canonical form since it needs a minimum amount of memory.

As shown in Figure 4.8, the direct-form II second-order IIR filter can be implemented as

$$y(n) = b_0w(n) + b_1w(n-1) + b_2w(n-2), \tag{4.33}$$

where

$$w(n) = x(n) - a_1w(n-1) - a_2w(n-2). \tag{4.34}$$

This realization can be expanded as in Figure 4.9 to realize the IIR filter defined in (4.22) with $M = L - 1$ and using the direct-form II structure.

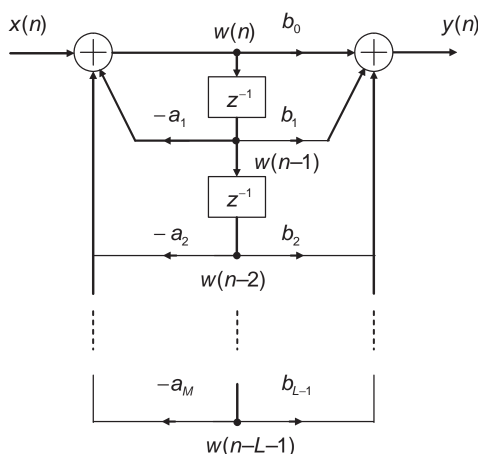


Figure 4.9 Direct-form II realization of general IIR filter, $L = M + 1$

4.3.2 Cascade Realizations

By factoring the numerator and denominator polynomials of the transfer function $H(z)$, an IIR filter can be realized as a cascade of multiple second-order IIR filters when the filter order M is an even number. Consider the transfer function $H(z)$ given in (4.22), which can be expressed as

$$H(z) = b_0 H_1(z) H_2(z) \ldots H_K(z) = b_0 \prod_{k=1}^{K} H_k(z), \tag{4.35}$$

where k is the section index, $K (= M/2)$ is the total number of sections, and $H_k(z)$ is the second-order filter expressed as

$$H_k(z) = \frac{(z - z_{1k})(z - z_{2k})}{(z - p_{1k})(z - p_{2k})} = \frac{1 + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 + a_{1k} z^{-1} + a_{2k} z^{-2}}. \tag{4.36}$$

If the filter order M is an odd number, one of the $H_k(z)$ is the first-order IIR filter expressed as

$$H_k(z) = \frac{z - z_{1k}}{z - p_{1k}} = \frac{1 + b_{1k} z^{-1}}{1 + a_{1k} z^{-1}}. \tag{4.37}$$

The realization of (4.35) in cascade structure is illustrated in Figure 4.10. Note that any pair of complex-conjugate roots must be grouped into the same section to guarantee that the coefficients of $H_k(z)$ are all real-valued numbers. Assuming that every $H_k(z)$ is the second-order IIR filter described by (4.36), the input/output equations describing the cascade realization are

$$w_k(n) = x_k(n) - a_{1k} w_k(n-1) - a_{2k} w_k(n-2), \tag{4.38}$$

$$y_k(n) = w_k(n) + b_{1k} w_k(n-1) + b_{2k} w_k(n-2), \tag{4.39}$$

$$x_{k+1}(n) = y_k(n), \tag{4.40}$$

for $k = 1, 2, \ldots, K$ where $x_1(n) = b_0 x(n)$ and $y(n) = y_K(n)$.

It is possible to have many different cascade realizations of the same transfer function $H(z)$ by different ordering and pairing. Ordering means the order of connecting $H_k(z)$, and pairing means the grouping of poles and zeros of $H(z)$ to form $H_k(z)$ as shown in (4.36). In theory, these different cascade realizations are identical; however, they will be different when the filter is implemented using finite-wordlength hardware. For example, each section will have a certain amount of roundoff noise, which is propagated to the next section. The total roundoff noise at the final output will depend on the particular pairing/ordering.

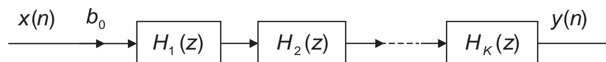


Figure 4.10 Cascade realization of digital filter

In the direct-form II realization shown in Figure 4.9, the variation of one parameter will affect all the poles of $H(z)$. In the cascade realization, the variation of one parameter in $H_k(z)$ will only affect the pole(s) in that section. Therefore, the cascade realization is preferred in practical implementations because it is less sensitive to parameter variation due to quantization effects.

Example 4.5

Consider the second-order IIR filter

$$H(z) = \frac{0.5(z^2 - 0.36)}{z^2 + 0.1z - 0.72}.$$

By factoring the numerator and denominator polynomials of $H(z)$, we obtain

$$H(z) = \frac{0.5(1 + 0.6z^{-1})(1 - 0.6z^{-1})}{(1 + 0.9z^{-1})(1 - 0.8z^{-1})}.$$

By different pairings of poles and zeros, there are four possible realizations of $H(z)$ in terms of using first-order sections. For example, we may choose

$$H_1(z) = \frac{1 + 0.6z^{-1}}{1 + 0.9z^{-1}} \quad \text{and} \quad H_2(z) = \frac{1 - 0.6z^{-1}}{1 - 0.8z^{-1}}.$$

The IIR filter can be realized by the cascade structure expressed as

$$H(z) = 0.5H_1(z)H_2(z) = 0.5H_2(z)H_1(z).$$

4.3.3 Parallel Realizations

The expression of $H(z)$ using a partial-fraction expansion leads to another canonical structure called the parallel realization, expressed as

$$H(z) = c_0 + H_1(z) + H_2(z) + \cdots + H_K(z), \tag{4.41}$$

where c_0 is a constant, and $H_k(z)$ is the second-order IIR filter expressed as

$$H_k(z) = \frac{b_{0k} + b_{1k}z^{-1}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}, \tag{4.42}$$

or one section is the first-order filter expressed as

$$H_k(z) = \frac{b_{0k}}{1 + a_{1k}z^{-1}} \tag{4.43}$$

if the filter order M is an odd number. The realization of (4.41) in a parallel structure is illustrated in Figure 4.11.

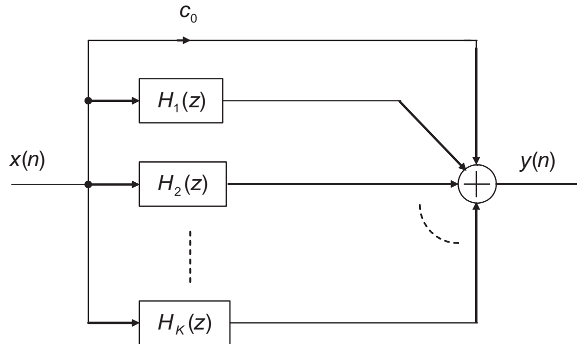


Figure 4.11 Parallel realization of digital IIR filter

Example 4.6

Consider the transfer function $H(z)$ given in Example 4.5, which can be expressed as

$$H'(z) = \frac{H(z)}{z} = \frac{0.5(1+0.6z^{-1})(1-0.6z^{-1})}{z(1+0.9z^{-1})(1-0.8z^{-1})} = \frac{A}{z} + \frac{B}{z+0.9} + \frac{C}{z-0.8},$$

where

$$\begin{aligned} A &= zH'(z)|_{z=0} = 0.25 \\ B &= (z+0.9)H'(z)|_{z=-0.9} = 0.147 \\ C &= (z-0.8)H'(z)|_{z=0.8} = 0.103. \end{aligned}$$

Therefore, we obtain

$$H(z) = 0.25 + \frac{0.147}{1+0.9z^{-1}} + \frac{0.103}{1-0.8z^{-1}}.$$

4.3.4 Realization of IIR Filters Using MATLAB®

Realization of IIR filters using the cascade structure involves the factorization of polynomials. This can be done using the MATLAB® function `roots`. For example, the statement

```
r = roots(b);
```

returns the roots of the numerator vector `b` in the output vector `r`. Similarly, we can obtain the roots of the denominator vector `a`. The coefficients of each section can be determined by pole–zero pairings.

The MATLAB® function `tf2zp` calculates the zeros, poles, and gain of a given transfer function. For example, the statement

```
[z, p, c] = tf2zp(b, a);
```

will return the zero locations in z, the pole locations in p, and the gain in c. Similarly, the function

```
[b, a] = zp2tf(z,p,c);
```

forms the rational transfer function $H(z)$ given a set of zeros in vector z, a set of poles in vector p, and a gain in scalar c.

Example 4.7

The zeros, poles, and gain of the system defined in Example 4.5 can be obtained using the MATLAB® script (example4_7.m) as follows:

```
b = [0.5, 0, -0.18];   % Numerator
a = [1, 0.1, -0.72];   % Denominator
[z, p, c] = tf2zp(b,a) % Display zeros, poles, and gain
```

Running the program, we obtain z = 0.6, −0.6, p = −0.9, 0.8, and c = 0.5. These results verify the derivation obtained in Example 4.5.

MATLAB® also provides the useful function zp2sos to convert a zero–pole–gain representation to the equivalent representation of second-order IIR sections [15]. The function

```
[sos, G] = zp2sos(z, p, c);
```

finds the overall gain G and the matrix sos containing the coefficients of every second-order IIR section determined from its zero–pole–gain form. The matrix sos is the $6 \times K$ matrix expressed as follows:

$$\text{sos} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0K} & b_{1K} & b_{2K} & 1 & a_{1K} & a_{2K} \end{bmatrix}, \tag{4.44}$$

where each row k ($k = 1, 2, \ldots, K$) represents the kth second-order IIR filter $H_k(z)$, which contains the numerator and denominator coefficients, b_{ik}, $i = 0, 1, 2$, and a_{jk}, $j = 1, 2$. The overall transfer function is expressed as

$$H(z) = G\prod_{k=1}^{K} H_k(z) = G\prod_{k=1}^{K} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}, \tag{4.45}$$

where G is a scalar which accounts for the overall gain of the system. Similarly, the MATLAB® function `[sos, G] = tf2sos(b, a)` computes the matrix sos and the gain G from the given numerator and denominator vectors b and a, respectively.

The parallel realizations discussed in Section 4.3.3 are supported by the MATLAB® function `residuez`. This function converts the transfer function expressed in (4.22) to the partial-fraction expansion (or residue) form expressed in (4.41). The function

```
[r, p, c] = residuez(b, a);
```

returns the vectors where `r` contains the residues, `p` contains the pole locations, and `c` contains the direct terms.

4.4 Design of IIR Filters Using MATLAB®

This section introduces MATLAB® functions that can be used to design IIR filters, realize and analyze the designed filters, and quantize filter coefficients for fixed-point implementations [15,16].

4.4.1 Filter Design Using MATLAB®

MATLAB® provides several functions for designing the Butterworth, Chebyshev type I, Chebyshev type II, elliptic, and Bessel IIR filters in four different types: lowpass, highpass, bandpass, and bandstop. In general, the IIR filter design requires two steps. First, compute the minimum filter order `N` and the frequency-scaling factor `Wn` from the given specifications. Second, calculate the filter coefficients using these two parameters. In the first step of IIR filter design, one of the following MATLAB® functions is used to estimate the filter order:

```
[N, Wn] = buttord(Wp, Ws, Rp, Rs);  % Butterworth filter
[N, Wn] = cheb1ord(Wp, Ws, Rp, Rs); % Chebyshev type I filter
[N, Wn] = cheb2ord(Wp, Ws, Rp, Rs); % Chebyshev type II filter
[N, Wn] = ellipord(Wp, Ws, Rp, Rs); % Elliptic filter
```

The parameters `Wp` and `Ws` are the normalized passband and stopband edge frequencies, respectively. The range of `Wp` and `Ws` is from 0 to 1, where 1 corresponds to $\omega = \pi$ or $f = f_s/2$. The parameters `Rp` and `Rs` are the passband ripple and the minimum stopband attenuation specified in dB, respectively. These four functions return the order `N` and the frequency-scaling factor `Wn`, which are needed for the second step of IIR filter design.

In the second step, the IIR filter design process computes the filter coefficients using `N` and `Wn` obtained from the first step. MATLAB® provides the following IIR filter design functions to calculate the filter coefficients:

```
[b, a] = butter(N, Wn);
[b, a] = cheby1(N, Rp, Wn);
[b, a] = cheby2(N, Rs, Wn);
[b, a] = ellip(N, Rp, Rs, Wn);
```

These functions return the filter coefficients in row vectors `b` (contains numerator coefficients, b_l) and `a` (contains denominator coefficients, a_m). We can use `butter(N,Wn,'high')` to design a highpass filter. If `Wn` is a two-element vector such as `Wn = [W1 W2]`, the function

butter will return a bandpass filter of order 2N with the passband in between W1 and W2, and butter(N,Wn,'stop') designs a bandstop filter.

Example 4.8

Design a lowpass Butterworth filter with less than 1.0 dB of ripple from 0 to 800 Hz, and at least 20 dB of stopband attenuation from 1600 Hz to the Nyquist frequency 4000 Hz.

The MATLAB® script (example4_8.m) for designing the filter is listed as follows:

```
Wp = 800/4000;     % Passband frequency
Ws= 1600/4000;     % Stopband frequency
Rp = 1.0;          % Passband ripple
Rs = 20.0;         % Stopband attenuation

[N, Wn] = buttord(Wp, Ws, Rp, Rs);      % First step
[b, a] = butter(N, Wn);                 % Second step
freqz(b, a, 512, 8000);                 % Display frequency response
```

Instead of using the MATLAB® function freqz to display magnitude and phase responses, we can use the flexible Filter Visualization Tool (FVTool) to analyze digital filters. The command

```
fvtool(b,a)
```

launches the GUI FVTool and computes the magnitude response of the filter defined by the numerator and denominator coefficients given in the vectors b and a, respectively.

Example 4.9

Design a bandpass filter with the passband from 100 to 200 Hz and the sampling rate is 1 kHz. The passband ripple is less than 3 dB and the stopband edges are 50 and 250 Hz with at least 30 dB of stopband attenuation.

The MATLAB® script (example4_9.m) for designing and evaluating this filter is listed as follows:

```
Wp = [100 200]/500;                   % Passband frequencies
Ws = [50 250]/500;                    % Stopband frequencies
Rp = 3;                               % Passband ripple
Rs = 30;                              % Stopband attenuation
[N, Wn] = buttord(Wp, Ws, Rp, Rs);    % Estimate filter order
[b, a] = butter(N, Wn);               % Design Butterworth filter
fvtool(b, a);                         % Analyze designed IIR filter
```

From the **Analysis** menu in the FVTool window, we select the **Magnitude and Phase Responses**. The magnitude and phase responses of the designed bandpass filter are shown in Figure 4.12.

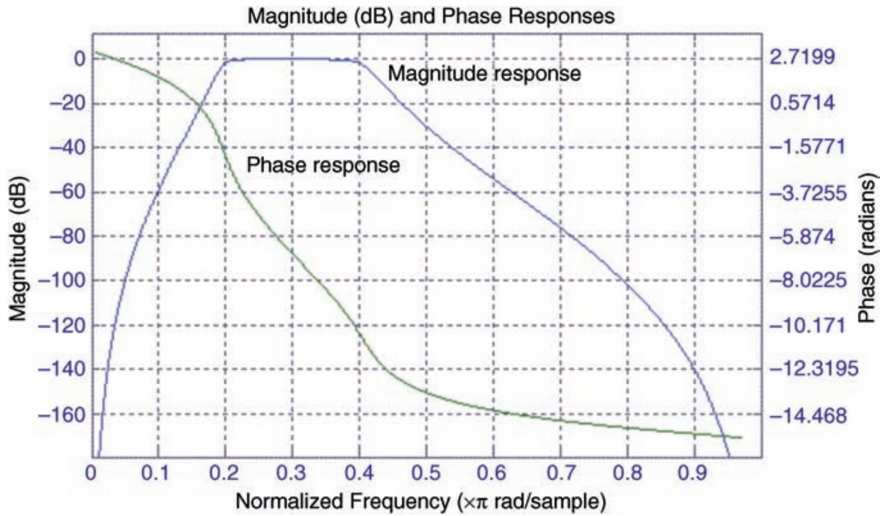


Figure 4.12 Magnitude and phase responses of bandpass filter

4.4.2 Frequency Transforms Using MATLAB®

The *Signal Processing Toolbox* provides the functions `lp2hp`, `lp2bp`, and `lp2bs` to convert the prototype lowpass filters to highpass, bandpass, and bandstop filters, respectively. For example, the command

```
[numt,dent] = lp2hp(num,den,wo);
```

transforms the prototype lowpass filter to the highpass filter with cutoff frequency `wo`.

4.4.3 Filter Design and Realization Using the FDATool

In this section, we use the FDATool for designing, realizing, and quantizing IIR filters. To design an IIR filter, select the radio button next to **IIR** in the **Design Method** region on the GUI. There are seven options (from the pull-down menu) for **Lowpass** types, and several different filter design methods for different response types.

Example 4.10

Design a lowpass IIR filter with the following specifications: sampling frequency $f_s = 8\,\text{kHz}$, passband cutoff frequency $\omega_p = 2\,\text{kHz}$, stopband cutoff frequency $\omega_s = 2.5\,\text{kHz}$, passband ripple $A_p = 1\,\text{dB}$, and stopband attenuation $A_s = 60\,\text{dB}$.

We can design an elliptic filter by clicking on the radio button next to **IIR** in the **Design Method** region and selecting **Elliptic** from the pull-down menu. We then enter parameters in the **Frequency Specifications** and **Magnitude Specifications** regions as shown in Figure 4.13. After pressing the **Design Filter** button to compute the filter coefficients, the **Filter Specifications** region changes to the **Magnitude Response (dB)** as shown in Figure 4.13.

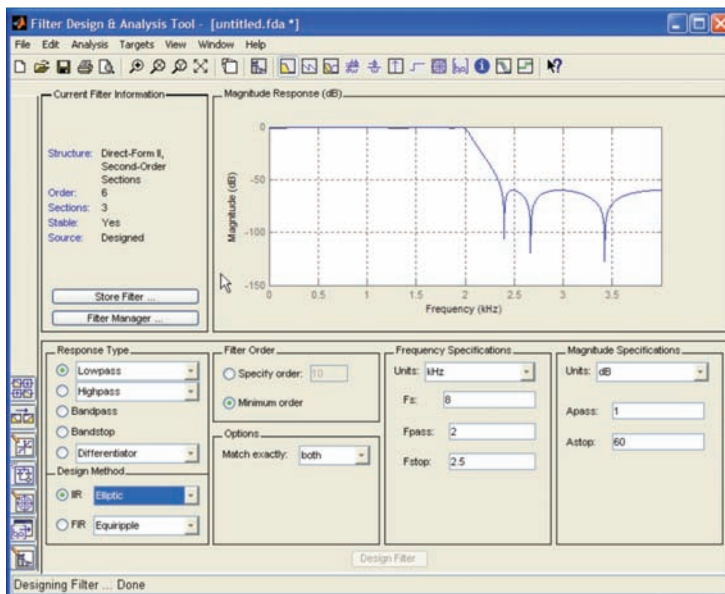
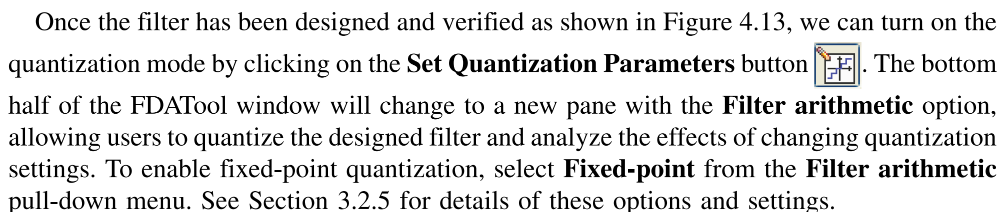


Figure 4.13 Design of an elliptic IIR lowpass filter

We can specify the desired filter order by clicking on the radio button **Specify order** and entering the filter order in the text box, or choose the default **Minimum order**. The order of the designed filter is 6, which is stated in the **Current Filter Information** region (top left) as shown in Figure 4.13. By default, the designed IIR filter is realized by cascading second-order IIR sections using the direct-form II biquads shown in Figure 4.8. We can change this default setting from **Edit** → **Convert Structure**, the dialog window displayed for selecting different structures. We can reorder and scale second-order sections by selecting **Edit** → **Reorder and Scale Second-order Sections**.

Once the filter has been designed and verified as shown in Figure 4.13, we can turn on the quantization mode by clicking on the **Set Quantization Parameters** button . The bottom half of the FDATool window will change to a new pane with the **Filter arithmetic** option, allowing users to quantize the designed filter and analyze the effects of changing quantization settings. To enable fixed-point quantization, select **Fixed-point** from the **Filter arithmetic** pull-down menu. See Section 3.2.5 for details of these options and settings.

Example 4.11

Design a quantized bandpass IIR filter for a 16-bit fixed-point digital signal processor with the following specifications: sampling frequency = 8000 Hz, lower stopband cutoff frequency $F_{stop1} = 1200$ Hz, lower passband cutoff frequency $F_{pass1} = 1400$ Hz, upper passband cutoff frequency $F_{pass2} = 1600$ Hz, upper stopband cutoff frequency $F_{stop2} = 1800$ Hz, passband ripple = 1 dB, and stopband (both lower and upper) attenuation = 60 dB.

Start the FDATool and enter the required parameters in the **Frequency Specifications** and **Magnitude Specifications** regions, select the elliptic IIR filter type, and click on **Design Filter**. The order of the designed filter is 16 with eight second-order sections. Click

```
Quantized SOS matrix:
1  -1.442626953125     1  1  -0.80755615234375  0.94775390625
1   0.2017822265625    1  1  -0.685546875       0.9462890625
1  -1.11370849609375   1  1  -0.8975830078125   0.98077392578125
1  -0.35791015625      1  1  -0.6136474609375   0.97955322265625
Quantized Scale Factors:
0.09100341796875
0.09100341796875
0.3792724609375
0.3792724609375

Reference SOS matrix:
1  -1.4426457356639371   1  1  -0.80758037773723901  0.9477601749661968
1   0.20175861869884926  1  1  -0.68556453894167657  0.94631236163284504
1  -1.1137377328480738   1  1  -0.8975583707515371   0.98080193156894147
1  -0.35788604023451132  1  1  -0.61365277529418827  0.97953808588048008
Reference Scale Factors:
0.090994140334780968
0.090994140334780968
0.37924885680628034
0.37924885680628034
```

Figure 4.14 Quantized and reference filter coefficients

on the **Set Quantization Parameters** button, and select the **Fixed-point** option from the pull-down menu of **Filter arithmetic** and use the default settings. After designing and quantizing the filter, select the **Magnitude Response Estimate** option on the **Analysis** menu to estimate the frequency response of the quantized filter. The magnitude response of the quantized filter is displayed in the analysis area. We observe that the quantized filter has a satisfactory magnitude response, primarily because the FDATool implements the IIR filter using a cascade of second-order sections, which is more resistant to coefficient quantization errors.

We also select **Filter Coefficients** from the **Analysis** menu, as display them in Figure 4.14. It shows both the quantized coefficients (top) and the original coefficients (bottom) using the double-precision floating-point format.

We can save the designed filter coefficients in a C header file by selecting **Generate C header** from the **Targets** menu. The **Generate C Header** dialog box appears. For an IIR filter, variable names in the C header file are numerator (`NUM`), numerator length (`NL`), denominator (`DEN`), denominator length (`DL`), and number of sections (`NS`). We can use these default variable names, or change them to match the names used in the C programs that will include this header file. Click on **Generate** and the **Generate C Header** dialog box will appear. Enter the filename and click on **Save** to save the file. The IIR filter coefficients used by the experiments presented in Section 4.7 are generated by the FDATool.

4.5 Implementation Considerations

This section discusses important considerations for implementing IIR filters, including stability and finite-wordlength effects.

4.5.1 Stability

The IIR filter is stable if all the poles lie inside the unit circle. That is,

$$|p_m| < 1, \quad m = 1, 2, \ldots, M. \tag{4.46}$$

In this case, $\lim_{n\to\infty} h(n) = 0$. If there is any pole located outside the unit circle, that is, $|p_m| > 1$ for any m, the IIR filter is unstable since $\lim_{n\to\infty} h(n) \to \infty$. In addition, the IIR filter is unstable if $H(z)$ has multiple-order (repeated) pole(s) on the unit circle. However, the IIR filter is marginally stable (or oscillatory bounded) if the poles on the unit circle are first-order (non-repeated) poles such that the impulse response

$$\lim_{n\to\infty} h(n) = c, \tag{4.47}$$

where c is a non-zero constant. For example, if $H(z) = 1/1 + z^{-1}$, there is a first-order pole on the unit circle at $z = -1$, and the impulse response oscillates between ± 1 since $h(n) = (-1)^n, n \geq 0$.

Example 4.12

Consider an IIR filter with the transfer function

$$H(z) = \frac{1}{1 - az^{-1}}.$$

The impulse response of the system is $h(n) = a^n, n \geq 0$. If $|a| < 1$, the pole is located inside the unit circle and $\lim_{n\to\infty} h(n) = \lim_{n\to\infty} a^n \to 0$, so this IIR filter is stable. If $|a| > 1$, the pole is outside the unit circle and $\lim_{n\to\infty} a^n \to \infty$, so this IIR filter is unstable. However, if $a = 1$, the pole is on the unit circle and $\lim_{n\to\infty} a^n \to 1$, so this IIR filter is marginally stable.

In addition, consider a system with the transfer function

$$H(z) = \frac{z}{(z-1)^2},$$

where there is a second-order pole at $z = 1$. The impulse response of the system is $h(n) = n$, which is an unstable system since $\lim_{n\to\infty} h(n) \to \infty$.

Consider the second-order IIR filter defined by (4.30). The denominator can be factored as

$$1 + a_1 z^{-1} + a_2 z^{-2} = \left(1 - p_1 z^{-1}\right)\left(1 - p_2 z^{-1}\right) = 1 - (p_1 + p_2)z^{-1} + p_1 p_2 z^{-2},$$

thus

$$a_1 = -(p_1 + p_2) \quad \text{and} \quad a_2 = p_1 p_2. \tag{4.48}$$

The poles must lie inside the unit circle for stability, that is, $|p_1| < 1$ and $|p_2| < 1$. Therefore, we must have

$$|a_2| = |p_1 p_2| < 1 \tag{4.49}$$

for a stable system. The corresponding condition on a_1 can be derived from the Schur–Cohn stability test as

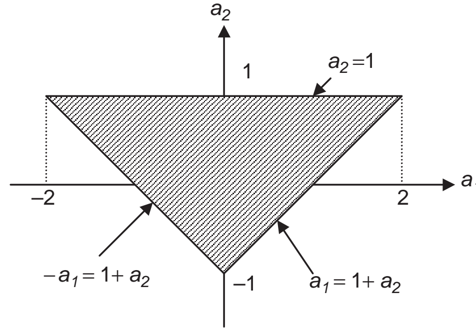


Figure 4.15 Region of coefficient values for the stable second-order IIR filters

$$|a_1| < 1 + a_2. \tag{4.50}$$

Stability conditions (4.49) and (4.50) for the second-order IIR filters are illustrated in Figure 4.15, which shows the resulting stability triangle in the a_1–a_2-plane. That is, the second-order IIR filter is stable if and only if the point (a_1, a_2) defined by the coefficients lies inside the stability triangle.

4.5.2 Finite-Precision Effects and Solutions

In practical applications, the coefficients obtained from filter design are quantized to a finite number of bits for implementation. The filter coefficients, b_l and a_m, obtained by MATLAB® are represented using the double-precision floating-point format. Let b'_l and a'_m denote the quantized values corresponding to b_l and a_m, respectively. The transfer function of the quantized IIR filter is expressed as

$$H'(z) = \frac{\sum_{l=0}^{L-1} b'_l z^{-l}}{1 + \sum_{m=1}^{M} a'_m z^{-m}}. \tag{4.51}$$

If the wordlength is not sufficiently large, some undesirable effects may occur. For example, the magnitude and phase responses of $H'(z)$ may be different from those of the desired $H(z)$. If the poles of $H(z)$ are close to the unit circle, some poles of $H'(z)$ may move outside the unit circle after coefficient quantization, resulting in an unstable filter. These undesired effects are more serious when higher order IIR filters are implemented using direct-form realizations. The cascade (or parallel) realization is recommended for implementation of high-order narrowband IIR filters that have closely clustered poles. Note that the cascade realization is used by MATLAB® to realize any IIR filter of order higher than 2.

Example 4.13

Consider an IIR filter with the transfer function

$$H(z) = \frac{1}{1 - 0.85z^{-1} + 0.18z^{-2}},$$

where the poles are located at $z = 0.4$ and $z = 0.45$. This filter can be realized in the cascade structure as $H(z) = H_1(z)H_2(z)$, where

$$H_1(z) = \frac{1}{1 - 0.4z^{-1}} \quad \text{and} \quad H_2(z) = \frac{1}{1 - 0.45z^{-1}}.$$

If this IIR filter is implemented using 4-bit (a sign bit plus three data bits, see Table 2.2) fixed-point hardware, 0.85 and 0.18 are quantized to 0.875 and 0.125, respectively. Therefore, the direct-form realization is described as

$$H'(z) = \frac{1}{1 - 0.875z^{-1} + 0.125z^{-2}}.$$

The poles of the direct-form $H'(z)$ become $z = 0.1798$ and 0.6952, which are significantly different from the original 0.4 and 0.45.

For the cascade realization, the coefficients 0.4 and 0.45 are quantized to 0.375 and 0.5, respectively. Thus, the quantized cascade filter is expressed as

$$H''(z) = \left(\frac{1}{1 - 0.375z^{-1}}\right)\left(\frac{1}{1 - 0.5z^{-1}}\right).$$

The poles of $H''(z)$ are $z = 0.375$ and 0.5. Therefore, the poles of cascade realization are closer to the desired $H(z)$ at $z = 0.4$ and 0.45.

Rounding of the $2B$-bit product to B bits introduces roundoff noise. The order of the cascade sections influences the power of roundoff noise at the filter output. In addition, when digital filters are implemented using fixed-point processors, we have to optimize the signal-to-quantization-noise ratio. This involves a trade-off with the probability of arithmetic overflow. The most effective technique in preventing overflow is to use scaling factors at various nodes within the cascaded filter sections. Optimal performance can be achieved by keeping the signal level as high as possible at each section without getting overflow.

Example 4.14

Consider the first-order IIR filter with the scaling factor α described by

$$H(z) = \frac{\alpha}{1 - az^{-1}},$$

where the stability condition requires that $|a| < 1$. The aim of including the scaling factor α is to ensure that the values of output $y(n)$ will not exceed one in magnitude. Suppose that

$x(n)$ is the sinusoidal signal of amplitude less than one at frequency ω_0; then the amplitude of the output is determined by the gain $|H(\omega_0)|$. The maximum gain of $H(z)$ is

$$\max_{\omega}|H(\omega)| = \frac{\alpha}{1-|a|}.$$

Thus, if the input signals are sinusoidal, a suitable scaling factor is $\alpha < 1-|a|$.

4.5.3 MATLAB® Implementations of IIR Filters

The MATLAB® function `filter` implements the IIR filter defined by (4.23). The basic forms of this function are

```
y = filter(b, a, x);
y = filter(b, a, x, zi);
```

The first element `a(1)` of the vector `a`, that is, the first coefficient a_0, must be one. The input vector is `x` and the filter output vector is `y`. At the beginning, the initial conditions (data in the signal buffers) are set to zero. However, they can be specified by the vector `zi` to reduce transient effects.

Example 4.15

A given sine wave (150 Hz) is corrupted by white noise with SNR of 0 dB, and the sampling rate is 1000 Hz. To enhance the sine wave, we need a bandpass filter with the passband centered at 150 Hz. Similar to Example 4.9, we design a bandpass filter with the following MATLAB® script:

```
Fs = 1000;                    % Sampling rate
Wp = [140 160]/(Fs/2);        % Passband edge frequencies
Ws = [130 170]/(Fs/2);        % Stopband edge frequencies
Rp = 3;                       % Passband ripple
Rs = 40;                      % Stopband ripple
[N, Wn] = buttord(Wp, Ws, Rp, Rs);  % Calculate filter order
[b, a] = butter(N, Wn);             % Design IIR filter
```

We can implement the designed filter to enhance the sinusoidal signal in the vector `xn` using the following function:

```
y = filter(b, a, xn); % IIR filtering
```

We plot the input and output signals in the `xn` and `y` vectors, which are displayed in Figure 4.16. The complete MATLAB® script for this example is given in `example4_15.m`.

MATLAB® also provides the second-order (biquad) IIR filtering function with the following syntax:

```
y = sosfilt(sos,x)
```

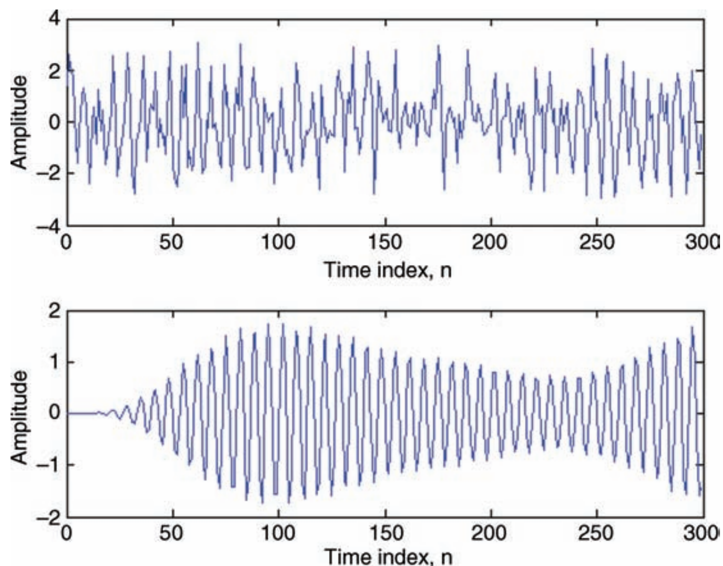


Figure 4.16 Input (top) and output (bottom) signals of the bandpass filter

This function applies the input signal in the vector `x` to the IIR filter $H(z)$ that is realized using the second-order sections in `sos` as defined in (4.44).

Example 4.16

In Example 4.15, we designed the bandpass filter and implemented the direct-form IIR filter using the function `filter`. In this example, we convert the direct-form filter to the cascade of second-order sections using the following function:

```
sos = tf2sos(b,a);
```

The obtained `sos` matrix is as follows:

```
sos =
    0.0000    0.0000   -0.0000    1.0000    -1.1077    0.8808
    1.0000    2.0084    1.0084    1.0000    -1.0702    0.8900
    1.0000    2.0021    1.0021    1.0000    -1.1569    0.8941
    1.0000    1.9942    0.9942    1.0000    -1.0519    0.9215
    1.0000   -2.0052    1.0053    1.0000    -1.2092    0.9268
    1.0000   -1.9925    0.9925    1.0000    -1.0581    0.9711
    1.0000   -1.9980    0.9981    1.0000    -1.2563    0.9735
```

We then perform the IIR filtering using the following function:

```
y = sosfilt(sos,xn);
```

The MATLAB® program for this example is given in `example4_16.m`.

The Signal Processing Tool (SPTool) helps users to analyze signals, design and analyze filters, perform filtering, and analyze the spectra of signals. We can open this tool by typing

```
sptool
```

in the MATLAB® Command Window. There are three windows that can be accessed within the SPTool:

1. The **Signals** window is used to analyze the input signals. Signals from the workspace or a file can be loaded into the SPTool by clicking on **File** → **Import**. The **Import to SPTool** window allows users to select the data from either a file or the workspace.
2. The **Filters** window is used to design, analyze, and implement filters. This column uses the `fdatool` to design and analyze filters.
3. We can compute the spectrum by selecting the signal, and then clicking on the **Create** button in the **Spectra** column. To view the spectra of input and output signals, select the spectrum of input and the spectrum of output, and click on the **View** button to display both of them in the same window for easy comparison.

4.6 Practical Applications

In this section, we briefly introduce some practical applications of IIR filtering such as signal generation and audio equalization. The design of the specific IIR filters for parametric equalizers will be introduced in Chapter 10.

4.6.1 Recursive Resonators

Consider a simple second-order filter whose frequency response is dominated by a single peak at frequency ω_0. To make a peak at frequency ω_0, we place a pair of complex-conjugate poles inside the unit circle at

$$p_i = r_\mathrm{p}\, \mathrm{e}^{\pm \mathrm{j}\omega_0}, \tag{4.52}$$

where the radius (the distance from the origin to the pole) is within $0 < r_\mathrm{p} < 1$. The transfer function of this second-order IIR filter can be expressed as

$$\begin{aligned} H(z) &= \frac{A}{\left(1 - r_\mathrm{p}\, \mathrm{e}^{\mathrm{j}\omega_0} z^{-1}\right)\left(1 - r_\mathrm{p}\, \mathrm{e}^{-\mathrm{j}\omega_0} z^{-1}\right)} = \frac{A}{1 - 2r_\mathrm{p}\cos(\omega_0) z^{-1} + r_\mathrm{p}^2 z^{-2}} \\ &= \frac{A}{1 + a_1 z^{-1} + a_2 z^{-2}}, \end{aligned} \tag{4.53}$$

where A is a fixed gain used to normalize the filter's gain at ω_0 to unity such that $|H(\omega_0)| = 1$. The direct-form realization is shown in Figure 4.17.

The filter gain at ω_0 can be computed by replacing z in (4.53) by $\mathrm{e}^{\mathrm{j}\omega_0}$ and taking the absolute value as

$$|H(\omega_0)| = \frac{A}{\left|\left(1 - r_\mathrm{p}\, \mathrm{e}^{\mathrm{j}\omega_0}\, \mathrm{e}^{-\mathrm{j}\omega_0}\right)\left(1 - r_\mathrm{p}\, \mathrm{e}^{-\mathrm{j}\omega_0}\, \mathrm{e}^{-\mathrm{j}\omega_0}\right)\right|} = 1. \tag{4.54}$$

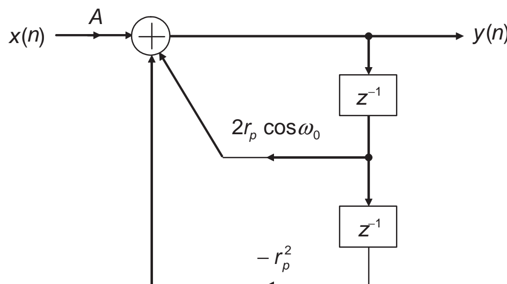


Figure 4.17 Signal-flow diagram of the second-order resonator filter

This condition can be used to set the gain value as

$$A = \left|\left(1 - r_{\mathrm{p}}\right)\left(1 - r_{\mathrm{p}}\, \mathrm{e}^{-2\mathrm{j}\omega_0}\right)\right| = \left(1 - r_{\mathrm{p}}\right)\sqrt{1 - 2r_{\mathrm{p}}\cos(2\omega_0) + r_{\mathrm{p}}^2}. \tag{4.55}$$

The 3-dB bandwidth of the resonator is equivalent to

$$|H(\omega)|^2 = \frac{1}{2}|H(\omega_0)|^2 = \frac{1}{2}. \tag{4.56}$$

There are two solutions on both sides of ω_0, and the bandwidth is the difference between these two frequencies. When the poles are close to the unit circle, the bandwidth is approximated as

$$\mathrm{BW} \cong 2\left(1 - r_{\mathrm{p}}\right). \tag{4.57}$$

This design criterion determines the value of r_{p} for a given BW. The closer r_{p} is to one, the sharper the peak.

From (4.53), the input/output equation of the resonator can be expressed as

$$y(n) = Ax(n) - a_1 y(n-1) - a_2 y(n-2), \tag{4.58}$$

where

$$a_1 = -2r_{\mathrm{p}}\cos\omega_0 \quad \text{and} \quad a_2 = r_{\mathrm{p}}^2. \tag{4.59}$$

This recursive oscillator is based on the marginally stable two-pole IIR filter where the complex-conjugate poles lie on the unit circle ($r_{\mathrm{p}} = 1$) at the angles $\pm\omega_0$ for generating the sinusoidal waveform at frequency ω_0. This recursive oscillator is the most efficient way of generating a sine wave, particularly if quadrature signals (sine and cosine signals) are required.

MATLAB® provides the function `iirpeak` for designing IIR peaking filters with the following syntax:

```
[NUM, DEN] = iirpeak(Wo, BW);
```

This function designs the second-order resonator with the peak at frequency `Wo` and the 3 dB bandwidth `BW`. In addition, we can use `[NUM,DEN] = iirpeak(Wo,BW,Ab)` to design a peaking filter with the bandwidth of `BW` at level `Ab` in dB.

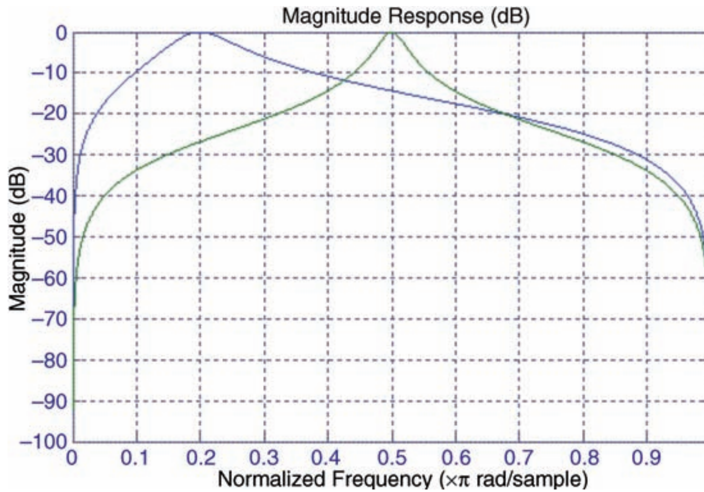


Figure 4.18 Magnitude responses of resonators

Example 4.17

Design two resonators to operate at 10 kHz sampling rate that have peaks at 1 kHz (0.2π) and 2.5 kHz (0.5π) and the 3 dB bandwidth of 500 Hz and 200 Hz, respectively. These filters can be designed using the following MATLAB® script (`example4_17.m`, adapted from the **Help** menu):

```
Fs = 10000;                  % Sampling rate
Wo = 1000/(Fs/2);            % First filter peak frequency
BW = 500/(Fs/2);             % First filter bandwidth
W1 = 2500/(Fs/2);            % Second filter peak frequency
BW1 = 200/(Fs/2);            % Second filter bandwidth
[b,a] = iirpeak(Wo,BW);      % Design first filter
[b1,a1] = iirpeak(W1,BW1);   % Design second filter
fvtool(b,a,b1,a1);           % Analyze both filters
```

The magnitude responses of both filters are shown in Figure 4.18. In the FVTool window, we select **Analysis** → **Pole/Zero Plot** to display the poles and zeros of both filters, which are shown in Figure 4.19. It can be clearly seen that the second filter (peak at 0.5π) has a narrower bandwidth (200 Hz) because its poles are close to the unit circle.

Similarly, MATLAB® provides the function `iirnotch` for designing IIR notch (narrow bandstop) filters with the following syntax:

```
[NUM, DEN] = iirnotch(Wo, BW);
```

This function designs second-order notch filters with the center of the stopband at frequency `Wo` and the 3-dB bandwidth `BW`.

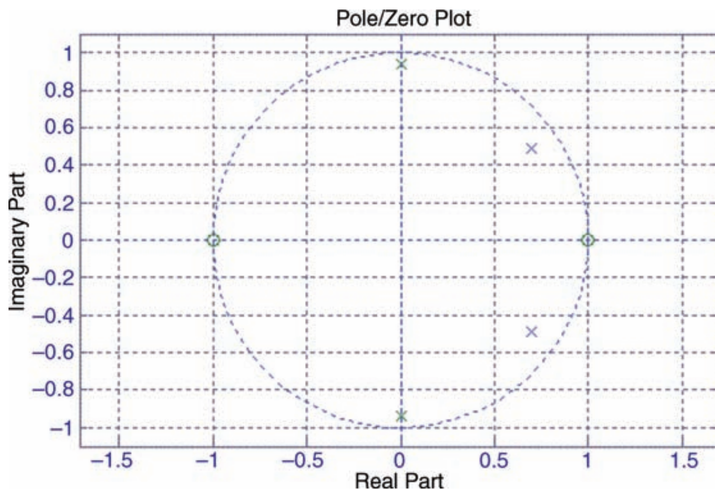


Figure 4.19 Pole/zero plot of two resonators

4.6.2 *Recursive Quadrature Oscillators*

Consider two causal impulse responses

$$h_c(n) = \cos(\omega_0 n)u(n) \tag{4.60a}$$

and

$$h_s(n) = \sin(\omega_0 n)u(n), \tag{4.60b}$$

where $u(n)$ is the unit-step function. Taking the z-transform of both functions, the corresponding system transfer functions are obtained as

$$H_c(z) = \frac{1 - \cos(\omega_0)z^{-1}}{1 - 2\cos(\omega_0)z^{-1} + z^{-2}} \tag{4.61a}$$

and

$$H_s(z) = \frac{\sin(\omega_0)z^{-1}}{1 - 2\cos(\omega_0)z^{-1} + z^{-2}}. \tag{4.61b}$$

The combined recursive structure with two outputs to realize these two transfer functions is illustrated in Figure 4.20. The implementation requires just two data memory locations and two multiplications. The difference equations for generating output signals are expressed as

$$y_c(n) = w(n) - \cos(\omega_0)w(n-1) \tag{4.62a}$$

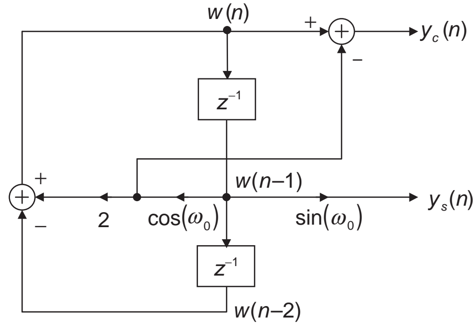


Figure 4.20 Recursive quadrature oscillators

and

$$y_s(n) = \sin(\omega_0)w(n-1), \tag{4.62b}$$

where $w(n)$ is the internal state variable that is updated as

$$w(n) = 2\cos(\omega_0)w(n-1) - w(n-2). \tag{4.63}$$

Applying the impulse signal $A\delta(n)$ to excite the oscillator is equivalent to presetting the following initial conditions:

$$w(-2) = -A \quad \text{and} \quad w(-1) = 0. \tag{4.64}$$

The waveform accuracy of the generated sine wave is limited primarily by the wordlength of the hardware. The quantization of the coefficient $\cos(\omega_0)$ in (4.62) and (4.63) will cause the actual output frequency to differ slightly from the desired frequency ω_0.

For some applications, only a sine wave is required. From (4.58) and (4.59), using the conditions that $x(n) = A\delta(n)$ and $r_p = 1$, we can generate the sinusoidal signal as

$$\begin{aligned} y_s(n) &= Ax(n) - a_1 y_s(n-1) - a_2 y_s(n-2) \\ &= 2\cos(\omega_0)y_s(n-1) - y_s(n-2) \end{aligned} \tag{4.65}$$

with the initial conditions

$$y_s(-2) = -A\sin(\omega_0) \quad \text{and} \quad y_s(-1) = 0. \tag{4.66}$$

The oscillating frequency defined by (4.65) is determined from its coefficient a_1 and its sampling frequency f_s, expressed as

$$f = \cos^{-1}\left(\frac{|a_1|}{2}\right)\frac{f_s}{2\pi} \text{ Hz}, \tag{4.67}$$

where the coefficient $|a_1| \leq 2$.

4.6.3 Parametric Equalizers

The design of IIR peak, low-shelf, and high-shelf filters specifically for audio parametric equalizers will be presented in Section 10.3. In this section, we design a simple IIR filter for a parametric equalizer from the resonator given in (4.53) by adding a pair of zeros near the poles at the same angles $\pm\omega_0$. That is, placing a pair of complex-conjugate zeros at

$$z_i = r_z \, e^{\pm j\omega_0}, \tag{4.68}$$

where $0 < r_z < 1$. Thus, the transfer function given in (4.53) becomes

$$\begin{aligned} H(z) &= \frac{\left(1 - r_z \, e^{j\omega_0} z^{-1}\right)\left(1 - r_z \, e^{-j\omega_0} z^{-1}\right)}{\left(1 - r_p \, e^{j\omega_0} z^{-1}\right)\left(1 - r_p \, e^{-j\omega_0} z^{-1}\right)} \\ &= \frac{1 - 2r_z \cos(\omega_0) z^{-1} + r_z^2 z^{-2}}{1 - 2r_p \cos(\omega_0) z^{-1} + r_p^2 z^{-2}} \\ &= \frac{1 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}. \end{aligned} \tag{4.69}$$

When $r_z < r_p$, the pole dominates the zero because it is closer to the unit circle than the zero. Thus, it will generate a peak at $\omega = \omega_0$. When $r_z > r_p$, the zero dominates the pole, thus providing a dip in the frequency response. When the pole and zero are very close to each other, the effects of the poles and zeros will be reduced, resulting in a flat response. Therefore, (4.69) provides a boost if $r_z < r_p$, or a cut if $r_z > r_p$. The amount of boost or cut is controlled by the difference between r_p and r_z. The distance from r_p to the unit circle will determine the bandwidth of the equalizer.

Example 4.18

Design a parametric equalizer with the peak at frequency 1.5 kHz and the sampling rate is 10 kHz. Assume the parameters $r_z = 0.8$ and $r_p = 0.9$. Part of the MATLAB® script (`example4_18.m`) is listed as follows:

```
rz=0.8; rp=0.9;                % Define radii of zero and pole
b=[1, -2*rz*cos(w0), rz*rz]; % Define numerator coefficients
a=[1, -2*rp*cos(w0), rp*rp]; % Define denominator coefficients
fvtool(b,a);                   % Analyze the equalizer
```

Since $r_z < r_p$, this filter provides a boost.

4.7 Experiments and Program Examples

This section implements different forms and realizations of IIR filters using C and TMS320C55xx assembly programs.

4.7.1 Direct-Form I IIR Filter Using Floating-Point C

The direct-form I IIR filter defined by the input/output equation (4.23) is implemented using the C function listed in Table 4.1. The input and output signal buffers are `x` and `y`, respectively. The current input data is passed to the function via variable `in`, and the filter output is saved on the top of the `y` buffer at the location `y[0]`. The IIR filter coefficients are stored in the arrays `a` and `b` with lengths `na` and `nb`, respectively. This IIR filter processes one sample at a time, that

Table 4.1 List of floating-point C function for direct-form I IIR filtering

```
void floatPoint_IIR(double in, double *x, double *y,
                    double *b, short nb, double *a, short na)
{
   double z1,z2;
   short i;

   for(i=nb-1; i>0; i--)          // Update the buffer x[]
      x[i] = x[i-1];

   x[0] = in;                     // Insert new data to x[0]

   for(z1=0, i=0; I<nb; i++)      // Filter x[] with coefficients in b[]
      z1 += x[i] * b[i];

   for(i=na-1; i>0; i--)          // Update y buffer
      y[i] = y[i-1];

   for(z2=0, i=1; I<na; i++)      // Filter y[] with coefficients in a[]
      z2 += y[i] * a[i];

   y[0] = z1 - z2;                // Place the final result to y[0]
}
```

is, sample-by-sample processing. The input signal for the experiment contains three sinusoids at frequencies 800, 1800, and 3300 Hz. The IIR bandpass filter is designed using the following MATLAB® script:

```
Rp=0.1;                                       % Passband ripple
Rs=60;                                        % Stopband attenuation
[N,Wn]=ellipord(836/4000,1300/4000,Rp,Rs);    % Filter order & scaling
                                              % factor
[b,a]=ellip(N,Rp,Rs,Wn);                      % Lowpass IIR filter
[num,den]=iirlp2bp(b,a,0.5,[0.25, 0.75]);     % Bandpass IIR filter
```

This bandpass filter will pass an 1800 Hz sine wave and attenuate 800 and 3300 Hz sinusoids. Table 4.2 lists the files used for the experiment.

Table 4.2 File listing for the experiment Exp4.1

Files	Description
floatPoint_directIIRTest.c	Program for testing floating-point IIR filter
floatPoint_directIIR.c	C function for floating-point IIR filter
floatPointIIR.h	C header file
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
input.pcm	Input data file

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the project to filter the signal in the given input data file.
3. Validate the output signal using CCS or MATLAB® to analyze the magnitude spectrum of the output signal to examine whether the 800 and 3300 Hz components are attenuated by 60 dB.
4. Design an IIR filter with two stopbands centered at frequencies 800 and 3300 Hz, each bandstop filter has bandwidth 400 Hz, and the stopband attenuation is 60 dB. Redo the experiment and compare the magnitude spectrum of output signal to the result obtained in step 3.

4.7.2 Direct-Form I IIR Filter Using Fixed-Point C

The fixed-point C implementation of IIR filters can be done by modifying the floating-point C program from the previous experiment. It is important to note that the data type `long` must be used for integer multiplication. As discussed in Chapter 2, the product of the multiplication resides in the upper portion of the `long` variables when using integers to represent fractional signal samples, such as using the Q15 data format. The fixed-point C function for the IIR filter is listed in Table 4.3, where we use the Q11 format for filter coefficients and Q15 for signal samples. Table 4.4 lists the files used for the experiment.

Table 4.3 Fixed-point C implementation of direct-form I IIR filter

```
void fixPoint_IIR(short in, short *x, short *y,
                  short *b, short nb, short *a, short na)
{
  long z1,z2,temp;
  short i;

  for(i=nb-1; i>0; i--)       // Update the buffer x[]
      x[i] = x[i-1];

  x[0] = in;                  // Insert new data to x[0]

  for(z1=0, i=0; i<nb; i++)  // Filter x[] with coefficients in b[]
      z1 += (long)x[i] * b[i];

  for(i=na-1; i>0; i--)       // Update y[] buffer
      y[i] = y[i-1];

  for(z2=0, i=1; i<na; i++)  // Filter y[] with coefficients in a[]
      z2 += (long)y[i] * a[i];

  z1 = z1 - z2;               // Q15 data filtered using Q11 coefficient
  z1 += 0x400;                // Rounding
  y[0] = (short)(z1 - z2);    // Place the final result in y[0]
}
```

Table 4.4 File listing for the experiment Exp4.2

Files	Description
`fixPoint_directIIRTest.c`	Program for testing fixed-point IIR filter
`fixPoint_directIIR.c`	C function for fixed-point IIR filter
`fixPointIIR.h`	C header file
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`input.pcm`	Input data file
`noise.pcm`	Experiment testing data file

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the project to filter the signal in the given input data file.
3. Use CCS to plot the magnitude spectrum of the output signal to examine whether 800 and 3300 Hz sinusoidal components are attenuated by 60 dB. Compare the result to the floating-point C implementation result obtained in Exp4.1 to examine the finite-precision effects, especially around the passband at 1800 Hz to observe the coefficients' quantization noise in fixed-point implementation of a high-order direct-form IIR filter.
4. Use `noise.pcm` (included in this experiment) as the input data file, and redo experiments Exp4.1 and Exp4.2. Plot and compare the magnitude spectra of output signals obtained from these two experiments. Also, compare these spectra to the magnitude response of the bandpass filter.
5. Design a lowpass filter using MATLAB® to pass 800 Hz tone and attenuate frequency components above 1200 Hz by 60 dB. Redo the experiment using this new filter and validate the experimental result by showing the magnitude spectrum of the output signal.
6. Design a highpass filter using MATLAB®, the passband starting at 2000 Hz with 45 dB attenuation in the stopband. Redo the experiment using this new filter and verify the filtering result.

4.7.3 Cascade IIR Filter Using Fixed-Point C

The cascade structure shown in Figure 4.8 uses direct-form II second-order IIR filters. As mentioned earlier, the zero-overhead repeat loops, multiply–accumulate instructions, and circular addressing mode are three important features of modern digital signal processors. To better utilize these features, the cascade structures of second-order IIR sections are implemented using the fixed-point C program. In the experiment, the C statements are arranged to mimic the multiply–accumulate and circular addressing operations of the C55xx processors. The fixed-point C program listed in Table 4.5 implements the IIR filter with `Ns` cascaded second-order sections.

The coefficient and signal arrays are configured as circular buffers that simulate the architecture of the C55xx. These circular buffers are shown in Figure 4.21. The signal buffer for each second-order section contains two elements, $w_k(n-1)$ and $w_k(n-2)$, as defined by the difference equations (4.38) and (4.39). The signal buffer is arranged to group together each section's signal elements at time $n-1$ in the first half of the buffer, and all the elements at

Table 4.5 Fixed-point C implementation of cascade IIR filter

```
void cascadeIIR(short *x, short Nx, short *y, short *coef, short Ns,
                short *w)
{
   short i,j,n,m,l,s;
   short temp16;
   long w_0,temp32;

   m=Ns*5;                              // Set up circular buffer coef[]
   s=Ns*2;                              // Set up circular buffer w[]

   for (j=0,l=0,n=0; n<Nx; n++)         // IIR filtering
   {
      w_0 = (long)x[n]<<12; // Scale input to prevent overflow
      for (i=0; i<Ns; i++)
      {
         temp32 = (long)(*(w+l)) * *(coef+j); j++; l=(l+Ns)%s;
         w_0 -= temp32<<1;
         temp32 = (long)(*(w+l)) * *(coef+j); j++;
         w_0 -= temp32<<1;
         w_0 += 0x4000;                 // Rounding
         temp16 = *(w+l);
         *(w+l) = (short)(w_0>>15); // Save in Q15 format
         w_0 = (long)temp16 * *(coef+j); j++;
         w_0 <<= 1;
         temp32 = (long)*(w+l) * *(coef+j); j++; l=(l+Ns)%s;
         w_0 += temp32<<1;
         temp32 = (long)*(w+l) * *(coef+j); j=(j+1)%m; l=(l+1)%s;
         w_0 += temp32<<1;
         w_0 += 0x800;                  // Rounding
   }
   y[n] = (short)(w_0>>12);             // Output in Q15 format
  }
}
```

time $n-2$ are placed in the second half of the buffer. The pointer address for the signal buffer is initialized to start at the first sample, $w_1(n-1)$. The address index offset equals the number of sections. The filter coefficients are arranged in the order a_{1k}, a_{2k}, b_{2k}, b_{0k}, and b_{1k} for each of `k` cascaded IIR sections, and the coefficient pointer is initialized to point at the first coefficient, a_{11}. It is important to note that the specific coefficient ordering for b_{2k}, b_{0k}, and b_{1k} is due to the implementation of the circular buffer required by the C55xx assembly programs, which is not required by C programs. Such a special arrangement for this C program is intended to show how it will be done by the assembly program in Section 4.7.5. The circular pointers for the coefficient buffer `C[]` and signal buffer `w[]` are updated by `j=(j+1)%m` and `l=(l+1)%s`, where `m` and `s` are the sizes of the coefficient and signal buffers, respectively. In this experiment, four second-order IIR sections ($N_s = 4$) are used so $m = 5N_s = 20$ and

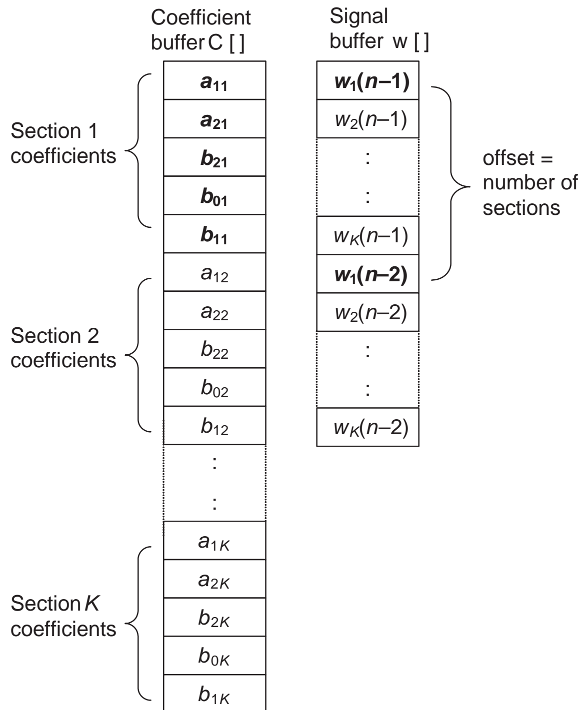


Figure 4.21 Configuration of IIR filter coefficient and signal buffers

$s = 2N_s = 8$. The filter coefficient vector `C[]` and the signal vector `w[]` are defined in the C program `fixPoint_cascadeIIRTest.c`. By changing the filter order N_s, different IIR filters with different orders can be tested using the same IIR filter function, `fixPoint_cascadetIIR.c`.

The test function reads in the filter coefficient header file, `fdacoefsMATLAB.h`, which is generated by the FDATool. These coefficients are arranged in the order of a_{1k}, a_{2k}, b_{2k}, b_{0k}, and b_{1k} for the experiment. Table 4.6 lists the files used for the experiment, where the input data file `in.pcm` consists of three sinusoids at 800, 1500, and 3300 Hz with 8000 Hz sampling rate. The IIR filter will attenuate the 800 and 3300 Hz components in the input signal.

Table 4.6 File listing for the experiment Exp4.3

Files	Description
`fixPoint_cascadeIIRTest.c`	Program for testing cascade IIR filter
`fixPoint_cascadetIIR.c`	C function for fixed-point second-order IIR filter
`cascadeIIR.h`	C header file
`tistdtypes.h`	Standard type define header file
`fdacoefsMATLAB.h`	FDATool-generated C header file
`tmwtypes.h`	Data type definition file for MATLAB® C header file
`c5505.cmd`	Linker command file
`in.pcm`	Input data file

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the project to filter the input signal in the given data file.
3. Use MATLAB® to plot the magnitude spectrum of input and output signals and compare them.
4. Design a highpass filter that will attenuate frequencies below 1800 Hz at least 60 dB. Redo the experiment using this highpass filter. Verify the filtering result. The 3300 Hz tone should pass, while the 800 and 1500 Hz tones are attenuated.
5. Convert the direct-form I realization of the IIR filter presented in Exp4.2 to a cascade second-order IIR filter. Redo experiment Exp4.2 using this cascade IIR filter. Compare and analyze the differences of the filtering results to Exp4.2, which are caused by using different structures.

4.7.4 Cascade IIR Filter Using Intrinsics

The C intrinsics are C functions that will produce optimized assembly statements in compile time. These intrinsics are specified with a leading underscore and can be called by C programs. For example, the multiply–accumulation operation, `z+=x*y`, can be implemented by the following intrinsic:

```
short x,y;
long z;
z = _smac(z,x,y);        // Perform signed z=z+x*y
```

This intrinsic performs the signed multiple–accumulation operation `z+=x*y` with the following assembly instruction:

```
macm Xmem,Ymem,Acx
```

Table 4.7 lists the intrinsics supported by the TMS320C55xx C compiler.

Table 4.7 Intrinsics supported by the TMS320C55xx C compiler

C Compiler Intrinsics	Description
`short _sadd(short src1, short src2);`	Adds two 16-bit integers with SATA set, producing a saturated 16-bit result
`long _lsadd(long src1, long src2);`	Adds two 32-bit integers with SATD set, producing a saturated 32-bit result
`short _ssub(short src1, short src2);`	Subtracts `src2` from `src1` with SATA set, producing a saturated 16-bit result
`long _lssub(long src1, long src2);`	Subtracts `src2` from `src1` with SATD set, producing a saturated 32-bit result
`short _smpy(short src1, short src2);`	Multiplies `src1` and `src2` and shifts the result left by one. Produces a saturated 16-bit result. (SATD and FRCT are set)
`long _lsmpy(short src1, short src2);`	Multiplies `src1` and `src2` and shifts the result left by one. Produces a saturated 32-bit result. (SATD and FRCT are set)
`long _smac(long src, short op1, short op2);`	Multiplies `op1` and `op2`, shifts the result left by one, and adds it to `src`. Produces a saturated 32-bit result. (SATD, SMUL, and FRCT are set)

(*continued*)

Table 4.7 (*Continued*)

C Compiler Intrinsics	Description
`long _smas(long src, short op1, short op2);`	Multiplies `op1` and `op2`, shifts the result left by one, and subtracts it from `src`. Produces a 32-bit result. (SATD, SMUL, and FRCT are set)
`short _abss(short src);`	Creates a saturated 16-bit absolute value. `_abss`(0x8000) => 0x7FFF (SATA set)
`long _labss(long src);`	Creates a saturated 32-bit absolute value. `_labss`(0x8000000) => 0x7FFFFFFF (SATD set)
`long _labss(long src);`	Creates a saturated 32-bit absolute value. `_labss`(0x8000000) => 0x7FFFFFFF (SATD set)
`short _sneg(short src);`	Negates the 16-bit value with saturation `_sneg`(0xffff8000) => 0x00007FFF
`long _lsneg(long src);`	Negates the 32-bit value with saturation. `_lsneg`(0x80000000) => 0x7FFFFFFF
`short _smpyr(short src1, short src2);`	Multiplies `src1` and `src2`, shifts the result left by one, and rounds by adding 2^{15} to the result. (SATD and FRCT are set)
`short _smacr(long src, short op1, short op2);`	Multiplies `op1` and `op2`, shifts the result left by one, adds the result to `src`, and then rounds the result by adding 2^{15}. (SATD, SMUL, and FRCT are set)
`short _smasr(long src, short op1, short op2);`	Multiplies `op1` and `op2`, shifts the result left by one, subtracts the result from `src`, and then rounds the result by adding 2^{15}. (SATD, SMUL, and FRCT set)
`short _norm(short src);`	Produces the number of left shifts needed to normalize `src`
`short _lnorm(long src);`	Produces the number of left shifts needed to normalize `src`
`short _rnd(long src);`	Rounds `src` by adding 2^{15}. Produces a 16-bit saturated result. (SATD set)
`short _sshl(short src1, short src2);`	Shifts `src1` left by `src2` and produces a 16-bit result. The result is saturated if `src2` is less than or equal to eight. (SATD set)
`long _lsshl(long src1, short src2);`	Shifts `src1` left by `src2` and produces a 32-bit result. The result is saturated if `src2` is less than or equal to eight. (SATD set)
`short _shrs(short src1, short src2);`	Shifts `src1` right by `src2` and produces a 16-bit result. Produces a saturated 16-bit result. (SATD set)
`long _lshrs(long src1, short src2);`	Shifts `src1` right by `src2` and produces a 32-bit result. Produces a saturated 32-bit result. (SATD set)
`short _addc(short src1, short src2);`	Adds `src1`, `src2`, and carry bit and produces a 16-bit result
`long _laddc(long src1, short src2);`	Adds `src1`, `src2`, and carry bit and produces a 32-bit result

In this experiment, the fixed-point C function for the cascade IIR filter from the previous experiment is modified by using intrinsics. Table 4.8 lists the fixed-point C implementation with coefficients represented by the Q14 format. Since $N_s = 4$, the filter length equals eight, which is a power-of-2 number, thus the modulo operation "`%s`" is replaced with the "`&s`" operation by changing $s = 2N_s - 1 = 7$ for updating the circular signal buffer. Table 4.9 lists the files used for the experiment.

Table 4.8 Implementation of cascade IIR filter using fixed-point C with intrinsics

```
void intrinsics_IIR(short *x, short Nx, short *y,
                     short *coef, short Ns, short *w)
{
  short i,j,n,m,l,s;
  short temp16;
  long w_0;

  m=Ns*5;                        // Set up circular buffer coef[]
  s=Ns*2-1;                      // Set up circular buffer w[]

  for (j=0,l=0,n=0; n<Nx; n++)   // IIR filtering
  {
    w_0 = (long)x[n]<<12;        // Scale input to prevent overflow
    for (i=0; i<Ns; i++)
    {
      w_0 = _smas(w_0,*(w+l),*(coef+j)); j++; l=(l+Ns)&s;
      w_0 = _smas(w_0,*(w+l),*(coef+j)); j++;

      temp16 = *(w+l);
      *(w+l) = (short)(w_0>>15); // Save in Q15 format

      w_0 = _lsmpy(temp16,*(coef+j)); j++;
      w_0 = _smac(w_0,*(w+l),*(coef+j)); j++; l=(l+Ns)&s;
      w_0 = _smac(w_0,*(w+l),*(coef+j)); j=(j+1)%m; l=(l+1)&s;
    }
    y[n] = (short)(w_0>>12);     // Output in Q15 format
  }
}
```

Table 4.9 File listing for the experiment Exp4.4

Files	Description
intrinsics_IIRTest.c	Program for testing IIR filter using intrinsics
intrinsics_IIR.c	Intrinsics implementation of second-order IIR filter
intrinsics_IIR.h	C header file
tistdtypes.h	Standard type define header file
fdacoefsMATLAB.h	FDATool-generated C header file
tmwtypes.h	Data type definition file for MATLAB® C header file
c5505.cmd	Linker command file
in.pcm	Input data file

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the project to filter the input signal in the given data file.
3. Use MATLAB® to plot the magnitude spectrum of the output signal and compare it to the input signal.
4. Profile the clock cycles needed for this experiment and compare them to Exp4.3 to evaluate the efficiency achieved by using C55xx intrinsics.
5. Use MATLAB® to design a bandstop filter (such as using MATLAB® function `iirnotch` introduced in Section 4.6.1) that will attenuate an 800 Hz sine wave by 60 dB. Repeat the experiment using the notch filter. Use MATLAB® to plot the magnitude spectrum of the output signal and compare it to the magnitude spectrum of the input signal.
6. Modify the fixed-point C program presented in Exp4.2 by using C55xx intrinsics and redo the experiment. Profile the required clock cycles needed for the fixed-point C with and without using intrinsics.

4.7.5 Cascade IIR Filter Using Assembly Program

This experiment implements a cascade IIR filter using the C55xx assembly program and block processing method. The IIR filter coefficients and signal samples are represented using the Q14 format. To compensate for the scaled input signal, the filter output $y(n)$ is scaled up to the Q15 format and stored in memory with rounding.

For the cascade IIR filter with K second-order sections, the signal and coefficient buffers are arranged as shown in Figure 4.21 for using the C55xx circular addressing mode. The fixed-point C program from the previous experiments serves as an example to show the flow of the C55xx assembly code, particularly the circular addressing operations. Table 4.10 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the project to filter the input signal from the given data file. Verify the result by comparing the output signal to the output signal obtained from the previous fixed-point C experiments.

Table 4.10 File listing for the experiment Exp4.5

Files	Description
`asmIIRTest.c`	Program for testing assembly IIR filter
`asmIIR.asm`	Assembly implementation of second-order IIR filter
`asmIIR.h`	C header file
`tistdtypes.h`	Standard type define header file
`fdacoefsMATLAB.h`	FDATool-generated C header file
`tmwtypes.h`	Data type definition file for MATLAB® C header file
`c5505.cmd`	Linker command file
`in.pcm`	Input data file

3. Modify experiments Exp4.1 through Exp4.5 such that they all use the same input data file `in.pcm` and the IIR bandpass filter `fdacoefsMATLAB.h` provided by Exp4.5. Use CCS to profile the required clock cycles of the IIR filters in experiments Exp4.1 through Exp4.5 and compare their efficiency.

4.7.6 Real-Time IIR Filtering

This experiment modifies the real-time FIR filtering program developed in Chapter 3 for real-time IIR filtering. Table 4.11 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Connect a headphone to the eZdsp audio output jack and connect an audio source to the eZdsp audio input jack. Load and run the program at a sampling rate of 48 000 Hz.
3. Listen to the audio playback from the headphone to verify that the high-frequency audio components in the input signal are attenuated by the lowpass filter with 2000 Hz cutoff frequency used by this experiment.
4. Use MATLAB® to design a new lowpass IIR filter with 1200 Hz cutoff frequency at 8000 Hz sampling rate. Configure the eZdsp to run at 8000 Hz sampling rate using this new lowpass filter for real-time experiments. Compare the result to the result from step 3 using subjective evaluation.
5. Design a highpass filter that has a cutoff frequency of 1200 Hz at 8000 Hz sampling rate. Modify the program such that the left-channel signal will be filtered by the lowpass filter with cutoff frequency 1200 Hz (step 4), and the right-channel audio will be filtered by this highpass filter. Listen to both channels of audio output to evaluate different filtering effects.
6. Use the MATLAB® FDATool to design a lowpass and a highpass filter with a cutoff frequency of 3000 Hz at 48 000 Hz sampling rate. Redo step 5 using the eZdsp running at 48 000 Hz sampling rate.

Table 4.11 File listing for the experiment Exp4.6

Files	Description
`realtimeIIRTest.c`	Program for testing real-time IIR filter
`iirFilter.c`	Filter control and initialization functions
`asmIIR.asm`	Assembly program for second-order IIR filter
`vector.asm`	Vector table for real-time experiment
`asmIIR.h`	Header file for assembly IIR experiment
`iir_lp_2khz_48khz.h`	FDATool-generated C coefficients header file
`tmwtypes.h`	Data type definition file for MATLAB® C header file
`tistdtypes.h`	Standard type define header file
`dma.h`	Header file for DMA functions
`dmaBuff.h`	Header file for DMA data buffer
`i2s.h`	i2s header file for i2s functions
`Ipva200.inc`	C5505 processor include file
`myC55xxUtil.lib`	BIOS audio library
`c5505.cmd`	Linker command file

Table 4.12 File listing for the experiment Exp4.7

Files	Description
parametric_equalizerTest.c	Program for testing parametric equalizer
fixPoint_cascadetIIR.c	C function of parametric equalizer
cascadeIIR.h	Header file for the experiment
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
input.pcm	Data file consists of two tones

4.7.7 Parametric Equalizer Using Fixed-Point C

This experiment designs and implements a parametric equalizer using resonators at 200 and 1000 Hz for the sampling rate of 8000 Hz. The experiment initializes the parameters of the IIR filters based on (4.69) using the selected `rz` and `rp` values from the table given in `parametric_equalizerTest.c`. This table provides the equalizer gain with an adjustable (dynamic) range of ±6 dB in 1 dB steps. The default setting is to attenuate the 1000 Hz component by 6 dB and boost the 200 Hz component by 6 dB. The direct-form II IIR filter is used as the resonator. The equalizer coefficients are generated by the function `coefGen()` during the initialization stage. The experiment is implemented using fixed-point C. Table 4.12 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the project using the data file provided by the companion software package.
3. Use the CCS graphical tool to plot and compare the magnitude spectrum of the equalizer's input and output signals. Change the default settings to use different resonator frequencies and boost/attenuation levels (by changing `rz` and `rp` values in `parametric_equalizerTest.c`) to examine the equalization effects.
4. Design a new equalizer for the sampling rate of 48 000 Hz. Use the interpolator experiment from Chapter 3 to convert the input data to 48 000 Hz. Redo the experiment using the new equalizer and audio file at the sampling rate of 48 000 Hz.
5. Design a three-band equalizer at normalized frequencies 0.05, 0.25, and 0.5 with 8000 Hz sampling rate based on the two-band equalizer used in previous steps. The equalizer must have a dynamic range of ±9 dB in 1 dB steps for each resonator (IIR filter) frequency. This requires the design of a new set of tables similar to the tables `gain200[][]` and `gain1000[][]` in `parametric_equalizerTest.c` used by the two-band equalizer. Redo the experiment to verify the performance of the three-band equalizer.

4.7.8 Real-Time Parametric Equalizer

This experiment implements a two-band equalizer and tests its performance in real time using the sampling rate of 8000 Hz. There are two default settings: the first one initializes coefficients in `C1[]` for the equalizer having 6 dB gain at 1000 Hz and −6 dB attenuation at 200 Hz for the treble effect. The second default setting initializes coefficients in `C2[]` for the equalizer having 6 dB gain at 200 Hz and −6 dB attenuation at 1000 Hz. The experiment

Table 4.13 File listing for the experiment Exp4.8

Files	Description
`realtimeEQTest.c`	Program for testing real-time equalizer
`equalizer.c`	Initialization and control functions
`asmIIR.asm`	Assembly implementation of second-order IIR filter
`vector.asm`	Vector table for real-time experiment
`asmIIR.h`	Header file for assembly IIR filter
`tistdtypes.h`	Standard type define header file
`dma.h`	Header file for DMA functions
`dmaBuff.h`	Header file for DMA data buffer
`i2s.h`	i2s header file for i2s functions
`Ipva200.inc`	C5505 processor include file
`myC55xxUtil.lib`	BIOS audio library
`c5505.cmd`	Linker command file

uses treble and bass filters having ± 6 dB range boost/attenuation and can be initialized with different gain values from -6 to $+6$ dB. Table 4.13 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Connect a headphone and an audio source to the eZdsp audio ports. Load and run the program.
3. Listen to the audio output to evaluate the performance of the equalizer. Modify the default setting by changing the resonator frequencies and gain/attenuation values. Redo the experiment to evaluate different equalization effects.
4. Modify the experiment to run properly at the sampling rate 48 000 Hz.
5. Design a three-band equalizer with normalized resonator frequencies at 0.05, 0.25, and 0.5 with 8000 Hz sampling rate based on the two-band equalizer. The equalizer must have a dynamic range of ± 9 dB in 1 dB steps for each IIR filter. Redo the real-time experiment to evaluate the performance of the new three-band equalizer.

Exercises

4.1. Compute the Laplace transform of the unit-impulse function $\delta(t)$ and the unit-step function $u(t)$.

4.2. Given the analog system transfer function

$$H(s) = \frac{2s+3}{s^2+3s+2},$$

find the poles and zero of the system and discuss its stability.

4.3. Given the digital system transfer function

$$H(z) = \frac{0.5(z^2+0.55z-0.2)}{z^3-0.7z^2-0.84z+0.544},$$

realize the system using a cascade of direct-form II sections.

4.4. Draw the direct-form I and II realizations of the transfer function

$$H(z) = \frac{(z^2 + 2z + 2)(z + 0.6)}{(z - 0.8)(z + 0.8)(z^2 + 0.1z + 0.8)}.$$

4.5. Given the IIR filter with transfer function

$$H(z) = \frac{(1 + 1.414z^{-1} + z^{-2})(1 + 2z^{-1} + z^{-2})}{(1 - 0.8z^{-1} + 0.64z^{-2})(1 - 1.0833z^{-1} + 0.25z^{-2})},$$

find the poles and zeros of the filter, and use the stability triangle to check if $H(z)$ is a stable filter. Verify the results using MATLAB®.

4.6. Consider the second-order IIR filter defined by the input/output equation

$$y(n) = x(n) + a_1 y(n-1) + a_2 y(n-2), \quad n \geq 0.$$

Find the transfer function $H(z)$, and discuss the stability conditions related to the following cases:
1. $a_1^2/4 + a_2 < 0$.
2. $a_1^2/4 + a_2 > 0$.
3. $a_1^2/4 + a_2 = 0$.

4.7. A first-order allpass filter has the transfer function

$$H(z) = \frac{z^{-1} - a}{1 - az^{-1}}.$$

(a) Draw the direct-form I and II realizations.
(b) Show that $|H(\omega)| = 1$ for all ω.
(c) Sketch the phase response of this filter.
(d) Plot the magnitude and phase responses using MATLAB® for different values of a.

4.8. Given the transfer function of a six-order IIR as

$$H(z) = \frac{6 + 17z^{-1} + 33z^{-2} + 25z^{-3} + 20z^{-4} - 5z^{-5} + 8z^{-6}}{1 + 2z^{-1} + 3z^{-2} + z^{-3} + 0.2z^{-4} - 0.3z^{-5} - 0.2z^{-6}},$$

factorize the transfer function and realize it as cascaded second-order IIR sections using MATLAB®.

4.9. Given the fourth-order IIR transfer function

$$H(z) = \frac{12 - 2z^{-1} + 3z^{-2} + 20z^{-4}}{6 - 12z^{-1} + 11z^{-2} - 5z^{-3} + z^{-4}}.$$

1. Factorize $H(z)$ using MATLAB®.
2. Develop two different cascade realizations.
3. Develop two different parallel realizations.

4.10. Design and plot the magnitude response of an elliptic IIR lowpass filter with the following specifications using MATLAB®: passband edge at 1600 Hz, stopband edge at 2000 Hz, passband ripple of 0.5 dB, and minimum stopband attenuation of 40 dB with a sampling rate of 8000 Hz. Analyze the designed filter using the FVTool.

4.11. Use the FDATool to design the IIR filter specified in Problem 4.10 using following methods:

1. Butterworth.
2. Chebyshev type I.
3. Chebyshev type II
4. Bessel.

Show both the magnitude and phase responses of the designed filters, and indicate the required filter orders

4.12. Redo Problem 4.10 using the FDATool, compare the results to Problem 4.10, and design a quantized filter using 16-bit fixed-point processors.

4.13. Redo Problem 4.12 for designing an 8-bit fixed-point filter. Compare the differences to the 16-bit filter designed in Problem 4.12.

4.14. Design a Butterworth IIR bandpass filter with the following specifications: passband edges at 450 and 650 Hz, stopband edges at 350 and 750 Hz, passband ripple of 1 dB, minimum stopband attenuation of 60 dB, and sampling rate of 8000 Hz. Analyze the designed filter using the FVTool.

4.15. Redo Problem 4.14 using the FDATool, compare the results to Problem 4.14, and design a quantized filter for 16-bit fixed-point digital signal processors.

4.16. Design a Chebyshev type I IIR highpass filter with passband edge at 700 Hz, stopband edge at 500 Hz, passband ripple of 1 dB, and minimum stopband attenuation of 32 dB. The sampling frequency is 2000 Hz. Analyze the designed filter using the FVTool.

4.17. Redo Problem 4.16 using the FDATool, compare the results to Problem 4.16, and design a quantized filter for 16-bit fixed-point processors.

4.18. Given the IIR lowpass filter with transfer function

$$H(z) = \frac{0.0662(1 + 3z^{-1} + 3z^{-2} + z^{-3})}{1 - 0.9356z^{-1} + 0.5671z^{-2} - 0.1016z^{-3}},$$

plot the impulse response using an appropriate MATLAB® function, and analyze the result using the FVTool.

4.19. It is interesting to examine the frequency response of the second-order resonator filter as the radius r_p and the pole angle ω_0 are varied. Use MATLAB® to compute and plot the magnitude responses for $\omega_0 = \pi/2$ and various values of r_p. Also, compute and plot the magnitude responses for $r_p = 0.95$ and various values of ω_0.

References

1. Ahmed, N. and Natarajan, T. (1983) *Discrete-Time Signals and Systems*, Prentice Hall, Englewood Cliffs, NJ.
2. Ingle, V.K. and Proakis, J.G. (1997) *Digital Signal Processing Using MATLAB V.4*, PWS Publishing, Boston, MA.
3. The MathWorks, Inc. (1994) *Signal Processing Toolbox for Use with MATLAB*.
4. Oppenheim, A.V. and Schafer, R.W. (1989) *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
5. Orfanidis, S.J. (1996) *Introduction to Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
6. Proakis, J.G. and Manolakis, D.G. (1996) *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd edn, Prentice Hall, Englewood Cliffs, NJ.
7. Mitra, S.K. (1998) *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York.
8. Grover, D. and Deller, J.R. (1999) *Digital Signal Processing and the Microcontroller*, Prentice Hall, Englewood Cliffs, NJ.
9. Taylor, F. and Mellott, J. (1998) *Hands-On Digital Signal Processing*, McGraw-Hill, New York.
10. Stearns, S.D. and Hush, D.R. (1990) *Digital Signal Analysis*, 2nd edn, Prentice Hall, Englewood Cliffs, NJ.
11. Soliman, S.S. and Srinath, M.D. (1998) *Continuous and Discrete Signals and Systems*, 2nd edn, Prentice Hall, Englewood Cliffs, NJ.
12. Jackson, L.B. (1989) *Digital Filters and Signal Processing*, 2nd edn, Kluwer Academic, Boston, MA.
13. The MathWorks, Inc. (2000) Using MATLAB, Version 6.
14. The MathWorks, Inc. (2004) Signal Processing Toolbox User's Guide, Version 6.
15. The MathWorks, Inc. (2004) Filter Design Toolbox User's Guide, Version 3.
16. The MathWorks, Inc. (2004) Fixed-Point Toolbox User's Guide, Version 1.

5

Frequency Analysis and the Discrete Fourier Transform

This chapter introduces the properties, applications, and implementations of the discrete Fourier transform (DFT) and the fast Fourier transform algorithms that are now widely used for frequency analysis, fast convolution, and many other applications [1–9].

5.1 Fourier Series and Fourier Transform

This section briefly introduces the Fourier series for periodic analog signals, and the Fourier transform for finite-energy analog signals.

5.1.1 Fourier Series

There are several different forms of Fourier series, such as trigonometric Fourier series. In this chapter, we introduce only the complex Fourier series, which has a similar form to the Fourier transform.

A periodic signal can be represented as the sum of an infinite number of harmonic-related sinusoids or complex exponentials. The complex Fourier series representation of a periodic signal $x(t)$ with period T_0, that is, $x(t) = x(t + T_0)$, is defined as

$$x(t) = \sum_{k=-\infty}^{\infty} c_k \mathrm{e}^{\mathrm{j}k\Omega_0 t}, \tag{5.1}$$

where c_k are the Fourier series coefficients, $\Omega_0 = 2\pi/T_0$ is the fundamental frequency, and k is the integer frequency index for the kth harmonic at frequency $k\Omega_0$.

The kth Fourier series coefficient c_k is defined as

$$c_k = \frac{1}{T_0}\int_0^{T_0} x(t)\mathrm{e}^{-\mathrm{j}k\Omega_0 t}\,\mathrm{d}t. \tag{5.2}$$

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

For an even function $x(t)$, it is easier to calculate the interval from $-T_0/2$ to $T_0/2$. The term

$$c_0 = \frac{1}{T_0}\int_0^{T_0} x(t)\mathrm{d}t$$

is called the DC (or dc) component because it equals the average of $x(t)$ over one period.

Example 5.1

A rectangular pulse train is a periodic signal with period T_0 and can be expressed as

$$x(t) = \begin{cases} A, & kT_0 - \tau/2 \le t \le kT_0 + \tau/2 \\ 0, & \text{otherwise.} \end{cases} \tag{5.3}$$

where $k = 0, \pm 1, \pm 2, \ldots$, and $\tau < T_0$ is the width of a rectangular pulse with amplitude A. Since $x(t)$ is an even function, its Fourier coefficients can be calculated as

$$c_k = \frac{1}{T_0}\int_{-T_0/2}^{T_0/2} A\mathrm{e}^{-\mathrm{j}k\Omega_0 t}\,\mathrm{d}t = \frac{A}{T_0}\left[\frac{\mathrm{e}^{-\mathrm{j}k\Omega_0 t}}{-\mathrm{j}k\Omega_0}\bigg|_{-\tau/2}^{\tau/2}\right] = \frac{A\tau}{T_0}\frac{\sin(k\Omega_0\tau/2)}{k\Omega_0\tau/2}. \tag{5.4}$$

This equation shows that c_k has the maximum value of $A\tau/T_0$ at the DC frequency $\Omega_0 = 0$, gradually decays to zero as $\Omega_0 \to \pm\infty$, and equals zero at frequencies that are multiples of π.

The plot of $|c_k|$ versus frequency index k is called the magnitude spectrum, which shows that the power of the periodic signal is distributed at the frequency components $k\Omega_0$. Since the power of the periodic signal exists only at discrete frequencies $k\Omega_0$, the signal has a line spectrum. The space between two adjacent spectral lines is equal to the fundamental frequency Ω_0. For the rectangular pulse train with a fixed period T_0, the effect of decreasing τ (narrowing the width of the rectangular pulse) is to spread the signal power over the entire frequency range. On the other hand, when τ is fixed but the period T_0 increases, the spacing between adjacent spectral lines decreases.

Example 5.2

Consider the pure sine wave expressed as

$$x(t) = \sin(2\pi f_0 t).$$

Using Euler's formula and (5.1), we obtain

$$\sin(2\pi f_0 t) = \frac{1}{2\mathrm{j}}\left(\mathrm{e}^{\mathrm{j}2\pi f_0 t} - \mathrm{e}^{-\mathrm{j}2\pi f_0 t}\right) = \sum_{k=-\infty}^{\infty} c_k\,\mathrm{e}^{\mathrm{j}k2\pi f_0 t}.$$

Therefore, the Fourier series coefficients are obtained as

$$c_k = \begin{cases} 1/2\mathrm{j}, & k = 1 \\ -1/2\mathrm{j}, & k = -1 \\ 0, & \text{otherwise.} \end{cases} \tag{5.5}$$

This equation indicates that the power of a pure sine wave is only distributed at the harmonics $k = \pm 1$, a perfect line spectrum.

5.1.2 Fourier Transform

We have shown that a periodic signal has a line spectrum, and the space between two adjacent spectral lines is equal to the fundamental frequency $\Omega_0 = 2\pi/T_0$. As the period T_0 increases, the line space decreases and the number of frequency components increases. If we increase the period without limit (i.e., $T_0 \to \infty$), the line spacing approaches zero with infinite frequency components. Therefore, the discrete line components converge to a continuum of frequency spectrum.

In practical applications, most real-world signals such as speech are not periodic. They can be approximated by periodic signals with infinite period, that is, $T_0 \to \infty$ (or $\Omega_0 \to 0$), and have continuous frequency spectra. Therefore, the number of exponential components in (5.1) approaches infinity, and the summation becomes integration over the range $(-\infty, \infty)$. Thus, (5.1) becomes

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\Omega)\mathrm{e}^{\mathrm{j}\Omega t}\,\mathrm{d}\Omega. \tag{5.6}$$

This is the inverse Fourier transform. Similarly, (5.2) becomes

$$X(\Omega) = \int_{-\infty}^{\infty} x(t)\mathrm{e}^{-\mathrm{j}\Omega t}\,\mathrm{d}t \tag{5.7}$$

or

$$X(f) = \int_{-\infty}^{\infty} x(t)\mathrm{e}^{-\mathrm{j}2\pi f t}\,\mathrm{d}t. \tag{5.8}$$

This is the Fourier transform (FT) of analog signal $x(t)$.

Example 5.3

Calculate the Fourier transform of $x(t) = \mathrm{e}^{-at}u(t)$, where $a > 0$ and $u(t)$ is the unit-step function.

From (5.7), we have

$$\begin{aligned} X(\Omega) &= \int_{-\infty}^{\infty} \mathrm{e}^{-at}u(t)\mathrm{e}^{-\mathrm{j}\Omega t}\,\mathrm{d}t = \int_{0}^{\infty} \mathrm{e}^{-(a+\mathrm{j}\Omega)t}\,\mathrm{d}t \\ &= \frac{1}{a+\mathrm{j}\Omega}. \end{aligned}$$

For the function $x(t)$ defined over a finite interval T_0, that is, $x(t)=0$ for $|t|>T_0/2$, the Fourier series coefficients c_k can be expressed in terms of $X(\Omega)$ using (5.2) and (5.7) as

$$c_k = \frac{1}{T_0} X(k\Omega_0). \tag{5.9}$$

Therefore, the Fourier transform $X(\Omega)$ of a finite interval function at a set of equally spaced points on the Ω axis is specified by the Fourier series coefficients c_k.

5.2 Discrete Fourier Transform

In this section, we introduce the discrete-time Fourier transform for the theoretical analysis of discrete-time signals and systems, and the discrete Fourier transform, which can be computed by digital hardware for practical applications.

5.2.1 Discrete-Time Fourier Transform

The discrete-time Fourier transform (DTFT) of a discrete-time signal $x(nT)$ is defined as

$$X(\omega) = \sum_{n=-\infty}^{\infty} x(nT)\mathrm{e}^{-\mathrm{j}\omega nT}. \tag{5.10}$$

It can be shown that $X(\omega)$ is a periodic function with period 2π. Thus, the frequency range of a discrete-time signal is unique over the range of $(-\pi, \pi)$ or $(0, 2\pi)$.

The DTFT of $x(nT)$ can also be defined using the normalized frequency as

$$X(F) = \sum_{n=-\infty}^{\infty} x(nT)\mathrm{e}^{-\mathrm{j}2\pi F n}, \tag{5.11}$$

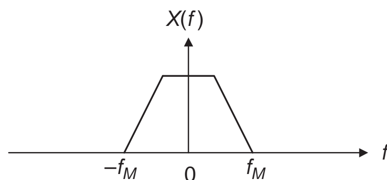
where

$$F = \frac{\omega}{\pi} = \frac{f}{(f_\mathrm{s}/2)}$$

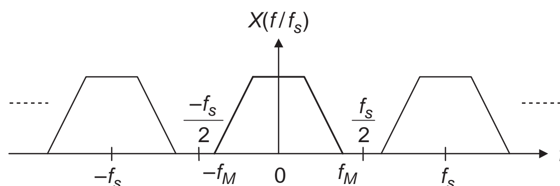
defined in (2.8) is the normalized digital frequency in cycles per sample. Compare this equation to (5.8): the periodic sampling imposes the relationship between the independent variables t and n as $t=nT=n/f_\mathrm{s}$. It can be shown that

$$X(F) = \frac{1}{T}\sum_{k=-\infty}^{\infty} X(f - kf_\mathrm{s}). \tag{5.12}$$

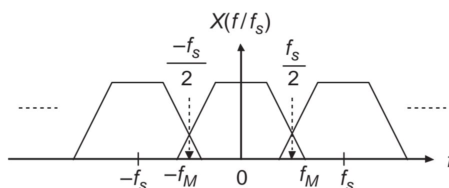
This equation states that $X(F)$ is the sum of an infinite number of $X(f)$, which is the Fourier transform of analog signal $x(t)$, scaled by $1/T$, and then frequency shifted to kf_s. It also states that $X(F)$ is a periodic function with period $T = 1/f_\mathrm{s}$.



(a) Spectrum of bandlimited analog signal.



(b) Spectrum of discrete-time signal when the sampling theorem $f_M \leq f_s/2$ is satisfied.



(c) Spectrum of discrete-time signal that shows aliasing when the sampling theorem is violated.

Figure 5.1 Spectrum replication of discrete-time signal caused by sampling

Example 5.4

Assume that the continuous-time signal $x(t)$ is a bandlimited signal, that is, $|X(f)| = 0$ for $|f| \geq f_M$, where f_M is the bandwidth of signal $x(t)$. The spectrum is zero for $|f| \geq f_M$ as shown in Figure 5.1(a).

As illustrated in (5.12), sampling extends the original spectrum $X(f)$ repeatedly on both sides of the f axis. When the sampling rate f_s is greater than or equal to $2f_M$, that is, $f_M \leq f_s/2$, the analog spectrum $X(f)$ is preserved (without overlap) in $X(F)$ as shown in Figure 5.1(b). In this case, there is no aliasing because the spectrum of the discrete-time signal is identical (except for the scaling factor $1/T$) to the spectrum of the analog signal within the frequency range $|f| \leq f_s/2$ or $|F| \leq 1$. Therefore, the analog signal $x(t)$ can be recovered from the sampled discrete-time signal $x(nT)$ by passing it through an ideal lowpass filter with bandwidth f_M and gain T. This verifies the sampling theorem, that is, $f_M \leq f_s/2$, introduced in Chapter 1.

However, if the sampling rate $f_s < 2f_M$, the shifted replicas of $X(f)$ will overlap with adjacent ones as shown in Figure 5.1(c). This phenomenon is called aliasing since the frequency components in the overlapped regions are corrupted, thus the analog signal $x(t)$ cannot be reconstructed from the sampled signal $x(nT)$.

The DTFT $X(\omega)$ is a continuous function of frequency ω and the computation requires an infinite-length sequence $x(n)$. These two problems make the DTFT very difficult to compute. We will define the discrete Fourier transform (DFT) in the following section for N samples of $x(n)$ and only compute DFT coefficients at N discrete frequencies. Therefore, the DFT is a numerically computable transform that can be used for practical applications.

5.2.2 Discrete Fourier Transform

The DFT of the finite-duration signal $x(n)$ of length N is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j(2\pi/N)kn}, \quad k = 0, 1, \ldots, N-1, \tag{5.13}$$

where k is the frequency index, $X(k)$ is the kth DFT coefficient, and the summation bounds reflect the assumption that $x(n) = 0$ outside the range $0 \leq n \leq N-1$. The DFT is equivalent to taking N samples of DTFT $X(\omega)$ over the interval $0 \leq \omega < 2\pi$, at N equally spaced discrete frequencies $\omega_k = 2\pi k/N$, $k = 0, 1, \ldots, N-1$. Therefore, the space between two successive $X(k)$ is $2\pi/N$ radians (or f_s/N Hz), which is the frequency resolution of the DFT.

Example 5.5

If the signal $x(n)$ is real valued and N is an even number, we can show that

$$X(0) = \sum_{n=0}^{N-1} x(n)e^{-j0} = \sum_{n=0}^{N-1} x(n)$$

and

$$X(N/2) = \sum_{n=0}^{N-1} e^{-j\pi n}x(n) = \sum_{n=0}^{N-1} (-1)^n x(n).$$

Therefore, the DFT coefficients $X(0)$ and $X(N/2)$ are real values. Note that if N is an odd number, $X(0)$ is still real but $X(N/2)$ is not available.

Example 5.6

Consider the finite-length signal

$$x(n) = a^n, \quad n = 0, 1, \ldots, N-1,$$

where $0 < a < 1$. The DFT of $x(n)$ is computed as

$$X(k) = \sum_{n=0}^{N-1} a^n \, e^{-j(2\pi k/N)n} = \sum_{n=0}^{N-1} \left(a \, e^{-j2\pi k/N}\right)^n$$
$$= \frac{1 - \left(a \, e^{-j2\pi k/N}\right)^N}{1 - a \, e^{-j2\pi k/N}} = \frac{1 - a^N}{1 - a \, e^{-j2\pi k/N}}, \quad k = 0, 1, \ldots, N-1.$$

The DFT defined in (5.13) can also be rewritten as

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \quad k = 0, 1, \ldots, N-1, \tag{5.14}$$

where

$$W_N^{kn} = e^{-j(2\pi/N)kn} = \cos\left(\frac{2\pi kn}{N}\right) - j\sin\left(\frac{2\pi kn}{N}\right), \quad 0 \le k, n \le N-1. \tag{5.15}$$

The parameters W_N^{kn} are called the twiddle factors of the DFT. It can be shown that $W_N^N = e^{-j2\pi} = 1 = W_N^0$, $W_N^{N/2} = e^{-j\pi} = -1$, and $W_N^k, k = 0, 1, \ldots, N-1$, are the N roots of unity in a clockwise direction on the unit circle. The twiddle factors have the symmetry property

$$W_N^{k+N/2} = -W_N^k, \quad 0 \le k \le N/2 - 1, \tag{5.16}$$

and the periodicity property

$$W_N^{k+N} = W_N^k. \tag{5.17}$$

The inverse discrete Fourier transform (IDFT) is used to transform the frequency-domain coefficients $X(k)$ back to the time-domain signal $x(n)$. The IDFT is defined as

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j(2\pi/N)kn} = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-kn}, \quad n = 0, 1, \ldots, N-1. \tag{5.18}$$

This is identical to the DFT with the exception of the normalizing factor $1/N$ and the opposite sign of the exponent of the twiddle factors.

The DFT coefficients are equally spaced on the unit circle in the z-plane at frequency intervals of f_s/N (or $2\pi/N$). Therefore, the frequency resolution of the DFT is $\Delta = f_s/N$. The frequency sample $X(k)$ represents the discrete frequency

$$f_k = k\frac{f_s}{N}, \quad \text{for } k = 0, 1, \ldots, N-1. \tag{5.19}$$

Because the DFT coefficient $X(k)$ is a complex variable, it can be expressed in polar form as

$$X(k) = |X(k)|\,e^{j\phi(k)}, \tag{5.20}$$

where the magnitude spectrum is defined as

$$|X(k)| = \sqrt{\{\mathrm{Re}\,[X(k)]\}^2 + \{\mathrm{Im}\,[X(k)]\}^2} \tag{5.21}$$

and the phase spectrum is defined as

$$\phi(k) = \begin{cases} \tan^{-1}\left\{\dfrac{\mathrm{Im}\,[X(k)]}{\mathrm{Re}\,[X(k)]}\right\}, & \text{if} \quad \mathrm{Re}\,[X(k)] \geq 0 \\ \pi + \tan^{-1}\left\{\dfrac{\mathrm{Im}\,[X(k)]}{\mathrm{Re}\,[X(k)]}\right\}, & \text{if} \quad \mathrm{Re}\,[X(k)] < 0. \end{cases} \tag{5.22}$$

5.2.3 Important Properties

This section introduces several important properties of the DFT that are useful for analyzing digital signals and systems.

A. **Linearity:** If $\{x(n)\}$ and $\{y(n)\}$ are digital signals of the same length, then

$$\begin{aligned} \mathrm{DFT}[ax(n) + by(n)] &= a\mathrm{DFT}[x(n)] + b\mathrm{DFT}[y(n)] \\ &= aX(k) + bY(k), \end{aligned} \tag{5.23}$$

where a and b are arbitrary constants. Linearity allows us to analyze complicated composite signals and systems by evaluating their individual frequency components. The overall frequency response is the summation of individual results evaluated at every frequency component.

B. **Complex Conjugate:** If the sequence $\{x(n), 0 \leq n \leq N-1\}$ is real valued, then

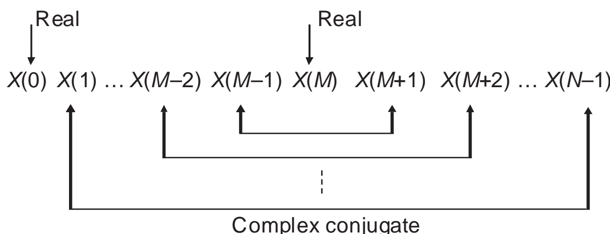
$$X(-k) = X^*(k), \quad 1 \leq k \leq N-1, \tag{5.24}$$

where $X^*(k)$ is the complex conjugate of $X(k)$. Defining $M = N/2$ if N is an even number, or $M = (N-1)/2$ if N is an odd number, we have

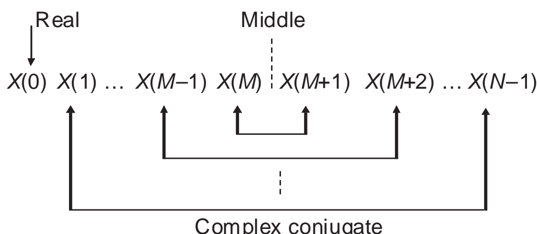
$$X(M+k) = X^*(M-k), \quad \text{for } 1 \leq k \leq M \quad \text{if } N \text{ is even} \tag{5.25a}$$

or

$$X(M+k) = X^*(M-k+1), \quad \text{for } 1 \leq k \leq M \quad \text{if } N \text{ is odd.} \tag{5.25b}$$



(a) N is an even number, $M = N/2$.



(b) N is an odd number, $M = (N–1)/2$.

Figure 5.2 Complex-conjugate property for N is (a) an even number and (b) an odd number

As illustrated in Figure 5.2, this property indicates that only the first $(M+1)$ DFT coefficients from $k=0$ to M are independent and need to be computed. For complex signals, however, all N complex DFT coefficients carry useful information.
From the symmetry property, we obtain

$$|X(k)| = |X(N-k)|, \quad k = 1, 2, \ldots, M \tag{5.26}$$

and

$$\phi(k) = -\phi(N-k), \quad k = 1, 2, \ldots, M. \tag{5.27}$$

These equations show that the magnitude spectrum is an even function, and the phase spectrum is an odd function.

C. **DFT and the z-transform:** The DFT coefficients can be obtained by evaluating the z-transform of the length N sequence $x(n)$ on the unit circle at N equally spaced frequencies $\omega_k = 2\pi k/N$, $k=0, 1, \ldots, N-1$. That is,

$$X(k) = X(z)\big|_{z=e^{j(2\pi/N)k}}, \quad k = 0, 1, \ldots, N-1. \tag{5.28}$$

D. **Circular Convolution:** If $x(n)$ and $h(n)$ are real-valued N-periodic sequences, $y(n)$ is the circular convolution of $x(n)$ and $h(n)$, which is defined as

$$y(n) = h(n) \otimes x(n) = \sum_{m=0}^{N-1} h(m)x\big((n-m)_{\text{mod } N}\big), \quad n = 0, 1, \ldots, N-1, \tag{5.29}$$

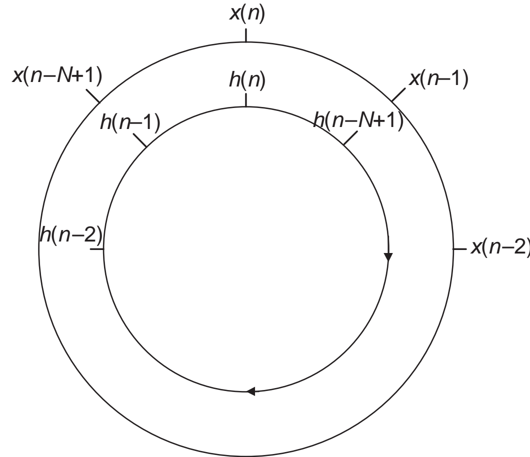


Figure 5.3 Circular convolution of two sequences using the concentric circle approach

where $\otimes$ denotes circular convolution and $(n-m)_{\mathrm{mod}\ N}$ is the non-negative modular N operation. Circular convolution in the time domain is equivalent to multiplication in the frequency domain expressed as

$$Y(k) = X(k)H(k), \quad k = 0, 1, \ldots, N-1. \tag{5.30}$$

Note that the shorter sequence must be padded with zeros in order to have the same length for computing circular convolution.

Figure 5.3 illustrates the cyclic property of circular convolution using two concentric circles. To perform circular convolution, N samples of $x(n)$ are equally spaced around the outer circle in a clockwise direction, and N samples of $h(n)$ are displayed on the inner circle in a counterclockwise direction starting at the same point. Corresponding samples on the two circles are multiplied, and the products are summed to obtain the output value. The successive value of the circular convolution is obtained by rotating the inner circle one sample in the clockwise direction, and repeats the operation of computing the sum of corresponding products. This process is repeated until the first sample of the inner circle lines up with the first sample of the exterior circle again.

Example 5.7

Given two four-point sequences $x(n) = \{1, 2, 3, 4\}$ and $h(n) = \{1, 0, 1, 1\}$, and using the circular convolution method illustrated in Figure 5.3, we obtain

$$\begin{aligned}
n &= 0, y(0) = 1 \cdot 1 + 1 \cdot 2 + 1 \cdot 3 + 0 \cdot 4 = 6 \\
n &= 1, y(1) = 0 \cdot 1 + 1 \cdot 2 + 1 \cdot 3 + 1 \cdot 4 = 9 \\
n &= 2, y(2) = 1 \cdot 1 + 0 \cdot 2 + 1 \cdot 3 + 1 \cdot 4 = 8 \\
n &= 3, y(3) = 1 \cdot 1 + 1 \cdot 2 + 0 \cdot 3 + 1 \cdot 4 = 7.
\end{aligned}$$

Therefore, we have

$$y(n) = x(n) \otimes h(n) = \{6, 9, 8, 7\}.$$

Note that the linear convolution of sequences $x(n)$ and $h(n)$ results in

$$y(n) = x(n) * h(n) = \{1, 2, 4, 7, 5, 7, 4\}.$$

This example can be verified by MATLAB® script `example5_7.m` [10].

To use the DFT to perform linear convolution, we have to eliminate the circular effect by using the zero-padding method. These two sequences must be zero padded to the length of $L+M-1$ or greater, since the linear convolution of two sequences of lengths L and M results in a sequence of length $L+M-1$. That is, append the sequence of length L with at least $M-1$ zeros, and pad the sequence of length M with at least $L-1$ zeros.

Example 5.8

Consider the same sequences $h(n)$ and $x(n)$ as given in Example 5.7. If those four-point sequences are zero padded to eight points as $x(n) = \{1, 2, 3, 4, 0, 0, 0, 0\}$ and $h(n) = \{1, 0, 1, 1, 0, 0, 0, 0\}$, the resulting circular convolution is

$$\begin{aligned}
&n = 0, y(0) = 1\cdot 1 + 0\cdot 2 + 0\cdot 3 + 0\cdot 4 + 0\cdot 0 + 1\cdot 0 + 1\cdot 0 + 0\cdot 0 = 1\\
&n = 1, y(1) = 0\cdot 1 + 1\cdot 2 + 0\cdot 3 + 0\cdot 4 + 0\cdot 0 + 0\cdot 0 + 1\cdot 0 + 1\cdot 0 = 2\\
&n = 2, y(2) = 1\cdot 1 + 0\cdot 2 + 1\cdot 3 + 0\cdot 4 + 0\cdot 0 + 0\cdot 0 + 1\cdot 0 + 1\cdot 0 = 4\\
&\vdots
\end{aligned}$$

We finally have

$$y(n) = x(n) \otimes h(n) = \{1, 2, 4, 7, 5, 7, 4, 0\}.$$

This result is identical to the linear convolution of the two sequences given in Example 5.7. Thus, linear convolution can be realized by circular convolution with proper zero padding. Zero padding can be implemented using the MATLAB® function `zeros(1, N)`, which generates a row vector of `N` zeros. For example, the four-point sequence $x(n)$ given in Example 5.8 can be zero padded to eight points with the following command:

```
x = [1, 2, 3, 4, zeros(1, 4)];
```

MATLAB® script `example5_8.m` performs linear convolution using the DFT and zero padding of sequences.

5.3 Fast Fourier Transforms

The difficulty in using the DFT for practical applications is its intensive computational requirements. To compute each coefficient $X(k)$ defined in (5.14), we need approximately N complex multiplications and additions. Thus, approximately N^2 complex multiplications and

$(N^2 - N)$ complex additions are required to compute N samples of $X(k)$ for $k = 0, 1, \ldots, N-1$. Since a complex multiplication requires four real multiplications and two real additions, the total number of arithmetic operations required for computing an N-point DFT is proportional to $4N^2$, which becomes a huge number for large N.

The twiddle factor W_N^{kn} defined in (5.15) is a periodic function with limited distinct values since

$$W_N^{kn} = W_N^{(kn)_{\mathrm{mod}\,N}}, \quad \text{for } kn > N \tag{5.31}$$

and $W_N^N = 1$. Therefore, different powers of W_N^{kn} have the same value. In addition, some twiddle factors have either real or imaginary parts equal to one or zero. By reducing these redundancies, a very efficient algorithm called the fast Fourier transform (FFT) can be derived, which requires only $N\log_2 N$ operations instead of N^2 operations. If $N = 1024$, the FFT requires about 10^4 operations instead of 10^6 operations for the DFT.

The generic term FFT covers many different computation-efficient algorithms with different features. Each FFT algorithm has different trade-offs in terms of code complexity, memory usage, and computational requirements. In this section, we introduce two basic classes of FFT algorithms: decimation-in-time and decimation-in-frequency. Also, we introduce only the radix-2 algorithm such that N is a power-of-2 integer, that is, $N = 2^m$.

5.3.1 Decimation-in-Time

For the decimation-in-time algorithms, the sequence $x(n)$, $n = 0, 1, \ldots, N-1$, is first divided into two shorter interwoven sequences: the even-numbered sequence

$$x_1(m) = x(2m), \quad m = 0, 1, \ldots, (N/2) - 1 \tag{5.32}$$

and the odd-numbered sequence

$$x_2(m) = x(2m+1), \quad m = 0, 1, \ldots, (N/2) - 1. \tag{5.33}$$

Applying the DFT defined in (5.14) to these two sequences of length $N/2$, and combining the resulting $N/2$-point $X_1(k)$ and $X_2(k)$, produces the final N-point DFT. This procedure is illustrated in Figure 5.4 for $N = 8$.

The structure shown on the right of Figure 5.4 is called the butterfly network because of its crisscross appearance, which can be generalized as in Figure 5.5. Each butterfly involves just a single complex multiplication by the twiddle factor W_N^k, one addition, and one subtraction.

Since N is a power-of-2 integer, $N/2$ is an even number. Each $N/2$-point DFT can be computed by two smaller $N/4$-point DFTs. The second process is illustrated in Figure 5.6.

By repeating the same process, we will finally obtain a set of two-point DFTs. For example, the $N/4$-point DFT became a two-point DFT in Figure 5.6 for $N = 8$. Since the first stage uses the twiddle factor $W_N^0 = 1$, the two-point butterfly network illustrated in Figure 5.7 requires only one addition and one subtraction.

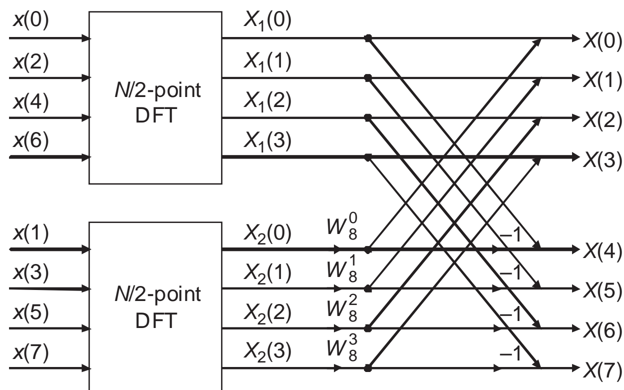


Figure 5.4 Decomposition of N-point DFT into two $N/2$-point DFTs, $N=8$

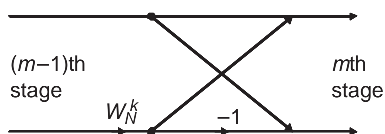


Figure 5.5 Flow graph for butterfly computation

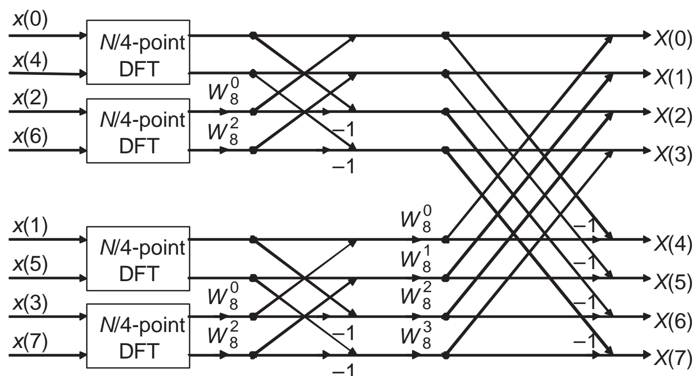


Figure 5.6 Flow graph illustrating second step of N-point DFT, $N=8$

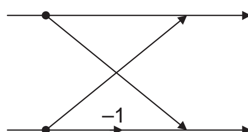


Figure 5.7 Flow graph of two-point DFT

Example 5.9

Consider the two-point FFT algorithm which has two input samples $x(0)$ and $x(1)$. The DFT output samples $X(0)$ and $X(1)$ can be computed as

$$X(k) = \sum_{n=0}^{1} x(n) W_2^{nk}, \quad k = 0, 1.$$

Since $W_2^0 = 1$ and $W_2^1 = \mathrm{e}^{-\pi} = -1$, we obtain

$$X(0) = x(0) + x(1) \quad \text{and} \quad X(1) = x(0) - x(1).$$

This computation is identical to the signal-flow graph shown in Figure 5.7.

As shown in Figure 5.6, the input sequence is arranged as if each index is written in binary form and then the order of binary digits is reversed. This bit-reversal process is illustrated in Table 5.1 for the case of $N = 8$. The input sample indices in decimal are first converted to their binary representations, then the binary bit streams are reversed, and the reversed binary numbers are converted back to decimal to give the reordered time indices. Most modern digital signal processors (including the TMS320C55xx) provide the bit-reversal addressing mode to efficiently support this process.

For the FFT algorithm shown in Figure 5.6, the input values are no longer needed after the computation of output values at a particular stage. Thus, the memory locations used for the FFT outputs can be the same locations used for storing the input data. This fact supports the in-place FFT algorithms that use the same memory locations for both the input and output numbers.

5.3.2 Decimation-in-Frequency

The development of the decimation-in-frequency FFT algorithm is similar to the decimation-in-time algorithm presented in the previous section. The first step is to divide the data sequence into two halves, each of $N/2$ samples. Figure 5.8 illustrates the first decomposition of the N-point DFT into two $N/2$-point DFTs.

Table 5.1 Example of bit-reversal process, $N = 8$ (3-bit)

Input sample index		Bit-reversed sample index	
Decimal	*Binary*	*Binary*	*Decimal*
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

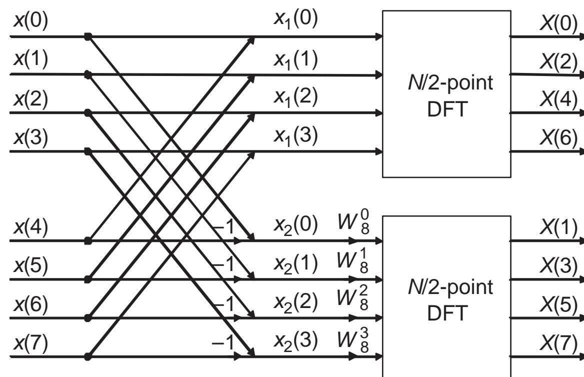


Figure 5.8 Decomposition of an N-point DFT into two $N/2$-point DFTs

The process of decomposition is continued until the last stage consists of two-point DFTs. The decomposition and symmetry relationships are reversed from the decimation-in-time algorithms. The bit reversal occurs at the output instead of the input and the order of the output samples $X(k)$ will be rearranged as in Table 5.1.

The FFT algorithms introduced in this chapter are based on two-input, two-output butterfly computations, and are classified as radix-2 FFT algorithms. It is possible to use other radix values to develop FFT algorithms. However, these algorithms only work well for some specific FFT lengths. In addition, these algorithms are more complicated than the radix-2 FFT algorithms and the programs for real-time implementation are not widely available for digital signal processors.

5.3.3 Inverse Fast Fourier Transform

The FFT algorithms introduced in the previous sections can be modified to efficiently compute the inverse FFT (IFFT). From the DFT defined in (5.14) and the IDFT defined in (5.18), we can use the FFT routine to compute the IFFT using two different methods. By taking the complex conjugates of both sides of (5.18), we get

$$x^*(n) = \frac{1}{N}\sum_{k=0}^{N-1} X^*(k) W_N^{kn}, \quad n = 0, 1, \ldots, N-1. \tag{5.34}$$

Comparing this equation to (5.14), we can first perform the complex conjugate of the DFT coefficients $X(k)$ to obtain $X^*(k)$, compute the DFT of $X^*(k)$ using the FFT algorithm, scaling the results by $1/N$ to obtain $x^*(n)$, and then perform the complex conjugate of $x^*(n)$ to obtain the output sequence $x(n)$. If the signal is real valued, the final conjugation operation is not required.

The second method is to take the complex conjugates of the twiddle factors W_N^{kn} to obtain W_N^{-kn}, compute the DFT of $X(k)$ using the FFT algorithm, and scale the results by $1/N$ to obtain $x(n)$. In Section 5.6.5, we will use the second method to conduct the IFFT experiment.

5.4 Implementation Considerations

Because FFTs are widely used for practical DSP applications, many FFT routines are available in C and assembly programs for several digital signal processors; however, it is important to understand the implementation issues in order to use FFT properly.

5.4.1 Computational Issues

The FFT routines assume that input signals are complex valued, therefore the memory locations required are $2N$ for N-point FFTs. To use the available complex FFT program for real-valued signals, we have to set the imaginary parts to zero. The complex multiplication has the form

$$(a + \mathrm{j}b)(c + \mathrm{j}d) = (ac + bd) + \mathrm{j}(bc + ad),$$

where a, b, c, and d are real numbers so one complex multiplication requires four real multiplications and two real additions. The number of multiplications and the memory storage requirements can be reduced if the signal has special properties. For example, if the signal $x(n)$ is real valued, only the first $N/2 + 1$ samples from $X(0)$ to $X(N/2)$ are needed, based on the complex-conjugate property discussed in Section 5.2.3.

For most FFT programs developed for general-purpose computers, the computation of twiddle factors W_N^{kn} defined in (5.15) is embedded in the program. However, the twiddle factors only need to be computed once during the program initialization stage for a given N. In the implementation of FFT algorithms on digital signal processors, it is preferable to tabulate the values of twiddle factors in memory and use the table lookup method for computation of the FFT.

The complexity of FFT algorithms is usually measured by the required arithmetic operations (multiplications and additions). In practical real-time implementations using digital signal processors, the architecture, instruction set, data structures, and memory organization of the processors are all critical factors. For example, modern digital signal processors usually provide bit-reversal addressing and a high degree of instruction parallelism to implement FFT algorithms.

5.4.2 Finite-Precision Effects

From the signal-flow graph of the FFT algorithm shown in Figure 5.6, $X(k)$ will be computed by a series of butterfly computations with a single complex multiplication per butterfly network. Note that some butterfly networks with coefficients ± 1 (such as the two-point FFT in the first stage) do not require multiplication. Figure 5.6 also shows that computation of the N-point FFT requires $M = \log_2 N$ stages. There are $N/2$ butterflies in the first stage, $N/4$ in the second stage, and so on. Thus, the total number of butterflies required is

$$\frac{N}{2} + \frac{N}{4} + \cdots + 2 + 1 = N - 1. \tag{5.35}$$

The quantization errors introduced at the mth stage are multiplied by the twiddle factors at each subsequent stage. Since the magnitude of each twiddle factor is always unity, the variances of the quantization errors do not change while propagating to the output.

The definition of the DFT in (5.14) shows that we can scale the input signal to satisfy the condition

$$|x(n)| < \frac{1}{N} \tag{5.36}$$

to prevent the overflow at the output because $|e^{-j(2\pi/N)kn}| = 1$. For example, in a 1024-point FFT, the input data must be shifted right by 10 bits ($1024 = 2^{10}$). If the original data is 16 bits, the effective wordlength of the input data is reduced to only 6 bits after scaling. This worst-case scaling substantially reduces the precision and resolution of the FFT results.

Instead of scaling the input samples by $1/N$ at the beginning of the FFT, we can scale the signals at each stage. Figure 5.5 shows that we can avoid overflow within the FFT by scaling the input at each stage by 0.5 because the outputs of each butterfly involve the addition of two numbers. This scaling process provides better performance than the scaling of the input signal by $1/N$ at the beginning.

The best precision can be achieved using the conditional scaling method, which examines the signals at each FFT stage to determine whether to scale the input signals at that stage. If all of the results in a particular stage have magnitude less than one, no scaling is necessary at that stage. Otherwise, the input signals to that stage are scaled by 0.5. This conditional scaling technique achieves much better accuracy, but at the cost of increasing software complexity and decreasing computational saving of FFT algorithms.

5.4.3 MATLAB® Implementations

As introduced in Section 2.2.6, MATLAB® provides the function `fft` with the syntax [11,12]

```
y = fft(x);
```

to compute the DFT of $x(n)$ stored in the vector `x`. If the length of `x` is a power-of-2 integer, the `fft` function employs the efficient radix-2 FFT algorithm. Otherwise, it will use a slower mixed-radix FFT algorithm or even a direct DFT for computation.

An alternative way of using the `fft` function efficiently is to use the syntax

```
y = fft(x, N);
```

to specify the N-point FFT, where N is a power-of-2 integer. If the length of `x` is less than `N`, the vector `x` will be padded with trailing zeros to length `N`. If the length of `x` is greater than `N`, the `fft` function will perform the FFT of the first `N` samples only.

The execution time of the `fft` function depends on the input data type and the sequence length. If the input data is real valued, it computes the real power-of-2 FFT algorithm that is faster than the complex FFT of the same length. The execution speed is fastest if the sequence length is exactly a power-of-2 integer. For example, if the length of `x` is 511, the function `y = fft(x, 512)` can compute faster than `fft(x)` which performs a 511-point DFT. It is important to note that the vectors in MATLAB® are indexed from 1 to N instead of from 0 to $N - 1$ as given in the DFT and IDFT definitions.

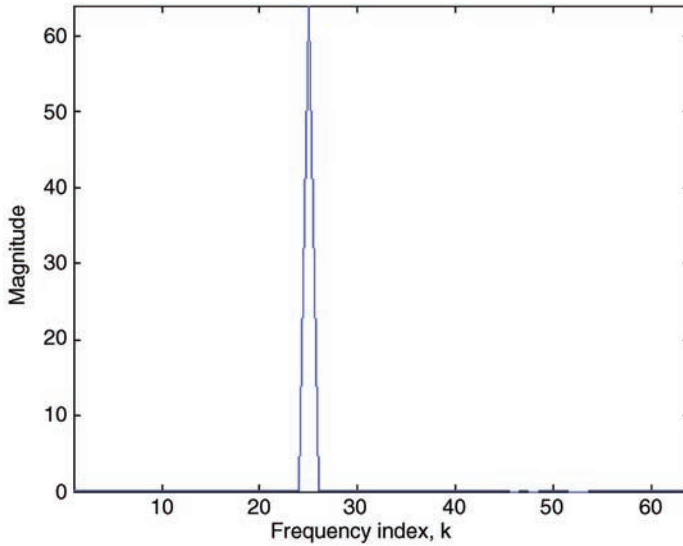


Figure 5.9 Spectrum of 50 Hz sine wave

Example 5.10

Consider a sine wave of frequency $f = 50$ Hz expressed as

$$x(n) = \sin(2\pi f n / f_s), \quad n = 0, 1, \ldots, 127,$$

where the sampling rate f_s is 256 Hz. We analyze this sine wave using the 128-point FFT given in the MATLAB® script (`example5_10.m`), and display the magnitude spectrum in Figure 5.9. It shows that the spectrum peak occurred at the frequency index $k = 25$. According to (5.19), the frequency resolution is 2 Hz so this line spectrum corresponds to 50 Hz.

The MATLAB® function `ifft` implements the IFFT algorithm as

```
y = ifft(x);
```

or

```
y = ifft(x,N);
```

The characteristics and usage of `ifft` are the same as those for `fft`.

5.4.4 Fixed-Point Implementation Using MATLAB®

MATLAB® provides the function `qfft` for quantizing an FFT object to support fixed-point implementation [13]. For example, the command

```
F = qfft
```

constructs the quantized FFT object `F` from default values. We can change the default settings by

```
F = qfft('Property1',Value1, 'Property2',Value2, ...)
```

to create the quantized FFT object with specific property/value pairs.

Example 5.11

We can change the default 16-point FFT to a 128-point FFT using the following command:

```
F = qfft('length',128)
```

We then obtain the following quantized FFT object in the command window:

```
F =
                Radix = 2
               Length = 128
    CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16 15])
          InputFormat = quantizer('fixed', 'floor', 'saturate', [16 15])
         OutputFormat = quantizer('fixed', 'floor', 'saturate', [16 15])
  MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16 15])
        ProductFormat = quantizer('fixed', 'floor', 'saturate', [32 30])
            SumFormat = quantizer('fixed', 'floor', 'saturate', [32 30])
     NumberOfSections = 7
          ScaleValues = [1]
```

This shows that the quantized FFT is a 128-point radix-2 FFT for the fixed-point data and arithmetic. The coefficients, input, output, and multiplicands are represented using the Q15 format `[16 15]`, while the product and sum use the Q30 format `[32 30]`. There are seven stages for $N = 128$, and no scaling is applied to the input at each stage by the default setting `ScaleValues = [1]`. We can set the scaling factor to 0.5 at the input of each stage as follows:

```
F.ScaleValues = [0.5 0.5 0.5 0.5 0.5 0.5 0.5];
```

or set different values at specific stages using different scaling factors.

Example 5.12

Similar to Example 5.10, we use the quantized FFT to analyze the spectrum of a sine wave. In `example5_12a.m`, we first generate the same sine wave as in Example 5.10, and then use the following functions to compute the fixed-point FFT with the Q15 format:

```
FXk = qfft('length',128); % Create quantized FFT object
qXk = fft(FXk, xn);       % Compute Q15 FFT of xn vector
```

When we run the MATLAB® script, it displays the warning “1135 overflows in quantized FFT” in the MATLAB® Command Window. Also, the displayed magnitude spectrum is totally different than in Figure 5.9. This result shows that without using proper scaling factors, the fixed-point FFT result is wrong.

We can modify the code by setting the scaling factor of 0.5 at each stage as follows (see `example5_12b.m`):

```
FXk = qfft('length',128); % Create quantized FFT object
FXk.ScaleValues = [0.5 0.5 0.5 0.5 0.5 0.5 0.5]; % Set scaling factors
qXk = fft(FXk, xn);         % Compute Q15 FFT of xn vector
```

When we run the modified MATLAB® script, there is no warning or error. Comparing the displayed magnitude spectrum plot to that in Figure 5.10, we verify that we can perform the fixed-point FFT properly using 16-bit processors with adequate scaling factors at each stage.

5.5 Practical Applications

This section introduces two important FFT applications: spectral analysis and fast convolution.

5.5.1 Spectral Analysis

The inherent properties of the DFT directly affect its performance in spectral analysis. The spectrum estimated from a finite number of samples is correct only if the signal is periodic and

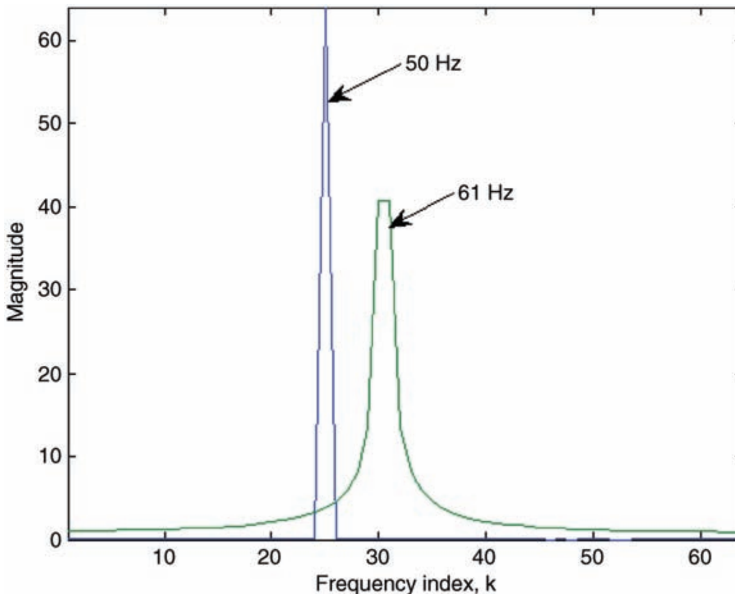


Figure 5.10 Spectra of two sine waves at 50 and 61 Hz

the sample set exactly covers one or multiple periods of signal. In practice, the signal given for analysis may not satisfy this condition, and we may have to break up a long sequence into smaller segments and analyze each segment individually using the DFT.

As discussed in Section 5.2, the frequency resolution of an N-point DFT is f_s/N. The DFT coefficients $X(k)$ represent frequency components that are equally spaced at frequencies f_k as defined in (5.19). The DFT cannot properly represent a frequency component that falls in between two adjacent samples in the spectrum, thus its energy will spread to neighboring frequency bins and distort their spectral amplitude.

Example 5.13

In Example 5.10, the frequency resolution (f_s/N) is 2 Hz using the 128-point FFT and the sampling rate is 256 Hz. The line component at 50 Hz can be represented by $X(k)$ at $k = 25$ as shown in Figure 5.9. In this example, we also compute the magnitude spectrum of the second sine wave at frequency 61 Hz. We display both spectra on the same plot using the MATLAB® script `example5_13.m`. Figure 5.10 shows both spectral components at 50 and 61 Hz. The sine wave at 61 Hz (located between $k = 30$ and $k = 31$) does not show a line because its energy spreads into adjacent frequency bins.

A possible solution to this spectral leakage problem is to have finer resolution f_s/N by using a larger FFT size N. If the number of data samples is not sufficiently large, the sequence may be expanded to length N by padding zeros to the end of the data. This process is equivalent to interpolation of the spectral curve between adjacent frequency components.

Other problems relating to FFT-based spectral analysis include aliasing, finite data length, spectral leakage, and spectral smearing. These issues will be discussed in the following section.

5.5.2 Spectral Leakage and Resolution

The data set that represents a signal of finite length N can be obtained by multiplying the signal with the rectangular window expressed as

$$x_N(n) = w(n)x(n) = \begin{cases} x(n), & 0 \le n \le N-1 \\ 0, & \text{otherwise,} \end{cases} \tag{5.37}$$

where the rectangular window function $w(n)$ of length N is defined in Section 3.2, and the length of $x(n)$ may be much longer than N. As the length of the window increases, the windowed signal $x_N(n)$ is closer to the original signal $x(n)$, and thus $X(k)$ becomes a better approximation of the DTFT $X(\omega)$.

The time-domain multiplication given in (5.37) is equivalent to convolution in the frequency domain. Thus, the DFT of $x_N(n)$ can be expressed as

$$X_N(k) = W(k) * X(k), \tag{5.38}$$

where $W(k)$ is the DFT of the window function $w(n)$, and $X(k)$ is the true DFT of the signal $x(n)$. Equation (5.38) clearly shows that the computed spectrum $X_N(k)$ is the true spectrum $X(k)$

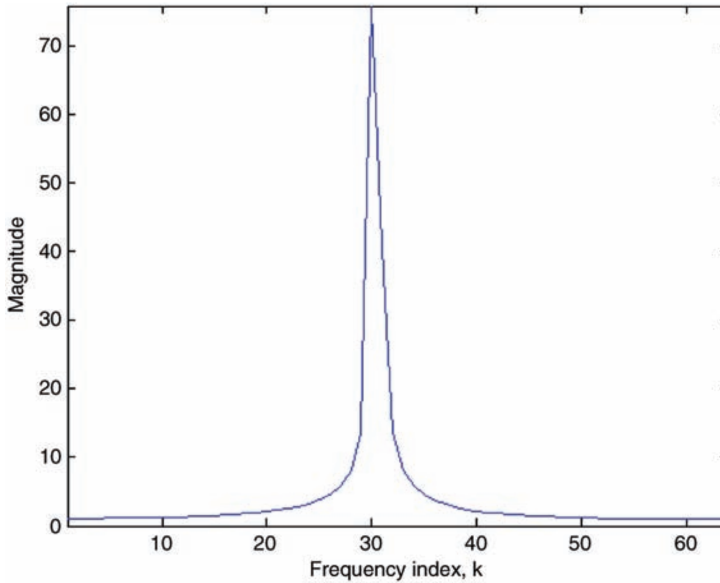


Figure 5.11 Spectra of mixing sinusoids at 60 and 61 Hz

convoluted with the window spectrum $W(k)$. Therefore, the spectrum of the finite-length signal $X_N(k)$ is corrupted by the rectangular window's spectrum, $W(k)$.

As discussed in Section 3.2.2, the magnitude response of the rectangular window consists of a mainlobe and several smaller sidelobes, see Figure 3.11. The frequency components that lie under the sidelobes represent the sharp transition of $w(n)$ at the endpoints. These sidelobes introduce spurious peaks into the computed spectrum, or cancel true peaks in the original spectrum. This phenomenon is known as spectral leakage. To avoid spectral leakage, it is necessary to use different windows with smaller sidelobes as introduced in Section 3.2.3 to reduce the sidelobe effects. Equation (5.38) indicates that the optimum window's spectrum is $W(k) = \delta(k)$, which suggests that suboptimal windows have a very narrow mainlobe and very small sidelobes.

Example 5.14

Given the sinusoidal signal $x(n) = \cos(\omega_0 n)$, the spectrum of the infinite-length sampled signal is

$$X(\omega) = \pi\delta(\omega \pm \omega_0), \qquad -\pi \leq \omega \leq \pi, \tag{5.39}$$

which consists of two line components at frequencies $\pm\omega_0$. However, the spectrum of the windowed sinusoid can be obtained as

$$X_N(\omega) = \frac{1}{2}[W(\omega - \omega_0) + W(\omega + \omega_0)], \tag{5.40}$$

where $W(\omega)$ is the spectrum of the window function.

Equations (5.39) and (5.40) show that the windowing process has the effect of smearing the original sharp spectral lines $\delta(\omega \pm \omega_0)$ at frequencies $\pm\omega_0$ and replacing them with $W(\omega \pm \omega_0)$. Thus, the power has been spread into the entire frequency range by the windowing operation. This undesired effect is called spectral smearing. Thus, windowing not only distorted the spectrum of the signal due to leakage effects, but also reduced spectral resolution.

Example 5.15

Consider a signal consisting of two sinusoidal components, which is expressed as $x(n) = \cos(\omega_1 n) + \cos(\omega_2 n)$. The spectrum of the windowed signal is

$$X_N(\omega) = \frac{1}{2}[W(\omega - \omega_1) + W(\omega + \omega_1) + W(\omega - \omega_2) + W(\omega + \omega_2)], \tag{5.41}$$

which shows the four sharp spectral lines replaced with their smeared versions. If the frequency separation, $\Delta\omega = |\omega_1 - \omega_2|$, of the two sinusoids is

$$\Delta\omega \leq \frac{2\pi}{N} \quad \text{or} \quad \Delta f \leq \frac{f_s}{N}, \tag{5.42}$$

the mainlobe of the two window functions $W(\omega - \omega_1)$ and $W(\omega - \omega_2)$ will smear. Thus, these two spectral lines in $X_N(\omega)$ are not distinguishable. MATLAB® script `example5_15.m` computes a 128-point FFT of the signal with a sampling rate of 256 Hz. From (5.42), the frequency separation of 1 Hz is less than the frequency resolution of 2 Hz, thus these two spectral lines are smeared as shown in Figure 5.11. This example shows that the two spectral lines of sine waves at frequencies of 60 Hz and 61 Hz are mixed, and thus cannot be distinguished by the magnitude spectrum plot.

To guarantee that the spectral lines of two sinusoids appear as two distinct ones, their frequency separation must larger than the frequency resolution to satisfy the condition

$$\Delta\omega > \frac{2\pi}{N} \quad \text{or} \quad \Delta f > \frac{f_s}{N} \tag{5.43}$$

Thus, the minimum DFT length to achieve the desired frequency resolution is

$$N > \frac{2\pi}{\Delta\omega} \quad \text{or} \quad N > \frac{f_s}{\Delta f} \tag{5.44}$$

In summary, the mainlobe width of the window determines the frequency resolution of the windowed signal spectrum. The sidelobes determine the amount of undesired frequency leakage. Therefore, the optimum window used for spectral analysis must have a narrow mainlobe and small sidelobes. The amount of leakage can be substantially reduced by using the non-rectangular window functions introduced in Section 3.2.3, but at the cost of decreased spectral resolution. For a given window length N, windows such as the rectangular, Hanning,

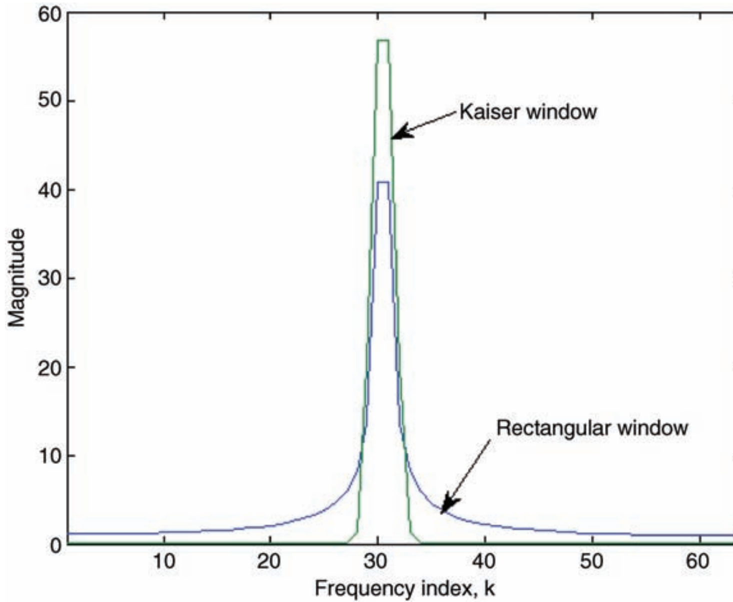


Figure 5.12 Spectra obtained using rectangular and Kaiser windows

and Hamming have a relatively narrow mainlobe compared to the Blackman and Kaiser windows. Unfortunately, the first three windows have relatively large sidelobes, thus having more leakage. There is a trade-off between frequency resolution and spectral leakage in choosing windows for a given spectral analysis application.

Example 5.16

Consider the 61 Hz sine wave in Example 5.13. We can apply the Kaiser window with $N = 128$ and $\beta = 8.96$ to the signal using the following commands:

```
beta = 8.96;                % Define beta value
wn = (kaiser(N,beta))'; % Kaiser window
x1n = xn.*wn;               % Apply Kaiser window to sine wave
```

The magnitude spectra of windowed sine waves from the rectangular and Kaiser windows are shown in Figure 5.12 using the MATLAB® script `example5_16.m`. This example shows that the Kaiser window can effectively reduce the spectral leakage. Note that the gain for using the Kaiser window has been scaled up by 2.4431 in order to compensate for the energy loss as compared to the rectangular window. The time- and frequency-domain plots of the Kaiser window can be evaluated using Wintool.

For a given window, increasing the length of the window reduces the width of the mainlobe, which leads to better frequency resolution. However, as the length increases, the ripples become narrower and move closer to $\pm\,\omega_0$, but the largest magnitude of the ripples remains

constant. Also, if the signal changes its frequency contents over time, the window cannot be too long in order to obtain meaningful estimation of the spectrum.

5.5.3 Power Spectral Density

Consider a signal $x(n)$ of length N with the DFT $X(k)$, the Parseval's theorem can be expressed as

$$E = \sum_{n=0}^{N-1} |x(n)|^2 = \frac{1}{N}\sum_{k=0}^{N-1} |X(k)|^2. \tag{5.45}$$

The term $|X(k)|^2$ is called the power spectrum that measures the power of the signal at frequency f_k. Therefore, squaring the DFT magnitude spectrum $|X(k)|$ produces the power spectrum, which is also called the periodogram.

The power spectral density (PSD) (power density spectrum or simply power spectrum) characterizes stationary random processes. The PSD is very useful in the analysis of random signals since it provides a meaningful measure for the distribution of the average power over the frequency. There are several different techniques for estimating the PSD. Since the periodogram is not a consistent estimate of the true PSD, the averaging method can reduce the statistical variation of the computed spectra.

One way of computing the PSD is to decompose $x(n)$ into M segments, $x_m(n)$, for $m = 1, 2, \ldots, M$, of N samples each. These signal segments are spaced $N/2$ samples apart, that is, there is a 50% overlap between successive segments for reducing discontinuity. In order to reduce spectral leakage, each $x_m(n)$ is multiplied by a non-rectangular window (such as a Hamming) function $w(n)$ of length N. The PSD is a weighted sum of the periodograms of the individual overlapped segment.

MATLAB® provides the function `spectrum` for spectral estimation using the following syntax:

```
h = spectrum.<estimator>
```

This function supports several estimators including `burg` (Burg), `periodogram` (periodogram), `cov` (covariance), `welch` (Welch), and others to estimate the PSD of the signal given in the vector `xn` using the following statements:

```
Hs = spectrum.periodogram;     % Create a periodogram object
psd(Hs,xn,'Fs',fs);            % Plots the two-sided PSD by default
```

where `fs` is the sampling frequency.

Example 5.17

Consider the noisy signal $x(n)$ which consists of two sinusoids (140 Hz and 150 Hz) and noise generated by `example5_17.m` (adapted from the MATLAB® **Help** menu). The PSD can be computed by creating the following periodogram object:

```
Hs = spectrum.periodogram;
```

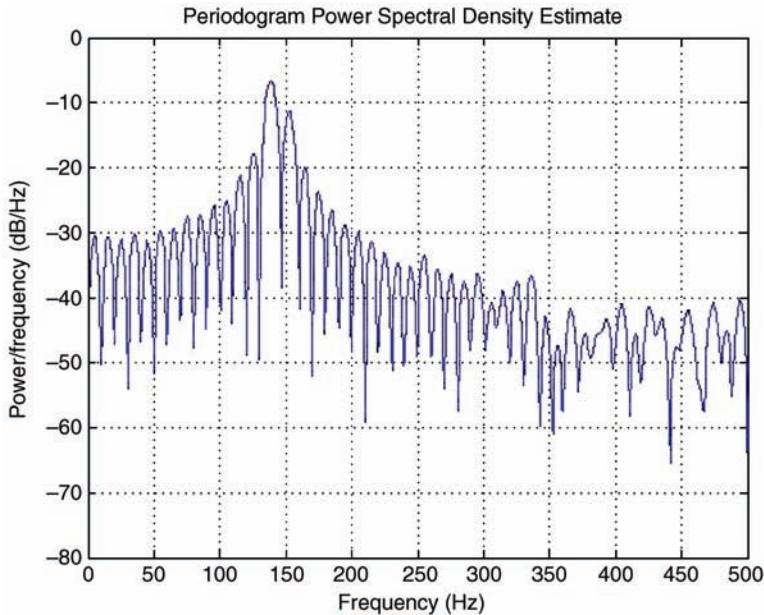


Figure 5.13 PSD of two sine waves embedded in noise

The `psd` function can also display the PSD (Figure 5.13) as follows:

```
psd(Hs,xn,'Fs',fs,'NFFT',1024)
```

For a time-varying signal, it is more meaningful to compute a local spectrum that measures spectral contents over a short-time interval. For this purpose, we can use a sliding window to break up a long signal sequence into several short blocks $x_m(n)$ of N samples, and then perform the FFT to obtain the time-dependent frequency spectrum at each segment as follows:

$$X_m(k) = \sum_{n=0}^{N-1} x_m(n) W_N^{kn}, \quad k = 0, 1, \ldots, N-1. \tag{5.46}$$

This process is repeated for the next block of N samples. This technique is called the short-time Fourier transform, since $X_m(k)$ is just the spectrum of the short segment of $x_m(n)$ that lies inside the sliding window $w(n)$. This method of computing time-dependent Fourier transforms has several applications in speech, sonar, and radar signal processing.

Equation (5.46) shows that $X_m(k)$ is a two-dimensional sequence. The index k represents frequency, and the block index m represents segment (or time). Since the result is a function of both time and frequency, a three-dimensional graphical display is needed. This can be done by plotting $|X_m(k)|$ using gray-scale (or color) images as a function of both k and m. The resulting three-dimensional graphic is called a spectrogram, which uses the x axis to represent time and

the y axis to represent frequency. The gray level (or color) at point (m, k) is proportional to $|X_m(k)|$.

MATLAB® provides the function `spectrogram` (`specgram` in older versions) to compute and display the spectrogram. This MATLAB® function has the form

```
B = spectrogram(x,window,noverlap,nfft,Fs);
```

where `B` is a matrix containing the complex spectrogram values $|X_m(k)|$, `window` is the window length, `noverlap` is the number of samples that each segment of `x` overlaps, `nfft` is the FFT size, and `Fs` is the sampling rate. Note that `noverlap` must be an integer smaller than `window`. More overlapped samples smoothes the spectrum moving from block to block. It is common to pick the overlap to be around 50%. The `spectrogram` function with no output arguments displays the scaled logarithm of the spectrogram in the current graphic window.

Example 5.18

The MATLAB® program `example5_18.m` loads the 16-bit, 8 kHz sampled speech file `timit2.asc`, plays it using the function `soundsc`, and computes and displays the spectrogram as shown in Figure 5.14. The color corresponding to the lower power regions in the figure indicates the periods of silence and the color corresponding to the higher

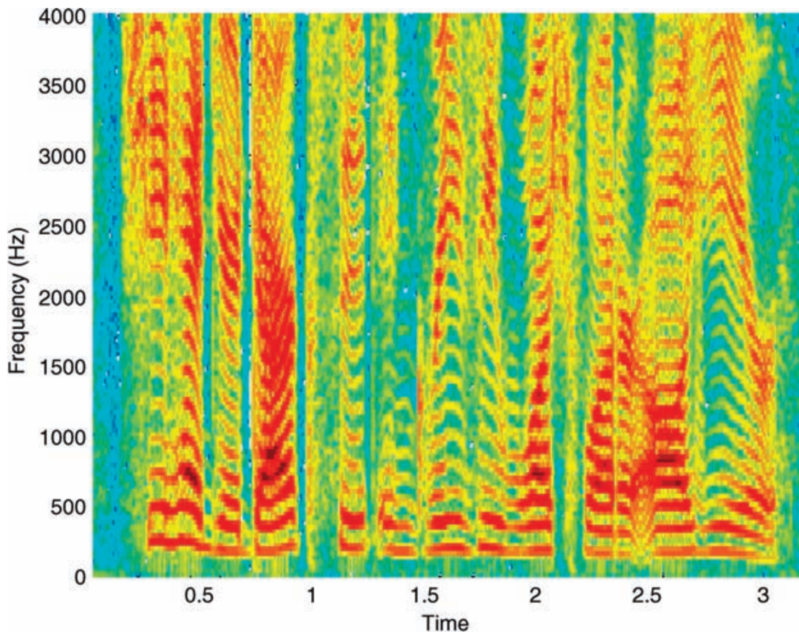


Figure 5.14 Spectrogram of speech signal.

power represents the speech signals. In the MATLAB® script, we use the following function to compute and display the spectrogram:

```
spectrogram(timit2,256,200,256,8000,'yaxis')
```

Note that we use the trailing input string 'yaxis' to display frequency on the y axis and time on the x axis.

5.5.4 Convolution

As discussed in Chapter 3, FIR filtering is linear convolution of the filter impulse response $h(n)$ with the input signal $x(n)$. If the FIR filter has L coefficients, we need L multiplications and $L-1$ additions to compute each output $y(n)$. To obtain N output samples, the number of operations (multiplication and addition) needed is proportional to LN. To take advantage of computationally efficient FFT and IFFT algorithms, we can use the fast convolution algorithm illustrated in Figure 5.15 for FIR filtering. Fast convolution provides a significant reduction in the computational requirements for higher order FIR filters, thus it is often used to implement FIR filtering in applications having a large number of signal samples.

It is important to note that fast convolution produces circular convolution as discussed in Section 5.2.3. In order to produce linear convolution, it is necessary to append zeros to both sequences as shown in Example 5.8. If the data sequence $x(n)$ has finite duration M, the first step is to zero-pad both impulse response and signal sequences to a length equal to the FFT size N, where N ($\geq L+M-1$) is a power-of-2 integer and L is the length of $h(n)$. The FFT is computed for both sequences to obtain $X(k)$ and $H(k)$, the corresponding complex products $Y(k)=X(k)H(k)$ are calculated, and the IFFT of $Y(k)$ is performed to obtain output $y(n)$. The desired linear convolution is contained in the first $(L+M-1)$ samples of the IFFT results. Since the filter impulse response $h(n)$ is known a priori, the FFT of $h(n)$ can be pre-calculated and stored as fixed coefficients $H(k)$.

For many applications, the input sequence is very long as compared to the FIR filter length L. This is especially true in real-time applications, such as in audio signal processing where the FIR filter order is extremely high due to a high sampling rate and input data is very long. In order to use the efficient FFT and IFFT algorithms, the input sequence must be partitioned into segments of N ($N>L$ and N is a size supported by the FFT algorithm) samples, process each segment using the FFT, and finally assemble the output sequence from the outputs of each segment. This procedure is called the block processing operation. The cost of using this efficient block processing is the buffering delay. Complicated algorithms have been devised to

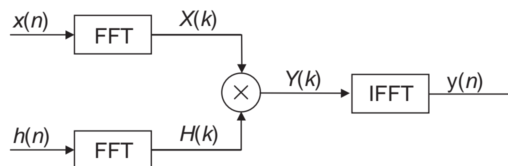


Figure 5.15 Basic idea of fast convolution

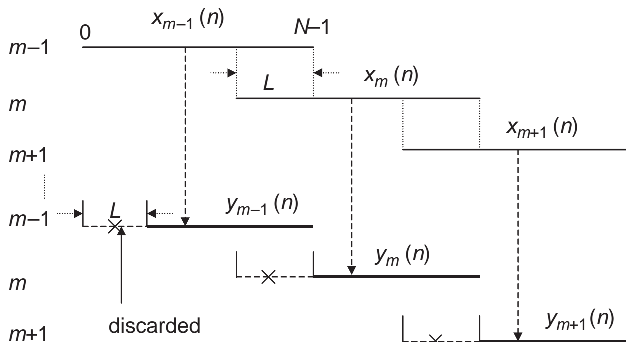


Figure 5.16 Overlap data segments for the overlap–save technique

have advantages of computational efficiency and zero latency as the direct time-domain FIR filtering [14].

There are two effective techniques for the segmentation and recombination of the data: the overlap–save and overlap–add algorithms.

Overlap–Save Technique

The overlap–save technique overlaps L input samples on each segment of length N where $L < N$. The output segments are truncated to be non-overlapping and then concatenated. The following steps describe the process illustrated in Figure 5.16:

1. Apply an N-point FFT to the expanded (zero-padded) impulse response sequence to obtain $H(k)$, $k = 0, 1, \ldots, N-1$. This process can be pre-calculated offline and stored in memory.
2. Select N signal samples $x_m(n)$ (where m is the segment index) from the input sequence $x(n)$ based on the overlap illustrated in Figure 5.16, and then use the N-point FFT to obtain $X_m(k)$.
3. Multiply the stored $H(k)$ (obtained in step 1) by the $X_m(k)$ (obtained in step 2) to get

$$Y_m(k) = H(k)X_m(k), \quad k = 0, 1, \ldots, N-1. \tag{5.47}$$

   Perform the N-point IFFT of $Y_m(k)$ to obtain $y_m(n)$ for $n = 0, 1, \ldots, N-1$.
4. Discard the first L samples from each IFFT output. The resulting segments of $N - L$ samples are concatenated to produce $y(n)$.

Overlap–Add Technique

The overlap–add process divides the input sequence $x(n)$ into non-overlapping segments of length $N - L$. Each segment is zero padded to produce $x_m(n)$ of length N. Steps 2, 3, and 4 of the overlap–save method are followed to obtain an N-point segment $y_m(n)$. Since the convolution is a linear operation, the output sequence $y(n)$ is the summation of all segments.

MATLAB® implements this efficient FIR filtering using the overlap–add technique as

```
y = fftfilt(b, x)
```

The `fftfilt` function filters the input signal in the vector `x` with the FIR filter represented by the coefficient vector `b`. The function chooses the FFT and data block length that automatically guarantees efficient execution time. However, we can specify the FFT length `N` by using

```
y = fftfilt(b, x, N)
```

The overlap–add technique is used for the FFT convolution experiments given in the next section.

Example 5.19

The speech data `timit2.asc` (used in Example 5.18) is corrupted by a tonal noise at frequency 1 kHz. We design a bandstop FIR filter with edge frequencies of 900 and 1100 Hz to filter the noisy speech using the MATLAB® script `example5_19.m`. This program plays the original speech first, and then plays the noisy speech that is corrupted by the 1 kHz tone and shows the spectrogram with the noise component (in red) at 1 kHz. In order to attenuate the tonal noise, a bandstop FIR filter is designed (using the function `fir1`) to filter the noisy speech using the function `fftfilt`. Finally, the filter output is played and its spectrogram is displayed with the 1 kHz tonal noise having been attenuated.

5.6 Experiments and Program Examples

This section implements the DFT and FFT algorithms for DSP applications. Computation of the DFT and FFT involves nested loops, complex multiplications, bit-reversal operations, in-place computations, and the generation of complex twiddle factors.

5.6.1 DFT Using Floating-Point C

For multiplying a complex signal sample $x(n) = x_\mathrm{r}(n) + \mathrm{j}x_\mathrm{i}(n)$ and a complex twiddle factor $W_N^{kn} = \cos(2\pi kn/N) - \mathrm{j}\sin(2\pi kn/N) = W_\mathrm{r} - \mathrm{j}W_\mathrm{i}$ defined in (5.15), the product can be expressed as

$$x(n)W_N^{kn} = x_\mathrm{r}(n)W_\mathrm{r} + x_\mathrm{i}(n)W_\mathrm{i} + \mathrm{j}[x_\mathrm{i}(n)W_\mathrm{r} - x_\mathrm{r}(n)W_\mathrm{i}],$$

where the subscripts r and i denote the real and imaginary parts of a complex variable, respectively. This equation can be rewritten as

$$X = X_\mathrm{r} + \mathrm{j}X_\mathrm{i}$$

where

$$X_\mathrm{r} = x_\mathrm{r}(n)W_\mathrm{r} + x_\mathrm{i}(n)W_\mathrm{i} \quad \text{and} \quad X_\mathrm{i} = x_\mathrm{i}(n)W_\mathrm{r} - x_\mathrm{r}(n)W_\mathrm{i}.$$

The floating-point C program listed in Table 5.2 computes the DFT coefficients $X(k)$. The program uses two arrays, `Xin[2*N]` and `Xout[2*N]`, to hold the complex input and output samples. The twiddle factors are computed at runtime. Since most real-world applications process real-valued signals, it is necessary to construct a complex data set from the given real data. The simplest way is to place the real-valued data into the corresponding

Table 5.2 List of floating-point C function for the DFT

```
#include <math.h>
#define PI 3.1415926536
void floating_point_dft(float Xin[], float Xout[])
{
  short i,n,k,j;
  float angle;
  float Xr[N],Xi[N];
  float W[2];
  for (i=0,k=0;k<N;k++)
  {
    Xr[k]=0;
    Xi[k]=0;
    for(j=0,n=0;n<N;n++)
    {
         angle=(2.0*PI*k*n)/N;    // Calculate twiddle factor angle
         W[0]=cos(angle);         // Compute complex twiddle factor
         W[1]=sin(angle);
         // Multiply by complex data with complex twiddle factor
         Xr[k]=Xr[k]+Xin[j]*W[0]+Xin[j+1]*W[1];
         Xi[k]=Xi[k]+Xin[j+1]*W[0]-Xin[j]*W[1];
         j+=2;                    // Advance data pointer
    }
    // Save the real and imaginary parts of DFT results
    Xout[i++] = Xr[k];
    Xout[i++] = Xi[k];
  }
}
```

real parts of the complex data array and set the imaginary parts to zero before calling the DFT function.

This experiment computes a 128-point DFT and the magnitude spectrum $|X(k)|$ of the signal from the given file `input.dat`. Table 5.3 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment.
3. Examine the magnitude spectrum saved in the array `spectrum[]` using the CCS graph tool. From **Tools** → **Graph** → **Single Time** to open the CCS graph tool, and in the **Graph Properties** dialog window, use the following parameters:

   Acquisition Buffer Size = 64
   Dsp Data Type = 16-bit signed integer
   Starting Address = `spectrum`
   Display Data Size = 64

Table 5.3 File listing for the experiment Exp5.1

Files	Description
float_dft128Test.c	Program for testing floating-point DFT
float_dft128.c	C function for 128-point floating-point DFT
float_mag128.c	C function for computing magnitude spectrum
float_dft128.h	C header file
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
input.dat	Input data file

and keep the default settings for the rest. Click on **OK**. The CCS plot shows three lines at the DFT bin numbers 8, 16, and 32, which correspond to the normalized frequencies at 0.125, 0.25, and 0.5.

4. Profile the code to find the required clock cycles for computing each DFT coefficient $X(k)$ (in the array `Xout[]`) using the floating-point C program.

5.6.2 DFT Using the C55xx Assembly Program

This experiment implements the DFT using TMS320C55xx assembly routines. The sine–cosine generator for the experiments given in Chapter 2 is used to generate the twiddle factors. The assembly function, `sine_cos.asm` (see Section 2.6.2), is a C-callable function that follows the C55xx C-calling convention. This function has two arguments: `angle` and `Wn`. The first argument contains the value of the angle in radians and is passed to the assembly routine using the C55xx temporary register T0. The second argument is a pointer to the array `Wn` that holds the twiddle factors, and is passed using the C55xx auxiliary register AR0.

The calculation of the angle depends on two variables, k and n, as

$$\text{angle} = (2\pi/N)kn.$$

As shown in Figure 2.25, 16-bit fixed-point representation of value π for the sine–cosine generator is 0x7FFF (32 767). Thus, the angle used to generate the twiddle factors can be expressed as

$$\text{angle} = (2 \times 32\,767/N)kn$$

for $n = 0, 1, \ldots, N-1$ and $k = 0, 1, \ldots, N-1$. Table 5.4 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment.
3. Examine the magnitude spectrum saved in the data array `spectrum[]` using the CCS graphic tool; see step 3 in Exp5.1 for details.

Table 5.4 File listing for the experiment Exp5.2

Files	Description
asm_dft128Test.c	Program for testing DFT experiment
dft_128.asm	Assembly function for 128-point DFT
mag_128.asm	Assembly function for computing magnitude spectrum
sine_cos.asm	Assembly function for computing twiddle factors
asm_dft128.h	C header file
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
input.dat	Input data file

4. Compare the resulting $X(k)$ (in the array `Xout[]`) to the floating-point implementation results obtained in Exp5.1.
5. Profile the clock cycles required to compute each DFT coefficient $X(k)$ using C55xx assembly routines in this experiment, and compare the result to the floating-point C implementation obtained in Exp5.1.

5.6.3 FFT Using Floating-Point C

This experiment implements the complex, radix-2, decimation-in-time FFT algorithm using the floating-point C program. The C code for the FFT function is listed in Table 5.5. The first argument is the pointer to the complex input data array. The second argument is the base-2 logarithm exponential value of the FFT. For the radix-2, 128-point FFT, this value equals 7 since $2^7 = 128$. The third argument is the pointer to the array of the twiddle factors. The next argument is the flag indicating if the computation is FFT or IFFT. The last argument is the flag indicating if scaling by 0.5 is used at each stage as explained in Section 5.4.2. The twiddle factors are computed and stored in the complex array `W[]` during the initialization stage.

This experiment also uses the bit-reversal function listed in Table 5.6, which rearranges the order of signal samples according to the bit-reversal algorithm illustrated in Table 5.1 before passing the samples to the FFT function. The files used for the experiment are listed in Table 5.7.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment.
3. Examine the magnitude spectrum `spectrum[]` for each FFT frame using the CCS graphic tool. Verify that the input signal is a sweeping tone by observing that the tone's frequency (indicated by the line position) is changing with time (frame).
4. Profile the clock cycles required to compute each coefficient $X(k)$ using the FFT and compare the result to the floating-point C implementation obtained in Exp5.1 (direct DFT).

5.6.4 FFT Using Fixed-Point C with Intrinsics

This experiment implements the FFT using fixed-point C with C55xx intrinsics (listed in Table 4.7) based on the floating-point C program used in the previous experiment. The

Table 5.5 List of floating-point C FFT function

```
void fft(complex *X, unsigned short EXP, complex *W, unsigned short iFlag,
unsigned short sFlag)
{
    complex temp;              /* Temporary storage of complex variable */
    complex U;                 /* Twiddle factor W^k */
    unsigned short i,j;
    unsigned short id;         /* Index for lower point in butterfly */
    unsigned short L;          /* FFT stage */
    unsigned short LE;         /* Number of points in DFT at stage L
                                  and offset to next DFT in stage */
    unsigned short LE1;        /* Number of butterflies in one DFT at
                                  stage L. Also is offset to lower point
                                  in butterfly at stage L */
    float scale;
    unsigned short N=1<<EXP;/* Number of points for FFT */

    if (sFlag == 1)            /* NOSCALE_FLAG=1 */
        scale = 1.0;           /* Without scaling */
    else                       /* SCALE_FLAG = 0 */
        scale = 0.5;           /* Scaling of 0.5 at each stage */
    if (iFlag == 1)            /* FFT_FLAG=0, IFFT_FLAG=1 */
        scale = 1.0;           /* Without scaling for IFFT */
    for (L=1; L<=EXP; L++)     /* FFT butterfly */
    {
        LE=1<<L;           /* LE=2^L=points of sub DFT */
        LE1=LE>>1;         /* Number of butterflies in sub-DFT */
        U.re = 1.0;
        U.im = 0.0;

        for (j=0; j<LE1;j++)
        {
            for(i=j; i<N; i+=LE) /* Butterfly computations */
            {
                id=i+LE1;
                temp.re = (X[id].re*U.re - X[id].im*U.im)*scale;
                temp.im = (X[id].im*U.re + X[id].re*U.im)*scale;

                X[id].re = X[i].re*scale - temp.re;
                X[id].im = X[i].im*scale - temp.im;

                X[i].re = X[i].re*scale + temp.re;
                X[i].im = X[i].im*scale + temp.im;
            }
            /* Recursive compute W^k as U*W^(k-1) */
            temp.re = U.re*W[L-1].re - U.im*W[L-1].im;
            U.im = U.re*W[L-1].im + U.im*W[L-1].re;
            U.re = temp.re;
        }
    }
}
```

Table 5.6 List of bit-reversal C function

```
void bit_rev(complex *X, short EXP)
{
    unsigned short i,j,k;
    unsigned short N=1<<EXP; /* Number of points for FFT */
    unsigned short N2=N>>1;
    complex temp; /* Temporary storage of the complex variable */

    for (j=0,i=1;i<N-1;i++)
    {
        k=N2;
        while(k<=j)
        {
             j-=k;
             k>>=1;
        }
        j+=k;
        if (i<j)
        {
            temp = X[j];
            X[j] = X[i];
            X[i] = temp;
        }
    }
}
```

program uses some intrinsics including `_lsmpy`, `_smas`, `_smac`, `_sadd`, and `_ssub` for arithmetic operations.

The implementation of the FFT using mixed fixed-point C and intrinsics to improve its runtime efficiency is listed in Table 5.8. Table 5.9 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment.

Table 5.7 File listing for the experiment Exp5.3

Files	Description
`float_fftTest.c`	Program for testing floating-point FFT
`fft_float.c`	C function for floating-point FFT
`fbit_rev.c`	C function performs bit reversal
`float_fft.h`	C header file for floating-point FFT
`fcomplex.h`	C header file defines floating-point complex data type
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`input_f.dat`	Input data file

Table 5.8 Fixed-point C implementation of the FFT using intrinsics

```
for (L=1; L<=EXP; L++)        /* FFT butterfly */
{
    LE=1<<L;                  /* LE=2^L=points of sub DFT */
    LE1=LE>>1;                /* Number of butterflies in DFT */
    U.re = 0x7fff;
    U.im = 0;

    for (j=0; j<LE1;j++)
    {
        for(i=j; i<N; i+=LE) /* Butterfly computations */
        {
            id=i+LE1;
            ltemp.re = _lsmpy(X[id].re, U.re);
            temp.re = _smas(ltemp.re, X[id].im, U.im);
            temp.re = (short)(ltemp.re>>SFT16);
            temp.re >>= scale;
            ltemp.im = _lsmpy(X[id].im, U.re);
            temp.im = _smac(ltemp.im, X[id].re, U.im);
            temp.im = (short)(ltemp.im>>SFT16);
            temp.im >>= scale;
            X[id].re = _ssub(X[i].re>>scale, temp.re);
            X[id].im = _ssub(X[i].im>>scale, temp.im);
            X[i].re = _sadd(X[i].re>>scale, temp.re);
            X[i].im = _sadd(X[i].im>>scale, temp.im);
        }
        /* Recursive compute W^k as W*W^(k-1) */
        ltemp.re = _lsmpy(U.re, W[L-1].re);
        ltemp.re = _smas(ltemp.re, U.im, W[L-1].im);
        ltemp.im = _lsmpy(U.re, W[L-1].im);
        ltemp.im = _smac(ltemp.im, U.im, W[L-1].re);
        U.re = ltemp.re>>SFT16;
        U.im = ltemp.im>>SFT16;
    }
}
```

Table 5.9 File listing for the experiment Exp5.4

Files	Description
`intrinsic_fftTest.c`	Program for testing FFT using intrinsics
`intrinsic_fft.c`	C function for FFT using intrinsics
`ibit_rev.c`	C function for performing fixed-point bit reversal
`intrinsic_fft.h`	C header file for fixed-point FFT
`icomplex.h`	C header file defines fixed-point complex data type
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`input_i.dat`	Input data file

Table 5.10 File listing for the experiment Exp5.5

Files	Description
FFT_iFFT_Test.c	Program for testing FFT and IFFT
intrinsic_fft.c	C function for FFT using intrinsics
ibit_rev.c	C function for performing fixed-point bit reversal
w_table.c	C function for generating twiddle factors
intrinsic_fft.h	C header file for fixed-point FFT
icomplex.h	C header file defines fixed-point complex data type
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
input.dat	Input data file

3. Examine the magnitude spectrum `spectrum[]` for each FFT frame using the CCS graphic tool. Verify that the input signal is a sweeping tone; see step 3 of Exp5.3 for details.
4. Profile the FFT function and compare the required clock cycles for the fixed-point C using intrinsics to the floating-point C implementation of the FFT given in Exp5.3.

5.6.5 Experiment with the FFT and IFFT

This experiment uses fixed-point C with intrinsics to implement the radix-2 FFT and IFFT algorithms. The twiddle factors are computed using the C function `w_table()` during the initialization process. In order to use the same FFT routine for the IFFT calculation, two simple changes are made. First, the complex conjugate of twiddle factors is done by changing the sign of the imaginary part of the complex numbers. Second, the normalization of $1/N$ for the IFFT is handled inside the FFT routine by using the scale factor 0.5 at each stage.

This experiment computes the 128-point FFT to obtain 128 DFT coefficients and performs the IFFT of these coefficients. We also compare the difference (or error) between the original input and the resulting FFT/IFFT output. The files used for the experiment are listed in Table 5.10.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment.
3. Examine the difference between the input signal and the resulting output signal from the FFT/IFFT operations.
4. Modify the program to compute 64- and 256-point FFT/IFFT experiments. Compare the difference (due to numerical errors) between the input signal and the resulting output signal.

5.6.6 FFT Using the C55xx Hardware Accelerator

C55xx processors have a built-in hardware accelerator to compute the FFT efficiently. The FFT module is located in the ROM section of the C55xx. It supports 8-, 16-, 32-, 64-, 128-, 256-, 512-, and 1024-point, radix-2 FFT. This hardware module uses a coprocessor to compute the complex FFT with parallel data memory accesses using B, C, and D data buses, see Appendix C for the C55xx architecture. The hardware FFT module includes a lookup table

consisting of 1024 complex twiddle factors to support up to 1024-point FFT. The bit-reversal function is also included in the ROM.

The C55xx hardware FFT function uses the following syntax [15]:

```
hwafft_NUMpts(long *in, long *out, short fftFlag, short scaleFlag)
```

The `NUM` within the function name `hwafft_NUMpts()` specifies the FFT size. For example, the 1024-point FFT function is called as `hwafft_1024pts()`. The arguments are defined as follows:

`in` – the complex input data array, which is ordered as interleaved real and imaginary parts;
`out` – the complex output data array and interleaved as the `in` array;
`fftFlag` – the FFT flag: `fftFlag=0` for the FFT and `fftFlag=1` for the IFFT;
`scaleFlag` – the scale flag: `scaleFlag=0` scales the input by 0.5 at each stage.

The bit-reversal function has the following syntax:

```
hwafft_br(long *in, long *out, short len)
```

where the argument `len` is the array length and the rest of the arguments are defined in the FFT function. The bit-reversal function is the assembly program using the C55xx bit-reversal addressing mode (see Appendix C) to effectively swap data in the bit-reversed order for the FFT algorithm.

There are several restrictions on using the hardware FFT module. First, the complex input and output data arrays must be arranged in an interleaved real and imaginary order. Second, these complex data arrays must be aligned on a boundary of two 16-bit words (32-bit) since the module uses double and dual memory accesses. Furthermore, in order to properly use the bit-reversal function `hwafft_br()`, the output array for the bit-reversal function needs to be placed at the data memory address with at least $\log_2(4N)$ binary zeros. For example, for the 1024-point FFT, `N` = 1024 and the address must end with 12 zero bits, such as 0x1000 or 0x2000. The address 0x2800 does not satisfy the requirement for the 1024-point FFT since it has only 11 zero bits. To ensure this condition is satisfied, `DATA_ALIGN` pragma can be used in the source code to force memory alignment. Data alignment can also be done by fixing the address location in the linker command file, see the experiment program file and linker command file for details. Finally, the hardware FFT module is included in the program linking process. To use the hardware FFT function, the bit-reversal function and FFT function are appended to the end of the linker command file with the function name associated with the function address. An example of adding the hardware bit-reversal function and 1024-point FFT function is given as follows:

```
_hwafft_br = 0x00ff6cd6;
_hwafft_1024pts = 0x00ff7a56;
```

This experiment demonstrates how to use the C55xx hardware FFT module to process the signal from a data file. The C5505 eZdsp reads the input data and computes the 128-point FFT to convert the time-domain signal $x(n)$ to DFT coefficients $X(k)$, then uses the

Table 5.11 File listing for the experiment Exp5.6

Files	Description
hwfftTest_.c	Program for testing hardware FFT module
tistdtypes.h	Standard type define header file
input.dat	Input data file
c5505.cmd	Linker command file modified for using hardware FFT

IFFT to convert it back to the time-domain signal. Table 5.11 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment.
3. Examine the difference between the input signal and the obtained FFT/IFFT result using the C5505 FFT hardware module.
4. Profile the hardware FFT module and compare the result to the FFT program using fixed-point C with intrinsics in the previous experiments.
5. Modify the program to use the 256-point hardware FFT and evaluate the hardware FFT performance in terms of processing speed (profile required clock cycles) and memory usage.

5.6.7 Real-Time FFT Using the C55xx Hardware Accelerator

This section uses the C55xx hardware FFT module for real-time experiments. The C5505 eZdsp digitizes the input signal from an audio source such as an audio player, computes the 512-point FFT, then performs the IFFT to convert the DFT coefficients $X(k)$ back to the time-domain signal $x(n)$ for real-time playback. Table 5.12 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Connect a headphone and an audio source to the eZdsp. Load and run the experiment.

Table 5.12 File listing for the experiment Exp5.7

Files	Description
realtime_hwfftTest.c	Program for testing real-time FFT
realtime_hwfft.c	C function manages real-time audio data process
vector.asm	Vector table for real-time experiment
icomplex.h	C header file defines fixed-point complex data type
tistdtypes.h	Standard type define header file
dma.h	Header file for DMA functions
dmaBuff.h	Header file for DMA data buffer
i2s.h	i2s header file for i2s functions
Ipva200.inc	C5505 processor include file
myC55xxUtil.lib	BIOS audio library
c5505.cmd	Linker command file modified for using hardware FFT

3. Evaluate the experiment results by comparing (listening to) the signal between the original audio source and the processed audio from FFT/IFFT operations.
4. Modify this hardware FFT experiment to zero-out several low-frequency DFT coefficients (FFT bins) after computation of the FFT. Listen to the IFFT output and describe the artifacts resulting from zeroing some FFT bins. Repeat the experiment by zeroing several DFT coefficients from high-frequency FFT bins.
5. Modify the experiment to use the 128-point hardware FFT, and repeat step 4 to evaluate the real-time FFT/IFFT experimental results.

5.6.8 Fast Convolution Using the Overlap–Add Technique

This experiment implements fast convolution using mixed fixed-point C and assembly programs. The lowpass FIR filter used for the experiment has 511 coefficients. Fast convolution uses the overlap–add method with the 1024-point FFT and IFFT. The filtering is done using complex multiplication in the frequency domain instead of the time-domain FIR filtering presented in Chapter 3. The array `OVRLAP` of length (`FFT_PTS-DATA_LEN`) is used to hold the overlap–add data samples. The buffer names and their lengths used for the experiment are defined as follows:

```
#define FLT_LEN 511                  // Filter length
#define FFT_PTS 1024                 // FFT size
#define DATA_LEN(FFT_PTS-FLT_LEN+1)  // Data segment length

complex X[FFT_PTS];                  // Signal buffer
complex H[FFT_PTS];                  // Frequency-domain filter
short OVRLAP[FFT_PTS-DATA_LEN];      // Frequency-domain overlap buffer
```

The input signal for the experiment contains three tones at 800, 1500, and 3300 Hz with a sampling rate of 8000 Hz. The lowpass filter has a cutoff frequency of 1000 Hz to attenuate the frequency components above 1000 Hz. The experiment performs the following tasks:

1. Pad (`FFT_PTS-FLT_LEN`) zeros to the impulse response (coefficients) of the FIR filter during the initialization stage, perform the 1024-point FFT, and store the frequency-domain filter coefficients in the complex buffer `H[FFT_PTS]`.
2. Segment the input signal of length `DATA_LEN` and pad `FLT_LEN` – 1 zeros at the end, and store the signal in the complex data buffer `X[FFT_PTS]`.
3. Process each segment of signal samples using the 1024-point FFT to obtain the frequency-domain data samples and place them back in the complex array `X[FFT_PTS]`.
4. Perform the filtering process by multiplying the complex coefficients in the array `H` by the corresponding data in the complex array `X` to obtain the frequency-domain output, and save back in the complex array `X[FFT_PTS]`.
5. Apply the 1024-point IFFT to compute the filtered time-domain signal. The result is placed back in the complex array `X[FFT_PTS]`.
6. Add the first (`FFT_PTS-FLT_LEN`) samples with the previous segment to form the output. The time-domain filtered signal samples are located in the real parts of the complex array `X`, which are written to an output file.
7. Update the overlap buffer with the next (`FLT_LEN` – `FFT_PTS`) samples for the next segment of the signal. For applications with real-valued signals like this experiment, the

Table 5.13 File listing for the experiment Exp5.8

Files	Description
fast_convoTest.c	Program for testing fast convolution
intrinsic_fft.c	FFT and IFFT functions
bit_rev.asm	Assembly function performs bit reversal
freqflt.asm	Assembly function performs fast convolution
olap_add.c	C function controls overlap–add process
w_table.c	C function generates twiddle factors
fast_convolution.h	C header file for fast convolution
icomplex.h	C header file defines fixed-point complex data type
tistdtypes.h	Standard type define header file
fdacoefs1k_8k_511.h	Lowpass filter coefficients
input.pcm	Input data file
c5505.cmd	Linker command file

imaginary parts of the overlap buffer should be updated with zeros; only the real parts of the overlap buffer will be updated with the new data results from fast convolution.

The files used for the experiment are listed in Table 5.13.
Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment using the input signal from the data file `input.pcm`.
3. Examine the experimental result by computing the magnitude spectrum of the output signal to show that the 800 Hz tone will be passed. Measure the attenuation (in dB) of the 1500 and 3300 Hz tones.
4. Modify the experiment to replace the lowpass filter with a bandpass FIR filter that will pass only the 1500 Hz tone. Redo the experiment to show the bandpass filter result and measure the attenuations achieved by the bandpass filter at frequencies 800 and 3300 Hz.
5. Design a narrow notch (bandstop) FIR filter that will remove the 1500 Hz tone with minimum impact on other frequencies. Show the magnitude spectrum of the filtered output signal. Perform the experiment using the input signal from the file `input.pcm`.
6. Generate white noise as the input signal, repeat step 5, and examine the magnitude spectrum of the filtered output signal.

5.6.9 Real-Time Fast Convolution

This experiment implements the fast convolution technique for real-time lowpass filtering using the C55xx hardware FFT module. The input signals used for this experiment can come from any audio source such as an audio player. A lowpass FIR filter with a 2000 Hz cutoff frequency and 48 000 Hz sampling frequency is used for the experiment. The process of fast convolution is the same as in the previous experiment. This experiment uses the 1024-point hardware FFT module for different filter lengths. Fast convolution applies to both left and right stereo channels. The high-frequency components of the audio signals will be attenuated by the lowpass filter. By listening to the input and output audio signals, the filtered audio signal's "muffle" effect can be heard. The files used for the experiment are listed in Table 5.14.

Table 5.14 File listing for the experiment Exp5.9

Files	Description
realtime_hwfftConvTest.c	Program for testing real-time fast convolution
freqflt.asm	Assembly function performs fast convolution
olap_add.c	C function controls overlap–add process
realtime_hwfftConv.c	C function manages real-time data process
vector.asm	Vector table for real-time experiment
icomplex.h	C header file defines fixed-point complex data type
tistdtypes.h	Standard type define header file
fdacoefs2k_48k_511.h	Lowpass filter coefficients
dma.h	Header file for DMA functions
dmaBuff.h	Header file for DMA data buffer
i2s.h	i2s header file for i2s functions
Ipva200.inc	C5505 processor include file
myC55xxUtil.lib	BIOS audio library
c5505.cmd	Linker command file

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Connect a headphone and an audio source to the C5505 eZdsp. Load and run the experiment.
3. Examine the experiment by listening to the audio output and compare it to the input audio.
4. Use different filter lengths (63, 127, 255, and 511) and repeat step 3 to verify the real-time experimental results.
5. Design a comb filter that has four equally spaced passbands (refer to Example 3.2 in Chapter 3). Use MATLAB® to create a signal file in WAV format which contains tones at the fundamental frequency $\pi/8$ and harmonics $\pi/4$, $\pi/2$, and $3\pi/4$. Use an audio player to play this WAV file as the input signal. Conduct a real-time experiment using the eZdsp to verify that the comb filter will attenuate these harmonic-related tones.

Exercises

5.1. Compute the Fourier series coefficients of the cosine function $x(t) = \cos(2\pi f_0 t)$.

5.2. Compute the four-point DFT of the sequence $\{1, 1, 1, 1\}$.

5.3. Compute $X(0)$ and $X(4)$ of the eight-point DFT of the sequence $\{1, 1, 1, 1, 2, 3, 4, 5\}$.

5.4. Prove the symmetry and periodicity properties of the twiddle factors defined as

(a) $W_N^{k+N/2} = -W_N^k$.
(b) $W_N^{k+N} = W_N^k$.

5.5. Consider the following two sequences:

$$x_1(n) = 1 \quad \text{and} \quad x_2(n) = n, \quad \text{for } 0 \leq n \leq 3.$$

(a) Compute the linear convolution of these two sequences.
(b) Compute the circular convolution of these two sequences.
(c) Pad zeros for these two sequences such that the circular convolution results are the same as the linear convolution results obtained in (a).
(d) Verify the above results using MATLAB®.

5.6. Construct the signal-flow diagram of the FFT for $N=16$ using the decimation-in-time method.

5.7. Construct the signal-flow diagram of the FFT for $N=8$ using the decimation-in-frequency method.

5.8. Similar to Table 5.1, show the bit-reversal process for the 16-point FFT.

5.9. Consider 1 second of a digitized signal with a sampling rate at 20 kHz. It is desired to obtain a spectrum with a frequency resolution of 100 Hz or smaller. Is this possible? If impossible, what FFT size N is needed?

5.10. A 1 kHz sinusoid is sampled at 8 kHz. The 128-point FFT is performed to compute $X(k)$. What is the frequency resolution? What frequency indices k will show peaks in $|X(k)|$? Can we observe the line spectrum? If spectral leakage occurs, how can we solve the problem?

5.11. A touch-tone phone with a dual-tone multi-frequency (DTMF) transmitter encodes each keypress as the sum of two sinusoids, with two frequencies taken from each of the following groups (see Chapter 7 for details):
Vertical group: 697, 770, 852, 941 Hz
Horizontal group: 1209, 1336, 1477, 1633 Hz.

What is the smallest DFT size N that can distinguish these two sinusoids from the computed magnitude spectrum? The sampling rate used in telecommunications is 8 kHz.

5.12. Similar to Example 5.7, and given $x(n)=\{1, 2, 3, 4, 5, 6, 7, 8\}$ and $h(n)=\{1, 0, 1, 0, 1, 0, 1, 0\}$, write MATLAB® scripts to implement the following tasks:
(a) Linear convolution.
(b) Circular convolution using the FFT and IFFT.
(c) Linear convolution using the FFT and IFFT.
(d) Fast convolution with adequate zero padding of two sequences $x(n)$ and $h(n)$.

5.13. Similar to Example 5.10, compute the fixed-point FFT using the Q15 format without scaling the input signal. Has overflow occurred? Compare the results of using the scaling factor 1/128 at the input of stage 1 and using the scaling factor 0.5 at the input of each stage.

5.14. Similar to Example 5.13 but using white noise with zero mean and variance 0.5 instead of a sine wave, compute the Q15 FFT without scaling. How many overflows are there in the quantized FFT? Try different scaling vectors, `F.ScaleValues`, and discuss the differences.

5.15. Redo Problem 5.14 using a sine wave as input. Compare the differences to the white-noise input obtained in Problem 5.14.

5.16. Similar to Example 5.15, use different techniques to distinguish two sine waves at 60 and 61 Hz.

5.17. Similar to Example 5.18, use the MATLAB® function `spectrogram` with different parameters (such as window size, percentage of overlap, FFT size) to analyze the given speech signal in file `timit2.asc`.

5.18. Write a C or MATLAB® program to compute the fast convolution of a long sequence (such as `timit2.asc`) with an FIR filter (design by MATLAB) with a short impulse response, employing the overlap–save method introduced in Section 5.5.4. Compare the results to the MATLAB® function `fftfilt` that uses the overlap–add method.

5.19. Most DSP applications process real-valued input samples using the complex FFT function by placing zeros in the imaginary parts of the complex buffer. This approach is simple but inefficient in terms of execution speed and memory requirements. For real-valued input, we can split the even and odd samples into two sequences, and compute both even and odd sequences in parallel. This approach will reduce the execution time by approximately 50%. Given a real-valued input signal $x(n)$ of $2N$ samples, we can define $c(n) = a(n) + \mathrm{j}b(n)$, where $a(n) = x(n)$ is the even-indexed sequence and $b(n) = x(n+1)$ is the odd-indexed sequences. We can represent these sequences as $a(n) = [c(n) + c^*(n)]/2$ and $b(n) = -\mathrm{j}[c(n) - c^*(n)]/2$; then they can be written in terms of DFT coefficients as $A_k(k) = [C(k) + C^*(N-k)]/2$ and $B_k(k) = -\mathrm{j}[C(k) - C^*(N-k)]/2$. Finally, the real-valued input FFT results can be obtained by $X(k) = A_k(k) + W_{2N}^k B_k(k)$ and $X(k+N) = A_k(k) - W_{2N}^k B_k(k)$, where $k = 0, 1, \ldots, N-1$. Modify the complex radix-2 FFT function to compute $2N$ real-valued signal samples.

5.20. Write the 128-point, decimation-in-frequency FFT function in fixed-point C and verify the results using the experiment given in Section 5.6.4.

5.21. Redo Problem 5.20 using fixed-point C with intrinsics, and compare the efficiency to the fixed-point C in Problem 5.20.

References

1. DeFatta, D.J., Lucas, J.G., and Hodgkiss, W.S. (1988) *Digital Signal Processing: A System Design Approach*, John Wiley & Sons, Inc., New York.
2. Ahmed, N. and Natarajan, T. (1983) *Discrete-Time Signals and Systems*, Prentice Hall, Englewood Cliffs, NJ.
3. Kuo, S.M. and Gan, W.S. (2005) *Digital Signal Processors*, Prentice Hall, Upper Saddle River, NJ.
4. Jackson, L.B. (1989) *Digital Filters and Signal Processing*, 2nd edn, Kluwer Academic, Boston.
5. Oppenheim, A.V. and Schafer, R.W. (1989) *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
6. Orfanidis, S.J. (1996) *Introduction to Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
7. Proakis, J.G. and Manolakis, D.G. (1996) *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd edn, Prentice Hall, Englewood Cliffs, NJ.
8. Bateman, A. and Yates, W. (1989) *Digital Signal Processing Design*, Computer Science Press, New York.
9. Stearns, S.D. and Hush, D.R. (1990) *Digital Signal Analysis*, 2nd edn, Prentice Hall, Englewood Cliffs, NJ.
10. The MathWorks, Inc. (2000) Using MATLAB, Version 6.
11. The MathWorks, Inc. (2004) Signal Processing Toolbox User's Guide, Version 6.
12. The MathWorks, Inc. (2004) Filter Design Toolbox User's Guide, Version 3.
13. The MathWorks, Inc. (2004) Fixed-Point Toolbox User's Guide, Version 1.
14. Tian, W. (2002) Method for efficient and zero latency filtering in a long-impulse-response system. European patent WO0217486A1, February.
15. Texas Instruments, Inc. (2010) FFT Implementation on the TMSVC5505, TMSC5505, TMSC5515 DSPs, SPRABB6A, June.

6

Adaptive Filtering

The principles and techniques for the analysis, design, and implementation of linear time-invariant FIR and IIR filters are introduced in Chapters 3 and 4, respectively. These filters with fixed coefficients calculated by filter design algorithms may not work properly for some practical applications where the characteristics of the signals and environment are unknown and/or changing. In this chapter, we introduce adaptive filters with coefficients updated automatically and continually by adaptive algorithms to track time-varying signals and systems [1–10].

6.1 Introduction to Random Processes

Real-world signals such as speech, music, and noise are time varying and random in nature. Some basic concepts of random variables are introduced in Section 2.3. This section briefly reviews some important properties of the random processes for understanding the fundamental concepts of adaptive algorithms.

The autocorrelation function of the random process $x(n)$ is defined as

$$r_{xx}(n,k) = E[x(n)x(k)]. \tag{6.1}$$

This function specifies the statistical relation of the signal at different time indices n and k, and determines the degree of dependence for the random variable at different time lags $(n-k)$ units apart.

Correlation is a useful tool for detecting signals that are corrupted by random noise, measuring the time delay between two signals, estimating the impulse response of an unknown system, and many other things. Correlation is often used in radar, sonar, digital communications, and other scientific and engineering areas. For example, in radar and sonar applications, the received signal reflected from the target is the delayed version of the transmitted signal. By measuring the round-trip delay using an appropriate correlation function, the radar and sonar can determine the distance of the target object. In Chapter 8, the correlation function will be used to estimate the time delay between speech and its echoes.

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

A random process is stationary if its statistics do not change with time. The most useful and relaxed form of stationarity is the wide-sense stationary (WSS) process that satisfies the following two conditions:

1. The mean of the process is independent of time. That is,

$$E[x(n)] = m_x, \tag{6.2}$$

   where the mean m_x is a constant.
2. The autocorrelation function depends only on the time difference. That is,

$$r_{xx}(k) = E[x(n+k)x(n)], \tag{6.3}$$

   where k is the time lag.

The autocorrelation function $r_{xx}(k)$ of a WSS process has the following important properties:

1. The autocorrelation function is an even function. That is,

$$r_{xx}(-k) = r_{xx}(k). \tag{6.4}$$

2. The autocorrelation function is bounded by

$$|r_{xx}(k)| \le r_{xx}(0), \tag{6.5}$$

   where $r_{xx}(0) = E[x^2(n)]$ is the mean-square value, or the power of random process $x(n)$. If $x(n)$ is a zero-mean random process, we have

$$r_{xx}(0) = E\left[x^2(n)\right] = \sigma_x^2. \tag{6.6}$$

Example 6.1

Consider the sinusoidal signal $x(n) = A\cos(\omega_0 n)$. Find the mean and the autocorrelation function of $x(n)$.

For the mean,

$$m_x = AE[\cos(\omega_0 n)] = 0.$$

For the autocorrelation function,

$$\begin{aligned} r_{xx}(k) &= E[x(n+k)x(n)] = A^2 E[\cos(\omega_0 n + \omega k)\cos(\omega_0 n)] \\ &= \frac{A^2}{2}E[\cos(2\omega_0 n + \omega_0 k)] + \frac{A^2}{2}\cos(\omega_0 k) = \frac{A^2}{2}\cos(\omega_0 k). \end{aligned}$$

Thus the autocorrelation function of a cosine wave is the cosine function of the same frequency ω_0.

Next, consider the widely used random signal for many applications, which is white noise $v(n)$ with zero mean and variance σ_v^2. Its autocorrelation function

$$r_{vv}(k) = \sigma_v^2 \delta(k) \tag{6.7}$$

is a delta function with amplitude σ_v^2 at lag $k = 0$, while its power spectrum is

$$P_{vv}(\omega) = \sigma_v^2, \quad |\omega| <= \pi, \tag{6.8}$$

which shows that the power of the random signal is uniformly distributed over the entire frequency range.

The cross-correlation function between two WSS processes $x(n)$ and $y(n)$ is defined as

$$r_{xy}(k) = E[x(n+k)y(n)]. \tag{6.9}$$

This cross-correlation function has the property

$$r_{xy}(k) = r_{yx}(-k). \tag{6.10}$$

Therefore, $r_{yx}(k)$ is simply the folded version of $r_{xy}(k)$.

Example 6.2

Consider the second-order FIR filter with the I/O equation

$$y(n) = x(n) + ax(n-1) + bx(n-2).$$

Assume white noise with zero mean and variance σ_x^2 is used as the input signal $x(n)$. Find the mean m_y and the autocorrelation function $r_{yy}(k)$ of the filter output signal $y(n)$.

For the mean,

$$m_y = E[y(n)] = E[x(n)] + aE[x(n-1)] + bE[x(n-2)] = 0.$$

For the autocorrelation function,

$$\begin{aligned}
r_{yy}(k) &= E[y(n+k)y(n)] \\
&= (1 + a^2 + b^2)r_{xx}(k) + (a + ab)r_{xx}(k-1) + (a + ab)r_{xx}(k+1) \\
&\quad + br_{xx}(k-2) + br_{xx}(k+2) \\
&= \begin{cases} (1 + a^2 + b^2)\sigma_x^2, & \text{if } k = 0 \\ (a + ab)\sigma_x^2, & \text{if } k = \pm 1 \\ b\sigma_x^2, & \text{if } k = \pm 2 \\ 0, & \text{otherwise.} \end{cases}
\end{aligned}$$

In many practical applications, we may only have one finite-length sequence $x(n)$. In this case, the sample mean of $x(n)$ is defined as

$$\bar{m}_x = \frac{1}{N}\sum_{n=0}^{N-1} x(n), \tag{6.11}$$

where N is the number of samples available for the short-time analysis. Similarly, the sample autocorrelation function is defined as

$$\bar{r}_{xx}(k) = \frac{1}{N-k} \sum_{n=0}^{N-k-1} x(n+k)x(n), \quad k = 0, 1, \ldots, N-1. \tag{6.12}$$

In theory, we can compute the time lag k up to $N-1$; however, in practice, we can only expect good results for lags of less than 10% of the length of the signal.

As discussed in Chapter 2, MATLAB® function `rand` generates uniformly distributed pseudo-random numbers between 0 and 1 [11]. In addition, `randn` generates normally distributed pseudo-random numbers, and `randi` generates uniformly distributed pseudo-random numbers in integers.

Example 6.3

Similar to Example 6.1, consider a cosine wave corrupted by zero-mean, unit-variance white noise $v(n)$. The corrupted signal can be expressed as

$$x(n) = A\cos(\omega_0 n) + v(n).$$

The mean of this corrupted signal is

$$m_x = AE[\cos(\omega_0 n)] + E[v(n)] = 0.$$

From Example 6.1, the autocorrelation function is

$$r_{xx}(k) = \frac{A^2}{2}\cos(\omega_0 k) + \sigma_v^2 \delta(k).$$

This equation shows that the noise energy is concentrated in zero lag ($k=0$), and the rest of the autocorrelation function for $k>1$ is the pure cosine wave with the same frequency ω_0. Therefore, the autocorrelation function can be used to estimate the frequency of a sinusoid embedded in white noise. In addition, the power spectrum is

$$P_{xx}(\omega) = \frac{A^2}{2}\delta(\omega_0) + \sigma_v^2, \quad |\omega| < \pi.$$

This equation shows that the power spectrum is basically flat with a line located at frequency ω_0. Therefore, either the autocorrelation function or the power spectrum can be used to detect a sinusoidal signal that is corrupted by white noise, see Example 2.21 and Figure 2.18.

MATLAB® provides the function `xcorr` for estimating the cross-correlation function as

```
c = xcorr(x,y)
```

where `x` and `y` are vectors of length N. If `x` and `y` have different lengths, the shortest one will be zero padded to the same length as the longer one. This function returns the length $2N-1$ cross-correlation sequence `c`. This function also can be used to estimate the autocorrelation function as

```
a = xcorr(x)
```

Note that the zeroth lag ($k=0$) of the estimated autocorrelation function is in the middle of the $2N-1$ sequence, that is, at element N.

6.2 Adaptive Filters

The signal characteristics in many practical applications are time varying and/or unknown. For example, consider high-speed data communication via a media channel for transmitting and receiving data. The application may require the use of a filter called a channel equalizer to compensate for channel distortion. Since the communication channels may be unknown and time varying each time the connection is established, the equalizer must be able to track and compensate for the unknown and changing channel characteristics.

An adaptive filter modifies its characteristics to achieve certain objectives by automatically updating its coefficients. Many adaptive filter structures and adaptation algorithms have been developed to suit different application requirements. This chapter only presents the widely used FIR-based adaptive filters with the least mean-square (LMS) algorithm and some simple modified LMS-type algorithms. These adaptive filters are relatively simple to design and implement with a robust performance, thus they are widely used in practical applications. They are well understood with regard to stability, convergence speed, steady-state performance, and finite-precision effects.

6.2.1 Introduction to Adaptive Filtering

An adaptive filter consists of two distinct parts: a digital filter to perform the desired filtering and an adaptive algorithm to automatically update the filter coefficients (or weights). A general form of adaptive filter is illustrated in Figure 6.1, where $d(n)$ is the desired (or primary input) signal, $y(n)$ is the output signal of the digital filter driven by the reference input signal $x(n)$, and $e(n)$ is the error signal, that is, the difference between $d(n)$ and $y(n)$. The sources of signals $x(n)$ and $d(n)$, and the characteristics of signals $y(n)$ and $e(n)$, depend on the given application. The adaptive algorithm adjusts the filter coefficients to minimize the given cost function of the error signal $e(n)$. Therefore, the filter weights are updated such that the error signal is progressively minimized by the adaptation algorithm based on the given objective.

There are several types of digital filters, such as FIR and IIR filters, which can be used for adaptive filtering purposes. As discussed in Chapter 3, the FIR filter is guaranteed to be stable with bounded coefficients and can provide the desired linear phase response. On the other hand, as discussed in Chapter 4, the poles of the IIR filter may move out of the unit circle

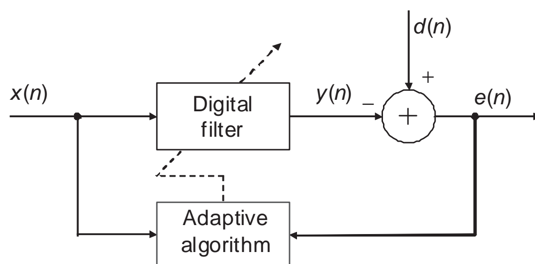


Figure 6.1 A generic block diagram of adaptive filter

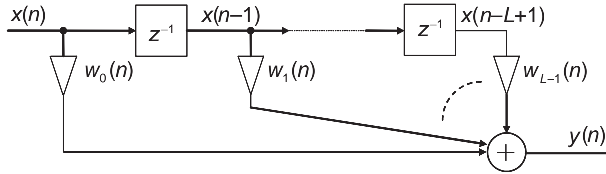


Figure 6.2 Block diagram of time-varying FIR filter for adaptive filtering

during adaptation of the coefficients, thus leading to an unstable system. Because the stability of adaptive IIR filters may not be guaranteed in real-time applications, adaptive FIR filters are widely used for practical applications. This chapter discusses adaptive FIR filters only.

The most widely used adaptive FIR filter is depicted in Figure 6.2. The filter output signal is computed as

$$y(n) = \sum_{l=0}^{L-1} w_l(n)x(n-l), \tag{6.13}$$

where the filter coefficients $w_l(n)$ are functions of time and updated by an adaptive algorithm. We define the input vector of length L at time n as

$$\mathbf{x}(n) \equiv [x(n)x(n-1)\ldots x(n-L+1)]^T, \tag{6.14}$$

and the coefficients vector as

$$\mathbf{w}(n) \equiv [w_0(n)w_1(n)\ldots w_{L-1}(n)]^T. \tag{6.15}$$

Equation (6.13) can be rewritten in vector form as

$$y(n) = \mathbf{w}^T(n)\mathbf{x}(n) = \mathbf{x}^T(n)\mathbf{w}(n). \tag{6.16}$$

The filter output $y(n)$ is compared to the desired signal $d(n)$ to obtain the error signal

$$e(n) = d(n) - y(n) = d(n) - \mathbf{w}^T(n)\mathbf{x}(n). \tag{6.17}$$

Our objective is to determine the weight vector $\mathbf{w}(n)$ to minimize the predetermined cost (or performance) function based on $e(n)$.

6.2.2 Performance Function

The adaptive filter shown in Figure 6.1 uses the adaptive algorithm to update the digital filter coefficients as shown in Figure 6.2 to achieve predetermined performance criteria. There are several performance functions, such as the least squares function, and this chapter introduces the commonly used cost function that is based on the mean-square error (MSE) defined as

$$\xi(n) \equiv E\left[e^2(n)\right]. \tag{6.18}$$

The MSE function can be obtained by substituting (6.17) into (6.18), resulting in

$$\xi(n) = E\left[d^2(n)\right] - 2\mathbf{p}^T\mathbf{w}(n) + \mathbf{w}^T(n)\mathbf{R}\mathbf{w}(n), \tag{6.19}$$

where $\mathbf{p}$ is the cross-correlation vector defined as

$$\begin{aligned}\mathbf{p} &\equiv E[d(n)\mathbf{x}(n)] \\ &= [r_{dx}(0) r_{dx}(1) \ldots r_{dx}(L-1)]^T,\end{aligned} \tag{6.20}$$

and

$$r_{dx}(k) \equiv E[d(n+k)x(n)] \tag{6.21}$$

is the cross-correlation function between $d(n)$ and $x(n)$. In (6.19), $\mathbf{R}$ is the input autocorrelation matrix defined as

$$\begin{aligned}\mathbf{R} &\equiv E[\mathbf{x}(n)\mathbf{x}^T(n)] \\ &= \begin{bmatrix} r_{xx}(0) & r_{xx}(1) & \ldots & r_{xx}(L-1) \\ r_{xx}(1) & r_{xx}(0) & \ldots & r_{xx}(L-2) \\ \vdots & \ldots & \ddots & \vdots \\ r_{xx}(L-1) & r_{xx}(L-2) & \ldots & r_{xx}(0) \end{bmatrix},\end{aligned} \tag{6.22}$$

where $r_{xx}(k)$ is the autocorrelation function of $x(n)$ defined in (6.3). The autocorrelation matrix defined in (6.22) is a symmetric matrix, called a Toeplitz matrix, since all the elements on the main diagonal are equal and the elements on the other diagonals parallel to the main diagonal are also equal.

Example 6.4

Consider the optimum filter with a fixed coefficient w_1 as illustrated in Figure 6.3. If the given signals $x(n)$ and $d(n)$ have characteristics $E[x^2(n)] = 1$, $E[x(n)x(n-1)] = 0.5$, $E\left[d^2(n)\right] = 4$, $E[d(n)x(n)] = -1$, and $E[d(n)x(n-1)] = 1$, find the MSE function ξ based on the fixed coefficient vector.

From (6.22), we get

$$\mathbf{R} = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix},$$

and from (6.20), we have

$$\mathbf{p} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

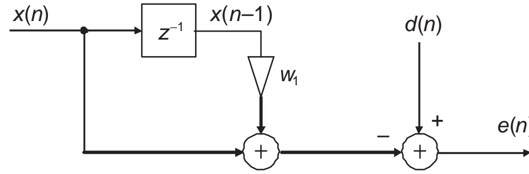


Figure 6.3 A simple optimum filter configuration

Therefore, from (6.19) we obtain

$$\begin{aligned}\xi &= E\left[d^2(n)\right] - 2\mathbf{p}^T\mathbf{w} + \mathbf{w}^T\mathbf{R}\mathbf{w} \\ &= 4 - 2[-1 \quad 1]\begin{bmatrix} 1 \\ w_1 \end{bmatrix} + [1 \quad w_1]\begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ w_1 \end{bmatrix} \\ &= w_1^2 - w_1 + 7.\end{aligned}$$

The optimum filter $\mathbf{w}^{\mathrm{o}}$ minimizes the MSE function $\xi(n)$. The optimum solution can be obtained by differentiation of (6.19) respect to $\mathbf{w}(n)$ and setting the result to $\mathbf{0}$. This operation results in the following normal equation:

$$\mathbf{R}\mathbf{w}^{\mathrm{o}} = \mathbf{p}. \tag{6.23}$$

Thus the optimum filter can be computed as

$$\mathbf{w}^{\mathrm{o}} = \mathbf{R}^{-1}\mathbf{p}. \tag{6.24}$$

By substituting the optimum weight vector in (6.24) for $\mathbf{w}(n)$ in (6.19), we obtain the minimum MSE as

$$\xi_{\min} = E\left[d^2(n)\right] - \mathbf{p}^T\mathbf{w}^{\mathrm{o}}. \tag{6.25}$$

Example 6.5

Consider an FIR filter with two coefficients w_0 and w_1, the desired signal $d(n) = \sqrt{2}\,\sin(\omega_0 n), n \geq 0$, and the reference signal $x(n) = d(n-1)$. Find $\mathbf{w}^{\mathrm{o}}$ and $\xi_{\min}$.

Similar to Example 6.4, we obtain $r_{xx}(0) = E[x^2(n)] = E\left[d^2(n)\right] = 1$, $r_{xx}(1) = \cos(\omega_0)$, $r_{xx}(2) = \cos(2\omega_0)$, $r_{dx}(0) = r_{xx}(1)$, and $r_{dx}(1) = r_{xx}(2)$. From (6.24), we have

$$\mathbf{w}^{\mathrm{o}} = \mathbf{R}^{-1}\mathbf{p} = \begin{bmatrix} 1 & \cos(\omega_0) \\ \cos(\omega_0) & 1 \end{bmatrix}^{-1}\begin{bmatrix} \cos(\omega_0) \\ \cos(2\omega_0) \end{bmatrix} = \begin{bmatrix} 2\cos(\omega_0) \\ -1 \end{bmatrix}.$$

From (6.25), we obtain

$$\xi_{\min} = 1 - [\cos(\omega_0) \quad \cos(2\omega_0)]\begin{bmatrix} 2\cos(\omega_0) \\ -1 \end{bmatrix} = 0.$$

In practical applications, the computation of the optimum filter using (6.24) requires continuous estimation of $\mathbf{R}$ and $\mathbf{p}$ when the signal is non-stationary. In addition, if the filter

length L is large, the dimension of the autocorrelation matrix ($L \times L$) is large, thus the calculation of inverse matrix $\mathbf{R}^{-1}$ requires intensive computation.

The cost function defined by (6.19) is a quadratic function for the adaptive FIR filter with coefficient vector $\mathbf{w}(n)$. For each coefficient vector $\mathbf{w}(n)$, there is a corresponding value of MSE $\xi(n)$. Therefore, the MSE values associated with $\mathbf{w}(n)$ form an $(L+1)$-dimensional space, which is commonly called the error surface, or the performance surface. The height of $\xi(n)$ corresponds to the power of the error signal $e(n)$. When the filter coefficients change, the power of the error signal will also change. Since the error surface is a quadratic function, a unique filter setting $\mathbf{w}(n) = \mathbf{w}^{o}$ will produce the minimum MSE, ξ_{min}. Because matrix $\mathbf{R}$ is positive semi-definite, the quadratic form on the right-hand side of (6.19) indicates that any departure of the weight vector $\mathbf{w}(n)$ from the optimum $\mathbf{w}^{o}$ would increase the MSE from its minimum value given by (6.25). This characteristic is useful for deriving search techniques in seeking the optimum weight vector. In such cases, the objective is to develop an algorithm that can automatically search the error surface to find the optimum weights in order to minimize $\xi(n)$. Therefore, a more useful algorithm can be obtained based on a recursive searching method, which will be introduced in the next section.

For example, if $L=2$ the error surface forms a three-dimensional space called an elliptic paraboloid. If we cut the paraboloid with planes above ξ_{min} that are parallel to the w_0–w_1-plane, we obtain concentric ellipses of constant MSE values. These ellipses are called the error contours.

Example 6.6

Consider an FIR filter with two coefficients w_0 and w_1. The reference signal $x(n)$ is the zero-mean white noise with unit variance. The desired signal is

$$d(n) = b_0 x(n) + b_1 x(n-1) = 0.3x(n) + 0.5x(n-1).$$

Compute and plot the error surface and error contours.

From (6.22), we obtain

$$\mathbf{R} = \begin{bmatrix} r_{xx}(0) & r_{xx}(1) \\ r_{xx}(1) & r_{xx}(0) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

From (6.20), we have

$$\mathbf{p} = \begin{bmatrix} r_{dx}(0) \\ r_{dx}(1) \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}.$$

Substituting these values into (6.19), we get

$$\begin{aligned} \xi &= E\left[d^2(n)\right] - 2\mathbf{p}^T\mathbf{w} + \mathbf{w}^T\mathbf{R}\mathbf{w} \\ &= \left(b_0^2 + b_1^2\right) - 2b_0 w_0 - 2b_1 w_1 + w_0^2 + w_1^2. \end{aligned}$$

Since $b_0 = 0.3$ and $b_1 = 0.5$, we have

$$\xi = 0.34 - 0.6w_0 - w_1 + w_0^2 + w_1^2.$$

The MATLAB® script `example6_6a.m` plots the error surface as shown in Figure 6.4 (top) and the script `example6_6b.m` plots the error contours shown in the bottom part of Figure 6.4. In the script `example6_6a.m`, we define the error function as

```
error=0.34-0.6.*w0-w1+w0.*w0+w1.*w1;
```

and use the MATLAB® function `meshgrid` to define `w0` (w_0) and `w1` (w_1) arrays for the three-dimensional plot of the error surface, and the function `mesh(w0,w1,error)` to plot the colored parametric mesh where the color is proportional to the mesh height. In the script `example6_6b.m`, we use

```
contour(w0(1,:),w1(:,1),error,15);
```

to plot 15 contours based on the error surface defined by the error function `error`.

6.2.3 Method of Steepest Descent

As shown in Figure 6.4, the MSE is a quadratic function of the filter coefficients that can be pictured as a positive-concave hyperparabolic surface with only one global minimum point. Adjusting the coefficients to minimize the error signal involves descending along this surface until reaching the "bottom of the bowl," where the gradient equals zero. Gradient-based algorithms are based on making local estimates of the gradient and moving toward the bottom of the bowl. The steepest descent method reaches the minimum point by following the negative-gradient direction in which the error surface has the greatest rate of decrease.

The steepest descent method is an iterative (recursive) technique that starts from some arbitrary initial weight vector $\mathbf{w}(0)$ and descends to the bottom of the bowl, $\mathbf{w}^o$, by moving on the error surface in the direction of the negative gradient estimated at that point. The mathematical development of the method of steepest descent can be obtained from a geometric approach using directional derivatives of the MSE surface. The gradient of the error surface, $\nabla\xi(n)$, is defined as the gradient of the MSE function with respect to $\mathbf{w}(n)$, and a negative sign indicates that the weight vector is updated in the negative-gradient direction. Thus, the concept of the steepest descent method can be realized as

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \frac{\mu}{2}\nabla\xi(n), \tag{6.26}$$

where μ is the convergence factor (also known as the step size) that determines the stability and convergence speed of the algorithm. Successive corrections of the weight vector in the direction of steepest descent on the error surface should eventually reach the optimum value $\mathbf{w}^o$, which corresponds to the minimum mean-square error $\xi_{\min}$.

When $\mathbf{w}(n)$ has converged to $\mathbf{w}^o$, it reaches the minimum point of the performance surface, $\xi_{\min}$. At this point the gradient $\nabla\xi(n) = \mathbf{0}$, and the adaptation process defined by (6.26) will stop since the update term equals zero, so the weight vector stays at the optimum solution. This convergence process can be viewed as a ball placed on the "bowl-shaped" performance surface at the point $[\mathbf{w}(0), \xi(0)]$. When the ball is released, it will roll down toward the bottom of the bowl, which is the minimum of the surface at the point $[\mathbf{w}^o, \xi_{\min}]$.

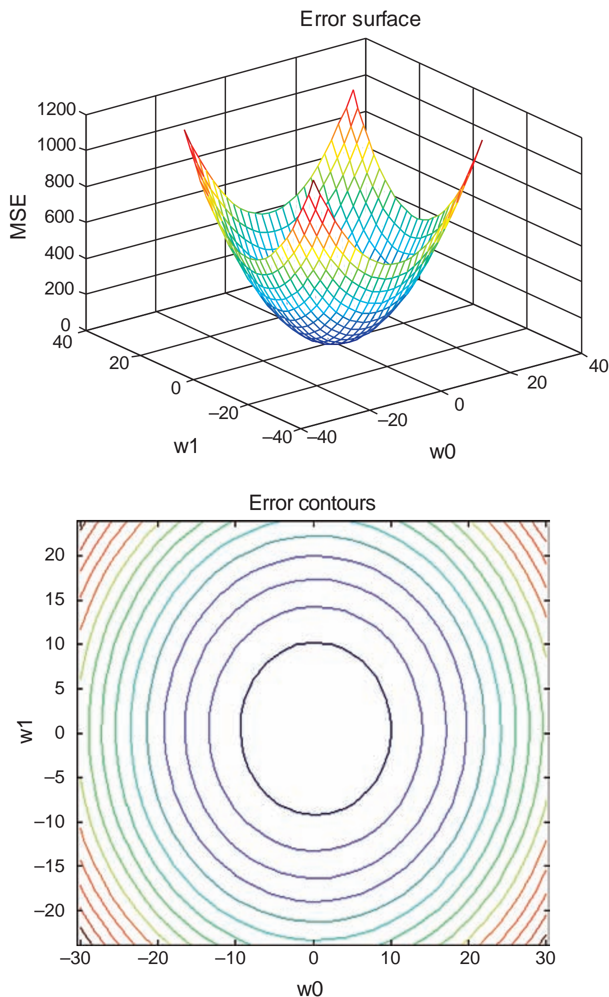


Figure 6.4 Examples of error surface (top) and error contours (bottom), $L = 2$.

6.2.4 LMS Algorithm

In many practical applications, the statistics of $d(n)$ and $x(n)$ are unknown. Therefore, the method of steepest descent cannot be used directly since it assumes the MSE is available to compute the gradient vector. The LMS algorithm developed by Widrow [9] uses the instantaneous squared error, $e^2(n)$, to estimate the MSE as

$$\hat{\xi}(n) = e^2(n). \tag{6.27}$$

Therefore, the gradient estimate is the partial derivative of this cost function with respect to the weight vector as

$$\nabla\hat{\xi}(n) = 2[\nabla e(n)]e(n). \tag{6.28}$$

Since $e(n) = d(n) - \mathbf{w}^T(n)\mathbf{x}(n)$, $\nabla e(n) = -\mathbf{x}(n)$, the gradient estimate becomes

$$\nabla\hat{\xi}(n) = -2\mathbf{x}(n)e(n). \tag{6.29}$$

Substituting this gradient estimate into the steepest descent algorithm defined in (6.26), we get

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\mathbf{x}(n)e(n). \tag{6.30}$$

This is the well-known LMS algorithm, or stochastic gradient algorithm. This algorithm is very simple to implement since it does not require squaring, averaging, or differentiating computations.

The LMS algorithm is illustrated in Figure 6.5 and its steps are summarized as follows:

1. Determine the values of L, μ, and $\mathbf{w}(0)$, where L is the length of the filter, μ is the step size, and $\mathbf{w}(0)$ is the initial weight vector at time $n = 0$. It is very important to determine these parameter values properly in order to achieve the best performance of the LMS algorithm.
2. Compute the adaptive filter output as

$$y(n) = \mathbf{w}^T(n)\mathbf{x}(n) = \sum_{l=0}^{L-1} w_l(n)x(n-l). \tag{6.31}$$

3. Compute the error signal as

$$e(n) = d(n) - y(n). \tag{6.32}$$

4. Update the weight vector using the LMS algorithm given in (6.30), which can be expressed using the following scalar form:

$$w_l(n+1) = w_l(n) + \mu x(n-l)e(n), \quad \text{for } l = 0, 1, \ldots, L-1. \tag{6.33}$$

The filtering operation defined in (6.31) required L multiplications and $L - 1$ additions. The coefficient update defined in (6.33) requires $L + 1$ multiplications and L additions if we multiply μ with $e(n)$ first outside the loop, and use the product to multiply with $x(n - l)$ inside the loop. Therefore, the computational requirement for an adaptive FIR filter with the LMS algorithm is $2L$ additions and $2L + 1$ multiplications.

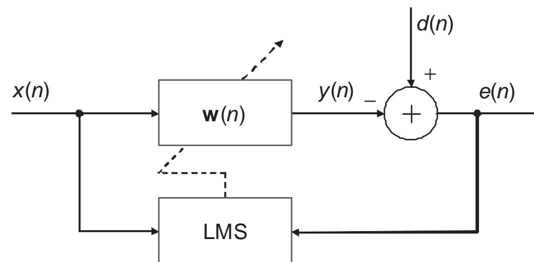


Figure 6.5 Block diagram of adaptive filter with the LMS algorithm

6.2.5 Modified LMS Algorithms

There are three simplified versions of the LMS algorithm that can further reduce the number of multiplications required by the LMS algorithm. However, the convergence rates of these LMS-type algorithms are slower than the LMS algorithm. The first modified algorithm, called the sign–error LMS algorithm, is expressed as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\mathbf{x}(n)\mathrm{sgn}[e(n)], \tag{6.34}$$

where

$$\mathrm{sgn}[e(n)] \equiv \begin{cases} 1, & e(n) \geq 0 \\ -1, & e(n) < 0. \end{cases} \tag{6.35}$$

This sign operation of the error signal is equivalent to a very harsh (1-bit) quantization of $e(n)$. If μ is a negative power-of-2 number, $\mu\mathbf{x}(n)$ can be computed using a right shift of $x(n)$. However, if the sign algorithm is implemented using digital signal processors with a hardware multiplier, the conditional tests in (6.35) require more instruction cycles than the multiplications needed by the LMS algorithm.

The sign operation can be performed on the input signal $x(n)$ instead of the error signal $e(n)$, which results in the following sign–data LMS algorithm:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu e(n)\mathrm{sgn}[\mathbf{x}(n)]. \tag{6.36}$$

Since L branch (IF–ELSE) instructions are required inside the adaptation loop to determine the signs of $x(n-l)$, $l = 0, 1, \ldots, L-1$, slower throughput than in the sign–error LMS algorithm is expected.

Finally, the sign operation can be applied to both $e(n)$ and $x(n)$, resulting in the sign–sign LMS algorithm expressed as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\mathrm{sgn}[e(n)]\mathrm{sgn}[\mathbf{x}(n)]. \tag{6.37}$$

This algorithm requires no multiplication, and is designed for VLSI or ASIC implementation to save multiplications. It is used in adaptive differential pulse code modulation (ADPCM) for speech compression.

Some practical applications dealing with complex signals and the frequency-domain adaptive filtering require complex operations to maintain their phase relationships. The complex adaptive filter uses the complex input vector $\mathbf{x}(n)$ and complex coefficient vector $\mathbf{w}(n)$ expressed as

$$\mathbf{x}(n) = \mathbf{x}_\mathrm{r}(n) + \mathrm{j}\,\mathbf{x}_\mathrm{i}(n) \tag{6.38}$$

and

$$\mathbf{w}(n) = \mathbf{w}_\mathrm{r}(n) + \mathrm{j}\,\mathbf{w}_\mathrm{i}(n), \tag{6.39}$$

where the subscripts r and i denote the real and imaginary parts, respectively.

The complex output signal $y(n)$ is computed as

$$y(n) = \mathbf{w}^T(n)\mathbf{x}(n), \tag{6.40}$$

where all multiplications and additions are complex operations. The complex LMS algorithm adapts the real and imaginary parts of $\mathbf{w}(n)$ simultaneously as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu e(n)\mathbf{x}^*(n), \tag{6.41}$$

where the superscript * denotes the complex-conjugate operation such that $\mathbf{x}^*(n) = \mathbf{x}_\mathrm{r}(n) - \mathrm{j}\,\mathbf{x}_\mathrm{i}(n)$. An example of decomposing complex calculations into real computations is given in Section 6.6.7 for adaptive channel equalizers.

6.3 Performance Analysis

In this section, we briefly discuss some important properties of the LMS algorithm such as stability, convergence rate, excess mean-square error, and finite-wordlength effects. Understanding these properties helps in selecting the best parameters, such as L and μ, to improve the performance of adaptive filters.

6.3.1 Stability Constraint

As shown in (6.30), the correction term of the LMS algorithm to the weight vector includes the feedback $e(n)$ and the step size μ. Thus, the convergence of the adaptive algorithm is determined by the step size value. The stability of the LMS algorithm is guaranteed when the step size satisfies

$$0 < \mu < 2/\lambda_{\max}, \tag{6.42}$$

where $\lambda_{\max}$ is the largest eigenvalue of the autocorrelation matrix $\mathbf{R}$ defined in (6.22).

The computation of $\lambda_{\max}$ is difficult when L is large. Also, true $\mathbf{R}$ may not be available during the design stage, and/or may be changed during operation of the adaptive filter. In practical applications, it is easier to approximate $\lambda_{\max}$ using a simple method. From (6.22) and linear algebra, we have

$$\lambda_{\max} \leq \sum_{l=0}^{L-1} \lambda_l = L r_{xx}(0) = L P_x, \tag{6.43}$$

where λ_l are the eigenvalues of matrix $\mathbf{R}$, and

$$P_x \equiv r_{xx}(0) = E[x^2(n)] \tag{6.44}$$

is the power of $x(n)$. Therefore, the stability condition given in (6.42) can be approximated as

$$0 < \mu < 2/LP_x. \tag{6.45}$$

This equation provides two important principles for determining the value of μ:

1. The upper bound of the step size μ is inversely proportional to the filter length L, thus a smaller μ must be used for a higher order filter, and vice versa.
2. Since the step size is inversely proportional to the input signal power, a larger μ can be used for a low-power signal, and vice versa. A more effective technique is to normalize the step size μ with respect to P_x such that the convergence rate of the algorithm is independent of signal power. The resulting algorithm is called the normalized LMS algorithm, which will be discussed in Section 6.3.4.

6.3.2 Convergence Speed

Convergence of the weight vector $\mathbf{w}(n)$ from $\mathbf{w}(0)$ to $\mathbf{w}^{\mathrm{o}}$ corresponds to the convergence of the MSE from $\xi(0)$ to $\xi_{\min}$. The average time needed for the MSE to decrease from $\xi(0)$ to its minimum value $\xi_{\min}$ is defined as τ_{mse}, which is the commonly used measurement of convergence rate for adaptive algorithms. In addition, a plot of the MSE versus time n is called the learning curve, which is an effective way to describe the transient behavior of adaptive algorithms.

Each adaptive mode has its own time constant for convergence, which is determined by the step size μ and the eigenvalue λ_l associated with that mode. Thus, the time needed for convergence is clearly limited by the slowest mode caused by the minimum eigenvalue, and can be approximated as

$$\tau_{\mathrm{mse}} \cong \frac{1}{\mu\lambda_{\min}}, \tag{6.46}$$

where $\lambda_{\min}$ is the minimum eigenvalue of the matrix $\mathbf{R}$. Because τ_{mse} is inversely proportional to the step size μ, using a smaller μ will result in a larger τ_{mse} (slower convergence). Note that the maximum time constant given in (6.46) is a conservative estimate since only large eigenvalues will exert significant influence on the convergence time in practical applications.

When $\lambda_{\max}$ is very large, only a small μ can satisfy the stability constraint given in (6.42). If $\lambda_{\min}$ is very small, the time constant can be very large, resulting in very slow convergence. The slowest convergence occurs when using the smallest step size $\mu = 1/\lambda_{\max}$. Substituting this step size into (6.46) results in

$$\tau_{\mathrm{mse}} \le \frac{\lambda_{\max}}{\lambda_{\min}}. \tag{6.47}$$

Therefore, the convergence rate is determined by the ratio of the maximum to minimum eigenvalues (called eigenvalue spread) of the matrix $\mathbf{R}$.

In practice, the eigenvalues $\lambda_{\max}$ and $\lambda_{\min}$ are difficult to compute or estimate if the matrix $\mathbf{R}$ is unknown and the order of the filter is high. The eigenvalue spread can be efficiently approximated by the spectral dynamic range, thus (6.47) can be modified as

$$\tau_{\mathrm{mse}} \le \frac{\lambda_{\max}}{\lambda_{\min}} \cong \frac{\max |X(\omega)|^2}{\min |X(\omega)|^2}, \tag{6.48}$$

where $X(\omega)$ is the DTFT of $x(n)$. Therefore, the convergence rate is determined by the characteristics of the magnitude spectrum of the input signal. For example, white noise with a flat spectrum will have a fast convergence speed.

6.3.3 Excess Mean-Square Error

The steepest descent algorithm defined by (6.26) requires the true gradient $\nabla\xi(n)$, which must be computed from the exact MSE for each iteration. After the algorithm has converged, the true gradient $\nabla\xi(n) = \mathbf{0}$, and the algorithm will stop at the optimum solution with the minimum MSE. However, the LMS algorithm uses the instantaneous squared error given by (6.27) to estimate MSE, thus the gradient estimate $\nabla\hat{\xi}(n) \neq \mathbf{0}$. As indicated by (6.26), this will cause $\mathbf{w}(n)$ to vary randomly around $\mathbf{w}^{\mathrm{o}}$, thus producing excess random noise at the filter output. The excess MSE, which is caused by the excess random noise in the weight vector after convergence, can be approximated as

$$\xi_{\text{excess}} \approx \frac{\mu}{2} L P_x \xi_{\min}. \tag{6.49}$$

This approximation shows that the excess MSE is directly proportional to μ. Thus the use of larger step size μ results in a faster convergence rate at the cost of degraded steady-state performance by producing more noise. Therefore, there is a design trade-off between the excess MSE and the convergence speed when choosing the value of μ.

The optimal step size μ is difficult to determine for practical applications. Improper selection of the μ value may make the algorithm unstable, decrease convergence speed, or introduce more excess MSE in the steady state. If the signal is non-stationary and the real-time tracking capability is crucial for a given application, a large μ may be chosen. If the signal is stationary and the initial convergence speed is not important, a small μ can be used to achieve better steady-state performance. In some practical applications, variable step sizes can be used, such as a larger μ at the beginning (or when the environment changed) to achieve faster convergence, and a smaller μ to get better steady-state performance after the adaptive system has converged. Several advanced algorithms called variable step size LMS algorithms have been developed to improve the convergence rate.

The excess MSE given in (6.49) is also proportional to the filter length L, which means that a large L results in more algorithm noise. In addition, (6.45) requires a small μ for a large L, and this also results in slow convergence. On the other hand, a large L may be needed for better filter quality, as in acoustic echo cancellation (will be introduced in Chapter 8). Again, there exists a design trade-off in choosing filter length L and step size μ for a given application.

6.3.4 Normalized LMS Algorithm

The stability, convergence rate, and excess MSE of the LMS algorithm are governed by the step size μ, the filter length L, and the input signal power. As shown in (6.45), the maximum step size μ is inversely proportional to the filter length L and the signal power. One important technique to make the convergence speed independent of signal power and filter length while maintaining the desired steady-state performance is the normalized LMS algorithm (NLMS) expressed as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu(n)\mathbf{x}(n)e(n), \tag{6.50}$$

where $\mu(n)$ is the time-varying step size normalized by the filter length and signal power as

$$\mu(n) = \frac{\alpha}{L\hat{P}_x(n) + c}, \tag{6.51}$$

where $\hat{P}_x(n)$ is the estimate of the power of $x(n)$ at time n, $0 < \alpha < 2$ is a constant, and c is a very small constant to prevent division by zero or using a very large step size for a very weak signal at time n. Note that $\hat{P}_x(n)$ can be estimated recursively using the technique introduced in Chapter 2.

Some useful implementation considerations for the NLMS algorithm are summarized as follows:

1. Choose $\hat{P}_x(0)$ as the best a priori estimate of the input signal power.
2. A software constraint may be required to ensure that $\mu(n)$ is bounded if $\hat{P}_x(n)$ is very small when the signal is absent.

6.4 Implementation Considerations

For many real-world applications, adaptive filters are implemented on fixed-point hardware. It is important to understand some practical issues such as the finite-wordlength effects for adaptive filters to meet the design specifications [12].

6.4.1 Computational Issues

The coefficient update defined in (6.33) requires $L + 1$ multiplications and L additions if we multiply μ by $e(n)$ outside the loop. Suppose the input vector $\mathbf{x}(n)$ is stored in the array `x[]`, the error signal is `en`, the coefficient vector is `w[]`, the step size is `mu`, and the filter length is `L`; then (6.33) can be implemented in C as follows:

```
uen=mu*en;               // Perform u*e(n) outside the loop
for (l=0; l<L; l++)      // l=0, 1, . . . , L-1
{
   w[l] += uen=x[l];     // Update weight using LMS algorithm
}
```

The architecture of most digital signal processors has been optimized for convolution operations to compute filter output $y(n)$ given in (6.31). However, the coefficient update operations in (6.33) cannot take advantage of this special architecture because each update involves loading the coefficient into the accumulator, performing a multiply–add operation, and storing the result back into memory. The computational complexity can be reduced by skipping part of the coefficient update. In this case, the update is performed only for a portion of filter coefficients in one sampling period. The update for the remaining portions may be done during the following sampling periods.

In some practical applications, the desired signal $d(n)$, and thus the error signal $e(n)$, is not available until several sampling intervals later. In addition, in the implementation of adaptive filters using processors with a pipeline architecture, the computational delay is an inherent

problem. The delayed LMS algorithm expressed as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu e(n-\Delta)\mathbf{x}(n-\Delta) \tag{6.52}$$

can be used to solve these problems. The delay in the coefficient adaptation has only a slight influence on the steady-state behavior of the LMS algorithm. The delayed LMS algorithm with delay $\Delta = 1$ is widely used in implementing adaptive FIR filtering on some digital signal processors with pipeline architecture, such as the TMS320C55xx.

6.4.2 Finite-Precision Effects

Finite-precision effects in adaptive filters are analyzed in this section using fixed-point arithmetic and methods for confining these effects to acceptable levels. We assume the input data samples are properly scaled so that their values lie between -1 and 1. As introduced in Chapter 2, the techniques used to prevent overflow are scaling, saturation arithmetic, and guard bits. For adaptive filters, the feedback path of using $e(n)$ to update filter coefficients makes scaling more complicated. Also, the dynamic range of the filter output is determined by the time-varying filter coefficients, which are unknown at the design stage.

For adaptive FIR filtering with the LMS algorithm, the scaling of the filter output and coefficients can be achieved by scaling the "desired" signal, $d(n)$. The scaling factor α, where $0 < \alpha \leq 1$, is used to prevent overflow of the filter coefficients during the coefficient update. Reducing the magnitude of $d(n)$ reduces the gain demand on the filter, thereby reducing the magnitude of the coefficient values. Since α only scales the desired signal, it will not affect the convergence rate, which depends on the magnitude spectrum and power of input signal $x(n)$.

The finite-precision LMS algorithm can be described as follows using rounding operations:

$$y(n) = R\left[\sum_{l=0}^{L-1} w_l(n)x(n-l)\right] \tag{6.53}$$

$$e(n) = R[\alpha d(n) - y(n)] \tag{6.54}$$

$$w_l(n+1) = R[w_l(n) + \mu x(n-l)e(n)], \quad l = 0, 1, \ldots, L-1, \tag{6.55}$$

where $R[x]$ denotes the fixed-point rounding of the quantity x.

When updating coefficients according to (6.55), the product $\mu x(n-l)e(n)$ is a double-precision number, which is added to the original stored weight value, $w_l(n)$, and then rounded to obtain the updated value, $w_l(n+1)$. The power of the overall roundoff noise comes mainly from the quantization of filter coefficients, which is inversely proportional to the step size μ. Although a small value of μ reduces the excess MSE, it will increase quantization error.

There is another factor to be considered in the selection of step size μ. As mentioned in Section 6.2, the adaptive algorithm is aimed at minimizing the error signal, $e(n)$. As the weight vector is updated to minimize the MSE, the amplitude of the error signal will decrease during the adaptation. The LMS algorithm updates the current coefficient value $w_l(n)$ by adding a correction term, $R[\mu x(n-l)e(n)]$, which becomes smaller and smaller when $e(n)$ gets smaller. Adaptation will stop when this update term is rounded to zero if its value is smaller than the LSB of the hardware. This phenomenon is known as "stalling" or "lockup." This problem may

be solved by using more bits, and/or using larger step size μ, to guarantee convergence of the algorithm. However, using a large step size will increase excess MSE.

The leaky LMS algorithm may be used to reduce numerical errors accumulated in the filter coefficients. This algorithm prevents coefficient update overflow from finite-precision implementation by providing a compromise between minimizing the MSE and constraining the energy of the adaptive filter. The leaky LMS algorithm can be expressed as

$$\mathbf{w}(n+1) = \nu\, \mathbf{w}(n) + \mu\, \mathbf{x}(n)e(n), \tag{6.56}$$

where ν is the leakage factor, $0 < \nu \leq 1$. The leaky LMS algorithm not only prevents unconstrained weight overflow, but also limits the power of output signal $y(n)$ in order to avoid nonlinear distortion of transducers (such as loudspeakers) driven by the filter output.

It can be shown that leakage is equivalent to adding low-level white noise to the weight vector. Therefore, this approach results in some performance degradation in adaptive filters. The value of the leakage factor is determined as a compromise between the robustness of algorithm and the loss of performance. The excess power of errors caused by the leakage is proportional to $[(1-\nu)/\mu]^2$. Therefore, $(1-\nu)$ should be kept smaller than μ in order to maintain an acceptable level of performance.

6.4.3 MATLAB® Implementations

MATLAB® provides the function `adaptfilt` to support adaptive filtering [13–15]. The syntax of this function is

```
h = adaptfilt.algorithm(input1,input2,...)
```

This function returns the adaptive filter object `h` that uses the adaptive algorithm specified by `algorithm`. The algorithm string determines which adaptive algorithm the `adaptfilt` object implements. Some LMS-type algorithms for adaptive FIR filters are summarized in Table 6.1. The adaptive filter objects use different LMS-type algorithms to update filter coefficients. For example,

```
h = adaptfilt.lms(l,stepsize,leakage,coeffs,states)
```

constructs the adaptive FIR filter `h` using the LMS algorithm. The input parameters are defined as follows:

Table 6.1 Adaptive FIR filter objects with various LMS-type algorithms

Object.Algorithm	Description
`adaptfilt.lms`	Direct-form, (leaky) LMS algorithm
`adaptfilt.sd`	Direct-form, sign–data LMS algorithm
`adaptfilt.se`	Direct-form, sign–error LMS algorithm
`adaptfilt.ss`	Direct-form, sign–sign LMS algorithm
`adaptfilt.nlms`	Direct-form, normalized LMS algorithm
`adaptfilt.dlms`	Direct-form, delayed LMS algorithm
`adaptfilt.blms`	Block-form, LMS algorithm

`l` – the filter length L;
`stepsize` – the step size μ, a positive scalar defaults to 0.1;
`leakage` – the leakage factor ν, which is a scalar between 0 and 1 (defaults to 1 for no leakage; if the leakage factor is less than one, the leaky LMS algorithm is implemented);
`coeffs` – the vector contains the initial filter coefficients, defaults to a zero vector; and
`states` – the vector consists of initial filter states, defaults to a zero vector.

Some default parameters of the adaptive filter `h=adaptfilt.lms()` can be changed as in the following MATLAB® command:

```
set(h,paramname,paramval)
```

Note that MATLAB® also supports other types of adaptive algorithms such as recursive least squares, affine projection, and frequency-domain adaptive filters.

Example 6.7

In this example, the reference input signal $x(n)$ is normally distributed random numbers. This signal is filtered by the FIR filter with coefficient vector $\mathbf{b} = \{0.1, 0.2, 0.4, 0.2, 0.1\}$ to generate the desired signal $d(n)$. The goal of the adaptive filter is to generate the output signal $y(n)$ to approximate $d(n)$ so that the error signal $e(n)$, the difference, is minimized.

The following MATLAB® script (`example6_7.m`, adapted from the MATLAB® **Help** menu) implements the adaptive FIR filter with the LMS algorithm:

```
randn('seed',12345);            % Seed for noise generator
x = randn(1,128);               % Reference input signal x(n)
b = [0.1,0.2,0.4,0.2,0.1];      % An FIR filter to be identified
d = filter(b,1,x);              % Desired signal d(n)
mu = 0.05;                      % Step size mu
h = adaptfilt.lms(5,mu);        % FIR filter with LMS algorithm
[y,e] = filter(h,x,d);          % Adaptive filtering
```

In the code, the filter length is $L = 5$ and the step size is $\mu = 0.05$. The desired signal $d(n)$, the output signal $y(n)$, and the error signal $e(n)$ are plotted in Figure 6.6. It shows that the filter output $y(n)$ is gradually approximated to $d(n)$, thus the error signal $e(n)$ converges (reduces) to zero in about 80 iterations. At the end of the adaptation (iteration 128), the unknown system described by the vector `b` has been identified by the adaptive FIR filter coefficients as [0.0998 0.1996 0.3994 0.1995 0.0995], which is very close to the actual FIR system defined in the vector `b` as [0.1, 0.2, 0.4, 0.2, 0.1].

In the MATLAB® code `example6_7.m`, the following adaptive filtering syntax is used:

```
[y,e] = filter(h,x,d);
```

This function filters the input signal vector `x` through the adaptive filter object `h` and uses `d` as the desired signal vector, producing the output signal vector `y` and the error signal vector `e`. The vectors `x`, `d`, and `y` must have the same length.

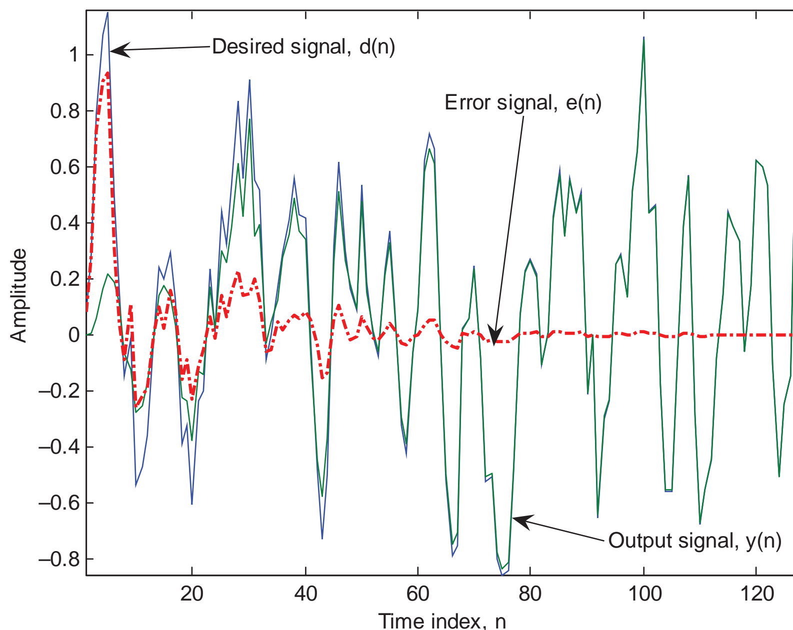


Figure 6.6 Performance of adaptive FIR filter with the LMS algorithm

The function `maxstep` (defaults to zero) can be used to determine a reasonable range of step size values for the signals being processed. The syntax

```
mumax = maxstep(h,x);
```

predicts the bound on the step size to guarantee convergence of the mean values of the adaptive filter coefficients.

6.5 Practical Applications

There are four general classes of adaptive filtering applications: system identification, prediction, noise cancellation, and inverse modeling. As shown in Figure 6.5, the essential difference among these applications is the configuration of signals $x(n)$, $d(n)$, $y(n)$, and $e(n)$.

6.5.1 Adaptive System Identification

System identification is a technique used to model (or identify) an unknown system [16]. The general block diagram of adaptive system identification is illustrated in Figure 6.7, where $P(z)$ is an unknown system to be identified by the adaptive filter $W(z)$. By exciting both $P(z)$ and $W(z)$ using the same excitation signal $x(n)$ and minimizing the difference of output signals $y(n)$ and $d(n)$, we can identify the characteristics of $P(z)$ using the model $W(z)$ from the I/O viewpoint.

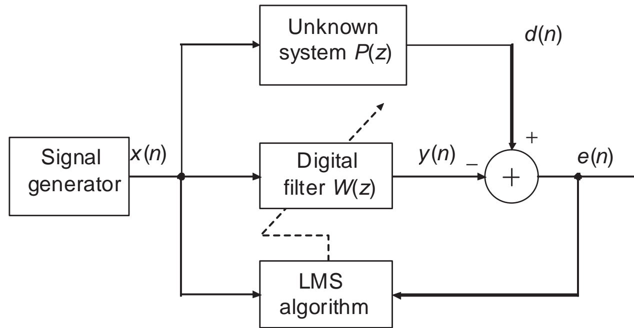


Figure 6.7 Adaptive system identification using the LMS algorithm

As shown in Figure 6.7, the modeling error is

$$\begin{aligned} e(n) &= d(n) - y(n) \\ &= \sum_{l=0}^{L-1} [p(l) - w_l(n)]x(n-l), \end{aligned} \tag{6.57}$$

where $p(l)$ is the impulse response of the unknown plant $P(z)$. Assuming that the unknown system $P(z)$ is an FIR system of length L, and using white noise as the excitation signal, minimizing $e(n)$ will make $w_l(n)$ approach $p(l)$, that is,

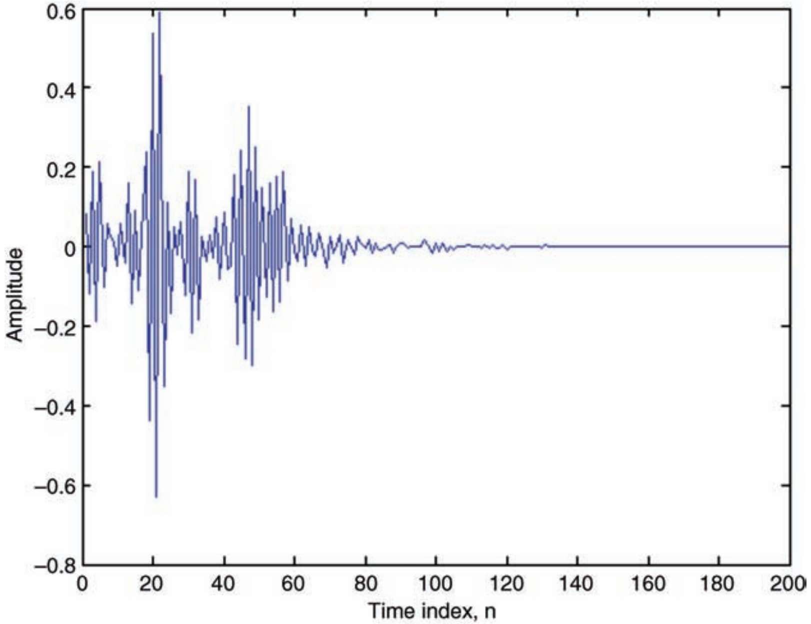
$$w_l(n) = p(l), \quad l = 0, 1, \ldots, L-1. \tag{6.58}$$

In this perfect system identification case, $W(z)$ identifies $P(z)$ after the adaptive filter has converged, and the error signal $e(n)$ converges to zero.

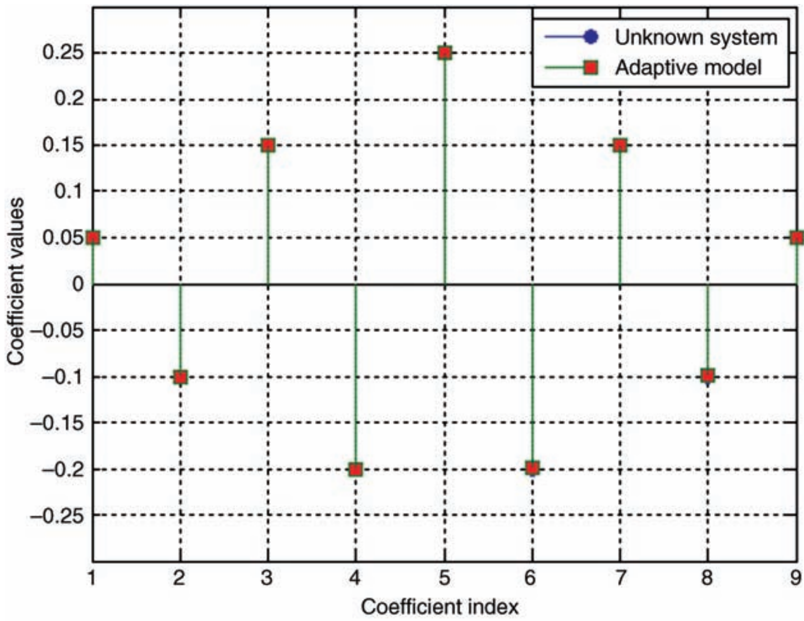
When the difference between the physical system response $d(n)$ and model response $y(n)$ has been minimized, the adaptive model $W(z)$ approximates $P(z)$ from the I/O viewpoint. When the plant is time varying, the adaptive algorithm will continually track time variations of the plant dynamics by minimizing the modeling error. We will apply the adaptive system identification technique in Chapter 8 for adaptive echo cancellation applications.

Example 6.8

Assume the excitation signal $x(n)$ shown in Figure 6.7 is a normally distributed random signal. This excitation signal is applied to the unknown system $P(z)$ that is the FIR filter defined by the coefficient vector `b` = [0.05, −0.1, 0.15, −0.2, 0.25, −0.2, 0.15, −0.1, 0.05]. The MATLAB® script (`example6_8.m`, adapted from the MATLAB® **Help** menu) implements adaptive system identification using the adaptive FIR filter of length $L = 9$ with the LMS algorithm. Figure 6.8(a) shows that the error signal $e(n)$ converges to zero in about 120 iterations. As predicted in (6.58) and verified by Figure 6.8(b), the adaptive filter coefficients converge to the corresponding unknown FIR system coefficients after convergence of the algorithm. This example shows that the adaptive model $W(z)$ can exactly identify the unknown system $P(z)$ since $P(z)$ is assumed to be an FIR system with known order.



(a) Error signal, $e(n)$



(b) Coefficients of $P(z)$ and $W(z)$

Figure 6.8 Adaptive system identification of unknown plant

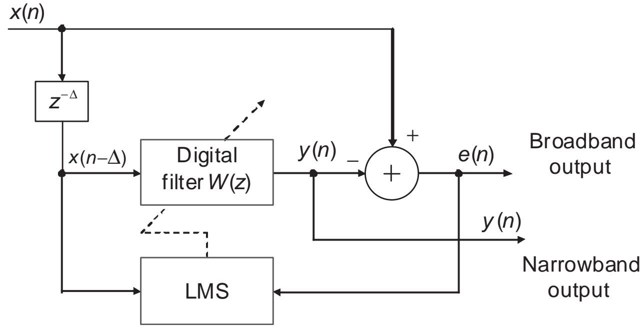


Figure 6.9 Adaptive predictor with the LMS algorithm

6.5.2 Adaptive Prediction

A linear predictor estimates the values of a signal at future time [17]. This technique has been successfully applied to a wide range of applications such as speech coding and separating signals from noise. As illustrated in Figure 6.9, the adaptive predictor consists of an adaptive filter using the reference input $x(n-\Delta)$ to predict its future value $x(n)$, where Δ is the number of delay samples. The predictor output $y(n)$ is expressed as

$$y(n) = \sum_{l=0}^{L-1} w_l(n)x(n-\Delta-l). \tag{6.59}$$

The filter coefficients are updated by the LMS algorithm as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\, \mathbf{x}(n-\Delta)e(n), \tag{6.60}$$

where $\mathbf{x}(n-\Delta) = [x(n-\Delta)x(n-\Delta-1)\ldots x(n-\Delta-L+1)]^T$ is the delayed reference signal vector, and $e(n) = x(n) - y(n)$ is the prediction error. For example, proper selection of the prediction delay Δ allows improved frequency estimation performance of multiple sinusoids embedded in white noise.

Consider the application of using an adaptive predictor to enhance the primary signal which consists of M sinusoids corrupted by white noise expressed as

$$\begin{aligned} x(n) &= s(n) + v(n) \\ &= \sum_{m=0}^{M-1} A_m \sin(\omega_m n + \phi_m) + v(n), \end{aligned} \tag{6.61}$$

where $v(n)$ is the zero-mean white noise with unit variance σ_v^2. In this application, the structure shown in Figure 6.9 is called an adaptive line enhancer, which can efficiently track the sinusoidal components in $x(n)$ and separate the narrowband signal $s(n)$ from the broadband noise $v(n)$. This technique is very effective in practical applications where the signal and noise parameters are unknown and/or time varying.

As shown in Figure 6.9, we want the highly correlated components of $x(n)$ to appear in $y(n)$. This is accomplished by adjusting the adaptive filter $W(z)$ to form a bandpass filter with multiple passbands centered at the frequencies of the sinusoidal components, and to compensate the phase differences (caused by Δ) of the narrowband signals so that they can cancel correlated components in $d(n)$ to minimize the error signal $e(n)$. In this case, the output $y(n)$ is the enhanced signal, which contains multiple sinusoids.

Example 6.9

Assume the signal $x(n)$ shown in Figure 6.9 consists of the desired sine wave that is corrupted by white noise. The delay $\Delta = 1$ can decorrelate the white-noise component in $x(n)$ and $x(n-\Delta)$, but the sinusoidal component is still highly corrected. Thus the adaptive filter corrects the amplitude and phase differences of the sinusoidal component and generates the enhanced output sine wave. This example is implemented in MATLAB® script `example6_9.m`. The enhanced output $y(n)$ and the error signal $e(n)$ are plotted in Figure 6.10. As shown in the figure, the error signal gradually converges to the broadband white noise, while the enhanced output signal $y(n)$ approaches the desired sine wave.

In some digital communications and signal detection applications, the desired broadband signal $v(n)$ is corrupted by an additive $s(n)$. The objective of the adaptive filter is to adjust the amplitude and phase of narrowband interference in $x(n-\Delta)$ to cancel the undesired narrowband interference at $x(n)$, thus resulting in an enhanced broadband signal at $e(n)$. In this application, the desired output signal from the adaptive predictor is the broadband signal $e(n)$ as shown in Figure 6.9. Note that in order to decorrelate broadband signals such as speech, the required delay Δ is usually larger than one.

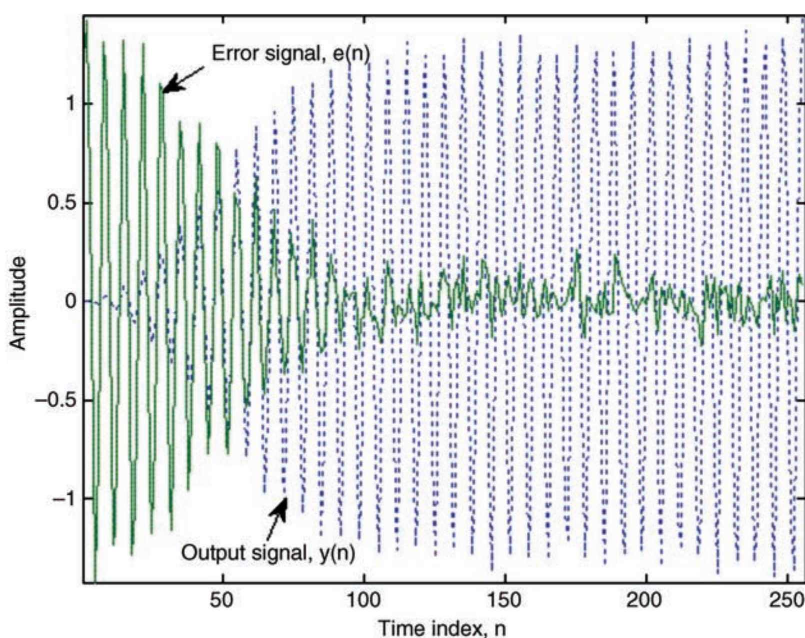


Figure 6.10 Performance of adaptive line enhancer

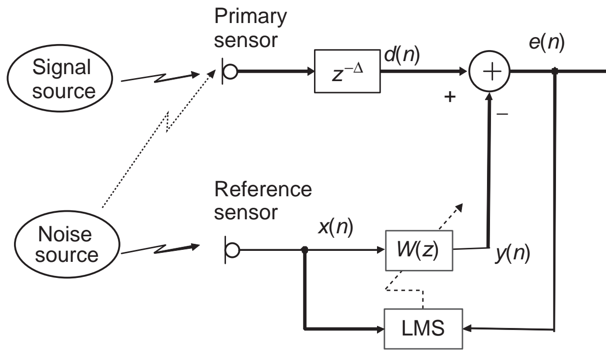


Figure 6.11 Basic concept of adaptive noise canceling

6.5.3 Adaptive Noise Cancellation

The widespread use of cell phones has significantly increased the use of voice devices under high acoustic noise environments. Unfortunately, intense background noise often corrupts speech and degrades the effectiveness of communication. The adaptive noise canceler employs an adaptive filter with the LMS algorithm and can be applied to cancel the noise components embedded in the primary signal [18].

As illustrated in Figure 6.11, the adaptive noise canceler has two input signals: the primary signal $d(n)$ and the reference signal $x(n)$. The primary sensor is placed close to the signal source to pick up the desired primary signal. The reference sensor is placed close to the noise source to sense only the noise. The suitable delay unit, $z^{-\Delta}$, is inserted in the primary channel to guarantee the causality of adaptive filter $W(z)$. Assume that the signal leakage from the signal source to the reference sensor can be prevented so the reference input $x(n)$ contains only noise. The primary signal $d(n) = s(n) + x'(n)$, that is, $d(n)$ consists of desired signal $s(n)$ plus the noise $x'(n)$, originates from the noise source. The noise $x'(n)$ is highly correlated with $x(n)$ since they come from the same noise source. The objective of the adaptive filter is to use the reference input $x(n)$ to estimate the undesired noise $x'(n)$. The filter output $y(n)$ is then subtracted from the primary signal $d(n)$ to attenuate noise $x'(n)$. If $y(n) = x'(n)$, we can obtain $e(n)$ as the desired signal $s(n)$.

Example 6.10

As shown in Figure 6.11, assume that $s(n)$ is a sine wave, $x(n)$ is white noise, and the path from the noise source to the primary sensor is a simple FIR system. We use an adaptive FIR filter with the LMS algorithm for noise cancellation, which is implemented in the MATLAB® script `example6_10.m`. The adaptive filter will approximate the path from the noise source to the primary sensor, thus the filter output $y(n)$ will approximate $x'(n)$ in order to cancel it. Therefore, the error signal $e(n)$ (enhanced output signal) will gradually approach the desired sinusoidal signal $s(n)$, as shown in Figure 6.12.

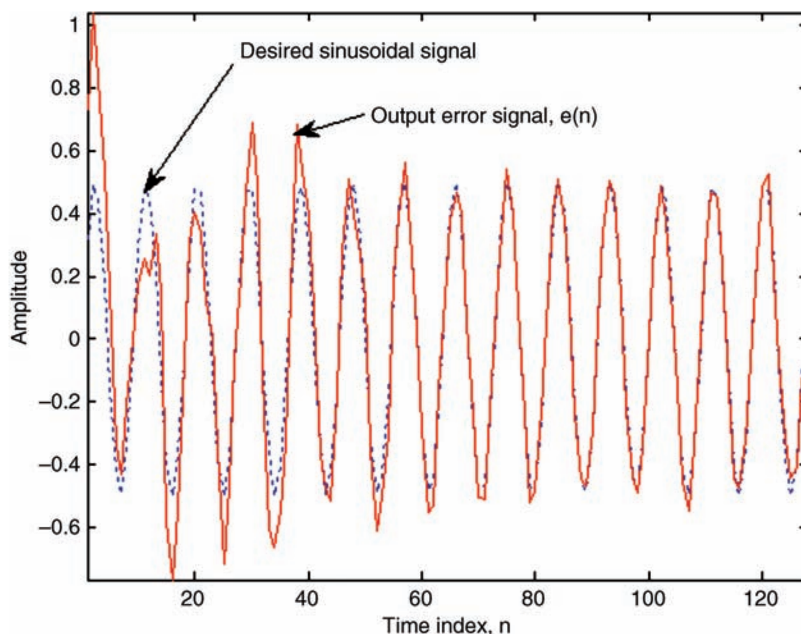


Figure 6.12 The enhanced sine wave given in $e(n)$ approaches the original $s(n)$

To apply adaptive noise cancellation effectively, the following two conditions must be satisfied:

1. The reference noise picked up by the reference sensor must be highly correlated with the noise components in the primary signal picked up by the primary sensor.
2. The reference sensor should only pick up noise, that is, it must avoid picking up signals from the signal source.

Unfortunately, these two conditions conflict with each other. The first condition usually requires a close placement between the primary and reference sensors; however, the second condition requires avoiding the signal components generated by the signal source being picked up by the reference sensor. The "signal leakage" from the signal source to the reference sensor will degrade the performance of adaptive noise cancellation because the presence of the signal components in the reference signal will cause the adaptive filter to cancel the desired signal components along with the undesired noise.

The signal leakage problem may be solved by placing the primary sensor far away from the reference sensor. Unfortunately, this arrangement requires a large-order filter to identify the long noise path from the noise source to the primary sensor. Furthermore, it is not always feasible to place the reference sensor far away from the signal source while still maintaining highly correlated noise components in both $d(n)$ and $x(n)$. An alternative method for reducing leakage is to place an acoustic barrier (oxygen masks used by pilots in an aircraft cockpit, for example) between the primary and reference sensors. However, many applications do not allow any acoustic barrier between sensors, and the barrier may

also reduce the correlation of the noise components in the primary and reference signals. Another technique is to control the adaptive algorithm to update filter coefficients only during the noise-only periods, such as the silent intervals in speech. Unfortunately, this method depends on a reliable speech activity detector (which will be presented in Chapter 8) that is very application dependent. Also, this technique fails to track changes in the environment during the speech periods since the adaptation is frozen.

In some practical applications, the primary input is a broadband signal corrupted by undesired sinusoidal noise. The conventional method of eliminating such sinusoidal interference is to use a notch filter with a center frequency that is tuned to the frequency of the interference. The adaptive notch filter has the capability to track the frequency of the interference, thus is especially useful when the interfering sinusoidal frequency is unknown and drifts in time.

For changing amplitude and phase of a sinusoidal signal, two filter coefficients are needed. A single-frequency adaptive notch filter with two adaptive weights is illustrated in Figure 6.13. Assume the frequency of sinusoidal noise is ω_0, and the reference input signal is the cosine signal $x_0(n) = A\cos(\omega_0 n)$. A 90° phase shifter is used to produce the quadrature signal $x_1(n) = A\sin(\omega_0 n)$ (or use the quadrature oscillators introduced in Section 4.6.2 to generate both the sine and cosine signals).

The LMS algorithm employed in Figure 6.13 is summarized as

$$w_l(n+1) = w_l(n) + ue(n)x_l(n), \quad l = 0, 1. \tag{6.62}$$

The center frequency of the adaptive notch filter will be tuned automatically to the frequency of the narrowband primary noise. Therefore, the noise at that frequency is attenuated. This adaptive notch filter provides a simple method for eliminating sinusoidal interference. For example, in many practical applications such as measured medical signals, the primary signal is a broadband signal corrupted by power line interference. In this case, $x_0(n) = A\cos(2\pi f_0 n)$ is used as the reference signal where the frequency f_0 is 60 Hz in the USA and many other countries.

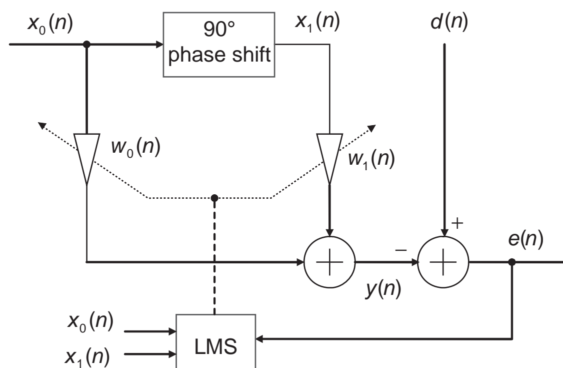


Figure 6.13 Single-frequency adaptive notch filter

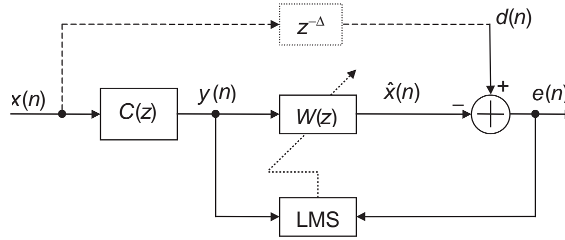


Figure 6.14 An adaptive channel equalizer as an example of inverse modeling

6.5.4 Adaptive Inverse Modeling

In many practical applications, we have to estimate the inverse model of an unknown system in order to compensate its effects. For example, in digital communications, the transmission of high-speed data through a channel is limited by intersymbol interference caused by channel distortion. Data transmission through channels with severe distortion can be solved by designing an adaptive equalizer in the receiver that counteracts and tracks the unknown and changing channels.

As illustrated in Figure 6.14, the received signal $y(n)$ is different from the original signal $x(n)$ because it was distorted (convoluted or filtered) by the overall channel transfer function $C(z)$, which includes the transmit filter, the transmission medium, and the receive filter. To recover the original signal, we need the equalizer filter $W(z)$ with the following transfer function:

$$W(z) = \frac{1}{C(z)}, \tag{6.63}$$

that is, $C(z)W(z) = 1$ such that $\hat{x}(n) = x(n)$. In other words, we have to design an equalizer which is the inverse modeling of the unknown system transfer function $C(z)$.

As shown in Figure 6.14, the adaptive filter requires the desired signal $d(n)$ for computing the error signal $e(n)$ for the adaptation of the LMS algorithm. In theory, the delayed version of the transmitted signal, $x(n-\Delta)$, is the desired response for the adaptive equalizer $W(z)$. However, since the adaptive filter is located in the receiver, the desired signal generated by the transmitter is not available at the receiver. The desired signal may be generated locally in the receiver using two methods. During the training stage, the adaptive equalizer coefficients are adjusted by transmitting a short training sequence. This known transmitted sequence is also generated in the receiver and is used as the desired signal $d(n)$ to compute the error signal. After the short training period, the transmitter begins to transmit the data sequence. In the data mode, the output of the equalizer, $\hat{x}(n)$, is used by a decision device to produce binary data. Assuming the output of the decision device is correct, the binary sequence can be used as the desired signal $d(n)$ to generate the error signal $e(n)$ for the LMS algorithm.

Example 6.11

The adaptive channel equalizer shown in Figure 6.14 is implemented using MATLAB® script `example6_11.m` [19]. A simple FIR filter is used as the channel $C(z)$, and an adaptive FIR filter is applied with the LMS algorithm as an adaptive equalizer. To ensure the causality of the adaptive filter, the delay used to generate $d(n)$ is half the filter length of $W(z)$, that is, $L/2$. As shown in Figure 6.15, the error signal $e(n)$ is minimized such that the adaptive filter approximates the inverse model of the channel.

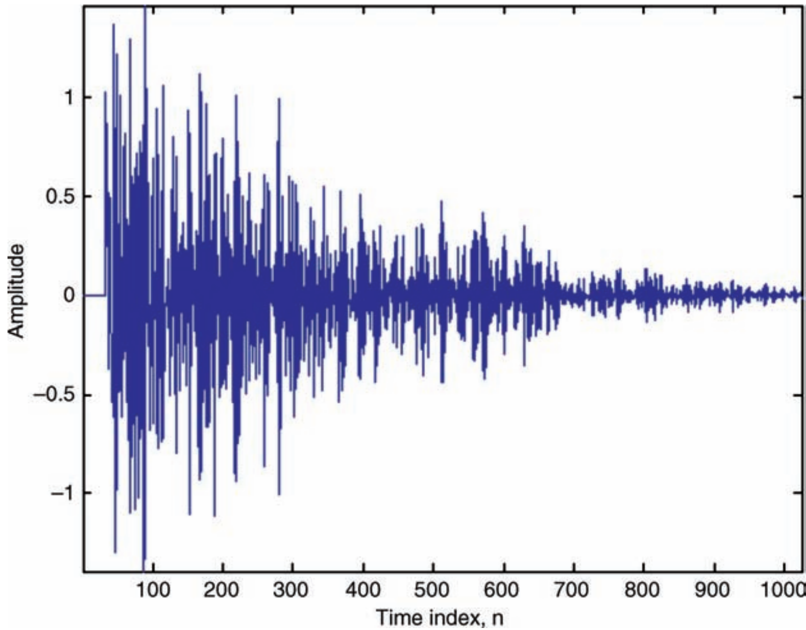


Figure 6.15 Convergence of adaptive channel equalizer indicated by the error signal $e(n)$

6.6 Experiments and Program Examples

This section presents adaptive filtering experiments based on adaptive FIR filters with the LMS-type algorithms.

6.6.1 LMS Algorithm Using Floating-Point C

The block diagram of an adaptive FIR filter with the LMS algorithm is shown in Figure 6.5. The floating-point C implementation is listed in Table 6.2.

In the experiment, the desired signal $d(n)$ is a sine wave and the reference input signal $x(n)$ is the same sine wave corrupted by white noise. This experiment reduces the noise in the reference signal using an adaptive FIR filter of length 128 and step size 0.003. The adaptive filter will converge in about 500 iterations. Table 6.3 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment. Verify the experimental result by comparing the magnitude spectrum of the filtered output signal $y(n)$ to the magnitude spectrum of the reference input signal $x(n)$. The white-noise component in $x(n)$ should be reduced, resulting in an enhanced sinusoidal component in $y(n)$.
3. Modify the experiment by using the CCS graphic tool to plot $x(n)$, $y(n)$, and $e(n)$.
4. Modify the experiment to compute $e^2(n)$ and use the CCS graphic tool to plot the squared error, $e^2(n)$. Examine and observe the number of iterations needed for $e^2(n)$ decreases to the power of the white noise.

Table 6.2 Floating-point C for LMS algorithm

```
void float_lms(LMS *lmsObj)
{
    LMS    *lms=(LMS *)lmsObj;
    double *w,*x,y,ue;
    short j,n;
    n = lms->order;
    w = &lms->w[0];
    x = &lms->x[0];

    // Update signal buffer
    for(j=n-1; j>0; j--)
    {
        x[j] = x[j-1];
    }
    x[0] = lms->in;

    // Compute filter output
    y = 0.0;
    for(j=0; j<n; j++)
    {
        y += w[j] * x[j];
    }
    lms->out = y;

    // Compute error signal
    lms->err = lms->des - y;

    // Update filter coefficients
    ue = lms->err * lms->mu;
    for(j=0; j<n; j++)
    {
        w[j] += ue * x[j];
    }
}
```

Table 6.3 File listing for the experiment Exp6.1

Files	Description
float_lmsTest.c	Program for testing adaptive FIR filter
float_lms.c	C function for floating-point LMS algorithm
float_lms.h	C header file for experiment
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
input.pcm	Reference input signal file
desired.pcm	Desired signal file

5. Fix the adaptive filter length, change the step size value, and compare the convergence rates and the steady-state value of $e^2(n)$.
6. Fix the step size value, change the adaptive filter length, and redo step 5 to compare the convergence rate and the steady-state value of $e^2(n)$.

6.6.2 Leaky LMS Algorithm Using Fixed-Point C

In this experiment, the Q15 format is used for fixed-point C implementation of the leaky LMS algorithm. The reference and desired signals $x(n)$ and $d(n)$ used by this experiment are the same as in the previous experiment. The use of the leaky LMS algorithm and rounding operations can reduce the finite-precision effects of using a fixed-point implementation. As mentioned in Section 6.4.2, it is important to choose a proper leaky factor. The files used for the experiment are listed in Table 6.4.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment. Verify the experimental result using the methods given in step 2 of Exp6.1.
3. Compute the difference (mainly due to finite-precision effects) of the filtered output signal $y(n)$ with the result obtained in the previous experiment using float-point implementation.
4. Fix the length of the adaptive FIR filter, change the step size value, and compare the convergence rates and the steady-state value of $e^2(n)$. Repeat the experiment using a fixed step size but changing the filter length.
5. Select the best combination of step size and filter length from step 4, rerun the experiment, and use a graphic plot to show the following results:
   (a) Observe the convergence rate changes with different leaky factor ν values.
   (b) Observe the change of steady-state value $e^2(n)$ with different leaky factor ν values.

6.6.3 Normalized LMS Algorithm Using Fixed-Point C and Intrinsics

The ETSI (European Telecommunications Standards Institute) operators (functions) are useful for developing DSP applications such as the GSM (Global System for Mobile communications) standards including speech coders. The original ETSI operators are fixed-point C functions. The C55xx compiler supports ETSI functions by mapping them directly to its intrinsics. This experiment implements the normalized leaky LMS algorithm

Table 6.4 File listing for the experiment Exp6.2

Files	Description
`fixPoint_leaky_lmsTest.c`	Program for testing leaky LMS algorithm
`fixPoint_leaky_lms.c`	C function for fixed-point leaky LMS algorithm
`fixPoint_leaky_lms.h`	C header file for experiment
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`input.pcm`	Reference input signal file
`desired.pcm`	Desired signal file

using fixed-point C with intrinsics. The reference and desired signals used for this experiment are the same as in the two previous experiments. Table 6.5 lists the ETSI operators and the corresponding intrinsics.

The fixed-point C implementation of the normalized LMS algorithm using the ETSI operators is listed in Table 6.6. Table 6.7 lists the files used for the experiment.

Table 6.5 ETSI functions and corresponding intrinsic functions

ETSI functions	Intrinsics representation	Description
`L_add(a,b)`	`_lsadd((a),(b))`	Add two 32-bit integers with SATD set, producing a saturated 32-bit result
`L_sub(a,b)`	`_lssub((a),(b))`	Subtract `b` from `a` with SATD set, producing a saturated 32-bit result
`L_negate(a)`	`_lsneg(a)`	Negate the 32-bit value with saturation. `_lsneg`(0x80000000)=> 0x7FFFFFFF
`L_deposite_h(a)`	`(long)(a<<16)`	Deposit the 16-bit `a` into the 16 MSBs of a 32-bit output and the 16 LSBs of the output are zeros
`L_deposite_l(a)`	`(long)a`	Deposit the 16-bit `a` into the 16 LSBs of a 32-bit output and the 16 MSBs of the output are sign extended
`L_abs(a)`	`_labss((a))`	Create a saturated 32-bit absolute value. `_labss`(0x8000000) => 0x7FFFFFFF (SATD set)
`L_mult(a,b)`	`_lsmpy((a),(b))`	Multiply `a` and `b` and shift the result left by 1. Produce a saturated 32-bit result. (SATD and FRCT are set)
`L_mac(a,b,c)`	`_smac((a),(b),(c))`	Multiply `b` and `c`, shift the result left by 1, and add it to `a`. Produce a saturated 32-bit result. (SATD, SMUL, and FRCT are set)
`L_macNs(a,b,c)`	`L_add_c((a),L_mult((b),(c)))`	Multiply `b` and `c`, shift the result left by 1, and add the 32-bit result to `a` without saturation
`L_msu(a,b,c)`	`_smas((a),(b),(c))`	Multiply `b` and `c`, shift the result left by 1, and subtract it from `a`. Produce a 32-bit result. (SATD, SMUL, and FRCT are set)
`L_msuNs(a,b,c)`	`L_sub_c((a),L_mult((b),(c)))`	Multiply `b` and `c`, shift the result left by 1, and subtract it from `a` without saturation
`L_shl(a,b)`	`_lsshl((a),(b))`	Shift `a` left by `b` and produce a 32-bit result. The result is saturated if `b` is less than or equal to 8. (SATD set)
`L_shr(a,b)`	`_lshrs((a),(b))`	Shift `a` right by `b` and produce a 32-bit result. Produce a saturated 32-bit result. (SATD set)
`L_shr_r(a,b)`	`L_crshft_r((a),(b))`	Same as `L_shr(a,b)` but with rounding
`abs_s(a)`	`_abss(a)`	Create a saturated 16-bit absolute value. `_abss`(0x8000)=>0x7FFF (SATA set)
`add(a,b)`	`_sadd((a),(b))`	Add two 16-bit integers with SATA set, producing a saturated 16-bit result.

(*continued*)

Table 6.5 (*Continued*)

ETSI functions	Intrinsics representation	Description
`sub(a,b)`	`_ssub((a),(b))`	Subtract `b` from `a` with SATA set, producing a saturated 16-bit result
`extract_h(a)`	`(unsigned short)((a)>>16)`	Extract the upper 16 bits of the 32-bit `a`
`extract_l(a)`	`(short)a`	Extract the lower 16 bits of the 32-bit `a`
`round(a)`	`(short)_rnd(a)>>16`	Round `a` by adding 2^{15}. Produce a 16-bit saturated result. (SATD set)
`mac_r(a,b,c)`	`(short)(_smacr((a),(b),(c))>>16)`	Multiply `b` and `c`, shift the result left by 1, add the result to `a`, and then round the result by adding 2^{15}. (SATD, SMUL, and FRCT are set)
`msu_r(a,b,c)`	`(short)(_smasr((a),(b),(c))>>16)`	Multiply `b` and `c`, shift the result left by 1, subtract the result from `a`, and then round the result by adding 2^{15}. (SATD, SMUL, and FRCT set)
`mult_r(a,b)`	`(short)(_smpyr((a),(b))>>16)`	Multiply `a` and `b`, shift the result left by 1, and round by adding 2^{15} to the result. (SATD and FRCT are set)
`mult(a,b)`	`_smpy((a),b))`	Multiply `a` and `b` and shift the result left by 1. Produce a saturated 16-bit result. (SATD and FRCT are set)
`norm_l(a)`	`_lnorm(a)`	Produce the number of left shifts needed to normalize `a`
`norm_s(a)`	`_norm(a)`	Produce the number of left shifts needed to normalize `a`
`negate(a)`	`_sneg(a)`	Negate the 16-bit value with saturation. `_sneg`(0xFFFF8000)=> 0x00007FFF
`shl(a,b)`	`_sshl((a),(b))`	Shift `a` left by `b` and produce a 16-bit result. The result is saturated if `b` is less than or equal to 8. (SATD set)
`shr(a,b)`	`_shrs((a),(b))`	Shift `a` right by `b` and produce a 16-bit result. Produce a saturated 16-bit result. (SATD set)
`shr_r(a,b)`	`crshft((a),(b))`	Same as `shr(a,b)` but with rounding
`shift_r(a,b)`	`shr_r((a),-(b))`	Same as `shl(a,b)` but with rounding
`div_s(a,b)`	`divs((a),(b))`	Produces a truncated positive 16-bit result which is the fractional integer division of `a` by `b`; `a` and `b` must be positive and `b` $\geq$ `a`

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment. Verify the experimental result using the methods given in step 2 of Exp6.1.
3. Profile the required clock cycles for key functions and compare to the fixed-point C results obtained in Exp6.2, which shows the effectiveness of using intrinsics.

Table 6.6 Normalized LMS algorithm using fixed-point C and intrinsics

```
void intrinsic_nlms(LMS *lmsObj)
{
     LMS     *lms=(LMS *)lmsObj;
     long    temp32;
     short   j,n,mu,ue,*x,*w;

     n = lms->order;
     w = &lms->w[0];
     x = &lms->x[0];

     // Update signal buffer
     for(j=n-1; j>0; j--)
     {
         x[j] = x[j-1];
     }
     x[0] = lms->in;
     // Compute normalized mu
     temp32 = mult_r(lms->x[0],lms->x[0]);
     temp32 = mult_r((short)temp32, ONE_MINUS_BETA);
     lms->power = mult_r(lms->power, BETA);
     temp32 = add(lms->power, (short)temp32);
     temp32 = add(lms->c, (short)temp32);
     mu = lms->alpha / (short)temp32;
     // Compute filter output
     temp32 = L_mult(w[0], x[0]);
     for(j=1; j<n; j++)
     {
         temp32 = L_mac(temp32, w[j], x[j]);
     }
     lms->out = round(temp32);
     // Compute error signal
     lms->err = sub(lms->des, lms->out);
     // Update filter coefficients
     ue = mult_r(lms->err, mu);
     for(j=0; j<n; j++)
     {
         w[j] = add(w[j], mult_r(ue, x[j]));
     }
}
```

4. This experiment uses the same set of data files as the previous two experiments. Rerun all three experiments to plot the error signals and compare the results to answer the following questions:
   (a) Which experiment has the fastest convergence rate and why?
   (b) Which experiment has the lowest steady-state error level and why?

Table 6.7 File listing for the experiment Exp6.3

Files	Description
intrinsic_nlmsTest.c	Program for testing NLMS algorithm
intrinsic_nlms.c	C function for NLMS algorithm using ETSI operators
intrinsic_nlms.h	C header file for experiment
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
input.pcm	Reference input signal file
desired.pcm	Desired signal file

6.6.4 *Delayed LMS Algorithm Using Assembly Program*

The TMS320C55xx provides the assembly instruction, `LMS`, for implementing the delayed LMS algorithm. This instruction utilizes the high parallelism of the C55xx architecture to perform the following two adaptive filtering equations (based on one coefficient) in one cycle:

$$y(n) = y(n) + w_l(n)x(n-l)$$
$$w_l(n+1) = w_l(n) + \mu e(n-1)x(n-l).$$

This `LMS` instruction computes the sum of one product and updates the same coefficient $w_l(n)$ using the same signal sample $x(n-l)$ and previous error $e(n-1)$, thus the `LMS` instruction effectively improves the runtime efficiency of the delayed LMS algorithm. Note that these two equations must be repeated L times to obtain the final filter output $y(n)$ and update all filter coefficients.

This experiment is written using the block processing method. The nested repeat loops are placed in the instruction buffer using the `repeatlocal` instruction, which further improves the real-time efficiency of the delayed LMS algorithm. The files used for the experiment are listed in Table 6.8.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment. Verify the experimental result using the methods given in step 2 of Exp6.1.

Table 6.8 File listing for the experiment Exp6.4

Files	Description
asm_dlmsTest.c	Program for testing delayed LMS algorithm
asm_dlms.asm	Assembly function for delayed LMS algorithm
asm_dlms.h	C header file for experiment
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
input.pcm	Reference input signal file
desired.pcm	Desired signal file

3. Use both signal waveform and magnitude spectrum plots to compare the delayed LMS algorithm result to the results obtained from Exp6.1, Exp6.2, and Exp6.3.
4. Profile the cycles needed for the delayed LMS algorithm using the `LMS` instruction and compare the runtime efficiency to the profile results from Exp6.1, Exp6.2, and Exp6.3.
5. Explain why the `LMS` instruction must use the previous error signal $e(n-1)$.

6.6.5 Experiment of Adaptive System Identification

This section performs the adaptive system identification experiment using mixed C-and-assembly programs. The block diagram of adaptive system identification is illustrated in Figure 6.7, and its procedures can be summarized as follows:

1. Place the current excitation signal $x(n)$ into `x[0]` of the signal buffer.
2. Compute the output of the unknown plant to get the desired signal $d(n)$.
3. Compute the adaptive FIR filter output $y(n)$.
4. Calculate the error signal $e(n)$.
5. Update L filter coefficients using the LMS algorithm.
6. Update the signal buffer as

$$x(n-l-1) = x(n-l), \quad l = L-2, L-1, \ldots, 1, 0.$$

The adaptive system identification shown in Figure 6.7 can be implemented in C as follows:

```
// Simulate an unknown system
   x1[0]=input;                     // Get input signal x(n)
   d = 0.0;
   for (i=0; i<N1; i++)             // Compute d(n)
      d += (coef[i]*x1[i]);
   for (i=N1-1; i>0; i--)           // Update signal buffer
      x1[i] = x1[i-1];              // of unknown system
// Adaptive system identification operation
   x[0]=input;                      // Get input signal x(n)
   y = 0.0;
   for (i=0; i<N0; i++)             // Compute output signal y(n)
      y += (w[i]*x[i]);
   e = d - y;                       // Calculate error signal e(n)
   uen = twomu*e;                   // uen = mu*e(n)
   for (i=0; i<N0; i++)             // Update coefficients
      w[i]+= (uen*x[i]);
   for (i=N0-1; i>0; i--)           // Update signal buffer
      x[i] = x[i-1];                // of adaptive filter
```

The unknown system used for this experiment is an FIR filter with the filter coefficients given in the vector `plant[]`. The excitation signal $x(n)$ is zero-mean white noise. The unknown system's output $d(n)$ is used as the desired signal for the adaptive filter to compute the error signal, and the adaptive filter coefficients are stored in `w[i]`. The files used for the experiment are listed in Table 6.9.

Table 6.9 File listing for the experiment Exp6.5

Files	Description
system_identificaitonTest.c	Program for testing adaptive system identification
sysIdentification.asm	Assembly function for adaptive filter
unknowFirFilter.asm	Assembly function for unknown FIR filter
system_identify.h	C header file for experiment
tistdtypes.h	Standard type define header file
unknow_plant.dat	Include file for unknown FIR system coefficients
c5505.cmd	Linker command file
x.pcm	Excitation input signal file

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment. Verify the experimental result by examining the error signal $e(n)$, which should be minimized after convergence of the LMS algorithm. Also, compare the final weight vector in `w[]` to the coefficient vector to be identified in `plant[]`.
3. Replace the assembly function, `sysIdentification.asm`, by the fixed-point C program using the operators given in Table 6.5. Redo the experiment and verify it by comparing to the results from step 2.
4. If the unknown plant (filter) is an IIR filter or the order of the unknown FIR filter is not available, the adaptive system identification using an adaptive FIR filter may not achieve a perfect model. Modify the program to perform the following experiments:
   (a) Change the adaptive FIR filter lengths to `N0` = 32 and `N0` = 64.
   (b) Use MATLAB® to design a simple IIR filter as the unknown plant. Redo the experiment using the adaptive FIR filter with different lengths.

6.6.6 Experiment of Adaptive Predictor

This experiment uses the block processing method to implement the leaky LMS algorithm for adaptive prediction. The input signal $x(n)$ consists of both the narrowband and broadband components. After the adaptive algorithm has converged, the adaptive filter output $y(n)$ will estimate the narrowband signal $s(n)$, and the error signal $e(n)$ will approximate the broadband signal $v(n)$. As discussed in Section 6.5.2, an adaptive predictor can be used to enhance the narrowband signal corrupted by broadband noise, or to cancel narrowband interference in the broadband signal. In this experiment, two different input signal files are used for these two applications.

For applications such as spread spectrum communications, the narrowband interference can be tracked and removed by the adaptive predictor. In the experiment, the input signal file `speech.pcm` contains the desired speech (broadband signal) corrupted by 120 Hz tonal interference (narrowband noise). For applications such as adaptive line enhancements, the adaptive filter forms a tunable bandpass filter to enhance the narrowband component at the filter output. In the experiment, the delay is set as $\Delta = 1$ and the input signal file `tone.pcm` contains a 1000 Hz tone and white noise. Table 6.10 lists the files used for the experiment.

Table 6.10 File listing for the experiment Exp6.6

Files	Description
`adaptive_predictorTest.c`	Program for testing adaptive predictor
`adaptivePredictor.asm`	Assembly function for adaptive predictor
`adaptive_predictor.h`	C header file for experiment
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`speech.pcm`	Data file containing 120 Hz sine wave and speech
`tone.pcm`	Data file containing 1000 Hz tone and white noise

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment using the data file `speech.pcm` for the experiment. Listen to the error signal (the enhanced speech in `e[]` is saved to the output file `wideband.pcm`) and compare it to the input signal file `speech.pcm`. Observe that the narrowband humming noise has been reduced. Also use the MATLAB® function `spectrogram` to compare spectrograms of the input and error signals.
3. Rewrite the adaptive predictor in fixed-point C. Redo step 2 using the provided input signal file `speech2.pcm` (containing speech and 1000 Hz sine wave interference). Evaluate the adaptive predictor result. Has the 1000 Hz tonal interference been reduced?
4. Modify the program by using a longer delay Δ (> 1) for the adaptive predictor and repeat steps 2 and 3. Observe the improvement in performance by using longer delay units.
5. Redo the experiment using the input signal file `tone.pcm` to verify the enhancement of the narrowband signal. Listen to the adaptive filter output in `y[]` saved to the output file `narrowband.pcm` and compare it to the input file `tone.pcm`. Also use the MATLAB® function `spectrogram` to evaluate the achieved noise reduction.
6. Use the fixed-point C adaptive predictor written for step 3, repeat step 5 using different delay units, and observe that a delay of $\Delta = 1$ is enough for decorrelating white noise.
7. The values of step size and leaky factor are defined in the assembly file `adaptivePre-dictor.asm`. The default leaky factor is set to 0x7FFE and the step size is 0x80. Conduct the following tasks:
   (a) Change the leaky factor value and observe the adaptive filter performance. Can the leaky factor be set to 0x7FFF?
   (b) Use different step size values and find the best value for the experiment.
8. Design new experiments for the following tasks:
   (a) Use the NLMS algorithm to replace the leaky LMS algorithm.
   (b) Implement the NLMS algorithm using the operators listed in Table 6.5.

6.6.7 Experiment of Adaptive Channel Equalizer

This experiment implements the adaptive equalizer using the complex LMS algorithm for a simplified ITU V.29 FAX modem [20]. According to the V.29 recommendation, the modem operates at speeds up to 9600 bits per second on the general switched telephone network lines.

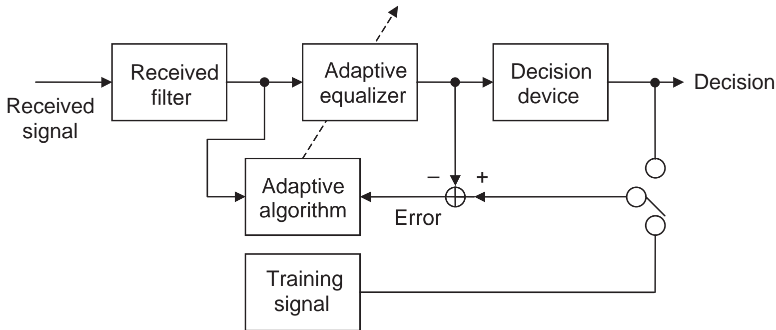


Figure 6.16 Simplified block diagram of V.29 modem with adaptive channel equalizer

The equalizer for modems can be realized using the adaptive FIR filter. In the absence of noise and intersymbol interference, the modem receiver decision logic output can precisely match the transmitted symbols and the error signal will converge to zero. Figure 6.16 shows the block diagram of the adaptive channel equalizer for the simplified V.29 modem, which uses a complex adaptive equalizer.

The decision-directed equalizer is only effective in tracking slow variations of channel. For this reason, the V.29 recommendation calls for force training using the predefined sequence of two symbols. These symbols are ordered according to the following random number generator:

$$1 \oplus x^{-6} \oplus x^{-7},$$

where $\oplus$ represents the exclusive-OR operation, and x^{-6} and x^{-6} are the sixth and seventh bits of the shift register of the pseudo-random binary sequence generator (to be introduced in Section 7.2.2). When the generated random number is zero, the constellation point (3, 0) will be transmitted. When the random number is one, the point (−3, 3) will be transmitted. In the receiver, the local generator will use the same generator with the same initial value to recreate the identical sequence and use it as the desired signal $d(n)$ for computing the error signal. The V.29 force training sequence consists of 384 symbols. This experiment shows the force training process for the adaptive equalizer that satisfies the simplified V.29 requirements. The files used for the experiment are listed in Table 6.11.

Table 6.11 File listing for the experiment Exp6.7

Files	Description
`channel_equalizerTest.c`	Program for testing adaptive equalizer
`adaptiveEQ.c`	C function for implementing adaptive equalizer
`channel.c`	C function simulates communication channel
`signalGen.c`	C function generates training sequence
`complexEQ.h`	C header file for experiment
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file

Table 6.12 File listing for the experiment Exp6.8

Files	Description
realtime_predictorTest.c	Program for testing adaptive line enhancer
adaptive_predictor.c	C function manages real-time audio process
adaptivePredictor.asm	Assembly function for adaptive line enhancer
vector.asm	Vector table for real-time experiment
adaptive_predictor.h	C header file for experiment
tistdtypes.h	Standard type define header file
dma.h	Header file for DMA functions
dmaBuff.h	Header file for DMA data buffer
i2s.h	i2s header file for i2s functions
Ipva200.inc	C5505 include file
myC55xxUtil.lib	BIOS audio library
c5505.cmd	Linker command file
tone_1khz.wav	Data file – narrowband signal corrupted by wideband noise
speech.wav	Data file – wideband signal corrupted by narrowband noise

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment. Use the CCS graphic tool to plot the error signal.
3. Modify the experiment based on different techniques and algorithms introduced in this chapter, and repeat the experiments to find better methods to improve the performance of the adaptive equalizer.

6.6.8 Real-Time Adaptive Prediction Using eZdsp

This experiment modifies the adaptive predictor implemented in Section 6.6.6 for real-time demonstration using the C5505 eZdsp. Similar to Exp6.6, this experiment uses two signal files: `tone_1khz.wav` is a tonal signal corrupted by wideband noise; and `speech.wav` is a wideband speech corrupted by a 120 Hz tone. These data files can be played via an audio player that supports WAV file format. The eZdsp digitizes the input signal from the audio player, processes the audio signal, and sends the enhanced output signal to a headphone (or loudspeaker) for playback. The files used for the experiment are listed in Table 6.12.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Connect an audio player (as the input) and a headphone (as the output) to the eZdsp audio input and output jacks for the experiment.
3. Load and run the experiment. Set the playback in loop mode to play the WAV file continuously. Adjust the audio player's volume to make sure that the eZdsp will have a proper dynamic range for digitizing the audio input for the experiment.
4. Listen to the processed result using the headphone. Note that the output signal is $y(n)$ when the input signal file is `tone_1khz.wav`, but the output signal is $e(n)$ when using the input signal from the file `speech.wav`. Verify that the noise will be reduced.
5. Replace the leaky LMS algorithm written in assembly functions by the NLMS algorithm implemented using the fixed-point C program with intrinsics. Repeat the real-time experiment given in step 4 to verify the result.

Exercises

6.1. Derive the autocorrelation functions of the following signals:
(a) $x(n) = A\ \sin(2\pi n/N)$.
(b) $y(n) = A\ \cos(2\pi n/N)$.

6.2. Find the cross-correlation functions, $r_{xy}(k)$ and $r_{yx}(k)$, for the $x(n)$ and $y(n)$ defined in Problem 1.

6.3. Let $x(n)$ and $y(n)$ be two independent zero-mean WSS random signals with variances σ_x^2 and σ_y^2, respectively. The random signal $w(n)$ is obtained as

$$w(n) = ax(n) + by(n),$$

where a and b are constants. Express $r_{ww}(k)$, $r_{wx}(k)$, and $r_{wy}(k)$ in terms of $r_{xx}(k)$ and $r_{yy}(k)$.

6.4. Similar to Example 6.6, the desired signal $d(n)$ is the output of an FIR filter with coefficient vector [0.2 0.5 0.3], where the input $x(n)$ is a zero-mean, unit-variance white noise. This white noise is also used as the reference input signal for the adaptive FIR filter with $L = 3$ using the LMS algorithm. Derive the error function in terms of $\mathbf{R}$ and $\mathbf{p}$, and compute the optimum solution $\mathbf{w}^{\mathrm{o}}$ and the minimum MSE $\xi_{\min}$. Use MATLAB® to verify the results by plotting (a) $e^2(n)$ vs. n, and (b) $w_0(n)$, $w_1(n)$, and $w_2(n)$ vs. n in a single plot. Note that this plot shows the weight tracks of the adaptive filter, which should converge to 0.2, 0.5, and 0.3.

6.5. Consider a second-order autoregressive (AR) process defined by

$$d(n) = v(n) - a_1 d(n-1) - a_2 d(n-2),$$

where $v(n)$ is a white noise of zero mean and variance σ_v^2. This AR process is generated by filtering $v(n)$ using the second-order IIR filter $H(z)$.
(a) Derive the IIR filter transfer function $H(z)$.
(b) Consider the second-order optimum FIR filter shown in Figure 6.3. If the desired signal is $d(n)$, the reference input signal $x(n) = d(n-1)$. Find the optimum weight vector $\mathbf{w}^{\mathrm{o}}$ and the minimum MSE $\xi_{\min}$.

6.6. Given the two finite-length sequences

$$x(n) = \{1\ 3\ -2\ 1\ 2\ -1\ 4\ 4\ 2\}$$
$$y(n) = \{2\ -1\ 4\ 1\ -2\ 3\}$$

use the MATLAB® function `xcorr` to compute and plot the cross-correlation function $r_{xy}(k)$ and the autocorrelation functions $r_{xx}(k)$ and $r_{yy}(k)$.

6.7. Write a MATLAB® script to generate a signal of length 1024 defined as

$$x(n) = 0.8 \sin(\omega_0 n) + v(n),$$

where $\omega_0 = 0.1\pi$, and $v(n)$ is a zero-mean random noise with variance $\sigma_v^2 = 1$ (see Section 3.3 for details). Compute and plot $r_{xx}(k)$, $k = 0, 1, \ldots, 127$, using

MATLAB®. Explain this simulation result using the theoretical derivations given in Example 6.1.

6.8. Redo Example 6.6 by using $x(n)$ as input to the adaptive FIR filter ($L = 2$) with the LMS algorithm. Implement this adaptive filter using MATLAB®. Plot the error signal $e(n)$ and $e^2(n)$ vs. the time index n, and show that the adaptive weights converge to the derived optimum values. Modify the MATLAB® code using $L = 8$ and 32, and compare the results to $L = 2$.

6.9. Implement the adaptive system identification technique illustrated in Figure 6.7 using MATLAB®, or the C program on the eZdsp. The input signal is a zero-mean, unit-variance white noise. The unknown system is an IIR filter defined in Problem 6.5. Use different filter lengths L and step sizes μ to plot $e(n)$ and $e^2(n)$ vs. the time index n for these parameters. Find the optimum values that result in fast convergence and low excess MSE.

6.10. Implement the adaptive line enhancer illustrated in Figure 6.9 using MATLAB®, or the C program on the eZdsp. The desired signal is given by

$$x(n) = \sqrt{2}\sin(\omega n) + v(n),$$

where frequency $\omega = 0.2\pi$ and $v(n)$ is zero-mean white noise with unit variance. The decorrelation delay $\Delta = 1$. Plot both $e(n)$ and $y(n)$. Evaluate the convergence speed and steady-state error level for different parameters L and μ.

6.11. Implement the adaptive noise cancellation illustrated in Figure 6.11 using MATLAB®, or the C program on the eZdsp. The primary signal is given by

$$d(n) = \sin(\omega n) + 0.8v(n) + 1.2v(n-1) + 0.25v(n-2),$$

where $v(n)$ is defined by Problem 6.5. The reference signal is $v(n)$. Plot $e(n)$ for different values of L and μ.

6.12. Implement the single-frequency adaptive notch filter illustrated in Figure 6.13 using MATLAB®, or the C program on the eZdsp. The desired signal $d(n)$ is given in Problem 6.11, and $x(n)$ is given by

$$x(n) = \sqrt{2}\,\sin(\omega n).$$

Plot $e(n)$ and the magnitude response of the second-order FIR filter after convergence. Also, plot the weight tracks as defined in Problem 6.4.

6.13. Use MATLAB® to generate the primary signal $d(n) = 0.25\,\cos(2\pi n f_1/f_s) + 0.25\,\sin(2\pi n f_2/f_s)$ and the reference input signal $x(n) = 0.125\,\cos(2\pi n f_2/f_s)$, where f_s is the sampling frequency, and f_1 and f_2 are the frequencies of the desired signal and interference, respectively. Implement the adaptive noise canceler to remove the interference using MATLAB®. Verify the results by plotting the magnitude spectrum of both the primary signal and the error signal.

6.14. Implement the functions developed in Problem 6.13 for an eZdsp experiment. Perform the real-time experiment by connecting the primary input and reference input signals to

the eZdsp stereo input, where the left channel is the primary signal with interference and the right channel contains only the interference. Test the adaptive noise canceler in real time using the C5505 eZdsp by listening to the noise-reduced output signal using a headphone or a loudspeaker.

References

1. Alexander, S.T. (1986) *Adaptive Signal Processing*, Springer-Verlag, New York.
2. Bellanger, M. (1987) *Adaptive Digital Filters and Signal Analysis*, Marcel Dekker, New York.
3. Clarkson, P.M. (1993) *Optimal and Adaptive Signal Processing*, CRC Press, Boca Raton, FL.
4. Cowan, C.F.N. and Grant, P.M. (1985) *Adaptive Filters*, Prentice Hall, Englewood Cliffs, NJ.
5. Glover, J.R. Jr. (1977) Adaptive noise canceling applied to sinusoidal interferences. *IEEE Trans. Acoust., Speech, Signal Process.*, **ASSP-25**, 484–491.
6. Haykin, S. (1991) *Adaptive Filter Theory*, 2nd edn, Prentice Hall, Englewood Cliffs, NJ.
7. Kuo, S.M. and Morgan, D.R. (1996) *Active Noise Control Systems – Algorithms and DSP Implementations*, John Wiley & Sons, Inc., New York.
8. Treichler, J.R., Johnson, C.R. Jr., and Larimore, M.G. (1987) *Theory and Design of Adaptive Filters*, John Wiley & Sons, Inc., New York.
9. Widrow, B. and Stearns, S.D. (1985) *Adaptive Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
10. Honig, M.L. and Messerschmitt, D.G. (1986) *Adaptive Filters: Structures, Algorithms, and Applications*, Kluwer Academic, Boston, MA.
11. The MathWorks, Inc. (2000) Using MATLAB, Version 6.
12. Kuo, S.M. and Chen, C. (1990) Implementation of adaptive filters with the TMS320C25 or the TMS320C30, in *Digital Signal Processing Applications with the TMS320 Family*, vol. **3** (ed. P. Papamichalis), Prentice Hall, Englewood Cliffs, NJ, pp. 191–271, Chapter 7.
13. The MathWorks, Inc. (2004) Signal Processing Toolbox User's Guide, Version 6.
14. The MathWorks, Inc. (2004) Filter Design Toolbox User's Guide, Version 3.
15. The MathWorks, Inc. (2004) Fixed-Point Toolbox User's Guide, Version 1.
16. Ljung, L. (1987) *System Identification: Theory for the User*, Prentice Hall, Englewood Cliffs, NJ.
17. Makhoul, J. (1975) Linear prediction: a tutorial review. *Proc. IEEE*, **63**, 561–580.
18. Widrow, B., Glover, J.R., McCool, J.M. *et al*. (1975) Adaptive noise canceling: principles and applications. *Proc. IEEE*, **63**, 1692–1716.
19. The MathWorks, Inc. (2005) Communications Toolbox User's Guide, Version 3.
20. ITU Recommendation (1988) V.29, 9600 Bits Per Second Modem Standardized for Use on Point-to-Point 4-Wire Leased Telephone-Type Circuits, November.

7

Digital Signal Generation and Detection

Digital signal generation and detection are useful techniques for DSP system design, analysis, and real-world applications [1,2]. For example, dual-tone multi-frequency (DTMF) generation and detection are widely used in telephone signaling and interactive control applications through public switched telephone networks and cellular networks [3]. This chapter introduces several techniques for the generation of digital signals such as sine wave and random noise, DTMF generation and detection, and some practical applications.

7.1 Sine Wave Generators

There are several characteristics that should be considered when designing algorithms for generating digital sinusoidal signals. These issues include total harmonic distortion, frequency and phase control, memory usage, computational complexity, and waveform accuracy.

Some trigonometric functions can be approximated by polynomials; for example, the generation of cosine and sine signals using polynomial approximation is presented in Section 2.6 and used for experiments in Section 2.6.2. In addition, sine wave generation using resonators realized as IIR filters is presented in Chapter 4. These polynomial approximations and IIR filters are realized using multiplications and additions. This section discusses only the most efficient lookup table method for generating sine wave and chirp signals.

7.1.1 Lookup Table Method

The lookup table (or table lookup) method is probably the most simple and flexible technique for generating periodic waveforms. This technique reads a series of stored data samples that represent one period of the waveform. These values can be obtained either by sampling of analog signals or by computation using mathematical algorithms.

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

A sine wave table containing one period of the waveform can be obtained by computing the following function:

$$x(n) = \sin\left(\frac{2\pi n}{N}\right), \quad n = 0, 1, \ldots, N-1. \tag{7.1}$$

The digital samples are represented in binary form in digital hardware, thus the accuracy is determined by the wordlength used for representing these numbers. The desired sine wave can be generated by reading these stored samples from the table at a constant step Δ. The data pointer wraps around at the end of the table for generating a waveform with length longer than one period. The frequency of the generated sine wave is determined by the sampling rate f_s, table length N, and the table address increment Δ as

$$f = f_s \frac{\Delta}{N} \text{ Hz}. \tag{7.2}$$

For a given sine wave table of length N, the sine wave with frequency f and sampling rate f_s can be generated using the following address pointer increment (or step)

$$\Delta = \frac{Nf}{f_s}, \tag{7.3}$$

with the following constraint to avoid aliasing

$$\Delta \leq \frac{N}{2}. \tag{7.4}$$

To generate L samples of sine wave $x(l)$, $l = 0, 1, \ldots, L-1$, we can use a circular pointer k as the address pointer for the table that is updated as

$$k = (l\Delta + m)_{\text{mod } N}, \tag{7.5}$$

where m determines the initial phase of the sine wave and the modulo operation (mod) returns the remainder after division by N. It is important to note that the step Δ given in (7.3) may not be an integer, thus k computed by (7.5) may be a real number. An easy solution is to round the non-integer index k to the nearest integer. A better but more complex method is to interpolate the value based on the adjacent samples in the existing table.

The following two errors will cause harmonic distortion:

1. Amplitude quantization errors due to the use of a finite wordlength to represent sample values in the table.
2. Time quantization errors caused by synthesizing sample values between adjacent table entries.

Increasing table length N can reduce the time quantization errors. To reduce the memory requirement, we can take advantage of the symmetry property of the sine function since the absolute values of a sine wave repeat four times in each period. Thus, only one-fourth of the

period is required. However, a more complex algorithm is needed to track which quadrant of the waveform is generated in order to determine the correct value and sign of data.

To decrease the harmonic distortion for a given table length, an interpolation technique can be used to compute the values between table entries when the address index k computed from (7.5) is a non-integer number. Simple linear interpolation assumes the value between two consecutive table entries lies on the straight line between these two values. Suppose the integer part of the pointer k is i $(0 \leq i < N)$ and the fractional part is f $(0 < f < 1)$. Then the interpolated value can be computed as

$$x(n) = s(i) + f[s(i+1) - s(i)], \tag{7.6}$$

where $[s(i+1) - s(i)]$ is the slope of the line between successive table entries $s(i)$ and $s(i+1)$.

Example 7.1

MATLAB® program `example7_1.m` is used to generate one period of a 200 Hz sine wave with a sampling rate of 4 kHz as shown in Figure 7.1. These 20 samples are stored in a table for generating the sine wave at different frequencies under $f_s = 4$ kHz. From (7.3), $\Delta = 1$ (i.e., read every sample in the table) is used to generate the 200 Hz sine wave, and $\Delta = 2$ (read every other sample in the table) is used for generating the 400 Hz sine wave. In order to generate the 300 Hz sine wave, $\Delta = 1.5$ is needed, but some samples are not directly stored in the table and must be computed from adjacent samples.

From Figure 7.1, when we access the lookup table with $\Delta = 1.5$ we get the first value, which is the first entry in the table. However, the second value is not available in the table since it is between the second and third entries. Therefore, linear interpolation results in the average of these two values. To generate the 250 Hz sine wave, $\Delta = 1.25$, and we can use (7.6) for computing sample values with a non-integer index.

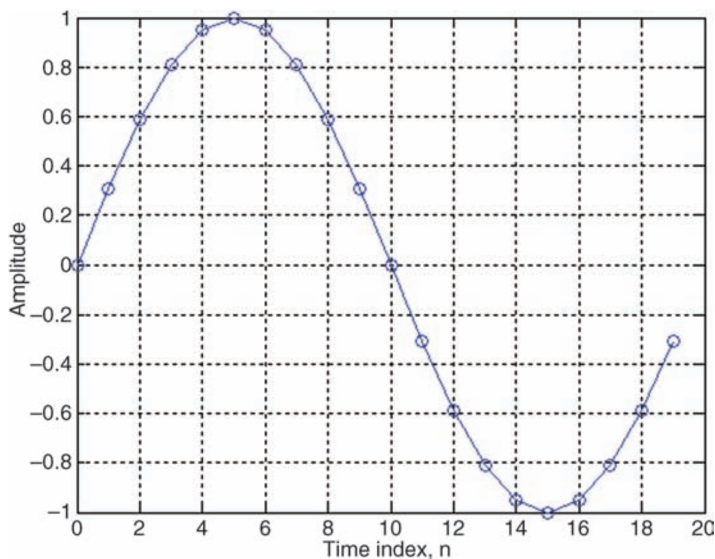


Figure 7.1 One period of sine wave, where stored samples are marked by ○

7.1.2 Linear Chirp Signal

A linear chirp signal is a waveform whose instantaneous frequency changes linearly with time between two specified frequencies. It is a waveform with the lowest possible peak to root-mean-square amplitude ratio in the desired frequency band. The digital chirp waveform can be expressed as

$$c(n) = A\sin[\phi(n)], \tag{7.7}$$

where A is a constant amplitude and $\phi(n)$ is a time-varying quadratic phase in the form of

$$\phi(n) = 2\pi\left[f_L n + \left(\frac{f_U - f_L}{2(N-1)}\right)n^2\right] + \alpha, \quad 0 \le n \le N-1, \tag{7.8}$$

where N is the total number of points in a single chirp. In (7.8), α is an arbitrary constant phase factor, and f_L and f_U are the normalized lower and upper frequency limits, respectively. The waveform periodically repeats with

$$\phi(n+kN) = \phi(n), \quad k = 1, 2, \ldots. \tag{7.9}$$

The instantaneous normalized frequency is defined as

$$f(n) = f_L + \left(\frac{f_U - f_L}{N-1}\right)n, \quad 0 \le n \le N-1. \tag{7.10}$$

This expression shows that the instantaneous frequency goes from $f(0) = f_L$ at time $n = 0$ to $f(N-1) = f_U$ at time $n = N-1$.

Because of the complexity of the linear chirp signal generator, it is more convenient to generate the chirp sequence by computer and store it in a lookup table for real-time applications. The lookup table method introduced in Section 7.1.1 can be used to generate the desired chirp signal using the stored table.

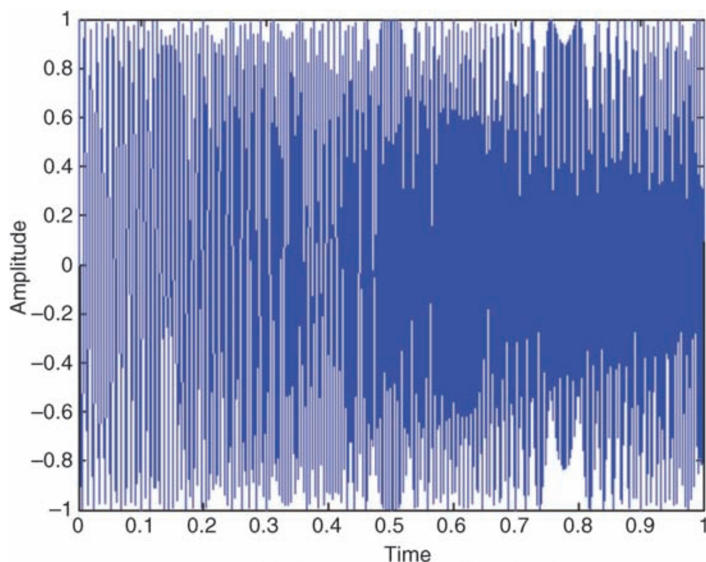
MATLAB® [4–6] provides the function `chirp(t, f0, t1, f1)` for generating a linear chirp signal at the time instances defined in the array `t`, where `f0` is the starting frequency at time 0 and `f1` is the ending frequency at time `t1`. The unit used for the variables `f0` and `f1` is hertz (Hz).

Example 7.2

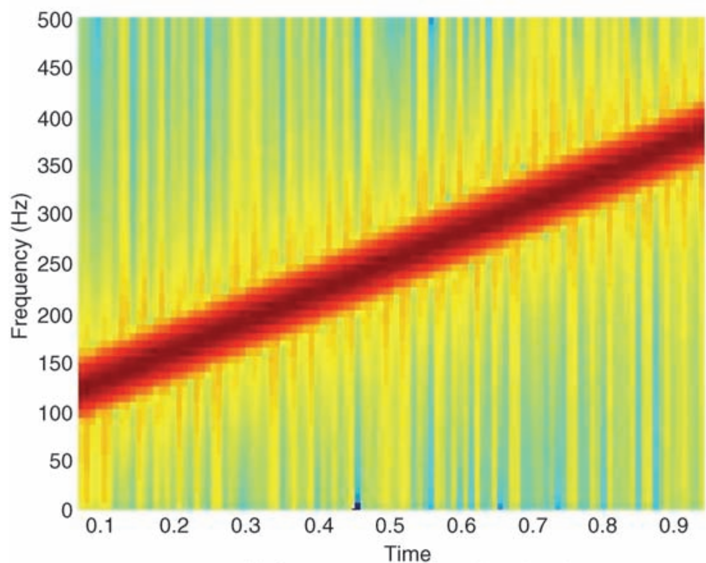
Compute the chirp signal with the sampling rate 1000 Hz. The signal sweeps from 100 to 400 Hz in 1 second. Part of the MATLAB® code for generating the chirp signal (`example7_2.m`, adapted from the MATLAB® **Help** menu) is listed as follows:

```
Fs = 1000;                  % Sampling rate
T = 1/Fs;                   % Sampling period
t = 0:T:1;                  % 1 second at 1 kHz sample rate
y = chirp(t,100,1,400);% From 100 Hz to 400 Hz in 1 second
spectrogram(y,128,120,128,Fs,'yaxis');% Spectrogram
```

The waveform and spectrogram of the generated chirp signal are illustrated in Figure 7.2.



(a) Waveform of the chirp signal.



(b) Spectrogram of the chirp signal.

Figure 7.2 Chirp signal from 100 to 400 Hz in 1 second.

A useful application of the chirp signal generator is to generate sirens. Electronic sirens are often produced by a generator inside the vehicle compartment. This generator drives a loudspeaker in the light bar mounted on the roof of the vehicle. The actual siren characteristics (bandwidth and duration) vary slightly among manufacturers. The wail type of siren sweeps between 800 to 1700 Hz with a sweep period of approximately 4.92 seconds. The yelp siren has similar characteristics but the period is 0.32 seconds.

Example 7.3

The chirp signal generator given in Example 7.2 is modified for generating sirens. The MATLAB® code `example7_3.m` generates a wail-type siren signal and plays it using the `soundsc` function.

7.2 Noise Generators

Random numbers are used in many practical applications such as adaptive system identification and adaptive channel equalizers discussed in Chapter 6. Although we cannot produce perfect random numbers by using digital hardware, it is possible to generate a sequence of numbers that are unrelated to each other, such as the white noise introduced in Section 6.1. Such numbers are called pseudo-random numbers. This section introduces some random number generation algorithms.

7.2.1 Linear Congruential Sequence Generator

The linear congruential method is widely used by many random number generators, and can be expressed as [7,8]

$$x(n) = [ax(n-1) + b]_{\mathrm{mod}\ M}, \tag{7.11}$$

where the mod operation returns the remainder after division by M. The constants a, b, and M can be chosen as

$$a = 4K + 1, \tag{7.12}$$

where K is an odd number such that a is less than M, and

$$M = 2^L \tag{7.13}$$

is a power-of-2 integer, and b can be any odd number. Good choices of parameter values are $M = 2^{20} = 1\,048\,576$, $a = 4 \cdot (511) + 1 = 2045$, and $x(0) = 12\,357$, where the initial condition $x(0)$ functions as a seed. The period of the sequence given by (7.11) has the full length of M.

Since some applications require real-valued random samples between 0 and 1, we can normalize the integer random samples $x(n)$ from (7.11) as

$$r(n) = \frac{x(n) + 1}{M + 1} \tag{7.14}$$

Table 7.1 C program for generating a linear congruential sequence

```
/*
* URAN - Generation of floating-point pseudo-random numbers
*/
static long n=(long)12357; // Seed x(0) = 12357
float uran()
{
float ran;                    // Random noise r(n)
n=(long)2045*n+1L;            // x(n)=2045*x(n-1)+1
n-=(n/1048576L)*1048576L;//x(n)=x(n)-INT[x(n)/1048576]*1048576
ran=(float)(n+1L)/(float)1048577; //r(n)=FLOAT[x(n)+1]/1048577
return(ran);                  // Return the generated random number
}
```

to produce the real-valued random samples $r(n)$. A floating-point C function (`uran.c`) that implements the random number generator defined by (7.11) and (7.14) is listed in Table 7.1. A fixed-point C function (`rand.c`) that is more efficient for implementation on fixed-point digital signal processors was introduced in Section 2.6.3. Also, as introduced in Chapters 2 and 6, MATLAB® provides the function `rand` with some variations to generate uniformly distributed pseudo-random numbers.

7.2.2 *Pseudo-random Binary Sequence Generator*

A shift register with feedback from specific bit locations can also generate a repetitive pseudo-random sequence. The schematic of a 16-bit generator is shown in Figure 7.3, where the functional operator labeled "XOR" performs the exclusive-OR operation on its two binary inputs. The sequence itself is determined by the position of the feedback bits in the shift register. In Figure 7.3, x_1 is the output of b_0 XOR with b_2, x_2 is the output of b_{11} XOR with b_{15}, and x is the output of x_1 XOR with x_2.

Each output sample from the sequence generator is the entire 16-bit content of the register. After the random number is generated, every bit in the register is shifted left by 1 bit (b_{15} is

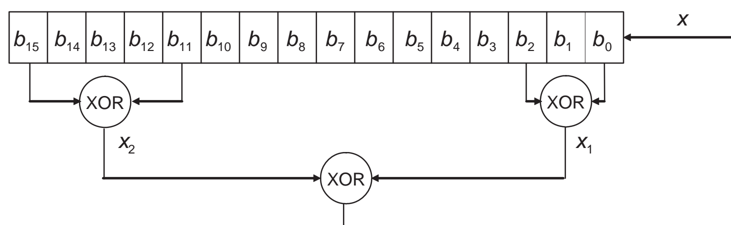


Figure 7.3 A 16-bit pseudo-random number generator

lost), and then x is shifted into the b_0 position. A shift register of length 16 bits can readily be accommodated by a single word on 16-bit processors. It is important to recognize, however, that sequential words formed by this process will be correlated. The maximum sequence length before repetition is

$$L = 2^M - 1, \tag{7.15}$$

where M is the number of bits in the shift register.

7.2.3 White, Color, and Gaussian Noise

We introduced the white noise $v(n)$ in Example 6.1 as a WSS noise that has the constant power density spectrum as defined in (6.8), and its autocorrelation function is given in (6.7). This kind of noise is called "white noise" because it contains all frequencies in its spectrum as white light. One of the important applications of white noise is as an excitation signal $x(n)$ for adaptive system identification as shown in Figure 6.7. The MATLAB® code `example2_19.m` generates the zero-mean, unit-variance white noise using the function `rand`.

If the white noise is filtered by a bandpass filter with predetermined bandwidth and center frequency, the output signal is called bandlimited white noise. This kind of noise is also called "colored noise" since it has a portion of the frequencies in its spectrum as colored light. Colored noise is widely used as a testing signal for the evaluation of DSP algorithms.

Similarly, if the white noise is filtered by a high-order FIR filter, the output signal consists of a large number of weighted sums of white noise that tend to approach Gaussian noise. One of the most important applications of Gaussian noise is to model the effects of "thermal" noise in electronic circuits. MATLAB® provides the `wgn` function for generating white Gaussian noise (WGN), which is widely used for modeling communication channels.

We can specify the power of the noise in dBW (decibels relative to 1 watt), dBm, or linear units, see Appendix A for details. We can generate either real or complex noise. For example, the MATLAB® command below generates a vector of length 50 containing real-valued WGN whose power is 2 dBW:

```
y1 = wgn(50,1,2);
```

The function assumes that the load impedance is 1 Ω (ohm).

Example 7.4

A WGN channel adds WGN to the signal that passes through it. To model a WGN channel, use the `awgn` function as follows:

```
y = awgn(x,snr)
```

This command adds WGN to the signal vector `x`. The scalar `snr` specifies the signal-to-noise ratio in dB. If `x` is complex, then `awgn` adds complex noise. This syntax assumes that the power of `x` is 0 dBW. The MATLAB® script (`example7_4.m`, adapted from the MATLAB® **Help** menu) adds WGN to a square wave signal. It then plots the original and noisy signals.

7.3 DTMF Generation and Detection

DTMF generation and detection are widely used in telephone signaling and interactive control applications through telephone and cellular networks [9]. DTMF signaling was initially developed for telephony signaling such as dialing and automatic redial. DTMF is also used in interactive remote access control with computerized automatic response systems such as an airline's information systems, remote voice mailboxes, electronic banking systems, as well as many semi-automatic services via telephone networks. DTMF signaling scheme, reception, testing, and implementation requirements are defined in ITU Recommendations Q.23 [10] and Q.24 [11].

7.3.1 DTMF Generator

A common application of the sine wave generator is the touch-tone telephone or cell (mobile) phone that uses the DTMF transmitter and receiver. Each keypress on the keypad generates the sum of two tones expressed as

$$x(n) = \cos(2\pi f_{\mathrm{L}} nT) + \cos(2\pi f_{\mathrm{H}} nT), \tag{7.16}$$

where T is the sampling period and the two frequencies f_{L} and f_{H} uniquely define the key that was pressed. Figure 7.4 shows a 4×4 grid matrix of the frequencies used to encode the 16 DTMF symbols defined by ITU Recommendation Q.23. This matrix represents 16 DTMF signals including numbers 0–9, special keys $*$ and #, and four English letters A–D. The letters A–D are assigned to unique functions for special communication systems such as military telephony systems.

The low-frequency group (697, 770, 852, and 941 Hz) selects the row frequencies of the 4×4 keypad, and the high-frequency group (1209, 1336, 1477, and 1633 Hz) selects the column frequencies. The values of these eight frequencies have been chosen carefully so that they do not interfere with speech. A pair of sinusoidal signals with f_{L} from the low-frequency group and f_{H} from the high-frequency group will represent a particular key.

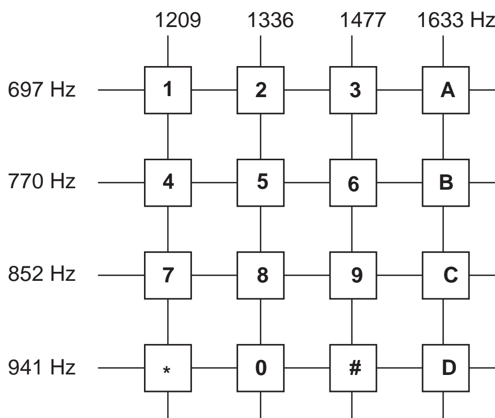


Figure 7.4 The 4×4 telephone keypad matrix

For example, the digit "3" is represented by two sine waves at the frequencies 697 and 1477 Hz.

The generation of dual tones can be implemented using two sine wave generators connected in parallel. The DTMF signal must meet timing requirements for the duration and spacing of digit tones. Digits are required to be transmitted at a rate of less than 10 per second. A minimum spacing of 50 ms between tones is required, and the tones must be presented for a minimum of 40 ms. The tone detection method used for the DTMF receiver must have sufficient time resolution to verify correct digit timing.

One common application of using DTMF is signaling for remote access control between individual users and an automated electronic database. In this example, the user follows the pre-recorded voice commands to key in the corresponding information, such as the account number and user authentication, using a touch-tone telephone keypad. The user's inputs are converted to a series of DTMF signals. The reception end processes these DTMF signals to reconstruct the digits for remote access control. The system with the automated electronic database sends the queries, responses, and confirmation messages via a voice channel to the user during the remote access process.

Besides DTMF, there are many other tones used in communications. For example, the call progress tones include dial tones, busy tones, ring-back tones, and modem and fax tones. The basic tone detection algorithm and implementation techniques [12–14] are similar. In this chapter, we focus on DTMF detection.

7.3.2 DTMF Detection

This section introduces techniques used for detecting DTMF signals in communication networks. Since the DTMF signaling may be used to set up a call and to control functions such as call forwarding or transferring, it is necessary to detect DTMF signaling in the presence of speech.

Goertzel Algorithm

The basic principle of DTMF detection is to examine the energy of the received signal for all eight frequencies defined in Figure 7.4, and to determine whether a valid DTMF tone pair exists. The detection algorithm can be implemented using the DFT or a filterbank. For example, an N-point DFT can calculate the energies of N evenly spaced frequency bins. To achieve the required frequency resolution to detect the DTMF frequencies within $\pm 1.5\%$, a 256-point FFT is needed for an 8 kHz sampling rate. Since DTMF detection only considers eight frequencies, it is more efficient to use a filterbank that consists of eight IIR bandpass filters. In this chapter, we will introduce the modified Goertzel algorithm as a filterbank for DTMF detection [14].

The DFT can be used to compute eight different $X(k)$ that correspond to the frequencies of DTMF signals as

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \tag{7.17}$$

where W_N^{kn} are the twiddle factors of the DFT defined in (5.15). Using the modified Goertzel algorithm, the DTMF decoder can be implemented as a matched filter for each frequency

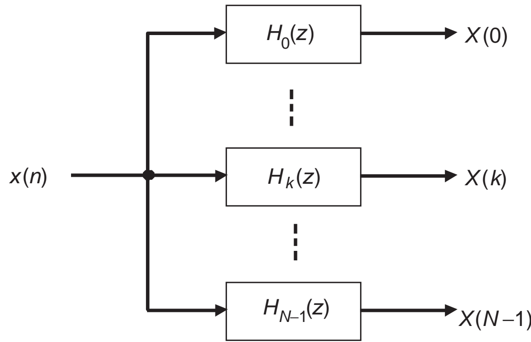


Figure 7.5 Block diagram of Goertzel filterbank

index k as illustrated in Figure 7.5, where $x(n)$ is the input signal, $H_k(z)$ is the transfer function of the kth filter, and $X(k)$ is the corresponding filter output.

From (5.15), we have

$$W_N^{-kN} = e^{j(2\pi/N)kN} = e^{j2\pi k} = 1. \tag{7.18}$$

Thus we can multiply the right-hand side of (7.17) by W_N^{-kN} as

$$X(k) = W_N^{-kN} \sum_{n=0}^{N-1} x(n) W_N^{kn} = \sum_{n=0}^{N-1} x(n) W_N^{-k(N-n)}. \tag{7.19}$$

We define the sequence

$$y_k(n) = \sum_{m=0}^{N-1} x(m) W_N^{-k(n-m)}, \tag{7.20}$$

which can be interpreted as the convolution between the finite-duration sequence $x(n)$ and the sequence $W_N^{-kn}u(n)$ for $0 \le n \le N-1$. Consequently, $y_k(n)$ can be viewed as the output of the filter with impulse response

$$h_k(n) = W_N^{-kn} u(n). \tag{7.21}$$

Thus, (7.20) can be expressed as

$$y_k(n) = x(n) * W_N^{-kn} u(n). \tag{7.22}$$

From (7.19) and (7.20), and the fact that $x(n) = 0$ for $n < 0$ and $n \ge N$, we can show that

$$X(k) = y_k(n)|_{n=N-1}. \tag{7.23}$$

That is, $X(k)$ is the output of the corresponding filter $H_k(z)$ at time $n = N - 1$.

Taking the z-transform of (7.22), we obtain

$$Y_k(z) = X(z)\frac{1}{1 - W_N^{-k}z^{-1}}. \tag{7.24}$$

The transfer function of the kth Goertzel filter is therefore defined as

$$H_k(z) = \frac{Y_k(z)}{X(z)} = \frac{1}{1 - W_N^{-k}z^{-1}}, \quad k = 0, 1, \ldots, N-1. \tag{7.25}$$

This filter has a pole on the unit circle at frequency $\omega_k = 2\pi k/N$. Thus, the DFT can be computed by filtering a block of input data using N filters in parallel as defined by (7.25). Each filter has a pole at the corresponding frequency of the DFT.

The parameter N must be chosen to ensure that $X(k)$ is the result representing the DTMF at frequency f_k that meets the requirement of frequency tolerance. The DTMF detection accuracy can only be ensured if N is chosen such that the following approximation is satisfied:

$$\frac{2f_k}{f_s} \cong \frac{k}{N}. \tag{7.26}$$

A block diagram of the transfer function $H_k(z)$ for the recursive computation of $X(k)$ is depicted in Figure 7.6. Since the coefficients W_N^{-k} are complex valued, the computation of each new value of $y_k(n)$ requires four multiplications and additions. All the intermediate values, $y_k(0), y_k(1), \ldots$, and $y_k(N-1)$, must be computed in order to obtain the final output $y_k(N-1) = X(k)$. Therefore, the computation of $X(k)$ given in Figure 7.6 requires $4N$ complex multiplications and additions for each frequency index k.

We can avoid the complex multiplications and additions by combining the pair of filters that have complex-conjugate poles. By multiplying both the numerator and denominator of $H_k(z)$ in (7.25) by the factor $(1 - W_N^k z^{-1})$, we have

$$\begin{aligned} H_k(z) &= \frac{1 - W_N^k z^{-1}}{(1 - W_N^{-k}z^{-1})(1 - W_N^k z^{-1})} \\ &= \frac{1 - e^{j2\pi k/N}z^{-1}}{1 - 2\cos(2\pi k/N)z^{-1} + z^{-2}}. \end{aligned} \tag{7.27}$$

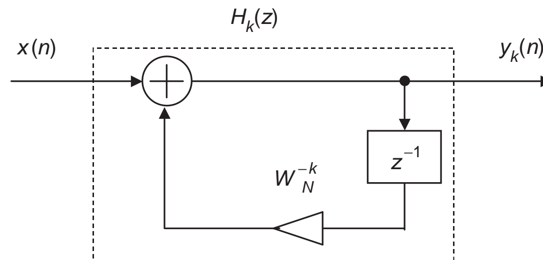


Figure 7.6 Block diagram of recursive computation of $X(k)$

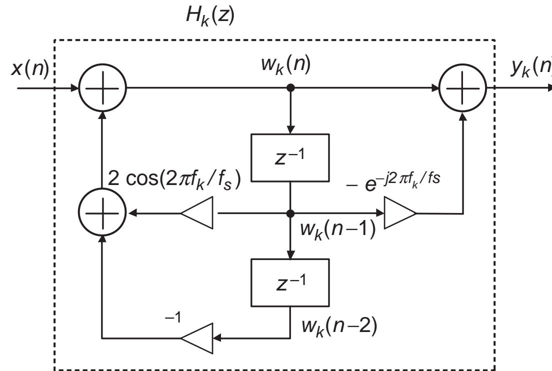


Figure 7.7 Detailed signal-flow diagram of Goertzel algorithm

This transfer function can be represented as the signal-flow diagram shown in Figure 7.7 using the direct-form II IIR filter. The recursive part of the filter is on the left, and the non-recursive part is on the right. Since the output $y_k(n)$ is required only at time $N-1$, we just need to compute the non-recursive part of the filter at the $(N-1)$th iteration. The recursive part of the algorithm can be expressed as

$$w_k(n) = x(n) + 2\cos(2\pi f_k/f_s)w_k(n-1) - w_k(n-2), \tag{7.28}$$

while the non-recursive calculation of $y_k(N-1)$ is expressed as

$$X(k) = y_k(N-1) = w_k(N-1) - e^{-j2\pi f_k/f_s}w_k(N-2). \tag{7.29}$$

The algorithm can be further simplified by realizing that only the squared $X(k)$ (magnitude) is needed for tone detections. From (7.29), the squared magnitude of $X(k)$ is computed as

$$|X(k)|^2 = w_k^2(N-1) - 2\cos(2\pi f_k/f_s)w_k(N-1)w_k(N-2) + w_k^2(N-2). \tag{7.30}$$

This equation avoids the complex arithmetic in (7.29), and requires only one coefficient, $2\cos(2\pi f_k/f_s)$, for computing each $|X(k)|^2$. Since there are eight possible tones, the detector needs eight filters as described by (7.28) and (7.30). Each filter is tuned to one of the eight frequencies. Note that (7.28) is computed for $n=0, 1, \ldots, N-1$, but (7.30) is computed only once at time $n=N-1$.

Implementation Considerations

The flowchart of the DTMF detection algorithm is illustrated in Figure 7.8. At the beginning of each frame, the state variables $x(n)$, $w_k(n)$, $w_k(n-1)$, $w_k(n-2)$, and $y_k(n)$ for each of the eight Goertzel filters and the energy are set to zero. For every new sample, the recursive part of filtering defined in (7.28) is executed. At the end of each frame, that is, $n=N-1$, the squared

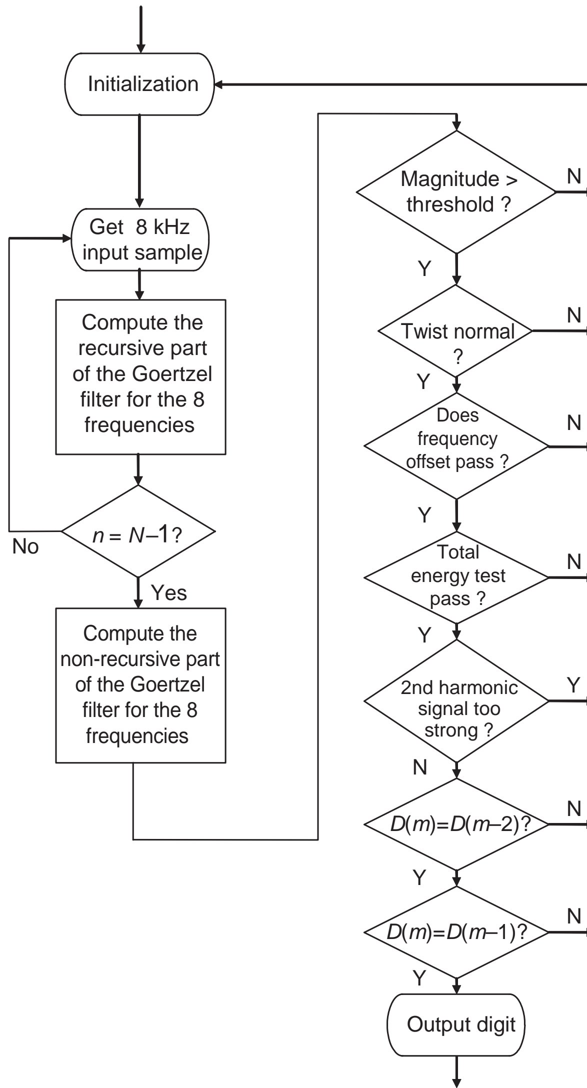


Figure 7.8 Flowchart for the DTMF detector

magnitude $|X(k)|^2$ for every DTMF frequency is computed based on (7.30). Six tests are followed to determine if a valid DTMF digit has been detected.

Magnitude Test

According to ITU Q.24, the maximum signal level transmitted to the public network shall not exceed -9 dBm. This limits the average voice range of -35 dBm for a very weak long-distance call to -10 dBm for a local call. A DTMF receiver is expected to operate at an average range of -29 to $+1$ dBm. Thus, the largest magnitude in each band must be greater

than the threshold of -29 dBm; otherwise, the DTMF signal would not be detected. For the magnitude test, the squared magnitude $|X(k)|^2$ defined in (7.30) for every DTMF frequency is computed. The largest magnitude in each group is obtained.

Twist Test

The tones may be attenuated according to the telephone system's gains at the tonal frequencies. Therefore, we do not expect the received tones to have the same amplitude, even though they may be transmitted with the same strength. Twist is defined as the difference, in decibels, between the low- and high-frequency tone levels. In practice, the DTMF digits are generated with forward twist to compensate for greater losses at higher frequency within a long telephone cable. For example, North America recommends not more than 8 dB of forward twist and 4 dB of reverse twist.

Frequency Offset Test

This test prevents some broadband signals, such as speech, from being incorrectly detected as DTMF tones. If the effective DTMF tones are present, the power levels at those two frequencies should be much higher than the power levels at the other frequencies. To perform this test, the largest magnitude in each group is compared to the magnitudes of other frequencies in that group. The difference must be greater than the predetermined threshold in each group.

Total Energy Test

Similar to the frequency offset test, the goal of the total energy test is to reject some broadband signals to further improve the robustness of the DTMF decoder. To perform this test, three different constants, c_1, c_2, and c_3, are used. The energy of the detected tone in the low-frequency group is weighted by c_1, the energy of the detected tone in the high-frequency group is weighted by c_2, and the sum of the two energies is weighted by c_3. Each of these terms must be greater than the summation of the energy from the rest of the filter outputs.

Second-Harmonic Test

The objective of this test is to reject speech that has harmonics close to f_k where they might be falsely detected as DTMF tones. Since DTMF tones are pure sinusoids, they contain very little second-harmonic energy. Speech, on the other hand, contains a significant amount of second harmonics. To test the level of second harmonics, the detector must evaluate the second-harmonic frequencies of all eight DTMF tones. These frequencies (1394, 1540, 1704, 1882, 2418, 2672, 2954, and 3266 Hz) also can be estimated using the Goertzel algorithm.

Digit Decoder

Finally, if all five tests are passed, the tone pair is decoded and mapped to one of the 16 keys on the telephone touch-tone keypad. This decoded digit is placed in the memory location designated as $D(m)$. If any of the tests fail, then "-1" representing "no detection" is placed in $D(m)$. For a new valid digit to be declared, the current $D(m)$ must be the same in three successive frames, that is, $D(m-2) = D(m-1) = D(m)$.

There are two reasons for checking three successive digits at each pass. First, the check eliminates the need to generate hits every time a tone is present. As long as the tone is present,

it can be ignored until it changes. Second, comparing digits $D(m-2)$, $D(m-1)$, and $D(m)$ improves noise and speech immunity.

7.4 Experiments and Program Examples

This section presents several experiments including real-time signal generation and DTMF generation and detection using the C5505 eZdsp.

7.4.1 Sine Wave Generator Using Table Lookup

This experiment uses the eZdsp, C5505 chip support library, and eZdsp board support library for generating sinusoidal signals. A similar experiment is given in Appendix C. Table 7.2 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Connect the eZdsp audio output to a headphone or a loudspeaker. Load and run the experiment to generate the tone. Verify the experimental result by listening to the eZdsp audio output.
3. Connect the eZdsp audio output to a digital recording device to digitize the tone in a PCM or WAV file. Compute the magnitude spectrum of the digital signal to verify that the tone generated in step 2 is a 1000 Hz sine wave.
4. In step 2, the eZdsp is set to an 8000 Hz sampling rate. Change the AIC3204 settings to run the experiment at the sampling rate of 48 000 Hz. Create a 400 Hz sine wave lookup table for the 48 000 Hz sampling rate. How many data samples are needed for this lookup table? Run the experiment and digitize the output signal. Examine the waveform and magnitude spectrum to verify that the signal is a 400 Hz sine wave at the 48 000 Hz sampling rate.
5. Modify the 400 Hz sine wave lookup table to contain only the data points from 0 to $\pi/2$. Modify the programs using the symmetric property of the sine wave to correctly use this quarter period of the lookup table to generate the 400 Hz tone at the 48 000 Hz sampling rate. Examine the waveform and magnitude spectrum to verify that the result is correct.
6. Repeat step 5 using a sampling rate of 8000 Hz.

Table 7.2 File listing for the experiment Exp7.1

Files	Description
`sineGenTest.c`	Program for testing sine wave generation
`tone.c`	C program performs sine wave table lookup
`initAIC3204.c`	C program configures AIC3204 for real-time experiment
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`C55xx_csl.lib`	C55xx chip support library
`USBSTK_bsl.lib`	C5505 eZdsp board support library

Table 7.3 File listing for the experiment Exp7.2

Files	Description
`sirenGenTest.c`	Program for testing siren generation
`siren.c`	C program performs siren signal table lookup
`initAIC3204.c`	C program configures AIC3204 for real-time experiment
`siren.h`	C header file for experiment
`wailSiren.h`	C include file contains siren signal lookup table
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`C55xx_csl.lib`	C55xx chip support library
`USBSTK_bsl.lib`	C5505 eZdsp board support library

7.4.2 Siren Generator Using Table Lookup

This experiment modifies the table lookup method presented in the previous experiment for the real-time generation of sirens using the eZdsp. Table 7.3 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Connect a headphone or a loudspeaker to the eZdsp's audio output jack. Load and run the experiment. Verify the experimental result by listening to the generated wail siren.
3. This experiment uses a pre-computed lookup table of length 19 680 for the wail siren generation at an 8000 Hz sampling rate. How many seconds of signal can be generated without wrapping around at the end of table? How many data points will be needed if this experiment is running at a 32 000 Hz sampling rate for the same time duration?
4. Review the interpolator design and implementation using the FIR filter presented in Chapter 3. Modify the experiment to generate the siren in real time at a sampling rate of 32 000 Hz.

7.4.3 DTMF Generator

This real-time experiment uses the eZdsp to generate DTMF tones using the table lookup method. Table 7.4 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Connect a headphone or a loudspeaker to the eZdsp. Load and run the experiment. Verify the experimental result by listening to the eZdsp audio output (refer to Figure 7.4 for the frequencies assigned to each digit). Since the public telephone switched network uses the 8000 Hz sampling frequency, the eZdsp sampling rate is set at 8000 Hz for the experiment.
3. This experiment generates DTMF tones using the table `digits[]` from `dtmfGenTest.c`. Modify the experiment to create an interactive user interface through the CCS console window such that the user can enter the desired digits, 0 through 9, to generate the corresponding DTMF tones.

Table 7.4 File listing for the experiment Exp7.3

Files	Description
dtmfGenTest.c	Program for testing DTMF tone generation
dtmfTone.c	C program using table lookup for DTMF generation
initAIC3204.c	C program configures AIC3204 for real-time experiment
dtmf.h	C header file for experiment
tone697.h	C include file contains 697 Hz tone table
tone770.h	C include file contains 770 Hz tone table
Tone852.h	C include file contains 852 Hz tone table
Tone941.h	C include file contains 941 Hz tone table
Tone1209.h	C include file contains 1209 Hz tone table
Tone1336.h	C include file contains 1336 Hz tone table
Tone1477.h	C include file contains 1477 Hz tone table
Tone1633.h	C include file contains 1633 Hz tone table
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
C55xx_csl.lib	C55xx chip support library
USBSTK_bsl.lib	C5505 eZdsp board support library

4. The lookup table method is simple and effective. This experiment uses eight tables for the frequencies listed in Table 7.4. Therefore, this method requires large memory space for storing these tables. Modify the experiment such that the DTMF signals are generated in real time using the algorithms presented in Chapter 2 or Chapter 4. Digitize the generated DTMF signal and validate the accuracy of the generated tone frequencies for each digit. Profile the required clock cycles and compare the result to the table lookup method. Also compare the memory requirements of these two methods.

7.4.4 DTMF Detection Using Fixed-Point C

The Goertzel algorithm is often used for DTMF detection in telephony signaling applications. This experiment uses fixed-point C to implement the Goertzel algorithm for detecting DTMF tones. Table 7.5 lists the files used for the experiment.

Table 7.5 File listing for the experiment Exp7.4

Files	Description
dtmfDecodeTest.c	Program for testing DTMF generation
checkKey.c	C program checks DTMF keys
gFreqDetect.c	C program detects DTMF tones
init.c	C program initializes variables for experiment
computeOutput.c	C program computes DTMF decode output
dtmfFreq.c	C program detects DTMF tone frequencies
gFilter.c	C program of Goertzel algorithm
dtmfDetect.h	C header file for experiment
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
DTMF16digits.pcm	DTMF test data file
DTMF_with_noise.pcm	DTMF test data file with noise

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project. Load and run the experiment.
2. This experiment uses file input/output to read a pre-computed DTMF signal from a data file, detect the DTMF tone frequencies and decode the digits, and write the detection result to a file. Two data files are provided for testing. The file `DTMF_with_noise` contains 11 pairs of tones to represent the phone number, 1 8 0 0 5 5 5 1 2 3 4, in a noisy environment; and the file `DTMF16digits` includes all 16 digits of a conventional telephone keypad, 1 2 3 4 5 6 7 8 9 0 A B C D $*$ #. Verify the experimental results are correct by checking the display on the CCS console window or from the experiment output file, `DTMFKEY.txt`.
3. Refer to Exp7.3 to design a real-time experiment to digitize the DTMF signal from the eZdsp line-in jack. (Hint: use the eZdsp board support library `USBSTK5505_I2S_readLeft()` and `USBSTK5505_I2S_readRight()` functions.) Use an audio player to play the test data file as the input signal to the eZdsp and verify that the modified real-time experiment can correctly detect the DTMF digits.

7.4.5 DTMF Detection Using Assembly Program

This experiment implements DTMF tone detection using the assembly program. Table 7.6 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project. Load and run the experiment.
2. Verify the experimental result. The DTMF detection should have identical results to the previous experiment using fixed-point C.
3. Develop a new real-time DTMF tone generation and detection experiment in a loopback setup that can perform the following tasks:
   (a) Generate DTMF tones for all 16 digits and play the generated DTMF tones in real time using the eZdsp.

Table 7.6 File listing for the experiment Exp7.5

Files	Description
`asmDTMFDetTest.c`	Program for testing DTMF generation
`checkKey.c`	C program checks DTMF keys
`gFreqDetect.c`	C program detects DTMF tone frequencies
`init.c`	C program initializes variables
`computeOutput.asm`	Assembly program computes DTMF decode output
`dtmfFreq.asm`	Assembly program detects DTMF frequencies
`gFilter.asm`	Assembly program of Goertzel algorithm
`dtmfDetect.h`	C header file for experiment
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`DTMF16digits.pcm`	DTMF test data file
`DTMF_with_noise.pcm`	DTMF test data file with noise

(b) Use an audio cable to connect the eZdsp audio output to the eZdsp audio input (for audio loopback). Demonstrate that the DTMF detector can correctly detect the DTMF digits based on the tones generated by the DTMF generator in real time using the eZdsp.

Exercises

7.1. Design a new experiment to perform real-time generation of a noisy sine wave using the eZdsp. Use both the sine wave and noise generators to generate a fixed-point sine wave embedded in white noise. Pay special attention to the overflow problem when adding two Q15 numbers. How can overflow be prevented? Try different sine wave frequencies and signal-to-noise ratios.

7.2. The yelp siren has similar characteristics to the wail siren but its period is 0.32 seconds. Use the wail siren generation experiment as reference to design a new eZdsp experiment that generates a yelp siren using the table lookup method.

7.3. ITU Q.24 allows the level of high-frequency DTMF tones to be higher than that of low-frequency tones. Redesign the experiment such that the level of high-frequency DTMF tones generated is 3 dB higher than the low-frequency tones. Demonstrate the result in a real-time experiment using the eZdsp. Examine the generated DTMF tones and verify that the high-frequency tone of each digit is 3 dB higher than its low-frequency counterpart.

7.4. For an *N*-point DFT, the frequency resolution is $f_s/N = 8000/N$ (8000 Hz sampling rate). In the Goertzel algorithm, the signal frequency f_k is approximated by $f_s k/N$ as defined in (5.19). If *N* is not properly selected, the signal frequency f_k may be off by more than 1.5% due to the DFT algorithm. If the frequency tolerance is 1.5%, calculate which $N \in [180, 256]$ makes all eight frequencies meet the requirement.

7.5. Instead of using the Goertzel algorithm, the IIR filterbank can also be used for DTMF detection. Use MATLAB® to design eight fourth-order IIR filters that can be used to replace the Goertzel algorithm. Plot the magnitude responses of these eight filters on a single figure.

7.6. Design a new experiment that uses eight IIR filters designed in Problem 7.5 for DTMF detection. Profile the performance and compare it to the decoder that uses the Goertzel algorithm.

References

1. Orfanidis, S.J. (1996) *Introduction to Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
2. Bateman, A. and Yates, W. (1989) *Digital Signal Processing Design*, Computer Science Press, New York.
3. Schulzrinne, H. and Petrack, S. (2000) RTP Payload for DTMF Digits, Telephony Tones and Telephony Signals, IETF RFC2833, May.
4. The MathWorks, Inc. (2000) Using MATLAB, Version 6.
5. The MathWorks, Inc. (2004) Signal Processing Toolbox User's Guide, Version 6.
6. The MathWorks, Inc. (2004) MATLAB, Version 7.0.1, Release 14, September.
7. Kuo, S.M. and Morgan, D.R. (1996) *Active Noise Control Systems – Algorithms and DSP Implementations*, John Wiley & Sons, Inc., New York.

8. Knuth, D.E. (1981) *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2, 2nd edn, Addison-Wesley, Reading, MA.
9. Hartung, J., Gay, S.L., and Smith, G.L. (1988) Dual-tone Multifrequency Receiver Using the WE DSP16 Digital Signal Processor, Application Note, AT&T.
10. ITU-T Recommendation (1993) Q.23, Technical Features of Push-Button Telephone Sets.
11. ITU-T Recommendation (1993) Q.24, Multifrequency Push-Button Signal Reception.
12. Analog Devices (1990) *Digital Signal Processing Applications Using the ADSP-2100 Family*, Prentice Hall, Englewood Cliffs, NJ.
13. Mock, P. (1986) Add DTMF generation and decoding to DSP-uP designs, in *Digital Signal Processing Applications with the TMS320 Family*, Texas Instruments, Inc., Chapter 19.
14. Texas Instruments, Inc. (2000) DTMF Tone Generation and Detection – An Implementation Using the TMS320C54x, SPRA096A, May.

8

Adaptive Echo Cancellation

The principles of adaptive filters with some practical applications are introduced in Chapter 6. This chapter focuses on one of the important applications of adaptive filtering, namely, adaptive echo cancellation, which is based on adaptive system identification for attenuating undesired echoes. In addition to canceling voice echoes in long-distance networks and acoustic echoes in hands-free speakerphones, adaptive echo cancelers are also widely used in full-duplex networks over two-wire circuits. This chapter introduces adaptive echo cancelers for long-distance networks, voice over internet protocol (VoIP), and speakerphone applications.

8.1 Introduction to Line Echoes

One of the problems associated with telephone communications is the line (or network) echo caused by impedance mismatches at various points in the network [1–3]. The deleterious effects of echoes depend upon their loudness, spectral distortion, and time delay. In general, a longer delay requires a higher degree of echo attenuation. If the time delay between the original speech and the echo is short, the echo may not be noticeable.

A simplified telecommunication network is illustrated in Figure 8.1, where a local telephone is connected to the central office by a two-wire line in which both directions of the transmission are carried on a single pair of wires. The connection between two central offices uses the four-wire facility, which physically segregates the transmission from the two-wire facilities. This is because long-distance transmission requires repeated amplification that is a one-way function. A hybrid (H) located in the central office makes the conversion between the two-wire and four-wire facilities. This type of telephone service system is used for most homes and small offices. In contrast, advanced landline telephone systems, such as the integrated services digital network (ISDN), which are often used by big corporations, the telephone line is digital without using a hybrid for two-wire to four-wire conversion.

An ideal hybrid is a bridge circuit with a balancing impedance that is equal to the impedance of the connected two-wire circuit. Therefore, it will couple all the energy on the

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

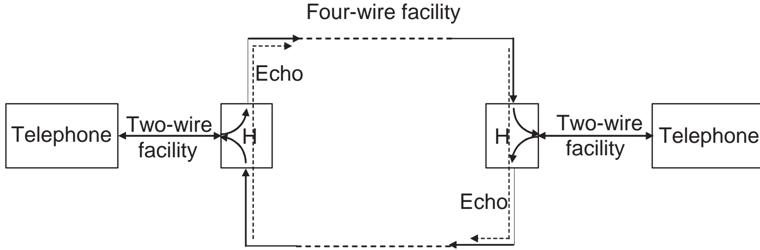


Figure 8.1 Long-distance telecommunication networks

incoming branch of the four-wire circuit into the two-wire circuit. In practice, the hybrid may be connected to any of the two-wire loops served by the central office. Thus, the balancing network can only provide a fixed and compromise impedance match. As a result, some of the incoming signals from the four-wire circuit leak into the outgoing four-wire circuit, and return to the source as an echo, shown in Figure 8.1. This echo requires special treatment if the round-trip delay exceeds 40 ms.

Example 8.1

For internet protocol (IP) trunk applications that use IP packets to relay the circuit switch network traffic, the round-trip delay can easily exceed 40 ms. Figure 8.2 shows a VoIP example using a gateway in which the voice is converted from the time-division multiplex (TDM) circuits to IP packets.

The delay includes speech compression and decompression, jitter compensation, and the network delay. The ITU-T G.729 speech coding standard is widely used for VoIP applications because of its good performance and its 15 ms low algorithm delay. When a 10 ms frame real-time protocol (RTP) packet and 10 ms jitter compensation are used, the round-trip delay of the G.729 speech coder-based system will be at least $2(15 + 10) = 50$ ms without counting the IP network delay and the processing delay. Such a long delay is the reason why adaptive echo cancellation is required for VoIP applications if one or both ends are connected by a TDM circuit.

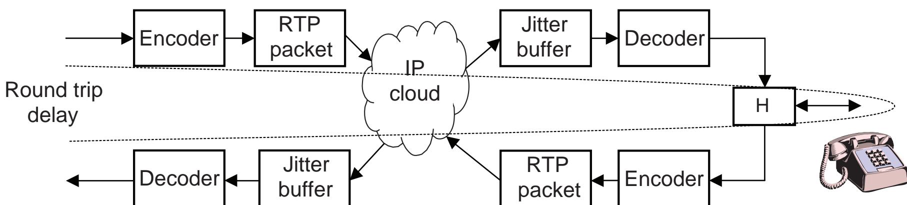


Figure 8.2 Example of round-trip delay for VoIP applications

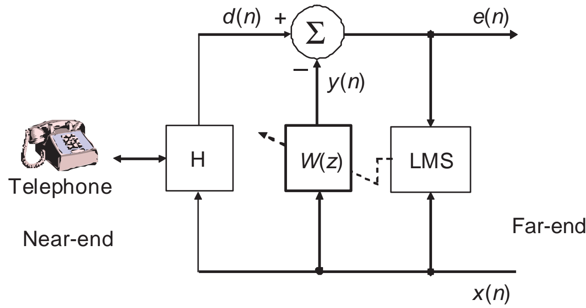


Figure 8.3 Block diagram of adaptive echo canceler

8.2 Adaptive Line Echo Canceler

For telecommunication networks using echo cancellation, the echo canceler is located in the four-wire section of the network near the origin of the echo source(s). The principle of adaptive echo cancellation is illustrated in Figure 8.3. To overcome the line echo problem in a full-duplex communication network, it is necessary to cancel the echoes in both directions of the trunk. We show only one echo canceler located at the left end of network. The reason for showing a telephone and two-wire line is to indicate that, in this chapter, this side is defined as the near-end, while the other side is referred to as the far-end.

8.2.1 Principles of Adaptive Echo Cancellation

To explain the principle of adaptive echo cancellation, the function of the hybrid is illustrated in Figure 8.4 in detail, where the far-end signal $x(n)$ passing through the echo path $P(z)$ results in an undesired echo $r(n)$. The primary signal $d(n)$ consists of the echo $r(n)$, near-end signal $u(n)$, and noise $v(n)$. Based on the principle of adaptive system identification introduced in Chapter 6[4], the adaptive filter $W(z)$ models the echo path $P(z)$ using the far-end speech $x(n)$

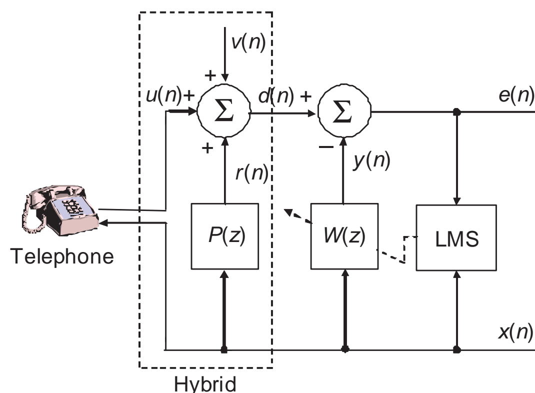


Figure 8.4 Block diagram of adaptive echo canceler with details of hybrid function

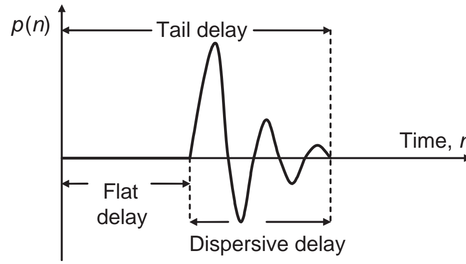


Figure 8.5 A typical impulse response of echo path

as an excitation signal. The output signal $y(n)$ generated by $W(z)$ will be subtracted from the primary signal $d(n)$ to yield the error signal $e(n)$. After the adaptive filter identifies the echo path, its output $y(n)$ (echo replica) approximates the echo, thus the error $e(n)$ contains the near-end speech, noise, and residual echo.

A typical impulse response $p(n)$ of an echo path is shown in Figure 8.5. The time span over the hybrid is usually about 4 ms, which is called the dispersive delay. Because the four-wire circuit is located between the echo canceler and the hybrid, the impulse response of the echo path has a flat delay. The flat delay depends on the transmission delay caused by the distance between the echo canceler and the hybrid, and the filtering delay associated with the frequency- or time-division multiplex equipment. The sum of the flat delay and the dispersive delay is called the tail delay.

Assuming that the echo path $P(z)$ is linear, time invariant, and with infinite impulse response $p(n), n = 0, 1, \ldots, \infty$, the primary signal $d(n)$ can be expressed as

$$\begin{aligned} d(n) &= r(n) + u(n) + v(n) \\ &= \sum_{l=0}^{\infty} p(l)x(n-l) + u(n) + v(n), \end{aligned} \tag{8.1}$$

where the additive noise $v(n)$ is assumed to be uncorrelated with the near-end speech $u(n)$ and the echo $r(n)$. The adaptive FIR filter $W(z)$ estimates the echo as

$$y(n) = \sum_{l=0}^{L-1} w_l(n)x(n-l), \tag{8.2}$$

where L is the filter length. The error signal can be expressed as

$$\begin{aligned} e(n) &= d(n) - y(n) \\ &= u(n) + v(n) + \sum_{l=0}^{L-1}[p(l) - w_l(n)]x(n-l) + \sum_{l=L}^{\infty} p(l)x(n-l). \end{aligned} \tag{8.3}$$

Due to the changing power of speech signals, the normalized LMS algorithm [5,6] introduced in Section 6.3 is commonly used for adaptive echo cancellation applications. Assuming that the disturbances $v(n)$ and the near-end speech $u(n)$ are uncorrelated with the far-end speech $x(n)$, then $W(z)$ will converge to $P(z)$, that is, $w_l(n) \approx p(l), l = 0, 1, \ldots, L-1$. Thus the adaptive filter $W(z)$ adapts its weights $w_l(n)$ to mimic the first L samples of the impulse response of the echo path. As shown in (8.3), the residual error after $W(z)$ has converged can be expressed as

$$e(n) \approx \sum_{l=L}^{\infty} p(l)x(n-l) + u(n) + v(n), \tag{8.4}$$

where the first term on the right-hand side is called the residual echo. By making the length of $W(z)$ sufficiently long to cover the tail delay, as shown in Figure 8.5, the residual echo can be minimized. However, as discussed in Section 6.3, the excess MSE produced by the adaptive algorithm and finite-precision errors are also proportional to the filter length. Therefore, there is an optimum length L for a given echo cancellation application.

The length L of the adaptive FIR filter is determined by the tail delay shown in Figure 8.5. As mentioned earlier, the impulse response of the hybrid (dispersive delay) is relatively short. However, the flat delay from the echo canceler to the hybrid depends on the physical location of the echo canceler and the processing delay of the transmission equipment.

8.2.2 Performance Evaluation

The effectiveness of the adaptive echo canceler is usually measured by the echo return loss enhancement (ERLE) defined as

$$\mathrm{ERLE} = 10 \log \left\{ \frac{E\left[d^2(n)\right]}{E\left[e^2(n)\right]} \right\}. \tag{8.5}$$

For a given application, the ERLE depends on the step size μ, the filter length L, the signal-to-noise ratio, and the nature of the signal in terms of power and spectral contents. A larger step size results in faster initial convergence, but the final ERLE will be smaller due to the excess MSE and quantization errors. If the filter length is long enough to cover the echo tail (or tail delay), further increases L will reduce the ERLE.

The ERLE achieved by an adaptive echo canceler is limited by many practical factors. Detailed requirements for adaptive echo cancelers are defined by ITU-T recommendations G.165 [7] and G.168 [8], including the maximum residual echo level, the echo suppression effect on the hybrid, the convergence time, the initial setup time, and the degradation in a double-talk situation.

In the past, adaptive echo cancelers were implemented using customized devices in order to handle the heavy computation for real-time applications. Disadvantages of VLSI implementation [9,10] are long development time, high development cost, lack of flexibility to meet new application-specific requirements, and inability to be upgraded for more advanced algorithms. Therefore, recently adaptive echo canceler design and development have been based on programmable digital signal processors.

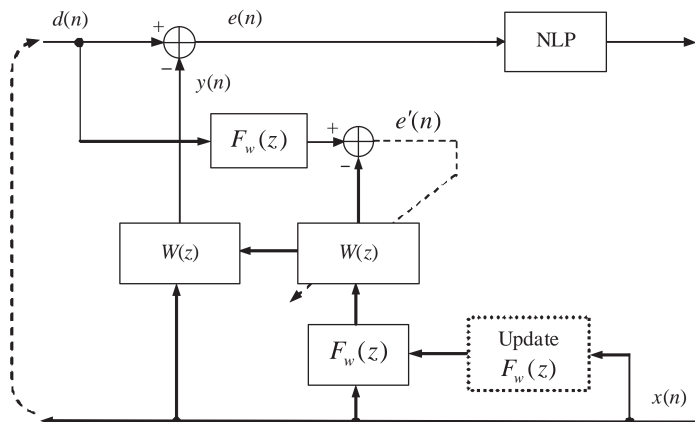


Figure 8.6 Block diagram of signal pre-whitening structure

8.3 Practical Considerations

This section discusses two practical issues in designing adaptive echo cancelers: pre-whitening and delay estimation.

8.3.1 Pre-whitening of Signals

As discussed in Chapter 6, the convergence time of the adaptive FIR filter using the LMS algorithm is proportional to the spectral ratio $\lambda_{max}/\lambda_{min}$. Since a speech signal is highly correlated with a non-flat spectrum, the convergence speed is generally slow. The decorrelation (whitening) of the input speech signal can be used to improve the convergence speed [11].

Figure 8.6 shows a typical pre-whitening structure for input signals where the whitening and adaptation are processed in the background. The same whitening filter $F_w(z)$ is used for both the far-end signal $x(n)$ and the near-end signal $d(n)$. The whitened signals are used to update the background adaptive filter $W(z)$ for improving the convergence rate. The foreground echo cancellation uses the original far-end and near-end signals, thus the resulting signal $e(n)$ will not be affected by the pre-whitening process. The function of the nonlinear processor (NLP) shown in Figure 8.6 will be introduced in Section 8.5.

The fixed filter $F_w(z)$ can be obtained using reversed statistical or temporal averaged spectrum values. One example is the anti-tile filter used to lift up the high-frequency components since the power of most speech signals is concentrated in the low-frequency region. The whitening filter can be updated based on the far-end signal $x(n)$, which is similar to the adaptive channel equalization discussed in Chapter 6.

8.3.2 Delay Estimation

As discussed in Section 8.2.1, the initial part of the impulse response of the echo path (flat delay in Figure 8.5) represents the transmission delay between the echo canceler and the hybrid. The structure illustrated in Figure 8.7 uses the delay unit $z^{-\Delta}$ to cover the flat delay,

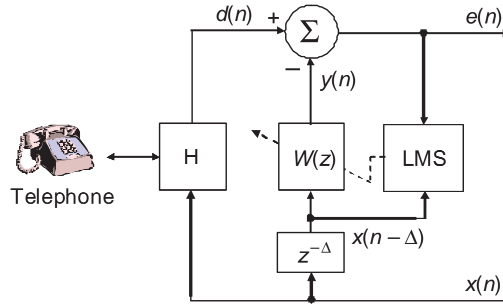


Figure 8.7 Adaptive echo canceler with effective flat-delay compensation

where Δ is the number of flat-delay samples. By estimating the length of the flat delay and using the delay unit $z^{-\Delta}$, the echo canceler $W(z)$ can be shortened by Δ samples since it covers only the dispersive delay. This technique effectively improves the convergence speed and reduces the excess MSE and computational requirements. However, there are three major difficulties in realizing this technique in real applications: the existence of multiple echoes, the difficulty in estimating the flat delay, and the delay variation during a call.

The cross-correlation function between the far-end signal $x(n)$ and the near-end signal $d(n)$ can be used to estimate the delay. The normalized cross-correlation function with lag k can be estimated as

$$\rho(k) = \frac{r_{xd}(k)}{\sqrt{r_{xx}(k) r_{dd}(k)}}, \tag{8.6}$$

where $r_{xd}(k)$ is the cross-correlation function defined in (6.9), and the autocorrelations $r_{xx}(k)$ and $r_{dd}(k)$ are defined in (6.3). They can be estimated using (6.12) with typical values of length N between 128 and 256 for an 8 kHz sampling rate.

Example 8.2

Given the far-end data file `rtfar.pcm` (as vector `y`) and the near-end data file `rtmic.pcm` (as vector `x`) and a sampling rate of 8 kHz, the MATLAB® program `example8_2.m` uses the function `xcorr(x,y,'biased')` to estimate the cross-correlation between the vectors `x` and `y` [12,13]. The option '`biased`' normalizes the calculated cross-correlation function with the length of the vectors. The maximum value (shown as the peak) in Figure 8.8 indicates the delay between the far-end and near-end signals. Thus, we can estimate the delay between the vectors `x` and `y` by locating the peak lag index `k`.

The flat delay is identified by the lag k that maximizes the normalized cross-correlation function defined by (8.6). Unfortunately, this method may have poor performance for speech signals, although it has good performance for signals with flat spectra such as white noise.

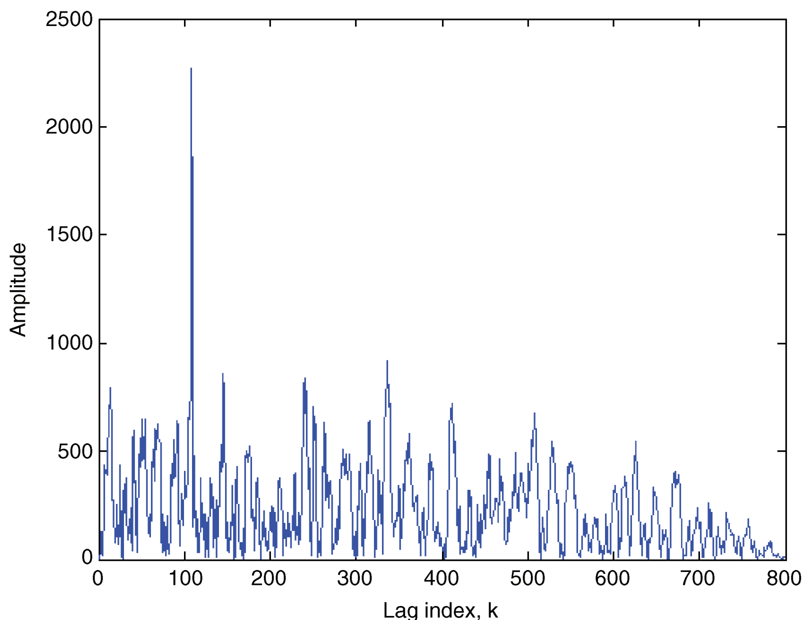


Figure 8.8 Cross-correlation function between near-end and far-end data

A bandpass filter with the passband covering two or three formants can be applied to improve the performance of the cross-correlation method to estimate the delay [14]. This technique makes the cross-correlation method more reliable by whitening the signals. Multirate filtering (introduced in Section 3.4) can also be used to further reduce the computational load. In this case, the normalized cross-correlation function $\rho(k)$ is computed using the subband signals. With a properly designed bandpass filter and down-sampling factor D, the decimated subband signals can have flat spectra similar to that of white noise.

Example 8.3

An adaptive echo canceler is located at a tandem switch as shown in Figure 8.9. For a typical impulse response of the echo path shown in Figure 8.5, assuming the sampling rate is 8 kHz, the flat delay is 15 ms, and the dispersive segment is 10 ms, find the adaptive filter length with and without using the delay estimate technique.

Since the flat delay is a pure delay, this part can be covered by the delay unit $z^{-\Delta}$ where $\Delta = 120$ (15 ms). So, using the delay estimate, the actual filter length is 80 to model the 10 ms of dispersive segment; that is, the FIR filter coefficients only need to approximate the dispersive delay of the hybrid. Without using the delay estimate method, the adaptive filter length will be 200 in order to cover the entire tail length including the flat delay and the dispersive delay. However, accurate delay estimation is extremely important because the flat delay may change for each connection.

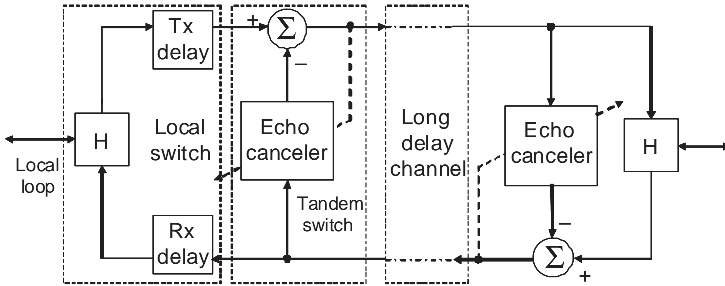


Figure 8.9 Configuration for echo cancellation with flat delay

8.4 Double-Talk Effects and Solutions

A very important issue in designing adaptive echo cancelers for practical applications is how to handle the double-talk problem, which occurs when the far-end and near-end talkers are speaking simultaneously. During the double-talk periods, signal $d(n)$ contains both the near-end speech $u(n)$ and the undesired echo $r(n)$ as shown in Figure 8.4, thus the error signal $e(n)$ described in (8.4) contains the residual echo, the uncorrelated noise $v(n)$, and the near-end speech $u(n)$. As discussed in Section 6.5.1 for adaptive system identification, $d(n)$ must be generated solely from its excitation input signal $x(n)$ in order to correctly identify the characteristics of $P(z)$.

In theory, the far-end signal $x(n)$ is uncorrelated with the near-end speech $u(n)$, and thus will not affect the asymptotic mean value of the adaptive filter coefficients. However, the variation in the filter coefficients about this mean will be increased substantially in the presence of near-end speech. The echo cancellation performance will be degraded because the previously converged adaptive filter coefficients will be corrupted by using the near-end speech as $e(n)$ for adaptation. An unprotected algorithm may exhibit unacceptable adaptive filter behaviors during double-talk periods.

An effective solution for the double-talk problem is to correctly detect the double-talk occurrences and immediately disable the adaptation of $W(z)$ during the double-talk periods. When the adaptation of coefficients is disabled as illustrated in Figure 8.10, the filter

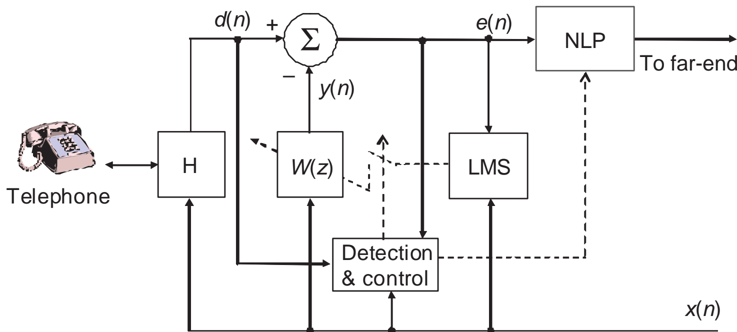


Figure 8.10 Adaptive echo canceler with speech detectors and nonlinear processor

coefficients are fixed at that moment. The filtering of $x(n)$ by $W(z)$ with the fixed coefficients to generate $y(n)$ for canceling the echo component in $d(n)$ will be continued during the double-talk periods. If the echo path does not change during the double-talk intervals and the detection of double talk is fast and accurate, the echo can still be canceled using the fixed $W(z)$ which have previously converged to model the echo path.

As shown in Figure 8.10, a double-talk detector (DTD) uses the detection and control block to control the adaptation of the adaptive filter $W(z)$, and uses the NLP to reduce residual echo. The DTD, which detects the presence of near-end speech when far-end speech is present, is very challenging to design for adaptive echo cancelers.

The echo return loss (ERL), or hybrid loss, can be expressed as

$$\mathrm{ERL} = 20\log_{10}\left\{\frac{E[|x(n)|]}{E[|d(n)|]}\right\}. \tag{8.7}$$

Note that the absolute value is used instead of the instantaneous power for computational savings in practical applications. For several adaptive echo cancelers defined by ITU standards, the ERL value is assumed to be at least 6 dB. Based on this assumption, a conventional DTD detects near-end speech if

$$|d(n)| > \frac{1}{2}|x(n)|. \tag{8.8}$$

The advantage of using the instantaneous absolute value (or power) of $d(n)$ is its fast response to near-end speech. However, it may increase the probability of a false trigger if noise exists in the network. Thus we consider the modified near-end speech detection algorithm that declares the presence of near-end speech if

$$|d(n)| > \frac{1}{2}\max\{|x(n)|, \ldots, |x(n-L+1)|\}. \tag{8.9}$$

This equation compares the instantaneous absolute value $|d(n)|$ to the maximum absolute value of $x(n)$ over a time window spanning the echo path.

A more robust version of speech detector can be obtained by replacing the instantaneous power $|x(n)|$ and $|d(n)|$ with the short-term power estimates $P_x(n)$ and $P_d(n)$. These short-term power estimates are implemented by the first-order IIR filter as

$$P_x(n) = (1-\alpha)P_x(n-1) + \alpha|x(n)| \tag{8.10}$$

and

$$P_d(n) = (1-\alpha)P_d(n-1) + \alpha|d(n)|, \tag{8.11}$$

where $0 < \alpha \ll 1$. The use of a larger value of α results in a more robust detector. However, it can cause a slower response for detecting near-end speech. With the modified short-term power estimate, near-end speech is present if

$$P_d(n) > \frac{1}{2} \max\{P_x(n), P_x(n-1), \ldots, P_x(n-L+1)\}. \tag{8.12}$$

It is important to note that a portion of the initial break-in near-end speech $u(n)$ may not be detected by this detector. Thus, adaptation would proceed at the beginning of double-talk periods. Furthermore, the requirement of maintaining a buffer to store L power estimates increases the memory requirements and the complexity of the algorithm.

The assumption that the ERL is a constant of 6 dB is not always correct. If the real ERL is higher than 6 dB, it will take longer to detect the presence of near-end speech. In this case, the adaptive filter coefficients will be corrupted by the near-end speech. If the ERL is lower than 6 dB, some far-end speech samples will be falsely detected as the presence of near-end speech. In this case, the adaptation will be stopped without the presence of near-end speech. For practical applications, it is better to dynamically estimate the time-varying threshold ERL by observing the signal levels of $x(n)$ and $d(n)$ when the near-end speech $u(n)$ is absent.

8.5 Nonlinear Processor

Nonlinearities in the echo path, noise in the circuits, and the presence of near-end speech can limit the achievable performance of a typical adaptive echo canceler. To further increase the overall echo reduction, the residual echo can be removed using an NLP realized as a center clipper. The insertion of comfort noise can minimize the adverse effects caused by the NLP.

8.5.1 Center Clipper

The NLP shown in Figure 8.10 removes the last vestiges of the remaining echoes. The most widely used NLP is the center clipper with the input/output characteristic illustrated in Figure 8.11. This nonlinear operation can be expressed as

$$y(n) = \begin{cases} 0, & |x(n)| \leq \beta \\ x(n), & |x(n)| > \beta, \end{cases} \tag{8.13}$$

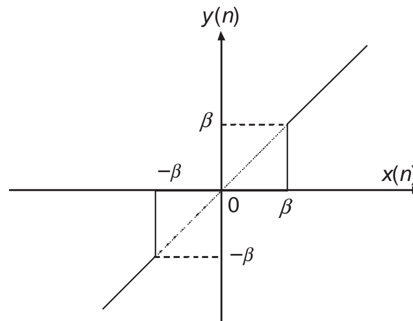


Figure 8.11 Input/output relationship of center clipper

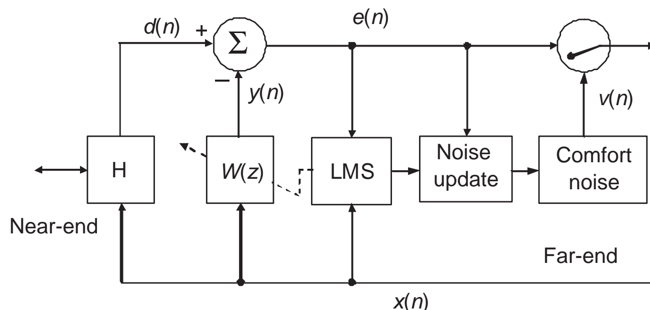


Figure 8.12 Implementation of G.168 with comfort noise insertion

where β is the clipping threshold. This center clipper completely attenuates signal samples below the clipping threshold β by setting them to zero while keeping signal samples above the clipping threshold unaffected. A larger β suppresses more residual echoes but also deteriorates the quality of the near-end speech. Usually the threshold is chosen to be equal to or exceed the peak amplitude of the return echo.

8.5.2 Comfort Noise

An NLP will completely eliminate the low-level residual echo and circuit noise. However, this can make the phone connection sound unrealistic. For example, if the near-end subscriber stops talking, the signal level transmitted to the far-end will suddenly drop to zero because of NLP clipping. If the difference is significant, the far-end subscriber may think the call has been disconnected. Therefore, complete suppression of the signal using the NLP has undesired effects. This problem can be solved by injecting low-level comfort noise when the residual echo is suppressed.

As specified by G.168, comfort noise must match the background noise level and frequency contents. In order to match the spectrum, the insertion of comfort noise [8,15] can be implemented in the frequency domain by capturing the frequency characteristic of the background noise. An alternate approach uses the linear predictive coding (LPC) coefficients to model the spectral information. In this case, comfort noise is synthesized using a pth-order LPC all-pole filter, where the order p is between 6 and 10. The LPC coefficients are computed during the segments of silence. G.168 recommends a level of comfort noise within ± 2 dB of near-end noise.

An effective way to implement the NLP with comfort noise is illustrated in Figure 8.12, where either the generated comfort noise $v(n)$ or echo canceler output $e(n)$ will be selected as the output according to the control logic. The background noise is generated with the matched level and spectrum during call connection, thus significantly improving perceptive speech quality.

8.6 Adaptive Acoustic Echo Cancellation

There has been growing interest in applying acoustic echo cancellation for hands-free communication in mobile environments and speakerphones for teleconferencing. For

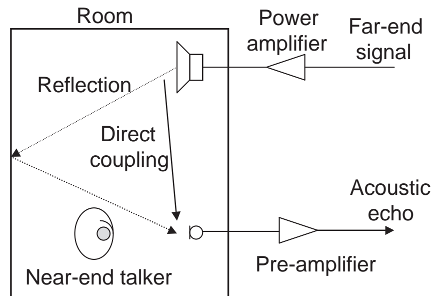


Figure 8.13 Acoustic echo generated by speakerphone in a room

example, the speakerphone has become important office equipment and provides mobile communication functionality because it makes hands-free conversation convenient [16–18]. In general, acoustic echoes consist of three major components: (1) acoustic energy coupling between the loudspeaker and the microphone; (2) multiple-path sound reflections of far-end speech played by the loudspeaker in the room; and (3) near-end speech signal reflections. In this section, we focus on the cancellation of the first two echo components.

8.6.1 Acoustic Echoes

For reference purposes, the person using the speakerphone is the near-end talker and the person at the other end of the communication link is the far-end talker. As shown in Figure 8.13, far-end speech is broadcast through the loudspeaker inside the room. Unfortunately, the far-end speech played by the loudspeaker will also be picked up by the microphone inside the room, and this acoustic echo is returned to the far-end.

The basic concept of acoustic echo cancellation is similar to line echo cancellation; however, the adaptive filter used for the acoustic echo canceler models the loudspeaker–room–microphone system instead of the hybrid. Thus, the acoustic echo canceler needs to cancel the longer echo tail by using a higher order adaptive FIR filter. One effective technique is to use a subband acoustic echo canceler, which splits the full-band signals into several overlapped subbands and uses individual low-order filters for processing decimated subband signals.

Example 8.4

To evaluate a real acoustic echo path, the impulse response of a rectangular room of $246 \times 143 \times 111$ in^3 or $6.25 \times 3.63 \times 2.82$ m^3 was measured using white noise as an excitation signal. The original data is sampled at 48 kHz, which is then decimated to 8 kHz. The room impulse response is saved in the file `roomImp.pcm` and is plotted in Figure 8.14 by the MATLAB® program `example8_4.m`.

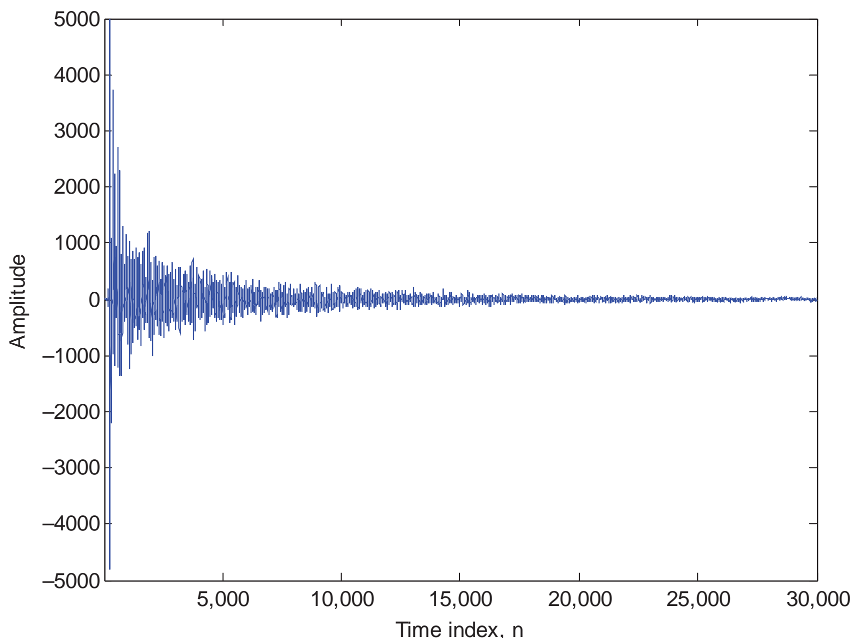


Figure 8.14 An example of room impulse response

There are three major factors making the acoustic echo cancellation far more challenging than line echo cancellation for real-world applications. They are summarized as follows:

1. The room reverberation causes a very long acoustic echo tail, see Figure 8.13. For example, 4000 taps are needed to cancel 500 ms of acoustic echo at the sampling rate of 8 kHz. As explained in Chapter 6, long filter length reduces the upper bound of step size, thus resulting in slow convergence.
2. The acoustic echo path may change rapidly due to the motion of people inside the room, the positional changes of the microphone, and some other factors like opening or closing doors and/or windows. Thus, the acoustic echo canceler usually requires a fast convergence algorithm to track these changes.
3. The design of double-talk detection is more difficult since the 6 dB acoustic loss cannot be assumed as the hybrid loss in the line echo canceler.

Therefore, acoustic echo cancelers require more computational power, fast convergence speed, and a more sophisticated double-talk detector.

8.6.2 Acoustic Echo Canceler

The block diagram of an acoustic echo canceler is illustrated in Figure 8.15. The acoustic echo path $P(z)$ includes the transfer functions of the ADCs and DACs, smoothing and antialiasing

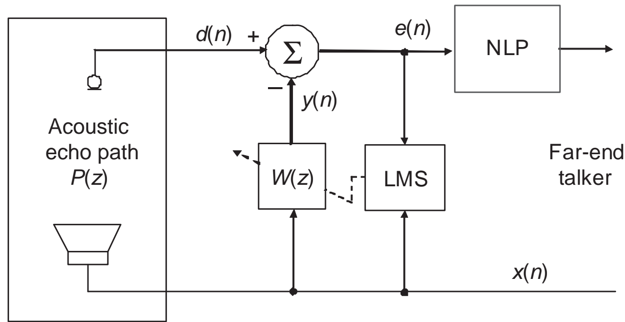


Figure 8.15 Block diagram of acoustic echo canceler

lowpass filters, power amplifier, loudspeaker, microphone, pre-amplifier, automatic gain control, and the room transfer function from the loudspeaker to the microphone. The adaptive filter $W(z)$ models the acoustic echo path $P(z)$ and yields the echo replica $y(n)$ to cancel acoustic echo components in $d(n)$.

8.6.3 Subband Implementations

Subband and frequency-domain adaptive filtering techniques [19] have been developed to cancel long and fast-changing acoustic echoes. The advantages of using subband acoustic echo cancelers are: (1) the signal whitening using a normalized step size at each subband results in fast convergence; and (2) the decimation of subband signals reduces the computational requirements. The decimation techniques are introduced in Chapter 3.

The typical structure of a subband echo canceler is shown in Figure 8.16, where $A_m(z)$ and $S_m(z)$ are the analysis and synthesis filters, respectively. The bandpass filters $A_m(z)$ and $S_m(z)$ are connected in parallel to form the analysis and synthesis filterbanks, respectively. The number of subbands is M, and the decimation factor D can be a number equal to or less than M, that is, $D \leq M$. When the decimation factor equals the number of subbands M as shown in Figure 8.16, we use the critical decimation rate, that is, $D = M$. The full-band signals $x(n)$ and $d(n)$ are split by the same analysis filterbank to form M subband signals and each subband signal is decimated by the factor M. These low-rate subband signals are processed by the corresponding adaptive FIR filters $W_m(z)$. The subband error signals after echo cancellation are up-sampled by the factor M and reconstructed to the full-band signal using the synthesis filterbank $S_m(z)$. These M adaptive filters are updated independently using the normalized LMS algorithm based on the estimated power of the corresponding input subband signals, which results in faster convergence speed. Usually, a lower order adaptive FIR filter is used for each subband and each filter is adapted using the decimated rate, which results in significant computational savings as compared to the full-band adaptive filter $W(z)$ shown in Figure 8.15.

The design of filterbanks with complex coefficients reduces the filter length due to the relaxed antialiasing requirement. The drawback is the increased computation because one complex multiplication requires four real multiplications. Nevertheless, the complex filterbank is still commonly used because of the difficulties in designing real-coefficient bandpass filters with sharp cutoffs and strict antialiasing requirements for adaptive echo cancellation.

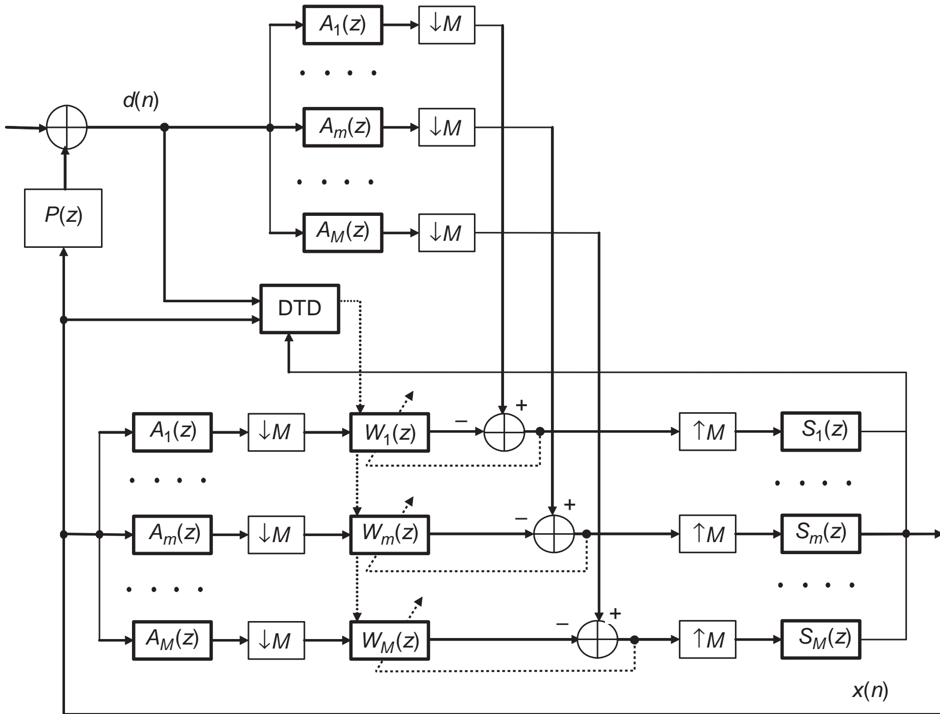


Figure 8.16 Block diagram of subband adaptive echo canceler

The procedure for subband adaptive echo cancellation using a 16-band ($M = 16$) filterbank with complex coefficients is highlighted as follows:

1. Use MATLAB® to design a prototype lowpass FIR filter with coefficients $h(n)$, $n = 0, 1, \ldots, N-1$, which meets the requirement of the 3 dB bandwidth at $\pi/(2M)$. Note that this baseband lowpass filter has real-valued coefficients. The magnitude response of the prototype filter is shown in Figure 8.17(a) and its impulse response is given in Figure 8.17(b).
2. Multiply

$$\cos\left[\pi\left(\frac{m-1/2}{M}\right)\left(n-\frac{N+1}{2}\right)\right] \quad \text{and} \quad \sin\left[\pi\left(\frac{m-1/2}{M}\right)\left(n-\frac{N+1}{2}\right)\right]$$

by the prototype filter coefficients to generate M bandpass analysis filters, $A_m(z)$, $m = 0, 1, \ldots, M-1$. Therefore, the analysis filterbank consists of M complex bandpass filters and each filter has bandwidth π/M. In this example, the synthesis filterbank is chosen to be the same as the analysis filterbank, that is, $S_m(z) = A_m(z)$ for $m = 0, 1, \ldots, M-1$. The magnitude responses of these analysis (or synthesis) filters are shown in Figure 8.17(c). Figure 8.17(d) shows the full-band magnitude response of the analysis (or synthesis) filterbank by summing M magnitude responses of the bandpass filters.

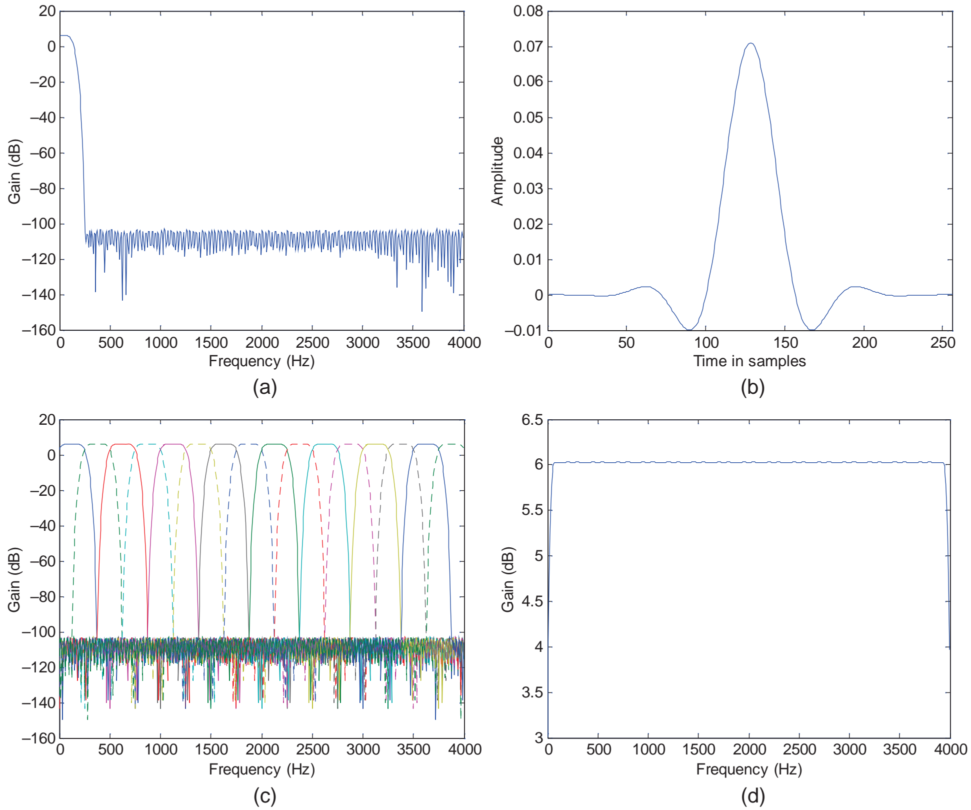


Figure 8.17 Example of filterbank with 16 complex subbands: (a) magnitude response of prototype filter; (b) impulse response of prototype filter; (c) magnitude responses of all filters; and (d) combined magnitude response of analysis (or synthesis) filterbank.

3. Use the analysis filterbank to split the full-band signals $x(n)$ and $d(n)$ into M subbands, and decimate each subband filter output by D ($\leq M$) to produce the low-rate far-end and near-end subband signals $x_m(n)$ and $d_m(n)$, respectively.
4. Perform the echo cancellation and adaptation of filter coefficients using the normalized LMS algorithm for each individual subband adaptive filter at the decimated sampling rate ($1/D$). This step produces subband error signals $e_m(n)$, $m = 0, 1, \ldots, M-1$.
5. These M subband error signals are interpolated by the factor D and synthesized back to the full-band signal using the synthesis filterbank which consists of M parallel bandpass filters $S_m(z)$.

Example 8.5

Follow the filterbank design procedure steps 1 and 2 to calculate the 16-band analysis filters coefficients. The complete MATLAB® program is given in `example8.5`. The prototype lowpass filter is a linear phase FIR filter of order 256, as shown by the symmetric filtered coefficients in Figure 8.17(b). The magnitude responses of these complex bandpass filters and the overall filterbank are shown in Figure 8.17.

Example 8.6

For the same echo tail length, compare the computational load of adaptive echo cancelers using a subband adaptive filter with two subbands (assume real-valued coefficients) and a full-band adaptive filter. Specifically, assume the length of the echo tail is 32 ms (256 samples at 8 kHz sampling rate), and estimate the required number of multiply and add operations.

Subband implementation requires 2×128 multiplications and additions for updating coefficients at half ($D = 2$) the original sampling rate. In comparison, the full-band adaptive filter needs 256 multiplications and additions at the original sampling rate. Thus the subband implementation needs only half the computations required for the full-band implementation. In the comparison, the computational load for analysis and synthesis filters is not counted since this computational load is relatively small as compared to the coefficients update using the adaptive algorithm, especially when the number of subbands M is large.

8.6.4 Delay-Free Structures

An inherent disadvantage of subband implementations is the extra delay introduced by the analysis filterbank to split the full-band signal into multiple subbands and also the synthesis filterbank to synthesize the processed subband signals back to the full-band signal. Several delay-less subband adaptive filters are discussed in the recent book titled *Subband Adaptive Filtering* [20]. In general, the delay-less subband adaptive filter uses a full-band FIR filter in the foreground for echo cancellation to avoid the extra delay caused by the analysis and synthesis filterbanks. The coefficients of the full-band foreground filter are transformed from the background subband adaptive filters $W_m(z)$ shown in Figure 8.16.

An example of a delay-free subband acoustic echo canceler is realized by adding an additional short full-band adaptive FIR filter $W_0(z)$, which covers the first part of the echo path and has a length equal to the delay introduced by the analysis and synthesis filters plus the block processing size. The subband adaptive filters model the rest of the echo path. This scheme is different from the one using the foreground full-band FIR filter, whose coefficients are transformed from the background subband adaptive filters. Figure 8.18 illustrates the structure of this delay-free subband acoustic echo cancellation.

8.6.5 Integration of Acoustic Echo Cancellation with Noise Reduction

There are many practical applications, such as using hands-free phones in noisy environments, which require the combination of acoustic echo cancellation (AEC) and noise reduction (NR) techniques. These two subsystems can be integrated using cascade structure. The NR techniques will be introduced in Section 9.3.

The combined structure is shown in Figure 8.19. In the first stage, the echo is canceled by using the AEC filter $W(z)$ to produce the echo-free signal. The second stage reduces noise through the NR techniques by a Wiener filter or spectral subtraction. The integration of AEC and NR will be presented as an experiment in Section 8.7.3.

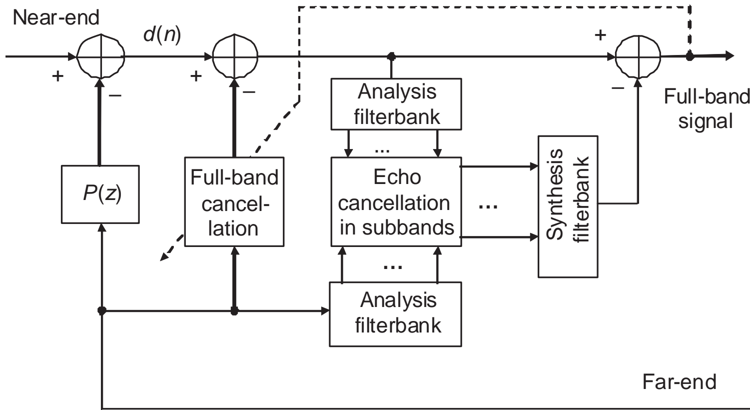


Figure 8.18 An example of delay-free subband acoustic echo canceler

8.6.6 Implementation Considerations

As shown in Figure 8.7, one of the effective techniques for using a shorter acoustic echo canceler filter is to employ a delay buffer (of Δ samples) before the adaptive filter. This buffer compensates for the propagation delay in the echo path between the loudspeaker and the microphone. Using this technique, the actual filtering process will not be required for the samples in the delay buffer, so the adaptive filter can be shortened. For example, if the distance between the loudspeaker and the microphone is 1.5 meters, then the pure time delay is about 4.526 ms based on the speed of sound traveling at 331.4 meters per second, which corresponds to $\Delta = 36$ at the 8 kHz sampling rate.

The step size μ used for the acoustic echo canceler is usually small due to the large L. This may produce higher excess MSE. Using fixed-point implementations may introduce higher numerical (coefficient quantization and roundoff) errors. Furthermore, the roundoff error may cause early adaptation termination if a small μ is used. One solution to these problems is to increase the dynamic range, which may require using floating-point arithmetic with more expensive hardware.

As mentioned earlier, the coefficient adaptation must be temporarily disabled when near-end speech is detected. Most DTDs for adaptive line echo cancelers are based on the 6 dB echo return loss assumption. This is no longer the case for acoustic echo cancelers. Their echo

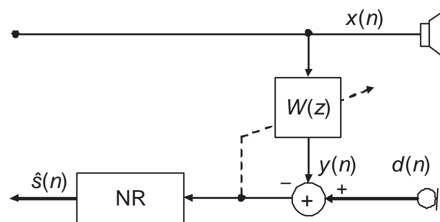


Figure 8.19 Integration of AEC with NR

return (or acoustic) loss is very small or may even be a gain because of the use of high-gain amplifiers to drive the loudspeakers. Therefore, the higher level of acoustic echoes makes the detection of near-end speech very difficult.

8.7 Experiments and Program Examples

This section presents the C5505 eZdsp experiments to examine the performance of the AEC algorithms introduced in this chapter.

8.7.1 Acoustic Echo Canceler Using Floating-Point C

This experiment uses floating-point C to implement an acoustic echo canceler with the leaky NLMS algorithm. AEC uses the adaptive FIR filter of length 512. The data files used for the experiment are digitized at the 8 kHz sampling rate using a PC sound card. The conversation is carried out in a room of size $11 \times 11 \times 8\ \text{ft}^3$, or $3.33 \times 3.33 \times 2.44\ \text{m}^3$.The far-end speech file `rtfar.pcm` and the near-end speech file `rtmic.pcm` are captured simultaneously. The near-end signal picked up by the microphone consists of the near-end speech and acoustic echoes generated from the far-end speech. The files used for the experiment are listed in Table 8.1.

The adaptive echo canceler operates in four different modes based on the far-end and near-end signals. These four operating modes are:

1. Receive mode – only the far-end speaker is talking.
2. Transmit mode – only the near-end speaker is talking.
3. Idle mode – both ends are silence.
4. Double-talk mode – both ends are talking simultaneously.

Different operations are required for the corresponding modes. For example, the adaptive filter coefficients will be updated only during the receive mode.

Figure 8.20 illustrates the performance of the acoustic echo canceler: (a) the far-end speech signal that is played via the loudspeaker in the room; (b) the near-end signal picked up by the microphone, which consists of the near-end speech as well as the acoustic echoes caused by playing the far-end speech in the room; and (c) the acoustic echo canceler output, which will be transmitted to the far-end. Examining the result shown in (c), it clearly shows that the echo

Table 8.1 File listing for the experiment Exp8.1

Files	Description
`fAecTest.c`	Program for testing acoustic echo canceler
`fAecInit.c`	AEC initialization function
`fAecUtil.c`	Echo canceler utility functions
`fAecCalc.c`	Main module for echo canceler
`fAec.h`	C header file
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`rtfar.pcm`	Far-end PCM data file
`rtmic.pcm`	Near-end PCM data file

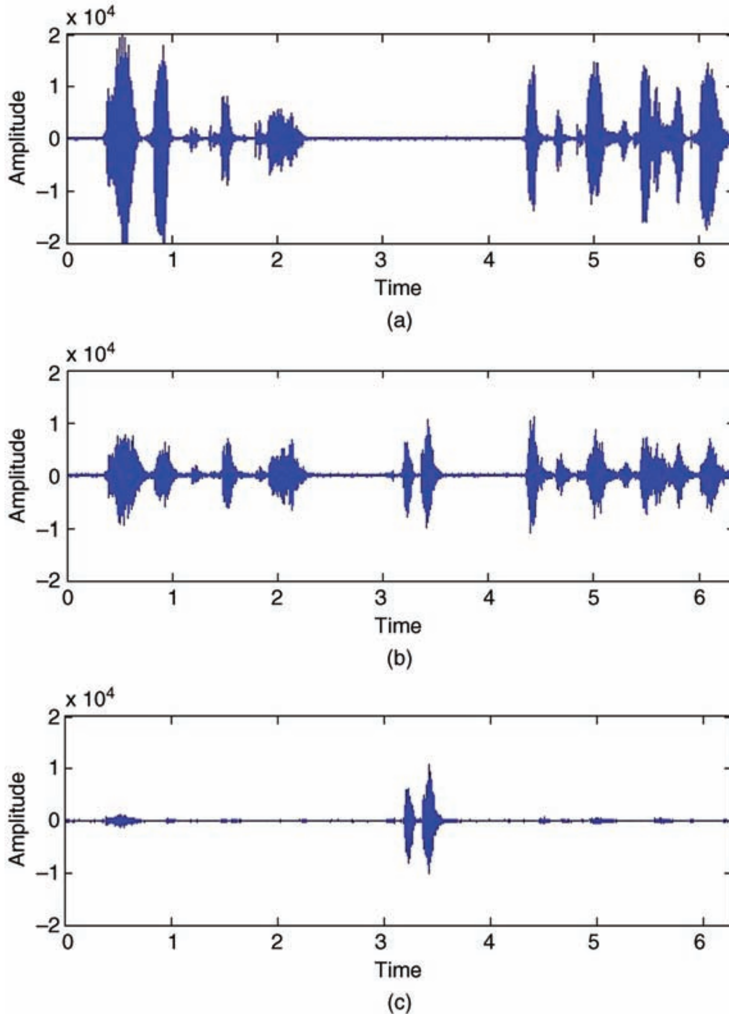


Figure 8.20 Experiment results of AEC: (a) far-end signal, (b) near-end signal picked up by the microphone; and (c) acoustic echo canceler's output signal

canceler's output contains the near-end speech with very low levels of residual echo. The digitized data files used in this experiment represent a typical telephone conversation between the near-end and far-end talkers. During the conversation, they usually talk at different times. Since the near-end and far-end talkers do not speak simultaneously, double talk does not occur in this experiment. The echo canceler reduces the acoustic echo by more than 20 dB.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment. Plot the AEC output signal (the error signal of the adaptive filter) and compare to Figure 8.20(c) to verify the experiment works properly.

3. The experiment initializes the AEC parameters in the C program `fAecInit.c`. Change the filter length from the original 512 to 256 and 128, Compare the AEC performance to the result shown in Figure 8.20(c) to explain the experimental results obtained using different filter lengths of 128, 256, and 512.
4. Increase the length of the adaptive FIR filter to 1024 and rerun the experiment. Observe and explain the result and correct the problem.
5. Conduct more experiments by using different parameter values, such as leaky factor (`leaky`), hangover time (`hangoverTime`), and step size μ (`mu`). These parameters are initialized in the function `aec_param_init()`.
6. Digitize different sets of near-end and far-end speech signals to replace the provided speech files `rtmic.pcm` and `rtfar.pcm` for new experiments. Especially, try the data sets with double-talk periods and observe the degradation of echo cancellation during the double talk. It is important to note that the near-end and far-end signals must be digitized simultaneously for the experiments.

8.7.2 Acoustic Echo Canceler Using Fixed-Point C with Intrinsics

This experiment uses the fixed-point C program with C55xx intrinsics to implement the acoustic echo canceler. We use the normalized LMS algorithm for time-varying speech signals, use the double-talk detector to avoid performance degradation during double-talk periods, and add the NLP function to further attenuate the residual echo. The files used for the experiment are listed in Table 8.2. Note that the same near-end and far-end speech files used in Exp8.1 are used for this experiment.

Fixed-point C implementation of the NLMS algorithm using intrinsic functions has been discussed in Chapter 6. Using the same technique, the double-talk detector can be implemented using fixed-point C with the C55xx intrinsics. Figure 8.21 shows the acoustic echo canceler's output. Comparing it to Figure 8.20(c), we can observe the differences mainly due to the use of the NLP and fixed-point implementation.

Table 8.2 File listing for the experiment Exp8.2

Files	Description
`fixPoint_nlmsTest.c`	Program for testing adaptive echo canceler
`fixPoint_aec_init.c`	Initialization function
`fixPoint_double_talk.c`	Double-talk detection function
`utility.c`	Utility functions
`fixPoint_nlms.c`	Major module of normalized LMS algorithm
`fixPoint_nlp.c`	NLP function
`utility.c`	Utility function for long division
`fixPoint_nlms.h`	Header file
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`rtfar.pcm`	PCM data file of far-end signal
`rtmic.pcm`	PCM data file of near-end signal

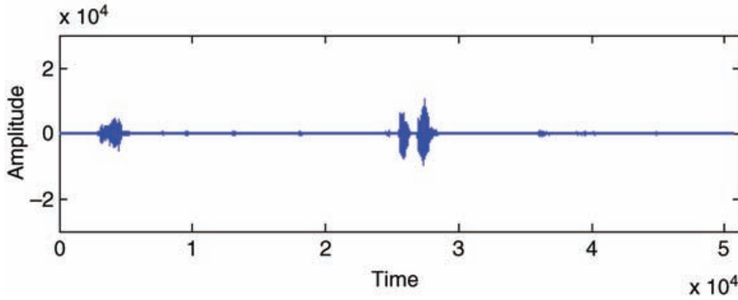


Figure 8.21 Acoustic echo canceler's output using fixed-point C implementation with NLP

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment. Plot the AEC output, $e(n)$, to verify the experiment using the fixed-point C program works properly as shown in Figure 8.21. Compare the result to the floating-point C result obtained in the previous experiment, and explain the difference.
3. Modify the program to disable the NLP function and redo the experiment. Compare the result to the floating-point C result obtained in the previous experiment, and explain the difference mainly due to numerical effects.
4. Change the AEC parameter values (initialized by `fixPoint_aec_init.c`) using the following different settings and evaluate the performance differences:
   (a) Near-end and far-end noise floors, `nfNear` and `nfFar`.
   (b) Training time, `trainTime`.
   Summarize the experiment results and explain what factors affect the AEC performances and why.
5. This experiment uses the normalized LMS algorithm. Modify the experiment to use the fixed-point leaky LMS algorithm. Redo the experiment using the same data files and compare the performance to the normalized LMS algorithm in terms of convergence rate and residual echo level at steady state.
6. The data files used for this experiment do not have double-talk periods. To verify the double-talk detection performance, we can use the data files from the next experiment (or digitize new data sets as explained in step 6 of Exp8.1). Copy the microphone and far-end data files from Exp8.3, and redo the experiment to observe the AEC performance. Pay special attention to the double-talk periods, explain the results, and consider some possible solutions presented in this chapter.

8.7.3 Integration of AEC and Noise Reduction

This experiment integrates AEC with noise reduction (NR) as illustrated in Figure 8.19. The files used for the experiment are listed in Table 8.3. The VAD (voice activity detection) and noise reduction concepts and techniques will be presented in Chapter 9. In this experiment, VAD is used for the noise reduction module to classify the input data into either voice or noise-only frames.

In this experiment, AEC uses sample-based processing while the NR uses frame-based processing. Similar to Figure 8.19, the names of signals used in the program and the processing scheme are illustrated in Figure 8.22.

Table 8.3 File listing for the experiment Exp8.3

Files	Description
floatPoint_aecNr_mainTest.c	Program for testing AEC integrated with NR
floatPoint_nr_vad.c	VAD used by NR
floatPoint_nr_hwindow.c	Generate Hanning window lookup table
floatPoint_nr_proc.c	NR algorithm
floatPoint_nr_ss.c	Data pre-processing before calling VAD
floatPoint_nr_init.c	Noise reduction initialization
floatPoint_nr_fft.c	FFT function and bit reversal
floatPoint_aec_calc.c	AEC algorithm
floatPoint_aec_util.c	AEC supporting functions
floatPoint_aec_init.c	AEC initialization
nr.h	NR C header file
aec.h	AEC C header file
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
micIn.pcm	PCM data file of microphone signal
farIn.pcm	PCM data file of far-end signal

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment. Compare the experiment's output to the microphone (near-end) and far-end speech signals as shown in Figure 8.20 to examine if the program works properly.
3. To verify the AEC and NR performance, we can also use the near-end and far-end signal files from the previous experiments. Copy the data files from Exp8.1 and redo the experiment. Compare the result to the result obtained from Exp8.1; we should observe some improvements in NR by adding the NR module.
4. Replace the floating-point C functions with the fixed-point C program using C55xx intrinsics to conduct fixed-point C experiments. Compare the results of echo cancellation, noise reduction, and computational requirements to the floating-point C results.

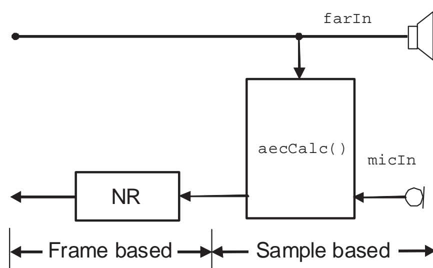


Figure 8.22 Processing scheme and signal flow of AEC and NR

Exercises

8.1. If the adaptive filter used by an adaptive echo canceler diverges during the simulation, how can the possible factors be identified and the problems solved?

8.2. Given the speech file `TIMIT1.ASC` with the sampling rate of 8 kHz, use the measured room impulse response file `roomImp.pcm` from Example 8.4 to produce an acoustic echo. Listen to both the original speech and the acoustic echo and plot both waveforms and spectrograms. Note that the speech file can be loaded using the MATLAB® `load` function as in the example given by `exercise8_2.m`.

8.3. Redo Problem 8.2 using the speech file `TIMIT3.ASC`.

8.4. Use MATLAB® to implement full-band acoustic echo cancellation which uses the original speech as $x(n)$ and the acoustic echo generated in Problem 8.2 as $d(n)$. Use the normalized LMS algorithm for different adaptive FIR filter lengths and step size values. Evaluate the echo cancellation performance by listening to the echo canceler output $e(n)$ and compare it to $d(n)$ and $x(n)$. Also, compare the spectrograms of these signals. Find the filter length that can obtain the best echo cancellation. (Hint: refer to `example6_7.m` on how to use the normalized LMS algorithm supported by MATLAB®.)

8.5. Redo Problem 8.4 by adding an NLP to further process $e(n)$, and compare the results to the results obtained from Problem 8.4 by both listening and using spectrograms.

8.6. Based on Example 8.5, design an analysis filterbank with eight subbands.

8.7. Redo Problem 8.4 using the subband adaptive filter technique presented in Section 8.6.3. Use different numbers of subbands such as M equal to 8, 16, and 32.

8.8. Assume that a full-band adaptive FIR filter is used and the sampling frequency is 8 kHz. Calculate the following factors:
(a) The number of coefficients needed to cover the echo tail of 128 ms.
(b) The number of multiplications needed for updating coefficients (use the filter length obtained in Problem 8.8(a)).
(c) The number of coefficients to cover the echo tail of 128 ms for a sampling rate of 16 kHz.
(d) How to decide a suitable filter length L for a given application?

8.9. In Problem 8.8, the full-band signal is sampled at 8 kHz. If we use a subband technique with 32 subbands, the filter coefficient is real, and the subband signal is critically sampled (i.e., decimation factor $D = 32$). Answer the following questions:
(a) What is the decimated sampling rate at each subband?
(b) What is the minimum number of coefficients for each subband adaptive FIR filter in order to cover the echo tail length of 128 ms?
(c) What is the total number of coefficients for this system? Is this number the same as that in Problem 8.8?
(d) How many multiplications are needed for coefficient adaptation at each sampling period (at 8 kHz rate)? Compare the computational loads to those in Problem 8.8.

8.10. For VoIP applications, for a conventional landline user calling an IP phone user via the VoIP gateway, draw a diagram to show which side will hear a line echo and which side needs an adaptive line echo canceler. Assuming the IP phone user uses a speakerphone and there is an acoustic echo problem, draw a diagram to show which side will hear an acoustic echo and which side needs acoustic echo cancellation. If both sides are using IP phones, is the line echo canceler still needed?

References

1. Gritton, C.W.K. and Lin, D.W. (1984) Echo cancellation algorithms. *IEEE ASSP Mag.*, 30–38.
2. Sondhi, M.M. and Berkley, D.A. (1980) Silencing echoes on the telephone network. *Proc. IEEE*, **68**, 948–963.
3. Sondhi, M.M. and Kellermann, W. (1992) Adaptive echo cancellation for speech signals, in *Advances in Speech Signal Processing* (eds. S. Furui and M. Sondhi), Marcel Dekker, New York, Chapter 11.
4. Kuo, S.M. and Morgan, D.R. (1996) *Active Noise Control Systems – Algorithms and DSP Implementations*, John Wiley & Sons, Inc., New York.
5. Texas Instruments, Inc. (1997) Echo Cancellation S/W for TMS320C54x, BPRA054.
6. Texas Instruments, Inc. (1997) Implementing a Line-Echo Canceller Using Block Update & NLMS Algorithms on the TMS320C54x, SPRA188.
7. ITU-T Recommendation (1993) G.165, Echo Cancellers, March.
8. ITU-T Recommendation (2012) G.168 Digital Network Echo Cancellers, February.
9. Duttweiler, D.L. (1978) A twelve-channel digital echo canceller. *IEEE Trans. Commun.*, **COM-26**, 647–653.
10. Duttweiler, D.L. and Chen, Y.S. (1980) A single-chip VLSI echo canceller. *Bell Syst. Tech. J.*, **59**, 149–160.
11. Tian, W. and Alvarez, A. (2002) Echo canceller and method of canceling echo. World Intellectual Property Organization, Patent WO 02/093774 A1, November.
12. The MathWorks, Inc. (2000) Using MATLAB, Version 6.
13. The MathWorks, Inc. (1992) MATLAB Reference Guide.
14. Lu, Y., Fowler, R., Tian, W., and Thompson, L. (2005) Enhancing echo cancellation via estimation of delay. *IEEE Trans. Signal Process.*, **53** (11), 4159–4168.
15. Tian, W. and Lu, Y. (2004) System and method for comfort noise generation. US Patent No. 6,766,020 B1, July.
16. Texas Instruments, Inc. (1997) Acoustic Echo Cancellation Software for Hands-Free Wireless Systems, SPRA162.
17. ITU-T Recommendation (1993) G.167, Acoustic Echo Controllers, March.
18. ITU-T Recommendation (2012) P.340 Transmission Characteristics and Speech Quality Parameters of Hands-free Terminals, May.
19. Eneman, K. and Moonen, M. (1997) Filterbank constrains for subband and frequency-domain adaptive filters. Proceedings of the IEEE ASSP Workshop.
20. Lee, K.A., Gan, W.S., and Kuo, S.M. (2009) *Subband Adaptive Filtering: Theory and Implementation*, John Wiley & Sons, Ltd, Chichester.

9

Speech Signal Processing

In recent years, communication infrastructures have changed dramatically to include voice, data, image, and video services. However, speech is still the most common and fundamental service provided by the telecommunication networks. With an increased bandwidth to preserve audio signal fidelity, wideband coding is also gaining popularity with improved voice quality. This chapter introduces speech coding techniques, speech enhancement methods, and voice over internet protocol (VoIP) applications for next generation networks.

9.1 Speech Coding Techniques

Speech coding is the digital representation of speech signals to provide efficient transmission and storage of digital data. It includes techniques to compress the digitized speech signals into digital codes and decompress the digital codes to reconstruct the speech signals to a satisfactory quality. Several sophisticated speech coding algorithms have been developed to preserve speech quality while achieving low bit rate. These algorithms usually require higher computational load and more memory for complicated programs and larger signal buffers. The trade-offs among the bit rate, speech quality, coding delay, and algorithm complexity are the main considerations when designing speech systems.

As introduced in Chapter 2, the simplest method to encode speech is to uniformly sample and quantize (digitize) the time-domain speech waveform for digital representation, known as pulse code modulation (PCM). Linear PCM quantization requires at least 12 bits per sample to maintain satisfactory speech quality, called toll quality. Since most telecommunication systems use the 8 kHz sampling rate, PCM coding requires a bit rate of 96 kilobits per second (kbit/s or kbps).

As introduced in Chapter 1, lower bit rate can be achieved by using logarithmic quantization such as μ-law or *A*-law companding (compression and expanding). The basic idea of the nonlinear quantizer is to use variable quantization steps that are proportional to the amplitude of the time-varying speech waveforms. This can be done by using a uniform quantizer to operate on the logarithm of the speech signal. The compressed signal can be reconstructed using the inverse operation, called expanding. Both μ-law and *A*-law

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

companding methods compress speech to 8 bits per sample, thus these methods reduce the bit rate from 96 to 64 kbps.

Further bit-rate reduction to 32 kbps can be achieved by using the adaptive differential PCM (ADPCM) algorithm, which uses an adaptive predictor and differential quantizer to track the changing amplitudes of the speech signals. The basic concept of differential quantization is to quantize the difference between the speech sample and its prediction. Because the amplitude difference will be smaller than the speech sample itself due to the high correlation between consecutive speech samples, fewer bits are required to encode the differences. ADPCM applies the adaptation concepts introduced in Chapter 6 for its adaptive predictor and/or quantizer.

Linear PCM, nonlinear companding (μ-law or A-law), and ADPCM are classified as waveform coding techniques, which operate on the amplitude of speech signals on a sample-by-sample basis. In contrast, the analysis-by-synthesis coding methods process signals on a frame-by-frame basis to achieve a higher compression rate by analyzing and coding the spectral parameters that represent the speech production model. The analysis-by-synthesis coding methods transmit the coded parameters to the receiver for speech synthesis. This type of coding algorithm is called a vocoder (voice coder) since it uses an explicit speech production model. Many vocoders are based on linear predictive coding (LPC) techniques to achieve low bit rates. The LPC coding techniques will be introduced in this chapter.

9.1.1 Speech Production Model Using LPC

The LPC method is based on the speech production model including the excitation input, gain, and vocal-tract filter. The simplified speech production model for an LPC vocoder is illustrated in Figure 9.1. In this model, the vocal tract is considered as a pipe extending from the vocal cords to the oral cavity with a coupled branch leading to the nasal cavity. The oral cavity is the most important component of the vocal tract because its shape always changes to produce different sounds based on the relative positions of the palate, tongue, and lips.

The vocal-tract resonances are called formants. The phonemes are distinguishable from one another by the resonances of the vocal tract. Typically, there are three to four distinguishable formants. As introduced in the IIR filter design, amplification to certain frequencies can be achieved by properly placing the poles in the filter transfer function. The LPC synthesis filter is an all-pole IIR filter to model this vocal-tract transfer function. The filter coefficients can be estimated from the segment of the speech signal by applying the Levinson–Durbin recursive algorithm, which will be introduced in Section 9.1.3.

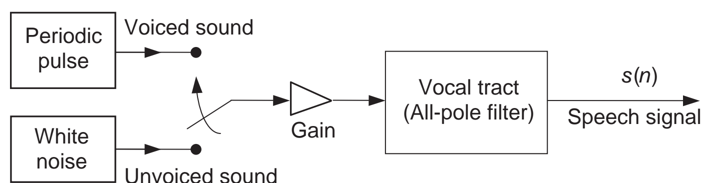


Figure 9.1 Speech production model

When air is pushed from the lung through the vocal tract and radiated through the mouth, it generates different sounds to form speech. For voiced sounds, the vocal cords vibrate (open and close) to produce pseudo-periodic sounds. The rate at which the vocal cords vibrate represents the pitch of the voice. The voiced sounds, for example, include vowels and some consonants. For unvoiced sounds (fricatives and plosive consonants), such as "*s*", "*sh*," and "*f*" sounds in the English language, the vocal cords do not vibrate but remain constantly open. Thus, the voiced or unvoiced sounds can be modeled using periodic pulses or white-noise excitation, respectively. To use the LPC model, the given segment of speech needs to be classified as either voiced or unvoiced. For the voiced signal, estimation of the fundamental frequency (also called pitch frequency) becomes very important. The inverse pitch frequency equals the pitch period, which is usually expressed in samples for the given sampling rate. The pitch period is used to generate the periodic pulse (also called pulse train) for exciting the LPC filter to produce the voiced speech. In addition, the amount of air coming from the lung determines the voice loudness, which is represented by the gain as shown in Figure 9.1.

Several LPC-based speech codecs (coder–decoders), especially code-excited linear predictive (CELP) codecs at the bit rate of 8 kbps or lower, have been developed for wireless and network applications. CELP-type speech codecs are widely used in mobile and IP telephony communications, media streaming services, audio and video conferencing, and digital radio broadcasting. These speech codecs include the 5.3 and 6.3 kbps ITU-T G.723.1 [1] coder for multimedia communications, low-delay G.728 [2] coding at 16 kbps, G.729 [3] coding at 8 kbps, 3GPP (the third-generation partnership project), AMR (adaptive multi-rate)-NB/WB (narrowband/wideband) [4–7], and MPEG-4 CELP coding. Some of these codecs will be introduced in Section 9.1.7.

9.1.2 CELP Coding

As mentioned earlier, vocoders process speech segment by segment (frame by frame), usually in the range of 5 to 30 ms. Segmentation is formed by multiplying the speech signal with a window function such as the Hamming window introduced in Chapter 3. The successive windows are overlapped. In general, the smaller frame size and higher overlap percentage can better capture speech transitions, thus attaining good speech quality.

CELP algorithms are based on the LPC approach using the analysis-by-synthesis scheme. The coded parameters are analyzed to minimize the perceptually weighted error in the synthesized speech via a closed-loop optimization procedure [8,17]. All CELP algorithms share the same basic functions including short-term synthesis filter, long-term pitch synthesis filter (or adaptive codebook), perceptual weighted error minimization procedure, and stochastic (or fixed) codebook excitation [11,12,14]. The basic structure of the CELP algorithm is illustrated in Figure 9.2, where the top portion of the figure is the encoder and the bottom portion is the decoder.

Good-quality synthesized speech can be achieved by optimizing the following three components:

1. The time-varying filters, including the short-term LPC synthesis filter $1/A(z)$, the long-term pitch synthesis filter $P(z)$ (adaptive codebook), and the postfilter $F(z)$ [2,16].

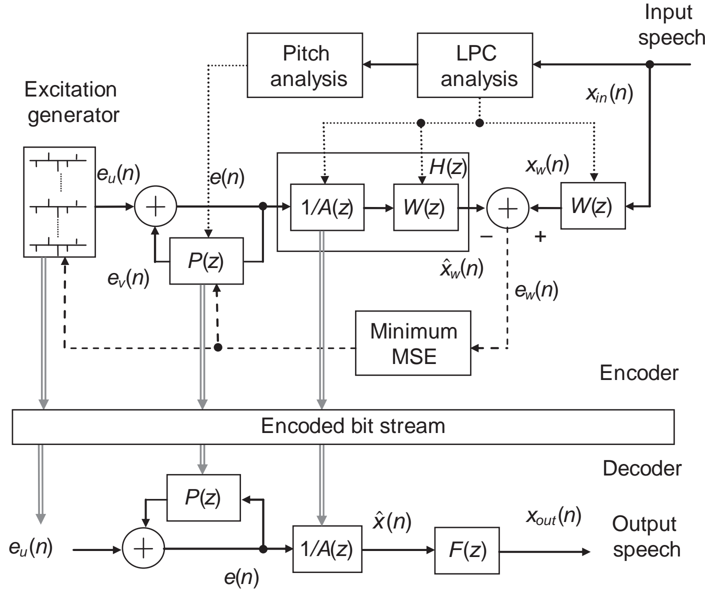


Figure 9.2 Block diagram of typical CELP algorithm

2. The perceptual weighting filter $W(z)$, which is used to optimize the error criterion. A reduction in perceived distortion can be achieved by shaping the noise spectrum to place the majority of errors in the formant regions where human ears are relatively insensitive due to auditory masking. On the other hand, more subjectively disturbing noise in the formant nulls must be reduced. Thus, the perceptual weighting filter will emphasize the weight of the errors between the formant frequencies.
3. The fixed codebook excitation signal $e_u(n)$, including the excitation signal shape and gain.

In the encoder, the LPC and pitch analysis modules analyze speech to estimate the parameters for the speech synthesis model. They are followed by the speech synthesis module to minimize the weighted error. To develop an efficient search procedure, the number of operations can be reduced by placing the weighting filter in the location shown in Figure 9.2. In the encoder, $x_{\mathrm{in}}(n)$ is the input speech, $x_{\mathrm{w}}(n)$ is the original speech weighted by the perceptual weighting filter $W(z)$, $\hat{x}_{\mathrm{w}}(n)$ is the weighted reconstructed speech by passing the excitation signal $e(n)$ through the combined filter $H(z)$, $e_u(n)$ is the excitation signal from the codebook, $e_v(n)$ is the output of pitch predictor $P(z)$, and $e_{\mathrm{w}}(n)$ is the weighted error.

Parameters including the excitation index, quantized LPC coefficients, and pitch predictor coefficients are encoded and transmitted. The decoder will use these parameters to synthesize the speech. The filter $W(z)$ is used only for minimizing the mean-square error so its coefficients are not encoded. The coefficients of the postfilter $F(z)$ are derived from the LPC coefficients and/or from the reconstructed speech.

In the decoder, the excitation signal $e(n)$ is first passed through the long-term pitch synthesis filter $P(z)$ and then the short-term LPC synthesis filter $1/A(z)$. The reconstructed signal $\hat{x}(n)$ is sent to the postfilter $F(z)$ which emphasizes speech formants and attenuates the spectral valleys between formants.

9.1.3 Synthesis Filter

The time-varying short-term synthesis filter $1/A(z)$ is updated frame by frame using the Levinson–Durbin recursive algorithm. The synthesis filter $1/A(z)$ is expressed as

$$1/A(z) = \frac{1}{1 - \sum_{i=1}^{p} a_i z^{-i}}, \tag{9.1}$$

where a_i are the short-term LPC coefficients and p is the filter order. These coefficients are used to estimate the current speech sample from the previous samples.

The widely used method to calculate the LPC coefficients is the autocorrelation method derived by minimizing the total squared errors between the speech samples and their estimates. A window (such as the Hamming window defined in Chapter 3) is applied to form a speech frame and calculate the autocorrelation coefficients based on the speech samples in the frame as

$$r_m(j) = \sum_{n=0}^{N-1-j} x_m(n) x_m(n+j), \quad j = 0, 1, 2, \ldots, p, \tag{9.2}$$

where N is the window (or frame) size, j is the autocorrelation coefficient index, m is the frame index, and n is the sample index inside the frame. The prediction filter coefficients a_i can be calculated by solving the following normal equation:

$$\begin{bmatrix} r_m(0) & r_m(1) & \ldots & r_m(p-1) \\ r_m(1) & r_m(0) & \ldots & r_m(p-2) \\ \vdots & \vdots & \ddots & \vdots \\ r_m(p-1) & r_m(p-2) & \ldots & r_m(0) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_p \end{bmatrix} = \begin{bmatrix} r_m(1) \\ r_m(2) \\ \vdots \\ r_m(p) \end{bmatrix}. \tag{9.3}$$

The matrix on the left is the Toeplitz matrix defined in Chapter 6. This square matrix is guaranteed to be invertible so we can always find the solution for a_i. Several recursive algorithms have been derived for solving (9.3). The most widely used algorithm is the Levinson–Durbin recursion, which can be summarized as follows:

$$E_m(0) = r_m(0) \tag{9.4}$$

$$k_i = \frac{r_m(i) - \sum_{j=1}^{i-1} a_j^{(i-1)} r_m(|i-j|)}{E_m(i-1)} \tag{9.5}$$

$$a_i^{(i)} = k_i \tag{9.6}$$

$$a_j^{(i)} = a_j^{(i-1)} - k_i a_{i-j}^{(i-1)}, \quad 1 \le j \le i-1 \tag{9.7}$$

$$E_m(i) = (1 - k_i^2) E_m(i-1). \tag{9.8}$$

After solving these equations recursively for $i = 1, 2, \ldots, p$, the prediction filter coefficients a_i are obtained as

$$a_i = a_i^{(p)}, \quad 1 \leq i \leq p. \tag{9.9}$$

The gain needed in the synthesis model can be computed as $G_m = \sqrt{E_m(p)}$ to form the synthesis filter $G_m/A(z)$, and the coefficients k_i are the reflection coefficients. A stable LPC filter is guaranteed if all the reflection coefficients satisfy $-1 < k_i < 1$. Also, instead of realizing the LPC filter using a direct-form IIR filter with coefficients a_i, a lattice filter with coefficients k_i can be used. However, the lattice filter is beyond the scope of this book.

Example 9.1

Given the order $p = 3$ and the autocorrelation coefficients $r_m(j)$, $j = 0$, 1, 2, and 3, for a frame of speech signal, calculate the LPC coefficients.

Using (9.3) for the order $p = 3$, the matrix equation

$$\begin{bmatrix} r_m(0) & r_m(1) & r_m(2) \\ r_m(1) & r_m(0) & r_m(1) \\ r_m(2) & r_m(1) & r_m(0) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} r_m(1) \\ r_m(2) \\ r_m(3) \end{bmatrix}$$

can be solved recursively using the Levinson–Durbin algorithm as follows:

For $i = 1$:

$$E_m(0) = r_m(0)$$

$$k_1 = \frac{r_m(1)}{E_m(0)} = \frac{r_m(1)}{r_m(0)}$$

$$a_1^{(1)} = k_1 = \frac{r_m(1)}{r_m(0)}$$

$$E_m(1) = \left(1 - k_1^2\right) r_m(0) = \left[1 - \frac{r_m^2(1)}{r_m^2(0)}\right] r_m(0).$$

For $i = 2$, $E_m(1)$ and $a_1^{(1)}$ are available from $i = 1$. Thus,

$$k_2 = \frac{r_m(2) - a_1^{(1)} r_m(1)}{E_m(1)} = \frac{r_m(0) r_m(2) - r_m^2(1)}{r_m^2(0) - r_m^2(1)}$$

$$a_2^{(2)} = k_2$$

$$a_1^{(2)} = a_1^{(1)} - k_2 a_1^{(1)} = (1 - k_2) a_1^{(1)}$$

$$E_m(2) = \left(1 - k_2^2\right) E_m(1).$$

For $i = 3$, $E_m(2)$, $a_1^{(2)}$, and $a_2^{(2)}$ are available from $i = 2$. Thus,

$$k_3 = \frac{r_m(3) - \left[a_1^{(2)} r_m(2) + a_2^{(2)} r_m(1)\right]}{E_m(2)}$$

$$a_3^{(3)} = k_3$$

$$a_1^{(3)} = a_1^{(2)} - k_3 a_2^{(2)}$$

$$a_2^{(3)} = a_2^{(2)} - k_3 a_1^{(2)}.$$

Finally,

$$a_0 = 1, \quad a_1 = a_1^{(3)}, \quad a_2 = a_2^{(3)}, \quad \text{and} \quad a_3 = a_3^{(3)}.$$

MATLAB® provides two functions (`levinson` and `lpc`) to calculate LPC coefficients. For example, the LPC coefficients can be calculated using the Levinson–Durbin recursion function `[a,e] = levinson(r,p)`. The parameter `r` is the vector containing autocorrelation coefficients, `p` is the order of $A(z)$, `a` $= [1\ a_1\ a_2\ \ldots\ a_p]$ is the LPC coefficients vector, and `e` is the prediction error.

The MATLAB® function `lpc` computes the LPC coefficients using the Levinson–Durbin recursion to solve the normal equation (9.3). The function `[a,g] = lpc(x,p)` computes `p` LPC coefficients based on the data samples in the vector `x`. This function returns the LPC coefficients in the vector `a` and the variance (power) of the prediction error in `g`.

Example 9.2

Given LPC order `p` = 10, Hamming window size 256, and speech file `voice4.pcm`, calculate the LPC coefficients using the function `levinson()`, compute the magnitude response of the synthesis filter $1/A(z)$, and compare the magnitude response of the speech spectrum to the contours.

The complete MATLAB® program is given in `example9_2.m`. The magnitude response of the synthesis filter represents the envelope of the corresponding speech magnitude spectrum as shown in Figure 9.3.

Example 9.3

Calculate the LPC coefficients of the same speech file given in Example 9.2 using the function `lpc` instead of `levinson`. In this case, `a` (LPC coefficients) and `g` (error variance) will be identical to the results from Example 9.2 if using the same order `p` (= 10).

Figure 9.4 shows the result of using a higher order filter (42 instead of 10 in Example 9.2) to calculate the LPC coefficients. By comparing Figure 9.4 to Figure 9.3, it is clear that the magnitude response of the synthesis filter can better match the speech spectrum with a high-order synthesis filter. The complete MATLAB® script is given in `example9_3.m`.

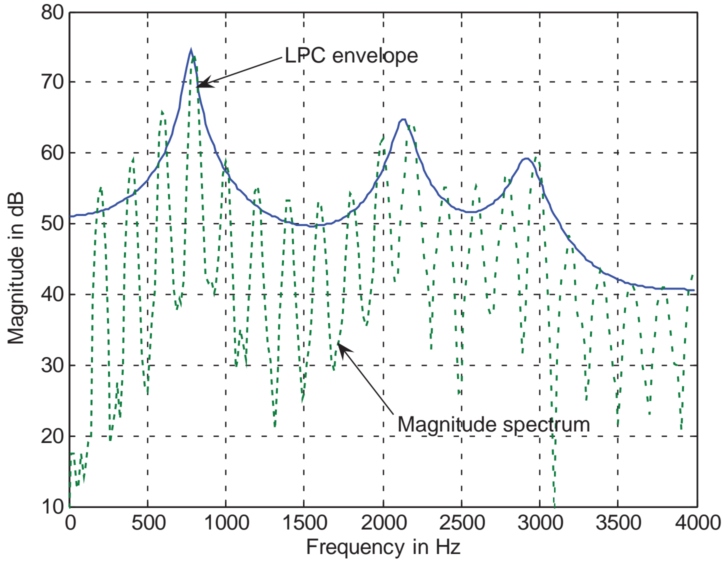


Figure 9.3 Magnitude spectrum of speech and its spectral envelope derived from the synthesis filter using LPC coefficients

9.1.4 Excitation Signals

Excitation signals can be classified into two different types: long term and short term. The short-term excitation signal is a stochastic signal while the long-term excitation signal is a period signal.

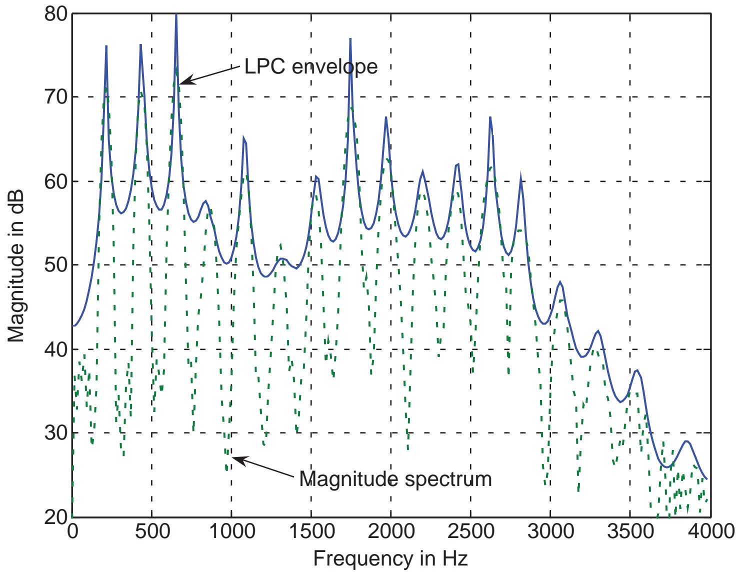


Figure 9.4 Magnitude response of high-order synthesis filter matches closely the magnitude spectrum of the speech

Long-Term and Short-Term Excitation Signals

The long-term prediction (pitch synthesis) filter $P(z)$ models the long-term correlation of the speech to provide a fine spectral structure and has the following general form:

$$P(z) = \sum_{i=-I}^{I} b_i z^{-(L_{\text{opt}}+i)}, \tag{9.10}$$

where L_{opt} is the optimum pitch period, b_i are the filter coefficients, and I determines the filter length. For example, $I=0$ for one-tap, $I=1$ for three-tap, and $I=2$ for five-tap pitch filters.

In some cases, the long-term prediction filter is also called the adaptive codebook since the excitation signals are adaptively updated. This filter can be found in ITU-T G.723.1, which uses a five-tap (i.e., $I=2$) long-term prediction filter.

For efficient temporal analysis, the speech frame is usually divided into several subframes. For example, there are two subframes defined in G.729 and four defined in G.723.1, AMR-NB, and AMR-WB. The excitation signal is generated for each subframe and the prediction error is minimized to find the optimum excitation signal for each subframe.

As discussed earlier, the excitation signal varies between the periodic pulse train and random noise. The general form of the excitation signal $e(n)$ shown in Figure 9.2 can be expressed as

$$e(n) = e_v(n) + e_u(n), \quad 0 \le n \le N_{\text{sub}} - 1, \tag{9.11}$$

where $e_u(n)$ is the excitation from the fixed or stochastic codebook given by

$$e_u(n) = G_u c_k(n), \quad 0 \le n \le N_{\text{sub}} - 1, \tag{9.12}$$

where G_u is the gain, N_{sub} is the length of the excitation vector (or the subframe), and $c_k(n)$ is the nth element of the kth vector in the codebook. In (9.11), $e_v(n)$ is the excitation signal from the long-term prediction filter $P(z)$ and expressed as

$$e_v(n) = \sum_{i=-I}^{I} b_i e(n + i - L_{\text{opt}}), \quad n = 0, 1, \ldots, N_{\text{sub}} - 1. \tag{9.13}$$

Passing $e(n)$ through the combined filter $H(z)$, as shown in Figure 9.2, the perceptually weighted synthesis speech can be expressed as

$$\hat{x}_{\text{w}}(n) = v(n) + u(n) = \sum_{i=0}^{n} h_i e_v(n-i) + \sum_{i=0}^{n} h_i e_u(n-i), \quad n = 0, 1, \ldots, N_{\text{sub}} - 1, \tag{9.14}$$

where $v(n) = \sum_{i=0}^{n} h_i e_v(n-i)$ is generated by the long-term predictor and $u(n) = \sum_{i=0}^{n} h_i e_u(n-i)$ is from the stochastic codebook. Therefore, the weighted error $e_{\text{w}}(n)$ can be described as

$$e_{\text{w}}(n) = x_{\text{w}}(n) - \hat{x}_{\text{w}}(n). \tag{9.15}$$

The squared error of $e_{\text{w}}(n)$ is computed as

$$E_{\text{w}} = \sum_{n=0}^{N_{\text{sub}}-1} e_{\text{w}}^2(n). \tag{9.16}$$

By computing the above equations for all possible parameters including the pitch predictor coefficients b_i (pitch gain), lag L_{opt} (delay), optimum stochastic excitation codebook vector $\mathbf{C}_k$ with the elements $c_k(n)$, $n=0, \ldots, N_{sub}-1$, and gain G_u, the minimum E_w^{min} can be obtained as

$$E_w^{min} = \min\{E_w\} = \min\left\{\sum_{n=0}^{N_{sub}-1} e_w^2(n)\right\}. \tag{9.17}$$

This minimization procedure is a joint optimization of the pitch prediction (adaptive codebook) and the stochastic (fixed codebook) excitations. Although a joint optimization for all the excitation parameters, including L_{opt}, b_i, G_u, and $\mathbf{C}_k$, is possible, it will need intensive computation. A significant simplification can be achieved by optimizing the pitch prediction parameters independently of the stochastic codebook excitation parameters. This is called the separate optimization procedure.

In separate optimization, the excitation $e(n)$ in (9.11) only contains $e_v(n)$ because $e_u(n) = 0$. The optimized pitch lag and pitch gain are first found using the historical excitations. The new target signal can be obtained by subtracting the contribution of the pitch prediction $v(n)$ from the target signal $x_w(n)$. The second round of minimization is carried out by approximating the stochastic codebook contribution to this new target signal. The separate optimization procedure is used by the G.729 and G.723.1 standards.

Algebraic CELP

Algebraic CELP (ACELP) uses the structured codebook to efficiently find the best excitation codebook vector. This structured codebook vector consists of a set of interleaved permutation codes containing a few non-zero elements. The ACELP fixed codebook structure is used by the G.729, G.723.1 (5.3 kbps), AMR-NB, and AMR-WB standards for achieving low bit rate. Table 9.1 shows the ACELP codebook structure used by the G.729 standard.

In Table 9.1, m_k is the pulse position, k is the pulse number, and the interleaving depth is 5. In this codebook, each vector contains four non-zero pulses indexed by i_k. Each pulse can have the amplitude of $+1$ or -1 and can have the positions as listed in Table 9.1. The codebook vector, $\mathbf{c}_k$, is determined by placing four unit pulses at the locations m_k multiplied by their sign s_k ($+1$ or -1) as follows:

$$c_k(n) = s_0\delta(n-m_0) + s_1\delta(n-m_1) + s_2\delta(n-m_2) + s_3\delta(n-m_3), \quad n = 0, 1, \ldots, 39, \tag{9.18}$$

where $\delta(n)$ is the unit pulse.

Table 9.1 G.729 ACELP codebook

Pulse i_k	Sign s_k	Position m_k	Number of bits (sign + position)
i_0	± 1	0, 5, 10, 15, 20, 25, 30, 35	1 + 3
i_1	± 1	1, 6, 11, 16, 21, 26, 31, 36	1 + 3
i_2	± 1	2, 7, 12, 17, 22, 27, 32, 37	1 + 3
i_3	± 1	3, 8, 13, 18, 23, 28, 33, 38, 4, 9, 14, 19, 24, 29, 34, 39	1 + 4

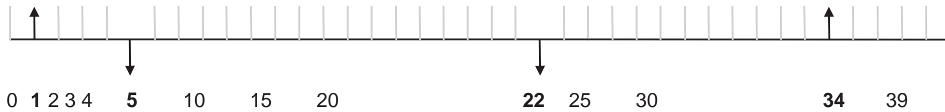


Figure 9.5 Four pulse locations in 40-sample frame

Example 9.4

Assume four pulses are presented in a frame as shown in Figure 9.5. The pulse locations and signs are obtained by minimizing the squared error defined by (9.16). The pulse positions are confined within certain positions with interleaving depth 5 as defined in Table 9.1. As an example, the second pulse shown in Figure 9.5 is located at position 5, which falls into the i_0 group. For this group, it needs 1 bit to code the sign and 3 bits to code eight possible positions. All the pulse positions and signs can be coded into codewords as shown in Table 9.2.

The sign bit is the MSB of the codeword: the positive sign is encoded with 0, and the negative sign is encoded with 1. This example shows how to encode ACELP excitations. To code this frame's ACELP information, a total of $4 + 4 + 4 + 5 = 17$ bits will be needed.

9.1.5 Perceptual Based Minimization Procedure

The synthesis filter defined in (9.1) can be used to construct the perceptual weighting filter with the following transfer function:

$$W(z) = \frac{1 - \sum_{i=1}^{p} a_i z^{-i} \gamma_1^i}{1 - \sum_{i=1}^{p} a_i z^{-i} \gamma_2^i} = \frac{A(z/\gamma_1)}{A(z/\gamma_2)}, \tag{9.19}$$

where $0 < \gamma_i < 1$ is the bandwidth expansion factor with typical values $\gamma_1 = 0.9$ and $\gamma_2 = 0.5$.

The synthesis filter $1/A(z)$ and perceptual weighting filter $W(z)$ can be combined to form the cascaded filter expressed as

$$H(z) = W(z)/A(z). \tag{9.20}$$

As shown in (9.14), the coefficients of the combined (cascaded) filter are denoted as h_i, $i = 0, 1, \ldots, N_{\text{sub}} - 1$, where N_{sub} is the subframe length.

Table 9.2 Example of coding ACELP pulses into codeword

Pulse i_k	Sign s_k	Position m_k	Sign + position = encoded code
i_0	+1	5 (k = 1)	0 << 3 + 1 = 0001
i_1	−1	1 (k = 0)	1 << 3 + 0 = 1000
i_2	−1	22 (k = 4)	1 << 3 + 4 = 1100
i_3	+1	34 (k = 14)	0 << 4 + 14 = 01110

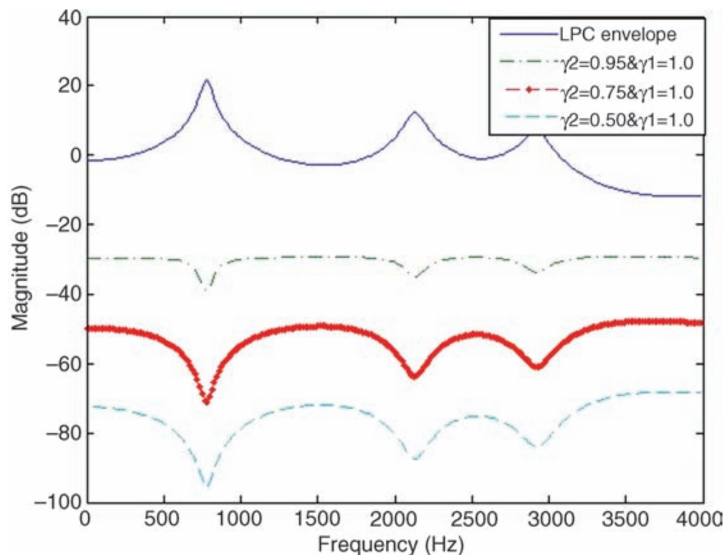


Figure 9.6 LPC envelope and the magnitude responses of the weighting filters

Example 9.5

Use the same speech file `voice4.pcm` given in Example 9.2 to calculate the coefficients of the perceptual weighting filter $W(z)$ for $\gamma_1 = 1.0$ and $\gamma_2 = 0.95$, 0.70, and 0.50.

The LPC coefficients can be calculated first and the bandwidth expansion factor γ_i is applied to compute the weighting filter transfer function defined in (9.19). The complete MATLAB® program is given in `example9_5.m`. Figure 9.6 shows the LPC envelope for the speech spectrum and the magnitude responses of three perceptual weighting filters with different bandwidth expansion factors. These magnitude responses show that the weighting filter with smaller values of γ_2 will allow more distortion at the formant frequencies.

9.1.6 Voice Activity Detection

Voice activity detection (VAD) is one of the most important functions for reducing the bandwidth in speech coding. VAD can be modified for use with the double-talk detector, which is a critical function for adaptive echo cancellation (introduced in Chapter 8). VAD is also a key technique for noise reduction [9], which will be discussed in Section 9.2. The basic assumptions for the VAD algorithm are: (1) the spectrum of the speech signal changes in a short time but the background noise is relatively stationary; and (2) the level (energy) of active speech is usually higher than the background noise. Several techniques have been developed for VAD including the voiced/unvoiced classification used by ITU-T G.723.1 and the zero-crossing method and spectral comparison scheme used by G.729. Some of these methods can be combined with power threshold validations to achieve a better performance for some specific applications.

In practical speech applications, the input signal is usually filtered by a highpass filter to remove the undesired low-frequency noise components. Let $X(k)$ be the FFT bins

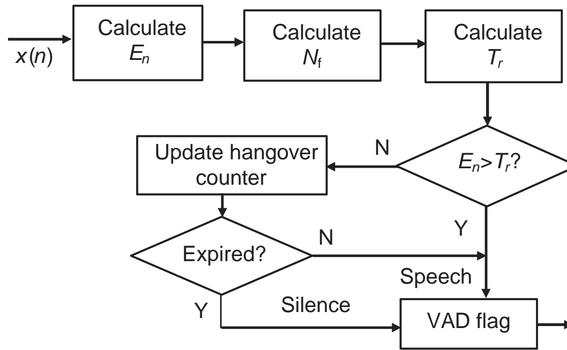


Figure 9.7 Block diagram of simple VAD algorithm

(coefficients) of the input signal $x(n)$; the FFT bins that correspond to the frequency range from 300 to 1000 Hz are usually used to estimate the power with different window sizes as shown in Figure 9.7. The reason for choosing a frequency range from 300 to 1000 Hz is because these FFT bins usually have relatively high speech energy. The VAD algorithm is depicted in Figure 9.7.

The partial signal energy E_n is calculated as

$$E_n = \sum_{k=K_1}^{K_2} |X(k)|^2, \tag{9.21}$$

where K_1 and K_2 are the nearest integers (of frequency index k) to 300 and 1000 Hz, respectively. The energy of the signal in a short window is estimated recursively as

$$E_s(j) = (1 - \alpha_s)E_s(j-1) + \alpha_s E_n, \tag{9.22}$$

and a long window signal energy is estimated as

$$E_l(j) = (1 - \alpha_l)E_l(j-1) + \alpha_l E_n, \tag{9.23}$$

where j is the frame index, and α_s and α_l are the inverse window lengths for the short and long windows used. For example, $\alpha_s = 1/16$ and $\alpha_l = 1/128$ are equivalent to window lengths of 16 and 128, respectively.

The noise level (noise floor) at the nth frame, N_f, is also updated recursively based on its previous value and the current energy E_n with different ratios controlled by α_s or α_l as follows:

$$N_f = \begin{cases} (1-\alpha_l)N_f + \alpha_l E_n, & \text{if } N_f < E_s(j) \\ (1-\alpha_s)N_f + \alpha_s E_n, & \text{if } N_f \geq E_s(j) \end{cases}. \tag{9.24}$$

This equation increases the noise floor slowly at the beginning of the speech, but it will decrease the noise floor faster at the end of the speech to better match the amplitude envelope of the speech waveform. The variable threshold T_r used for the signal energy comparison can

be updated as

$$T_r = \frac{N_f}{1-\alpha_1} + \beta, \tag{9.25}$$

where β is a safety margin to avoid toggling between voice and silence if the noise level is flat.

The signal energy is compared to the threshold to detect if speech is present (VAD = 1) in current frame as follows:

$$\text{VAD flag} = \begin{cases} 1, & \text{if } E_n > T_r \\ 1, & \text{if } E_n \le T \text{ and hangover not expired} \\ 0, & \text{if } E_n \le T \text{ and hangover expired.} \end{cases} \tag{9.26}$$

The hangover period is used for smoothing transition from active speech to silence to avoid false detection of silences at the tail end of the speech (which has relatively lower energy). During the tail period of the speech and before the hangover counter has been expired, the signal frames are classified as active speech. The typical hangover time is about 90 ms.

For non-stationary noise with changing levels and low-level speech, this VAD algorithm may not work properly, especially in the tail of the speech segments. It may need extra measurements such as the zero-crossing rate and voiced segment detection. More details can be found in the G.729 and AMR standards.

9.1.7 ACELP Codecs

This section discusses some popular ACELP codecs such as ITU-T G.729, G.722.2, and 3GPP AMR.

Overview of ITU-T G.729

G.729 is a low-bit-rate, toll-quality, speech codec using the CS (conjugate structure) ACELP-based algorithm. The coder operates on 10 ms frames (80 samples) at the 8 kHz sampling rate. Each frame is further divided into two 40-sample subframes. For every frame, the speech signal is processed to extract the CELP parameters including the LPC coefficients and the adaptive and fixed codebook indices and gains. These parameters are quantized and encoded, then either transmitted or stored. The block diagram of the G.729 encoder is shown in Figure 9.8.

As illustrated in Figure 9.9, the LPC analysis window uses 6 subframes, where 120 samples are coming from past speech frames, 80 samples from the present speech frame, and 40 samples from the future subframe. The future 40-sample subframe is used for 5 ms look-ahead in the LPC analysis that introduces an extra 5 ms of encoding delay.

Calculating the LPC coefficients requires the past, current, and future subframes. The LPC synthesis filter is defined as

$$\frac{1}{A_m(z)} = \frac{1}{1 - \sum_{i=1}^{10} a_{i,m} z^{-i}}, \tag{9.27}$$

where the subframe index $m = 0, 1$.

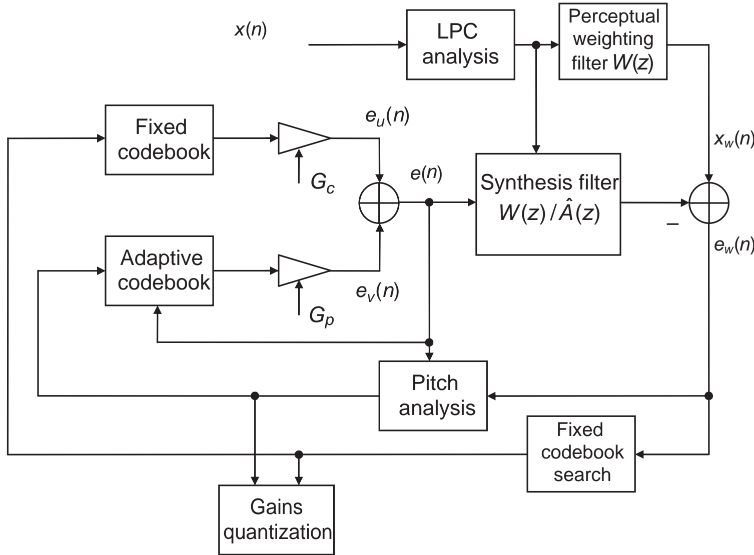


Figure 9.8 Block diagram of the CS-ACELP encoder (adapted from G.729)

The LPC filter coefficients of the last subframe are converted to line spectrum pairs (LSP) coefficients. The LSP is related to the poles of the LPC filter $1/A(z)$, or the zeros of $A(z)$. The reasons for converting LPC coefficients to LSP are because the LSP coefficients can be used to verify the stability of the filter and have higher correlation. The first property can be used to make the synthesis filter stable after quantization and the second property can be used to further remove the redundancy.

The conversion from LPC to LSP coefficients is by decomposing the pth-order analysis filter into two parts as follows:

$$\begin{aligned} A(z) = 1 - \sum_{i=1}^{p} a_i z^{-i} &= \frac{1}{2}\left\{\left[A(z) - z^{-(p+1)}A(z^{-1})\right] + \left[A(z) + z^{-(p+1)}A(z^{-1})\right]\right\} \\ &= \frac{1}{2}\left[S^{p+1}(z) + Q^{p+1}(z)\right], \end{aligned} \tag{9.28}$$

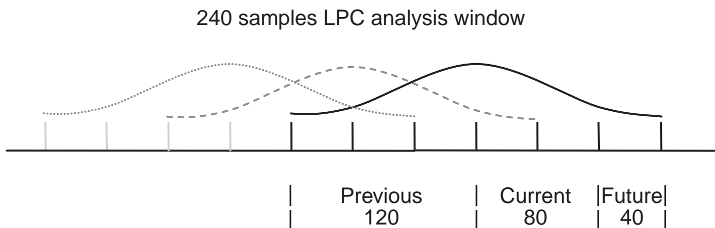


Figure 9.9 LPC analysis window operations

where the first part $S^{p+1}(z) = A(z) - z^{-(p+1)}A(z^{-1})$ is the differential transfer function and the second part $Q^{p+1}(z) = A(z) + z^{-(p+1)}A(z^{-1})$ is the sum transfer function; both are of order $p+1$.

Given the fact that $S^{p+1}(+1) = 0$ and $Q^{p+1}(-1) = 0$, the order of $S^{p+1}(z)$ and $Q^{p+1}(z)$ from $p+1$ to p can be reduced by introducing two new transfer functions $S(z)$ and $Q(z)$ defined as follows:

$$S(z) = S^{p+1}(z)/(1-z) = A_0 z^p + A_1 z^{(p-1)} + \cdots + A_p \tag{9.29a}$$

$$Q(z) = Q^{p+1}(z)/(1+z) = B_0 z^p + B_1 z^{(p-1)} + \cdots + B_p. \tag{9.29b}$$

The coefficients of A_i and B_i can be derived from the analysis filter coefficients a_i recursively as follows:

$$A_0 = 1 \tag{9.30a}$$

$$B_0 = 1 \tag{9.30b}$$

$$A_i = (a_i - a_{p+1-i}) + A_{i-1}, \quad \text{for } i = 1, \ldots, p, \tag{9.30c}$$

$$B_i = (a_i + a_{p+1-i}) + B_{i-1}, \quad \text{for } i = 1, \ldots, p. \tag{9.30d}$$

As the $S(z)$ coefficients are antisymmetric and the $Q(z)$ coefficients are symmetric, it can be proven that all roots of $S(z)$ and $Q(z)$ are on the unit circle and alternate with each other. Each polynomial has five complex-conjugate roots on the unit circle at $e^{\pm j\omega_i}$, and they can be written as

$$S(z) = \prod_{i=2,4,\ldots,p} \left(1 - 2q_i z^{-1} + z^{-2}\right) \tag{9.31a}$$

$$Q(z) = \prod_{i=1,3,\ldots,p-1} \left(1 - 2q_i z^{-1} + z^{-2}\right), \tag{9.31b}$$

where $q_i = \cos(\omega_i)$. The coefficients q_i are LSP coefficients and the coefficients ω_i are the line spectral frequencies with the following ordering property:

$$0 < \omega_1 < \omega_2 < \cdots < \omega_p < \pi. \tag{9.32}$$

G.729 uses very specific methods to find the LSP coefficients by evaluating the polynomials $Q(z)$ and $S(z)$ defined in (9.29). G.729 performs LSP coefficient vector quantization, LSP coefficient interpolation, and LSP to LPC coefficient conversion for each subframe.

In general, the LPC coefficient calculation and quantization are summarized in the following procedures:

1. Compute the 10th-order LPC coefficients $a_{i,m}$ ($i = 1, \ldots, 10$) for both subframes ($m = 0, 1$). Realize (9.27) using LPC coefficients $a_{i,m}$. These unquantized LPC coefficients are used

for the short-term perceptual weighting filter $W_m(z)$ which filters the entire frame to obtain the perceptually weighted speech signal.
2. Convert the last subframe's LPC coefficients to LSP coefficients.
3. Use vector quantization to quantize these 10 LSP coefficients to get the LSP indices for transmission or local storage.
4. Dequantize the LSP coefficients from the vector indices to find the LSP coefficients. Use the LSP coefficients obtained from the current and the last frame to interpolate the LSP coefficients for each subframe.
5. Convert the dequantized LSP coefficients back to the LPC coefficients and construct the synthesis filter $1/\hat{A}_m(z)$ for each subframe. Note that, even in the encoder side, this is required in order to provide the same set of synthesis filters for the decoder. However, the decoder side never has unquantized LPC coefficients.
6. The reconstructed and combined synthesis filter and perceptual weighting filter are used for the synthesis filter, $W(z)/\hat{A}(z)$ in Figure 9.6.

As shown in Figure 9.8, the excitation signal is the sum of the signals from the fixed codebook vector $e_u(n)$ and the adaptive codebook vector $e_v(n)$, which are determined for each subframe. The closed-loop pitch analysis is used to find the adaptive codebook delay with fractional resolutions and the gain. Different from the long-term predictor $P(z)$ in Figure 9.2, the fractional pitch is used to accurately represent the periodic signals. In the first subframe, the fractional pitch delay T_1 with a resolution of 1/3 is used in the range of $[19\frac{1}{3}, 84\frac{2}{3}]$ and the integer delay is used in the range of [85, 143]. For the second subframe, only the fractional delay T_2 with a resolution of 1/3 pitch delay is used. The pitch delay is encoded using 8 bits for the first subframe, but differentially encoded using 5 bits for the second subframe.

A 17-bit algebraic codebook is used for the fixed codebook excitation. For the algebraic codebook, each vector contains four non-zero pulses. The bit allocation for the algebraic codebook is listed in Table 9.1. The 40-sample vector with four unit pulses is defined in (9.18). The adaptive codebook gain (G_p) and the fixed codebook gain (G_c) are vector quantized to 7 bits with moving-average prediction being applied to the fixed codebook gain.

The decoder block diagram is shown in Figure 9.10. The excitation and synthesis filter parameters are retrieved from the received bit stream. First, the parameters' indices are extracted from the bit stream. These indices are decoded to obtain the parameters corresponding to a 10 ms speech frame. The parameters include the LSP coefficients, two fractional

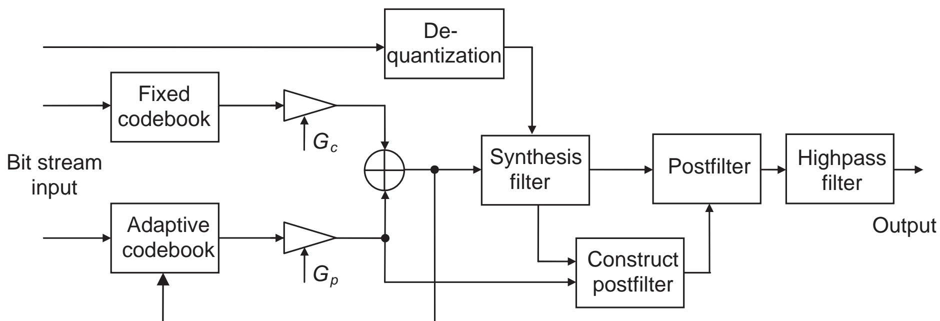


Figure 9.10 Block diagram of G.729 decoder

Table 9.3 Summary of G.729 bit rates of different annexes

Annexes	Annex A or C	Annex D	Annex E	SID[a] (Annex B)	Silence (Annex B)
Bits per 10 ms frame	80	64	118	16	0
Bit rate (kbps)	8	6.4	11.8	1.6	0

[a]SID = Silence insertion descriptor.

pitch delays, two fixed codebook vectors, and two sets of adaptive and fixed codebook gains. The LSP coefficients are interpolated and converted to the LPC filter coefficients for each subframe. For each 5 ms subframe, the following processes will be performed:

- The excitation is constructed by adding the adaptive codebook vector with the fixed codebook vector scaled by their respective gains.
- The speech is reconstructed by filtering the excitation signal using the LPC synthesis filter.
- The reconstructed speech signal passes through the post-processing stage that includes the adaptive postfilter based on the long-term and short-term synthesis filters, and followed by a highpass filter with a cutoff frequency of 100 Hz.

The related annexes for the G.729 standard are used for different applications. The bit-rate information on these annexes is listed in Table 9.3.

Overview of AMR

The AMR codec is a speech coding standard introduced by 3GPP for toll-quality speech for mobile telephony applications; 3GPP AMR is also called AMR-NB (narrowband). The AMR codec operates at eight different bit rates: 12.2, 10.2, 7.92, 7.40, 6.70, 5.90, 5.15, and 4.75 kbps. The data rate is selectable at runtime by the system.

The AMR vocoder processes signals in 20 ms frames. For compatibility with other legacy systems, the 12.2 and 7.4 kbps modes are compatible versions with the GSM-enhanced full-rate codec (GSM EFR) and the North American TDMA (Interim Standard 136) digital cellular system enhanced full-rate codec, respectively. In addition, the AMR codec is designed to allow seamless switching on a frame-by-frame basis between the following different modes:

1. Channel mode: GSM half-rate operation at 9.4 kbps or full-rate operation at 22.8 kbps.
2. Channel mode adaptation: full-rate or half-rate channel mode control and selection.
3. Codec mode: speech bit rate partitioned for a given channel mode.
4. Codec mode adaptation: bit-rate control and selection by the system.

The AMR speech codec provides high-quality speech service with the flexibility to support multi-rate, thus allowing trade-offs between voice quality and network capacity. This flexibility is applicable to the second and third generations of mobile networks. Similar to the G.723.1 and G.729 algorithms, AMR also uses the ACELP technique. Therefore, its rate adaptation for application in wireless networks rather than its detailed algorithm will be presented here.

Figure 9.11 Block diagram of AMR system (adapted from GSM 06.71, version 7.0.2)

In Figure 9.11, the speech encoder input is either the 13-bit PCM signal from the mobile station, or the 13-bit linear PCM signal through 8-bit expanding (A-law/μ-law) from the public switched telephone network (PSTN). The encoder encodes the PCM signal to bit stream according to the bit rate controlled by the transcoding and rate adapter unit (TRAU). The bit rates of the source codec for the adaptive multi-rate full-rate and half-rate channels are listed in Table 9.4. The encoded bit stream is sent to the channel coding function to produce the encoded block. Full-rate bit stream has 456 bits and half-rate has 228 bits.

The inverse operations are performed at the receiving end. The GSM 06.90 specification describes the detailed mapping between the 160 input speech samples in 13-bit PCM format to the encoded bit stream. The decoder will reconstruct these 160 speech samples from the bit stream.

Figure 9.11 presents the block diagram of the high-level AMR system including the uplink and downlink radio channel, where DL, UL, and Rx denote downlink, uplink, and received, respectively. GSM 05.09 defines the AMR link adaptation, channel quality estimation, and in-band signaling. The AMR multi-rate adaptation is based on the UL and DL quality indicators from the radio channel quality measurements. The UL quality indicator is mapped to the UL

Table 9.4 Summary of speech bit rate, full rate, and half rate

		Encoded speech bit rate							
Speech bit rate (kbps)		12.2	10.2	7.95	7.40	6.70	5.90	5.15	4.75
Encoded bits per 20 ms frame		244	204	159	148	134	118	103	95
Gross bit rate with channel coding	Full-rate mode	22.8	22.8	22.8	22.8	22.8	22.8	22.8	22.8
	Half-rate mode			11.4	11.4	11.4	11.4	11.4	11.4

mode command and the DL quality indicator is mapped to the DL mode request. The UL mode command and the DL mode request are sent to the transmitter using the communication reverse link. The UL codec mode request and DL codec mode request are sent as in-band signals in the UL radio channel. The DL codec mode command and UL mode command are sent as in-band signals in the DL channel.

The GSM system uses the AMR speech codec and may select the optimum channel (half or full rate) and codec modes (speech and channel bit rates) to deliver the best combination of speech quality and system capacity. This flexibility provides many important benefits.

Improved speech quality in both half-rate and full-rate modes can be achieved by codec mode adaptation, that is, by balancing between the speech and channel coding for the same gross bit rate. When the radio channel quality is good, the channel bit error rate is low so the higher bit-rate codec will be used. When the radio channel quality becomes worsened, the lower bit-rate codec will be used while the overall channel bit rate remains unchanged. Table 9.4 lists the bit allocations for all the rates.

Overview of AMR-WB

Wideband AMR (AMR-WB) is named for 3GPP GSM AMR-WB and also specified as ITU-T G.722.2. Wideband AMR is an adaptive multi-rate wideband codec with bit rates ranging from 6.6 to 23.85 kbps as listed in Table 9.5. The coder works on speech sampled at 16 kHz with a frame size of 20 ms (320 speech samples). AMR-WB is mainly used for wideband telephony services. Its VAD module supports discontinuous transmission and comfort noise generation (CNG).

For AMR-WB, the wideband speech is divided into two subbands: the lower subband from 75 Hz to 6.4 kHz and the upper subband from 6.4 to 7.0 kHz. The simplified encoding/decoding block diagram is illustrated in Figure 9.12, where HB denotes higher subband.

The low-band encoder and decoder share the similar CELP coding techniques described in Section 9.1.2. The ACELP model is used to provide the large codebook and low-complexity searching procedure, which is similar to G.729. The LPC analysis filter uses order 16 because of the wider bandwidth at 6.4 kHz as compared to the narrowband codecs such as G.729 and AMR-NB which use a 10th-order LPC filter. The low-band decoder performs the reverse process as the encoder. Down-sampling and up-sampling processing blocks (discussed in Chapter 3) are integrated with analysis and synthesis filterbanks to split the full-band signal for subband processing and synthesize the subband signals to form the full-band signal [10,13].

The high-frequency band (6.4 to 7 kHz) is encoded using the silence compression method in the 23.85 kbps mode. In this mode, the high-band excitation signal is random noise and the gain is encoded using 4 bits per subframe. For other models, no information about the high-band will be transmitted.

For the AMR-WB decoder, the high band is reconstructed using the low-band parameters and the 16 kHz random noise excitation. In the 23.85 kbps mode, high-band gain is decoded,

Table 9.5 Source bit rates for the AMR-WB codec

Bits per 20 ms	477	461	397	365	317	285	253	177	132	35
Bit rate	23.85	23.05	19.85	18.25	15.85	14.25	12.65	8.85	6.6	1.75[a]

[a]Assuming SID frames are continuously transmitted.

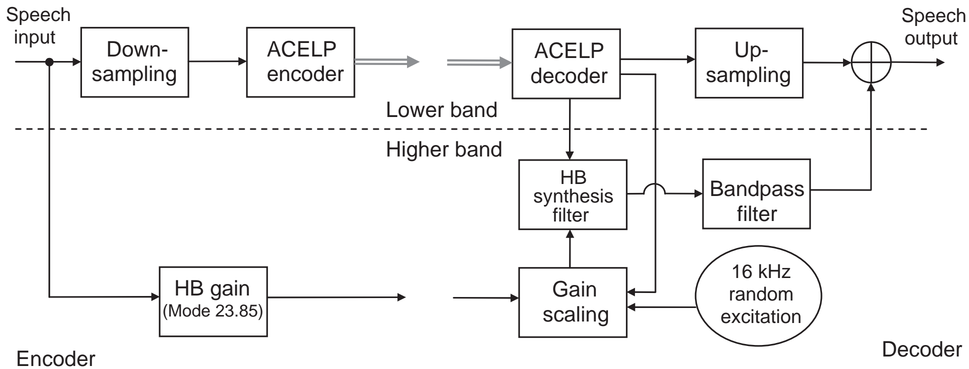


Figure 9.12 Block diagram of AMR-WB

and then applied to the reconstructed signal. In other modes, the high-band gain is derived from the low-band parameters along with the voicing information. The spectrum of the high band is reconstructed using the wideband LPC synthesis filter generated from the low-band LPC filter.

The modes at bit rates of 12.65 kbps and higher offer better quality for wideband speech. Two of the lowest modes, 8.85 and 6.6 kbps, are used only temporarily during severe radio channel conditions or during network congestion. AMR-WB also includes the background noise mode, which is designed to be used in discontinuous transmission operation in GSM and as the low-bit-rate source-dependent mode for coding background noise for other systems. In GSM, the bit rate of this mode is 1.75 kbps. More details can be found in 3GPP technical specification TS 26.290.

9.2 Speech Enhancement

Cell (mobile) phones are often used in noisy environments, such as vehicles, restaurants, shopping malls, manufacturing plants, airports, and other similar places. High background noise will degrade the quality or intelligibility of speech signals in these applications. Excessive noise levels will also degrade the performance of the existing signal processing techniques, such as speech coding, speech recognition, speaker identification, and adaptive echo cancellation, which are developed under the low-noise assumption. The purpose of many speech enhancement algorithms is to reduce the noise or suppress the undesired interferences to improve speech quality. Noise reduction becomes an increasingly important requirement for improving voice quality in noisy environments for hands-free applications.

9.2.1 Noise Reduction Techniques

In general, there are three different types of noise reduction techniques: single channel, dual channel, and multiple channel. The dual-channel technique is based on the adaptive noise cancellation introduced in Chapter 6. The multiple-channel methods can be realized as adaptive beamforming and blind source separation. In this section, we focus on single-channel techniques.

There are three general classes of single-channel speech enhancement techniques: noise subtraction, harmonic-related noise suppression, and speech re-synthesis using vocoders.

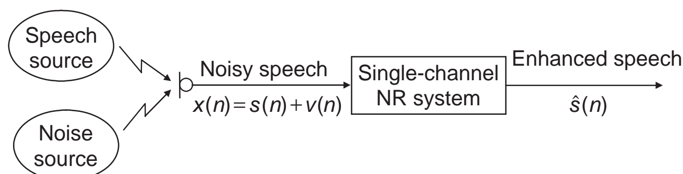


Figure 9.13 A single-channel speech enhancement system

Each technique has its own assumptions, advantages, and limitations. The first technique suppresses noise by subtracting the estimated amplitude spectrum of the noise from the noisy signal, which will be discussed in following sections. The second method employs fundamental frequency tracking using adaptive comb filters for reducing periodic noise. The third technique focuses on estimating speech-modeling parameters using iterative methods and uses these parameters to re-synthesize noise-free speech.

A typical single-channel speech enhancement or noise reduction (NR) system is shown in Figure 9.13. The noisy speech $x(n)$ is the only available input signal for the system. The noisy speech $x(n)$ contains the desired signal $s(n)$ from the speech source and noise $v(n)$ from the noise source(s). The output signal is the enhanced speech $\hat{s}(n)$ (noise reduced).

The single-channel speech enhancement algorithms usually assume that the background noise is stationary (or quasi-stationary) and the characteristics of the noise can be estimated during the silent periods between utterances. Since the system estimates noise characteristics during the non-speech periods and different noise reduction methods can be applied to noisy speech or to noise-only signals, an accurate and robust VAD plays an important role in the performance of the system. In this section, the VAD algorithm introduced in Section 9.1.6 is used.

Noise subtraction algorithms can be implemented either in the time domain or in the frequency domain. The frequency-domain implementation based on short-time magnitude spectra estimation for noise subtraction is called spectral subtraction. As illustrated in Figure 9.14, the spectral subtraction algorithm uses the DFT to obtain the short-time magnitude spectrum of the noisy speech, subtracts the estimated noise magnitude spectrum, reconstructs the DFT coefficients using the subtracted magnitude spectrum with the original phase spectrum, and performs the IDFT to obtain enhanced speech. In practical applications, the spectral subtraction technique uses the computationally efficient FFT and IFFT algorithms.

Frequency-domain noise suppression can also be implemented in the time domain by replacing the DFT with a filterbank that consists of many subbands to decompose the corrupted speech signal into overlapped frequency bands. A filterbank design example is given in Chapter 8. The noise power for each subband is estimated during the non-speech periods. Noise suppression is achieved by using an attenuation factor that corresponds to the ratio of the temporal noisy speech power over the estimated noise power.

9.2.2 Short-Time Spectrum Estimation

Assume a noisy signal $x(n)$ consists of both speech $s(n)$ and uncorrelated noise $v(n)$. As shown in Figure 9.14, this noisy speech is segmented and windowed. The FFT and magnitude spectrum are computed frame by frame. A VAD is used to determine if the

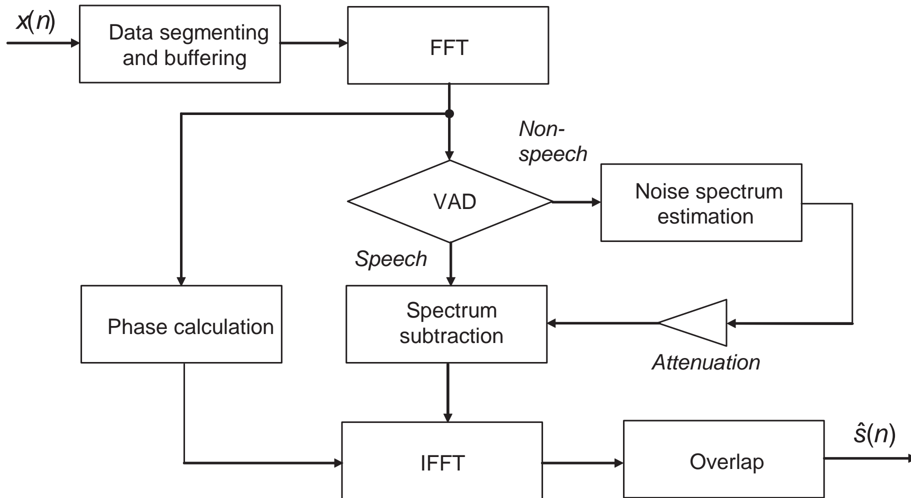


Figure 9.14 Block diagram of the spectral subtraction algorithm

current frame contains speech or noise. For a speech frame, the algorithm performs spectral subtraction to generate the enhanced speech signal $\hat{s}(n)$. For non-speech frames, the algorithm estimates the amplitude of the noise spectrum and attenuates the samples in the buffer to reduce noise.

There are two methods to generate the output (noise-reduced) signal for the non-speech (noise-only) frames: attenuate the signal by a fixed scaling factor that is less than one, or set the output to zero. Subjective evaluation shows that having some residual noise during non-speech frames results in better speech quality. This is because setting the output to zero has the effect of amplifying the noise in the speech frames. The magnitude and characteristics of the noise perceived during the speech and noise frames must be balanced to avoid undesirable audio effects such as clicking, fluttering, or even slurring of the processed speech signal. A reasonable attenuation amount is about 30 dB. This concept is similar to the residual echo suppressor design for adaptive echo cancelers discussed in Chapter 8.

The input signal is segmented using the Hanning window (or other windows introduced in Section 3.2.3) with 50% overlap between the successive data buffers. After noise subtraction, the enhanced speech is reconstructed into the time-domain waveform by the IFFT. These output frames are overlapped and added to produce the output signal.

Example 9.6

Given a frame of 256 samples, calculate the algorithm delay if 50% overlap is used. Compare the computational load for the algorithms with and without using overlap.

If 50% overlap is used, the algorithm delay is 256 samples or 32 ms at 8 kHz sampling rate. The computational load will be doubled since the same block of data has been used twice for the operation.

9.2.3 Magnitude Spectrum Subtraction

Several assumptions are made for the magnitude spectrum subtraction algorithm. First, the algorithm assumes the background noise is stationary such that the estimated noise spectrum will not change during the following speech frames. If the environment changes, there will be enough time for the algorithm to estimate the new noise spectrum before the presence of speech. Therefore, the algorithm must have an effective VAD to determine proper operations and update the noise spectrum. The algorithm also assumes that noise reduction can be achieved by removing the noise from the magnitude spectrum.

If the speech $s(n)$ has been degraded by the zero-mean uncorrelated noise $v(n)$, the corrupted noisy signal can be expressed as

$$x(n) = s(n) + v(n). \tag{9.33}$$

Taking the DFT of both sides results in

$$X(k) = S(k) + V(k). \tag{9.34}$$

Thus the estimation of $|S(k)|$ can be realized as

$$\left|\hat{S}(k)\right| = |X(k)| - E|V(k)|, \tag{9.35}$$

where $E|V(k)|$ is the magnitude spectrum of the noise estimated during the non-speech frames.

Assuming that human hearing is relatively insensitive to noise in the phase spectrum, the enhanced speech using the estimated short-time speech magnitude spectrum $|\hat{S}(k)|$ with the original noisy phase spectrum $\theta_x(k)$ can be reconstructed. Thus, the spectrum of enhanced speech can be reconstructed as

$$\hat{S}(k) = \left|\hat{S}(k)\right| e^{j\theta_x(k)}, \tag{9.36}$$

where

$$e^{j\theta_x(k)} = \frac{X(k)}{|X(k)|}. \tag{9.37}$$

Substituting (9.35) and (9.37) into (9.36), the speech estimate can be expressed as

$$\begin{aligned} \hat{S}(k) &= [|X(k)| - E|V(k)|]\frac{X(k)}{|X(k)|} \\ &= H(k)X(k), \end{aligned} \tag{9.38}$$

where

$$H(k) = 1 - \frac{E|V(k)|}{|X(k)|}. \tag{9.39}$$

It is important to note that the spectral subtraction algorithm given in (9.38) and (9.39) avoids the phase spectrum $\theta_x(k)$ computation by using the arctangent function, which is too complicated to implement on most digital signal processors for real-time applications.

Example 9.7

In the derivation of the spectral subtraction algorithm, identify what are the major sources that could contribute to the distortion of the reconstructed speech.

Using (9.39) and (9.34), the estimation can be further decomposed as

$$\begin{aligned}\hat{S}(k) = X(k)H(k) &= [S(k) + V(k)]\left[1 - \frac{E|V(k)|}{|S(k) + V(k)|}\right] \\ &= S(k) + V(k) - E|V(k)|\frac{S(k) + V(k)}{|S(k) + V(k)|} \\ &= S(k) + V(k) - E|V(k)|\,\mathrm{e}^{\theta_x(k)}.\end{aligned}$$

If $V(k) - E|V(k)|\,\mathrm{e}^{\theta_x(k)} = 0$, this results in perfect noise reduction. In practice, this is unlikely for the following reasons: (1) $V(k)$ is a temporal observation and the computation of true $E[V(k)]$ may be very difficult in real-time applications; (2) the simplified algorithm uses $\theta_x(k)$ to represents $\theta_v(k)$ in order to avoid the heavy computation; and (3) the VAD will not be perfect, thus resulting in an inaccurate estimation of the noise spectrum.

Example 9.8

Use a frame of data from the speech file `voice4.pcm` as the original speech signal $s(n)$. Add the WGN noise $v(n)$ to $s(n)$ to form the noisy speech signal $x(n)$ as defined in (9.33). The corresponding frequency-domain signals are $S(k)$, $V(k)$, and $X(k)$. The phases of $S(k)$ and $X(k)$ are computed using the MATLAB® function `angle()`. The phase differences between $S(k)$ and $X(k)$ are calculated in the range of $(-\pi, \pi)$. The original magnitude spectra of the speech $|S(k)|$ and white Gaussian noise $|V(k)|$ are illustrated in Figure 9.15 using the MATLAB® script `example9_8.m`. The frame size is 256 and the white Gaussian noise is generated by `v = wgn(frame,1,40)`. The noise level is relatively small as compared to the original speech $s(n)$ as shown in Figure 9.15.

The phase difference between the original speech and noisy speech is relatively large in the regions where the noise level is comparable to or even higher than the speech. If VAD is accurate, most of these large phase-difference regions should be classified as silence.

In order to reduce musical tone effects, the reduction of over-subtraction may be modified as

$$H(k) = 1 - \varepsilon\frac{E|V(k)|}{|X(k)|}, \tag{9.40}$$

where $\varepsilon \leq 1$. Using $\varepsilon < 1$ can relieve the effect of over-subtraction, but mildly decreases the SNR of the processed signal.

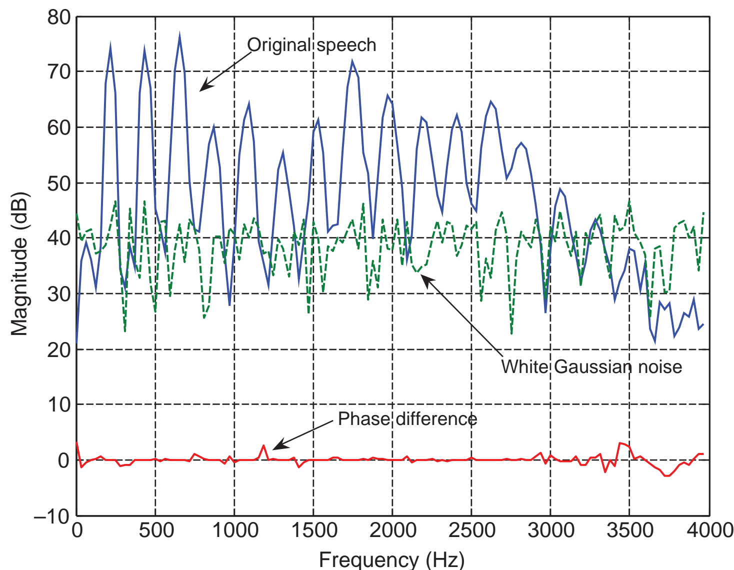


Figure 9.15 Magnitude spectra of speech and noise, and the phase difference between the original and noisy speeches

9.3 VoIP Applications

VoIP refers to IP (internet protocol) telephony services. The voice and data are integrated to provide converged services via packet-switched data networks. VoIP development has led to the cost-effective gateway equipment that bridges analog telephony circuits and IP ports. IP telephony converts the voice or fax into packet data suitable for transport over the networks. As a result, communication systems are now capable of replacing the traditional PSTN with integrated voice and data over the IP. This section uses an example to demonstrate the still-evolving VoIP system, its applications, and some factors that affect the voice quality.

9.3.1 Overview of VoIP

A simplified diagram of VoIP applications is shown in Figure 9.16. It includes the end user devices, that is, the traditional analog phones, SIP (session initial protocol) controlled phones, and pure software soft phones. It also includes network components such as the PBX (private branch exchange), VoIP gateway, and VoIP phone adapter.

Similar to the traditional PSTN, the VoIP application also involves signaling and media processing. The media processing topics include efficient compression of media data to save network bandwidth, voice noise reduction, packet loss concealment, and overall transmission delay reduction. Some VoIP-related topics are introduced in Chapter 8 when adaptive echo cancelers are discussed. As shown in Figure 8.3, the echo canceler is a very important function to make the VoIP application feasible due to the long delay for transmitting the media packets over the Internet. In addition, there are many other key modules enabling the VoIP applications to achieve acceptable voice quality.

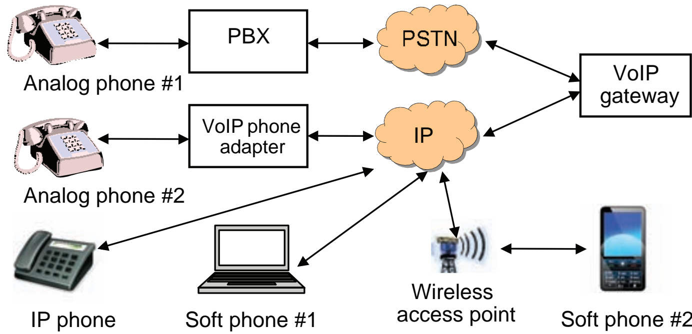


Figure 9.16 Topology of VoIP applications

In Figure 9.16, the media path is the end-to-end call processing path between analog phone #1 and the IP phone or any soft phone. The media processing uses DSP algorithms for media conversion and is usually implemented in the VoIP gateway that bridges the traditional circuits and packet-switched networks. It can also be implemented in the VoIP phone adapter, which adapts the packet data to the traditional user-loop two-wire analog phones. Both the VoIP gateway and phone adapter generally share similar functional blocks for signal processing. Figure 9.17 shows processing units that use many of the DSP techniques introduced in previous chapters, such as the line echo canceler (LEC), the nonlinear processor (NLP), comfort noise insertion (CNI), noise reduction (NR), automatic level control (ALC), tone detection and generation, VAD, voice encoder (ENC) and decoder (DEC), as well as some new modules. These new modules, including discontinuous transmission (DTX), comfort noise generation (CNG), and packet loss concealment (PLC), are very important media processing components in VoIP applications for improving bandwidth efficiency and reducing the packet loss impairment.

The IP provides a standard encapsulation for data transmission over the networks. It contains the source and destination routing addresses. First, the signal from an analog phone is filtered by the LEC to remove the line echoes introduced by the hybrid (H). The LEC output signal is then fed to the tone detection module to detect various tones, such as DTMF tones, call progress tones, or modem and fax tones. The detected tones will be sent to the IP side via the out-of-band message to inform the other side. NLP and CNI modules are part of the LEC to

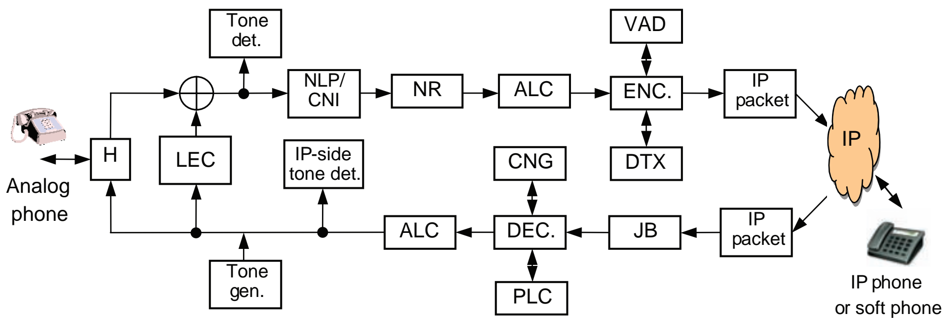


Figure 9.17 Simplified block diagram of media signal processing in VoIP gateway or adapter

further eliminate residual echoes. NR is used to reduce noise and ALC is responsible for gain adjustment. Finally, the speech signal is encoded to produce the bit stream. Depending on the VAD output and DTX decision, there may not be any data to send out or only noise energy information is sent. The noise energy information sent over the network is the silence insertion descriptor (SID). The bit stream, either the encoded speech frame or the SID frame, is sent over the IP network to which the IP phone is connected. On the IP phone side, this bit stream is processed, decoded, and played if it is a speech frame, or comfort noise is generated for the SID frame.

In the reverse processing, the encoded bit stream is received from the IP network and a jitter buffer (JB) is used to compensate for network jitters. The jitter buffer holds incoming packets for a specified amount of time before presenting them to the decoder. This will smooth the packet flow and increase the resilience to delayed packets or out-of-order packets. The jitter buffer size is configurable, typically 20 to 100 ms, and can be optimized for a given network condition. Depending on the frame type, the bit stream from the jitter buffer will be processed by the decoder (DEC) for normal speech frames, processed by the PLC for erased frames, or replaced with comfort noise generated by the CNG for the SID frames.

Tone detection is used for monitoring the telephony events to determine the call progress and to respond to the events. Sometimes, these events can trigger the codec to switch operation mode to handle the media data. One example is the fax tones, which can trigger switches from the low-bit-rate codec (e.g., G.729) to the high-bit-rate codec (e.g., G.711), or even to the fax relay codec (e.g., T.38). The tone generation module is used to generate the tones locally for the end user if this event is an out-of-band message sent from the IP side, an example being the locally generated DTMF or modem tones.

9.3.2 Discontinuous Transmission

DTX is an effective way of achieving bandwidth efficiency. The basic principle of DTX is to switch on the transmitter only when active speech is present. The basic problem of DTX is the potential degradation in speech quality and clipping due to noise contrast effects. The design of the voice activity detector has to balance the risk of clipping speech against the risk of misclassifying noise as speech.

DTX is very useful to save bandwidth. As described in Section 9.2, both G.729 and AMR use the VAD-controlled DTX mechanism. Once VAD indicates that the current segment is a silent frame and its energy has been changed beyond the threshold from previous silent frames, the SID will be sent. At the decoder side, this silent frame will be replaced by comfort noise generated with a similar level. Figure 9.18 shows the diagram of a typical silence compression method. The input signal is first buffered to form a frame of input data. The VAD module classifies the input data as an active or inactive speech frame. The DTX module is used to further classify the inactive frame as either an SID or untransmitted frame according to the variation of amplitude and frequency components.

The DTX module is responsible for tracking the change in noise level from one silent frame to another. If the noise power has noticeable changes, for example, more than 6 dB, this frame will be classified as an SID frame. The SID algorithm module is then used to encode the input data as the comfort noise payload.

At the decoding side, the speech decoder module or the CNG module will be selected to decode the packet bit stream according to the frame type. If the current frame is an inactive

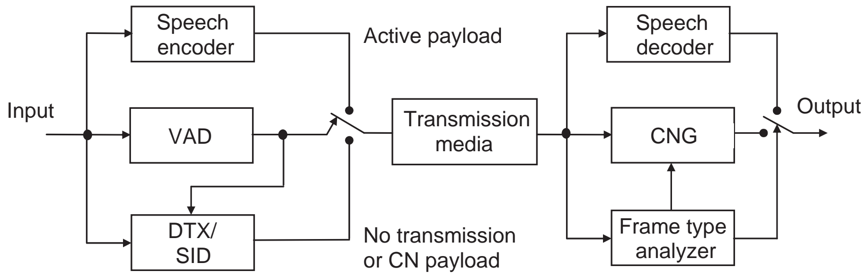


Figure 9.18 Bock diagram of silence compression scheme

speech frame (VAD = Inactive) and the previous frame is active speech (VAD = Active), that is, the first inactive frame, the frame type will be classified as an SID frame. For the other inactive frames, the decision between the SID frame and untransmitted frame will be made based on the following criteria from the differences in the spectrum and power.

The frame energy is calculated and compared to the energy from the previous frame. If the difference in amplitude is greater than 6 dB, this frame is classified as an SID frame. Assuming the SID frame energy is defined as

$$E_{\text{sid}} = \sum_{k=0}^{L-1} x_m^2(k),$$

where m is the frame index, the same method can be used to calculate the energy in the subsequent silent frame. Assuming the current silent frame energy is calculated as

$$E_{\text{sil}} = \sum_{k=0}^{L-1} x_{m+l}^2(k)$$

where $m+l$ is the frame index and $l \geq 1$, this frame will be considered as a new SID frame if it satisfies the following condition:

$$|20\log(E_{\text{sil}}/E_{\text{sid}})| \geq 6\,\text{dB}. \tag{9.41}$$

Otherwise, nothing needs to be transmitted for this silent frame.

The noise level is expressed by the level compared to the maximum range that the system can support, and is expressed as dBov (dB overload). For example, within a 16-bit linear system, the reference will be a square wave with values ±32 767 and this square wave represents 0 dBov. One byte can represent values from 0 to −127 dBov.

9.3.3 Packet Loss Concealment

The packet loss concealment (PLC) technique, also known as the frame-erasure concealment technique, is used to regenerate lost or out-of-order packets. The PLC is used to synthesize the speech segment based on the history buffer to recover the lost data in the received data stream. Many CELP-based speech coders such as G.729 and AMR use an LPC synthesis filter to

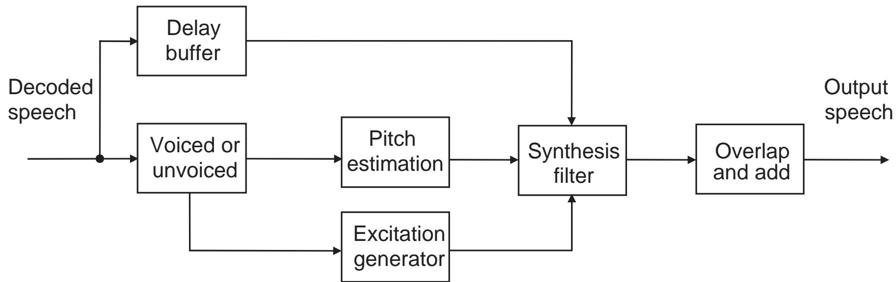


Figure 9.19 Bock diagram of typical PLC

regenerate the lost packets from the available decoded parameters such as pitch, gain, and synthesis filter coefficients.

Appendix I of G.711 is designed for sample-based codecs such as G.711 or G.726, which does not include the LPC-based synthesis filter or pitch estimation module. The PLC technique described here is entirely receiver based. A simplified diagram of typical PLC implementation is given in Figure 9.19.

As in vocoders, the buffered speech data is first processed to determine if it is a voiced frame or an unvoiced frame. For a voiced frame, the pitch estimation module is used to estimate the pitch by searching for the peak locations from the normalized autocorrelation function of the residual signal. Pitch periods ranging from 20 to 140 samples will be searched at a resolution of 0.125 ms (e.g., 1/8000 for the 8 kHz sampling frequency). The pitch period in terms of samples is passed to the excitation generator. These samples are also stored and used in the situation where consecutive packet loss has occurred. For the unvoiced frame, random noise is generated for synthesizing unvoiced speech.

Since the speech signal is often locally stationary, it is possible to use the signal's past history to generate a reasonable approximation for the missing segment without using LPC synthesis. If the erasures are not too long and the erasure does not occur in the region where the signal is changing rapidly, the erasures may be inaudible after the concealment.

The overlap–add module is used to produce the final output of reconstructed speech. The output is delayed by about 1/4 of the maximum pitch period before it is sent out. This algorithm delay is used for the overlap–add at the start of erasure. It allows the PLC processing to make a smooth transition between the real (in the good frame) and synthesized signal (reconstructed in the erasure frame). As an example, the PLC defined in Appendix I of G.711 uses 3.75 ms (30 samples) of delay for the overlap–add purpose.

9.3.4 Quality Factors of Media Stream

The network delays, packet losses, packet jitters, and echoes are major factors that will affect the perceived VoIP quality.

The ITU-T G.114 standard states that the end-to-end one-way delay should not exceed 150 ms. The overall voice packet delay can be improved by prioritizing the voice packet in the network. The speech coding algorithms [2,10,15] can also compensate the network delay (e.g., the delay can be shortened by choosing low-delay speech codecs). A good jitter buffer algorithm can effectively compensate the jitters with the minimum buffer size to reduce the overall delay. Furthermore, the efficient packet loss concealment algorithms

make the lost or discarded packet effects less noticeable. Network echoes can be effectively canceled or suppressed using the adaptive echo cancellation algorithms introduced in Chapter 8.

Voice band data (VBD) is the data transmitted through the packet networks' voice channel using an appropriate encoding method. These data signals include data modems, fax machines, and text-phones for the hearing-impaired community.

A VoIP system is configured to handle voice services by default. Hence, in order to serve modem data, it is necessary to turn off the devices that are specifically used for voice but will impair the data in the network. This is usually done by switching the system from voice mode to VBD mode. Auto-switching is the easiest and preferred way once the signaling tones for media have been detected, such as the V.25/V.8 answering tones (2100 Hz with or without phase reversal) used for data, fax modems, or fax calling tone. A typical switch procedure will disable the PLC, VAD, NLP, and dynamic jitter buffer during the switch from voice mode to VBD mode. For higher rate modems, such as the V.34 standard, the LEC will be disabled if the detected answering tone has the reversed phase.

9.4 Experiments and Program Examples

This section implements VAD, noise reduction, LPC coefficient calculation, and speech codecs using C programs for experiments.

9.4.1 LPC Filter Using Fixed-Point C with Intrinsics

The ETSI (Europe Telecommunications Standards Institute) library that can be used for fixed-point operators to implement codecs includes G.723.1, G.729, and AMR. The floating-point C program can be efficiently converted to fixed-point C programs using these basic operators provided by ETSI.

For real-time DSP applications using fixed-point C55xx processors, converting the floating-point C code to fixed-point C code is often required. According to the dynamic range, the input signal should be normalized to avoid overflow. An example of vector normalization and LPC analysis filter calculation is listed in Table 9.6. The basic operators provided by the fixed-point C library do not deliver high efficiency for real-time processing. Replacing these fixed-point C operators with C55xx intrinsics improves runtime efficiency. The results using C55xx intrinsics have the same results (bit exactness) as those obtained from using fixed-point C operators from ETSI. In this experiment, the intrinsics functions (basic operators) `round`, `norm_l`, `L_add`, `L_sub`, `L_shl`, and `L_shr` are used for rounding, normalization, 32-bit addition, subtraction, and shift. After the normalization, the Levinson–Durbin recursive algorithm is used to compute LPC coefficients.

In Table 9.6, the operator `round()` converts a 32-bit value to a 16-bit value by rounding the higher 16-bit value of the 32-bit value and saving the rounded upper 16-bit result to the 16-bit memory location `autoc[]`. The operator `norm_l()` calculates the leading sign bit of the variable. Use the same approach, the Levinson–Durbin algorithm can be written as shown in Table 9.7. The program uses the Q13 format to represent the LPC coefficients `a[]` and the reflection coefficients `K[]`, thus the LPC coefficients are in the range of -4 (`0x8000/0x2000`) and $+3\frac{8191}{8192}$ (`0x7fff/0x2000`). The prediction error `E[]` and the correlation `R[]` are represented using the Q14 format to avoid overflow. The files used for the experiment are listed in Table 9.8.

Table 9.6 Compute autocorrelation coefficients using intrinsics

```
/*
|     calc_autoc()              : Autocorrelation
|     Input        ws           : ws[0,.,frame_size-1]
|                  p_order      : LPC order
|                  frame_size   : Frame size
|     Output       autoc        : autoc[0,.,p_order]*/
void calc_autoc(short *ws, short p_order, short frame_size, short *autoc)
{
    short k,m,i,Exp;
    long  acc0,acc1;

    // Compute autocorrelation:autoc[0]
    acc1 = (long) 0;
    for ( i = 0; i < frame_size; i ++)
    {
        acc0 = (long)ws[i]*ws[i];
        acc1 = L_add(acc1,acc0);
    }
    /* Normalize the energy */
    Exp = norm_l( acc1);
    acc1 = L_shl( acc1, Exp);
    autoc[0] = round( acc1);

    // Compute autocorrelation: autoc[k], k=1,...,10
    for ( k = 1; k <= p_order; k ++)
    {
       acc1 = (long) 0;
       for ( m = k; m < frame_size; m ++)
       {
           acc0 = (long)ws[m]*ws[m-k];
           acc1 = L_add(acc1,acc0);
       }
       acc0 = L_shl( acc1, Exp);
       autoc[k] = round( acc0);
    }
}
```

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package. Build and run the program to compute the LPC coefficients. The experiment generates 10 coefficients for each frame and these coefficients are saved in the file `lpcCoeff.txt` in the `data` folder. The experimental result has 120 frames.
2. Plot the magnitude response of the LPC filter $1/A(z)$ against the magnitude spectrum of the input signal. This can be done by modifying MATLAB® scripts `example9_2.m`:
   (a) Choose the sixth speech frame (180 samples) from file `voice4.wav` to compute the magnitude spectrum of the speech.

Table 9.7 LPC filter coefficient computation using intrinsics

```
/*
|      calc_lpc()            : LPC coefficients
|       Input      autoc     : autoc[0,...,p_order]
|                  p         : LPC order
|       Output    lpc        : lpc[0,...,p_order]
 */
short K[LPCORDER+1];                    // Reflection coefficients in Q13
short a[(LPCORDER+1)*(LPCORDER+1)]; // LPC coefficients in Q13
short E[LPCORDER+1];                    // Prediction error in Q14
void calc_lpc(short *autoc, short *lpc, short p)
{
  short i,j,p1;
  long acc0,acc1;
  short *R;                             // Correlation coefficients in Q14
  short sign;
  p1= p+1;

  // Calculate first-order LPC coefficients
  R = autoc;
  E[0] = R[0];
  acc0 = L_shl(R[1],13);
  sign = signof(&acc0);
  acc0 = L_shl(acc0,1);
  K[1] = div_l(acc0, E[0]);

  if(sign== (short)-1) K[1] = negate(K[1]);
  a[1*p1+1]=K[1];

  /*E[1]=((8192-((K[1]*K[1])>>13)) * E[0])>>13;*/
  acc0 = (long)K[1] * K[1];
  acc0 = L_shr(acc0,13);
  acc0 = L_sub(8192, acc0);
  acc0 = (long)E[0]* (short)(acc0);
  acc0 = L_shr(acc0,13);
  E[1] = (short)acc0;

  // Recursive calculation of LPC coefficients from order 2 to p
  for (i=2;i<=p;i++)
  {
    acc0=0;
    for (j=1;j<i;j++)
    {
      acc1 = (long)a[j*p1+i-1] * R[i-j];
      acc0 = L_add(acc0,acc1);
    }
    acc1 = L_shl(R[i],13);
    acc0 = L_sub(acc1, acc0);
    sign = signof(&acc0);
    if (acc0 > L_shl(E[i-1],13))
```

Table 9.7 (*Continued*)

```
        break;
    acc0 =L_shl(acc0,1);
    K[i] = div_l(acc0,E[i-1]);
    if(sign == (short)-1) K[i] = negate(K[i]);
    a[i*p1+i] = K[i];

    /* a[j*p1+i]=a[j*p1+(i-1)]-((K[i]*a[(i-j)*p1+(i-1)])>>13);*/
    for (j=1;j<i;j++)
    {
        acc0 = (long)K[i] * a[(i-j)*p1+(i-1)];
        acc1 = L_shl((long)a[j*p1+(i-1)],13);
        acc0 = L_sub(acc1,acc0);
        acc0 = L_shr(acc0,13);
        a[j*p1+i] = (short)acc0;
    }
    /* E[i]=((8192-((K[i]*K[i])>>13)) * E[i-1])>>13;*/
    acc0 = (long)K[i] * K[i];
    acc0 = L_shr(acc0,13);
    acc0 = L_sub((short)8192,acc0);
    acc0 = (long)E[i-1]* (short)(acc0);
    acc0 = L_shr(acc0,13);
    E[i] = (short) acc0;
  }
  for (j=1;j<=p;j++)
    lpc[j] = negate(a[j*p1+p]);
}
```

(b) Use the obtained sixth-frame LPC coefficients to compute the magnitude response of $1/A(z)$.
(c) Plot both the magnitude response of the filter and the corresponding magnitude spectrum of the speech signal.

Table 9.8 File listing for the experiment Exp9.1

Files	Description
`intrinsic_lpc_mainTest.c`	Program for testing LPC experiment
`intrinsic_lpc_lpc.c`	Fixed-point function computes LPC coefficients
`intrinsic_lpc_auto.c`	Fixed-point function computes autocorrelation function
`intrinsic_lpc_hamming.c`	Hamming window function using intrinsics
`intrinsic_lpc_hamTable.c`	Function generates Hamming window lookup table
`lpc.h`	C header file for experiment
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`voice4.wav`	Speech file in WAV format

The magnitude response of the LPC filter represents the envelope of the speech magnitude spectrum and the plot should be similar to Figure 9.3.

3. Modify the test program using different parameters such as changing the LPC filter order from 10 to 8, 12, and 15, to perform experiments and observe the differences.
4. Repeat the experiment with different speech files, such as speeches from male and female speakers with voiced and unvoiced segments.

9.4.2 Perceptual Weighting Filter Using Fixed-Point C with Intrinsics

The perceptual weighting filter coefficients can be computed from the LPC coefficients for the CELP-based speech coding algorithms. This experiment calculates the weighting filter coefficients using the fixed-point C `intrinsic_pwf_wz.c`. The program `intrinsic_pwf_wz.c` is listed in Table 9.9, where the variables `gamma1` and `gamma2` represent the parameters γ_1 and γ_2 defined in (9.19). The files used for the experiment are listed in Table 9.10, where PWF represents the perceptual weighting filter.

Table 9.9 Computation of perceptual weighting filter coefficients

```
/*
|     calc_wz()              : Perceptual weighting filter
|                              W(Z)=(wf1[z])/(wf2[z])
|       Input     lpc        : lpc[0,...,p_order]
|                 gamma1     : gamma1
|                 gamma2     : gamma2
|                 p_order    : LPC order
|       Output    wf1        : wf1[0,.,p_order]
|                 wf2        : wf2[0,.,p_order]
*/
void calc_wz(short *lpc, short gamma1,short gamma2, short p_order,
short *wf1, short *wf2)
{
      short i,gam1,gam2;
      wf1[0]=32767;      // Constant 1.0 in Q15
      wf2[0]=32767;      // Constant 1.0 in Q15
      gam1 = gamma1;     // Bandwidth expansion factor in Q15
      gam2 = gamma2;     // Bandwidth expansion factor in Q15

      for (i=1; i<=p_order; i++) // Calculate weighting filter
                                 // coefficients
      {
         wf1[i] = mult_r(lpc[i],gam1);
         wf2[i] = mult_r(lpc[i],gam2);
         gam1 = mult_r(gam1, gamma1);
         gam2 = mult_r(gam1, gamma2);
      }
}
```

Table 9.10 File listing for the experiment Exp9.2

Files	Description
intrinsic_pwf_mainTest.c	Program for testing PWF experiment
intrinsic_pwf_lpc.c	Fixed-point function computes LPC coefficients
intrinsic_pwf_auto.c	Fixed-point function computes autocorrelation function
intrinsic_pwf_hamming.c	Hamming window function using intrinsics
intrinsic_pwf_hamTable.c	Function generates Hamming window lookup table
intrinsic_pwf_wz.c	Function calculates perceptual weighting filter coefficients
pwf.h	C header file
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
voice4.wav	Speech file in WAV format

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package. Build and run the program to generate the perceptual weighting filter coefficients and save the result in the file `pwfCoeff.txt`. There are a total of 120 frames, and each frame has three lines of coefficients. The first line (`LPC`) contains 10 LPC coefficients; the second (`WF1`) and third lines (`WF2`) contain the numerator and denominator coefficients of the perceptual weighting filter $W(z)$, respectively.
2. Examine the experimental results by plotting the magnitude response of the weighting filter and compare it to the LPC envelope obtained from the previous experiment. This can be done by modifying the MATLAB® script `example9_5.m`. For example, use the coefficients from the sixth frame in file `pwfCoeff.txt` to plot the magnitude responses of the filers $1/A(z)$ and $W(z)$. The plot is expected to be similar to Figure 9.6, but with reversed peaks and valleys.
3. Change the parameter γ_2 to different values and observe the different weighting effects.
4. Change the order of the LPC filter as suggested in the previous experiment and repeat the experiment to observe different results.
5. Use different frame sizes to calculate the LPC coefficients and observe different results.

9.4.3 VAD Using Floating-Point C

This experiment uses a 256-point complex FFT to implement the VAD algorithm using the floating-point C program. The FFT bins that cover the frequency range from 250 to 820 Hz are used for power estimation as shown in Figure 9.7. Table 9.11 lists the C implementation of the VAD algorithm presented in Section 9.1.

In this program, the signal power is computed by summing the FFT bins from `ss` to `ee`, which correspond to frequencies at 250 and 820 Hz, respectively. Using the 256-point FFT with the 8000 Hz sampling rate, the low-frequency 250 Hz corresponds to the FFT bin `ss` $= (250 \times 256)/8000 = 8$ and the high-frequency 820 Hz corresponds to the FFT bin `ee` $= (820 \times 256)/8000 = 26$. The short-term energy is estimated using (9.21). The detector uses the threshold, `Em`, to determine if the current frame contains a speech or non-speech signal. If the energy is greater than the threshold, the detector will set the VAD flag. If the

Table 9.11 C program for VAD algorithm

```
short vad_vad(VAD_VAR *pvad)
{
  short k,VAD;
  float En;                          // Current frame power
  VAD_VAR *p = (VAD_VAR *)pvad;
  En = 0;                            // VAD algorithm
  for (k=p->ss; k<=p->ee; k++)       // Power from 250 to 820 Hz
  {
     En += (float)(sqrt(p->D[k].real*p->D[k].real
        + p->D[k].imag*p->D[k].imag));
  }
  p->Em = p->am1*p->Em + p->alpham*En;
  if (p->Nf Em)                      // Update noise floor
     p->Nf = p->al1*p->Nf + p->alphal*En;
  else
     p->Nf = p->am1*p->Nf + p->alpham*En;
  p->thres = p->Nf + p->margin;
  VAD = 0;
  if (p->Em >= p->thres)
     VAD=1;
  if (VAD)                   // Speech is detected since Em >= threshold
     p->hov = HOV;
  else                       // Silence is detected since Em < threshold
  {
     if (p->hov- <=0)
        p->hov =0;
     else
        VAD=1;
  }
  return VAD;
}
```

energy is less than the threshold, the VAD flag will be either set or cleared based on the status of the hangover counter (`hov`). In order to prevent the detector from oscillating, a small number is added to the threshold as a safety margin. Table 9.12 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package. Build and run the experiment to obtain the output file `VAD_ref.xls`, which contains the speech with the corresponding VAD results.
2. Plot the speech and the VAD results to evaluate the accuracy of the VAD detector.
3. Modify the program to generate the output determined by the VAD results. When the VAD flag is set, it copies the input directly to the output, otherwise it writes zeros to the output.

Table 9.12 File listing for the experiment Exp9.3

Files	Description
floatPoint_vad_mainTest.c	Program for testing VAD experiment
floatPoint_vad_vad.c	Noise reduction function uses VAD
floatPoint_vad_hwindow.c	Tabulated data table for Hanning window
floatPoint_vad_ss.c	FFT and pre-processing for VAD detection
floatPoint_vad_init.c	VAD initialization
floatPoint_vad_fft.c	FFT function and bit-reversal function
floatPoint_vad.h	C header file defines constant and function prototyping
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
speech.wav	Speech data file

In this way, the output speech samples are set to zero for the noise-only frames. Listen to the output file to evaluate the VAD results.

4. Adjust the length of hangover time (`HOV`) and listen to the transition effects under different hangover lengths.

9.4.4 VAD Using Fixed-Point C

This experiment converts the floating-point C program used in the previous experiment to the fixed-point C program. Table 9.13 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package. Build and run the experiment to obtain the output file `VAD_ref.xls` which contains the original speech and the VAD results.
2. Plot the speech and the VAD results to examine the accuracy of VAD.

Table 9.13 File listing for the experiment Exp9.4

Files	Description
mix_VAD_mainTest.c	Program for testing VAD experiment
mixed_vad_vad.c	Noise reduction function uses VAD
mixed_vad_tableGen.c	Tabulated data table for Hanning window
mixed_vad_ss.c	FFT and pre-processing for VAD detection
mixed_vad_init.c	VAD initialization
mixed_vad_wtable.c	Twiddle factor table generation
intrinsic_fft.c	FFT function
ibit_rev.c	Bit-reversal function
dspFunc55.asm	DSP supporting functions in assembly language
mixed_vad.h	C header file defines constant and function prototyping
icomplex.h	C header file defines complex data type
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
speech.wav	Speech data file

3. Compare the fixed-point detection results to the floating-point C results obtained from the previous experiment. Observe any performance degradation caused by numerical precision.
4. Profile the required clock cycles and compare the runtime efficiency between the floating-point and fixed-point C implementations.
5. Repeat steps 3 and 4 of experiments given in Section 9.4.3 for this fixed-point C experiment.

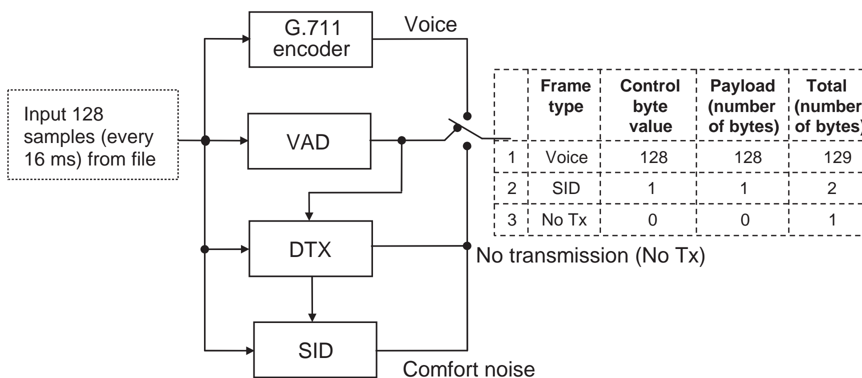
9.4.5 Speech Encoder with Discontinuous Transmission

Based on the previous VAD experiments, the DTX and SID modules will be implemented in this experiment using the floating-point C program. In the experiment, the G.711 (μ-law) codec is used to compress the input speech from 16-bit PCM format to 8-bit log-PCM format. The block diagram for VAD/DTX/SID processing is illustrated in Figure 9.20. The switch shown in the diagram represents the selection of three different frame types: voice, SID, or no transmission.

In this experiment, 1 byte is used to represent the frame type (the frame type column in Figure 9.20). As shown in Figure 9.20, 129 bytes are for the voice frame, 2 bytes are for the SID frame, or 1 byte is for no transmission frame. This experiment sets the DTX threshold (`dtxThre`) to 2 (6 dB) and this threshold can be adjusted to achieve different DTX results. The experiment generates the DTX data file, `parameter.dtx`, which is G.711 compressed speech data with DTX. The speech quality of the DTX compressed result can be verified by listening to the decoded voice; this will be presented along with the G.711 decoder in the next experiment. Table 9.14 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package. Build and run the experiment to obtain the output file, `parameter.dtx`, which contains the encoded speech.
2. Modify the DTX threshold (`dtxThre`) value from 2 to 1.414 and 4, redo the experiment, and check the size of the encoded speech file `parameter.dtx` for using these new DTX threshold values.
3. Modify the VAD hangover time (`HOV`) by using a longer value (4) and a shorter value (0), redo the experiment, and save the results. Examine the file size of `parameter.dtx` to see



	Frame type	Control byte value	Payload (number of bytes)	Total (number of bytes)
1	Voice	128	128	129
2	SID	1	1	2
3	No Tx	0	0	1

Figure 9.20 Media processing diagram with VAD/DTX/SID

Table 9.14 File listing for the experiment Exp9.5

Files	Description
`floatPoint_dtx_mainTest.c`	Program for testing DTX experiment
`floatPoint_vad_vad.c`	Noise reduction function uses VAD
`floatPoint_vad_hwindow.c`	Tabulated data table for Hanning window
`floatPoint_vad_ss.c`	FFT and pre-processing for VAD detection
`floatPoint_vad_init.c`	VAD initialization
`floatPoint_vad_fft.c`	FFT function and bit reversal
`floatPoint_dtx.c`	DTX functions
`floatPoint_sid.c`	SID functions
`g.711.c`	G.711 standard implementation
`floatPoint_dtx.h`	C header file defines constant and function prototyping
`floatPoint_vad.h`	C header file defines constant and function prototyping
`g.711.h`	C header file for G.711 standard implementation
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`speech.wav`	Speech data file

if there is any difference and explain why. Save these files for experiment Exp9.6 (to reconstruct the speech and listen to the transition effects under different hangover lengths).

4. Change the companding law from the μ-law to A-law by replacing the function call `ulaw_compress()` with `alaw_compress()`. Repeat steps 2 and 3, and save the results for the next experiment.

9.4.6 Speech Decoder with CNG

In the previous experiment, DTX and the G.711 encoder are used for speech processing at the transmitter side. In this experiment, the G.711 speech decoder with the CNG module is used at the receiver end. The G.711 decoder expands the log-scale samples back to the 16-bit PCM format and the CNG module will generate comfort noise based on the SID information. The block diagram of the processing modules is shown in Figure 9.21.

Repeating the previous data frame is a simple packet loss concealment (PLC) technique. Compared to the method using pitch-synchronized voice data, the packet repeating technique intends to preserve the relative power and a similar waveform. In real applications, the PLC

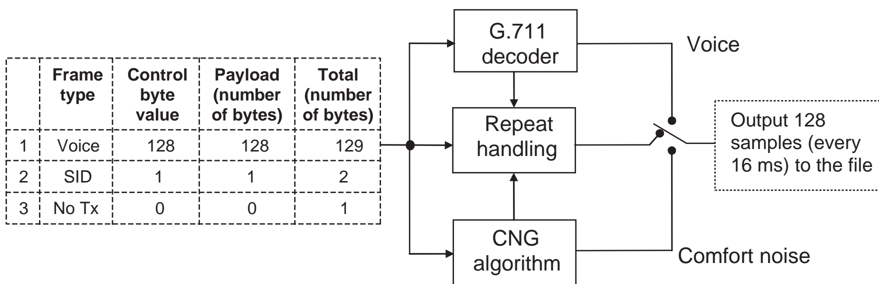


Figure 9.21 Media processing diagram with CNG

Table 9.15 File listing for the experiment Exp9.6

Files	Description
cng_mainTest.c	Program for testing CNG experiment
cng.c	CNG functions
g711.c	G.711 standard implementation
cng.h	C header file defines constant and function prototyping
g711.h	C header file for G.711 standard implementation
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
parameter.dtx	Encoded data file obtained from Exp9.5

module will replace the repeat handling module shown in Figure 9.21 to reduce side effects, such as the noticeable clicking noise. Table 9.15 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package. Build and run the experiment to obtain the output file, `result.wav`, which is the decoded speech.
2. Listen to the `result.wav` and compare it to the original speech (`speech.wav`) given in the previous experiment.
3. Use different DTX thresholds (`dtxThre`) and VAD hangover time (`HOV`) from the DTX experiment and compare the decoded speech quality.
4. Change the companding law from the μ-law to A-law by replacing the function call `ulaw_expand()` with `alaw_expand()` to process the signals encoded previously by the A-law encoder (use the saved files from step 4 in the previous experiment). Examine if there is any performance difference between these two companding laws.

9.4.7 Spectral Subtraction Using Floating-Point C

The core part of the spectral subtraction algorithm for noise reduction is listed in Table 9.16. The input signal is placed in the FFT bins `TB[k]` and the estimated noise is placed in the FFT bins `NS[k]`. Based on the VAD information, either spectrum subtraction or amplitude attenuation will be applied. The number `N` equals half the FFT size and the array `h[k]` contains filter coefficients.

The experiment will generate enhanced speech file `nr_output.wav` and the reference file `nr_ref.xls`. The first column of the reference file contains the original signal and the second column is the processed data. The files used for the experiment are listed in Table 9.17.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package. Build and run the experiment to obtain the output file.
2. Plot the original speech file `speech.wav` and the noise-reduced file `nr_output.wav`. Compare these waveforms to evaluate the performance of the NR algorithm. It is recommended to use the MATLAB® function `spectrogram` for frequency-domain evaluation.

Table 9.16 Partial list of C program for spectral subtraction

```
if (VAD)          // Speech is detected since VAD=1
{
    for (k=0;k<=N;k++)
    {
        tmp = TB[k] - (127./128.)*NS[k];
        h[k] = tmp / TB[k];
    }
}
else                             // Silence is detected since VAD=0
{
    Npw = 0.0;
    for (k=0;k<=N;k++)           // Update the noise spectrum
    {
        NS[k] = (1-alpha)*NS[k] + alpha*TB[k];
        Npw += NS[k];
    }
    Npw = Npw/Npw_normalfact;                  // Normalized noise power
    margin = (127./128.)*margin+(1./128.)*En; // New margin
}
```

3. Select different values of ε defined in (9.40), such as 63/64, 31/32, or 15/16, to obtain different results, and compare the spectral subtraction algorithm performance. Use subjective evaluation (listening) to observe the musical tone effect.
4. Use different attenuation factors, such as 0.5, 0.1, and 0.01, for noise-only frames and compare the different noise reduction results.
5. Use different FFT sizes, such as 128 and 512, redo the experiment, and compare the results.

Table 9.17 File listing for the experiment Exp9.7

Files	Description
`floatPoint_nr_mainTest.c`	Program for testing spectral subtraction experiment
`floatPoint_nr_vad.c`	Voice activity detector
`floatPoint_nr_hwindow.c`	Tabulated data table for Hanning window
`floatPoint_nr_ss.c`	Program for pre-processing before FFT
`floatPoint_nr_init.c`	Noise reduction experiment initialization
`floatPoint_nr_fft.c`	FFT function and bit reversal
`floatPoint_nr_proc.c`	Noise reduction control function
`floatPoint_nr.h`	C header file defines constant and function prototyping
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`speech.wav`	Speech data file

Table 9.18 File listing for the experiment Exp9.8

Folder/Files		Description
G722	warm.sln;	AMR-WB workspace
.\c-code	*.c;*.h;*.tab	All source code unzipped from 26173-a00.zip
.\testv	*.cod;*.out;*.inp;*.bat	All test vectors
.\release	wamr_enc.exe & wamr_dec.exe	Executable files
.\wamr-enc	wamr_enc.vcproj	Encoder project
.\wamr-dec	wamr_dec.vcproj	Decoder project

9.4.8 G.722.2 Using Fixed-Point C

This experiment tests the G.722.2 (GSM AMR-WB) encoder and decoder based on the ANSI-C reference source code from 3GPP. The completed package `26173-a00.zip` (dated April 5, 2011) including the test vectors can be downloaded from the 3GPP website.

The zip file `26173-a00.zip` contains the zipped source code as `26173-a00_ANSI-C_source_code.zip`. Extract the zip file; the complete G.722.2 will be stored as shown in Table 9.18, assuming `G722` is the working directory. Use the workspace file `wamr.sln` with the Microsoft Visual C environment to compile the project; the executable encoder and decoder programs will be placed in the `release` folder. The batch files `test_enc.bat` and `test_dec.bat` in the folder `testv` are used to run the encoder and decoder operations for experiments.

Procedures of the experiment are listed as follows:

1. Download the G.722.2 software package and unzip the `26173-a00_ANSI-C_source_code.zip` file into the folder `.\c-code`. Launch the Microsoft Visual Studio and load `warm.sln`. Build the whole solution. The executable programs, `wamr_enc.exe` and `wamr_dec.exe`, are located in the directory `.\release`.
2. Run the batch file `test_enc.bat` from the folder `.\testv` to encode the test vectors to obtain the encoded bit files (bit streams). Compare the encoded speech bit streams against the provided reference test vectors, and make sure they are identical.
3. Run the batch file `test_dec.bat` from the folder `.\testv` to decode the encoded bit streams. Compare the decoded speech files against the provided reference test vectors; they should also be identical.
4. Use the audio file `audioIn.pcm` at the 16 kHz sampling rate from Example 10.7 as the input, generate the processed audio with bit rates of 23.85, 23.05, 19.85, 18.25, 15.85, 14.25, 12.65, 8.85, and 6.6 kbps. Compare the audio quality among these rates.
5. Examine the C source code in the folder `c-code`, to understand how these functions work. For example, study the following functions:
   (a) `Levinson()` in the file `levinson.c`, to understand how to calculate the LPC coefficients recursively as defined in (9.4) through (9.9). The MATLAB® function `levinson()` used in Example 9.2 can be used for comparison.
   (b) `Weight_a()` in the file `weight_a.c`, to understand how to compute the weighting filter coefficients.
   (c) `Oversamp_16k()`, `Decim_12k8()`, `Down_samp()`, and `Up_samp()` in the file `decim54.c`, to understand how to change the sampling rate.
   (d) `norm_l()` and `round()` in the file `basicop2.c`, to understand how to normalize 16-bit numbers and round the lower 16 bits with saturation.

Table 9.19 File listing for the experiment Exp9.9

Files	Description
`g711Test.c`	Program for testing G.711 experiment
`g711.c`	G.711 implementation
`g711.h`	C header file defines constant and function prototyping
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`c55DSPAudioTest.wav`	Speech data file for experiment

9.4.9 G.711 Companding Using Fixed-Point C

This experiment uses the fixed-point C program to implement the ITU-T G.711 speech codec. Table 9.19 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package. Build and run the experiment to obtain the G.711 encoded bit stream `g711Compressed.cod` and G.711 decoded speech file `g711Expanded.wav`.
2. Listen to the expanded (linear) speech file `g711Expanded.wav` and compare it to the original speech file `c55DSPAudioTest.wav`.
3. Redo the experiment with different speech files and music files sampled at the 8 kHz sampling rate. Observe if G.711 companding affects the music quality.
4. Change the companding law from the μ-law to A-law. Compare the processed results to the ones obtained by using the μ-law standard. Examine if there is any difference by listening to the results vs. the original sources.
5. Texas Instruments provides several speech and audio codec libraries [18]. The C55xx-based codecs are available at http://www.ti.com/tool/c55xcodecs. Download the G.722.2 speech codec and replace the G.711 with the G.722.2 libraries for C55xx to conduct a new experiment using the G.722.2 speech codec. Compare the G.722.2 speech codec encode and decode performances and voice quality to the experimental results using the G.711 (at the 8 kHz sampling rate).

9.4.10 Real-Time G.711 Audio Loopback

This experiment implements G.711 using the fixed-point C for real-time demonstration. Table 9.20 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Connect an audio source and a headphone to the eZdsp. Import the CCS project from the companion software package. Build and run the experiment.
2. Listen to the output audio and compare the G.711 audio quality to the uncompressed signal.
3. Change the sampling rate from 8 to 16 kHz. Listen to the output audio and compare the G.711 audio quality between the 16 and 8 kHz sampling rates.
4. Change the sampling frequency from 8 to 16 kHz. Repeat step 5 in Section 9.4.9. Compare the audio quality between G.711 at the 16 kHz sampling rate and G.722.

Table 9.20 File listing for the experiment Exp9.10

Files	Description
realtime_g711Test.c	Program for testing real-time G.711 experiment
realtime_g711.c	Real-time framework for audio loopback experiment
g711.c	G.711 implementation
vector.asm	C5505 interrupt vector table
g711.h	C header file defines constant and function prototyping
tistdtypes.h	Standard type define header file
dma.h	Header file for DMA functions
dmaBuff.h	Header file for DMA data buffer
i2s.h	i2s header file for i2s functions
Ipva200.inc	C5505 processor include file
myC55xUtil.lib	BIOS audio library
c5505.cmd	Linker command file

5. Replace the G.711 codec with the Texas Instruments G.722.2 C55xx libraries to create a G.722.2 real-time audio loopback experiment.

Exercises

9.1. The combined filter $H(z)$ defined in (9.20) can be simplified by using $\gamma_1 = 1$ in (9.19), which is the actual mechanism used by G.729A. The magnitude responses of the weighting filter with three different values of γ_2 are plotted in Figure 9.6. Use the MATLAB® script from Example 9.5 to plot the magnitude response of the filter for $\gamma_1 = 0.98$ and compare the corresponding magnitude responses.

9.2. Explain why the weighting filter will not work properly by setting $\gamma_2 = 1$ and $\gamma_1 < 1$ in (9.19) for perceptive weighting. Which spectral components will be emphasized in such settings? Can this filter function properly for other applications? (Hint: it can be used as a postfilter for the decoder to make the frequency components under the formant peaks sound sharper, especially when the LPC coefficients are quantized.)

9.3. For the ACELP representation, how many bits are needed to encode the eight possible positions? If the pulse amplitude is either $+1$ or -1, how many bits are needed to encode the pulse position and sign? If the pitch interval is from 20 to 147, how many bits are needed for encoding? If we need higher resolution, say the resolution of a 1/2 sample, how many bits are needed for the pitch range from 20 to 147?

9.4. The LSP coefficient vector defined in G.723.1 is a 10th-order vector. This vector is divided into three sub-vectors of lengths 3, 3, and 4. If each sub-vector is vector quantized using an 8-bit codebook, how many bits are needed to represents the LSP quantization index?

9.5. The LSP coefficients are very important for reconstructing the speech. In order to keep the LSP coefficients from corrupting during the transmission, some applications need to protect the LSP index with an error correction code. In the case where the LSP index has been corrupted and is not able to recover, what is the best way to reconstruct the LSP coefficients for the current frame based on the history information available from previous frames?

9.6. If we use 16 bits to represent the LPC coefficients, and assume the values of these coefficients are always between ±3.00, what is the most efficient 16-bit fixed-point representation for these coefficients?

9.7. Assume VoIP applications use 60 bytes for Ethernet/IP/UDP/RTP headers. If G.729 is used (20 ms packet frame size), what is the actual bit rate over the IP network? If the voice activity is 40% (which means that 60% of the time the signal is silence), how much can be saved in bandwidth if nothing is transmitted for the silent frames? If we increase the packet frame size from 20 to 40 ms, what is the actual bit rate over networks during the active frames?

9.8. Repeat Problem 9.7 by using G.711. Compare the payload bit-rate ratio and the actual bit-rate ratio over the IP for G.729 and G.711. Note that the payload is the pure bits produced by the codec, where G.711 produces 64 kbps and G.729 produces 8 kbps.

9.9. For an ACELP codec such as G.729, if the first three pulse locations have been found to be at 5, 6, and 7, is it possible for the fourth pulse to be at 8? Is it possible for the fourth pulse to be at 9? Why or why not?

9.10. Given the speech file `TIMIT1.ASC` in Problem 8.2 of Chapter 8, implement the energy estimator defined in (9.23) using both long and short windows. Plot the estimated energy on the same plot with the speech signal.

9.11. Given the speech file `TIMIT1.ASC` in Problem 8.2 of Chapter 8, implement the noise floor estimator defined in (9.24). Plot the estimated noise floor on the same plot with the speech signal. Observe the noise floor at the beginning and end of the speech.

9.12. Add low-level white noise to the speech file `TIMIT1.ASC` and use the VAD algorithm to detect the silent and speech frames. Plot the VAD result and the speech on the same figure for comparison.

9.13. Explain why different window sizes are required for calculating the signal energy used by the VAD algorithm. During the onset of speech, which output is bigger using (9.22) and (9.23)? What happens at the offset (tail) of speech?

9.14. The musical tone effect from the spectrum subtraction process is due to the over-subtraction of certain frequencies. In order to reduce this effect, a mild subtraction should be applied. As a result, the amount of noise cancellation will be decreased as compared to the full subtraction. Write a MATLAB® function to control this parameter among the different subtraction factors expressed in (9.35).

9.15. Use the experiment given in Section 9.4.3 to calculate the percentage of silent frames for speech file `speech.wav`. In a normal conversation, there should be over 50% of silent frames.

References

1. ITU-T Recommendation (1996) G.723.1, Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 & 6.3 kbit/s, March.
2. CCITT Recommendation (1992) G.728, *Coding of Speech at 16 kbit/s Using Low-delay Code Excited Linear Prediction.*

3. ITU-T Recommendation (1995) G729, Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear Prediction (CS-ACELP), December.
4. ITU-T Recommendation (2002) G722.2, Wideband Coding of Speech at Around 16 kbit/s Using Adaptive Multi-rate Wideband (AMR-WB), January.
5. 3G TS 26.190 V1.0.0 (2000-12) (2000) Mandatory Speech CODEC Speech Processing Functions; AMR Wideband Speech CODEC; Transcoding Functions (Release 4), December.
6. 3GPP TS 26.171 (2001) Universal Mobile Telecommunications System (UMTS); AMR Speech CODEC, Wideband; General Description (Release 5), March.
7. Bessette, B., Salami, R., Lefebvre, R. *et al*. (2002) The adaptive multi-rate wideband speech codec (AMR-WB). *IEEE Trans. Speech Audio Process.*, **10** (8), 620–636.
8. Kondoz, A.M. (1995) *Digital Speech Coding for Low Bit Rate Communications Systems*, John Wiley & Sons, Inc., New York.
9. Kuo, S.M. and Morgan, D.R. (1996) *Active Noise Control Systems – Algorithms and DSP Implementations*, John Wiley & Sons, Inc., New York.
10. Tian, W., Wong, W.C., and Tsao, C. (1997) Low-delay subband CELP coding of wideband speech. *IEE Proc. Vision Image Signal Process.*, **144**, 313–316.
11. Tian, W. and Wong, W.C. (1998) Multi-pulse embedded coding of speech. Proceedings of IEEE APCC/ICCS'98, Singapore, November, pp. 107–111.
12. Tian, W., Wong, W.C., Law, C.Y., and Tan, A.P. (1999) Pitch synchronous extended excitation in multi-mode CELP. *IEEE Commun. Lett.*, **3**, 275–276.
13. Tian, W. and Wong, W.C. (1999) 6 kbit/s partial joint optimization CELP. Proceedings of ICICS'99, Sydney, December, CDROM #1D1.1.
14. Tian, W. and Alvarez, A. (1999) Embedded coding of G.729x. Proceedings of ICICS'99, Sydney, December, CDROM #2D3.3.
15. Tian, W., Hui, G., Ni, W., and Wang, D. (1993) Integration of LD-CELP codec and echo canceller. Proceedings of IEEE TENCON, Beijing, October, pp. 287–290.
16. Chen, J.H. and Gersho, A. (1995) Adaptive postfiltering for quality enhancement of coded speech. *IEEE Trans. Speech Audio Process.*, **3**, 59–71.
17. Papamichalis, P.E. (1987) *Practical Approaches to Speech Coding*, Prentice Hall, Englewoods Cliffs, NJ.
18. Texas Instruments, Inc. (1997) A-Law and mu-Law Companding Implementations Using the TMS320C54x, SPRA163A, December.

10

Audio Signal Processing

Digital audio signal processing techniques are widely used in consumer electronics such as CD players, portable audio players, high-definition televisions, and home theaters. For professional audio, the applications can be found in the fields of digital sound broadcasting, audio program distribution, computer music, studio audio recording and editing, digital cinema, and digital storage. This chapter introduces some audio processing applications including audio coding algorithms, audio graphic and parametric equalizers, and several audio effects. Examples and experiments are designed to help better understand DSP algorithms for audio applications.

10.1 Introduction

The audio CD used to be a very popular digital audio format. It stores audio signals using the 16-bit pulse code modulation (PCM) format at a sampling rate of 44.1 kHz, so the two audio tracks for stereo audio require a bit rate of 1411.2 kbps. The digitized audio signals using this uncompressed PCM format are referenced as CD-quality. Unfortunately, this simple method presents a challenge to channel bandwidth and storage capacity for emerging digital audio applications such as multimedia streaming over the Internet, wireless mobile systems, teleconferencing, and digital radio broadcasting. The increasing demand for higher quality digital audio, such as multi-channel audio coding (5.1 and 7.1 channels) and higher sampling rate (96 kHz) for professional audio, requires more sophisticated encoding and decoding techniques in order to minimize the cost associated with transmission and storage.

The efficient speech coding techniques based on the vocal-tract model discussed in Chapter 9 are not applicable to audio coding in general. In order to satisfy stereo or multi-channel signal presentation, higher sampling rate, higher resolution, wider dynamic range, and even higher listener expectations, audio signal processing algorithms have been developed using advanced techniques including psychoacoustics, transform coding, Huffman coding, and inter-channel redundancy reductions. The combination of these techniques leads to the developments of perceptually transparent high-fidelity coding algorithms for CD-quality digital audio. In addition to the basic requirements of low bit rate and high

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

quality, robustness to channel bit errors and packet loss, and low complexity encoders and decoders, are also required for some applications.

Many audio coding algorithms have become international standards for commercial products. In particular, MPEG-1 layer 3 (MP3) is a very popular media format for internet audio delivery and portable audio players. MP3 provides bit rates ranging from 32 to 320 kbps, and supports bit-rate switching between audio frames. MP3 at 320 kbps produces comparable audio quality to the CD at 1411.2 kbps. In this chapter, the MP3 coding standard [1–4] will be discussed in detail as an audio coding example.

This chapter also introduces some audio effects such as bass and treble boosters, audio graphic and parametric equalizers [5], as well as techniques used for these applications. The DSP principles and algorithms discussed in previous chapters will be used to realize these interesting applications. The techniques introduced in this chapter include the design of shelf filters, peak and notch filters, comb filters, and allpass filters.

10.2 Audio Coding

Speech coding techniques have been introduced in Chapter 9. Compared to speech, music has a much wider bandwidth and comes with multiple channels. Psychoacoustic principles, including human hearing characteristics in determining the quantization levels at different frequency bands, have been studied and used for audio coding. For example, if the frequency component is below the hearing threshold, this signal component will not be coded based on the principle of auditory masking [6].

10.2.1 Basic Principles

Lossy compression can be used for speech and audio signals based on noise shaping, where the noise below the masking threshold may not be audible. Entropy coding can also be applied to audio because of the large amount of data from the high sampling rate and multiple channels. This section introduces the principles of lossy audio compression based on psychoacoustics and entropy coding using Huffman coding.

Figure 10.1 shows the basic structure of an audio coder–decoder (CODEC or codec) device. The function of each module will be briefly described as follows, and some modules will be further discussed in the subsequent sections.

Filterbank and transform: As shown in Figure 10.1, there are two highly related filterbanks used for the coder and decoder. The coder uses an analysis filterbank to split the full-band signal into several uniformly spaced subbands or according to the critical band model of the human auditory system, while the decoder uses a synthesis filterbank to reconstruct subband signals into a full-band signal. For example, MPEG-1 uses 32 subbands. The transform converts the time-domain signals to frequency-domain (spectral) coefficients for processing. MP3 uses the modified discrete cosine transform (MDCT). MPEG-2 AAC (advanced audio coding) and Dolby AC-3 [7,8] use the MDCT as a filterbank to split full-band signals.

Psychoacoustic model: Calculates the masking threshold according to human auditory masking effects from the spectral coefficients, and uses the masking threshold to quantize the MDCT coefficients.

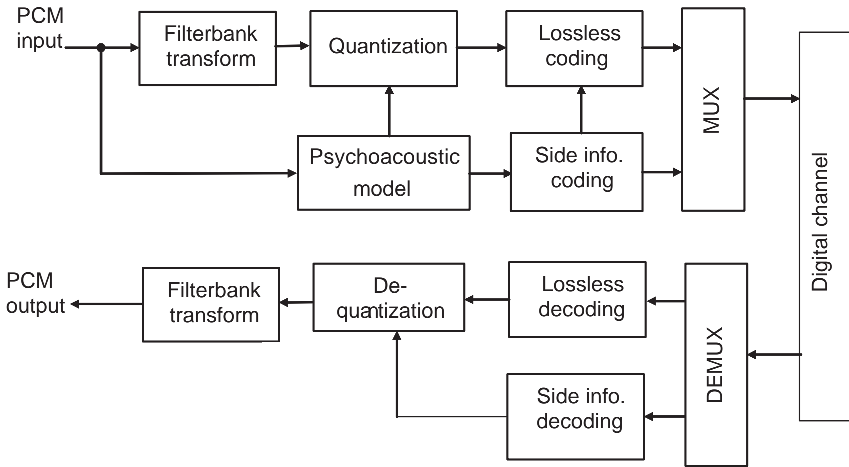


Figure 10.1 Basic structure of audio CODEC

Lossless coding: Further reduces the redundancy of the coded bit stream using entropy coding. For example, MP3 uses Huffman coding.

Quantization and de-quantization: Quantize the MDCT coefficients to indices based on the masking threshold provided by the psychoacoustic model in the coder, and convert the indices back to the spectral coefficients in the decoder.

Side information coding: Codes bit allocation information needed for the decoder.

Multiplexer and de-multiplexer: Pack the coded bits into the bit stream and unpack the bit stream back to the coded bits.

Figure 10.2 shows the format of the encoded bit stream to be sent over the digital channel for transmission [9,10]. Each field is briefly defined as follows:

Header: Contains information about the frame format. It starts with a synchronization word used to identify the beginning of a frame. The header also contains information about the bit stream, including the layer number, bit rate, sampling frequency, and stereo encoding parameters. The length of the header field varies from one audio standard to another. For example, the MP3 header field uses 32 bits.

CRC: Uses the error detection code CRC (cyclic redundancy checksum) to protect the header. This field is optional. If it is present, the decoder calculates the header field CRC in each frame and compares the result to the CRC embedded in the stream. If two CRC codes do not match, the decoder looks for a new sync word. For example, the sync word for MP3 is a fixed 12-bit word 0xFFF.

Header	CRC (optional)	Side information	Main data	Ancillary information

Figure 10.2 Typical encoded audio bit-stream format

Side information: Contains information for the decoder to process the main data. Several decode processing steps will use this global information. For example, when MP3 performs Huffman decoding, the information for the Huffman table used by the encoder is stored in the side information. The Huffman encoded data and the side information are combined in a single bit stream.

Main data: Consists of coded spectral coefficients and lossless encoded data. For example, MP3 data includes scaling factors to reconstruct the original frequency lines from information in the data.

Ancillary data: Holds user-defined information such as the title of a song or other optional album information.

In this section, we focus on the auditory masking effects for perceptual coding. Auditory masking is based on the psychoacoustic principle that a low-level signal (the maskee) becomes inaudible when a louder signal (the masker) occurs simultaneously. This is because human hearing does not respond equally to all frequency components. This phenomenon can be exploited in speech and audio coding by an appropriate noise shaping in the encoding process. The auditory masking depends on the spectral distribution of masker and maskee, and on their variations with time. The perceptual weighting method used for speech coding has been discussed in Chapter 9. Audio coding is also based on similar auditory masking effects but with wider signal bandwidth.

The quiet (absolute) threshold is approximated by the following nonlinear function [11]:

$$T_{\mathrm{q}}(f) = 3.64(f/1000)^{-0.8} - 6.5\mathrm{e}^{-0.6(f/1000-3.3)^2} + 10^{-3}(f/1000)^4 \text{ dB SPL}, \qquad (10.1)$$

where f is the frequency in Hz, and SPL represents sound pressure level.

The quiet threshold defined by (10.1) represents a listener with acute hearing such that any signal level below this threshold will not be perceived. The approximation of quiet threshold using a log scale to represent frequency is shown in Figure 10.3. Most humans cannot sense frequencies outside the range from 20 Hz to 20 kHz. This range becomes narrower as a person becomes older. For example, a middle-aged man may not hear many of the frequency components above 16 kHz. Frequencies in the region from 2 to 4 kHz are the easiest to perceive at relatively low levels. Figure 10.3 also shows the frequency-domain phenomenon called simultaneous masking, in which a low-level signal can be masked (inaudible) by a simultaneously occurring stronger signal if the maskee is close to the frequency of masker (e.g., 1 kHz tone).

Since the playback volume is unknown during the analysis, we need SPL normalization. As shown in Figure 10.3, the lowest threshold is at 4 kHz. SPL normalization ensures that a 4 kHz signal of ± 1-bit amplitude will correspond to the SPL of 0 dB. In this way, a 16-bit full-scale sinusoid will be associated with the SPL of 90 dB. Assuming a Hanning window with 6 dB attenuation is used, a 16-bit full-scale sinusoid that can be precisely resolved by the 512-point FFT will yield a spectral line of 84 dB SPL. After normalization, the playback SPL estimate for the lowest amplitude 16-bit input signal will be at or below the absolute threshold.

The first step of perceptual coding uses the absolute hearing threshold to shape the coding distortion spectrum. However, the most useful threshold is the masking threshold when stimuli (audio signals) are present. Therefore, the detection threshold for spectral quantization noise

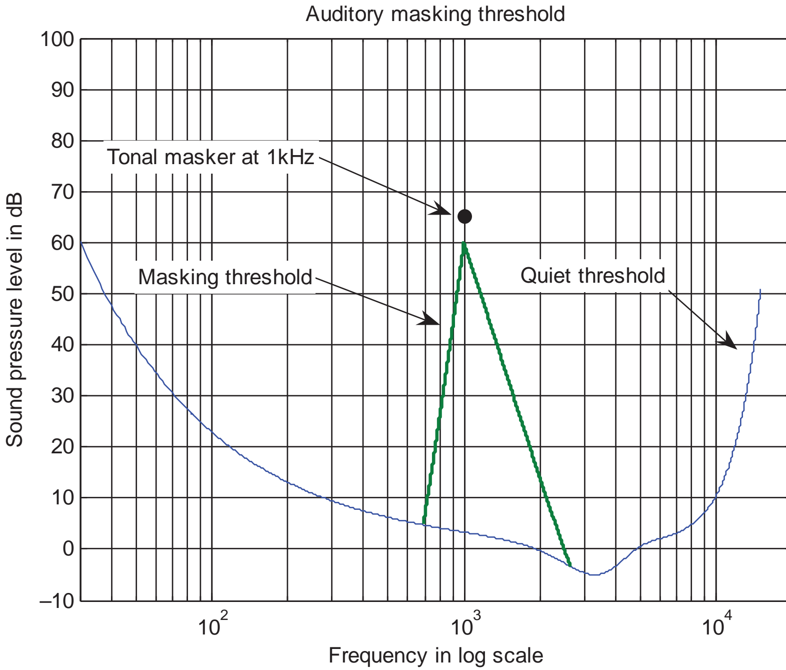


Figure 10.3 Auditory masking thresholds

consists of the absolute threshold and the shape determined by the stimuli at a given time. Since stimuli are time varying, the masking threshold is also a time-varying function.

Human hearing does not respond linearly to all frequency components. The auditory system can be roughly divided into 26 critical bands; each band can be represented as a bandpass filter with bandwidth from 50 to 100 Hz for a frequency below 500 Hz, and with bandwidth up to 5000 Hz at higher frequencies. Within each critical band (or bark), the auditory (also called psychoacoustic) masking threshold can be determined. Different frequencies within the same critical band are difficult to distinguish by human hearing.

The conversion from frequency to bark (critical band) can be approximated using the following equation [11]:

$$z(f) = 13\tan^{-1}(0.000\,76f) + 3.5\tan^{-1}[(f/7500)^2]\ \text{bark}. \tag{10.2}$$

This equation converts frequency in Hz to the bark scale. Thus, one critical band comprises 1 bark. For example, if $f = 1000$ Hz, $z(f) = 8.51$. This frequency falls into the ninth critical band, that is, ninth bark.

Suppose there is a dominant tonal component in the audio signal. This dominant frequency will introduce a masking threshold that masks out frequencies inside the same critical band. In other words, we do not need to code other frequency components within this critical band. This frequency-domain masking phenomenon within a critical band is known as simultaneous masking. The masking effect crossing the adjacent critical bands is known as the masking

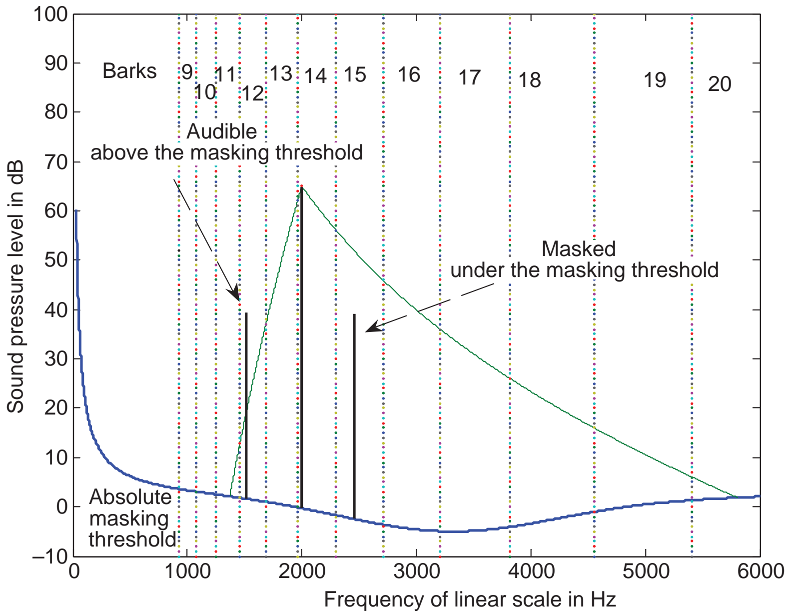


Figure 10.4 Masking effect of a 2 kHz tone

spread. It is often modeled in coding applications by the triangular spreading function that has a slope of 25 dB per bark for the lower frequencies and a slope of −10 dB per bark for the higher frequencies. The 2 kHz tone masking threshold shown in Figure 10.4 is calculated based on this model.

Example 10.1

Given a 65 dB tone at 2 kHz and two 40 dB test tones at 2.5 and 1.5 kHz, calculate if it is necessary to code these two test tones based on the masking threshold and the triangular spreading function that has slopes of 25 dB and −10 dB per bark to calculate the masking effect.

We first use (10.2) to calculate the masking threshold corresponding to the 2 kHz tone with 65 dB. The magnitude contours of masker and masking threshold are plotted in Figure 10.4. It is clear that there are several bark bands between 1.5 and 2.5 kHz. Due to the sharper slope (25 dB per bark) on the left side (lower frequency) of the masker tone, the test tone at 1.5 kHz is above the masking threshold. Therefore, the 1.5 kHz tone cannot be masked and will be left as a masker tone. On the other hand, due to the slower slope (−10 dB per bark) on the right side (higher frequency) and wider frequency coverage of the bark bands at higher frequencies, the 2.5 kHz tone is under the masking threshold of the 2 kHz masker tone. This example shows that the masking effect is not symmetric about the masker tone, with more masking effect at higher frequencies.

10.2.2 Frequency-Domain Coding

MDCT is widely used in audio coding techniques [12,13]. In addition to the same capability of energy compaction as the DCT, the MDCT can simultaneously achieve critical sampling (maximum decimation with compliance to the sampling theorem), reduction of block effects, and flexible window switching.

Before the development of MDCT, the transform-domain audio coding techniques used the DFT or DCT with window functions. These early coding techniques cannot meet the contradictory requirements, that is, critical sampling vs. block effects. For example, when a rectangular windowed DFT (or DCT) analysis/synthesis system is critically sampled, the system suffers from poor frequency resolution and block effects. An overlapped window provides better frequency response but with the penalty of requiring additional blocks in the frequency domain.

The MDCT of $x(n)$, $n = 0, 1, \ldots, N-1$, is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n)\cos\left[\left(n + \frac{N+2}{4}\right)\left(k + \frac{1}{2}\right)\frac{2\pi}{N}\right], \quad k = 0, 1, \ldots, (N/2) - 1, \tag{10.3}$$

where $X(k)$ is the kth MDCT coefficient. It is important to note that if $x(n)$ is a real-valued signal, the MDCT (or DCT) coefficients $X(k)$ are real-valued coefficients, while the DFT coefficients are complex numbers. The inverse MDCT is defined as

$$x(n) = \frac{2}{N}\sum_{k=0}^{N/2-1} X(k)\cos\left[\left(n + \frac{N+2}{4}\right)\left(k + \frac{1}{2}\right)\frac{2\pi}{N}\right], \quad n = 0, 1, \ldots, N-1. \tag{10.4}$$

The relationship between the MDCT and the DFT can be established via the shifted DFT. By shifting the time index n by $(N+2)/4$ and the index k by 1/2, the DFT defined in (5.13) becomes

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j\left[\left(n+\frac{N+2}{4}\right)\left(k+\frac{1}{2}\right)\frac{2\pi}{N}\right]}, \quad k = 0, 1, \ldots, N/2 - 1. \tag{10.5}$$

Similarly, the IDFT with the same shifts is derived as

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)e^{j\left[\left(n+\frac{N+2}{4}\right)\left(k+\frac{1}{2}\right)\frac{2\pi}{N}\right]}, \quad n = 0, 1, \ldots, N-1. \tag{10.6}$$

For real-valued signals, it is easy to prove that the MDCT coefficients in (10.3) and (10.4) are equivalent to the real part of the shifted DFT coefficients in (10.5) and (10.6), respectively.

This fact provides the basis to compute the MDCT using FFT algorithms. To satisfy time-domain aliasing cancellation, the window function must satisfy the following conditions to achieve perfect reconstruction [14]:

1. The analysis and synthesis windows must be the same and the window length N must be an even number.
2. The window coefficients must be symmetric as

$$w(n) = w(N-1-n), \quad \text{for } n = 0, 1, \ldots, (N/2)-1. \tag{10.7}$$

The window coefficients must satisfy the power complementary requirement expressed as

$$w^2(n) + w^2(n+N/2) = 1, \quad \text{for } n = 0, 1, \ldots, (N/2)-1. \tag{10.8}$$

Several windows satisfy these conditions. The simplest but rarely used one is the modified rectangular window expressed as

$$w(n) = 1/\sqrt{2}, \quad 0 \le n \le N-1. \tag{10.9}$$

The window used by MP3 and AC-3 is a sine window defined as

$$w(n) = \sin\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)\right], \quad 0 \le n \le N-1. \tag{10.10}$$

Note that applying the window to the MDCT is different than applying the window for other types of signal analysis. One major difference is that the window must be applied twice to the MDCT and twice to the inverse MDCT.

Example 10.2

This example illustrates the time-domain overlap using 64-point DFT–IDFT block processing. A window is usually applied to the signal before the DFT computation to reduce the passband and stopband ripples. In this example, the Hanning window is used for the 1 kHz tone signal. With a 50% overlap, we need 32 new samples and 32 previous samples to compose the 64-sample block. For each block, the Hanning window will be applied as shown in Figure 10.5(a). For each transformed block, the IDFT is applied to reconstruct the time-domain waveform. Due to the 50% overlap, the output is composed by adding the first 32 samples in the current block to the last 32 samples from the previous block. The output of the processed signal is shown in Figure 10.5(b).

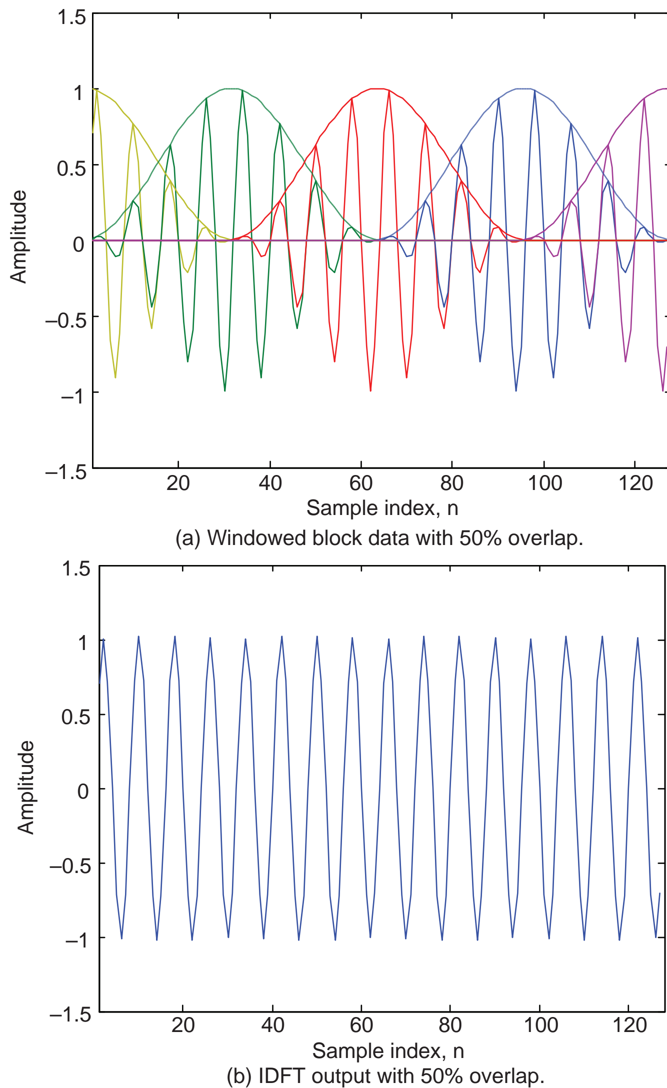


Figure 10.5 DFT/IDFT block processing with 50% overlap.

Pre-echoes are the artifacts that occur even before the presence of music. Pre-echoes are common in the perceptual audio coding schemes that use high-frequency resolution. Switching windows between the length $N = 512$ and $N = 64$ can effectively resolve this problem. For example, AC-3 uses different window sizes to achieve different resolutions. The MP3 decoder also supports window switching between the length of 36 and 12 to increase time-domain resolution. The pre-echo effects will be further discussed in Section 10.5.3.

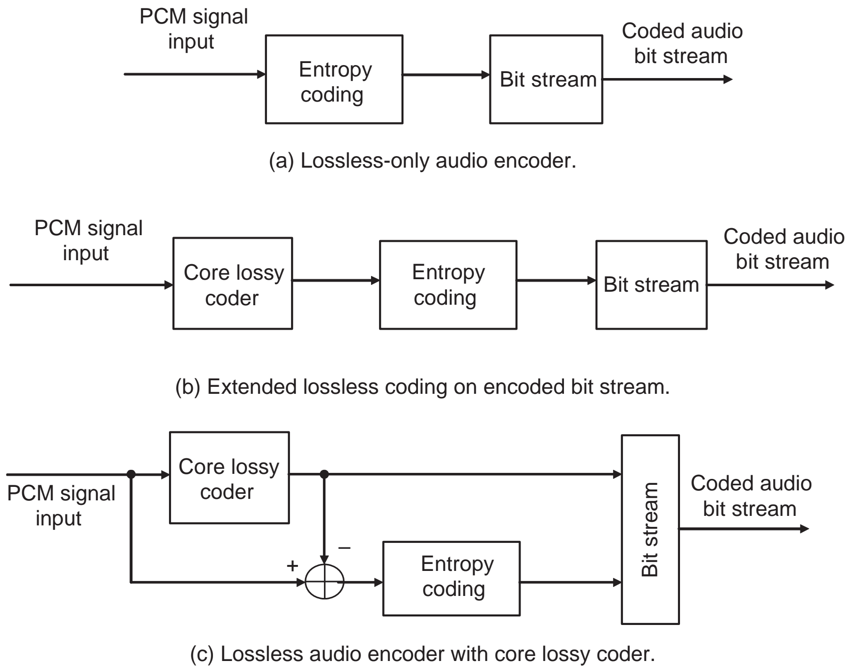


Figure 10.6 Three lossless coder structures

10.2.3 Lossless Audio Coding

Lossless audio coding uses the entropy coding technique to remove the redundancy from the coded data without any loss in quality. Figure 10.6 shows three types of lossless coders. Figure 10.6(a) is a pure entropy coding (or lossless-only) structure using the Huffman encoding scheme. Based on the statistical contents of the input sequence, the symbols are mapped to Huffman codes. Symbols that occur more frequently are coded with shorter code words, while symbols that occur less frequently are coded with longer codes. On average, this method can compress the total number of bits. Huffman coding is extremely fast because it utilizes the lookup table method for mapping quantized coefficients to Huffman codes. The advantage of this technique is that it is simple, but it is difficult to achieve higher compression gain.

Figure 10.6(b) shows a lossy coder followed by entropy coding to further reduce the redundancy. MP3 and MPEG-2 AAC use this method. In MP3, the second compression process, entropy coding, is used after the perceptual coding process. An average of 20% compression gain can be achieved by using Huffman entropy coding.

The MPEG-4 AAC [15] scalable lossless coding standard can generate the bit stream that can reproduce bit-exact audio as the input audio. For this coding scheme, the input audio is first encoded by the AAC encoder. The residual error between the original audio and the AAC encoder output is encoded using the entropy coding method as shown in Figure 10.6(c). The compressed bit stream contains two rates: the lossy bit rate (the same as the encoded AAC bit

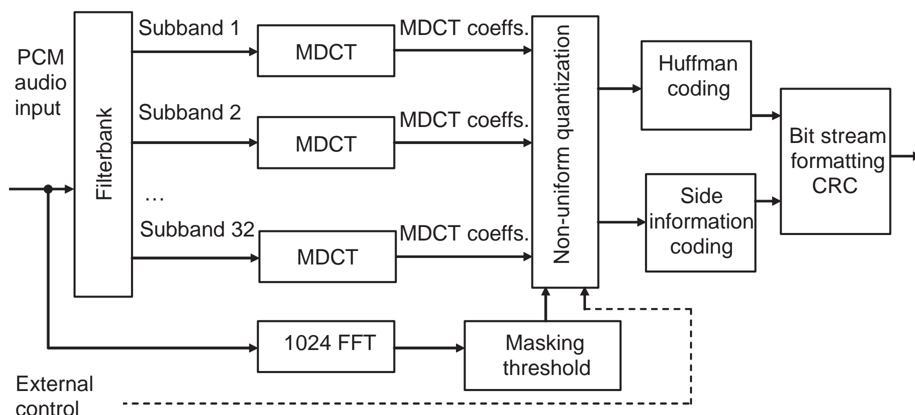


Figure 10.7 Block diagram of MP3 encoder

stream) and the lossless bit rate. At the decoding side, the decoder may use the core lossy encoded bit stream to produce a lossy reproduction of the original audio with lower quality, or use both bit streams to produce the highest quality lossless audio. This flexible scheme is used by the MPEG-4 AAC scalable lossless coding standard [16].

10.2.4 Overview of MP3

The MP3 algorithm uses a filterbank to split the full-band audio signal into 32 subbands. Each subband is critically decimated in the ratio of 32 to 36 MDCT coefficients. Therefore, there are $32 \times 36 = 1152$ samples per frame (about 26 ms). The MP3 encoder is illustrated in Figure 10.7.

The filterbank used by MP3 consists of 32 bandpass filters connected in parallel to split the input full-band signal into multiple subbands. The same polyphase filterbank design discussed in Section 8.6.3 can be used to design the filterbank to produce 32 subband signals. Within each subband, the audio coding process is carried out independently at the decimated rate.

MP3 uses two different MDCT block lengths: a long block of 18 samples and a short block of 6 samples. There is 50% overlap between successive windows so the window sizes are 36 and 12 for the long and short blocks, respectively. The longer block length has a better frequency resolution for stationary audio signals, while the shorter block length provides a better time resolution for the transients. As discussed in Section 10.2.2, the window switching technique can reduce pre- or post-echoes.

As mentioned earlier, MP3 uses 1152 samples per frame, thus each frame has 576 (1152/2) MDCT coefficients. These MDCT coefficients are quantized using a psychoacoustic model with a masking threshold calculated based on the 1024-point FFT coefficients. In Figure 10.7, the control parameters are the sampling rate and the bit rate, as summarized in Table 10.1.

The encoder arranges 576 quantized MDCT coefficients in order of increasing frequency for the mode using the long MDCT block. Since higher energy audio components are concentrated at lower frequencies, the increasing frequency order moves the large values of MDCT coefficients to the lower frequencies and the small values to the higher frequencies.

Table 10.1 MP3 configurations for different sampling and bit rates

Parameters	Configurations
Sampling rate (kHz)	48, 44.1, 32
Bit rate (kbps)	320, 256, 224, 192, 160, 128, 112, 96, 80, 64, 56, 48, 40, 32

This ordered stream is more efficient for Huffman coding. This method is similar to the zigzag operation used in image coding, which will be introduced in Chapter 11. For the short blocks, there are three sets of window values for a given frequency and 192 [(1152/3)/2] MDCT coefficients per block.

Example 10.3

The amplitude distribution of MDCT coefficients is shown in Figure 10.8. The data is obtained using the averaged absolute values of the MDCT coefficients. There are 32 subbands and each subband has 18 samples. These coefficients are scaled by 32 768. It is clear that large coefficients are near DC frequency (indicated by the subband index 0) and very small coefficients with values close to zero are at the higher frequencies.

After ordering the MDCT coefficients, the frequency bins are divided into three regions: run-zero, count-1, and big-value. Starting at the highest frequency, the encoder identifies the continuous run of all-zero values as the run-zeros region. The encoder will remove these frequency bins inside this region and the decoder will add zeros during the decoding process. The count-1 region is the area containing the values of 1, 0 or −1. Finally, low-frequency components in the big-value region are coded in high precision. The big-value region is

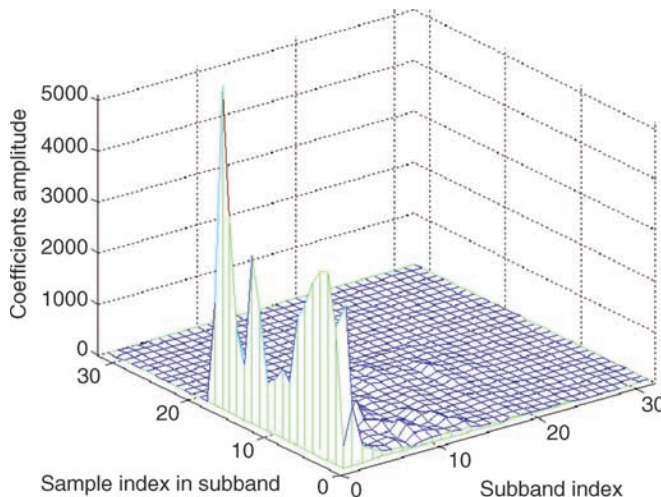


Figure 10.8 Amplitude distribution of MDCT coefficients, where subband sample index is from 0 to 17

further divided into three sub-regions and each region is coded with a different Huffman table that matches the statistics of that region.

MP3 uses 32 Huffman tables for big values. These tables are predefined based on the statistics that are suitable for audio compression. The side information specifies which table is to be used for decoding the current frame. The output from the Huffman decoder is 576 scaled frequency lines represented by integer values.

10.3 Audio Equalizers

Audio spectral equalization uses filtering techniques to reshape the magnitude spectrum of audio signals for sound recording and reproduction. Some hi-fi systems may use relatively simple filters (parametric equalizer) to adjust bass and treble. Some audio systems use equalizers to correct the responses of the microphones, instrument pick-ups, loudspeakers, and hall acoustics. In general, parametric equalizers can provide better specific compensation or alteration around the frequencies of interest, but require more knowledge to operate than for graphic equalizers.

10.3.1 Graphic Equalizers

Graphic equalizers [17] are commonly used in both consumer and professional audio applications to change the magnitude spectra of audio signals. The graphic equalizer uses several frequency bands to display and adjust the power of audio frequency components. It splits the input signal with a bank of parallel bandpass filters, multiplies filtered signals with the corresponding scaling factors, and sums all subband signals to form the full-band signal. Bu using the control (gain) knobs one can adjust the signal amplitudes for each bandpass filter output. The signal power within a band is estimated and displayed using a bar. Figure 10.9 illustrates an example to divide the full band into N bands using an octave scale. The full band is divided into M sections, where $M = 2^N$. Thus for a 10-band equalizer with an octave scale, $M = 1024$. The center frequencies of these bandpass filters are π/M, $2\pi/M$, $4\pi/M$, . . . , $\pi/2$. Two adjacent bandpass filters are overlapped at the frequency that is 3 dB below the passband.

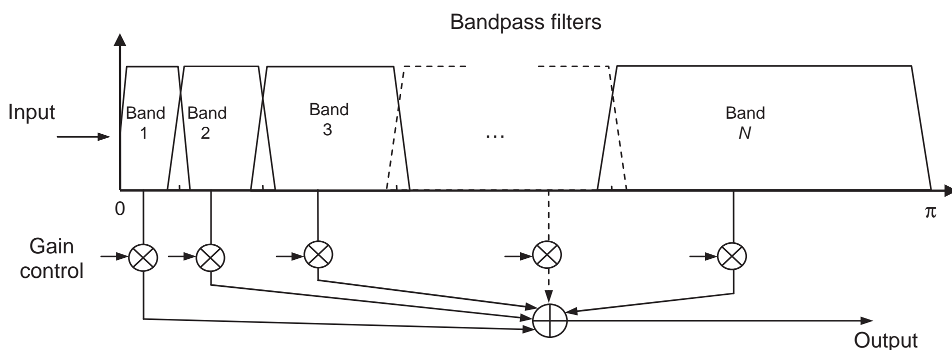


Figure 10.9 An example of an N-band graphic equalizer using octave scale

There are several ways to realize the bandpass filters in order to apply the gains (amplification or attenuation) to the bands of interest, or display the audio signal energy of these bands. One effective way is to use the IIR bandpass filter presented in Chapter 4. Another is to use subband techniques, or the short-time DFT with overlap–add techniques. Example 10.4 uses the DFT and applies the gains to specific bands.

As discussed in Chapter 5, the frequency components of digitized audio signals can be computed using the DFT. In audio applications, the frequency scale usually is not linear, as shown in Figure 10.9. Thus the frequency bins of DFT coefficients $X(k)$ need to be combined to form the desired frequency bands. The principle of combining these bins is simply to consider the octave scale among these bands as crossing the whole spectrum, as illustrated in Example 10.4.

Example 10.4

This is an example of a 10-band graphic equalizer realized using MATLAB® script `example10_4.m`. As shown in Figure 10.10, the gains for 10 bands will be displayed when running the equalizer program. These gains will be applied to the corresponding subband signals, and finally the equalized signal is saved and played. The sampling frequency used for this example is 48 000 Hz.

The frame-by-frame processing is based on the 4096-point FFT with 50% overlap. The 50% overlap–add part is modified from `example10_2.m`. The 10-band center frequencies are specified in the vector `bandFeqs` and their corresponding gains are given in the vector `bandGainIndB`.

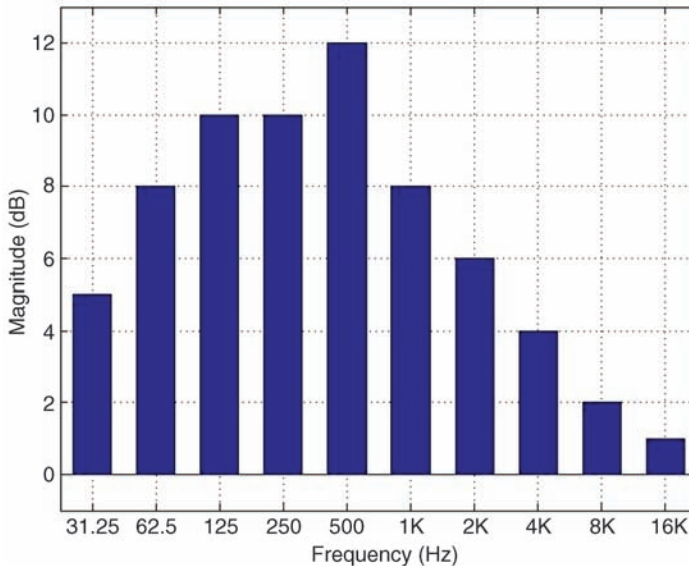


Figure 10.10 An example of a graphic equalizer realized using MATLAB®

```
bandFreqs = {'31.25','62.5','125','250','500','1k','2k','4k','8k','16k'};
bandGainIndB = [5.0,8.00, 10.0,10.0,12.0,8.0,6.0,4.0,2.0,1.0];
bandGainLinear = 10.^(bandGainIndB/20.0); % Linear gain
equalizer(bandGainLinear);                % Equalization

% 10-band gain distribution among 2048 FFT bins
function [out] = GainTbl(bandGain)
   out = ones(2048,1);
   out(1:4) = bandGain(1:1);                  % 0.00000-46.8750 Hz
   out(5:8) = bandGain(2:2);                  % 46.8750-93.7500 Hz
   out(9:16) = bandGain(3:3);                 % 93.7500-187.500 Hz
   out(15:32) = bandGain(4:4);                % 187.500-375.000 Hz
   out(31:64) = bandGain(5:5);                % 375.000-750.000 Hz
   out(65:128) = bandGain(6:6);               % 750.000-1500.00 Hz
   out(129:256) = bandGain(7:7);              % 1500.00-3000.00 Hz
   out(257:512) = bandGain(8:8);              % 3000.00-6000.00 Hz
   out(513:1024) = bandGain(9:9);             % 6000.00-12000.0 Hz
   out(1025:2048) = bandGain(10:10);          % 12000.0-24000.0 Hz
end
```

In the example, 2048 frequency bins are uniformly distributed from DC to 24 kHz, resulting in a frequency resolution of $24\,000/2048 = 11.718\,75$ Hz. These bins are grouped into 10 bands. For example, the fourth bin at 35.156 25 Hz is the bin closest to and less than the cutoff frequency $(31.25 + 62.5)/2$. Thus the first four bins (including the DC bin) are combined as the first band that represents the first frequency band of 31.25 Hz. The FFT bins from 1025 to 2048 are grouped to form the last band with the center frequency 16 kHz. The gains of the 10-band equalizer can be changed by changing the values of `bandGainIndB[]`.

10.3.2 Parametric Equalizers

The graphic equalizer discussed in the previous section contains multiple bandpass filters connected in parallel, where each filter has fixed bandwidth and center frequency. Equalizer gain only adjusts the amplitude of each filter output. A parametric equalizer provides a set of filters connected in cascade form that are tunable in terms of both the spectral shape and filter gain. This section introduces the parametric equalizer using second-order IIR filters with coefficients calculated according to tuning parameters.

There are two types of filters that are specifically designed for parametric equalizers: shelf filers and peak filters. A low-shelf filter will attenuate or boost frequency components below the given cutoff frequency f_c while frequency components higher than f_c will be passed. For example, a low-shelf filter as shown in Figure 10.11(a) boosts frequency components below the cutoff f_c. On the other hand, a high-shelf filter boosts (or attenuates) frequency components above the cutoff f_c, as shown in Figure 10.11(b). The dashed line in Figure 10.11 shows the ideal response, while the solid line is the actual response of the shelf filter that can be achieved using a second-order IIR filter.

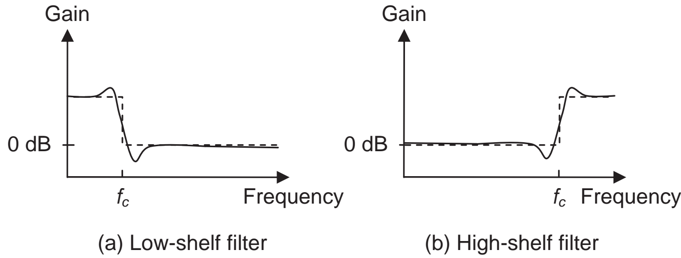


Figure 10.11 Magnitude responses of shelf filters

It is clear that the low-shelf filter is different from the lowpass filter defined in Chapter 3, which drastically attenuates frequency components above the cutoff f_c. Similar comparisons apply to the highpass and high-shelf filters. Usually, a shelf filter may not be used to attenuate or boost more than 12 dB.

The adjustable parameters of the shelf filter are defined as follows:

- f_s – sampling rate;
- f_c – cutoff frequency which is the center (peak) frequency or midpoint (shelf) frequency;
- Q – quality factor, resonance for peak filter or slope for low- and high-shelf filters;
- *Gain* – boost or attenuation in dB.

These parameters determine the coefficients of the filter.

In Chapter 4, Figure 4.7 illustrates the direct-form I realization of the IIR filters. The second-order IIR filter transfer function, defined by (4.30), is expressed as

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}. \tag{10.11}$$

As introduced in Chapter 4, a typical method to design digital IIR filters from the given specifications begins with designing an analog filter in the Laplace domain, then converting it to a digital filter using the bilinear transform. We define the following intermediate variables to simplify the process of transforming the analog filter into the digital filter:

1. K is the normalized gain derived from the *Gain* parameter as

$$|H(\Omega_c)|^2 = 10^{Gain/40\text{ dB}} \equiv K^4, \tag{10.12}$$

   where Ω_c is the analog center frequency.
2. ω_c is the de-warped angular frequency

$$\varpi_c \equiv \tan\left(\frac{\omega_c}{2}\right), \tag{10.13}$$

   where ω_c is the digital center frequency.

Peak Filter Design

Peak (or notch) filters can be used for amplifying (or attenuating) certain narrow frequency bands. In audio terms, they can be used for adjusting the loudness of certain frequency components. The transfer function of the second-order IIR analog filter can be written as

$$H(s) = \frac{s^2 + (K/Q)\Omega_c s + \Omega_c^2}{s^2 + [(1/K)/Q]\Omega_c s + \Omega_c^2}. \tag{10.14}$$

Based on the requirements of the peak filters, the following conditions must be satisfied for this analog prototype filter:

$$\frac{\partial}{\partial\Omega}|H(\Omega)|^2\Big|_{\Omega=\Omega_c} = 0, \tag{10.15a}$$

$$|H(\Omega)|^2\Big|_{\Omega=\Omega_c} = K^4, \tag{10.15b}$$

$$|H(\Omega)|^2\Big|_{\Omega\to 0} = 1, \tag{10.15c}$$

and

$$|H(\Omega)|^2\Big|_{\Omega\to\infty} = 1. \tag{10.15d}$$

These conditions require the peak at Ω_c to have a gain of K^4, and at the start and end of the frequency range (0, ∞), the gain approaches one.

Using the bilinear transform discussed in (4.26), $H(s)$ can be transformed to $H(z)$ by

$$\Omega_c = \frac{2}{T}\tan\left(\frac{\omega_c}{2}\right)$$

and pre-warped $s = s/\Omega_c = s/\left[\frac{2}{T}\tan\left(\frac{\omega_c}{2}\right)\right] = s/\left(\frac{2}{T}\varpi_c\right)$. Replacing s with (4.26), the pre-warped s can be expressed as

$$s/\Omega_c = \frac{2}{T}\left(\frac{1-z^{-1}}{1+z^{-1}}\right)\Big/\left[\frac{2}{T}\tan\left(\frac{\omega_c}{2}\right)\right] = \left(\frac{1-z^{-1}}{1+z^{-1}}\right)\Big/\varpi_c, \tag{10.16a}$$

$$s = \Omega_c\left(\frac{1-z^{-1}}{1+z^{-1}}\right)\Big/\varpi_c, \tag{10.16b}$$

$$\begin{aligned} H(z) &= H(s)\Big|_{s=\Omega_c\left(\frac{1-z^{-1}}{1+z^{-1}}\right)/\varpi_c} \\ &= \frac{(1+\varpi_c K/Q + \varpi_c^2) - 2(1-\varpi_c^2)z^{-1} + (1-\varpi_c K/Q + \varpi_c^2)z^{-2}}{(1+\varpi_c/(KQ) + \varpi_c^2) - 2(1-\varpi_c^2)z^{-1} + [1-\varpi_c/(KQ) + \varpi_c^2]z^{-2}}. \end{aligned} \tag{10.17}$$

Further normalization with respect to $a_0 = 1 + \varpi_c/(KQ) + \varpi_c^2$ is required to obtain the second-order IIR filter as follows:

$$H(z) = \frac{(1+\varpi_c K/Q + \varpi_c^2)/a_0 - [2(1-\varpi_c^2)/a_0]z^{-1} + [(1-\varpi_c K/Q + \varpi_c^2)/a_0]z^{-2}}{1 - [2(1-\varpi_c^2)/a_0]z^{-1} + \{[(1-\varpi_c/(KQ) + \varpi_c^2)]/a_0\}z^{-2}}. \tag{10.18}$$

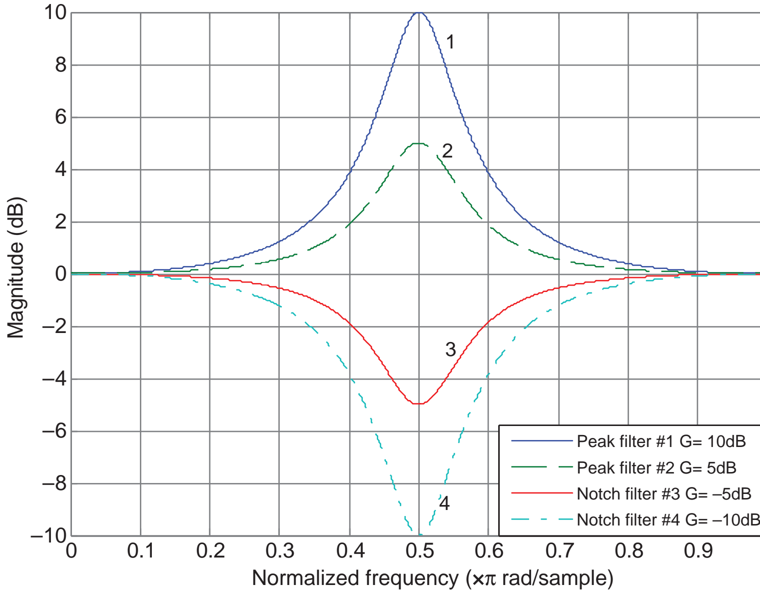


Figure 10.12 Magnitude responses of peak and notch filters with different gain values

Example 10.5

With $f_s = 16$ kHz, $f_c = 4$ kHz, $Q = 2$, and $Gain = 10$, 5, −5, and −10 dB, MATLAB® script `example10_5.m` uses (10.18) to calculate the peak filter coefficients and plot the magnitude responses for four different gain values. Figure 10.12 shows that a peak (first two cases) or a notch (last two cases) filter can be obtained by using a positive or a negative gain.

Design of Low-Shelf and High-Shelf Filters

A low-shelf filter can be used for amplifying (boosting) or attenuating (cutting) frequency components below the predefined frequency to modify the bass loudness of an audio signal. The transfer function of the low-shelf filter can be written as

$$H(s) = \frac{s^2 + (\Omega_c\sqrt{K}/Q)s + K\Omega_c^2}{s^2 + [(\Omega_c/\sqrt{K})/Q)]s + \Omega_c^2/K}, \tag{10.19}$$

where Ω_c is the cutoff frequency. Based on the amplitude responses of shelf filters shown in Figure 10.11, the analog prototype filter must satisfy the following conditions:

$$|H(\Omega)|^2\big|_{\Omega\to 0} = K^4, \tag{10.20a}$$

$$|H(\Omega)|^2\big|_{\Omega=\Omega_c} = 1/K^2, \tag{10.20b}$$

$$|H(\Omega)|^2\big|_{\Omega\to\infty} = 1. \tag{10.20c}$$

Applying (10.16b), we have the following second-order IIR filter for the low-shelf filter:

$$H(z) = \frac{(1+\varpi_c\sqrt{K}/Q+K\varpi_c^2) - 2(1-K\varpi_c^2)z^{-1} + (1-\varpi_c\sqrt{K}/Q+K\varpi_c^2)z^{-2}}{[1+\varpi_c/(\sqrt{K}Q)+1/(K\varpi_c^2)] - 2[1-1/(K\varpi_c^2)]z^{-1} + [1-\varpi_c/(\sqrt{K}Q)+1/(K\varpi_c^2)]z^{-2}}. \tag{10.21}$$

Similar, further normalization with respect to $a_0 = 1+\varpi_c/(\sqrt{K}Q)+1/(K\varpi_c^2)$ is required to normalize a_0 to one in the standard second-order IIR filters.

A high-shelf filter can be used to boost (or cut) frequency components above the cutoff frequency f_c to adjust the loudness or treble of the audio signals. The transfer function of the filter is written as

$$H(s) = \frac{K^2s^2+\Omega_cK\sqrt{K}/Qs+K\Omega_c^2}{s^2+\Omega_c\sqrt{K}/Qs+K\Omega_c^2} \tag{10.22}$$

to satisfy following conditions:

$$|H(\Omega)|^2\Big|_{\Omega\to 0} = 1, \tag{10.23a}$$

$$|H(\Omega)|^2\Big|_{\Omega=\Omega_c} = K^2, \tag{10.23b}$$

$$|H(\Omega)|^2\Big|_{\Omega\to\infty} = K^4. \tag{10.23c}$$

Equation (10.22) is actually the reversed case of the low-shelf filter with a gain of K^4. Applying (10.16b) to (10.22), the IIR filter transfer function can be expressed as

$$H(z) = K\frac{(K+\varpi_c\sqrt{K}/Q+\varpi_c^2) - 2(K-\varpi_c^2)z^{-1} + (K-\varpi_c\sqrt{K}/Q+\varpi_c^2)z^{-2}}{(1+\varpi_c\sqrt{K}/Q+K\varpi_c^2) - 2(1-K\varpi_c^2)z^{-1} + (1-\varpi_c\sqrt{K}/Q+K\varpi_c^2)z^{-2}}. \tag{10.24}$$

Again, further normalization with respect to $a_0 = 1+\varpi_c\sqrt{K/}Q+K\varpi_c^2$ is required to normalize a_0 to one.

Example 10.6

Given $f_s = 16$ kHz, $f_c = 2$ kHz for the low-shelf filters and $f_c = 6$ kHz for the high-shelf filters, $Q = 1$ (or 2), *Gain* = 10, 5, −5, and −10, use (10.21) and (10.24) to calculate the low-shelf filter and high-shelf filter coefficients and plot their magnitude responses.

The MATLAB® script `example10_6.m` uses the function `shelfFilter(Fc, Gain, Q, Fs, sfType)` to design the shelf filters. The magnitude responses of the low-shelf filters for

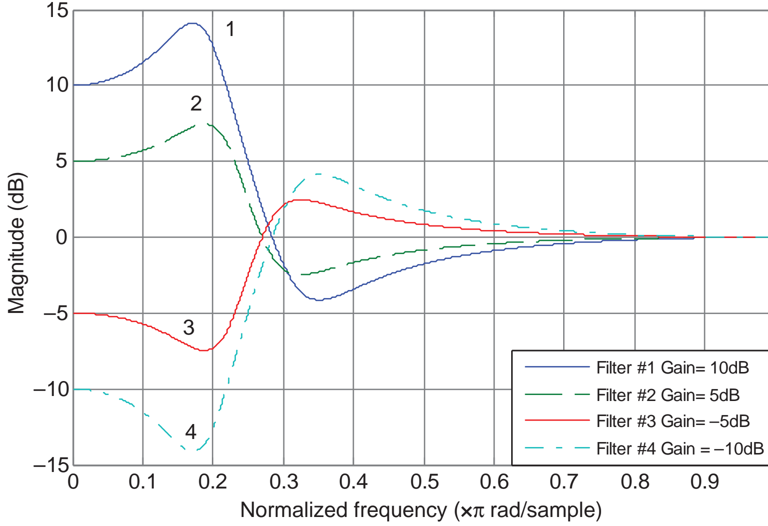


Figure 10.13 Magnitude responses of low-shelf filters for four different gain values

four different gains (fixed Q) are shown in Figure 10.13, and the magnitude responses of the high-shelf filters for four different gains and Q values are shown in Figure 10.14. This latter figure clearly shows that the value of Q (1 or 2) determines the slope of the transition bandwidth, and a higher Q value results in a narrower transition bandwidth.

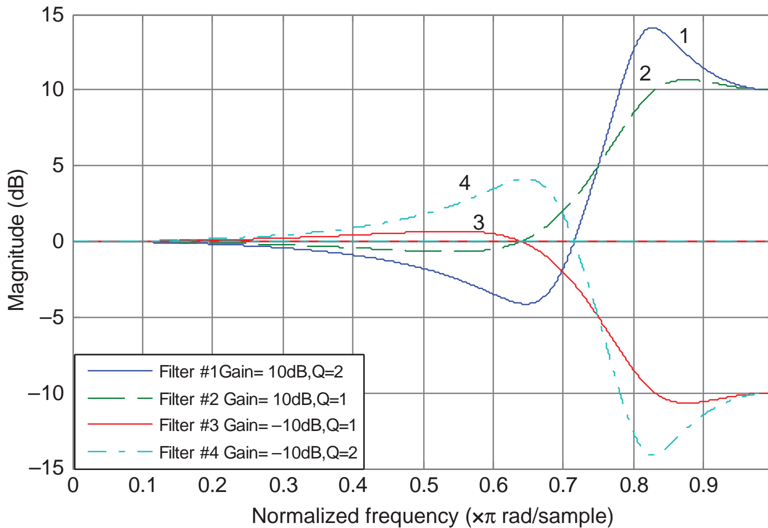


Figure 10.14 Magnitude responses of high-shelf filters for different gain and Q values

Example 10.7

The MATLAB® script `example10_7.m` implements a parametric equalizer using $f_s = 16\,000$ Hz and the following specifications:

Low-shelf filters: $f_c = 1000$ Hz, $Gain = -10$ dB, and $Q = 1.0$
High-shelf filters: $f_c = 4000$ Hz, $Gain = 10$ dB, and $Q = 1.0$
Peak filters: $f_c = 7000$ Hz, $Gain = 10$ dB, and $Q = 1.0$.

The input audio file is `audioIn.pcm`, and the MATLAB® program listed as follows will play the equalized output signal:

```
Fs = 16000;                                  % Sampling frequency
[az1, bz1] = ShelfFilter(1000,-10,1,Fs,'L');
[az2, bz2] = PeakFilter(4000,10,1,Fs);
[az3, bz3] = ShelfFilter(7000,10,1,Fs,'H');

% Combine 3 cascaded IIR filters
da = cascad2x2 (az1,az2);
db = cascad2x2 (bz1,bz2);
az = cascad4x2 (da,az3);
bz = cascad4x2 (db,bz3);

fid1 = fopen('audioIn.pcm','rb');
x = fread(fid1,'short');
fclose(fid1);

y = filter(bz,az,x);                        % IIR filtering
out(:,1)=x/32767;                           % Original audio in left channel
out(:,2)=y/32767;                           % Equalized audio in right channel
wavwrite(out, Fs, 'audioOut.wav');          % Write to output file
disp('play input audio (left channel) vs. output audio (right channel)');
sound(out, Fs);
```

Note that the MATLAB® function `wavwrite` is used to save the equalized signal in the `out` vector to the sound file `audioOut.wav`. This type of wave file can be read using the function `wavread`.

10.4 Audio Effects

Audio effects (or sound effects) can be realized using audio signal processing techniques to emphasize artistic contents of the movies, television shows, live performances, animations, video games, and many others. This section emphasizes DSP-related techniques such as filtering.

Typical sound effects used in recording and broadcasting are reverberation, flanger, phaser, pitch shift, time stretching, spatial sounds, tremolo, and modulation. We will introduce these audio effects and conduct experiments in Section 10.5 using the C programs. Other similar techniques such as echo, equalization, chorus, and vibrato will not be discussed in this book.

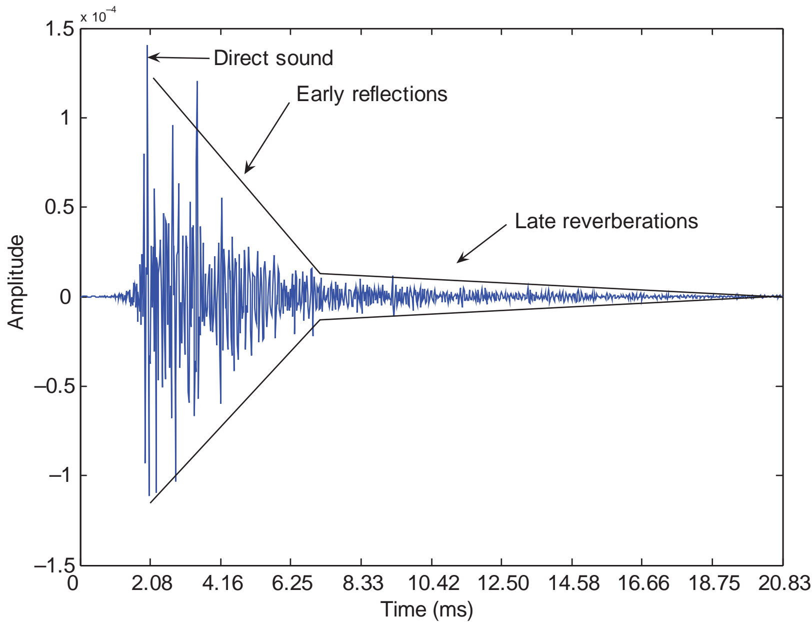


Figure 10.15 An example of a room impulse response

10.4.1 Sound Reverberation

Reverberation is a series of reflected sounds (echoes) in a confined space that reach the listener's ears or microphones. The combination of room size, structure complexities, as well as the angles of the walls, furniture, and decorations in the room, and the density of the surfaces will determine the room impulse response. The impulse response provides a useful time-domain representation of room acoustics.

Figure 10.15 shows a measured room impulse response. Assuming the impulse response of a room is denoted as $h(l)$, $l = 0, \ldots, L-1$, and $x(n)$ is the audio signal played inside the room, the reverberated signal $y(n)$ picked up by a microphone can be calculated using linear convolution as

$$y(n) = \sum_{l=0}^{L-1} h(l)x(n-l). \tag{10.25}$$

For most descriptive reverberation models, the direct sound is defined as the sound wave that first reaches the ears via a direct path without bouncing off surrounding surfaces. Reflected sound, or reverberation, is the wave from the sound source that reaches the listener indirectly, such as sound bouncing off surfaces. Depending on the time that the sound takes to reach the listener, one type of reflection is categorized as early reflection or direct sound reflection. The total amount of early reflections with amplitudes above the noise floor reaching the receiver will depend upon the proximity of the reflecting surfaces to the

measurement point. More spatially diffuse reverberations called late reverberations will follow the early reflections. Figure 10.15 illustrates the segmented impulse response measured in a room of size 246 × 143 × 111 cubic inches, which consists of the direct sound, early reflections, and late reverberations.

Example 10.8

Generate the reverberated sounds of the pre-recorded audio using the given simulated hall impulse response. The audio data and this hall impulse response are digitized at 8 kHz. The MATLAB® script, `example10_8.m`, demonstrates the audio reverberation effect.

In Figure 10.15, the room impulse response has been measured from a real room. There are other methods for creating (simulating) room or hall impulse responses, such as using a model based on considerations of direct path, early reflections, and late-field reverberation.

10.4.2 Time Stretch and Pitch Shift

Time stretch is a process of changing the speed of audio playback without affecting the pitch. This technique can be used for audio editing, for example, to fit a given time interval within an existing audio clip. The application can be found in the fast (or slow) replay of audio or A/V (video with audio) signals.

Pitch shift is the technique that changes the speech's pitch, which is the fundamental frequency of sound. For example, the pitch of an audio signal may be shifted by any preset octave (or interval), up or down. This pitch shift is applied to all frequency components of the entire signal. It is often used in recording and playback systems to raise or lower the pitch. It is also used intentionally for esthetic effects in some pop songs or distinctive animal voices in cartoons.

The phase vocoder [18] is a well-established tool for scaling audio signals using phase information in both the frequency and time domains. Pitch shift of audio and speech signals by the phase vocoder is usually achieved by using a combination of time stretch and sampling rate conversion. For example, to raise the pitch by a factor of 2, one can time-stretch the signal by a factor of 2 and then resample it at half the original sampling rate. The half-rate resampling (or decimation by 2) is used to restore the original duration without altering the original sampling rate. In this way, the resampling shifts the frequency contents by a factor of 2.

There are three key steps: (1) the frequency shift is used to achieve the required pitch shift; (2) the phase alignment is used to maintain the phase coherence of the sound; and (3) the overlap–add method is used to avoid audio boundary effects resulting from DFT block processing.

Frequency Shift

As discussed in Section 5.2.2, the DFT is equivalent to computing N samples of $X(\omega)$ over the interval $0 \le \omega < 2\pi$ at N discrete frequencies $\omega_k = 2\pi k/N$, $k = 0, 1, \ldots, N-1$. The DFT of the windowed sequence $x(n)$ of length N is defined as

$$X(k) = \sum_{n=0}^{N-1} w(n)x(n)e^{-j(2\pi/N)kn}, \quad k = 0, 1, \ldots, N-1, \tag{10.26}$$

where $w(n)$ is the window function (e.g., the Hanning window), and $X(k)$ is the kth DFT coefficient. Since the frequency shift by $\Delta\omega$ is equivalent to Δk expressed as

$$\Delta\omega = 2\pi\Delta k/N, \tag{10.27}$$

the process of the pitch shift is to add Δk to the original signal as

$$X(k+\Delta k) = X(k)\mathrm{e}^{-\mathrm{j}(2\pi/N)\Delta kn}, \quad k = 0, 1, \ldots, N-1. \tag{10.28}$$

Because of the DFT discrete property, the non-integer frequency shift of the DFT bin requires an interpolation process. Assuming the kth DFT bin is to be shifted by Δk, the shifted bin can be calculated using the interpolation of two adjacent bins expressed as

$$X_{\text{shift}}(k) \equiv r_k X(\lfloor k+\Delta k \rfloor) + (1-r_k)X(\lceil k+\Delta k \rceil), \quad k = 0, 1, \ldots, N-1, \tag{10.29}$$

where $\lfloor k + \Delta k \rfloor$ is the largest integer that is not greater than $k + \Delta k$, and $\lceil k + \Delta k \rceil$ is the smallest integer that is not less than $k + \Delta k$; r_k is the interpolation radial, which is linearly proportional to the points between these two bins.

Phase Adjustment

In order to maintain phase coherence from one frame to the next, the phases of the DFT bins must be adjusted to compensate for the modification of the frequencies. Without frequency shift, that is, $\Delta\omega = 0$, the successive short-time Fourier transform (STFT) is phase coherent since it corresponds to the original signal. Assuming that a given DFT bin is frequency shifted by $\Delta\omega$, that is, $\Delta\omega \neq 0$, in order to maintain the phase coherence, the phase difference of the DFT bin between two successive frames must be adjusted by a proper amount based on the modified frequency. For each frequency index k, the phase is pre-adjusted by

$$\Delta\phi(k) = \frac{2\pi(k+1)R}{2N}, \quad k = 0, \ldots, N-1, \tag{10.30}$$

where R is the phase vocoder hop size, which is defined as the number of non-overlapped samples. For example, R is $N/2$ or $N/4$ for 50% or 75% overlap, respectively.

With the shifted DFT bins defined in (10.29) and the phase adjustment given in (10.30), we can combine them to form a new set of DFT bins expressed as

$$Y(k) = X_{\text{shift}}(k)\mathrm{e}^{-\mathrm{j}\Delta\phi(k)}, \quad k = 0, 1, \ldots, N-1. \tag{10.31}$$

Finally, the desired time-domain signal $y(n)$ can be obtained by the IDFT of $Y(k)$ expressed as

$$y(n) = \frac{1}{N}\sum_{k=0}^{N-1} Y(k)\mathrm{e}^{\mathrm{j}(2\pi/N)kn}, \quad n = 0, 1, \ldots, N-1. \tag{10.32}$$

Overlap–Add

Due to the use of the STFT, the time-domain signal components will spread over multiple frames and multiple DFT bins. The result is a combination of using a window and the pre-echo effect. To avoid these border effects due to the block-by-block processing, we overlap two consecutive blocks of time-domain signal. We can use either 50% overlap or a larger 75% overlap. Usually, the integer pitch shift may use 50% overlap but the fractional pitch shift will require a larger overlap in order to maintain phase continuity between two consecutive blocks.

Example 10.9

We can change the pitch and time stretch of an audio signal by using the phase vocoder for the desired audio effects. The phase vocoder source code used for this example can be found at the website http://www.ee.columbia.edu/~dpwe/resources/matlab/pvoc/. The file usage is described as follows:

`pvoc.m` – top-level routine;
`stft.m` – calculate the STFT time–frequency representation;
`pvsample.m` – interpolate/reconstruct the new STFT on the modified time base;
`istft.m` – overlap–add the modified STFT back to the time-domain signal.

In addition, two MATLAB® files, `example10_9a.m` and `example10_9b.m`, are provided for this example. The first program is used for pitch shift and the second one is for time stretch. The spectrogram shown in Figure 10.16(b) shows the pitch shifted up by 1/4 from the original signal in Figure 10.16(a). The time stretch result is shown in Figure 10.16(d) where the time scale has been made 1/4 longer than that of the original signal in Figure 10.16(c).

10.4.3 Modulated and Mixed Sounds

Sound modulations include (1) delay modulation or flanger, (2) phase modulation or phaser, and (3) amplitude modulation or tremolo. These audio effects share the same basic techniques and can be modified from one to another.

Flanger

Flanger creates a modulated sound by adding the variable delayed version of the signal to its original signal [6]. This effect was originally generated by playing the same recording on two synchronized tape players and then mixing the signals together. The flanger effect can be produced easily using DSP techniques.

Flanging can be modeled as the comb filter introduced in Section 3.1.2. However, the fixed delay L is replaced by the time variable $L(n)$ for flanging applications. As shown in Figure 10.17, the input/output relationship of a basic flanger can be expressed as

$$y(n) = x(n) + gx[n - L(n)], \tag{10.33}$$

(a) Spectrogram of the original signal.

(b) Spectrogram after pitch shifted up.

(c) Waveform of the original signal.

(d) Waveform after time stretch.

Figure 10.16 Comparison of pitch shift and time stretch effects.

where $x(n)$ is the input signal at time n, g determines the "depth" of the flanging effect, and $L(n)$ is the variable delay at time n. Usually, g is chosen between [0, 1], where $g = 1$ has the strongest flanging effect.

The amplitude response of the "comb-shaped" filter is shown in Figure 3.3. For $g > 0$, there are L peaks in the amplitude response, centered at frequencies $(2l + 1)\pi/L$, $l = 0, 1, \ldots,$

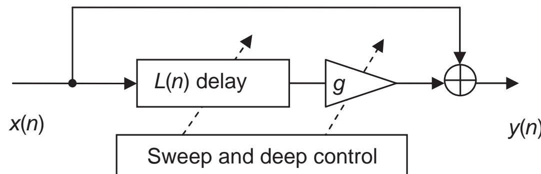


Figure 10.17 Block diagram of basic flanger

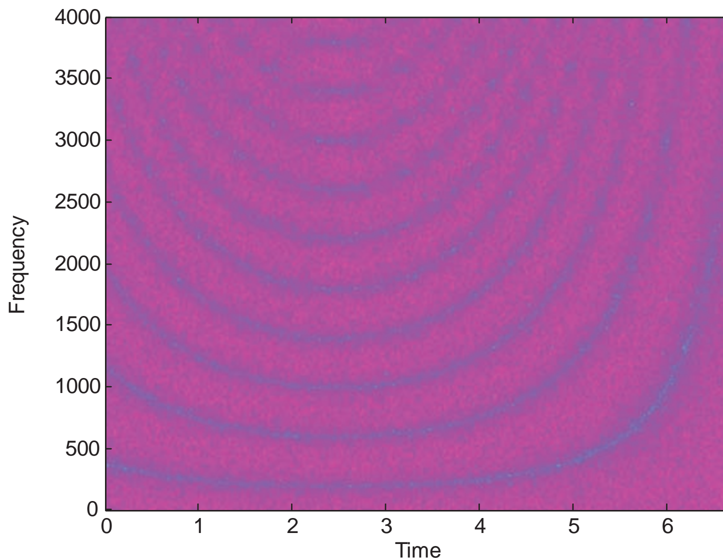


Figure 10.18 Flanger effect of white noise.

$L-1$, with L notches occurring between them at frequencies $z_l = 1, e^{2\pi/L}, \ldots, e^{2(L-1)\pi/L}$. For $g = 1$, these peaks have maximum values.

The time variation of the delay $L(n)$ results in the "sweeping" of uniformly spaced notches in the spectrum. The flanging effect is created by moving the notches in the spectrum, thus notch motion is essential for the flanging effect. Since $L(n)$ must vary smoothly over time, it is necessary to use interpolation techniques to provide smoother non-integer values of $L(n)$.

The delay $L(n)$ is typically changed according to a triangular or sinusoidal waveform, or the delay is modulated by a low-frequency oscillator. The oscillator waveform is usually triangular, sinusoidal, or exponential (triangular on log-frequency scale). For the sinusoidal case, the delay variation can be expressed as

$$L(n) = L_0[1 + A\sin(2\pi f_r nT)], \tag{10.34}$$

where f_r is the speed (or rate) of the flanger in cycles per second, A is the maximum delay swing, T is the sampling period, and L_0 is the average delay controlling the average number of notches. The sine waveform produces a very smooth sound as the pitch is constantly changing. To obtain the maximum effect, the depth control parameter g should be set to one. This parameter is basically the weighting factor between the original (or dry) signal $x(n)$ and the delayed (or wet) signal $x[n - L(n)]$.

Figure 10.18 shows the flanger effects of white noise. Frequency f_r is reflected in this spectrogram. The period of the low-frequency oscillator is 16 000 samples and the maximum delay is 12 samples, which, in turn, produces the maximum frequency shift. Table 10.2 summarizes the parameter values used for generating the flanger effect as shown in Figure 10.18.

As the delay $L(0)$ starts as L_0 at the time index 0, there are six notches crossing the frequency range from 0 to 4000 Hz as shown in Figure 10.18. As $L(n)$ increases with the

Table 10.2 Parameter values used for generating flanger effect in Figure 10.18

Average delay	Maximum delay swing	Flanger speed	Deep	Sampling rate
L_0	A	f_r	g	f_s
6	1.0	0.1	1.0	8000 Hz

time n, the number of notches increases to the maximum of 10, between the time indices 2 and 3. Due to the oscillating effect, $L(n)$ decreases with the time index n after reaching its maximum and repeats such behavior with the rate defined by f_r. As a result of using the oscillated $L(n)$, any sound passing through the flanger will be filtered by a variable comb filter to produce the flanger effect.

Phaser

A phaser creates a synthesized or electronic effect by filtering the audio signal with a serial of allpass filters. As shown in Figure 10.19, the phaser can be realized by mixing the original audio signal and the filtered signal to produce a phase-shifted signal at specific frequencies.

In Figure 10.19, a serial of second-order allpass filters, $H_m(z), m = 1, \ldots, M$, where M is the total number of stages, is cascaded to create different phase shift points. The transfer function of the phaser shown in Figure 10.19 can be expressed as

$$H(z) = 1 + g \prod_{m=1}^{M} H_m(z), \tag{10.35}$$

where the second-order allpass filter $H_m(z)$ can be expressed as

$$H(z) = \frac{b_0 + b_1 z^{-1} + z^{-2}}{1 + b_1 z^{-1} + b_0 z^{-2}} = z^{-2} \frac{D(z^{-1})}{D(z)}, \tag{10.36}$$

where $D(z) = 1 + b_1 z^{-1} + b_0 z^{-2}$. It is easy to show that $|H(\omega)| = 1$, for all ω.

Instead of having a fixed frequency with phase shift, it is typical to apply a slow time-varying oscillation around this frequency. Thus, this frequency component is modulated to create a sweeping effect. Therefore, the digital electronic phaser uses a series of variable allpass phase shift filters to alter the phases of different frequency components.

As discussed in Chapter 4, the allpass filter has flat (equal) amplitude response for all frequencies, thus it changes only the phase of the signal. Human hearing is not very sensitive to phase differences, but the allpass filter creates audible interferences when its output is mixed with the original signal, in which notches have been created.

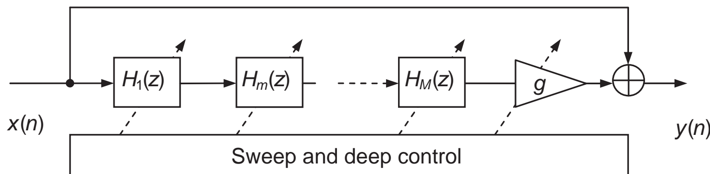


Figure 10.19 A block diagram of a phaser using allpass filters

The tunable allpass filter can be realized with the transfer function

$$H(z) = \frac{c + d(1+c)z^{-1} + z^{-2}}{1 + d(1+c)z^{-1} + cz^{-2}}, \tag{10.37}$$

where $c = [1 - \tan(\pi f_b/f_s)]/[1 + \tan(\pi f_b/f_s)]$ and $d = -\cos(2\pi f_c/f_s)$. Parameter c determines the bandwidth and parameter d adjusts the cutoff frequency. The cutoff frequency f_c determines the point where the phase response passes $-180°$. The width or slope of the phase transition around the cutoff frequency is controlled by the bandwidth parameter f_b. Since Q (quality factor) and f_c are the most popular control parameters in audio recoding or processing, Q is often used to replace f_b in order to have direct control. Thus, the parameter c can be rewritten as

$$c = \frac{1 - \sin(2\pi f_c/f_s)/(2Q)}{1 + \sin(2\pi f_c/f_s)/(2Q)}, \tag{10.38}$$

where $Q = \sin(2\pi f_c/f_s)/2\tan(\pi f_b/f_s)$. Now the allpass filter can be defined by parameters Q and f_c.

The magnitude and phase responses of a second-order allpass filter are shown in Figure 10.20(a), where $Q = 1, f_c = 6000, f_s = 16\,000$, and $g = 1$. It verifies that the magnitude response of the allpass filter is flat and equal to one (0 dB). The phase shift at 6000 Hz (at normalized frequency 0.75) passes $-180°$, and the phase response approaches $-360°$ at high-end frequency. The magnitude and phase responses of the single-stage phaser are shown in Figure 10.20(b), where the phase shift at 6000 Hz (normalized frequency 0.75) passes $-90°$, and the phase response approaches $0°$ at high frequencies.

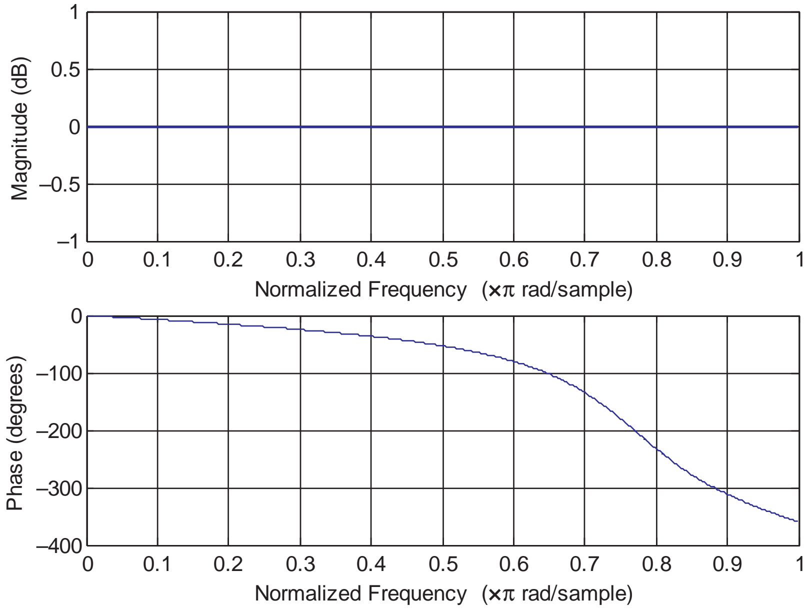
The magnitude and phase responses of a three-stage phaser are shown in Figure 10.21. There are three notches corresponding to the preset normalized frequencies at 0.2, 0.5, and 0.8. Using the low-frequency oscillator with slow changing frequency, the notch frequencies will vary accordingly.

The phaser structure defined in (10.35) and the low-frequency oscillator defined in (10.34) can be used to generate phase modulation effects. This slow frequency-varying effect can be analyzed using a spectrogram to show the changing notch frequencies at different time. The spectrogram of a three-stage phaser modulated by the low-frequency oscillator using white noise is shown in Figure 10.22. It can be seen that the notch frequency varies as a low-frequency sine wave. The notch frequencies are not uniformly spaced. This is different from the flanger effect in which the notch frequencies are always equally spaced. As an example, in Figure 10.22 the upper two notch frequencies at the time index 0.5 are about 1000 Hz apart, while they are more than 3000 Hz apart at the time index 1.5.

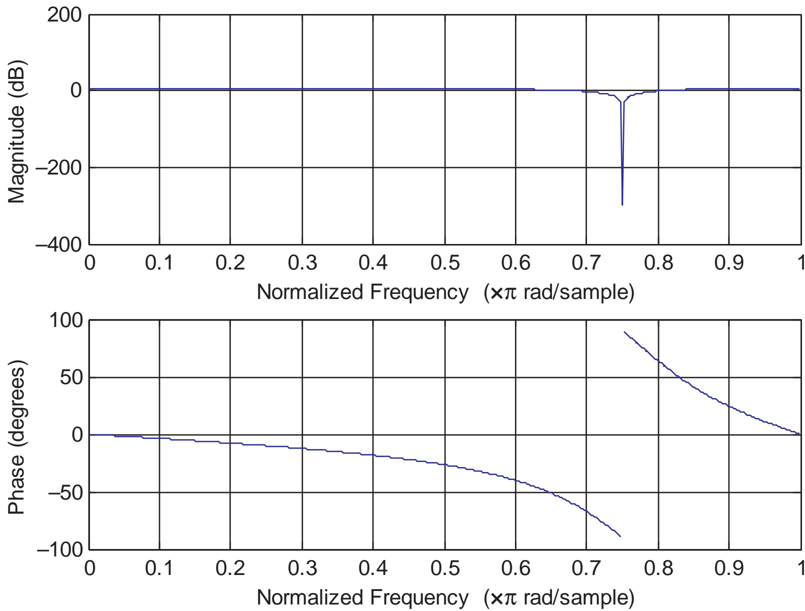
Tremolo

Tremolo is a common audio application for electronic instruments that can be realized using amplitude modulation. This effect can be implemented by multiplying sound with a lower frequency waveform, expressed as

$$y(n) = [1 + AM(n)]x(n), \tag{10.39}$$



(a) Magnitude (top) and phase (bottom) responses of a second-order allpass filter.



(b) Magnitude (top) and phase (bottom) responses of a one-stage phaser.

Figure 10.20 Magnitude and phase responses of allpass filter and phaser

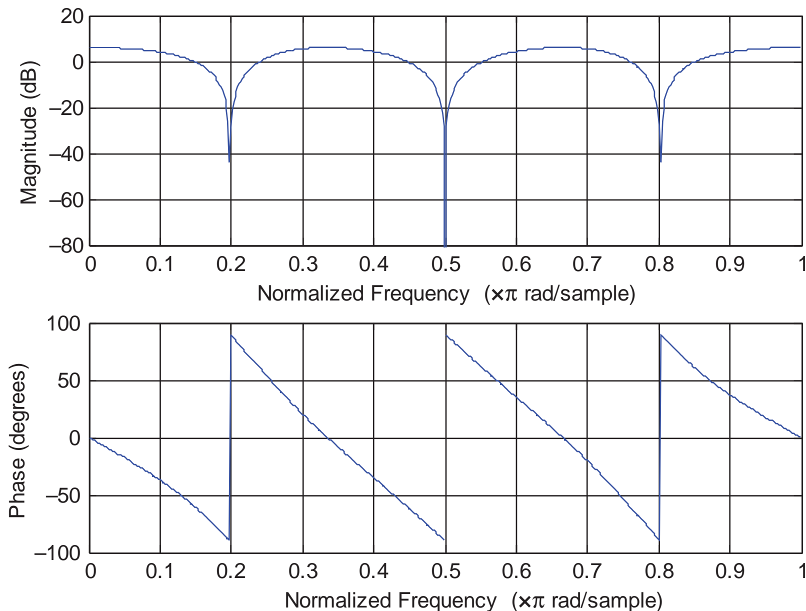


Figure 10.21 Magnitude and phase responses of a three-stage phaser

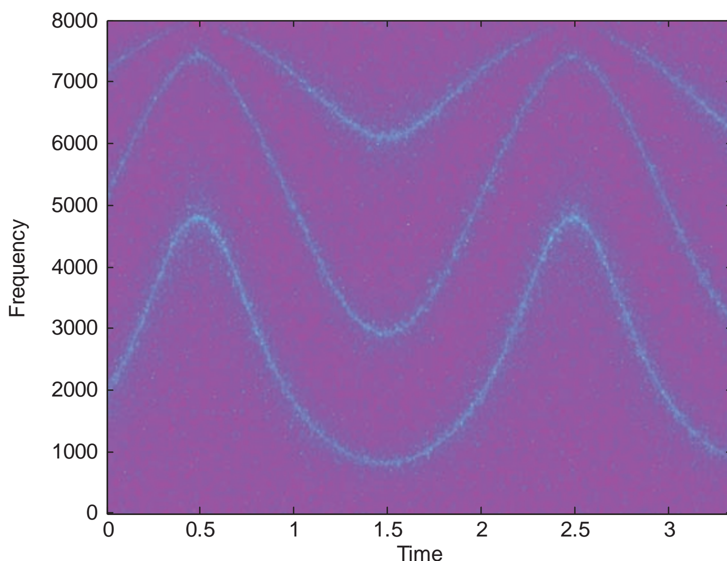


Figure 10.22 Spectrogram of three-stage phasing effect using low-frequency oscillator.

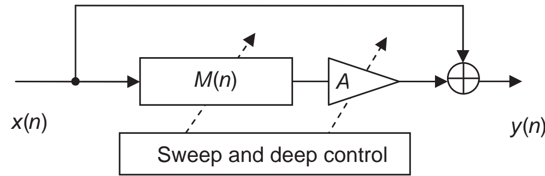


Figure 10.23 A block diagram of tremolo using low-frequency modulation

where A is the maximum modulation amplitude and $M(n)$ is the slow modulation oscillator. The block diagram for implementing the tremolo effect is illustrated in Figure 10.23.

The similar low-frequency oscillator introduced in (10.34) can be used for modulation, which can be expressed as

$$M(n) = \sin(2\pi f_r nT), \tag{10.40}$$

where f_r is the modulation rate and T is the sampling period.

Example 10.10

Refer to the tremolo module as shown in Figure 10.23, given the tremolo method defined in (10.39) with the depth $A = 1$ and the modulation rate at 1 Hz. Use the white noise $x(n)$ as the input signal at a sampling frequency of 8000 Hz to show the tremolo effect.

The resulting tremolo effect for the given conditions is displayed in Figure 10.24, which clearly shows that the signal has been modulated according to the low-frequency 1 Hz oscillator.

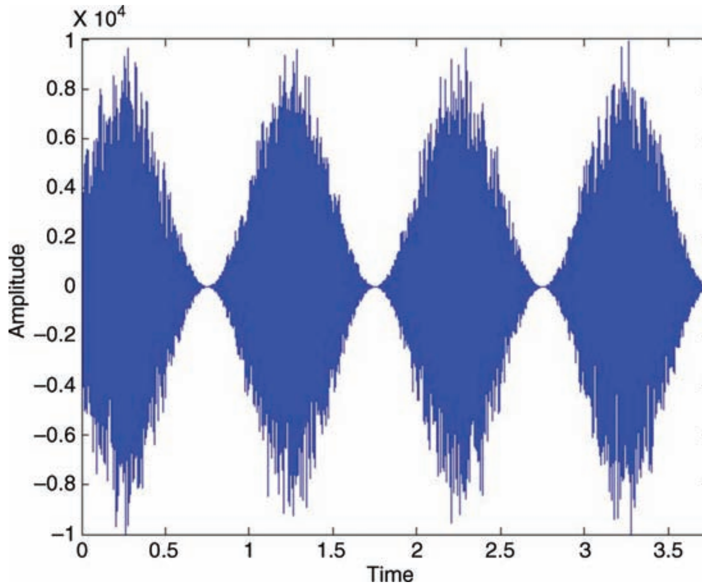


Figure 10.24 Tremolo effect of white noise with modulation rate of 1 Hz

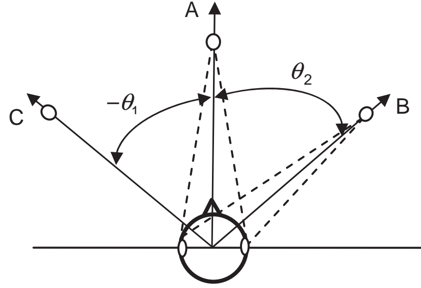


Figure 10.25 The sound source A directly in front of the head and the source B displaced at the θ_2^o azimuth

10.4.4 Spatial Sounds

The audio source location can be determined by its time delay and the intensity differences received by a human listener. Figure 10.25 shows the relative positions of the sound source related to the ears. When the source is at position A (at 0° azimuth), the distances from the sound source to both the ears are equal, thus the sound waveforms arrive at both eardrums at the same time with equal intensity. At position B, the sound source is at θ_2^o azimuth to the right of the listener, so the distances from the sound source and the listener's left and right ears are unequal. The sound waveform arrives at the right ear first.

The sound path differences between the left and right ears are the basis of the interaural time difference (ITD) cues and the interaural intensity difference (IID) cues. The sound source at position A in Figure 10.25 reaches both ears at the same time and same level, and thus does not yield any cues. The sound source at position B or C will yield ITD and IID cues, but only for those frequency components with wavelengths that are smaller than the diameter of the head, that is, for frequencies higher than about 1.5 kHz [19]. For the signal originating from point B, higher frequencies will be attenuated at the left ear because the head will create a shadow effect between the sound source B and the listener's left ear.

Stereo sound can be generated from a mono sound based on this principle. For example, a listener can perceive sound location cues at a concert hall in acoustic space from a given sound source location. The displacement scale can be uniformly divided within 0° to 90° azimuth with ±5 corresponding to ±90° off-center. Figure 10.26 illustrates a typical stereo sound, left channel $y_L(n)$and right channel $y_R(n)$, generated from a single mono sound source $x(n)$. D_L^n and D_R^n are the delays, and g_L^n and g_R^n are the gains, at time n for the stereo channels.

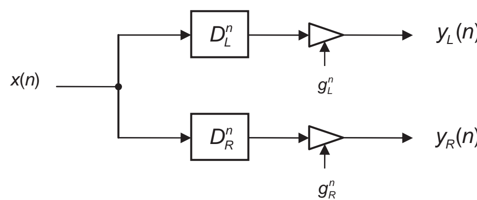


Figure 10.26 Two-channel spatial sound created from mono source with time-varying delays and gains

For example, to perceive the sound source coming from location B as depicted in Figure 10.25, we can set the left-channel delay D_L^n greater than the right-channel D_R^n and set the left-channel gain g_L^n smaller than the right channel gain g_R^n.

The IID-based panning [20] uses the loudness difference for sound source localization. Considering the case shown in Figure 10.25, when the sound source is moving from location C to location B along the front half-circle, the total audio signal power reaching the listener's ears will need to be constant. The most practical and simple way to ensure a constant power is to use the trigonometric identity

$$\sin^2(\theta_s) + \cos^2(\theta_s) = 1, \tag{10.41}$$

where θ_s is the linearly mapped angle from the panning angle θ. When the left-channel gain varies as $\sin(\theta_s)$ and the right-channel gain also varies as $\cos(\theta_s)$, the total power is guaranteed to be constant. To support a wide angle between two sound source locations, θ is first mapped to angle θ_s between 0° and 90° as

$$\theta_s = \left(\frac{\theta - \theta_1}{\theta_2 - \theta_1}\right)90^\circ, \quad \theta_2 \geq \theta \geq \theta_1, \tag{10.42}$$

where θ_1 is the azimuth from the center to the left sound source location, and θ_2 is the azimuth from the center to the right sound source location, as shown in Figure 10.25. The gains for the left and right channels are determined as a function of θ_s as

$$g_L = \sin(\theta_s) \quad \text{and} \quad g_R = \cos(\theta_s). \tag{10.43}$$

Equations (10.42) and (10.43) do not consider the maximum displacement, that is, the maximum IID is 10 dB, but this can be done by adding the limitation and remapping the angles, as explained by Example 10.11.

Example 10.11

This example uses (10.42) and (10.43) to calculate the linear angles and gains for the sound source locations. The maximum displacement is set to 10 dB and the difference between the left-channel gain and right-channel gain must be within this limitation.

By adding the limitation of the maximum IID to 10 dB, we have

$$\theta_s = 17.5^\circ + \left(\frac{\theta - \theta_1}{\theta_2 - \theta_1}\right)55^\circ. \tag{10.44}$$

Thus, assuming 10 dB gain difference at the maximum displacement at (θ_1, θ_2) and letting $-\theta_1 = \theta_2 = 90^\circ$, then approximately

$$\frac{g_L}{g_R} = \frac{\cos(\theta_s)}{\sin(\theta_s)}\bigg|_{\theta=\theta_1} = \frac{1}{\tan(17.5^\circ)} = 3.16 = 10^{10\text{ dB}/20}.$$

This limits the mapped angle range to 55° [17.5°, 72.5°]. Figure 10.27 shows the curves for the constant power gains g_L^n and g_R^n within the IID 10 dB limitation.

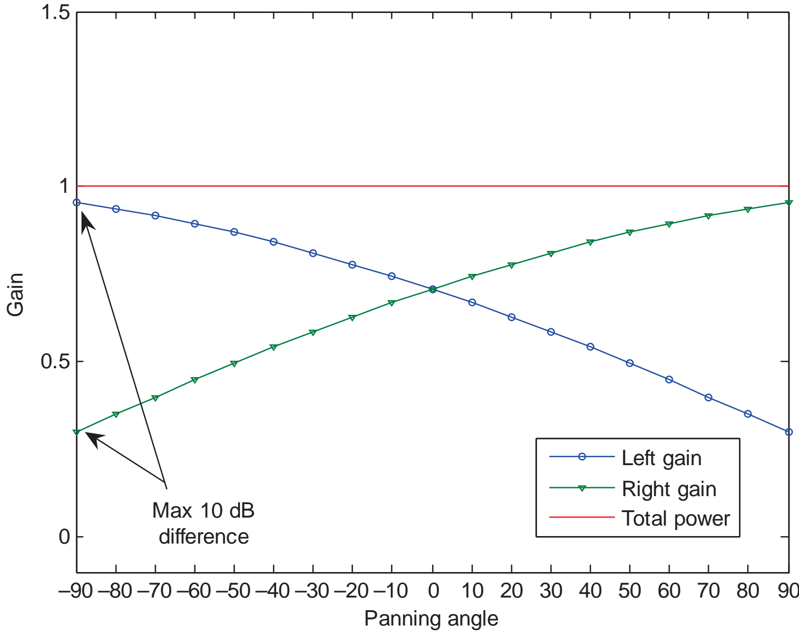


Figure 10.27 Constant power panning for stereo channels

In Figure 10.27, the sound sources are located at $\theta_1 = -90°$ and $\theta_2 = 90°$. The panning angle can be any azimuth along the x axis from $-90°$ to $90°$. If we simulate a sound coming from the left side at $\theta = -40°$, the mapped θ_s will be $32.8°$, thus the left- and right-channel gains will be $g_L = 0.84$ and $g_R = 0.54$, respectively.

10.5 Experiments and Program Examples

This section implements the MDCT and some audio applications introduced in this chapter for experiments using C and TMS320C55xx assembly programs.

10.5.1 MDCT Using Floating-Point C

Table 10.3 lists a portion of the floating-point C program implementing the MDCT and inverse MDCT (IMDCT) used by MP3. The code computes the MDCT of the signal from the given data file and then performs IMDCT on the MDCT coefficients to generate the reconstructed signal. Figure 10.28 shows the flow diagram of functions including windowing, MDCT, and IMDCT, where the quantization operation is not included in this experiment. The programs for the MDCT and IMDCT functions are listed in Tables 10.4 and 10.5, respectively.

In this experiment, three tables are generated in the initialization stage: the sine window table `win`, the `cos_enc` table used by MDCT, and the `cos_dec` table used by IMDCT. The original signal, `input.pcm`, is shown in Figure 10.29(a). The difference between the original input signal and the reconstructed (IMDCT output) signal, `mdctProc.pcm`, is plotted in Figure 10.29(b). The maximum distortion is ± 3. Table 10.6 lists the files used for the experiment.

Table 10.3 Partial C program `floatPoint_mdctTtest.c` for testing MDCT module

```
{
   mdct(pcm_data_in,mdct_enc16,FRAME);   // Perform MDCT of N samples
   // Save latest subband samples to be used in the next MDCT call
   for (j=0; j<M; j++)
   {
      pcm_data_in[j] = pcm_data_in[j+M];
   }
   inv_mdct(mdct_enc16,mdct_proc,FRAME); // Inverse MDCT
   for(j=0; j<M; j++)                    // Overlap and add
   {
      tempOut[j] = mdct_proc[j] + prevblck[j];
      prevblck[j] = mdct_proc[j+M];
   }
}
```

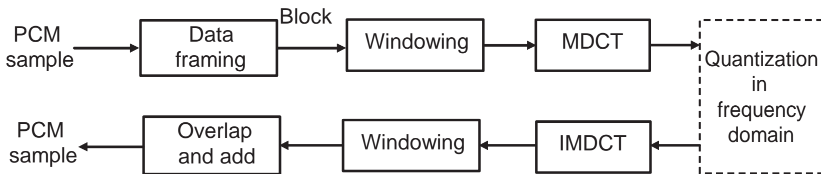


Figure 10.28 Program flow of block processing to compute MDCT and IMDCT

Table 10.4 C code for implementing MDCT

```
// Function: Calculation of the direct MDCT
void mdct(short *in, short *out, short N)
{
    short k,j;
    float acc0;

    for (j=0; j<N/2; j++)        // Calculate N/2 MDCT coefficients
    {
      acc0 = 0.0;
      for (k = 0; k<N; k++)      // Calculate j coefficients
      {
        acc0 += in[k]*(float)win[k] * cos_enc[j][k];
      }
      out[j] = float2short(acc0);// Convert j coefficients to 16-bit word
    }
}
```

Table 10.5 C code for implementing IMDCT

```
// Function: Calculation of the inverse MDCT
void inv_mdct(short *in, short *out, short N)
{
    short j,k;
    float acc0;

    for(j=0;j<N;j++)
    {
        acc0 = 0.0;
        for(k=0;k<N/2;k++)              // Calculate j components
        {
          acc0 += (float)in[k]*cos_dec[((2*j+1+N/2)*(2*k+1))%(4*N)];
        }
        acc0 = acc0*win[j];             // Windowing
        out[j] = float2short(acc0);     // Convert to 16-bit word
    }
}
```

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project. Load and run the experiment.
2. Check the experimental results by comparing the inverse MDCT output file against the original input.
3. Change the frame size, `FRAME`, to 12 and 36 and compare the results.
4. Optimize the experiment to reduce the table size. This can be done by replacing the two-dimensional array `cos_enc[][]` with the one-dimensional array `cos_enc[]` and taking advantage of the periodic property.
5. Three tables, `win[]`, `cos_enc[][]`, and `cos_dec[]`, become larger with increasing frame size. Modify the program such that the MDCT and IMDCT will compute these values using cosine and sine functions at runtime instead of using pre-calculated tables. Verify the implementation using frame sizes of 64, 256, and 512.

Table 10.6 File listing for the experiment Exp10.1

Files	Description
floatPoint_mdctTest.c	Program for testing MDCT experiment
floatPoint_mdct.c	MDCT and IMDCT functions
floatPoint_mdct_init.c	Generate window and coefficient tables
floatPoint_mdct.h	C header file
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
input.pcm	Linear 16-bit PCM data file

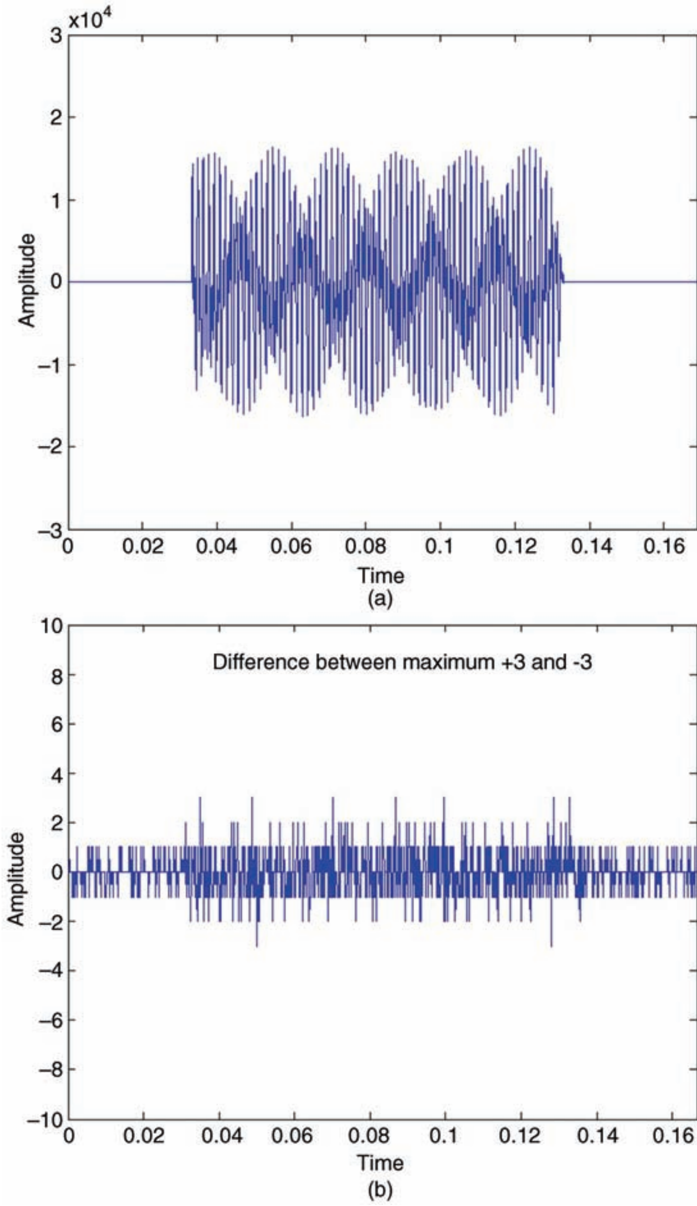


Figure 10.29 The original signal (a) and (b) the amplitude difference between the original and reconstructed signals

Table 10.7 Implementation of MDCT using intrinsics

```
void mdct(short *in, short *out, short N)
{
    short k,j;
    long acc0;
    short temp16;

    for (j = 0; j < (N>>1); j++)
    {
      acc0 = 0;
      for (k = 0; k < N; k++)        // Calculate the j component
      {
         temp16 = mult_r(win[k],(in[k]));
         acc0 = L_mac(acc0,temp16, cos_enc[j][k]);
      }
      out[j] = (short)round(acc0); // Convert to 16-bit word with
                                   // rounding
    }
}
```

10.5.2 *MDCT Using Fixed-Point C and Intrinsics*

This experiment uses fixed-point C to implement the MDCT by replacing all floating-point variables used in the previous floating-point C experiment with integer data types. Intrinsics are also used for some specific functions to improve runtime efficiency. The implementation of the MDCT using fixed-point C with intrinsics is listed in Table 10.7. The main difference between the floating-point and fixed-point implementation is the fixed-point code listed in Table 10.7 using the `mult_r`, `L_mac` and `round` functions to implement the multiplication and accumulation.

Similar to the experiment given in Section 10.5.1, the IMDCT result is compared against the original data. Compared with the 32-bit floating-point implementation shown in Figure 10.29 (b), the distortion from the 16-bit fixed-point implementation shown in Figure 10.30 is larger due to the numerical errors. As illustrated in Figure 10.30, at the active segments, the maximum distortion reaches ±9. The files used for the experiment are listed in Table 10.8.

Table 10.8 File listing for the experiment Exp10.2

Files	Description
`intrinsic_mdctTest.c`	Program for testing MDCT experiment
`intrinsic_mdct.c`	MDCT and IMDCT functions
`intrinsic_mdctInit.c`	Experiment initialization
`intrinsic_mdct.h`	C header file
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`input.pcm`	Linear 16-bit PCM data file

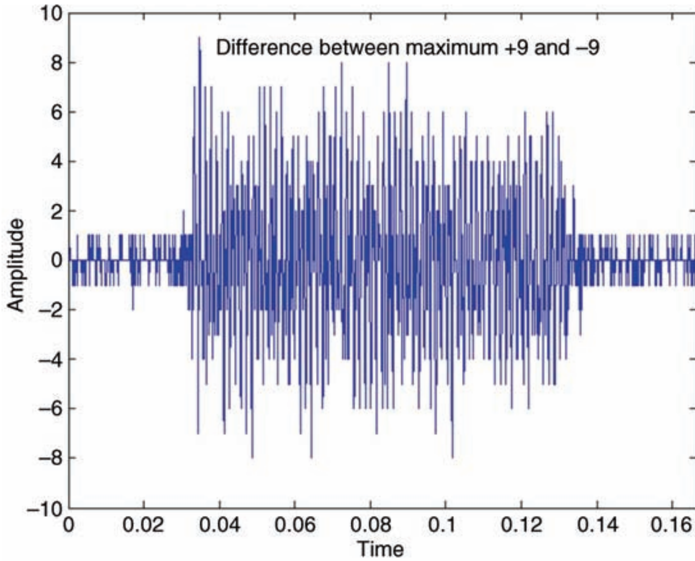


Figure 10.30 Amplitude difference between the original and reconstructed signals

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project. Load and run the experiment.
2. Rerun the experiment using different frame sizes and check the experimental results. Plot the differences between the MDCT/IMDCT experiment input and output signals to show the errors are larger than the floating-point C implementation.
3. Profile the required clock cycles for executing the MDCT function and compare the result to the floating-point implementation obtained in the previous experiment.
4. Use 32 × 32 fixed-point multiplication to replace the 16 × 16 fixed-point multiplication to increase the resolution in the `mdct()` and `inv_mdct()` functions. The input and output data are 16-bit numbers. Check the result and compare it to the result obtained from the floating-point C implementation. Note that a 32 × 32 fixed-point multiplication results in a 64-bit product and a proper scale is required.

10.5.3 Pre-echo Effects

In order to demonstrate the pre-echo effects, this experiment adds the quantization function shown in Figure 10.28 to quantize the MDCT coefficients. The experiment compares two pre-echo effects from 512-point and 64-point MDCT block sizes. The original signal and the pre-echo results are shown in Figure 10.31. In `floatPoint_preEcho.h`, the constant `FRAME` is the MDCT frame size and `NUM_QNT` is the number of levels in log scale used for quantizing the MDCT coefficients. The experiment uses the 512-point MDCT/IMDCT with 50% overlap and the absolute values of the MDCT coefficients are quantized using 64 steps in log scale where the maximum 16-bit value (32 767) corresponds to the highest step. First, the absolute values of the MDCT coefficients are converted to base 10 logarithms. Then, the

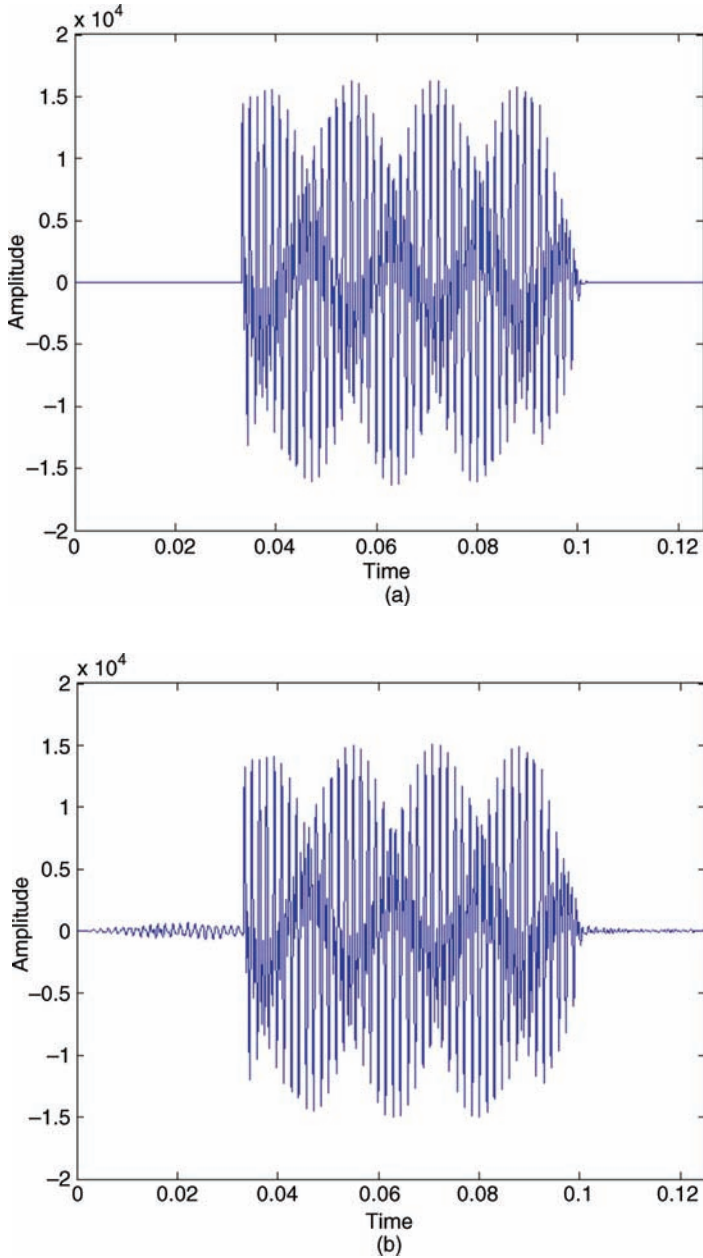


Figure 10.31 The original and reconstructed signals using 64 quantization steps in log scale: (a) original signal; (b) reconstructed signal with pre-echo effect from 512-point MDCT/IMDCT; and (c) reconstructed signal with reduced pre-echo effect from 64-point MDCT/IMDCT

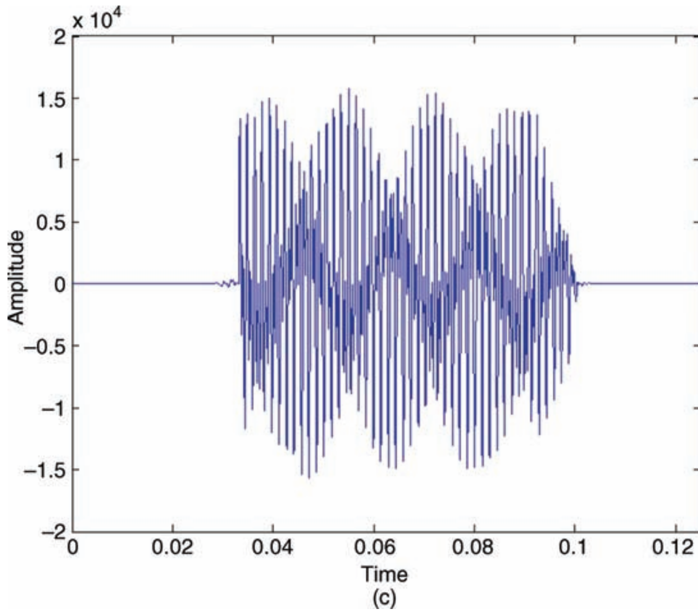


Figure 10.31 (*Continued*)

logarithm coefficients are uniformly quantized with the step size $\log_{10}(32\,768)/64$. The ripples before and after the signal segment from the quantization errors are clearly shown in Figure 10.31(b). For comparison purposes, the 64-point MDCT/IMDCT with 50% overlap is also performed. As shown in Figure 10.31(c), the ripples are reduced by using a shorter MDCT to increase time-domain resolution. The files used for the experiment are listed in Table 10.9.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the experiment to examine the resulting data files.
3. Plot the difference between the original input signal and the reconstructed output signal. Find the range of errors for the 64-point MDCT/IMDCT and 512-point MDCT/IMDCT.

Table 10.9 File listing for the experiment Exp10.3

Files	Description
`floatPoint_preEchoTest.c`	Program for testing pre-echo effects
`floatPoint_preEchoMdct.c`	MDCT and IMDCT functions
`floatPoint_preEchoInit.c`	Generate widow and coefficient tables
`floatPoint_preEchoQnt.c`	Simulate log quantization
`floatPoint_preEcho.h`	C header file
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`dtmf_digit2.pcm`	Data file

Table 10.10 Listing of files used for the experiment Exp10.4

Files		Description
lsfDec.dsw		MP3 decoder workspace
src:	musicout.c	Main file to parse the parameters and access all individual functions
	common.c	Common functions with sampling frequency conversion, bit-rate conversion, file I/O access
	decode.c	Bit-stream decoding, parameter decoding, sample de-quantization, synthesis filters, decoder used functions
	huffman.c	Huffman decoding functions
	ieeefloat.c	Data format conversion
	portableio.c	I/O functions
inc:	common.h	Header file for common.c
	decoder.h	Header file for decoder.h
	huffman.h	Header file for huffman.h
	ieeefloat.h	Header file for ieeefloat.h
	portableio.h	Header file for portableio.h
tables:	1cb0-1cb6, 1th0-1th6	Constant data
	2cb0-2cb6, 2th0-2th6	
	absthr_0-absthr_2	
	alloc_0-alloc_4	
	dewindow, enwindow	
	huffdec	
debug:	lsfDec.exe	Executable files
data:	musicD_44p1_128bps.mp3	Input file for decoder
	mp3_dec.bat	Batch file for running the experiment
	pcm2wav.m	MATLAB® script to convert PCM data to WAV data

4. Repeat step 3 using the 128-point and 256-point MDCT/IMDCT.
5. Convert the floating-point C program to fixed-point C using intrinsics. Rerun the fixed-point C program and compare the results to those obtained using floating-point C. Plot the differences to show the numerical errors using the 64-point and 512-point MDCT/IMDCT.
6. Use higher resolution quantization (e.g., 128 steps) and examine the result to check if it can resolve the pre-echo problem. Specifically, use the 512-point MDCT/IMDCT with 50% overlap and quantize the MDCT coefficients using 128-step log scales. Verify that the amplitude of the ripples (before and after the signal segment) will be smaller than quantizing the MDCT coefficients using 64 steps.

10.5.4 MP3 Decoding Using Floating-Point C

This section presents the MP3 decoding experiment using the floating-point C program. The ISO (International Organization for Standardization) reference source code for the MPEG-1 Layer I, II and III can be downloaded from websites [2] (`dist10.tgz`). These files are listed in Table 10.10, where the file `musicout.c` is the main function that parses

Table 10.11 Decoding information displayed on PC screen when running `mp3_dec.bat`

```
input file = '..\data\musicD_44p1_128bps.mp3'
output file = '..\data\musicD_44p1_128bps.mp3.dec'
the bit stream file ..\data\musicD_44p1_128bps.mp3 is a BINARY file
HDR: s=FFF, id=1, l=3, ep=on, br=9, sf=0, pd=1, pr=0,
     m=1, js=2, c=0, o=0, e=0
alg.=MPEG-1, layer=III, tot bitrate=128, sfrq=44.1
mode=j-stereo, sblim=32, jsbd=8, ch=2
```

the parameters and controls the I/O file. To extract compressed tar archive files, the WinRAR tool can be used.

Extract the tar file `dist10.tgz`, and place the corresponding files under the folders `src`, `inc`, and `tables` as listed in Table 10.10. Use the included workspace file `lsfDec.dsw` under the Microsoft Visual C environment to compile these files to get the executable code under the `Debug` folder. In this experiment, we use Microsoft Visual Studio 2008. After running the MP3 decoder, the program will display the information on the PC console output as listed in Table 10.11.

Procedures of the experiment are listed as follows:

1. The executable program `lsfDec.exe` is located in the directory `..\debug`. Run the batch file `mp3_dec.bat` in the `..\data` folder to generate the experimental results. For this experiment, the MP3 data file, `musicD_44p1_128bps.mp3`, is used as the input file.
2. The output information from the MP3 decoder is listed in Table 10.11. It shows the MP3 audio file, `musicD_44p1_128bps.mp3`, encoded in stereo format using the MPEG-1 Layer 3 algorithm at 44.1 kHz sampling rate with a bit rate of 128 kbps.

```
fid1=fopen('musicD_44p1_128bps.mp3.dec', 'rb', 'ieee-be');
x = fread(fid1,'short');                    % Open to read big-endian file
fclose(fid1);                               % Close file
n=length(x);
y=zeros(n/2, 2);                            % Create output data array
i=1;
for k=1:2:n                                 % Place stereo data to y
    y(i,1)=x(k);
    y(i,2)=x(k+1);
    i = i+1;
end
y = y/32768;
Fs = 44100;                                 % Sampling frequency
nbit=16;                                    % Number of bits/sample
wavwrite(y,Fs,nbit,'decodedMP3Audio'); % Write stereo WAV file
sound(y,Fs,nbit);
```

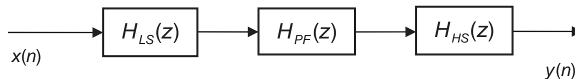


Figure 10.32 Cascade of IIR filters for parametric equalizer

3. The output of the MP3 decoder, `musicD_44p1_128bps.mp3.dec`, is a stereo PCM file and its stereo data samples are arranged in the order of left, right, left, right, ... The 16-bit PCM data is in (MSB, LSB) format. MATLAB® can be used to verify the decoded audio PCM data. The following MATLAB® script converts the PCM format to Intel PCM format:
4. In `decod.c`, the IMDCT function `inv_mdct()` uses double-precision (64-bit) floating-point multiplication. Modify the code to use 32-bit floating-point precision, and check the audio quality by comparing the differences of decoded audio signals. Also, modify the code to use fixed-point multiplication and check the results again.

10.5.5 Real-Time Parametric Equalizer Using eZdsp

This section uses Example 10.7 to calculate the peak and shelf filter coefficients for the real-time parametric equalizer experiment. The sampling frequency is set at 48 kHz for the real-time experiment. Three second-order IIR filters, low-shelf filter $H_{\mathrm{LS}}(z)$, peak filter $H_{\mathrm{PF}}(z)$, and high-shelf filter $H_{\mathrm{HS}}(z)$, are in cascaded form to implement bass, band, and treble boosters as shown in Figure 10.32. The files used for the experiment are listed in Table 10.12

This experiment uses `example10_7.m` to generate the filter coefficients. The magnitude responses of the original and combined cascade filters are shown in Figure 10.33. In the experiment, the filter magnitude responses are scaled to unit gain for fixed-point implementation. Partial code is listed as follows to show the parameter values:

```
Fs = 48000;                                    % Sampling frequency
[az1, bz1] = ShelfFilter(8000,9,1,Fs,'L');     % Fc=8000Hz, Gain=3dB, Q=1
[az2, bz2] = PeakFilter(12000,9,1,Fs);         % Fc=12000Hz, Gain=3dB, Q=1
[az3, bz3] = ShelfFilter(16000,9,1,Fs,'H');    % Fc=16000Hz, Gain=3dB, Q=1
```

Table 10.12 File listing for the experiment Exp10.5

Files	Description
`realtime_parametricTest.c`	Program for testing parametric equalizer
`parametricEQ.c`	Equalizer functions
`asmIIR.asm`	IIR filter using assembly function
`vector.asm`	C5505 interrupt vector for real-time experiment
`asmIIR.h`	C header file
`tistdtypes.h`	Standard type define header file
`dma.h`	Header file for DMA functions
`dmaBuff.h`	Header file for DMA data buffer
`i2s.h`	i2s header file for i2s functions
`Ipva200.inc`	C5505 include file
`myC55xUtil.lib`	BIOS audio library
`c5505.cmd`	Linker command file

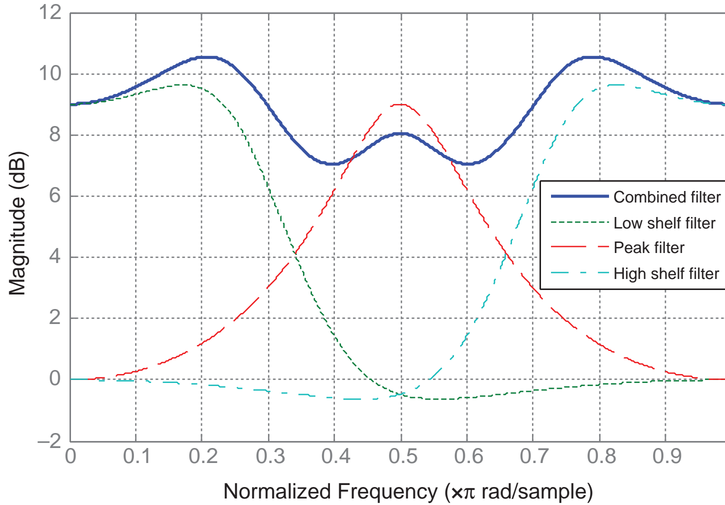


Figure 10.33 Magnitude responses of filters for parametric equalizer

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Connect an audio source such as an MP3 player to the eZdsp audio input jack and connect a headphone or stereo speaker to the eZdsp audio headphone output jack.
3. Load and run the program. Listen to the audio output to compare the real-time equalization effect using the cascaded low-shelf filter, peak filter, and high-shelf filter.
4. Replace the parametric equalizer filters with the following filter combinations and redo the experiment to observe the differences in the equalization effects:
   (a) Low-shelf filter: `Fs` = 48 000 Hz, `Fc` = 6000 Hz, `Gain` = 9 dB, and `Q` = 1.
   High-shelf filter: `Fs` = 48 000 Hz, `Fc` = 16 000 Hz, `Gain` = 9 dB, and `Q` = 1.
   Peak filter: `Fs` = 48 000 Hz, `Fc` = 12 000 Hz, `Gain` = −9 dB, and `Q` = 1.
   (b) Low-shelf filter: `Fs` = 48 000 Hz, `Fc` = 8000 Hz, `Gain` = −9 dB, and `Q` = 1.
   High-shelf filter: `Fs` = 48 000 Hz, `Fc` = 16 000 Hz, `Gain` = −9 dB, and `Q` = 1.
   Peak filter: `Fs` = 48 000 Hz, `Fc` = 12 000 Hz, `Gain` = 9 dB, and `Q` = 1.
   (c) Low-shelf filter: `Fs` = 48 000 Hz, `Fc` = 8000 Hz, `Gain` = 9 dB, and `Q` =1.
   High-shelf filter: `Fs` = 48 000 Hz, `Fc` = 18 000 Hz, `Gain` = 9 dB, and `Q` = 1.
   Peak filter: `Fs` = 48 000 Hz, `Fc` = 12 000 Hz, `Gain` = −9 dB, and `Q` = 1.
   (d) Low-shelf filter: `Fs` = 48 000 Hz, `Fc` = 8000 Hz, `Gain` = 9 dB, and `Q` = 1.
   High-shelf filter: `Fs` = 48 000 Hz, `Fc` = 16 000 Hz, `Gain` = −9 dB, and `Q` = 1.
   Peak filter: `Fs` = 48 000 Hz, `Fc` = 1000 Hz, `Gain` = 9 dB, and `Q` = 0.4.

10.5.6 Flanger Effects

This experiment demonstrates flanger effects using an audio data file with the 8000 Hz sampling rate. The average delay is chosen as 100 samples, the maximum swing is set to 0.5, and the depth is set to 1.0. The files used for the experiment are listed in Table 10.13.

Table 10.13 File listing for the experiment Exp10.6

Files	Description
flangerTest.c	Program for testing flanger effects
flanger.c	Flanger effect processing functions
flanger.h	C header file
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
Soxphone8kHz.pcm	Linear 16-bit PCM audio data file

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project. Load and run the program.
2. Listen to the processed audio file to examine the flanger effects. Compare the magnitude spectrum of the input and output signals to verify that the processed (flanger) audio after the sweeping effect has a wider bandwidth than that of the original audio.
3. Repeat the experiment by changing the swing parameter, `A_maxSwing`, from 0.5 to 1.0. Observe the audio flanger effects changes with a larger swing factor.
4. Change the delay parameter, `delay`, from 100 to 50 and repeat the experiment. Examine if the flanger effects have noticeable changes. Summarize the changes caused by using a shorter delay.
5. Change the depth parameter, `G_depth`, from 1.0 to 0.5, and repeat the experiment. Observe the flanger effects with different values of depth.

10.5.7 Real-Time Flanger Effects Using eZdsp

This section uses the eZdsp for a real-time experiment to demonstrate flanger effects. The audio input can be from an audio player connected to the eZdsp and the audio output is sent to a headphone for listening to the audio effects. This experiment uses mixed C and assembly functions. Updating of the signal buffer requires intensive processing due to the large amount of data to be refreshed as discussed in Chapter 3. The for-loop in C is replaced with the assembly function, `dataMove.asm`, to shift samples in the data buffer. The eZdsp is configured at the 8000 Hz sampling rate. The files used for the experiment are listed in Table 10.14.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project. Load and run the program.
2. Connect an audio player to the eZdsp and use a headphone to examine the real-time flanger effects by listening to the processed audio output.
3. Change the values of the swing, delay, and depth parameters and rebuild the experiment. Compare the differences of the flanger effects to different control parameters.
4. Replace the assembly function `dataMove` in `dataMove.asm` with the C for-loop statement. Build and run the experiment. Observe the problem by listening to the real-time audio playback. The problem occurred because the overall processing load has

Table 10.14 File listing for the experiment Exp10.7

Files	Description
realtime_flangerTest.c	Program for testing flanger effects
realtime_flanger.c	Flanger effect processing functions
dataMove.asm	Assembly function for moving data samples
vector.asm	C5505 interrupt vector table for real-time experiment
realtime_flanger.h	C header file
tistdtypes.h	Standard type define header file
dma.h	Header file for DMA functions
dmaBuff.h	Header file for DMA data buffer
i2s.h	i2s header file for i2s functions
Ipva200.inc	C5505 include file
myC55xUtil.lib	BIOS audio library
c5505.cmd	Linker command file

exceeded the capability of the C5505 processor. Modify the program by replacing floating-point computation with fixed-point computation and using the table lookup method to improve the performance.

10.5.8 Tremolo Effects

This experiment implements the tremolo functions using the C program to demonstrate the audio tremolo effects introduced in Section 10.4.3. The files used for the experiment are listed in Table 10.15.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the program. Listen to the resulting audio file. Also, use the MATLAB® function `spectrogram` to examine the processed audio signal.
3. Change the depth parameter, `TDEPTH`, from 1.0 to 0.5. Repeat the experiment to examine the change in tremolo effects by listening and plotting the spectrogram of the processed signal.
4. Change the modulation rate parameter, `FR`, from 1.0 to 3.0. Repeat the experiment to observe the tremolo effects changed by using a larger modulo parameter.

Table 10.15 File listing for the experiment Exp10.8

Files	Description
tremoloTest.c	Program for testing tremolo effects
tremolo.c	Tremolo effect processing functions
tremolo.h	C header file
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
piano8kHz.pcm	Linear 16-bit PCM audio data file

Table 10.16 File listing for the experiment Exp10.9

Files	Description
realtime_tremoloTest.c	Program for testing tremolo effects
realtime_tremolo.c	Tremolo effect processing functions
vector.asm	C5505 interrupt vector table for real-time experiment
realtime_tremolo.h	C header file
tistdtypes.h	Standard type define header file
dma.h	Header file for DMA functions
dmaBuff.h	Header file for DMA data buffer
i2s.h	i2s header file for i2s functions
Ipva200.inc	C5505 include file
myC55xUtil.lib	BIOS audio library
c5505.cmd	Linker command file

10.5.9 Real-Time Tremolo Effects Using eZdsp

Based on the previous experiment, the eZdsp is used for a real-time experiment to evaluate the tremolo effects. The input audio signal can be from an audio player, digitized at the 8000 Hz sampling rate for the eZdsp experiment. A headphone is used to listen to the audio output for examining audio effects. The files used for the experiment are listed in Table 10.16.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project. Load and run the program.
2. Listen to the output audio using a headphone to evaluate the real-time tremolo effects.
3. Change the values of the control parameters for the experiment as given in the previous tremolo experiment. Examine the audio effects in real time by listening to the output audio.

10.5.10 Spatial Sound Effects

Consider the listening configuration as shown in Figure 10.25, where the sound source moves from the angle θ_1 to θ_2 (in maximum range from $-90°$ to $90°$). Process a mono audio signal (from the sound source) to produce the stereo audio outputs that mimic the sound source moving from the far left side to the far right side. Table 10.17 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load and run the program. Listen to the processed stereo audio file.
3. In this experiment, the sound source is assumed to move from the left ending point ($-90°$) to the right ending point ($90°$) during audio playback. The moving speed can be slowed down by changing the sound source movement from the left ending point to the center in the same audio playback duration. This change can be done by modifying the parameter, `samples`, in the `spacitalTest.c` listed as follows. In the code, the first C statement sets the experiment to move the sound source from left to right in the time duration for all samples in the experiment's input file, while the second statement will only move to the center for the same amount of time.

Table 10.17 File listing for the experiment Exp10.10

Files	Description
spatialTest.c	Program for testing spatial sound effects
spatialSound.c	Spatial effect processing functions
spatialSound.h	C header file
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
audioIn.pcm	Mono linear PCM audio data file

```
// From the left to the right
samples = (unsigned long)((samples>>1)/SAMPLEPOINTS);
// From the left to the center
samples = (unsigned long)((samples)/SAMPLEPOINTS);
```

4. Swap the left and right channels to listen to the stereo sound. Observe if the sound source moves from the left to the right or from the right to the left.

10.5.11 Real-Time Spatial Effects Using eZdsp

This section uses the eZdsp for a real-time experiment to demonstrate the audio spatial effects. The audio input signal is from an audio player. To achieve efficient data-move operations, the C55xx assembly function with single repeat instruction is used. The sampling rate of the eZdsp is set at 8000 Hz. The experimental results can be examined by listening using a headphone. The files used for the experiment are listed in Table 10.18.

Table 10.18 File listing for the experiment Exp10.11

Files	Description
realtime_spatialTest.c	Program for testing spatial sound effects
realtime_spatial.c	Spatial effect processing functions
dataMove.asm	Assembly function for moving data samples
vector.asm	C5505 interrupt vector table for real-time experiment
realtime_spatial.h	C header file
tistdtypes.h	Standard type define header file
dma.h	Header file for DMA functions
dmaBuff.h	Header file for DMA data buffer
i2s.h	i2s header file for i2s functions
Ipva200.inc	C5505 include file
myC55xUtil.lib	BIOS audio library
c5505.cmd	Linker command file

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project. Load and run the program.
2. Listen to the processed audio using a headphone to evaluate the real-time spatial effects.
3. In this experiment, the sound source is moved from the left to the right within 22 seconds. This rate is controlled by parameters `SEGMENTSAMPLES` and `SAMPLEPOINTS`. Use different values of `SEGMENTSAMPLES` to simulate the movement of the sound source from the left to the right at different speeds.

Exercises

10.1. Use MATLAB® to play a 2 kHz tone (masker) at the constant level of 65 dB, and select one of the tones listed in the following table from band numbers 1 to 24. Preset the level of tone chosen to make sure it can be heard. Decrease the level of tone and record the level for which one just cannot hear the tone as the masking threshold. Draw a masking threshold curve similar to Figure 10.3 based on this psychoacoustic test. The bark bands and corresponding center frequency (in Hz) are provided in the following table for reference [11]:

Band no.	Center frequency	Band no.	Center frequency	Band no.	Center frequency	Band no.	Center frequency
1	50	7	700	13	1850	19	4800
2	150	8	840	14	2150	20	5800
3	250	9	1000	15	2500	21	7000
4	350	10	1175	16	2900	22	8500
5	450	11	1370	17	3400	23	10 500
6	570	12	1600	18	4000	24	13 500

10.2. Based on the 36-point MDCT used by MP3, calculate the minimum number of multiplications and additions needed for computing the direct MDCT. Also, compare these numbers to three individual 18-point MDCT blocks.

10.3. If the bit rate of MP3 bit stream is 128 kbps using the constant coding scheme at a sampling rate of 48 kHz, calculate the compression ratio. If the sampling rate is 32 kHz and the bit rate remains the same, calculate the compression ratio. For both cases, assume the input is a stereo signal with left and right channels.

10.4. The downloaded MPEG-1 Layer I, II, and III files in `dist10.tgz` also contain the source code of the MP3 encoder. Using the experiment given in Section 10.5.4 as reference, compile the MP3 encoder program, and encode the decoded linear stereo PCM data from Section 10.5.4 to the MP3 bit stream. Check the resulting MP3 bit stream by playing it back with a commercially available audio player or a PC audio player.

10.5. Based on the transfer function given in (10.14), prove (10.15a). The derivative formula

$$\left(\frac{u}{v}\right)' = \frac{u'v - uv'}{v^2}$$

can be used to prove that

$$\frac{\partial}{\partial \Omega} \left| H(\Omega) \right|^2 \Big|_{\Omega=\Omega_c} = 0.$$

10.6. Given the low-shelf filter transfer function in (10.19), prove (10.20a), (10.20b), and (10.20c). Also prove (10.23a), (10.23b), and (10.23c), given the high-shelf filter in (10.22).

10.7. Replace the input audio file `audioIn.pcm` in Example 10.7 with white noise. Redo Example 10.7 and use MATLAB® to plot the input and output spectra to verify that the shelf and peak filters match the specifications.

10.8. Develop a real-time eZdsp experiment that combines Exp10.7, Exp10.9, and Exp10.11 into one, with a user interface via the CCS command console for selecting which experiment to run.

References

1. Noll, P. and Pan, D. (1997) ISO/MPEG audio coding. *Int. J. High Speed Electron. Syst.*, **8** (1), 69–118.
2. ISO Reference Source Code of MPEG-1 Layer I, II and III, http://www.mp3-tech.org/programmer/sources/dist10.tgz (accessed April 29, 2013).
3. Brandenburg, K. and Popp, H. (2000) An Introduction to MPEG Layer-3, available from http://www.mp3-tech.org/programmer/docs/trev_283-popp.pdf (accessed April 29, 2103).
4. Brandenburg, K. (1999) MP3 and AAC explained. Proceedings of the AES 17th International Conference on High Quality Audio Coding, http://www.aes.org/events/17/papers.cfm (accessed April 29, 2103).
5. Sarkka, S. and Huovilainen, A. (2011) Accurate discretization of analog audio filters with application to parametric equalizer design. *IEEE Trans. Audio, Speech, Lang. Process.*, **19** (8), 2486–2493.
6. Smith, J.O.III (2013) Physical Audio Signal Processing, online book, https://ccrma.stanford.edu/~jos/pasp/ (accessed April 29, 2103).
7. ETSI (2005) Digital Audio Compression (AC-3, Enhanced AC-3) Standard, TS 102 366 V1.1.1, February.
8. ATSC Standard (2001) Digital Audio Compression (AC-3), Revision A, August.
9. Gadd, S. and Lenart, T. (2001) A Hardware Accelerated MP3 Decoder with Bluetooth Streaming Capabilities, MS Thesis, Lund University, available from http://books.google.com/books?id=GoPsPTcnlK0C (accessed April 29, 2103).
10. Raissi, R. (2002) The Theory Behind MP3, December, http://rassol.com/cv/mp3.pdf (accessed April 29, 2103).
11. Painter, T. and Spanias, A. (2000) Perceptual coding of digital audio. *Proc. IEEE*, **88** (4), 415–513.
12. Wang, Y., Yaroslavsky, L., Vilermo, M., and Vaananen, M. (2000) Some peculiar properties of the MDCT. Proceedings of the 5th International Conference on Signal Processing, pp. 61–64.
13. Wang, Y. and Vilermo, M. (2002) The modified discrete cosine transform: its implications for audio coding and error concealment. Proceedings of the AES 22nd International Conference on Virtual, Synthetic and Entertainment Audio, pp. 223–232.
14. Ferreira, A.J. (1998) Spectral Coding and Post-Processing of High Quality Audio, PhD Thesis, University of Porto.
15. Dimkovic, I. (n.d.) Improved ISO AAC Coder, available from http://www.mp3-tech.org/programmer/docs/di042001.pdf (accessed April 29, 2103).

16. ISO/IEC JTC1/SC29/WG11/N7018 (2005) Scalable Lossless Coding, January.
17. Doke, J. (2009) Equalizer GUI for Winsound, http://www.mathworks.com/matlabcentral/fileexchange/10569-equalizer (accessed April 29, 2103).
18. Laroche, J. and Dolson, M. (1999) New phase-vocoder techniques for pitch-shifting, harmonizing and other exotic effects. Proceedings of the 1999 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, pp. 91–94.
19. Begualt, D.R. (1994) *3-D Sound for Virtual Reality and Multimedia*, Academic Press, San Diego, CA.
20. West, J.R. (1998) IID-based Panning Methods, A Research Project, http://www.music.miami.edu/programs/mue/Research/jwest/Chap_3/Chap_3_IID_Based_Panning_Methods.html (accessed April 29, 2103).
21. Li, J. (2002) Embedded audio coding (EAC) with implicit auditory masking. Proceedings of ACM Multimedia'02, pp. 592–601, December, available from http://portal.acm.org/citation.cfm?doid=641126 (accessed April 29, 2103).

11

Introduction to Digital Image Processing

This chapter introduces fundamental digital image processing techniques, with a focus on DSP implementations and some practical applications such as two-dimensional (2-D) image filtering and transforms. We use MATLAB® for digital image visualization, analysis, processing, and algorithm development, and use the C5505 eZdsp for experiments.

11.1 Digital Images and Systems

Digital image processing applications such as digital cameras and camcorders, high-definition televisions (HDTVs), network TVs, smart TVs, portable media players, and home media gateways are now popular features of our daily lives. Image processing has become a skill in high demand for DSP professionals.

Digital image processing, also called 2-D signal processing, has many common principles to the one-dimensional (1-D) DSP introduced in previous chapters, though digital image processing has many specific aspects as well. For example, due to the tremendous amounts of data samples to be processed, most image and video processing applications need efficiently optimized algorithms, require powerful processors with fast input/output throughputs, and are often equipped with hardware accelerators for specific image processing functions. In this section, we will introduce the basic concepts of digital images and systems.

11.1.1 Digital Images

A digital image is a set of data samples mapped onto a 2-D grid of points [1]. Each image sample is called a pixel, which stands for picture element. Similar to a 1-D signal $x(n)$ where n is the time index, an $M \times N$ digital image consists of pixels $x(m, n)$ in 2-D space where $m = 0, 1, \ldots, M-1$ is the column (width) index and $n = 0, 1, \ldots, N-1$ is the row (height) index. Thus, an $M \times N$ image contains MN pixels stored as a 2-D array.

For a black-and-white (B&W) image, each pixel is represented by an 8-bit (byte) number. A color image consists of multiple numbers for each pixel. To display a B&W image, each

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

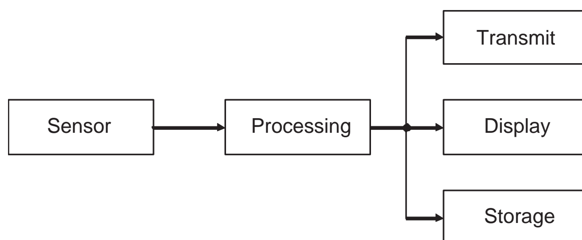
number represents a gray level between 0 and 255, where "0" is a black pixel and "255" corresponds to a white pixel. For a color image, each pixel can be represented using three numbers for the three primary colors, red (R), green (G), and blue (B), often referred to as RGB data. Proper mixing of these three primary colors can create different colors. Most RGB images use 8 bits for each color, and thus need 24-bit data per pixel. Today, color images are widely used in consumer electronics such as photographs, TVs, and computer displays.

The image resolution determines the ability to distinguish spatial details of images. The terms dots per inch and pixels per inch are commonly used to describe image resolutions. In audio applications, a higher sampling rate generally yields a wider bandwidth for better audio fidelity with the trade-off of higher data rate. Likewise, an image with more pixels generally has better (or finer) spatial resolution at the cost of higher data rate.

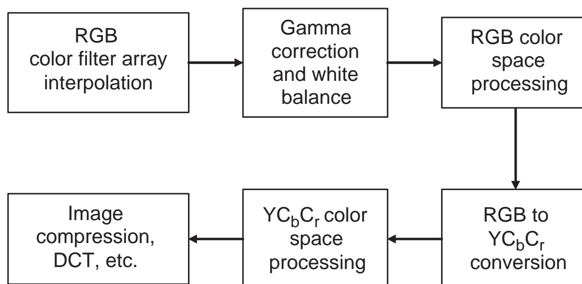
Another frequently used term is pixel dimension, which means the width and height of an image in terms of the numbers of pixels. For example, the National Television System Committee (NTSC) defines North American standard TVs as 720 pixels (width) by 480 pixels (height) so the standard definition TV has a resolution of 720 × 480 pixels.

11.1.2 Digital Image Systems

A simplified digital image system is shown in Figure 11.1(a) . The image captured by the array of sensors is sent to the processing unit and the processed image is presented for display, storing on media storage devices, or transmitting over networks. The image acquisition unit



(a) Block diagram of simplified digital image system.



(b) An example of image processing system.

Figure 11.1 Block diagram of digital image system

contains an array of either charge-coupled device (CCD) or complimentary metal-oxide semiconductor (CMOS) image sensors. These image sensors convert the light or scene into analog electrical signals, digitized by a device called a frame grabber, and stored as digital integer numbers in memory as an M by N array.

There are several different representations (also referred to as color spaces or color models) for color images. The RGB color space is widely used in color image processing. An example shown in Figure 11.1(b); there are other color spaces used for specific applications, so the RGB data may need to be converted to other color spaces. We will introduce color space conversion in the following section.

11.2 Color Spaces

The RGB color space is based on the technical reproduction of color. However, human vision is more sensitive to brightness variations than color changes. This means that humans perceive a similar image with the same brightness even if the color varies slightly. This has led to different representations such as the YC_bC_r color space, which is used by the JPEG (Joint Photographic Experts Group) standard. The ITU BT.601 standard [2] defines the number Y with 220 positive integers, ranging from 16 to 235. The C_b and C_r values range from 16 to 240, centered at 128. The numbers 0 and 255 should not be used for 8-bit image coding because they are used by some manufactures for special synchronization codes. The relationship between the RGB color space and the YC_bC_r color space is defined by the ITU BT.601 standard. Conversion from the RGB space to the YC_bC_r space can be expressed as

$$\begin{bmatrix} \mathrm{Y} \\ \mathrm{C_b} \\ \mathrm{C_r} \end{bmatrix} = \begin{bmatrix} 0.257 & 0.504 & 0.098 \\ -0.148 & -0.291 & 0.439 \\ 0.439 & -0.368 & -0.071 \end{bmatrix} \begin{bmatrix} \mathrm{R} \\ \mathrm{G} \\ \mathrm{B} \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}. \tag{11.1}$$

The YC_bC_r color space is mainly used for computer images, while the YUV color space is generally used for composite color video standards, such as NTSC and PAL (phase alternating line; used in Europe and part of Asia). The conversion from the RGB color space to the YUV color space is defined by the ITU BT.601 standard as

$$\begin{bmatrix} \mathrm{Y} \\ \mathrm{U} \\ \mathrm{V} \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} \mathrm{R} \\ \mathrm{G} \\ \mathrm{B} \end{bmatrix}, \tag{11.2}$$

where Y represents brightness information, and U and V represent color information. The MATLAB® *Image Processing Toolbox* provides several color space conversion functions to transform color images from one color space to another.

Example 11.1

The MATLAB® function `rgb2ycbcr` converts an image from the RGB color space to the YC_bC_r color space. The MATLAB® function `imshow` displays an image using either RGB color or gray level. The function `ycbcr2rgb` converts the YC_bC_r color space to the RGB color space. For example,

```
YCbCr = rgb2ycbcr(RGB);        % RGB to YCbCr conversion
imshow(YCbCr(:,:,1));          % Show Y component of YCbCr data
```

where the function `imshow` displays the luminance of the image `YCbCr(:,:,1)` as a gray-scale image. C_b and C_r are represented by the matrices `YCbCr(:,:,2)` and `YCbCr(:,:,3)`, respectively.

The YUV and YC_bC_r data can be arranged in two different ways: sequential and interleaved. The sequential scheme is backward compatible with B&W TV signals. This method stores all the luminance (Y) data in one continuous memory block, followed by the color difference in the U color block, and finally the V color block. The data memory is arranged as YY . . . YYUU . . . UUVV . . . VV. This arrangement allows the TV decoder to access Y data continuously for B&W TVs. For image processing, the data is usually arranged as interleaved YC_bC_r format for fast access and less memory requirements.

11.3 YC_bC_r Sub-sampled Color Space

The YC_bC_r color space can represent a color image efficiently by sub-sampling the color space. This is because human vision does not perceive chrominance with the same clarity as luminance. Therefore, reduction of chrominance data loses less of the visual contents. Figure 11.2 shows four common YC_bC_r sampling patterns. All four schemes have identical

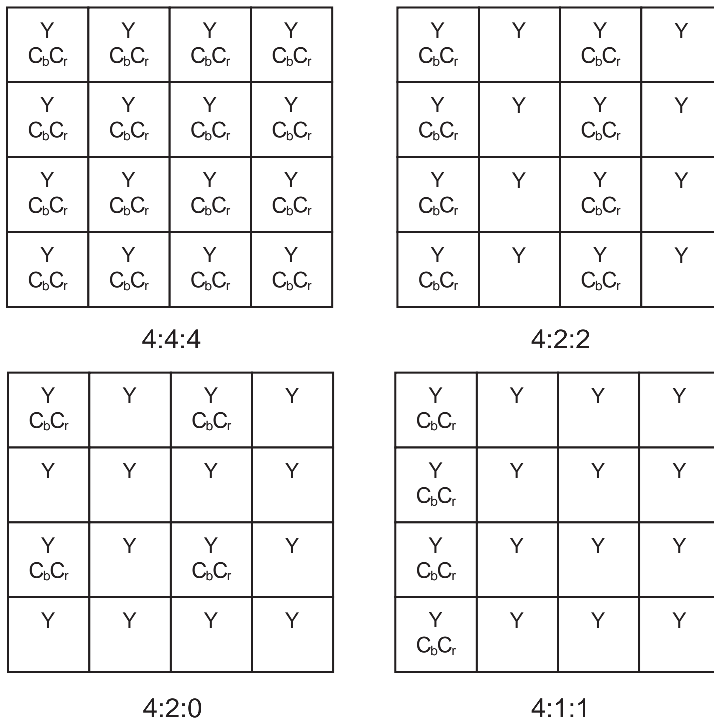


Figure 11.2 Four YC_bC_r sampling patterns

Table 11.1 Number of bits used for four YC_bC_r sub-sampling schemes

720 × 480 pixels	Bits for Y	Bits for C_b	Bits for C_r	Total bits
YC_rC_b4:4:4	720 × 480 × 8	720 × 480 × 8	720 × 480 × 8	8 294 400
YC_rC_b4:2:2	720 × 480 × 8	360 × 480 × 8	360 × 480 × 8	5 529 600
YC_rC_b4:2:0	720 × 480 × 8	360 × 240 × 8	360 × 240 × 8	4 147 200
YC_rC_b4:1:1	720 × 480 × 8	180 × 480 × 8	180 × 480 × 8	4 147 200

4×4 image resolutions. However, the total numbers of bits used to represent these four digital images are different.

In Figure 11.2, the YC_bC_r 4:4:4 pattern does not involve sub-sampling. Every Y (8-bit) sample has its associated C_b (8-bit) and C_r (8-bit) samples. Thus, the YC_bC_r 4:4:4 pattern uses 48 bytes to represent 16 pixels as shown in the figure. This scheme preserves the full color fidelity of chrominance. The YC_bC_r 4:2:2 pattern sub-samples the chrominance by 2 to remove half of the chrominance samples. In this scheme, every four Y samples are only associated with two C_b and C_r samples. Thus, it uses 32 bytes to represent 16 pixels. As shown in Figure 11.2, three-quarters of the chrominance samples are removed by the YC_bC_r 4:2:0 and YC_bC_r 4:1:1 sub-sample schemes, thus only 24 bytes is required to represent 16 pixels. Table 11.1 lists the total number of bits needed for a 720×480 digital image. For MPEG-4 video and JPEG image compression standards, the YC_bC_r 4:2:0 scheme is used to reduce the memory requirement with satisfactory image quality.

11.4 Color Balance and Correction

The images captured by image sensors may not exactly represent the real scene viewed by human eyes. Many factors can cause differences, such as the image sensor's electronic characteristics, which are not the same over the entire color spectrum; lighting variations when the images are captured; reflection of the object under different lighting sources; image acquisition system architectures; as well as display or printing devices. Therefore, color correction techniques including color balance, color adjustment, and gamma correction are necessary in digital cameras, camcorders, and image printers.

11.4.1 Color Balance

The color balance is also called the white balance, which corrects the color bias caused by lighting and other variations in the conditions. For example, a picture taken under indoor incandescent light may appear reddish, while a picture taken at noon on a sunny day may appear bluish. The white-balance algorithm mimics the human visual system to adjust the images. The white balance of the RGB color space can be performed as follows:

$$R_{\mathrm{w}} = R g_{\mathrm{R}}, \tag{11.3a}$$

$$G_{\mathrm{w}} = G g_{\mathrm{G}}, \tag{11.3b}$$

$$B_{\mathrm{w}} = B g_{\mathrm{B}}, \tag{11.3c}$$

where the subscript w indicates the white-balanced RGB color components, and g_{R}, g_{G}, and g_{B} are the gain factors for the red, green, and blue color pixels, respectively. The white-balance

algorithms can be applied in the color domain or spectral domain. The spectral domain-based algorithm requires spectral information of the image sensors and lighting sources, and thus is more accurate but computationally intensive. On the other hand, the RGB color space-based white-balance algorithm is simple, easy to implement, and less expensive. In order to obtain accurate gain factors, the RGB data must contain a rich spectrum of colors. For example, the calculation of gain factors may not be correct if the RGB color space consists of the red color only.

Example 11.2

The MATLAB® function `imread` can read JPEG, TIF, GIF, BMP, and PNG images. The `imread` function will return a three-dimensional (3-D) array for color images and a 2-D array for gray-scale images. For most images, the image array is converted to the 8-bit RGB color space from these files. In real applications, the white-balance gains are usually normalized to G (green) pixels. The following MATLAB® script computes the white-balance gains from a RGB image:

```
R = sum(sum(RGB(:,:,1)));        % Compute sum of R
G = sum(sum(RGB(:,:,2)));        % Compute sum of G
B = sum(sum(RGB(:,:,3)));        % Compute sum of B
gr = G/R;                        % Normalized gain factor for R
gb = G/B;                        % Normalized gain factor for B
Rw=RGB(:,:,1)*gr;                % Apply gain factor to R
Gw=RGB(:,:,2);                   % G has gain factor of 1
Bw=RGB(:,:,3)*gb;                % Apply gain factor to B
```

If the image appears reddish, the sum of the red color elements `R` will be larger than the sum of the blue color elements `B`. Thus the normalized red color gain `gr = G/R` is smaller than the blue color gain `gb = G/B`. Applying the gains `gr` and `gb` to the `RGB` image, the `Rw` values will be reduced while the `Bw` values will be increased, since the gain factor `gr` is smaller than `gb`. Thus, the resulting image will be less reddish.

11.4.2 Color Correction

The RGB color in a digital camera or a camcorder may not be the same as perceived by human eyes. Chromatic correction (also called color correction or color saturation correction) can compensate the color offset to make digital images close to what humans see. The color correction can be achieved by applying a 3×3 matrix to the white-balanced RGB color space, as

$$\begin{bmatrix} R_c \\ G_c \\ B_c \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \begin{bmatrix} R_w \\ G_w \\ B_w \end{bmatrix}, \tag{11.4}$$

where the coefficients of the 3×3 color correction matrix are obtained as

$$\min \left\{ \sum_{n=1}^{3} \sum_{m=1}^{3} [c_{nm} x_{\mathrm{w}}(m,n) - x_{\mathrm{ref}}(m,n)]^2 \right\} \quad \text{for } n \neq m, \tag{11.5}$$

$$c_{nm} = 1 \qquad \text{for } n = m, \tag{11.6}$$

where $x_{\mathrm{w}}(m,n)$ are the pixels of the white-balanced image and $x_{\mathrm{ref}}(m,n)$ are the pixels of the reference image with known values, such as the standard color chart. Therefore, the color correction is achieved by determining the coefficients c_{nm} to form the optimal 3×3 color correction matrix that minimizes the mean-square error between $x_{\mathrm{w}}(m,n)$ and $x_{\mathrm{ref}}(m,n)$. In order to preserve the white balance and luminance, the diagonal elements of the matrix are normalized to one, as defined in (11.6).

11.4.3 Gamma Correction

When TVs or computer monitors fail to produce linear output values from linear input values, gamma correction can be used to compensate the nonlinearity of the display devices to achieve a linear response. Figure 11.3 shows the gamma correction curve vs. the nonlinearity of the display devices, where the x axis represents the input image data values (R, G, or B) to be displayed and the y axis represents the output image data values of the display devices. In order to display the image with linear output as a diagonal line, the gamma correction (convex line) multiplies the input RGB data by the corresponding gamma values to compensate for the display device's nonlinear (concave) characteristics. The gamma values for TV systems are specified by ITU BT.624-4 [3]. Computer monitors use gamma values ranging from 1.80 to 2.20.

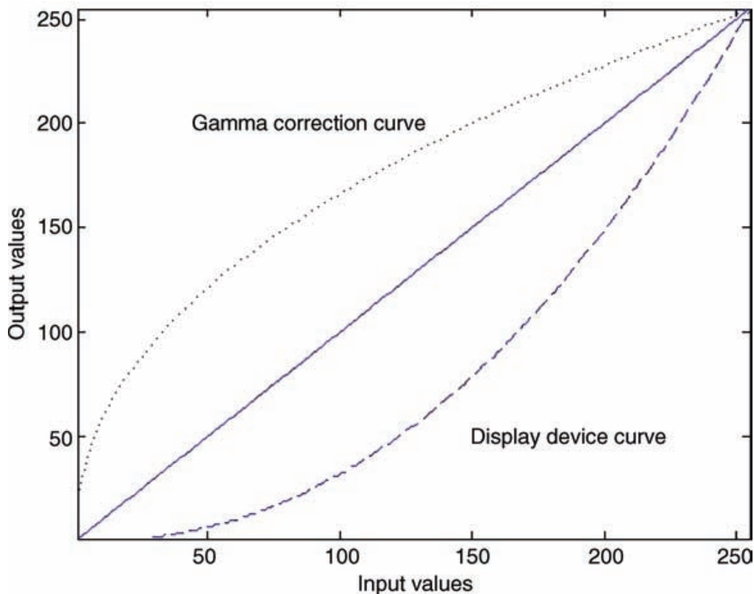


Figure 11.3 Gamma correction curve for 8-bit display devices

The operations of gamma correction are defined as

$$R_\gamma = gR_c^{1/\gamma}, \tag{11.7a}$$

$$G_\gamma = gG_c^{1/\gamma}, \tag{11.7b}$$

$$B_\gamma = gB_c^{1/\gamma}, \tag{11.7c}$$

where g is the correction gain factor, γ is the gamma value, and R_c, G_c, and B_c are the input values of the color-corrected RGB color elements. The output values of the gamma-corrected RGB color elements are denoted by R_γ, G_γ, and B_γ.

Example 11.3

This example uses (11.7) to generate the gamma correction curve for $\gamma = 2.20$; the computed values are stored in a table with 256 entries, and the gamma correction is applied to the given bit-mapped (BMP) image ($\gamma = 1.00$). The BMP file format will be briefly introduced in Section 11.9. The original image is shown in Figure 11.4(a) . The image after gamma correction is shown in Figure 11.4(b) using the MATLAB® script `example11_3.m`.

11.5 Histogram Equalization

A histogram displays important characteristics of digital images, which represents the distribution of the image's pixel values by counting the numbers of pixels that have the same values. For an 8-bit image, the histogram has $L = 256$ entries. The first entry is the total number of pixels in the image with value "0," the second entry is the total number of pixels with value "1," and so on. Therefore, the sum of all values in the histogram equals the total number of pixels in the $M \times N$ image as

$$MN = \sum_{l=0}^{L-1} h_l, \tag{11.8}$$

where L is the total number of entries in the histogram, h_l is the individual histogram value, and MN equals the total number of pixels in the image.

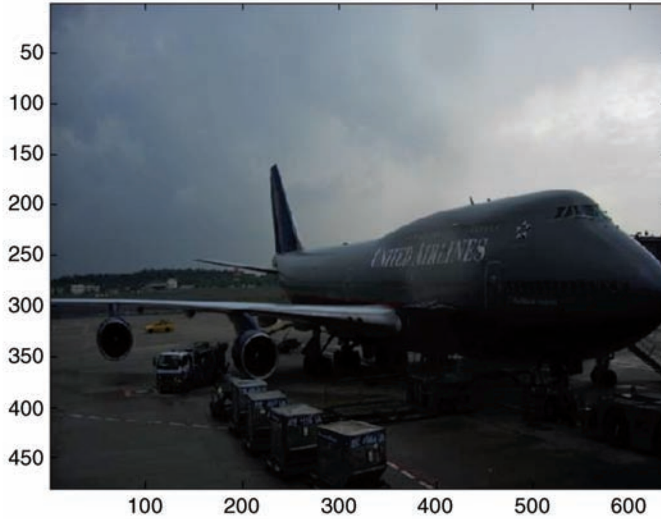
Since the image $x(m, n)$ may contain millions of pixels, we can compute the mean m_x and the variance σ_x^2 of the image from the histogram as follows:

$$m_x = \frac{1}{MN}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1} x(m,n) = \frac{1}{MN}\sum_{l=0}^{L-1} lh_l \tag{11.9}$$

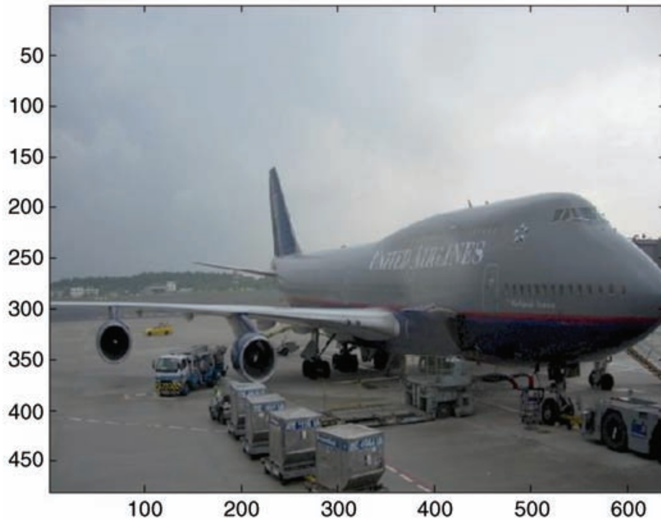
$$\sigma_x^2 = \frac{1}{MN}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}[x(m,n) - m_x]^2. \tag{11.10}$$

MATLAB® provides the functions `mean2` and `std2` to compute the mean and standard derivation, respectively, of the pixel values.

Proper brightness and contrast can make the image easier to view. The brightness is the overall luminance level of the image, and the contrast is the difference in brightness. The brightness is associated with the exposure when the picture is taken using a digital camera. Brightness adjustment can improve the viewing ability of the darker or brighter areas by



(a) The original image (γ = 1.00)



(b) Gamma-corrected image (γ = 2.20)

Figure 11.4 Gamma-corrected image vs. the original image

increasing or decreasing the luminance value of every pixel. Since the modification of brightness changes the value of every pixel in the image, the entire image will become brighter or darker. Changing the brightness of the image will not affect the contrast of the image. Contrast can be adjusted by varying the luminance value of each pixel. The adjustment of brightness and contrast may result in saturation if the adjusted pixel values reach the limit.

Histogram equalization uses the monotonic nonlinear mapping process to redistribute the intensity values of the image such that the resulting image has a uniform distribution of

intensities. Histogram equalization may not work well for some images because the redistribution does not use prior knowledge of the image. However, histogram equalization works well for fine detail in the darker regions of B&W images. MATLAB® provides the function `histeq` to enhance contrast using histogram equalization. The histogram equalization given in MATLAB® script `example11_4.m` consists of the following three steps:

1. Compute the histogram of the image:

```
for i=1:height
   for j=1:width
      index = uint8((Y(i,j))+1);
      hist(index) = hist(index) + 1;
   end
end
```

2. Normalize the histogram:

```
len = 256;
eqFactor = 255 / (width * height);
sum = 0;
for i=1:len
   sum = sum + hist(i);
   eqValue = sum * eqFactor;
   eqTable(i) = uint8(eqValue);
end
```

3. Equalize the image using the normalized histogram:

```
for i=1:height
   for j=1:width
      index = uint8((Y(i,j))+1);
      newY(i,j) = eqTable(index);
   end
end
```

Example 11.4

This example computes the histogram of a given image and equalizes the image based on the histogram using the MATLAB® script `example11_4.m`. As shown in Figure 11.5(a), the buildings in the original image are very dark because the picture was taken in the evening without sufficient light. The top plot of Figure 11.5(c) shows that most of the pixels are concentrated in the lower portion of the histogram. The viewing quality of this image can be improved by increasing the contrast using histogram equalization. The equalized image in Figure 11.5(b) shows that most of the dark scenes have been clearly revealed. The bottom plot of Figure 11.5(c) shows the histogram of the equalized image with more evenly distributed pixel values.

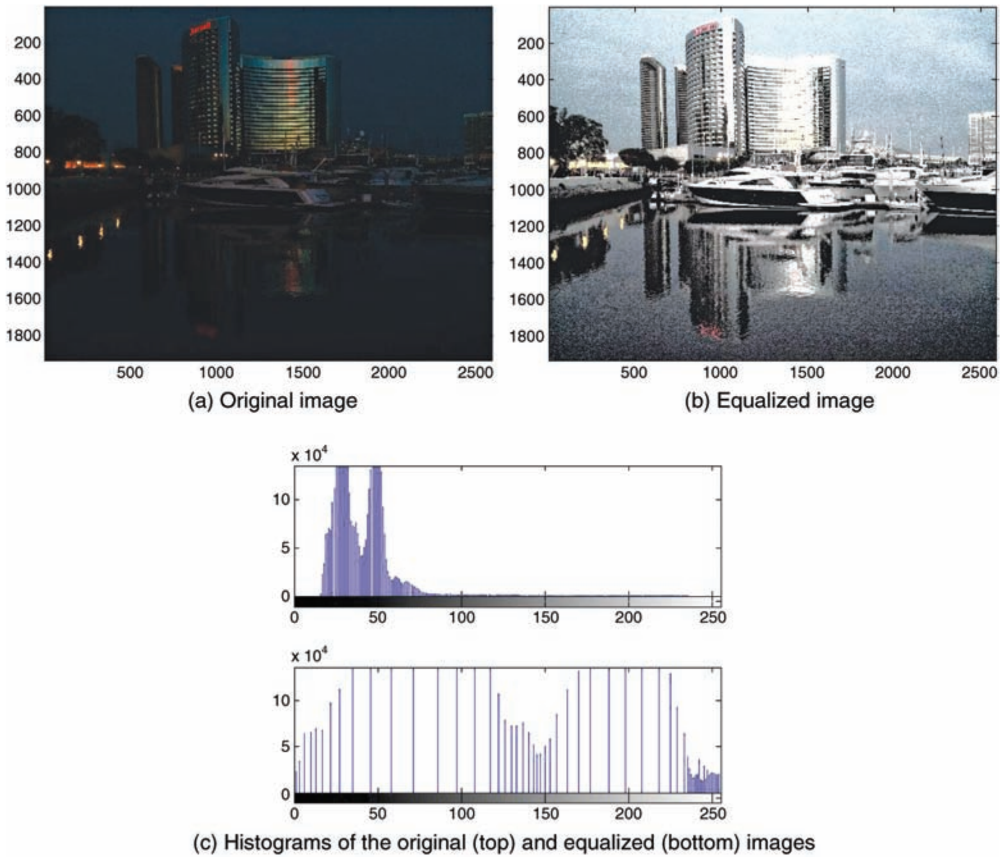


Figure 11.5 Histogram equalization of dark image.

The histogram equalization used by Example 11.4 is an automated process that replaces the trial-and-error adjustment of the brightness. It is an effective technique to enhance the contrast of the image. Since histogram equalization applies an approximated histogram uniformly, some undesired effects may occur. These problems may be overcome by using adaptive histogram equalization, which divides the image into regions and equalizes each smaller region individually instead of the entire image. To reduce the artifacts caused by the boundaries of these small regions, an additional smoothing process should be considered. MATLAB® provides the function `adapthisteq` for performing adaptive histogram equalization. An example of histogram application is the gray stretch of fingerprint images, which will be introduced in Section 11.9.6.

11.6 Image Filtering

Many video and image applications use 2-D filters to accomplish the required tasks. If the input to a 2-D system is the impulse (delta) function $\delta(m, n)$ at the origin, the output is the system's impulse response $h(m, n)$. When the system's response remains the same regardless

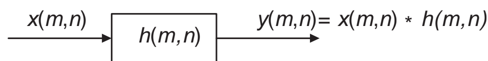


Figure 11.6 A linear space-invariant 2-D system

of the position of the impulse function, the system is defined as a linear space-invariant (LSI) system. The input/output relationship of an LSI system can be described by its impulse response as illustrated in Figure 11.6, where $*$ denotes linear 2-D convolution.

In general, computation of 2-D filtering requires four nested loops, and thus it becomes a very computationally intensive operation. If the pixels in the horizontal direction (row) and vertical direction (column) are uncorrelated, we can implement the 2-D filtering as two 1-D filtering operations that filter the row direction first and then the column direction. Similar to the 1-D FIR filtering introduced in Chapter 3, image filtering to produce the output image $y(m,n)$ can be realized as the 2-D convolution of the $I \times J$ filter impulse response $h(i,j)$ with the $M \times N$ image $x(m,n)$ expressed as

$$y(m,n) = \sum_{j=J-1}^{0} \sum_{i=I-1}^{0} h(i,j)x(m-i+1, n-j+1), \tag{11.11}$$

for $m = 0, 1, \ldots, M-1$ and $n = 0, 1, \ldots, N-1$. The filter coefficients are denoted by $h(i,j)$ for $i = I-1, I-2, \ldots, 0$ and $j = J-1, J-2, \ldots, 0$. Note that the filter coefficient indices are in decreasing order since the linear convolution is realized by folding, shifting, multiplication, and summation operations as described in Section 3.1.5.

For image filtering, the 2-D filter coefficients are folded twice (up–down and left–right). Figure 11.7 shows an example of using a 3×3 filter ($I = J = 3$) to obtain the filtered image pixel at $y(4, 4)$, which can be expressed in detail as

x(0,0)	x(0,1)	x(0,2)	x(0,3)	x(0,4)	x(0,5)	...
x(1,0)	x(1,1)	x(1,2)	x(1,3)	x(1,4)	x(1,5)	...
x(2,0)	x(2,1)	x(2,2)	x(2,3)	x(2,4)	x(2,5)	...
x(3,0)	x(3,1)	x(3,2)	x(3,3)	x(3,4)	x(3,5)	...
x(4,0)	x(4,1)	x(4,2)	x(4,3)	x(4,4)	x(4,5)	...
x(5,0)	x(5,1)	x(5,2)	x(5,3)	x(5,4)	x(5,5)	...
...	...	...	...	...	...	...

h(2,2)	h(2,1)	h(2,0)
h(1,2)	h(1,1)	h(1,0)
h(0,2)	h(0,1)	h(0,0)

Figure 11.7 An example of 2-D convolution using a 3×3 filter

$$
\begin{aligned}
y(4,4) = {} & h(2,2)x(3,3) + h(2,1)x(3,4) + h(2,0)x(3,5) \\
& + h(1,2)x(4,3) + h(1,1)x(4,4) + h(1,0)x(4,5) \\
& + h(0,2)x(5,3) + h(0,1)x(5,4) + h(0,0)x(5,5).
\end{aligned}
$$

This clearly shows that the output pixel value is the sum of products obtained from multiplying the filter coefficients by the overlaid image pixels.

Image filtering can be used for noise reduction, edge enhancement, sharpening, blurring, and many other applications. Linear filters are widely used in image processing and editing to create many special effects for digital photos. Common image noises are Gaussian noise introduced by image sensors, impulse noise caused by sudden intensity change in white values (may also result from bad image sensors), and B&W high-frequency noise, also known as salt and pepper noise. An edge is a boundary between two image regions indicated by a relative discontinuity of intensity. Edge detection is one of the most critical functions in image recognition and computer vision. MATLAB® provides the function `edge` to find edges in the intensity image.

Linear smoothing (or lowpass) filters are effective for reducing high-frequency noise, but at the cost of blurring edges. The linear filters usually use weighted coefficients such that the sum of the filter coefficients equals one in order to maintain the same image intensity. When the sum is greater than one, the resulting image will be brighter; otherwise, it will be darker.

MATLAB® provides several functions for designing 2-D filters. For example, the function `fwind2` uses a 2-D window specification to design 2-D FIR filters based on the desired frequency response. Some commonly used 2-D filters are the 3×3 kernels summarized as follows:

(a) Delta filter: $h(i,j) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$.

(b) Lowpass filter: $h(i,j) = \dfrac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$.

(c) Highpass filter: $h(i,j) = \dfrac{1}{6}\begin{bmatrix} -1 & -4 & -1 \\ -4 & 26 & -4 \\ -1 & -4 & -1 \end{bmatrix}$.

(d) Sobel filter: $h(i,j) = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ or $h(i,j) = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$.

(e) Prewitt filter: $h(i,j) = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$ or $h(i,j) = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$.

(f) Laplacian filter: $h(i,j) = \begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix}$.

(g) Emboss filter: $h(i,j) = \begin{bmatrix} -4 & -4 & 0 \\ -4 & 1 & 4 \\ 0 & 4 & 4 \end{bmatrix}$.

(h) Engrave filter: $h(i,j) = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$.

Most 2-D filters such as the delta, lowpass, highpass, and Laplacian filters are symmetric in both the horizontal and vertical directions. In general, lowpass filters use different values of coefficients; however, the 3×3 smoothing filter given in (b) is a special case of a lowpass filter that has the same value for every coefficient. A highpass filter may be obtained by subtracting a lowpass filter kernel from the delta filter kernel. A highpass filter can be used for image sharpening to enhance edges. Edges represent sudden changes of local intensity levels in an image. Edge detection is a key step for image analysis and recovery, and it is often used to recognize objects in machine vision applications. The Sobel filter is widely used as an edge filter to enhance the horizontal image edges. Similarly, the Prewitt filter can enhance the image's vertical edges. The operations of these two filters apply only to one direction at a time. The horizontal Sobel kernel can be rotated by 90° to obtain the vertical kernel and the vertical Prewitt kernel can be rotated by 90° to get the horizontal kernel. Examples of horizontal Sobel filtering and vertical Prewitt filtering are illustrated in the MATLAB® script `example11_5.m`. The Laplacian kernel can be used to check the number of zero crossings.

Example 11.5

The MATLAB® *Image Processing Toolbox* provides the function `filter2` to implement 2-D filters as follows:

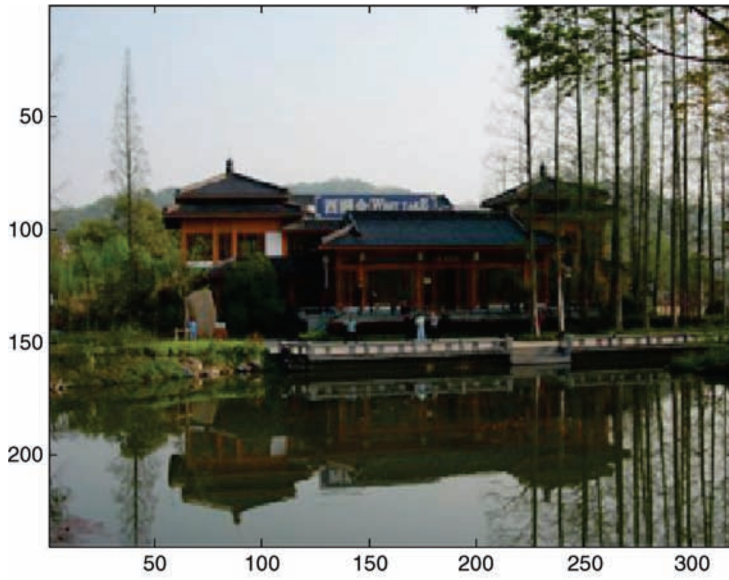
```
R = filter2(coeff, RGB(:,:,1));
G = filter2(coeff, RGB(:,:,2));
B = filter2(coeff, RGB(:,:,3));
```

The parameter `coeff` is the 2-D filter kernel, `RGB` is the input image matrix, and `R`, `G`, and `B` are the arrays of filtered output of the R, G, and B components, respectively. MATLAB® also provides the built-in function `imfilter` for image filtering using the syntax

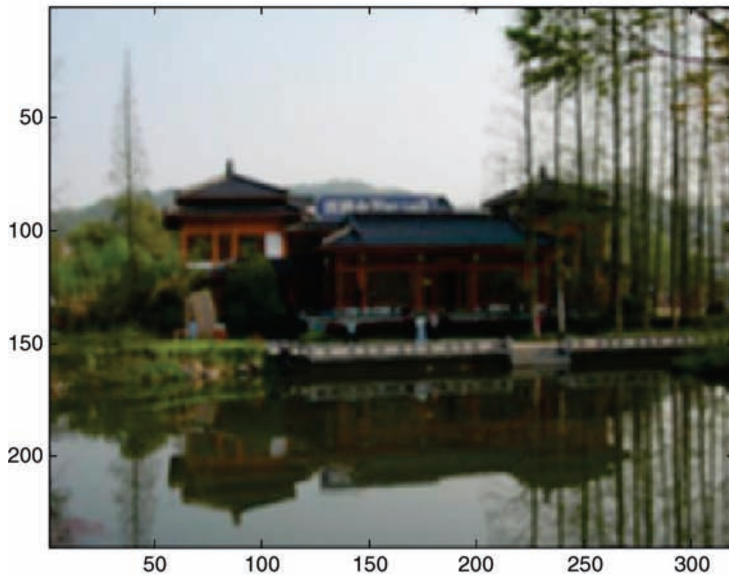
```
newRGB = imfilter(RGB, coeff);
```

The 2-D filtering results by different filter kernels are shown in Figure 11.8, where the delta kernel result represents the original picture for comparison to the different 2-D filtering effects.

When implementing a 2-D filter using fixed-point hardware for image processing, the overflow problem must be handled carefully. Since 2-D filtering is computationally intensive, it often uses low-order filter kernels such as 3×3 or 5×5 for many real-time image and video applications.

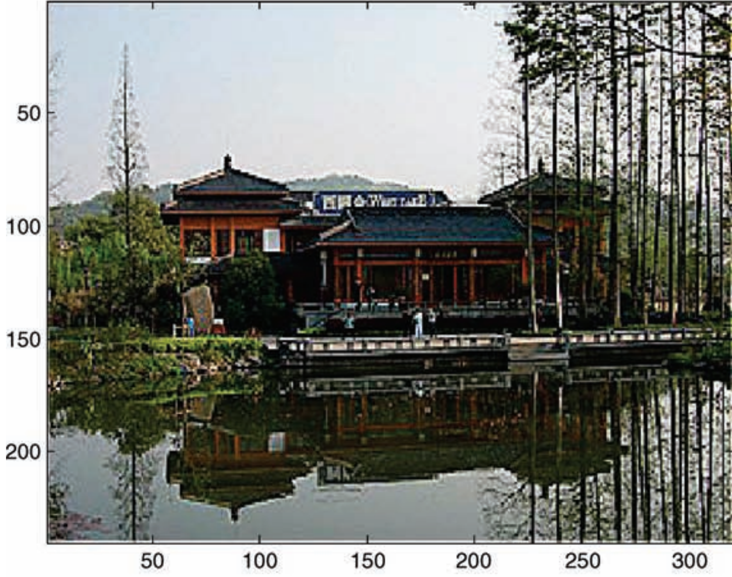


(a) Delta kernel (the original image).

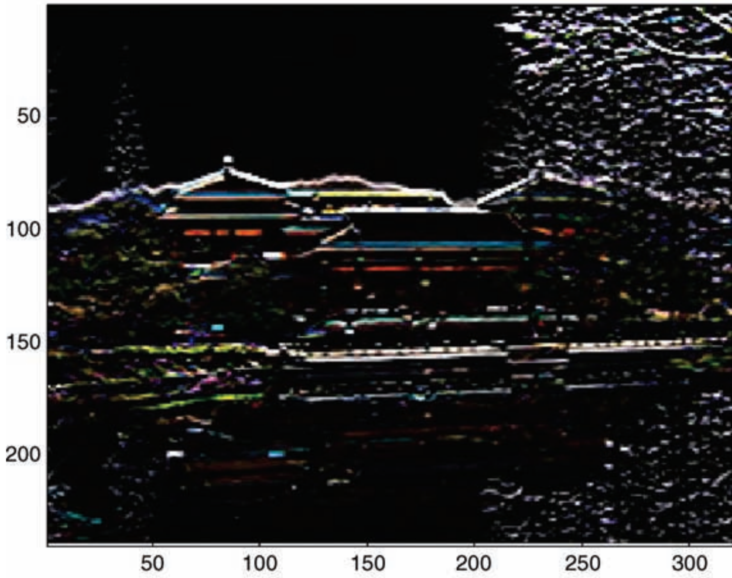


(b) Lowpass filter (image blurrring).

Figure 11.8 Results and effects of image filtering using different 3×3 filter kernels.

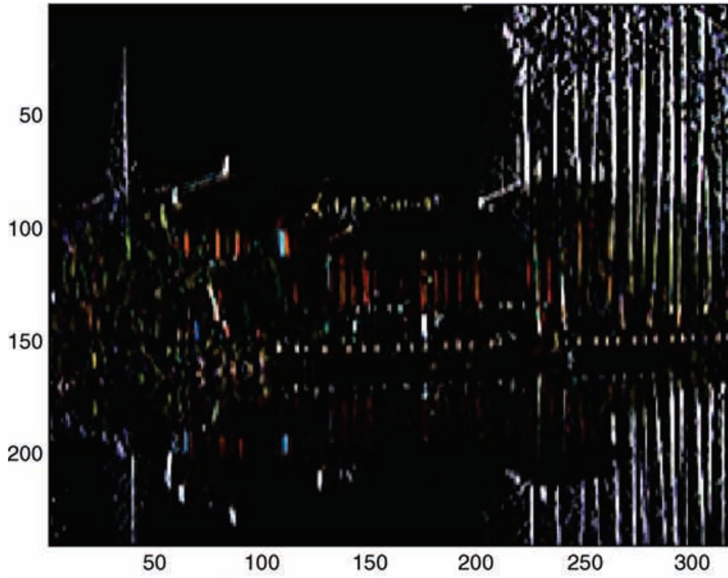


(c) Highpass filter (image sharpening).

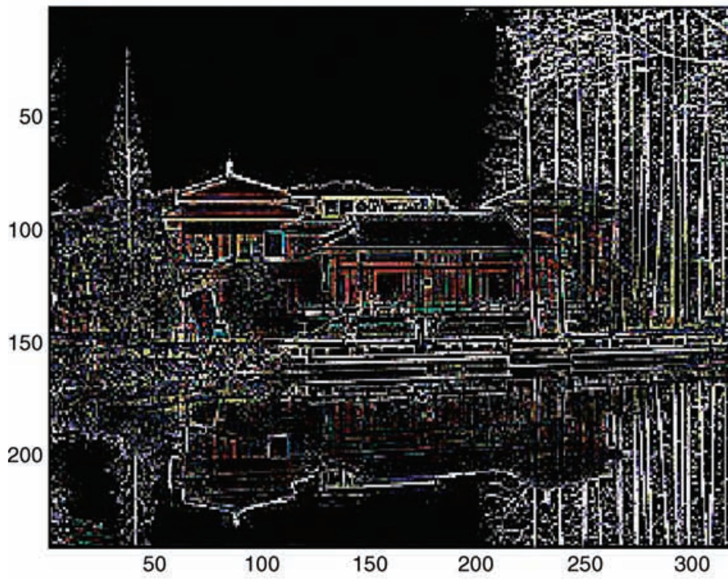


(d) Sobel filter (for horizontal edge detection).

Figure 11.8 (*Continued*)

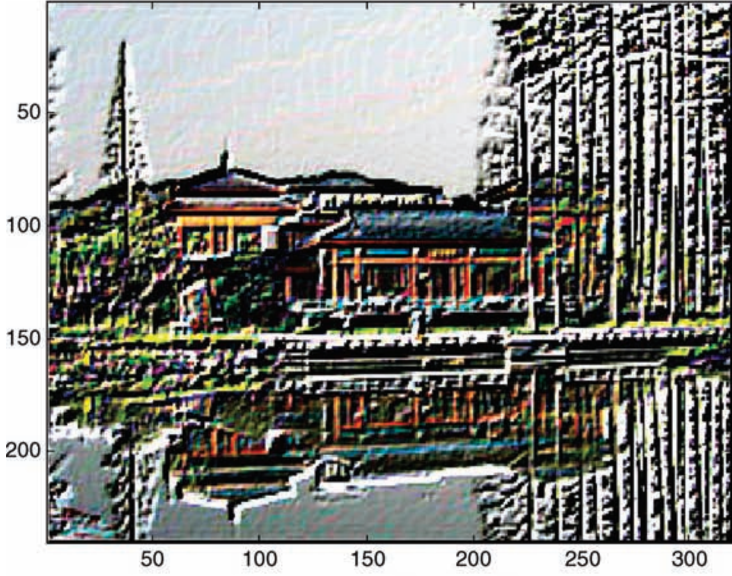


(e) Prewitt filter (for vertical edge detection).

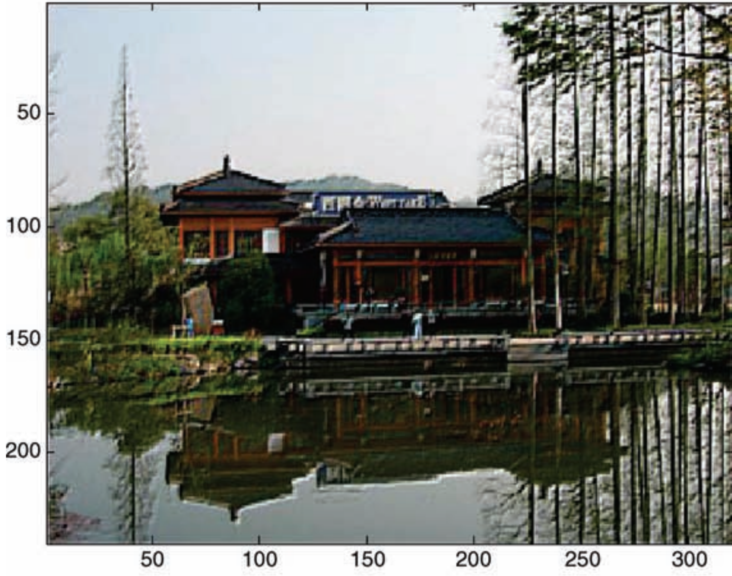


(f) Laplacian filter (edge sharpening).

Figure 11.8 (*Continued*)



(g) Emboss filter (3-D shadow effect).



(h) Engrave filter (engrave effect).

Figure 11.8 (*Continued*)

11.7 Fast Convolution

A 720×480 RGB image consists of $720 \times 480 \times 3$ data samples. As shown in (11.11), image filtering uses nested loops for each pixel of every filter coefficient in the kernel, which requires intensive computation if the filter kernel is large. For a large 2-D filter, the more efficient approach is to use fast convolution that utilizes the computationally efficient 2-D FFT and IFFT. This convolution is considerably efficient for filtering images using a large-size filter kernel because the spatial-domain convolution becomes multiplication in the frequency domain. For 2-D fast convolution, we can first apply the filter in the horizontal direction, and then in the vertical direction.

Given an $M \times N$ image, the 2-D DFT is defined as

$$X(k,l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x(m,n) \mathrm{e}^{-\mathrm{j}(2\pi/N)lm} \, \mathrm{e}^{-\mathrm{j}(2\pi/M)kn}, \tag{11.12}$$

where m and n are spatial indices for the pixel $x(m,n)$, k and l are frequency indices for $k = 0$, 1, . . . , $M - 1$ and $l = 0$, 1, . . . , $N - 1$. The 2-D IDFT is defined as

$$x(m,n) = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} X(k,l) \mathrm{e}^{\mathrm{j}(2\pi/N)lm} \, \mathrm{e}^{\mathrm{j}(2\pi/M)kn}. \tag{11.13}$$

MATLAB® provides the 2-D FFT function `fft2` and the IFFT function `ifft2`.

The 2-D fast convolution can be realized using the following procedures:

1. Pad zeros to both the filter coefficient matrix and image data matrix so they have the same power-of-2 number for width and height in order to use the efficient 2-D FFT and IFFT algorithms.
2. Compute the 2-D FFT of both the image and coefficient matrices.
3. Multiply the frequency-domain coefficient matrix by the image matrix using the dot product, starting in the row direction and then in the column direction.
4. Apply the 2-D IFFT function to obtain the filtered image.

Example 11.6

The example of using MATLAB® functions `fft2` and `ifft2` on the R component of an RGB image for fast convolution is listed as follows:

```
fft2R = fft2(double(RGB(:,:,1)));                % 2-D FFT of R component
[imHeight imWidth] = size(fft2R);
fft2Filt = fft2(coeff, imHeight, imWidth);       % 2-D FFT of filter kernel
fft2FiltR = fft2Filt .* fft2R;                   % 2-D fast convolution
newRGB(:,:,1) = uint8(ifft2(fft2FiltR));         % 2-D inverse FFT
```

In the code, `fft2FiltR` is the 2-D fast convolution result of the red component of the RGB image. The filter coefficients are zero padded and the frequency-domain matrix is

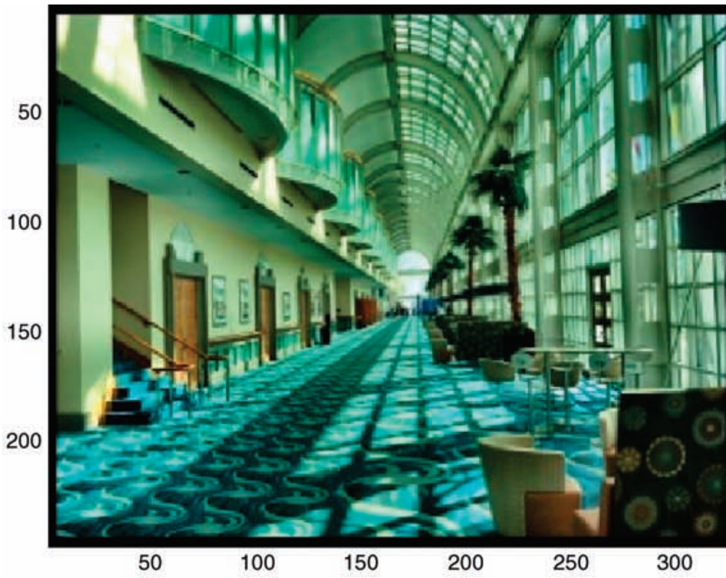
obtained using the `fft2` function. The symbol "`.*`" in the fourth line of the MATLAB® script is the dot-product operator. The complete MATLAB® script `example11_6.m` is available in the companion software package. Figure 11.9 shows the image filtering results using the following three 9×9 kernels as the edge, motion, and Gaussian filters:

$$\text{(a) Edge filter: } h(i,j) = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 & 31 & 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}.$$

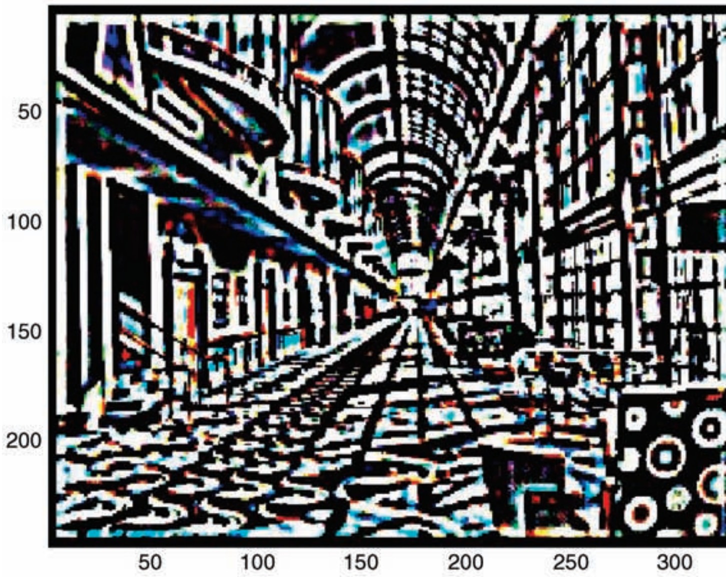
$$\text{(b) Motion filter: } h(i,j) = 1/9 \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$\text{(c) Gaussian filter: } h(i,j) = 1/256 \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 4 & 6 & 4 & 1 & 0 & 0 \\ 0 & 0 & 4 & 16 & 24 & 16 & 4 & 0 & 0 \\ 0 & 0 & 6 & 24 & 36 & 24 & 16 & 0 & 0 \\ 0 & 0 & 4 & 16 & 24 & 16 & 4 & 0 & 0 \\ 0 & 0 & 1 & 4 & 6 & 4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The example of fast convolution given in Example 11.6 shows that image filtering can be efficiently performed in the frequency domain for large-size 2-D filters using the fast convolution method. The 2-D convolution of an $N \times N$ image with a $J \times J$ filter requires approximately N^2J^2 multiplications (N^4 if $J = N$). Fast convolution, including the 2-D FFT, IFFT, and frequency-domain multiplication, requires approximately $NJ + 2\log_2(NJ)$ multiplications (or $N^2 + 4\log_2(N)$ if $J = N$). Therefore, the computational requirements for fast convolution are much less than those needed in the spatial domain using linear 2-D convolution.

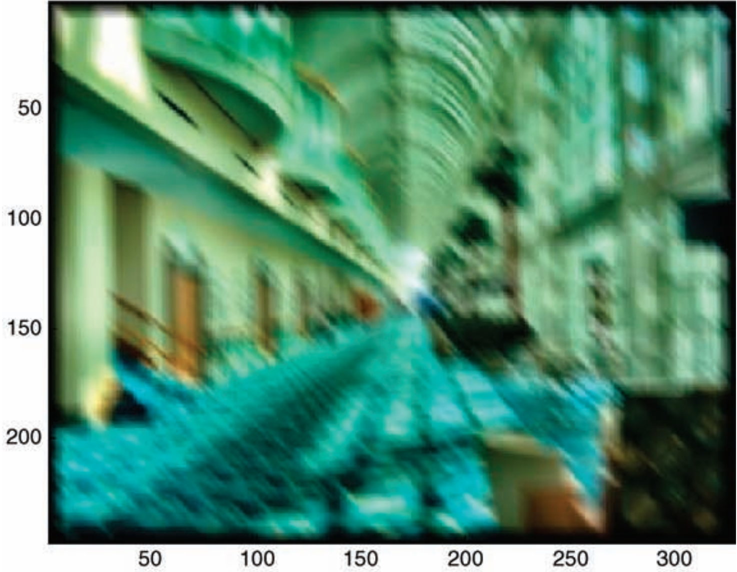


(a) Original image.

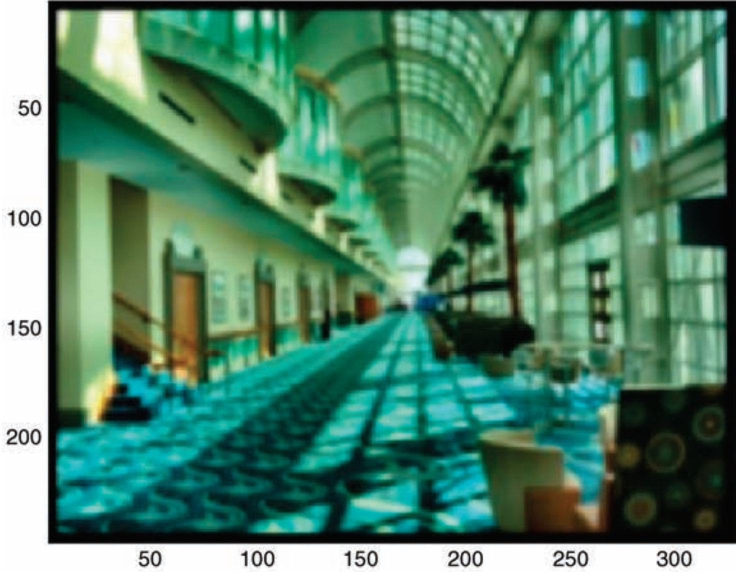


(b) Output image from the edge filter.

Figure 11.9 Results of 2-D filtering using fast convolution.



(c) Output image from the motion filter.



(d) Output image from the Gaussian filter.

Figure 11.9 (*Continued*)

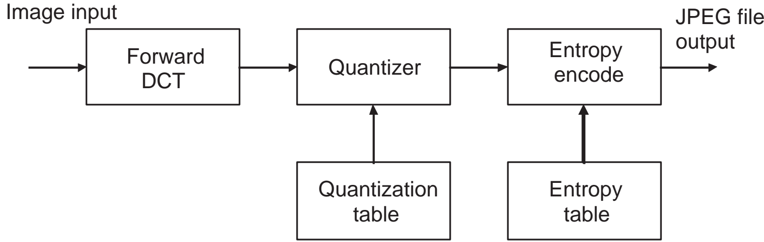


Figure 11.10 Block diagram of baseline JPEG encoder

11.8 Practical Applications

Digital image processing has many practical applications. For example, JPEG is the most widely used, still image compression standard based on the 8×8 discrete cosine transform (DCT). JPEG can achieve up to 10:1 compression ratio with satisfactory quality. However, for higher compression ratios, JPEG images may show block artifacts. This problem can be solved by the newer compressing standard, JPEG2000, which uses wavelet technology to compress images at higher compression ratio without the block artifacts. This section introduces these two image compression techniques.

11.8.1 DCT and JPEG

The JPEG standard, defined in ITU-T Recommendation T.81 [4], is widely used in printing, digital cameras, video editing, security, and medical imaging applications. Figure 11.10 shows the basic block diagram of the JPEG encoder.

The commonly used 1-D type II DCT (forward DCT) and type III DCT (inverse DCT) are adapted by the JPEG standard. The baseline JPEG is a sequential DCT-based operation with image pixels grouped into 8×8 blocks. The JPEG encoder processes one block of pixels at a time from left to right, starting from the first row of blocks, repeating for the second row, and so on from the top to the bottom. Each block is transformed by the forward DCT to obtain 64 DCT coefficients. The first coefficient is called the DC coefficient, and the rest of the 63 coefficients are AC coefficients. These 64 DCT coefficients are quantized using one of 64 corresponding values from the quantization table specified by the T.81 standard.

Instead of quantizing the current DC coefficient directly, JPEG encodes the difference between the previous and current DC coefficients, which has a smaller value. The rest of the 63 AC coefficients are quantized based on their actual values. These 64 quantized DCT coefficients are arranged as the 1-D zigzag sequence shown in Figure 11.11. The quantized and reordered coefficients are then passed to the entropy coder for further compression. There are two entropy coding techniques defined by the JPEG standard: Huffman encoding and arithmetic encoding. Each encoding method uses its own encoding table specified by T.81.

11.8.2 Two-Dimensional DCT

Two-dimensional DCT and inverse DCT (IDCT) are used in many digital image and video compression techniques including the JPEG standard [5]. The 2-D DCT and IDCT of an $N \times N$ image are defined as

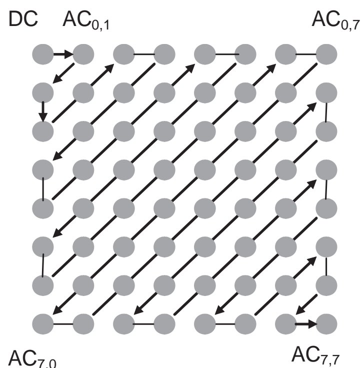


Figure 11.11 Ordering of DCT coefficients in zigzag fashion

$$X(k,l) = \frac{2}{N} C(k)C(l) \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x(m,n) \cos\left[\frac{(2n+1)l\pi}{2N}\right] \cos\left[\frac{(2m+1)k\pi}{2N}\right], \tag{11.14}$$

$$x(m,n) = \frac{2}{N} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} C(k)C(l)X(k,l) \cos\left[\frac{(2n+1)l\pi}{2N}\right] \cos\left[\frac{(2m+1)k\pi}{2N}\right], \tag{11.15}$$

where $x(m,n)$ is the image pixel and $X(k,l)$ is the corresponding DCT coefficient, and

$$C(k) = C(l) = \begin{cases} \sqrt{2}/2, & \text{if } k = l = 0 \\ 1, & \text{otherwise.} \end{cases} \tag{11.16}$$

Most image compression algorithms use $N = 8$, and the 8×8 DCT and IDCT specified by T.81 can be obtained by substituting $N = 8$ in (11.14) and (11.15).

The 2-D DCT and IDCT can be implemented using two separate 1-D operations: one for the horizontal direction (column by column) and the other for the vertical direction (row by row). Using efficient fast 1-D DCT and IDCT algorithms, the computation can be reduced dramatically. The 1-D, eight-point DCT and IDCT can be implemented as

$$X(k) = \frac{1}{2} C(k) \sum_{m=0}^{7} x(m) \cos\left[\frac{(2m+1)k\pi}{16}\right], \tag{11.17}$$

$$x(m) = \frac{1}{2} \sum_{k=0}^{7} C(k)X(k) \cos\left[\frac{(2m+1)k\pi}{16}\right], \tag{11.18}$$

where

$$C(k) = \begin{cases} \sqrt{2}/2, & \text{if } k = 0 \\ 1, & \text{otherwise.} \end{cases}$$

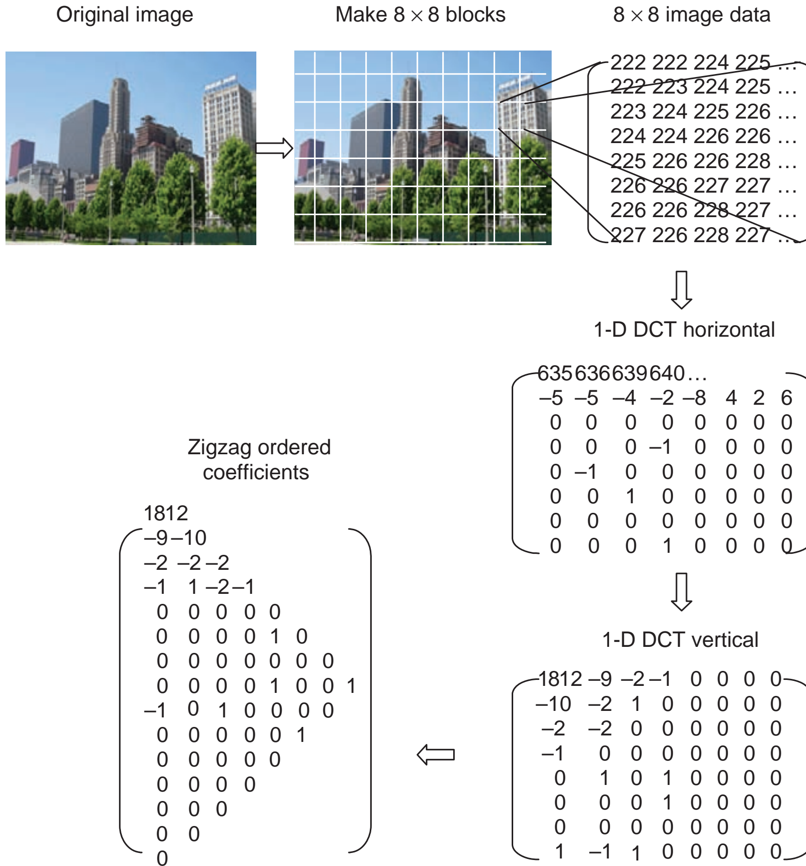


Figure 11.12 DCT block transform and reorder in JPEG image coding process

The JPEG compression algorithm applies the 1-D DCT to a block of 8 × 8 (64) image pixels at a time. The resulting DCT coefficients are reordered in a zigzag fashion, and then quantized by the entropy encoder. This process with intermediate results is illustrated in Figure 11.12.

Example 11.7

MATLAB® provides the functions `dct` (`idct`) and `dct2` (`idct2`) for the 1-D and 2-D DCT (IDCT), respectively. The use of 1-D functions given in (11.17) and (11.18) is identical to the use of the 2-D functions defined in (11.14) and (11.15). The MATLAB® implementation using the 1-D DCT and IDCT is given in the MATLAB® script `example11_7.m`. First, the image file is read using the `imread` function and the RGB color space is converted to YC_bC_r color space using `rgb2ycbcr`. The image is then divided into 8 × 8 blocks and the 1-D DCT applied to these 8 × 8 image blocks, first to the row and then to the column as follows:

```
for n=1:8:imHeight                          % Each block is 8 pixels in height
  for m=1:8:imwidth                         % Each block is 8 pixels in width
    for i=0:7
      for j=0:7
        mbY(i+1,j+1) = Y(n+i,m+j,1);        % Form 8x8 blocks
      end
    end
    mbY = dct(double(mbY));                 % Perform 1-D DCT horizontally
    mbY = dct(double(mbY'));                % Perform 1-D DCT vertically
  end
end
```

The IDCT is operated on every 8×8 block of the DCT coefficients to transfer them back to the YC_bC_r color space. Finally, the image in the YC_bC_r color space is converted to the RGB color space using the MATLAB® function `ycbcr2rgb`. The difference (error) between the original and reconstructed images is small, which verifies that the DCT/IDCT operations are correct.

For JPEG compression, once the zigzag ordered sequence of DCT coefficients is formed, these DCT coefficients will be quantized and encoded. The encoding process stops when the remaining coefficients of the sequence are all zeros.

11.8.3 Fingerprint

Fingerprint is a reliable, unique, and consistent identification technique. In recent years, automated biometric identification based on fingerprint has been integrated into many applications. A fingerprint image consists of many dark friction ridge lines and with white shallow valleys between the ridges. The fingerprint identification algorithm compares the locations of the ridge endings and directions of the ridge lines to determine if two patterns are identical. As illustrated in Figure 11.13, a typical fingerprint biometric application uses several image processing algorithms such as image filtering, transform, and edge enhancement. The fingerprint identification process following image edge thinning is omitted in the figure because it is beyond the scope of this book.

The fingerprint image is segmented into regions such that the pixels in each segment will contain the same aspects of the object or area in the image. This segmentation prepares the image for edge enhancement and detection. The basic task of fingerprint segmentation is to decide the foreground image obtained by contacting the finger with the sensors. Histogram

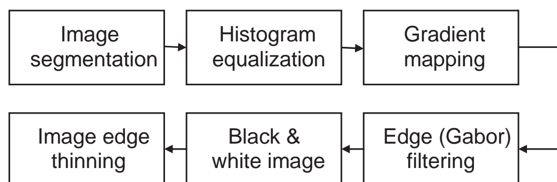


Figure 11.13 Flow diagram of image processing algorithms used for fingerprint application

equalization (or gray stretch) is used to increase the fingerprint's contrast by making the histogram of the fingerprint image more evenly distributed. Gradient mapping is then applied to the equalized image. The directional process uses the gradient algorithm to create a map based on the image gradient magnitude and searches for the local gradient magnitude maxima along the gradient direction. Next, the image is processed using a filter, such as the Laplacian filter introduced in Section 11.6, to extract edge information from the image. The filtered image is passed to the decision block, which uses a threshold to set the pixel to one of two binary values, either black or white. The threshold value is a critical element of this binary decision block. An incorrect threshold may produce faulty results from noisy images or miss subtle information due to image fragmentation. Finally, the enhanced image is passed to the thinning process to remove all unwanted spurious points near the edge of the pixel line and create a sharp single-line image. This single-line image enables efficient fingerprint feature identification, comparison, and recognition. Figure 11.14 shows the series of resulting images from each of the processes. The complete program will be introduced by the experiment given in Section 11.9.6.

11.8.4 Discrete Wavelet Transform

Discrete wavelet transforms (DWTs) are used for many image processing applications such as image compression, noise reduction, and fingerprint identification. In general, a wavelet function is a small wave, which must be oscillatory in certain ways to discriminate between different frequencies. In comparison to the DFT based on sinusoidal functions, the analysis function of the wavelet transform can be chosen from several basis functions called mother wavelets. Popular mother wavelets include the Daubechies, Coiflets, Haar, Morlet, and Symlets wavelets, which are satisfied by certain requirements. Since the theory of wavelet technology is beyond the scope of this book, we will only briefly present the 2-D DWT to be used for the experiments.

The main concept of discrete wavelet technology, or wavelet transform, is subband filtering (or decomposition) using filterbanks. The DWT uses multi-level filterbanks with specific wavelet filters for the decomposition (analysis) and reconstruction (synthesis) of signals. Wavelet decomposition can be accomplished by a pair of highpass and lowpass filters as shown in Figure 11.15. This process can be repeated to form multiple cascaded stages (or levels). One important difference between the wavelet-based and DCT-based techniques is the region used for image processing. The DCT-based method requires image data to be segmented in blocks, such as the most popular block of 8×8 pixels, while the wavelet technique is free of this requirement. Thus, wavelet-based image processing avoids the block artifacts resulting from the DCT-based compression. The JPEG standard uses the DCT-based algorithm while the JPEG2000 standard uses the wavelet-based transform. Figure 11.15 shows the block diagram of the discrete wavelet decomposition and reconstruction. In the figure, notations $h(\cdot)$, $g(\cdot)$, $W_{\mathrm{LL}}(m,n)$, $W_{\mathrm{LH}}(m,n)$, $W_{\mathrm{HL}}(m,n)$, $W_{\mathrm{HH}}(m,n)$, and so on will be defined later in this section.

As shown in Figure 11.16, wavelet technology decomposes the signal into high- and low-frequency bands. As depicted in Figure 11.16(b), the first stage (level 1) 2-D subband decomposition contains four decomposition results (sub-sampled images) labeled as LL, LH, HL, and HH, where the letters L and H denote lowpass and highpass filtered image bands, respectively. In wavelet analysis, the lowpass filter output

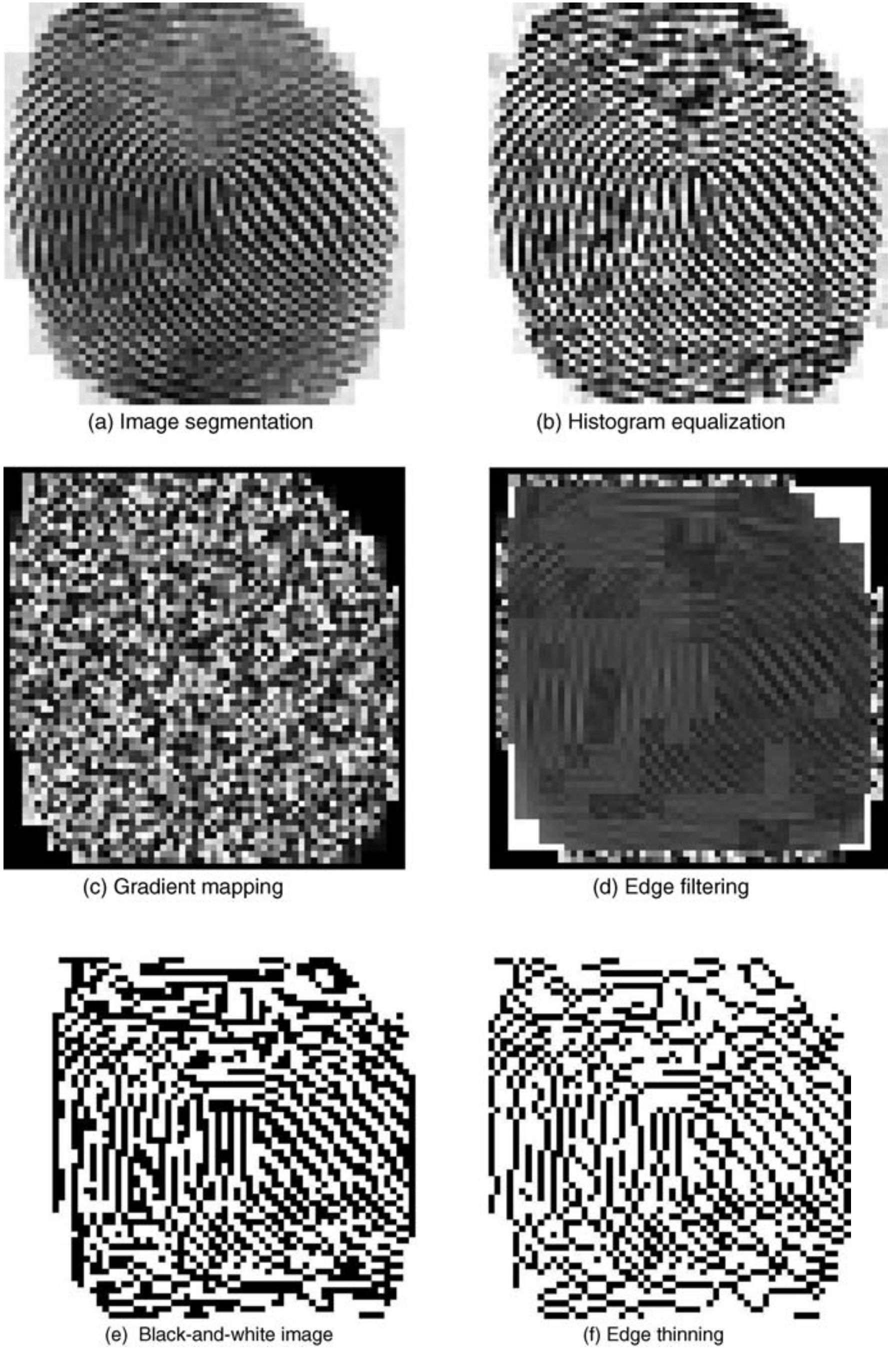


Figure 11.14 Results of fingerprint processing for edge enhancement

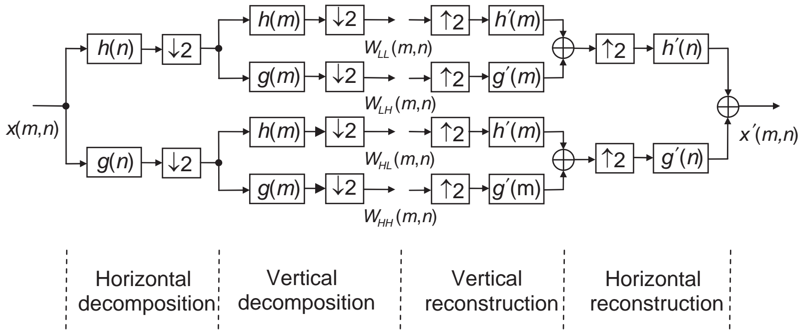


Figure 11.15 Discrete wavelet decomposition and reconstruction

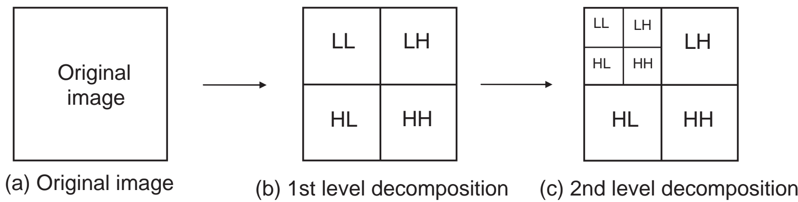


Figure 11.16 Two levels of 2-D discrete wavelet decomposition

contains the most important large-amplitude, low-frequency components called approximations, while the highpass filter output contains the small-amplitude, high-frequency components called details. Therefore, as shown in the top left corner of Figure 11.16(b), the LL decomposition image is the result of row-by-row lowpass filtering followed by column-by-column lowpass filtering, which retains most of the original image information. The size of the resulting decomposed image equals one-quarter (half of the width and half of the height) of the original image. Similarly, the LH subband image is the result of lowpass filtering of the image row by row followed by column-by-column highpass filtering; and the HL subband image is the result of row-by-row highpass filtering followed by column-by-column lowpass filtering. Finally, the HH subband image is the result of row-by-row highpass filtering followed by column-by-column highpass filtering, and thus it contains the high-frequency residuals only.

This decomposition process can be continued to the second stage (level 2), as shown in Figure 11.16(c). It is important to note that only the low-frequency subband image LL in Figure 11.16(b) is used for the second level (or stage) of wavelet decomposition. If the third stage of decomposition is needed, only the LL subband image in Figure 11.16(c) will be decomposed.

As shown in Figure 11.15, $W_{\mathrm{LL}}(m,n)$, $W_{\mathrm{LH}}(m,n)$, $W_{\mathrm{HL}}(m,n)$, and $W_{\mathrm{HH}}(m,n)$ are the DWT coefficients of the LL, LH, HL, and HH subband images, respectively. Consider an

$M \times N$ image $x(m,n)$; the one-level decomposition of the 2-D DWT can be defined as follows:

$$W_{\mathrm{LL}}(m,n) = \frac{1}{\sqrt{MN}} \sum_{l=0}^{L-1} \left\{ \left[\sum_{k=0}^{L-1} x(m,n)h(n-k) \right] h(m-l) \right\}, \tag{11.19a}$$

$$W_{\mathrm{LH}}(m,n) = \frac{1}{\sqrt{MN}} \sum_{l=0}^{L-1} \left\{ \left[\sum_{k=0}^{L-1} x(m,n)h(n-k) \right] g(m-l) \right\}, \tag{11.19b}$$

$$W_{\mathrm{HL}}(m,n) = \frac{1}{\sqrt{MN}} \sum_{l=0}^{L-1} \left\{ \left[\sum_{k=0}^{L-1} x(m,n)g(n-k) \right] h(m-l) \right\}, \tag{11.19c}$$

$$W_{\mathrm{HH}}(m,n) = \frac{1}{\sqrt{MN}} \sum_{l=0}^{L-1} \left\{ \left[\sum_{k=0}^{L-1} x(m,n)g(n-k) \right] g(m-l) \right\}, \tag{11.19d}$$

where $h(l) = \{h_0, h_1, \ldots, h_{L-1}\}$ and $g(l) = \{g_0, g_1, \ldots, g_{L-1}\}$ are the 1-D lowpass and highpass filters associated with the selected wavelet functions, and L is the order of the filters. Note that the decimation operation is not reflected in (11.19). It is important that these filters are designed as quadrature mirror filters to allow perfect reconstruction under certain conditions. These filters include the ones developed by Ingrid Daubechies, called Daubechies' wavelets. The highpass and lowpass filters are related to each other. For example, we can obtain the highpass filter coefficients as follows:

$$g(L-1-l) = (-1)^l h(l), \quad \text{for } l = 0, 1, \ldots, L-1, \tag{11.20a}$$

or

$$g_{L-1-l} = (-1)^l h_l, \quad \text{for } l = 0, 1, \ldots, L-1. \tag{11.20b}$$

On the synthesis side, there are the lowpass filter h'_l and the highpass filter g'_l. The synthesis filter coefficients h'_l and g'_l are the reverse order of the analysis filter coefficients h_l and g_l, respectively, expressed as

$$h'_l = h_{L-1-l} \quad \text{for } l = 0, 1, \ldots, L-1. \tag{11.21a}$$

and

$$g'_l = g_{L-1-l} \quad \text{for } l = 0, 1, \ldots, L-1. \tag{11.21b}$$

It is important to note that we just need to compute the lowpass analysis filter coefficients h_l since the other three coefficient sets g_l, h'_l, and g'_l can be derived from h_l for perfect reconstruction. These relationships can be verified by the MATLAB® function `wfilters`, which will be introduced in Example 11.8.

In Figure 11.15, the inverse DWT can be expressed as

$$\begin{aligned} x'(m,n) = & \frac{1}{\sqrt{MN}} \sum_{l=0}^{L-1} \left\{ \left[\sum_{k=0}^{L-1} W'_{\mathrm{LL}}(m,n) h'(n-k) \right] h'(m-l) \right\} \\ & + \frac{1}{\sqrt{MN}} \sum_{l=0}^{L-1} \left\{ \left[\sum_{k=0}^{L-1} W'_{\mathrm{LH}}(m,n) h'(n-k) \right] g'(m-l) \right\} \\ & + \frac{1}{\sqrt{MN}} \sum_{l=0}^{L-1} \left\{ \left[\sum_{k=0}^{L-1} W'_{\mathrm{HL}}(m,n) g'(n-k) \right] h'(m-l) \right\} \\ & + \frac{1}{\sqrt{MN}} \sum_{l=0}^{L-1} \left\{ \left[\sum_{k=0}^{L-1} W'_{\mathrm{HH}}(m,n) g'(n-k) \right] g'(m-l) \right\}, \end{aligned} \tag{11.22}$$

where $W'_{\mathrm{LL}}(m,n)$, $W'_{\mathrm{LH}}(m,n)$, $W'_{\mathrm{HL}}(m,n)$, and $W'_{\mathrm{HH}}(m,n)$ are the processed DWT coefficients. For example, a threshold method can be used to process the DWT coefficients for data compression. When the high-frequency residuals are smaller than the chosen threshold, these residuals can be set to zero. By varying the threshold value, one can adjust the compression ratio.

For wavelet image processing, the decomposition can be repeated continuously until the last pixel. After the wavelet decomposition, most of the important information will be located in the LL band, and the highpass filtered results will eventually become insignificant or zero after several stages of decomposition. Since the decomposition acts on image pixels, theoretically we can reconstruct the decomposition image without losing the information. This concept is important in wavelet-based image processing, especially for lossless compression. This wavelet process is known as pyramid decomposition and is used for the JPEG2000 standard. Other popular wavelet applications are edge and corner detection for pattern recognition, and noise removal for electrocardiogram (ECG) analysis.

Example 11.8

MATLAB® provides the 2-D DWT and inverse DWT functions `dwt2` and `idwt2`. In this example, we implement one level of image decomposition and a simple wavelet-based operation for edge enhancement. Figure 11.17 shows a B&W image and its wavelet transform for one level of decomposition. Note that the horizontal edges of the original image are present in the LH subband while the vertical edges are present in the HL subband. To enhance the edges, we simply zero out the LL subband image (at the top left of Figure 11.17(b)), combine these subband images into a single image, and compute the inverse wavelet transform. The reconstructed image with enhanced edges is displayed in Figure 11.17(c). The partial MATLAB® script for this example is listed as follows:

```
[X,map] = imread('reference.bmp');     % Read the image to X
[L_D,H_D,L_R,H_R] = wfilters('db2'); % Compute filter coefficients
[LL,LH,HL,HH] = dwt2(X,L_D,H_D);       % 2-D DWT using L_D and H_D
LL = zeros(size(LL));                  % Zero out LL subband image
Y = idwt2(LL,LH,HL,HH,L_R,H_R);        % Inverse 2-D DWT using L_R and H_R
```

In the script, LL, LH, HL, and HH contain the approximation, horizontal detail, vertical detail, and diagonal detail coefficients, respectively. The MATLAB® function wfilters computes four wavelet filter coefficients, where L_D and H_D are the lowpass and highpass analysis filters, respectively, and L_R and H_R are the lowpass and highpass reconstruction, respectively. Available wavelet names for the Daubechies filter are 'db1', 'db2', . . . , 'db45'.

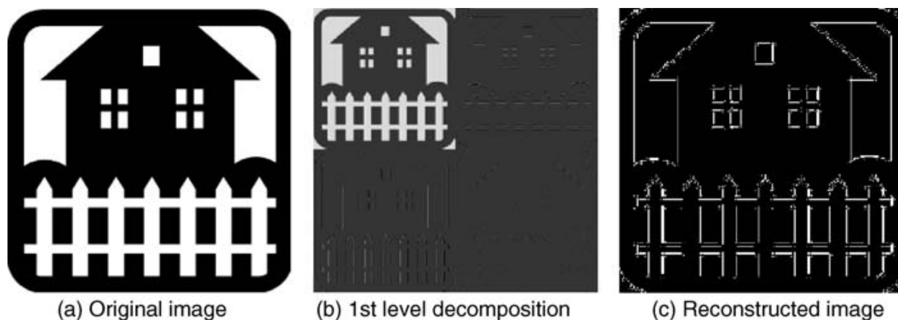


Figure 11.17 One-level wavelet transform and processing for enhancing edges

11.9 Experiments and Program Examples

This section presents image processing experiments using the C5505 eZdsp. For digital image and video processing, the computational speed is usually very critical, especially for real-time video applications. In addition, finite-wordlength effects are important for fixed-point implementations.

This section uses BMP files for the experiments. The BMP image file format is commonly used because it can be viewed by many devices including computers. BMP files can be either B&W or color images. There are some variations in the BMP file formats. For the experiments presented in this section, the Microsoft uncompressed 24-bit RGB BMP image format [6] is used. A BMP image includes a file header to describe the information of the image. A partial listing of the header for the Microsoft BMP file format is shown in Table 11.2.

Table 11.2 Partial listing of BMP file format, RGB2BMPHeader.c

```
typedef struct {
  unsigned short bfType;          // Where BMP image file starts
  unsigned long bfSize;           // BMP file size
  unsigned short bfReserved1;
  unsigned short bfReserved2;
  unsigned long bfOffBits;        // BMP data starts offset from
                                  // beginning
  unsigned long biSize;           // BMP file header info block size
  unsigned long biWidth;          // BMP image width
  unsigned long biHeight;         // BMP image height
  unsigned short biPlanes;        // BMP number of planes (must be
                                  // zero)
  unsigned short biBitCount;      // BMP number of bits per pixel
```

Table 11.2 (*Continued*)

```
  unsigned long biCompression;      // BMP compression type, 0=no
                                    // compression
  unsigned long biSizeImage;        // BMP image data length
  unsigned long biXPelsPerMeter;    // BMP X-direction number of pixels
                                    // per meter
  unsigned long biYPelsPerMeter;    // BMP Y-direction number of pixels
                                    // per meter
  unsigned long biClrUsed;          // BMP number of color used
  unsigned long biClrImportant;     // BMP number of color important
} BMPHEADER;
void createBMPHeader(unsigned short *bmpHeader, unsigned short width,
unsigned short height)
{
  BMPHEADER bmp;
  // Prepare the BMP file header
  bmp.bfType          = 0x4D42;    // Microsoft BMP file
  bmp.bfSize          = (width*height*3)+54;
  bmp.bfReserved1     = 0;
  bmp.bfReserved2     = 0;
  bmp.bfOffBits       = 54;        // Total bytes of file header
  bmp.biSize          = 40;        // Size of information block
  bmp.biWidth         = width;
  bmp.biHeight        = height;
  bmp.biPlanes        = 1;
  bmp.biBitCount      = 24;        // 8-bit RGB color data
  bmp.biCompression   = 0;         // No compression
  bmp.biSizeImage     = (width*height*3);
  bmp.biXPelsPerMeter = 0;
  bmp.biYPelsPerMeter = 0;
  bmp.biClrUsed       = 0;
  bmp.biClrImportant  = 0;
}
```

The RGB data in a BMP image file is arranged in rows (image width) from the bottom up. For example, a BMP file for resolution 120×160 is illustrated in Figure 11.18. Detailed descriptions of BMP file definitions and the variations of BMP files can be found in [6].

11.9.1 YC_bC_r to RGB Conversion

This experiment implements the color space conversion between the YC_bC_r format and the RGB format introduced in Section 11.3 using the C5505 eZdsp. The 8-bit color space conversion is realized using the fixed-point C program. The conversion supported by the MATLAB® function `ycbcr2RGB` can be expressed as

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 0.046 & 0 & 0.0063 \\ 0.046 & -0.0015 & -0.0032 \\ 0.046 & 0.0079 & 0 \end{bmatrix} \begin{bmatrix} Y - 16 \\ C_b - 128 \\ C_r - 128 \end{bmatrix}.$$

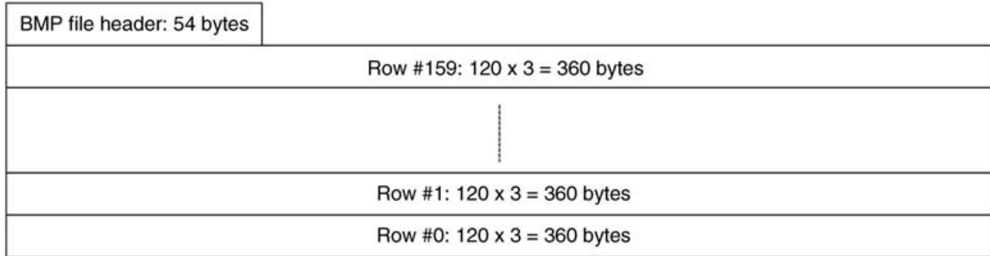


Figure 11.18 Microsoft BMP format for 120 × 160 image file

Because the YC_bC_r format uses only 8-bit data, the wordlength of data in the conversion matrix will affect the accuracy and mitigation of cumulative numerical errors during the conversion process. In order to achieve higher precision and minimize the finite-wordlength effects, an integer that provides an adequate dynamic range should be used to represent the coefficients. In this experiment, the fixed-point coefficients are represented by using 24-bit integers with a range from 0x800000 (-1.0) to 0x7FFFFF (1.0 to 2^{-23}).

The conversion results must be scaled back to maintain the proper 8-bit RGB data format. For 8-bit data, the dynamic range is from 0 to 255. Since two 8-bit data items can be packed into one 16-bit word memory location, this experiment uses the packed data format and processes two pixels at the same time. In the experiment, the 8-bit Y, C_b, and C_r input data files are converted to the RGB image and stored as a BMP file. The resulting BMP image can be viewed by a computer. Table 11.3 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load the program to the eZdsp and run the experiment to generate the BMP image file. Compare the experimental result to the provided reference BMP image (`reference_butterfly160x120.bmp`) to verify that the experimental result is the 160 × 120 resolution landscape BMP image.

Table 11.3 File listing for the experiment Exp11.1

Files	Description
`YCbCr2RGB.c`	YC_bC_r to RGB color space conversion function
`YCbCr2RGBTest.c`	Program for testing experiment
`RGB2BMPHeader.c`	C function uses RGB data to create BMP file
`ycbcr2rgb.h`	C header file
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`butterfly160x120Y8.dat`	Y component data file
`butterfly160x120Cb8.dat`	C_b component data file
`butterfly160x120Cr8.dat`	C_r component data file

3. Use MATLAB® to generate YC_bC_r files from 120 × 160 resolution portrait images. (Hint: use the MATLAB® function `imread` to read an image file; it can be a JPG, BMP, PNG, or GIF image. The function `imread` can convert the image to the RGB color space. Use the function `imresize` to resize the image to obtain the required resolution, convert the image from the RGB color space to the YC_bC_r color space, and save the YC_bC_r file for the experiment.) Modify the experiment such that it can read YC_bC_r image files and convert to 120 × 160 portrait BMP images.
4. In order to work with large-size images, it is necessary to properly define and assign eZdsp memory for the experiment. Repeat step 3 for processing 320 × 240 resolution images.

11.9.2 White Balance

Under different light sources, an object may appear in different colors. Human vision will adapt to these differences when viewing the object. However, machine vision devices, digital cameras, and camcorders cannot automatically adjust for different colors. Traditional mechanical cameras use special optical filters for color correction when taking pictures. Digital cameras use the auto-white-balance and manual-white-balance algorithms to correct the unrealistic color resulting from different light sources or conditions. The auto-white-balance technique uses data from the image to compute the "true" white color gain to balance the R, G, and B color components. Manual (or fixed) white balance uses predefined color balance settings for particular lighting conditions, such as beach and snow scenes, outdoor daylight, and indoor candescent light. The light sources are usually described using color temperature. Table 11.4 lists some useful light sources and the corresponding temperature in kelvins.

Incorrect color balance may produce bluish or reddish images. Section 11.4.2 contains the example (Example 11.2) of performing auto-white balance. In this experiment, the C5505 eZdsp is used. Figure 11.19 shows the flow diagram of the auto-white-balance method for color correction.

In this experiment, a digital camera with fixed white balance at 4150 K was used to capture three images under different lighting conditions: candescent light source (2850 K), fluorescent light source (4150 K), and daylight (6500 K). Before the white balance, the picture taken under candescent light looks reddish and the picture taken in bright daylight appears bluish.

As illustrated in Figure 11.19, this experiment consists of three steps: (1) compute the sums of R, G, and B pixels; (2) calculate the gain values for R, G, and B white-balance correction; and (3) perform the white balance on all pixels. The white-balance function will correct the

Table 11.4 Color temperatures of different light sources

Light sources	Color temperature (K)
Candle light	1000–2000
Candescent (household) light	2500–3500
Sunrise or sunset	3000–4000
Fluorescent (office) light	4000–5000
Electronic flash light	5000–5500
Sunny daylight	5000–6500
Bright overcast sky	6500–8000

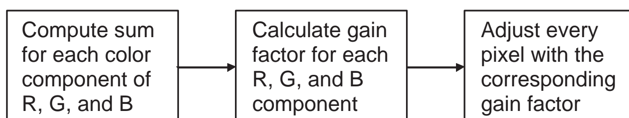


Figure 11.19 Color correction using auto-white balance

Table 11.5 File listing for the experiment Exp11.2

Files	Description
`whitebalance.c`	Fixed-point C function for white balance
`whitebalanceTest.c`	Program for testing experiment
`whiteBalance.h`	C header file
`tistdtypes.h`	Standard type define header file
`c550c.cmd`	Linker command file
`tory_2850k.bmp`	Image file taken under indoor candescent light
`tory_4150k.bmp`	Image file taken under indoor fluorescent light
`tory_6500k.bmp`	Image file taken under outdoor light at noon

unrealistic color to make them look as if they were all taken under the same fluorescent light source. The files used for the experiment are listed in Table 11.5.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Run the experiment using the eZdsp to generate the white-balanced BMP images. Verify the experimental results by comparing the processed images to the original images.
3. Compare the experimental results to observe that the white-balanced images have different intensities. This is because this experiment corrects color balance without adjusting the overall image brightness (luminance). Modify the experiment to generate white-balanced images while maintaining similar brightness levels.
4. Modify the experiment to support 320×240 resolution images instead of using 160×120 resolution BMP images given in this experiment.

11.9.3 Gamma Correction and Contrast Adjustment

Gamma correction and contrast adjustment of images are usually implemented using table lookup methods. An 8-bit image system requires a table with 256 values for each color. Figure 11.20 shows the relationships (pixel values) between the input image (solid line) and the table-mapped output images (dotted lines), where Figure 11.20(b) shows the gamma curve (dotted line) for $\gamma = 2.20$, Figure 11.20(c) maps the input image to become a low-contrast image, and Figure 11.20(d) produces the output image with high contrast. In these figures, the x axis represents input pixel values and the y axis represents output pixel values.

The tables used for the lookup method are usually generated offline or during system initialization. In this experiment, the gamma table, low-contrast table, and high-contrast table are generated during the initialization process. The functions used to generate lookup tables for the curves shown in Figure 11.20 are given in the C program `tableGen.c`. As discussed in

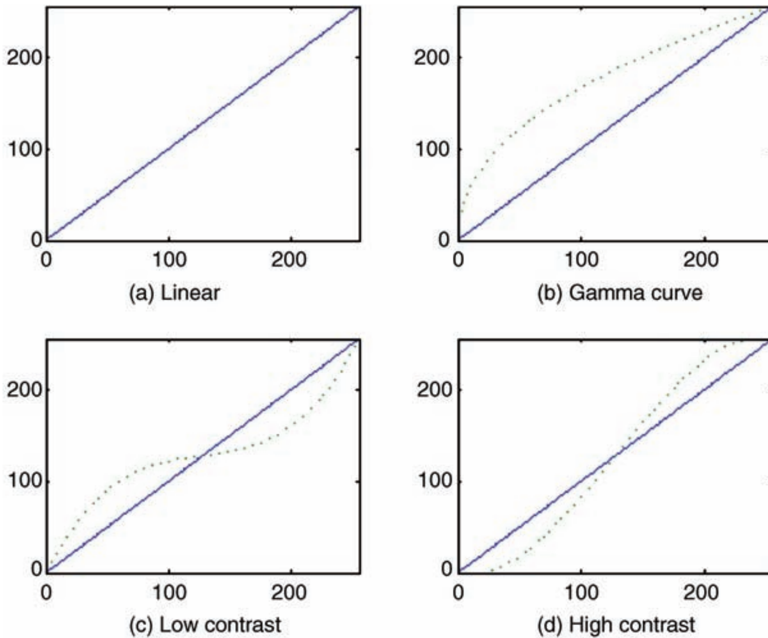


Figure 11.20 Table lookup methods for different image mappings

Section 11.4.2, gamma correction is a pre-warping process to compensate for the nonlinear characteristics of the output display devices. Most personal computers use a gamma value of 2.20. The contrast adjustment is achieved by changing the pixel distribution as described by the contrast curves shown in Figure 11.20. Table 11.6 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Run the experiment to generate the BMP images from the gamma correction and high- and low-contrast adjustments. Verify the experimental results by comparing the resulting images to the original images.

Table 11.6 File listing for the experiment Exp11.3

Files	Description
`tableGen.c`	Fixed-point C function for generating lookup tables
`imageMapping.c`	Fixed-point C function for image correction
`gammaContrastTest.c`	Program for testing experiment
`gammacontrast.h`	C header file
`tistdtypes.h`	Standard type define header file
`c5505cmd`	Linker command file
`boat.bmp`	BMP image file for gamma experiments
`temple.bmp`	BMP image file for high- and low-contrast experiments

3. Repeat the experiment using different gamma values 1.0, 1.8, 2.2, and 2.5, and compare the differences.
4. Examine `tabelGen.c` to derive the mathematical equation used to generate the high-contrast lookup table. Modify the equation such that it can achieve higher contrast than the current experimental setting. Rerun the experiment and observe the differences. Use different mathematical equations to create different contrast curves similar to Figure 11.20 (c) and Figure 11.20(d), redo the experiment, and observe the results.

11.9.4 Image Filtering

Image filtering usually requires intensive computation because of the large number of pixels to be processed. For real-time applications, image filtering can be efficiently implemented using assembly routines and/or specific hardware accelerators.

This experiment uses three 3×3 filter (lowpass, highpass, and Laplacian) kernels. The input image pixels are placed in the data buffer `pixel[3×IMG_WIDTH]`, which contains three rows of image pixels. The 3×3 filter coefficients are placed in the array `filter[I×J]` for the experiment. The image pixel $x(m, n)$ and its eight neighboring pixels (refer to Figure 11.7) are multiplied by the corresponding filter coefficients and the products are summed to produce the filtered output pixel $y(m, n)$. The processing starts with three rows of data. We use index n for the current row of the image data. The row indexed by $n - 1$ is the previous row and $n + 1$ is the next row. The filtering process continues from the first column to the last column for each row. The image data buffer is updated one row at a time by adjusting the data buffer index n. In this experiment, three different resolutions of images are used for image filtering. Table 11.7 lists the files used for the experiment.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Run the following experiments to generate the 2-D filter output BMP images:
   (a) Select the highpass filter option. Compare the output image to the original image to check if the output image appears sharper than the original image.
   (b) Select the Laplacian filter option. Verify that the edges of the output image are enhanced for edge detection as compared to the original image.
   (c) Select the lowpass filter option. Verify that the resulting image has smoother (blurred) edges.
3. Repeat the experiment using the 3×3 emboss and engrave filters defined in Section 11.6. Examine the output images to observe the filtering effects.

Table 11.7 File listing for the experiment Exp11.4

Files	Description
`filter2D.asm`	C55xx assembly 2-D filter function
`filter2DTest.c`	Program for testing experiment
`filter2D.h`	C header file
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`eagle160x160.bmp`	BMP image file for highpass filter experiment
`hallway160x120.bmp`	BMP image file for Laplacian filter experiment
`kingProtea160x128.bmp`	BMP image file for lowpass filter experiment

4. Modify the program such that it is able to filter the image twice. Perform the experiment using the 3×3 Sobel and Prewitt filters defined in Section 11.6. Perform the horizontal filtering first and then the vertical filtering. Examine the output images to observe the filtering effects.
5. Generate some B&W images using MATLAB® for experiments. Repeat the experiment using the emboss, engrave, Sobel, and Prewitt filters for 2-D filtering of these B&W images and observe the results. (Hint: B&W images can be obtained using the MATLAB® function `rgb2gray`.)
6. Modify the experiment to process large-size images such as 320×240 resolution.

11.9.5 DCT and IDCT

This experiment implements the most popular 8×8 DCT and IDCT for image processing using the C5505 eZdsp. In the experiment, the DCT is applied to the input image to obtain the DCT coefficients, and then the IDCT is performed to reconstruct the image for comparison. C55xx assembly routines for DCT and IDCT functions are used to improve image processing efficiency. In the experiment, 64 pixels used for 8×8 DCT are placed sequentially in a data array of size 64 and the DCT coefficients are also stored in an array of size 64. A 1-D data array is used to store 2-D image pixels as this a common programming practice since the physical memory is addressed sequentially, even though a 2-D image is viewed as a block of $M \times N$ pixels. The 2-D DCT is carried out using two 1-D DCTs. First, the 1-D DCT is applied to each column of the 8×8 image. The results are stored in an intermediate data buffer. Then the DCT is applied to each row of the intermediate 8×8 data buffer to obtain the results. In the experiment, the resulting 8×8 DCT coefficients are stored in the same data buffer used by the input image. This is the in-place computation method (introduced in Chapter 5) because the results overwrite the original data after the transform is completed.

The DCT and IDCT coefficients are computed according to (11.17) and (11.18). In order to reduce numerical errors, the DCT coefficients are represented by 12-bit integers to keep the DCT process at higher precision. The DCT and IDCT functions will scale the data back to 8-bit format when the results are stored back to memory. The coefficients used by the DCT and IDCT are computed during the initialization stage by the function `DCTcoefGen()`.

The C5505 eZdsp has limited memory to load the entire image for processing. In this experiment, the width and height of the digital image are constrained to multiples of eight pixels. We load only eight rows of image data each time for performing the 8×8 DCT from the first column (contains eight rows of pixels) to the last column. Once all eight rows of data are processed, we will read the next eight rows of pixels until finishing the last row of image data. For this experiment, the image files must have their width and height in multiples of eight pixels for 8×8 DCT and IDCT operations. The files used for the experiment are listed in Table 11.8.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Run the experiment to perform DCT/IDCT operations on the provided YC_bC_r image files in the companion software package, convert the DCT/IDCT result from the YC_bC_r color space to the RGB color space, and save the image as a BMP file. Compare the resulting image with the reference BMP image (`reference_hat160x120.bmp`) to verify the experimental result.

Table 11.8 File listing for the experiment Exp11.5

Files	Description
`DCT.asm`	C55xx assembly DCT function
`IDCT.asm`	C55xx assembly IDCT function
`dctTest.c`	Program for testing experiment
`DCTcoefGen.c`	C program generates DCT and IDCT coefficients
`YCbCr2RGB.c`	YC_bC_r to RGB color space conversion function
`RGB2BMPHeader.c`	C function to construct BMP file using RGB data
`dct.h`	C header file
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`hat160x120Y8.dat`	Y component data file
`hat160x120Cb8.dat`	C_b component data file
`hat160x120Cr8.dat`	C_r component data file

3. Modify the experiment to read a color BMP image from the input file and convert it to a gray-scale image for the experiment. Apply the DCT/IDCT to the gray-scale image. Write the resulting image to a BMP file. Verify that, except when the output is a gray-scale image, it should not have any visual errors or artifacts.
4. Modify step 2 of the experiment by zeroing out the DCT coefficients that are smaller than a predetermined threshold before the computation of the IDCT. Compare the reconstructed image to the original image to observe if there is any artifact. Redo this experiment using different threshold values and observe the distortions.
5. Modify the experiment to work on one block of 8×8 image pixels at a time. Redo the experiment described in step 2. This modification provides a more flexible experiment for processing large-size images since the DCT/IDCT operations will be based on the 8×8 image blocks instead of eight rows of image data.

11.9.6 Image Processing for Fingerprints

As discussed in Section 11.8.4, fingerprint applications use several image processing algorithms such as image filtering for edge enhancement. This experiment presents a simplified fingerprint application [7] using the C5505 eZdsp. For real-world fingerprint applications, there are several requirements for fingerprint acquisitions. However, the fingerprint image used for this experiment is a standard BMP file for simplicity. Also, this experiment uses only the green pixels, that is, the G components of the RGB image. The files used for the experiment are listed in Table 11.9.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load the program to the eZdsp and run the experiment to obtain the resulting fingerprint. Compare the output images to the original fingerprint as shown in Figure 11.21. Observe the following differences:
   (a) Figure 11.21(b) shows that the histogram equalization process improves image contrast.

Table 11.9 File listing for the experiment Exp11.6

Files	Description
fingerPrintTest.c	Program for testing experiment
fingerPrint.c	C program for fingerprint algorithms
TA_demo_algo.h	C header file
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file
fingerprint64x64.bmp	Fingerprint BMP file

(b) Figure 11.21(c) shows that the image filtering enhances the edges.
(c) Figure 11.21(d) is the single-pixel fingerprint image which can be used for identification.

3. Profile the fingerprint program to identify five key functions that consume most of the processing power. Optimize these functions by using C55xx intrinsics. Once the program has been modified and fully tested, rerun the experiment to profile the computational saving achieved by using intrinsics.

11.9.7 The 2-D Wavelet Transform

This experiment implements the 2-D DWT using the C5505 eZdsp. The program reads a YC_bC_r image, applies the 2-D DWT to decompose the image and IDWT for reconstruction, converts the resulting YC_bC_r image to the RGB format and, stores it as a BMP file for comparison. For the 2-D DWT, the decomposition process is first applied to the rows and then to the columns. The resulting one-stage decomposition has four subband images. In this experiment, the decomposition process uses four Daubechies filters with coefficients depending upon vanish-moment factor p. For example, the decomposition lowpass filter coefficients for the factor $p = 2$ are -0.1294, 0.2241, 0.8365, and 0.4830; and for the factor $p = 4$, the filter coefficients are -0.0106, 0.0329, 0.0308, -0.1870, -0.0280, 0.6309, 0.7148, and 0.2304.

These Daubechies filter coefficients are converted to the fixed-point Q15 format by multiplying by the 15-bit integer 32 767 (0x7FFF). For the vanish-moment factor $p = 2$, the analysis and reconstruction filters coefficients are obtained using the MATLAB® function `[L_D,H_D,L_R,H_R] = wfilters('db2')` and the results are displayed as follows:

Lowpass decomposition filter h_i: `L_D = {-4240,7345,27410,15825},`
Highpass decomposition filter g_i: `H_D = {-15825,27410,-7345,-4240},`
Lowpass reconstruction filter h'_i: `L_R = {15825, 27410,7345,-4240},`
Highpass reconstruction filter g'_i: `H_R = {-4240,-7345,27410,-15825},`

where `L_D` $\{h_0, h_1, h_2, h_3\}$ and `H_D` $\{g_0, g_1, g_2, g_3\}$ are the coefficients for the lowpass and highpass analysis filters, respectively; and `L_R` $\{h'_0, h'_1, h'_2, h'_3\}$ and `H_R` $\{g'_0, g'_1, g'_2, g'_3\}$ are the coefficients for the lowpass and highpass synthesis filters, respectively (same as `example11_8.m`). Their values satisfy the perfect reconstruction conditions defined in (11.20) and (11.21).

(a) Original fingerprint image
(b) Equalized image
(c) Edge-enhanced image
(d) Single-pixel image

Figure 11.21 Fingerprint experimental results

This experiment arranges Daubechies filter coefficients in the following order for efficiently using the C55xx circular addressing mode with dual-MAC read and write architecture:

$$\begin{bmatrix} N & & & \\ h_3 & h_2 & h_1 & h_0 \\ h_0 & -h_1 & h_2 & -h_3 \\ h_1 & h_2 & h_3 & h_0 \\ h_0 & -h_3 & h_2 & -h_1 \end{bmatrix} = \begin{bmatrix} 4 & & & \\ 15\,825 & 27\,410 & 7345 & -4240 \\ -4240 & -7345 & 27\,410 & -15\,825 \\ 7345 & 27\,410 & 15\,825 & -4240 \\ -4240 & -15\,825 & 27\,410 & -7345 \end{bmatrix},$$

where $N = 4$ is the filter length for the vanish-moment factor $p = 2$.

This experiment also supports filter coefficients for different vanish-moment factors in the range of $2 \leq p \leq 10$. For example, when $p = 4$ the analysis lowpass, analysis highpass,

synthesis lowpass, and synthesis highpass filter coefficients are expressed as

$$\begin{bmatrix} N & & & & & & & \\ h_7 & h_6 & h_5 & h_4 & h_3 & h_2 & h_1 & h_0 \\ h_0 & -h_1 & h_2 & -h_3 & h_4 & -h_5 & h_6 & -h_7 \\ h_1 & h_6 & h_3 & h_4 & h_5 & h_2 & h_7 & h_0 \\ h_0 & -h_7 & h_2 & -h_5 & h_4 & -h_3 & h_6 & -h_1 \end{bmatrix},$$

where the filter length $N = 8$, and the coefficients represented in the Q15 format are $h_0 = -347$, $h_1 = 1077$, $h_2 = 1011$, $h_3 = -6129$, $h_4 = -917$, $h_5 = 20\,672$, $h_6 = 23\,423$, and $h_7 = 7549$. These analysis and reconstruction filter coefficients are obtained using the MATLAB® function `[L_D,H_D,L_R,H_R] = wfilters('db4')`. More filter coefficients are provided for the experiment by the software `2DWavelet.c` [8,9].

A longer (higher order) filter can provide better smoothness and finer intermediate results with the trade-off of higher computational loads. Since the DWT, realized by analysis and synthesis filterbanks, has perfect reconstruction capability, a shorter filter should be considered if the intermediate results are not required for other processing. The files used for the experiment are listed in Table 11.10.

The 2-D wavelet decomposition results are shown in Figure 11.22. The original image is given in Figure 11.22(a). The resulting images from the first-level decomposition are shown in Figure 11.22(b). The top left image is the LL band image; top right, the LH band image; bottom left, the HL band image; and bottom right, the HH band image. To continue with the level 2 decomposition, the LL subband image from the level 1 decomposition is used for the transform. The results of level 2 decomposition images are shown in Figure 11.22(c). The reconstructed image from the level 2 DWT/IDWT with the operation of zeroing values smaller than a predetermined threshold is shown in Figure 11.22(d).

Table 11.10 File listing for the experiment Exp11.7

Files	Description
`2DWaveletTest.c`	Program for testing experiment
`2DWavelet.c`	C program for 2-D wavelet
`YCbCr2RGB.c`	YC_bC_r to RGB color space conversion function
`RGB2BMPHeader.c`	C function uses RGB data to create BMP file
`col2rowmn.asm`	Program copies column data to buffer
`decInplcemn.asm`	Program for 1-D wavelet decomposition
`decomColmn.asm`	Program for decomposition of column
`interColmn.asm`	Program interlaces lowpass and highpass filter outputs
`recInplcemn.asm`	Program for wavelet reconstruction
`reconColmn.asm`	Program for 1-D column reconstruction
`wavelet.h`	C header file
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file
`totem128x128Y8.dat`	Y component data file
`totem128x128Cb8.dat`	C_b component data file
`totem128x128Cr8.dat`	C_r component data file

Figure 11.22 Wavelet decomposition and reconstruction of images.

Procedures of the experiment are listed as follows:

1. Import the CCS project from the companion software package and rebuild the project.
2. Load the program to the eZdsp and run the experiment using the following options:
   (a) Select option 0 to generate the reference image (`reference.bmp`) for the experiment. This is done by converting the YC_bC_r image to the RGB image as shown in Figure 11.22(a). This image will be used as the reference for comparison to the DWT/IDWT results obtained from the experiment.

(b) Rerun the experiment with option 1 to generate the composite level 1 wavelet decomposition image, as shown in Figure 11.22(b).
(c) Rerun the experiment with option 2 to generate the composite level 2 wavelet decomposition image, as shown in Figure 11.22(c).
(d) Rerun the experiment and select option 3 to generate the synthesized full-band image and compare the output image to the original image (`reference.bmp`) to examine if the perfect wavelet reconstruction of image is achieved.
(e) Rerun the experiment and select option 4 to generate the subband images with the small DWT coefficients set to zero based on a predetermined threshold, and then reconstruct the full-band image using the level 2 wavelet process as shown in Figure 11.22(d). Compare to the original image (`reference.bmp`) to verify that the reconstructed image has no obvious artifacts or errors. Redo this experiment with different threshold values and observe the resulting artifacts.

3. Modify the experiment for the level 1 decomposition to zero-out the HH band image. Can we reconstruct a good-quality image without the HH band image? Redo the experiment to zero-out the LL band image and reconstruct the full-band image; observe the processing effects.
4. Modify the experiment for the level 2 decomposition to zero-out the HH, LH, and HL images obtained from the level 2 decompositions. Can we reconstruct a satisfactory quality of image by using only the level 2 LL band image?

Exercises

11.1. Based on Example 11.1, write a MATLAB® script that will read in the RGB data file (`BGR.RGB`) from the experiment given in Section 11.9.1, and display the RGB image using the MATLAB® function `imshow`. Use visual inspection to compare the RGB image displayed by MATLAB® and the bit-map image generated by Exp11.1.

11.2. Develop an experiment that converts the RGB color space to the YC_bC_r color space, and then converts the YC_bC_r color space back to the RGB color space using the C5505 eZdsp. In this experiment, use the input image from a BMP file, and generate the resulting image using the BMP file for viewing.

11.3. Use the MATLAB® function `imadjust` to adjust image intensity values to achieve a similar result to the one obtained in Example 11.3, which uses gamma correction to brighten the original dark image.

11.4. Compute the histogram of the original and gamma-corrected images given in Example 11.3. Verify the results using the MATLAB® function `imhist`, which displays a histogram for the image intensity.

11.5. Redo Example 11.4 using the MATLAB® function `histeq`, which enhances the contrast using histogram equalization.

11.6. Example 11.6 implements a 2-D filter in the frequency domain. The resulting image has been shifted toward the bottom right. Modify the MATLAB® script such that the edge effects will be minimized.

11.7. Develop a C5505 eZdsp experiment based on Example 11.4 to equalize the image evenly using histogram equalization.

11.8. In the 2-D image filtering experiment given in Section 11.9.4, the pixels along the image boundaries are not properly filtered. In order to properly filter the image, we need to pad zeros to the image such that the pixels in the image perimeters can be placed in the center of the 3×3 filter kernel. Modify the 2-D filtering experiment by (1) padding zeros and (2) using (duplicating) the same perimeter pixels, and compare the differences of these two methods.

11.9. Modify the fingerprint experiment such that the fingerprint algorithm uses the luma (Y) components of the 128×128 resolution fingerprint image in the YC_bC_r color space. (Hint: use MATLAB® to convert the BMP fingerprint image given by Exp11.6 to the YC_bC_r color space and extract the Y data components.)

11.10. Example 11.8 uses the MATLAB® 2-D functions `dwt2()` and `idwt2()`. As shown in Figure 11.15 and equations (11.19) and (11.22), the 2-D DWT is equivalent to applying the 1-D DWT to the rows followed by the columns. Use the MATLAB® 1-D DWT function `dwt()` and IDWT function `idwt()` to implement the image decomposition and reconstruction operations and compare the result to Example 11.8.

References

1. Sakamoto, T., Nakanishi, C., and Hase, T. (1998) Software pixel interpolation for digital still cameras suitable for a 32-bit MCU. *IEEE Trans. Consum. Electron.*, **44**, 1342–1352.
2. ITU-R Recommendation (2011) BT.601-7, Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-screen 16:9 Aspect Ratios, March.
3. ITU Report (1990) BT.624-4, Characteristics of Systems for Monochrome and Color Television.
4. ITU-T Recommendation (1992) T.81, Information Technology – Digital Compression and Coding of Continuous-tone Still Images – Requirements and Guidelines.
5. McGovern, F.A., Woods, R.F., and Yan, M. (1994) Novel VLSI implementation of (8×8) point 2-D DCT. *Electron. Lett.*, **30**, 624–626.
6. Charlap, D. (1995) The BMP file format, Part 1. *Dr. Dobb's J. Software Tools*, **20** (228).
7. Texas Instruments, Inc. (2011) TMS320C5515 Fingerprint Development Kit (FDK) Software Guide, SPRUH47, April.
8. Texas Instruments, Inc. (2002) Wavelet Transforms in the TMS320C55x, SPRA800, January.
9. Texas Instruments, Inc. (2004) TMS320C55x Image/Video Processing Library Programmer's Reference, SPRU037C, January.

Appendix A

Some Useful Formulas and Definitions

This appendix briefly summarizes some basic formulas and definitions of algebra that are used extensively in this book [1].

A.1 Trigonometric Identities

Trigonometric identities are often required in the manipulation of Fourier series, transforms, and harmonic analysis. Some of the most common identities are as follows:

$$\sin(-\alpha) = -\sin \alpha \tag{A.1a}$$

$$\cos(-\alpha) = \cos \alpha \tag{A.1b}$$

$$\sin(\alpha \pm \beta) = \sin \alpha \cos \beta \pm \cos \alpha \sin \beta \tag{A.2a}$$

$$\cos(\alpha \pm \beta) = \cos \alpha \cos \beta \mp \sin \alpha \sin \beta \tag{A.2b}$$

$$2 \sin \alpha \sin \beta = \cos(\alpha - \beta) - \cos(\alpha + \beta) \tag{A.3a}$$

$$2 \cos \alpha \cos \beta = \cos(\alpha + \beta) + \cos(\alpha - \beta) \tag{A.3b}$$

$$2 \sin \alpha \cos \beta = \sin(\alpha + \beta) + \sin(\alpha - \beta) \tag{A.3c}$$

$$\sin \alpha \pm \sin \beta = 2 \sin\left(\frac{\alpha \pm \beta}{2}\right)\cos\left(\frac{\alpha \mp \beta}{2}\right) \tag{A.4a}$$

$$\cos \alpha + \cos \beta = 2 \cos\left(\frac{\alpha + \beta}{2}\right)\cos\left(\frac{\alpha - \beta}{2}\right) \tag{A.4b}$$

$$\cos \alpha - \cos \beta = -2 \sin\left(\frac{\alpha + \beta}{2}\right)\sin\left(\frac{\alpha - \beta}{2}\right) \tag{A.4c}$$

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

$$\sin(2\alpha) = 2\sin\alpha\cos\alpha \quad \text{(A.5a)}$$

$$\cos(2\alpha) = 2\cos^2\alpha - 1 = 1 - 2\sin^2\alpha \quad \text{(A.5b)}$$

$$\sin\left(\frac{\alpha}{2}\right) = \sqrt{\frac{1}{2}(1 - \cos\alpha)} \quad \text{(A.6a)}$$

$$\cos\left(\frac{\alpha}{2}\right) = \sqrt{\frac{1}{2}(1 + \cos\alpha)} \quad \text{(A.6b)}$$

$$\sin^2\alpha + \cos^2\alpha = 1 \quad \text{(A.7a)}$$

$$\sin^2\alpha = \frac{1}{2}[1 - \cos(2\alpha)] \quad \text{(A.7b)}$$

$$\cos^2\alpha = \frac{1}{2}[1 + \cos(2\alpha)] \quad \text{(A.7c)}$$

$$e^{\pm j\alpha} = \cos\alpha \pm j\sin\alpha \quad \text{(A.8a)}$$

$$\sin\alpha = \frac{1}{2j}\left(e^{j\alpha} - e^{-j\alpha}\right) \quad \text{(A.8b)}$$

$$\cos\alpha = \frac{1}{2}\left(e^{j\alpha} + e^{-j\alpha}\right). \quad \text{(A.8c)}$$

In Euler's theorem, Equations (A.8), $j = \sqrt{-1}$. The basic concepts and manipulations of complex number will be reviewed in Section A.3.

A.2 Geometric Series

The geometric series is used in discrete-time signal analysis to evaluate functions in closed form. Its basic form is

$$\sum_{n=0}^{N-1} x^n = \frac{1 - x^N}{1 - x}, \quad x \neq 1. \quad \text{(A.9)}$$

This is a widely used identity. For example,

$$\sum_{n=0}^{N-1} e^{-j\omega n} = \sum_{n=0}^{N-1} \left(e^{-j\omega}\right)^n = \frac{1 - e^{-j\omega N}}{1 - e^{-j\omega}}. \quad \text{(A.10)}$$

If the magnitude of x is less than one, the infinite geometric series converges to

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1 - x}, \quad |x| < 1. \quad \text{(A.11)}$$

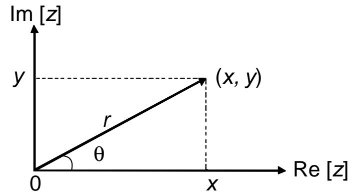


Figure A.1 Complex numbers represented in Cartesian form

A.3 Complex Variables

A complex number z can be expressed in rectangular (Cartesian) form as

$$z = x + \mathrm{j}y = \mathrm{Re}[z] + \mathrm{j}\,\mathrm{Im}[z], \tag{A.12}$$

where $\mathrm{Re}[z] = x$ and $\mathrm{Im}[z] = y$. Since the complex number z represents the point (x, y) in the two-dimensional plane, it can be illustrated as a vector, as shown in Figure A.1. The horizontal coordinate x is called the real part, and the vertical coordinate y is the imaginary part.

As shown in Figure A.1, the vector z also can be defined by its length (radius) r and its direction (angle) θ. The x and y coordinates of the vector are given by

$$x = r\cos\theta \quad \text{and} \quad y = r\sin\theta. \tag{A.13}$$

Substituting (A.13) into (A.12) and using Euler's theorem defined in (A.8a), the vector z can be expressed in polar form as

$$z = r\cos\theta + \mathrm{j}\,r\sin\theta = r(\cos\theta + \mathrm{j}\sin\theta) = r\,\mathrm{e}^{\mathrm{j}\theta}, \tag{A.14}$$

where

$$r = |z| = \sqrt{x^2 + y^2} \tag{A.15}$$

is the magnitude of the vector z and

$$\theta = \begin{cases} \tan^{-1}\left(\dfrac{y}{x}\right), & \text{if } x \geq 0 \\ \tan^{-1}\left(\dfrac{y}{x}\right) \pm \pi, & \text{if } x < 0 \end{cases} \tag{A.16}$$

is the angle (or phase) in radians.

The basic arithmetic operations for two complex numbers, $z_1 = x_1 + \mathrm{j}\,y_1$ and $z_2 = x_2 + \mathrm{j}\,y_2$, are listed as follows:

$$z_1 \pm z_2 = (x_1 \pm x_2) + \mathrm{j}(y_1 \pm y_2) \tag{A.17}$$

$$z_1 z_2 = (x_1 x_2 - y_1 y_2) + \mathrm{j}(x_1 y_2 + x_2 y_1) \tag{A.18a}$$

$$z_1 z_2 = (r_1 r_2) e^{j(\theta_1+\theta_2)} \quad (A.18b)$$

$$z_1 z_2 = (r_1 r_2)[\cos(\theta_1 + \theta_2) + j \sin(\theta_1 + \theta_2)] \quad (A.18c)$$

$$\frac{z_1}{z_2} = \frac{(x_1 x_2 + y_1 y_2) + j(x_2 y_1 - x_1 y_2)}{x_2^2 + y_2^2} \quad (A.19a)$$

$$\frac{z_1}{z_2} = \frac{r_1}{r_2} e^{j(\theta_1-\theta_2)} \quad (A.19b)$$

$$\frac{z_1}{z_2} = \frac{r_1}{r_2}[\cos(\theta_1 - \theta_2) + j \sin(\theta_1 - \theta_2)]. \quad (A.19c)$$

Note that addition and subtraction are straightforward in rectangular form, but complicated in polar form. Division is simple in polar form, but complicated in rectangular form.

The complex arithmetic of the complex number z can be listed as

$$z^* = x - j\, y = r\, e^{-j\theta}, \quad (A.20)$$

where * denotes complex-conjugate operation. In addition,

$$z + z^* = 2 \,\mathrm{Re}[z] = 2x \quad (A.21a)$$

$$zz^* = |z|^2 = x^2 + y^2 \quad (A.21b)$$

$$|z| = \sqrt{zz^*} = \sqrt{x^2 + y^2} \quad (A.21c)$$

$$z^{-1} = \frac{1}{z} = \frac{1}{r} e^{-j\theta}, \quad (A.22)$$

$$z^N = r^N e^{jN\theta} = r^N[\cos(N\theta) + j \sin(N\theta)]. \quad (A.23)$$

The solutions of

$$z^N = 1 \quad (A.24)$$

are

$$z_k = e^{j\theta_k} = e^{j(2\pi k/N)}, \quad k = 0, 1, \ldots, N-1. \quad (A.25)$$

As illustrated in Figure A.2, these N solutions are equally spaced around the unit circle $|z| = 1$ where $r = 1$. The angular spacing between the adjacent points is $\theta = 2\pi/N$.

A.4 Units of Power

Power and energy calculations are important in circuit analysis. Power is defined as the time rate of expending or absorbing energy, and can be expressed in the form of a derivative as

$$P = \frac{dE}{dt}, \quad (A.26)$$

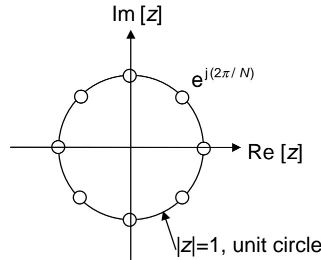


Figure A.2 Graphical display of the Nth roots of unity, $N = 8$

where P is the power in watts, E is the energy in joules, and t is the time in seconds. The power associated with the voltage and current can be expressed as

$$P = vi = \frac{v^2}{R} = i^2 R, \tag{A.27}$$

where v is the voltage in volts, i is the current in amperes, and R is the resistance in ohms.

In engineering applications, the most popular description of signal strength is the decibel (dB) defined as

$$N = 10 \log_{10}\left(\frac{P_x}{P_y}\right) \text{ dB}. \tag{A.28}$$

Therefore, the decibel unit is used to describe the ratio of two powers and requires a reference value, P_y, for comparison.

It is important to note that both the current $i(t)$ and voltage $v(t)$ can be considered as an analog signal $x(t)$, thus the power of a signal is proportional to the square of the signal amplitude. For example, if the signal $x(t)$ is amplified (or attenuated) by a factor g, that is, $y(t) = gx(t)$, the signal gain can be expressed in dB as

$$Gain = 10 \log_{10}\left(\frac{P_y}{P_x}\right) = 20 \log_{10}(g), \tag{A.29}$$

since the power is a function of the square of the voltage (or current) as shown in (A.28).

Some useful units which are used in this book are defined as follows.

SNR: One of the widely used terms in practice is the signal-to-noise ratio (SNR) between the signal $x(n)$ and the noise $v(n)$ defined as

$$\text{SNR} = 10 \log_{10}\left(\frac{P_x}{P_v}\right) \text{ dB}, \tag{A.30}$$

where

$$P_x = \lim_{N\to\infty} \frac{1}{N}\sum_{n=0}^{N-1} x^2(n) \quad \text{and} \quad P_v = \lim_{N\to\infty} \frac{1}{N}\sum_{n=0}^{N-1} v^2(n)$$

are the power of $x(n)$ and $v(n)$, respectively.

SPL: Sound pressure level, or sound level, L_p, is a logarithmic measurement of sound relative to the standard reference level. It is defined in dB units as

$$L_p = 10\,\log_{10}\left(\frac{P_{\text{rms}}^2}{P_{\text{ref}}^2}\right) = 20\,\log_{10}\left(\frac{P_{\text{rms}}}{P_{\text{ref}}}\right), \tag{A.31a}$$

where P_{rms} is the sound pressure measured using the root-mean-square (RMS or rms) value defined as

$$P_{\text{rms}} = \sqrt{\frac{1}{N}\sum_{n=0}^{N-1} x^2(n)} \tag{A.31b}$$

and P_{ref} is the RMS value of the reference sound pressure. The commonly used reference sound pressure in air is $P_{\text{ref}} = 20\,\mu\text{Pa}$ (micropascals).

dBm: When the reference signal $x(t)$ has power P_x equal to 1 milliwatt, the power unit of $y(t)$ is called dBm (dB with respect to 1 milliwatt).

dBm0: The digital reference level for the dBm0 is the digital milliwatt defined in ITU-T Recommendation G.168 [2], in which the dBm0 is measured as

$$P_k = 3.14 + 20\,\log\left[\frac{\sqrt{(2/N)\sum_{i=k}^{k-N+1} x_i^2}}{4096}\right] \quad (A\text{-law encoding}), \tag{A.32a}$$

$$P_k = 3.17 + 20\,\log\left[\frac{\sqrt{(2/N)\sum_{i=k}^{k-N+1} x_i^2}}{8159}\right] \quad (\mu\text{-law encoding}), \tag{A.32b}$$

where P_k is the signal level in dBm0, x_i is the linear equivalent of the PCM (pulse code modulation) encoded signal at time i, k is the time index, and N is the number of samples used for the measurement of RMS.

dBov: dBov [3] is the level, in decibels, relative to the overload point of the system; that is, the highest intensity signal that can be encoded by the codec. Representation relative to the overload point of a system is particularly useful for digital implementations, since one does not need to know the relative calibration of the analog circuitry. For example, in the case of G.711 μ-law audio [4], the 0 dBov reference would be a square wave with values in the range of ± 8031. This is translated to 6.18 dBm0, relative to the μ-law's dBm0 defined in G.168.

References

1. Tuma, Jan J. (1979) *Engineering Mathematics Handbook*, McGraw-Hill, New York.
2. ITU-T Recommendation (2000) G.168, Digital Network Echo Cancellers.
3. IETF (2002) RFC 6464, A Real-time Transport Protocol (RTP) Header Extension for Client-to-Mixer Audio Level Indication, http://tools.ietf.org/html/rfc6464 (accessed April 30, 2013).
4. ITU-T Recommendation (2000) G.711, Appendix II, A Comfort Noise Payload Definition.

Appendix B

Software Organization and List of Experiments

The companion software package includes the entire program and data files used for this book. These MATLAB®, floating-point and fixed-point C, and TMS320C55xx assembly programs along with the associated data files can be downloaded from the website http://www.wiley.com/go/kuo_dsp. Figure B.1 shows the directory structure of the software, including examples, exercises, and experiments provided by the companion software package for this book. In the figure, Chapter 4 (`Ch4`) is used as an example for showing the detailed structure of subdirectories.

Under the root software folder, there are 12 subfolders: Appendix C and Chapters 1 to 11. These folders contain the `Examples`, `Experiments`, and `Exercises` directories.

The `Exercises` folder contains the necessary data files and programs for the exercise problems listed at the end of that chapter.

The `Examples` folder consists of one or more directories. Each directory is named according to the example number referred to by the chapter. For example, in the directory `Example 4.8` the software program `example4_8.m` is the MATLAB® program written for *Example 4.8* in Chapter 4. Some example folders also contain the required data files for the MATLAB® program.

The `Experiments` directory contains the experiments for the chapter. The experiments directory is named with the chapter number and the experiment number. For example, experiment folder `Exp4.6` as shown in Figure B.1 is the sixth experiment in Chapter 4. There are two folders under the experiment folder. The first, Code Composer Studio (CCS), automatically creates the directory `.metadata`. The other folder is the DSP project directory that is named for the specific function of the experiment. For example, as shown in Figure B.1, the project folder for `Exp4.6` is `realtimeIIR`, the real-time IIR filtering experiment. Inside the DSP project folder, there are several directories. The `data` directory holds the input data files used by the given experiment, and is also used to keep the experiment output results. The `Debug` directory is used by the CCS to store temporary files and the DSP executable program. The `inc` directory has the C header files (`.h`) and assembly include files (`.dat` and `.inc`).

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

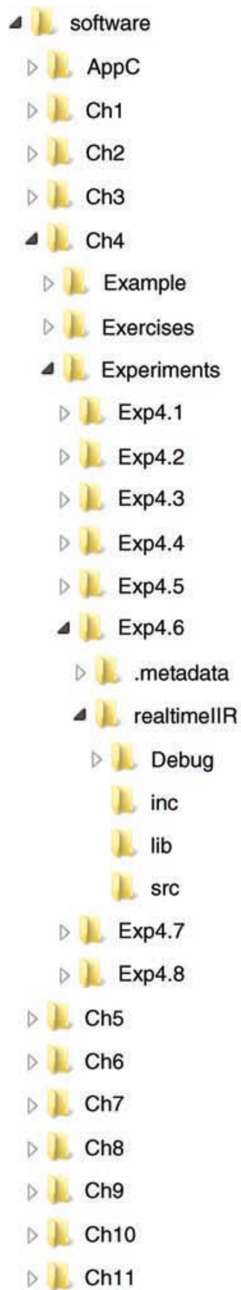


Figure B.1 Software directory structure

Table B.1 File types and formats used in the book

File extension	File type and format	Description
.asc	ASCII text	ASCII text file
.asm	ASCII text	C55xx assembly program source file
.bin	Binary	Data file
.bmp	Binary	Bit-map image file
.c	ASCII text	C program source file
.cmd	ASCII text	C55xx linker command file
.dat	ASCII text	Data file or parameter file
.exe	Binary	Microsoft Visual C IDE executable file
.fig	Binary	MATLAB® figure M-file
.h	ASCII text	C program header file
.inc	ASCII text	C55xx assembly program include file
.jpg	Binary	JPEG image file
.lib	Binary	C55xx CCS runtime support library
.m	ASCII text	MATLAB® script file
.map	ASCII text	C55xx linker generated memory map file
.mp3	Binary	MP3 audio file
.out	Binary	C55xx linker generated executable file
.pcm	Binary	Linear PCM data file
.txt	ASCII text	ASCII text file
.wav	Binary	Microsoft linear PCM wave file
.yuv	Binary	YUV or YC_bC_r image file

Table B.2 List of experiments for the book

Chapter	Experiment	Experiment project	Experiment purpose
1	Exp1.1	CCS_eZdsp	Getting started with CCS and eZdsp
	Exp1.2	fileIO	Getting familiar with C file I/O functions
	Exp1.3	userInterface	Introduce user interface for CCS and eZdsp
	Exp1.4	playTone	Real-time audio playback using eZdsp
	Exp1.5	audioLoop	Real-time audio loopback using eZdsp
2	Exp2.1	overflow	Overflow and saturation arithmetic
	Exp2.2	funcAppro: A_floatingPointC	Compute cosine using floating-point C program
		funcAppro: B_fixedPointC	Compute cosine using fixed-point C program
		funcAppro: C_c55xASM	Compute cosine using assembly implementation
		funcAppro: D_design4DSP	Compute sine and cosine using assembly implementation
	Exp2.3	signalGen: A_floatingPointC	Real-time tone and noise generation using eZdsp: floating-point C program
		signalGen: B_toneGen	Real-time tone generation using eZdsp: assembly implementation
		signalGen: C_randGenC	Real-time noise generation using eZdsp: fixed-point C implementation

Table B.2 (*Continued*)

Chapter	Experiment	Experiment project	Experiment purpose
		signalGen: D_randGen	Real-time noise generation using eZdsp: assembly implementation
		signalGen: E_signalGen	Real-time tone and noise generation using eZdsp: assembly implementation
3	Exp3.1	fixedPoint_BlockFIR	FIR filtering using fixed-point C
	Exp3.2	asm_BlockFIR	FIR filtering using C55xx assembly program
	Exp3.3	symmetric_BlockFIR	Symmetric FIR filtering using C55xx assembly program
	Exp3.4	dualMAC_BlockFIR	Optimization using dual-MAC architecture
	Exp3.5	realtimeFIR	Real-time FIR filtering
	Exp3.6	decimation	Decimation using C and assembly programs
	Exp3.7	interpolation	Interpolation using fixed-point C
	Exp3.8	SRC	Sampling rate conversion
	Exp3.9	realtimeSRC	Real-time sampling rate conversion
4	Exp4.1	floatPoint_directIIR	Direct-form I IIR filter using floating-point C
	Exp4.2	fixedPoint_directIIR	Direct-form I IIR filter using fixed-point C
	Exp4.3	fixedPoint_cascadeIIR	Cascade IIR filter using fixed-point C
	Exp4.4	intrisics_implementation	Cascade IIR filter using intrinsics
	Exp4.5	asm_implementation	Cascade IIR filter using assembly program
	Exp4.6	realtimeIIR	Real-time IIR filtering
	Exp4.7	parametric_equalizer	Parametric equalizer using fixed-point C
	Exp4.8	realtimeEQ	Real-time parametric equalizer
5	Exp5.1	floatingPoint_DFT	DFT using floating-point C
	Exp5.2	asm_DFT	DFT using C55xx assembly program
	Exp5.3	floatingPoint_FFT	FFT using floating-point C
	Exp5.4	intrinsics_FFT	FFT using fixed-point C with intrinsics
	Exp5.5	FFT_iFFT	Experiment of FFT and IFFT
	Exp5.6	hwFFT	FFT using C55xx hardware accelerator
	Exp5.7	realtime_hwFFT	Real-time FFT using C55xx hardware accelerator
	Exp5.8	fastconvolution	Fast convolution using overlap–add technique
	Exp5.9	realtime_hwfftConv	Real-time fast convolution
6	Exp6.1	floatingPoint_LMS	LMS algorithm using floating-point C
	Exp6.2	fixPoint_LeakyLMS	Leaky LMS algorithm using fixed-point C
	Exp6.3	intrinsic_NLMS	Normalized LMS algorithm using fixed-point C and intrinsics
	Exp6.4	asm_dlms	Delayed LMS algorithm using assembly program
	Exp6.5	system_identificaiton	Experiment of adaptive system identification
	Exp6.6	adaptive_predictor	Experiment of adaptive predictor
	Exp6.7	channel_equalizer	Experiment of adaptive channel equalizer
	Exp6.8	realtime_predictor	Real-time adaptive prediction using eZdsp
7	Exp7.1	sineGen	Sine wave generator using table lookup
	Exp7.2	sirenGen	Siren generator using table lookup
	Exp7.3	dtmfGen	DTMF generator
	Exp7.4	dtmfDetect	DTMF detection using fixed-point C

(*continued*)

Table B.2 (*Continued*)

Chapter	Experiment	Experiment project	Experiment purpose
	Exp7.5	`asmDTMFDet`	DTMF detection using assembly program
8	Exp8.1	`floatingPointAec`	Acoustic echo canceler using floating-point C
	Exp8.2	`intrinsicAec`	Acoustic echo canceler using fixed-point C with intrinsics
	Exp8.3	`floatPointAECNR`	Integration of acoustic echo cancellation and noise reduction
9	Exp9.1	`LPC`	LPC filter using fixed-point C with intrinsics
	Exp9.2	`PWF`	Perceptual weighting filter using fixed-point C with intrinsics
	Exp9.3	`floatPointVAD`	Voice activity detection using floating-point C
	Exp9.4	`mix_VAD`	Voice activity detection using fixed-point C
	Exp9.5	`VAD.DTX.SID`	Speech encoder with discontinuous transmission
	Exp9.6	`CNG`	Speech decoder with comfort noise generation
	Exp9.7	`floatPointNR`	Spectral subtraction using floating-point C
	Exp9.8	`G722`	G.722.2 using fixed-point C
	Exp9.9	`G711`	G.711 companding using fixed-point C
	Exp9.10	`realtime_G711`	Real-time G.711 audio loopback
10	Exp10.1	`floatingPointMdct`	MDCT using floating-point C
	Exp10.2	`intinsicMdct`	MDCT using fixed-point C and intrinsics
	Exp10.3	`preEcho`	Pre-echo effects
	Exp10.4	`isoMp3Dec`	MP3 decoding using floating-point C
	Exp10.5	`realtime_parametricEQ`	Real-time parametric equalizer using eZdsp
	Exp10.6	`flanger`	Flanger effects
	Exp10.7	`realtime_flanger`	Real-time flanger effects using eZdsp
	Exp10.8	`tremolo`	Tremolo effects
	Exp10.9	`realtime_tremolo`	Real-time tremolo effects using eZdsp
	Exp10.10	`spatial`	Spatial sound effects
	Exp10.11	`realtime_spatial`	Real-time spatial effects using eZdsp
11	Exp11.1	`YCbCr2RGB`	YC_bC_r to RGB conversion
	Exp11.2	`whiteBalance`	White balance
	Exp11.3	`gammaContrast`	Gamma correction and contrast adjustment
	Exp11.4	`2DFilter`	Image filtering
	Exp11.5	`DCT`	DCT and IDCT
	Exp11.6	`fingerPrint`	Image processing for fingerprints
	Exp11.7	`2DWavelet`	2-D wavelet transform
App. C	ExpC.1	`appC_examples`	Examples
	ExpC.2	`assembly`	Assembly program
	ExpC.3	`multiply`	Multiplication
	ExpC.4	`loop`	Loops
	ExpC.5	`modulo`	Modulo operator
	ExpC.6	`c_assembly`	Use mixed C-and-assembly programs
	ExpC.7	`audioPlayback`	Working with AIC3204
	ExpC.8	`audioExp`	Analog input and output

The `src` directory has all the source programs including `.c` and `.asm` programs. Except for the `Debug` directory, the `data`, `inc`, and `src` directories are user-created folders. Besides the C and assembly programs, there are other files used by the book. Table B.1 lists the file types and formats used by the book with a brief description.

Table B.2 lists the experiments provided by the book. These experiments are primarily designed for use with the C5505 eZdsp; however, very limited experiments are conducted based on the available software package of some standards, and these software packages can be compiled and run on personal computers.

Appendix C

Introduction to the TMS320C55xx Digital Signal Processor

This appendix introduces the basic architecture and assembly and C programming for the Texas Instruments TMS320C55xx fixed-point processor family.

C.1 Introduction

The TMS320C55xx fixed-point processor family consists of a variety of digital signal processors including C5501, C5502, C5503, C5505, C5509, C5510, C5515, and so on. The C55xx processor delivers optimum performance with balanced low-power consumption and high code density. Its dual multiply-and-accumulate (MAC) architecture doubles the cycle efficiency for computing the inner product of vectors (e.g., FIR filtering) and its scalable instruction set provides good code density. In this appendix, we introduce the low-cost C5505 eZdsp USB stick [1] and use it for experiments.

C.2 TMS320C55xx Architecture

The C55xx central processing unit (CPU) [2] consists of four processing units, namely, the instruction buffer unit (IU), program flow unit (PU), address-data flow unit (AU), and data computation unit (DU), together with 12 different address and data buses connected to these units as shown in Figure C.1.

C.2.1 Architecture Overview

The instruction buffer unit (IU) fetches instructions from the memory into the CPU. C55xx instructions have different lengths for optimum code density. Simple instructions use only 8 bits (1 byte) while complicated instructions may contain as many as 48 bits (6 bytes). For each clock cycle, the IU fetches 4 bytes of instruction code via its 32-bit program-read data bus and places them into the 64-byte instruction buffer queue. At the same time, the instruction decoder decodes the instruction and passes it to the PU, AU, or DU as shown in Figure C.2.

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

24-bit program-read address bus (PAB)

32-bit program-read data bus (PB)

Three 24-bit data-read address buses (BAB, CAB, DAB)

Three 16-bit data-read data buses (BB, CB, DB)

32 bits

CB DB

BB CB DB

Instruction buffer unit IU

Program flow unit PU

Address-data flow unit AU

Data computation unit DU

C55xx CPU

Two 16-bit data-write data buses (EB, FB)

Two 24-bit data-write address buses (EAB, FAB)

Figure C.1 Block diagram of the TMS320C55xx CPU

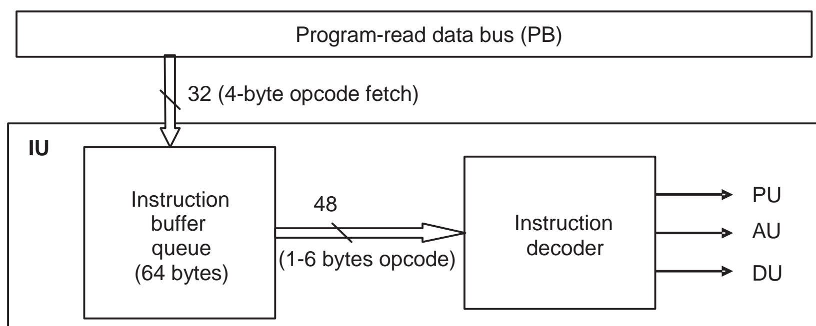


Figure C.2 Simplified block diagram of the C55xx IU

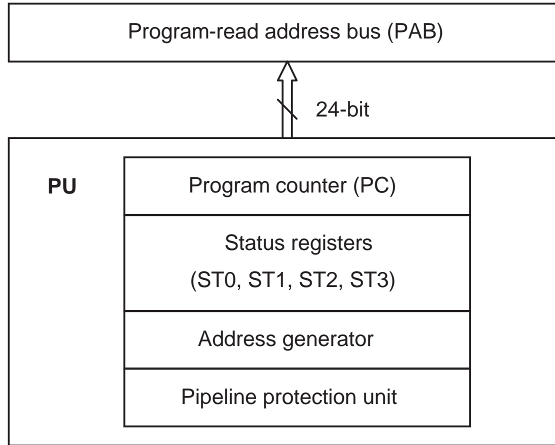


Figure C.3 Simplified block diagram of the C55xx PU

The IU improves program execution by maintaining the instruction flow between the four units within the CPU. If the IU is able to hold a complete loop segment code inside the buffer queue, the program can be repeated many times without fetching new instructions from memory. Such capability improves the efficiency of loop execution.

The program flow unit (PU) manages program execution. As illustrated in Figure C.3, the PU consists of a program counter (PC), four status registers, a program address generator, and a pipeline protection unit. The PC tracks program execution every clock cycle. The program address generator produces a 24-bit address that accesses 16 megabytes (Mbytes) of memory space. Since most instruction executions will be sequential, the C55xx utilizes a pipeline architecture to improve its execution efficiency. Instructions such as branch, call, return, conditional execution, and interrupt will break sequential program execution. The pipeline protection unit prevents program flow from any pipeline vulnerabilities from non-sequential execution.

The address-data flow unit (AU) serves as the data access manager. The block diagram illustrated in Figure C.4 shows that the AU generates the data-space addresses for data read and data write. The AU consists of eight 23-bit extended auxiliary registers (XAR0–XAR7), four 16-bit temporary registers (T0–T3), a 23-bit coefficient pointer (XCDP), and a 23-bit stack pointer (XSP). It has an additional 16-bit ALU for simple arithmetic operations. The AU allows up to two address registers and a coefficient pointer working together to access two data samples and one coefficient simultaneously in one clock cycle. The AU also supports up to five circular buffers.

The data computation unit (DU) handles intensive computations for C55xx applications. As illustrated in Figure C.5, the DU consists of a pair of MAC units, a 40-bit ALU, four 40-bit accumulators (AC0, AC1, AC2, and AC3), a barrel shifter, and rounding and saturation control logic. Three data-read data buses allow two data paths and a coefficient path accessed by dual-MAC units simultaneously. In one clock cycle, each MAC unit can perform a 17-bit by 17-bit multiplication plus a 40-bit addition (or subtraction) with saturation option. The ALU can perform 40-bit arithmetic, logic, rounding, and saturation operations using these

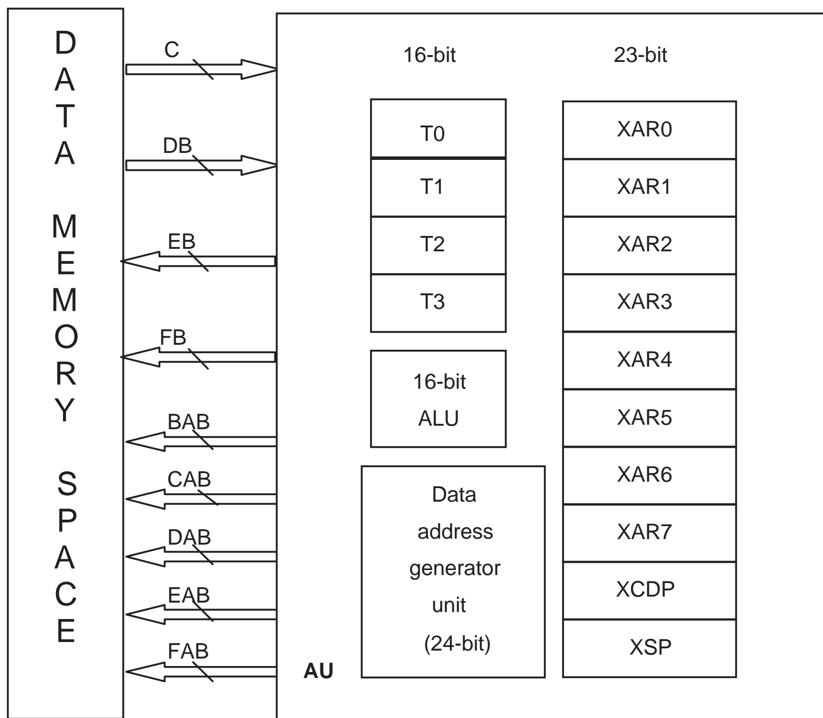


Figure C.4 Simplified block diagram of the C55xx AU

accumulators. It can also perform two 16-bit arithmetic operations in both the upper and lower portions of an accumulator at the same time. The ALU can accept immediate values from the IU as data and communicate with other AU and PU registers. The barrel shifter can perform data shift in the range of 2^{-32} (shift right 32 bits) to 2^{31} (shift left 31 bits).

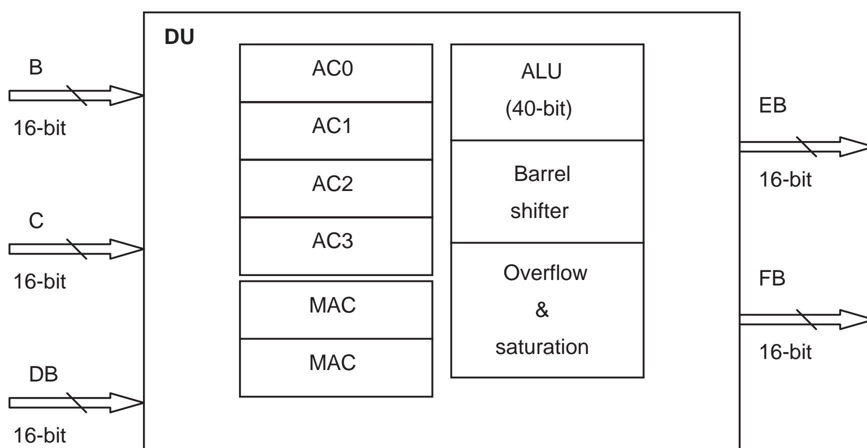


Figure C.5 Simplified block diagram of the C55xx DU

C.2.2 On-Chip Memories

The C55xx uses unified program and data memory configurations with a separated I/O space. All 16Mbytes of memory space can be used for both program and data. The memory-mapped registers (MMRs) also reside in data memory space. The processor fetches instructions from the program memory space using the program-read address bus in the 8-bit byte unit and accesses the data memory space as a 16-bit word entity. As shown in Figure C.6, the 16Mbyte memory map consists of 128 data pages (0–127), and each page has 128Kbytes (64Kwords).

The C55xx on-chip memory from addresses 0x0000 to 0xFFFF uses dual-access RAM (DARAM). The 64Kbyte DARAM consists of eight 8Kbyte blocks, with address ranges as given in Table C.1. DARAM allows C55xx to perform two accesses (two reads, two writes, or one read and one write) per cycle.

The C55xx on-chip memory also includes single-access RAM (SARAM). The SARAM location starts from the byte address 0x10000 to 0x4FFFF, and consists of 32 blocks of 8Kbytes each, as shown in Table C.2. At each clock cycle, SARAM allows only single access (one read or one write).

	Data space addresses (word in hexadecimal)	C55xx memory program/data space	Program space addresses (byte in hexadecimal)
Page 0	MMRs 00 0000-00 005F		00 0000-00 00BF Reserved
	00 0060 00 FFFF		00 00C0 01 FFFF
Page 1	01 0000 01 FFFF		02 0000 03 FFFF
Page 2	02 0000 02 FFFF		04 0000 05 FFFF
	⋮	⋮	⋮
Page 127	7F 0000 7F FFFF		FE 0000 FF FFFF

Figure C.6 TMS320C55xx memory map for program space and data space

Table C.1 C55xx DARAM blocks and addresses

DARAM byte address range	DARAM memory blocks
0x0000–0x1FFF	DARAM 0
0x2000–0x3FFF	DARAM 1
0x4000–0x5FFF	DARAM 2
0x6000–0x7FFF	DARAM 3
0x8000–0x9FFF	DARAM 4
0xA000–0xBFFF	DARAM 5
0xC000–0xDFFF	DARAM 6
0xE000–0xFFFF	DARAM 7

Table C.2 C55xx SARAM blocks and addresses

SARAM byte address range	SARAM memory blocks
0x10000–0x11FFF	SARAM 0
0x12000–0x13FFF	SARAM 1
0x14000–0x15FFF	SARAM 2
:	:
:	:
:	:
0x4C000–0x4DFFF	SARAM 30
0x4E000–0x4FFFF	SARAM 31

C.2.3 Memory-Mapped Registers

The C55xx processor uses MMRs for internal managing, controlling, operating, and monitoring. These MMRs are located in the reserved RAM block from 0x00000 to 0x005F. Table C.3 lists all the TMS320C5505 CPU registers with their MMR addresses and function descriptions.

The accumulators AC0, AC1, AC2, and AC3 are 40-bit registers. Each accumulator consists of two 16-bit registers and one 8-bit register as shown in Figure C.7. The guard bits, AG, hold a data result that is more than 32 bits to prevent overflow during accumulation.

The temporary data registers, T0, T1, T2, and T3, are 16-bit registers. They can hold data less than or equal to 16 bits. There are eight auxiliary registers, AR0–AR7, for several purposes such as data pointers for indirect addressing mode and circular addressing mode. The coefficient data pointer is a unique addressing register used for accessing coefficients via the coefficient data bus during multiple data access operations. The stack pointer tracks the data-memory address position at the top of the stack. Auxiliary registers, coefficient pointer

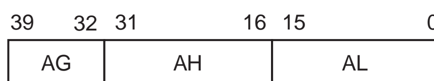


Figure C.7 TMS320C55xx accumulator structure

Table C.3 C5505 MMRs

Reg.	Addr.	Function description	Reg.	Addr.	Function description
IER0	0×00	Interrupt mask register 0	AC3L	0×28	Accumulator 3 [15 0]
IFR0	0×01	Interrupt flag register 0	AC3H	0×29	Accumulator 3 [31 16]
ST0_55	0×02	Status register 0 for C55xx	AC3G	0×2A	Accumulator 3 [39 32]
ST1_55	0×03	Status register 1 for C55xx	DPH	0×2B	Extended data-page pointer
ST3_55	0×04	Status register 3 for C55xx		0×2C	Reserved
	0×05	Reserved		0×2D	Reserved
ST0	0×06	ST0 (for 54x compatibility)	DP	0×2E	Memory data-page start address
ST1	0×07	ST1 (for 54x compatibility)	PDP	0×2F	Peripheral data-page start address
AC0L	0×08	Accumulator 0 [15 0]	BK47	0×30	Circular buffer size register for AR[4–7]
AC0H	0×09	Accumulator 0 [31 16]	BKC	0×31	Circular buffer size register for CDP
AC0G	0×0A	Accumulator 0 [39 32]	BSA01	0×32	Circular buffer start addr. reg. for AR[0–1]
AC1L	0×0B	Accumulator 1 [15 0]	BSA23	0×33	Circular buffer start addr. reg. for AR[2–3]
AC1H	0×0C	Accumulator 1 [31 16]	BSA45	0×34	Circular buffer start addr. reg. for AR[4–5]
AC1G	0×0D	Accumulator 1 [39 32]	BSA67	0×35	Circular buffer start addr. reg. for AR[6–7]
T3	0×0E	Temporary register 3	BSAC	0×36	Circular buffer coefficient start addr. reg.
TRN0	0×0F	Transition register	BIOS	0×37	Data-page ptr storage for 128-word data table
AR0	0×10	Auxiliary register 0	TRN1	0×38	Transition register 1
AR1	0×11	Auxiliary register 1	BRC1	0×39	Block repeat counter 1
AR2	0×12	Auxiliary register 2	BRS1	0×3A	Block repeat save 1
AR3	0×13	Auxiliary register 3	CSR	0×3B	Computed single repeat
AR4	0×14	Auxiliary register 4	RSA0H	0×3C	Repeat start address 0 high
AR5	0×15	Auxiliary register 5	RSA0L	0×3D	Repeat start address 0 low
AR6	0×16	Auxiliary register 6	REA0H	0×3E	Repeat end address 0 high
AR7	0×17	Auxiliary register 7	REA0L	0×3F	Repeat end address 0 low
SP	0×18	Stack pointer register	RSA1H	0×40	Repeat start address 1 high
BK03	0×19	Circular buffer size register	RSA1L	0×41	Repeat start address 1 low
BRC0	0×1A	Block repeat counter	REA1H	0×42	Repeat end address 1 high
RSA0L	0×1B	Block repeat start address	REA1L	0×43	Repeat end address 1 low
REA0L	0×1C	Block repeat end address	RPTC	0×44	Repeat counter
PMST	0×1D	Processor mode status register	IER1	0×45	Interrupt mask register 1

Table C.3 (*Continued*)

Reg.	Addr.	Function description	Reg.	Addr.	Function description
	0×1E	Reserved	IFR1	0×46	Interrupt flag register 1
	0×1F	Reserved	DBIER0	0×47	Debug IER0
T0	0×20	Temporary data register 0	DBIER1	0×48	Debug IER1
T1	0×21	Temporary data register 1	IVPD	0×49	Interrupt vector pointer, DSP
T2	0×22	Temporary data register 2	IVPH	0×4A	Interrupt vector pointer, HOST
T3	0×23	Temporary data register 3	ST2_55	0×4B	Status register 2 for C55xx
AC2L	0×24	Accumulator 2 [15 0]	SSP	0×4C	System stack pointer
AC2H	0×25	Accumulator 2 [31 16]	SP	0×4D	User stack pointer
AC2G	0×26	Accumulator 2 [39 32]	SPH	0×4E	Extended data-page pointer for SP and SSP
CDP	0×27	Coefficient data pointer	CDPH	0×4F	Main data-page pointer for the CDP

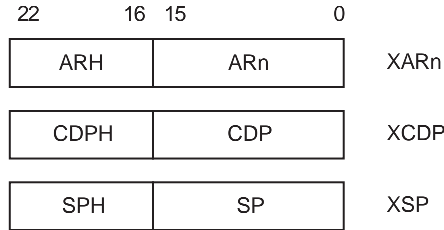


Figure C.8 TMS320C55xx 23-bit memory-mapped registers.

registers, and stack pointer registers are all 23-bit registers as shown in Figure C.8. The data in the lower 16-bit portion will not carry into the higher 7-bit portion of the register.

The C55xx processor has four system status registers, ST0_C55, ST1_C55, ST2_C55, and ST3_C55. These registers contain system control bits and flag bits. The control bits directly affect the operational conditions of the C55xx. The flag bits report the status of the processor. Figure C.9 shows the names and positions of these bits.

C.2.4 Interrupts and Interrupt Vector

The C55xx has the interrupt vector shown in Table C.4 to serve all the internal and external interrupts. The interrupt vector lists the functions and priorities of all internal and external interrupts for the C5505. The addresses of the interrupts are the offsets from the interrupt vector pointer.

We use the interrupt enable registers IER0 and IER1 to enable (unmask) or disable (mask) the interrupts. The interrupt flag registers, IFR0 and IFR1, indicate if an interrupt has occurred. The interrupt enable bit and flag bit assignments are illustrated in Figure C.10. When a flag bit of the IFR is set to 1, it indicates an interrupt has happened and that interrupt is pending to be served.

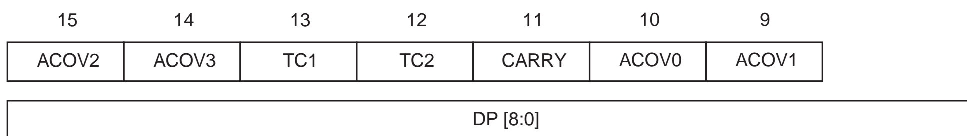
C.3 TMS320C55xx Addressing Modes

The C55xx can access 16Mbytes of memory space using the following six addressing modes:

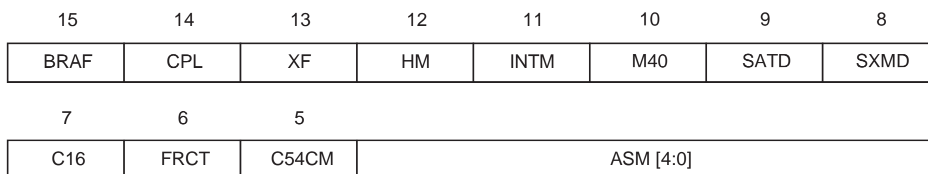
- Direct addressing mode.
- Indirect addressing mode.
- Absolute addressing mode.
- Memory-mapped register addressing mode.
- Register bits addressing mode.
- Circular addressing mode.

These addressing modes are very important for assembly language programming. When a program is written in C, the C compiler will generate the proper addressing mode. However, the compiler cannot generate all the addressing modes automatically and efficiently. The circular addressing mode is an example of using the processor efficiently. When programming in assembly language, we can effectively use circular buffers with the circular addressing mode. Table C.5 uses the move (`mov`) instruction with different syntaxes to show the C55xx addressing modes.

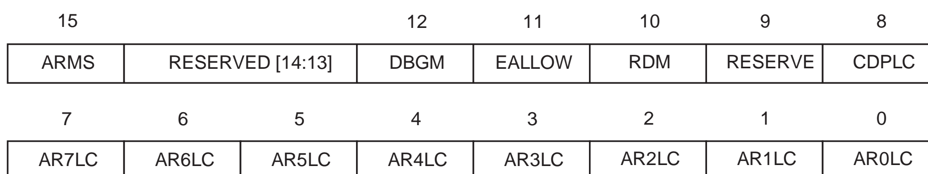
ST0_55:



ST1_55:



ST2_55:



ST3_55:

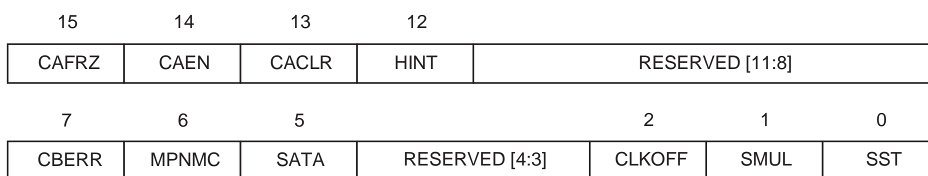
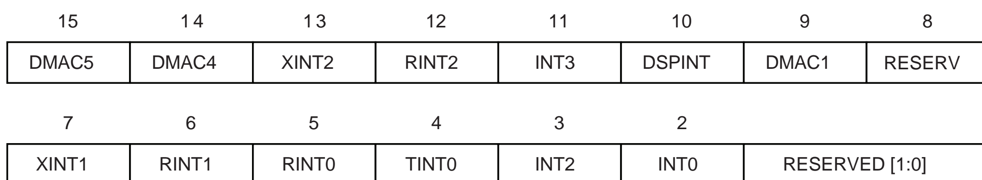


Figure C.9 TMS320C55xx status registers

IFR0/IER0:



IFR1/IER1:

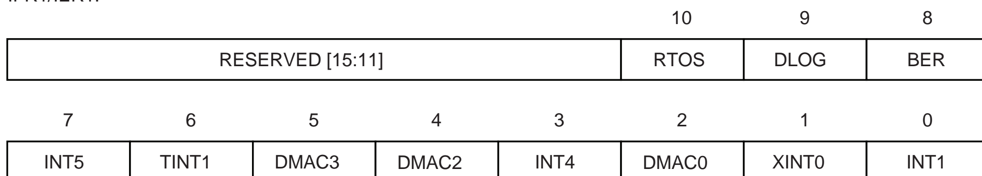


Figure C.10 TMS320C55xx interrupt enable and flag registers

Table C.4 C5505 interrupt vector

Name	Offset	Priority	Function description
RESET	0×00	0	Reset (hardware and software)
MNI	0×08	1	Non-maskable interrupt
INT0	0×10	3	External interrupt #0
INT2	0×18	5	External interrupt #2
TINT0	0×20	6	Timer #0 interrupt
RINT0	0×28	7	McBSP #0 receive interrupt
RINT1	0×30	9	McBSP #1 receive interrupt
XINT1	0×38	10	McBSP #1 transmit interrupt
SINT8	0×40	11	Software interrupt #8
DMAC1	0×48	13	DMA channel #1 interrupt
DSPINT	0×50	14	Interrupt from host
INT3	0×58	15	External interrupt #3
RINT2	0×60	17	McBSP #2 receive interrupt
XINT2	0×68	18	McBSP #2 transmit interrupt
DMAC4	0×70	21	DMA channel #4 interrupt
DMAC5	0×78	22	DMA channel #5 interrupt
INT1	0×80	4	External interrupt #1
XINT0	0×88	8	McBSP #0 transmit interrupt
DMAC0	0×90	12	DMA channel #0 interrupt
INT4	0×98	16	External interrupt #4
DMAC2	0×A0	19	DMA channel #2 interrupt
DMAC3	0×A8	20	DMA channel #3 interrupt
TINT1	0×B0	23	Timer #1 interrupt
INT5	0×B8	24	External interrupt #5
BERR	0×C0	2	Bus error interrupt
DLOG	0×C8	25	Data log interrupt
RTOS	0×D0	26	Real-time operating system interrupt
SINT27	0×D8	27	Software interrupt #27
SINT28	0×E0	28	Software interrupt #28
SINT29	0×E8	29	Software interrupt #29
SINT30	0×F0	30	Software interrupt #30
SINT31	0×F8	31	Software interrupt #31

As illustrated in Table C.5, each addressing mode uses following one or more operands:

- `Smem` is a short data word (16-bit) from data memory, I/O memory, or MMRs.
- `Lmem` is a long data word (32-bit) from either data memory or MMRs.
- `Xmem` and `Ymem` are used by an instruction to perform two 16-bit data memory accesses simultaneously.
- `src` and `dst` are source and destination registers, respectively.
- `#k` is a signed immediate constant, for example, `#k16` is a 16-bit constant ranging from −32 768 to 32 767.
- `dbl` is a memory qualifier for memory access for a long data word.
- `xdst` is an extended register (23-bit).

Table C.5 C55xx `mov` instruction with different operand forms

Instruction	Description
1. `mov #k,dst`	Load the 16-bit signed constant `k` to the destination register `dst`
2. `mov src,dst`	Load the content of source register `src` to the destination register `dst`
3. `mov Smem,dst`	Load the content of memory location `Smem` to the destination register `dst`
4. `mov Xmem,Ymem,ACx`	The content of `Xmem` is loaded into the lower part of `ACx`, while the content of `Ymem` is sign extended and loaded into upper part of `ACx`
5. `mov dbl(Lmem),pair(TAx)`	Load upper 16-bit data and lower 16-bit data from `Lmem` to `TAx` and `TA(x+1)`, respectively
6. `amov #k23,xdst`	Load the effective address of `k23` (23-bit constant) into the extended destination register (`xdst`)

C.3.1 Direct Addressing Modes

There are four types of direct addressing modes: data-page pointer (DP) direct, stack pointer (SP) direct, register bits direct, and peripheral data-page pointer (PDP) direct.

The DP-direct addressing mode uses the main data page specified by the 23-bit extended data-page pointer (XDP). As shown in Figure C.11, the upper 7 bits of DPH determines the main data page (0–127), and the lower 16 bits of DP defines the starting address in the data page pointed by DPH. The instruction contains a 7-bit offset (`@x`) that directly points to the variable `x(Smem)` in the data page. The data-page registers DPH, DP, and XDP can be loaded by the `mov` instruction as

```
mov  #k7,DPH   ; Load DPH with a 7-bit constant k7
mov  #k16,DP   ; Load DP with a 16-bit constant k16
```

The first instruction loads the high portion of the extended data-page pointer, DPH, with a 7-bit constant `k7` to set up the main data page. The second instruction initializes the starting address of the data-page pointer. Example C.1 shows how to initialize the DPH and DP pointers.

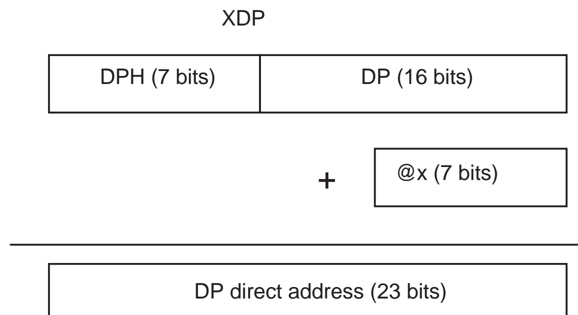
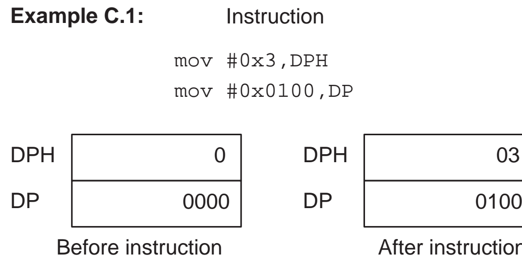


Figure C.11 Using the DP-direct addressing mode to access variable x



The XDP can also be initialized in one instruction using a 23-bit constant as

```
amov   #k23,XDP   ; Load XDP with a 23-bit constant
```

The syntax used in the assembly code is `amov #k23,xdst`, where `#k23` is a 23-bit address, and the destination `xdst` is an extended register. For example, `amov #14000,XDP` initializes the XDP to data page 1 with starting address `0x4000`.

The following code shows how to use the DP-direct addressing mode:

```
X    .set   0x1FFEF
     mov    #0x1,DPH          ; Load DPH with 1
     mov    #0x0FFEF,DP       ; Load DP with starting address
     .dp    X
     mov    #0x5555,@X        ; Store 0x5555 to memory location X
     mov    #0xFFFF,@(X+5)    ; Store 0xFFFF to memory location X+5
```

In this example, the symbol `@` tells the assembler that this instruction uses the direct addressing mode. The directive `.dp` indicates the base address of the variable `X` without using memory space.

The SP-direct addressing mode is similar to the DP-direct addressing mode. The 23-bit address can be formed with the extended stack pointer (XSP) in the same way as XDP. The upper 7 bits (SPH) select the main data page and the lower 16 bits (SP) determine the starting address. The 7-bit stack offset is contained in the instruction. When SPH = 0 (main page 0), the stack must avoid using the reserved memory space for MMRs from address 0 to 0x5F.

The I/O-space addressing mode only has a 16-bit addressing range. The 512 peripheral data pages can be selected by the upper 9 bits of the PDP register. The 7-bit offset in the lower portion of the PDP register determines the location inside the selected peripheral data page as illustrated in Figure C.12.

C.3.2 Indirect Addressing Modes

There are four types of indirect addressing modes. The AR-indirect mode uses one of the eight auxiliary registers as a pointer to data memory, I/O space, and MMRs. The dual AR-indirect mode uses two auxiliary registers for dual data memory accesses. The coefficient data pointer (CDP) indirect mode uses the CDP for pointing to coefficients in data memory space. The

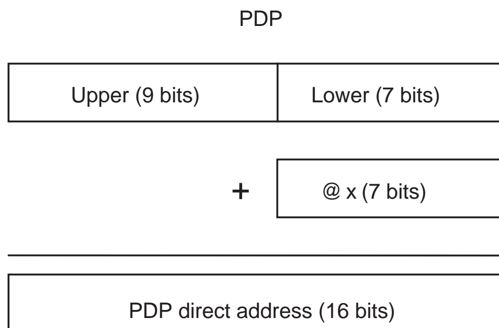


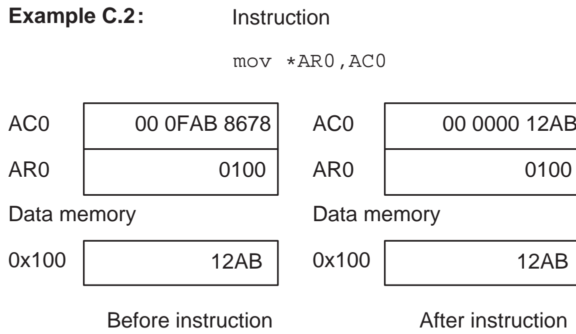
Figure C.12 Using PDP-direct addressing mode to access variable x

coefficient dual AR-indirect mode uses the CDP and the dual AR-indirect modes to generate three addresses. The indirect addressing is the most frequently used in addressing mode since it supports pointer updates and modification schemes as listed in Table C.6.

The AR-indirect addressing mode uses auxiliary registers (AR0–AR7) to point to data memory space. The upper 7 bits of the extended auxiliary register (XAR) points to the main data page while the lower 16 bits points to the data location in that page. Since the I/O-space address is limited to a 16-bit range, the upper portion of the XAR must be set to zero when accessing I/O space. The maximum block size (32Kwords) of the indirect addressing mode is limited by using 16-bit auxiliary registers. Example C.2 uses the indirect addressing mode to copy the data stored in data memory pointed at by AR0 to the destination register AC0.

Table C.6 AR- and CDP-indirect addressing pointer modification schemes

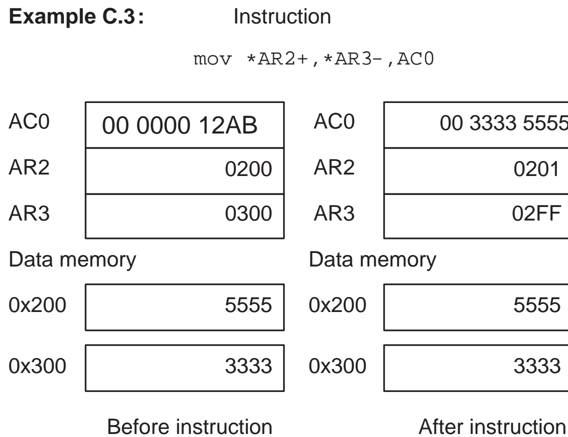
Operand	ARn/CDP pointer modifications
`*ARn` or `*CDP`	`ARn` (or `CDP`) is not modified
`*ARn±` or `*CDP±`	`ARn` (or `CDP`) is modified after the operation by: ±1 for 16-bit operation (`ARn=ARn±1`) ±2 for 32-bit operation (`ARn=ARn±2`)
`*ARn(#k16)` or `*CDP(#k16)`	`ARn` (or `CDP`) is not modified The signed 16-bit constant `k16` is used as the offset from the base pointer `ARn` (or `CDP`)
`*+ARn(#k16)` or `*+CDP(#k16)`	`ARn` (or `CDP`) is modified before the operation The signed 16-bit constant `k16` is added as the offset to the base pointer `ARn` (or `CDP`) before generating a new address
`*(ARn±T0/T1)`	`ARn` is modified after the operation by ±16-bit content in `T0` or `T1`, (`ARn=ARn±T0/T1`)
`*ARn(T0/T1)`	`ARn` is not modified `T0` or `T1` is used as the offset for the base pointer `ARn`



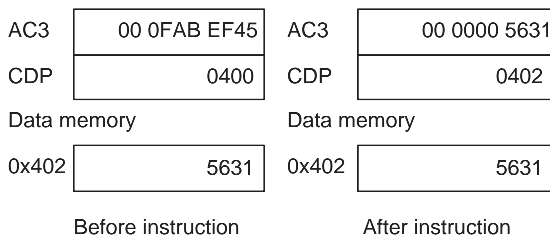
The dual-AR indirect addressing mode allows two data-memory accesses using the auxiliary registers. It can access two 16-bit data in memory using the syntax given in Table C.5 as follows:

```
mov Xmem,Ymem,ACx
```

Example C.3 shows how to perform two 16-bit data loads using AR2 and AR3 as data pointers to the `Xmem` and `Ymem`, respectively. The data pointed at by AR3 is sign extended to 24 bits, and loaded into the upper portion of the destination accumulator `AC0(39:16)`, while the data pointed at by AR2 is loaded into the lower portion of `AC0(15:0)`. The data pointers AR2 and AR3 are also updated.

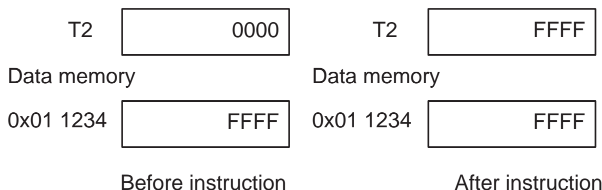


The extended coefficient data pointer (XCDP) is the concatenation of the CDPH (the upper 7 bits) and the CDP (the lower 16 bits). The CDP-indirect addressing mode uses the upper 7 bits to define the main data page and the lower 16 bits to point to the memory location within the specified data page. Example C.4 shows how to use the CDP-indirect addressing mode where CDP contains the address of the coefficient in data memory. This instruction first increases the CDP pointer by 2, then loads the coefficient pointed by the updated coefficient pointer to the destination register AC3.



C.3.3 Absolute Addressing Modes

The memory can also be addressed using the `k16` or `k23` absolute addressing mode. The `k23` absolute mode specifies an address as a 23-bit unsigned constant. Example C.5 shows an example of loading the data content at address `0x1234` on the main data page `1` into the temporary register T2, where the symbol `*()` indicates the use of the absolute addressing mode.



The `k16` absolute addressing mode uses the operand `*abs(#k16)`, where `k16` is a 16-bit unsigned constant. In this mode, the DPH (7 bits) is forced to zero and concatenated with the unsigned constant `k16` to form a 23-bit data-space memory address. The I/O absolute addressing mode uses the operand `port(#k16)`. The absolute address can also have the variable name such as the variable `x` in the following example:

```
mov *(x),AC0
```

This instruction loads the accumulator AC0 with the content of variable `x`. When using the absolute addressing mode, we do not need to worry about the data-page pointer. The drawback is that it needs more code space to represent a 23-bit address.

C.3.4 MMR Addressing Mode

The absolute, direct, and indirect addressing modes can address MMRs located in the data memory from address 0x0 to 0x5F on the main data page 0 as shown in Figure C.6. For example, the instruction `mov *(#AR2),T2` uses the absolute addressing mode to load the 16-bit content of AR2 into the temporary register T2.

For the MMR-direct addressing mode, the DP-direct addressing mode must be selected. Example C.6 shows how to use the direct addressing mode to load the content of the lower portion of the accumulator AC0(15:0) into the temporary register T2. When the `mmap()` qualifier is used for the MMR-direct addressing mode, it forces the data address generator to access the main data page 0. That is, XDP = 0.

Example C.6: Instruction

```
mov mmap(@AC0L),T2
```

Before instruction		After instruction	
AC0	00 3333 5555	AC0	00 3333 5555
T2	0201	T2	5555

Accessing the MMRs using the indirect addressing mode is the same as addressing the data memory space. Since the MMRs are located in data page 0, the XAR and XCDP must be initialized to page 0 by setting the upper 7 bits to zero. The following instructions load the content of AC0 into T1 and T2 registers:

```
amov   #AC0H,XAR6
mov    *AR6-,T2
mov    *AR6+,T1
```

In this example, the first instruction loads the effective address of the upper portion of the accumulator AC0 (AC0H, located at address 0x9 on page 0) to the extended auxiliary register XAR6. That is, XAR6 = 0x000009. The second instruction uses AR6 as a pointer to copy the content of AC0H into the T2 register, and then decrements the pointer by 1 to point to the lower portion of AC0 (AC0L, located at address 0x8). Thus, the third instruction copies the content of AC0L into the register T1 and modifies AR6 to point to AC0H again.

C.3.5 Register Bits Addressing Mode

Both direct and indirect addressing modes can address a bit or a pair of bits in a specific register. The direct addressing mode uses a bit offset to access a particular register's bit. The offset is the number of bits counting from the LSB. The instruction of the registers bit direct addressing mode is shown in Example C.7. The bit test instruction `btstp` will update the test condition bits (TC1 and TC2) of the status register ST0 (bit 12 and bit 13).

Example C.7: Instruction

```
btstp  @28,AC0
```

Before instruction		After instruction	
AC0	00 3333 5555	AC0	00 3333 5555
ST0	0802	ST0	3802

We also can use the indirect addressing modes to specify register bit(s) as follows:

```
mov     #2,AR4    ; AR4 contains the bit offset 2
bset    *AR4,AC3  ; Set the AC3 bit pointed by AR4 to 1
btstp   *AR4,AC1  ; Test AC1 bit pair pointed by AR4
```

The register bits addressing mode supports only the bit test, bit set, bit clear, and bit complement instructions in conjunction with the accumulators (AC0–AC3), auxiliary registers (AR0–AR7), and temporary registers (T0–T3).

C.3.6 Circular Addressing Mode

The circular addressing mode updates data pointers in modulo fashion for continuously accessing data buffers without resetting the pointers. When the pointer reaches the end of the buffer, it will wrap back to the beginning of the buffer for the next iteration. Auxiliary registers (AR0–AR7) and CDP can all be used as circular pointers in the indirect addressing mode. The following steps set up circular buffers:

1. Initialize the most significant 7 bits of the extended auxiliary registers (ARnH or CDPH) to select the main data pages for the circular buffers. For example, `mov #k7,AR2H`.
2. Initialize the 16-bit circular pointers (ARn or CDP). The pointer can point to any memory location within the buffer. For example, `mov #k16,AR2`. Note that the initialization of the address pointers in steps 1 and 2 can be combined using the single instruction `amov #k23,XAR2`.
3. Initialize the 16-bit circular buffer starting address registers BSA01 (for AR0 or AR1), BSA23 (for AR2 or AR3), BSA45 (for AR4 or AR5), BSA67 (for AR6 or AR7), or BSAC (for CDP) that are associated with the auxiliary registers. For example, if AR2 (or AR3) is used as the circular pointer, we have to use BSA23 and initialize it using `mov #k16, BSA23`. The main data page concatenated with the content of this register defines the 23-bit starting address of the circular buffer.
4. Initialize the data buffer size registers BK03, BK47, or BKC. When using AR0–AR3 (or AR4–AR7) as the circular pointer, BK03 (or BK47) should be initialized. The instruction `mov #16, BK03` sets up the circular buffer of 16 elements for the auxiliary registers AR0–AR3.
5. Enable the circular buffers by setting the appropriate bits in the status register ST2. For example, the instruction `bset AR2LC` enables AR2 for circular addressing.

The following example demonstrates how to initialize the circular buffer `COEFF[4]` with four integers, and use the circular addressing mode to access data in the buffer:

```
amov   #COEFF,XAR2   ; Main data page for COEFF[4]
mov    #COEFF,BSA23  ; Buffer base address is COEFF[0]
mov    #0x4,BK03     ; Set buffer size of 4 words
mov    #2,AR2        ; AR2 points to COEFF[2]
bset   AR2LC         ; AR2 is configured as circular pointer
mov    *AR2+,T0      ; T0 is loaded with COEFF[2]
mov    *AR2+,T1      ; T1 is loaded with COEFF[3]
mov    *AR2+,T2      ; T2 is loaded with COEFF[0]
mov    *AR2+,T3      ; T3 is loaded with COEFF[1]
```

Since circular addressing uses the indirect addressing modes, the circular pointers can be updated using the modifications listed in Table C.6. As discussed in Chapter 3, FIR filtering can be efficiently implemented using circular buffers.

C.4 TMS320C55xx Assembly Language Programming

We can use either C or assembly language for C55xx programming [3]. In this section, we introduce the basics [4] of C55xx assembly programming, including assembly instructions in four categories: arithmetic, logic and bit manipulation, move (load and store), and program flow control.

C.4.1 Arithmetic Instructions

The arithmetic instructions include addition (`add`), subtraction (`sub`), and multiplication (`mpy`). Combinations of these basic operations produce subsets of instructions such as the multiply–accumulation (`mac`) and multiply–subtraction (`mas`) instructions. Most arithmetic operations can be executed conditionally. The C55xx also supports extended precision arithmetic such as add-with-carry, subtract-with-borrow, signed/signed, signed/unsigned, and unsigned/unsigned instructions. In Example C.8, the instruction `mpym` multiplies the data pointed at by AR1 and CDP, stores the product in the accumulator AC0, and updates AR1 and CDP after the multiplication.

Example C.8: Instruction

```
mpym *AR0+,*CDP-,AC0
```

Before instruction		After instruction	
AC0	00 3333 5555	AC0	00 0649 04BB
FRCT	0	FRCT	0
AR0	0100	AR0	0101
CDP	0402	CDP	0401
Data memory		Data memory	
0x100	12AB	0100	12AB
0x402	5631	0x402	5631

In Example C.9, the `macmr40` instruction performs MAC operation using AR1 and AR2 as data pointers. In addition, it also carries out the following operations:

- The keyword "`r`" produces a rounded result in the higher portion of the accumulator AC3. After rounding, the lower portion of AC3(15:0) is cleared.
- The 40-bit overflow detection is enabled by the keyword "`40`". If overflow occurs, the result in accumulator AC3 will be saturated to a 40-bit maximum value.
- The option "`T3 = *AR1 +`" loads the data pointed at by AR1 into T3.
- Finally, AR1 and AR2 are incremented by 1 to point to the next data memory location.

Example C.9: Instruction

```
macmr40 T3=*AR1+,*AR2+,AC3
```

	Before instruction	After instruction
AC3	00 0000 1234	00 2222 0000
FRCT	1	1
T3	FFF0	5555
AR1	0200	0201
AR2	0300	0301
Data memory		
0x200	5555	5555
0x300	3333	3333

C.4.2 Logic and Bit Manipulation Instructions

Logic operation instructions such as `and`, `or`, `not`, and `xor` (exclusive-OR) on data values are widely used in decision making and execution flow control. Logic operations are found in applications such as error correction coding in data communications. For example, the instruction

```
and #0xf,AC0
```

clears all upper bits in the accumulator AC0 except the four LSBs.

The bit manipulation instructions act on an individual bit or a pair of bits in the register or data memory. These instructions include bit clear, bit set, and bit test. Similar to logic operations, bit manipulation instructions are often used with logic operations to support decision-making processes. In Example C.10, the bit clear instruction clears the carry bit (bit 11) of the status register ST0.

Example C.10: Instruction

```
bclr #11,ST0_55
```

	Before instruction	After instruction
ST0	3802	3002

C.4.3 Move Instruction

The move instruction copies data values between registers and memory locations, and from register to memory, or memory to register. For example, the instruction

```
mov #1<<12,AC1
```

loads a constant to the accumulator AC1 with a shift. If the value of the original AC1 is zero, it becomes 00 0001 0000 after execution of the instruction. In general, we can use the instruction

```
mov #k<<n,AC1
```

where the constant k is shifted left by n bits first.

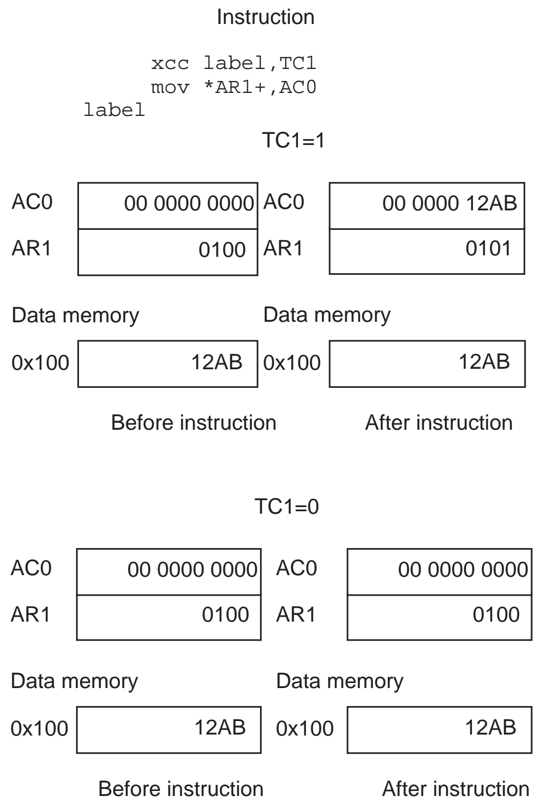
C.4.4 Program Flow Control Instructions

The program flow control instructions determine the execution flow of the program, including branching (b), subroutine call (call), loop operation (rptb), return to caller (ret), and so on. These instructions can be conditional, such as conditional call (callcc), conditional branch (bcc), and conditional return (retcc) used to control the program flow according to certain conditions. For example,

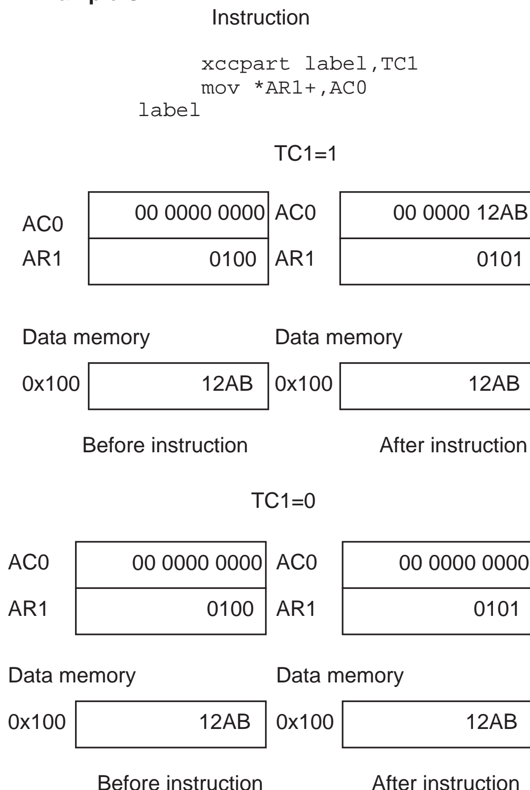
```
callcc my_routine, TC1
```

will call the subroutine my_routine only if the test control bit TC1 of the status register ST0 is set.

The conditional execution instruction, xcc, can be used in either conditional execution or partial conditional execution. In Example C.11, the conditional execution instruction tests the TC1 bit. If TC1 is set, the following instruction, mov *AR1+,AC0, will be executed, and both AC0 and AR1 will be updated. If the condition is false, AC0 and AR1 will not be changed. Conditional execution instruction xcc allows conditional execution of one instruction or two parallel instructions. The label is used for readability, especially when two parallel instructions are used.



In addition to conditional execution, the C55xx also supports partial conditional execution of an instruction. As shown in Example C.12, when the condition is true, both AR1 and AC0 will be updated. However, if the condition is false, the execution phase of the pipeline will not be carried out. Since the first operand (the address pointer AR1) is updated in the read phase of the pipeline, AR1 will be updated no matter of the condition, while the accumulator AC0 will remain unchanged at the execution phase. That is, the instruction has partially executed.



Many DSP applications such as filtering require repeated execution of instructions. These arithmetic operations may be located inside nested loops. If the number of instructions in the inner loop is small, the percentage of overhead for loop control may be very high compared to the instructions used in the inner loop. The loop control instructions, such as testing and updating the loop counter(s), pointer(s), and branches back to the beginning of the loop, impose a heavy overhead for any loop processing. To minimize this loop overhead, the C55xx provides built-in hardware for zero-overhead loop operations.

The single-repeat instruction (`rpt`) repeats the following single-cycle instruction or two single-cycle instructions that can be executed in parallel. For example,

```
rpt   #N-1              ; Repeat next instruction N times
mov   *AR2+,*AR3+
```

The immediate number, `N − 1`, is loaded into the single-repeat counter (RPTC) by the `rpt` instruction. It allows the following instruction, `mov *AR2+,*AR3+`, to be executed `N` times.

The block-repeat instruction (`rptb`) forms a loop that repeats a block of instructions. It supports a nested loop with an inner loop placed inside the outer loop. Block-repeat operations use block-repeat counters BRC0 and BRC1. For example,

```
      mov    #N-1,BRC0          ; Repeat outer loop N times
      mov    #M-1,BRC1          ; Repeat inner loop M times
      rptb   outloop-1          ; Repeat outer loop up to outloop
      mpy    *AR1+,*CDP+,AC0
      mpy    *AR2+,*CDP+,AC1
      rptb   inloop-1           ; Repeat inner loop up to inloop
      mac    *AR1+,*CDP+,AC0
      mac    *AR2+,*CDP+,AC1
inloop                          ; End of inner loop
      mov    AC0,*AR3+          ; Save result in AC0
      mov    AC1,*AR4+          ; Save result in AC1
outloop                         ; End of outer loop
```

This example uses two block-repeat instructions to control nested repetitive operations. The block-repeat structure

```
   rptb label_name-1
   (a block of instructions)
label_name
```

executes a block of instructions between the `rptb` instruction and the end label `label_name`. The maximum size of code that can be used inside a block-repeat loop is limited to 64Kbytes, and the maximum number of times that a loop can be repeated is limited to 65 536 ($= 2^{16}$) due to the use of 16-bit block-repeat counters. Because of the pipeline scheme, the minimum number of cycles within a block-repeat loop is two. For a single loop, we must use BRC0 as the repeat counter. When implementing nested loops, we will use BRC1 for the inner-loop counter and BRC0 for the outer-loop counter. Since repeat counter BRC1 will be reloaded each time when it reaches zero, it only needs to be initialized once.

The local block-repeat structure is illustrated as follows:

```
   rptblocal label_name-1
   (Instructions of 56 bytes or less)
label_name
```

This has a similar structure as the previous block-repeat loop, but is more efficient because the loop code is placed inside the instruction buffer queue (IBQ). Unlike the previous block-repeat loop, the local block-repeat loop only fetches instructions from memory once. These instructions are stored in the IU and are used throughout the entire looping operation. The size of the IBQ limits the size of the local-repeat loop to 56 bytes or less.

Finally, we list some basic C55xx mnemonic instructions in Table C.7. The listed examples are just a small set of the C55xx assembly instructions, especially the conditional instructions.

Table C.7 C55xx instruction set

Syntax	Meaning	Example
aadd	Modify AR	aadd AR0, AR1
abdst	Absolute distance	abdst *AR0+, *AR1+, AC0, AC1
abs	Absolute value	abs AC0, AC1
add	Addition	add uns(*AR4), AC1, AC0
addsub	Addition and subtraction	addsub *AR3, AC1, TC2, AC0
amov	Modify AR	amov AR0, AR1
and	Bitwise AND	and AC0<#16, AC1
asub	Modify AR	asub AR0, AR1
b	Branch	bcc AC0
bclr	Bit field clear	bclr AC2, *AR2
bcnt	Bit field counting	bcnt AC1, AC2, TC1, T1
bfxpa	Bit field expand	bfxpa #0x4022, AC0, T0
bfxtr	Bit field extract	bfxtr #0x2204, AC0, T0
bnot	Bit complement	bnot AC0, *AR3
brand	Bit field comparison	band *AR0, #0x0040, TC1
bset	Bit set	bset INTM
btst	Bit test	btst AC0, *AR0, TC2
btstclr	Bit test and clear	btstset #0xA, *AR1, TC0
btstset	Bit test and set	btstset #0x8, *AR3, TC1
call	Function call	call AC1
delay	Memory delay	delay *AR2+
cmp	Compare	cmp *AR1+ == #0x200, TC1
firsadd	FIR symmetric add	firsadd *AR0, *AR1, *CDP, AC0, AC1
firssub	FIR symmetric sub	firssub *AR0, *AR1, *CDP, AC0, AC1
idle	Force processor to idle	idle
intr	Software interrupt	intr #3
lms	Least mean square	lms *AR0+, *AR1+, AC0, AC1
mant	Normalization	mant AC0
		:: nexp AC0, T1
mac	Multiply–accumulate	macr *AR2, *CDP, AC0
		:: macr *AR3, *CDP, AC1
mack	Multiply–accumulate	mack T0, #0xff00, AC0, AC1
mar	Modify AR register	amar *AR0+, *AR1-, *CDP
mas	Multiply–subtraction	mas uns(*AR2), uns(*CDP), AC0
max	Maximum value	max AC0, AC1
maxdiff	Compare and select maximum	maxdiff AC0, AC1, AC2, AC1
min	Minimum value	min AC1, T0
mindiff	Compare and select minimum	mindiff AC0, AC1, AC2, AC1
mov	Move data	mov *AR3<<T0, AC0
mpy	Multiply	mpy *AR2, *CDP, AC0
		:: mpy *AR3, *CDP, AC1
mpyk	Multiply	mpyk #-54, AC0, AC1
neg	Negate	neg AC0, AC1
not	Bitwise complement	not AC0, AC1
or	Bitwise OR	or AC0, AC1
pop	Pop from stack	popboth XAR5

(*continued*)

Table C.7 (*Continued*)

Syntax	Meaning	Example
psh	Push to stack	psh AC0
reset	Software reset	reset
ret	Return	retcc
reti	Return from interrupt	reti
rol	Rotate left	rol CARRY, AC1, TC2, AC1
ror	Rotate right	ror TC2, AC0, TC2, AC1
round	Rounding	round AC0, AC2
rpt	Repeat	rpt #15
rptb	Repeat block	rptblocal label-1
sat	Saturate	sat AC0, AC1
sftl	Logic shift	sftl AC2, #-1
sfts	Signed shift	sfts AC0, T1, AC1
sqr	Square	sqr AC1, AC0
sqdst	Square distance	sqdst *AR0, *AR1, AC0, AC1
sub	Subtraction	sub dual(*AR4), AC0, AC2
subadd	Subtraction and addition	subadd T0, *AR0+, AC0
swap	Swap register	swap AR4, AR5
trap	Software trap	trap #5
xcc	Execute conditionally	xcc *AR0 != #0 add *AR2+, AC0
xor	Bitwise XOR	xor AC0, AC1

The complete mnemonic instruction set can be found in the TMS320C55xx Mnemonic Instruction Set Reference Guide [5].

C.4.5 Parallel Execution

The C55xx uses multiple-bus architecture, dual-MAC units, and separated PU, AU, and DU for supporting two types of parallel processing: implied (built-in) and explicit (user-built). The implied parallel instructions use the parallel column symbol "::" to separate the pair of instructions that will be processed in parallel. The explicit parallel instructions use the parallel bar symbol "| |" to indicate the pair of parallel instructions. Some of these two types of parallel instructions can be used together to form a combined parallel instruction. The following examples show the user-built, built-in, and combined parallel instructions that can be carried out in just one clock cycle.

User-built:

```
   mpym   *AR1+,*AR2+,AC0   ; User-built parallel instruction
|| and    AR4,T1            ;   using DU and AU
```

Built-in:

```
   mac  *AR2-, *CDP-,AC0  ; Built-in parallel instruction
:: mac  *AR3+,*CDP-,AC1  ;   using dual-MAC units
```

Built-in and user-built combination:

```
   mpy *AR2+,*CDP+,AC0  ; Combined parallel instruction
:: mpy *AR3+,*CDP+,AC1  ;   using dual-MAC units and PU
|| rpt #15
```

Some of the restrictions for using parallel instructions are summarized as follows:

1. For either the user-built or built-in parallelism, only two instructions can be executed in parallel, and these two instructions must not exceed 6 bytes.
2. Not all instructions can be used for parallel operations.
3. When addressing memory space, only the indirect addressing mode is allowed.
4. Parallelism is allowed between and within execution units, but there cannot be any hardware resource conflicts between units, buses, or within the unit itself.

There are several restrictions on the parallelism within each unit when applying parallel operations in assembly code, see the Mnemonic Instruction Set Reference Guide [5] for detailed descriptions.

The PU, AU, and DU can be involved in parallel operations. Understanding the register files and buses in these units will help to be aware of the potential conflicts when using the parallel instructions. Table C.8 lists some of the registers and buses in the PU, AU, and DU.

The parallel instructions used in the following example are incorrect because the second instruction uses the direct addressing mode:

```
   mov  *AR2,AC0
|| mov  T1,@x
```

We can correct this problem by replacing the direct addressing mode, `@x`, with an indirect addressing mode, `*AR1`, so both memory accesses are using the indirect addressing mode as follows:

Table C.8 Partial list of the C55xx registers and buses

PU registers/buses	AU registers/buses	DU registers/buses
RPTC	T0, T1, T2, T3	AC0, AC1, AC2, AC3
BRC0, BRC1	AR0, AR1, AR2, AR3,	TRN0, TRN1
RSA0, RSA1	AR4, AR5, AR6, AR7	
REA0, REA1	CDP	
	BSA01, BSA23, BSA45, BSA67	
	BK01, BK23, BK45, BK67	
Read buses: CB, DB	Read buses: CB, DB	Read buses: BB, CB, DB
Write buses: EB, FB	Write buses: EB, FB	Write buses: EB, FB

```
   mov  *AR2,AC0
|| mov  T1,*AR1
```

Consider the example

```
   mov   AC0,AR2
|| call  AC3
```

where the first instruction loads the content of AC0 that resides inside the DU to the auxiliary register AR2 inside the AU. The second instruction attempts to use the content of AC3 as the program address for a function call. Because there is only one link between the AU and DU, when both instructions try to access the accumulators in the DU via the single link, it creates a conflict.

To solve this problem, we can change the subroutine call from call by accumulator to call by address as follows:

```
   mov   AC0,AR2
|| call  my_func
```

This is because the instruction, `call my_func`, uses only the PU.

The coefficient dual AR-indirect addressing mode is used to perform operations with the dual AR-indirect addressing mode. The coefficient indirect addressing mode supports three simultaneous memory accesses (`Xmem`, `Ymem`, and `Cmem`). FIR filtering can effectively use this mode. The following code is an example of using the coefficient indirect addressing mode:

```
     mpy   *AR2+,*CDP+,AC2    ; AR1 pointer to data x1
::   mpy   *AR3+,*CDP+,AC3    ; AR2 pointer to data x2
||   rpt   #6                 ; Repeat the following 7 times
     mac   *AR2+,*CDP+,AC2    ; AC2 has accumulated result
::   mac   *AR3+,*CDP+,AC3    ; AC3 has another result
```

In this example, the memory buffers `Xmem` and `Ymem` are pointed at by the AR2 and AR3, respectively, while the coefficient array is pointed at by the CDP. The multiplication results are added to the contents in accumulators AC2 and AC3.

C.4.6 Assembly Directives

Assembly directives control assembly processes such as the source file listing format, data alignment, and section contents. They also enter data into the program, initialize memory, define global variables, set conditional assembly blocks, and reserve memory space for code and data. Some commonly used C55xx assembly directives are described in this section.

The `.bss` directive reserves uninitialized memory for data variables defined in the `.bss` section. It is often used to allocate data into RAM for runtime variables such as I/O buffers. For example, as

```
.bss  xn_buffer,size_in_words
```

where the `xn_buffer` points to the first location of the reserved memory space, and the `size_in_words` specifies the number of words to be reserved in the `.bss` section. If we do

not specify names for uninitialized data sections, the assembler will put all the uninitialized data into the `.bss` section, which is word addressable.

The `.data` directive tells the assembler to begin assembling the source code into the `.data` section, which usually contains data tables, or pre-initialized data, such as a sine table. The `.data` section is word addressable.

The `.sect` directive defines a code or data section and tells the assembler to begin assembling source code or data into that section. It is often used to separate long programs into logical partitions. It can separate subroutines from the main program, or separate constants that belong to different tasks. For example,

```
.sect  "user_section"
```

assigns the code into the user-defined section called `user_section`. Code sections from different source files with the same section name will be placed together.

Similar to the `.bss` directive, the `.usect` directive reserves memory space in an uninitialized section. It places data into user-defined sections. It is often used to separate large data sections into logical partitions, such as separating the transmitter variables from the receiver variables. The syntax for the `.usect` directive is

```
symbol  .usect "section_name", size_in_words
```

where `symbol` is the variable, or the starting address of a data array, which will be placed into the section named `section_name`.

The `.text` directive tells the assembler to begin assembling source code into the `.text` section, which is the default section for program code. If a code section is not specified, the assembler will put all the programs into the `.text` section.

The `.int` (or `.word`) directive places one or more 16-bit integer values into consecutive words in the current memory section. This allows users to initialize memory with constants. For example,

```
data1  .word 0x1234
data2  .int 1010111b
```

In these examples, `data1` is initialized to the hexadecimal number 0x1234 (decimal number 4660), while `data2` is initialized to the binary number 1010111b (decimal 87).

The `.set` (or `.equ`) directive assigns values to symbols. These types of symbols are known as assembly-time constants. They can then be used by source statements in the same manner as a numeric constant. The `.set` directive has the form

```
symbol .set value
```

where the `symbol` must appear in the first column. This equates the constant `value` to the `symbol`. The symbolic name used in the program will be replaced with the constant value by the assembler during assembly time, thus allowing programmers to write more readable programs. The `.set` and `.equ` directives can be used interchangeably.

The `.global` (`.def` or `.ref`) directive makes the symbols global for the external functions. The `.def` directive indicates a defined symbol that is given in the current file

and known to other external files. The .ref directive references an external defined symbol that is defined in another file. The .def directive has the form

```
.def symbol_name
```

The symbol_name can be a function name or a variable name that is defined in this file and can be referenced by other functions in different files. The .global directive can be interchanged with either the .def or .ref directive.

The .include (or .copy) directive reads the source file from another file. The .include directive has the form

```
.include "file_name"
```

The file_name is used to tell the assembler which file to read in as part of the source file.

C.4.7 Assembly Statement Syntax

The basic syntax for C55xx assembly statements may be separated into four ordered fields as

```
[label][:] mnemonic [operand list] [;comment]
```

The elements inside the square brackets are optional. Statements must begin with a label, blank, asterisk, or semicolon. Each field must be separated by at least one blank space. For ease of reading and maintenance, it is strongly recommended to use meaningful mnemonics for labels, variables, and subroutine names. An example of a C55xx assembly program is given in Table C.9.

The label field can contain up to 32 alphanumeric characters (A–Z, a–z, 0–9, _, and $). It associates a symbolic address with a unique program location. The line labeled in the assembly program can be referenced by the defined symbolic name. This is useful for modular programming and branch instructions. Labels are optional, but if used they must begin in the first column. Labels are case sensitive and must start with an alphabetic letter or underscore. In the example, the symbol start is defined as a global function entry point so functions in other files are able to reference it. The complex_data_loop symbol is another label in the text section. This is a local label for setting up the block-repeat loop by the assembler.

The mnemonic field can contain a mnemonic instruction, assembler directive, macro directive, or macro call. Note that the mnemonic field cannot start in the first column; otherwise, it would be interpreted as a label.

The operand field is a list of operands. An operand can be a constant, symbol, or combination of constants and symbols in an expression. An operand can also be an assembly-time expression that refers to memory, I/O ports, or pointers. Another category of the operands can be the registers and accumulators. Constants can be expressed in binary, decimal, or hexadecimal formats. For example, a binary constant is a string of binary digits (0s and 1s) followed by the suffix B (or b), and a hexadecimal constant is a string of hexadecimal digits (0, 1, . . . , 9, A, B, C, D, E, and F) followed by the suffix H (or h). A hexadecimal number can also use the prefix 0x similar to those used by the C language. The prefix # is used to indicate an immediate constant. For example, #123 indicates the operand is a constant of decimal number 123, while #0x53CD is the hexadecimal number of 53CD (equal to a decimal number of 21 453). Symbols defined in an assembly program with assembler directives may

Table C.9 An example of a C55xx assembly program

```
;
;    Assembly program example
;
N              .set 128
_Xin           .usect ".in_data",(2*N)    ; Input data array
_Xout          .usect ".out_data",(2*N)   ; Output data array
_Spectrum      .usect ".out_data",N       ; Data spectrum array
          .sect .data
input     .copy input.inc                 ; Copy input.inc into program

          .def   _start                   ; Define this program's entry
                                          ; point
          .def   _Xin,_Xout,_Spectrum     ; Make these data global data
          .ref   _dft_128,_mag_128        ; Reference external functions
          .sect  .text

_start
     bset  SATD                           ; Set up saturation for D unit
     bset  SATA                           ; Set up saturation for A unit
     bset  SXMD                           ; Set up sign extension mode
     mov   #N-1,BRC0                      ; Init. counter for loop N times
     amov  #input,XAR0                    ; Input data array pointer
     amov  #_Xin,XAR1                     ; Xin array pointer

     rptblocal complex_data_loop-1        ; Form complex data
     mov   *AR0+,*AR1+
     mov   #0,*AR1+
complex_data_loop
     amov  #_Xin,XAR0                     ; Xin array pointer
     amov  #_Xout,XAR1                    ; Xout array pointer
     call  _dft_128                       ; Perform 128-point DFT
     amov  #_Xout,XAR0                    ; Xout pointer
     amov  #_Spectrum,XAR1                ; Spectrum array pointer
     call  _mag_128                       ; Compute squared-magnitude
                                          ; response
     ret
```

be labels, register names, constants, and so on. For example, we use the `.set` directive to assign a value to the symbol `N` in Table C.9. Thus, the symbol `N` becomes a constant value of 128 during assembly time.

The assembly instruction

```
mov *AR0+,*AR1+
```

located inside the repeat loop copies the data pointed at by address pointer AR0 to a different memory location pointed at by address pointer AR1. The operand can also be a function name such as

```
call _dft_128
```

Comments are notes used to explain the important meanings of program. A comment can begin with an asterisk or a semicolon in column 1. Comments that begin in any other column must begin with a semicolon.

Assembly programming involves many considerations: allocating sections for programs, data, constants, and variables; initializing the processor mode; deciding on the proper addressing mode; and writing the assembly program. The example given in Table C.9 has a text section, `.sect text`, where the assembly program code resides, a data section, `.sect .data`, where the data file is copied into this program, and three uninitialized data sections, `.usect`, for stacks and data arrays.

C.5 C Programming for TMS320C55xx

In recent years, high-level languages such as C and C++ have become more popular for DSP applications since C compilers are getting better at generating efficient code. In this section, we will introduce some basic C programming considerations for the C55xx.

C.5.1 Data Types

The C55xx C compiler supports standard C data types. However, the C55xx uses three different data types as compared to other computer-based devices, see Table C.10.

For most general-purpose computers and microprocessors, C data type `char` is 8 bits, but the C55xx C data type `char` is a 16-bit word. The C55xx `long long` data type is 40 bits (as the size of accumulators). Also, the data type `int` is defined as 16-bit data by C55xx, while many other computers define `int` as 32-bit data. Therefore, in order to write portable C programs, avoid using `char`, `int`, and `long long` if possible. In this book, we use the C header file, `tistdtypes.h`, to define data types for C55xx to avoid mistakes.

C.5.2 Assembly Code Generation by C Compiler

The C55xx C compiler translates a C source code into an assembly source code first, then the C55xx assembler converts the assembly code into a binary object file, and finally the C55xx linker combines it with other object files and library files to create an executable program. Knowing how the C compiler generates the assembly code can help to write correct and efficient C programs.

Table C.10 C data types

Data type	Data size (bits)	
	C55xx	Computer
`char`	16	8
`int`	16	32
`short`	16	16
`long`	32	32
`long long`	40	64
`float`	32	32
`double`	64	64

Multiplication and addition are the two most widely used statements in C programs for DSP applications. One common mistake of working with `short` variables is not to use data type cast to force them into 32-bit data. For example, the correct C statements for multiplying two 16-bit numbers are

```
long   mult_32bit;
short  data1_16bit, data2_16bit;
mult_32bit = (long)data1_16bit * data2_16bit;
```

The C55xx compiler will treat the above multiplication as a 16-bit number times another 16-bit number with a 32-bit product because the compiler knows both data operands are 16 bits. The following statements are incorrectly written for the C55xx C compiler:

```
mult_32bit = data1_16bit * data2_16bit;

mult_32bit = (long)(data1_16bit * data2_16bit);

add_32bit = data1_16bit + data2_16bit;

sub_32bit = (long)(data1_16bit - data2_16bit);
```

where `add_32bit` and `sub_32bit` are defined as `long`.

For example, Table C.11 shows the C program that multiplies two 16-bit data using three different statements, and Table C.12 is the resulting CCS debugger's mixed mode display. The C55xx compiler generates the assembly code (in gray), multiplying the 16-bit data `a` with other 16-bit data `b`. Only the last C statement, `c3 = (long)a * b;`, generates the correct 32-bit result `c3`.

When we develop DSP code, repeated operations are often written using loop instructions. For example, the loop statement such as `for(i=0; i<count; i++)` is widely used in C programs for general-purpose computers. As discussed in Section C.4.4, this for-loop can be efficiently implemented using the C55xx repeat instructions `rpt` and `rptb`. However, the block-repeat counters BRC0 and BRC1 are 16-bit registers, and this limits the loop counter to 16 bits. The for-loop example in C shown in Table C.13 initializes a two-dimensional array. Table C.14 shows the CCS disassembly display of the nested for-loop. It is clearly shown that the C55xx C compiler-generated assembly code is not very efficient. By using the CCS

Table C.11 Multiplication example

```
short a=0x4000, b=0x6000;
static long c1,c2,c3;

void main(void)
{
    // Wrong: only the lower 16-bit result in AC0 is saved to c
    c1 = a * b;
    // Wrong: only the lower 16-bit result in AC0 is saved to c
    c2 = (long)(a * b);
   // Correct: the 32-bit result in AC0 is saved to c
  c3 = (long)a * b;
}
```

Table C.12 Assembly code generated by C compiler for multiplication of two numbers

```
          main:
21         c1 = a * b;
02010a:     a531001861          MOV *(#01861h),T1
02010f:     d33105001860        MPYM *(#01860h),T1,AC0
020115:     a010_98 MOV         mmap(@AC0L),AC0
020118:     eb3108001862        MOV AC0,dbl(*(#01862h))
23         c2 = (Int32)(a * b);
02011e:     d33105001860        MPYM *(#01860h),T1,AC0
020124:     a010_98              MOV mmap(@AC0L),AC0
020127:     eb3108001864        MOV AC0,dbl(*(#01864h))
26         c3 = (Int32)a * b;
02012d:     d33105001860        MPYM *(#01860h),T1,AC0
020133:     eb3108001866        MOV AC0,dbl(*(#01866h))
```

simulator, this short program takes 12 654 CPU cycles to complete. In the experiment in Section C.6, we will present the much more efficient assembly program in Table C.15, which takes only 624 cycles to perform the same task.

If it is possible, avoiding the use of compiler built-in supporting library functions can also improve real-time efficiency. This is because these library functions require function calls, which need to be set up and pass the arguments before the call and collect return values. Some of the C55xx registers need to be saved by the caller function so they can be used by the library functions. If we have to use the compiler's built-in library functions, we must find an efficient way of using them. An example of a modulo statement in C is listed in Table C.16. It is clear that the C55xx C compiler will generate more efficient code when using $2^n - 1$ for modulo-2^n operation as shown in Table C.17.

C.5.3 Compiler Keywords and Pragma Directives

The C55xx C compiler supports the `const` and `volatile` keywords. The `const` keyword controls the allocation of data objects. It ensures that the data objects will not be changed by

Table C.13 Nested for-loop example in C

```
short a[2][300];

void main (void)
{
    short i, j;

    for (i=0; i<2; i++)
    {
       for (j=0; j<300; j++)
       {
          a[i][j] = 0xffff;
       }
    }
}
```

Table C.14 The assembly program generated by the C compiler for the nested for-loop

```
        main:
020000:   4efd                       AADD  #-3,SP
19        for (i=0; i<2; i++)
020002:  e60000                      MOV   #0,*SP(#00h)
020005:  a900_3d2a                   MOV   *SP(#00h),AR1 || MOV #2,AR2
020009:  1298a0                      CMP   AR1 >= AR2, TC1
02000c:  04643f                      BCC   C$DW$L$_main$4$E,TC1
21          for (j=0; j<300; j++)
        C$DW$L$_main$2$B, C$L1:
02000f:  e60200                      MOV   #0,*SP(#01h)
020012:  76012ca8                    MOV   #300,AR2
020016:  a902                        MOV   *SP(#01h),AR1
020018:  1298a0                      CMP   AR1 >= AR2, TC1
02001b:  046422                      BCC   C$DW$L$_main$3$E,TC1
23              a[i][j] = 0;
        C$DW$L$_main$3$B, C$L2, C$DW$L$_main$2$E:
02001e:   b000                       MOV   *SP(#00h) << #16,AC0
020020:  79012c00                    MPYK  #300,AC0,AC0
020024:  2209                        MOV   AC0,AR1
020026:  a402                        MOV   *SP(#01h),T0
020028:  ec31be001860                AMAR  *(#01860h),XAR3
02002e:  1490b0                      AADD  AR1,AR3
020031:  e66b00                      MOV   #0,*AR3(T0)
21          for (j=0; j<300; j++)
020034:  f7020001                    ADD   #1,*SP(#01h)
020038:  a902                        MOV   *SP(#01h),AR1
02003a:  1294a0                      CMP   AR1 < AR2, TC1
02003d:  0464de                      BCC   C$DW$L$_main$2$E,TC1
19        for (i=0; i<2; i++)
        C$DW$L$_main$4$B, C$L3, C$DW$L$_main$3$E:
020040:  f7000001                    ADD   #1,*SP(#00h)
020044:  a900_3d2a                   MOV   *SP(#00h),AR1 || MOV #2,AR2
020048:  1294a0                      CMP   AR1 < AR2, TC1
02004b:  0464c1                      BCC   C$L1,TC1
```

Table C.15 Assembly code performs the same task as the C nested for-loop

```
    .global _asmLoop

    .text
_asmLoop:
    rpt #(2*300)-1
    mov #0xffff,*AR0+
    ret
```

Table C.16 Example of modulo operation

```
void main()
{
    short a=30;
    // Modulo operation calls library function
    a = a % 7;
    // Inefficient modulo operation of a power-of-2 number
   a = a % 8;
   // Efficient modulo operation of a power-of-2 number
   a = a & (8-1);
}
```

placing constant data into ROM space. The `volatile` keyword is especially important when the compiler optimization feature is turned on. This is because the optimizer may aggressively rearrange code and may even remove segments of code and data variables. However, the optimizer will keep all the variables with the `volatile` keyword.

The C55xx compiler also supports three new keywords: `ioport`, `interrupt`, and `onchip`. The keyword `ioport` is used for the compiler to distinguish memory space related to the I/O. Peripheral registers such as EMIF, DMA, Timer, McBSP, and so on are all located in the I/O memory space [6]. To access these registers, we must use the `ioport` keyword. This

Table C.17 Assembly program for modulo operations

```
           main:
020146:     4eff                AADD #-1,SP
10         short a=30;
020148:     e6001e              MOV #30,*SP(#00h)
13            a = a % 7;
02014b:     a400                MOV *SP(#00h),T0
02014d:     6c0200f5_3d75       CALL I$$MOD || MOV #7,T1
020153:     c400                MOV T0,*SP(#00h)
16            a = a % 8;
020155:     2249                MOV T0,AR1
020157:     2290_3d7a           MOV AR1,AC0 || MOV #7,AR2
02015b:     10053e_37aa         SFTS AC0,#-2,AC0 || NOT AR2,AR2
020160:     76e000b0            BFXTR #57344,AC0,AR3
020164:     249b                ADD AR1,AR3
020166:     28ba                AND AR3,AR2
020168:     26a9                SUB AR2,AR1
02016a:     c900                MOV AR1,*SP(#00h)
19            a = a & (8-1);
02016c:     f4000007            AND #7,*SP(#00h)
```

Table C.18 C program of enabling DPLL

```
// Function for enabling or disabling clock generator PLL
#define PLLENABLE_SET      1      // PLL enable
#define CLKMD_ADDR     0x1c00
#define CLKMD          (ioport volatile unsigned short *)CLKMD_ADDR
#pragma CODE_SECTION(pllEnable, ".text:C55xxCode");
void pllEnable(short enable)
{
     short clkModeReg;
     clkModeReg = *CLKMD;
     if (enable)
         *CLKMD = clkModeReg | (PLLENABLE_SET<<4);
     else
         *CLKMD = clkModeReg & 0xFFEF;
}
```

keyword can only be applied to global variables and used for local or global pointers. Since I/O is only addressable in the 16-bit range, all variables including pointers are 16 bits even if the program is compiled for the large-memory model.

Interrupts and interrupt services are common in programs for real-time DSP applications. Due to the fact that the interrupt service routine (ISR) requires specific register handling and relies on special sequences for entry and return, the `interrupt` keyword supported by the C55xx compiler specifies the function that is actually an ISR.

To utilize its dual-MAC feature in C, the C55xx compiler uses the keyword `onchip` to qualify the memory that may be used by dual-MAC instructions. This memory must be located at the on-chip DARAM.

The `CODE_SECTION` pragma allocates program memory space and places the function associated with the pragma into that section. The `DATA_SECTION` pragma allocates a data memory space and places the data associated with the pragma into that data section instead of the `.bss` section.

Table C.18 shows a C program example that enables the digital phase lock loop (DPLL). In this example, the function `pllEnable` is placed in the program memory space inside the `.text` section with the subsection of `C55xxCode`.

C.6 Mixed C and Assembly Programming

As discussed in Chapter 1, mixed C and assembly programs are used for many DSP applications. High-level C code provides portability, ease of development, and maintenance, while assembly code has the advantages of runtime efficiency and code density. In this section, we will introduce how to interface C with assembly programs, and provide guidelines for C function calling conventions. The assembly routines called by a C function can have arguments and return values just like C functions. The following guidelines are important for writing the C55xx assembly code that is callable by C functions:

Naming convention. Use the underscore "_" as the prefix for all variables and routine names for the assembly program that will be accessed by the C functions. For example, use `_asm_func` as the name of the assembly routine that will be called by a C function. If a variable is defined in the assembly routine, it must use the underscore prefix, such as `_asm_var`, for a C function to access it. The prefix "_" is used only by the C compiler. When we access assembly routines or variables from C functions, we do not need to use the underscore prefix.

Variable definition. The variables that are accessed by both C and assembly routines must be defined as global variables using the directive `.global`, `.def`, or `.ref` by the assembler.

Compiler mode. By using the C compiler, the C55xx CPL (compiler mode) bit is automatically set for using the stack pointer relative addressing mode when entering an assembly routine. The indirect addressing modes are preferred under this configuration. If we need to use direct addressing modes to access data memory in a C callable assembly routine, we must change to the DP-direct addressing mode. This can be done by clearing the CPL bit. However, before the assembly routine returns to its C caller function, the CPL bit must be restored. The bit clear and bit set instructions, `bclr CPL` and `bset CPL`, can be used to reset and set the CPL bit in the status register ST1, respectively. The following example code can be used to check the CPL bit, turn off the CPL bit if it is set, and restore the CPL bit before returning it to the caller:

```
    btstclr #14,*(ST1),TC1      ; Turn off CPL bits if it is set
    (more instructions . . . )
    xcc continue,TC1            ; TC1 is set if we turn off CPL bit
    bset CPL                    ; Turn on CPL bit
continue
    ret
```

Passing arguments. To pass arguments from a C function to an assembly routine, we must follow the strict rules of C callable conversions set by the C55xx compiler. When passing an argument, the C compiler assigns it to a particular data type and then places it using a register according to its data type. The C55xx C compiler uses the following three classes to define the data types:

- Data pointers: `short *`, `int *`, or `long *`.
- 16-bit data: `char`, `short`, or `int`.
- 32-bit data: `long`, `float`, `double`, or function pointers.

If the arguments are pointers to data memory, they are treated as data pointers. If the argument can fit into a 16-bit register such as `short`, `int`, and `char`, it is considered to be 16-bit data. Otherwise, it is 32-bit data. The arguments can also be structures. A structure of two words (32 bits) or less is treated as a 32-bit data argument and is passed using a 32-bit register. For structures larger than two words, the arguments are passed by reference. The C compiler will pass the address of a structure as a pointer, and this pointer is treated like a data argument.

For a subroutine call, the arguments are assigned to registers in the order that the arguments are listed by the function. They are placed in the registers according to their data type, in the order shown in Table C.19.

Table C.19 shows the overlap between the AR registers used for data pointers and the registers used for 16-bit data. For example, if T0 and T1 hold 16-bit data arguments and

Table C.19 Argument classes assigned to registers

Argument type	Register assignment order
16-bit data pointer	AR0, AR1, AR2, AR3, AR4
23-bit data pointer	XAR0, XAR1, XAR2, XAR3, XAR4
16-bit data	T0, T1, AR0, AR1, AR2, AR3, AR4
32-bit data	AC0, AC1, AC2

AR0 already holds a data pointer argument, a third 16-bit data argument would be placed into AR1. See the second example in Figure C.13. If the registers of the appropriate type are not available, the arguments are passed onto the stack, as the third example given in Figure C.13.

Return values. The calling function collects the return values from the called function/ subroutine. The 16-bit data is returned in the register T0, and 32-bit data is returned in the accumulator AC0. A data pointer is returned in (X)AR0, and a structure is returned on the local stack.

Register use and preservation. When calling a function, the register assignments and preservations between the caller and called functions are strictly defined. Table C.20 describes how the registers are preserved during a function call. The called function must save the contents of the save-on-entry registers (T2, T3, AR5, AR6, and AR7) if it uses these registers. The calling function must push the contents of any other save-on-call registers onto the stack if these registers' contents are needed after the function/ subroutine call. A called function can freely use any of the save-on-call registers (AC0–AC3, T0, T1, and AR0–AR4) without saving the save-on-call register's value. More detailed descriptions can be found in the TMS320C55xx Optimizing C Compiler User's Guide [7].

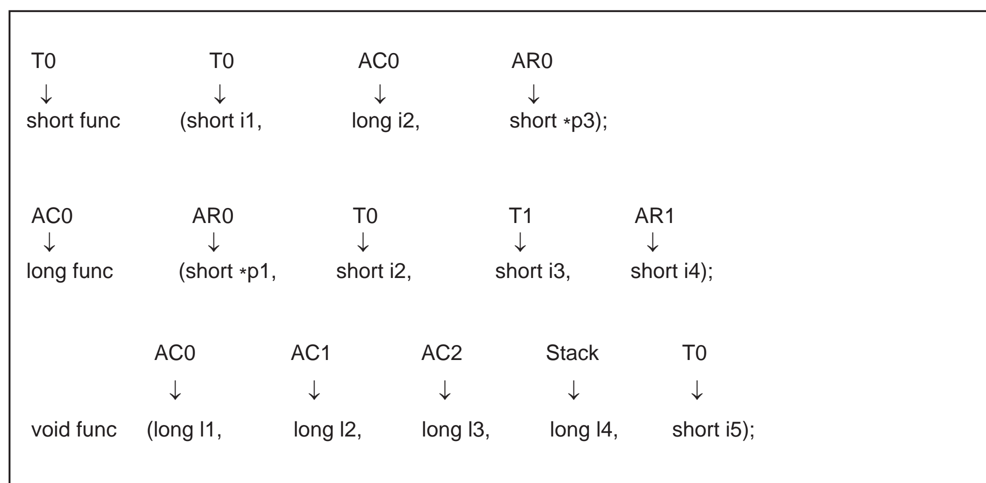


Figure C.13 Examples of argument-passing conventions

Table C.20 Register use and preservation conventions

Registers	Preserved by	Used for
AC0–AC2	Calling function	16-, 32-, or 40-bit data
	Save-on-call	24-bit code pointers
(X)AR0–(X)AR4	Calling function	16-bit data
	Save-on-call	16- or 23-bit pointers
T0 and T1	Calling function	16-bit data
	Save-on-call	
AC3	Called function	16-, 32-, or 40-bit data
	Save-on-entry	
(X)AR5–(X)AR7	Called function	16-bit data
	Save-on-entry	16- or 23-bit pointers
T2 and T3	Called function	16-bit data
	Save-on-entry	

As an example of mixed C and assembly programming, the following C program calls the assembly function `findMax()` that returns the maximum value of a given array:

```
extern short findMax(short *p, short n);
void main()
{
    short a[8]={19, 55, 2, 28, 19, 84, 12, 10};
    static short max;
    max = findMax(a, 8);
}
```

The assembly function `findMax()` is listed as follows:

```
;
;    Function prototype:
;      short findMax(short *p, short n);
;
;    Entry: AR0 is the pointer of p and T0 contains length n
;    Exit: T0 contains the maximum data value found
;
    .def _findMax               ; Using "_" prefix for C-callable
    .text
_findMax:
    sub #2,T0
    mov T0,BRC0                 ; Set up loop counter
    mov *AR0+,T0                ; Place the first data in T0
||  rptblocal loop-1            ; Loop through entire array
    mov *AR0+,AR1               ; Place the next data in AR1
    cmp T0<AR1,TC1              ; Check to see if new data is greater
||  nop                         ; than the maximum?
    xccpart TC1                 ; If find new maximum, place it in T0
||  mov AR1,T0
loop
    ret                         ; Return with maximum data in T0
```

The assembly routine findMax returns the 16-bit maximum value of the array pointed at by the data pointer *p. The assembly function uses the underscore "_" as the prefix for the function name. The first argument is a 16-bit data pointer and it is passed via auxiliary register AR0. The array size is the second argument, which is 16-bit data, and is passed via the temporary register T0. The return value is 16-bit data in the T0 register.

C.7 Experiments and Program Examples

In this section, the CCS, simulator, and C5505 eZdsp are used to conduct experiments to show and demonstrate several examples given in this appendix. By doing these examples, we can become more familiar with the C55xx addressing modes, register updates, conditional execution, loop implementation, as well as mixed C and assembly functions. The experiments using the C5505 eZdsp to play audio signals in real time are also presented. These basic audio experiments are modified for other real-time experiments throughout the book. The basic knowledge of the CCS introduced by experiments given in Chapter 1 is required to complete the experiments presented in this section.

C.7.1 Examples

This experiment shows Examples C.1 to C.12 presented in this appendix. The files used for the experiment are listed in Table C.21.

Procedures of the experiment are listed as follows:

1. Start CCS and create a working folder.
2. Copy the files from the companion software package to the experiment folder.
3. Import the project using the CCS import feature.
4. Build and load the program. Launch the selected configuration from the CCS Target Configurations window to load the program for the experiment. To do this, from the CCS menu bar, **View** → **Target Configurations**, right-click on the configuration file to launch it. Although the C55xx simulator is used for this experiment, the eZdsp can also be used as the target device for the experiment by modifying the target configuration file AppC_examples.ccxml using CCS.
5. From the CCS **View** pull-down menu, select **View** → **Registers**. Expand **CPU Registers** in the **View** window to view the registers used by the experiment.
6. Use single-step command (F5) to walk through each example to observe the C5505 registers' values while stepping through each instruction.

Table C.21 File listing for the experiment ExpC.1

Files	Description
examples.asm	Program contains examples in Appendix C
appC_examplesTest.c	Program for showing and testing examples
c5505.cmd	Linker command file

Table C.22 File listing for the experiment ExpC.2

Files	Description
`assembly.asm`	Assembly program
`dft_128.asm`	Assembly program for computing DFT
`mag_128.asm`	Assembly program for computing magnitude
`assembly_programTest.c`	Program for testing assembly programs
`input.inc`	Data include file using assembly syntax
`lpva200.inc`	Program include file using assembly syntax
`c5505.cmd`	Linker command file

C.7.2 Assembly Program

This experiment demonstrates the assembly program syntax as described in Section C.4.7. The files used for the experiment are listed in Table C.22.

Procedures of the experiment are listed as follows:

1. Create a working folder, and copy the files from the companion software package to the folder.
2. Import the CCS project for this experiment. Build and load the program.
3. Use the single-step (F5) command to walk through the assembly program.
4. Get familiar with the single steps of C statements in C and assembly instructions in the assembly program. Show that the single-step results are as expected.
5. What is the difference between the CCS **Step Into** and **Step Over** operations? How does the CCS **Step Return** operation work?
6. How can the data section, `_Xin .usect ".in_data",(2*N)`, be defined in the assembly program and linker command file?

C.7.3 Multiplication

This experiment uses an example of multiplying two 16-bit integers to demonstrate the use of proper data types for C programming. The files used for the experiment are listed in Table C.23.

Procedures of the experiment are listed as follows:

1. Copy the files from the companion software package to the working folder. Import the project, and build and load the program.

Table C.23 File listing for the experiment ExpC.3

Files	Description
`multiplyTest.c`	Program for testing multiplication
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file

2. For the multiplication results displayed in the CCS console window, which one is correct? Why does the program generate wrong results?
3. Restart the experiment and single-step through the multiplication instructions, watching the registers to understand how the C55xx performs the multiplication.

C.7.4 Loops

This experiment demonstrates how the C compiler may generate inefficient assembly programs and compares the effectiveness to hand-coded assembly programs for real-time applications. The experiment also shows how to use the CCS profile feature to measure the clock cycles needed for performing specific operations. The files used for the experiment are listed in Table C.24.

Procedures of the experiment are listed as follows:

1. Copy the files from the companion software package to the working folder, import the project, and build and load the program.
2. Set breakpoints at the lines indicated by the program, see Figure C.14. Run to the first breakpoint.

Table C.24 File listing for the experiment ExpC.4

Files	Description
`asmLoop.asm`	Assembly program performs loop operations
`loopTest.c`	Program for testing loop operation
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file

*loopTest.c   0xffff04

```
20 {
21     Int16 i, j;
22     Int16 *p;
23
24     printf("Exp C.4 --- C Loop and assembly loop\n");
25
26     for (i=0; i<2; i++)          // <-set break point #1 at this line
27     {
28         for (j=0; j<300; j++)
29         {
30             a[i][j] = 0xffff;
31         }
32     }
33
34     p = (short *)a;              // <-set break point #2 at this line
35     asmLoop(p);
36
37     printf("\nExp. completed\n");  // <-set break point #3 at this line
38 }
39
```

Licensed   : 622   Writable   Sm...rt

Figure C.14 The assembly function used 622 clock cycles

3. From the CCS top menu bar, click on **Run** → **Clock** → **Enable** to enable the clock profile tool.
4. From the CCS top menu bar, click on **Run** → **Clock** → **Setup . . .** to select the manual rest option.
5. Run the program to each breakpoint, and record the clock counts at each breakpoint to obtain the required clock cycles. The C program for-loop uses 13 249 cycles while the assembly loop `asmLoop()` uses only 622 cycles as shown in Figure C.14.

C.7.5 Modulo Operator

This experiment demonstrates how to apply programming techniques properly in C programs to improve runtime efficiency. The files used for the experiment are listed in Table C.25.

Procedures of the experiment are listed as follows:

1. Copy the files from the companion software package to the working folder. Import the CCS project, and build and load the program.
2. Set breakpoints at the lines indicated by the program, see Figure C.15.

Table C.25 File listing for the experiment ExpC.5

Files	Description
`moduloTest.c`	Program for testing modulo operation
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file

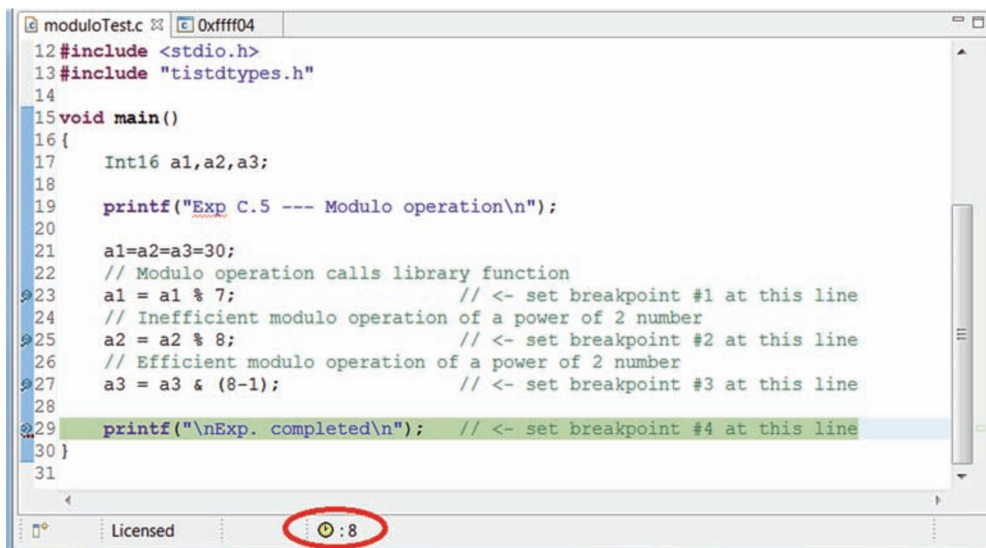


Figure C.15 Measure CPU cycles for each operation

3. Enable the CPU clock, and run the program to measure and compare the clock cycles required for each operation. This experiment shows that if the modulo operation uses a power-of-2 number, then using the "`&`" (`AND`) instruction is much more efficient than calling the C library.

C.7.6 Use Mixed C and Assembly Programs

This experiment demonstrates the mixing of C with assembly programs for sorting numbers. It shows how the C program calls assembly functions and how the assembly program calls C functions. The files used for the experiment are listed in Table C.26.

Procedures of the experiment are listed as follows:

1. Copy the files from the companion software package to the working folder. Import the CCS project, and build and load the program.
2. Run the experiment to verify the sorting results.
3. Use the CCS single-step tool to examine the CPU registers' values before and after calling the assembly function `arraySort()` and examine the registers' values before and after the assembly function calls the C function `sort()`. Review Section C.6 to understand the save-on-call registers during the function calls.

Table C.26 File listing for the experiment ExpC.6

Files	Description
`arraySort.asm`	Assembly program performs sorting operation
`findMax.asm`	Assembly program finds the largest number
`sort.c`	C function performs sorting operation
`c_assemblyTest.c`	Program for testing mixed C and assembly programs
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file

C.7.7 Working with AIC3204

This experiment uses the C5505 eZdsp USB stick with the board support functions. These software programs are located in the companion software package under the folder `USBSTK_bsl`, where the C programs are in the `bsl` subfolder and the include files are in the `inc` subfolder. These files are developed specifically for the C5505 eZdsp. The experiment also includes the C55xx chip support functions. These C programs and include files are located in the `C55xx_csl` folder, where the C programs are in the `src` subfolder and the include files are in the `inc` subfolder. The experiment shows how to set up and initialize the eZdsp and the analog interface chip, AIC3204. The program generates a 1 kHz tone and plays it using the eZdsp. The files used for the experiment are listed in Table C.27.

Procedures of the experiment are listed as follows:

1. Copy the files from the companion software package to the working folder and import the project into the CCS.

Table C.27 File listing for the experiment ExpC.7

Files	Description
`initAIC3204.c`	C program initializes AIC3204
`tone.c`	C function generates a tone
`audioPlaybackTest.c`	Program for testing eZdsp audio playback
`tistdtypes.h`	Standard type define header file
`c5505.cmd`	Linker command file

2. Connect the eZdsp to the computer's USB port, then from the CCS **Target Configurations** window launch the configuration and connect the eZdsp as the target device.
3. Connect a headphone or a pair of loudspeakers to the eZdsp HP OUT jack.
4. Build and load the experiment, run the program, and listen to the 1 kHz tone played by the C5505 eZdsp.
5. Change the eZdsp sampling rate (`SAMPLING`) to verify that 48, 32, 24, 12, and 8 kHz are all working properly. Explain how to generate a 1 kHz tone at different sampling rates.
6. Change the D/A output gain (`GAIN`) to verify that the audio output level varies according to the D/A gain.
7. This experiment utilizes the C5505 chip support libraries (CSL) and the eZdsp board support libraries (BSL). The files for these libraries are included in this experiment. Interested users can examine these files to understand how to configure the AIC3204 and operate the eZdsp. For some of the experiments presented in this book, we will use the libraries `USBSTK_bsl.lib` and `C55xx_csl.lib` which are built using the files in the folders `USBSTK_bsl` and `C55xx_csl` from this experiment.

C.7.8 Analog Input and Output

In the previous experiment, we used the C5505 eZdsp for real-time audio playback. Many real-time applications need to process, convert, and transfer signals between analog and digital forms. Therefore, it is often necessary to digitize an analog signal for processing, and convert the processed signal back to analog form for output. This experiment presents the analog I/O program, which digitizes an analog signal from the STEREO IN jack on the eZdsp, converts the digital signal back to analog again, and then outputs to the eZdsp headphone HP OUT jack. The experiment begins with initializing the C5505 eZdsp, and then sets up the AIC3204 similar to the previous experiment. The difference from the previous experiment is that this program uses direct memory access (DMA) for data transfer between the AIC3204 and the C5505 processor. The files used for the experiment are listed in Table C.28.

Procedures of the experiment are listed as follows:

1. Copy the files from the companion software package to the working folder and import the project.
2. Connect the C5505 eZdsp to the computer's USB port.
3. Connect a headphone or a pair of loudspeakers to the eZdsp HP OUT jack. Connect an audio source (MP3 player or radio) to the eZdsp STEREO IN jack.
4. From the CCS, launch the configuration file and connect the eZdsp as target devices. Build and load the program.

Table C.28 File listing for the experiment ExpC.8

Files	Description
AIC3204_init.asm	Assembly program initializes AIC3204
audio.c	C function sends data samples to AIC3204 for playback
audioExpTest.c	Program for testing audio loopback experiment
dma.c	DMA support functions
i2s.c	I2C support functions
i2s_register.asm	I2S register definition and read/write functions
vector.asm	C5505 interrupt vector table
tistdtypes.h	Standard type define header file
c5505.cmd	Linker command file

5. Play audio using the audio player, and listen to the audio playback with the headphone (or speakers) connected to the eZdsp HP OUT jack to make sure that the program works properly.
6. Several programs are used for this experiment. `AIC3204_init.asm`, `dma.c`, `i2s.c`, and `i2s_register.asm` are the supporting functions for configuring and operating the AIC3204 device. They are included in this experiment so readers have the complete set of files that can be modified for different applications. For many real-time experiments in this book, the library `myC55xUtil.lib` is used, which is created using `AIC3204_init.asm`, `dma.c`, `i2s.c` and `i2s_register.asm`. Interested readers can create this library from these files using the CCS.

References

1. Texas Instruments, Inc. (2012) TMS320C5505, Fixed-Point Digital Signal Processor, SPRS660E, January.
2. Texas Instruments, Inc. (2009) C55x V3.x, CPU Reference Guide, SWPU073E.
3. Texas Instruments, Inc. (2001) TMS320C55x, Programmer's Reference Guide, SPRU376A.
4. Texas Instruments, Inc. (2011) TMS320C55x, Assembly Language Tools v4.4, SPRU280I, November.
5. Texas Instruments, Inc. (2009) TMS320C55x, CPU Mnemonic Instruction Set Reference Guide, SWPU067E, June.
6. Texas Instruments, Inc. (2011) TMS320C55x, DSP Peripherals Overview, SPRU317K, December.
7. Texas Instruments, Inc. (2011) TMS320C55x, Optimizing C/C++ Compiler v4.4, SPRU281G, December.

Index

2's complement, 72, 74, 91
2-D
fast convolution, 448
FFT, 448
image filtering, 430
3-dB bandwidth, 151, 175, 319
60 Hz hum, 6, 104
AAC, see MPEG-4 AAC
Absolute addressing modes, 506
AC-3, 7, 378, 384
Acoustic
echo, 254, 304
echo cancellation, 254, 315, 321, 325
echo path, 317
Adaptive
algorithm, 239, 243, 252, 256, 260, 266, 276, 308, 321
channel equalization, 309
filter, 67, 239, 243, 251, 255, 267, 276, 304, 306, 312, 316, 318, 321
line enhancer, 262
echo cancellation, 306
noise cancellation, 264
notch filter, 266
prediction, 262
system identification, 259, 288, 290, 304, 306, 312
Adaptive multi-rate (AMR), 332, 348
Adaptive PCM (ADPCM), 251, 331
ADC (Analog-to-digital converter), 3, 9, 14, 75, 112, 317
Adder, 47, 82
Address-data flow unit (AU), 491, 493
Addressing modes, 13, 499
Advanced audio coding, 6, 378
Algebraic CELP (ACELP), 339
Aliasing, 4, 9, 130, 155, 200, 215, 284, 317, 384
Allpass filter, 105, 378, 404
Amplitude spectrum, see Magnitude spectrum
AMR (Adaptive multi-rate)
AMR-NB, 339, 347, 349
AMR-WB, 338, 349
Analog
filter, 9, 130, 148, 151, 154, 392
interface chip (AIC), 9, 35, 37, 533
Ancillary data, 380
Antialiasing filter, 5, 9, 130
Antisymmetric, 109, 137, 345
ASIC (Application specific integrated circuits), 11
Arithmetic logic unit (ALU), 13, 492
Arithmetic
error, 2, 44, 75
instructions, 508
ASCII (text) file, 26, 46, 124, 486
Assembler directives, 518
Assembly statement syntax, 518
Audio effect, 352, 377, 397, 401, 423, 425
Auditory masking, 333, 378, 380
Autocorrelation matrix, 245, 247, 252
Auxiliary register (AR0-AR7), 226, 492, 495, 502, 506, 516, 528

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.

© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

Bandlimited, 5, 290
Bandpass filter, 5, 64, 104, 123, 135, 154, 165, 263, 276, 290, 292, 311, 318, 381, 387, 389
Bandstop (band-reject) filter, 104, 106, 153, 165, 176
Bandwidth, 2, 5, 12, 15, 131, 133, 154, 156, 175, 179, 288, 290, 330, 340, 349, 355, 377, 380, 391, 396, 405, 431
Bark, 381
Barrel shifter, 492
Bessel filter, 151, 153
Bilinear transform, 155, 392
Binary
  file, 27
  point, 15, 73
Biquad, 159, 167, 172
Bit manipulation instruction, 509
Bit-reversal, 208, 210, 224, 227, 232
Blackman window, 145
Block
  FIR filter, 128, 135
  processing, 16, 188, 222, 274, 276, 321, 384, 399, 401, 412
BMP, 461, 469, 470
Breakpoint, 21, 30, 33
Broadband signal, 266, 276, 297
Butterfly network, 206, 210
Butterworth filter, 151
Cascade, 56, 160, 168, 170, 182, 185, 188, 391
Causal, 45, 51, 53, 114, 148, 177
Canonical form, 159
CCS, see Code composer studio
CD player, 377
Center
  clipper, 314
  frequency, 104, 154, 266, 290, 391
Channel equalization, 309
Characteristic equation, 63
Charge-coupled device (CCD), 432
Chebyshev
  approximation, 90, 121
  filter, 152
Circular
  addressing, 14, 85, 128, 136, 142, 182, 188, 471, 495, 498, 507
  buffer, 85, 113, 492, 498, 507
  convolution, 203, 222
  pointer, 85, 183, 284, 507
Clipping threshold, 315
Complimentary metal-oxide semiconductor (CMOS), 432
Code composer studio (CCS), 21, 484
Coefficient
  data pointer (CDP), 495, 502, 504
  quantization, 75, 78, 124, 170, 322
Coder/decoder (CODEC), see Codec
Code-excited linear predictive (CELP), 332
Codec, 9, 332, 343, 347, 349, 357, 359, 378, 482
Color
  balance, 434, 464
  correction, 434, 464
Comb filter, 105, 351, 378, 401, 404
Combined parallel instruction, 514
Comfort noise, 314, 349, 356, 369
Companding, 8, 330, 373
Compiler optimization, 524
Complex
  arithmetic, 295, 480
  LMS algorithm, 252, 277
  variable, 148, 150, 202, 224, 479
Complex-conjugate, 63, 160, 174, 179, 210, 252, 294, 345, 480
Contrast adjustment, 465
Convergence
  factor, see Step size
  speed, 243, 248, 253, 309, 317
Convolution, 54, 111, 149, 205, 222, 255, 293, 398, 441
Correlation function, 239
Critical
  band, 378, 381
  frequencies, 156
  sampling, 383
Crosscorrelation (Cross-correlation), 241, 245, 310
Cumulative probability distribution function, 67
Cutoff frequency, 5, 9, 104, 121, 130, 133, 151, 166, 347, 391, 394, 405
DAC (Digital-to-analog converter), 3, 14, 317
Data
  computation unit (DU), 490, 492
  page pointer (DP), 501, 505
dBm, 290, 296, 482
dBm0, 482
dBov, 358, 482
DC
  component, 196
  offset, 6
Decibel (dB), 66, 107, 290, 297, 481

Decimation, 8, 10, 130, 133, 141, 318, 383, 399, 459
Delay
estimation, 309, 311
unit, 47, 264, 309
Delta function, 44, 241, 440
Deterministic signal, 44
Differentiator, 123
Digital
filter, 41, 44, 51, 78, 102, 104, 106, 122, 127, 154, 171, 243, 392
signal, 1, 5, 7
signal processing, 1
signal processors, 12
Direct addressing modes, 501
Direct form, 78, 158, 161, 170, 174, 179, 181, 295, 335, 392
Direct memory access (DMA), 14, 38, 552
Discrete
cosine transform (DCT), 383, 452, 468
Fourier transform (DFT), 65, 195, 198, 200, 224, 226, 383, 400, 448
Discontinuous transmission (DTX), 349, 356, 368
Discrete-time
Fourier transform (DTFT), 61, 114, 198
signal, 1, 4, 7, 75, 78, 198, 478
Dispersive delay, 307, 310
Dolby AC-3, see AC-3
Double-talk, 308, 312, 317, 341
DSP, see Digital signal processing
DTMF (Dual-tone multi-frequency)
frequencies, 292, 296
tone detection, 283, 292, 294
tone generator, 283, 291, 298
Dual slope, see Analog-to-digital converter
Dynamic range, 2, 8, 15, 73, 75, 81, 83, 190, 253, 256, 322, 360, 377, 463
Echo
cancellation, 254, 260, 304, 308, 312, 315, 341, 350, 360
path, 306, 313, 317, 321
return loss (ERL), 308, 313, 322
return loss enhancement (ERLE), 308
Eigenvalue spread, 253
Elliptic filter, 153
Encoding process, 4, 7, 380
Equalizer
graphic equalizer, 389, 391
parametric equalizer, 174, 179, 377, 389, 391, 397, 421
Error
contours, 247
surface, 247
Euler's
formula, 196
theorem, 478
Even function, 61, 65, 104, 196, 203, 240
Excess mean-square error (MSE), 244, 252, 256, 308, 310, 322
Excitation signal, 259, 290, 307, 333, 337, 346, 349
Expectation
operation, 68
value, see Mean
Far-end, 306, 308, 312, 318, 323
Fast
convolution, 195, 214, 222, 234, 448
Fourier transform (FFT), 11, 13, 16, 65, 195, 205, 208, 217, 220
Feedback coefficients, 52
Feedforward coefficients, 52
Field-programmable gate array (FPGA), 11
Filter
adaptive, 67, 239, 243, 247, 250, 255, 262, 304, 306, 312, 316, 318, 321
allpass, 105, 378, 404
all-pole, 152, 315
notch, 104, 176, 266, 378, 394, 403
peak, 391, 397, 402, 421
shelf, 378, 391, 394, 421
weighting, 333, 340, 345, 364
Filter specifications, 104, 106, 122, 155, 166,
Filterbank, 292, 318, 321, 349, 351, 378, 387, 456, 472
analysis, 318, 378
synthesis, 318, 321, 349, 378, 472
Finite impulse response, see FIR filter
Finite wordlength (precision) effects, 75, 125, 158, 168, 170, 210, 252, 255, 461
FIR filter, 49, 56, 62, 82, 102, 105, 108, 112, 118, 120, 127, 130, 135, 222, 224, 234, 243, 247, 257, 278, 290, 307, 311, 316, 318, 321, 442
Fixed-point, 13, 72, 74, 78, 124, 164, 167, 212, 255, 322, 443
Flanger, 397, 401, 403, 422
Flash ADCs, see Analog-to-digital converter
Flat delay, 307

Floating-point, 13, 78, 86, 123, 128, 170, 322, 411, 415, 419
Folding frequency, see Nyquist frequency
Fourier
  coefficient, 196
  series, 114, 119, 195, 197
  series (window) method, 114, 120
  transform, 61, 65, 114, 149, 195, 197, 200, 220
Four-wire facility, 304
Fractional number, 72
Frequency
  domain coding, 383
  index, 65, 195, 200, 294, 400
  offset test, 297
  resolution, 65, 200, 212, 217, 292, 383, 385, 387
  response, 61, 105, 108, 114, 120, 125, 130, 149, 168, 442
  shift, 198, 399, 403
  transform, 148, 153, 166
  units, 123
  warping, 156
Fundamental frequency, 195, 332, 351, 399
Gamma correction, 434, 436
Geometric series, 53, 57, 478
Gibbs phenomenon, 116
Goertzel algorithm, 292, 297, 300
Gradient estimate, 248, 254
Group delay, 103, 109, 153
GSM, 347
Hamming window, 118, 332
Hands-free telephone, see Speakerphone
Hanning window, 119, 352, 400
Harmonics, 106, 197, 297
Harvard architecture, 12
Highpass filter, 104, 153, 341, 443, 456, 458
Hilbert transformer, 105
Histogram equalization, 437
Hybrid, 304
  loss, see Echo return loss
Ideal
  filter, 106, 151
  sampler, 4
Interaural intensity difference (IID), 409
IIR Filters, 52, 56, 78, 148, 154, 158, 162, 164, 166, 168, 170, 172, 239, 243, 391, 395, 421
Image filtering, 430, 440, 449
Impedance, 290, 304
Implied parallel instructions, 514
Impulse
  function, 441
  response, 48, 51, 61, 114, 149, 169, 177, 222, 260, 293, 307, 398, 441
Impulse-invariant method, 155
Indirect addressing modes, 503, 506, 526, 528
Infinite impulse response, see IIR filters
Input
  autocorrelation matrix, 245
  quantization, 76
  vector, 127, 134, 172, 244, 255
Instantaneous
  power, 313
  squared error, 249, 254
Instruction
  buffer queue (IBQ), 490, 512
  buffer unit (IU), 490
  set, 12, 210, 490, 514
Interleave, 232, 339, 433
International Telecommunication Union (ITU)
  ITU, 6, 277, 291, 296, 313, 331, 359, 432, 436
  ITU-T, 305, 308, 332, 338, 341, 343, 349, 452, 482
Internet protocol (IP), 304, 330, 355
IP telephony, see Voice over IP (VoIP)
Interpolation, 8, 130, 133, 142, 215, 285, 400, 403
Interrupt service routine (ISR), 525
Intrinsics, 20, 185, 271
Inverse
  discrete cosine transformation (IDCT), 452, 468
  discrete Fourier transform (IDFT), 114, 201, 209, 211, 351, 383, 400, 448
  fast Fourier transform (IFFT), 209, 212, 222, 351, 448
  Fourier transform, 197
  Laplace transform, 148, 150
  z-transform, 54, 58
IP network, 305, 357
Interaural time difference (ITD), 409
Jitter buffer, 357, 359
JPEG (Joint Photographic Experts Group), 432, 434, 452, 455
JPEG2000, 452, 456, 460
Kaiser window, 120, 135, 218
Kronecker delta function, 44
Laplace transform, 52, 148
Leakage factor, 257
Leakage or smearing, 117
Leaky LMS algorithm, 257

Learning curve, 253
Least-mean-square, see LMS algorithm
Least significant bit (LSB), 72
Line
  echo, 304, 306, 356
  spectrum, 196, 344
Linear
  chirp signal, 286
  congruential method, 288
  convolution, 50, 54, 111
  interpolation, 50, 54, 111, 149, 205, 222, 398, 441
  phase, 103, 108, 114, 120, 243
  predictive coding (LPC), 315, 331
  time-invariant, 102, 111, 149
Linearity (superposition), 54
Linker, 13, 19, 21, 520
Linker command file, 24
LMS algorithm, 243, 249
Logarithmic quantizer, 9
Lookup table method, 283, 286, 386
Lossless audio coding, 386
Lossy compression, 378
Lowpass filter, 5, 9, 103, 107, 119, 130, 133, 151, 153, 156, 166, 199, 318, 392, 443, 456, 458, 470
Magnitude
  bits, 72
  response, 61, 64, 102, 151, 153, 155, 166, 168, 319
  spectrum, 65, 196, 202, 214, 219, 254, 256, 351, 353, 389
  test, 296
Mainlobe, 117, 216
Mapping properties, 148, 150
Marginally stable, 60, 169, 175
Maskee, 380
Masker, 380, 382
Masking threshold, 378, 387
Mean, 68, 76, 125, 240, 312, 437
MDCT, see Modified discrete cosine transform
Mean-square error (MSE), 244, 248, 252, 254, 333, 436
Mean-square value, 69, 240
Microprocessor, 12
Microcontrollers, 10
Minimum MSE, 246, 254
Modified discrete cosine transform, 378, 383, 387, 411
Modulo operation, 96, 186, 284
Most significant bit (MSB), 72
Moving average filter, 50, 57
MP3, 378, 385
MPEG-1 Layer-3, see MP3
MPEG-2 AAC, 6, 378, 386
MPEG-4 AAC, 378, 386
MPEG-4 CELP, 332
MSE surface, 248
Narrowband noise, 106, 276
National Television System Committee (NTSC), 431
Near-end, 306, 309
Negative symmetry, see Antisymmetric
Network echo, 304
Noise
  Gaussian, 290
  generators, 94
  reduction, 321, 326, 456
  subtraction, 350
Nonlinear processor (NLP), 309, 314
Normalized
  digital frequency, 45, 198
  frequency, 190, 198
  LMS algorithm, 254
  step size, 318
Notch filter, 378, 393
Nyquist
  frequency, 5, 65
  interval, 5
  rate, 5
Odd function, 61, 203
One-sided z-transform, 53
Operands, 500, 518
Optimum
  filter, 246
  solution, 246, 248
  weight vector, 246
Oscillatory behavior, 116
Overflow, 81, 211
Overlap-add, 223, 390, 399, 401
Overlap-save, 223
Overload, 282
Oversampling, 9, 130
Overshoot, 102
Packet loss concealment (PLC), 356, 369
Panning, 410
Parallel
  connection, 56
  converter, 9
  execution, 514
  form, 161

Parametric equalizer, 179, 190
Parks–McClellan algorithm, 120
Parseval's theorem, 219
Partial-fraction expansion, 161, 164
Passband, 104
  edge (cutoff) frequency, 107
  ripple, 107
Perceptive (perceptual) weighting filter, 333, 340, 344, 364
Performance
  (or cost) function, 244, 247
  surface, 247
Periodic signal, 195
Periodicity, 201
Periodogram, 219
Phase
  distortion, 103, 105
  response (shift), 61, 103
  spectrum, 202
Phaser, 401, 404
Pixel, 430, 434, 441, 465
PN sequence, see Pseudo-random numbers
Polar form, 53, 61, 202, 479
Pole, 58, 63, 149, 294, 331, 344
Pole-zero
  cancellation, 58
  plot (diagram), 58
Polynomial approximation, 86, 151, 283
Positive symmetric, 109
Postfilter, 332, 347
Power, 69
Power density spectrum (PDS), 219, 290
Power spectral density, see PDS
Power spectrum, see PDS
Pre-echo effect, 401, 417
Pre-whitening, 309
Probability density function, 67
Processing time, 15, 113
Profiler (profiling), 21
Pseudo-random
  binary sequence generator, 278, 289
  numbers, 70, 242, 278, 288
Psychoacoustics
  model, 378
  masking, 381
Pulse code modulation (PCM), 330, 377
Q format, 74
Quadrature mirror filterbank (QMF), 459
Quantization, 4, 212
  effect, 77, 125
  errors (noise), 7, 75, 284, 418
  process, 4, 7, 76, 124
  step (interval, width, resolution), 7, 75
Quiet threshold, 380
Radix-2 FFT, 206
Raised cosine, 118
Random
  number generation, 96, 288
  process (signals), 68, 219, 239
  variable, 66, 239
Realization of IIR filter, 158, 162, 166
Real-time
  signal processing, 2
  constraint, 15
Real-time transport protocol (RTP), 305
Reconstruction filter, 9, 472
Rectangular
  pulse train, 196
  window, 116, 215
Recursive
  algorithm, 334
  computation (calculation), 294
  oscillator, 174, 177
Register bits addressing mode, 506
Region of convergence, 53
Residual
  echo, 307, 312
  echo suppressor, 352
  noise reduction, 352
Resonator (peaking) filter, 175
Reverberation, 398
RGB color space, 432, 434
Room transfer function, 318
Roundoff error (noise), 78, 160, 171, 256
Sample
  autocorrelation function, 242
  mean, 241
  space, 67
Sampling
  frequency (rate), 4, 45, 133
  process, 3
  theorem, 5, 199
Saturation arithmetic, 81
Scalar, 47
Schur-Cohn stability test, 169
Second harmonic test, 297
Serial converter, 9
Settling time, 102
Shannon's sampling theorem, 5
Sidelobes, 117, 216

Sigma-delta ADC, see Analog-to-digital converter
Signal buffer, 112
Signal-to-noise ratio (SNR), 70
Signal-to-quantization-noise ratio, 7, 77, 83
Sign bit, 72
Sinewave (sine wave)
  generation, 86, 90, 175, 283
  table, 284
Sinusoidal signal, 44, 172, 298
Steady-state response, 102
Sirens, 288, 299
Smearing, 117, 215, 217
Smoothing filter, see Reconstruction filter
Sound pressure level (SPL), 380, 482
Sound reverberation, 398
Spatial sound, 409, 425
Speakerphone, 304, 316
Spectral
  analysis, 214
  dynamic range, 253
  envelope, 337
  leakage, 215
  resolution, 215
  smearing, 217
  subtraction, 321, 351, 370
Spectrogram, 220
Speech
  enhancement, 350
  (source) coding, 330
  recognition, 350
Square wave, 358, 482
Stability, 60, 150, 168
  constraint, 252
  triangle, 170
Stalling, 256
Standard deviation, 69
Stationary, 240, 353
Steady-state response, 102
Steepest descent, 248
Step size, 308, 322, 352
Stochastic gradient algorithm, see LMS algorithm
Stochastic process, see Random process
Stopband, 104, 151
  edge frequency, 107
  ripple (or attenuation), 107
Successive approximation, see Analog-to-digital converter
Superposition, 54
Symmetry, 108, 201, 203
Synthesis filter, 319, 332, 459
System
  identification, 259, 275
  stack pointer (SSP), 497
Tail delay, 307
Taylor series, 86
Time delay, 15, 54, 103, 239, 304, 409
Time division multiplex (TDM), 305
Time-quantization error, 284
Toeplitz matrix, 245, 334
Tone generation, 94
Tone detection, 292, 356
Total energy test, 297
Total harmonic distortion, 283
Training signal, 278
Transfer function, 54, 149
Transient response, 102
Transition band, 107
Transversal filter, see FIR filter
TRAU, 348
Tremolo, 401, 405, 408, 424
Trigonometric function, 283
Twiddle factor, 201, 224
Twist test, 297
Two-sided z-transform, 53
Uncorrelated noise, 312, 351
Uniform density function, 68
Unit
  circle, 53, 150
  delay, 47
Unit-impulse
  signal, 44, 48
  sequence, 44
Units of power, 480
Unit-step sequence, 44
Variance, 69
Voice activity detector (VAD), 357
Voice over IP (VoIP), 305, 355
von Neumann architecture, 12
Wavelet
  Daubechies, 459, 470
  Coiflets, 456
  Haar, 456
  Morlet, 456
  Symlets, 456
Weight vector, 244, 246
White balance, 434, 464
White Gaussian noise (WGN), 290
White noise, 241, 262, 290
Whitening, 309

Wideband, 6, 349
Wide-sense stationary (WSS), 240
Window function, 118, 216, 384
Wordlength, 4, 7, 10, 72, 160
YUV color spaces, 432
YCbCr color spaces, 432
Zero, 58, 149
Zero-crossing, 343
Zero-mean, 68, 262
Zero-overhead looping, 14
Zero-padding, 205
Zigzag sequence, 452
Z-plane, 150
Z-transform, 52, 150