Academic year
2025-2026

Faculty of Applied Engineering

Digital Signal Processing

Signal Processing Systems – Textbook

Walter Daems

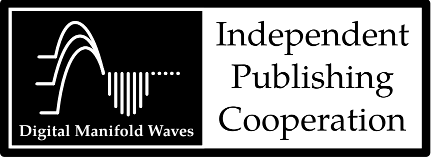
Master of Science in Electronics and ICT Engineering Technology
Master of Science in de industriële wetenschappen: elektronica-ICT

2210FTIESY I-Electronic Systems
2212FTIESY I-Elektronische Systemen

This document has been typeset using LaTeX and the `uantwerpendocs` package.
Calculations have been performed using Matlab/Octave.
Graphics have been composed using PGF, TiKZ and InkScape.
All this material has been prepared on a GNU/Linux workstation.

All trademarks are copyright of their respective owners.

Typesetting of this document was enabled by:



This document is under copyright. However, if you want to obtain a free license to use and distribute it (whether it as a lecturer or as a student), send an e-mail with your request to the author (walter.daems@uantwerpen.be).

DSP-SPS-2025-3.11

CONFIDENTIAL AND PROPRIETARY.

© 2025 University of Antwerp, All rights reserved.

Contents

1 Introduction **1**

I Systems, convolution and filtering **3**

2 Systems **5**

2.1 Introduction 5

2.2 Definition and representation 6

2.3 Classification (I) 6

2.3.1 MIMO, MISO, SIMO and SISO systems 6

2.3.2 Stable vs. nonstable systems 7

2.3.3 Linear vs. nonlinear systems 7

2.3.4 Time invariance 9

2.4 LTI systems 10

2.4.1 Benefits 10

2.4.2 System description 12

2.5 Classification (II) 15

2.5.1 Causal vs. noncausal systems 15

2.5.2 FIR vs. IIR systems 17

2.6 Building causal LTI systems 17

2.6.1 Discrete-time causal LTI systems 17

2.6.2 Continuous-time causal LTI systems 20

2.7 Classification (III) 22

2.7.1 Recursive vs. nonrecursive systems 22

2.8 Basic forms of causal discrete-time LTI systems 23

2.8.1 Nonrecursive systems 23
2.8.2 Recursive systems 25
2.9 Non LTI systems 31

3 System architectures 33
3.1 Introduction 33
3.2 Incremental vs. frame-based architectures 34
3.3 Organizing frames 35
3.3.1 Single-buffer architectures 35
3.3.2 Double-buffer architectures 38
3.3.3 Mixed-buffer architectures 39
3.3.4 Comparison 39

4 Convolution and FFT-convolution 47
4.1 Introduction 47
4.2 Convolution 48
4.3 Convolution properties 48
4.3.1 Commutativity 48
4.3.2 Associativity 49
4.3.3 Distributivity 49
4.4 Causal Convolution 49
4.5 Time-limited convolution 50
4.5.1 Convolution with a time-limited signal 50
4.5.2 Convolution of two time-limited signals 51
4.6 Basic implementation schemes 52
4.6.1 Input-side algorithms 52
4.6.2 Output-side algorithms 53
4.6.3 Buffer types 53
4.7 Implementation schemes in the time domain 57
4.7.1 Incremental 57

4.7.2 Frame-based single buffer operation . . . 60
4.7.3 Frame-based double buffer operation . . . 67
4.8 Implementation schemes in the frequency domain . . . 72
4.8.1 Basic principle . . . 72
4.8.2 Frame-based single buffer operation . . . 74
4.8.3 Frame-based double buffer operation . . . 75
4.9 Choosing the right convolution method . . . 84
4.9.1 Incremental algorithms . . . 84
4.9.2 Frame-based algorithms in the time domain . . . 86
4.9.3 Frame-based algorithms in the frequency domain . . . 88
4.9.4 Comparison . . . 90
4.10 Conclusions . . . 94

5 Digital Filtering 95

5.1 Introduction . . . 95
5.2 What are we filtering for? . . . 96
5.2.1 Noise removal . . . 96
5.2.2 Band selection . . . 98
5.2.3 Deconvolution . . . 101
5.2.4 Correlation detection - Matched filtering . . . 102
5.3 Filter requirements description for band-selection filters . . . 105
5.3.1 Basic filter types . . . 105
5.3.2 Basic terminology . . . 107
5.4 Plotting frequency responses Octave/Matlab . . . 108

6 FIR Filter Design 113

6.1 Introduction . . . 113
6.2 Zero placement . . . 114
6.2.1 Arbitrary placement . . . 114
6.2.2 Feedforward comb filters . . . 116

6.2.3 Strengths and weaknesses 120
6.3 Impulse response invariance method 120
6.3.1 Design procedure 120
6.3.2 Example 122
6.3.3 Strengths and weaknesses 123
6.4 Frequency sampling design method 123
6.4.1 Design procedure 123
6.4.2 Example 128
6.4.3 Strengths and weaknesses 133
6.5 Optimal linear-phase filter design 133
6.5.1 Introduction 133
6.5.2 Least-squares method 137
6.5.3 Parks-McClellan / Remez-exchange algorithm 150
6.6 Conclusion 160

7 IIR Filter Design 163

7.1 Introduction 163
7.2 Pole-zero placement 164
7.2.1 Arbitrary placement 164
7.2.2 Biquad sections 166
7.2.3 Feedback comb filters 167
7.2.4 Classical band-selection filters 171
7.2.5 Strengths and weaknesses 175
7.3 Impulse invariance method 175
7.3.1 Fundamentals 175
7.3.2 Design procedure 176
7.3.3 Example 177
7.3.4 Strengths and weaknesses 178
7.4 Bilinear transformation 178

7.4.1 Fundamentals . . . 178
7.4.2 Frequency warping and frequency-prewarping . . . 182
7.4.3 Example . . . 183
7.4.4 Strengths and weaknesses . . . 186
7.5 Reverse filtering . . . 186
7.5.1 Principle . . . 186
7.5.2 Fundamentals . . . 187
7.5.3 Strengths and weaknesses . . . 188
7.6 Conclusion . . . 189

II Advanced Signal Transformations 191

8 Signal Transforms — Short-time Fourier Transform 193
8.1 Time-frequency atoms . . . 193
8.1.1 Position and size of the atoms in the time domain . . . 194
8.1.2 Position and size of the atoms in the frequency domain . . . 194
8.1.3 Heisenberg boxes . . . 195
8.2 Linear time-frequency transforms . . . 195
8.3 Generating Dictionaries . . . 197
8.4 Gabor's short-time Fourier transform . . . 197
8.4.1 Forth... . . . 197
8.4.2 Intermezzo: its Heisenberg boxes . . . 198
8.4.3 ...and back . . . 200
8.5 Cutting Gabor's elephant to pieces . . . 201
8.5.1 Forth... . . . 202
8.5.2 Intermezzo: its Heisenberg boxes . . . 203
8.5.3 ...and back . . . 207
8.6 Examples . . . 208
8.6.1 Analyzing an arbitrary signal . . . 208

8.6.2 Creating a sound-level meter . . . 212
8.7 Conclusion . . . 213

9 Signal Transforms — Wavelets 215

9.1 Introduction: why do we need wavelets? . . . 215
9.2 The ball park . . . 218
9.3 The Haar transform . . . 219
9.3.1 The level-1 Haar transform . . . 221
9.3.2 One step back: the level-1 Haar transform from a wavelet perspective . . . 223
9.3.3 Trend and fluctuation . . . 227
9.3.4 Example . . . 227
9.3.5 The level-n Haar transform . . . 230
9.3.6 The level-n Haar transform from the perspective of wavelets . . . 234
9.3.7 Multi-resolution analysis . . . 239
9.4 Wavelet construction . . . 240
9.4.1 The essence of wavelets . . . 240
9.4.2 Finding scaling and wavelet numbers . . . 244
9.4.3 The Haar transform revisited . . . 245
9.5 Wavelet transforms — an overview . . . 246
9.5.1 Haar wavelets . . . 246
9.5.2 Daubechies wavelets . . . 247
9.5.3 Coifman wavelets . . . 258
9.5.4 Biorthonormal wavelets . . . 267
9.6 Wavelet packet transforms . . . 276
9.7 Edge effects . . . 277
9.8 Wavelets and Digital Filtering . . . 278
9.8.1 The Haar transform as subband filter . . . 278
9.8.2 Generic wavelet transform as subband filter . . . 280
9.8.3 Recursive subband decomposition and recomposition . . . 281

9.9 Two-dimensional wavelet transforms 281
9.9.1 Principles 281
9.9.2 Example 286
9.9.3 The base vectors of the two-dimensional wavelet transforms 292
9.10 Applications 298
9.10.1 Signal compression 299
9.10.2 Denoising 308

III Capita Selecta 313

10 CIC Filters 315
10.1 Decimation and interpolation 315
10.2 Noble identities 316
10.3 Rate conversion and CIC filters 318
10.3.1 The attractive nature of moving average filters 318
10.3.2 Crossing the rate-change border 324
10.4 Correcting the pass-band distorsion 327

11 Polyphase filters 329
11.1 What's in a name? 329
11.2 The maths - for 3 phases 329
11.3 The maths - for M phases 331
11.4 Crossing the rate-change border 333
11.4.1 Decimation 333
11.4.2 Interpolation 334
11.5 Conclusion 335

A Circular Library 337

B Mathematical Bits and Pieces 343
B.1 Orthogonal signal decompositions 343

B.1.1 Elements . . . 343
B.1.2 Operations . . . 344
B.1.3 Geometric notions . . . 345
B.1.4 Vector space . . . 345
B.1.5 Set notions . . . 346
B.1.6 Base and dimension of a vector space . . . 346
B.1.7 Decomposition of vectors in terms of the base vectors . . . 347
B.1.8 Parseval's identity . . . 347
B.2 Taylor's theorem . . . 348
B.3 The 'Big-O' notation . . . 348
B.3.1 Formal definition . . . 348
B.3.2 Practical use . . . 349

C Amdahl's law **351**

D Stochastic Theory **353**
D.1 Experiments and outcomes . . . 353
D.2 Events . . . 354
D.3 Probability function . . . 354
D.4 Stochastic or random variables . . . 355
D.5 Describing stochastic variables . . . 355
D.5.1 Cumulative probability distribution . . . 355
D.5.2 Ordinary probability distribution . . . 356
D.5.3 Properties . . . 356
D.6 Describing distributions using moments . . . 356
D.6.1 Expected value . . . 356
D.6.2 Raw moments . . . 357
D.6.3 Centralized moments . . . 357
D.6.4 Characteristic values . . . 357
D.7 Transformations of stochastic variables . . . 358

D.8 Orthogonal signal decompositions 359
D.8.1 Elements 359
D.8.2 Operations 360
D.8.3 Geometric notions 360
D.8.4 Vector space 361
D.8.5 Set notions 361
D.8.6 Base and dimension of a vector space 362
D.8.7 Decomposition of vectors in terms of the base vectors 362
D.8.8 Parseval's identity 363

Preface

The scenery

For a description of the scenery of Digital Signal Processing, please refer to the first volume in this series.

The available literature on DSIP is vast. Many good books exist on the subject. This work in particular was inspired by some of them you can find in the reference list at the end [Smi03, Smi08b, Smi08a, Lyo04, vdW92, OSB99, Sij04, vdEV87, MO94, Boz94, GW07, GWE09, BtMvdBJvdV93, VK13, Wal08, DLW06, Mal09]. So, why write another introductory-level text on the very same subject? It allows treating the subjects at a level appropriate to undergraduates in Applied Engineering. There is almost no textbook available that has the correct mix of mathematics and engineering. Writing my own course material also allows elaborating difficult subjects based on teaching experience and updating evolving subjects swiftly.
It avoids also having to purchase multiple expensive books to cover the subject. The reader should be warned that the mathematical treatment in this textbook is not very rigorous.

The script

This book is the third in a series of three books on digital signal and image processing:

1. Digital Signal Processing — Signals and Transformations
2. Digital Image Processing
3. Digital Signal Processing — Signal Processing Systems

The first volume focuses on signals and signal transformations. The second volume focuses on image processing. This third focuses on building digital signal processing systems, and adds some advanced transforms to the picture.

On the language used in this textbook: I tried to write this book from the perspective of a tutor guiding his tutees. Therefore, the text lacks the formality of many scientific text books. I hope you like this style. Where appropriate, I left out some mathematical derivations in order not to clutter the overall picture. I tried to add as much hints as needed to allow you working your way through the (sometimes difficult) material on your own if you like. The "he-him-his" formulation that has been used is not to emphasize the lack of women in engineering. This wording (instead of the more elaborate he/she, him/her and his/hers) has been chosen to keep this text simple.

A lot of effort has been put into this edition.

- This edition has been equipped with a solution book containing the solutions to the exercises. Making exercises is thé way to make sure you understand the theory. Exercises marked with (*) are a bit harder than the standard non-marked exercises. Those marked with (**) are for the enthousiastic reader (to fill any rare rainy days).
- This edition has been equipped with a formula collection that brings the important equations and definitions within reach in a convenient survey. If you think equations are missing from this collection, please inform me about this and I will consider adding them.

Though I try to avoid any errors, human erring is of all times. Do not hesitate to check with me if you find any errors. Even when you think there is an error and there's not, you will gain my appreciation for taking the exercises seriously.

Most of the material in this textbook is my original work. Some of it has been taken from other (free/open) sources. In case you notice a copyright infringement (or a reference that is not clear), please contact me. It is my firm desire to be 100% in line with the copyright legislation. If there happens to be an infringment, I appologize for it even right now. I will do all that is reasonably possible to overcome that issue as soon as possible when it occurs.

The crew

Finally, I would like to thank many people who contributed to this text.
First, a special thanks to my editor, Paul Levrie, for helping me by reviewing this text and supporting me with references, his profound experience, joyful humor and music.
Thanks to Maggy Goossens, Eric Paillet and Jan Steckel for bringing interesting background material to my attention.
Finally, my deepest gratitude goes to my beloved wife and children for enduring me devoting my time to writing this text, instead of spending my time with them.

Any contribution to this work is welcome. You won't get any money for it. I can only offer you to be listed in the hall of fame in this "Preface". You can contact me by e-mail to `walter.daems@uantwerpen.be`.

I hope you enjoy discovering digital signal and image processing!

Walter Daems
Summer 2025
Jordan Green, Norfolk (UK)

Symbol Table

Symbol	Meaning
Number sets	
$\mathbb{N}$	set of natural numbers (positive integer numbers)
$\mathbb{Z}$	set of integer numbers
$\mathbb{Q}$	set of rational numbers
$\mathbb{I}$	set of irrational numbers (real numbers that are not rational)
$\mathbb{R}$	set of real numbers
$\mathbb{C}$	set of complex real numbers
$\mathbb{X}^+$	set $\mathbb{X}$ restricted to positive numbers (not for $\mathbb{C}$)
$\mathbb{X}^-$	set $\mathbb{X}$ restricted to negative numbers (not for $\mathbb{C}$)
$\mathbb{X}_0$	set $\mathbb{X}$ with 0 excluded
Transform symbols - Forward	
$C_k = \mathrm{FS}\left(x(t)\right)$	C_k are the harmonic numbers of the Fourier Series of $x(t)$
$X(\omega) = \mathcal{F}\left(x(t)\right)$	$X(\omega)$ is the Fourier Transform of $x(t)$
$X(\omega) = \mathrm{DtFT}\left(x[n]\right)$	$X(\omega)$ is the Discrete-time Fourier Transform of $x[n]$
$X[k] = \mathrm{DFT}\left(x[n]\right)$	$X[k]$ is the Discrete Fourier Transform of $x[n]$
$X(s) = \mathcal{L}\left(x(t)\right)$	$X(s)$ is the Laplace Transform of $x(t)$
$X(z) = \mathcal{Z}\left(x[n]\right)$	$X(z)$ is the Z-Transform of $x[n]$
$c[k] = {}_m\mathcal{W}\left(x[n]\right)$	$c[k]$ is the m-level Wavelet-Transform of $x[n]$
$c[k] = {}_m\mathcal{H}\left(x[n]\right)$	$c[k]$ is the m-level Haar-Transform of $x[n]$
$c[k] = {}_m^l\mathcal{D}\left(x[n]\right)$	$c[k]$ is the m-level Daubechies-l Transform of $x[n]$
$c[k] = {}_m^l\mathcal{C}\left(x[n]\right)$	$c[k]$ is the m-level Coifman-l Transform of $x[n]$
Transform symbols - Inverse	
$x(t) = \mathrm{FS}^{-1}\left(C_k\right)$	C_k are the harmonic numbers of the Fourier Series of $x(t)$
$x(t) = \mathcal{F}^{-1}\left(X(\omega)\right)$	$X(\omega)$ is the Fourier Transform of $x(t)$
$x[n] = \mathrm{DtFT}^{-1}\left(X(\omega)\right)$	$X(\omega)$ is the Discrete-time Fourier Transform of $x[n]$
$x[n] = \mathrm{DFT}^{-1}\left(X[k]\right)$	$X[k]$ is the Discrete Fourier Transform of $x[n]$

continued on next page

continued from previous page

Symbol	Meaning
$x(t) = \mathcal{L}^{-1}(X(s))$	$X(s)$ is the Laplace Transform of $x(t)$
$x[n] = \mathcal{Z}^{-1}(X(z))$	$X(z)$ is the Z-Transform of $x[n]$
$x[n] = {}_{\mathrm{m}}\mathcal{W}^{-1}(c[k])$	$x[n]$ is the m-level inverse Wavelet-Transform of $c[k]$
$x[n] = {}_{\mathrm{m}}\mathcal{H}^{-1}(c[k])$	$x[n]$ is the m-level inverse Haar-Transform of $c[k]$
$x[n] = {}_{\mathrm{m}}^{1}\mathcal{D}^{-1}(c[k])$	$x[n]$ is the m-level inverse Daubechies-l -Transform of $c[k]$
$x[n] = {}_{\mathrm{m}}^{1}\mathcal{C}^{-1}(c[k])$	$x[n]$ is the m-level inverse Coifman-l -Transform of $c[k]$
Mapping symbols	
$x(t) \xrightarrow{\text{FS}} C_k$	C_k are the harmonic numbers of the Fourier Series of $x(t)$
$x(t) \xrightarrow{\mathcal{F}} X(\omega)$	$X(\omega)$ is the Fourier Transform of $x(t)$
$x[n] \xrightarrow{\text{DtFT}} X(\omega)$	$X(\omega)$ is the Discrete-time Fourier Transform of $x[n]$
$x[n] \xrightarrow{\text{DFT}} X[k]$	$X[k]$ is the Discrete Fourier Transform of $x[n]$
$x(t) \xrightarrow{\mathcal{L}} X(s)$	$X(s)$ is the Laplace Transform of $x(t)$
$x[n] \xrightarrow{\mathcal{Z}} X(z)$	$X(z)$ is the Z-Transform of $x[n]$
Morphological operations	
$x[n] \star y[n]$	convolution of $x[n]$ and $y[n]$
$x[n] \,☆\, y[n]$	correlation of $x[n]$ and $y[n]$
$A \ominus B$	erosion of binary image A by structuring element B
$A \oplus B$	dilation of binary image A by structuring element B
$A \circ B$	opening of binary image A by structuring element B
$A \bullet B$	closing of binary image A by structuring element B
$f[x,y] \ominus b[x,y]$	erosion of intensity image $f[x,y]$ by structuring element $b[x,y]$
$f[x,y] \oplus b[x,y]$	dilation of intensity image $f[x,y]$ by structuring element $b[x,y]$
$f[x,y] \circ b[x,y]$	opening of intensity image $f[x,y]$ by structuring element $b[x,y]$
$f[x,y] \bullet b[x,y]$	closing of intensity image $f[x,y]$ by structuring element $b[x,y]$
$A \cap B$	intersection of binary images A and B
$A \cup B$	union of binary images A and B
$f[x,y] \wedge g[x,y]$	intersection of intensity images f and g
$f[x,y] \vee g[x,y]$	union of intensity images f and g
DAC/ADC related acronyms	
DNL	Differential nonlinearity
INL	Integral nonlinearity

continued on next page

continued from previous page

Symbol	Meaning
DAC	Digital to Analog Converter
ADC	Analog to Digital Converter
S/N	Signal to Noise ratio
THD	Total Harmonic Distorsion
RMS	Root-Mean Square value
SNAD	Signal to Noise and Distorsion ratio
SFDR	Spurious Free Dynamic Range

Introduction

For an introduction on Digital Signal Processing, please, refer to the first volume in this series, *Digital Signal Processing — Signals and Transformations*.

In that volume, we focussed on the mathematical background required to study digital signal processing systems. We studied signals and signal transforms. Given that knowledge, we are now able to study signal processing systems.

We will start by studying systems from a generic point of view. Then, we will study digital convolution in detail. This automatically leads to a treatment of linear time-invariant digital filtering and the two basic types to implement those filters, using finite impulse response systems and infinite impulse response systems.

Next, we will extend our knowledge of signal transforms, by taking a look at two advanced signal transforms: the Short-time Fourier transform and the wavelet transform.

Finally, we will treat some capita selecta from the vast book DSP.

Systems, convolution and filtering

Chapter 2

Systems

In this chapter, you will learn about:

- electronic systems,
- how to classify systems using their properties,
- the importance of linear time-invariant systems,
- how to describe the latter using their impulse response and their transfer function in the time and the frequency domain,
- basic forms of LTI systems.

After having read this chapter, some questions will still be left unanswered:

- how do I design such systems?

After having read/studied this chapter, you are expected to be able to

- calculate the impulse response and the transfer function when given a system description and vice versa,
- classify electronic systems (regarding dimensionality, stability, causality, linearity, time-invariance, impulse response length, recursiveness, a.o.) when given a system description or some input and corresponding output signals,
- understand the relevance and the consequences of such a classification,
- draw the basic forms of LTI systems.

2.1 Introduction

Continuous and discrete-time systems are very similar. Most — if not all — of the definitions and properties are independent of the nature of the system: the math looks a little different, but that's mainly because we've chosen different symbols for the two domains. We therefore treat both types of systems in a single pass. Many of the equations will be presented in two columns: the left column for the continuous-time domain, the right column for the discrete-time domain. In addition, if we want to denote a signal regardless whether it is a continuous-time or a discrete-time signal, we'll just write x referring to both $x(t)$ and $x[n]$.

2.2 Definition and representation

A system is an entity H that transforms a set of input signals into a set of output signals. Mathematically:

$$\left.\begin{array}{r} x_1(t) \\ x_2(t) \\ \vdots \\ x_q(t) \end{array}\right\} \xrightarrow{H} \left\{\begin{array}{l} y_1(t) \\ y_2(t) \\ \vdots \\ y_r(t) \end{array}\right. \qquad \left.\begin{array}{r} x_1[n] \\ x_2[n] \\ \vdots \\ x_q[n] \end{array}\right\} \xrightarrow{H} \left\{\begin{array}{l} y_1[n] \\ y_2[n] \\ \vdots \\ y_r[n] \end{array}\right.$$

or in vector notation:

$$\vec{x}(t) \xrightarrow{H} \vec{y}(t) \qquad \vec{x}[n] \xrightarrow{H} \vec{y}[n]$$

Using block diagram symbols, we can represent this graphically as in Figure 2.1.

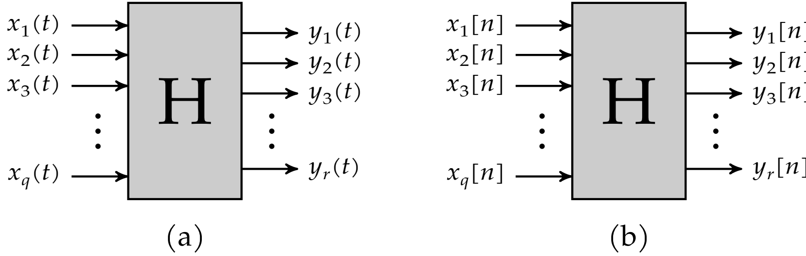


Figure 2.1: Graphical representation of a system as a block diagram for (a) a continuous-time system, and (b) a discrete-time system

2.3 Classification (I)

Using some basic properties, we can classify the generic *system* concept into some basic categories.

2.3.1 MIMO, MISO, SIMO and SISO systems

Depending on the number of inputs and outputs, we can distinguish:

- MIMO: multiple input, multiple output system
- MISO: multiple input, single output system
- SIMO: single input, multiple output system
- SISO: single input, single output system

The generic system represented in Figure 2.1 is a MIMO system. A MIMO system can be considered to be a composition of r MISO systems. Likewise, a SIMO system can be considered to be composition of r SISO systems. Therefore, from a mathematical point of view, we only need to study MISO and SISO systems.

2.3.2 Stable vs. nonstable systems

Consider a sytem H that maps $\vec{x} \xrightarrow{H} \vec{y}$. The system H is stable if any bounded input corresponds to a bounded output.

$$H \text{ is stable} \iff \left\| \vec{x}(t) \right\| < +\infty, \forall t \in \mathbb{R} \Downarrow \left\| \vec{y}(t) \right\| < +\infty, \forall t \in \mathbb{R}$$

$$H \text{ is stable} \iff \left\| \vec{x}[n] \right\| < +\infty, \forall n \in \mathbb{Z} \Downarrow \left\| \vec{y}[n] \right\| < +\infty, \forall n \in \mathbb{Z}$$

Any suitable norm definition will do for $\|\cdot\|$. For example,

- the 1-norm (the so-called Manhattan norm):

$$\left\| [x_1 x_2 x_3 \cdots x_l] \right\|_1 = |x_1| + |x_2| + |x_3| + \cdots + |x_l|$$

- the 2-norm (the so-called Euclidean norm):

$$\left\| [x_1 x_2 x_3 \cdots x_l] \right\|_2 = \sqrt{x_1^2 + x_2^2 + x_3^2 + \cdots x_l^2}$$

- the ∞-norm:

$$\left\| [x_1 x_2 x_3 \cdots x_l] \right\|_\infty = \max_{i=1}^{l} |x_i|$$

2.3.3 Linear vs. nonlinear systems

Usually the linearity property is split into two aspects:

- homogeneity
- superposition

Homogeneity A system H is called *homogeneous* if for an arbitrary signal $\vec{x}$ the following relationship holds:

$$\forall \vec{x}, \forall a \in \mathbb{R} : \qquad \vec{x} \xrightarrow{H} \vec{y} \quad \Longrightarrow \quad a\vec{x} \xrightarrow{H} a\vec{y}$$

Superposition principle A system H is said to obey the *superposition principle* if for any two arbitrary signals $\vec{x}_1$ and $\vec{x}_2$ the following relationship holds:

$$\forall \vec{x}_1, \vec{x}_2 : \qquad \left. \begin{array}{l} \vec{x}_1 \xrightarrow{H} \vec{y}_1 \\ \vec{x}_2 \xrightarrow{H} \vec{y}_2 \end{array} \right\} \quad \Longrightarrow \quad \vec{x}_1 + \vec{x}_2 \xrightarrow{H} \vec{y}_1 + \vec{y}_2$$

Linearity In short, if a system is homogeneous and it obeys the superposition principle, then the system is said to be *linear*.

This definition boils down to the following: consider two arbitrary signals $\vec{x}_1$ and $\vec{x}_2$, two arbitrary real numbers a and b, and a system H; the system H is said to be *linear* if the following holds

$$\forall \vec{x}_1, \vec{x}_2, \forall a, b \in \mathbb{R} : \left.\begin{array}{l} \vec{x}_1 \xrightarrow{H} \vec{y}_1 \\ \vec{x}_2 \xrightarrow{H} \vec{y}_2 \end{array}\right\} \implies a\vec{x}_1 + b\vec{x}_2 \xrightarrow{H} a\vec{y}_1 + b\vec{y}_2$$

Example 1 Consider a continuous-time system H_1:

$$x(t) \xrightarrow{H_1} y(t) = \frac{d^2x(t)}{dt^2} - 5x(t)$$

Is this a linear system? Yes, it is:

Homogeneity The system is homogeneous as:

$$\begin{aligned} ax(t) &\xrightarrow{H_1} \frac{d^2}{dt^2}\left(ax(t)\right) - 5\left(ax(t)\right) \\ &\xrightarrow{H_1} a\left(\frac{d^2}{dt^2}x(t) - 5x(t)\right) = ay(t) \end{aligned}$$

Superposition The system obeys superposition. Consider $x_1(t)$ and $x_2(t)$ and let's apply the sum of them to the system's input:

$$\begin{aligned} x_1(t) + x_2(t) &\xrightarrow{H_1} \frac{d^2}{dt^2}\left(x_1(t) + x_2(t)\right) - 5\left(x_1(t) + x_2(t)\right) \\ &\xrightarrow{H_1} \frac{d^2}{dt^2}x_1(t) + \frac{d^2}{dt^2}x_2()t - 5x_1(t) - 5x_2(t) \\ &\xrightarrow{H_1} \frac{d^2}{dt^2}x_1(t) - 5x_1(t) + \frac{d^2}{dt^2}x_2(t) - 5x_2(t) = y_1(t) + y_2(t) \end{aligned}$$

Example 2 Consider a discrete time system H_2:

$$x[n] \xrightarrow{H_2} y[n] = 7x^2[n]$$

Is this a linear system? No, it is not:

Homogeneity The system is inhomogeneous as:

$$\begin{aligned} ax[n] &\xrightarrow{H_2} 7\left(ax[n]\right)^2 \\ &\xrightarrow{H_2} a^2 7x^2[n] \\ &\xrightarrow{H_2} a^2 y[n] \neq ay[n] \end{aligned}$$

Superposition The system does not obey superposition. Consider $x_1[n]$ and $x_2[n]$ and let's apply the sum of them to the system's input:

$$\begin{aligned} x_1[n] + x_2[n] &\xrightarrow{H_2} 7\left(x_1[n] + x_2[n]\right)^2 \\ &\xrightarrow{H_2} 7x_1^2[n] + 14x_1[n]x_2[n] + 7x_2^2[n] \\ &\xrightarrow{H_2} y_1[n] + y_2[n] + 14x_1[n]x_2[n] \neq y_1[n] + y_2[n] \end{aligned}$$

Example 3 Consider a discrete-time system H_3:

$$x[n] \xrightarrow{H_3} y[n] = nx[n+3]$$

Is this a linear system? Yes, it is:

Homogeneity The system is homogeneous as:

$$ax[n] \xrightarrow{H_3} nax[n+3] = ay[n]$$

Superposition The system obeys superposition. Consider $x_1[n]$ and $x_2[n]$ and let's apply the sum of them to the system's input:

$$\begin{aligned} x_1[n] + x_2[n] &\xrightarrow{H_3} n(x_1[n+3] + x_2[n+3]) \\ &\xrightarrow{H_3} nx_1[n+3] + nx_2[n+3]) = y_1[n] + y_2[n] \end{aligned}$$

Exercises
Some exercises on linearity

Exercise 2.3.3-1: Is the following system linear?

$$x(t) \xrightarrow{H} y(t) = \frac{5}{x(t)}$$

Exercise 2.3.3-2: Is the following system linear?

$$x[n] \xrightarrow{H} y[n] = 5\sin(x[n])$$

Exercise 2.3.3-3: Is the following system linear?

$$x[n] \xrightarrow{H} y[n] = 3nx[n]$$

Exercise 2.3.3-4: Is the following system linear?

$$x[n] \xrightarrow{H} y[n] = 0.5y[n-1] + 3x[n] - 2x[n-1]$$

2.3.4 Time invariance

A system H is said to be time-invariant if the following property holds:

$$\begin{array}{cc} \forall \tau \in \mathbb{R}: & \forall N \in \mathbb{Z}: \\ \vec{x}(t) \xrightarrow{H} \vec{y}(t) & \vec{x}[n] \xrightarrow{H} \vec{y}[n] \\ \Downarrow & \Downarrow \\ \vec{x}(t-\tau) \xrightarrow{H} \vec{y}(t-\tau) & \vec{x}[n-N] \xrightarrow{H} \vec{y}[n-N] \end{array}$$

Basically, this means that any signal applied to the system provokes the same response, irrespective of the moment in time the signal is applied.

Example 1 Consider a continuous-time system H_1:

$$x(t) \xrightarrow{H_1} y(t) = tx(t)$$

Is this a time-invariant system? No, it is not. Assume applying a delayed signal $x(t-\tau)$:

$$x(t-\tau) \xrightarrow{H_1} tx(t-\tau) \neq (t-\tau)x(t-\tau) = y(t-\tau)$$

Example 2 Consider a discrete-time system H_2:

$$x[n] \xrightarrow{H_2} y[n] = 2x[n] - 3x^2[n+5]$$

Is this a time-invariant system? Yes, it is. Assume applying an expedited signal $x[n+n_0]$:

$$x[n+n_0] \xrightarrow{H_2} 2x[n+n_0] - 3x^2[n+n_0+5] = y[n+n_0]$$

Exercises
Some exercises on time invariance

Exercise 2.3.4-1: Is the following system time invariant?

$$x(t) \xrightarrow{H} y(t) = \frac{5}{x(t)}$$

Exercise 2.3.4-2: Is the following system time invariant?

$$x[n] \xrightarrow{H} y[n] = 5\sin(x[n])$$

Exercise 2.3.4-3: Is the following system time invariant?

$$x[n] \xrightarrow{H} y[n] = 3nx[n]$$

Exercise 2.3.4-4: Is the following system time invariant?

$$x[n] \xrightarrow{H} y[n] = 0.5y[n-1] + 3x[n] - 2x[n-1]$$

2.4 LTI systems

Based on two of the properties described above, an interesting subclass of systems can be identified: the linear, time-invariant systems, LTI systems in short.

2.4.1 Benefits

There are many reasons why these systems are so attractive.

2.4.1.1 Predictability

The time invariance of a system makes the system predictable. If we test the behavior of a system by applying a (time-limited) specific input signal, then we know that the observed response of the system will reappear if the same signal is applied to the system on any future moment. This makes the system predictable.

2.4.1.2 Reduction of multiple-input systems

Consider an Q-dimensional input signal $\vec{x}$ that is applied to a Q-input MISO system:

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_q \end{bmatrix} \xrightarrow{H} y$$

This signal can be decomposed into Q "one-dimensional" input signals:

$$\vec{x} = \begin{bmatrix} x_1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ x_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ x_3 \\ \vdots \\ 0 \end{bmatrix} + \cdots + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ x_q \end{bmatrix} \tag{2.1}$$

As a linear system obeys the superposition principle, this means we can calculate the response y of the system by adding the partial responses y_i caused by the Q components of the decomposition of (2.1).

As all but one input is zero for those components, this means that we can analyze an Q-input MISO system by analyzing Q SISO systems. The resulting MISO system is just the superposition of the Q SISO systems.

Schematically this can be represented as follows:

$$\begin{array}{ccccccccccccc} \vec{x} & = & \begin{bmatrix} x_1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} & + & \begin{bmatrix} 0 \\ x_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} & + & \begin{bmatrix} 0 \\ 0 \\ x_3 \\ \vdots \\ 0 \end{bmatrix} & + & \cdots & + & \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ x_q \end{bmatrix} \\ | & & | & & | & & | & & & & | \\ H & & H_1 & & H_2 & & H_3 & & \cdots & & H_q \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & & & \downarrow \\ y & = & y_1 & + & y_2 & + & y_3 & + & \cdots & + & y_q \end{array}$$

2.4.1.3 Decomposition

Consider applying an input signal x to an LTI SISO system H, resulting in an output signal y:

$$x \xrightarrow{H} y$$

As the system H is linear, we can analyze the system's output by decomposing the input signal and applying superposition on the components after they were treated by the system.

An interesting decomposition is the frame decomposition. This decomposition technique allows chopping a time-unlimited signal into an infinite but countable number of time-limited frames. Divide and conquer is one of the most basic engineering techniques. Frame decomposition is such a technique.

Another particularly interesting decomposition is the impulse decomposition. In view of the importance of this impulse decomposition technique for LTI systems, we will devote the next full subsection to the subject.

2.4.2 System description

So, let's elaborate the impulse decomposition a little further. Let's start by considering the time domain. In a second phase, we will consider the frequency domain.

2.4.2.1 Time-domain system description

Because of the inherent simplicity of the impulse decomposition of discrete-time signals, we'll start by analyzing these first.

a) Discrete-time systems Mathematically, the discrete-time impulse decomposition boils down to:

$$x[n] = \sum_{i=-\infty}^{+\infty} x[i]\delta[n-i]$$

In a similar fashion as in section 2.4.1.2 on the preceding page, we can calculate the response $y[n]$ as follows:

$$\begin{array}{ccccccccccccc}
x[n] & = & \cdots & + & x[-1]\delta[n+1] & + & x[0]\delta[n-0] & + & x[1]\delta[n-1] & + & x[2]\delta[n-2] & + & \cdots \\
| & & & & | & & | & & | & & | & & \\
H & & & & H & & H & & H & & H & & \\
\downarrow & & & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\
y[n] & = & \cdots & + & y_{-1}[n] & + & y_0[n] & + & y_1[n] & + & y_2[n] & + & \cdots
\end{array}$$

Note that the output is a combination of responses to scaled, time-shifted unit impulses. Taking into account that the system is homogeneous (i.e. invariant to scaling) and

time-invariant (i.e. invariant to time shifts), we only need to study the system's reponse to a unit impulse appearing at $n = 0$.

Now, let's denote the system's reponse to a unit impulse as $h[n]$, the so-called system's *impulse response*:

$$\delta[n] \xrightarrow{H} h[n]$$

This allows to write the system's reponse to the input signal $x[n]$ as:

$$\begin{aligned} y[n] &= \sum_{i=-\infty}^{+\infty} x[i]h[n-i] \\ &= h[n] \star x[n] \end{aligned} \tag{2.5}$$

This is a most important conclusion:

> **Impulse response description of a discrete-time LTI system** *The response of a linear, time-invariant system can be calculated as the convolution of that signal with the impulse response of the system.*

Elaborating (2.4.2.1) a littlebit further, makes the decomposition even more clear:

$$\begin{array}{ccccccccccccc} x[n] & = & \cdots & + & x[-1]\delta[n+1] & + & x[0]\delta[n-0] & + & x[1]\delta[n-1] & + & x[2]\delta[n-2] & + & \cdots \\ | & & & & | & & | & & | & & | & & \\ H & & & & H & & H & & H & & H & & \\ \downarrow & & & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\ y[n] & = & \cdots & + & x[-1]h[n+1] & + & x[0]h[n-0] & + & x[1]h[n-1] & + & x[2]h[n-2] & + & \cdots \end{array}$$

b) Continuous-time systems In a similar way, we can apply the idea of impulse decomposition to a continuous-time system.

Mathematically, the continuous-time impulse decomposition boils down to:

$$x(t) = \int_{-\infty}^{+\infty} x(\tau)\delta(t-\tau)\,\mathrm{d}\tau \tag{2.6}$$

If we interpret the integral in (2.6) as a Riemann sum, we can consider $x(t)$ to be a superposition of shifted, scaled impulses. Knowing that our system is linear and time invariant, we again can restrict our system analysis to analyzing its impulse reponse.

Let's denote the system's reponse to a unit impulse $\delta(t)$ as $h(t)$, the so-called system's *impulse response*:

$$\delta(t) \xrightarrow{H} h(t)$$

This allows to write the system's reponse to the input signal $x(t)$ as:

$$\begin{aligned} y(t) &= \int_{-\infty}^{+\infty} x(\tau)h(t-\tau)\,\mathrm{d}\tau \\ &= h(t) \star x(t) \end{aligned} \tag{2.7}$$

This, again, is a most important conclusion:

> **Impulse response description of a continuous-time LTI system** *The response of a linear, time-invariant system can be calculated as the convolution of that signal with the impulse response of the system.*

2.4.2.2 Frequency-domain system description

Let's again start by considering discrete-time systems.

a) Discrete-time systems Let's start from the time-domain description that we've derived for LTI systems in section 2.4.2.1 on page 12.

$$y[n] = h[n] \star x[n]$$

As we know, time-domain convolution corresponds to frequency-domain multiplication. Therefore, applying the Discrete-time Fourier transform to (2.4.2.1), results in:

$$Y_p(\omega) = H_p(\omega)X_p(\omega)$$

and similarly, using the Z-transform, we can write:

$$Y(z) = H(z)X(z)$$

or alternatively:

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

The function $H(z)$ is the so-called *transfer function*. It is the Z-transform of the discrete-time impulse response.

> **Transfer function description of a discrete-time LTI system** *The frequency-domain response of a linear, time-invariant system can be calculated as the multiplication of the signal's frequency-domain description with the transfer function.*

b) Continuous-time systems Let's start from the time-domain description that we've derived for LTI systems in section 2.4.2.1 on the preceding page.

$$y(t) = h(t) \star x(t)$$

As we know, time-domain convolution corresponds to frequency-domain multiplication. Therefore, applying the Fourier transform to (2.7), results in:

$$Y(\omega) = H(\omega)X(\omega)$$

and similarly, using the Laplace transform, we can write:

$$Y(s) = H(s)X(s)$$

or alternatively:

$$H(s) = \frac{Y(s)}{X(s)}$$

The function $H(s)$ is the so-called *transfer function*. It is the Laplace transform of the continuous-time impulse response.

Transfer function description of an continuous-time LTI system *The frequency-domain response of a linear, time-invariant system can be calculated as the multiplication of the signal's frequency-domain description with the transfer function.*

Remarks Note that the concept of a transfer function is only valid for LTI systems. However, the concept of an impulse response is more general. Regardless whether the system is LTI or not, one can still apply an impulse to a system and observe the corresponding response. However, for non-LTI systems, one cannot use that response to convolve it with the input signals to obtain the real output signal. The result obtained in that way will be bogus.

2.5 Classification (II)

Our new insights into the time-domain impulse reponse description of LTI systems, allow classifying those systems a little bit further.

2.5.1 Causal vs. noncausal systems

A system H is said to be causal if and only if its impulse response is zero for negative time points.

$$H \text{ is causal} \quad\Updownarrow\quad \forall t < 0 : h(t) = 0 \qquad\qquad H \text{ is causal} \quad\Updownarrow\quad \forall n < 0 : h[n] = 0$$

Remarks

1. As the concept of an impulse response is not limited to the scope of LTI systems, also the concept of causality is a more general concept.
2. All physical systems known so far (irrespective whether they are LTI or not) are causal. In plain English: a system cannot respond before it gets excited. Alternatively: a system cannot respond to data that will appear in the future.

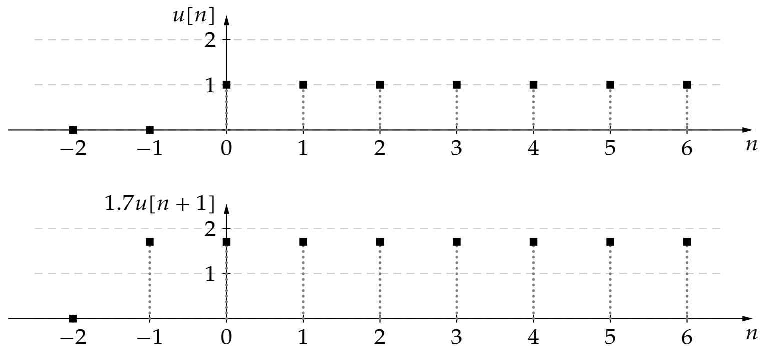
Example 1 Consider a discrete-time system H_1:

$$x[n] \xrightarrow{H_1} y[n] = 1.7x[n+1]$$

Is this system causal? No, as one can easily check, applying a unit step (Heaviside function) $u[n]$ to the system causes the system to respond with a unit step that leads the original one.

$$u[n] \xrightarrow{H_1} 1.7u[n+1]$$

Graphically:



Example 2 Consider a discrete-time system H_2:

$$x[n] \xrightarrow{H_2} y[n] = x[n] + 7x^2[n-1] - 3x[n-2] + 2x^{-1}[n-3] + x[n-4]$$

Is this system causal? Yes, it is. For a time point n the system only uses data that are in the past (at time points $n, n-1, ..., n-4$).

Exercises
Some exercises on causality

Exercise 2.5.1-1: Is the following system causal?

$$x[n] \xrightarrow{H} y[n] = 0.5y[n-1] - 1.4y[n-2] + 2x[n]$$

Exercise 2.5.1-2: Is the following system causal?

$$x[n] \xrightarrow{H} y[n] = -1.1x[n-1] + 2.5x^2[n-2]$$

Exercise 2.5.1-3: Is the following system causal?

$$x[n] \xrightarrow{H} y[n] = -1.4y[n+1] + 2x[n]$$

Exercise 2.5.1-4: Is the following system causal?

$$x[n] \xrightarrow{H} y[n] = 0.5y[n] - 1.4y[n-1] + 2x[n+1]$$

Exercise 2.5.1-5: Is the LTI-system described by the following transfer function $H(z)$ causal?

$$H(z) = 1 - 0.2z^{-1} + 4.3z^{-2}$$

Exercise 2.5.1-6: Is the LTI-system described by the following transfer function $H(z)$ causal?

$$H(z) = z^2 + 0.34z - 1 - 0.2z^{-1} + 4.3z^{-2}$$

Exercise 2.5.1-7: Is the LTI-system described by the following transfer function $H(z)$ causal?

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

Exercise 2.5.1-8: Is the LTI-system described by the following transfer function $H(z)$ causal?

$$H(z) = \frac{z^2 + 3z + 1}{3.2z^2 - 1}$$

Exercise 2.5.1-9: Is the LTI-system described by the following transfer function $H(z)$ causal?

$$H(z) = \frac{z - 2}{z^2 - 3z + 1.23}$$

Exercise 2.5.1-10: (*) Is the following system causal?

$$x[n] \xrightarrow{H} y[n] = -0.1y[n] + y[n-1] - 0.5x[n-2]$$

2.5.2 FIR vs. IIR systems

We can classify LTI systems further based on the length of the impulse response.
If the impulse response length is infinite, we label the system as an *Infinite Impulse Response* (*IIR*) system.
If the impulse response length is finite, we label the system as a *Finite Impulse Response* (*FIR*) system.

Remarks

1. Though the concept of impulse reponse is more general than the LTI systems scope, the notion of FIR and IIR does not extend beyond LTI systems. For non-LTI systems the impulse reponse can be finite or infinite depending on the amplitude of the impulse or the moment in time on which the impulse is applied.
2. Though, in theory, FIR systems are viable systems in both the discrete-time and continuous-time domain, in practice, continuous-time FIR systems do not exist. The possibility of making FIR systems in the discrete-time domain is one of the core competitive elements that make discrete-time systems so unique (and attractive). We will see why later on.

2.6 Building causal LTI systems

When building analog systems, we're bound by the rules of physics (or electronics to be more specific). When building digital systems, we're only bound by the rules of mathematics, i.e. a much more liberal system.

In addition, as this is a course on DSP, let's first take a look at the digital side.

2.6.1 Discrete-time causal LTI systems

Let's face the problem of building a causal discrete-time signal processing system from the perspective of the signal processing system itself: "how can it produce the next output sample, only knowing the input samples that appeared so far".

First version The obvious idea is: combine the input samples that appeared so far to generate the next output sample. Nice, but of course the system better be linear and time invariant.

From the examples that appeared so far, you probably figured out yourself that there are only few operations that maintain LTI properties:

1. addition of (delayed) samples,
2. multiplication with a scalar.

What we certainly should not do is:

1. multiply samples (destroys linearity),
2. multiply with time (destroys time-invariance).

System description Exploiting these ideas, yields our first discrete-time system description:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] \tag{2.10}$$

in which M is a positive integer. In theory M could be infinite, but that destroys the feasibility of the system. Indeed, such a system would require infinite memory and an infinite number of adders and scalers.

Impulse response Finding the impulse response belonging to this system is most easy: trigger it with a Dirac impulse! The result is the time sequence:

$$\delta[n] \xrightarrow{H} h[n] = \begin{cases} 0, & \text{if } n < 0 \\ b_n & \text{if } 0 \leq n \leq M \\ 0, & \text{if } M < n \end{cases} \tag{2.11}$$

or in tabular format:

n	$\cdots$	-2	-1	0	1	2	3	$\cdots$	M	$M+1$	$\cdots$
$h[n]$	0	0	0	b_0	b_1	b_2	b_3	$\cdots$	b_M	0	0

Note that this impulse response is finite. Hence, we call the system a *finite impulse response system* (*FIR system*).

Transfer function So, how about the transfer function in the Z-domain? Applying the Z-transform to (2.11) is most easy:

$$H(z) = \mathcal{Z}(h[n]) = b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} + \cdots + b_M z^{-M} \tag{2.12}$$

Alternatively, we may derive $H(z)$ from (2.10) by Z-transforming both members of the equation:

$$\begin{aligned} \mathcal{Z}(y[n]) &= \mathcal{Z}\left(\sum_{k=0}^{M} b_k x[n-k]\right) \\ Y(z) &= \sum_{k=0}^{M} b_k \mathcal{Z}(x[n-k]) \\ &= \sum_{k=0}^{M} b_k z^{-k} X(z) \end{aligned}$$

leading to:

$$H(z) = \frac{Y(z)}{X(z)} = \sum_{k=0}^{M} b_k z^{-k}$$

which is — as expected — identical to (2.12).

Second version We can go even further: so far we only used delayed input samples to compose the next output sample; how about also using delayed output samples?

System description Exploiting this idea, yields our second discrete-time system description:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] + \sum_{l=1}^{N} a_l y[n-l] \tag{2.13}$$

in which M and N are positive integers. In theory they could be infinite, but that destroys the feasibility of the system. Indeed, such a system would require infinite memory and an infinite number of adders and scalers.

Impulse response Finding the impulse response belonging to this system is not so easy: triggering it with a Dirac impulse, yields a rather involved time sequence. In tabular format:

n	$h[n]$
$\vdots$	0
-2	0
-1	0
0	b_0
1	$b_1 + a_1 b_0$
2	$b_2 + a_2 b_0 + a_1(b_1 + a_1 b_0)$
3	$b_3 + a_3 b_0 + a_2(b_1 + a_1 b_0) + a_1(b_2 + a_2 b_0 + a_1(b_1 + a_1 b_0))$
$\vdots$	$\vdots$

Note that in general this impulse response is infinite. Hence, we call the system an *infinite impulse response system* (*IIR system*).

Transfer function So, how about the transfer function in the Z-domain? Applying the Z-transform to the impulse response will not give us a closed-form solution. However, we may derive $H(z)$ from (2.13) by Z-transforming both members of the equation:

$$\begin{aligned}\mathcal{Z}(y[n]) &= \mathcal{Z}\left(\sum_{k=0}^{M} b_k x[n-k] + \sum_{l=1}^{N} a_l y[n-l]\right)\\ Y(z) &= \sum_{k=0}^{M} b_k \mathcal{Z}(x[n-k]) + \sum_{l=1}^{N} a_l \mathcal{Z}(y[n-l])\\ &= \sum_{k=0}^{M} b_k z^{-k} X(z) + \sum_{l=1}^{N} a_l z^{-l} Y(z)\end{aligned}$$

leading to:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{l=1}^{N} a_l z^{-l}}$$

2.6.2 Continuous-time causal LTI systems

We can repeat the same mathematical derivation for continuous-time causal LTI systems (and we will do so!). However, we don't have the same mathematical freedom. All continuous-time systems are subject to the laws of physics, in the case of electronic systems these are Maxwell's laws, or, in the case of lumped circuit theory, Kirchhoff's laws. These laws prohibit us of making finite impulse response systems in practice.

The law of action and reaction in combination with the fact that even a pure resistive circuit suffers some reactive parasitics in reality, makes that the first part is only useful from a theoretical point of view.

First version The obvious idea is: combine the input signal and delayed versions of it to generate the output signal. Nice, but of course the system better be linear and time invariant. Therefore, again we restrict ourselves to linear operations:

1. derivation in time,
2. multiplication with a scalar,
3. addition.

What we certainly should not do is:

1. multiply signals (destroys linearity),
2. multiply signals with time (destroys time-invariance).

Hey, wait a minute. Why don't we consider time delay as we did for the discrete-time systems? The reason is very simple. Though time-delay is a linear operation, it is impossible to realize in practice. This would mean storing an exact copy of the signal for "replay" later on. Alas, nature does not offer us such a capabilities.

System description Exploiting this idea, yields our first continuous-time system description:

$$y(t) = \sum_{k=0}^{M} b_k \frac{d^k x(t)}{dt^k} \tag{2.14}$$

in which M is a positive integer. In theory M could be infinite, but that destroys the feasibility of the system. Indeed, such a system would require infinite memory (i.e. an infinite number of energy storage elements like inductors or capacitors) and an infinite number of adders and scalers.

Impulse response Finding the impulse response belonging to this system is not so trivial. It would require calculus involving the first (and higher-order) derivatives of a Dirac impulse. In order to make this purely theoretical section not needlessly more complicated, we'd rather not go there. Therefore, let's skip this step.

Transfer function So, how about the transfer function in the s-domain? Applying the Laplace transform to both members of (2.14) yields

$$\begin{aligned}
\mathcal{L}(y(t)) &= \mathcal{L}\left(\sum_{k=0}^{M} b_k \frac{d^k x(t)}{dt^k}\right) \\
Y(s) &= \sum_{k=0}^{M} b_k \mathcal{L}\left(\frac{d^k x(t)}{dt^k}\right) \\
&= \sum_{k=0}^{M} b_k s^k X(s)
\end{aligned}$$

leading to:

$$H(s) = \frac{Y(s)}{X(s)} = \sum_{k=0}^{M} b_k s^k$$

In the above, we assumed the bilateral Laplace transform, or the unilaterial transform with zero initial conditions.

Second version Now, back to the real world. Any system's output will be influenced by derivatives of that output.

System description Exploiting this idea, we obtain our second continuous-time system description:

$$y(t) = \sum_{k=0}^{M} b_k \frac{\mathrm{d}^k x(t)}{\mathrm{d}t^k} + \sum_{l=1}^{N} a_l \frac{\mathrm{d}^l y(t)}{\mathrm{d}t^l} \tag{2.15}$$

in which M and N are positive integers. In theory they could be infinite, but that destroys the feasibility of the system. Indeed, such a system would require infinite memory (i.e. an infinite number of energy storage elements like inductors and capacitors) and an infinite number of adders and scalers.

Impulse response Finding the impulse response belonging to this system is not so trivial. It would require calculus involving the first (and higher-order) derivatives of a Dirac impulse. We'd rather not go there. Therefore, let's skip this step, just for now.

Transfer function So, how about the transfer function in the s-domain? We can derive $H(s)$ from (2.15) by Laplace transforming both members of the equation:

$$\begin{aligned}\mathcal{L}(y(t)) &= \mathcal{L}\left(\sum_{k=0}^{M} b_k \frac{\mathrm{d}^k x(t)}{\mathrm{d}t^k} + \sum_{l=1}^{N} a_l \frac{\mathrm{d}^l y(t)}{\mathrm{d}t^l}\right) \\ Y(s) &= \sum_{k=0}^{M} b_k \mathcal{L}\left(\frac{\mathrm{d}^k x(t)}{\mathrm{d}t^k}\right) + \sum_{l=1}^{N} a_l \mathcal{L}\left(\frac{\mathrm{d}^l y(t)}{\mathrm{d}t^l}\right) \\ &= \sum_{k=0}^{M} b_k s^k X(s) + \sum_{l=1}^{N} a_l s^l Y(s)\end{aligned}$$

leading to:

$$H(s) = \frac{Y(s)}{X(s)} = \frac{\sum_{k=0}^{M} b_k s^k}{1 - \sum_{l=1}^{N} a_l s^l}$$

Impulse response (2) The impulse response can now be found (in case $M < N$) by partial fraction decomposition of the transfer function, followed by a simple term-by-term inverse Laplace transform. This is a technique that should be very familiar to you.

2.7 Classification (III)

2.7.1 Recursive vs. nonrecursive systems

Take a look at the system description of the causal LTI discrete-time systems in (2.10) and (2.13).

The latter shows an output which is dependent on a delayed version of itself. It is defined in a *recursive* way. We call this a *recursive system*.

The former shows an output that is *not* dependent on a delayed version of itself. We therefore call this a *nonrecursive system*.

Remarks

1. In the definition given above, we only referred to discrete-time systems for the simple reason that in practice continuous-time systems all are recursive systems. Therefore, the distinction is not very often made in the analog domain.
2. Note that very often a recursive system exhibits an infinite impulse response. However, this is not necessarily so. It is possible to conceive recursive systems that exhibit a finite impulse response (e.g., a recursive implementation of a moving average filter). On the contrary, it is not possible to conceive a nonrecursive system that has an infinite impulse response.
3. Nonrecursive systems have a very beneficial property w.r.t. to stability: they are 100% guaranteed stable. How come? The output is a linear combination of input samples. If all these samples are bounded, than also the output is bounded. In a recursive filter, we may suffer a run-away of the output due to the memory effect that is inherent to recursive filters. The following system exhibits such a run-away phenomenon. Just apply a Dirac impulse to it and see what happens!
$$x[n] \xrightarrow{H} y[n] = x[n] + 2y[n-1]$$
Can you explain this instability by analyzing the poles and zeros of the transfer function?

2.8 Basic forms of causal discrete-time LTI systems

It is easy to check in (2.10) and (2.13) that all we need in terms of hardware to build causal discrete-time LTI systems are:

1. unit delay elements,
2. scalers, and
3. adders.

Let's represent these signal-processing systems using block diagrams to see how the two prototype systems translate into hardware.

2.8.1 Nonrecursive systems

The system description we start from is:

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

2.8.1.1 Direct form

An obvious way to implement this system is displayed in Figure 2.2. This is the so-called *direct form*. As you can see, this implementation consists of a long delay chain (delaying the input

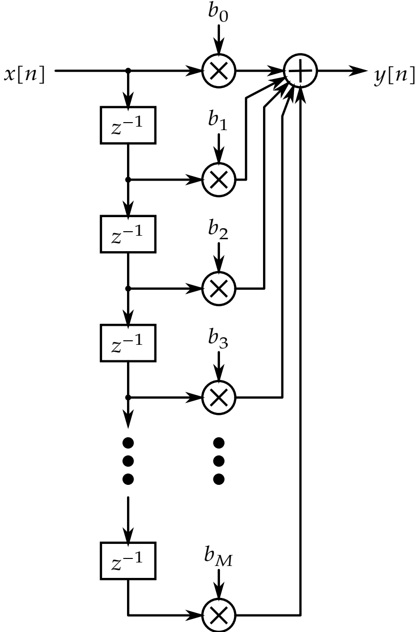


Figure 2.2: Block diagram of the *direct form* implementation of a nonrecursive system

signal $x[n]$) that is consistently *tapped* by the *filter coefficient* multipliers to be summed in the end and in this way form the output signal. Often, the number of *taps* is mentioned to indicate

the length of nonrecursive filters. Using this terminology, the figure displays an $M+1$-*tap* filter.

2.8.1.2 Transposed form

Let's consider the Z-transform of (2.10):

$$Y(z) = \sum_{k=0}^{M} b_k z^{-k} X(z)$$

We can partially factorize the right-hand side of this equation:

$$Y(z) = b_0 X(z) + z^{-1}\left(b_1 X(z) + z^{-1}\left(b_2 X(z) + z^{-1}\left(b_3 X(z) + z^{-1}\left(\cdots + z^{-1} b_M X(z)\right)\right)\right)\right)$$

This corresponds to the system implementation of Figure 2.3. This is the so-called *transposed form*. The name stems from the fact that this form can be obtained by *transposing* the direct form, i.e.

- reversing all signal flow arrows,
- replacing every summation by a node,
- replacing every node by a summation, and
- interchanging the role of $x[n]$ and $y[n]$.

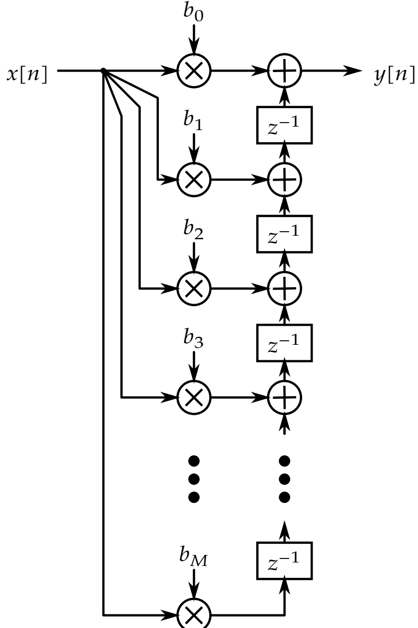


Figure 2.3: Block diagram of the *transposed form* implementation of a nonrecursive system

Again, the figure displays an $M+1$-tap filter.

2.8.1.3 Which form to use in practice?

From a computational complexity point of view, both forms are very similar: they require the same number of additions and multiplications and the same number of delay elements. If

infinite precision arithmetic is used, both implementations are equally well. In reality they differ (w.r.t. rounding). The transposed form is easier to parallellize in hardware. In software the type of processor determines our preference:

- in case of multicore or superscalar processors the transposed form is better;
- in a single-threaded processor or in serial hardware, the direct form is equally good;
- in an SIMD-processor, the direct form is better.

Exercises
Some exercises on the basic forms for nonrecursive systems

Exercise 2.8.1.3-1: Consider the LTI-system described by the following transfer function:

$$H(z) = 1 - 0.2z^{-1} + 0.3z^{-2} - 1.34z^{-3}$$

1. Draw the direct form implementation
2. Draw the transposed form implementation

Exercise 2.8.1.3-2: Consider the LTI-system described by the following transfer function:

$$H(z) = 1 - 3.14z^{-3}$$

1. Draw the direct form implementation
2. Draw the transposed form implementation

Exercise 2.8.1.3-3: Consider the LTI-system described by the following transfer function:

$$H(z) = 1.23z^{-2} - 0.013z^{-3}$$

1. Draw the direct form implementation
2. Draw the transposed form implementation

Exercise 2.8.1.3-4: Consider the LTI-system described by the following transfer function:

$$H(z) = z^{-2}\left(1 - 0.2z + 0.3z^{-1} - 1.34z^{-3}\right)$$

1. Draw the direct form implementation
2. Draw the transposed form implementation

2.8.2 Recursive systems

The system description we start from is:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] + \sum_{l=1}^{N} a_l y[n-l]$$

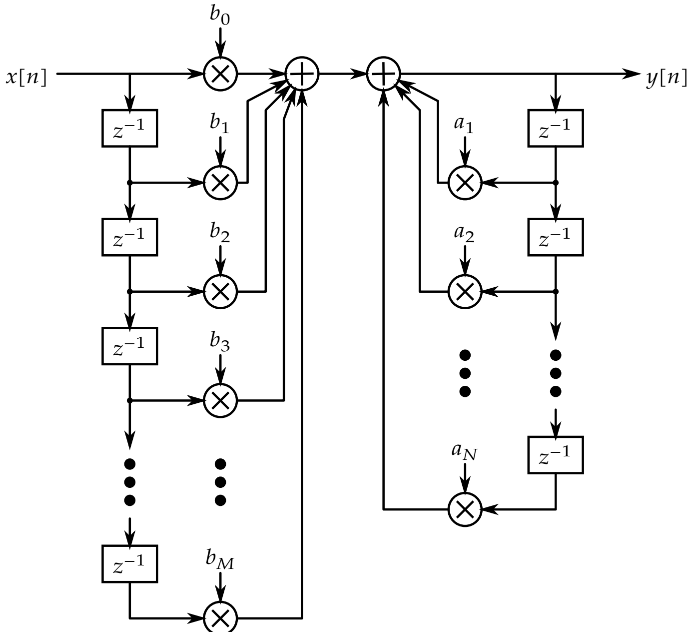


Figure 2.4: Block diagram of the *direct form I* implementation of a recursive system

2.8.2.1 Direct form I

An obvious way to implement this system is displayed in Figure 2.4. This is the so-called *direct form I* implementation.

2.8.2.2 Transposed form I

Let's consider the Z-transform of (2.13):

$$Y(z) = \sum_{k=0}^{M} b_k z^{-k} X(z) + \sum_{l=1}^{N} a_l z^{-l} Y(z)$$

We can factorize the right-hand side of this equation:

$$\begin{aligned} Y(z) = b_0 X(z) &+ z^{-1}\left(b_1 X(z) + z^{-1}\left(b_2 X(z) + z^{-1}\left(b_3 X(z) + z^{-1}\left(\cdots + z^{-1} b_M X(z)\right)\right)\right)\right) \\ &+ z^{-1}\left(a_1 Y(z) + z^{-1}\left(a_2 Y(z) + z^{-1}\left(a_3 Y(z) + z^{-1}\left(\cdots + z^{-1} a_N Y(z)\right)\right)\right)\right) \end{aligned}$$

This corresponds to the system implementation of Figure 2.5. This is the so-called *transposed form I*. The name stems from the fact that this form can be obtained by *transposing* the direct form I, i.e.

- reversing all signal flow arrows,
- replacing every summation by a node,
- replacing every node by a summation, and
- interchanging the role of $x[n]$ and $y[n]$.

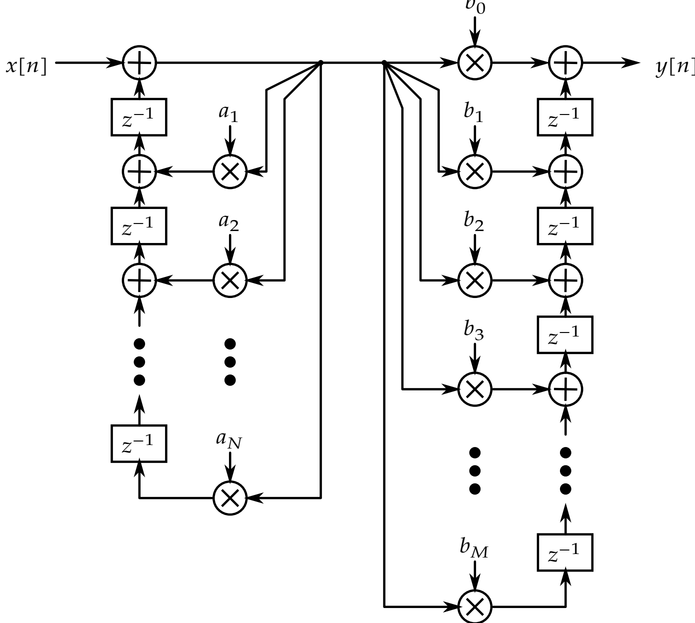


Figure 2.5: Block diagram of the *transposed form I* implementation of a recursive system

2.8.2.3 Direct form II

As the *direct form I* implementation is a cascaded connection of two parts (the a-chain and the b-chain), we can change the order of these parts without affecting the system's function.

This has been displayed in Figure 2.6 on the next page.

This arrangement can be further simplified by using the same delay elements for the a-chain and the b-chain. This has been depicted in Figure 2.7 on the following page. This is the so-called *direct form II* implementation. Note the assumption we made to ease the drawing of Figure 2.7: $M = N + 1$. This assumption was made completely arbitrarily.

The savings in terms of hardware are considerable: if N and M are of comparable size, we save about half of the delay elements!

2.8.2.4 Transposed form II

Let's consider the Z-transform of (2.13):

$$Y(z) = \sum_{k=0}^{M} b_k z^{-k} X(z) + \sum_{l=1}^{N} a_l z^{-l} Y(z)$$

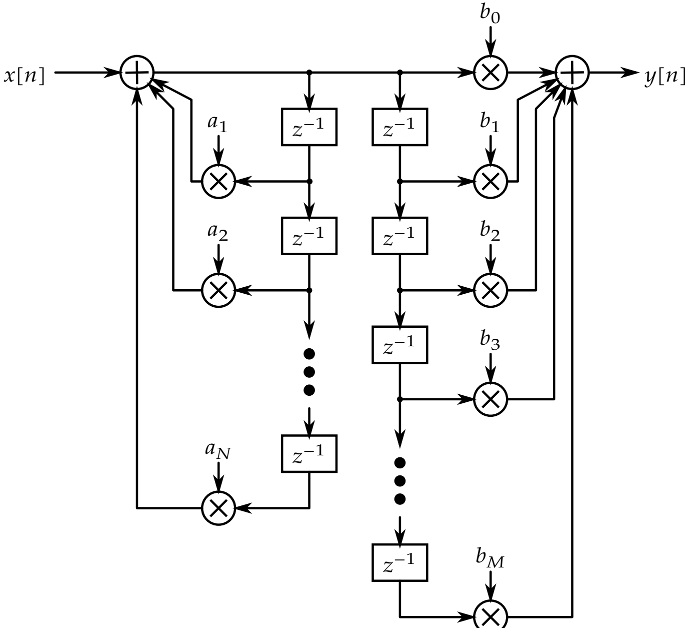


Figure 2.6: Block diagram of the *modified direct form I* implementation of a recursive system

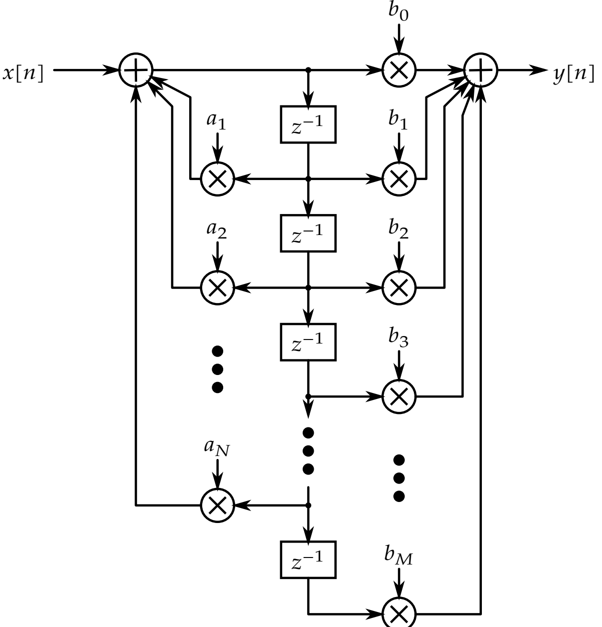


Figure 2.7: Block diagram of the *direct form II* implementation of a recursive system (assuming $M = N + 1$)

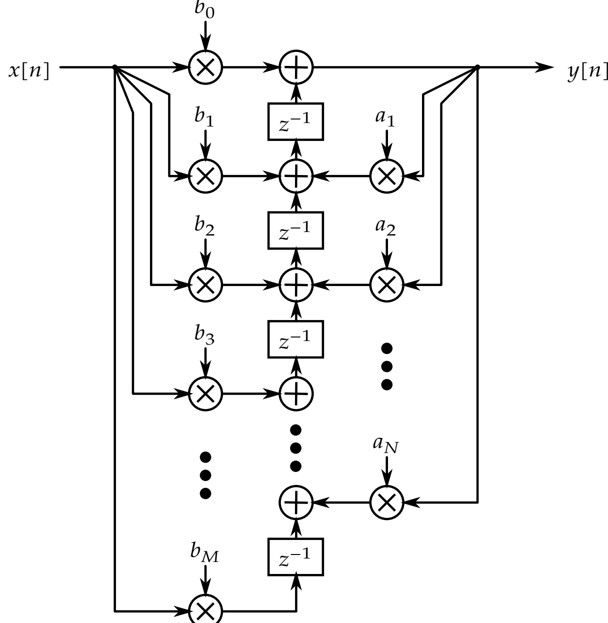


Figure 2.8: Block diagram of the *transposed form II* implementation of a recursive system

We factorized the right-hand side of this equation before (when treating transposed form I). However, we can take it a little further and rearrange the terms:[1]

$$
\begin{aligned}
Y(z) &= b_0X(z) + z^{-1}\left(b_1X(z) + z^{-1}\left(b_2X(z) + z^{-1}\left(b_3X(z) + z^{-1}\left(\cdots + z^{-1}b_MX(z)\right)\right)\right)\right) \\
&\qquad + z^{-1}\left(a_1Y(z) + z^{-1}\left(a_2Y(z) + z^{-1}\left(a_3Y(z) + z^{-1}\left(\cdots + z^{-1}a_NY(z)\right)\right)\right)\right) \\
&= b_0X(z) \\
&\quad + z^{-1}(b_1X(z) + a_1Y(z) \\
&\quad + z^{-1}(b_2X(z) + a_2Y(z) \\
&\quad + z^{-1}(b_3X(z) + a_3Y(z) \\
&\quad + z^{-1}(\cdots \\
&\quad + z^{-1}(b_{M-1}X(z) + a_NY(z) + z^{-1}b_MX(z))))))
\end{aligned}
$$

This corresponds to the system implementation of Figure 2.8. This is the so-called *transposed form II*. The name stems from the fact that this form can be obtained by *transposing* the direct form II, i.e.

- reversing all signal flow arrows,
- replacing every summation by a node,
- replacing every node by a summation, and
- interchanging the role of $x[n]$ and $y[n]$.

2.8.2.5 Which form to use in practice?

From computational complexity point of view, all forms are similar: they require the same number of additions and multiplications; regarding delay elements the II-forms are obviously

[1] Again we made the arbitrary choice $M = N + 1$

more economical. If infinite precision arithmetic is used, all implementations are equally well. In reality they differ (w.r.t. rounding and coefficient quantization effects).
The transposed form is easier to parallellize in hardware. In software the type of processor determines our preference:

- in case of multicore or superscalar processors the transposed form is better;
- in a single-threaded processor or in serial hardware, the direct form is equally good;
- in an SIMD-processor, the direct form is better.

The transposed form II is superior w.r.t. stability if integer arithmetic is used. The direct form I is in general less prone to coefficient quantization effects and stability problems.

Exercises
Some exercises on the basic forms for recursive systems

Exercise 2.8.2.5-1: Consider the LTI-system described by the following transfer function:

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

1. Draw the direct form I implementation
2. Draw the direct form II implementation
3. Draw the transposed form I implementation
4. Draw the transposed form II implementation

Exercise 2.8.2.5-2: Consider the LTI-system described by the following transfer function:

$$H(z) = \frac{1 + z^{-1} + 0.3z^{-2}}{1 + z^{-1} - 0.2z^{-2}}$$

1. Draw the direct form I implementation
2. Draw the direct form II implementation
3. Draw the transposed form I implementation
4. Draw the transposed form II implementation

Exercise 2.8.2.5-3: (*) Consider the LTI-system described by the following transfer function:

$$H(z) = z^{-2}\frac{1 + z^{-1} + 0.3z^{-2}}{1 + z^{-1} - 0.2z^{-2}}$$

1. Draw the direct form I implementation
2. Draw the direct form II implementation
3. Draw the transposed form I implementation
4. Draw the transposed form II implementation

Exercise 2.8.2.5-4: Consider the LTI-system described by the following transfer function:

$$H(z) = \frac{z^2 + 0.3z - 3.14}{z^3 + z^2 - 0.2z + 9.25}$$

1. Draw the direct form I implementation
2. Draw the direct form II implementation
3. Draw the transposed form I implementation
4. Draw the transposed form II implementation

Exercise 2.8.2.5-5: (*) Consider the LTI-system described by the following transfer function:

$$H(z) = \frac{0.3z - 3.14}{-0.23z^3 + 5.43z^2 - 8.34}$$

1. Draw the direct form I implementation
2. Draw the direct form II implementation
3. Draw the transposed form I implementation
4. Draw the transposed form II implementation

Exercise 2.8.2.5-6: (*) Take any of the direct-form implementations you created above and try to reconstruct the original transfer function by analyzing the block diagram.

Exercise 2.8.2.5-7: (**) Take any of the transposed-form implementations you created above and try to reconstruct the original transfer function by analyzing the block diagram.

2.9 Non LTI systems

A system may not fall into the category of LTI systems, if it is either

- nonlinear,
- time-variant,

or both. Though these systems fall outside the scope of this limited course, there is one further distinction regarding time-variant systems which is good to know on this beginner's level:

variable systems — systems that change their behavior in time based on external control parameters (e.g., variable gain amplifiers)

adaptive systems — systems that change their behavior in time based on the part of the input signals that already has been processed (e.g., adaptive filters)

Chapter 3

System architectures

In this chapter you will get to know the basic system architecture choices that need to be made in signal processing systems:

- incremental vs. frame-based architectures, and
- within the frame-based architectures: single or double-buffer architectures or a mixture of both.

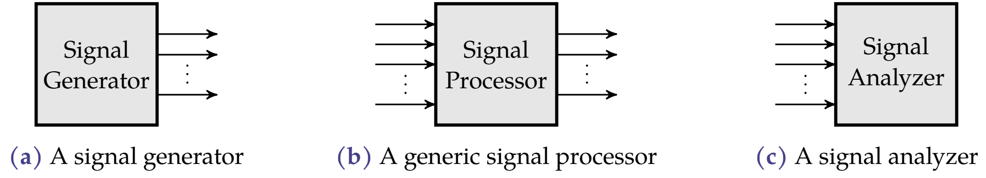
After having studied this chapter, you are expected to be able to

- select an appropriate system architecture given the requirements of the application,
- assess the implications of the choice that has been made.

Given the fact that modern computing architectures are often multicore architectures, it makes sense to study first Appendix C, which explains the basics of the speed-up one can gain from a multicore setup. Please, do so!

3.1 Introduction

Signal processing systems take zero or more input signals, perform operations on those signals and produce zero or more output signals (see Figure 3.1b).



(a) A signal generator (b) A generic signal processor (c) A signal analyzer

Figure 3.1: Signal processing systems

The special cases (for which the number of inputs or outputs are zero) are: signal generators (see Figure 3.1a) and signal analyzers (see Figure 3.1c). However, in general we deal with systems with at least one input and one output (see Figure 3.1b).

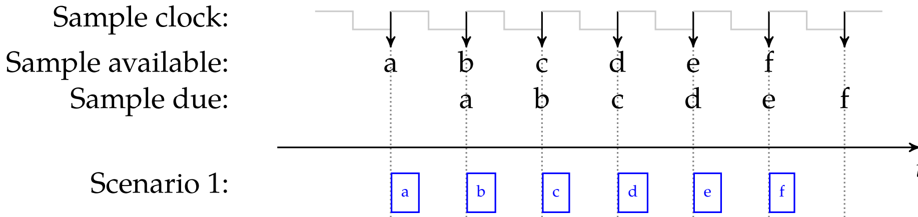
3.2 Incremental vs. frame-based architectures

In real-time operation two possible situations can occur:

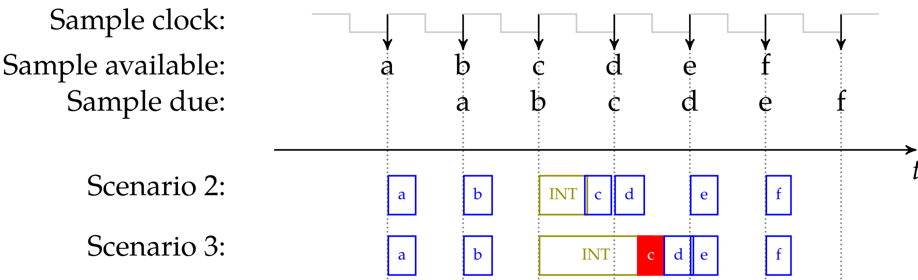
- the input signals are available one sample at a time and we will treat the signals sample by sample (incremental operation),
- the input signals are available in frames of length M and we will treat those samples together (frame-based operation).

The former seems a logical choice, as it avoids latency (we don't have to wait until a full frame of samples is available before we can start processing). However, this strength is also its direct weakness.

We'll explain this with an example. Consider the diagram below: a sample clock indicates whenever a new sample is available (on the rising edge). Let's assume we need to perform some work (some DSP-operations programmed in a low-level language, like C or assembler) on every incoming sample. Our goal is to finish the work before the next clock tick. Modern as we are, we execute the work to be done as tasks on a real-time operating system, running on a digital signal processor. This has been indicated as scenario 1. As soon as a sample arrives, the scheduler invokes the corresponding task (indicated with a blue box) and executes it. The horizontal dimension represents time.

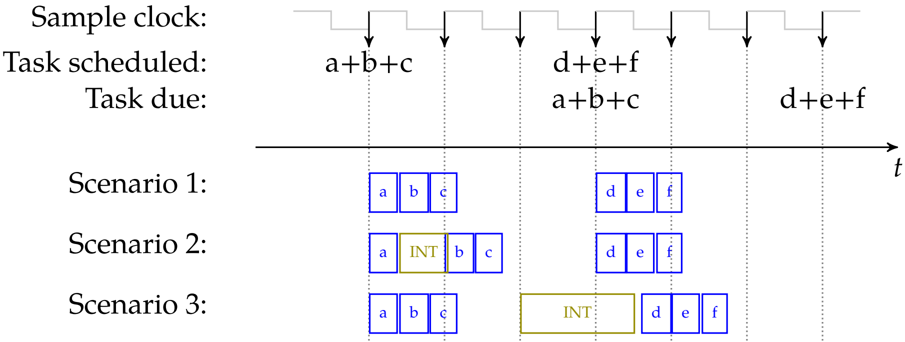


However, our real-time operating system has other processes running, with higher-priority tasks. Some of them are interrupt driven and therefore their scheduling time is highly unpredictable. This means that from time-to-time an interrupt may stop or prevent the processor from executing its normal task and forces it to do something else first (e.g. react to some timers). If the interrupt is short enough, we still have sufficient time to finish our work (scenario 2). However, if the interrupt routine takes more than a sample period, then we are in trouble (scenario 3): we miss the deadline for sample c.



The obvious solution is to conglomerate the work into bigger frames of samples, delaying the deadline to deliver the result. The price we pay is extra latency (in the order of twice the frame

length). The gain is robustness w.r.t. interrupts. This has been illustrated in the diagram below. Samples are now gathered in frames of length 3, freeing up bigger slots for interrupts, such that no deadlines are missed due to a longer interrupt service routine (scenario 3).



Of course, there is no guarantee that we never will miss a deadline. But the chance of it happening can be minimized by choosing an appropriate frame length.

Related to missing deadlines, the following terminology is worthwile to know:

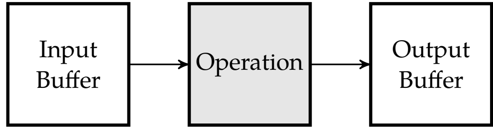
Buffer underrun failure to write the samples in the output buffer before the deadline, which means that the following operation will not have the required data to work on;

Buffer overrun failure to read the samples from the input buffer before the deadline, which means that these samples will be overwritten by new samples, before you can process them.

Now that we know why it makes sense to perform signal processing in frames, let's see how we can make that happen.

3.3 Organizing frames

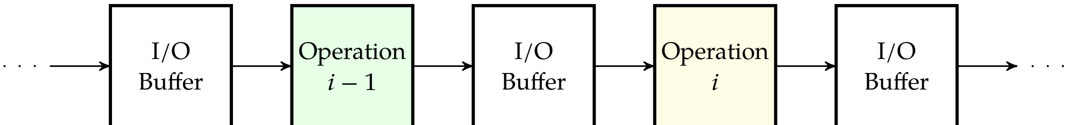
In a frame-based operation scenario, a signal operation will have to read its samples from an input buffer and write its samples to an output buffer.



We will start with single-buffer operation and then continue with double-buffer operation.

3.3.1 Single-buffer architectures

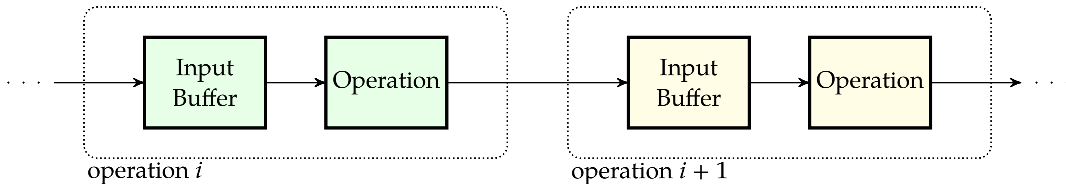
If we cascade multiple operations, we can clearly see that buffer sharing is possible. The input buffer of an operation is equal to the output buffer of the previous operation and the output buffer of an operation also can serve as the input buffer of the next operation.



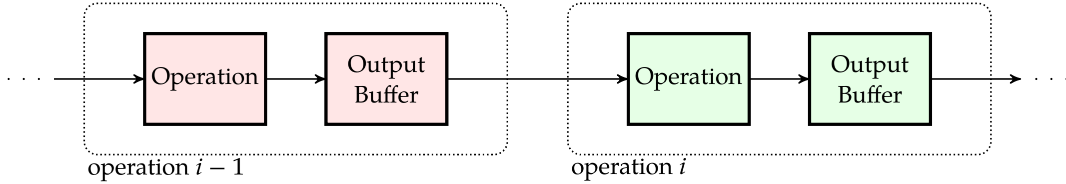
This leads us to a decision that needs to be made as to *who owns the buffer*. You may ask youself: 'Why? Couldn't we just make the buffers stand on their own, not belonging to any operation?' The benefit of assigning the buffers to an operation is that this operation may make smart use of the buffer (e.g. making it part of a bigger circular buffer to optimize/ease the processing, or to keep essential information in between frames).

We have two options:

- *input-buffer style* — the operation owns its input-buffer and makes use of the input-buffer of the subsequent operation to write its result into:



- *output-buffer style* — the operation owns its output buffer and makes use of the output buffer of the previous operation to read its inputs from:



As the input/output buffer that is owned by an operation needs to be accessed by the operation up-/down-stream, they need to be accessible by these operations. We therefore often refer to them as *accessible buffers*. Buffers that are not accessible ('public') are called *local or private buffers*. The setup in input-buffer and output-buffer style mode has been illustrated in Figure 3.2 and Figure 3.3.

Switching from one style to another, can be accomplished using special glue blocks that

- offer an intermediate buffer, when switching from input-buffer to output-buffer style,
- copy data from an existing output buffer to an existing input buffer, when switching from output-buffer to input-buffer style.

Note that these glue blocks cause overhead and are to be avoided as much as possible. As such, let's not consider the glue blocks any further.

Remarks

- Note that for both styles, the sequence is only valid if all operations are executed in order (from earliest in the chain to latest in the chain). This hinders parallel execution of filter tasks (as has become very common, given the availability of multicore processors). We can solve this by using *double-buffer operation*.

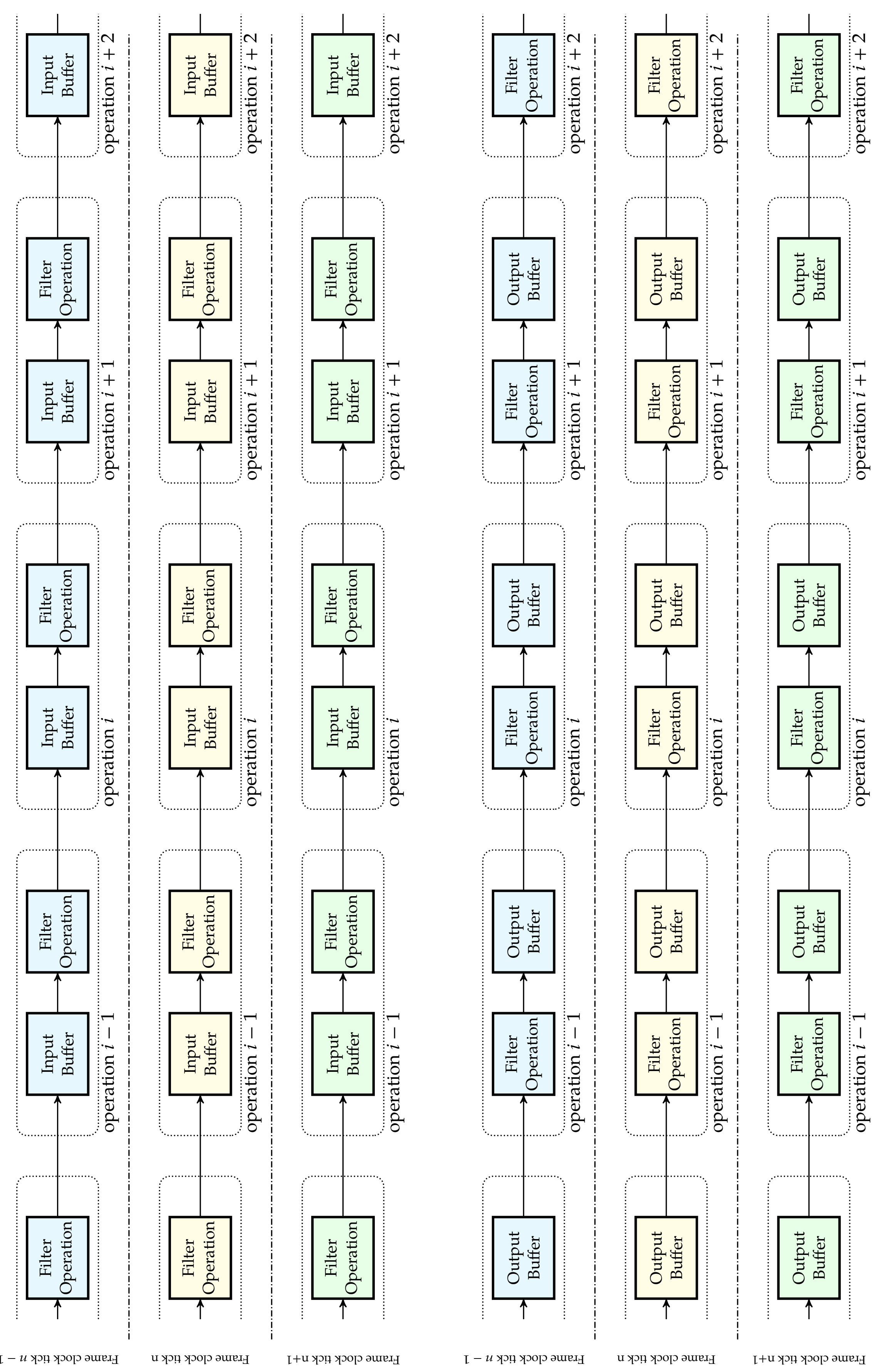
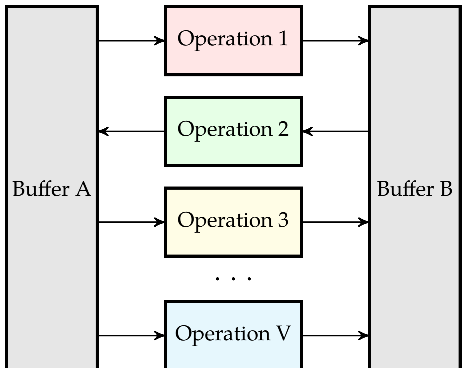


Figure 3.2: Input-buffer style single buffer operation

Figure 3.3: Output-buffer style single buffer operation

- On platforms with little memory, further memory optimization is possible by reusing the public buffers in a ping-pong-style operation as indicated below:



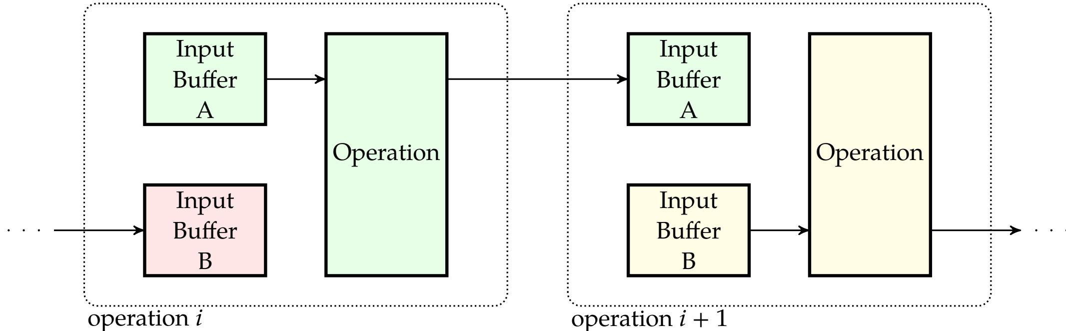
In this case, private buffer bookkeeping might become cumbersome. In addition, abutment of public and private memories might no longer be an option.

3.3.2 Double-buffer architectures

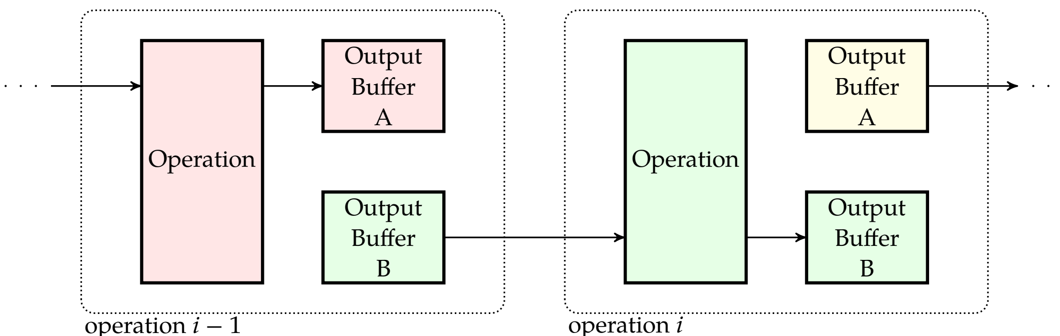
In this case, every operation has two buffers. This allows every operation to be run independently, providing maximal flexibility for scheduling in a multicore or multithreading environment and allows to make the fraction of parallelizable work p in Amdahl's law to approximate 1.

Again, we can distinguish two variants:

- *input-buffer style*



- *output-buffer style*



Of course, this way of working introduces pipelining in the system. The data progresses one stage whenever all filter operations have been run once. As such, the signals progress

gradually through the architecture, improving efficiency by exploiting parallelism, at the cost of an increased overall latency. This has been illustrated in Figure 3.4 for an input-buffer style setup and in Figure 3.5 for an output-buffer style setup.

3.3.3 Mixed-buffer architectures

In order to minimize the overall latency while enabling parallel execution, a mixed-buffer architecture is often used. This means that all operations are partitioned in multiple separate high-level blocks that can be executed in parallel. The blocks behave as double-buffer blocks, but internally use a single-buffer implementation. In the example of Figure 3.6, the choice between input- and output-style buffers has been left ambiguous (buffers connected with a dotted line are in fact the same buffer). In order to make sure that the appropriate frames are treated by an operation, it might be necessary to introduce extra *waiting stages* (queues), as is done on the top right.

3.3.4 Comparison

Now let's discuss and compare the advantages/disadvantages of the single- and double-buffer setup. Extrapolating this discussion to the mixed-buffer case is left to the user.

Let's assume a setup

- with a single input and output,
- running at f_s samples per second (remember: $f_s = 1/T_s$),
- with frames of size M, that
- requires V operations for the full flow, each requiring a single-core processing time T_i (for mathematical calculations), R_i memory cells (with $i = 1, 2, \ldots, V$), and
- hardware that offers Q equal parallel cores.[1]

This allows calculating the implications introduced by this setup. When you understand the principles of this simple case, you should be able to generalize it to more involved cases.

Single-buffer setup

Required computing power In the setup described above, the *computing power loading fraction* equals

$$CPLF = \frac{f_s}{M} \cdot \frac{\sum_{i=1}^{V} T_i}{Q}$$

The value of $CPLF$ indicates what fraction of the available computing power has been used.

Note the dimensions: $[CPLF] = 1$.

However, this assumes perfect parallelism. According to Amdahl's law (see Appendix C), this is too optimistic. The portion of the work that can be parallelized will be limited and depends

[1] This is called symmetrical multiprocessing (SMP).

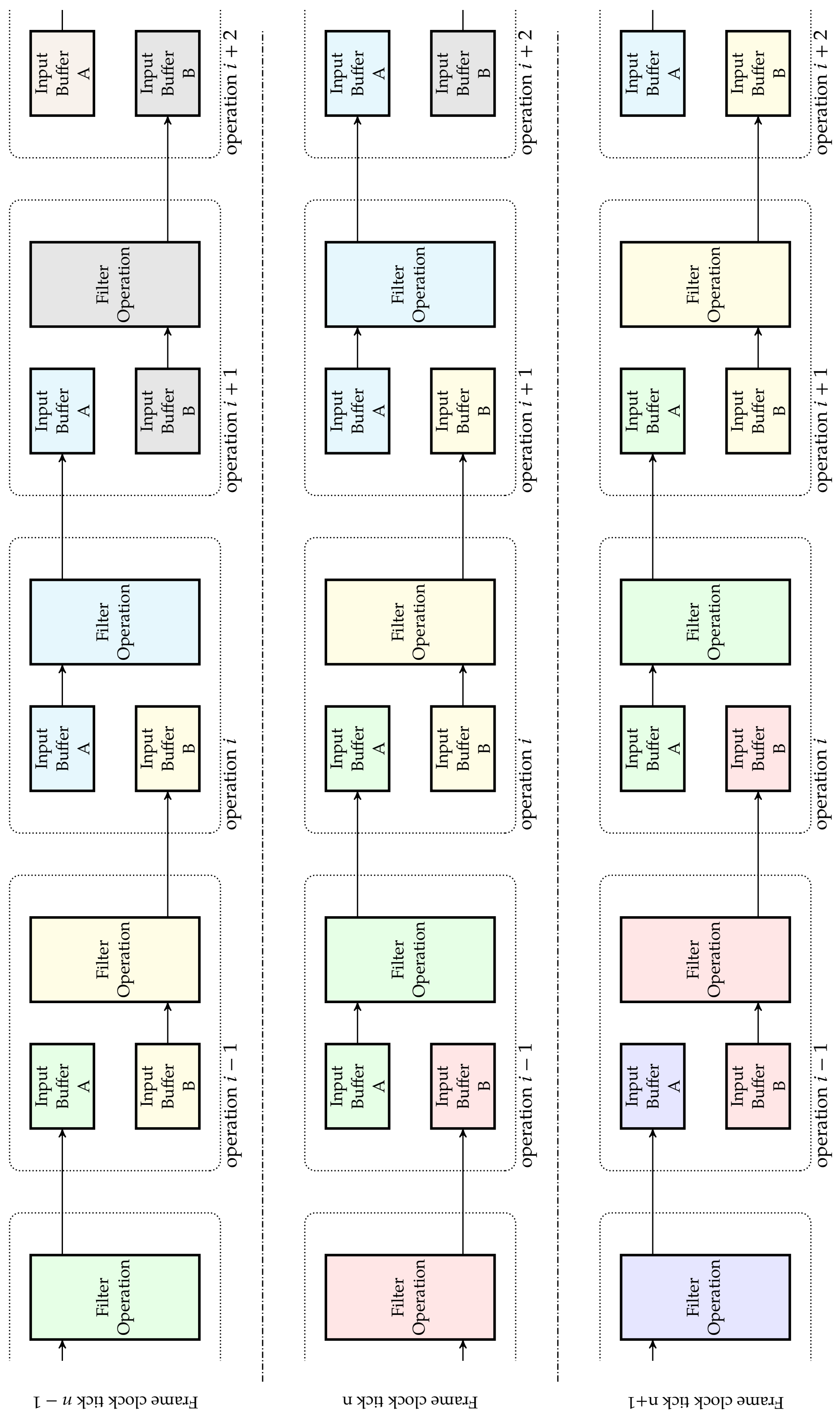


Figure 3.4: Input-buffer style double-buffer operation

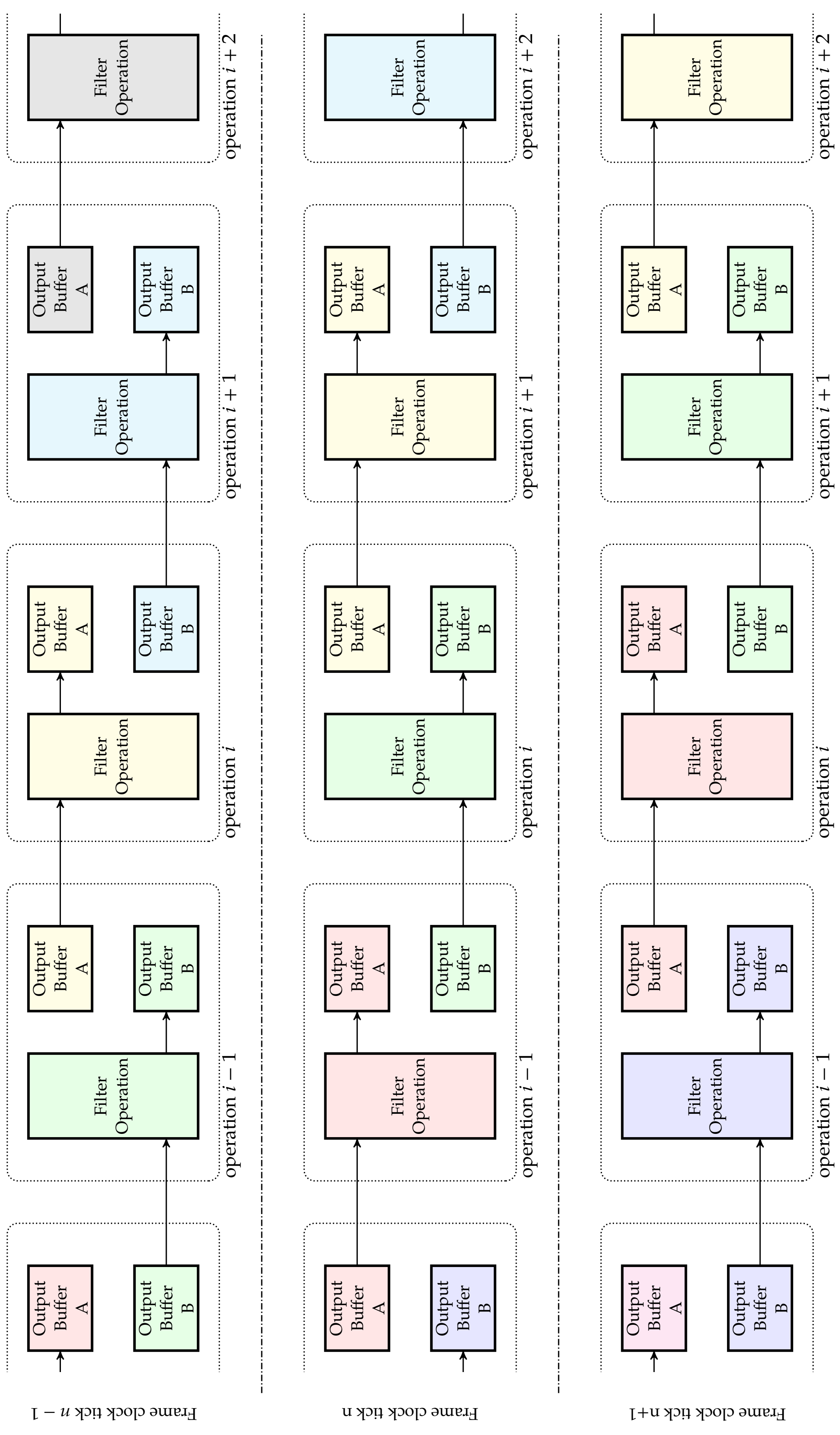


Figure 3.5: Output-buffer style double-buffer operation

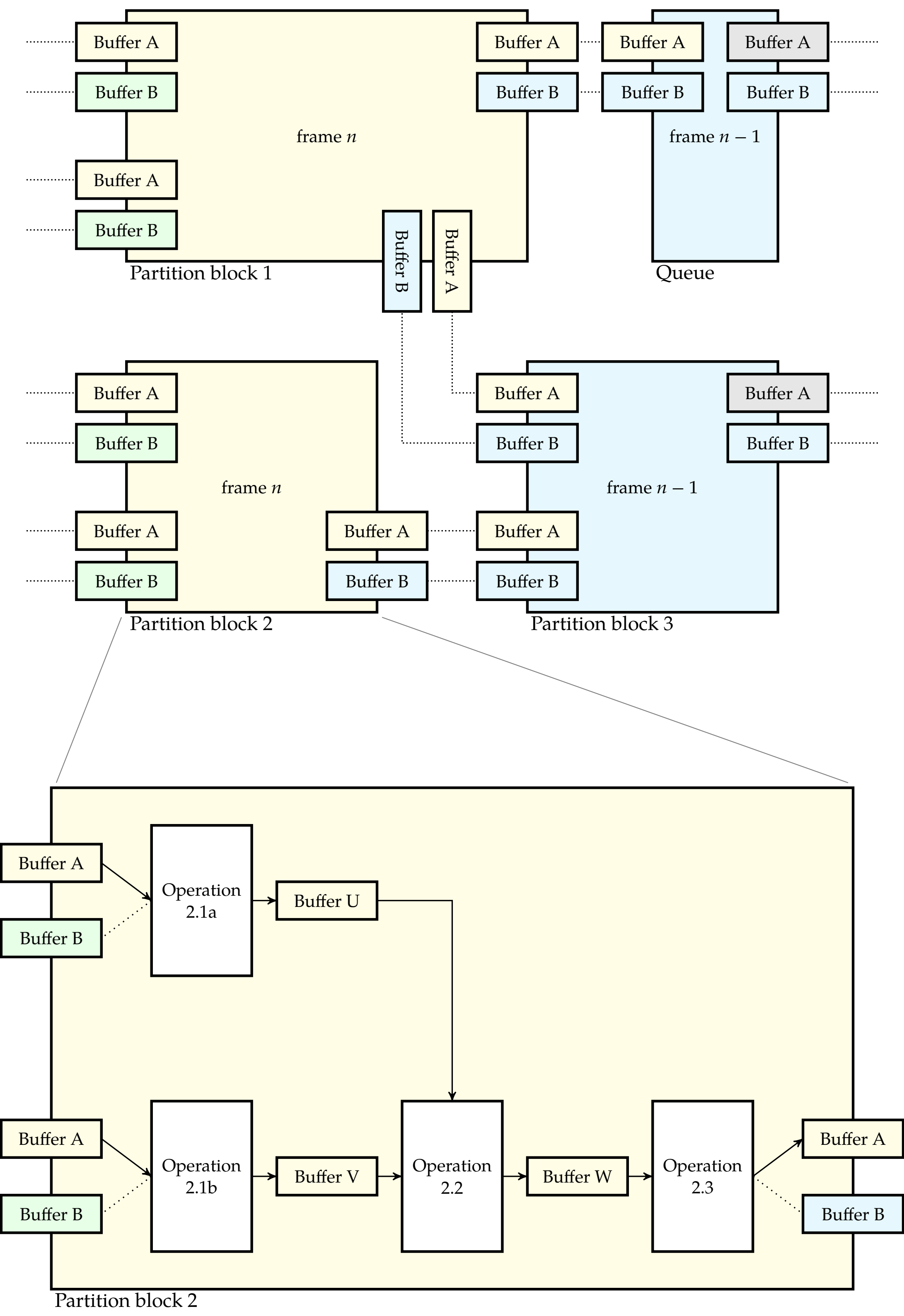


Figure 3.6: Example of a mixed-buffer architecture: (top) overall architecture using double-buffer partitions, (bottom) detail of partition block 2 with single-buffer internal structures.

on:

- the degree to which the calculations within the operations can be parallelized,
- the amount of parallel, independent signal paths in the system.

Applying Amdahl's law, assuming a fraction p of the work can be parallelized, leads to:

$$CPLF = \frac{f_s}{M} \cdot \left(1 - p + \frac{p}{Q}\right) \sum_{i=1}^{V} T_i$$

Note that if $p = 1$, this equation reduces itself to the previous one.

To be feasible, we must ensure $CPLF < 1$. In addition, very often $CPLF \approx 1$ is not feasible, as the operating system and related services also requires a part of the budget.

Required memory The total amount of memory required is:

$$R = 3M + \sum_{i=1}^{V} \underbrace{(M + R_{i,extra})}_{\equiv R_i} \tag{3.1}$$

with R_i representing the memory an operation requires. This consists of its public buffers plus the exta (local/private) buffers, represented by $R_{i,extra}$.

The first term in (3.1) accounts for the extra input and output buffer(s) that the operations do not yet provide. Why do we need that additional memory of three frames?

- In an input-buffer style setup, the first stage needs to be a double-buffer block to allow for new data coming in while processing the current frame, and we need a double-buffer output.
- In an output-buffer style setup, we need a double input buffer, to allow for new data coming in while processing the current frame and we need a double-buffer output block.

Often the first term is neglectable compared to the total amount of memory required by the individual operations.

Introduced latency The introduced latency L amounts to

$$L = 2 \cdot M \cdot T_s = 2\frac{M}{f_s}$$

as we need to wait M clockticks before a full frame is loaded and M more clockticks before the result is ready.

Double-buffer setup

Required computing power In the setup described above, the *computing power loading fraction* equals

$$CPLF = \frac{f_s}{M} \cdot \frac{\sum_{i=1}^{V} T_i}{Q}$$

However, this assumes perfect parallelism. Again Amdahl's law applies, leading to:

$$CPLF = \frac{f_s}{M} \cdot \left(1 - p + \frac{p}{Q}\right) \sum_{i=1}^{V} T_i$$

However, because of the perfect decoupling of the double-buffer setup, we may expect p to be very close to 1.

To be feasible, again, we must ensure $CPLF < 1$, and again most often the maximal allowable $CPLF$ will be considerably lower.

Required memory The total amount of memory required is:

$$R = 2M + \sum_{i=1}^{V} \underbrace{(2M + R_{i,extra})}_{\equiv R_i}$$

with $R_{i,extra}$ the extra amount of memory an operation requires in surplus to its publicly accessible buffers. The first term accounts for the extra input or output buffers that the operations do not yet provide. Often, this term is neglectable compared to the total amount of memory required by the individual operations.

Introduced latency The price paid for the perfect decoupling is in terms of extra latency. The total latency introduced, amounts to:

$$L = (1 + V) \cdot M \cdot T_s = (1 + V)\frac{M}{f_s}$$

Mixed-buffer setup

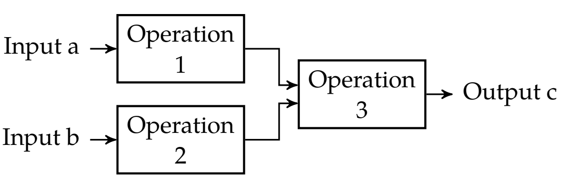
As we already have taken double-buffers at the input and output into account in the equations of the single-buffer setup, the equations for a mixed-buffer block are the same as for the single-buffer setup.

Remarks Note that the equations in the comparison above, assume a single-input, single-output system with a linear signal flow. This does not need to be the case. Making generic equations for a generic multi-flow MIMO setup could be done, but these equations will rather be complex than useful. In case of a MIMO setup, use your commons sense and make a drawing of the situation. This will ease your insight into the amount of buffers required and the latency introduced. The exercise below is an example of such a system.

Exercises

Exercise 3.3.4-1: Assume the signal processing flow indicated below, consisting of three tasks, each with indicated required (single-core) computing time T_i and extra (local/private) memory requirement $R_{i,extra}$. The desired sample frequency amounts to 750 kHz. The frame length amounts to 128. We have 4 cores available on our DSP.

Compute both for a single-buffer and double-buffer setup:



Op	T_i	$R_{i,extra}$
1	120 µs	16
2	50 µs	0
3	75 µs	32

- the computing power loading fraction *CPLF*
- the maximal achievable sampling frequency f_s
- the amount of memory R required
- the latency L at the maximal achievable sample frequency

Chapter 4 Convolution and FFT-convolution

In this chapter, you will learn about:

- the discrete-time convolution operation,
- its properties,
- how this generic operation can be made tractable for a computer,
- how to implement this operation,
- how to use the FFT to perform convolution.

After having read this chapter, some questions will still be left unanswered:

- How do I program convolution on real DSP hardware?
- How are circular buffers implemented in DSP hardware? How do I use them?

After having read/studied this chapter, you are expected to be able to

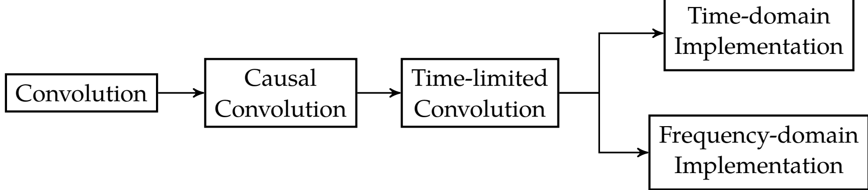
- explain the convolution operation and its properties,
- apply convolution and its properties when given an LTI system and an input signal,
- implement index-based convolution algorithms (incremental and frame-based) including frame sequencing,
- implement FFT-convolution including frame sequencing,
- explain and assess pros and cons of the overlap–add vs. overlap–save frame-sequencing techniques,
- decide when to use direct convolution and when to use FFT-convolution.

4.1 Introduction

It is time to study convolution in a little more depth. Though convolution is a concept that is applicable to both continuous-time and discrete-time systems, we will limit ourselves in this chapter to the discrete-time domain. However, the properties listed in section 4.3 on the next page are more generic than that restricted scope.

We will spend quite some time in this chapter on trying to implement convolution on a digital computer or a DSP. For this we need to restrict the generic convolution concept to causal convolution, and even further to time-limited convolution. As soon as we understand this, we

can investigate how to implement convolution in the time domain and in the frequency domain. The mental map below may help you finding your way through this chapter.



4.2 Convolution

Generic discrete-time convolution Consider two signals $x[n]$ and $y[n]$. The convolution of these two signals results in a signal $z[n]$ defined as

$$z[n] = x[n] \star y[n] = \sum_{i=-\infty}^{+\infty} x[n-i]y[i] \tag{4.1}$$

In the next section, we will take a look at the properties of this operation.

4.3 Convolution properties

The convolution operation is:

1. commutative
2. associative
3. distributive w.r.t. the addition operation

4.3.1 Commutativity

The convolution operation is commutative, i.e.

$$x[n] \star y[n] = y[n] \star x[n]$$

This means that the order in which we convolve two signals is of no importance. Drawing a block diagram that corresponds to this property, sheds a different light on this property.

$$x[n] \longrightarrow \boxed{y[n]} \longrightarrow z[n] \quad \Leftrightarrow \quad y[n] \longrightarrow \boxed{x[n]} \longrightarrow z[n]$$

This means that if we interchange the input signal with the LTI system's impulse response, the output remains the same! The practical use of this property is limited.

4.3.2 Associativity

The convolution operation is associative, i.e.

$$x[n]\star y[n]\star z[n] = (x[n]\star y[n])\star z[n] = x[n]\star(y[n]\star z[n])$$

This means that the order in wich we calculate the convolutions is not important. Drawing a block diagram of this property reveals the essence of this property:

$x[n] \rightarrow \boxed{y[n]} \rightarrow \boxed{z[n]} \rightarrow q[n] \quad \Leftrightarrow \quad x[n]\star y[n] \rightarrow \boxed{z[n]} \rightarrow q[n] \quad \Leftrightarrow \quad x[n] \rightarrow \boxed{y[n]\star z[n]} \rightarrow q[n]$

As such, this property allows merging cascaded convolution operations.

4.3.3 Distributivity

The convolution operation is distributive w.r.t. the addition operation, i.e.

$$x[n]\star(y[n] + z[n]) = x[n]\star y[n] + x[n]\star z[n]$$

Drawing a block diagram of this property reveals the essence of this property:

$x[n] \rightarrow \boxed{y[n]+z[n]} \rightarrow q[n] \quad \Leftrightarrow \quad x[n]$ → parallel branches $\boxed{y[n]}$ and $\boxed{z[n]}$ → $\oplus \rightarrow q[n]$

This property allows for the elimination of parallel convolution paths.

Exercises

Exercise 4.3.3-1: Prove the commutatitvity property (a) using the convolution's definition, (b) using the Z-transform.

Exercise 4.3.3-2: Prove the associativity property (a) using the convolution's definition, (b) using the Z-transform.

Exercise 4.3.3-3: Prove the distributive property (a) using the convolution's definition, (b) using the Z-transform.

4.4 Causal Convolution

Implementing the generic convolution operation of (4.1) using a digital computer is kind of a problem: we cannot deal with infinite summation boundaries.

Luckily, in DSP we're using convolution to calculate the response of causal LTI-systems, i.e. one of the two signals is the (causal) impulse response[1], hence *one of the signals is causal*.

Consider two signals $x[n]$ and $h[n]$, with $h[n]$ causal, starting at $n = 0$. The convolution $x[n] \star h[n]$ is a signal $y[n]$ defined as:

$$y[n] = x[n] \star h[n] = \sum_{i=0}^{+\infty} x[n-i]h[i]$$

or

$$= \sum_{i=-\infty}^{n} h[n-i]x[i]$$

Alas, apparently, despite our efforts, we're still stuck with an infinite summation boundary. Let's take it one step further, let's assume *both signals are causal*. Consider two $x[n]$ and $h[n]$, both causal, starting at $n = 0$.
The convolution $x[n] \star h[n]$ is a signal $y[n]$ defined as:

$$y[n] = x[n] \star h[n] = \sum_{i=0}^{n} x[n-i]h[i]$$

or

$$= \sum_{i=0}^{n} h[n-i]x[i]$$

Implementing this fully 'causal' convolution seems feasible. Still, the computational effort increases linearly with n. No problem if n is not too big, you might think. Alas, time always goes by, so n *will* go to infinity, just because of nature. So: close but no cigar...

To make the convolution operation tractable for real-time operation (what DSP is all about), we still need to simplify the situation a little more.

4.5 Time-limited convolution

4.5.1 Convolution with a time-limited signal

Let's take one step back and remove the restriction that both signals should be causal. Instead let's assume the causal impulse response is time-limited (i.e. we're dealing with a finite impulse response system).

Discrete-time convolution with a time-limited-signal Consider a time-unlimited signal $x[n]$ in combination with a time-limited signal $h[n]$ with length N, starting at $n = 0$ (i.e.

[1]Just like we consider a system to be causal when its impulse response is zero for negative times, by extension, we label signals 'causal' if they are zero for negative times.

causal). The convolution $x[n] \star h[n]$ is a signal $y[n]$ defined as:

$$y[n] = x[n] \star h[n] = \sum_{i=0}^{N-1} x[n-i]h[i]$$

or

$$= \sum_{i=n-N+1}^{n} h[n-i]x[i] \tag{4.2}$$

At last: a fixed number of operations indepent of the time index! That's what we need for real-time DSP operation.

4.5.2 Convolution of two time-limited signals

Now, let's consider both signals to be time-limited.

Discrete-time convolution of two time-limited signals Consider two time-limited signals $x[n]$ and $h[n]$ with length M and N respectively, both starting at $n = 0$. The convolution $x[n] \star h[n]$ is a signal $y[n]$ of length $N + M - 1$, defined as:

$$y[n] = x[n] \star h[n] = \sum_{i=\max(0,n-M+1)}^{\min(N-1,n)} x[n-i]h[i]$$

or

$$= \sum_{i=\max(n-N+1,0)}^{\min(n,M-1)} h[n-i]x[i]$$

Don't mind the complex summation boundaries. The important thing to remember is the length of the resulting signal: $N + M - 1$.

4.6 Basic implementation schemes

Now that we've found a computer tractable convolution concept, *convolution with a time-limited signal*, let's take a look at how we could implement this operation. One might argue about the usefulness of going into detail. Plenty of libraries exist that contain a most efficient convolution implementation. True. However, as this is our first acquaintance with DSP algorithms, it is worth taking our time for this. We'll first take a look at two basic algorithmic schemes, that we will implement later on in some practical algorithms.

The code examples shown here have been written in C++ [Str97] without making explicit[2] use of C++'s object oriented features or standard library data types (like `std::vector`, `std::set`, `std::list`, a.o.).

The reasons why we've chosen for this solution (C in C++ disguise), are multiple:

- C++ allows declaring variables at the appropriate place;
- C++ allows the pass-by reference construct (`&`) without using pointers (this avoids "distraction by pointers" on places where pointers are not essential);
- C++ allows performing operator overloading (allowing to implement a true modulo operator);
- We don't make use of C++'s object oriented features or standard data types, as this would be a distraction from the barebone implementation: therefore, C in C++ disguise.

Morphing this C++ code into C or embedding it in true C++ class hierarchies should be straightforward for anyone familiar with C and/or C++.

That said, both C and C++ are languages that allow producing very readable code, while still being tightly coupled with the underlying hardware. It is therefore not that surprising that very often DSP-hardware is programmed in C or C++.

4.6.1 Input-side algorithms

Let's focus on what one input sample does to the output. This means decomposing the input signal using impulse decomposition. This has been illustrated in Figure 4.1 on page 54. As one can see every sample is multiplied with the impulse response and as such contributes a scaled impulse response to the output. Considering the implementation from this perspective is called *input-side*.

[2] We will use the object oriented features in defining a circular integer and pointer class. As a user of these classes you will not notice the object oriented nature. That's what we mean by non-explicit use.

4.6.2 Output-side algorithms

Let's take a look at (4.2) from the output side, i.e. let's focus on one sample of the output and let's see how that's influenced by the input $x[n]$:

$$y[n] = x[n] \star h[n] = \sum_{i=n-N+1}^{n} h[n-i]x[i]$$

It means that the sample $y[n]$ is caused by the samples $x[i]$ starting at $i = n - N + 1$ until $i = n$. That portion of the x-signal is multiplied one by one by an inverted $(-i)$ and shifted $(+n)$ impulse response $h[n-i]$.

For the case $n = 5$, this has been illustrated in Figure 4.2 on page 55.

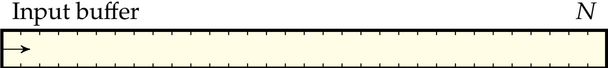
Considering the implementation from this perspective is called *output-side*.

4.6.3 Buffer types

Before we can dive into the implementation of the filter operations, we need to devote some time to the data types that are used for the buffers.

Linear buffers The most standard buffer that is used to store frame-data is the ordinary *linear buffer* that you know from regular programming languages as arrays or vectors. These consist of a contiguous range in memory that can hold a desired data type (N-bit integers or fixed point values, floats, doubles, ...). These buffers can be addressed in a linear way using indices or pointers. In DSP, we *always* use 0-indexed indices.

In the computation diagrams that will appear when discussing the implementations, we will draw a linear buffer as follows:



The name of the buffer is indicated on the left. The arrow indicates the increasing index or pointer direction. The number near the end of the buffer specifies its length (N in this case).

Circular buffers A circular buffer is a circular memory structure that can be addressed using indices or pointers. The special thing about these indices or pointers is that they "wrap around". This means that increasing an index/pointer that is pointing to a particular memory cell will yield an index/pointer that is pointing to the next cell (clockwise) and decreasing it will point to the previous cell (counterclockwise), no matter what the starting position was (see Figure 4.3a).

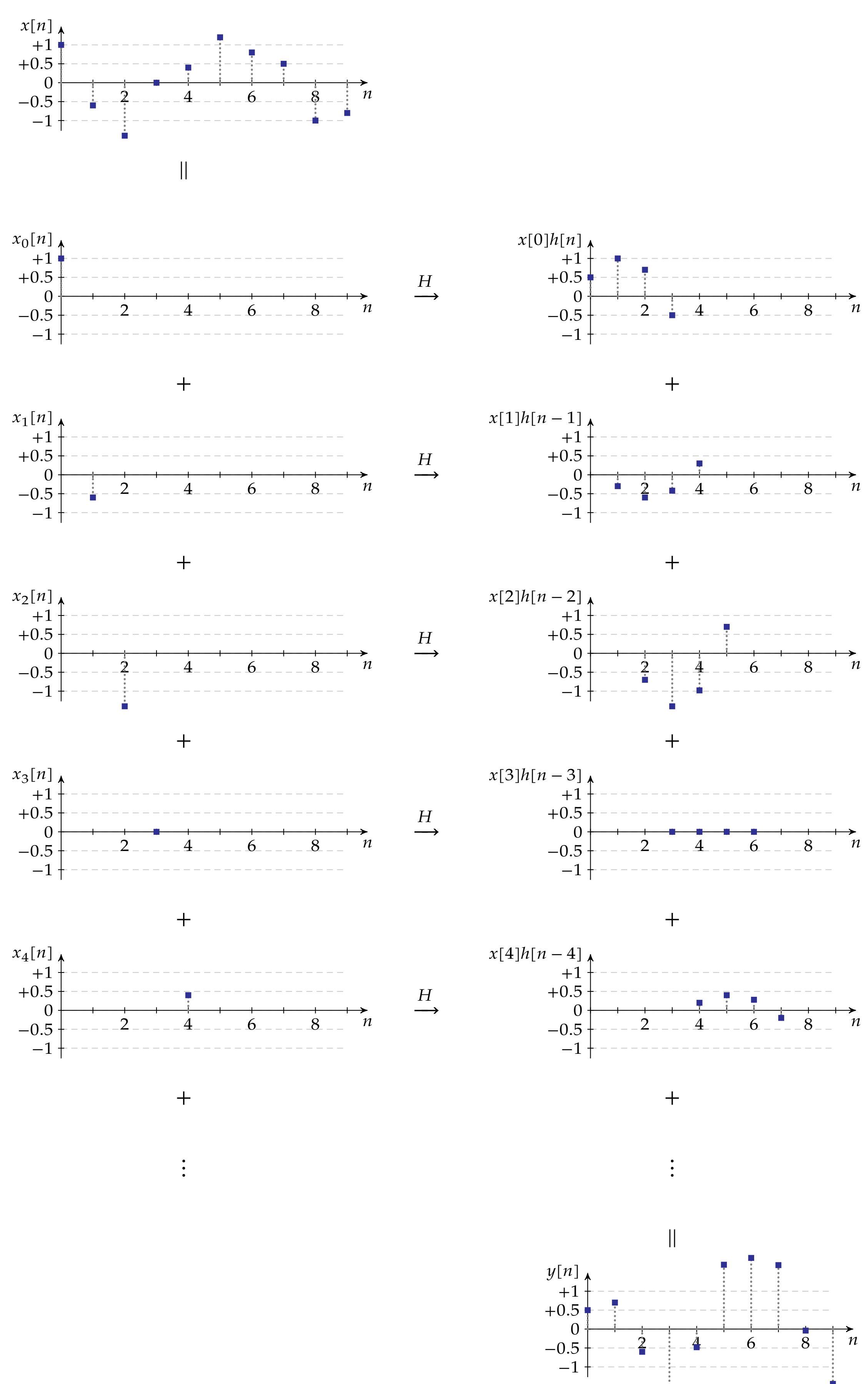


Figure 4.1: Illustration of the input-side convolution algorithm

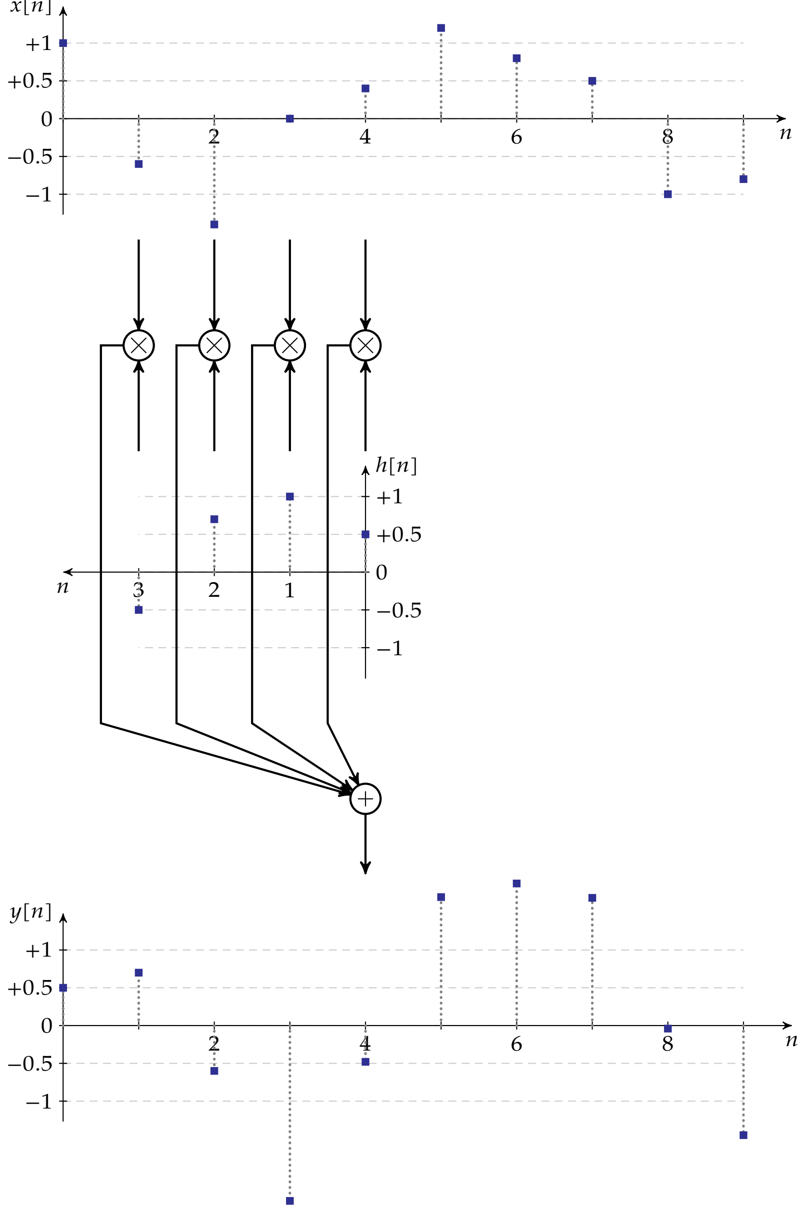
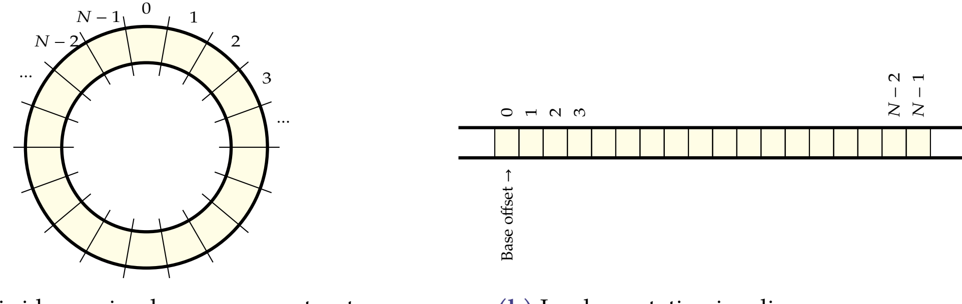


Figure 4.2: Illustration of the output-side convolution algorithm



(a) Basic idea: a circulary memory structure

(b) Implementation in a linear memory

Figure 4.3: Circular buffers: principle and implementation

Obviously, such a structure is constructed by taking a regular linearly addressed memory that has been equipped with circular indices or circular pointers (see Figure 4.3b). This is easily achieved for indices by using modulo-N arithmetic. In case the memory's size is a power of two ($N = 2^L$), the modulo-N arithmetic can be easily achieved by using a L-bit counter as index. Accessing the appropriate memory cell can then be achieved by offsetting the counter with the appropriate base memory address. In case the buffer is well alligned in memory, the offsetting can be accomplished by joining the counter with a base address that has L LSB zeroes.

In nowadays DSP hard- and software, circular buffers are implemented most efficiently, so making use of them makes perfect sense.

In many libraries, circular indices (and buffers) are invoked (i.e., recognized by the compiler or optimizer) by using the 'modulo'-operator `%` after every index manipulation. We therefore will use the same convention below. However, if you're trying these examples using your compiler at home, you will run into the fact that the standard 'modulo' operator `%` in C and C++ is no true modulo operator, but a remainder operator.[3] To overcome this, we'll use a small *circular integer* class: `cint`. You can find the source code in appendix A.

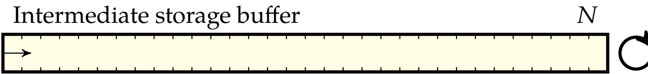
Though understanding the `cint`-class requires some familiarity with C++, using it is most simple. The following example, constructs a 5-element buffer `a`, a circular integer `i`, manipulates it and uses it to index the (circular) buffer at position with index 3:

```
#include <iostream>
#include "Circular/cint.hh"

int main()
{
  const int n = 5;
  double a[n];

  cint i = 0;
  ++i %= n;

  a[ ( i - 3 ) % n ] = 3.14;

  std::cout << a[3] << std::endl;

  return 0;
}
```

[3]To be correct: the remainder of negative integers is even not defined according to the ISO/ANSI standards, but often allowed and handled hardware/compiler dependent. In most cases the `%`-operator will not behave as a modulo operator. E.g., we would expect `-7 % 4 = 1`, but intel/GCC yields `-7 % 4 = -3`.

In the algorithms presented in the next subsections, we omitted the include directives in order not to overload the listings.

In the computational diagrams that illustrate the implementation schemes, we indicate a circular buffer as a normal buffer with an arced arrow to the right of it:



4.7 Implementation schemes in the time domain

In the time-domain we have four options, that are the product set of incremental vs. frame based an input side vs. output side:

- Incremental
  - input-side algorithm
  - output-side algorithm
- Frame-based
  - input-side algorithm (called the *overlap–add* method)
  - output-side algorithm (called the *overlap–save* method)

4.7.1 Incremental

Input-side algorithm As samples become available one at a time, an input-side algorithm is implemented most easily.

We can see from Figure 4.1 on page 54 that with every incoming sample we can calculate its effect on the current output sample and the $N-1$ next ones. These next (partial) output samples need be stored. To this end, we use a circular buffer memory.

Index-based implementation Take a look at the index-based implementation below. Note the multiply-and-accumulate (MAC) operation that is very typical for DSP algorithms. Very often this MAC operation is highly optimized in hardware such that it only takes one clockcycle to execute it. The algorithm is executed at every clock tick of the sample clock.

Listing 4.1: Incremental input-side convolution algorithm using indexed arrays

```
double iconvinc_i( double   x,          // next input sample
                   double*  h,          // array containing impulse response
                   double*  z,          // circular buffer array
                   cint&    i,          // index into circular buffer array
                   int      n )         // impulse reponse length
{
  for ( int j = 0; j < n; ++j )
    z[( i + j ) % n] += x * h[j];       // calculate x[i].h[n] and accmulate
                                        // (MAC)

  double y = z[i];
```

```
  z[i] = 0.0;

  ++i %= n;                             // modulo n index increment

  return y;
}
```

Pointer-based implementation As an alternative, consider the pointer-based implementation below.

Listing 4.2: Incremental input-side convolution algorithm using pointers

```
double iconvinc_p( double           x,  // next input sample
                   double*          h,  // impulse response array
                   cptr< double >&  zi, // circular buffer array pointer
                   int              n ) // impulse reponse length
{
  double*const endh = h + n;
  for ( ; h < endh; ++h, ++zi )
    *zi += x * *h;                      // calculate x[i].h[n] (MAC)

  double y = *zi;

  *zi++ = 0;                            // circular pointer increment

  return y;
}
```

The operation can be summarized in the following diagram:

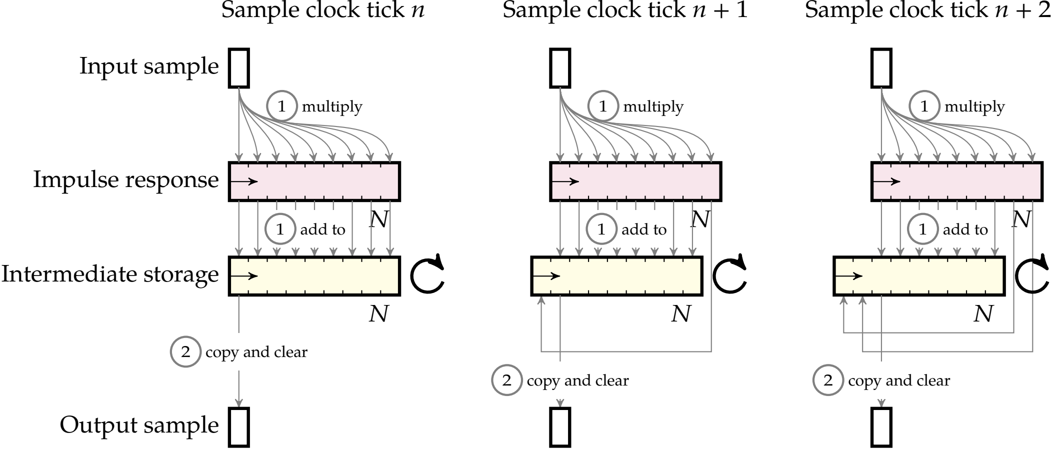


Figure 4.4: Computation diagram of an incremental input-side implementation in the time domain

Exercises
Some exercises on incremental input-side convolution algorithms

Exercise 4.7.1-1: (*) Rewrite the index-based algorithm in C++ using plain pointers.

Exercise 4.7.1-2: (**) Implement a circular buffer class in C++ and equip it with circular iterators. Write an iterator-based algorithm, using your circular buffer class.

Exercise 4.7.1-3: Write a test program to test the algorithms using some arbitrary waveforms. Check whether the outputs of all implementations are identical.

Output-side algorithm As we need N input samples to calculate one output sample, we need some intermediate storage for these samples. Again, we use a circular buffer.

Index-based implementation Take a look at the index-based implementation below. Note the multiply-and-accumulate (MAC) operation in the heart of the algorithm. This algorithm is executed at every clock tick of the sample clock.

Listing 4.3: Incremental output-side convolution algorithm using indexed arrays

```
double oconvinc_i( double   x,          // next input sample
                   double*  h,          // impulse response array
                   double*  z,          // circular buffer array
                   cint&    i,          // circular buffer array index
                   int      n )         // impulse reponse length
{
  z[i] = x;                             // store current input sample

  double y = 0.0;
  for ( int j = 0; j < n; ++j )
    y += z[( i - j ) % n] * h[j];       // calculate output for n = i (MAC)

  ++i %= n;                             // modulo n index increment

  return y;
}
```

Pointer-based implementation As an alternative, consider the pointer-based implementation below.

Listing 4.4: Incremental output-side convolution algorithm using pointers

```
double oconvinc_p( double           x,  // next input sample
                   double*          h,  // impulse response array
                   double*          z,  // circular buffer array
                   cptr< double >&  zi, // circular pointer into
                                        //   circular buffer array
                   int              n ) // impulse reponse length

{
  *zi = x;                              // store current input sample

  double y = 0.0;
  double*const endh = h + n;
  for ( ; h < endh; ++h, --zi )
    y += *zi * *h;                      // calculate output for n = i (MAC)

  ++zi;                                 // modulo n index increment

  return y;
}
```

The operation can be summarized in the following diagram:

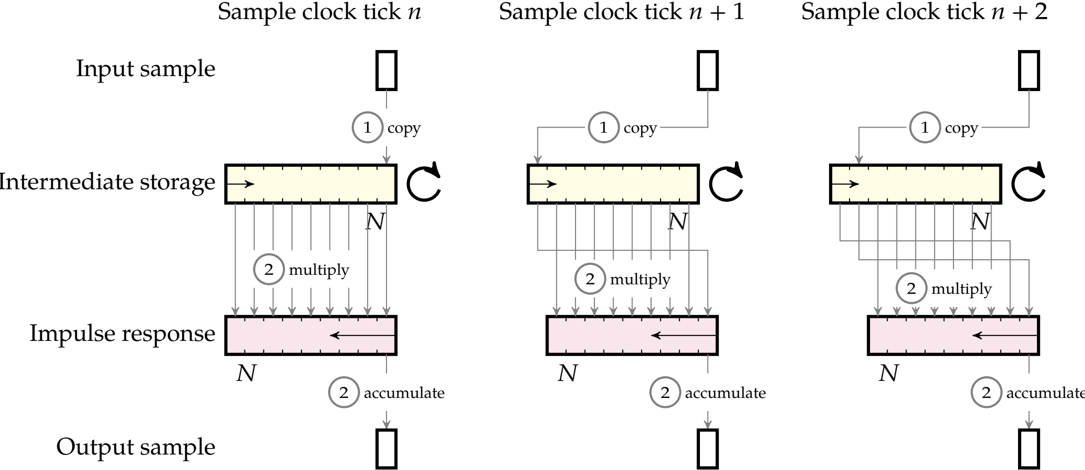


Figure 4.5: Computation diagram of an incremental output-side implementation in the time domain

---

Exercises
Some exercises on incremental output-side convolution algorithms

Exercise 4.7.1-4: (*) Rewrite the algorithm in C++ using plain pointers.

Exercise 4.7.1-5: (**) Implement a circular buffer class in C++ and equip it with circular iterators. Write an iterator-based algorithm, using your circular buffer class.

Exercise 4.7.1-6: Write a test program to test the algorithms using some arbitrary waveforms. Check whether the outputs of all implementations are identical.

4.7.2 Frame-based single buffer operation

We split the sample data in frames of length M and treat a single frame at every clock tick of the frame clock, that runs at a rate equal to the sample clock divided by M.

We will not provide code examples. Instead, we will illustrate the algorithms using computation diagram such that we can focus on the major operations and the frame sequencing instead of on the tiny little code details.

Before diving into to the details, be aware that we only treat the case in which the convolution kernel length N is smaller than the frame length M. What you read below is therefore not an all-encompassing body of knowledge on frame-based convolution. However, if you understand its principles you will be able to extend it to the case where $M < N$. Making proper computation drawings like the ones that can be found here, is a first step in making such an implementation.

OK. Now let's get our hands dirty.

overlap–add
The overlap–add technique is an input-side method. As the output-buffer style implementation is easier than the input-buffer style implementation, we start with the former.

Output-buffer style This is the first rather involved computation diagram that we consider. Therefore, we'll treat it in detail. The goal of this discussion is that you understand the operation of this algorithm, but also how to read the diagrams for the subsequent algorithms.

Consider Figure 4.6 on page 63 and focus on the top left-hand side. You'll see three separate lines containing buffers. We will use it to explain operation i.
The *first line* contains the output buffer of the previous operation (indicated in yellow) for frame number $n-1$. As such it is the input buffer to the operation i.
The *third line* contains the output buffer of the operation we are considering (frame number n). It is a circular buffer. We will write the results of this filter operation in the yellow and white parts of the buffer.
The *middle line* represents a single buffer (the filter kernel) drawn in three positions during the execution of the algorithm ($2a$ to $2c$). The left one is the start position, the middle one a position in the middle of the execution and the right one is the very last position.

The method starts by clearing all but the first $N-1$ one cells of the yellow and white output buffer. As the overlapp-add method is an *input-side* method, the algorithm takes the first sample of the input buffer and adds the impulse reponse (the filter kernel) scaled by this sample value to the output buffer (position $2a$). Then the second sample is taken and the effect on the output is written into the output buffer. This continues (the intermediate position $2b$ has been indicated). When this process reaches the end of the input buffer (position $2c$), we see that the tail of the scaled impulse response that we write in the output buffer extends beyond the yellow part in the white part. We need to keep this data to add to the next frame (hence the name overlap-*add* method).

Now concentrate on the central part of the figure, containing the execution of the next frame (at the next tick of the frame clock). The *input buffer* is now the output buffer of the previous operation but for the frame number n. The output buffer actually starts with the white portion of the output buffer (at frame clock tick $n-1$) and as such contains the overlap that is to be added to the new calculations. Because of the fact that this portion of the output buffer is not cleared in step 1, and the results of the input-side convolution are added to this buffer, this implicitly adds the tail from the previous frame to this frame. We indicated this with the arrow labeled 'implicit add'.

Input-buffer style This version is less straightforward than the previous one. It also requires an additional copy operation, that makes it less favorable than the previous one.

Consider Figure 4.7 on page 64 and focus on the top left-hand side. You'll see three separate lines containing buffers.
The *second line* contains the input buffer of the current operation (indicated in blue) for frame number $n-1$ preceded by an overlap part (indicated in gray). It is a circular buffer.
The *third line* contains the input buffer of the next operation we are considering. It will be our output buffer.
The *first line* represents a single buffer (the filter kernel) drawn in three positions during the execution of the algorithm ($1a$ to $1c$). The left one is the start position, the middle one a

position in the middle of the execution and the right one is the very last position.

The method starts by reading the first sample of the input buffer, clearing its value (after being read) and adds the impulse reponse (the filter kernel) scaled by this sample value to the input buffer, but starting at the very first position. Note that the final value of the scaled impulse response ends up in the position of the sample value we just read and cleared. Then the second sample is read, cleared and the effect on the output is written into the circular input buffer one position to the right . This continues (the intermediate position $1b$ has been indicated). When this process reaches the end of the input buffer (position $1c$), we see that the $N - 1$ final values of the circular (hence the name overlap-*add* method) contain the tail of the data that we will need to add to the next frame. Finally the first N samples of the circular buffer ar written to the input buffer of the next operation (step 2).

Now concentrate on the central part of the figure, containing the execution of the next frame (at the next tick of the frame clock). The *input buffer* now contains the tail of the previous frame the output buffer of the previous operation (blue part) followed by he new input samples (yellow part). The filter kernel is being run over all input samples again, adding the new scaled impulse responses to this blue tail (effectively causing the implict overlap add) and the yellow samples that have been cleared one by one (just in time).

Overlap–save

The overlap–save technique is an output-side method. As, this time, the input-buffer style implementation is easier than the output-buffer style implementation, we start with the former.

Input-buffer style Consider Figure 4.8 and focus on the top left-hand side. You'll see three separate lines containing buffers.
The *first line* contains the input buffer of the current operation (indicated in blue) for frame number $n - 1$ preceded by the final samples of the previous frame (the overlap part) that have been *saved* from the previous execution of the operation (indicated in gray). It is a circular buffer.
The *third line* contains the input buffer of the next operation we are considering. It will be our output buffer.
The *middle line* represents a single buffer (the filter kernel) drawn in three positions during the execution of the algorithm ($1a$ to $1c$). The left one is the start position, the middle one a position in the middle of the execution and the right one is the very last position.

The method starts by reading the saved old overlap values and the first input sample of the new frame to calculate the first output sample ($1a$). This process continues (calculating the next output sample), over the middle position ($1b$) to reach the end position ($1c$). Note that input buffer contains the original input samples.

Now, concentrate on the central part of the figure. The last samples of the previous frame are still present at the start of the buffer, the new samples haven been written after that saved overlap. Because of this access to the saved samples of the previous frame, all output samples of this frame can be calculatd appropriately.

Output-buffer style This version is less straightforward than the previous one. It also requires an additional copy operation, that makes it less favorable than the previous one.

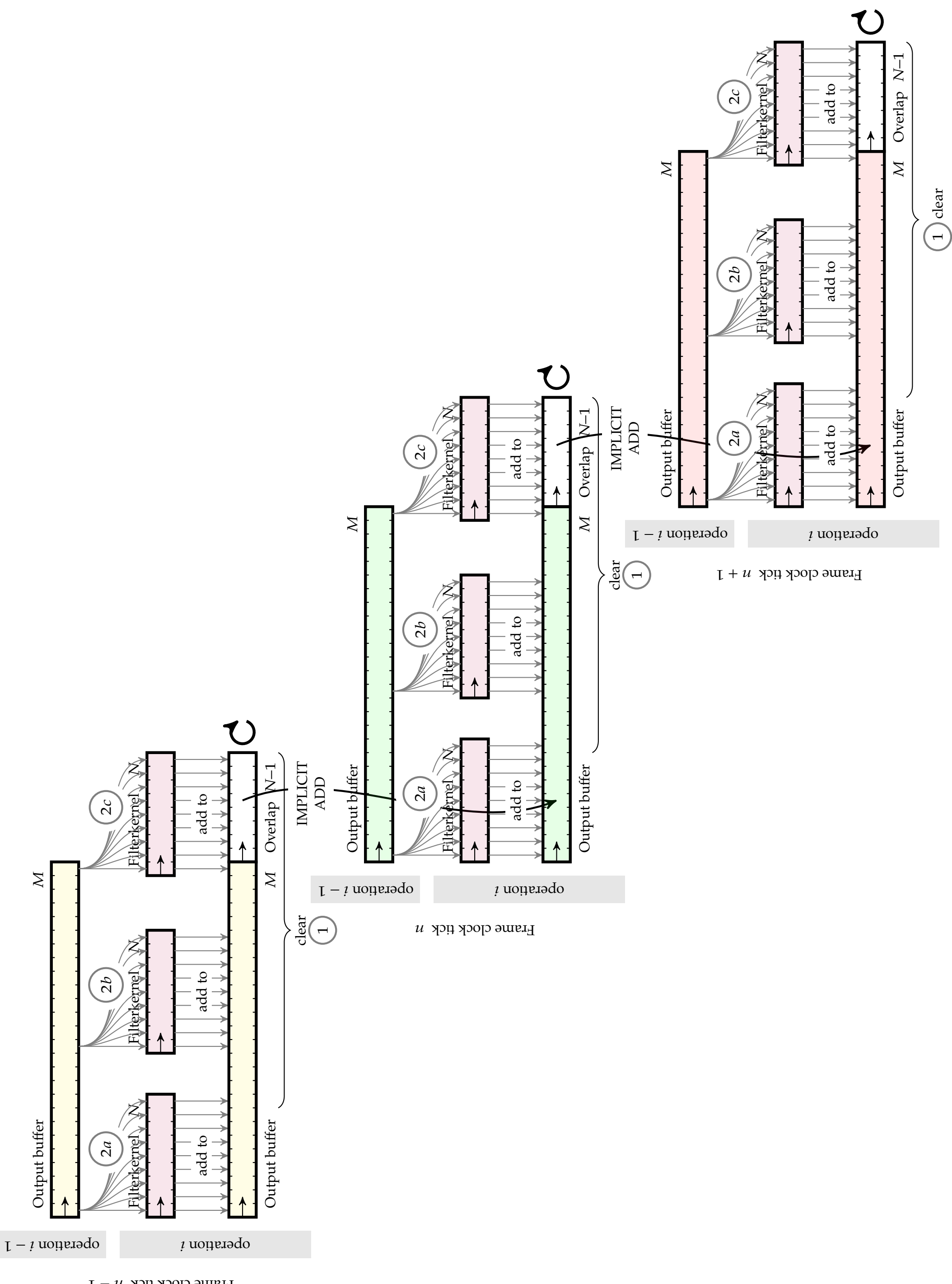


Figure 4.6: Computation diagram of a single-buffer overlap–add output-buffer style implementation in the time domain

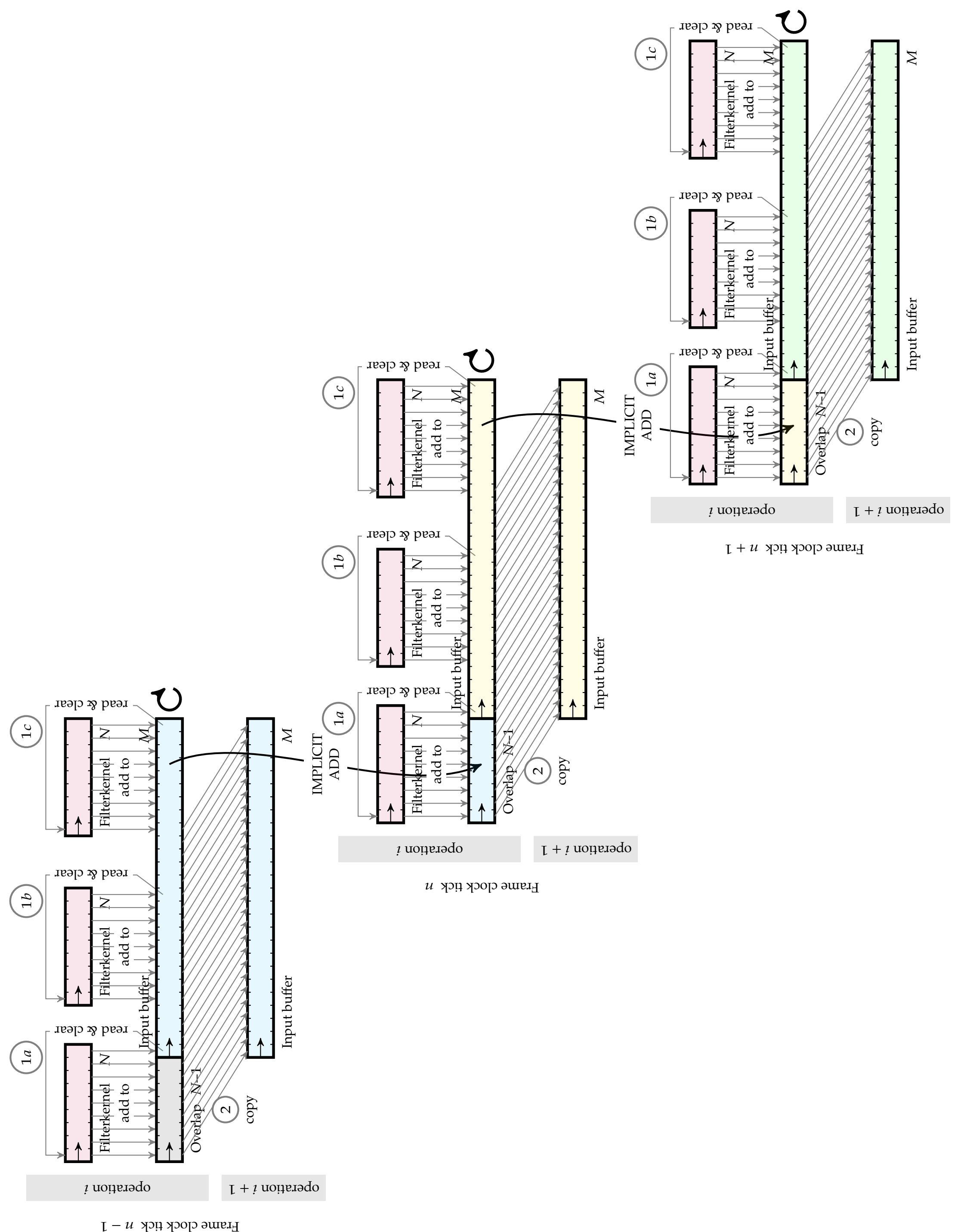


Figure 4.7: Computation diagram of a single-buffer overlap–add input-buffer style implementation in the time domain

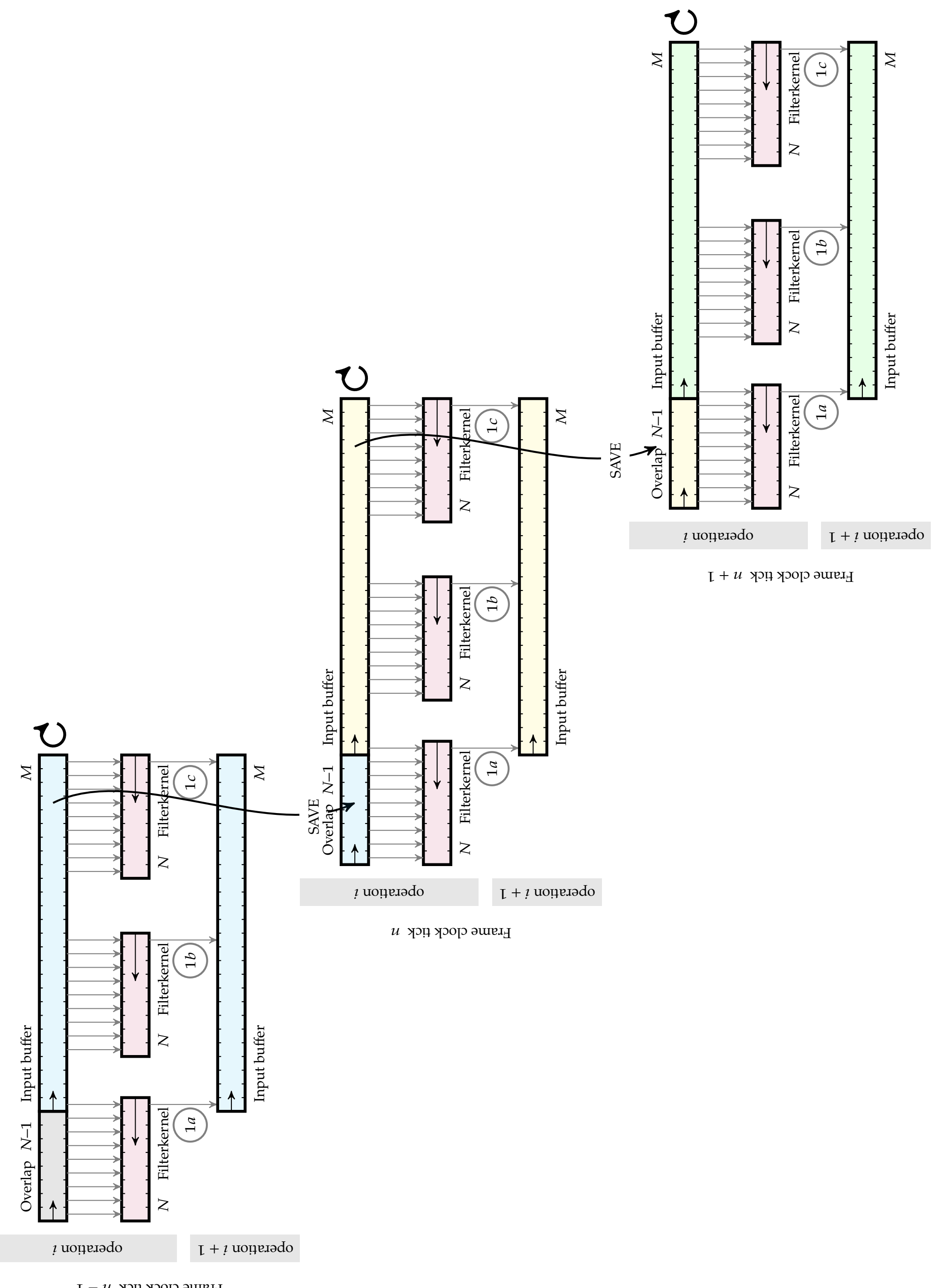


Figure 4.8: Computation diagram of a single-buffer overlap–save input-buffer style implementation in the time domain

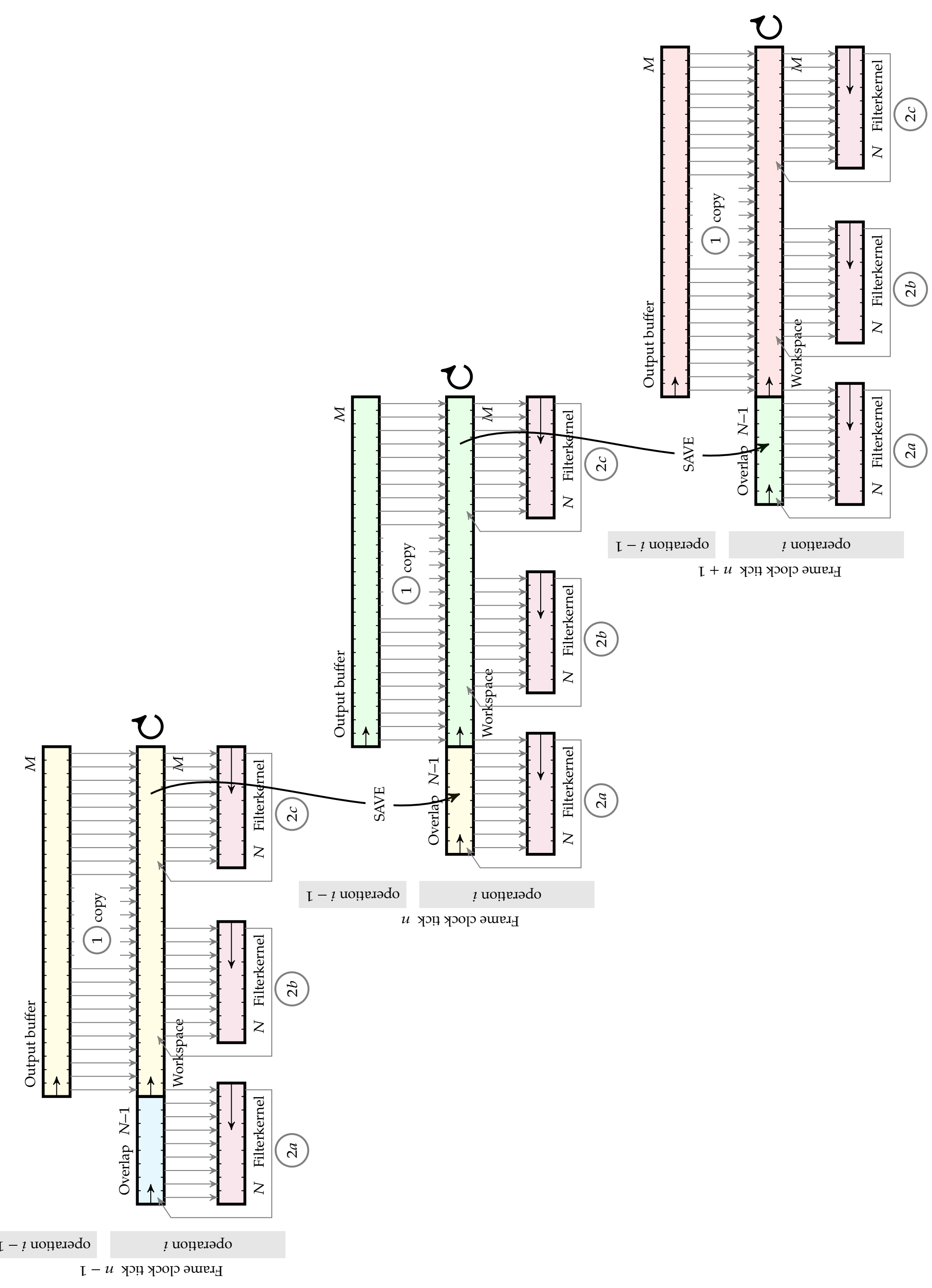


Figure 4.9: Computation diagram of a single-buffer overlap–save output-buffer style implementation in the time domain

Consider Figure 4.9 and focus on the top left-hand side. You'll see three separate lines containing buffers.
The *first line* contains the output buffer of the previous operation (indicated in yellow) for frame number $n-1$.
The *second line* contains our own output buffer that starts with the saved overlap part of the previous frame (the blue part) followed by the yellow part in which we will perform our work. It is a cicrcular buffer.
The *third line* represents a single buffer (the filter kernel) drawn in three positions during the execution of the algorithm ($2a$ to $2c$). The left one is the start position, the middle one a position in the middle of the execution and the right one is the very last position.

The method starts by copying the new input samples from the output buffer of the previous operation to our workspace right behind the saved overlap (step 1). Then the filter kernel is used to calculate the first output value (based on all saved overlap values and the first new sample), step $2a$. This output is written in the first position of the circular buffer (we no longer need that sample for the further calculations). This continues with the second sample, over the intermedidate situation (step $2b$) and the final situation (step $2c$). Note that at that moment the final $N-1$ samples in the workspace still contain the input samples that need to be saved.

Now, concentrate on the central part of the figure. The last samples of the previous frame are still present at the start of the output buffer of operation i (yellow part). Together with the new input samples that have been copied to follow the saved samples, we can calculate the correct outputs using the output-side algorithm.

4.7.3 Frame-based double buffer operation

We split the sample data in frames of length M and treat a single frame at every clock tick of the frame clock, that runs at a rate equal to the sample clock divided by M. However, now we have double buffers to allow for pipelining the entire singal processing chain.

By now, you should have familiarized yourself enough to be able to read the sequence of operations in these diagrams without any further help. Make sure you understand them!

You will find the following diagrams:

- overlap–add - output-buffer style (see Figure 4.10 on the next page)
- overlap–add - input-buffer style (see Figure 4.11 on page 69)
- overlap–add - input-buffer style (see Figure 4.12 on page 70)
- overlap–add - output-buffer style (see Figure 4.13 on page 71)

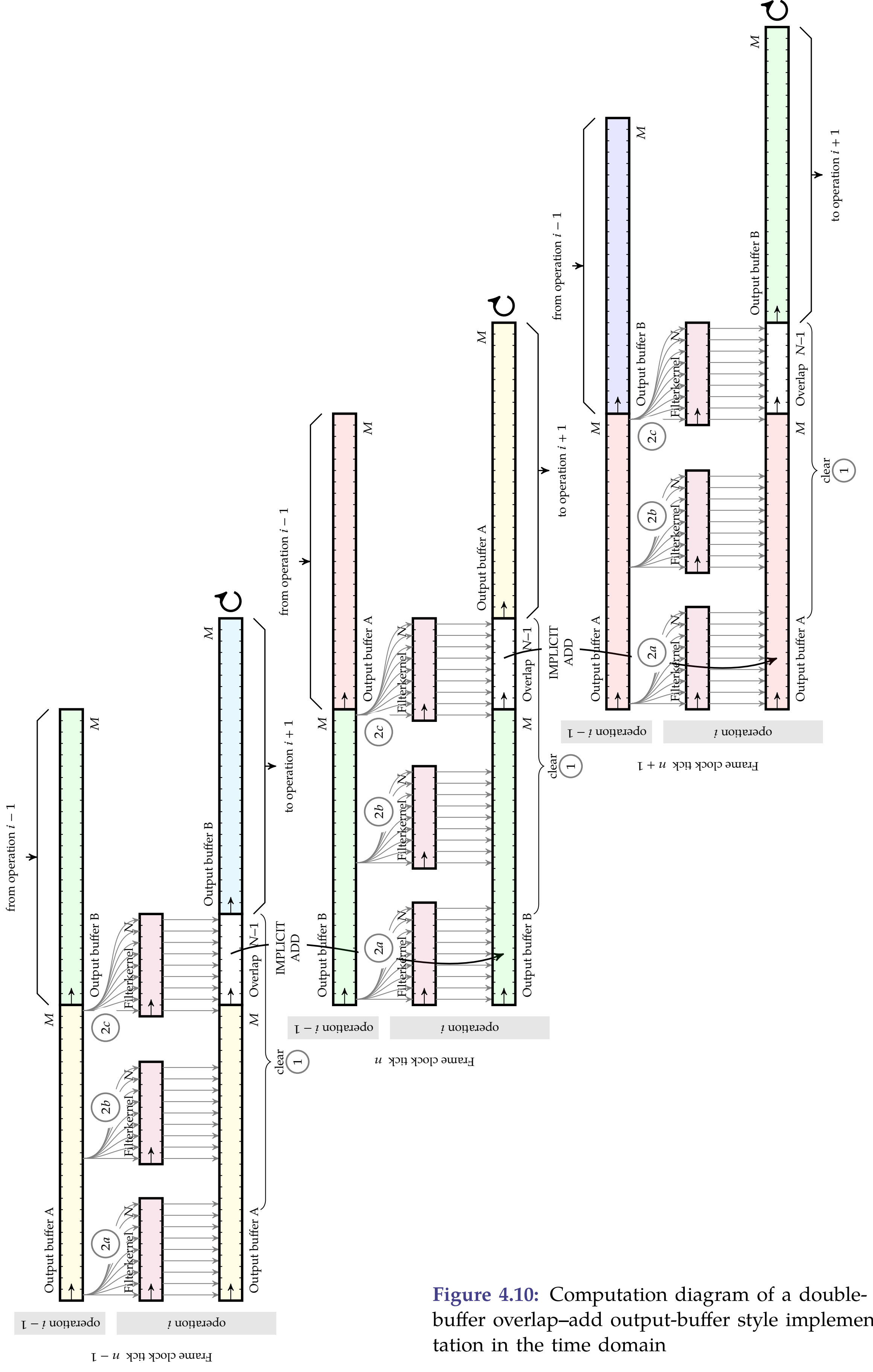


Figure 4.10: Computation diagram of a double-buffer overlap–add output-buffer style implementation in the time domain

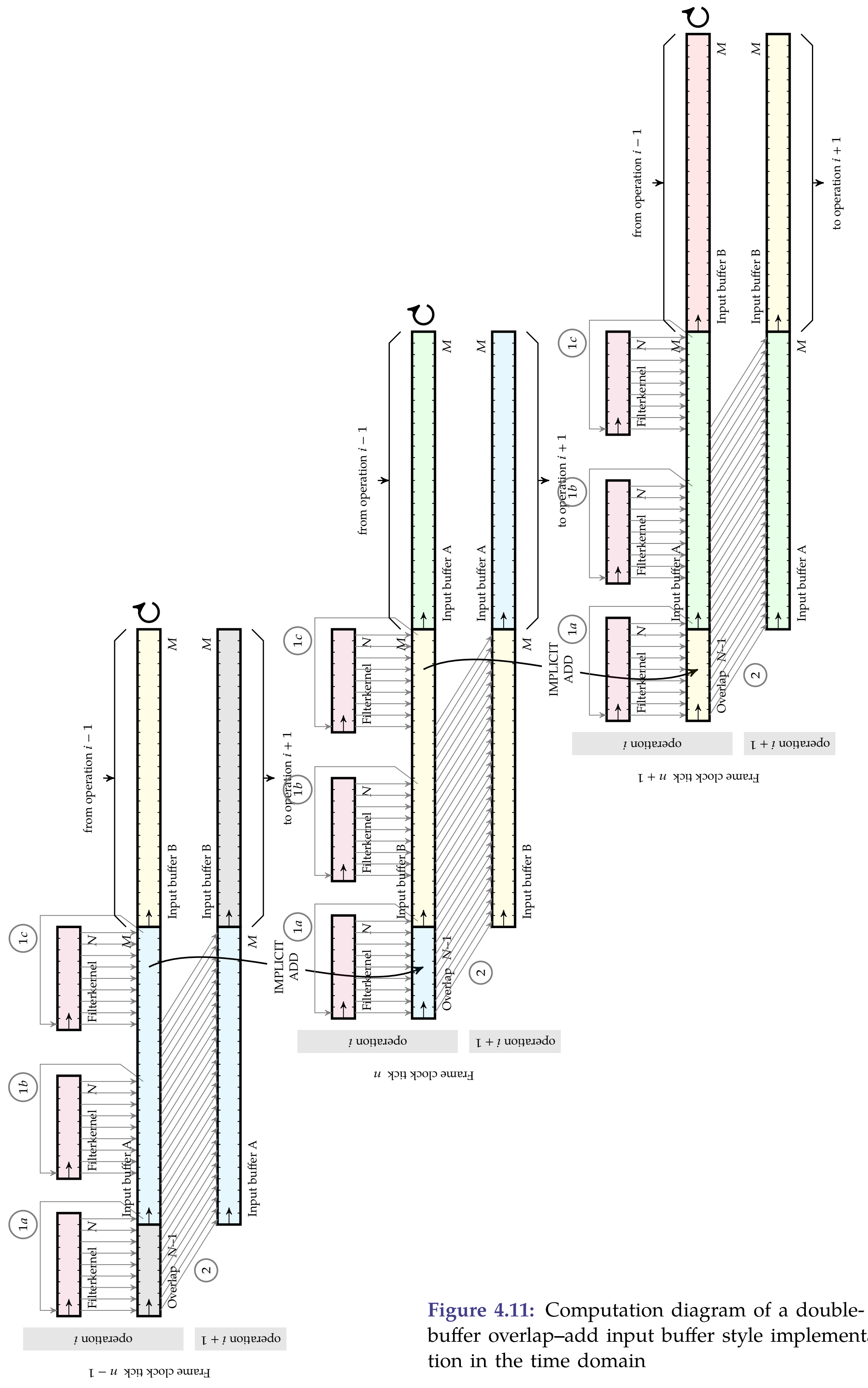


Figure 4.11: Computation diagram of a double-buffer overlap–add input buffer style implementation in the time domain

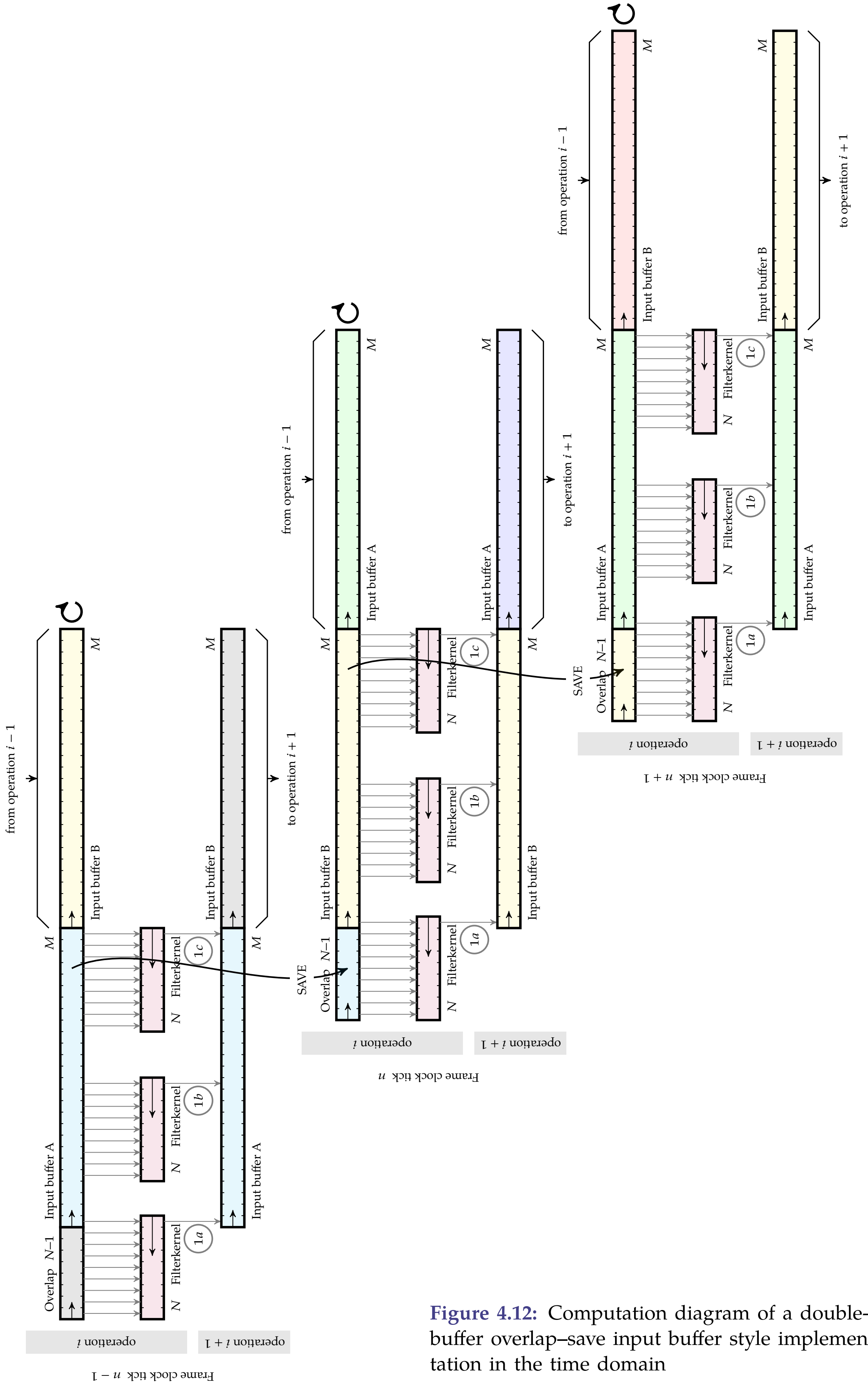


Figure 4.12: Computation diagram of a double-buffer overlap–save input buffer style implementation in the time domain

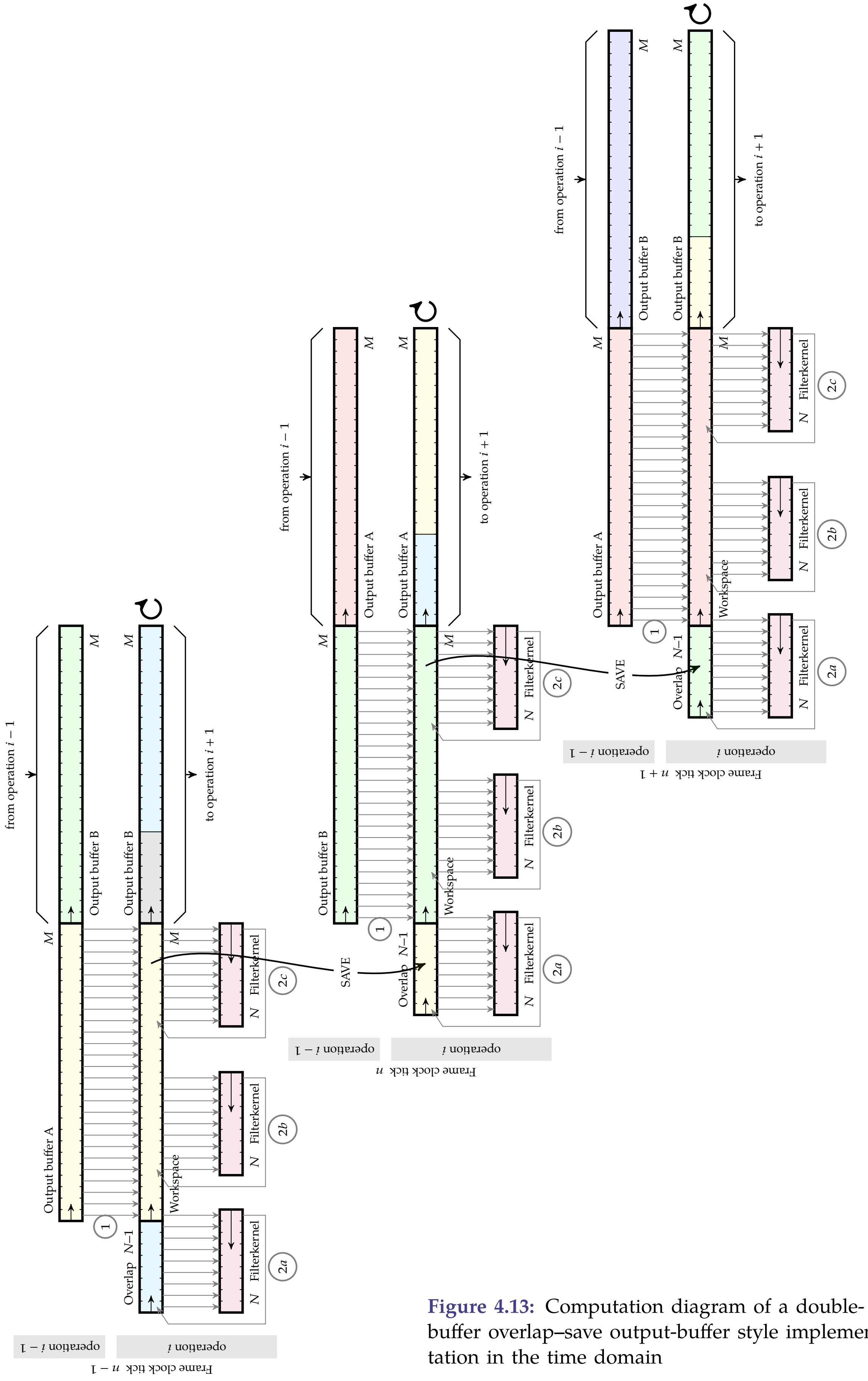


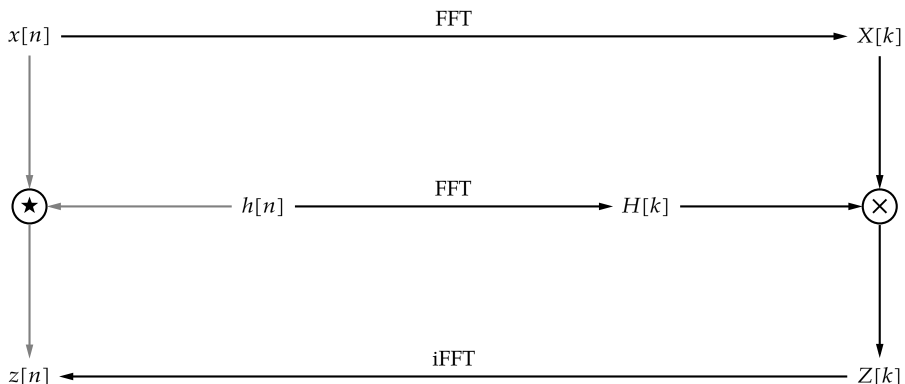
Figure 4.13: Computation diagram of a double-buffer overlap–save output-buffer style implementation in the time domain

4.8 Implementation schemes in the frequency domain

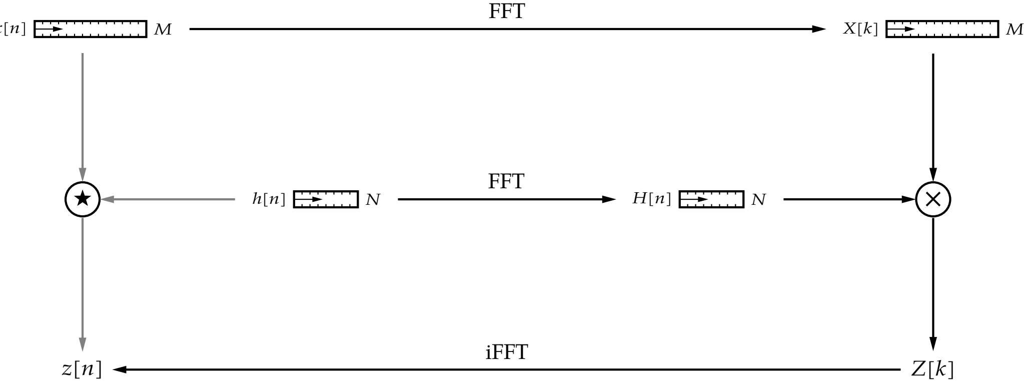
4.8.1 Basic principle

As you will remember from studying the Fourier family of transformations, convolution in the time-domain corresponds to multiplication in the frequency domain. In view of this, a very plausible idea probably did pop up in your mind: can't we use the combination FFT—multiplication—iFFT to perform convolution?

We denote this technique for short as *FFT-convolution*. This idea has been depicted below:

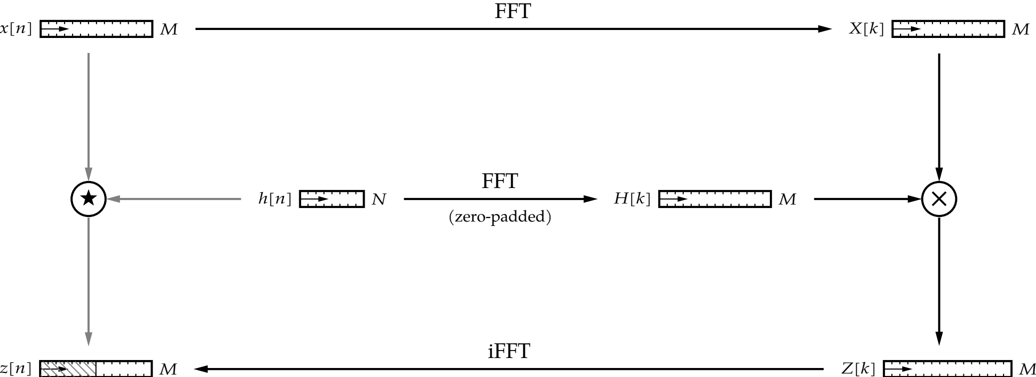


Let's take this idea a step further and replace the $x[n]$, $h[n]$, $X[k]$, and $H[k]$ vector symbols by some more explicit vector pictograms showing the vector lengths. Take your time to make sure you understand the indicated lengths on the drawing below.



A first problem arises as the multiplication on the right hand-side requires its inputs to be of the same length. We can solve this by zero-padding the impulse response $h[n]$ (or zero-padding $x[n]$ in case it is shorter than the latter).

This results in the drawing below in which we also replaced $Z[k]$ and $z[n]$ by their vector pictograms.



It may be not so obvious from the drawing above, but: “Houston, we have a problem!” Indeed, the direct convolution of an M-point signal with a N-point impulse response, yields an $M + N - 1$-point output signal. On the contrary, the FFT—multiplication—iFFT route results in an M-point signal. What's wrong?

If you do remember well: the last step in deriving the DFT, involved frequency sampling, which causes a repetition of the signal in the time domain. Stated differently, FFT-convolution does not realize a direct convolution, but it realizes a circular convolution. This means that the first $N - 1$ output samples are polluted by the last $N - 1$ samples of the input signal. We indicated this with the hatching pattern in the drawing above. Subtle, isn't it?

Hence, in order to be able to use FFT-convolution as a full replacement for a direct convolution, we need to take measures to avoid “the tail hitting the head”, or we need to make sure that we can live with this destructive process.

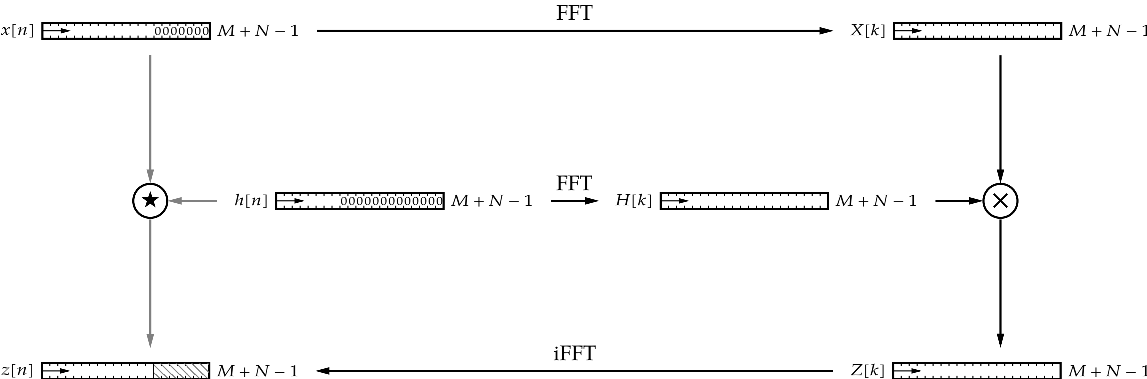
The two possible solutions are two techniques well known from earlier in this chapter:

- the overlap–add method, and
- the overlap–save method.

Overlap–add method

The idea is to pad the input signal with $N - 1$ zeros and to pad the impulse response with $M - 1$ zeros, such that the result of the FFT-convolution fits in without destructive aliasing.

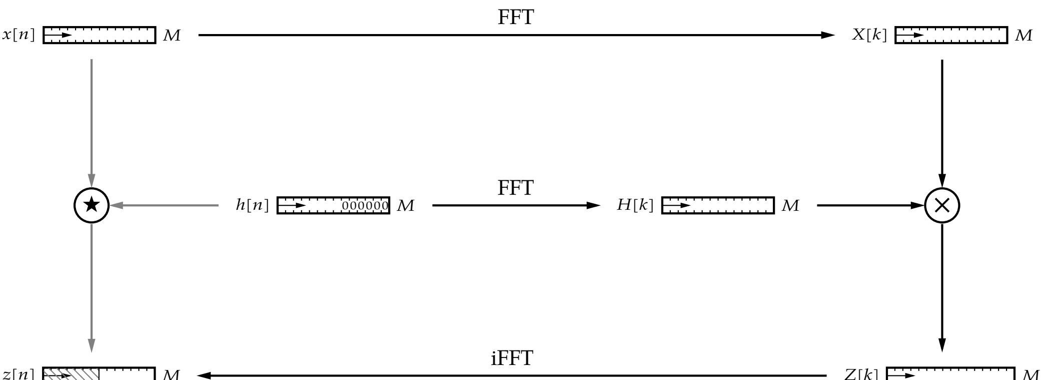
This principle has been depicted below:



The zero-padding has been indicated, and the tail that needs to be added to the head of the next frame has been indicated by right hatching.

Overlap–save method

The idea here is to discard the samples that are corrupted due to destructive circular convolution. These are the first $N-1$ samples. They haven been crosshatched in the drawing below:



4.8.2 Frame-based single buffer operation

The ideas presented in the previous section have been implemented in the following single-buffer overlap–add and overlap–save FFT implementations.

You will find the following diagrams:

- overlap–add - output-buffer style (see Figure 4.14 on page 76)
- overlap–add - input-buffer style (see Figure 4.15 on page 77)
- overlap–save - output-buffer style (see Figure 4.16 on page 78)
- overlap–save - input-buffer style (see Figure 4.17 on page 79)

4.8.3 Frame-based double buffer operation

The same ideas have also been implemented in the following double-buffer overlap–add and overlap–save FFT implementations.

You will find the following diagrams:

- overlap–add - output-buffer style (see Figure 4.18 on page 80)
- overlap–add - input-buffer style (see Figure 4.19 on page 81)
- overlap–save - output-buffer style (see Figure 4.20 on page 82)
- overlap–save - input-buffer style (see Figure 4.21 on page 83)

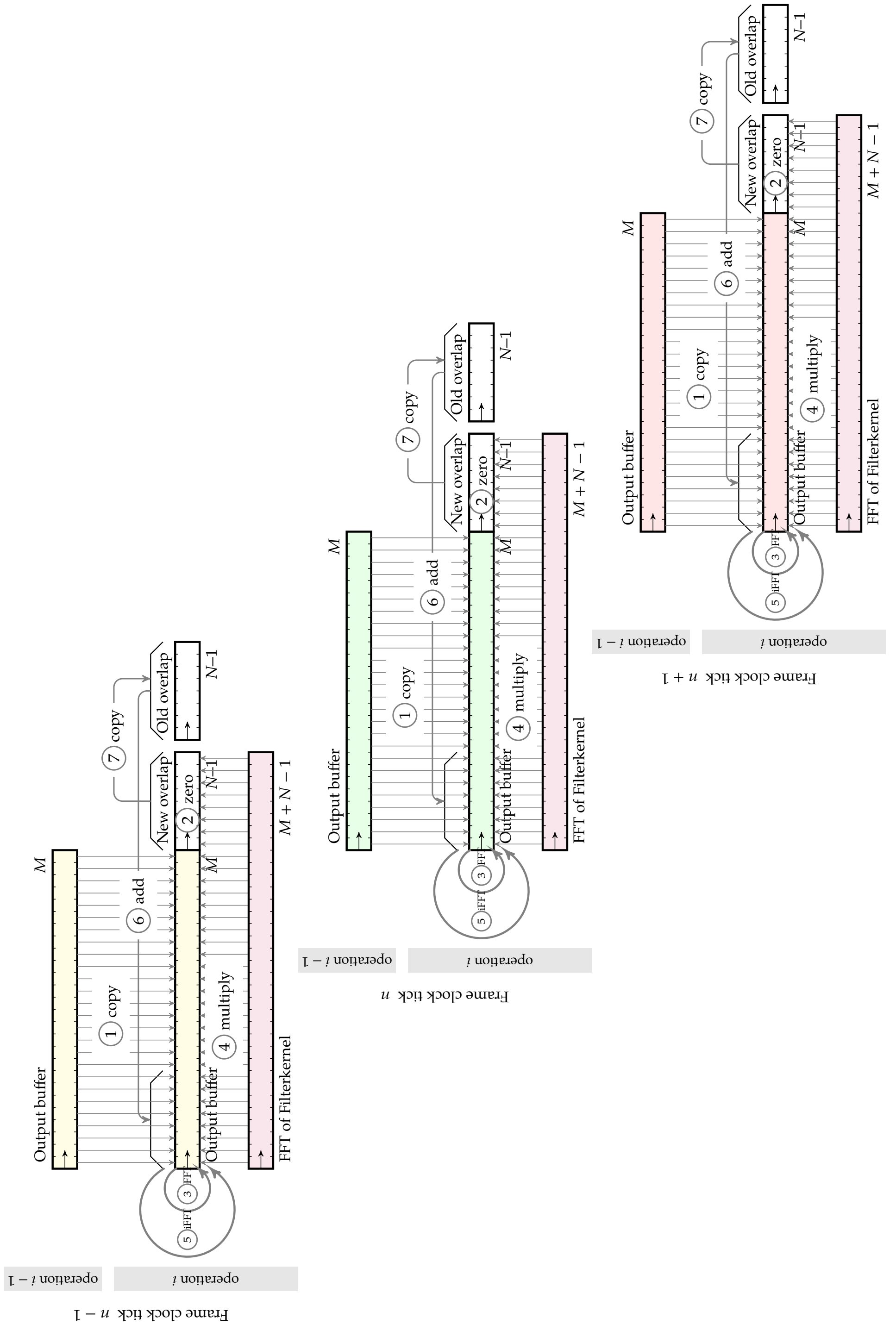


Figure 4.14: Computation diagram of a single-buffer overlap–add output buffer style implementation in the frequency domain

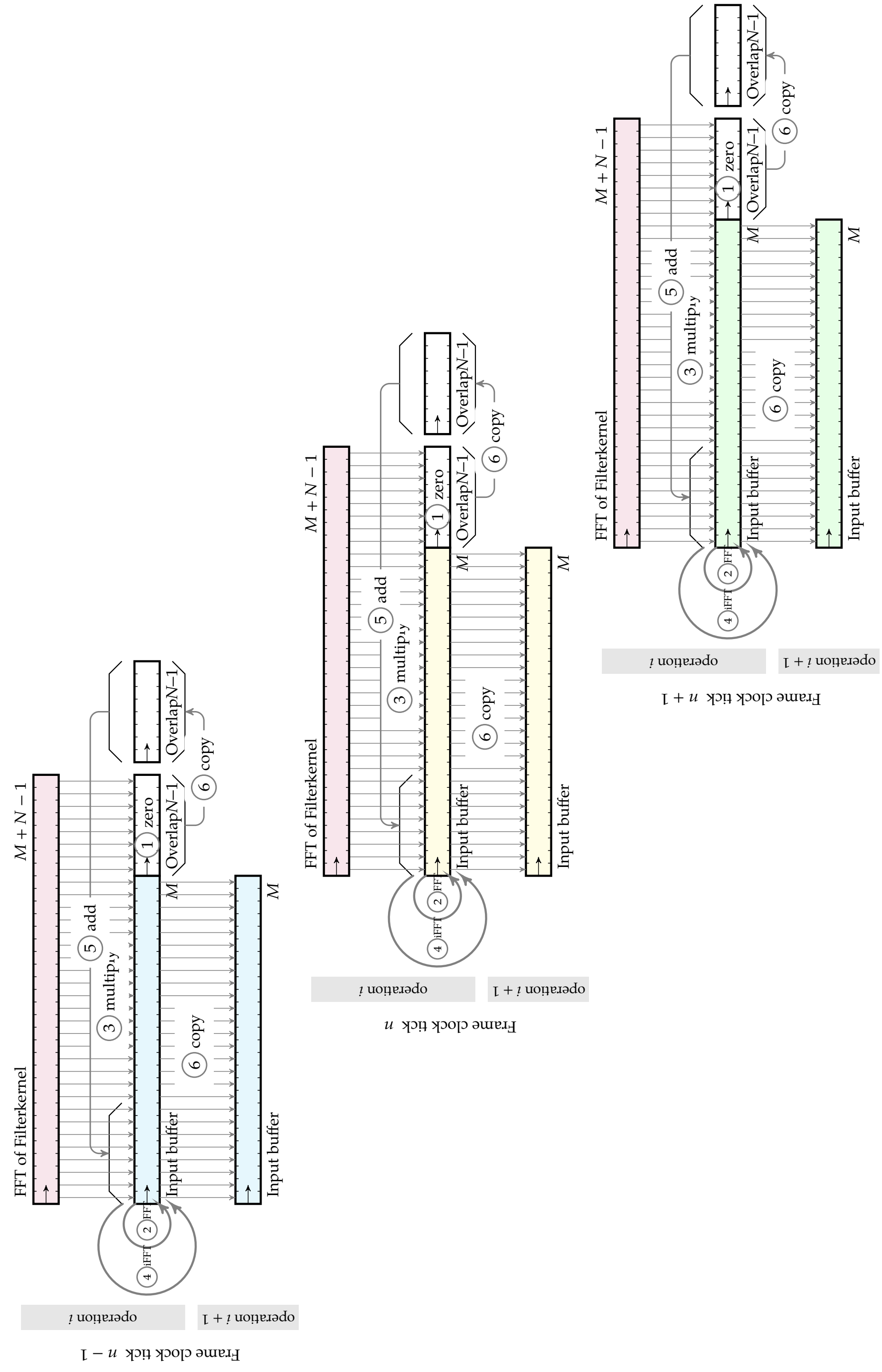


Figure 4.15: Computation diagram of a single-buffer overlap–add input buffer style implementation in the frequency domain

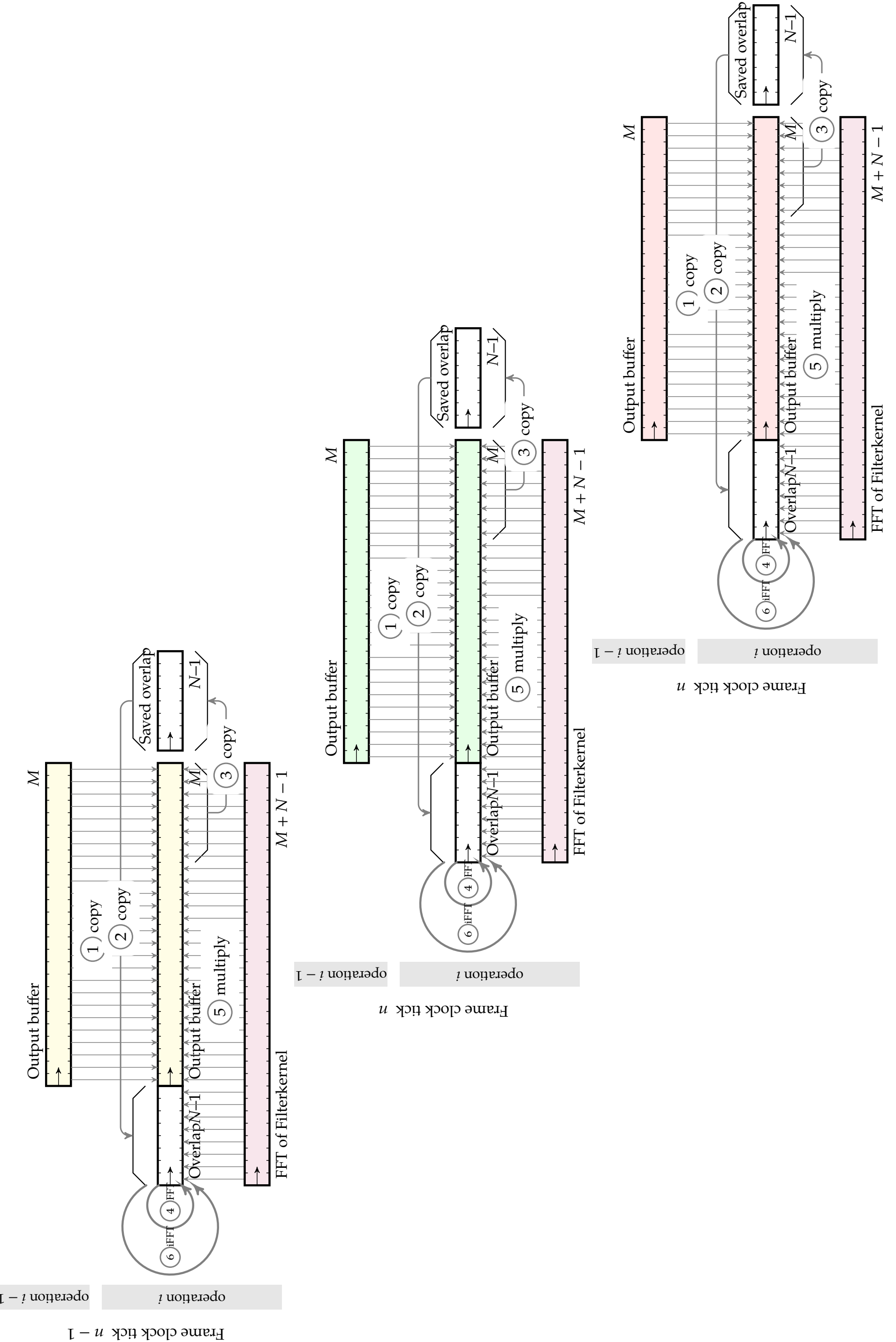


Figure 4.16: Computation diagram of a single-buffer overlap–save output buffer style implementation in the frequency domain

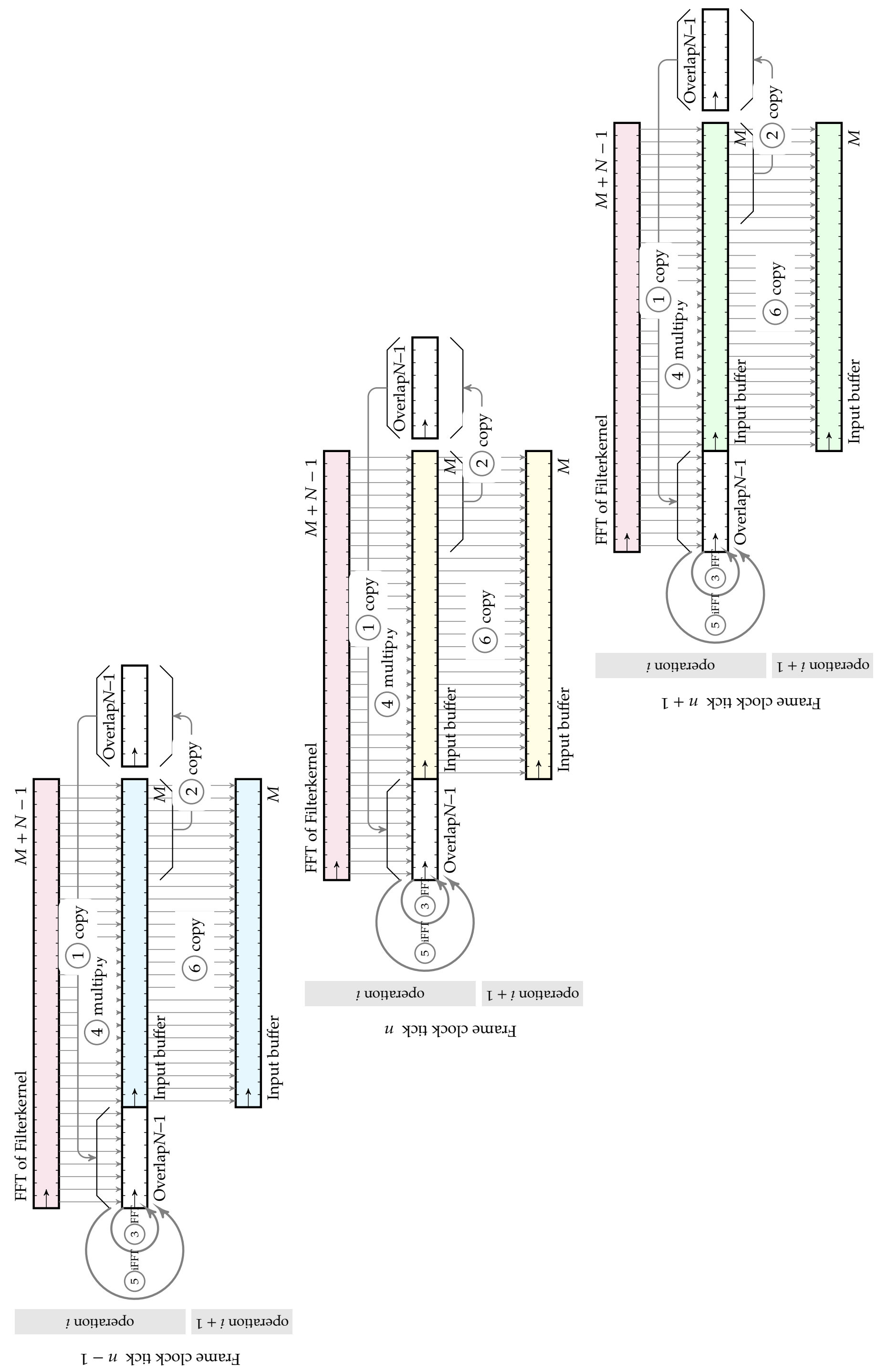


Figure 4.17: Computation diagram of a single-buffer overlap–save input buffer style implementation in the frequency domain

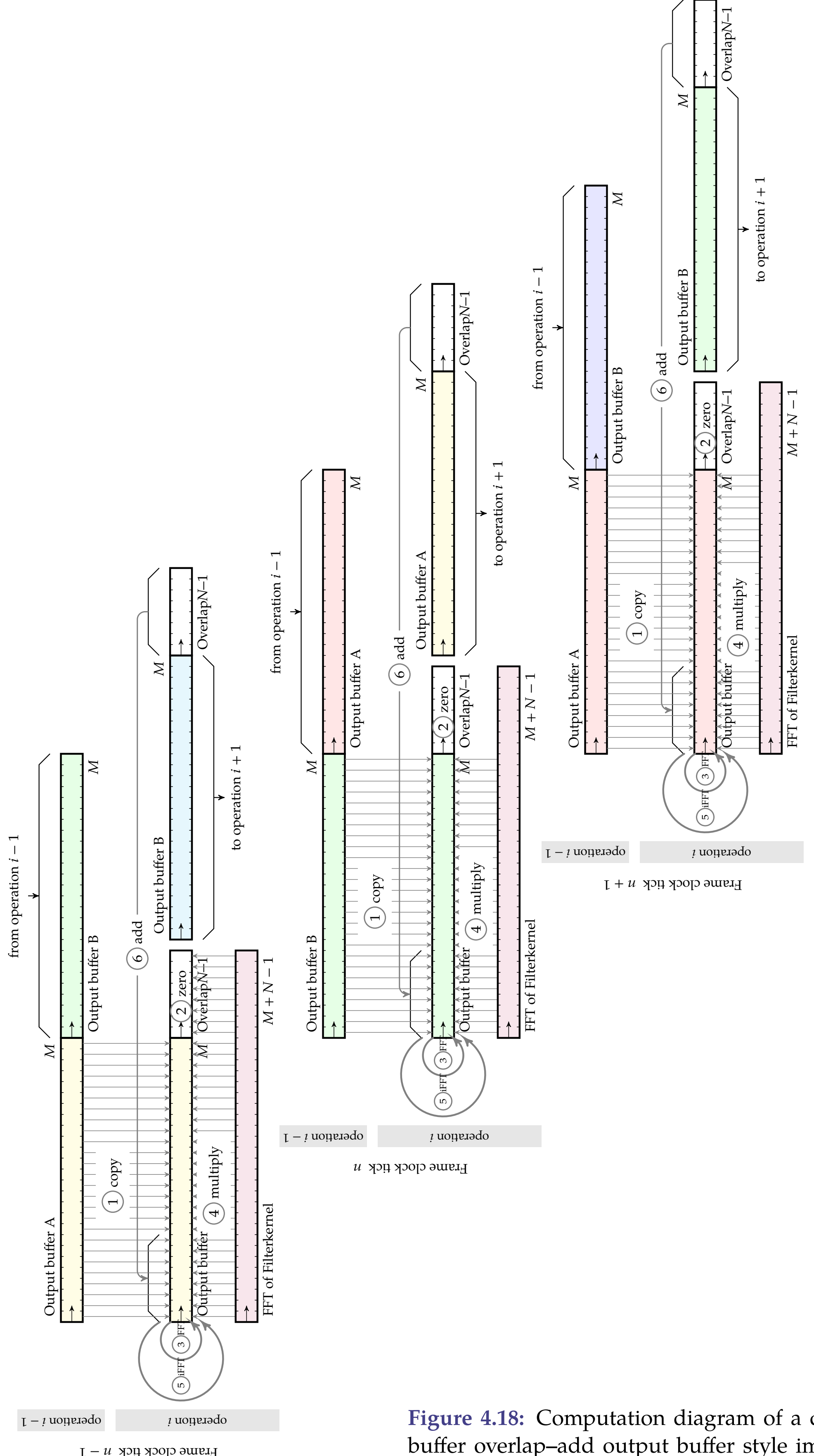


Figure 4.18: Computation diagram of a double-buffer overlap–add output buffer style implementation in the frequency domain

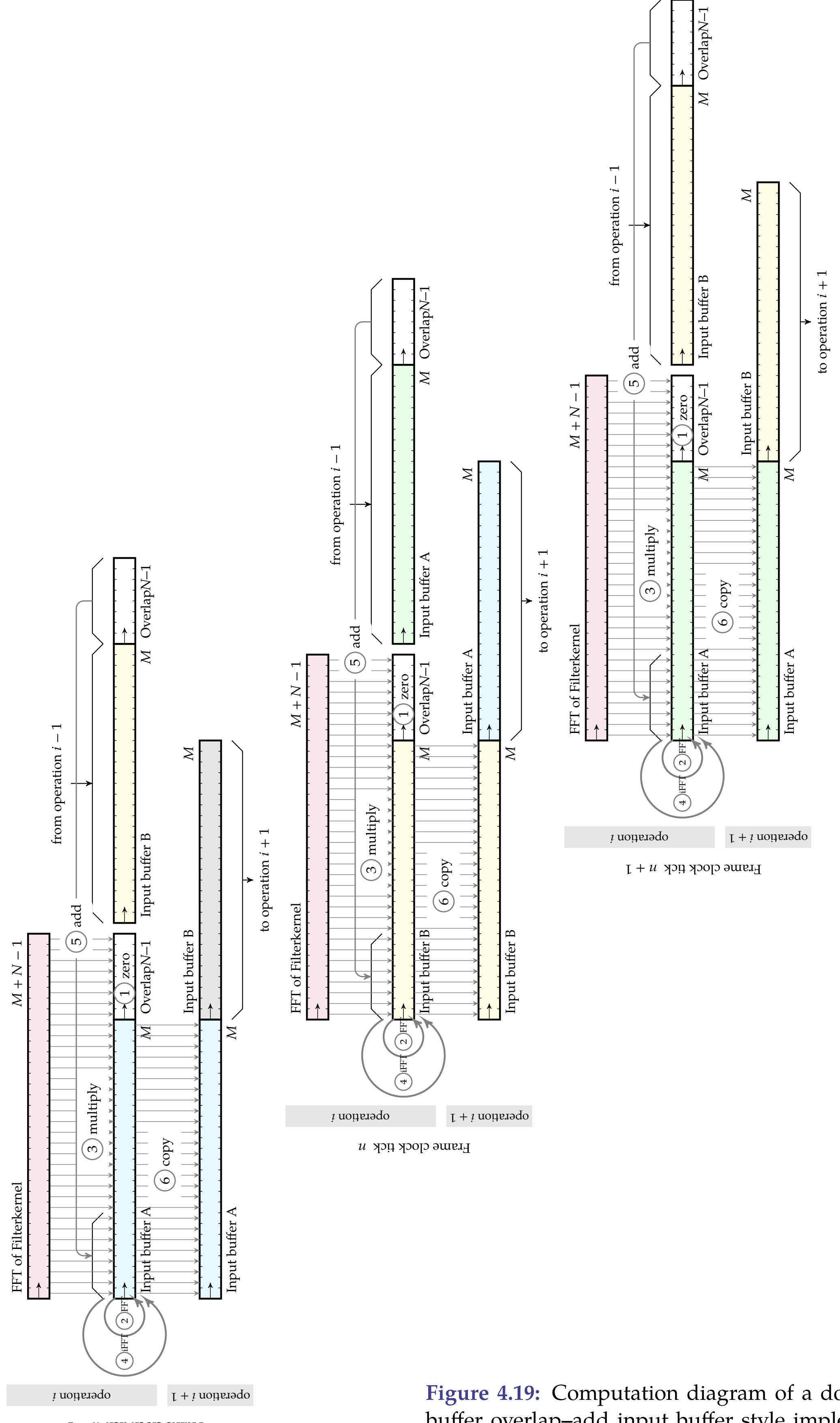


Figure 4.19: Computation diagram of a double-buffer overlap–add input buffer style implementation in the frequency domain

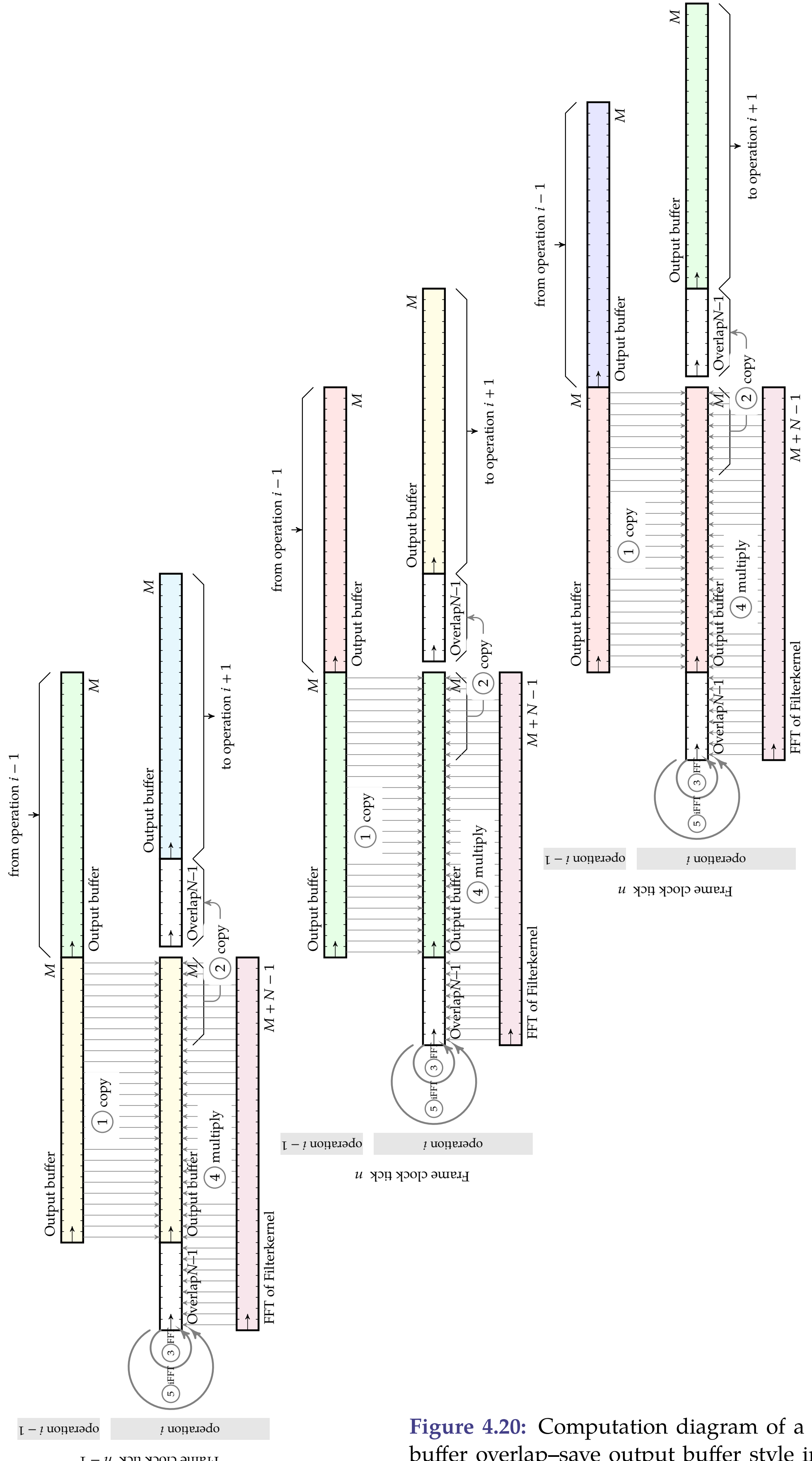


Figure 4.20: Computation diagram of a double-buffer overlap–save output buffer style implementation in the frequency domain

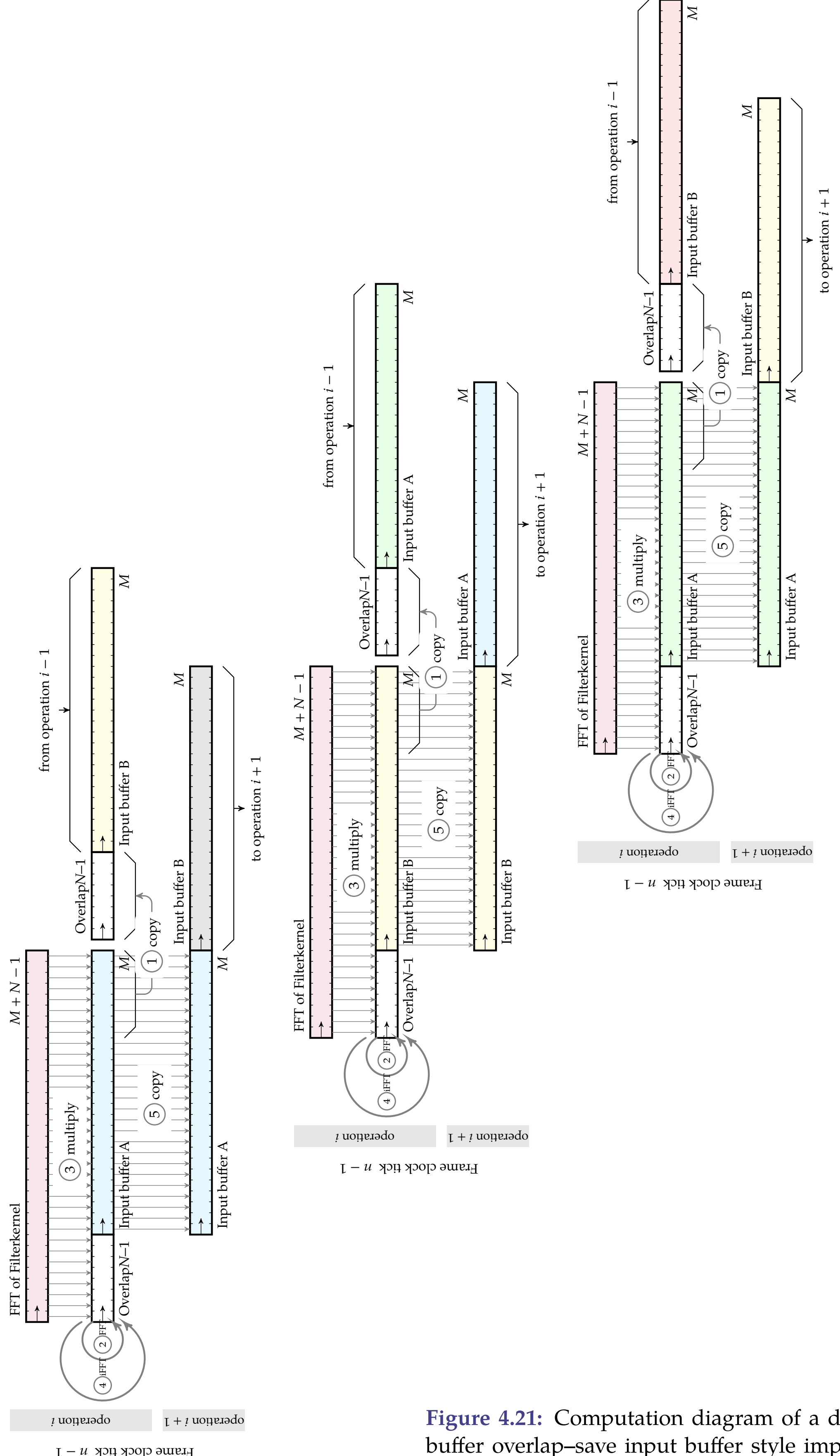


Figure 4.21: Computation diagram of a double-buffer overlap–save input buffer style implementation in the frequency domain

4.9 Choosing the right convolution method

It is time to take a step back from all the algorithms and computation diagrams that we've swallowed so far. To summarize:

- incremental algorithms in the time domain[4]
  - input-side algorithms
  - output-side algorithms
- frame-based algorithms in the time domain
  - overlap–add / input-side algorithms
  - overlap–save / output-side algorithms
- frame-based algorithms in the frequency domain
  - overlap–add / input-side algorithms
  - overlap–save / output-side algorithms

The frame-based algorithms all can be constructed in an input-buffer style and output-buffer style setup, either using a single-buffer approach or a double-buffer approach. It's nice to have these 18 alternatives, but...which one is best? We'll take a look at this "contest" for every relevant aspect in the next subsections.

What are the relevant aspects? Obviously:

- Memory consumption
- Required computational power
- Optimal use of the available computational power
- Latency

The results of these analyses is given in Table 4.1 on the next page. It's up to you to pick your favorite algorithm. Let's disucss them one by one, such that you understand each and every entry of the table.

4.9.1 Incremental algorithms

Consider the diagrams of Figure 4.4 on page 58 and Figure 4.5 on page 60. For both these diagrams, the calculations below hold.

Latency The incremental algorithms are the champions when it comes to latency. Only a single sample clock tick is required to calculate the next output.

$$L = T_s$$

The unit of L is s.

[4]Note that there are no incremental algorithms in the frequency domain. Can you figure out why?

Domain	Type	Buffer strategy	Method	Buffer style	Total	Per operation			Per sample
					Latency (L)	# Mem cells (R_i)	# COPs	# MACs	# MACs
Time	incremental	-	I-side	-	1	$2N+2$	1	N	N
	incremental	-	O-side	-	1	$2N+2$	1	N	N
Time	frame-based	single	OA	I	$2M$	$K+N$	M	MN	N
				O	$2M$	$K+N$	0	MN	N
			OS	I	$2M$	$K+N$	0	MN	N
				O	$2M$	$K+N$	M	MN	N
		double	OA	I	$(1+V)M$	$2K+1$	M	MN	N
				O	$(1+V)M$	$2K+1$	0	MN	N
			OS	I	$(1+V)M$	$2K+1$	0	MN	N
				O	$(1+V)M$	$2K+1$	M	MN	N
Frequency	frame-based	single	OA	I	$2M$	$3K-M$	K	$2K(1+2\log_2 K)+N-1$	$\leftarrow /M$
				O	$2M$	$3K-M$	K	$2K(1+2\log_2 K)+N-1$	$\leftarrow /M$
			OS	I	$2M$	$3K-M$	$K+N-1$	$2K(1+2\log_2 K)$	$\leftarrow /M$
				O	$2M$	$3K-M$	$K+N-1$	$2K(1+2\log_2 K)$	$\leftarrow /M$
		double	OA	I	$(1+V)M$	$3K$	$K-(N-1)$	$2K(1+2\log_2 K)+N-1$	$\leftarrow /M$
				O	$(1+V)M$	$3K$	$K-(N-1)$	$2K(1+2\log_2 K)+N-1$	$\leftarrow /M$
			OS	I	$(1+V)M$	$3K$	K	$2K(1+2\log_2 K)$	$\leftarrow /M$
				O	$(1+V)M$	$3K$	K	$2K(1+2\log_2 K)$	$\leftarrow /M$

with: $K = M + N - 1$

Legend

Symbol	Meaning	Abbrev.	Meaning	Abbrev.	Meaning
M	the frame buffer size	OA	overlap add	MAC	multiply-accumulate operation
N	convolution kernel length	OS	overlap–save	COP	copy operation
V	total number of operations	I	input		
		O	output		

Table 4.1: Computational comparison of the different convolution implementations

Amount of memory Also regarding memory consumption, incremental algorithms are unbeatable. The amount of required memory cells is easily calculated as:

$$R_i = \underbrace{1}_{\text{input sample}} + \underbrace{N}_{\text{impulse response}} + \underbrace{N}_{\text{intermediate storage}} + \underbrace{1}_{\text{output sample}} = 2N + 2$$

The unit of R_i is number of memory cells. To obtain the memory requirement in number of bytes, we need to multiply with the number of bytes per sample.

Computational complexity Every sample requires N multiply-and-accumulate operations, i.e.

$$T_i = NT_{MAC} + T_{COP} \approx NT_{MAC}$$

In the latter step, we neglected the copy operation. The unit of T_i is s.

4.9.2 Frame-based algorithms in the time domain

4.9.2.1 Single buffer implementations

Consider Figure 4.6 until Figure 4.9. Using these diagrams we can derive the following properties:

Latency The input buffer needs to be filled before we can start, and then it takes another frame to perform the computations:

$$L = 2 \cdot M \cdot T_s$$

Amount of memory The amount of memory can clearly be inspected on the diagrams:

$$R_i = \underbrace{N}_{\text{Filterkernel}} + \underbrace{M}_{\text{Input or output buffer}} + \underbrace{N - 1}_{\text{Overlap}} = M + 2N - 1$$

Computational complexity The computational complexity per frame can be derived from the diagrams. For the OA version with output-style buffers and the OS version with input-style buffers, this amounts to:

$$T_i = M \cdot N \cdot T_{MAC}$$

For the OA version with input-style buffers and the OS version with output-style buffers, we need an extra copy operation, and therefore:

$$T_i = M \cdot N \cdot T_{MAC} + M \cdot T_{COP} \approx M \cdot N \cdot T_{MAC}$$

Again, if copy operations are well optimized in the processor, then they can often be neglected.

Note that the computational complexity (per sample) is the same as for the incremental implementations.

4.9.2.2 Double buffer implementations

Consider Figure 4.10 until Figure 4.13. Using these diagrams we can derive the following properties:

Latency The input buffer needs to be filled before we can start, therefore the inherent latency for the operation itself amounts to

$$L = M \cdot T_s$$

However, we need to keep in mind that chosing for the double-buffer operation implies pipelining and this will increase the overall latency to

$$L = (1 + V)M \cdot T_s$$

with V the total number of operations.

Amount of memory The amount of memory can clearly be inspected on the diagrams:

$$R_i = \underbrace{N}_{\text{Filterkernel}} + \underbrace{M}_{\text{Input or output buffer A}} + \underbrace{M}_{\text{Input or output buffer B}} + \underbrace{N-1}_{\text{Overlap}} = 2M + 2N - 1$$

Computational complexity The computational complexity per frame can also be derived from the diagrams. For the OA version with output-style buffers and the OS version with input-style buffers, this amounts to:

$$T_i = M \cdot N \cdot T_{MAC}$$

For the OA version with input-style buffers and the OS version with output-style buffers, we need an extra copy operation, and therefore:

$$T_i = M \cdot N \cdot T_{MAC} + M \cdot T_{COP} \approx M \cdot N \cdot T_{MAC}$$

Again, if copy operations are well optimized in the processor, then they can often be neglected.

Note that the computational complexity (per sample) is the same as for the incremental implementations.

4.9.3 Frame-based algorithms in the frequency domain

Before we start with considering all the implementations, let's focus on the computational complexity of a DFT and its efficient cousin, the FFT.

Both algorithms are based on complex multiply-accumulate operations (MACs). Let's check their computational complexity. Consider multiplying $a + jb$ with $c + jd$, adding its result to $u + jv$ (with $a, b, c, d, u, v \in \mathbb{R}$):

$$u + jv + (a + jb)(c + jd) = (\underbrace{\underbrace{u + ac}_{MAC1} - bd}_{MAC2}) + j(\underbrace{\underbrace{v + bc}_{MAC3} + ad}_{MAC4})$$

The (i)DFT of a K-point complex signal requires K^2 complex MACs. The (i)FFT does this quite a bit better, requiring only $K \log_2 K$ complex MACs.

However, knowing that we are treating real signals, we only need to compute half of the coefficients (or alternately can pack two real signals into a single complex one) and therefore we can halve the work.

This has been summarized in the table below:

Algorithm	# Real MACs
Complex (i)DFT of length K	$4K^2$
Real (i)DFT of length K	$2K^2$
Complex (i)FFT of length K	$4K \log_2 K$
Real (i)FFT of length K	$2K \log_2 K$

4.9.3.1 Single buffer implementations

Consider Figure 4.14 until Figure 4.17. Using these diagrams we can derive the following properties:

Latency The input buffer needs to be filled before we can start, and then it takes another frame to perform the computations:

$$L = 2 \cdot M \cdot T_s$$

Amount of memory The amount of memory can clearly be inspected on the diagrams:

$$R_i = \underbrace{M + N - 1}_{\text{FFT of Filterkernel}} + \underbrace{M}_{\text{Input or output buffer}} + \underbrace{N - 1}_{\text{Old overlap}} + \underbrace{N - 1}_{\text{New overlap}} = 2M + 3N - 3$$

Computational complexity The computational complexity per frame can be derived from the diagrams. For the OA versions, this amounts to an FFT, a complex multiplication in the frequency domain for half of the coefficients (because of Hermiticity), and an iFFT followed by adding the overlap. Counting the latter as full MACs[5], results in:

$$\begin{aligned} T_i &= (\underbrace{2K\log_2 K}_{\text{FFT}} + \underbrace{2K}_{\text{multiplication}} + \underbrace{2K\log_2 K}_{\text{iFFT}} + \underbrace{N-1}_{\text{Add overlap}}) \cdot T_{MAC} + K \cdot T_{COP} \\ &= (2K(1+2\log_2 K) + N - 1) \cdot T_{MAC} + K \cdot T_{COP} \\ &\approx (2K(1+2\log_2 K) + N - 1) \cdot T_{MAC} \end{aligned}$$

with $K = M + N - 1$.

For the OS version we can avoid adding the overlap, but we need an extra copy operation of the same length, and therefore:

$$\begin{aligned} T_i &= (\underbrace{2K\log_2 K}_{\text{FFT}} + \underbrace{2K}_{\text{multiplication}} + \underbrace{2K\log_2 K}_{\text{iFFT}}) \cdot T_{MAC} + (K + N - 1) \cdot T_{COP} \\ &= 2K(1+2\log_2 K) \cdot T_{MAC} + (K + N - 1) \cdot T_{COP} \\ &\approx 2K(1+2\log_2 K) \cdot T_{MAC} \end{aligned}$$

Again, if copy operations are well optimized in the processor, then they can often be neglected (as has been indicated by the $\approx$-signs).

4.9.3.2 Double buffer implementations

Consider Figure 4.18 until Figure 4.20. Using these diagrams we can derive the following properties:

Latency The input buffer needs to be filled before we can start, therefore the inherent latency for the operation itself amounts to

$$L = M \cdot T_s$$

However, we need to keep in mind that chosing for the double-buffer operation implies pipelining and this will increase the overall latency to

$$L = (1 + V)M \cdot T_s$$

with V the total number of operations.

[5] which makes sense as on modern platforms it takes as long to perform an addition as a multiplication followed by an addition.

Amount of memory The amount of memory can clearly be inspected on the diagrams:

$$R_i = \underbrace{M+N-1}_{\text{Filterkernel}} + \underbrace{M}_{\text{Input or output buffer A}} + \underbrace{M}_{\text{Input or output buffer B}} + \underbrace{N-1}_{\text{Overlap buffer A}} + \underbrace{N-1}_{\text{Overlap buffer B}} = 3M + 3N - 3$$

Computational complexity The computational complexity per frame can be derived from the diagrams. For the OA versions, this amounts to an FFT, a complex multiplication in the frequency domain for half of the coefficients (because of Hermiticity), an iFFT followed by adding the overlap. Counting the latter as full MACs[6], results in:

$$\begin{aligned} T_i &= (\underbrace{2K\log_2 K}_{\text{FFT}} + \underbrace{2K}_{\text{multiplication}} + \underbrace{2K\log_2 K}_{\text{iFFT}} + \underbrace{N-1}_{\text{Add overlap}}) \cdot T_{MAC} + (K-(N-1)) \cdot T_{COP} \\ &= (2K(1+2\log_2 K) + N - 1) \cdot T_{MAC} + (K-(N-1)) \cdot T_{COP} \\ &\approx (2K(1+2\log_2 K) + N - 1) \cdot T_{MAC} \end{aligned}$$

with $K = M + N - 1$. Note that the double-buffer setup avoids having to copy the overlap, compared to the single-buffer setup.

For the OS version we can avoid adding the overlap, but we need an extra copy operation of the same length, and therefore:

$$\begin{aligned} T_i &= (\underbrace{2K\log_2 K}_{\text{FFT}} + \underbrace{2K}_{\text{multiplication}} + \underbrace{2K\log_2 K}_{\text{iFFT}}) \cdot T_{MAC} + K \cdot T_{COP} \\ &= 2K(1+2\log_2 K) \cdot T_{MAC} + K \cdot T_{COP} \\ &\approx 2K(1+2\log_2 K) \cdot T_{MAC} \end{aligned}$$

Again, if copy operations are well optimized in the processor, then they can often be neglected (as has been indicated by the ≈-signs).

4.9.4 Comparison

Incremental vs. frame-based The first important choice, *incremental vs. frame-based* will be be a trade-off between the robustness w.r.t. incoming interrupts and higher priority tasks, and the cost of latency[7] that determines the size of the buffer. The drawbacks are only in terms of latency and memory required. The computational burden is the same per sample.

[6] which makes sense as on modern platforms it takes as long to perform an addition as a multiplication followed by an addition.

[7] Cost can refer to a financial cost, but also to a negative technical implication, e.g. increased latency can reduce the controllability of system that needs to be controlled.

Single vs. double buffer In the case of a frame-based setup, the second important choice is *a single vs a double buffer* implementation. Key here is the degree of parallelizability one wants to achieve. In a single-threaded implementation, double buffering makes no sense. However, in a parallel computing environment, it makes sense. Latency will be the blocking factor for adopting a full double-buffer implementation. Therefore a meet in the middle is a very common practice, subdividing the entire system in islands that from the outside behave as double-buffer blocks, but are implemented using a single-buffer flow on the inside. This strategy has been illustrated in Figure 3.6 on page 42.

Time domain vs. frequency domain Making a choice between *time-domain vs. frequency domain*, depends on the values of N and M. Let's analyze this in detail.

Overlap–add

Let's start by comparing the time-domain overlap–add method with the FFT-convolution overlap–add method. To this end, we define the *coefficient of overperformance*, i.e. the division of the computational complexity per sample of the (direct) time-domain overlap-add method by the computational complexity per sample of the (indirect/FFT) frequency-domain overlap-add method. In the equations, we neglect the copy operations.

$$R_{OA} = \frac{T_{i,t-dom,OA,sample}}{T_{i,f-dom,OA,sample}} = \frac{N}{\frac{2K(1+2\log_2 K)+N-1}{M}} = \frac{MN}{2K(1+2\log_2 K)+N-1}$$

This ratio indicates the number of times the overlap–add FFT-convolution is quicker than the time-domain convolution.

Figure 4.22 on the next page displays a contour graph of this ratio on a detailed and a coarse scale.

Now, let's compare the time-domain overlap–save method with the FFT-convolution overlap–save method. To this end, we again define the *coefficient of overperformance*, i.e. the division of the computational complexity per sample of the (direct) time-domain overlap-save method by the computational complexity per sample of the (indirect/FFT) frequency-domain overlap-save method. In the equations, we neglect the copy operations.

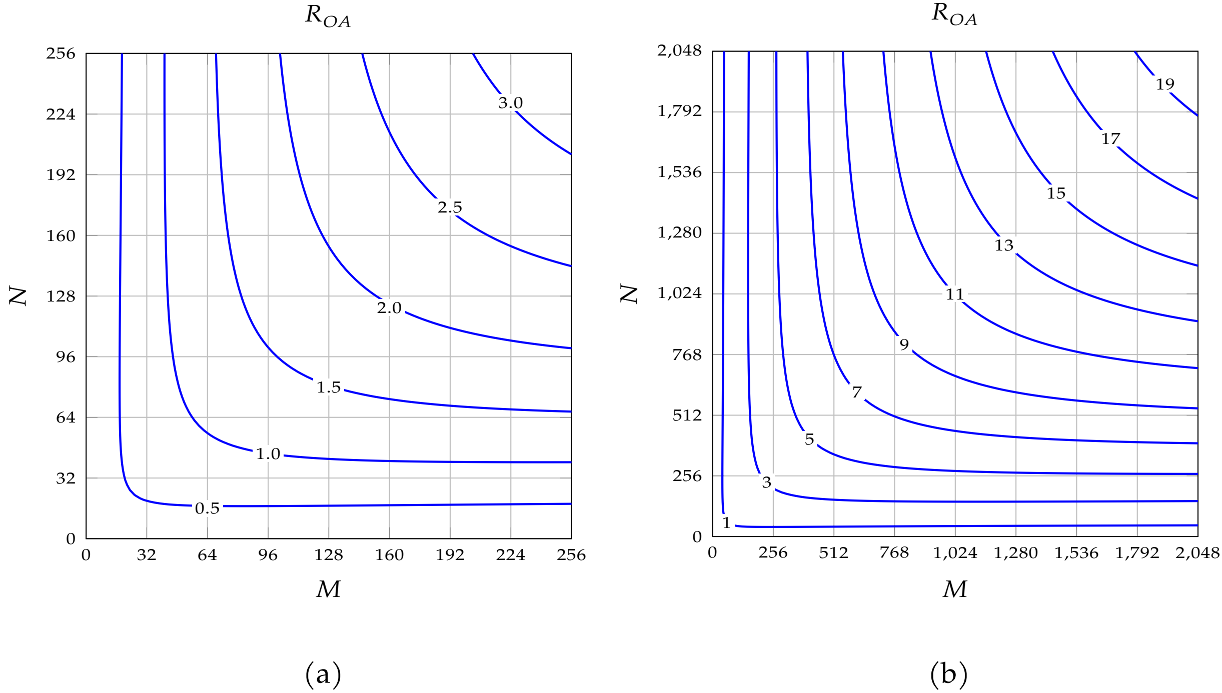


Figure 4.22: Coefficient of overperformance R_{OA} of the FFT-convolution over the direct convolution for the overlap–add method: (a) on a detailed scale, and (b) on a coarse scale

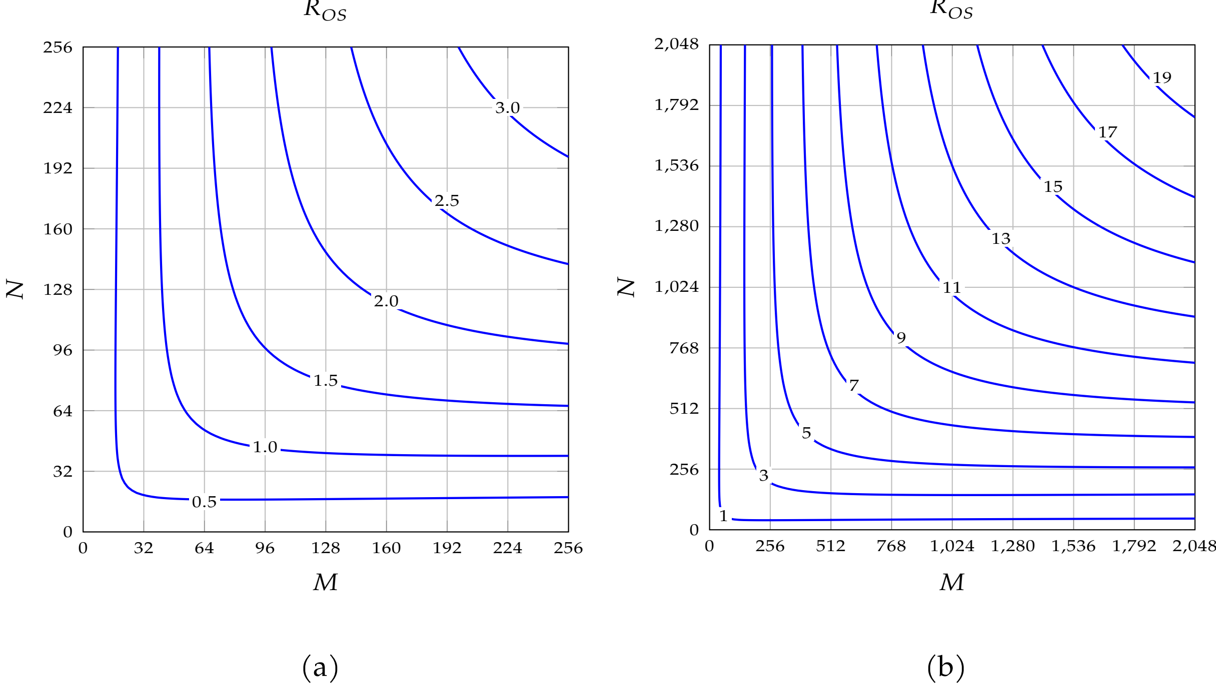


Figure 4.23: Coefficient of overperformance R_{OS} of the FFT-convolution over the direct convolution for the overlap–save method under the assumption $M > N$: (a) on a detailed scale, and (b) on a coarse scale

Overlap–save

The ratio we'd like to visualize is:

$$R_{OS} = \frac{T_{i,t-dom,OS,sample}}{T_{i,f-dom,OS,sample}} = \frac{N}{\frac{2K(1+2\log_2 K)}{M}} = \frac{MN}{2K(1+2\log_2 K)}$$

This ratio indicates the number of times the overlap–save FFT-convolution is quicker than the direct convolution. Figure 4.23 on the facing page displays a contour graph of this ratio on a detailed and a coarse scale.

The conclusions are obvious:

1. for large M and/or N the FFT-convolution is quite a bit faster than the direct convolution. Especially for large M and/or N the gain is huge (approximately a factor of 10 for $M \approx N \approx 1024!$);
2. for small M and N the direct convolution is still faster;
3. there is a memory penalty for using FFT-convolution, but in many applications it is quite tolerable;
4. the overlap–save FFT-convolution method slightly outperforms the overlap–add method when copy operations are cheaper than MACs. If not, the opposite may be true.

Exercises

Some exercises on algorithm and architectur selection

Exercise 4.9.4-1: Consider a system with a sampling clock $f_s = 100\,\text{kHz}$, and a sample size of 2 B.

For each of the cases below, determine the best convolution scheme:

Case	M	N	Buffer strategy
1	1	24	none
2	32	24	single
3	256	32	double
4	1024	512	mixed

Determine for every case:

1. the amount of memory required
2. the latency
3. the total amount of work per frame

4. the *CPLF*, assuming a single core and $T_{MAC} = 25$ ns and $T_{COP} = 10$ ns

If you can make a choice, choose the fastest implementation.

Exercise 4.9.4-2: Consider a studio-grade convolutional reverberation (audio) system, processing sampled stereo audio data in 24-bit format at a sample frequency of 192 kHz.

The 2-core signal processor has the following specifications:

Parameter	Value
T_{MAC}	10 ns
T_{COP}	5 ns

The total latency of the system needs to be less than 10 ms.

Answer the following questions:

1. Is a buffer length of 1024 samples feasible in this application? If so, use this buffer length. If not, calculate the maximal buffer length that is feasible, which is also a power of two.
2. Determine the best implementation scheme for the reverb convolution with a kernel length of 384.
3. Using that scheme, calculate the *CPLF* of a single stereo-channel convolution. Assume 10% of the work not to be parallelizable. Also calculate the *CPLF* for the full stereo system.
4. Calculate the amount of memory required for the full stereo system.

Finally, calculate the total reverb time that can be accomplished with this system, to convince you that it is not realistic at all.

Exercise 4.9.4-3: (*) Compose an architecture of the reverb system above, to allow for reverb times up until 10 ms. Calculate the amount of memory required an the loading fraction assuming the time required for a addition is equal to T_{MAC}.

4.10 Conclusions

In this chapter we've seen how to make the convolution operation tractable for a computer. As a consequence of this, the convolution operation is only useful for LTI FIR systems.

The optimal convolution technique depends on the frame size and the impulse response length. Direct convolution is to be preferred for small frame sizes and/or impulse response lengths. FFT-convolution pays of for larger frame sizes and impulse response lengths.

Though we've spent quite some time diving into C++ code, most DSP libraries contain some very well optimized algorithms to perform direct and FFT-convolution. Make use of these algorithms unless you are confident you can do better.

Chapter 5

Digital Filtering

In this chapter, you will learn about:

- the purpose of filtering,
- how to describe band-selection filter requirements.

After having read this chapter, some questions will still be left unanswered:

- how do I design digital filters?

After having read/studied this chapter, you are expected to be able to

- apply the concepts of band selection, noise removal, matched filtering and deconvolution,
- specify appropriate filter requirements for band-selection.

5.1 Introduction

Digital filtering is one of the operations one can perform in digital signal processing. In addition, it is our first acquaintance with digital signal processing itself. So far we've seen (in chapters 2 and 3.3.4) how we can build and implement causal LTI systems, now it's about time to see how we can design these systems.

Before we dive into the actual design of digital filters, we need to answer a few questions:

1. what are we filtering for?
2. how do we describe some basic filter requirements?

This (short) chapter is just about answering these questions.

5.2 What are we filtering for?

5.2.1 Noise removal

One of the problems with measurements (whether it is a measurement of a physical phenomenon or the measurement by an antenna of — let's say — a GSM signal), is that they are always corrupted by noise. One of the purposes of filtering is to *remove* this noise. It sounds very simple — remove the noise. However, in practice it is not that simple. The key of the problem is that we don't know which part of the signal is true 'signal' and which part is 'noise'. This is true for any external distortion happening to a signal: once it corrupts the signal, the harm is done. Therefore, the only thing we have available to work on is 'a noisy signal'.

However, in the case of noise, we're lucky because we do know something about noise phenomena: they are happening according to some noise statistics laws we hope to be substantially different from our signal statistics. That is, we assume that the noise component is a random signal governed by some (higher-order) noise statistic, while the signal is determined by the application.

We all know an obvious way to remove noise from a measurement: perform the measurement multiple times and average. Because of the fact the noise is random, it will *cancel* in the averaging process. The measured physical quantity hopefully will not have changed too much in order to keep an average value that makes sense for the time point we are observing.

Let's make a filter that does just that: the *moving-average filter*. The nice thing about this filter, is that it is a finite impulse response filter, but we can implement it in two ways, i.e.

- non-recursively, and
- recursively.

5.2.1.1 Moving-average filter — non-recursive implementation

The moving-average filter of length N is simply defined as:

$$\begin{aligned} y[n] &= \frac{1}{N} \sum_{i=0}^{N-1} x[n-i] \\ &= \frac{1}{N} \left(x[n] + x[n-1] + \cdots + x[n-N+1] \right) \end{aligned}$$

From this description, we can easily derive that the moving-average filter is a FIR filter. It exhibits a few very nice features:

1. it has a symmetric filter kernel[1], therefore it will exhibit a linear frequency response.
2. it uses many delay elements and adders, but only one multiplier!

If we'd use one of the basic forms to implement this filter, we'd need $N-1$ delay elements, a multiplier and and a one N-input adder.

[1] The term kernel is in this case a synonym for impulse response.

In fact, the impulse response of this filter corresponds to a rectangular window function. We know that the DtFT of a rectangular window function is a sync function, and hence exhibits a low pass character. It is this low-pass filtering function that removes the noise.

By replacing the rectangular window kernel by more advanced windows, we can change the performance of the filter.

Exercises
Some exercises on moving average filters

Exercise 5.2.1.1-1: Calculate the transfer function of this filter (in the Z-domain).

Exercise 5.2.1.1-2: Calculate the DtFT of the impulse response of this filter.

Exercise 5.2.1.1-3: (a) Draw a direct form implementation of this filter;
(b) Draw a transposed form implementation of this filter.

Exercise 5.2.1.1-4: Write a C++ function that implements this filter.

5.2.1.2 Moving-average filter — recursive implementation

A peculiar thing about this filter is that we can also implement it as a recursive filter. The recursion relation is easily derived by taking a look at $y[n]$ and $y[n-1]$:

$$\begin{aligned} y[n] &= \frac{1}{N}\left(x[n] + x[n-1] + x[n-2] + \cdots + x[n-N+2] + x[n-N+1] \right) \\ y[n-1] &= \frac{1}{N}\left(x[n-1] + x[n-2] + \cdots + x[n-N+2] + x[n-N+1] + x[n-N]\right) \end{aligned}$$

In this way, it is easy to see that:

$$\begin{aligned} y[n] - y[n-1] &= \frac{1}{N}\left(x[n] - x[n-N]\right) \\ y[n] &= y[n-1] + \frac{1}{N}\left(x[n] - x[n-N]\right) \end{aligned}$$

This implementation saves us a considerable amount of hardware (or processor cycles): we only need two adders and one multiplier. In exchange we need two delay elements extra. However, as the result is derived from a previous output, rounding or truncation errors due to limited floating- or fixed-point precision, may accumulate. In rare cases, this might lead to problems.

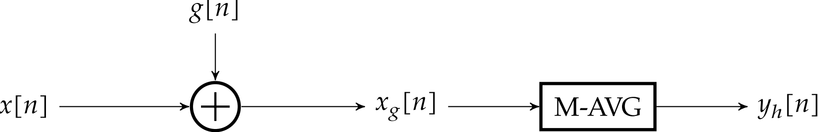
Exercises
Some exercises on moving average filters (recursive implementation)

Exercise 5.2.1.2-1: Calculate the transfer function of this filter (in the Z-domain) and prove that this transform is equal to the expression you derived for the non-recursive implementation.

Exercise 5.2.1.2-2: (a) Draw a direct form I and II implementation of this filter for $N = 5$;
(b) Draw a transposed form I and II implementation of this filter for $N = 5$.

5.2.1.3 Filtering effect

Let's consider the filtering effect by considering a small test setup. Consider an arbitrary sinusoid signal $x[n]$ that is affected by additive Gaussian noise $g[n]$ to become $x_g[n]$. Our goal is to try to get rid of the noise by employing a moving-average filter of length $N = 16$, resulting in a signal $y_h[n]$. This setup has been illustrated graphically below:



The filtering effect has been illustrated in Figure 5.1 on the next page. To reduce spectral leak when applying the FFT, we applied a Blackman window function (of length 256 before applying the DFT/FFT. Note how the moving-average filter filters away the high frequencies, but also affects the signal itself: it clearly reduces its peak-to-peak value. Also note how the filter introduces a delay of $N/2$. This holds for any linear-phase FIR filter: it introduces a delay of half the number of taps.

Exercises
Some exercises on the frequency response of moving average filters

Exercise 5.2.1.3-1: Use OCTAVE/MATLAB to calculate the frequency response of an 5-point moving-average filter using the built-in FFT and window functions.

Exercise 5.2.1.3-2: Write some OCTAVE/MATLAB code to check how the filter behaves for various values of N.

5.2.2 Band selection

Many signals are limited to a specific range of frequencies (or wavelengths in case we're considering electromagnetic or light waves):

- visible light
- human audible spectrum
- an FM channel on your radio
- an analog PAL cable television channel
- WiMAX wireless 'last mile' connections
- ...

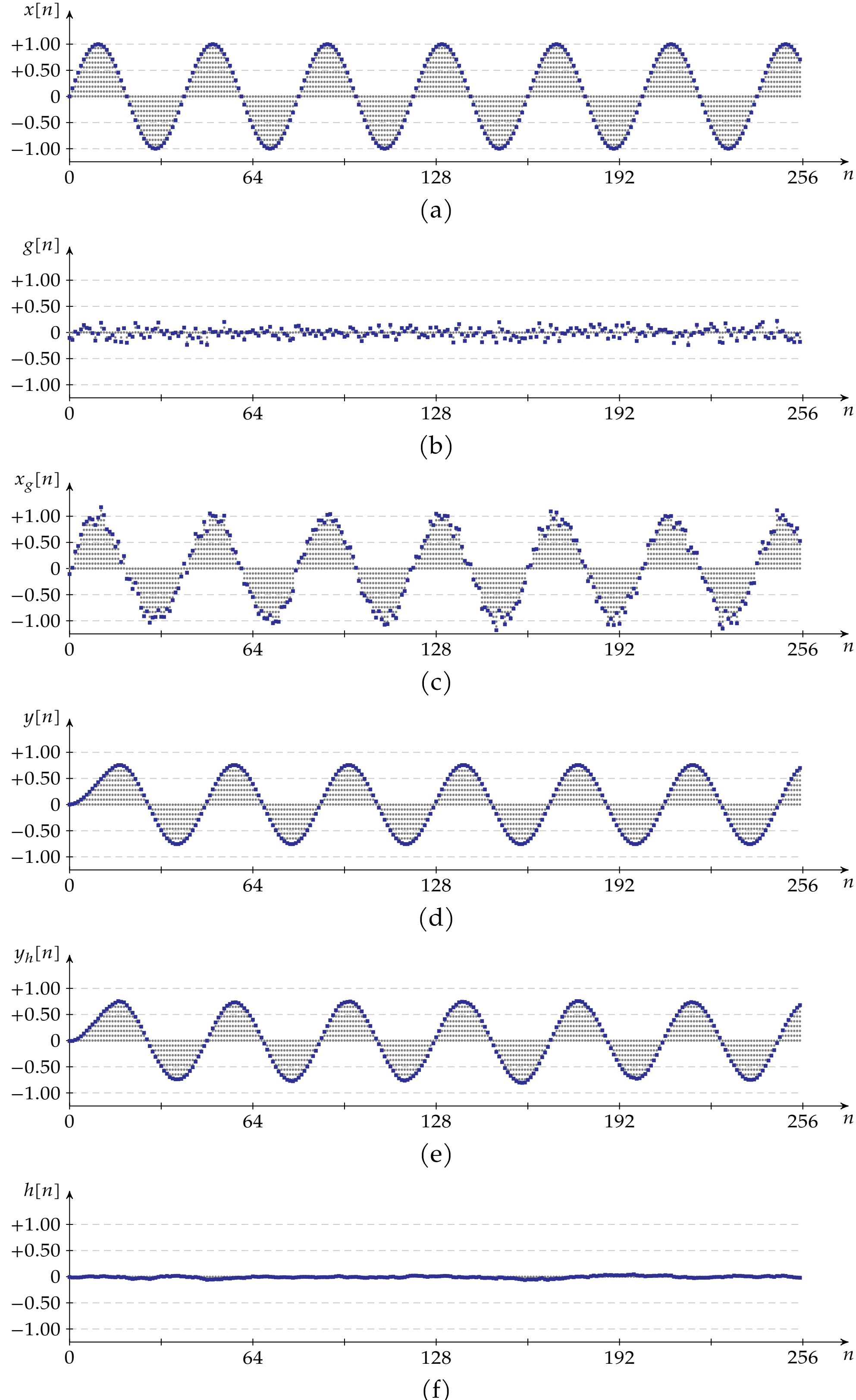


Figure 5.1: Illustration of the low-pass filtering effect of a $N = 16$ moving-average filter, part I — time domain: (a) noiseless signal $x[n]$, (b) Gaussian noise $g[n]$, with $\sigma = 0.1$, (c) noisy signal $x_g[n]$, (d) filtered signal $y[n]$, (e) filtered noisy signal $y_h[n]$, (f) remaining noise after filtering $h[n]$

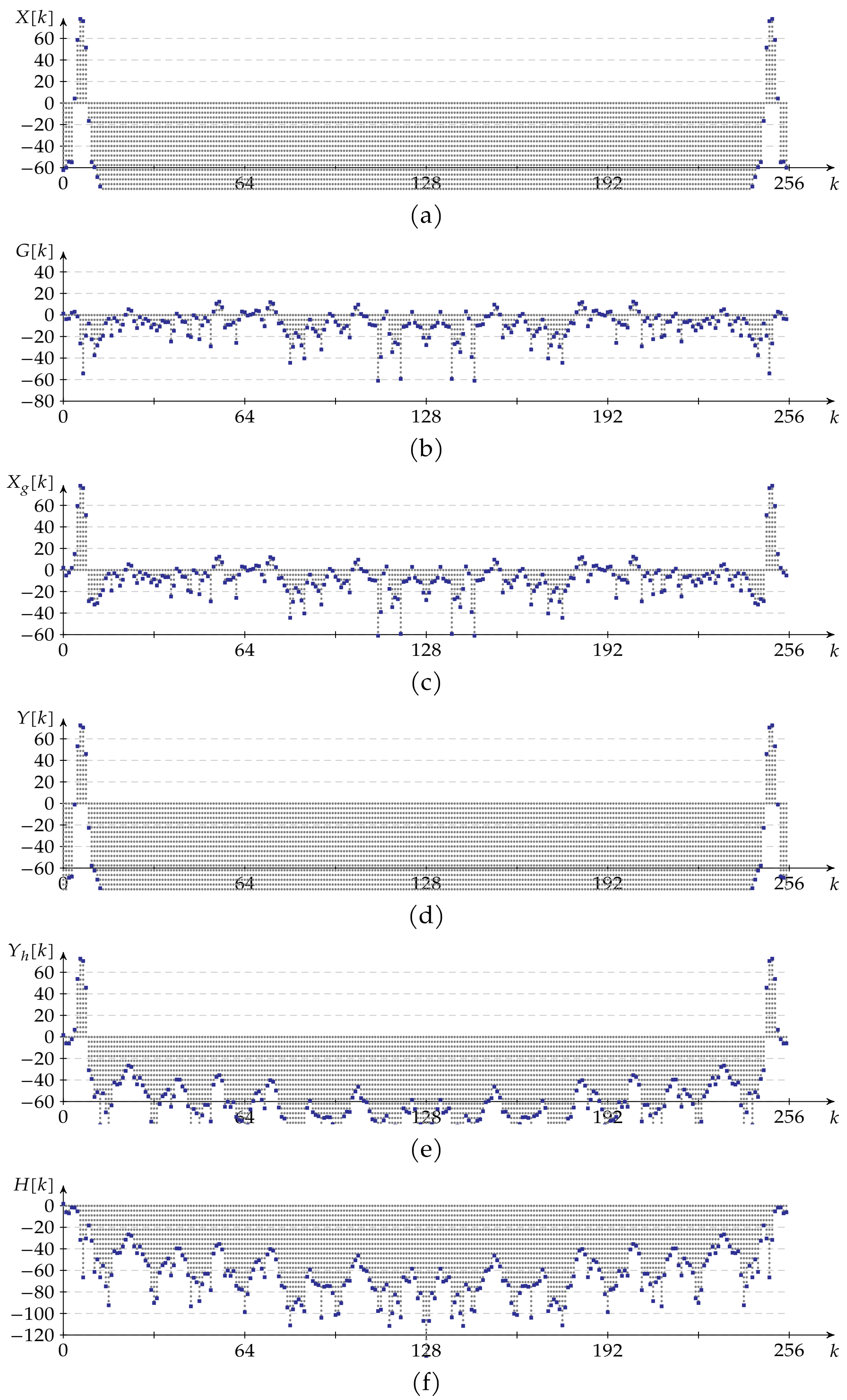


Figure 5.1: (cont'd) Illustration of the low-pass filtering effect of the moving-average filter, part II — frequency domain: (a) noiseless signal $X[k]$, (b) gaussian noise $G[k]$, with $\sigma = 0.1$, (c) noisy signal $X_g[k]$, (d) filtered signal $Y[k]$, (e) filtered noisy signal $Y_h[k]$, (f) remaining noise after filtering $H[k]$

In all of these applications you will need a filter to select the frequency range of interest out of the range your detector selects from the real world.

In addition, any DSP system that converts continuous-time signals to a discrete-time representation needs a sampling and a reconstruction filter. Remember, these have to be analog filters. However, for multi-rate conversion (upsampling and downsampling), we also needed digital decimation and interpolation filters. These are also band-selection (or band-rejection) filters.

5.2.3 Deconvolution

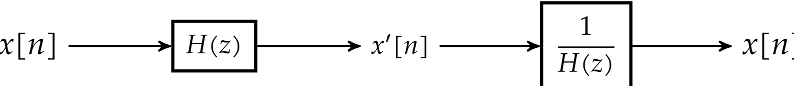
Signals cannot only suffer from noise, but also from a systematic deterioration by a transmission channel (we cannot avoid). A few examples may help to illustrate what we mean:

- Consider for example a picture taken with a digital camera. If the photographer moves his camera while taking the picture, the photo taken will be unsharp.
- Assume recording your favorite opera diva's voice in a poor concert theatre that resonates at frequencies of about 1 100 Hz. The recording of the aria of the Queen of the Night in Mozart's "Die Zauberflöte" (The Magic Flute) will increase in volume whenever she reaches out for her high notes.

We might ask ourselves the question: can't we undo the deterioration using an LTI "restoration"-filter? We can — if the deterioration was linear and time-invariant.

These type of filters are so-called *deconvolution filters* as they undo the effect of an unwanted *convolution*.

The principle has been outlined in the figure below. The signal $x[n]$ is deteriorated by convolution with $h[n] \xrightarrow{Z} H(z)$ to $x'[n]$. The deconvolution filter $1/H(z)$ restores it to $x[n]$.



Of course, here it is assumed that

- we know the characteristics of the convolution that corrupted the signal,
- the deconvolution filter is stable.

The two conditions above may prove to be problematic depending on the application at hand. Let's go into a little bit more detail.

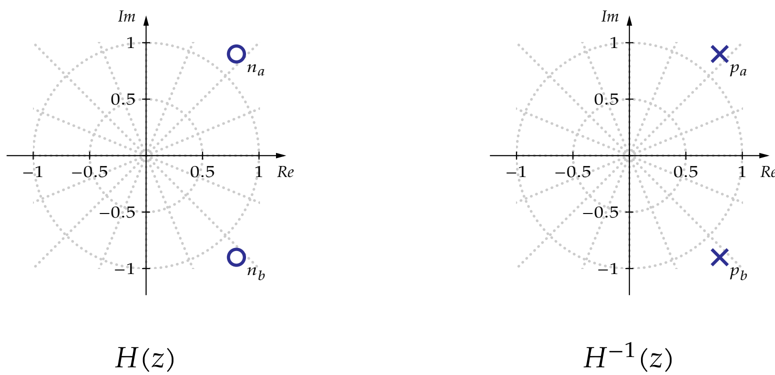
Blind deconvolution In case of the camera example, we probably did not know in what direction the camera was shaking when the picture was taken.[2] This is a case in which we don't know enough about the original deterioration to come up with a good deconvolution

[2] Even assuming that the camera was shaken along a straight line is an assumption too many. The photographer might have performed a circular shake!

filter. However, we're not totally lost in this case. Actually, there's something we know about sharp pictures: they usually contain a lot of sharp edges (i.e. sharp transitions from light to dark and the other way around). If we can put a quantitative measure on the "sharpness" of a picture, we might 'blindly' try some very probable deconvolution schemes (e.g., linear deconvolution for a radial motion every 5 degrees) and see which one gives the sharpest result. This is so-called *blind deconvolution*.

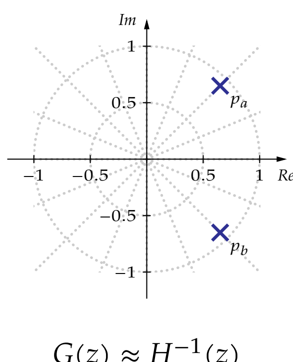
Blind convolution does not always have to be a dumb trial and error process. In many cases some hints can be found in the end-result (or in the history of the application at hand) as to which deconvolution filter kernel might be more succesful.

Limited deconvolution Assume we are faced with a system $H(z)$ that exhibits two zeros outside the unit circle. The deconvolution filter required to compensate for those two zeros is a system $H^{-1}(z)$ with two poles outside the unit circle, and hence unstable. This has been illustrated in the figure below.



$H(z)$ $H^{-1}(z)$

The solution may be to accept the fact that we'd better not make unstable filters and apply a filter that undoes the original deterioration as much as possible. In the example presented above, this might be done by putting the poles just inside the unit circle, attempting to make a $G(z) \approx H^{-1}(z)$. Don't be mistaken by this simple solution. A real world case may be way more complex.



$G(z) \approx H^{-1}(z)$

5.2.4 Correlation detection - Matched filtering

Correlation...

The correspondence between two real signals $x_1[n]$ and $x_2[n]$ can be mathematically expressed as their *correlation*, a metric that is defined as:

$$r_{x_1,x_2}[k] = \sum_{n=-\infty}^{+\infty} x_1[n+k]x_2[n] \tag{5.6}$$

The parameter k is a translation parameter that shifts $x_1[n]$ w.r.t $x_2[n]$ before multiplying the corresponding time points.

The correlation operation is important enough to devote a dedicated symbol to it:

$$r_{x_1,x_2}[k] = x_1[n] \star x_2[n]$$

If one of the signals is time-limited, the summation in the definition loses its infinite boundaries. E.g., if $x_2[n]$ is only nonzero for $n_{lo} \leq n \leq n_{hi}$, then (5.2.4) reduces to:

$$r_{x_1,x_2}[k] = \sum_{n=n_{lo}}^{n_{hi}} x_1[n+k]x_2[n]$$

Very often, in signal processing, *correlation* is also denoted as *cross-correlation*. This is opposed to *auto-correlation* that compares a signal $x[n]$with a translated copy of itself:

$$r_{x,x}[k] = x[n] \star x[n] = \sum_{n=-\infty}^{+\infty} x[n+k]x[n]$$

The auto-correlation peaks to the energy content of the signal for $k = 0$.

It is this very property that we can use for the better in the perspective of digital filtering. Suppose we'd like to recognize a known subsignal in a larger signal. As humans we can "scan" through the graph of the larger signal and we will be easily able to spot the location of the subsignal. Mathematically, we have performed a correlation operation (at which we, humans, are very good). This is also the way we can have a machine spot a subsignal in a larger signal: have it calculate the correlation values for different k, and see for which k it peaks (close) to the subsignal's energy content.

...and convolution

As you will have noticed, the correlation definition (5.2.4) above is rather similar to the definition of discrete convolution as stated in (4.1). Indeed, the relationship is rather obvious:

$$\begin{aligned} r_{x_1,x_2}[k] &= x_1[n] \star x_2[n] \\ &= x_1[-n] \bigstar x_2[n] \end{aligned} \tag{5.7}$$

The similarity between the correlation and convolution operations expressed in (5.7) is the key to implementing the correlation operation using a digital filter: just take the signal you want to spot in a larger signal, reverse it in time, shift it to make it causal and use it as the impulse response of your digital filter. The output of the filter will be the correlation of both signals.

The procedure is illustrated in Figure 5.2 on the following page assuming you want to recognize a time-limited subsignal $v[n]$ in a signal $x[n]$.

This technique of *correlation detection* is called *matched filtering*, because the impulse response matches the signal to be discovered.

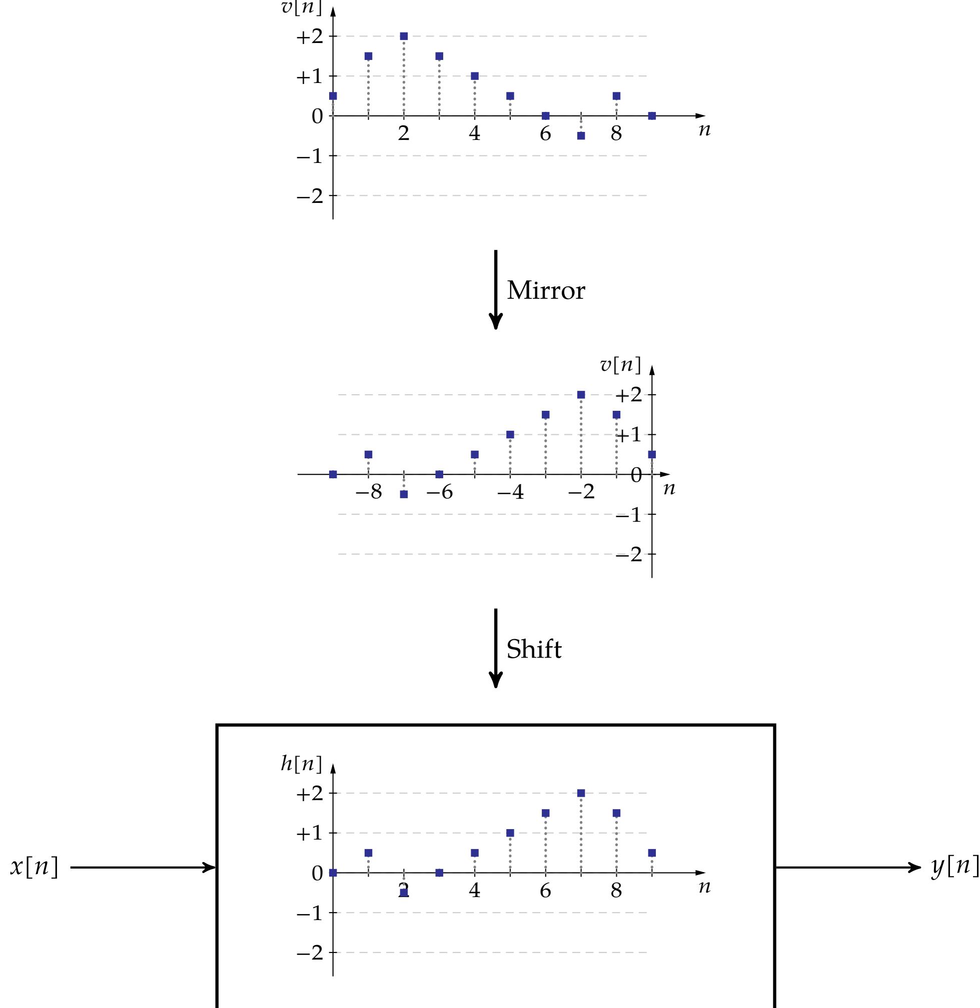


Figure 5.2: Illustration of the principle of matched filtering, deriving the desired impulse response $h[n]$ from the signal to match $v[n]$

5.3 Filter requirements description for band-selection filters

As *band selection* is one of the primary purposes of digital filtering, some extra classification and basic terminology for this type of applications makes sense.

5.3.1 Basic filter types

Depending on which part of the spectrum we remove and/or keep, several basic filter types can be recognized:

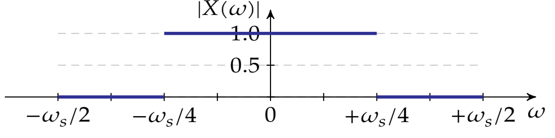
- low-pass filters
- high-pass filters
- band-pass filters
- band-stop filters
- all-pass filters

Low-pass filter

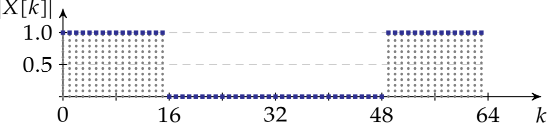
An ideal low-pass filter has a discrete-time frequency response

$$|X(\omega)| = \begin{cases} 1 & |\omega| \leq \omega_0 \\ 0 & |\omega| > \omega_0 \end{cases}$$

This has been illustrated on the right for $\omega_0 = \frac{\omega_s}{4}$.



Note that a standard N-point DFT/FFT will show a different picture — at first sight — due to the fact that the result is not centered around zero. This has been illustrated for $N = 64$ and $|k_0| = 15$ on the right.

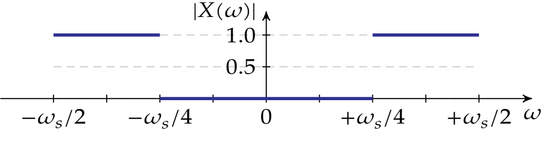


High-pass filter

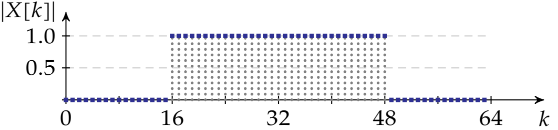
An ideal high-pass filter has a discrete-time frequency response

$$|X(\omega)| = \begin{cases} 0 & |\omega| \leq \omega_0 \\ 1 & |\omega| > \omega_0 \end{cases}$$

This has been illustrated on the right for $\omega_0 = \omega_s/4$.



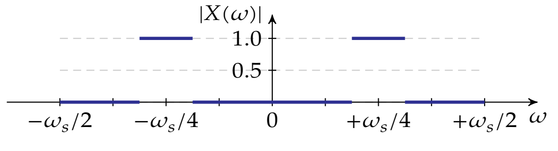
Note that a standard N-point DFT/FFT (for the reasons mentioned above) will show a different picture. This has been illustrated for $N = 64$ and $k_0 = 15$ on the right.



Band-pass filter

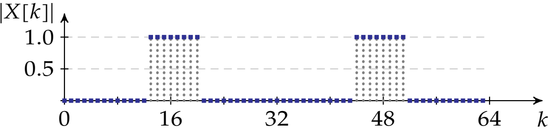
An ideal band-pass filter has a discrete-time frequency response

$$|X(\omega)| = \begin{cases} 0 & |\omega| < \omega_1 \\ 1 & \omega_1 \leq |\omega| \leq \omega_2 \\ 0 & |\omega| > \omega_2 \end{cases}$$



This has been illustrated on the right for $\omega_1 = \frac{3}{16}\omega_s$ and $\omega_2 = \frac{5}{16}\omega_s$.

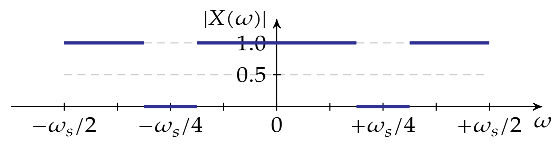
Note that a standard N-point DFT/FFT (for the reasons mentioned above) will show a different picture. This has been illustrated for $N = 64$ and $k_1 = 8$ and $k_2 = 20$ on the right.



Band-stop filter (notch filter)

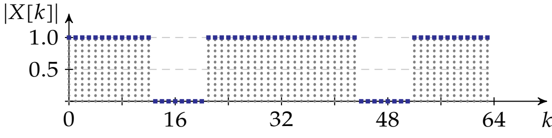
An ideal band-stop filter has a discrete-time frequency response

$$|X(\omega)| = \begin{cases} 1 & |\omega| < \omega_1 \\ 0 & \omega_1 \leq |\omega| \leq \omega_2 \\ 1 & |\omega| > \omega_2 \end{cases}$$



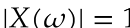
This has been illustrated on the right for $\omega_1 = \frac{3}{16}\omega_s$ and $\omega_2 = \frac{5}{16}\omega_s$.

Note that a standard N-point DFT/FFT will show a different picture (for the reasons mentioned above). This has been illustrated for $N = 64$ and $k_1 = 8$ and $k_2 = 20$ on the right.

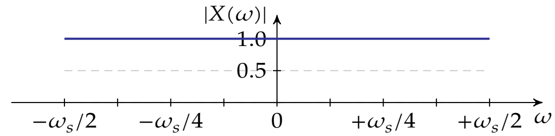


All-pass filter (phase shaping filter)

An ideal all-pass filter has a discrete-time frequency response

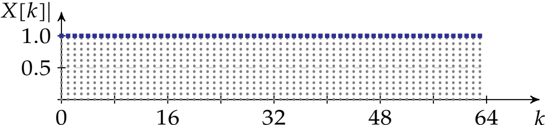


$$|X(\omega)| = 1$$



This has been illustrated on the right. The purpose of an all-pass filter is to perform phase correction without altering the magnitude of the signals.

The standard N-point DFT/FFT will show a very similar picture. This has been illustrated for $N = 64$ on the right.



5.3.2 Basic terminology

Filter requirements are defined using a specific terminology and a standard set of parameters:

- passband
- passband ripple R_p
- passband corner (or edge) frequency ω_P
- transition band
- stopband corner (or edge) frequency ω_S
- stopband
- stopband attenuation R_s

The *passband* is a contiguous frequency region where the magnitude of the filter transfer function spectrum is in between $-R_p$dB and 0 dB. Hence, in the passband, the spectrum exhibits a maximum *passband ripple* of R_pdB. The extreme frequencies of the passband are the *passband corner (or edge) frequencies*.

The *stopband* is a contiguous frequency region where the magnitude of the filter transfer function spectrum is below $-R_S$dB. Hence, the *stopband attenuation* is R_SdB. The extreme frequencies of this region are the *stopband corner (or edge) frequencies*.

The *transition band* is a contiguous frequency region where the magnitude of the filter transfer function spectrum is in between $-R_P$dB and $-R_S$dB. The transition band separates a passband from the adjacent stopband.

Let's see how these parameters appear on the frequency spectrum magnitudes of the basic filter types.

5.3.2.1 Low-pass filter

The frequency spectrum magnitude of the transfer function of a non-ideal low-pass filter is depicted in Figure 5.3 on page 110. The parameters mentioned above have been indicated on the figure.

5.3.2.2 High-pass filter

The frequency spectrum magnitude of the transfer function of a non-ideal high-pass filter has been depicted in Figure 5.4 on page 110. The parameters mentioned above have been indicated on the figure.

5.3.2.3 Band-pass filter

The frequency spectrum magnitude of the transfer function of a non-ideal band-pass filter has been depicted in Figure 5.5 on page 110. The parameters mentioned above have been indicated on the figure. We need to distinguish a lower and an upper transition band. Hence the subscripts L and U.

5.3.2.4 Band-stop filter

The frequency spectrum magnitude of the transfer function of a non-ideal band-stop filter has been depicted in Figure 5.6 on page 111. The parameters mentioned above have been indicated on the figure. We need to distinguish a lower and an upper transition band. Hence the subscripts L and U.

5.4 Plotting frequency responses OCTAVE/MATLAB

In the next chapters we will often need to draw a frequency response of a digital filter, given its transfer function:

$$H(z) = \frac{\sum_{m=0}^{M-1} b_m z^m}{\sum_{n=0}^{N-1} a_n z^n}$$

The frequency response that corresponds to this transfer function can be easily found in OCTAVE as:

```
[H, W] = freqz( B, A, N, "whole" );
```

in which B and A are vectors containing the numerator and denominator coefficients, N is the number of frequency points to use and `"whole"` forces the function to sample the interval 0 to 2π instead of the interval 0 to π. This function returns the frequency vector W and the transfer function values H.

The function `freqz` has alternative calling schemes with different functionality. Check them out using

```
help freqz
```

Plotting the results obtained with `freqz` can be easily done using `freqz_plot`:

```
freqz_plot( W, H )
```

However, this function doesn't allow easy customization, and therefore you will soon find yourself defining your own `freqz_plot` function using the custom `plot` function. Check the help pages of `plot` for more information.

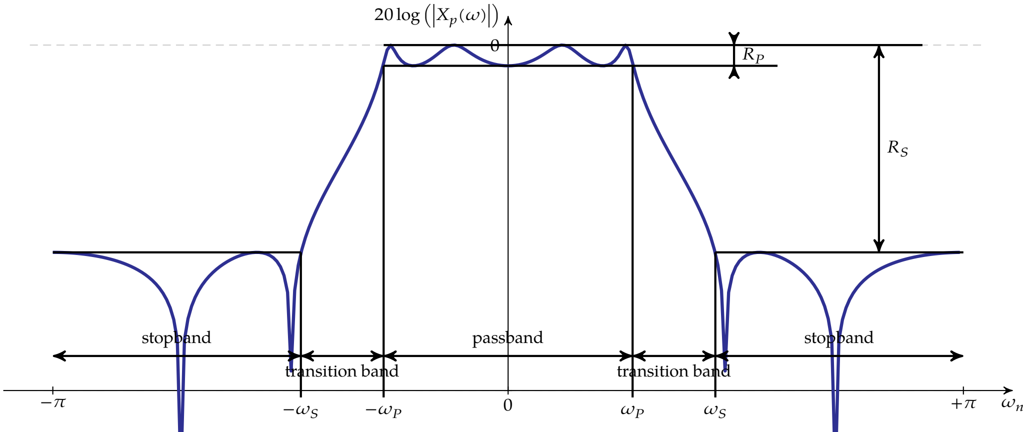


Figure 5.3: Frequency spectrum magnitude of the transfer function of a non-ideal low-pass filter

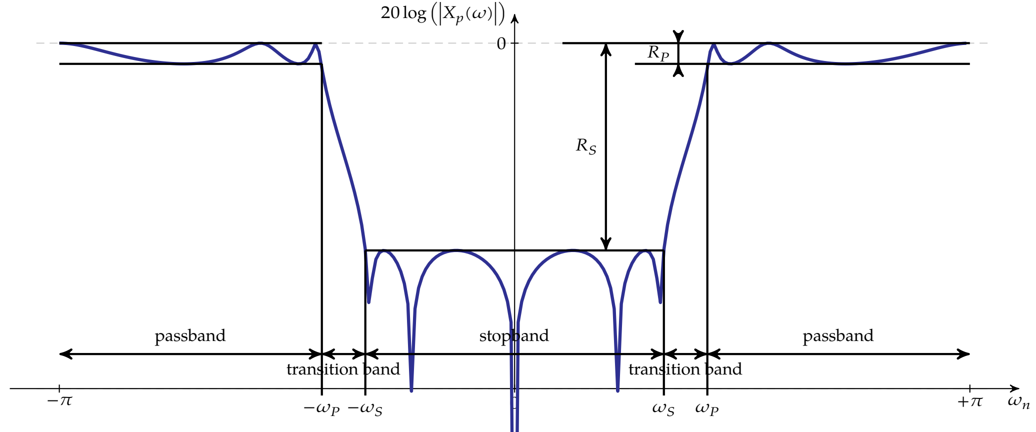


Figure 5.4: Frequency spectrum magnitude of the transfer function of a non-ideal high-pass filter

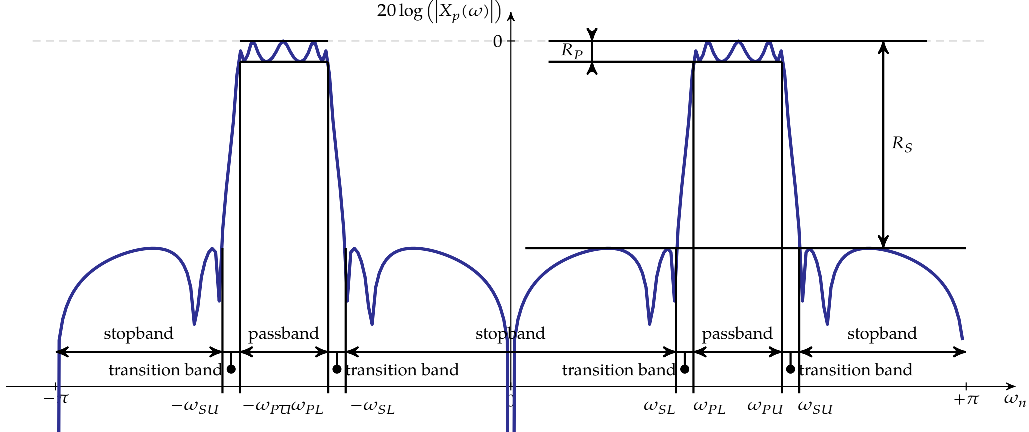


Figure 5.5: Frequency spectrum magnitude of the transfer function of a non-ideal band-pass filter

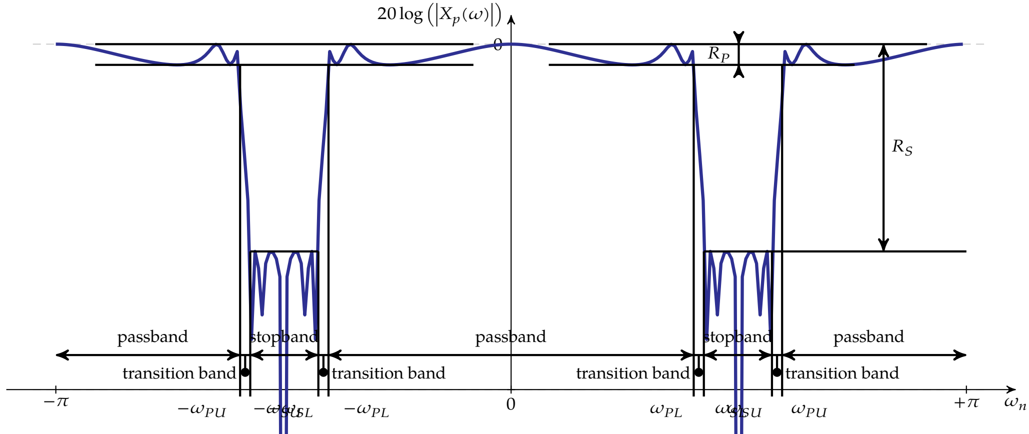


Figure 5.6: Frequency spectrum magnitude of the transfer function of a non-ideal band-stop filter

Chapter 6

FIR Filter Design

In this chapter, you will learn about:

- various ways to design discrete-time FIR filters,
- how to perform the actual design using Octave or Matlab,
- the advantages and drawbacks of these design methods.

After having read this chapter, some questions will still be left unanswered:

- what about IIR filters?

After having read/studied this chapter, you are expected to be able to

- explain the various design methods,
- judge the weaknesses and the strengths of every design method,
- apply the various design methods, using pencil and paper and using a mathematical computer program like Octave/Matlab, when given a filter specification.

6.1 Introduction

There are many good methods to design discrete-time FIR filters.

In this chapter we will discuss some of these methods:

- the zero placement method
- the impulse response invariance method
- the frequency sampling design method
- optimal filter design

Most of these methods have been implemented in mathematical toolboxes like the signal processing toolbox of Matlab or Octave. If all the magic is readily available, why bother studying the mathematical principles behind these techniques? The answer is obvious: using tools without any knowledge about their nature or the circumstances in which to use them is most dangerous. A small trivial example probably makes this clear: a car is a useful *transportation mean*. But putting the steering wheel in the hands of a drunk lunatic turns it into a *deadly weapon*. In, addition poor knowledge about tools, may lead to suboptimal use. An

example along the same lines to illustrate this: using a car to collect your snail-mail from your private mailbox (unless you live in the outback) is probably suboptimal in many ways: bad for your wallet, bad for the environment, bad for your health, etc.

So, our goal is twofold:

- get to know the basic principles of the techniques,
- get a flavor of how to use these techniques in MATLAB/OCTAVE.

Of course we cannot treat all the available methods. You should be aware that there are a number of filter types and (corresponding) design methods that we will not treat:

- frequency sampling filters
- interpolated low-pass FIR filters
- constrained optimal FIR-filter design
- ...

6.2 Zero placement

A very simple way to design FIR filters is to place zeros in the (complex) Z-plane. To create filter functions that correspond to a real impulse response, zeros need to be *real* or appear in *complex conjugate pairs*.

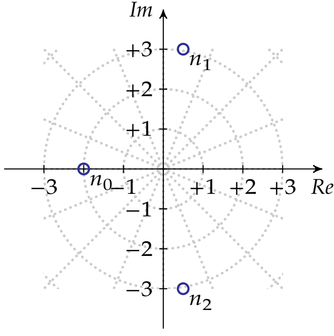
However, every zero we place needs to be accompanied by a pole at $z = 0$ to ensure causality of the filter. This can be easily understood by realizing that the transfer function of a causal filter can only contain terms with a factor of z^{-k}.

6.2.1 Arbitrary placement

Consider the following simple example in which we arbitrarily place three zeros:

$$n_0 = -2$$
$$n_{1,2} = 0.5 \pm j3$$

This zero configuration has been depicted below:



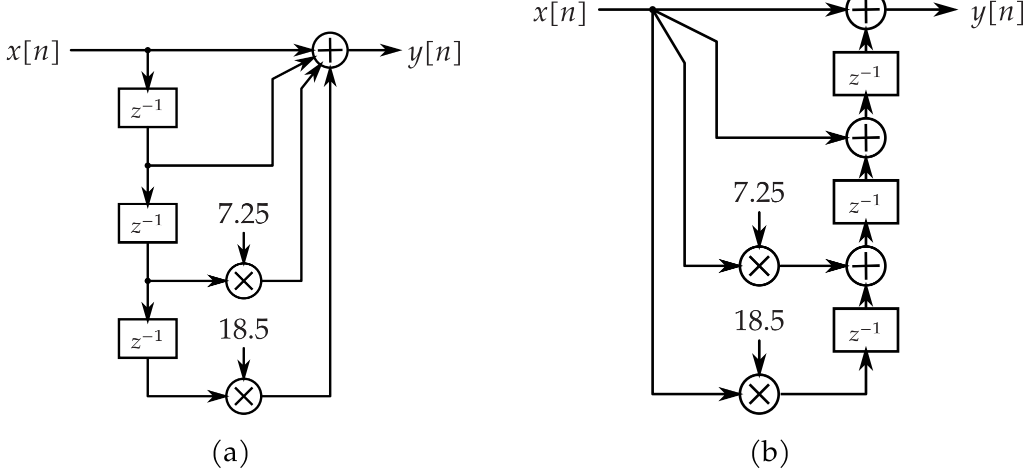


Figure 6.1: Implementation of (6.2): (a) direct form and (b) transposed form

Deriving the corresponding transfer function in the Z-plane is easy:

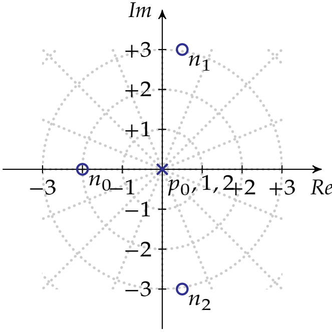
$$\begin{aligned} H_{NC}(z) &= (z-(-2))\,(z-(0.5+\mathrm{j}3))\,(z-(0.5-\mathrm{j}3)) \\ &= (z+2)\left(z^2-z+9.25\right) \\ &= z^3+z^2+7.25z+18.5 \end{aligned} \tag{6.1}$$

Alas, this function does not correspond to a causal filter, as it contains terms in positive powers of z. Dividing (6.1) by a pole at $z=0$ with multiplicity 3 does the job. In this way, the transfer function becomes

$$\begin{aligned} H_C(z) &= \frac{H_{NC}}{z^3} \\ &= 1+z^{-1}+7.25z^{-2}+18.5z^{-3} \end{aligned} \tag{6.2}$$

which is causal.

The corresponding pole-zero diagram has been depicted below:



This leads directly to the direct and transposed form implementations of Figure 6.1.

Exercises
Some exercises on the zero placement method:

Exercise 6.2.1-1: Make a filter that realizes the following zeros:

$$n_1 = -1.5$$
$$n_2 = 2.1$$

Use OCTAVE to make a graph displaying the transfer function's spectrum (Bode plot).

Exercise 6.2.1-2: Make a filter that realizes the following zeros:

$$n_{1,2} = -3 \pm j2$$

Use OCTAVE to make a graph displaying the transfer function's spectrum (Bode plot).

Exercise 6.2.1-3: Make a filter that realizes the following zeros:

$$n_1 = 1.5$$
$$n_{2,3} = -3 \pm j2$$

Use OCTAVE to make a graph displaying the transfer function's spectrum (Bode plot).

Exercise 6.2.1-4: Make a filter that realizes the following zeros:

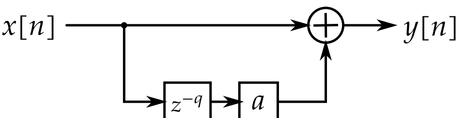
$$n_0 = 0.5$$
$$n_1 = -2$$
$$n_{2,3} = 0.25 \pm j3$$

Use OCTAVE to make a graph displaying the transfer function's spectrum (Bode plot).

6.2.2 Feedforward comb filters

Definition

If we place the zeros according to a particular pattern, we obtain a special kind of filter called a *feedforward comb filter*.[1],[2] In this type of filter a scaled and delayed version of the input signal is added to the original input signal.



This leads to the following system description:

$$y[n] = x[n] + ax[n-q]$$

This system description can be easily transformed to the Z-domain:

$$\begin{aligned} Y(z) &= X(z) + az^{-q}X(z) \\ &= \left(1 + az^{-q}\right) X(z) \end{aligned}$$

This yields the following transfer function:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{z^q + a}{z^q}$$

[1]These comb filters are totally unrelated to the 2-line, 3-line, or 3D comb filters advertised as features for TV sets.

[2]A related type of filter is the *feedback comb filter* (see section 7.2.3 on page 167)

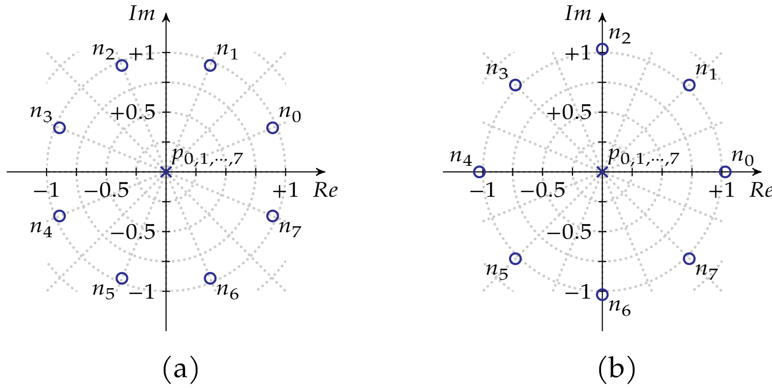


Figure 6.2: Pole zero diagram of a feedforward comb filter, for the case $q = 8$, with (a) $a = 0.75$ and (b) $a = -1.25$

Location of poles and zeros

Obviously, this transfer function has a pole with multiplicity q at $z = 0$:

$$p_{0,1,2,\ldots,q-1} = 0$$

The zeros can be easily derived by setting the numerator to zero

$$\begin{aligned} z^q + a &= 0 \\ z^q &= -a \end{aligned} \tag{6.3}$$

First, let's assume a is positive. Rewriting (6.3) with $-a$ in polar notation yields:

$$\begin{aligned} z^q &= a\, e^{j(\pi + k2\pi)} \\ z &= \sqrt[q]{a}\, e^{j\left(\frac{\pi}{q} + \frac{2k\pi}{q}\right)} \end{aligned}$$

with k integer.
Now, let's assume a is negative and substitute $b = -a$. Rewriting (6.3) with $-a$ in polar notation yields:

$$\begin{aligned} z^q &= b\, e^{jk2\pi} \\ z &= \sqrt[q]{b}\, e^{j\frac{2k\pi}{q}} \end{aligned}$$

with k integer.

As an example, the pole-zero diagrams for $q = 8$ in case $a = 0.75$ and in case $a = -1.25$ have been displayed in Figure 6.2.

The frequency spectra of these filters have been illustrated in Figure 6.3 on the next page. They also show why these filters are called *comb* filters. Notice how the number of zeros corresponds to the number of dips in the spectrum.

Comb ratio

An important parameter in designing a comb filter is the ratio between peaks and dips, the so-called *comb ratio*. This ratio can be determined by analyzing the magnitude response of the

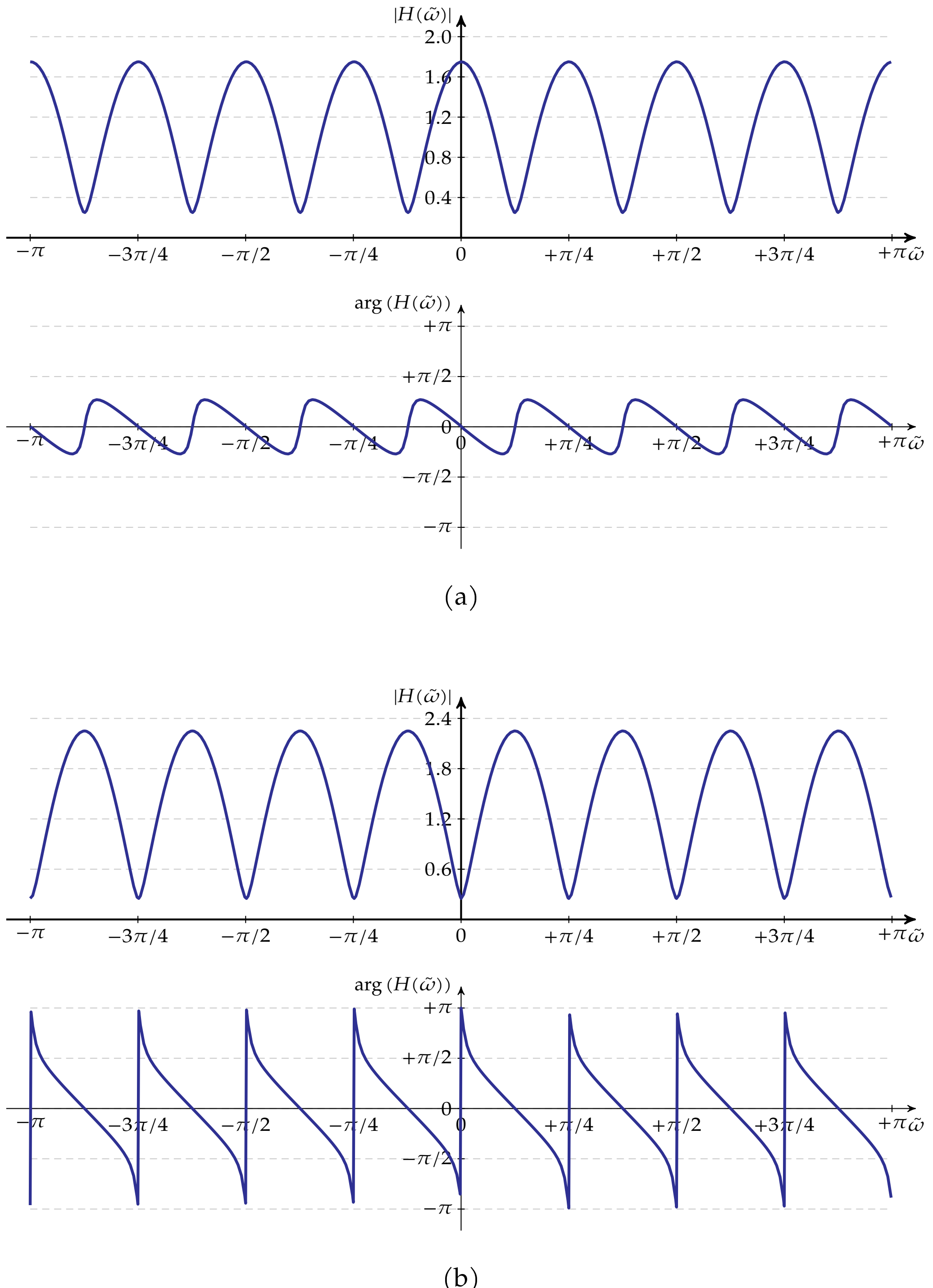


Figure 6.3: Frequency spectra of the feedforward comb filters of Figure 6.2 on the previous page, for the case $q = 8$, with (a) $a = 0.75$ and (b) $a = -1.25$

filter:

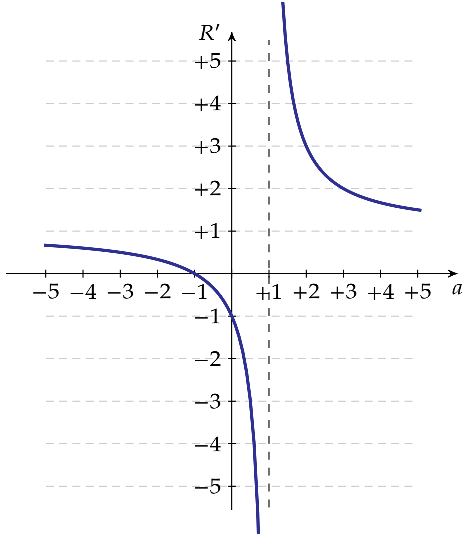
$$\begin{aligned}\left|H\left(e^{j\tilde{\omega}}\right)\right| &= \left|\frac{e^{jq\tilde{\omega}}+a}{e^{jq\tilde{\omega}}}\right| \\ &= \left|e^{jq\tilde{\omega}}+a\right| \\ &= \sqrt{(\cos(q\tilde{\omega})+a)^2+\sin^2(q\tilde{\omega})} \\ &= \sqrt{1+2a\cos(q\tilde{\omega})+a^2}\end{aligned}$$

The extrema of this filter are attained for $q\tilde{\omega} = m\pi$ with m integer. As minima and maxima alternate, we can easily calculate the comb ratio R:

$$\begin{aligned}R &= \frac{\left|H\left(e^{j0}\right)\right|}{\left|H\left(e^{j\frac{\pi}{q}}\right)\right|} = \sqrt{\frac{1+2a\cos 0+a^2}{1+2a\cos\pi+a^2}} = \sqrt{\frac{1+2a+a^2}{1-2a+a^2}} \\ &= \left|\frac{a+1}{a-1}\right|\end{aligned}$$

Absolute values are a nuisance when solving equations, so let's start by investigating

$$R' = \frac{a+1}{a-1}$$



Take a look at the graph of R' on the right. As you can see R' is positive outside the interval $[-1, 1]$. We'll therefore treat this interval separately from the rest.
Therefore:

- $-1 < a < 1$ leads to:

  $$R = \frac{1+a}{1-a}$$

  and therefore

  $$a = \frac{R-1}{R+1} \qquad (6.4)$$

- $|a| > 1$ leads to:

  $$R = \frac{a+1}{a-1}$$

  and therefore

  $$a = \frac{R+1}{R-1} \qquad (6.5)$$

Therefore, depending on the range of a, we have to pick one of the design equations (6.4) and (6.5).
The two equations yield a valid result for a specific R. Choosing either of them, influences the absolute values of the magnitude and the phase characteristic. You might want to check both, and decide which one is best for your application after analyzing the Bode plots of the two alternatives.

Exercises

Some exercises on the design of feedforward comb filters

Exercise 6.2.2-1: Design a feedforward comb filter with 6 dips and a comb ratio of 100 (i.e. a maximum at $\omega = 0$).
Verify the frequency response of all equal alternatives using OCTAVE.
Compose a direct form and a transposed form implementation for this filter.

Exercise 6.2.2-2: Design a feedforward comb filter with 7 dips and a comb ratio of 0.05 (i.e. a minimum at $\omega = 0$).
Verify the frequency response of all equal alternatives using OCTAVE.
Compose a direct form and a transposed form implementation for this filter.

Exercise 6.2.2-3: Design a feedforward comb filter with a magnitude spectrum exhibiting 42 peaks/dips, with a dip at zero frequency amounting to a gain of 2 and a peak level amounting 23.

6.2.3 Strengths and weaknesses

The zero-placement technique is a *very simple* design technique. However, it assumes that you were able to compose a zero-pattern (including poles at the origin) that accommodates your needs.

In addition, arbitrarily placing zeros does not guarantee the corresponding impulse response to be symmetrical. Hence, the generated filters will most likely not exhibit a linear-phase characteristic.

Because of the specific placement of poles and zeros, creating a specific spectrum, comb filters are an interesting subclass.

6.3 Impulse response invariance method

6.3.1 Design procedure

Assume that we want to approximate a system having an infinite (but causal) impulse response $h[n]$ using a finite impulse response filter (with impulse response $\tilde{h}[n]$ of length N).

The discrete-time infinite impulse response may even originate from sampling a continuous-time (analog) impulse response.[3]

There are two ways to approximate the discrete-time infinite impulse response by a finite one: (a) by truncation, (b) by windowing.

[3] Remember that this operation makes the corresponding spectrum periodic.

6.3.1.1 Truncation

As both impulse responses are vectors, a measure of the quality of such an approximation could be the distance between the two vectors[4], i.e. we want to solve

$$\min_{\tilde{h}[n]} \left\| \tilde{h}[n] - h[n] \right\|$$

Any valid norm will do in the definition above. In case we use a 2-norm, this definition corresponds to a least-squares approximation:

$$\min_{\tilde{h}[n]} \sqrt{\sum_{i=0}^{+\infty} \left(\tilde{h}[i] - h[i]\right)^2}$$

It is obvious that the solution to this minimization problem is obtained by setting:

$$\tilde{h}[n] = \begin{cases} h[n] & \text{if } 0 \leq n < N \\ 0 & \text{otherwise} \end{cases} \tag{6.6}$$

One can easily prove (using Parseval's theorem for the Discrete-time Fourier Transform) that this is also a best fit in the frequency domain from a least-squares point of view.

Actually, the solution as specified in (6.6) corresponds to truncating the original impulse response to the desired length. As we know, such a truncation corresponds to multiplying the original impulse response with a rectangular window function:

$$\tilde{h}[n] = h[n].r_N[n]$$

When analyzing *window functions* in depth, it is straightforward to derive that such a truncation operation corresponds to convolving the original frequency spectrum with a *sinc*-like spectrum.

$$\tilde{H}_p(\omega) = H_p(\omega) \star \underbrace{\mathrm{e}^{-\mathrm{j}\omega\frac{N-1}{2}T_s} \frac{\sin\frac{\omega N T_S}{2}}{\sin\frac{\omega T_S}{2}}}_{\text{sinc-like spectrum}}$$

Hence, the well-known effect of spectral leak will also occur when using this filter design method. This observation leads us to the obvious idea of replacing the rectangular window by a more advanced window function that reduces spectral leak. We will treat this in the next section.

6.3.1.2 Windowing

Instead of cutting the infinite impulse response to length using a rectangular window function (as we did in the previous section), we can also use a more advanced window function.

In this way, the *impulse invariance* is less optimal both in the time-domain and from a least-squares point of view in the frequency domain, but at least the spectral leak is lower. This translates itself in the frequency domain in a filter frequency spectrum that looks more like the original desired spectrum.

Usually, a trial-and-error procedure is used to find a window function that suits your needs.

[4]To this end, we padded the N-dimensional vector $\tilde{h}[n]$ with zeros to become of infinite length.

6.3.2 Example

Assume we'd like to make a discrete-time FIR-filter with impulse response length 64 to mimic the following impulse response:

$$h(t) = u(t)\,\mathrm{e}^{-\frac{t}{12}}$$

Let's assume that the sampling frequency is 2 Hz. This leads to an ideal discretized impulse response of

$$h[n] = u[n]\,\mathrm{e}^{-\frac{n}{24}}$$

Let's use OCTAVE/MATLAB to perform the calculations. Before we start, allow me to sermon you on some small but important details. In the code below, the alignment is just for better readability. Comments have been removed for the sake of simplicity. Remember, however, that proper axis labeling, proper titles and clear, appropriate comments are good engineering practice! Don't abandon that practice in scripts you write yourself. Proper documentation might seem a waste of time when writing code, but will be of utmost value when having to reconsider your design at a later stage.

The examples below are intended to be entered at the interpreter prompt. This allows you — if desired — to inspect the data after every command you enter. Alternatively you may collect all the commands in a script and execute them all at once. Probably, this is the way you will work in the future. Writing scripts allows you to archive your work.

Let's start by generating the truncated impulse response:

```
N = 64;
h = exp( - linspace(0,N-1,N)' / 24 );
```

We can plot this impulse response using:

```
plot(h);
title( "Truncated␣Impulse␣Response" );
ylabel( "h_t[n]" );
xlabel( "n" );
```

However, we'd better window this impulse response, e.g., using a Bartlett window:

```
hb = h;
hb .*= bartlett(N+1)(1:N);
```

Plot it using:

```
plot(hb);
title( "Windowed␣Impulse␣Response" );
ylabel( "h_w[n]" );
xlabel( "n" );
```

However, as you can see a symmetrical window severely impacts this originally exponential impulse response. Therefore, you may want to try an asymmetrical window:

```
hb = h;
hb .*= bartlett(2*N)(N+1:2*N);
```

We can observe the result by:

```
plot(hb);
title( "Windowed␣Impulse␣Response" );
ylabel( "h_w[n]" );
xlabel( "n" );
```

Exercises
Some exercises on the impulse invariance design method

Exercise 6.3.2-1: Make a digital filter that mimicks the following impulse response:

$$h(t) = u(t)\cos(2\pi t)\,\mathrm{e}^{-5t}$$

The sampling frequency is 10 Hz. The digital impulse response should be of length 64. Check the frequency response of the filter (by using the DtFT).

Exercise 6.3.2-2: Make a digital filter that mimicks the following impulse response:

$$h(t) = u(t)\frac{t}{1+t^2}$$

The sampling frequency is 1 Hz. The digital impulse response should be of length 32. Check the frequency response of the filter (by using the DtFT). Now, use the second half of a Hann window to shape your impulse reponse. Then, check the frequency response of the filter again (by using the DtFT).

6.3.3 Strengths and weaknesses

The impulse invariance method is appropriate if the time-domain behavior of the filter is important. A prerequisite for obtaining an acceptable filter is that almost all impulse response energy is contained within the truncation window. Applying a window function can smooth the truncation effects that can be observed in the frequency domain.

It must be noted that applying a window function to take care of unwanted frequency effects is a bit contradictory to the rationale for using this design method, i.e. preservation of the time-domain impulse response. Indeed, such a window severely impacts the impulse response shape. Therefore, in some cases asymmetrical windows are used (only affecting the tail of the impulse response). In most cases, no window is used at all.

6.4 Frequency sampling design method

6.4.1 Design procedure

In section 6.2.3 on page 120 we tried to make a best fit of the time-domain impulse response of an ideal filter we envisioned. Now, let's try to the same thing, but starting from the filter's desired (frequency domain) magnitude spectrum.

Because we want to obtain a discrete-time filter, we consider the desired spectrum to be periodic and therefore only consider the primary frequency range from $-\omega_s/2$ to $\omega_s/2$.

Again, there are two ways to elaborate this idea:
(a) by truncation, (b) by windowing.

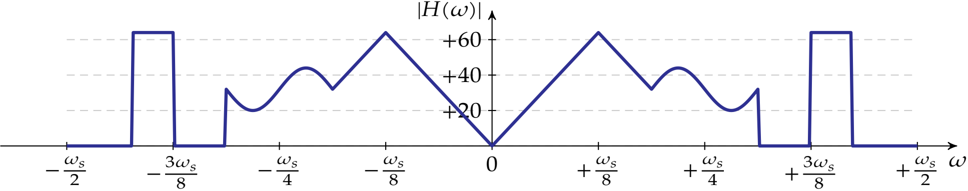
6.4.1.1 Truncation

Consider the following design procedure:

1. Consider the spectrum from $\omega = 0$ to $\omega = \omega_s$. Keep in mind that the spectrum is periodic.
2. sample the (frequency-domain) magnitude spectrum using $N+1$ equidistant points, putting the extreme points at 0 and $+\omega_s$; assume the phase spectrum to be zero
3. calculate the corresponding time-domain impulse response using the iFFT
4. move the samples at $n \geq N/2$ to $n - N$ to make the impulse response center-symmetric
5. shift the impulse response to make it causal
6. check the result using the FFT
7. finally check the result using a zero-padded FFT

In order to make the design procedure a little-bit less abstract, we'll use an arbitrary example to illustrate every step in the procedure.

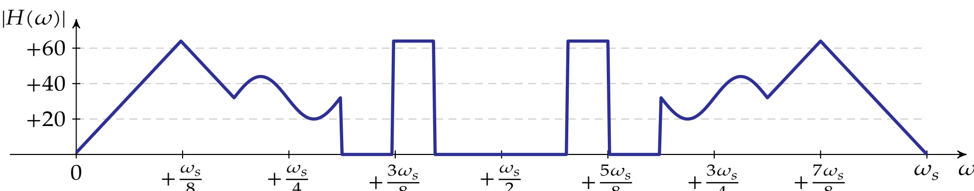
Starting point
Consider the following artistic example magnitude spectrum:



Take your time to be impressed by the spectrum above. The linear region, the sinusoidal region, the sharp edges are unachievable using analog electronics. Yet, using digital signal processing, we can!

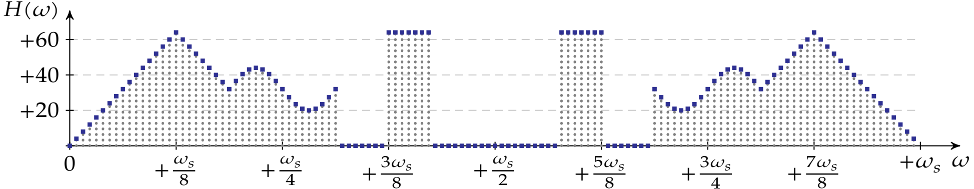
Step 1: Consider the spectrum from 0 to ω_s

The spectrum from the starting point (above) ranges from $-\omega_s/2$ to $+\omega_s/2$. However, the FFT that we are about to use, assumes the spectrum to range from 0 to ω_s. We will therefore perform step 1, i.e. consider the spectrum from $\omega = 0$ to $\omega = \omega_s$.
This results in the following spectrum:



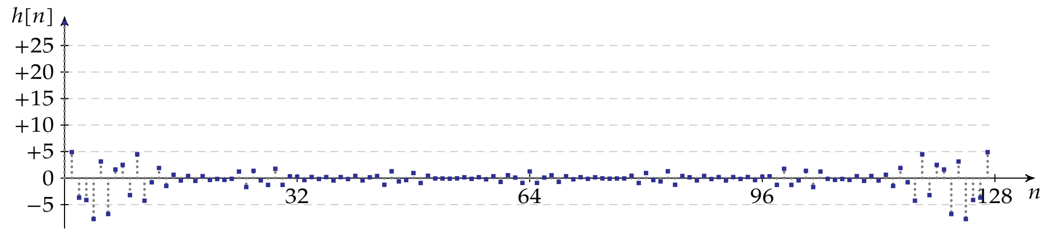
Step 2: sample the spectrum

Let's take 128 equidistant samples of this spectrum, by dividing the range 0 to ω_s in 128 equal parts, and taking the lower boundaries of these parts as sample points.



Step 3: Calculate the corresponding time-domain impulse response using the inverse FFT

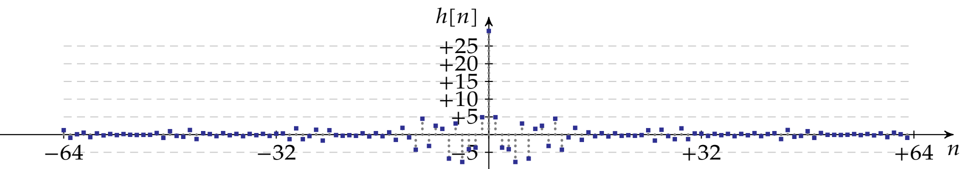
This results in the following time-domain signal:



However, the result of the inverse FFT gives us time-domain samples from the range 0 to N, while the true center-symmetric time-domain result is only visible if we look at the range from $-N/2$ tot $N/2 - 1$.[5] We will therefore perform step 4.

Step 4: move the samples at $n \geq N/2$ to $n - N$ to make the impulse response center-symmetrical

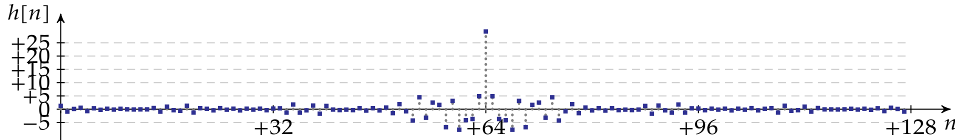
This results in the following time-domain signal:



As we can see, this signal is not causal. Therefore, let's shift it such that it becomes causal. We will therefore perform step 5.

Step 5: shift the signal such that it becomes causal

This results in the following time-domain signal:

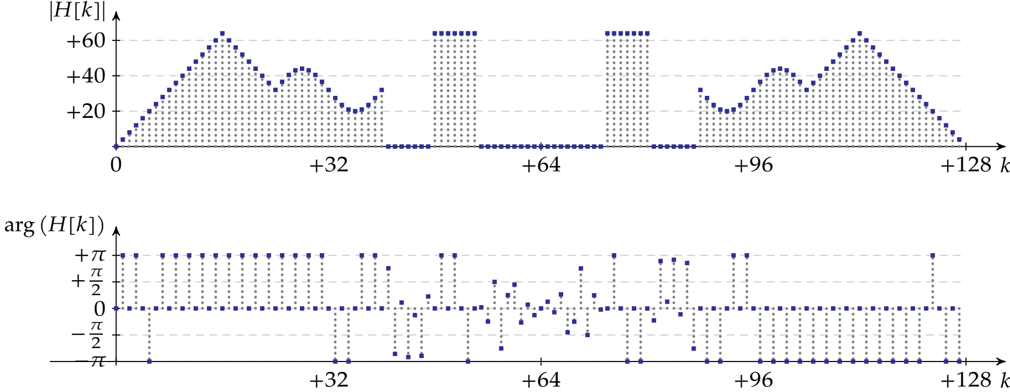


This time-domain shift corresponds to multiplying the frequency spectrum, with a factor $\mathrm{e}^{-\mathrm{j}k\pi}$. This causes the phase spectrum to be linear instead of of zero phase.

[5]We need the symmetry in the impulse response to ensure the filter has a linear-phase characteristic.

Step 6: check the result using the FFT

Let's again calculate the FFT of the resulting impulse response to see what the corresponding filter spectrum looks like.

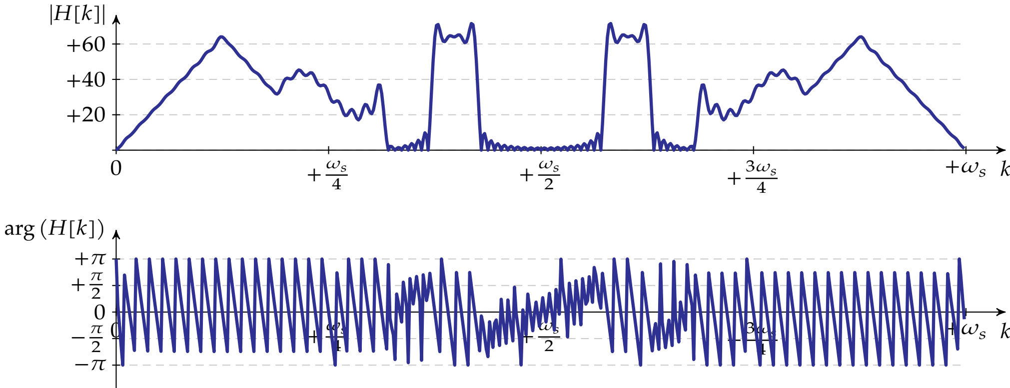


We can indeed observe the same magnitude spectrum as the spectrum of step 2. The phase values equal $k\pi$ except for the frequency points where the magnitude equals zero. In those points, the phase value deviates from the theoretical values of $k\pi$ due to numerical instability. This numerical instability is harmless because the magnitude of theses samples equals zero.

Step 7: finally check the result using a zero-padded FFT

The resulting FFT in the check phase above shows us a perfect result. However, we started from a continuous spectrum. How does the DtFT of the impulse response look like? We can calculate the DtFT in a numerical way, by applying zero-padding.

The (interpolated) result can be seen below.



This doesn't look very good, does it? How come? The frequency sampling of step 1, corresponds to truncating the infinite time-domain impulse response. This truncation causes spectral leak, causing all the bumps in the magnitude spectrum you can observe above. The spectral leak is only apparent in between the frequency points used for the frequency sampling and is therefore not visible on the earlier discrete-frequency graphs.

We can smooth out this effect by shaping the truncated impulse response using a window function. That's the subject of the next section.

6.4.1.2 Windowing

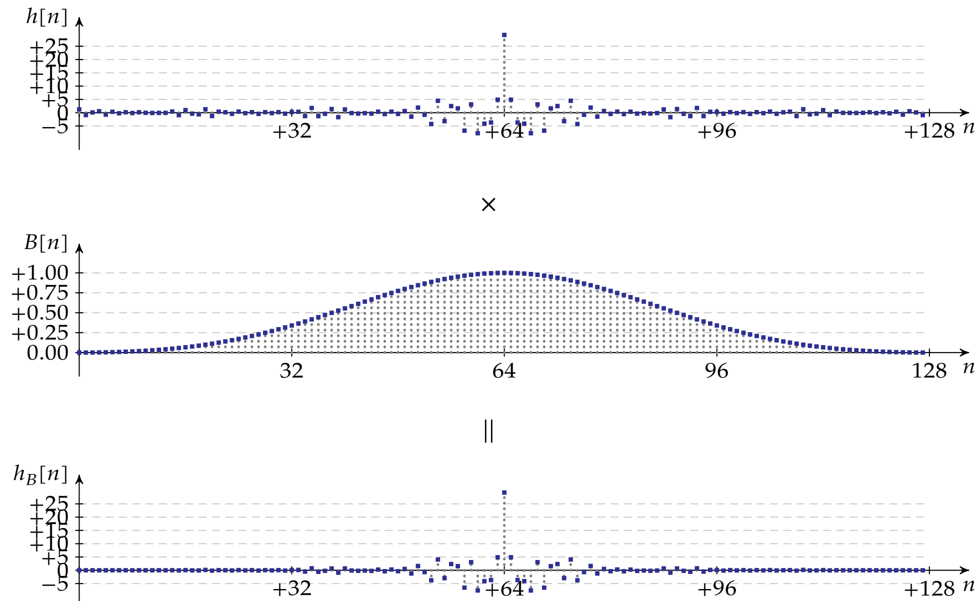
As we've seen in the previous section, frequency sampling without shaping the resulting impulse response obtained by inverse FFT, results in severe spectral leak in between the frequency points used for the frequency sampling.

Extending the procedure with a few steps cures this problem:

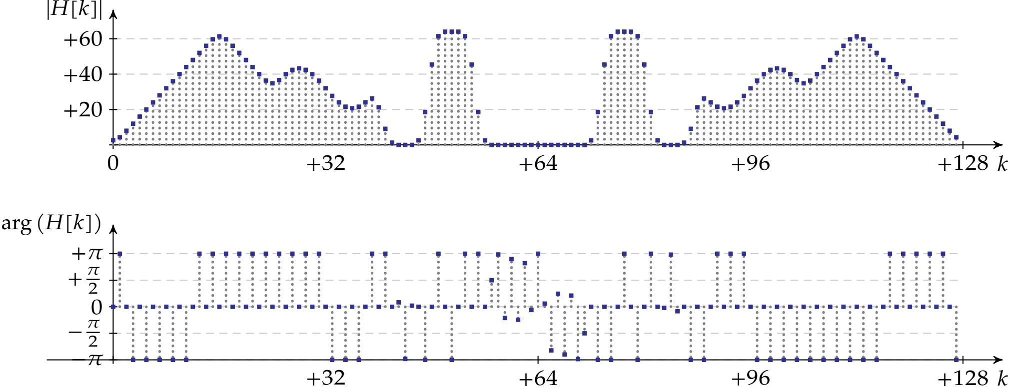
8. apply a window function to the resulting impulse response
9. check the result using the FFT
10. finally check the result using a zero-padded FFT

To illustrate the effect of a window function, we'll use the example of last section.

Step 8: apply a window function
Let's apply for example a Blackman window function:



Step 9: check the result using the FFT
Let's again calculate the FFT of the windowed impulse response to see what the corresponding filter spectrum looks like.



What probably strikes you, is the smoothing of the magnitude spectrum: the sharp transitions

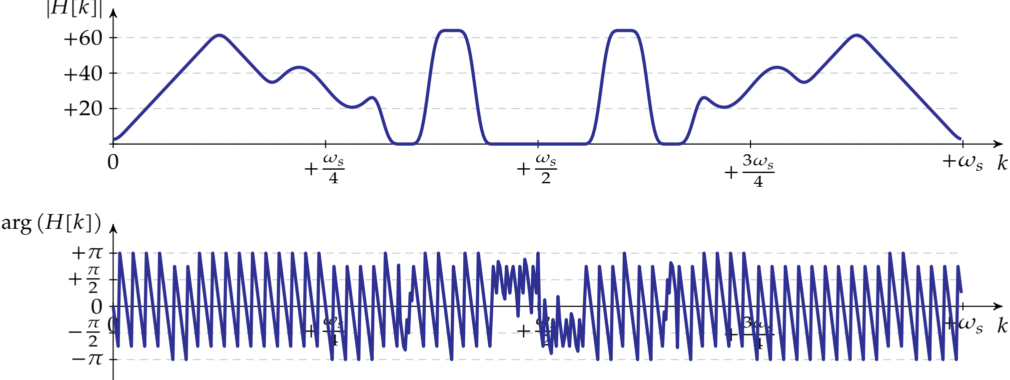
are totally gone: that's the price you pay for avoiding Gibbs phenomena. One can explore the trade-off curve between sharpness and Gibbs phenomena by trying different window functions.

Again, note that the phase values equal $k\pi$ except for the frequency points where the magnitude equals zero. In those points, the phase value deviates from the theoretical values of $k\pi$ due to numerical instability. Again, this numerical instability is harmless because the magnitude of theses samples equals zero.

At first sight, the width of the transition bands seems unpredictable. However, knowing that multiplication with a window function in the time domain corresponds to convolution in the frequency-domain, leads us to the simple conclusion: the transition-bands will have the same width as the main-lobe of the window's magnitude spectrum.

Step 10: finally check the result using a zero-padded FFT
How does the DtFT of the windowed impulse response look like? We can calculate the DtFT in a numerical way, by applying zero-padding.

The (interpolated) result can be seen below.



This looks much better, doesn't it? Again, we can observe the smoothing, but at least the Gibbs oscillations are gone.

6.4.2 Example

Let's implement an ideal low-pass filter using the frequency sampling design method. The filter transfer function is given by:

$$H(\omega) = \begin{cases} 1 & \text{for } |\omega| < 0.2\omega_s \\ 0 & \text{otherwise} \end{cases}$$

We can easily do this in Octave. To obtain normalized graphs we'll assume that $\omega_S = 2\pi$.

6.4.2.1 The hard way

Step 1 Let's sample this ideal filter (transfer) function using $N = 256$ samples.

```
N               = 256;
frequencies     = linspace( -pi, pi, N+1 )(1:N)';
filterfunction = (abs(frequencies) < 0.2 * 2*pi );
```

You can verify the result by plotting the filter's transfer function:

```
plot( frequencies, filterfunction, "s" );
title( "Desired Filter Spectrum" );
ylabel( "|H(w)|" );
xlabel( "w" );
```

Step 2 Now, let's shift the filter function such that it is ready to be submitted to the inverse FFT:

```
filterfunction = shift( filterfunction, -floor(N/2) );
```

Notice the floor function in calculating the shift length. The floor function serves for odd values of N.

You may check the result using:

```
plot( filterfunction, "s" );
title( "Desired Filter Spectrum (ready for iFFT)" );
ylabel( "|H[k]|" );
xlabel( "k" );
```

Step 3 Now let's calculate the corresponding impulse response using the inverse FFT:

```
impulseresponse = real( ifft( filterfunction ) );
```

Note the fact that we select the real part of the iFFT's output. Indeed, due to numerical noise, the imaginary part of the output will be close to, but not exactly, zero.

Check the result, using:

```
plot( impulseresponse, "s" );
title( "Impulse response" );
ylabel( "h[n]" );
xlabel( "n" );
```

Step 4 and 5 Make the impulse response center-symmetrical and causal:

```
impulseresponse = shift( impulseresponse, floor(N/2) );
```

And, check the result:

```
plot( impulseresponse, "s" );
title( "Impulse response (made symmetrical/causal)" );
ylabel( "h[n]" );
xlabel( "n" );
```

Step 6 Now, let's check the result using the FFT:

```
filterfunction  = fft( impulseresponse );
```

Shift it again to create a center-symmetrical view:

```
filterfunction  = shift( filterfunction, floor(N/2) );
```

Let's check the result:

```
subplot(2,1,1);
plot( frequencies, abs( filterfunction ), "s" );
ylabel( "|H[k]|" );
subplot(2,1,2);
plot( frequencies, angle( filterfunction ), "s" );
ylabel( "angle(H[k])" );
xlabel( "k" );
title( "Resulting␣FFT␣spectrum" );
```

Step 7 But, how does the result look like after zero-padding (mimicking the DtFT)? Let's first organize the zero-padding:

```
M = N*8;
frequencies_zp          = linspace(-pi, pi, M )(1:M);
impulseresponse_zp      = zeros( M, 1 );
impulseresponse_zp(1:N) = impulseresponse;
```

And then let's calculate the center-symmetrical FFT:

```
filterfunction_zp  = fft( impulseresponse_zp );
filterfunction_zp  = shift( filterfunction_zp, floor(M/2) );
```

This allows us to observe the true resulting spectrum:

```
subplot(2,1,1);
plot( frequencies_zp, abs( filterfunction_zp ) );
ylabel( "|H(w)|" );
subplot(2,1,2);
plot( frequencies_zp, angle( filterfunction_zp ) );
ylabel( "angle(H(w))" );
xlabel( "w" );
title( "Resulting␣DtFT␣spectrum" );
```

This does not look too good, does it? Considerable Gibbs effect spoils the fun. Let's apply a window function to solve this issue.

Step 8 The window function used below is the Blackman window. The trick with selecting N points of an $N+1$-point Blackman window, is due to OCTAVE's implementation of the Blackman window that does not correspond to ours. Ours keeps the impulse response symmetrical. OCTAVE's does not.[6]

[6] The full truth is the following: if N is odd, than OCTAVE's window functions are appropriate. In case N is even, ours are better.

```
impulseresponse_wd = impulseresponse .* blackman(N+1)(1:N);
```

The result can be inspected using:

```
plot( impulseresponse_wd, "s" );
ylabel( "h[n]" );
xlabel( "n" );
title( "Windowed␣impulse␣response" );
```

Step 9 Let's check the result using the FFT, generating a center-symmetrical spectrum:

```
filterfunction  = fft( impulseresponse_wd );
filterfunction  = shift( filterfunction, floor(N/2) );
```

A graphical representation can be generated using:

```
subplot(2,1,1);
plot( frequencies, abs( filterfunction ), "s" );
ylabel( "|H[k]|" );
subplot(2,1,2);
plot( frequencies, angle( filterfunction ), "s" );
ylabel( "angle(H[k])" );
xlabel( "k" );
title( "Resulting␣FFT␣spectrum␣(after␣windowing)" );
```

Step 10 Finally, let's check the end result using a zero-padded FFT (mimicking the DtFT). Let's start with the zero-padding:

```
M = N*8;
frequencies_zp          = linspace(-pi, pi, M )(1:M);
impulseresponse_zp      = zeros( M, 1 );
impulseresponse_zp(1:N) = impulseresponse_wd;
```

Calculate the center-symmetrical spectrum:

```
filterfunction_zp  = fft( impulseresponse_zp );
filterfunction_zp  = shift( filterfunction_zp, floor(M/2) );
```

Plot it!

```
subplot(2,1,1);
plot( frequencies_zp, abs( filterfunction_zp ) );
ylabel( "|H(w)|" );
subplot(2,1,2);
plot( frequencies_zp, angle( filterfunction_zp ) );
ylabel( "angle(H(w))" );
xlabel( "w" );
title( "Resulting␣DtFT␣spectrum␣(after␣windowing)" );
```

6.4.2.2 The easy way

Octave contains two functions that ease the process of designing filters using the frequency sampling design method:

- `fir1`: to design low-pass, high-pass, band-pass, band-stop or alternating pass/stop filters
- `fir2`: to design filters with a (somewhat) arbitrary frequency response

Check out the help facility for usage information. Neither `fir1`, nor `fir2` have the same flexibility as our fully manual design. However, they're quite convenient.

After reading the documentation, it's not too hard to find out that the low-pass filter that we implemented above, can simply be designed using the following piece of code:

```
N = 256;
impulseresponse = fir1( N-1, 0.4, "blackman", "noscale" );
```

Careful inspection will reveal that the result is not exactly the same. However, the correspondence is more than satisfactory.

You can refer to previous section when trying to verify this by plotting the FFT and the DtFT spectrum.

Exercises
Some exercises on the frequency sampling design method (FSDM)

Exercise 6.4.2.2-1: Use the FSDM to design a bandpass filter that realizes the following transfer function magnitude:

$$|H(\omega)| = \begin{cases} 0 & \text{for } |\omega| < 0.15\omega_s \\ 1 & \text{for } 0.15\omega_s \leq |\omega| < 0.3\omega_s \\ 0 & \text{for } 0.3\omega_s \leq |\omega| \end{cases}$$

Make the design

1. by hand for $N = 6$ (i.e. only using a calculator)
2. using Octave for $N = 64$, following "the hard way"
3. using Octave's `fir1` function for $N = 64$

Make the design twice:

- using truncation
- using a Blackman window

Check the result of both designs using the DtFT (i.e. a zero-padded DFT). For this step you may use Octave's FFT routines in all three cases.

Exercise 6.4.2.2-2: Use the FSDM to design a filter that realizes the following transfer function magnitude:

$$|H(\omega)| = \begin{cases} 0.5 & \text{for } |\omega| < 0.2\omega_s \\ \frac{-5}{\omega_s}(\omega - 0.4\omega_s) & \text{for } 0.2\omega_s \leq |\omega| < 0.4\omega_s \\ 0 & \text{for } 0.4\omega_s \leq |\omega| \end{cases}$$

Make the design

1. using OCTAVE for $N = 256$, following "the hard way"
2. using OCTAVE's `fir2` function for $N = 256$

Make the design twice:

- using truncation
- using a Hamming window

Check the result of both designs using the DtFT (i.e. a zero-padded DFT). For this step you may use OCTAVE's FFT routines in all four cases.

6.4.3 Strengths and weaknesses

The obvious strength of this filter design method is that one can specify an arbitrary frequency-domain magnitude spectrum, and generate a filter that implements this spectrum very accurately while guaranteeing a linear-phase characteristic.

The disadvantage of this filter design method is that we need window functions to avoid Gibbs oscillations. This also implies that zero-width transition bands that occur in our original specification will translate themselves into transition bands with a finite (uncontrollable and unpredictable) length. Trial and error using different window functions is required to obtain a satisfactory result. Knowledge about the width of the main lobe of the window's spectrum helps, but the deviation in the transition bands are not under your control.

A more theoretical objection is that this design method is not optimal in any particular way.

6.5 Optimal linear-phase filter design

6.5.1 Introduction

The next two methods are based on optimal approximation of a desired filter response. These methods allow detailed control of the approximation errors (even in the transition bands).

With both methods we intend to make filters that exhibit a linear-phase characteristic. Therefore, we need to study some basic properties of linear-phase FIR filters.

Linear-phase FIR filters can be subdivided into 4 categories (type I, type II, type III and type IV) according to the following table:

	FIR symmetry	
FIR Length	Symmetrical	Antisymmetrical
Odd	I	III
Even	II	IV

In the following paragraphs, we will derive generic expressions for the frequency response of these filter types to demonstrate that they realize linear-phase filters.

Remarks

1. If you find the math too complicated, try demonstrating the properties using specific examples for low values of N. In this way, you can avoide the generic summation signs. This may make the math more tractable. Understanding the generalized mathematical derivation should be more easy afterwards.
2. We also could denote the FIR symmetry properties as 'even' (symmetrical) and 'odd' (antisymmetrical). However, this would make the terminology rather confusing. Therefore, let's stick to the terms 'symmetrical' and 'antisymmetrical'.

Let's start with an arbitrary impulse response $h[n]$ of odd length $N = 2M + 1$ (left column) and even length $N = 2M$ (right column) with in both cases M a positive integer.

$$h[n] = \begin{cases} h_n & \text{for } 0 \leq n \leq 2M \\ 0 & \text{otherwise} \end{cases} \qquad h[n] = \begin{cases} h_n & \text{for } 0 \leq n \leq 2M-1 \\ 0 & \text{otherwise} \end{cases}$$

The (two-sided) Z-transform of this impulse response equals:

$$H(z) = \sum_{n=0}^{2M} h[n] z^{-n} \qquad H(z) = \sum_{n=0}^{2M-1} h[n] z^{-n}$$

The corresponding Discrete-time Fourier Transform can easily be obtained by setting

$$z = \mathrm{e}^{\mathrm{j}\omega T_s} = \mathrm{e}^{\mathrm{j}2\pi\frac{\omega}{\omega_s}}$$

To ease the math that follows, let's introduce the *normalized frequency*:

$$\tilde{\omega} = 2\pi \frac{\omega}{\omega_s}$$

This leads to:

$$H(\tilde{\omega}) = \sum_{n=0}^{2M} h_n \, \mathrm{e}^{-\mathrm{j}n\tilde{\omega}} \qquad (6.7) \qquad\qquad H(\tilde{\omega}) = \sum_{n=0}^{2M-1} h_n \, \mathrm{e}^{-\mathrm{j}n\tilde{\omega}} \qquad (6.8)$$

FIR — Type I (symmetrical, odd)

We know that the impulse response is symmetrical w.r.t. $n = M$, i.e.

$$h_n = h_{2M-n}, \qquad \text{for } n = 0, 1, 2, \ldots, M$$

Therefore, it makes sense combining the terms in (6.7) with equal coefficients, leading to:

$$H(\tilde{\omega}) = h_M \,\mathrm{e}^{-\mathrm{j}M\tilde{\omega}} + \sum_{n=0}^{M-1} h_n \left(\mathrm{e}^{-\mathrm{j}n\tilde{\omega}} + \mathrm{e}^{-\mathrm{j}(2M-n)\tilde{\omega}} \right)$$

From this equation, we can bring the factor $\mathrm{e}^{-\mathrm{j}M\tilde{\omega}}$ upfront[7], making the argument of the summation even more symmetrical such that we can recognize a cosine function in it.

$$\begin{aligned} H(\tilde{\omega}) &= \mathrm{e}^{-\mathrm{j}M\tilde{\omega}} \left(h_M + \sum_{n=0}^{M-1} h_n \left(\mathrm{e}^{-\mathrm{j}(n-M)\tilde{\omega}} + \mathrm{e}^{-\mathrm{j}(M-n)\tilde{\omega}} \right) \right) \\ &= \mathrm{e}^{-\mathrm{j}M\tilde{\omega}} \left(h_M + 2 \sum_{n=0}^{M-1} h_n \cos\left((M-n)\tilde{\omega}\right) \right) \end{aligned}$$

As one can readily see, this spectrum has linear phase introducing a delay of M samples. Often, this frequency decomposition is written as:

$$H(\tilde{\omega}) = \mathrm{e}^{-\mathrm{j}M\tilde{\omega}} \sum_{n=0}^{M} q_n \cos\left((M-n)\tilde{\omega}\right) \tag{6.9}$$

with

$$q_i = \begin{cases} h_M & \text{if } i = M \\ 2h_i & \text{if } i = 0,1,2,3,\dots,M-1 \end{cases}$$

FIR — Type II (symmetrical, even)

We know that the impulse response is symmetrical w.r.t. $n = M - \frac{1}{2}$, i.e.

$$h_n = h_{2M-1-n}, \qquad \text{for } n = 0,1,2,\dots,M-1$$

Therefore, it makes sense combining the terms in (6.8) with equal coefficients, leading to:

$$H(\tilde{\omega}) = \sum_{n=0}^{M-1} h_n \left(\mathrm{e}^{-\mathrm{j}n\tilde{\omega}} + \mathrm{e}^{-\mathrm{j}(2M-1-n)\tilde{\omega}} \right)$$

From this equation, we can bring the factor $\mathrm{e}^{-\mathrm{j}\frac{2M-1}{2}\tilde{\omega}}$ upfront[8], making the argument of the summation even more symmetrical such that we can recognize a cosine function in it.

$$\begin{aligned} H(\tilde{\omega}) &= \mathrm{e}^{-\mathrm{j}\frac{2M-1}{2}\tilde{\omega}} \sum_{n=0}^{M-1} h_n \left(\mathrm{e}^{-\mathrm{j}(n-\frac{2M-1}{2})\tilde{\omega}} + \mathrm{e}^{-\mathrm{j}(\frac{2M-1}{2}-n)\tilde{\omega}} \right) \\ &= \mathrm{e}^{-\mathrm{j}\frac{2M-1}{2}\tilde{\omega}} \sum_{n=0}^{M-1} 2h_n \cos\left(\left(\frac{2M-1}{2} - n\right)\tilde{\omega}\right) \end{aligned}$$

As one can readily see, this spectrum has linear phase introducing a delay of $\frac{2M-1}{2}$ samples. Often, this frequency decomposition is written as:

$$H(\tilde{\omega}) = \mathrm{e}^{-\mathrm{j}\frac{2M-1}{2}\tilde{\omega}} \sum_{n=0}^{M-1} q_n \cos\left(\left(\frac{2M-1}{2} - n\right)\tilde{\omega}\right)$$

with

$$q_i = 2h_i \qquad \text{for } i = 0,1,2,3,\dots,M-1$$

[7] This factor is the geometric average of the two exponential terms.
[8] This factor is the geometric average of the two exponential terms.

FIR — Type III (antisymmetrical, odd)
We know that the impulse response is asymmetrical w.r.t. $n = M$, i.e.

$$h_n = -h_{2M-n}, \qquad \text{for } n = 0, 1, 2, \ldots, M$$

Notice that for $n = M$ this leads to $h_M = -h_M$ and therefore $h_M = 0$.

In view of this antisymmetry, it makes sense combining the terms in (6.7) with opposite coefficients, leading to:

$$H(\tilde{\omega}) = \sum_{n=0}^{M} h_n \left(\mathrm{e}^{-\mathrm{j}n\tilde{\omega}} - \mathrm{e}^{-\mathrm{j}(2M-n)\tilde{\omega}} \right)$$

From this equation, we can bring the factor $\mathrm{e}^{-\mathrm{j}M\tilde{\omega}}$ upfront[9], making the argument of the summation even more (anti)symmetrical such that we can recognize a sine function in it.

$$\begin{aligned} H(\tilde{\omega}) &= \mathrm{e}^{-\mathrm{j}M\tilde{\omega}} \sum_{n=0}^{M-1} h_n \left(\mathrm{e}^{-\mathrm{j}(n-M)\tilde{\omega}} - \mathrm{e}^{-\mathrm{j}(M-n)\tilde{\omega}} \right) \\ &= \mathrm{e}^{-\mathrm{j}M\tilde{\omega}} 2\mathrm{j} \sum_{n=0}^{M-1} h_n \sin\left((M-n)\tilde{\omega}\right) \end{aligned}$$

As one can readily see, this spectrum has linear phase introducing a delay of M samples + an extra leading phase shift of $\pi/2$. Often, this frequency decomposition is written as:

$$H(\tilde{\omega}) = \mathrm{e}^{-\mathrm{j}(M\tilde{\omega}+\pi/2)} \sum_{n=0}^{M-1} q_n \sin\left((M-n)\tilde{\omega}\right) \tag{6.10}$$

with

$$q_i = 2h_i \qquad \text{for } i = 0, 1, 2, 3, \ldots, M-1$$

Also, notice that (6.10) implies that type III filters have a zero for $\tilde{\omega} = 0$.

FIR — Type IV (antisymmetrical, even)
We know that the impulse response is antisymmetrical w.r.t. $n = M - \frac{1}{2}$, i.e.

$$h_n = -h_{2M-1-n}, \qquad \text{for } n = 0, 1, 2, \ldots, M-1$$

Therefore, it makes sense combining the terms in (6.8) with opposite coefficients, leading to:

$$H(\tilde{\omega}) = \sum_{n=0}^{M-1} h_n \left(\mathrm{e}^{-\mathrm{j}n\tilde{\omega}} - \mathrm{e}^{-\mathrm{j}(2M-1-n)\tilde{\omega}} \right)$$

From this equation, we can bring the factor $\mathrm{e}^{-\mathrm{j}\frac{2M-1}{2}\tilde{\omega}}$ upfront[10], making the argument of the summation even more symmetrical such that we can recognize a sine function in it.

$$\begin{aligned} H(\tilde{\omega}) &= \mathrm{e}^{-\mathrm{j}\frac{2M-1}{2}\tilde{\omega}} \sum_{n=0}^{M-1} h_n \left(\mathrm{e}^{-\mathrm{j}(n-\frac{2M-1}{2})\tilde{\omega}} - \mathrm{e}^{-\mathrm{j}(\frac{2M-1}{2}-n)\tilde{\omega}} \right) \\ &= \mathrm{e}^{-\mathrm{j}\frac{2M-1}{2}\tilde{\omega}} \sum_{n=0}^{M-1} 2\mathrm{j}h_n \sin\left(\left(\frac{2M-1}{2} - n\right)\tilde{\omega}\right) \end{aligned}$$

[9] This factor is the geometric average of the two exponential terms.
[10] This factor is the geometric average of the two exponential terms.

As one can readily see, this spectrum has linear phase introducing a delay of $\frac{2M-1}{2}$ samples with an extra phase shift of $\pi/2$. Often, this frequency decomposition is written as:

$$H(\tilde{\omega}) = \mathrm{e}^{-\mathrm{j}\left(\frac{2M-1}{2}\tilde{\omega}+\pi/2\right)} \sum_{n=0}^{M-1} q_n \sin\left(\left(\frac{2M-1}{2} - n\right)\tilde{\omega}\right) \tag{6.11}$$

with

$$q_i = 2h_i \qquad \text{for } i = 0, 1, 2, 3, \ldots, M-1$$

Also, notice that (6.11) implies that type IV filters have a zero for $\tilde{\omega} = 0$.

Summary

Time to summarize the results for the four linear-phase FIR filter types. While summarizing, we can further enhance the similarity of the equations describing these filters by introducing a new variable K:

$$K = \frac{N-1}{2}$$

You can find an overview in Table 6.1 on the next page. The overview requires a little comment: for type II and type IV filters, K is not an integer, but an integer plus or minus a half, hence the 'floor' (round to the nearest smaller or equal integer) symbols $\lfloor \cdot \rfloor$.

In summary, for all linear-phase FIR filter types, we can write:

$$H(\tilde{\omega}) = \mathrm{e}^{-\mathrm{j}(K\tilde{\omega}+\alpha)} \sum_{n=0}^{\lfloor K \rfloor} q_n f\left((K-n)\,\tilde{\omega}\right) \tag{6.12}$$

with appropriate values for q_n (see Table 6.1 on the following page) and with f denoting cos or sin, and α being 0 or $\pi/2$.

Hence:

$$|H(\tilde{\omega})| = \left| \sum_{n=0}^{\lfloor K \rfloor} q_n f\left((K-n)\,\tilde{\omega}\right) \right| \tag{6.13}$$

Let's also define the following variant of this magnitude function:

$$\hat{H}(\tilde{\omega}) = \sum_{n=0}^{\lfloor K \rfloor} q_n f\left((K-n)\,|\tilde{\omega}|\right) \tag{6.14}$$

For type I and II filters, this function is equal to the magnitude of $H(\tilde{\omega})$ except for sign changes when it crosses zero.
For type III and type IV filters, the same statement holds for $\omega \geq 0$. Yet, in addition, $\overline{H}(\tilde{\omega})$ is made even, such that it resembles more the magnitude spectrum of a real signal.

6.5.2 Least-squares method

Consider a desired center-symmetrical magnitude spectrum $G(\tilde{\omega})$. We assume $G(\tilde{\omega})$ to be of zero-phase. Our goal is to find coefficients h_n defining $h[n]$ and $H(\omega)$ of a linear-phase FIR filter, such that the weighted integral square error

$$\epsilon_2 = \|W(\tilde{\omega})\left(|H(\tilde{\omega})| - G(\tilde{\omega})\right)\|_2$$

Type I

$$H(\tilde{\omega}) = \mathrm{e}^{-\mathrm{j}K\tilde{\omega}} \sum_{n=0}^{K} q_n \cos\left((K-n)\tilde{\omega}\right)$$

with

$$q_n = \begin{cases} h_K & \text{if } n = K \\ 2h_i & \text{for } n \text{ integer and } 0 \leq n < K \end{cases}$$

Type III

$$H(\tilde{\omega}) = \mathrm{e}^{-\mathrm{j}(K\tilde{\omega}+\pi/2)} \sum_{n=0}^{K} q_n \sin\left((K-n)\tilde{\omega}\right)$$

with

$$q_n = \begin{cases} h_K = 0 & \text{if } n = K \\ 2h_i & \text{for } n \text{ integer and } 0 \leq n < K \end{cases}$$

Type II

$$H(\tilde{\omega}) = \mathrm{e}^{-\mathrm{j}K\tilde{\omega}} \sum_{n=0}^{\lfloor K \rfloor} q_n \cos\left((K-n)\,\tilde{\omega}\right)$$

with

$$q_n = 2h_i \qquad \text{for } n \text{ integer and } 0 \leq n < K$$

Type IV

$$H(\tilde{\omega}) = \mathrm{e}^{-\mathrm{j}(K\tilde{\omega}+\pi/2)} \sum_{n=0}^{\lfloor K \rfloor} q_n \sin\left((K-n)\,\tilde{\omega}\right)$$

with

$$q_n = 2h_i \qquad \text{for } n \text{ integer and } 0 \leq n < K$$

$$K = \frac{N-1}{2}$$
$$\tilde{\omega} = 2\pi\frac{\omega}{\omega_s}$$

Table 6.1: Overview of the spectral equations of the four linear-phase FIR filter types with kernel length N

is minimal.

W is a center-symmetrical weighting function defined on $[-\pi, +\pi]$. The value of ϵ_2 is by mathematicians often referred to as the L_2-error.

In English: we're looking for filter coefficients h_n (or q_n) such that the magnitude spectrum of $H(\tilde{\omega})$ is a weighted least-squares approximation of the magnitude spectrum of $G(\tilde{\omega})$.

Solving this minimization problem is cumbersome, because of the magnitude operator it contains. Instead, we will try to minimize a variant of L_2-error:

$$\hat{\epsilon}_2 = \left\| W(\tilde{\omega}) \left(\hat{H}(\tilde{\omega}) - G(\tilde{\omega}) \right) \right\|_2$$

A solution that minimizes $\hat{\epsilon}_2$ might not be a perfect minimum for ϵ_2, but will be very close to it. Also, notice that using $\hat{\epsilon}_2$ also allows letting $G(\tilde{\omega}) < 0$.

6.5.2.1 Design Method 1: grid approach

Let's start by sampling the normalized frequency range from 0 to π. We only sample the positive half of the principal frequency range, because all amplitude spectra are assumed to be symmetrical. To this end, consider a grid of L frequencies $\tilde{\omega}_k, k = 1, 2, \ldots, L$.

For now, we'll consider the weighting function to be equaling one (for all frequencies). This lightens the mathematical treatment. Besides, we can reintroduce the weighting later again.

Filter objective Let's try to make a FIR filter for which $\hat{H}(\tilde{\omega})$ matches the desired spectrum $G(\tilde{\omega})$ on the frequency-grid points, i.e.

$$\begin{cases} \hat{H}(\tilde{\omega}_1) & = & G(\tilde{\omega}_1) \\ \hat{H}(\tilde{\omega}_2) & = & G(\tilde{\omega}_2) \\ \hat{H}(\tilde{\omega}_3) & = & G(\tilde{\omega}_3) \\ \quad\vdots \\ \hat{H}(\tilde{\omega}_L) & = & G(\tilde{\omega}_L) \end{cases} \tag{6.15}$$

Using (6.14), and knowing that $\tilde{\omega}_i \geq 0$ for all $i = 1, 2, 3, \ldots, L$, we can rewrite (6.15):

- for type I

$$\begin{cases} q_0 f(K\tilde{\omega}_1) + q_1 f((K-1)\tilde{\omega}_1) + q_2 f((K-2)\tilde{\omega}_1) + \cdots + q_{K-1} f(\tilde{\omega}_1) + q_K f(0) = G(\tilde{\omega}_1) \\ q_0 f(K\tilde{\omega}_2) + q_1 f((K-1)\tilde{\omega}_2) + q_2 f((K-2)\tilde{\omega}_2) + \cdots + q_{K-1} f(\tilde{\omega}_2) + q_K f(0) = G(\tilde{\omega}_2) \\ q_0 f(K\tilde{\omega}_3) + q_1 f((K-1)\tilde{\omega}_3) + q_2 f((K-2)\tilde{\omega}_3) + \cdots + q_{K-1} f(\tilde{\omega}_3) + q_K f(0) = G(\tilde{\omega}_3) \\ \quad\vdots \\ q_0 f(K\tilde{\omega}_L) + q_1 f((K-1)\tilde{\omega}_L) + q_2 f((K-2)\tilde{\omega}_L) + \cdots + q_{K-1} f(\tilde{\omega}_L) + q_K f(0) = G(\tilde{\omega}_L) \end{cases}$$

  In matrix notation this becomes:

$$\boldsymbol{A}\vec{X} = \vec{B}$$

assuming:

$$\boldsymbol{A} = \begin{bmatrix} f(K\tilde{\omega}_1) & f((K-1)\tilde{\omega}_1) & f((K-2)\tilde{\omega}_1) & \cdots & f(\tilde{\omega}_1) & f(0) \\ f(K\tilde{\omega}_2) & f((K-1)\tilde{\omega}_2) & f((K-2)\tilde{\omega}_2) & \vdots & f(\tilde{\omega}_2) & f(0) \\ f(K\tilde{\omega}_3) & f((K-1)\tilde{\omega}_3) & f((K-2)\tilde{\omega}_3) & \cdots & f(\tilde{\omega}_3) & f(0) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ f(K\tilde{\omega}_L) & f((K-1)\tilde{\omega}_L) & f((K-2)\tilde{\omega}_L) & \cdots & f(\tilde{\omega}_L) & f(0) \end{bmatrix}$$

and

$$\vec{B} = \begin{bmatrix} G(\tilde{\omega}_1) \\ G(\tilde{\omega}_2) \\ G(\tilde{\omega}_3) \\ \vdots \\ G(\tilde{\omega}_L) \end{bmatrix} \qquad \vec{X} \qquad = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ \vdots \\ q_K \end{bmatrix}$$

- for type III

$$\begin{cases} q_0 f(K\tilde{\omega}_1) + q_1 f((K-1)\tilde{\omega}_1) + q_2 f((K-2)\tilde{\omega}_1) + \cdots + q_{K-1} f(\tilde{\omega}_1) = G(\tilde{\omega}_1) \\ q_0 f(K\tilde{\omega}_2) + q_1 f((K-1)\tilde{\omega}_2) + q_2 f((K-2)\tilde{\omega}_2) + \cdots + q_{K-1} f(\tilde{\omega}_2) = G(\tilde{\omega}_2) \\ q_0 f(K\tilde{\omega}_3) + q_1 f((K-1)\tilde{\omega}_3) + q_2 f((K-2)\tilde{\omega}_3) + \cdots + q_{K-1} f(\tilde{\omega}_3) = G(\tilde{\omega}_3) \\ \vdots \\ q_0 f(K\tilde{\omega}_L) + q_1 f((K-1)\tilde{\omega}_L) + q_2 f((K-2)\tilde{\omega}_L) + \cdots + q_{K-1} f(\tilde{\omega}_L) = G(\tilde{\omega}_L) \end{cases}$$

and $q_K = 0$.
In matrix notation this becomes:

$$\boldsymbol{A}\vec{X} = \vec{B}$$

assuming:

$$\boldsymbol{A} = \begin{bmatrix} f(K\tilde{\omega}_1) & f((K-1)\tilde{\omega}_1) & f((K-2)\tilde{\omega}_1) & \cdots & f(\tilde{\omega}_1) \\ f(K\tilde{\omega}_2) & f((K-1)\tilde{\omega}_2) & f((K-2)\tilde{\omega}_2) & \vdots & f(\tilde{\omega}_2) \\ f(K\tilde{\omega}_3) & f((K-1)\tilde{\omega}_3) & f((K-2)\tilde{\omega}_3) & \cdots & f(\tilde{\omega}_3) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ f(K\tilde{\omega}_L) & f((K-1)\tilde{\omega}_L) & f((K-2)\tilde{\omega}_L) & \cdots & f(\tilde{\omega}_L) \end{bmatrix}$$

and

$$\vec{B} = \begin{bmatrix} G(\tilde{\omega}_1) \\ G(\tilde{\omega}_2) \\ G(\tilde{\omega}_3) \\ \vdots \\ G(\tilde{\omega}_L) \end{bmatrix} \qquad \vec{X} \qquad = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ \vdots \\ q_{K-1} \end{bmatrix}$$

and $q_K = 0$.

- for type II and IV

$$\begin{cases} q_0 f(K\tilde{\omega}_1) + q_1 f((K-1)\tilde{\omega}_1) + q_2 f((K-2)\tilde{\omega}_1) + \cdots + q_{K-1/2} f(\tilde{\omega}_1/2) = G(\tilde{\omega}_1) \\ q_0 f(K\tilde{\omega}_2) + q_1 f((K-1)\tilde{\omega}_2) + q_2 f((K-2)\tilde{\omega}_2) + \cdots + q_{K-1/2} f(\tilde{\omega}_2/2) = G(\tilde{\omega}_2) \\ q_0 f(K\tilde{\omega}_3) + q_1 f((K-1)\tilde{\omega}_3) + q_2 f((K-2)\tilde{\omega}_3) + \cdots + q_{K-1/2} f(\tilde{\omega}_3/2) = G(\tilde{\omega}_3) \\ \vdots \\ q_0 f(K\tilde{\omega}_L) + q_1 f((K-1)\tilde{\omega}_L) + q_2 f((K-2)\tilde{\omega}_L) + \cdots + q_{K-1/2} f(\tilde{\omega}_L/2) = G(\tilde{\omega}_L) \end{cases}$$

In matrix notation this becomes:

$$\boldsymbol{A}\vec{X} = \vec{B}$$

assuming:

$$\boldsymbol{A} = \begin{bmatrix} f(K\tilde{\omega}_1) & f((K-1)\tilde{\omega}_1) & f((K-2)\tilde{\omega}_1) & \cdots & f(\tilde{\omega}_1/2) \\ f(K\tilde{\omega}_2) & f((K-1)\tilde{\omega}_2) & f((K-2)\tilde{\omega}_2) & \vdots & f(\tilde{\omega}_2/2) \\ f(K\tilde{\omega}_3) & f((K-1)\tilde{\omega}_3) & f((K-2)\tilde{\omega}_3) & \cdots & f(\tilde{\omega}_3/2) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ f(K\tilde{\omega}_L) & f((K-1)\tilde{\omega}_L) & f((K-2)\tilde{\omega}_L) & \cdots & f(\tilde{\omega}_L/2) \end{bmatrix}$$

and

$$\vec{B} = \begin{bmatrix} G(\tilde{\omega}_1) \\ G(\tilde{\omega}_2) \\ G(\tilde{\omega}_3) \\ \vdots \\ G(\tilde{\omega}_L) \end{bmatrix} \qquad \vec{X} \qquad = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ \vdots \\ q_{K-1/2} \end{bmatrix}$$

Least-squares solution Assuming $\boldsymbol{A}$ is of full rank, the system of equations $\boldsymbol{A}\vec{X} = \vec{B}$ is

1. underdetermined if the number of unknowns is greater than the number of equations
2. exactly determined if the number of unknowns is equal to the number of equations
3. overdetermined if the number of unkowns is smaller than the number of equations

Assuming we want to use a dense frequency grid, the third case is most likely. In this case the system has no solution. However, we can calculate a least-squares approximation $\tilde{X}$ using the generalized inverse $A^{\dagger}$:

$$\vec{\tilde{X}} = \boldsymbol{A}^{\dagger}\vec{B}$$

with

$$\boldsymbol{A}^{\dagger} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T$$

In practice, the generalized inverse is not computed explicitly (for numerical reasons[11]). Instead, the overdetermined system is solved using techniques like QR-factorization. Using OCTAVE or MATLAB we don't need to care about this as the left division operator '\' implements the least-squares solution:

```
X = A \ B
```

Remarks By weighting the rows of $\boldsymbol{A}$ and the elements of $\vec{B}$, we can increase or decrease the importance of the corresponding frequency points. This corresponds to specifying the weighting function $W(\tilde{\omega})$ for the gridpoints $\tilde{\omega}_1$ tot $\tilde{\omega}_L$.

Conclusion By following the procedure above, we obtain estimates for q_n and hence for h_n, the values of our impulse response. Applying a window function as the last step will reduce the Gibbs phenomena on the resulting spectrum.

[11] $\boldsymbol{A}^T\boldsymbol{A}$ has in general a bad condition number and therefore numerically computing the inverse yields an inaccurate result.

6.5.2.2 Design Method 2: integral approach

Filter objective Let's start again by considering the variant of the L_2-error, and let's elaborate its definition:

$$\begin{aligned}\hat{\epsilon}_2 &= \left\| W(\tilde{\omega})\left(\hat{H}(\tilde{\omega}) - G(\tilde{\omega})\right)\right\|_2 \\ &= \sqrt{\frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega})\Big(\hat{H}(\tilde{\omega}) - G(\tilde{\omega})\Big)^2 \mathrm{d}\tilde{\omega}}\end{aligned}$$

Using (6.13) we can write:

$$\epsilon_2 = \sqrt{\frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega})\left(\sum_{n=0}^{\lfloor K\rfloor} q_n f((K-n)|\tilde{\omega}|) - G(\tilde{\omega})\right)^2 \mathrm{d}\tilde{\omega}}$$

with f respresenting cos (filter types I and II) or sin (filter types II and IV).

Our goal is to minimize this L_2-error. This is an unconstrained *least-squares* minimization problem. The minimum for ϵ_2 can be found by setting the partial derivatives for q_k equal to zero:

$$\begin{aligned}\frac{\partial}{\partial q_k}\left[\frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega})\left(\sum_{n=0}^{\lfloor K\rfloor} q_n f((K-n)|\tilde{\omega}|) - G(\tilde{\omega})\right)^2 \mathrm{d}\tilde{\omega}\right] &= 0 \\ \frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega})\, 2\left(\sum_{n=0}^{\lfloor K\rfloor} q_n f((K-n)|\tilde{\omega}|) - G(\tilde{\omega})\right)\frac{\partial}{\partial q_k}\left(\sum_{n=0}^{\lfloor K\rfloor} q_n f((K-n)|\tilde{\omega}|) - G(\tilde{\omega})\right)\mathrm{d}\tilde{\omega} &= 0 \\ \int_{-\pi}^{+\pi} W(\tilde{\omega})\left(\sum_{n=0}^{\lfloor K\rfloor} q_n f((K-n)|\tilde{\omega}|) - G(\tilde{\omega})\right) f((K-k)|\tilde{\omega}|)\,\mathrm{d}\tilde{\omega} &= 0\end{aligned}$$

The latter can be separated in a left-hand part that depends on q_n and a right-hand part that does not depend on q_n:

$$\begin{aligned}\int_{-\pi}^{+\pi} W(\tilde{\omega})\sum_{n=0}^{\lfloor K\rfloor} q_n f((K-n)|\tilde{\omega}|) f((K-k)|\tilde{\omega}|)\,\mathrm{d}\tilde{\omega} &= \int_{-\pi}^{+\pi} W(\tilde{\omega})G(\tilde{\omega})f((K-n)|\tilde{\omega}|)\,\mathrm{d}\tilde{\omega} \\ \sum_{n=0}^{\lfloor K\rfloor} q_n\int_{-\pi}^{+\pi} W(\tilde{\omega}) f((K-n)|\tilde{\omega}|) f((K-k)|\tilde{\omega}|)\,\mathrm{d}\tilde{\omega} &= \int_{-\pi}^{+\pi} W(\tilde{\omega})G(\tilde{\omega})f((K-n)|\tilde{\omega}|)\,\mathrm{d}\tilde{\omega} \qquad (6.18)\end{aligned}$$

Now, if we define[12]

$$\begin{aligned}A(k,n) &= \frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega}) f((K-n)|\tilde{\omega}|) f((K-k)|\tilde{\omega}|)\,\mathrm{d}\tilde{\omega} \\ b(k) &= \frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega})G(\tilde{\omega}) f((K-k)|\tilde{\omega}|)\,\mathrm{d}\tilde{\omega} \qquad (6.19)\end{aligned}$$

then we can rewrite (6.18) as:

$$\sum_{n=0}^{\lfloor K\rfloor} q_n A(k,n) = b(k)$$

Considering we have such an equation for every $k = 0, 1, 2, \ldots, \lfloor K\rfloor$, this yields a linear system of equations that can be written in matrix form:

$$A\vec{Q} = \vec{B}$$

[12] We reintroduce the factor $1/2\pi$ for it will come in handy, later on. You can safely omit it (in both definitions), if you like.

with

$$\boldsymbol{A} = \begin{bmatrix} A(0,0) & A(0,1) & A(0,1) & \cdots & A(0,\lfloor K\rfloor) \\ A(1,0) & A(1,1) & A(1,1) & \cdots & A(1,\lfloor K\rfloor) \\ A(2,0) & A(2,1) & A(2,1) & \cdots & A(2,\lfloor K\rfloor) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A(\lfloor K\rfloor,0) & A(\lfloor K\rfloor,1) & A(\lfloor K\rfloor,1) & \cdots & A(\lfloor K\rfloor,\lfloor K\rfloor) \end{bmatrix}$$

and

$$\vec{B} = \begin{bmatrix} b(0) \\ b(1) \\ b(2) \\ \vdots \\ b(\lfloor K\rfloor) \end{bmatrix} \qquad \vec{Q} \qquad = \begin{bmatrix} q(0) \\ q(1) \\ q(2) \\ \vdots \\ q(\lfloor K\rfloor) \end{bmatrix}$$

This is a square system that can be easily solved.[13]

Special properties of A Moreover it turns out that the matrix $\boldsymbol{A}$ is a special matrix that allows calculating it's entries efficiently (and allows an efficient solution of the system $\boldsymbol{A}\vec{Q} = \vec{B}$).

Indeed:

$$\begin{aligned} A(k,n) &= \frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega}) f((K-n)\,|\tilde{\omega}|) f((K-k)\,|\tilde{\omega}|)\,\mathrm{d}\tilde{\omega} \\ &\Big\downarrow f((K-n)\,|\tilde{\omega}|) f((K-k)\,|\tilde{\omega}|) = \frac{1}{2}\cos((k-n)\tilde{\omega}) \pm \frac{1}{2}\cos((2K-k-n)\tilde{\omega}) \\ &= \frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega}) \left(\frac{1}{2}\cos((k-n)\tilde{\omega}) \pm \frac{1}{2}\cos((2K-k-n)\tilde{\omega})\right) \mathrm{d}\tilde{\omega} \\ &= \frac{1}{2}A_1(k,n) \pm \frac{1}{2}A_2(k,n) \end{aligned}$$

with the plus sign in case $f = \cos$ and the minus sign in case $f = \sin$ and with

$$\begin{aligned} A_1(k,n) &= \frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega}) \cos((k-n)\tilde{\omega})\,\mathrm{d}\tilde{\omega} \\ A_2(k,n) &= \frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega}) \cos((2K-k-n)\tilde{\omega})\,\mathrm{d}\tilde{\omega} \end{aligned}$$

We can even further simplify the notation:

$$\begin{aligned} A_1(k,n) &= a(k-n) \\ A_2(k,n) &= a(2K-k-n) \end{aligned}$$

with:

$$a(m) = \frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega}) \cos(m\tilde{\omega})\,\mathrm{d}\tilde{\omega}$$

Please, notice: $a(m) = a(-m)$.

[13] For a type III filter, the last column and last row of $\boldsymbol{A}$ will contain all zeros. As we know that in that case $b(\lfloor K\rfloor) = 0$ and $q(\lfloor K\rfloor)$ is zero, we can reduce the system, by removing the last row and column from $\boldsymbol{A}$, and the last element of both $\vec{B}$ and $\vec{q}$.

Using this notation, we can decompose matrix $\boldsymbol{A}$ into $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ as follows:

$$\boldsymbol{A} = \frac{\boldsymbol{A}_1 \pm \boldsymbol{A}_2}{2}$$

with:

$$\boldsymbol{A}_1 = \begin{bmatrix} a(0) & a(1) & a(2) & \cdots & a(\lfloor K \rfloor) \\ a(1) & a(0) & a(1) & \cdots & a(\lfloor K \rfloor - 1) \\ a(2) & a(1) & a(0) & \cdots & a(\lfloor K \rfloor - 2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a(\lfloor K \rfloor) & a(\lfloor K \rfloor - 1) & a(\lfloor K \rfloor - 2) & \cdots & a(0) \end{bmatrix}$$

$$\boldsymbol{A}_2 = \begin{bmatrix} a(2K - 0) & a(2K - 1) & a(2K - 2) & \cdots & a(2K - \lfloor K \rfloor) \\ a(2K - 1) & a(2K - 2) & a(2K - 3) & \cdots & a(2K - \lfloor K \rfloor + 1) \\ a(2K - 2) & a(2K - 3) & a(2K - 4) & \cdots & a(2K - \lfloor K \rfloor + 2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a(2K - \lfloor K \rfloor) & a(2K - \lfloor K \rfloor + 1) & a(2K - \lfloor K \rfloor + 2) & \cdots & a(2K - 2\lfloor K \rfloor) \end{bmatrix}$$

As one can see $\boldsymbol{A}_1$ is a symmetric Toeplitz matrix[14] and $\boldsymbol{A}_2$ is a (symmetric) Hankel matrix[15]. This fact allows to generate these matrices by only calculating $2\lfloor K \rfloor + 1$ entries, instead of $(\lfloor K \rfloor + 1)^2$ entries. In addition, efficient algorithms exist to solve $(\boldsymbol{A}_1 \pm \boldsymbol{A}_2)\, Q = B$ with a computational complexity of $\lfloor K \rfloor \log \lfloor K \rfloor$.

Let's take a closer look at at

$$a(m) = \frac{1}{2\pi} \int_{-\pi}^{+\pi} W(\tilde{\omega}) \cos(m\tilde{\omega})\, \mathrm{d}\tilde{\omega}$$

Because $W(\tilde{\omega})$ is an even function, we know that:

$$\frac{1}{2\pi} \int_{-\pi}^{+\pi} W(\tilde{\omega}) \sin(m\tilde{\omega})\, \mathrm{d}\tilde{\omega} = 0$$

and therefore:

$$a(m) = \frac{1}{2\pi} \int_{-\pi}^{+\pi} W(\tilde{\omega}) f(m\tilde{\omega})\, \mathrm{d}\tilde{\omega} + \mathrm{j} \underbrace{\frac{1}{2\pi} \int_{-\pi}^{+\pi} W(\tilde{\omega}) \sin(m\tilde{\omega})\, \mathrm{d}\tilde{\omega}}_{=0}$$

and therefore:

$$a(m) = \frac{1}{2\pi} \int_{-\pi}^{+\pi} W(\tilde{\omega})\, \mathrm{e}^{\mathrm{j}m\tilde{\omega}}\, \mathrm{d}\tilde{\omega}$$

In this equation, we recognize the definition of the inverse Discrete-time Fourier Transform! Therefore

$$a(m) \xrightarrow{\text{DtFT}} W(\tilde{\omega})$$

[14]Toeplitz matrix: a square matrix has constant primary diagonals
[15]Hankel matrix: a square matrix that has constant secondary diagonals

Let's take a closer look at at

$$b(k) = \frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega})G(\tilde{\omega})f((K-k)\,|\tilde{\omega}|)\,\mathrm{d}\tilde{\omega}$$

For type I and II filters, $f = \cos$, and therefore even, i.e.

$$b(k) = \frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega})G(\tilde{\omega})f((K-k)\tilde{\omega})\,\mathrm{d}\tilde{\omega}$$

Because $W(\tilde{\omega})$ and $W(\tilde{\omega})G(\tilde{\omega})$ are also even functions, we know that:

$$\frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega})G(\tilde{\omega})\sin((K-k)\tilde{\omega})\,\mathrm{d}\tilde{\omega} = 0$$

and therefore:

$$b(k) = \frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega})G(\tilde{\omega})f((K-k)\tilde{\omega})\,\mathrm{d}\tilde{\omega} + \mathrm{j}\underbrace{\frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega})G(\tilde{\omega})\sin((K-k)\tilde{\omega})\,\mathrm{d}\tilde{\omega}}_{=0}$$

and therefore:

$$b(k) = \frac{1}{2\pi}\int_{-\pi}^{+\pi} W(\tilde{\omega})G(\tilde{\omega})\,\mathrm{e}^{\mathrm{j}k\tilde{\omega}}\,\mathrm{d}\tilde{\omega}$$

In this equation, we recognize the definition of the inverse Discrete-time Fourier Transform! Therefore, for type I and II filters:

$$b(k) \xrightarrow{\text{DtFT}} W(\tilde{\omega})G(\tilde{\omega})$$

For type III and IV filters, we'll have to revert to using (6.19).

Conclusion By following the procedure above, we obtain estimates for q_n and hence for h_n, the values of our impulse response. Just shift the impulse response to make it causal. This will merely change our zero-phase filter into a linear-phase filter. Applying a window function as the last step will reduce the Gibbs phenomena on the resulting spectrum.

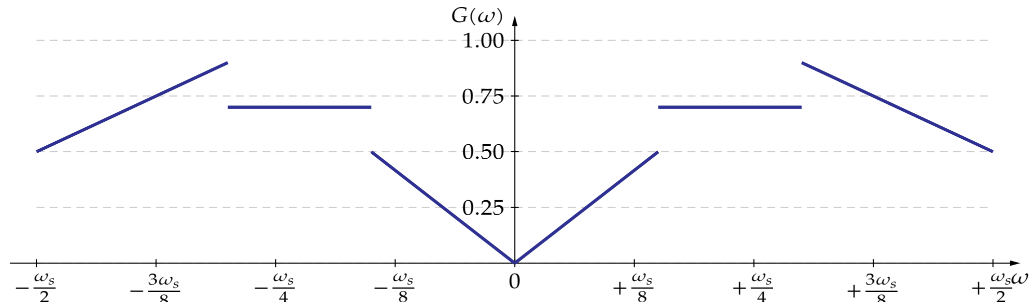
6.5.2.3 Example

Though arbitrary filter spectra can be fitted using the least-squares approach, we'll use a spectrum that's piecewise linear to keep it simple.

Let's design a 63-point filter that is optimal in the least-squares sense in approximating the following desired filter magnitude spectrum:

$$G(\omega) = \begin{cases} \frac{10}{3\omega_s}\,|\omega| & \text{if } |\omega| < 0.15\omega_s \\ 0.7 & \text{if } 0.15\omega_s \le |\omega| < 0.3\omega_s \\ 0.9 - \frac{2}{\omega_s}(|\omega| - 0.3\omega_s) & \text{if } 0.3\omega_s \le |\omega| < \frac{\omega_s}{2} \end{cases}$$

Graphically, this magnitude spectrum function looks like this:



The hard way

Let's start by defining $G(\tilde{\omega})$:

```
function y = gnormfreq( w )
  y = zeros(size(w));
  y+= ( abs(w) < 0.3*pi ) .* (5/3*abs(w)/pi);
  y+= ( abs(w) >= 0.3*pi & abs(w) < 0.6*pi ) * 0.7;
  y+= ( abs(w) >= 0.6*pi ) .* (0.9 - (abs(w)-0.6*pi)/pi);
end
```

Let's use this function to design an $2M+1$-point filter to comply with a L-point grid.

```
M = 50;
N = 2*M+1;
L = 200;
wk = linspace( 0, pi, L )';
```

And now, let's calculate the matrices A and $\vec{B}$:

```
A = cos( wk * (M:-1:0) );
B = gnormfreq( wk );
```

Did you expect it to be so simple?

As one cannot check enough, let's check on the shape of $\vec{B}$ to see whether it corresponds to the original magnitude spectrum.

```
plot( wk, B );
```

Now, we can solve for $\vec{Q}$:

```
Q = A \ B;
```

Starting from Q, we can generate the impulse response (and make it causal at the same time):

```
h = zeros( N, 1 );
h(M+1) = Q(M+1);
for i = 1:M
  h(i) = h(N+1-i) = Q(i) / 2;
end
```

Let's take a look at the result:

```
plot( h );
ylabel( "h[n]" );
xlabel( "n" );
title( "Impulse␣response" );
```

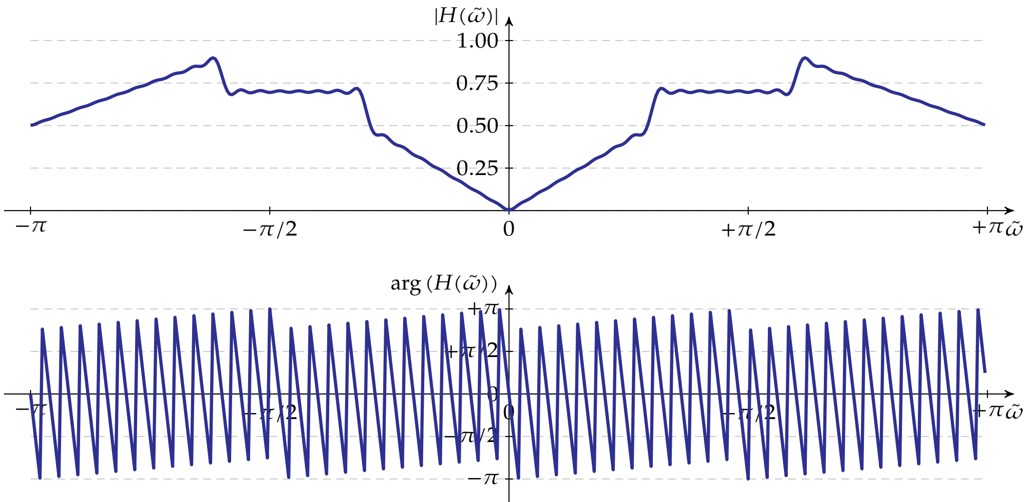
And, let's check the resulting spectrum

```
H = shift( fft(h), floor(N/2) );
```

and plot it:

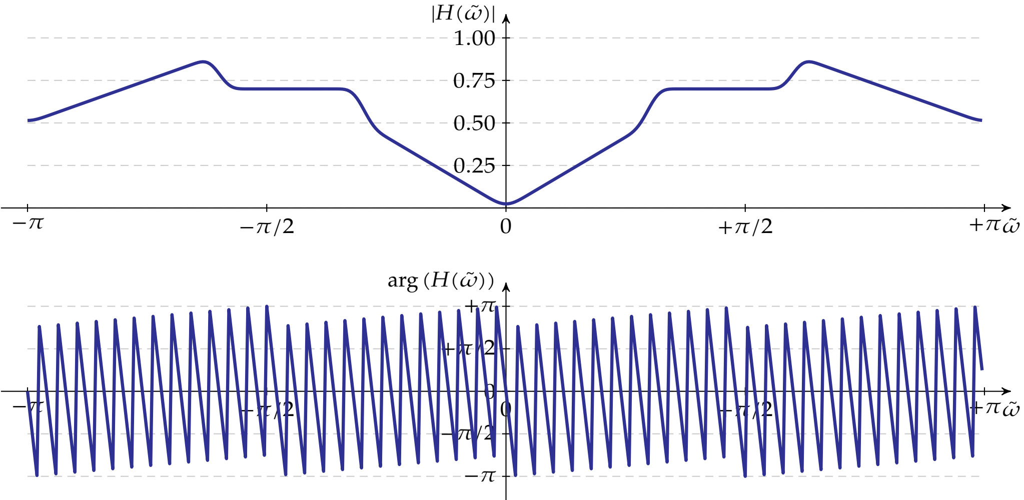
```
subplot(2,1,1);
plot( abs(H) );
ylabel( "|H(w)|" );
subplot(2,1,2);
plot( angle(H) );
ylabel( "angle(H(w))" );
xlabel( "w" );
title( "Resulting␣FFT␣spectrum" );
```

To give yourself a little exercise, perform some zero-padding and check the resulting spectrum. The result should look like this:



Then, apply a window function and perform zero-padding to see the effect.

The result after applying a Blackman window should look like this:



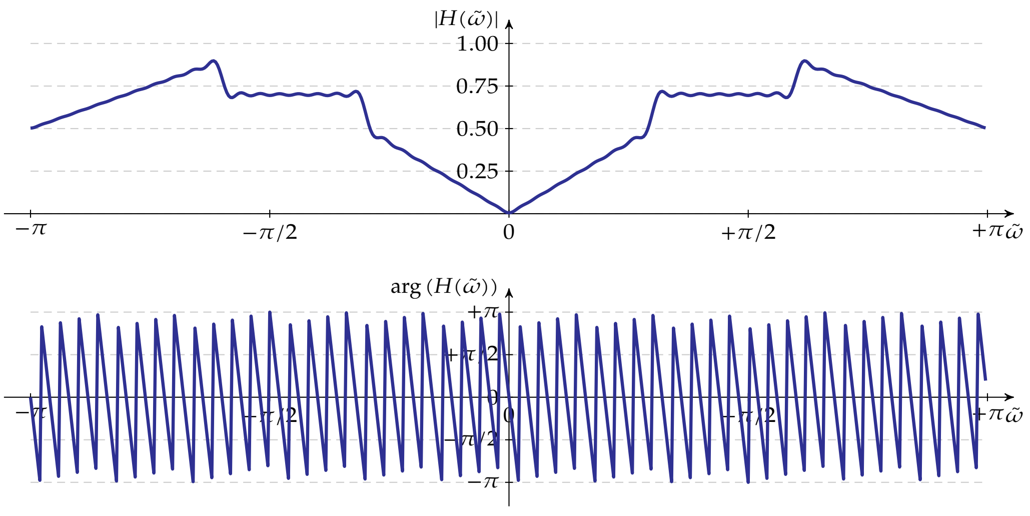
The easy way

Octave has standard support for designing piecewise linear FIR filters using the integral approach by the function `firls`. Check the help facility for some more information on how to use this function.

After heaving read the documentation, it should be very obvious that the following line does the job:

```
M = 100;
h = firls(M, [ 0 0.3 0.3 0.6 0.6 1.0 ],
             [ 0 0.5 0.7 0.7 0.9 0.5 ] );
```

Plotting the impulse response and the resulting spectrum should be a piece of cake to you by now. The resulting spectrum should look like this:



Exercises

Some exercises on least-squares optimal FIR-filter design

Exercise 6.5.2.3-1: Design a FIR filter that implements a transfer function with the following spectral magnitude using the least squares optimal design method:

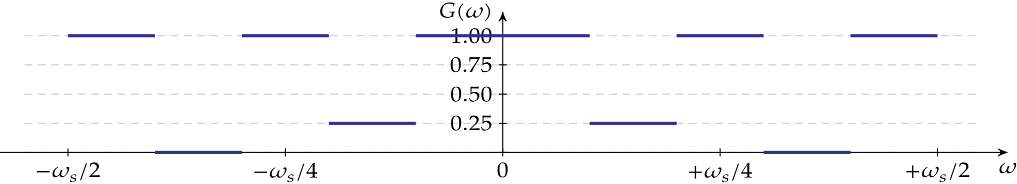
$$G(\omega) = \begin{cases} 1 - \frac{|\omega|}{\omega_s} & \text{if } |\omega| < 0.2\omega_s \\ 0.8 & \text{if } 0.2\omega_s \le |\omega| < 0.25\omega_s \\ 0.8 - \frac{4}{\omega_s}(|\omega| - 0.25\omega_s) & \text{if } 0.25\omega_s \le |\omega| < 0.3\omega_s \\ 0.2 & \text{if } 0.3\omega_s \le |\omega| \end{cases}$$

Do this

1. using the grid method (using Octave)
2. using the integral method (with Octave's `firls`-method)

Check the result using the DtFT (i.e., use a zero-padded FFT).
Try using an appropriate window function to milden the Gibbs effect.

Exercise 6.5.2.3-2: Design a FIR filter that implements a transfer function with the following spectral magnitude using the least squares optimal design method:



Do this

1. using the grid method (using Octave)
2. using the integral method (with Octave's `firls`-method)

Check the result using the DtFT (i.e., use a zero-padded FFT).
Try using an appropriate window function to milden the Gibbs effect.

6.5.2.4 Strengths and weaknesses

The least-squares design methods allow optimally approximating a desired magnitude spectrum. A weighting function enabled differentiating on the desired accuracy as a function of frequency. However, the actual error is not fully controllable.

We'll tackle this issue in the Parks-McClellan design method.

6.5.3 Parks-McClellan / Remez-exchange algorithm

In the previous section (on least-squares design) we tried to minimize the L_2 error. The Parks-McClellan filter design method will try to minimize the L_∞ error.

6.5.3.1 Design method

Consider a desired center-symmetrical magnitude spectrum $G(\tilde{\omega})$. Our goal is to find coefficients h_n defining $h[n]$ and $H(\omega)$ according to (6.12) such that the L_∞ error

$$\epsilon_\infty = \|W(\tilde{\omega})(|H(\tilde{\omega})| - G(\tilde{\omega}))\|_\infty$$

is minimal. W is a center-symmetrical weighting function defined on $[-\pi, +\pi]$.

The L_∞ norm can be rewritten as:

$$\epsilon_\infty = \max_{\tilde{\omega}} \left(W(\tilde{\omega})(H(\tilde{\omega}) - G(\tilde{\omega})) \right)$$

In English: we're looking for filter coefficients h_n (or q_n) such that $H(\tilde{\omega})$ is an approximation for which the maximum error (and this error may occur at a single or at multiple frequencies) on the considered frequency range is minimal.

Polynomial frequency response In section 6.5.1 on page 133 we've proven that the spectrum of a symmetric or antisymmetric impulse response can be written as:

$$|H(\tilde{\omega})| = \sum_{n=0}^{\lfloor K \rfloor} q_n f((K-n)\tilde{\omega})$$

with, depending on the filter type, appropriate q_i according to Table 6.1 on page 138.

Equation (6.9) can be further rewritten as:

$$|H(\tilde{\omega})| = \sum_{n=0}^{\lfloor K \rfloor} r_n f^n(\tilde{\omega})$$

by repeatedly applying

$$\cos(n\alpha) = 2\cos((n-1)\alpha)\cos\alpha - \cos((n-2)\alpha)$$

in the case $f = \cos$, and

$$\sin(n\alpha) = 2\sin((n-1)\alpha)\cos\alpha - \sin((n-2)\alpha)$$
$$\cos^2\alpha = 1 - \sin^2\alpha$$

in the case $f = \sin$.

Repeatedly applying the above recursion formulae, the values r_n can be derived from q_n and vice versa.

Limiting $\tilde{\omega}$ to the interval $[0, \pi]$ allows substituting $x = f(\tilde{\omega})$, because this is a bijective relationship. In this way, we can even further simplify:

$$|H(x)| = \sum_{n=0}^{\lfloor K \rfloor} r_n x^n \tag{6.40}$$

Hence, the frequency response is a polynomial in x.

Chebyshev's Alternation Theorem Consider an N-th order polynomial $H(x)$[16]:

$$H(x) = \sum_{n=0}^{N} r_n x^n$$

defined on a closed real interval $X = [x_{lo}, x_{lo}]$.

Let $G(x)$ be a desired function of x that's continuous on X, and let $W(x)$ be a positive weighting function of x that's also continuous on X. Using these functions we can define:

$$E(x) = W(x)(H(x) - G(x)) \tag{6.41}$$

One can prove that a unique and optimal N-th order approximation polynomial $\tilde{H}(x)$ exists that minimizes the L_∞ norm of $E(x)$.

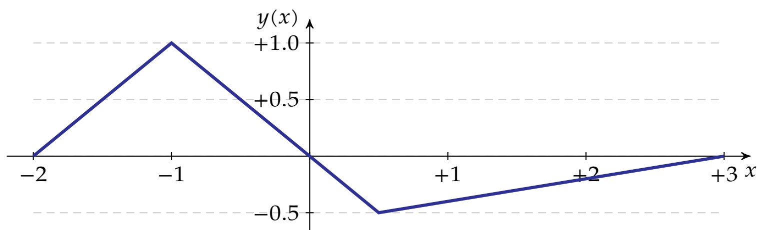
$\tilde{H}(x)$ is the optimal N-th order polynomial if $E(x)$ shows at least $N + 2$ consecutive, equal and maximal *alternations*, i.e. $N + 2$ values $x_k \in X, k = 0, 1, 2, \dots, N + 1$ exist with $x_0 < x_1 < x_2 < \dots < x_{N+1}$ such that

$$E(x_k) = -E(x_{k+1}) = \pm \|E(x)\|_\infty, \qquad k = 0, 1, 2, \dots, L$$

■

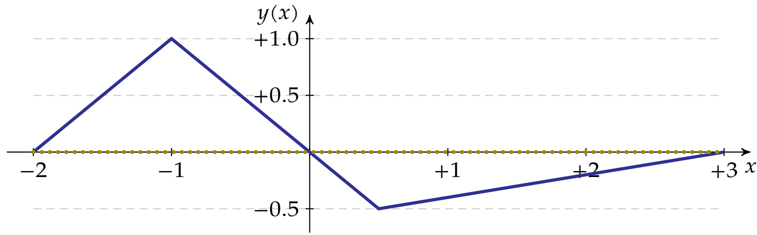
This is quite a complex theorem at first sight.[17] Let's make it a little-bit more visual with an example.

Consider the function in the graph below:



Let's try to find a good 1st-order polynomial (i.e. a linear function) to approximate this function on the range $[-2, 3]$. To keep things simple, we assume no weighting (i.e. $W(x) = 1$). Our first idea is to approximate this function by $y = 0$.

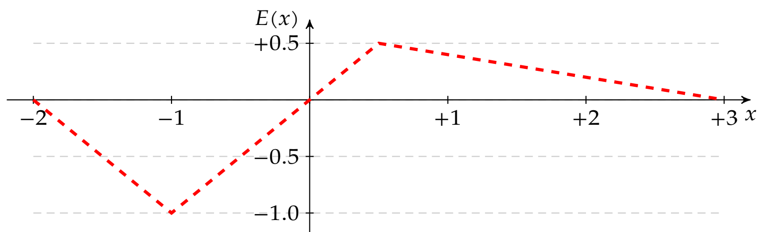
The solution (indicated using a dotted line) looks like this:



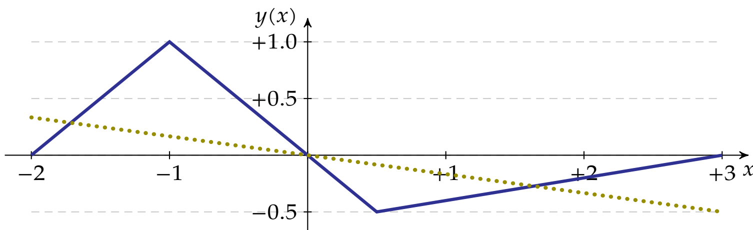
[16] We explicitly omit the absolute value stripes here, because the Chebyshev theorem doesn't require the function values to be positive.

[17] In addition, proving this theorem is not trivial. Even Chebyshev didn't manage to confirm his theorem by proof.

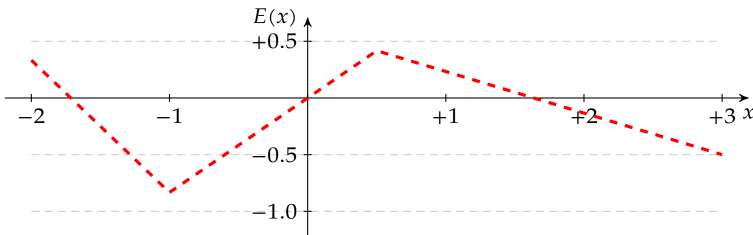
The error function $E(x)$ (depicted below) shows 2 non-equal and non-maximal alternations. However, we need 3 equal and maximal alternations to be sure that our approximation is the best possible approximation. Therefore, this is not the best possible approximation.



Our second idea is to try $y = -1/6x$. The solution (indicated using a dotted line) looks like this:

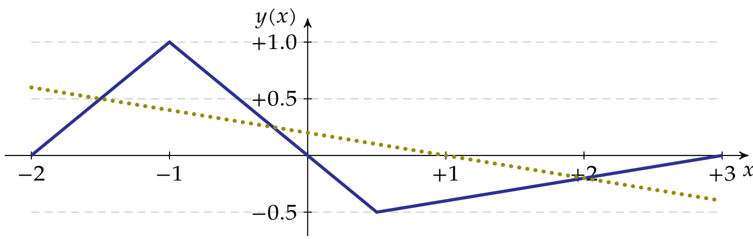


The error function $E(x)$ below again shows that our proposal is not the best one. We have four alternations, but no 3 consecutive of them are equal and maximal.

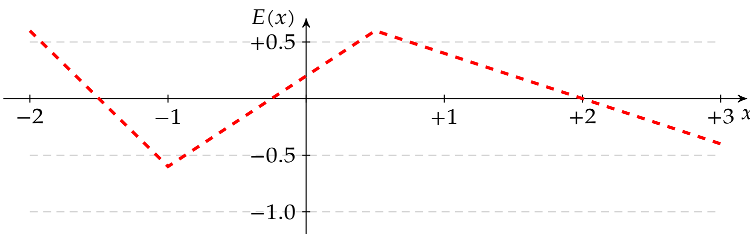


You'll find that finding an optimal solution "at first sight" is quite difficult even in very simple cases, like the one we're considering right now.

Luckily there is a systematic way to find the solution. This is the so-called *Remez-exchange* algorithm. Using this algorithm, we obtained the optimal approximation $y = -1/5x + 1/5$ as shown below:



The corresponding error function has been depicted below. As you can see it exhibits 3 consecutive equal and maximal alternations for $x_k = -2, -1, 0.5$.



Remember:

$$|H(x)| = \sum_{n=0}^{\lfloor K \rfloor} r_n x^n$$

The Remez-exchange algorithm is based on the improvement of the $\lfloor K \rfloor$-th order polynomial solution template $|H(x)|$. The template is improved using the idea that the solution has $\lfloor K \rfloor + 2$ consecutive points x_k for which the weighted error δ is equal and alternates.

Let's start by writing down this idea for the first point x_1 by starting from (6.41)[18]:

$$\begin{aligned} W(x_1)(\,|H(x_1)| - |G(x_1)|\,) &= \delta \\ |H(x_1)| &= |G(x_1)| + \delta / W(x_1) \end{aligned}$$

Taking into account the alternation for every subsequent point, this leads to the following equation system:

$$\begin{cases} |H(x_1)| = |G(x_1)| - (-1)^1 \delta / W(x_1) \\ |H(x_2)| = |G(x_2)| - (-1)^2 \delta / W(x_2) \\ |H(x_3)| = |G(x_3)| - (-1)^3 \delta / W(x_3) \\ \vdots \\ \left|H(x_{\lfloor K \rfloor+2})\right| = \left|G(x_{\lfloor K \rfloor+2})\right| - (-1)^{\lfloor K \rfloor+2} \delta / W(x_{\lfloor K \rfloor+2}) \end{cases}$$

Elaborating $|H(x)|$ using (6.40), we can rearrange this equation system and rewrite it in matrix form:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{\lfloor K \rfloor} & (-1)^1 / W(x_1) \\ 1 & x_2 & x_2^2 & \cdots & x_2^{\lfloor K \rfloor} & (-1)^2 / W(x_2) \\ 1 & x_3 & x_3^2 & \cdots & x_3^{\lfloor K \rfloor} & (-1)^3 / W(x_3) \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_{\lfloor K \rfloor+2} & x_{\lfloor K \rfloor+2}^2 & \cdots & x_{\lfloor K \rfloor+2}^{\lfloor K \rfloor} & (-1)^{\lfloor K \rfloor+2} / W(x_{\lfloor K \rfloor+2}) \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ \vdots \\ r_{\lfloor K \rfloor} \\ \delta \end{bmatrix} = \begin{bmatrix} |G(x_1)| \\ |G(x_2)| \\ |G(x_3)| \\ \vdots \\ \left|G(x_{\lfloor K \rfloor+2})\right| \end{bmatrix} \quad (6.43)$$

This is a $\lfloor K \rfloor + 2$-equation system with $\lfloor K \rfloor + 2$ unknowns and hence can be easily solved. Let's call this the "Remez system".

[18]We now again use the absolute value stripes, because we are again dealing with amplitude spectra.

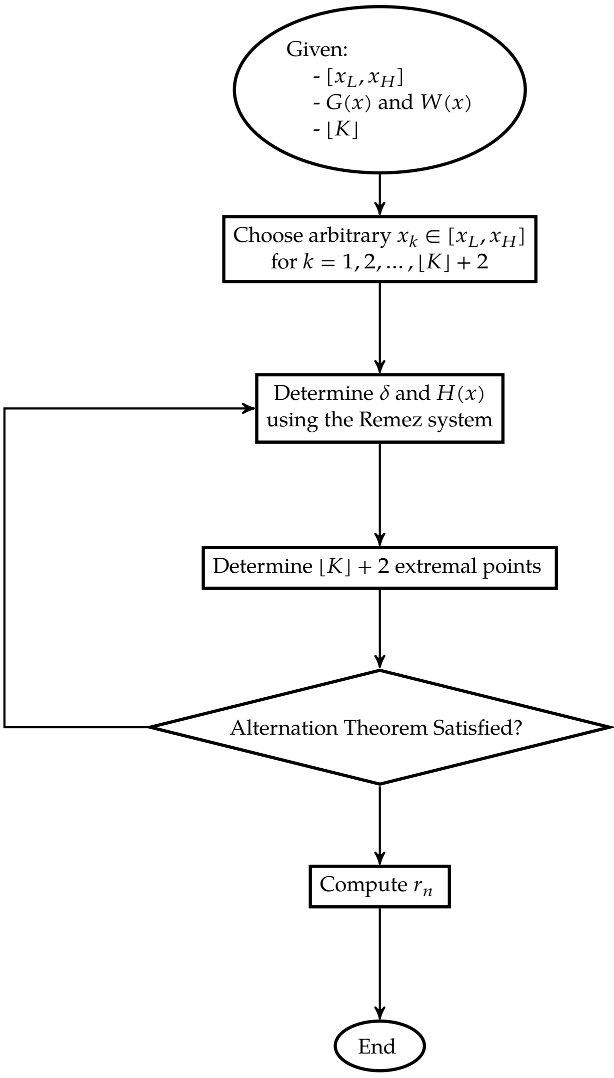


Figure 6.4: Simplified Flowchart of the Remez-exchange algorithm

Of course, the question remains: how do we know which values to use for x_k? Well, that brings us to the “exchange” part of the algorithm. Remez proposed to take $\lfloor K \rfloor + 2$ arbitrary points x_k, solve (6.43) and then find the extremities of $E(x)$ and have them replace the original points. This leads to the Remez-exchange algorithm of Figure 6.4.

In reality, the Remez-exchange algorithm will not solve (6.43) to calculate $H(x)$. Instead, only δ will be calculated and then $H(x)$ can be determined using Lagrange interpolation. An alternative strategy is to replace the Lagrange interpolation by interpolation to an FFT grid, apply an inverse FFT, zero-padding and a forward FFT to find the extremal frequencies on a higher-resolution grid.

The Remez-exchange algorithm has been implemented by Parks and McClellan with the objective to design linear-phase FIR filters.

6.5.3.2 Example

OCTAVE/MATLAB have built-in support for designing filters using the Parks-McClellan/Remez-exchange technique by means of the function `remez`.

Extensive design

Let's test this facility by designing a band-stopfilter according to the following specs:

Parameter	Value
$\tilde{\omega}_{PL}$	0.200π
$\tilde{\omega}_{SL}$	0.300π
$\tilde{\omega}_{SH}$	0.500π
$\tilde{\omega}_{PH}$	0.700π
R_P	± 1 dB
R_S	-40 dB

Designing filters using the `remez` function requires some feeling for the stubbornness of the algorithm.

Here are some typical problems that arise, and the tricks that may be used to solve them:

Problem	Solution
Bumps in the transition bands	- Shrink the transition bands
	- Specify the transition bands
	- Change the filter length
Convergence failure	- Reduce $\lfloor K \rfloor$
	- Shrink the transition bands
Insufficient extremals	- Increase the frequency grid oversampling
	- Shrink the transition bands
	- Specify the transition bands
Too many extremals	- Reduce $\lfloor K \rfloor$
Passband ripple too large	- Extend transition band
	- Increase passband weight
	- Increase $\lfloor K \rfloor$
Stopband ripple too large	- Exend transition band
	- Increase stopband weight
	- Increase $\lfloor K \rfloor$

We'll encounter some of these problems during the design. When having to choose between 'specifying the transition bands' and 'shrink the transition bands' (i.e. extend the stopbands), I'd go for the latter.

Let's start by specifying the vectors with corner frequencies, corresponding corner amplitudes, and band weights:

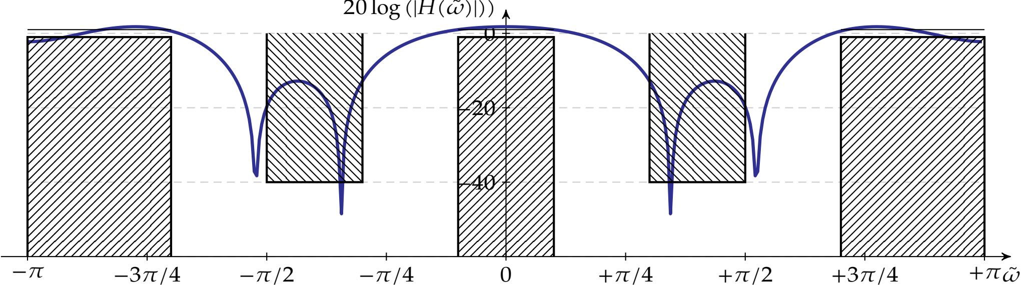
```
f = [ 0.000 0.200 0.300 0.500 0.700 1.000 ];
a = [ 1.000 1.000 0.000 0.000 1.000 1.000 ];
w = [ 1.000       1.000       1.000       ];
```

Designing a filter using the `remez` function is as simple as:

```
b = remez( 8, f, a, w );
```

Draw the logarithmic magnitude plot of this filter's spectrum yourself (using zero-padding and the FFT).

The result will look like this:

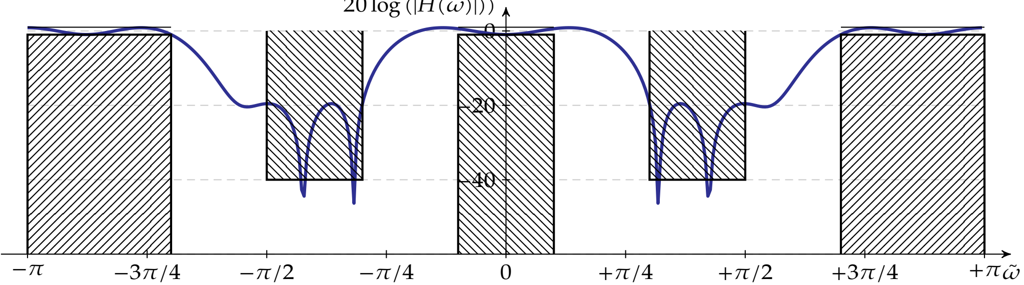


The hatched regions indicate the forbidden regions for the magnitude spectrum.

Apparently the ripple is not within spec. The table above makes three suggestions. Extending the transition band is not an option, because this would violate the specs. Increasing weights will not be effective as the ripple in all bands is too large. Changing weights will only allow influencing the distribution of the ripple. Therefore, let's go for an increase of $\lfloor K \rfloor$.

```
b = remez( 16, f, a, w );
```

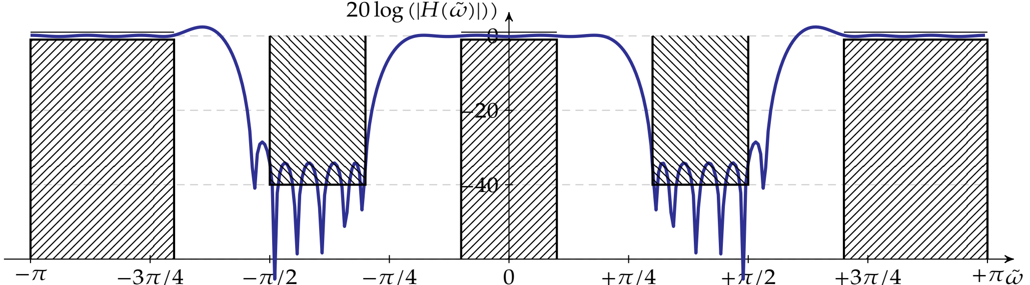
The corresponding newly designed filter function looks like this:



We're coming close, but the stopband ripple is still way out of spec. Let's again increase the order.

```
b = remez( 32, f, a, w );
```

This yields:

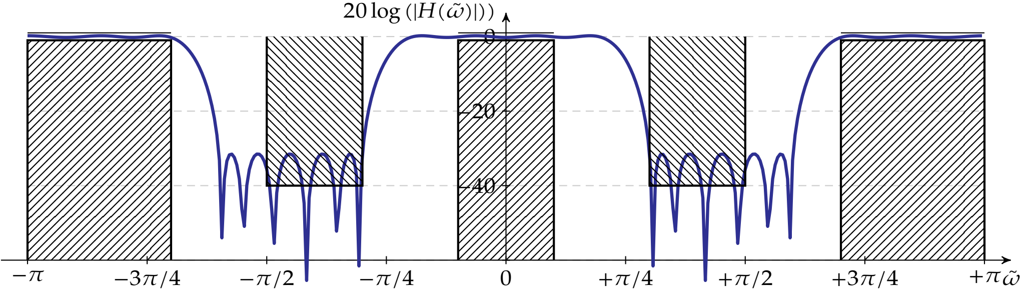


This looks quite a bit better. We could consider fiddling with the weight function to exchange more passband ripple for less stopband ripple. However, we need to take care of the upward bump in the transition from the highpassband to the stopband.

Therefore, let's try to shrink the upper transition band.

```
f = [ 0.000 0.200 0.300 0.600 0.700 1.000 ];
    % a = [ 1.000 1.000 0.000 0.000 1.000 1.000 ];
    % w = [ 1.000       1.000       1.000       ];
b = remez( 32, f, a, w );
```

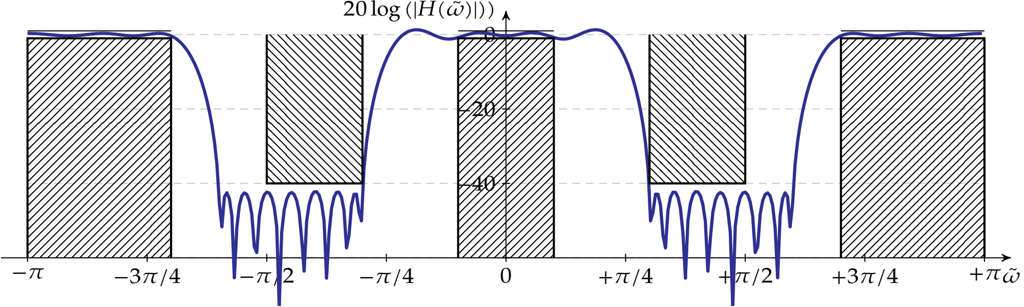
Let's check the result:



Hmmm... this is not bad. Let's fiddle with the weight function.

```
    % f = [ 0.000 0.200 0.300 0.600 0.700 1.000 ];
    % a = [ 1.000 1.000 0.000 0.000 1.000 1.000 ];
w = [ 1.000       5.000       1.000       ];
b = remez( 32, f, a, w );
```

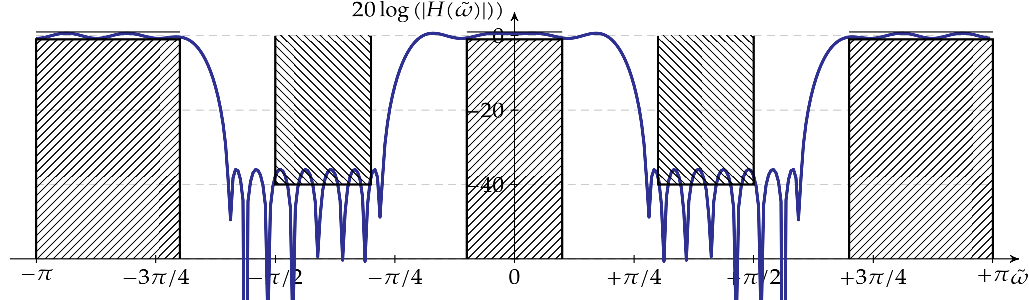
Here's the result:



Let's reduce the transition-band bump a little bit more. We do this by shrinking the transition band:

```
f = [ 0.000 0.200 0.275 0.600 0.700 1.000 ];
    % a = [ 1.000 1.000 0.000 0.000 1.000 1.000 ];
    % w = [ 1.000       5.000       1.000       ];
b = remez( 32, f, a, w );
```

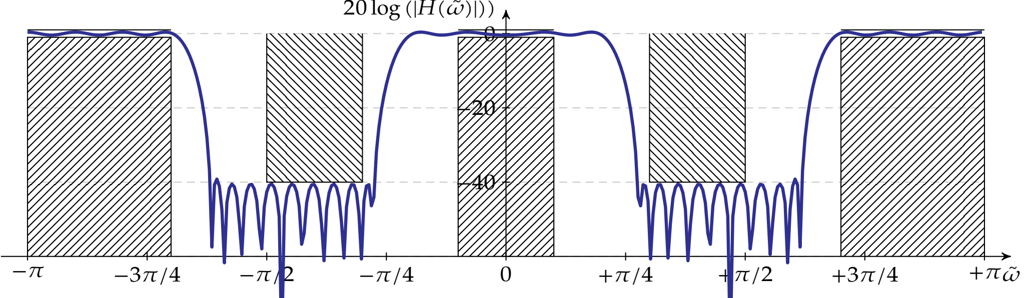
This yields:



Again, we're out of spec. Let's correct by increasing $\lfloor K \rfloor$ tot 40.

```
b = remez( 40, f, a, w );
```

This yields:



Perfect!

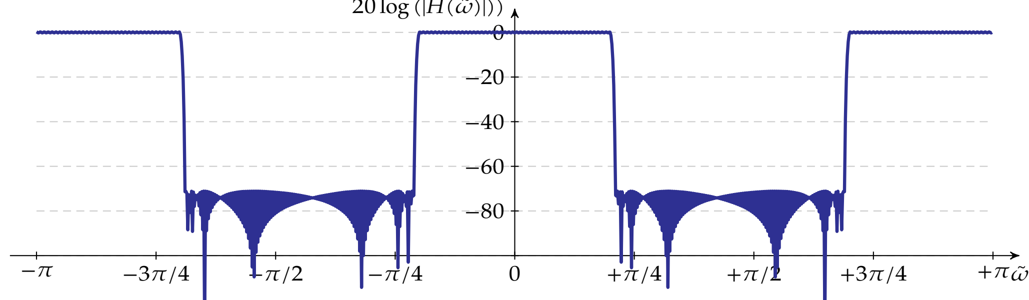
In the above, we increased and changed parameters in a very coarse way. In reality, more fine-grained optimization is appropriate. Especially every increase in $\lfloor K \rfloor$ is the crucial factor, because it implies an increase in computation effort, that you will pay at the counter when leaving the shop.

Particular cases

One can design high-order filters with very tight transition bands. E.g.,

```
f = [ 0.000 0.200 0.210 0.690 0.700 1.000 ];
a = [ 1.000 1.000 0.000 0.000 1.000 1.000 ];
w = [ 1.000       1.000       1.000       ];
b = remez( 512, f, a, w );
```

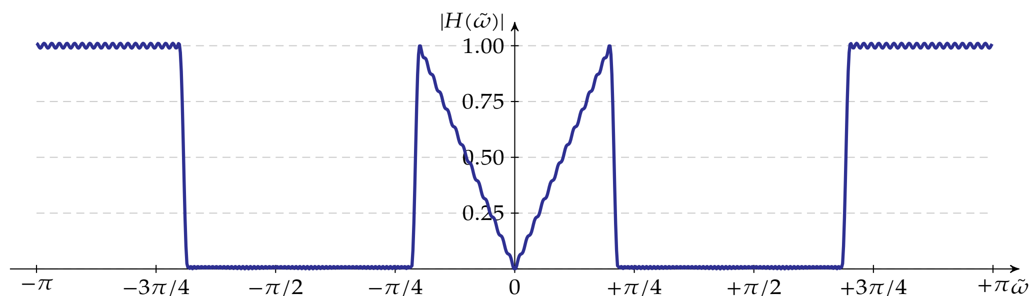
leading to:



One can also design filters with non-flat bands. E.g.,

```
f = [ 0.000 0.200 0.215 0.685 0.700 1.000 ];
a = [ 0.000 1.000 0.000 0.000 1.000 1.000 ];
w = [ 1.000       1.000       1.000       ];
b = remez( 256, f, a, w );
```

leading to:



By implementing your own Remez-exchange algorithm, you even could build arbitrary filters.

Anyway, when experimenting a little-bit further, you will find out that the convergence of the `remez` function is often a problem. In general, building filters with more than a few hundred taps, will get you some convergence problems. In addition, when comparing Octave to Matlab, you'll find the latter to be considerable more robust. Anyway, take a good generic piece of advice: *always* check your end result.[19]

Exercises
Some exercises on using the Remez-exchange optimal design method

Exercise 6.5.3.2-1: Design a FIR band-stop filter with as few taps as possible (using the Remez-exchange method) that implements the following filter function:

[19]In the extensive example that we've treated, you will even have observed convergence failure notices. As long as the end-result looks good, all's well.

Parameter	Value
$\tilde{\omega}_{PL}$	0.200π
$\tilde{\omega}_{SL}$	0.300π
$\tilde{\omega}_{SH}$	0.560π
$\tilde{\omega}_{PH}$	0.740π
R_P	± 0.5 dB
R_S	-20 dB

Verify your filter's transfer function spectrum using the DtFT.

Exercise 6.5.3.2-2: Design a FIR band-pass filter with as few taps as possible (using the Remez-exchange method) that implements the following filter function:

Parameter	Value
$\tilde{\omega}_{SL}$	0.200π
$\tilde{\omega}_{PL}$	0.300π
$\tilde{\omega}_{PH}$	0.500π
$\tilde{\omega}_{SH}$	0.700π
R_P	± 1 dB
R_S	-40 dB

Verify your filter's transfer function spectrum using the DtFT.

6.5.3.3 Strengths and weaknesses

The Remez-exchange (or Parks-McClellan) design method allows designing filters with specified passband and stopband ripple (or stopband attenuation), and specified transition bands, that provide optimal L_∞ ripple given a particular filter order.

One has to iterate to find the correct order when given desired values of the passpand and stopband ripple (or use the `remezord` function in MATLAB). It requires some experience to make a good filter using the Parks-McClellan technique.

6.6 Conclusion

The design methods we've treated all have their particular weaknesses and strengths. Table 6.2 on the next page summarizes the most important ones. In this table, the following abbreviations are used:

- l, m, and h for low, medium and high,

Design method	Impulse response oriented	Frequency response oriented	Linear phase	Gibbs sensitivity	Transition band control	Ripple control	Flexibility	Design procedure convergence	Design procedure complexity	Design procedure computation time
Zero placement		•	n	l	-	-	-	+	l	l
Impulse response invariance	•		n	m	-	-	+	+	l	l
Frequency sampling design method		•	y	h	-	-	+	+	m	l
Least-squares		•	y	m	-	-	+	±	m	m
Parks-McClellan		•	y	l	+	+	+	-	h	h

Table 6.2: Overview of the FIR-filter design methods

- *y* and *n* for yes and no,
- +, ± and - for good, medium and bad.

Chapter 7

IIR Filter Design

In this chapter, you will learn about:

- various ways to design discrete-time IIR filters,
- how to perform the actual design using OCTAVE or MATLAB,
- the advantages and drawbacks of these design methods.

After having read/studied this chapter, you are expected to be able to

- explain the various design methods,
- judge the weaknesses and the strengths of every design method,
- apply the various design methods, using pencil and paper and using a mathematical computer program like OCTAVE/MATLAB, when given a filter specification.

7.1 Introduction

Directly designing discrete-time IIR filters is quite a difficult job from a theoretical point of view. Therefore, in most cases, the design of a discrete-time IIR filter, starts from an analog filter design (for which the mathematics are well understood) that is afterwards converted to the discrete-time domain using a transformation method.

In this chapter we will discuss some of these methods:

- the pole-zero placement method
- the impulse invariance method
- the bilinear transformation method

In general these methods do not result in linear-phase IIR filters. But, if we process signals post-factum (in batch), there is a trick we can use to make linear-phase IIR filters, the so-called *reverse filtering* technique.

Many of the methods above have been implemented in mathematical toolboxes like the signal processing toolbox of MATLAB or OCTAVE. If all the magic is readily available, why bother studying the mathematical principles behind these techniques? The answer is the same as the one we've given on page 113: using tools without any knowledge about their nature or the circumstances in which to use them is most dangerous or leads at least to suboptimal use.

So, our goal is twofold:

- get to know the basic principles of the techniques,
- get a flavor of how to use these techniques in MATLAB/OCTAVE.

Of course we cannot treat all the available methods. You should be aware that there are a number of filter types and (corresponding) design methods that we will not treat:

- Prony's method
- Shank's method
- optimization-based design methods
- (statistical) Linear prediction filters (Yule Walker)
- ...

7.2 Pole-zero placement

A very simple way to design IIR filters is to place poles and zeros in the (complex) Z-plane. Remember, to create filter functions that correspond to a real impulse responses, poles and zeros need to be *real* or appear in *complex conjugate pairs*.

However, for every zero we place we need to add at least one pole as well, to ensure the causality of the filter. Stated differently: the degree of the transfer function's denominator must be higher than or equal to the degree of its numerator. This can be easily understood by realizing that after polynomial long-division, the transfer function of a causal filter can only contain terms with a factor of z^{-k}.

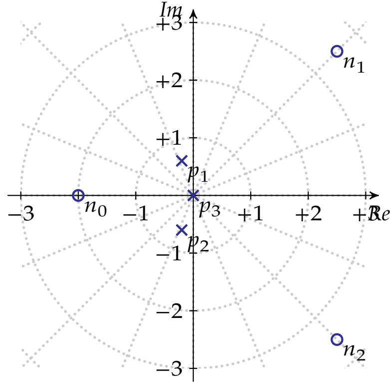
An additional requirement holds: all poles must be placed within the unit circle in order for the filter to be stable.

7.2.1 Arbitrary placement

Consider a simple example in which we arbitrarily place a number of poles and zeros.

$$\begin{aligned} n_0 &= -2 \\ n_{1,2} &= 2.5 \pm \mathrm{j}2.5 \\ p_{1,2} &= -0.2 \pm \mathrm{j}0.6 \\ p_3 &= 0 \end{aligned}$$

This pole-zero configuration has been depicted below:



Deriving the corresponding transfer function in the Z-plane is easy:

$$
\begin{aligned}
H(z) &= \frac{(z - n_0)\,(z - n_1)\,(z - n_2)}{(z - p_1)\,(z - p_2)\,(z - 0)} \\
&= \frac{(z - (-2))\,(z - (2.5 + j2.5))\,(z - (2.5 - j2.5))}{(z - (-0.2 + j0.6))\,(z - (-0.2 - j0.6))\,z} \\
&= \frac{(z - (-2))\left(z^2 - 5z + 12.5\right)}{\left(z^2 + 0.4z + 0.4\right)z} \\
&= \frac{z^3 - 3z^2 + 2.5z + 25}{z^3 + 0.4z^2 + 0.4z}
\end{aligned}
\tag{7.1}
$$

This leads directly to the direct and transposed form implementations of Figure 7.1 on the following page.

Exercises
Some exercises on the pole-zero placement method:

Exercise 7.2.1-1: Make a filter that realizes the following zeros and poles:

$$
\begin{aligned}
n_1 &= -1.5 \\
p_1 &= 0.1
\end{aligned}
$$

Use OCTAVE to make a graph displaying the transfer function's spectrum (Bode plot).

Exercise 7.2.1-2: Make a filter that realizes the following zeros and poles:

$$
\begin{aligned}
n_1 &= 3 \\
p_1 &= -2.5
\end{aligned}
$$

Use OCTAVE to make a graph displaying the transfer function's spectrum (Bode plot).

Exercise 7.2.1-3: Make a filter that realizes the following zeros and poles:

$$
\begin{aligned}
n_{1,2} &= -3 \pm j2 \\
p_0 &= -0.3 \\
p_{1,2} &= 0.5 \pm j0.8
\end{aligned}
$$

and Use OCTAVE to make a graph displaying the transfer function's spectrum (Bode plot).

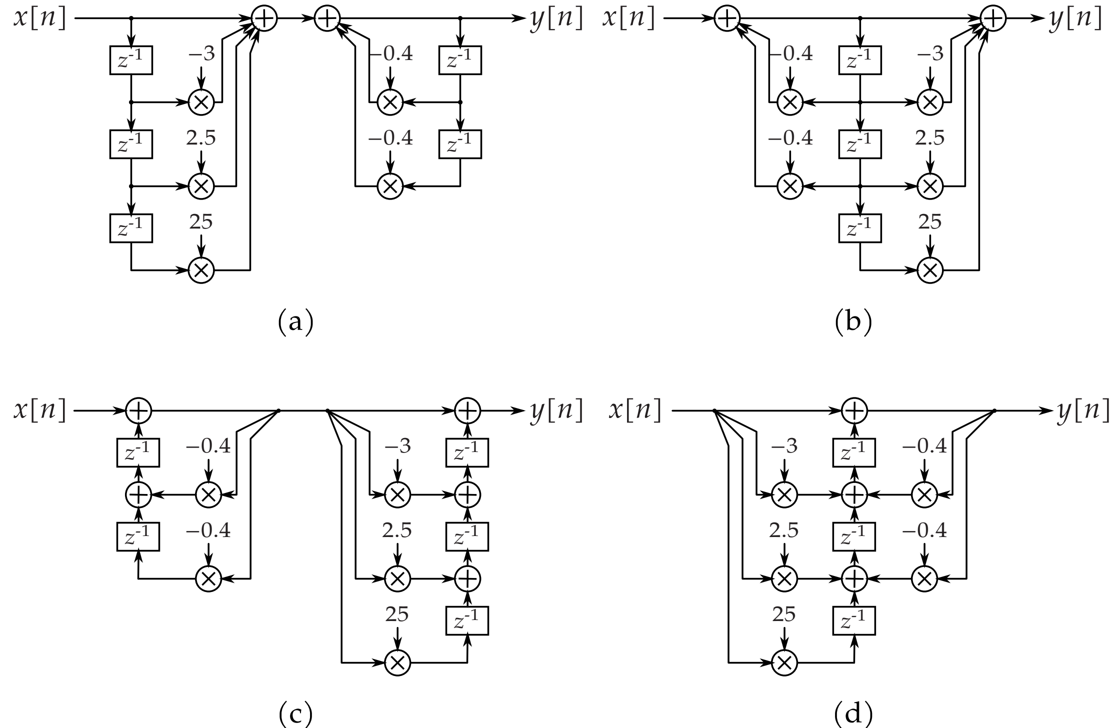


Figure 7.1: Implementation of (7.1): (a) direct form I, (b) direct form II, (c) transposed form I, and (d) transposed form II

7.2.2 Biquad sections

If a high-order transfer function has to be implemented, directly mapping the transfer function $H(z)$ onto a direct or transposed form may lead to stability problems. The reason for this is that small changes on a high-order polynomial's coefficients (the numerator and the denominator of the transfer function) may lead to considerable pole and zero errors because of coefficient rounding. If a pole close to the unit circle "by accident" falls out of the unit circle, the filter becomes unstable.

Therefore, a high-order transfer function is usually factorized, such that poles and zeros can be gathered in pairs, in *biquad* sections.

$$
\begin{aligned}
H(z) &= \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{l=1}^{N} a_l z^{-l}} \\
&= \left(\frac{b_{1,0} + b_{1,1} z^{-1} + b_{1,2} z^{-2}}{1 - a_{1,1} z^{-1} - a_{1,2} z^{-2}} \right) \left(\frac{b_{2,0} + b_{2,1} z^{-1} + b_{2,2} z^{-2}}{1 - a_{2,1} z^{-1} - a_{2,2} z^{-2}} \right) \cdots \left(\frac{b_{Q,0} + b_{Q,1} z^{-1} + b_{Q,2} z^{-2}}{1 - a_{Q,1} z^{-1} - a_{Q,2} z^{-2}} \right)
\end{aligned}
$$

Every biquad section is implemented using its own direct or transposed form implementation (see Figure 7.2 on the next page). Advice on which form to use in practice can be found in section 2.8.2.5 on page 29.

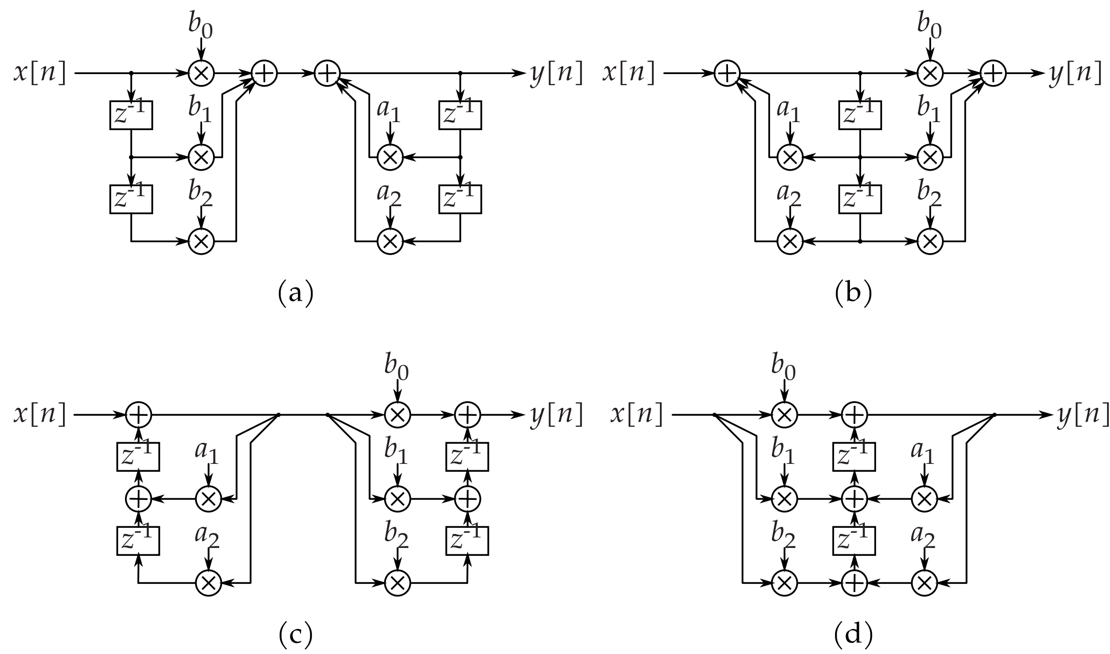
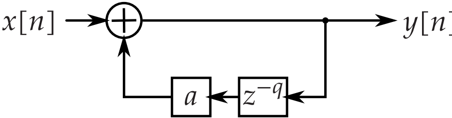


Figure 7.2: Biquad implementations: (a) direct form I, (b) direct form II, (c) transposed form I, and (d) transposed form II

7.2.3 Feedback comb filters

Definition

If we place the poles and zeros according to a particular pattern, we obtain a special kind of filter called a *feedback comb filter*.[1],[2] In this type of filter, a scaled and delayed version of the output signal is added to the input signal.



This leads to the following system description:

$$y[n] = x[n] + ay[n-q]$$

This system description can be easily transformed to the Z-domain:

$$\begin{aligned} Y(z) &= X(z) + az^{-q}Y(z) \\ &= \frac{1}{1-az^{-q}}X(z) \end{aligned}$$

This yields the following transfer function:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{z^q}{z^q - a}$$

[1]These comb filters are totally unrelated to the 2-line, 3-line, or 3D comb filters advertised as features for TV sets.
[2]A related type of filter is the *feedforward comb filter* (see section 6.2.2 on page 116).

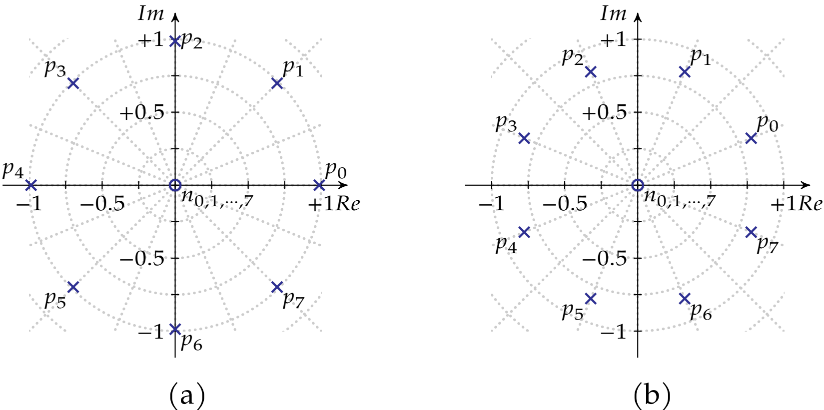


Figure 7.3: Pole-zero diagram of a feedback comb filter, for the case $q = 8$, with (a) $a = 0.9$ and (b) $a = -0.25$

Location of poles and zeros

Obviously, this transfer function has a zero with multiplicity q at $z = 0$:

$$n_{0,1,2,\ldots,q-1} = 0$$

The poles can be easily derived by setting the denominator to zero

$$\begin{aligned} z^q - a &= 0 \\ z^q &= a \end{aligned} \tag{7.2}$$

First, let's assume a is positive. Rewriting (7.2) with a in polar notation yields:

$$\begin{aligned} z^q &= a\,\mathrm{e}^{\mathrm{j}k2\pi} \\ z &= \sqrt[q]{a}\,\mathrm{e}^{\mathrm{j}\frac{2k\pi}{q}} \end{aligned}$$

with k integer.
Now, let's assume a is negative and substitute $b = -a$. Rewriting a in (7.2) in polar notation yields:

$$\begin{aligned} z^q &= b\,\mathrm{e}^{\mathrm{j}(\pi+k2\pi)} \\ z &= \sqrt[q]{b}\,\mathrm{e}^{\mathrm{j}\left(\frac{\pi}{q}+\frac{2k\pi}{q}\right)} \end{aligned}$$

with k integer.

If we want our filter to be stable, we must ensure that:

$$|a| < 1$$

As an example, the pole-zero diagrams for $q = 8$ in case $a = 0.9$ and in case $a = -0.25$ have been displayed in Figure 7.3.

The frequency spectra of these filters have been illustrated in Figure 7.4 on the facing page. They also show the reason for naming these filters *comb* filters.

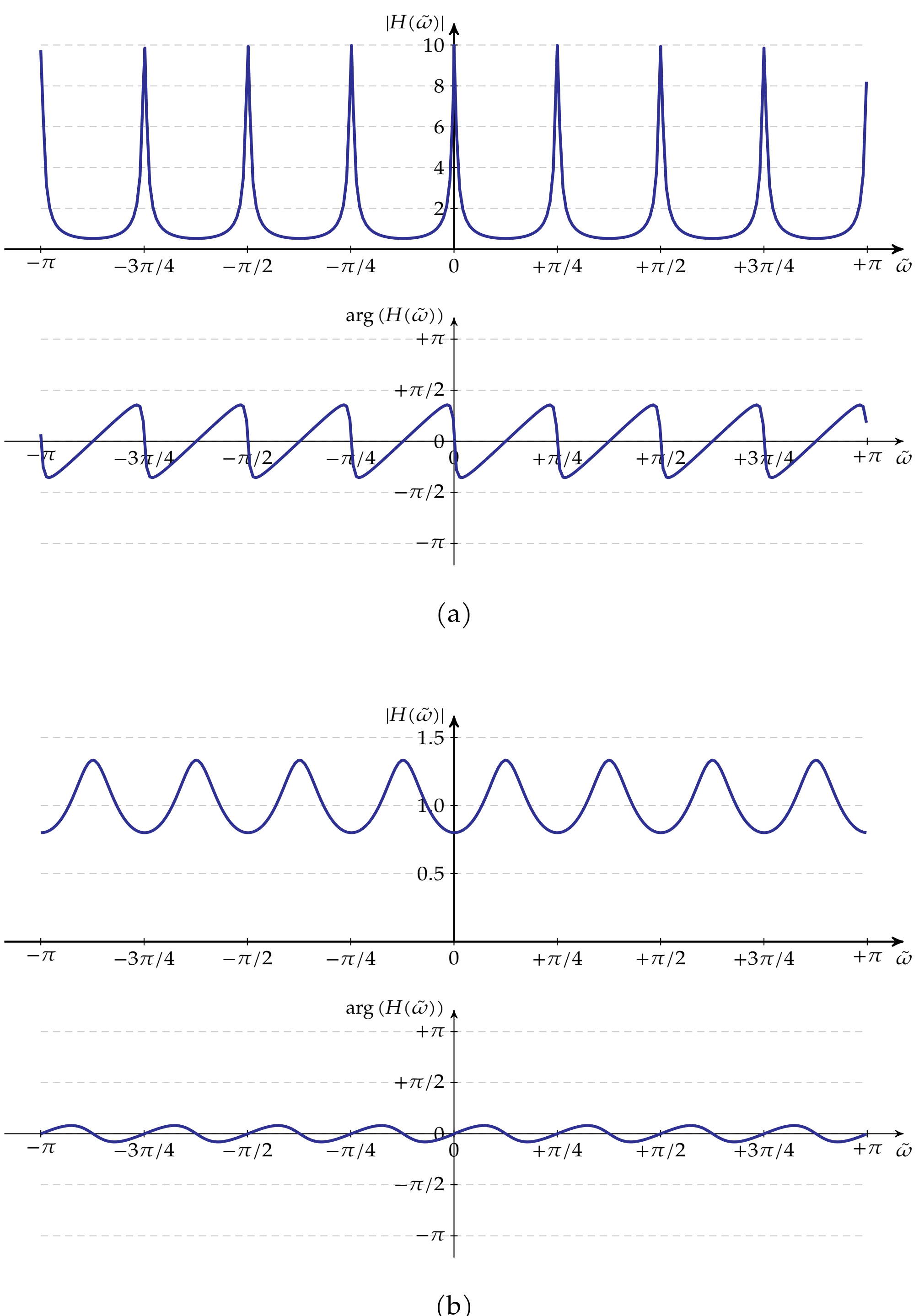


Figure 7.4: Frequency spectra of the feedback comb filters of Figure 7.3 on the facing page, for the case $q = 8$, with (a) $a = 0.9$ and (b) $a = -0.25$

Comb ratio

An important parameter in designing a comb filter is the ratio between peaks and dips, the so-called *comb ratio*. This ratio can be determined by analyzing the magnitude response of the filter:

$$\begin{aligned}\left|H\left(\mathrm{e}^{\mathrm{j}\tilde{\omega}}\right)\right| &= \left|\frac{\mathrm{e}^{\mathrm{j}q\tilde{\omega}}}{\mathrm{e}^{\mathrm{j}q\tilde{\omega}}-a}\right| \\ &= \frac{1}{\left|\mathrm{e}^{\mathrm{j}q\tilde{\omega}}-a\right|} \\ &= \frac{1}{\sqrt{\left(\cos(q\tilde{\omega})-a\right)^2+\sin^2(q\tilde{\omega})}} \\ &= \frac{1}{\sqrt{1-2a\cos(q\tilde{\omega})+a^2}}\end{aligned}$$

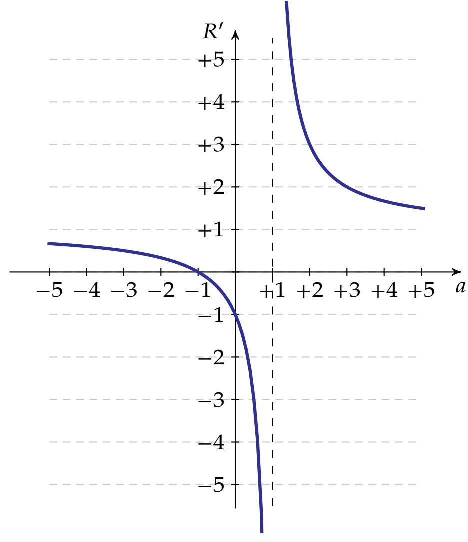
The extrema of this filter are attained for $q\tilde{\omega} = m\pi$ with m integer. As minima and maxima alternate, we can easily calculate the comb ratio R:

$$\begin{aligned}R &= \frac{\left|H\left(\mathrm{e}^{\mathrm{j}0}\right)\right|}{\left|H\left(\mathrm{e}^{\mathrm{j}\frac{\pi}{q}}\right)\right|} = \sqrt{\frac{1-2a\cos\pi+a^2}{1-2a\cos 0+a^2}} = \sqrt{\frac{1+2a+a^2}{1-2a+a^2}} \\ &= \left|\frac{a+1}{a-1}\right|\end{aligned}$$

Absolute values are a nuisance when solving equations, so let's start by investigating

$$R' = \frac{a+1}{a-1}$$

Take a look at the graph of R' on the right. As you can see R' is negative in the interval $[-1, 1]$. In order to keep our comb filter stable, we need to restrict a to this interval. Therefore:



- $-1 < a < 1$ leads to:

  $$R = \frac{1+a}{1-a}$$

  and therefore

  $$a = \frac{R-1}{R+1} \qquad (7.3)$$

- $|a| > 1$ does not lead to stable filter designs.

Therefore (7.3) is our design equation when a desired comb ratio is given.

Exercises

Some exercises on the design of feedback comb filters

Exercise 7.2.3-1: Design a feedback comb filter with 5 peaks and a comb ratio of 200 (i.e. a maximum at $\omega = 0$).
Verify the frequency response of your design using Octave.
Compose a direct form I and a transposed form I implementation for this filter.

Exercise 7.2.3-2: Design a feedback comb filter with 8 peaks and a comb ratio of 0.02 (i.e. a minimum at $\omega = 0$).
Verify the frequency response of your design using Octave.
Compose a direct form II and a transposed form II implementation for this filter.

7.2.4 Classical band-selection filters

Finding good pole-zero patterns is not easy. Luckily, the design of most classical band-selection filters has been implemented in Octave/Matlab.

7.2.4.1 Butterworth filters

The function `butter` allows designing Butterworth filters whose property it is to exhibit a maximally flat magnitude spectrum. Check out the documentation for more detailed information on this filter design function.

Let's design filters of order 8.

```
N = 8;      % = filter order
```

We can build a low-pass filter with normalized cut-off frequency $\tilde{\omega}_p = 0.25\pi$, i.e. $\omega_P = \omega_s/8$ using:

```
[B, A] = butter( N, 0.25 );
```

The numerator and denominator coefficients are returned as vectors `B` and `A` respectively. Specifying `'high'` as a third argument allows generating a high-pass filter.

Plot the spectrum yourself using the functions `freqz` and `freqz_plot` (see page 108) to observe the result in the frequency domain.

The function `butter` also allows designing band-pass filters. A band-pass filter with normalized corner frequencies of $\tilde{\omega}_{P,L} = 0.3\pi$ and $\tilde{\omega}_{P,U} = 0.6\pi$ can be designed using:

```
[B, A] = butter( n, [ 0.3, 0.6 ] );
```

Specifying `'stop'` as a third argument allows generating a band-stop filter.

If the order of the filter is too high, we might end-up in the (non)stability danger zone due to coefficient rounding. In those cases, it is better to calculate the poles and zeros making up the filter an combine them in biquad sections.

We can obtain the gain, the poles and the zeros of a Butterworth filter using:

```
[Z, P, g] = butter( N, 0.25 );
```

This function returns the zeros and poles as vectors Z and P and the gain g as a scalar. The same variations as mentioned above allow designing high-pass, band-pass and band-stop filters.

Observing those pole and zero-locations may teach you a lot about how the specific pole-zero patterns of Butterworth filters look like. Also try to find the relationship between the order of the filter and the roll-off slope in a Bode plot.

7.2.4.2 Bessel filters

The function `besself` allows designing Bessel filters whose property it is to exhibit a maximally flat phase spectrum. Alas, this function has not yet been implemented in Octave. Check out the Matlab documentation for more detailed information on this filter design function.

Let's design filters of order 8.

```
matlab:1> N = 8;      % = filter order
```

We can build a low-pass filter with normalized cut-off frequency of $\tilde{\omega}_P = 0.25\pi$, i.e. $\omega_P = \omega_s/8$ using:

```
matlab:2> [B, A] = besself( N, 0.25 );
```

The numerator and denominator coefficients are returned as vectors B and A respectively.

Plot the spectrum yourself using the functions `freqz` and `freqz_plot` (see page 108) to observe the result in the frequency domain.

If the order of the filter is too high, we might end-up in the (non)stability danger zone due to coefficient rounding. In those cases, it is better to calculate the poles and zeros making up the filter an combine them in biquad sections.

We can obtain the gain, the poles and the zeros of a Bessel filter using:

```
matlab:3> [Z, P, g] = besself( N, 0.25 );
```

This function returns the zeros and poles as vectors Z and P and the gain g as a scalar.

7.2.4.3 Chebyshev filters

Chebyshev filters allow ripple in passband (type I) or stopband (type II) in return for a steeper roll-off (i.e. the transition from passband to stopband).

Chebyshev I

The function `cheby1` allows designing Chebyshev I filters whose property it is to exhibit a maximal roll-off given an allowable passband ripple. Check out the documentation for more detailed information on this filter design function.

Let's design filters of order 8 allowing 1 dB ripple in the passband.

```
N  = 8;      % = filter order
Rp = 1;      % = passband ripple (in [dB])
```

We can build a low-pass filter with normalized cut-off frequency of $\tilde{\omega}_P = 0.25\pi$, i.e. $\omega_P = \omega_s/8$ using:

```
[B, A] = cheby1( N, Rp, 0.25 );
```

The numerator and denominator coefficients are returned as vectors `B` and `A` respectively. Specifying `'high'` as a fourth argument allows generating a high-pass filter.

Plot the spectrum yourself using the functions `freqz` and `freqz_plot` (see page 108) to observe the result in the frequency domain.

The function also allows designing band-pass filters. A band-pass filter with normalized corner frequencies of $\tilde{\omega}_{P,L} = 0.3\pi$ and $\tilde{\omega}_{P,U} = 0.6\pi$ can be designed using:

```
[B, A] = cheby1( n, Rp, [ 0.3, 0.6 ] );
```

Specifying `'stop'` as a fourth argument allows generating a band-stop filter.

If the order of the filter is too high, we might end-up in the (non)stability danger zone due to coefficient rounding. In those cases, it is better to calculate the poles and zeros making up the filter an combine them in biquad sections.

We can obtain the gain, the poles and the zeros of a Chebyshev I filter using:

```
[Z, P, g] = cheby1( N, Rp, 0.25 );
```

This function returns the zeros and poles as vectors `Z` and `P` and the gain `g` as a scalar. The same variations as mentioned above allow designing high-pass, band-pass and band-stop filters.

Observing those pole and zero-locations may teach you a lot about how the specific pole-zero patterns of Chebyshev I filters look like.

Chebyshev II

The function `cheby2` allows designing Chebyshev II filters whose property it is to exhibit a maximal roll-off given an allowable stopband ripple. Check out the documentation for more detailed information on this filter design function. The stopband ripple is specified as a stopband attenuation R_S.

To check out the function `cheby2`, just repeat the paragraph above, substituting `cheby1` by `cheby2` and `Rp` by `Rs`.

7.2.4.4 Elliptical / Cauer filters

Elliptical (or Cauer) filters allow ripple in the passband and in the stopband in return for an even steeper roll-off than was the case for the Chebyshev filters.

Let's design filters of order 8 allowing 1 dB ripple in the passband and requiring 60 dB stopband attenuation.

```
N  = 8;      % = filter order
Rp = 1;      % = passband ripple (in [dB])
Rs = 60;     % = stopband attenuation (in [dB])
```

We can build a low-pass filter with normalized cut-off frequency of $\tilde{\omega}_P = 0.25\pi$, i.e. $\omega_P = \omega_s/8$ using:

```
[B, A] = ellip( N, Rp, Rs, 0.25 );
```

The numerator and denominator coefficients are returned as vectors `B` and `A` respectively. Specifying `'high'` as a fifth argument allows generating a high-pass filter.

Plot the spectrum yourself using the functions `freqz` and `freqz_plot` (see page on page 108) to observe the result in the frequency domain.

The function also allows designing band-pass filters. A band-pass filter with normalized corner frequencies of $\tilde{\omega}_{P,L} = 0.3\pi$ and $\tilde{\omega}_{P,U} = 0.6\pi$ can be designed using:

```
[B, A] = ellip( n, Rp, Rs, [ 0.3, 0.6 ] );
```

Specifying `'stop'` as a fifth argument allows generating a band-stop filter.

If the order of the filter is too high, we might end-up in the (non)stability danger zone due to coefficient rounding. In those cases, it is better to calculate the poles and zeros making up the filter an combine them in biquad sections.

We can obtain the gain, the poles and the zeros of a Chebyshev I filter using:

```
[Z, P, g] = ellip( N, Rp, Rs, 0.25 );
```

This function returns the zeros and poles as vectors `Z` and `P` and the gain `g` as a scalar. The same variations as mentioned above allow designing high-pass, band-pass and band-stop filters.

Exercises
Some exercises on designing band-selection IIR filters

Exercise 7.2.4.4-1: Design a band-stop Butterworth filter of order 16 with corner frequencies $\tilde{\omega}_{P,L} = 0.12\pi$ and $\tilde{\omega}_{P,U} = 0.3\pi$.
Determine the frequencies at which the attenuation equals -100 dB.

Exercise 7.2.4.4-2: Design a Chebyshev high-pass filter without passband ripple, and realizing 46 dB attenuation in the stopband. The cut-off frequency of the filter should be $\omega_P = 0.3\omega_s$. The stopband corner frequency should be $\omega_S = 0.29\omega_s$.
(*) Try increasing the order of the filter above 50 and check the result.

Exercise 7.2.4.4-3: Design a Cauer bandpass filter for a DSP system operating with a sample frequency of 50 kHz with corner frequencies 5 kHz and 12.5 kHz. Allow 0.5 dB ripple in the passband and enforce 80 dB attenuation in the stopbands. Make the design for various values of N.

- $N = 2$
- $N = 4$
- $N = 6$
- $N = 8$

Determine for every case the values of the passband an the stopband corners.

7.2.5 Strengths and weaknesses

The pole-zero placement technique is a *very simple* design technique. However, it assumes that you were able to compose a pole-zero pattern that accommodates your needs. For some very specific filter types (i.e. comb and band-selection filters) this gap has been smoothly bridged.

7.3 Impulse invariance method

7.3.1 Fundamentals

Imagine having designed an analog filter whose impulse response is exactly your application's requirement. An obvious question might be: can we design a discrete-time IIR filter that exhibits a discrete-time (infinite) impulse response that matches the analog impulse response. The answer is: yes, we can.

Assume the filter's transfer function is:

$$H(s) = \frac{\sum_{k=0}^{M} b_k s^k}{\sum_{l=0}^{N} a_l s^l} \tag{7.4}$$

We can easily apply partial fraction decomposition to (7.4). Assuming for the sake of simplicity that there are no coinciding poles and no coinciding zeros, we obtain:

$$H(s) = \sum_{l=1}^{N} \frac{A_l}{s - p_l}$$

As our filter is assumed to be stable, all p_j are located in the left-hand half-plane. As our filter is assumed to exhibit a real impulse response, all poles appear in complex-conjugate pairs.

The corresponding impulse response can be obtained by inverse Laplace transform and equals:

$$h(t) = \sum_{l=1}^{N} A_l u(t)\,\mathrm{e}^{p_l t}$$

We can sample this impulse response at a rate $1/T_s$ to obtain:

$$h[n] = h(nTs) = \sum_{l=1}^{N} A_l u[n]\,\mathrm{e}^{p_l n T_s}$$

Applying the Z-transform to both members of the equations and taking into account that

$$\mathrm{e}^{-\alpha n} u[n] \xrightarrow{Z} \frac{z}{z - \mathrm{e}^{-\alpha}}$$

yields:

$$H(z) = \sum_{l=1}^{N} A_l \frac{z}{z - \mathrm{e}^{p_l T_s}} \tag{7.5}$$

As we can easily observe, stable continuous-time poles (in the Laplace plane) yield stable discrete-time poles (in the Z-plane). In (7.5) complex conjugate term-pairs may occur. If we combine these and the remaining real terms in pairs they can be mapped to real (as opposed to complex) biquad filter sections.

The deriviation above can easily be extended to cover the case of poles and/or zeros with a multiplicity higher than one.

The basic Z-transform pairs that come in most handy are:

$$\begin{aligned}
u[n]\,\mathrm{e}^{-an} &\xrightarrow{Z} \frac{z}{z - \mathrm{e}^{-a}} \\
u[n]n\,\mathrm{e}^{-an} &\xrightarrow{Z} \frac{z\,\mathrm{e}^{-a}}{(z - \mathrm{e}^{-a})^2} \\
u[n]n^2\,\mathrm{e}^{-an} &\xrightarrow{Z} \frac{z\,\mathrm{e}^{-a}\left(z^2 - \mathrm{e}^{-2a}\right)}{(z - \mathrm{e}^{-a})^4}
\end{aligned}$$

Note that every subsequent transform pair above can derived from its predecessor using the property:

$$nx[n] \xrightarrow{Z} -z\frac{\mathrm{d}X(z)}{\mathrm{d}z}$$

7.3.2 Design procedure

The design procedure is most simple:

1. determine the analog filter's transfer function $H(s)$
2. decompose $H(s)$ using partial fraction decomposition
3. apply the inverse Laplace transform to obtain the continuous-time impulse $h(t)$
4. sample $h(t)$ to obtain a discrete-time impulse response $h[n]$
5. convert $h[n]$ to $H(z)$ using the Z-transform.

6. gather the complex-conjugate and real poles and zeros poles in biquad sections
7. make a parallel connection of all the biquad sections (using an extra summation node at the output) to obtain the desired filter

Of course, it is possible to use higher-order sections than pure biquad sections. However, take into account that high-orders will put you up with stability problems.

7.3.3 Example

Let's consider the following analog filter:

$$H(s) = \frac{s^2 - 4s - 5}{s^4 + 16s^3 + 98s^2 + 280s + 325}$$

The hard way

Let's first decompose this transfer by partial fraction decomposition using the `residue` function[3]:

```
[R, P, K, E] = residue( [ 1 -4 -5], [1 16 98 280 325] );
```

This allows rewriting the transfer function as follows:

$$H(s) = \frac{0.75}{s-(-5)} + \frac{5}{(s-(-5))^2} + \frac{-0.375 + \mathrm{j}0.625}{s-(-3+\mathrm{j}2)} + \frac{-0.375 - \mathrm{j}0.625}{s-(-3-\mathrm{j}2)}$$

Applying the inverse Laplace transform results in:

$$h(t) = u(t)\left(0.75\,\mathrm{e}^{-5t} + 0.75t\,\mathrm{e}^{-5t} + (-0.375+\mathrm{j}0.625)\,\mathrm{e}^{(-3+\mathrm{j}2)t} + (-0.375-\mathrm{j}0.625)\,\mathrm{e}^{(-3-\mathrm{j}2)t}\right)$$

Sampling this continuous-time impulse response results in:

$$h[n] = u[n]\left(0.75\,\mathrm{e}^{-5nT_s} + 0.75nT_s\,\mathrm{e}^{-5nT_s} + (-0.375+\mathrm{j}0.625)\,\mathrm{e}^{(-3+\mathrm{j}2)nT_s} + (-0.375-\mathrm{j}0.625)\,\mathrm{e}^{(-3-\mathrm{j}2)nT_s}\right)$$

Applying the Z-transform yields:

$$H(z) = \frac{0.75z}{z-\mathrm{e}^{-5T_s}} + \frac{0.75z}{(z-\mathrm{e}^{-5T_s})^2} + \frac{(-0.375+\mathrm{j}0.625)z}{z-\mathrm{e}^{(-3+\mathrm{j}2)T_s}} + \frac{(-0.375-\mathrm{j}0.625)z}{z-\mathrm{e}^{(-3-\mathrm{j}2)T_s}}$$

Gathering terms pairwise yields:

$$H(z) = \frac{0.75z^2 + 0.75(1-\mathrm{e}^{-5T_s})z}{z^2 - 2\,\mathrm{e}^{-5T_s} + \mathrm{e}^{-10T_s}} + \frac{0.75z + \mathrm{e}^{-3T_s}(0.75\cos(2T_s) + 1.25\sin(2T_s))}{z^2 - 2\,\mathrm{e}^{-3T_s}\cos(2T_s)z + \mathrm{e}^{-6T_s}}$$

These two sections can simply be mapped on two biquad sections, that are connected in parallel to the input and whose outputs are summed.

[3]Check the documentation of the `residue` function to see what output this function generates.

The easy way

The above mathematics is quite prone to errors. However, the procedure is very straightforward and therefore easily automated. This has been done in the MATLAB function `impinvar`. It's use is most simple. Assuming $T_s = 0.25\,\mathrm{s}$, we can obtain the same result using

```
matlab:1> [bz, az] = impinvar( [ 1 -4 -5], [1 16 98 280 325], 0.25 );
```

with `bz` and `az` the numerator and denominator coefficients in the Z-plane.

Exercises

An exercise on the impulse invariance method

Exercise 7.3.3-1: Design a digital IIR filter using the impulse invariance method that mimicks the impulse response of the following system:

$$H(s) = \frac{s^2 - 1s - 6}{s^3 + 3s^2 + 5s - 1}$$

assuming $T_s = 0.1\,\mathrm{s}$.
Verify your result using the MATLAB-function `impinvar`.

7.3.4 Strengths and weaknesses

The impulse invariance method is appropriate if the time-domain behavior of the filter is important. Using this technique to approximate analog filters that were selected because of their frequency-domain behavior is not a wise thing to do. Indeed, because of the time discretization, frequency aliasing will ruin the beneficial properties.

7.4 Bilinear transformation

7.4.1 Fundamentals

Directly designing discrete-time IIR filters is not easy. Therefore, in many cases the design of a discrete-time IIR filter, starts from an analog filter design (for which the mathematics are well understood). Afterwards, this design is converted to the discrete-time domain using a transformation method.

The question therefore is: how can we map the continuous-time Laplace domain onto the discrete-time Z-domain?

Actually, the Z-transform was defined by a mapping:

$$z = e^{sT_s} \tag{7.6}$$

However, this theoretical mapping is not an easy one. Transforming a continuous-time transfer function $H_c(s)$ into a discrete-time transfer function $H_d(z)$ would mean performing

the substitution:

$$s = \frac{\ln z}{T_s} \tag{7.7}$$

and thus

$$H_d(z) = H_c\left(s \mapsto \frac{\ln z}{T_s}\right)$$

The net result of this substitution is that simple continuous-time transfer functions (that are ratios of polynomials in s) become highly nonlinear functions of z. Implementing those nonlinear functions is a nightmare.

Obviously we need a workaround. Can we find an approximate mapping that avoids the nasty logarithm in (7.7)? Moreover, this cannot be just any approximation. The approximate mapping must exhibit a number of properties:

- stable continuous-time systems are to be mapped on stable discrete-time systems, and
- the $\mathrm{j}\omega$ axis is to be mapped onto the unit circle $|z| = 1$.

The approximations we will see below, are all based on the power series of e^x:

$$\mathrm{e}^x = \sum_{n=0}^{+\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

Let's give it a try!

Approximation attempt 1

Let's approximate the exponential in (7.6) by truncating its power series after the linear term:

$$\mathrm{e}^x \approx 1 + x$$

Applied to our mapping problem, this leads to:

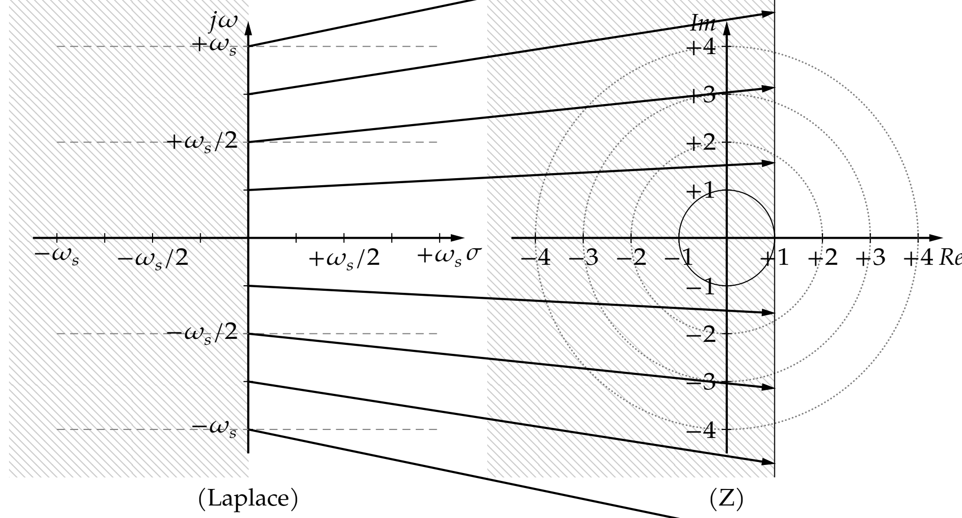
$$z = \mathrm{e}^{sT_s} \approx 1 + sT_s$$

And therefore:

$$s \approx \frac{z-1}{T_s}$$

This is the so-called *forward-Euler* approximation.

The mapping of the continuous-time Laplace domain to the discrete-time Z-domain is depicted below. The left-hand half-plane in the Laplace domain has been hatched, so has been the corresponding mapped region in the Z-domain. The mapping of the imaginary axis $\mathrm{j}\omega$ has been indicated by arrows.



As you can see a part of the left-hand half-plane of the Laplace domain is mapped inside the unit-circle in the Z-domain. Good. However, a large part is mapped to a region that extends beyond the unit circle. This means that stable continuous-time filters might be mapped onto unstable discrete-time filters. This is not good.

In addition, the $j\omega$ axis is not mapped onto the unit-circle, but on the line defined by $Re(z) = 1$.

We cannot but conclude that this attempt is not really successful.

Approximation attempt 2

Let's approximate the exponential in (7.6) by truncating its power series after the quadratic term:

$$e^x \approx 1 + x + \frac{x^2}{2}$$

Applied to our mapping problem, this leads to:

$$z = e^{sT_s} \approx 1 + sT_s + \frac{s^2T_s^2}{2}$$

Solving for s leads to:

$$s \approx -1 \pm \sqrt{2z - 1}$$

Alas, this substitution exhibits the same problem as the original exact mapping: it morphs simple continuous-time transfer functions into complicated nonlinear transfer functions. Continuing down this path makes no sense.

Approximation attempt 3

Let's try a different approach and first rewrite (7.6) as

$$z = e^{sT_s} = \frac{1}{e^{-sT_s}}$$

and then substitute the exponentials by their power series truncated after the first term, leading to

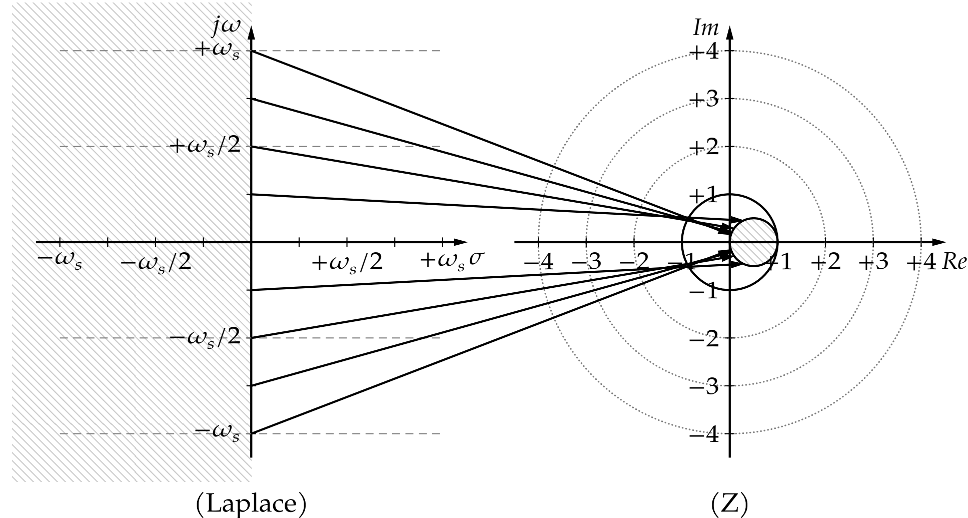
$$z \approx \frac{1}{1 - sT_s}$$

Solving for s leads to:

$$s \approx \frac{1}{T_s}\frac{z-1}{z}$$

This is the so-called *backward-Euler* approximation.

The mapping of the continuous-time Laplace domain to the discrete-time Z-domain is depicted below. The left-hand half-plane in the Laplace domain has been hatched, so has been the corresponding mapped region in the Z-domain. The mapping of the imaginary axis jω has been indicated by arrows.



As you can see the entire left-hand half-plane of the Laplace domain is mapped inside the unit-circle in the Z-domain. This means that stable continuous-time filters are mapped onto stable discrete-time filters. Good.

However, the jω axis is not mapped onto the unit-circle, but on the circle with center point 0.5 and radius 0.5.

So, close, but no cigar.

Approximation attempt 4

Reconsidering our attempts so far, we need something in between attempt 1 and attempt 3. This leads to the idea of rewriting (7.6) as

$$z = e^{sT_s} = \frac{e^{\frac{sT_s}{2}}}{e^{-\frac{sT_s}{2}}}$$

and then substitute the exponentials by their power series truncated after the first term, leading to

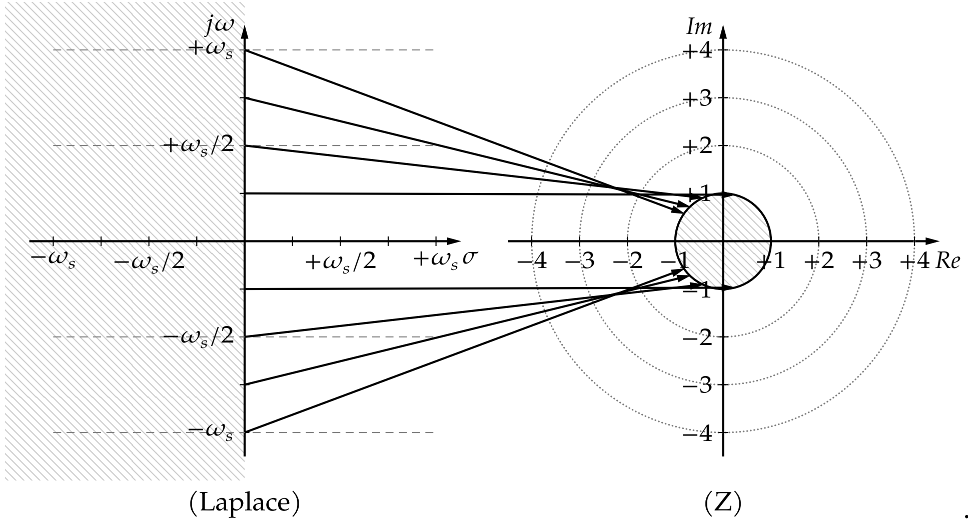
$$z \approx \frac{1 + s\frac{T_s}{2}}{1 - s\frac{T_s}{2}}$$

Solving for s leads to:

$$s \approx \frac{2}{T_s}\frac{z-1}{z+1} \tag{7.8}$$

This is the so-called *bilinear* approximation.

The mapping from the continuous-time Laplace domain to the discrete-time Z-domain is depicted below. The left-hand half-plane in the Laplace domain has been hatched, so has been the corresponding mapped region in the Z-domain. The mapping of the imaginary axis $j\omega$ has been indicated by arrows.



;

This looks too good to be true. The entire left-hand half-plane of the Laplace domain is mapped inside the unit-circle in the Z-domain: no space is wasted as in approximation attempt 3. In addition, the $j\omega$ axis is perfectly mapped onto the unit-circle!

No kidding: it is true.

Applying the bilinear transformation is very simple now: given a continuous-time filter with transfer function $H_c(s)$, we obtain our discrete-time filter with transfer function $H_d(z)$ by using the substitution formula of (7.8):

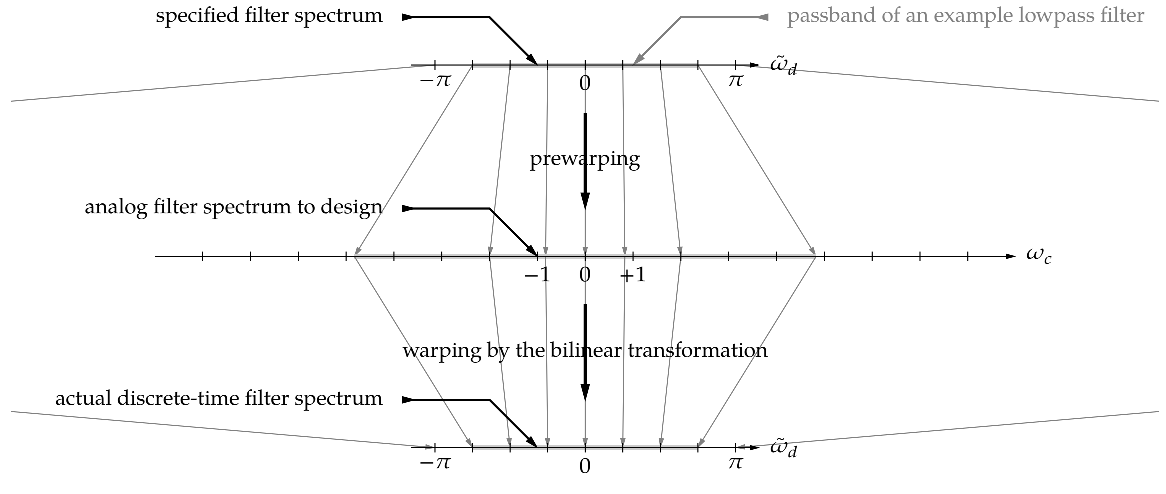
$$H_d(z) = H_c\left(s \mapsto \frac{2}{T_s}\frac{z-1}{z+1}\right)$$

7.4.2 Frequency warping and frequency-prewarping

The imaginary axis in the complex Laplace domain is mapped onto the unit circle in a nonlinear way. The mapping can be derived starting from:

$$s = \frac{2}{T_s}\frac{z-1}{z+1}$$

Now, substitute $s = j\omega_c$ and $z = e^{j\omega_d}$ to see how the continuous-time frequency ω_c relates to the discrete-time frequency ω_d.



;

Figure 7.5: Illustration of the prewarping and warping effect related to the bilinear transformation

en

$$
\begin{aligned}
\mathrm{j}\omega_c &= \frac{2}{T_s}\frac{\mathrm{e}^{\mathrm{j}\omega_d T_s}-1}{\mathrm{e}^{\mathrm{j}\omega_d T_s}+1} \\
&= \frac{2}{T_s}\frac{\mathrm{e}^{\mathrm{j}\frac{\omega_d T_s}{2}}}{\mathrm{e}^{\mathrm{j}\frac{\omega_d T_s}{2}}}\frac{\mathrm{e}^{\mathrm{j}\frac{\omega_d T_s}{2}}-\mathrm{e}^{-\mathrm{j}\frac{\omega_d T_s}{2}}}{\mathrm{e}^{\mathrm{j}\frac{\omega_d T_s}{2}}+\mathrm{e}^{-\frac{\mathrm{j}\omega_d T_s}{2}}} \\
&= \mathrm{j}\frac{2}{T_s}\frac{\sin\left(\frac{\omega_d T_s}{2}\right)}{\cos\left(\frac{\omega_d T_s}{2}\right)} \\
\omega_c &= \frac{2}{T_s}\tan\left(\frac{\omega_d T_s}{2}\right)
\end{aligned}
\tag{7.12}
$$

and vice versa:

$$
\omega_d = \frac{2}{T_s}\arctan\left(\frac{\omega_c T_s}{2}\right) \tag{7.13}
$$

The mapping of the continuous frequency ω_c onto the corresponding discrete-time frequency ω_d using (7.13) is called "warping".

Now, how does this influence our filter design? If we want a particular event (for example a pass-band to stop-band transition to occur at a frequency ω_d), we need to start from an analog filter that has that transition for the continuous-time frequency ω_c. The process of mapping ω_d to ω_c using (7.4.2) is called "prewarping" because it is as if one applies a (frequency) pre-counterdistortion to correct the actual frequency distortion of the bilinear transform.

This principle has been outlined in Figure 7.5 for $T_s = 1$.

7.4.3 Example

Consider the first-order analog low-pass filter

$$
H(s) = \frac{-a}{s-a}
$$

with a real and $a < 0$ to ensure stability. The cut-off frequency of this filter equals:

$$\omega_c = \pm a$$

The filter above is a Butterworth filter.

Assume we'd like to build a low-pass filter with a cut-off frequency at $\omega_d = 0.25\omega_s$ with $T_s = 1$ ms.

The hard way

Let's start by prewarping the discrete-time cut-off frequency to obtain a continuous-time cut-off frequency that we can plug into our analog filter design.

$$\begin{aligned}\omega_c &= \frac{2}{T_s}\tan\left(\frac{\omega_d T_s}{2}\right)\\ &= \frac{2}{T_s}\tan(0.25\pi)\\ &= \frac{2}{T_s}\end{aligned}$$

The following analog filter has the appropriate cut-off frequency:

$$H(s) = \frac{\frac{2}{T_s}}{s + \frac{2}{T_s}}$$

Applying the bilinear transform $s \mapsto \frac{2}{T_s}\frac{z-1}{z+1}$ results in:

$$\begin{aligned}H(z) &= \frac{2/T_s}{\frac{2}{T_s}\frac{z-1}{z+1} + \frac{2}{T_s}}\\ &= \frac{1}{\frac{z-1}{z+1} + 1}\\ &= \frac{(z+1)}{2z}\\ &= \frac{z+1}{2z}\end{aligned}$$

The easy way

Our workhorses Octave and Matlab both implement the `bilinear` function to ease the design of IIR filters based on the bilinear transformation. However, the frequency pre-warping is so simple that it has not been implemented as a separate function.

```
Ts = 1e-3;                    % sample period
wd = 0.25 * pi;               % discrete-time corner freq
wc = 2/Ts * tan( wd * Ts / 2 ) % prewarped cont.-time corner freq
```

Designing the analog first-order Butterworth filter can also be done using our math tool:

```
[b, a] = butter( 1, wc, 's' );
```

Next, we can perform the bilinear transformation:

```
[zb, za] = bilinear( b, a, Ts );
```

Finally, let's calculate the spectrum

```
[H, W] = freqz( zb, za, 256, "whole" );
```

and plot it to verify our result:

```
freqz_plot( W, H );
```

Actually, the classical band-selection filter functions that we've seen in section 7.2.4 on page 171 have been implemented using prewarping, analog filter design and the bilinear transformation as a final step.

Exercises
Some exercises on IIR filter design using the bilinear transformation

Exercise 7.4.3-1: Design a discrete-time IIR highpass filter using the bilinear transformation with frequency prewarping for a DSP system with a sampling frequency $f_s = 1\,\text{MHz}$. The cut-off frequency of the filter is to be $f_{co} = 200\,\text{kHz}$.

The following analog filter is a first-order high-pass Butterworth filter with cut-off frequency ω_c:

$$H(s) = \frac{s}{s + \omega_c}$$

Draw a transposed form I implementation of the filter you designed.

Exercise 7.4.3-2: Design a discrete-time IIR bandstop filter using bilinear transformation with frequency prewarping for a DSP system with a sampling frequency $f_s = 1\,000\,\text{Hz}$. The frequency $\omega_0 = 2\,400\,\text{rad/s}$ needs to be suppressed.

An appropriate analog band-stop filter suppressing the frequency ω_n equals:

$$H(s) = \frac{s^2 + \omega_n^2}{s^2 + \frac{\omega_n}{10}s + \omega_n^2}$$

Draw a transposed form type II implementation of the filter you designed.

Exercise 7.4.3-3: (*) Vinyl records used to be recorded using a pre-emphasis filter that boosts high frequencies and attenuates low frequencies. The goal of this pre-emphasis was twofold: (a) reducing the influence of noise and clicks due to imperfections in the vinyl and reducing the swing of the groove at low frequencies. The latter allows increasing the groove density, leading to longer recording times on standard 7 inch 'singles' and 12 inch 'LPs'.

The RIAA pre-emphasis curve is defined as:

$$H_p(s) = \frac{(1 + \tau_1 s)(1 + \tau_3 s)}{1 + \tau_2 s}$$

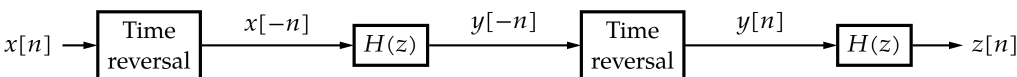


Figure 7.6: The principle of reverse filtering

with

$$\tau_1 = \frac{1}{2\pi} 3\,180\,\mu\text{s}$$
$$\tau_2 = \frac{1}{2\pi} 318\,\mu\text{s}$$
$$\tau_3 = \frac{1}{2\pi} 75\,\mu\text{s}$$

These pre-emphasis recordings require the turntable (or rather the amplifier) to contain a de-emphasis filter. Assume that we have digital recordings (at $f_s = 44.1\,\text{kHz}$) that have been recorded using a turntable without a RIAA de-emphasis filter and that we want to restore the original frequence response of these recordings.

To this end, design a digital de-emphasis filter using the bilinear transformation technique.

Draw a direct-form type II implementation of the filter you designed.

Compare the resulting frequency response with the original analog frequency response curve.

7.4.4 Strengths and weaknesses

The bilinear transformation technique is a *simple* and (taking into account prewarping and warping) a very effective design technique. However, it assumes that you are able to design an analog filter that suites the pre-warped discrete-time requirements.

7.5 Reverse filtering

In general, IIR filters don't exhibit a linear phase characteristic. However, if we're processing signals in batch (i.e. not in real-time), there is a trick we can use to correct this nonlinear phase characteristic: reverse filtering.

7.5.1 Principle

The idea is to time-reverse the input signal, then send it through an appropriate IIR filter, then again reverse that output signal in time and use it as the input of the very same filter again! This principle has been depicted in Figure 7.6.

7.5.2 Fundamentals

Reverse filtering is based on the behavior of the Discrete-time Fourier Transform for *real* signals under *time-reversal*.

Real signals

A particular thing to remember from studying Fourier theory, is that the spectrum of real signals is Hermitic, i.e. it has an

- even magnitude spectrum
- odd phase spectrum

Time reversal

Consider a signal $x[n]$. The discrete-time Fourier transform is defined as follows:

$$X_p(\omega) = \sum_{n=-\infty}^{+\infty} x[n]\, \mathrm{e}^{-\mathrm{j}\omega n T_s}$$

Now, let's consider the time-reversed signal $x[-n]$. Its discrete-time Fourier transform is defined by:

$$X_{p,-n}(\omega) = \sum_{n=-\infty}^{+\infty} x[-n]\, \mathrm{e}^{-\mathrm{j}\omega n T_s}$$

If we perform the substitution $m = -n$, we can write:

$$\begin{aligned} X_{p,-n}(\omega) &= \sum_{m=-\infty}^{+\infty} x[m]\, \mathrm{e}^{-\mathrm{j}(-\omega) m T_s} \\ &= X_p(-\omega) \end{aligned}$$

Our conclusion can be written in short:

$$\begin{aligned} x[n] &\xrightarrow{\text{DtFT}} X_p(\omega) \\ x[-n] &\xrightarrow{\text{DtFT}} X_p(-\omega) \end{aligned}$$

The effect of time reversal for real signals

Now, take a second look at Figure 7.6 on the facing page and consider a particular frequency component ω_1 that's present in the input signal $x[n] \xrightarrow{\text{DtFT}} X_p(\omega)$.

The considered frequency component therefore has a value of

$$X_p(\omega_1)$$

Time reversal transforms the value of this frequency component into:

$$X_p(-\omega_1)$$

That frequency is filtered through $H(z)$ and will be present in the output signal $y[-n]$ with a value

$$X_p(-\omega_1)\, \left|H(\mathrm{e}^{\mathrm{j}\omega_1})\right|\, \mathrm{e}^{\mathrm{j}\arg(H(\mathrm{e}^{\mathrm{j}\omega_1}))}$$

Time reversing $y[-n]$ into $y[n]$ corresponds to reversing the spectrum, therefore, the frequency component will have the following value:

$$X_p(\omega_1)\left|H(\mathrm{e}^{-\mathrm{j}\omega_1})\right| \mathrm{e}^{\mathrm{j}\arg(H(\mathrm{e}^{-\mathrm{j}\omega_1}))}$$

As we know that the magnitude spectrum of real signals is even and their phase spectrum is odd, we can reduce this to:

$$X_p(\omega_1)\left|H(\mathrm{e}^{\mathrm{j}\omega_1})\right| \mathrm{e}^{-\mathrm{j}\arg(H(\mathrm{e}^{\mathrm{j}\omega_1}))}$$

If we filter $y[n]$ again through $H(z)$, we obtain

$$X_p(\omega_1)\left|H(\mathrm{e}^{\mathrm{j}\omega_1})\right| \mathrm{e}^{-\mathrm{j}\arg(H(\mathrm{e}^{\mathrm{j}\omega_1}))}\left|H(\mathrm{e}^{\mathrm{j}\omega_1})\right| \mathrm{e}^{\mathrm{j}\arg(H(\mathrm{e}^{\mathrm{j}\omega_1}))}$$

and this reduces to:

$$X_p(\omega_1)\left|H(\mathrm{e}^{\mathrm{j}\omega_1})\right|^2$$

This means that the original phase of the signal $x[n]$ has not been altered by the sequence of time-reversal, filtering, time-reversal and filtering.

Exercises

An exercise on reverse filtering.

Exercise 7.5.2-1: Consider audio signals that have been sampled at a rate $f_s = 48\,\mathrm{kHz}$. We'd like to filter away low frequencies below 20 Hz and high frequencies above 20 kHz.

Consider the following analog-filter transfer function:

$$H(s) = \frac{a_1 s}{1 + a_1 s + a_2 s^2}$$

If the poles of this transfer function are in the left-hand plane and sufficiently split (say $|p_2| > 100\,|p_1|$), it implements a band-pass filter with edge frequencies $\omega_{P,L} = \frac{1}{a_1}$ and $\omega_{P,U} = \frac{a_1}{a_2}$.

If our signals would be analog, then passing the signal twice through this filter would be considered sufficient. For simplicity you may assume that passing the signal twice through the filter realizes a filter with the corner frequencies $\omega_{P,L}$ and $\omega_{P,U}$ mentioned above.

Design a discrete-time *linear-phase* IIR bandpass filter for batch processing of these signals.

7.5.3 Strengths and weaknesses

The technique of reverse filtering yields a filter that exhibits zero-phase (so far unseen for IIR filters) and a magnitude shaping equal to the square of the magnitude spectrum of the original filter.

A weakness is that the technique has limited flexibility. It can only be applied for non-real-time filtering, the so-called batch filtering.

7.6 Conclusion

The design methods we've treated all have their particular weaknesses and strengths. Table 7.1 summarizes the most important ones. In this table, the following abbreviations are used:

- l, m, and h for low, medium and high,
- y and n for yes and no,
- +, ± and - for good, medium and bad.

Design method	Impulse response oriented	Frequency response oriented	Linear phase	Gibbs sensitivity	Transition band control	Ripple control	Flexibility	Design procedure convergence	Design procedure complexity	Design procedure computation time
Zero placement		•	n	l	-	-	-	+	l	l
Impulse response invariance	•		n	l	-	-	+	+	m	l
Bilinear transformation		•	n	l	+	+	+	+	m	l
Reverse filtering		•	y	l	+	+	-	+	l	l

Table 7.1: Overview of the IIR-filter design methods

Advanced Signal Transformations

Chapter 8

Signal Transforms — Short-time Fourier Transform

In this chapter, you will learn:

- what time-frequency atoms and transforms are,
- how the dictionaries corresponding to these transforms can be generated,
- why Heisenberg appears in DSP,
- how Gabor exploited the time-frequency ideas to create the short-time Fourier Transform,
- how to use the very practical short-time discrete Fourier transform.

After having read/studied this chapter, you are expected to be able to

- explain what time-frequency transforms are in general,
- explain what time-frequency atoms are, and how their size and position is calculated,
- explain the short-Time Fourier transform and its discretized version,
- to apply the discrete short-Time Fourier transform to filter and reconstruct signals, or to analyze them.

8.1 Time-frequency atoms

So far we've studied the decomposition of signals in waveforms that

- are very concentrated in time, but are spread out over the entire frequency range: the *impulse decomposition*;
- are very concentrated in frequency, but are spread out over the entire time range: the *Fourier decomposition*.

Now, let's take a position in between those two extremes. Consider real/complex unit waveforms $\phi_\gamma(t)$ (i.e. $\left\|\phi_\gamma\right\|^2 = \int_{-\infty}^{+\infty} \left|\phi_\gamma(t)\right|^2 \, \mathrm{d}t = 1$) that are concentrated in time and in frequency.[1]

[1] The parameter $\gamma \in \Gamma$ is a an index (it may be a multiparameter index).

This means that both $\phi_\gamma(t)$ and $\Phi_\gamma(\omega) = \mathcal{F}(\phi_\gamma)$ are active around a central time and frequency value and quickly decay when one moves away from that value. Such a waveforms are called *time-frequency atoms*.

8.1.1 Position and size of the atoms in the time domain

As with true atoms, one might ask the question what the position and the size of these atoms are.

Of course, the position and the size of functions is not a common concept. Therefore, we borrow a concept from probability theory. There, two well know concepts, the expected value (the 'mean') and the variance of a random variable X with probability density function $f_X(x)$, are defined as:

$$\alpha_{1,X} = E\{X\} = \int_{-\infty}^{+\infty} x f_X(x)\,\mathrm{d}x$$
$$\sigma_X^2 = \int_{-\infty}^{+\infty} \left(x - \alpha_{1,x}\right)^2 f_X(x)\,\mathrm{d}x$$

The expected value corresponds to the position of the probability distribution; the variance indicates how 'wide' it is. We often refer to $f_X(x)$ as the *PDF*.

If these definitions put you in total darkness, please read Appendix D.

Given the facts that

$$\left|\phi_\gamma(t)\right|^2 \geq 0$$

and

$$\int_{-\infty}^{+\infty} \left|\phi_\gamma(t)\right|^2 \mathrm{d}t = 1$$

the function $\left|\phi_\gamma(t)\right|^2$ can be considered as a proper PDF of the atom.

Using that parallelism, we can define the position of the atom in time as

$$t_\gamma = \int_{-\infty}^{+\infty} t \left|\phi_\gamma(t)\right|^2 \mathrm{d}t$$

and the variance of the atom as

$$\sigma_{t,\gamma}^2 = \int_{-\infty}^{+\infty} (t - t_\gamma)^2 \left|\phi_\gamma(t)\right|^2 \mathrm{d}t.$$

The square root of the variance, a.k.a. the standard deviation $\sigma_{t,\gamma}$, then can be considered as a measure for the width of the atom in the time domain.

8.1.2 Position and size of the atoms in the frequency domain

Parseval's theorem states that:

$$\left\|\phi_\gamma(t)\right\|^2 = \int_{-\infty}^{+\infty} \left|\phi_\gamma(t)\right|^2 \mathrm{d}t = \frac{1}{2\pi} \int_{-\infty}^{+\infty} \left|\Phi_\gamma(\omega)\right|^2 \mathrm{d}\omega = \frac{1}{2\pi} \left\|\Phi_\gamma(\omega)\right\|^2$$

Therefore, $\frac{1}{2\pi}\left|\Phi_\gamma(\omega)\right|^2$ qualifies as a probability density function of the atom in the frequency domain.

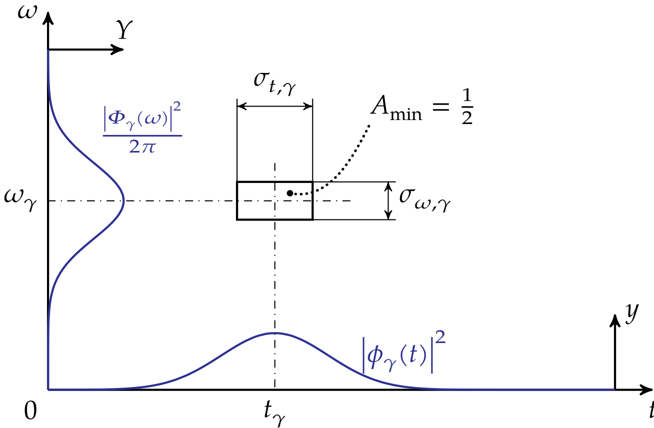


Figure 8.1: The Heisenberg box associated with an atom $\phi_\gamma(t)$

Using the same parallelism, we can define the position of the atom in the frequency domain as

$$\omega_\gamma = \frac{1}{2\pi}\int_{-\infty}^{+\infty} \omega \left|\Phi_\gamma(\omega)\right|^2 \,\mathrm{d}\omega$$

and the variance of the atom as

$$\sigma^2_{\omega,\gamma} = \frac{1}{2\pi}\int_{-\infty}^{+\infty} (\omega - \omega_\gamma)^2 \left|\Phi_\gamma(\omega)\right|^2 \,\mathrm{d}\omega.$$

The square root of the variance, a.k.a. the standard deviation $\sigma_{\omega,\gamma}$, then can be considered as a measure for the width of the atom in the frequency domain.

8.1.3 Heisenberg boxes

We will not put any effort in it, but one can prove that

$$\sigma_{t,\gamma}\sigma_{\omega,\gamma} \geq \frac{1}{2}$$

This is the so-called *Heisenberg uncertainty principle*. The name has been borrowed from quantum mechanics, where it describes the a lower limit to our ability of determining the position and the momentum (velocity vector) of a particle at the same time.

Here it shows us that we cannot keep shrinking the width of an atom in time, without allowing it to dilate in frequency and vice versa.

Graphically, a *Heisenberg box* can be associated with an atom, and the box has a lower boundary on the area of 1/2 that we cannot cross. This has been illustrated in Figure 8.1. However, note that the time-frequency atom is *not* a two-dimensional function of the pair (t, ω).

8.2 Linear time-frequency transforms

Now, let's consider an entire family of time-frequency atoms, a so-called *dictionary of atoms*

$$\mathcal{A} = \{\phi_\gamma(t)\} \tag{8.10}$$

with γ a multiparameter index that consists of a time and a frequency part.

Definition This dictionary defines a *linear time-frequency transform*:

Linear time-frequency transform

The time-frequency transform is denoted by

$$f(t) \xrightarrow{\mathcal{T}} \mathcal{T}(f(t)) = F(\gamma)$$

with

$$F(\gamma) = \langle f, \phi_\gamma \rangle = \int_{-\infty}^{+\infty} f(t)\phi_\gamma^*(t)\,\mathrm{d}t \tag{8.11}$$

Why is this called a linear time-frequency transform? It is linear because of the definition based on linear projection. It is a time-frequency transform because the paramter γ is a combined representation of time and frequency.

If γ allows it, F(□) is displayed by graphing its energy value for every value of γ, i.e. by considering

$$|F(\gamma)|^2$$

The resulting graph is a so called *spectrogram*.

Energy Assuming that the Fourier transforms of f and ϕ_γ exist, with $F(\omega) = \mathcal{F}(f(t))$ and $\Phi_\gamma(\omega) = \mathcal{F}(\phi_\gamma(t))$, the Parseval equation states that the integral of the right-hand side of (8.11) can be computed in the frequency domain as well:

$$\int_{-\infty}^{+\infty} f(t)\phi_\gamma^*(t)\,\mathrm{d}t = \frac{1}{2\pi}\int_{-\infty}^{+\infty} F(\omega)\Phi_\gamma^*(\omega)\,\mathrm{d}\omega$$

Remarks

Translation invariant dictionaries Consider an arbitrary atom $\phi_\gamma(t)$ in a dictionary. If, given this atom, all translated versions of the atom $\phi_\gamma(t-\tau)$ up until a multiplicative constant $k_{\tau,\gamma}$ are in the dictionary also, then we call the dictionary *translation invariant*.
In symbols:

$$\mathcal{A} \text{ is translation invariant } \Leftrightarrow \forall \phi_\gamma(t) \in \mathcal{A}, \forall \tau \in \mathbb{R} : \exists k_{\tau,\gamma} \in \mathbb{C} : k_{\tau,\gamma}\phi_\gamma(t-\tau) \in \mathcal{A}$$

Scaling invariant dictionaries Consider an arbitrary atom $\phi_\gamma(t)$ in a dictionary. If, given this atom, all scaled versions of the atom $\frac{1}{\sqrt{s}}\phi_\gamma\left(\frac{t}{s}\right)$ are in the dictionary also, then we call the dictionary *scale invariant*.
In symbols:

$$\mathcal{A} \text{ is scale invariant } \Leftrightarrow \forall \phi_\gamma(t) \in \mathcal{A}, \forall s \in \mathbb{R}_0^+ : \frac{1}{\sqrt{s}}\phi_\gamma\left(\frac{t}{s}\right) \in \mathcal{A}$$

The factor $\frac{1}{\sqrt{s}}$ ensures that when the original atom is of unit length ($\left\|\phi_\gamma(t)\right\|$), the derived atom is also of unit length. Indeed:

$$\left\|\frac{1}{\sqrt{s}}\phi_\gamma\left(\frac{t}{s}\right)\right\|^2 = \int_{-\infty}^{+\infty}\left|\frac{1}{\sqrt{s}}\phi_\gamma\left(\frac{t}{s}\right)\right|^2\mathrm{d}t = \frac{1}{s}\int_{-\infty}^{+\infty}\left|\phi_\gamma\left(\frac{t}{s}\right)\right|^2\mathrm{d}t \underset{u=\frac{t}{s}}{=} \int_{-\infty}^{+\infty}\left|\phi_\gamma(u)\right|^2\mathrm{d}u = \left\|\phi_\gamma(t)\right\|^2 = 1$$

Translation and scaling invariant dictionaries Consider an arbitrary atom $\phi_\gamma(t)$ in a dictionary. If, given this atom, all scaled translated versions of the atom $\frac{1}{\sqrt{s}}\phi_\gamma\left(\frac{t-\tau}{s}\right)$ up until a multiplicative constant are in the dictionary also, then we call the dictionary *scale and translation invariant*.
In symbols:

$$\mathcal{A} \text{ is scale and translation invariant } \Leftrightarrow \forall \phi_\gamma(t) \in \mathcal{A}, \forall \tau \in \mathbb{R}, \forall s \in \mathbb{R}_0^+, \exists k_{\tau,\gamma} \in \mathbb{C} : \frac{k_{\tau,\gamma}}{\sqrt{s}}\phi_\gamma\left(\frac{t-\tau}{s}\right) \in \mathcal{A}$$

The factor $\frac{1}{\sqrt{s}}$ once again ensures that when the original atom is of unit length ($\|\phi_\gamma(t)\|$), the derived atom is also of unit length.

8.3 Generating Dictionaries

Often, the dictionary $\mathcal{A}$ is generated starting from a a single (or a number of) atomic template function(s) $\phi_\gamma(t)$, the so-called generator function(s).

The dictionary is then generated using a fixed *generation principle*. The principle is chosen often as to guarantee a number of desirable properties. Very common are:

- Translation invariant generation:
  Fill the dictionary with all $\phi_\gamma(t)$ and complete it with $k_{\tau,\gamma}\phi_\gamma(t-\tau)$ for all relevant values of τ.
- Scale invariant generation:
  Fill the dictionary with all $\phi_\gamma(t)$ and complete it with scaled versions of the generator functions, i.e. $\frac{1}{\sqrt{s}}\phi_\gamma(\frac{t}{s})$ for all relevant values of s.
- Scale and translation invariant generation:
  Fill the dictionary with all $\phi_\gamma(t)$ and complete it with translated scaled versions of the generator functions, i.e. $\frac{k_{\tau,\gamma}}{\sqrt{s}}\phi_\gamma(\frac{t-\tau}{s})$ for all relevant values of s and τ.

8.4 Gabor's short-time Fourier transform

8.4.1 Forth...

Dennis Gabor (a Hungarian electrical engineer, inventor of holography, for which he received the Nobel Prize in Physics), proposed to use the family of windowed complex Fourier kernel as generator:

$$\phi_\xi(t) = w(t)\,\mathrm{e}^{\mathrm{j}\xi t}$$

with $\xi \in \mathbb{R}$ and $w(t)$ a normalized real symmetric window function, centered at $t = 0$. This means:

$$w(t) = w(-t)$$
$$\|w(t)\|^2 = \int_{-\infty}^{+\infty} |w(t)|^2 \,\mathrm{d}t = 1 \tag{8.12}$$

The window function limits the Fourier kernel to a short time span in which it is active.

The corresponding dictionary is then generated using the *translation invariant generation principle*, i.e.

$$\mathcal{A} = \{\phi_{\tau,\xi}(t)\}$$

with

$$\begin{aligned}\phi_{\tau,\xi}(t) &= k_{\tau,\xi}\phi_{\xi}(t-\tau)\\ &\downarrow\ k = e^{j\xi\tau}\\ &= e^{j\xi\tau}w(t-\tau)\,e^{j\xi(t-\tau)}\\ &= w(t-\tau)\,e^{j\xi t}\end{aligned}$$

In short, the window is being translated, but the frequency kernel is not.

Note, that in the above the parameter γ that was introduced in (8.2), now is being represented by the parameter combo (τ,ξ). The parameter τ represents the time at which the window is centered, and ξ the frequency at which it is centered.[2]

The resulting dictionary leads to the definition of the so-called (continuous time) *short-time Fourier transform*, a.k.a the *Gabor transform*. This transform is just a particular case of the more general *linear time-frequency transforms*.

Short-time Fourier transform

The short-time Fourier transform is denoted as:

$$f(t) \xrightarrow{\mathcal{S}} \mathcal{S}(f(t)) = F(\tau,\xi)$$

with

$$F(\tau,\xi) = \langle f, \phi_{\tau,\xi}\rangle = \int_{-\infty}^{+\infty} f(t)w(t-\tau)\,e^{-j\xi t}\,dt.$$

Often the resulting two-dimensional complex function of $F(\tau,\xi)$ is displayed by graphing its energy value, i.e. by considering

$$|F(\tau,\xi)|^2$$

The resulting graph is a so-called *spectrogram*.

8.4.2 Intermezzo: its Heisenberg boxes

Before diving head over heels into the Heisenberg-sea, let's check how Gabor's dictionary $\mathcal{A} = \{\phi_{\tau,\xi}(t)\}$, looks like in the frequency domain.
Therefore, let's calculate $\Phi_{\tau,\xi}(\omega) = \mathcal{F}(\phi_{\tau,\xi}(t))$.

$$\begin{aligned}\Phi_{\tau,\xi}(\omega) &= \int_{-\infty}^{+\infty} \phi_{\tau,\xi}(t)\,e^{-j\omega t}\,dt\\ &= \int_{-\infty}^{+\infty} w(t-\tau)\,e^{j\xi t}\,e^{-j\omega t}\,dt = \int_{-\infty}^{+\infty} w(t-\tau)\,e^{-j(\omega-\xi)t}\,dt\\ &\downarrow\ \text{substitute } u = t-\tau\\ &= e^{-j(\omega-\xi)\tau}\int_{-\infty}^{+\infty} w(u)\,e^{-j(\omega-\xi)u}\,du\\ &= e^{-j(\omega-\xi)\tau}W(\omega-\xi)\end{aligned}$$

[2] We selected these symbols to maximize our ability to remember them: the Greek letter τ is a variant of our western letter t, and the ξ looks a bit like a rotated ω.

with $W(\omega) = \mathcal{F}(w(t))$. Note that given the fact that $w(t)$ is real and symmetric, so is $W(\omega)$.

So far, so good. Now, back to the boxes.
Gabor's Heisenberg boxes can be easily determined.

Position The box corresponding to $\phi_{\tau,\xi}(t)$ is centered at τ and ξ respectively.
Indeed:

$$\begin{aligned} t_{\tau,\xi} &= \int_{-\infty}^{+\infty} t \left|\phi_{\tau,\xi}(t)\right|^2 \,\mathrm{d}t \\ &= \int_{-\infty}^{+\infty} t \left|w(t-\tau)\,\mathrm{e}^{\mathrm{j}\xi t}\right|^2 \,\mathrm{d}t \\ &= \int_{-\infty}^{+\infty} t \left|w(t-\tau)\right|^2 \left|\mathrm{e}^{\mathrm{j}\xi t}\right|^2 \,\mathrm{d}t \\ &\downarrow \quad \left|\mathrm{e}^{\mathrm{j}\xi t}\right| = 1 \text{ and substitute } u = t-\tau \\ &= \int_{-\infty}^{+\infty} (u+\tau) \left|w(u)\right|^2 \,\mathrm{d}u \\ &= \underbrace{\int_{-\infty}^{+\infty} u \left|w(u)\right|^2 \,\mathrm{d}u}_{=0 \text{ (iff } w(u) \text{ is even)}} + \tau \underbrace{\int_{-\infty}^{+\infty} \left|w(u)\right|^2 \,\mathrm{d}u}_{=1} = \tau \end{aligned}$$

The final step could be made in view of the fact that the first term is a symmetrical integral of an odd function (this only holds if the window function is even!) and the second term contains (8.12).
In case $w(u)$ is not even, the first term requires explicit calculation.

A similar reasoning can be made in the frequency domain:

$$\begin{aligned} \omega_{\tau,\xi} &= \int_{-\infty}^{+\infty} \omega \frac{\left|\Phi_{\tau,\xi}(\omega)\right|^2}{2\pi} \,\mathrm{d}\omega \\ &\downarrow \quad \text{see} \\ &= \int_{-\infty}^{+\infty} \omega \frac{\left|\mathrm{e}^{-\mathrm{j}(\omega-\xi)\tau} W(\omega-\xi)\right|^2}{2\pi} \,\mathrm{d}\omega \\ &= \int_{-\infty}^{+\infty} \omega \frac{\left|W(\omega-\xi)\right|^2}{2\pi} \,\mathrm{d}\omega = \xi \end{aligned}$$

The final step could be deduced by realizing that $|W(\omega-\epsilon)|^2/2\pi$ is a proper probability density function and even w.r.t. symmetry axis $\omega = \xi$.

Size The box corresponding to $\phi_{\tau,\xi}(t)$ has sizes in the time and frequency domain that are independent of τ and ξ.
The variance in the time domain can be calculated to be

$$\begin{aligned} \sigma^2_{t,\tau,\xi} &= \int_{-\infty}^{+\infty} (t-\tau)^2 \left|\phi_{\tau,\xi}(t)\right|^2 \,\mathrm{d}t \\ &= \int_{-\infty}^{+\infty} (t-\tau)^2 \left|w(t-\tau)\,\mathrm{e}^{\mathrm{j}\xi t}\right|^2 \,\mathrm{d}t \\ &\downarrow \quad \text{substitute } u = t-\tau \\ &= \int_{-\infty}^{+\infty} u^2 \left|w(u)\,\mathrm{e}^{\mathrm{j}\xi(u+\tau)}\right|^2 \,\mathrm{d}u \\ &= \int_{-\infty}^{+\infty} u^2 \left|w(u)\right|^2 \,\mathrm{d}u \end{aligned}$$

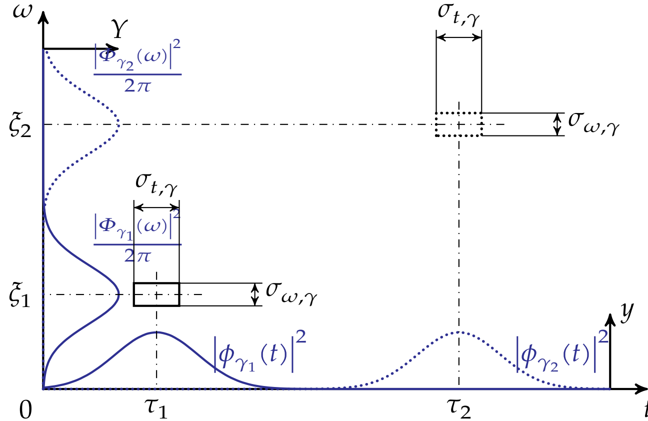


Figure 8.2: The Heisenberg boxes associated with the short-time Fourier transform are constant in size throughout the entire time-frequency plane. This has been illustrated for two points in the plane $\gamma_1 = (\tau_1, \xi_1)$ and $\gamma_2 = (\tau_2, \xi_2)$.

and therefore is dependent on the actual shape of $w(t)$, but is *independent of* τ *and* ξ. Therefore, we usually denote it by σ_t^2.

The variance in the frequency domain can be calculated to be

$$\begin{aligned}
\sigma^2_{\omega,\tau,\xi} &= \int_{-\infty}^{+\infty} (\omega-\xi)^2 \frac{\left|\Phi_{\tau,\xi}(\omega)\right|^2}{2\pi}\,\mathrm{d}\omega \\
&\downarrow \text{ see} \\
&= \int_{-\infty}^{+\infty} (\omega-\xi)^2 \frac{\left|\mathrm{e}^{-\mathrm{j}(\omega-\xi)\tau} W(\omega-\xi)\right|^2}{2\pi}\,\mathrm{d}\omega \\
&= \int_{-\infty}^{+\infty} (\omega-\xi)^2 \frac{|W(\omega-\xi)|^2}{2\pi}\,\mathrm{d}\omega \\
&\downarrow \text{ substitute } \lambda = \omega - \xi \\
&= \int_{-\infty}^{+\infty} \lambda^2 \frac{|W(\lambda)|^2}{2\pi}\,\mathrm{d}\lambda.
\end{aligned}$$

Again, the variance proves to be dependent on the actual shape of $w(t)$, but *independent of* τ *and* ξ. Therefore, we usually denote it by σ_ω^2.

Conclusion The size of the Heisenberg boxes of the short-time Fourier transform is independent of their position. Stated differently: the resolution of the transform is constant throughout the entire time-frequency plane. This has been illustrated in Figure 8.2.

8.4.3 ...and back

The Heisenberg boxes cover the entire time-frequency plane. Therefore, it seems that all information is present to accomplish the inverse transformation. Indeed, one can prove that the inverse short-time Fourier transformation exists.

Inverse short-time Fourier transform

Given

$$f(t) \xrightarrow{\mathcal{S}} F(\tau,\xi)$$

the following holds:

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} F(\tau,\xi)w(t-\tau)\,\mathrm{e}^{\mathrm{j}\xi t}\,\mathrm{d}\xi\,\mathrm{d}\tau$$

In addition, one can prove that the energy is conserved in the process:

$$\int_{-\infty}^{+\infty} |f(t)|^2\,\mathrm{d}t = \frac{1}{2\pi}\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} |F(\tau,\xi)|^2\,\mathrm{d}\xi\,\mathrm{d}\tau$$

8.5 Cutting Gabor's elephant to pieces

The continuous-time short-time Fourier transform is a big shot. When staying on a theoretic level, working the integrals is no big deal. Everything works out beautifully. However, when trying to deal with real-life complicated functions and windows, calculating all these integrals is a daunting job.

Let's cut this elephant to pieces by discretizing time.[3]

The basis is a windowed complex Fourier kernel as generator:

$$\phi_\kappa[n] = w[n]\,\mathrm{e}^{\mathrm{j}\frac{2\pi}{N}\kappa n}$$

with $\kappa \in \mathbb{Z}$ and $w[n]$ a symmetrical window. This means:

$$w[n] = w[-n]$$
$$\|w[n]\|^2 = \sum_n w^2[n] = 1$$

The generator in combination with *the translation invariant generation principle*, yields the following dictionary:

$$\mathcal{A} = \left\{\phi_{\eta,\kappa}[n]\right\}$$

with

$$\phi_{\eta,\kappa}[n] = w[n-\eta]\,\mathrm{e}^{\mathrm{j}\frac{2\pi}{N}\kappa n}.$$

Again, note, that in the above the parameter γ that was introduced in (8.2), now is being represented by the integer parameter combo (η,κ). The parameter η represents the discrete time index at which the window is centered, and κ the discrete frequency index at which it is centered.[4]

[3] Note that this is strictly a figure of speech: shooting and cutting elephants to pieces is strictly prohibited as the elephant (and especially Gabor's elephant) is an endangered species.

[4] We selected these symbols to maximize our ability to remember them: the Greek letter η looks a bit like our western letter n and the κ looks a bit like our western letter k.

Exercises

Exercise 8.5-1: Which of the following functions qualify as window functions for the Gabor generator function $w[n] \cdot \mathrm{e}^{\mathrm{j}\frac{2\pi}{N}\kappa n}$. The value for $n = 0$ has been circled.

- $f[n] = \begin{bmatrix} 1 & \circled{2} & 1 \end{bmatrix}$
- $g[n] = \begin{bmatrix} -\frac{1}{\sqrt{6}} & \circled{\frac{\sqrt{2}}{\sqrt{3}}} & -\frac{1}{\sqrt{6}} \end{bmatrix}$
- $h[n] = \begin{bmatrix} 0.2 & 0.8 & \circled{0.9} & 1.0 & 0.9 & 0.8 & 0.2 \end{bmatrix}$

Why (not)?
Can you correct the ones that do not qualify?

Exercise 8.5-2: Use the Blackman window function to generate a proper window (of length 7) to use in the Gabor generator function.

Exercise 8.5-3: Write some OCTAVE/MATLAB function to generate a proper Hamming window of arbitrary length (specified as a parameter to the function).

Exercise 8.5-4: Make the function of the previous exercise more generic by specifying the window type (Hamming in the case above) as a parameter.

8.5.1 Forth...

Short-time discrete Fourier transform

The short-time discrete Fourier transform is denoted as:

$$f[n] \xrightarrow{\mathcal{S}} \mathcal{S}(f[n]) = F(\eta, \kappa)$$

with

$$F(\eta, \kappa) = \langle f, \phi_{\eta,\kappa} \rangle = \sum_n f[n] w[n - \eta] \, \mathrm{e}^{-\mathrm{j}\frac{2\pi}{N}\kappa n} \tag{8.13}$$

In this definition, we used a sloppy summation index notation to keep things simple and robust w.r.t. to the remarks that are made below. It means: sum over all relevant values of n. Note that calculating (8.13) corresponds to calculating the DFT/FFT of the original function, multiplied with an appropriate window.

Remarks

1. **Time-limited windows** For practical purposes, the window will be in most cases time-limited (finite support), i.e.

$$w[n] = 0, n < 0 \text{ or } n \geq N$$

2. **Symmetry** In view of this aberrant definition, the symmetry constraint needs to be reformulated as
$$w[n] = w[N-1-n].$$
The symmetry center point is then located halfway at $n = (N-1)/2$. This will introduce an extra time shift, that we need to be aware of, when making conclusions in the time-frequency domain.
3. **Computational complexity** Note that the calculation of (8.13) requires the calculation of an FFT for every value of η. An FFT requires $N \log_2 N$ calculations. Therefore, calculating the Gabor transform for an M-point signal (with usually $M \geq N$) requires $MN \log_2 N$) additions and multiplications.[5]

8.5.2 Intermezzo: its Heisenberg boxes

Again, let's start by taking a look at the dictionary in the frequency domain. In a similar way to the continuous case, we can derive:

$$\Phi_{\eta,\kappa}[k] = \text{DFT}\left(\phi_{\eta,\kappa}[n]\right) = \mathrm{e}^{-\mathrm{j}\frac{2\pi}{N}(k-\kappa)\eta} W[k-\kappa] \tag{8.14}$$

with $W[k] = \text{DFT}\,(w[n])$. As $w[n]$ is real, $W[k]$ is hermitic. Note that in case $w[n]$ is symmetrical w.r.t. $n = 0$, that $W[k]$ is also real.

The starting point is very similar to the continuous case: we borrow the calculation of position and size from probability theory. The expected value (the 'mean') and the variance of a discrete random variable $\mathcal{N}$ with probability mass function $f_{\mathcal{N}}[n]$, are defined as:

$$\alpha_{1,\mathcal{N}} = E\{\mathcal{N}\} = \sum_n n f_{\mathcal{N}}[n]$$
$$\sigma^2_{\mathcal{N}} = \sum_n \left(n - \alpha_{1,\mathcal{N}}\right)^2 f_{\mathcal{N}}[n]$$

The expected value corresponds to the position of the probability mass function; the variance indicates how 'wide' it is. We often refer to $f_{\mathcal{N}}[n]$ as the *PMF*.[6]

In our case the PMF in the time domain is:

$$f_{\mathcal{N}}[n] = \left|\phi_{\eta,\kappa}[n]\right|^2$$

And given (8.14) and Parseval's theorem, the PMF in the frequency domain equals:

$$F_{\mathcal{N}}[k] = \frac{1}{N}\left|\Phi_{\eta,\kappa}[k]\right|^2$$

[5] We neglected the multiplication of the signal with the weight function $w[n]$. For realistic values of N this effect is negligible.

[6] As the time scale is no longer continuous, we cannot speak of a distribution, the probability is concentrated (as a mass) in the time points.

Position in the time domain We start from the definition of position of an atom in time:

$$
\begin{aligned}
n_{\eta,\kappa} &= \sum_n n \left| w[n-\eta]\, \mathrm{e}^{\mathrm{j}\frac{2\pi}{N}\kappa n} \right|^2 \\
&= \sum_n n\, |w[n-\eta]|^2 \underbrace{\left| \mathrm{e}^{\mathrm{j}\frac{2\pi}{N}\kappa n} \right|^2}_{=1} \\
&= \sum_n n\, |w[n-\eta]|^2 \\
&\Big\downarrow \quad \text{substitution: } \nu = n - \eta \\
&= \sum_\nu (\nu + \eta)\, |w[\nu]|^2 \\
&= \sum_\nu \nu\, |w[\nu]|^2 + \eta \underbrace{\sum_\nu |w[\nu]|^2}_{=1(\text{ (by definition)}}
\end{aligned}
$$

In case of a symmetric window, the first term equals zero. In this case:

$$n_{\eta,\kappa} = \eta$$

In case of a right-winged window, i.e. nonzero for $n = 0 \ldots N-1$, the first term, needs more careful investigation:

$$
\begin{aligned}
\sum_\nu \nu\, |w[\nu]|^2 = {} & 0 \cdot |w[0]|^2 + 1 \cdot |w[1]|^2 + 2 \cdot |w[2]|^2 \\
& + \cdots \\
& + (N-3) \cdot |w[N-3]|^2 + (N-2) \cdot |w[N-2]|^2 + (N-1) \cdot |w[N-1]|^2
\end{aligned}
$$

Combining the terms connected by the lines, and realizing that the window is symmetric, w.r.t. $(N-1)/2$, i.e. $|w[i]|^2 = |W[N-1-i]|^2, \forall i$ leads to:

$$
\begin{aligned}
\sum_\nu \nu\, |w[\nu]|^2 &= (N-1) \cdot |w[0]|^2 + (N-1) \cdot |w[1]|^2 + (N-1) \cdot |w[2]|^2 + \cdots + (N-1)\, |w[N/2-1]|^2 \\
&= (N-1) \cdot \left(\underbrace{|w[0]|^2 + |w[1]|^2 + |w[2]|^2 + \cdots |w[N/2-1]|^2}_{=1/2} \right) \\
&= \frac{N-1}{2}
\end{aligned}
$$

In the derivation above, we assumed N to be even. A similar derivation can be made for N odd, also resulting in:

$$n_{\eta,\kappa} = \frac{N-1}{2} + \eta$$

Position in the frequency domain Now, let's investigate the position in the frequency domain:

$$
\begin{aligned}
k_{\eta,\kappa} &= \sum_k k \frac{\left| e^{-j\frac{2\pi}{N}(k-\kappa)\eta} W[k-\kappa] \right|^2}{N} \\
&= \sum_k k \underbrace{\left| e^{-j\frac{2\pi}{N}(k-\kappa)\eta} \right|^2}_{=1} \frac{|W[k-\kappa]|^2}{N} \\
&\downarrow \text{ substitution: } \beta = k - \kappa \\
&= \sum_\beta (\beta + \kappa) \frac{|W[\beta]|^2}{N} \\
&= \sum_\beta \beta \frac{|W[\beta]|^2}{N} + \kappa \sum_\beta \frac{|W[\beta]|^2}{N}
\end{aligned}
$$

The first term is zero, as $|W[\beta]|$ is symmetrical w.r.t. $\beta = 0$ given the hermiticity of $W[\beta]$. The summation factor in the second term equals 1 given $\sum_n |w[n]|^2 = 1$ and Parseval's theorem.

Therefore:

$$
k_{\eta,\kappa} = \kappa
$$

Size in the time domain We start from the definition of the size of an atom in the time domain:

$$
\begin{aligned}
\sigma^2_{n,\eta,\kappa} &= \sum_n (n-\eta)^2 \left| w[n-\eta]\, e^{j\frac{2\pi}{N}\kappa n} \right|^2 \\
&= \sum_n (n-\eta)^2 |w[n-\eta]|^2 \underbrace{\left| e^{j\frac{2\pi}{N}\kappa n} \right|^2}_{=1} \\
&\downarrow \text{ subsitution: } \nu = n - \eta \\
&= \sum_\nu \nu^2 |w[\nu]|^2
\end{aligned}
$$

We can see clearly that the size of the atom is constant, independent of η and κ. Only the shape of window function determines the size of the full atom.

Note: in calculating the size, we assumed a symmetric window w.r.t. $n = 0$. In case the window is not symmetric, a similar reasoning can be made (see also exercise 8.5.2-3).

Size in the frequency domain Let's start once more from the definition of the size of an atom in the frequency domain:

$$\begin{aligned}
\sigma^2_{k,\eta,\kappa} &= \sum_k (k-\kappa)^2 \frac{\left| e^{-j\frac{2\pi}{N}(k-\kappa)\eta} W[k-\kappa] \right|^2}{N} \\
&= \sum_k (k-\kappa)^2 \underbrace{\left| e^{-j\frac{2\pi}{N}(k-\kappa)\eta} \right|^2}_{=1} \frac{|W[k-\kappa]|^2}{N} \\
&= \sum_k (k-\kappa)^2 \frac{|W[k-\kappa]|^2}{N} \\
&\downarrow \quad \text{substitution: } \beta = k - \kappa \\
&= \sum_\beta \beta^2 \frac{|W[\beta]|^2}{N}
\end{aligned}$$

Again, we can clearly see that the size of the atom in frequency is constant, independent of η and κ. Only the magnitude spectrum of the window function determines the size of the full atom in the frequency domain.

Conclusion The size of the Heisenberg boxes of the short-time Fourier transform is independent of their position. Stated differently: the resolution of the transform is constant throughout the entire time-frequency plane.

Exercises
Some exercises on calculating the position and size of the Heisenberg boxes of the STDFT.

Exercise 8.5.2-1: Calculate the time position and size of the Gabor atoms that use the following window function:

$$w[n] = \begin{bmatrix} \frac{1}{\sqrt{6}} & \boxed{-\frac{\sqrt{2}}{\sqrt{3}}} & \frac{1}{\sqrt{6}} \end{bmatrix}$$

Exercise 8.5.2-2: Calculate the frequency position and size of Gabor atoms that use the following window function:

$$w[n] = \begin{bmatrix} \frac{1}{\sqrt{6}} & \boxed{\frac{\sqrt{2}}{\sqrt{3}}} & \frac{1}{\sqrt{6}} \end{bmatrix}$$

Exercise 8.5.2-3: Calculate the time position and size of the Gabor atoms that use a right-winged Bartlett window of length 7.

Exercise 8.5.2-4: Calculate the frequency position and size of the Gabor atoms that use a right-winged Bartlett window of length 7.

Exercise 8.5.2-5: (*) Write a OCTAVE/MATLAB script to calculate the time position and size of an arbitrary window function. Make your function take a single parameter, the window vector.

Exercise 8.5.2-6: (*) Write a Octave/Matlab script to calculate the frequency position and size of an arbitrary window function. Make your function take a single parameter, the window vector.

Exercise 8.5.2-7: (*) Combine the window generation function you wrote for exercise 8.5-4 with the functions you wrote for the two previous exercises to create a function that generates a window, calculates its size in time and frequency and returns the vector and the size information.

8.5.3 ...and back

The Heisenberg boxes cover the entire time-frequency plane. Therefore, it seems that all information is present to accomplish the inverse transformation. Indeed, one can prove that the inverse short-time Fourier transformation exists.

Inverse short-time discrete Fourier transform

Given

$$f[n] \xrightarrow{\mathcal{S}} F(\eta, \kappa)$$

the following holds:

$$\begin{aligned} f[n] &= \frac{1}{N} \sum_{\kappa} \sum_{\eta} F(\eta, \kappa) w(n-\eta)\, e^{j\frac{2\pi}{M}\kappa n} \\ &= \frac{1}{N} \sum_{\eta} w(n-\eta) \sum_{\kappa} F(\eta, \kappa)\, e^{j\frac{2\pi}{M}\kappa n}. \end{aligned} \tag{8.15}$$

Once again, we used a sloppy summation index notation to keep things simple and robust. It means: sum over all relevant values of η or κ.
Note that calculating (8.15) corresponds to calculating the inverse DFT/FFT of the original function and gathering those results in a weighted sum for all relevant η.

In addition, one can prove that the energy is conserved in the process:

$$\sum_{n} |f[n]|^2 = \frac{1}{N} \sum_{\kappa} \sum_{\eta} |F[\eta, \kappa]|^2$$

Remarks

1. **Computational complexity** Note that the calculation of (8.15) requires the calculation of an FFT for every value of η. An FFT requires $N \log_2 N$ calculations. Therefore, calculating the inverse Gabor transform for an M-point signal (with usually $M \geq N$) requires $MN \log_2 N$ additions and multiplications.[7]
2. **Skipping values of η** Sometimes the short-time Fourier transform is only calculated for a decimated amount of values of η, e.g., $\eta = 0, 4, 8, \ldots$. It should be noted that this still gives useful information on the time-frequency content of the signal, but the inverse transform, will no longer regain the original function.

[7] Again, we neglected the multiplication of the signal with the weight function $w[n]$. For realistic values of N this effect is negligible.

8.6 Examples

8.6.1 Analyzing an arbitrary signal

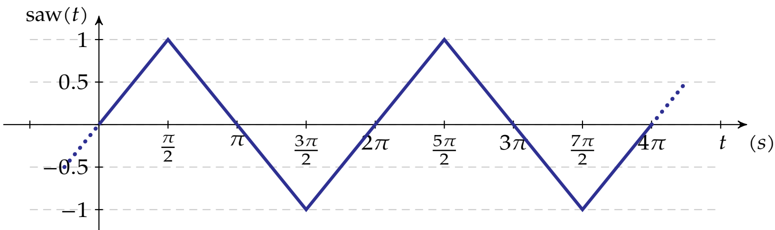
Consider the following example signal $f(t)$, consisting of

- a chirp[8] that rises quadratically from 100 to 800 Hz on an interval of 4 s,
- a chirp that falls linearly from 600 to 200 Hz on an interval of 4 s,
- a square pulse lasting from 0.5 s to 0.75 s,
- a sine wave lasting from 1.5 s to 2.25 s, and
- a saw-tooth wave lasting from 2.75 s to 3.5 s.

Mathematically:

$$
\begin{aligned}
f(t) = \sin\left(2\pi\left(\frac{700}{48}t^3 + 100t\right)\right) \\
+ 0.5\sin\left(2\pi\left(-50t^2 + 600t\right)\right) \\
+ u(t-0.5) - u(t-0.75) \\
+ [u(t-1.5) - u(t-2.25)]\sin(2\pi 800t) \\
+ [u(t-2.75) - u(t-3.5)]\,\mathrm{saw}(2\pi 400t)
\end{aligned}
\tag{8.16}
$$

with saw t a periodic wave with period 2π, oscillating sine-like between -1 and 1 as illustrated below:



We measured this signal using a data acquisition card, and therefore had to sample this signal from $t = 0$ s to $t = 4$ s with $f_s = 2\,048$ Hz.

The time-domain waveform of this sampled signal (see Figure 8.3a) does not reveal much about the nature of the signal, except for the pulse that is slightly visible.

However, when applying the short-time discrete Fourier transform to this signal, the entire composition becomes crystal clear. We used a Blackman window of length $N = 128$, leading to obtain the result of Figure 8.3b.

The scripts used to generate this result, can be found below. The short-time discrete Fourier transform is implemented as a function:

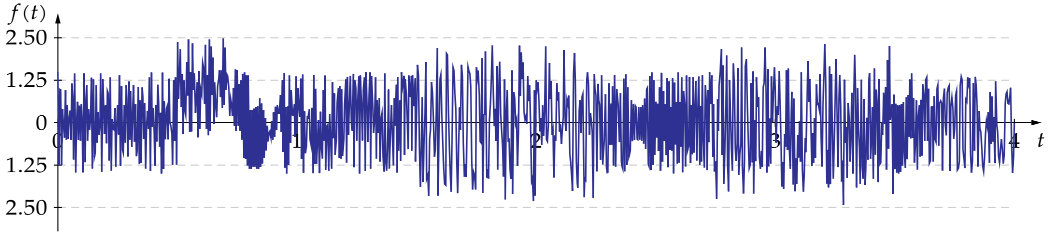
```
function A = stft( x, window, skipfactor )
  % STFT - short time fourier transform
  %
  % PARAMETERS
```

[8] A chirp is a sine wave with changing frequency.

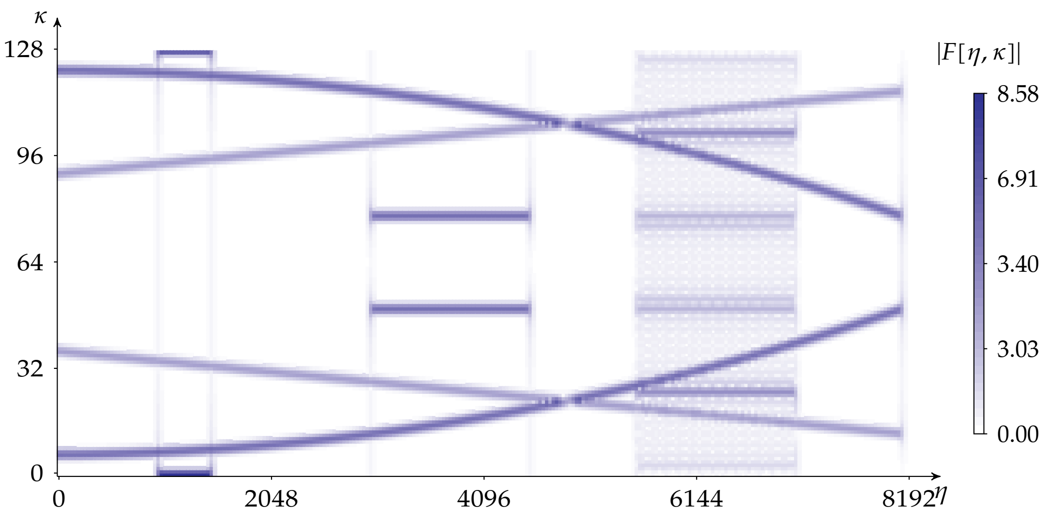
```
  %   - x: matrix containing signal with
  %     - time as first dimension
  %     - multiple channels in the second dimension
  %   - window: vector containing a proper window (the length and
  %       shape of this window also defines the time-frequency
  %       resolution)
  %   - skipfactor: factor causing decimation in time in the end
  %       result
  %
  % RETURN VALUE
  %   - matrix A containing stft with
  %     - time as first dimension
  %     - frequency as second dimension
  %     - channels as second dimension
  % (C) 2015 W. Daems

  % normalize window
  window = window / norm(window);

  % determine dimensions
  N = length(window);
  [M,C] = size(x);
  MM = ceil(M/skipfactor);

  % pad the signal
  % - prepend the signal with zeros, half a window length
  %   such that the first window is applied symmetrical
  %   w.r.t. n = 0
  x = [ zeros(N/2,1); x ];
  % - append zeros, usch that the last window can be applied
  %   without generating an input signal out of bounds
  x(MM*skipfactor+N/2,C) = 0;

  % generate empty stft matrix
  A = zeros(MM,N,C);

  % calculate STFT
  for i = 1:MM
    k = (i - 1) * skipfactor + 1;
    for j = 1:C
      z = x(k:k+N-1,j) .* window;
      A(i,:,j) = fft(z);
    end
  end
end
```

This function is used in the following test script:

```
% parameters related to time and definion a time scale t
fs = 2048;              % sampling rate
duration = 4;           % fragment length
t = linspace(0,4, fs*duration);

% the test signal
x = transpose(...
               chirp(t,100,4,800,'q')...
              + 0.5* chirp( t, 600, 4,200 )...
              + (t > 0.5 & t < 0.75) ...
              + (t > 1.5 & t < 2.25 ) .* sin (2*pi*800*t) ...
              + (t > 2.75 & t < 3.5 ) .* sawtooth(2*pi*400*t)...
             );


% saving the time signal (not in course notes)
f(:,1) = t.';
f(:,2) = x.';
ff = extremedecimate(f, 2, 8 );
save( 'f.dat', 'ff', '-ascii' );
```



(a) Time-domain representation of the example signal $f(t)$



(b) The time-frequency representation of the same signal, obtained by applying the short-time Fourier transform (Gabor transformation) using a Blackman window of length $N = 128$.

Figure 8.3: Illustration of the short-time Fourier transform (Gabor transformation) using the example signal of (8.16)

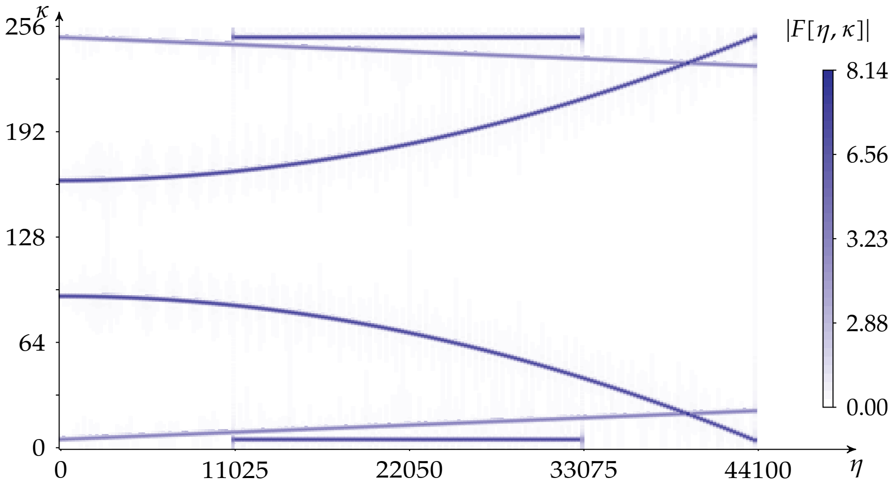
```
% create a blackman window for the StFt
N = 128;
window = blackman(N);

% we want to create a true StFt analysis, so no decimation in time
skipfactor = 1;

% calculate the StFt
X = stft( xzp, window, skipfactor );

% display it
pcolor( abs(X) );  % or surf(abs(X)) for a 3D view
```

Some questions to trigger you into exploring this case a bit further:

- You'll notice that it takes quite some computing power to display the result. Try experimenting with the skipfactor to improve the situation.
- Can you explain the reason for the zero-padding?
- Would you be able to write a function for the inverse short-time discrete Fourier transform?

Exercises
Some exercises on reading/interpreting STDFT spectrograms.

Exercise 8.6.1-1: Consider the spectrogram below.



Now, answer the following questions:

- How long was the window used to generate this spectrogram?
- If the signal that was analyzed is 1 s in length, then what was the sample frequency?
- From what time until what time does the constant sine wave last?
- There are some frequency sweeps (chirps) visible:
  - The linear one
  - A nonlinear one

  For both determine:
  - whether it is rising or falling
  - what the start frequency is
  - what the final frequency is
- Which frequency is denoted by $\kappa = 192$?

Exercise 8.6.1-2: (*) Can you think of a way to increase the frequency resolution, without increasing the size of the atom in time?

8.6.2 Creating a sound-level meter

A sound-level meter monitors the sound level of an audio source at different frequencies as a function of time.

In the old analog days, graphical sound-level meters were created using a fractional octave filter bank, with a band-pass filter for every frequency to monitor. Luckily, knowing the short-time discrete Fourier-transform and having quite some DSP computing power at our disposition, we can easily write a sound-level meter using very few lines of MATLAB code.

The main function `soundlevelplayer` that can be found below, reads in an audio file, hands it over to an audio player an attaches a callback function (`drawSoundLevel()` to the player, that will take care of displaying the sound levels.

```
function player = soundlevelplayer ( filename )
  % SOUNDLEVELPLAYER - play a music file accompanied by a sound level meter
  %
  % ARGUMENTS
  %   - musicfile: specifies the name of the audio file
  %
  % RETURN VALUE(s)
  %   audioplayer object that can be used to manipulate the player
  %
  % (C) 2015 W. Daems

  % read in audio data
  [ X, fs ] = audioread( filename );

  % parameters for STFT
  updaterate = 8;            % update rate in Hz
  window = @blackman;        % window used in the STFT
  Nbins = 25;                % number of logarithmic frequency bins

  % setup player
  player = audioplayer( X, fs );
  player.UserData = X;
  player.TimerPeriod = 1/updaterate;
  player.TimerFcn = {@drawSoundLevel, 1/updaterate, window, Nbins };

  % start player
  play(player);

end
```

The callback function is invoked every T seconds and applies a windowed FFT on the samples available in the coming T seconds. The quadratic norm of the resulting FFT is displayed, for a set of *nbins* logarithmic frequency bins.

```
function drawSoundLevel( player, event, T, window, nbins )
  % DRAWSOUNDLEVEL - timer call back function to draw sound level
  %
  % ARGUMENTS
  %    - player: audioplayer object
  %    - event : event that triggered this callback function (unused)
  %    - period: update period
  %
  % RETURN VALUE(s)
  %     none
  % (C) 2015 W. Daems

  persistent mywindow N Nold nbinsold freqs;

  % make sure the persistent variables get initialized/updated when required
```

```
  if isempty(N)
    Nold = 0;
  end

  if isempty(nbinsold)
    nbinsold = 0;
  end

  % setup window
  N = floor(player.SampleRate * T);
  if N ~= Nold
    mywindow = window(N);
    Nold = N;
  end

  % perform STFT
  i = player.CurrentSample;
  left = player.UserData(i:i+N-1,1);
  right = player.UserData(i:i+N-1,2);
  Xleft = abs(fft( left .* mywindow ));
  Xright = abs(fft( right .* mywindow ));

  % create logarithmic frequency bins
  if nbins ~= nbinsold
    freqs = floor(logspace( log10(1), log10(N/2), nbins + 1));
  end

  % gather total energy in bins
  for i = 1:nbins
    energy(nbins+1 + i) = norm( Xright( freqs(i):freqs(i+1) ) );
    energy(nbins+1 - i) = norm( Xleft( freqs(i):freqs(i+1) ) );
  end

  % plot graph
  bar( -(nbins):(nbins), energy, 'b' );
  axis manual;
  axis( [-nbins nbins 0 20 * sqrt(N / nbins) ] );
  title( 'Uncalibrated Sound Level Meter' );
  xlabel( '\kappa^*' );
  ylabel( '||F(\eta,\kappa^*)||   (\eta in real time)' );

end
```

A snapshot of the player in action can be found in Figure 8.4.
A feature that is missing, is the A-weighting of the spectrum? Can you add that to these scripts?
The script still offers quite a bit of room for improvement. Can you optimize it further?

8.7 Conclusion

Linear time-frequency transforms and more specifically the short-time Fourier transform is a powerful analysis tool to analyze signals in the combined time-frequency domain. It can be used as a transform, allowing full reconstruction of the original (e.g., after the application of some smart filtering operation in the time-frequency domain). It can also be used as an analysis tool in which case the reconstructability is less of an issue and the time-sampling rate can be reduced.

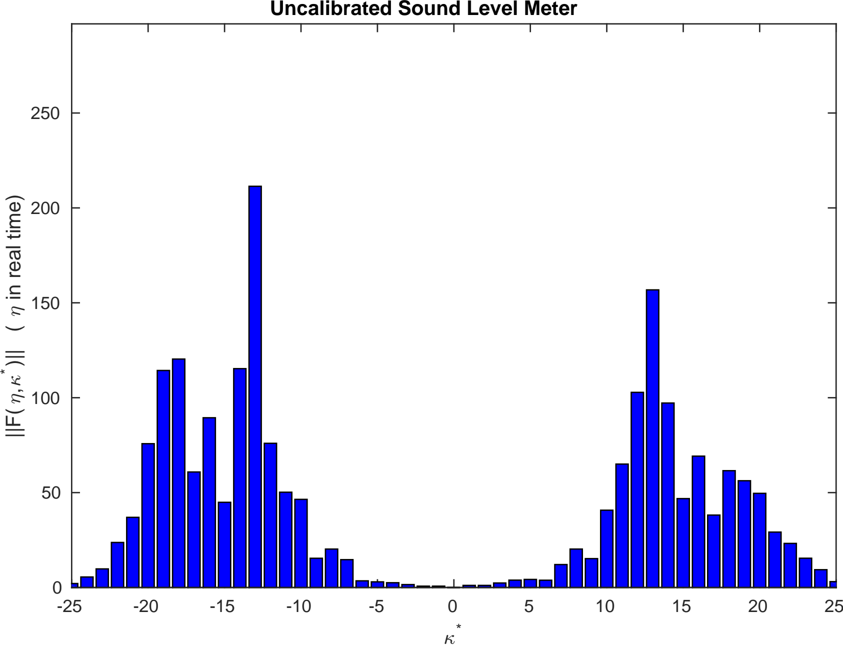


Figure 8.4: Snapshot of the sound level player in action. The absolute value of κ^* corresponds to the number of the logarithmic frequency bin (increasing numbers for increasing frequencies). Positive numbers for the right channel, and negative numbers for the left channel.

Chapter 9

Signal Transforms — Wavelets

In this chapter, you will learn about:

- The weak point of using a natural basis or a Fourier basis
- The route to take to overcome these problems: wavelets
- The Haar transform and how it leads us to wavelets
- Many more wavelets, how they are constructed and their properties, e.g., Daubechies wavelets
- What wavelet packet transforms are
- The application opportunities offered by wavelets

After having read this chapter, some questions will still be left unanswered:

- What is a lifting scheme?
- How do I create my own wavelets, tailored to my specific application?

After having read/studied this chapter, you are expected to be able to

- Explain the reason of existence of wavelets
- Understand/explain and apply the Haar transform
- Understand/explain and apply more complex Wavelet transforms
- Apply wavelet and wavelet packet transforms, given a specific set of wavelets and scaling signals
- Employ wavelet transforms for signal compression and denoising

9.1 Introduction: why do we need wavelets?

So far, we've seen a number of ways to decompose a signal into more basic parts:

- the natural decomposition (or impulse decomposition)
- the frame decomposition
- the interlaced decomposition
- the even/odd decomposition
- the Fourier decomposition

For now, let's focus on the first (the natural decomposition) and the last (the Fourier decomposition). The first considers a signal per time point. The latter looks at a signal by frequency. Note that they both are orthogonal decompositions. For your convenience, a short introduction on vector spaces has been given in section D.8 on page 359.

Seen from the perspective of orthogonal decompositions, they are opposite extremes. The former uses very short base signals, allowing to pinpoint time events with the utmost detail, but provides no information on the frequency content of the signal. The latter uses very long base signals in the time domain, allowing to pinpoint frequencies with the utmost detail, but provides not information on the timing of events.

Let's illustrate this with a simple example. Consider a very old tune you probably know from your childhood: Brother John. You can find a musical discription of the first few strokes in C major below:[1]

A mathematical description version of these strokes could be:

$$
\begin{aligned}
g(t) &= \mathrm{rect}_{0.00,0.45}(t)g(2\pi f_{c'}t) + \mathrm{rect}_{0.50,0.95}(t)g(2\pi f_{d'}t) + \mathrm{rect}_{1.00,1.45}(t)g(2\pi f_{e'}t) \\
&+ \mathrm{rect}_{1.50,1.95}(t)g(2\pi f_{c'}t) + \mathrm{rect}_{2.00,2.45}(t)g(2\pi f_{c'}t) + \mathrm{rect}_{2.50,2.95}(t)g(2\pi f_{d'}t) \\
&+ \mathrm{rect}_{2.00,2.45}(t)g(2\pi f_{e'}t) + \mathrm{rect}_{3.50,3.95}(t)g(2\pi f_{c'}t) + \mathrm{rect}_{4.00,4.45}(t)g(2\pi f_{e'}t) \\
&+ \mathrm{rect}_{4.50,4.95}(t)g(2\pi f_{f'}t) + \mathrm{rect}_{5.00,5.95}(t)g(2\pi f_{g'}t) + \mathrm{rect}_{6.00,6.45}(t)g(2\pi f_{e'}t) \\
&+ \mathrm{rect}_{6.50,6.95}(t)g(2\pi f_{f'}t) + \mathrm{rect}_{7.00,7.95}(t)g(2\pi f_{g'}t)
\end{aligned}
\tag{9.1}
$$

with

$$
\begin{aligned}
\mathrm{rect}_{a,b}(t) &= u(t-a) - u(t-b) \\
g(x) &= \sum_{i=1}^{4} \frac{1}{2^{i-1}} \sin(ix)
\end{aligned}
$$

and $u(t)$ denoting the Heaviside step function as usual. The function g results in a fundamental frequency and 3 overtones (harmonics).

The fundamental frequencies of the individual notes can be found in the table below:

Tone[2]	Note name	Symbol	Frequency [Hz]
c′	do	$f_{c'}$	261.63
d′	re	$f_{d'}$	293.66
e′	mi	$f_{e'}$	329.63
f′	fa	$f_{f'}$	349.23
g′	sol	$f_{g'}$	392.00

[1] If you are reading an electronic hyperlinked version of this text, you can click on the graph below to listen to the corresponding audio fragment.

We sampled this musical fragment with a sample frequency $f_s = 44.1$ kHz. Now consider this music fragment in the time domain (i.e. decomposed in the natural basis), depicted in Figure 9.1. Though the signal has the discretized time n as independent value, we used the continuous-time variable t on the x-axis to ease referencing the signal to its mathematical description. We can clearly see when the notes are played (i.e. we have a detailed view on the timing), but it's impossible to distinguish the frequencies on this graph. The density of the graph gives some clue on the frequency, but it's not a clear view.[3]

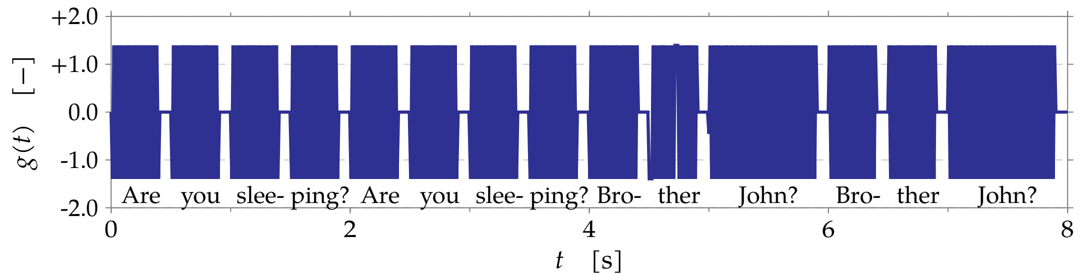


Figure 9.1: Brother John's tune w.r.t. to the natural basis (time)

Now consider this music fragment in the frequency domain (decomposed in the Fourier basis), depicted in Figure 9.2. We can clearly distinguish the 5 different tones used in the tune (though we have some spectral leak), but it's impossible to distinguish the timing of these tones. The height of the peaks gives a clue about the length of the notes, but, again, it's not a clear view.

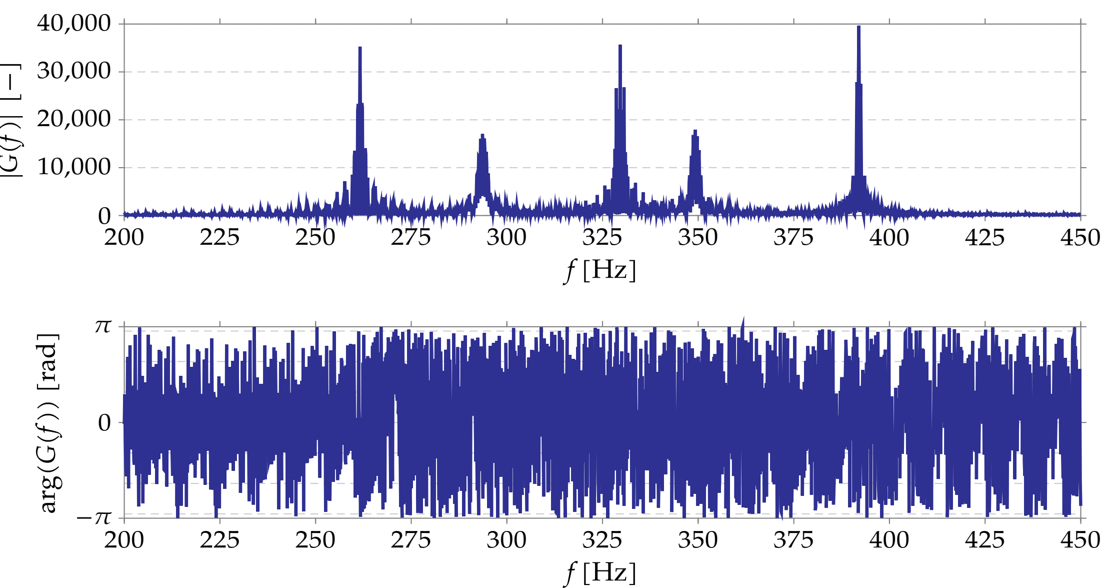


Figure 9.2: Brother John's tune w.r.t. to the Fourier basis (frequency)

The conclusion is rather obvious: neither of the two decompositions gives a good view on what the music fragment is all about. It is about waveforms ánd about their timing. They each

[2] The Helmholtz pitch notation is used.

[3] Of course, we can zoom in on the graph to investigate the period length of each of the tones, but that requires extra calculations: the frequency is not readily observable in the graph.

only give a view on one of the two aspects.
We therefore need a more appropriate basis, one that still reveals specific patterns/shapes (or trends) in the signal, and that allows pinpointing these in time. Wavelet analysis will bring us right that.

Remarks

- The observation that both the natural decomposition and the Fourier analysis are no good at analyzing signals extends beyond the domain of music. In general we are interested in events that exhibit a certain pattern/shape. Of these patterns and shapes we are also interested in when they occur.
- We could consider dividing the time axis into small (possibly overlapping) intervals and investigating each of the intervals seperately using the FFT. This leads to the short-time Fourier Transform concept (also known as the Gabor transformation). Though valuable, we will not investigate it further in the course of this chapter.
- The overtones in the signal $g(x)$ (with frequencies $2f$, $3f$ and $4f$) are not visible in the Fourier analysis of Figure 9.2 because we limited the frequency range. Therefore these overtone frequencies are not visible on the graph.

Before reading on: if your knowledge about orthogonal signal decompositions is a bit faded, you might want to read Appendix B as a refresher.

Exercises

Exercise 9.1-1: Below you can find the chromatic scale ranging from a (220 Hz) to a' (440 Hz).[4]

Let's assume an equally tempered musical instrument producing these tones. If you know that for this kind of instrument the tones are geometrically equally spread, can you reconstruct the frequencies used in Brother John's tune?

Exercise 9.1-2: (*) Try plotting (9.1) in MATLAB/OCTAVE with a limited number of points (e.g., 500). What effect do you observe? Does your version of the graph correspond to the graph of Figure 9.1 on the previous page? Try to find a strategy to overcome the observed effect.

9.2 The ball park

The theory of wavelets is very broad. We will only discover a small portion of it. The classification diagram of Figure 9.3, allows you to keep track of where we are. Let's take a

[4] If you are reading an electronic hyperlinked version of this text, you can click on the graph above to listen to the corresponding audio fragment.

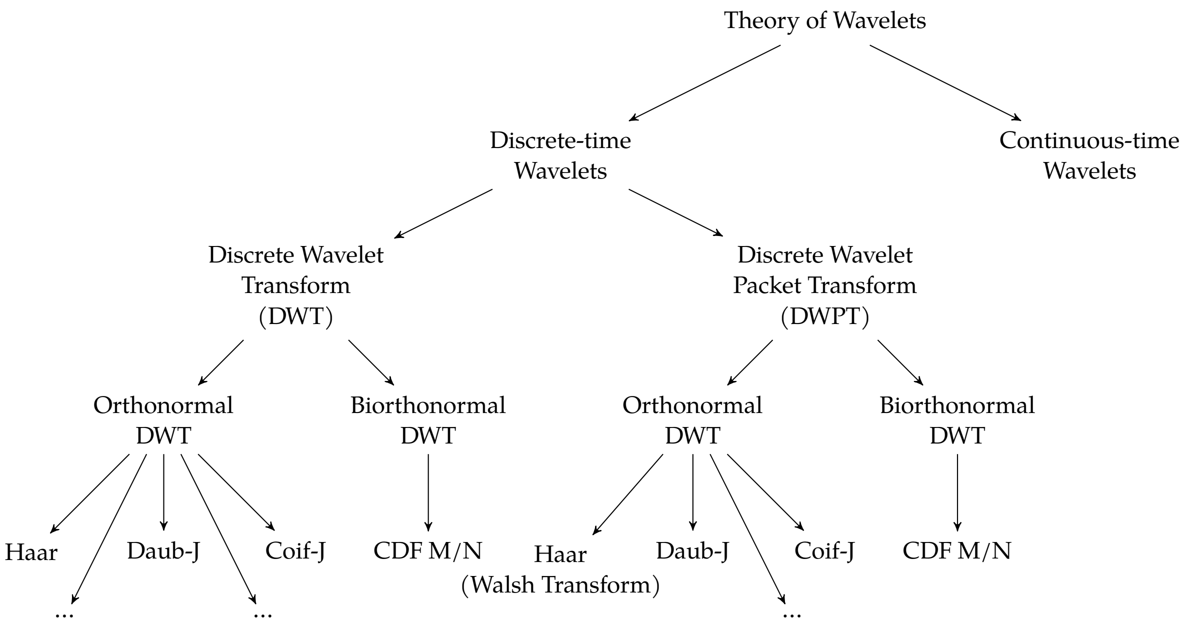


Figure 9.3: Classification diagram for the theory of wavelets

minute to browse through it. We start at the top of the tree. Wavelets can be considered in the continuous time domain and in the discrete-time domain. We will focus on discrete time. Within that category, we can distinguish between ordinary wavelet transforms (DWT) and packet transforms (DWPT). Both can be subdivided into orthonormal and biorthonormal versions. For each of them a multitude of wavelets can be employed: Haar wavelets, Daublets, Coiflets, Cohen-Daubechies-Feauveau wavelets, a.s.o.[5]

Of course, the tree also extends in the continuous-time domain, but since this is not the focus of this course, we chose not to elaborate that part.

Besides this standard classification, we can also distinguish between 1-dimensional, 2-dimensional wavelets and even n-dimensional wavelets.

As the Haar transform for *real-valued* signals (the bottom-left leaf of the tree diagram) is a gentle step up into the world of wavelets, we will use this transform to get started.

Note that most of the wavelets (Daublets, Coiflets, a.s.o.) are most recent (and date from the 1980s). However, the Haar transform dates from the beginning of the 20th century, long before the term wavelets was coined.

9.3 The Haar transform

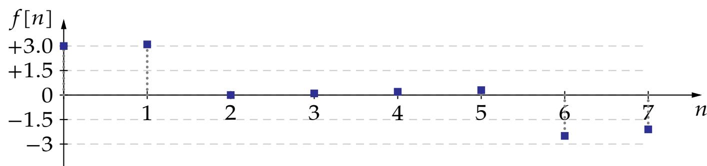
We'll introduce the Haar transform by example. To avoid unnecessary details due to complex numbers, we will use a real-valued signal out of S_8 as an example.

[5] Daublets are named after Ingrid Daubechies, a Belgian mathematician/physicist, who can be rightfully called the inventor of wavelets. A breakthrough on the level of a Nobel prize winner! Coiflets have been invented by Ingrid Daubechies, but named after Ronald Coifman who suggested them to Daubechies.

Consider:

$$f[n] = \left[3, \quad 3.1, \quad 0, \quad 0.1, \quad 0.2, \quad 0.3, \quad -2.5, \quad -2.1\right] \tag{9.2}$$

Graphically:



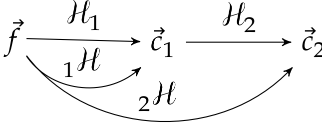
Trend and fluctuation The basic idea of the Haar transform is that a real signal is a superposition of *desired* information and *undesired* information. The desired information could be a speech signal detected using a microphone, the undesired information then would be the noise that is picked up. The undesired information doesn't need to be noise, however. Even more, the desired information could be a systolic aberration visible on an electrocardiogram (ECG) of the heart muscle contraction, the undesired information the normal systolic operation of the heart. You undoubtedly perceive the inappropriate meaning of the terms desired and undesired.

Therefore, we choose to adopt the more neutral terms: *trend* and *fluctuation*. The trend is what we are looking for in the signal, the fluctuation the deviation of the signal w.r.t. the trend.

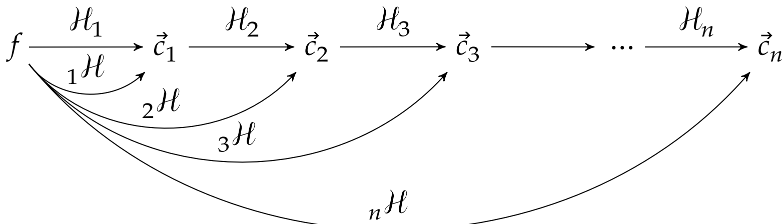
Levels The Haar transform is applied in several consecutive so-called *levels*. A 1-level Haar transformed signal (denoted by $\vec{c}_1 = {}_1\mathcal{H}(\vec{f})$) is obtained by applying the level-1 Haar transform (denoted by $\mathcal{H}_1$) to the original signal. Graphically

$$\vec{f} \xrightarrow{\mathcal{H}_1} \vec{c}_1$$

The notation (with the pre and post subscripts) seems a bit awkward. However, it becomes clear when considering the subsequent application of a level-2 Haar transform ($\mathcal{H}_2$) to obtain as result the 2-level Haar transformed signal $\vec{c}_2$. In this way, the 2-level Haar transform is the concatenation of level-1 Haar transform followed by the level-2 Haar transform. Symbolically: ${}_2\mathcal{H}(f) = \mathcal{H}_2(\mathcal{H}_1(f))$.

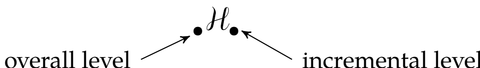


Of course, we can continue this process:



In general the level-n Haar transform gets you from level $n-1$ to level n, while the n-level Haar transform gets you all the way from the time-domain signal to level n. Note that by replacing the word 'level' by $\mathcal{H}$ in the description of the transform, there cannot be any confusion about the notation.

For your reference, the meaning of the sub- and superscripts is summarized below:

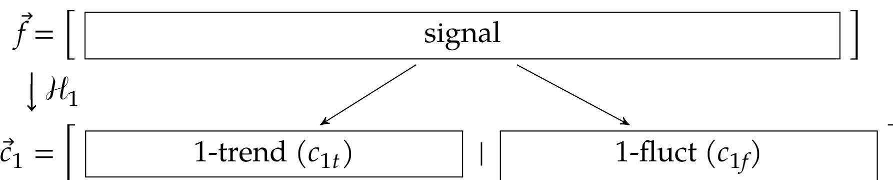


The term level is often also called the *resolution level*. When discussing the multi-resolution analysis (MRA) later on, the reason for this will become clear.

We will start with level 1 in the following section.

9.3.1 The level-1 Haar transform

The level-1 Haar transform (denoted by $\mathcal{H}_1$) will transform a signal into two half-length subsignals, the so-called *first trend subframe* and the *first fluctuation subframe*.



The symbol $\mathcal{H}_1$ denotes the level-1 Haar transform. To indicate the separation between the first trend and the first fluctuation, we will write a vertical bar between them.

Now, the basic idea is to look at pairs of samples. If we consider any two consecutive samples f_a and f_b, we can write them as:

$$f_a = \frac{f_a + f_b}{2} + \frac{f_a - f_b}{2}$$
$$f_b = \frac{f_a + f_b}{2} - \frac{f_a - f_b}{2}$$

Haar proposed to use the average of f_a and f_b (the first term on the right) as trend and half the difference of f_a and f_b (the second term on the right) as fluctuation. However he also added a

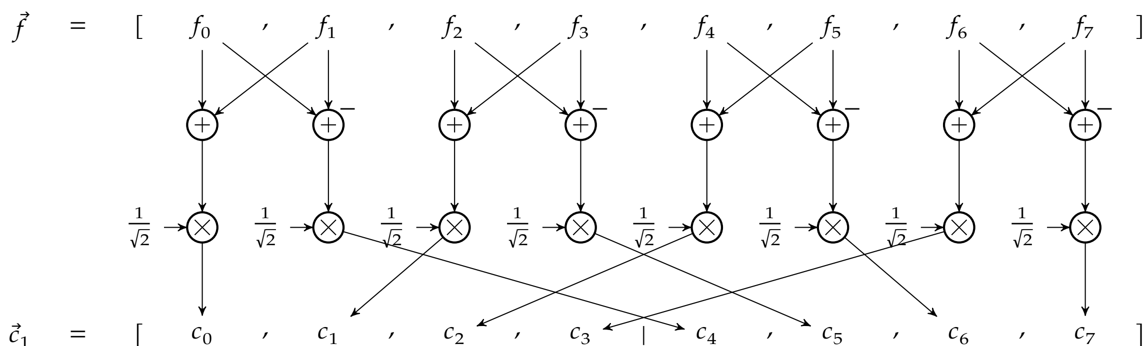
scaling factor factor $\sqrt{2}$, such that:[6]

$$c_{trend} = \sqrt{2}\frac{f_a + f_b}{2} = \frac{f_a + f_b}{\sqrt{2}} \tag{9.3}$$

$$c_{fluct} = \sqrt{2}\frac{f_a - f_b}{2} = \frac{f_a - f_b}{\sqrt{2}} \tag{9.4}$$

If f_a and f_b are the first pair of samples of f, we will take c_{trend} to be the first sample of the first trend and c_{fluct} to be the first sample of the first fluctuation. The second pair of samples yield the second samples of the first trend and the first fluctuation in the very same way, a.s.o.

Applying this principe to every pair of two consecutive samples, leads to the following *forward transform calculation scheme* (given a signal of length 8; however the principle can be extended to any 2^N-point signal):



Let's apply this to our example:

$$f[n] = [3, 3.1, 0, 0.1, 0.2, 0.3, -2.5, -2.1]$$

$$\downarrow \mathcal{H}_1$$

$$c[k] = \left[\frac{3+3.1}{\sqrt{2}}, \frac{0+0.1}{\sqrt{2}}, \frac{0.2+0.3}{\sqrt{2}}, \frac{-2.5-2.1}{\sqrt{2}} \;\middle|\; \frac{3-3.1}{\sqrt{2}}, \frac{0-0.1}{\sqrt{2}}, \frac{0.2-0.3}{\sqrt{2}}, \frac{-2.5-(-2.1)}{\sqrt{2}}\right]$$

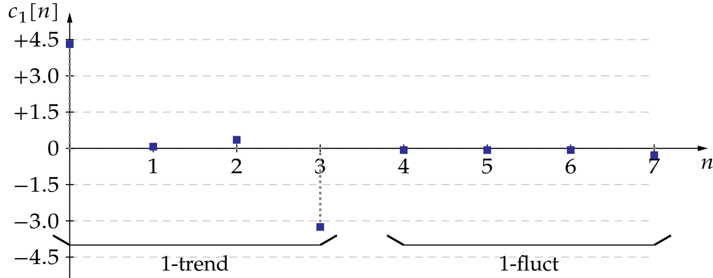
$$= \left[4.3134, \; 0.0707, \; 0.3536, \; -3.2527 \;\mid\; -0.0707, \; -0.0707, \; -0.0707, \; -0.2828\right]$$

Note that for sample pairs that are about the same, the trend value is large when compared to the corresponding fluctuation value and vice versa. Therefore, the Haar transform is said to detect a constant trend.

Also note that if the trend detection performs well (i.e. the fluctuation values are considerably smaller than the trend values), the trend subframe is just a coarse version of the original signal (sampled with half the resolution). This observation will give rise to the concept of multi-resolution analysis (MRA) introduced later on.

The graph of the transformed signal can be found below:

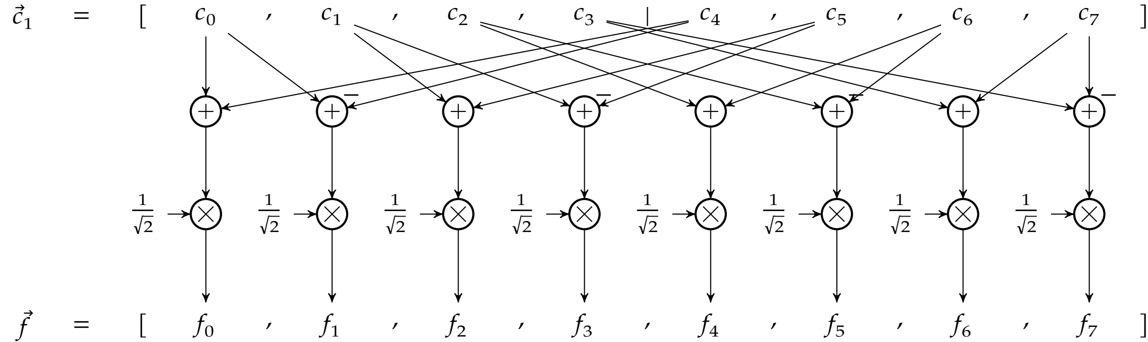
[6]For your peace of mind: don't worry about the odd scaling factor. For now, just take it for granted. Its necessity will become clear later on.



As we claim that the Haar transform is a *transform*, there should be an easy route back. Indeed, solving (9.3) and (9.4) for f_a and f_b results in:

$$f_a = \frac{c_{trend} + c_{fluct}}{\sqrt{2}}$$
$$f_b = \frac{c_{trend} - c_{fluct}}{\sqrt{2}}$$

Applying this principle systematically, results in the following *inverse transform calculation scheme* (given a signal length of 8; however the principle can be extended to any 2^N-point signal):



9.3.2 One step back: the level-1 Haar transform from a wavelet perspective

Let's take one step back and take a look at the level-1 Haar transform from the perspective of an orthonormal vector transform.

The idea is to decompose the signal at hand using a set of base vectors B, of which half are trend vectors ($\vec{s}_{1,0}$ to $\vec{s}_{1,3}$), and half are fluctuation vectors ($\vec{w}_4$ to $\vec{w}_7$).

The base The Level-1 Haar transform for a signal with a length of eight uses the following trend base vectors:

$$\vec{s}_{1,0} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1, & 1, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{s}_{1,1} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 1, & 1, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{s}_{1,2} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 1, & 1, & 0, & 0 \end{bmatrix}$$
$$\vec{s}_{1,3} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 1, & 1 \end{bmatrix}$$

It also uses the following fluctuation base vectors:

$$\vec{w}_4 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1, & -1, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{w}_5 = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 1, & -1, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{w}_6 = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 1, & -1, & 0, & 0 \end{bmatrix}$$
$$\vec{w}_7 = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 1, & -1 \end{bmatrix}$$

Most often the trend detection base vectors are called *scaling signals* and the fluctuation base vectors are called *wavelets*. Hence the used symbols s and w. The name *wavelet* stems from the fact that it contains one or more alternations (zero-crossings), and therefore resembles the most basic of waves, a sine wave (therefore the base *wave*), yet it is only a small part of a full wave (therefore, the suffex *let*). The term *scaling signal* will become clear later on.

Note that the wavelet base vectors can be generated using the inverse transform calculation scheme feeding it with natural base vectors.

Independence — One can verify that the set defined above is independent, by writing down the matrix equation $c_0\vec{s}_{1,0} + c_1\vec{s}_{1,1} + c_2\vec{s}_{1,2} + c_3\vec{s}_{1,3} + c_4\vec{w}_4 + c_5\vec{w}_5 + c_6\vec{w}_6 + c_7\vec{w}_7 = 0$ with the c_i as unknowns and verifying that the coefficient matrix is of full (column) rank.

Completeness — One can verify that the base is complete, as it counts 8 elements, while the dimension of the vector space is also 8.

Orthogonality — Note that

$$\begin{aligned} \langle \vec{s}_{1,i}, \vec{s}_{1,j} \rangle &= 0, \quad \forall i \neq j \\ \langle \vec{w}_i, \vec{w}_j \rangle &= 0, \quad \forall i \neq j \\ \langle \vec{s}_{1,i}, \vec{w}_j \rangle &= 0, \quad \forall i, j \end{aligned}$$

i.e. the base is orthogonal.

Orthonormality — Note that

$$\begin{aligned} \|\vec{s}_{1,i}\| &= 1 \\ \|\vec{w}_j\| &= 1 \end{aligned}$$

and therefore the base is also orthonormal.[7] This explains why Haar added the scaling factor $\sqrt{2}$ to its trend and fluctuation formulae. He did this to ensure normalization of the results.

Conclusion: $B_{_1\mathcal{H}} = \left\{\vec{s}_{1,0}, \vec{s}_{1,1}, \vec{s}_{1,2}, \vec{s}_{1,3}, \vec{w}_4, \vec{w}_5, \vec{w}_6, \vec{w}_7\right\}$ is an orthonormal base for S_8.

The forward transformation According to (D.2) we can decompose $\vec{f} = [f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7]$ into its base $B = \left\{\vec{s}_{1,0}, \vec{s}_{1,1}, \vec{s}_{1,2}, \vec{s}_{1,3}, \vec{w}_4, \vec{w}_5, \vec{w}_6, \vec{w}_7\right\}$ as:

$$\vec{f} = c_0\vec{s}_{1,0} + c_1\vec{s}_{1,1} + c_2\vec{s}_{1,2} + c_3\vec{s}_{1,3} + c_4\vec{w}_4 + c_5\vec{w}_5 + c_6\vec{w}_6 + c_7\vec{w}_7 \tag{9.7}$$

with:

$$c_i = \left\langle \vec{f}, \vec{s}_{1,i} \right\rangle, \quad \forall i \in \{0,1,2,3\}$$
$$c_i = \left\langle \vec{f}, \vec{w}_i \right\rangle, \quad \forall i \in \{4,5,6,7\}$$

If we arrange the coefficients c_i into a vector $\vec{c}_1 = [c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7]$ we obtain the level-1 Haar transform, for which we can use many common notations:

$$\vec{c}_1 = \mathcal{H}_1(\vec{f})$$

or

$$\vec{f} \xrightarrow{\mathcal{H}_1} \vec{c}_1$$

As the vector $\vec{f}$ is referred to as the signal in the time domain, we refer to $\vec{c}_1$ as the signal in the *wavelet spectrum* at the first (resolution) level.[8]

As introduced earlier, we often write a separation marker in the middle:

$$[f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7] \xrightarrow{\mathcal{H}_1} [c_0, c_1, c_2, c_3 \mid c_4, c_5, c_6, c_7]$$

Let's take it one step further and consider the explicit equations for the coefficients c_i:

$$c_0 = \left\langle \vec{f}, \vec{s}_{1,0} \right\rangle = \frac{1}{\sqrt{2}}(f_0 + f_1) \qquad c_4 = \left\langle \vec{f}, \vec{w}_4 \right\rangle = \frac{1}{\sqrt{2}}(f_0 - f_1)$$
$$c_1 = \left\langle \vec{f}, \vec{s}_{1,1} \right\rangle = \frac{1}{\sqrt{2}}(f_2 + f_3) \qquad c_5 = \left\langle \vec{f}, \vec{w}_5 \right\rangle = \frac{1}{\sqrt{2}}(f_2 - f_3)$$
$$c_2 = \left\langle \vec{f}, \vec{s}_{1,2} \right\rangle = \frac{1}{\sqrt{2}}(f_4 + f_5) \qquad c_6 = \left\langle \vec{f}, \vec{w}_6 \right\rangle = \frac{1}{\sqrt{2}}(f_4 - f_5)$$
$$c_3 = \left\langle \vec{f}, \vec{s}_{1,3} \right\rangle = \frac{1}{\sqrt{2}}(f_6 + f_7) \qquad c_7 = \left\langle \vec{f}, \vec{w}_7 \right\rangle = \frac{1}{\sqrt{2}}(f_6 - f_7)$$

This just confirms that working with projection on the base vectors boils down to the calculation scheme we introduced earlier.

One could consider this transformation to be a matrix multiplication starting from the column vectors $\vec{c}_1$ and $\vec{f}$:

$$\vec{c}_1 = H_1\vec{f} \tag{9.8}$$

[7] We like the base to be normalized such that we can use the simplified Parseval's theorem (the energy preservation theorem).

[8] We reserve the term spectrum for the Fourier decomposition. Always add the prefix 'wavelet' when referring to the space of a transformed wavelet signal.

with

$$H_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \tag{9.9}$$

Though this is a very inefficient implementation of the level-1 Haar-transformation, and therefore has no practical use, it serves one purpose: it shows that the transform is linear. We will need this fact when considering the fluctuation and trend signals separately in section 9.3.3 on the facing page.

A final thing to note about the matrix H_1 is it's orthonormality (which is evident, as we composed it by placing orthonormal basevectors as its rows), allowing us to write:

$$H_1^T H_1 = I = H_1 H_1^T$$

with I the identity matrix and H_1^T denoting the transpose of H_1. This fact allows easy calculation of the inverse of H_1 to be:

$$H_1^{-1} = H_1^T$$

The inverse transformation Performing the inverse transform boils down to applying (9.7). The obvious common notation applies:

$$\vec{f} = \mathcal{H}_1^{-1}(\vec{c}_1)$$

However, we keep writing the Haar-transform pair as:

$$\vec{f} \xrightarrow{\mathcal{H}_1} \vec{c}_1$$

Solving (9.7) for the individual values f_i yields:

$$f_0 = \frac{1}{\sqrt{2}}(c_0 + c_4) \qquad f_4 = \frac{1}{\sqrt{2}}(c_2 + c_6) \tag{9.10}$$

$$f_1 = \frac{1}{\sqrt{2}}(c_0 - c_4) \qquad f_5 = \frac{1}{\sqrt{2}}(c_2 - c_6)$$

$$f_2 = \frac{1}{\sqrt{2}}(c_1 + c_5) \qquad f_6 = \frac{1}{\sqrt{2}}(c_3 + c_7)$$

$$f_3 = \frac{1}{\sqrt{2}}(c_1 - c_5) \qquad f_7 = \frac{1}{\sqrt{2}}(c_3 - c_7) \tag{9.11}$$

This just confirms that working with projection on the base vectors boils down to the calculation scheme we introduced earlier.

One could consider this transformation to be a matrix multiplication starting from the column vectors $\vec{c}_1$ and $\vec{f}$ (though this is not an efficient implementation):

$$\begin{aligned}\vec{f} &= H_1^{-1}\vec{c}_1 \\ &= H_1^T\vec{c}_1\end{aligned}$$

with H defined as in (9.9).

9.3.3 Trend and fluctuation

The fact that the idea of the Haar transform (and more in general, all wavelet transforms) is to separate the trend from the fluctuation, should trigger us to consider these separately.

We made the observation before that we can divide $\vec{c}_1$ into two subframes (the *first trend subframe* and the *first fluctuation subframe*). We go one step further now and consider the transformed signal to be a superposition of a *first trend signal* ($\vec{c}_{1,T}$) and a *first fluctuation signal* ($\vec{c}_{1,F}$).

$$\begin{aligned}\vec{f} \xrightarrow{\mathcal{H}_1} \vec{c}_1 &= \left[\ \boxed{1-trend}\ |\ \boxed{1-fluct}\ \right] \\ &= \underbrace{\left[\ \boxed{1-trend}\ |\ \boxed{\quad 0 \quad}\ \right]}_{=\vec{c}_{1,T}} + \underbrace{\left[\ \boxed{\quad 0 \quad}\ |\ \boxed{1-fluct}\ \right]}_{=\vec{c}_{1,F}} \\ &= \vec{c}_{1,T} + \vec{c}_{1,F}\end{aligned} \tag{9.12}$$

In this way, the trend signal consists of the trend frame followed by a zero frame, and the fluctuation signal consists of a zero frame followed by the fluctuation frame.

This definition in combination with the fact that the inverse level-1 Haar transform is linear, allows us to bring the distinction between trend and fluctuation into the time domain. We just apply the inverse level-1 Haar transform to both sides of (9.12):

$$\begin{aligned}\vec{c}_1 &= \vec{c}_{1,T} + \vec{c}_{1,F} \\ &\downarrow \mathcal{H}_1^{-1} \\ \vec{f} = \mathcal{H}_1^{-1}(\vec{c}_1) &= \mathcal{H}_1^{-1}(\vec{c}_{1,T} + \vec{c}_{1,F}) \\ &= \underbrace{\mathcal{H}_1^{-1}(\vec{c}_{1,T})}_{=\vec{f}_{1,T}} + \underbrace{\mathcal{H}_1^{-1}(\vec{c}_{1,F})}_{=\vec{f}_{1,F}} \\ \vec{f} &= \vec{f}_{1,T} + \vec{f}_{1,F}\end{aligned} \tag{9.15}$$

We will refer to $\vec{f}_{1,T}$ as the *first trend signal in the time domain* and to $\vec{f}_{1,F}$ as the *first fluctuation signal in the time domain*.

9.3.4 Example

Example - by hand Let's reconsider our example of (9.2), that has been illustrated in Figure 9.3 on page 220.

$$f[n] = \begin{bmatrix}3, & 3.1, & 0, & 0.1, & 0.2, & 0.3, & -2.5, & -2.1\end{bmatrix}$$

We calculated the level-1 Haar transform before. Summarizing:

$$\begin{aligned}
f[n] &= \begin{bmatrix}3, & 3.1, & 0, & 0.1, & 0.2, & 0.3, & -2.5, & -2.1\end{bmatrix} \\
&\big\downarrow \mathcal{H}_1 \\
c_1[k] &= \begin{bmatrix}4.3134, & 0.0707, & 0.3536, & -3.2527 & | & -0.0707, & -0.0707, & -0.0707, & -0.2828\end{bmatrix} \\
&\big\downarrow \mathcal{H}_1^{-1} \\
f[n] &= \begin{bmatrix}3, & 3.1, & 0, & 0.1, & 0.2, & 0.3, & -2.5, & -2.1\end{bmatrix}
\end{aligned}$$

Secondly, we can calculate the total energy of the signal:

$$\left\|\vec{f}\right\|^2 = 3^2 + 3.1^2 + 0.1^2 + 0.2^2 + 0.3^2 + (-2.5)^2 + (-2.1)^2 = 29.41$$

The first trend signal is readily obtained by zeroing the second half of $c_1[k]$:

$$c_{1,T}[k] = \begin{bmatrix}4.3134, & 0.0707, & 0.3536, & -3.2527 & | & 0, & 0, & 0, & 0\end{bmatrix}$$

The first trend signal in the time domain can be calculated by applying the inverse Haar transform to $\vec{c}_{1,T}$:

$$\begin{aligned}
f_{1,T}[n] &= \Big[\tfrac{4.3134+0}{\sqrt{2}}, \quad \tfrac{4.3134-0}{\sqrt{2}}, \quad \tfrac{0.0707+0}{\sqrt{2}}, \quad \tfrac{0.0707-0}{\sqrt{2}}, \\
&\qquad \tfrac{0.3536+0}{\sqrt{2}}, \quad \tfrac{0.3536+0}{\sqrt{2}}, \quad \tfrac{-3.2527+0}{\sqrt{2}}, \quad \tfrac{-3.2527-0}{\sqrt{2}}\Big] \\
&= \begin{bmatrix}3.05, & 3.05, & 0.05, & 0.05, & 0.25, & 0.25, & -2.3, & -2.3\end{bmatrix}
\end{aligned}$$

The latter has been graphed in Figure 9.4(a).

The total energy embedded in the transformed trend signal is easily calculated as:

$$\left\|\vec{c}_{1,T}\right\|^2 = 4.3134^2 + 0.0707^2 + 0.3536^2 + (-3.2527)^2 = 29.315$$

and because of the energy preservation theorem this is equal to the total energy embedded in the trend signal in the time domain:

$$\left\|\vec{f}_{1,T}\right\|^2 = 3.05^2 + 3.05^2 + 0.05^2 + 0.05^2 + 0.25^2 + 0.25^2 + (-2.3)^2 + (-2.3)^2 = 29.315$$

The first fluctuation signal is readily obtained by zeroing the first half of $c_1[k]$:

$$c_{1,F}[k] = \begin{bmatrix}0, & 0, & 0, & 0 & | & -0.0707, & -0.0707, & -0.0707, & -0.2828\end{bmatrix}$$

The first fluctuation signal in the time domain is then calculated as:

$$\begin{aligned}
f_{1,F}[n] &= \Big[\tfrac{0+(-0.0707)}{\sqrt{2}}, \quad \tfrac{0-(-0.0707)}{\sqrt{2}}, \quad \tfrac{0+(-0.0707)}{\sqrt{2}}, \quad \tfrac{0-(-0.0707)}{\sqrt{2}}, \\
&\qquad \tfrac{0+(-0.0707)}{\sqrt{2}}, \quad \tfrac{0-(-0.0707)}{\sqrt{2}}, \quad \tfrac{0+(-0.2828)}{\sqrt{2}}, \quad \tfrac{0-(-0.2828)}{\sqrt{2}}\Big] \\
&= \begin{bmatrix}-0.05, & 0.05, & -0.05, & 0.05, & -0.05, & 0.05, & -0.2, & 0.2\end{bmatrix}
\end{aligned}$$

The latter has been graphed in Figure 9.4(b).

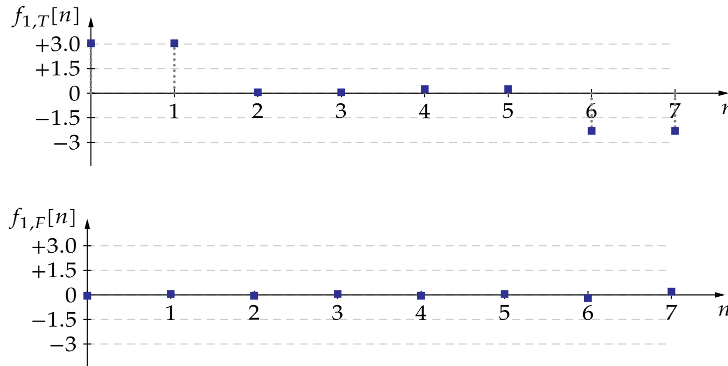


Figure 9.4: The signal of (9.2) decomposed in (a) its first Haar trend signal and (b) its first Haar fluctuation signal

The total energy embedded in the transformed fluctuation signal is then calculated as:

$$\left\|\vec{c}_{1,F}\right\|^2 = (-0.0707)^2 + (-0.0707)^2 + (-0.0707)^2 + (-0.2828)^2 = 0.095$$

and is equal to the total energy embedded in the fluctuation signal in the time domain:

$$\left\|\vec{f}_{1,F}\right\|^2 = (-0.05)^2 + 0.05^2 + (-0.05)^2 + 0.05^2 + (-0.05)^2 + 0.05^2 + (-0.2)^2 + (0.2)^2 = 0.095$$

The reader can verify that

$$\left\|\vec{f}\right\|^2 = \left\|f_{1,T}\right\|^2 + \left\|f_{1,F}\right\|^2 = \left\|\vec{c}_{1,T}\right\|^2 + \left\|\vec{c}_{1,F}\right\|^2$$

Note however, that most of the energy of the signal ($29.315/29.41 = 99.7\%$) ends up in the trend signal, whereas only a small amount ends up in the fluctuation signal ($0.095/29.41 = 0.3\%$). This can also be seen in the graphs of Figure 9.4. The original signal $f[n]$ is quite well represented by the trend signal $f_{1,T}[n]$. The fluctuation signal $f_{1,F}[n]$ is close to energyless, i.e. zero.

Example - using MATLAB Let's do the same thing again, but this time using MATLAB.[9]

Before we start, we set the *discrete wavelet mode* to periodic. We'll get back to the issue of wavelet modes in section 9.7 on page 277.

```
dwtmode('per');
```

We'll start by defining our input signal:

```
f = [ 3, 3.1, 0, 0.1, 0.2, 0.3, -2.5, -2.1];
```

We will use the matlab function `dwt` to calculate level-1 transforms. The Haar transformation is also called the 'Daubechies1' transform, abbreviated to `db1`.

[9]Warning: the OCTAVE implementation for wavelets is incomplete and contains many problematic issues. We will therefore restrict ourselves to MATLAB when considering wavelets.

```
[ct1,cf1] = dwt(f,'db1')
```

The trend part ends up in `ct1`, the fluctuation part in `cf1`.

The full level-1 Haar transform therefore yields:

```
c = [ct1,cf1]
```

We can go back using the inverse level-1 Haar transform, starting from the trend and the fluctuation signals, using `idwt`:

```
f = idwt( ct1, cf1, 'db1' )
```

You will see that except for rounding errors $f = g$.

The trend signal is calculated using

```
ft1 = idwt( ct1, [], 'db1' )
```

The fluctuation signal is given by

```
ff1 = idwt( [], cf1, 'db1' )
```

The energies of the various signals may be calculated using the `norm` function:

```
e = norm(c)^2
ect1 = norm(ct1)^2
ecf1 = norm(cf1)^2
eft1 = norm(ft1)^2
eff1 = norm(ff1)^2
```

Exercises

Exercise 9.3.4-1: Calculate the level-1 Haar transform of the following discrete-time signal:

$$\vec{f} = \begin{bmatrix} -3, & 2, & 0, & 1, & 3, & 4, & 2, & 0, & 0, & 1, & -1, & -2 \end{bmatrix}$$

Exercise 9.3.4-2: Calculate the level-1 inverse Haar transform of the following discrete-time signal:

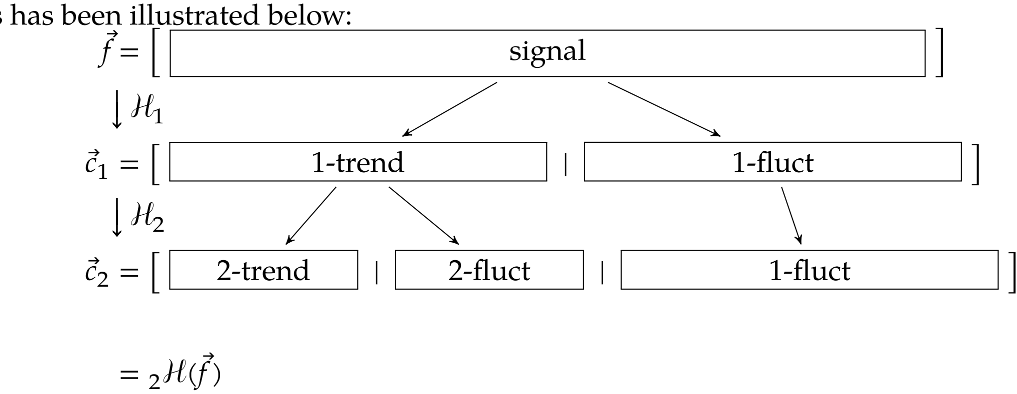
$$\vec{f} = \begin{bmatrix} -3, & 2, & 0, & 1, & 3, & 4, & | & 2, & 0, & 0, & 1, & -1, & -2 \end{bmatrix}$$

9.3.5 The level-n Haar transform

If you have a good understanding of the level-1 Haar transform, then the level-2 Haar transform should be a piece of cake. If not, first go back and study the level-1 Haar transform.

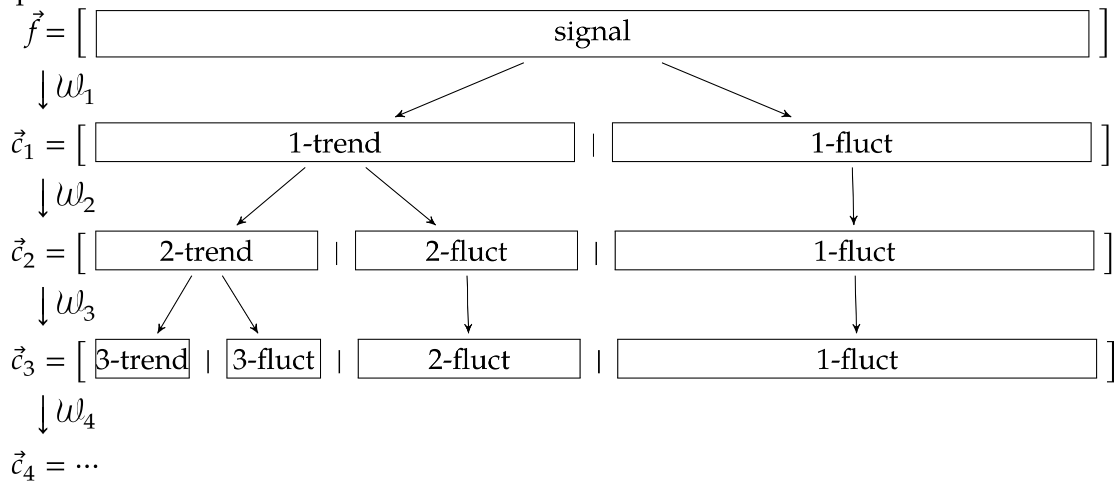
The forward transformation The basic idea of the level-n Haar transform is to continue after the level-1 Haar transform by re-applying the level-1 Haar transform to the first trend subframe, and leaving the first fluctuation subframe untouched. In this way, one obtains the 2-level Haar transform of $\vec{f}$.

This has been illustrated below:



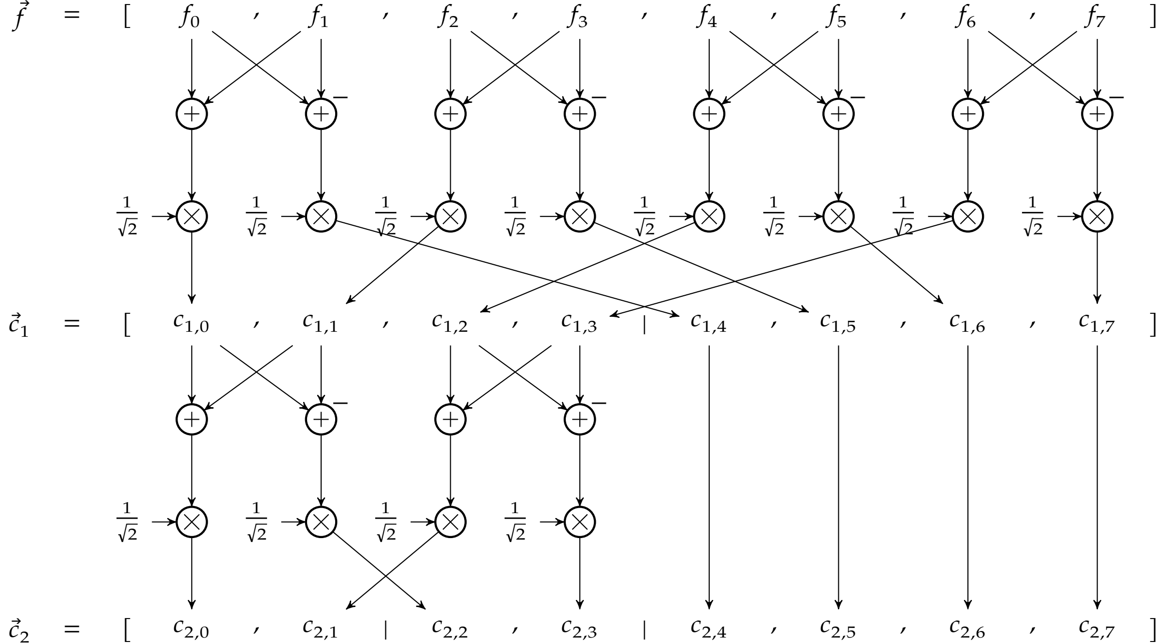
Of course one can continue the decomposition of the resulting level-n Haar transform to obtain the level-(n+1) Haar transform. In every step the n-trend subframe is decomposed into a half-length $n+1$-trend and a half-length $n+1$-fluctuation subframe.

The process has been outlined below:



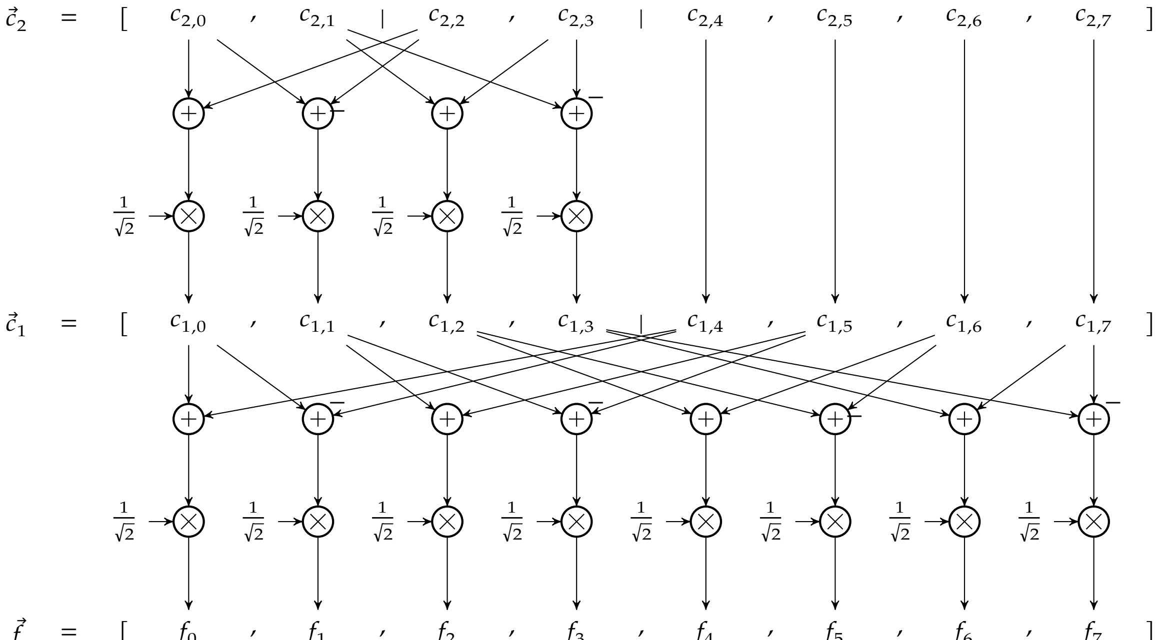
We can continue the decomposition process until the length of the n-th subframe is reduced to 1. Further decomposition is no longer possible beyond that limit.

The *forward transform calculation scheme* is easily composed for the 2-level Haar transform (given a signal of length 8; however, the principle can be extended to any 2^N-point signal). The upper half has been recycled from the 1-level Haar transform.



Completing this scheme for the 3-level Haar transform is left to the reader.

The inverse transformation Going back from $c_2[k]$ back to $c_1[k]$ is easily done using the following *inverse transform calculation scheme* (given a signal length of 8, however the principle can be extended to any 2^N-point signal). The bottom part has been recycled from the inverse 1-level Haar transform.



Extending this scheme for the third resolution level is straightforward and is left to the reader.

Example - by hand Let's apply this to our example:

$$
\begin{aligned}
f[n] &= \left[3,\quad 3.1,\quad 0,\quad 0.1,\quad 0.2,\quad 0.3,\quad -2.5,\quad -2.1\right] \\
&\downarrow \mathcal{H}_1 \\
c_1[k] &= \left[4.3134,\quad 0.0707,\quad 0.3536,\quad -3.2527 \;\mid\; -0.0707,\quad -0.0707,\quad -0.0707,\quad -0.2828\right] \\
&\downarrow \mathcal{H}_2 \\
c_2[k] &= \left[\frac{4.3134+0.0707}{\sqrt{2}},\quad \frac{0.3536+(-3.2527)}{\sqrt{2}} \;\middle|\; \frac{4.3134-0.0707}{\sqrt{2}},\quad \frac{0.3536-(-3.2527)}{\sqrt{2}} \;\middle|\right. \\
&\qquad \left. -0.0707,\quad -0.0707,\quad -0.0707,\quad -0.2828\right] \\
&= \left[3.1,\quad -2.05 \;\mid\; 3.0,\quad 2.55 \;\mid\; -0.0707,\quad -0.0707,\quad -0.0707,\quad -0.2828\right] \\
&\downarrow \mathcal{H}_3 \\
c_3[k] &= \left[\frac{3.1+(-2.05)}{\sqrt{2}} \;\middle|\; \frac{3.1-(-2.05)}{\sqrt{2}} \;\middle|\; 3.0,\quad 2.55 \;\middle|\; -0.0707,\quad -0.0707,\quad -0.0707,\quad -0.2828\right] \\
&= \left[0.7425 \;\mid\; 3.6416 \;\mid\; 3.0,\quad 2.55 \;\mid\; -0.0707,\quad -0.0707,\quad -0.0707,\quad -0.2828\right]
\end{aligned}
$$

Taking a look at the result of the 2-level Haar transform ($c_2[k]$) and comparing the corresponding values of the second trend ($[3.1, -2.05]$) and the second fluctuation ($[3.0, 2.55]$), reveals that the difference in order of magnitude that was apparent when comparing the first trend with the first fluctuation is no longer there. This means that the 'constant' trend was not recognized at the second resolution level.

Taking a look at the result of the 3-level Haar transform ($c_3[l]$) and comparing the corresponding values of the third trend ($[0.7425]$ and the third fluctuation ($[3.6416]$) shows us that most of the energy of the second trend ended up in the third fluctuation, indicating that the 3rd resolution-level wavelet has been recognized in the signal, rather than the 3rd resolution-level scaling signal.

Example - using MATLAB Let's do the same thing again, but this time using MATLAB.

We'll start by defining our input signal:

```
f = [ 3, 3.1, 0, 0.1, 0.2, 0.3, -2.5, -2.1];
```

We could calculate the n-level Haar transform by repeatedly using the `dwt` function on the trend subframe. However, MATLAB offers a convenience function `wavedec` that does the job for you.

Let's e.g., calculate $c_2[k]$ straightaway:

```
[c2,l] = wavedec( f, 2, 'db1' );
```

The resulting vector `c2` contains the result of the 2-level Haar transform. The `l` vector contains information about the trend and fluctuation information present in `c2`. Check it out!

MATLAB offers a convenient way to calculate the trend and fluctuation subframes: the `appcoef` and `detcoef` functions.

We can calculate the i-th trend subframe as (taking $i = 1$ as example):

```
i = 1;
sft1 = appcoef( c2, l, 'db1', i);
```

The 'app' in `appcoef` stands for 'approximation', an alternative name for 'trend'.

We can calculate the i-th fluctuation subframe as (taking $i = 2$ as example):

```
i = 2;
sff2 = detcoef( c2, l, 'db1', i);
```

The 'det' in `detcoef` stands for 'detail', an alternative name for 'fluctuation'.

MATLAB also offers a convenient way to calculate the trend and fluctuation signals in the time domain: the `wrcoef` function.

We can calculate the i-th trend signal in the time domain as (taking $i = 2$ as example):

```
i = 2;
ft2 = wrcoef( 'a', c2, l, 'db1', i);
```

where `'a'` denotes 'approximation coefficients', an alternative name for the trend signal in the time domain.

Likewise, we can calculate the i-th fluctuation signal in the time domain as (taking $i = 1$ as example):

```
i = 1;
ff1 = wrcoef( 'd', c2, l, 'db1', i);
```

where `'d'` denotes 'detail coefficients', an alternative name for the fluctuation signal in the time domain.

Exercises

Exercise 9.3.5-1: Calculate the 3-level Haar transform of the following discrete-time signal:

$$\vec{f} = \begin{bmatrix} -3, & 2, & 0, & 1, & 3, & 4, & 2, & 0 \end{bmatrix}$$

Exercise 9.3.5-2: Calculate the 3-level inverse Haar transform of the following discrete-time signal:

$$\vec{f} = \begin{bmatrix} -3, & 2, & 0, & 1, & 3, & 4, & 2, & 0 \end{bmatrix}$$

9.3.6 The level-n Haar transform from the perspective of wavelets

Again, let's take one step back and take a look at the matter from the perspective of an orthonormal transform.

The base of the 2-level Haar transform As the second level Haar transform leaves the second half of the wavelet coefficients untouched, this also means that these base vectors (the fluctuation base vectors of level 1) remain unchanged:

$$\vec{w}_4 = \frac{1}{\sqrt{2}}\left[\,1,\ \ -1,\ \ 0,\ \ 0,\ \ 0,\ \ 0,\ \ 0,\ \ 0\right]$$
$$\vec{w}_5 = \frac{1}{\sqrt{2}}\left[\,0,\ \ 0,\ \ 1,\ \ -1,\ \ 0,\ \ 0,\ \ 0,\ \ 0\right]$$
$$\vec{w}_6 = \frac{1}{\sqrt{2}}\left[\,0,\ \ 0,\ \ 0,\ \ 0,\ \ 1,\ \ -1,\ \ 0,\ \ 0\right]$$
$$\vec{w}_7 = \frac{1}{\sqrt{2}}\left[\,0,\ \ 0,\ \ 0,\ \ 0,\ \ 0,\ \ 0,\ \ 1,\ \ -1\right]$$

So far for the simple part. Now, what happens with the other base vectors? When considering the level-2 inverse calculation scheme (see page 232), and feeding it with vectors with all elements but one equal to zero, the new base vectors can be generated.

This yields two new scaling signals:

$$\vec{s}_{2,0} = \frac{1}{2}\left[\,1,\ \ 1,\ \ 1,\ \ 1,\ \ 0,\ \ 0,\ \ 0,\ \ 0\right]$$
$$\vec{s}_{2,1} = \frac{1}{2}\left[\,0,\ \ 0,\ \ 0,\ \ 0,\ \ 1,\ \ 1,\ \ 1,\ \ 1\right]$$

and two new wavelets:

$$\vec{w}_2 = \frac{1}{2}\left[\,1,\ \ 1,\ \ -1,\ \ -1,\ \ 0,\ \ 0,\ \ 0,\ \ 0\right]$$
$$\vec{w}_3 = \frac{1}{2}\left[\,0,\ \ 0,\ \ 0,\ \ 0,\ \ 1,\ \ 1,\ \ -1,\ \ -1\right]$$

Again, one can check that the set $B_{2\mathcal{H}} = \{\vec{s}_{2,0}, \vec{s}_{2,1}, \vec{w}_2, \vec{w}_3, \vec{w}_4, \vec{w}_5, \vec{w}_6, \vec{w}_7\}$ is independent, complete, orthogonal and normalized. Therefore it is an orthonormal base for S_8.

The base of the 3-level Haar transform Going to level 3 involves maintaining the final 6 wavelets obtained so far, i.e.:

$$\vec{w}_2 = \frac{1}{2}\left[\,1,\ \ 1,\ \ -1,\ \ -1,\ \ 0,\ \ 0,\ \ 0,\ \ 0\right]$$
$$\vec{w}_3 = \frac{1}{2}\left[\,0,\ \ 0,\ \ 0,\ \ 0,\ \ 1,\ \ 1,\ \ -1,\ \ -1\right]$$
$$\vec{w}_4 = \frac{1}{\sqrt{2}}\left[\,1,\ \ -1,\ \ 0,\ \ 0,\ \ 0,\ \ 0,\ \ 0,\ \ 0\right]$$
$$\vec{w}_5 = \frac{1}{\sqrt{2}}\left[\,0,\ \ 0,\ \ 1,\ \ -1,\ \ 0,\ \ 0,\ \ 0,\ \ 0\right]$$
$$\vec{w}_6 = \frac{1}{\sqrt{2}}\left[\,0,\ \ 0,\ \ 0,\ \ 0,\ \ 1,\ \ -1,\ \ 0,\ \ 0\right]$$
$$\vec{w}_7 = \frac{1}{\sqrt{2}}\left[\,0,\ \ 0,\ \ 0,\ \ 0,\ \ 0,\ \ 0,\ \ 1,\ \ -1\right]$$

We can then extend it with a new scaling signal:

$$\vec{s}_{3,0} = \frac{1}{2}\left[\,1,\ \ 1,\ \ 1,\ \ 1,\ \ 1,\ \ 1,\ \ 1,\ \ 1\right]$$

and a new wavelet:

$$\vec{w}_1 = \frac{1}{2}\begin{bmatrix} 1, & 1, & 1, & 1, & -1, & -1, & -1, & -1 \end{bmatrix}$$

The set obtained $B_{3\mathcal{H}} = \{\vec{s}_{3,0}, \vec{w}_1, \vec{w}_2, \vec{w}_3, \vec{w}_4, \vec{w}_5, \vec{w}_6, \vec{w}_7\}$ is an orthonormal base for S_8.

Generalization towards S_N Let's first generalize from S_8 to S_N. Let's define $M_n = N/(2^n)$.

It is logical to generalize the level-1 Haar scaling signals to:

$$\vec{s}_{1,0} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1, & 1, & 0, & 0, & 0, & 0, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{s}_{1,1} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 1, & 1, & 0, & 0, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{s}_{1,2} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 1, & 1, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{s}_{1,3} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 1, & 1, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vdots$$
$$\vec{s}_{1,M_1-2} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 1, & 1, & 0, & 0 \end{bmatrix}$$
$$\vec{s}_{1,M_1-1} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 1, & 1 \end{bmatrix}$$

Likewise, it is logical to generalize the level-1 Haar wavelets to:

$$\vec{w}_{M_1} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1, & -1, & 0, & 0, & 0, & 0, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{w}_{M_2} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 1, & -1, & 0, & 0, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{w}_{M_3} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 1, & -1, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{w}_{M_4} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 1, & -1, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vdots$$
$$\vec{w}_{N-2} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 1, & -1, & 0, & 0 \end{bmatrix}$$
$$\vec{w}_{N-1} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 1, & -1 \end{bmatrix}$$

The generalization of the level-2 Haar scaling signals should not surprise the reader:

$$\vec{s}_{2,0} = \frac{1}{2}\begin{bmatrix} 1, & 1, & 1, & 1, & 0, & 0, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{s}_{2,1} = \frac{1}{2}\begin{bmatrix} 0, & 0, & 0, & 0, & 1, & 1, & 1, & 1, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vdots$$
$$\vec{s}_{2,M_2-2} = \frac{1}{2}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \cdots, & 1, & 1, & 1, & 1, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{s}_{2,M_2-1} = \frac{1}{2}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 1, & 1, & 1, & 1 \end{bmatrix}$$

Neither should the generalization of the level-2 Haar wavelets:

$$\vec{w}_{M_2} = \frac{1}{2}\begin{bmatrix} 1, & 1, & -1, & -1, & 0, & 0, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{w}_{M_2+1} = \frac{1}{2}\begin{bmatrix} 0, & 0, & 0, & 0, & 1, & 1, & -1, & -1, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vdots$$
$$\vec{w}_{M_1-2} = \frac{1}{2}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \cdots, & 1, & 1, & -1, & -1, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vec{w}_{M_1-1} = \frac{1}{2}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \cdots, & 0, & 0, & 0, & 0, & 1, & 1, & -1, & -1 \end{bmatrix}$$

Likewise for level-3 Haar scaling signals:

$$\vec{s}_{3,0} = \frac{1}{2\sqrt{2}}\begin{bmatrix} 1, & 1, & 1, & 1, & 1, & 1, & 1, & 1, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vdots$$
$$\vec{s}_{3,M_3-1} = \frac{1}{2\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \cdots, & 1, & 1, & 1, & 1, & 1, & 1, & 1, & 1 \end{bmatrix}$$

and level-3 Haar wavelets:

$$\vec{w}_{M_3} = \frac{1}{2\sqrt{2}}\begin{bmatrix} 1, & 1, & 1, & 1, & -1, & -1, & -1, & -1, & \cdots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$
$$\vdots$$
$$\vec{w}_{M_2-1} = \frac{1}{2\sqrt{2}}\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \cdots, & 1, & 1, & 1, & 1, & -1, & -1, & -1, & -1 \end{bmatrix}$$

By now, you will have picked up the pattern.

The base of the n-level Haar transform Straight analysis reveals that we can use the following procedure to generate the base of the n-level Haar transform. We will not spend time proving these relationships. The ambitious reader can spend some time on it on a rainy afternoon.

Again, we define:

$$M_n = N/(2^n)$$

For the n-level Haar transform of an N-point signal, one takes the $N - M_n$ wavelets of the $n-1$-level Haar transform and adds to that

- M_n new scaling signals $s_{n,i}$ according to

$$s_{n,i} = \frac{1}{\sqrt{2}} s_{n-1,2i} + \frac{1}{\sqrt{2}} s_{n-1,2i+1}$$

  with $i = 0, 1, \ldots, M_n - 1$;
- M_n new wavelets w_{M_n+i} according to

$$w_{M_n+i} = \frac{1}{\sqrt{2}} s_{n-1,2i} - \frac{1}{\sqrt{2}} s_{n-1,2i+1}$$

  with $i = 0, 1, \ldots, M_n - 1$.

These recursion relationships are initialized by defining:

$$s_{0,i}[n] = \delta[n-i]$$

or more elaborately:

$$\begin{aligned}
\vec{s}_{0,0} &= \begin{bmatrix} 1, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \ldots, & 0, & 0, & 0, & 0 \end{bmatrix} \\
\vec{s}_{0,1} &= \begin{bmatrix} 0, & 1, & 0, & 0, & 0, & 0, & 0, & 0, & \ldots, & 0, & 0, & 0, & 0 \end{bmatrix} \\
\vec{s}_{0,2} &= \begin{bmatrix} 0, & 0, & 1, & 0, & 0, & 0, & 0, & 0, & \ldots, & 0, & 0, & 0, & 0 \end{bmatrix} \\
&\vdots \\
\vec{s}_{0,N-2} &= \begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \ldots, & 0, & 0, & 1, & 0 \end{bmatrix} \\
\vec{s}_{0,N-1} &= \begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \ldots, & 0, & 0, & 0, & 1 \end{bmatrix}
\end{aligned}$$

This means that the natural base (related to the impulse decomposition) is the 0-level Haar base and consists solely of scaling signals.

These recursion relationships carry the essence of the hierarchical decomposition nature of wavelets. They are the heart of the wavelet transform. We will elaborate on it in section 9.3.7 on page 240.

Exercises

Exercise 9.3.6-1: For a Haar transform with $N = 64$, determine $\vec{s}_{1,17}$.

Exercise 9.3.6-2: For a Haar transform with $N = 64$, determine $\vec{w}_{35}$.

Exercise 9.3.6-3: For a Haar transform with $N = 64$, determine $\vec{s}_{4,3}$.

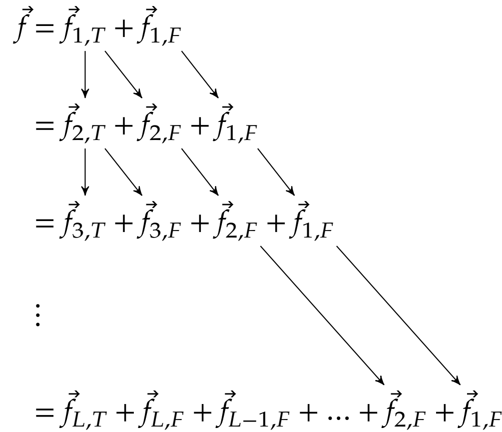
Exercise 9.3.6-4: For a Haar transform with $N = 64$, determine $\vec{w}_9$.

Exercise 9.3.6-5: For a Haar transform with $N = 64$, determine $\vec{s}_{2,20}$.

9.3.7 Multi-resolution analysis

The Haar transform converts our signal into two subframes, a trend and a fluctuation subframe. Each of these frames can be associated to a time-domain signal: the trend signal in the time domain and the fluctuation signal in the time domain. This became apparent in the earlier derived equation (9.3.3).

Now, let's take it one step further and continue decomposing our signal in the time domain. We can do this as long as the trend subframe can be divided in two. Symbolically, for a signal of $N = 2^L$ points:



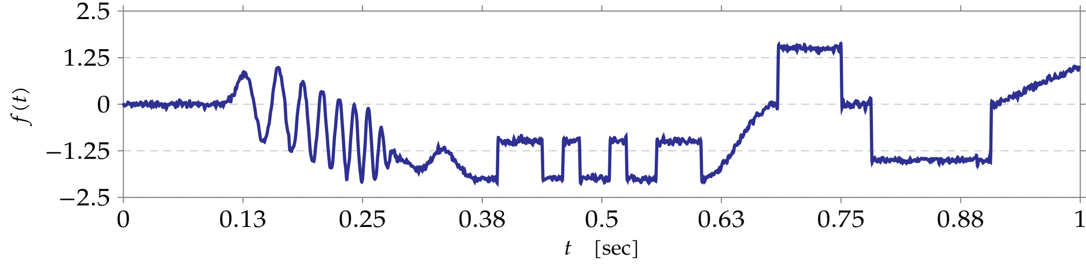
$$
\begin{aligned}
\vec{f} &= \vec{f}_{1,T} + \vec{f}_{1,F} \\
&= \vec{f}_{2,T} + \vec{f}_{2,F} + \vec{f}_{1,F} \\
&= \vec{f}_{3,T} + \vec{f}_{3,F} + \vec{f}_{2,F} + \vec{f}_{1,F} \\
&\vdots \\
&= \vec{f}_{L,T} + \vec{f}_{L,F} + \vec{f}_{L-1,F} + \ldots + \vec{f}_{2,F} + \vec{f}_{1,F}
\end{aligned}
$$

In fact, $\vec{f}_{L,T}$ is a version of $\vec{f}$ on the most coarse resolution. The corresponding scaling signal consists of all ones (except for the scaling factor). The L-th trend signal therefore corresponds to the DC-level of the signal $\vec{f}$. The L-th fluctuation signal adds a little more detail to that. It corresponds to a wavelet vector containing one in the first half of the positions and minus one in the second half of the positions (again, except for the scaling factor). The $L-1$-th fluctuation again adds a litte more detail to that.

In this sense, the fluctuations keep adding an increased level of detail to the signal in the time domain.

Analyzing a signal from a coarse to a fine resolution is called *multi-resolution analysis*.

An example will make things more clear. Consider the continuous-time signal $f(t)$ below:



It starts off with a frequency sweeping sine-wave burst, going down towards a steady level, followed by some square-wave pulses, going up again, followed by a wider rectangular positive pulse and negative pulse, ending with a linearly increasing part at the very end. It also clearly exhibits some noise.

Then, we sample it at 256Hz, to obtain the 256-point discrete time signal $f[n]$.

The latter signal and its Haar transform on multiple levels can be found in Figure 9.5 on page 242 in the left column, while the cumulative energy distribution is displayed in the right column.

This cumulative energy distribution is simply defined as the following running sum:

$$CE[k] = \sum_{i=0}^{k} c^2[i]$$

Take a look at the left-hand column, second row. One can clearly see that the 1-level Haar transform results in a trend subframe (left-half of the picture) that looks like a compacted version of the original signal (in the top row). This is also visible on the cumulative energy distribution (right-hand column): the total energy is almost fully present in the trend part. The corresponding fluctuation frame (the right-half of the picture) shows the 'difference' w.r.t. the original and is in general much smaller than the trend signal. There is one notable exception: the sine-wave burst results in a larger fluctuation part. The reason is obvious: the Haar transform is good at detecting a constant trend, but not at detecting a sine-wave trend.

The corresponding trend and fluctuation signals in the time-domain can be found in Figure 9.6 on page 243. Again, take a look at the left-hand-column, second row. One can see that the trend signal in the time domain is a pretty good approximation of the original signal. Again, the approximation for the sine-wave part is not so good.

When you go down along the rows on Figure 9.5 on page 242, you will see that the energy compaction further continues until ${}_4\mathcal{H}$, but afterwards there's no further improvement. In addition, the truthfulness of trend time-domain signals deteriorates quickly after ${}_1\mathcal{H}$ for non-constant parts, because of the fact that the Haar transform is only good at approximating piecewise constant signals.

The left-hand column of Figure 9.6 on page 243 actually shows the multi-resolution analysis in its very essence. The bottom row, shows the signal at a 1-point resolution. The one but last row shows it at a 2-point resolution. Going further up, you encounter a 4-point resolution, 8-point resolution, 16-point resolution (for ${}_4\mathcal{H}$), a.s.o. until the top row, displaying the signal at its original 256-point resolution.

9.4 Wavelet construction

Instead of just listing more arbitrary wavelets, let's take a while to analyze how wavelets (even the most simple of them, the Haar wavelets) are constructed.

To simplify things a bit, we will limit ourselves in this section to wavelets that on level 1 have a support of an even number of points L. To ease the mathematical discourse below, we will set $L = 6$, but the principles outlined are easily generalized for any arbitrary even L.

9.4.1 The essence of wavelets

The basic idea of wavelet construction is to:

1. generate some scaling signals and wavelets on level 1 to detect a specific trend and fluctuation
2. use a recursion formula to generate the scaling signals and additional wavelets of the higher levels.

We keep using the auxiliary variable: $M_n = N/2^n$.

And we'll use some (for now) arbitrary numbers α_i, the so-called *scaling numbers* and β_i, the so-called *wavelet numbers*.

The scaling signals on level 1 are generically defined by:

$$\begin{aligned}
\vec{s}_{1,0} &= [\,\alpha_0, \;\alpha_1, \;\alpha_2, \;\alpha_3, \;0, \;0, \;0, \;0, \;\dots, \;0, \;0, \;\alpha_{-2}, \;\alpha_{-1}] \\
\vec{s}_{1,1} &= [\,\alpha_{-2}, \;\alpha_{-1}, \;\alpha_0, \;\alpha_1, \;\alpha_2, \;\alpha_3, \;0, \;0, \;\dots, \;0, \;0, \;0, \;0] \\
\vec{s}_{1,2} &= [\,0, \;0, \;\alpha_{-2}, \;\alpha_{-1}, \;\alpha_0, \;\alpha_1, \;\alpha_2, \;\alpha_3, \;\dots, \;0, \;0, \;0, \;0] \\
&\;\vdots \\
\vec{s}_{1,M_1-2} &= [\,0, \;0, \;0, \;0, \;0, \;0, \;0, \;0, \;\dots, \;\alpha_0, \;\alpha_1, \;\alpha_2, \;\alpha_3] \\
\vec{s}_{1,M_1-1} &= [\,\alpha_2, \;\alpha_3, \;0, \;0, \;0, \;0, \;0, \;0, \;\dots, \;\alpha_{-2}, \;\alpha_{-1}, \;\alpha_0, \;\alpha_1]
\end{aligned}$$

Note that the scaling numbers of the first scaling signal are centered on position column zero, such that there's one more scaling number to the right than there are scaling numbers to left. This specific balancing makes sure that higher-order scaling base vectors do not drift to the right.

Similarly, the wavelets on level 1 are defined by:

$$\begin{aligned}
\vec{w}_{M_1} &= [\,\beta_0, \;\beta_1, \;\beta_2, \;\beta_3, \;0, \;0, \;0, \;0, \;\dots, \;0, \;0, \;\beta_{-2}, \;\beta_{-1}] \\
\vec{w}_{M_1+1} &= [\,\beta_{-2}, \;\beta_{-1}, \;\beta_0, \;\beta_1, \;\beta_2, \;\beta_3, \;0, \;0, \;\dots, \;0, \;0, \;0, \;0] \\
\vec{w}_{M_1+2} &= [\,0, \;0, \;\beta_{-2}, \;\beta_{-1}, \;\beta_0, \;\beta_1, \;\beta_2, \;\beta_3, \;\dots, \;0, \;0, \;0, \;0] \\
&\;\vdots \\
\vec{w}_{N-2} &= [\,0, \;0, \;0, \;0, \;0, \;0, \;0, \;0, \;\dots, \;\beta_0, \;\beta_1, \;\beta_2, \;\beta_3] \\
\vec{w}_{N-1} &= [\,\beta_2, \;\beta_3, \;0, \;0, \;0, \;0, \;0, \;0, \;\dots, \;\beta_{-2}, \;\beta_{-1}, \;\beta_0, \;\beta_1]
\end{aligned}$$

Note that the wavelet numbers of the first wavelet are centered on position column zero, such that there's one more wavelet number to the right than there are wavelet numbers to left. This specific balancing makes sure that higher-order wavelet base vectors do not drift to the right.

Note the wrap-around of the scaling and wavelet numbers for first few and last few scaling signals and wavelets. If one does not like this wrap-around (that in fact assumes that the analyzed signal is periodical), one can pad the signal

- with zeros, or
- by mirroring the neighboring values.

One can prove that if these scaling signals and wavelets form an orthonormal base, the following recursion relationships also yield the new scaling signals and wavelets for the

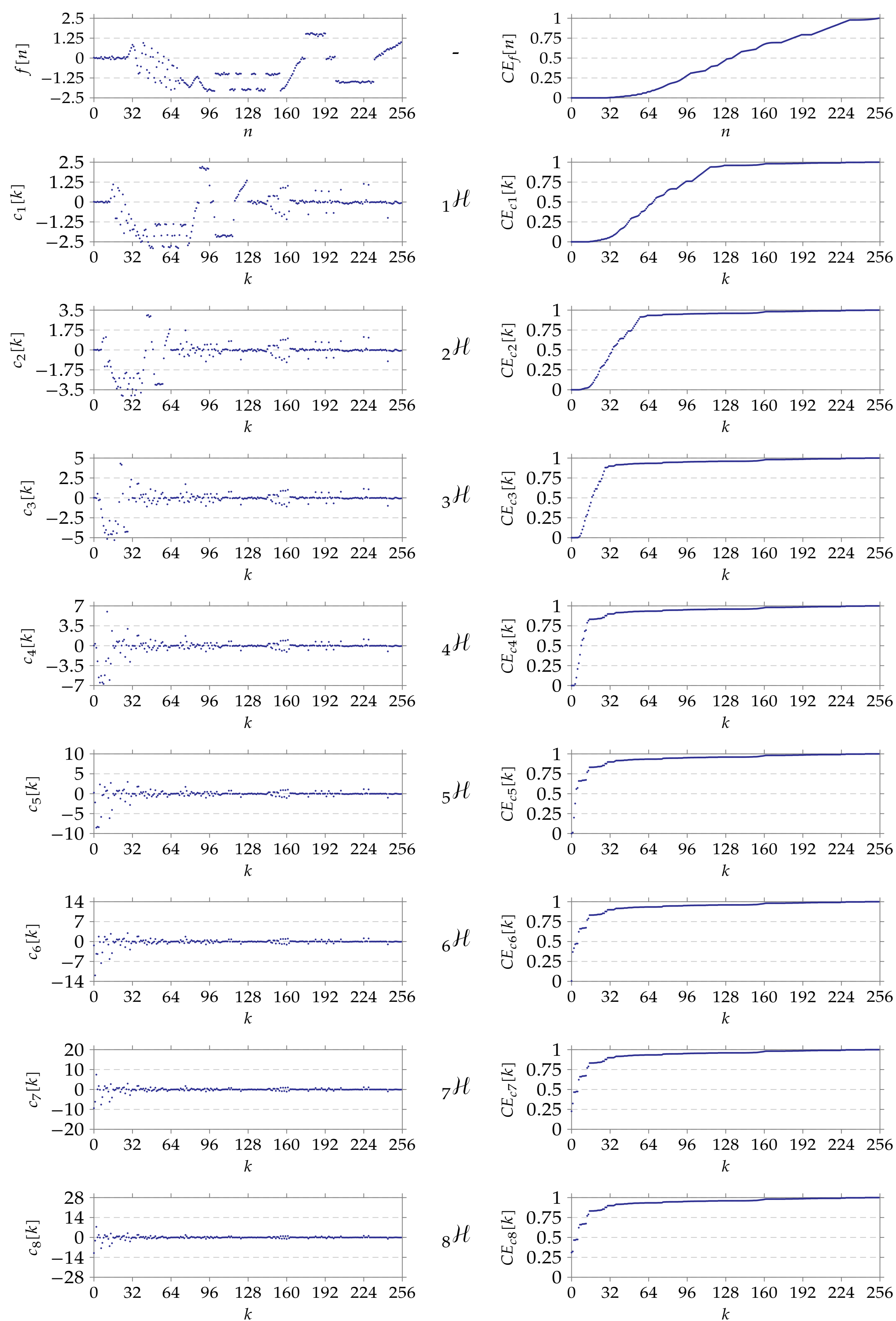


Figure 9.5: Example signal $f[n]$ with its Haar transforms (${}_1\mathcal{H}$ to ${}_8\mathcal{H}$); the left column contains the (transformed) signal, the right column contains its cumulative energy distribution.

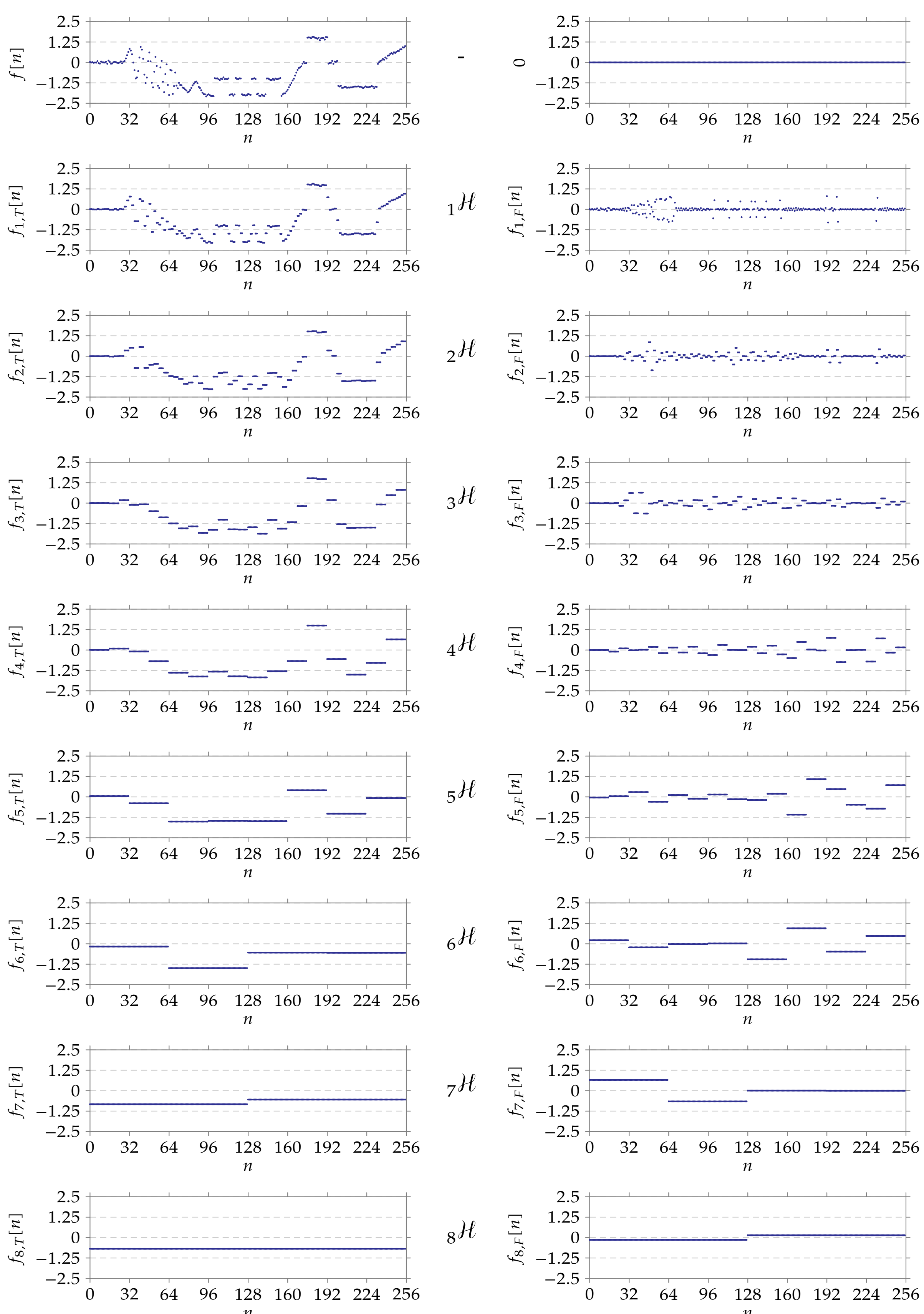


Figure 9.6: Example signal $f[n]$ with its trend and fluctuation signals in the time domain corresponding to the Haar transforms; left column contains the trend signals in the time domain, right column contains the fluctuation signals in the time domain.

higher levels, i.e. ensuring all the characteristics of an orthonormal base.

$$\begin{aligned}
\vec{s}_{n,i} &= \sum_k \alpha_k \vec{s}_{n-1,2i+k} \\
&= \alpha_{-2}\vec{s}_{n-1,2i-2} + \alpha_{-1}\vec{s}_{n-1,2i-1} + \alpha_0\vec{s}_{n-1,2i} + \alpha_1\vec{s}_{n-1,2i+1} + \alpha_2\vec{s}_{n-1,2i+2} + \alpha_3\vec{s}_{n-1,2i+3} \\
\vec{w}_{M_n+i} &= \sum_k \beta_k \vec{s}_{n-1,2i+k} \\
&= \beta_{-2}\vec{s}_{n-1,2i-2} + \beta_{-1}\vec{s}_{n-1,2i-1} + \beta_0\vec{s}_{n-1,2i} + \beta_1\vec{s}_{n-1,2i+1} + \beta_2\vec{s}_{n-1,2i+2} + \beta_3\vec{s}_{n-1,2i+3}
\end{aligned}$$

(9.136)

(9.137)

In these equations $i = 0, 1, \ldots, M_n - 1$.

Again to make things more general, we can define the level-0 scaling signals to equal the natural base:

$$s_{0,i}[n] = \delta[n-i]$$

Again, one has to consider the scaling base vectors to be periodical. In general:

$$s_{k,-i}[n] = s_{k,M_k-i}$$

E.g., $\delta[n+1] = \delta[N-1]$ and $\vec{s}_{1,-2} = \vec{s}_{1,N/2-2}$.

9.4.2 Finding scaling and wavelet numbers

Given the procedure outlined in the section above, there is only one remaining issue: how do we find the scaling numbers α_i and the wavelet numbers β_i?

A number of boundary conditions are imminent:

- The scaling signals and wavelets of level 1 need to be normalized, i.e.:

$$\alpha_{-2}^2 + \alpha_{-1}^2 + \alpha_0^2 + \alpha_1^2 + \alpha_2^2 + \alpha_3^2 = 1 \quad (9.138)$$

$$\beta_{-2}^2 + \beta_{-1}^2 + \beta_0^2 + \beta_1^2 + \beta_2^2 + \beta_3^2 = 1 \quad (9.139)$$

  For any value of L this condition yields 2 equations.

- The scaling signals of level 1 need to be internally orthogonal:

$$\alpha_0\alpha_{-2} + \alpha_1\alpha_{-1} + \alpha_2\alpha_0 + \alpha_3\alpha_1 = 0 \quad (9.140)$$

$$\alpha_2\alpha_{-2} + \alpha_3\alpha_{-1} = 0 \quad (9.141)$$

  For any value of L this condition yields $L/2 - 1$ equations.

- The wavelets of level 1 need to be internally orthogonal:

$$\beta_0\beta_{-2} + \beta_1\beta_{-1} + \beta_2\beta_0 + \beta_3\beta_1 = 0 \quad (9.142)$$

$$\beta_2\beta_{-2} + \beta_3\beta_{-1} = 0 \quad (9.143)$$

  For any value of L this condition yields $L/2 - 1$ equations.

- The scaling signals and wavelets of level 1 need to be mutually orthogonal:

$$\alpha_{\text{-}2}\beta_{\text{-}2} + \alpha_{\text{-}1}\beta_{\text{-}1} + \alpha_0\beta_0 + \alpha_1\beta_1 + \alpha_2\beta_2 + \alpha_3\beta_3 = 0 \tag{9.144}$$

$$\alpha_0\beta_{\text{-}2} + \alpha_1\beta_{\text{-}1} + \alpha_2\beta_0 + \alpha_3\beta_1 = 0 \tag{9.145}$$

$$\alpha_2\beta_{\text{-}2} + \alpha_3\beta_{\text{-}1} = 0$$

$$\beta_0\alpha_{\text{-}2} + \beta_1\alpha_{\text{-}1} + \beta_2\alpha_0 + \beta_3\alpha_1 = 0$$

$$\beta_2\alpha_{\text{-}2} + \beta_3\alpha_{\text{-}1} = 0 \tag{9.146}$$

  For any value of L this condition yields $L - 1$ equations.

The number of equations adds up to $2L - 1$, the number of unknowns totals $2L$, so apparently there's only one degree of freedom left. However, this is not correct. The equations above are not all independent.

Making the following choice

$$\begin{aligned}
\alpha_{\text{-}2} &= -\beta_3 \\
\alpha_{\text{-}1} &= \beta_2 \\
\alpha_0 &= -\beta_1 \\
\alpha_1 &= \beta_0 \\
\alpha_2 &= -\beta_{\text{-}1} \\
\alpha_3 &= \beta_{\text{-}2}
\end{aligned}$$

reduces the number of effective unknowns to L (the values of β_i). It also automatically ensures (9.144) and given (9.140)-(9.141) ensures (9.145)-(9.146). Additionally, given (9.139), it also realizes (9.138) and given (9.142) and (9.143) it also realizes (9.140) and (9.141).

Therefore, for the remaining number of unkowns (the values of β_i), only the following equations remain:

$$\begin{aligned}
\beta_{\text{-}2}^2 + \beta_{\text{-}1}^2 + \beta_0^2 + \beta_1^2 + \beta_2^2 + \beta_3^2 &= 1 \\
\beta_0\beta_{\text{-}2} + \beta_1\beta_{\text{-}1} + \beta_2\beta_0 + \beta_3\beta_1 &= 0 \\
\beta_2\beta_{\text{-}2} + \beta_3\beta_{\text{-}1} &= 0
\end{aligned}$$

In general, this adds up to $L/2$ equations, leaving $L/2$ degrees of freedom. We will refer to this set of equations as *the mandatory boundary conditions*.

And then the creative part starts: we can impose extra constraints to fix the remaining degrees of freedom. We'll refer to these extra constraints as *the free boundary conditions*.

9.4.3 The Haar transform revisited

The wavelet and scaling numbers of the Haar transform can easily be derived using the procedure outline in the previous section.

Writing down the mandatory boundary conditions for the Haar transform, yields:

$$\beta_0^2 + \beta_1^2 = 1$$

This leaves us one additional free boundary condition to choose.

The essence of the Haar transform was to make sure that the fluctuation would be zero in case the signal was constant. Naming this constant signal value c, this implies:

$$\beta_0 c + \beta_1 c = 0$$

Solving this system of two equations yields two possible solutions:

$$A : \begin{cases} \beta_0 = \frac{1}{\sqrt{2}} \\ \beta_1 = -\frac{1}{\sqrt{2}} \end{cases} \qquad B : \begin{cases} \beta_0 = -\frac{1}{\sqrt{2}} \\ \beta_1 = \frac{1}{\sqrt{2}} \end{cases}$$

Both solutions are equally good, and give the same results (except for sign change). Usually, solution A is chosen.

This fully defines the Haar transform, as:

$$\beta_0 = \frac{1}{\sqrt{2}} \qquad \alpha_0 = -\beta_1 = \frac{1}{\sqrt{2}}$$
$$\beta_1 = -\frac{1}{\sqrt{2}} \qquad \alpha_1 = \beta_0 = \frac{1}{\sqrt{2}}$$

9.5 Wavelet transforms — an overview

Usually, when one speaks about wavelet transforms, we mean the orthonormal wavelet transforms (see Figure 9.3 on page 219). In this category, we will discuss

- Haar wavelets
- Daubechies wavelets
- Coifman wavelets

When speaking about wavelets shortly, we usually refer to wavelets and scaling signals.

We will also treat a different category, the biorthonormal wavelets (see Figure 9.3). In this category, we will discuss

- the Cohen-Daubechies-Feauveau wavelets

In MATLAB one can get more information about wavelets using the `waveinfo` command. E.g., to obtain generic information, issue:

```
waveinfo
```

To obtain information about Coifman wavelets, issue:

```
waveinfo( 'coif' )
```

9.5.1 Haar wavelets

So far, we've seen quite a lot on Haar wavelets (see section 9.3 on page 219). To complete our image, let's consider how they look for the different levels of the transform.

To avoid the manual labor of calculating the higher-order wavelets using the recursion formulae (9.136) and (9.137), we have MATLAB do the work for us. The following script can be used to generate a view on a wavelet.

```
wavelet = 'db1';      % select the wavelet of choice
depth = 5;            % the maximum level to consider
N = 128;              % the length of our signal
x = zeros(1,N);
dwtmode('per');       % set the mode (see section on 'Edge effects')

for depth = 1:5

  % calculate the border between scaling signals and wavelets
  Mn = N / (2^depth);

  % decompose
  [c,l] = wavedec(x,depth,wavelet);

  % generate a natural base vector mid-way the scaling coefficient region
  spot = 1/2 * Mn + 1;
  x(spot) = 1;

  % generate the corresponding scaling base vector and plot it
  s = waverec(x,l,wavelet);
  subplot(5,2,2*depth-1);
  plot(s,'.');
  x(spot) = 0;        % remove the scaling spot

  % generate a natural base vector mid-way the wavelet coeffient region
  spot = 3/2 * Mn + 1;
  x(spot) = 1;

  % generate the corresponding wavelet base vector and plot it
  w = waverec(x,l,wavelet);
  subplot(5,2,2*depth);
  plot(w,'.');
  x(spot) = 0;        % remove the wavelet spot

end
```

The result for the Haar wavelets can be found in Figure 9.7.

9.5.2 Daubechies wavelets

The Daubechies wavelets improve on the basic Haar idea. The Haar wavelets intend to discover a constant trend. The Daub-4 wavelets intend to discover a linear trend, the Daub-6 wavelets a quadratic trend and so on.

The number J of the Daub-J wavelets indicates the support length of the wavelets on level 1.

The Daub-2 wavelet transform actually corresponds to the Haar transform.

Notation Again the Daubechies wavelet transform is a multi-level transform for which we'll use a notation that is very similar to the Haar notation. The only new thing is that the north west subscript indicates the support length J of the level-1 Daubechies transform.

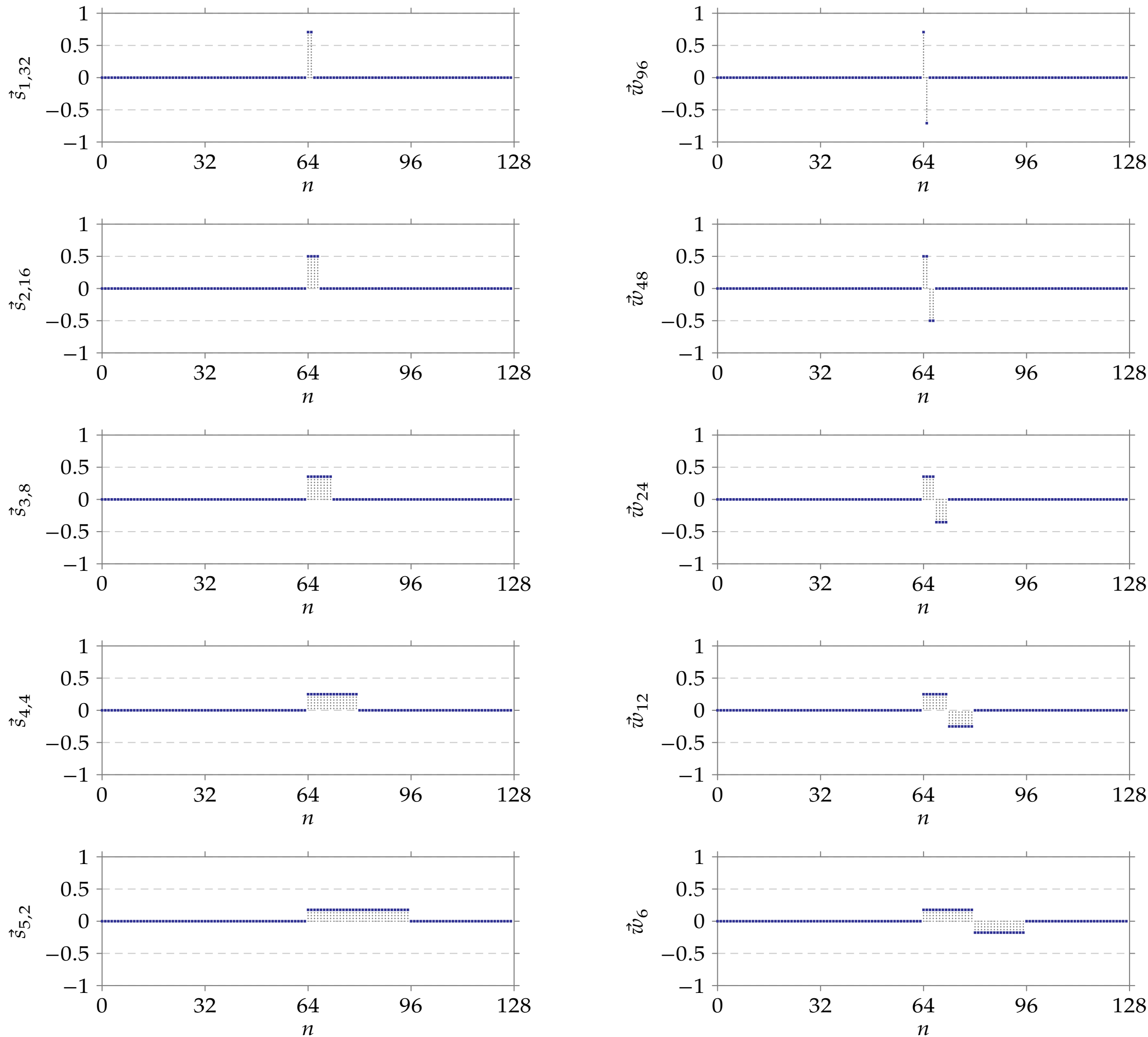
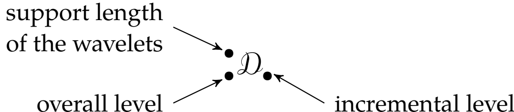
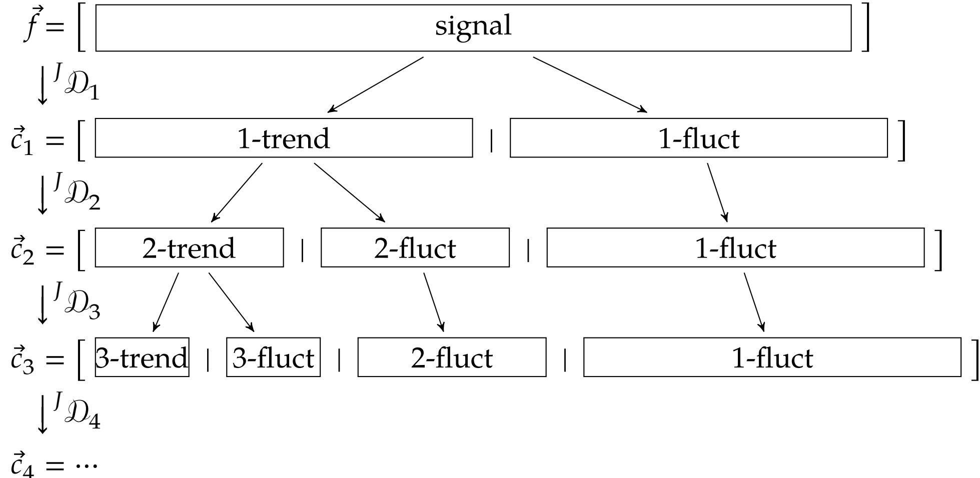


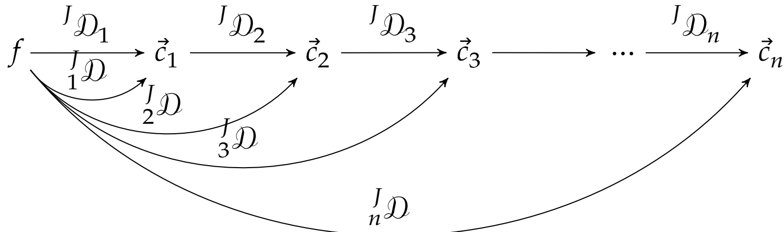
Figure 9.7: Illustration of the Haar scaling signals (left column) and wavelets (right column)



Using this notation, the following decomposition is most similar to the Haar decomposition:



The relationship between the n-level Daubechies transform and the level-n Daubechies transform, for n integer is given below:



Daubechies-4 wavelets (Daub-4)

The Daub-4 wavelets are wavelets with support length 4. Therefore, we have 4 wavelet numbers β_{-1}, β_0, β_1 and β_2 and 4 scaling numbers, α_{-1}, α_0, α_1 and α_2.

Writing down the mandatory boundary conditions yields:

$$\begin{aligned} \beta_{-1}^2 + \beta_0^2 + \beta_1^2 + \beta_2^2 &= 1 \\ \beta_{-1}\beta_1 + \beta_0\beta_2 &= 0 \end{aligned}$$

That leaves us two free boundary conditions of choice. We'll impose that a linear trend, represented by $f[n] = an + b$ with a and b arbitrary scalars, must result in a zero fluctuation. This means that for any a, b and time point n the following should hold:

$$\beta_{-1}(a(n-1)+b) + \beta_0(an+b) + \beta_1(a(n+1)+b) + \beta_2(a(n+2)+b) = 0$$

Sorting this equation w.r.t. a and b yields:

$$((n-1)\beta_{-1} + n\beta_0 + (n+1)\beta_1 + (n+2)\beta_2)a + (\beta_{-1} + \beta_0 + \beta_1 + \beta_2)b = 0$$

Given the fact that this equation should hold for any a and b, this implies:

$$\begin{cases} (n-1)\beta_{-1} + n\beta_0 + (n+1)\beta_1 + (n+2)\beta_2 = 0 \\ \beta_{-1} + \beta_0 + \beta_1 + \beta_2 = 0 \end{cases}$$

which can be reduced by subtracting n times the second equation from the first, to:

$$\begin{cases} -1\beta_{-1} + 0\beta_0 + 1\beta_1 + 2\beta_2 = 0 \\ \beta_{-1} + \beta_0 + \beta_1 + \beta_2 = 0 \end{cases} \tag{9.156}$$

Solving the system that consists of these equations and the mandatory boundary conditions is easily done, by subsitution. It yields two solutions, the values below and the same values but in reverse order.

$$\beta_{-1} = \frac{1-\sqrt{3}}{4\sqrt{2}} \qquad \alpha_{-1} = \frac{1+\sqrt{3}}{4\sqrt{2}}$$

$$\beta_0 = -\frac{3-\sqrt{3}}{4\sqrt{2}} \qquad \alpha_0 = \frac{3+\sqrt{3}}{4\sqrt{2}}$$

$$\beta_1 = \frac{3+\sqrt{3}}{4\sqrt{2}} \qquad \alpha_1 = \frac{3-\sqrt{3}}{4\sqrt{2}}$$

$$\beta_2 = -\frac{1+\sqrt{3}}{4\sqrt{2}} \qquad \alpha_2 = \frac{1-\sqrt{3}}{4\sqrt{2}}$$

The values α_i have been obtained using:

$$\alpha_{-1} = -\beta_2$$
$$\alpha_0 = \beta_1$$
$$\alpha_1 = -\beta_0$$
$$\alpha_2 = \beta_{-1}$$

The resulting scaling signals and wavelets have been illustrated in Figure 9.8.

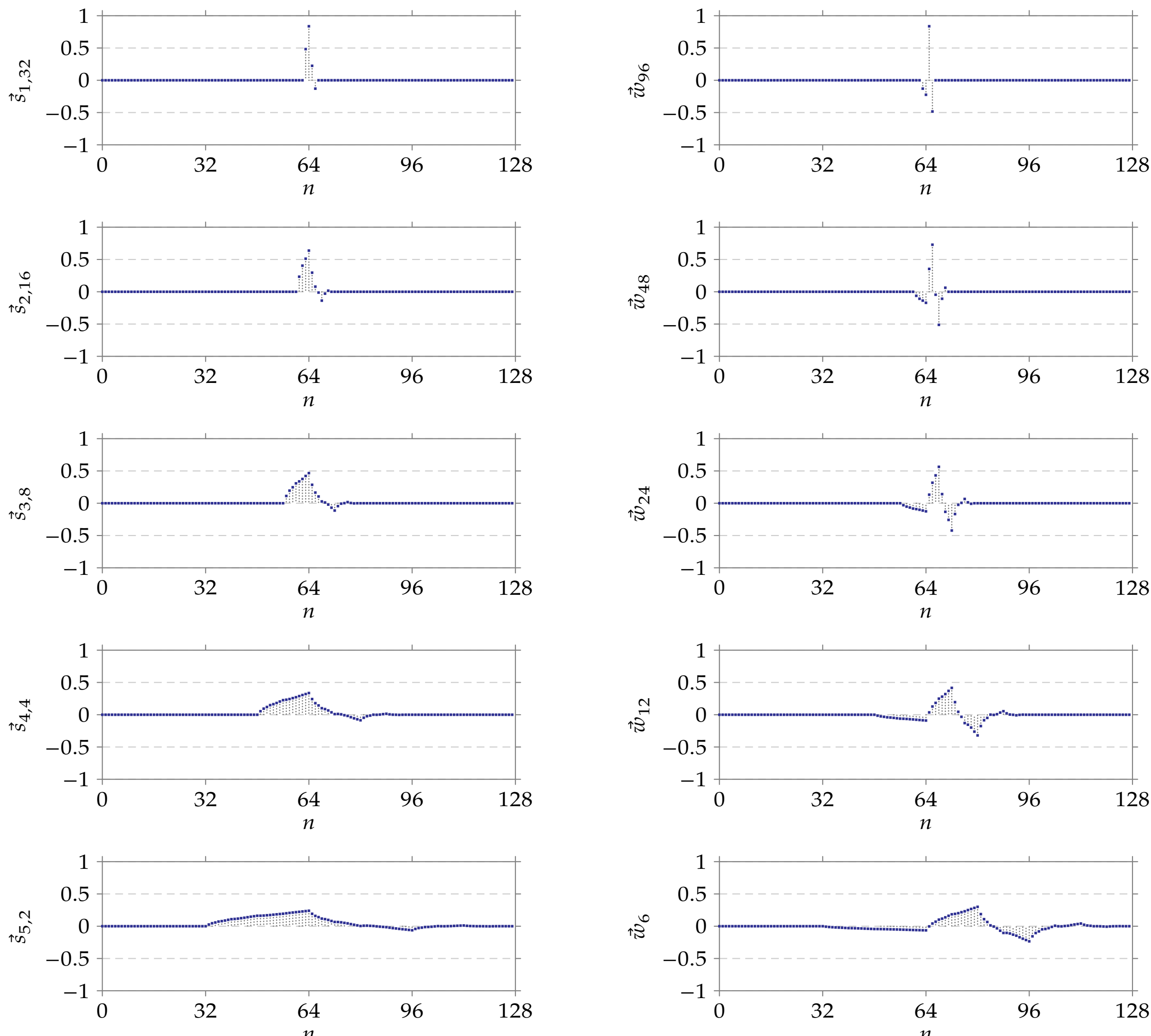


Figure 9.8: Illustration of the Daub-4 scaling signals (left column) and wavelets (right column)

As these scaling and wavelet numbers originate from the fact that linear signals yield zero fluctuation values, the performance will be excellent for near-linear signals.

We can extend this into a more generic theorem, that states that in general, for real-life signals, the fluctuations tend to be small.

Daub-4 small fluctuations theorem For a signal that originated from sampling a continuous-time signal with continuous derivatives up to second order, the Daub-4 transform generates small fluctuation signals. Assuming $M_k = N/2^k$:

$$c_n[M_k + i] \approx O\left(T_s^2\right), \qquad i = 0, 1, \dots, M_k - 1$$

with T_s the sampling period.

Note: you can find more information on the Big-O notation $O(\bullet)$ in section B.3 on page 348.

Proof for the case $k = 1$:
Consider the continuous time signal $f(t)$ that has been sampled with a sampling period T_s, yielding a discrete-time signal $f[n]$:

$$f[n] = f(nT_s)$$

given the fact that $f(t)$ has continuous first- and second-order derivatives, we can use Taylor's theorem (see section B.2 on page 348) and find a $c \in [0, \Delta t]$ such that:

$$f(t_0 + \Delta t) = f(t_0) + f'(t_0)(\Delta t) + \frac{f''(t_0 + c)}{2}(\Delta t)^2 \tag{9.157}$$

Using the Big-O notation, we abbreviate this as:

$$f(t_0 + \Delta t) = f(t_0) + f'(t_0)(\Delta t) + O\left((\Delta t)^2\right) \tag{9.158}$$

Now, let's project this function on an arbitrary wavelet vector. This means calculating the scalar product:

$$c_1[M_1 + i] = \left\langle \vec{f}, \vec{w}_{M_1+i} \right\rangle$$

If we make this scalar product explicit in terms of the scaling numbers, we obtain:

$$c_1[M_1 + i] = \beta_{-1} f[2i-1] + \beta_0 f[2i] + \beta_1 f[2i+1] + \beta_2 f[2i+2]$$

However, given (9.158) and assuming that the discrete time point with index $2i$ corresponds to $t = t_0$ we know that:

$$\begin{aligned}
\beta_{-1} f[2i-1] &= \beta_{-1}\left(f(t_0) + f'(t_0)(-T_s) + O\left(T_s^2\right)\right) \\
\beta_0 f[2i] &= \beta_0\left(f(t_0)\right) \\
\beta_1 f[2i+1] &= \beta_1\left(f(t_0) + f'(t_0)(T_s) + O\left(T_s^2\right)\right) \\
\beta_2 f[2i+2] &= \beta_2\left(f(t_0) + f'(t_0)(2T_s) + O\left(T_s^2\right)\right)
\end{aligned}$$

Summing the left-hand and right-hand sides and rearranging the terms yields:

$$\begin{aligned}
c_1[M_n + i] = {} & (\beta_{-1} + \beta_0 + \beta_1 + \beta_2) f(t_0) \\
& + f'(t_0) T_s \left(-1\beta_{-1} + 0\beta_0 + 1\beta_1 + 2\beta_2\right) \\
& + O\left(T_s^2\right)
\end{aligned}$$

However, in view of (9.156) this can be simplified to:

$$c_1[M_n + i] = O\left(T_s^2\right)$$

This proves the theorem for $n = 1$. ■

For subsequent transformation levels, the theorem will also hold but the term $O\left(T_s^2\right)$ *will* become big.

To complete our discussion of the Daubechies wavelets of length 4, we ran our 256-point test signal of page 239 through the multilevel Daub-4 wavelet transform. The results can be found in Figure 9.9 on the following page and its facing-page figure. Compare these graphs with the graphs of Figure 9.5 and its facing page. You will agree that it is very difficult to see the improvement over the Haar-wavelets. As we can see, this transform is lousy at tracking the sine-wave burst.

The increased quality can better be observed by comparing the percentages of the energy that ended up in the different trend signals. We made a comparison in Table 9.1 on page 256. Don't be disappointed by the fact that the fraction of energy in the trend subframes goes down for increasing levels. Keep in mind that for every level increase, the length of the trend subframe halves. Table 9.2 on page 256 takes that into account by considering the energy fraction *per sample*.

Daubechies-6 wavelets (Daub-6)

The essence of the Daub-6 transform is to make the fluctuation values of a quadratic signal $f[n] = an^2 + bn + c$ zero.
Using a similar derivation as for the Daub-4 wavelets, one can find:

$$\begin{aligned}
\beta_{-2}^2 + \beta_{-1}^2 + \beta_0^2 + \beta_1^2 + \beta_2^2 + \beta_3^2 &= 1 \\
\beta_{-2}\beta_0 + \beta_{-1}\beta_1 + \beta_0\beta_2 + \beta_1\beta_3 &= 0 \\
\beta_{-2}\beta_2 + \beta_{-1}\beta_3 &= 0 \\
\beta_{-2} + \beta_{-1} + \beta_0 + \beta_1 + \beta_2 + \beta_3 &= 0 \qquad (9.159) \\
(-2)\beta_{-2} + (-1)\beta_{-1} + 0\beta_0 + 1\beta_1 + 2\beta_2 + 3\beta_3 &= 0 \qquad (9.160) \\
(-2)^2\beta_{-2} + (-1)^2\beta_{-1} + 0^2\beta_0 + 1^2\beta_1 + 2^2\beta_2 + 3^2\beta_3 &= 0 \qquad (9.161)
\end{aligned}$$

Solving this set of simultaneous equations, yields the following wavelet and scaling numbers (that can be reversed in order):

$$\begin{aligned}
\beta_{-2} &= 0.0352262918857095 & \alpha_{-2} &= 0.332670552950083 \\
\beta_{-1} &= 0.0854412738820267 & \alpha_{-1} &= 0.806891509311092 \\
\beta_0 &= -0.135011020010255 & \alpha_0 &= 0.459877502118491 \\
\beta_1 &= -0.459877502118491 & \alpha_1 &= -0.135011020010255 \\
\beta_2 &= 0.806891509311092 & \alpha_2 &= -0.0854412738820267 \\
\beta_3 &= -0.332670552950083 & \alpha_3 &= 0.0352262918857095
\end{aligned}$$

The resulting scaling signals and wavelets have been illustrated in Figure 9.11.

As these scaling and wavelet numbers originate from the fact that quadratic signals yield zero fluctuation values, the performance will be excellent for near-quadratic signals.

[10]Remark: MATLAB labels the Coif-6 wavelet as `coif1`.
[11]Remark: MATLAB labels the Coef-18 wavelet as coef3.

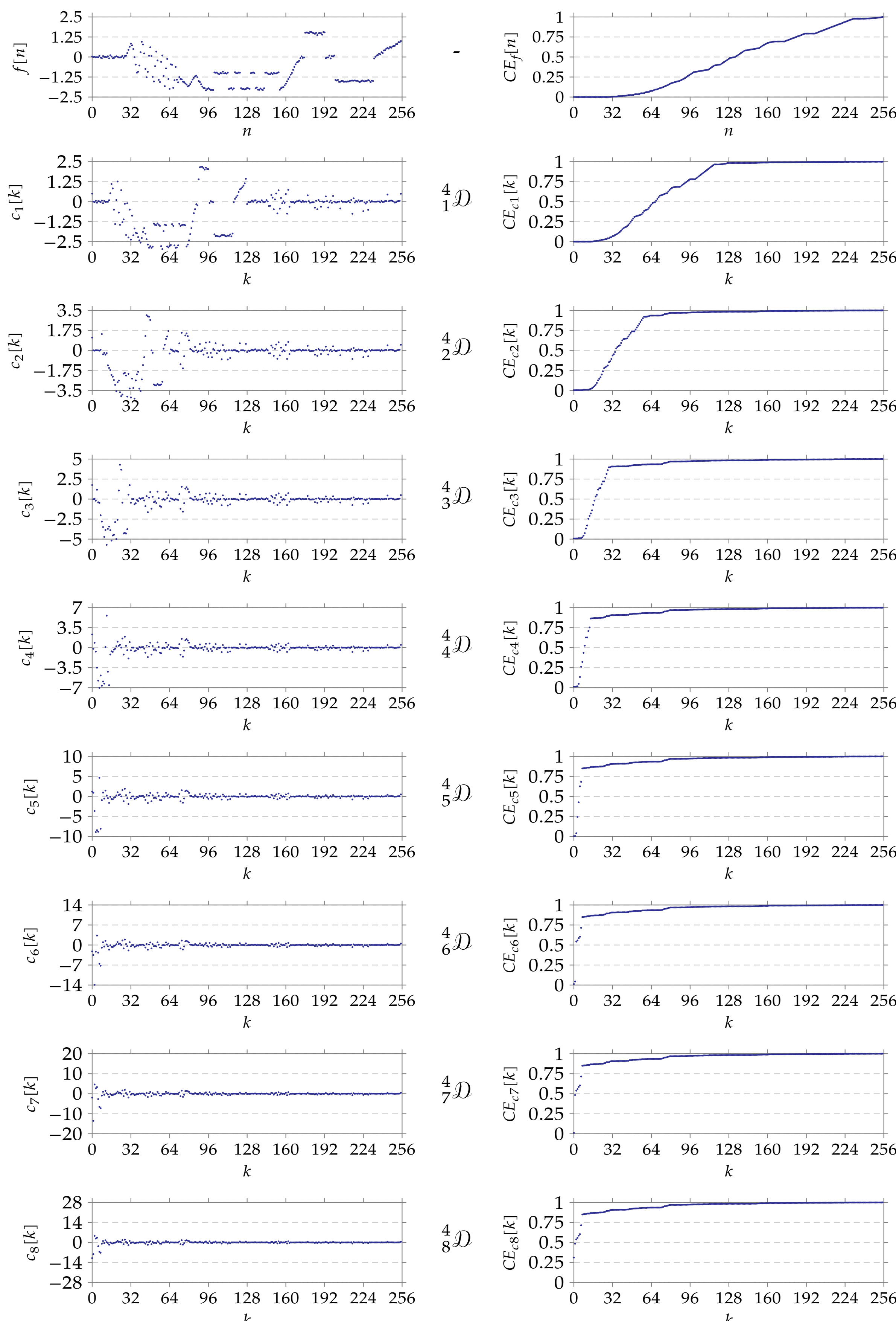


Figure 9.9: Example signal $f[n]$ with its Daub-4 transforms (${}^4_1\mathcal{D}$ to ${}^4_8\mathcal{D}$); the left column contains the (transformed) signal, the right column contains its cumulative energy distribution.

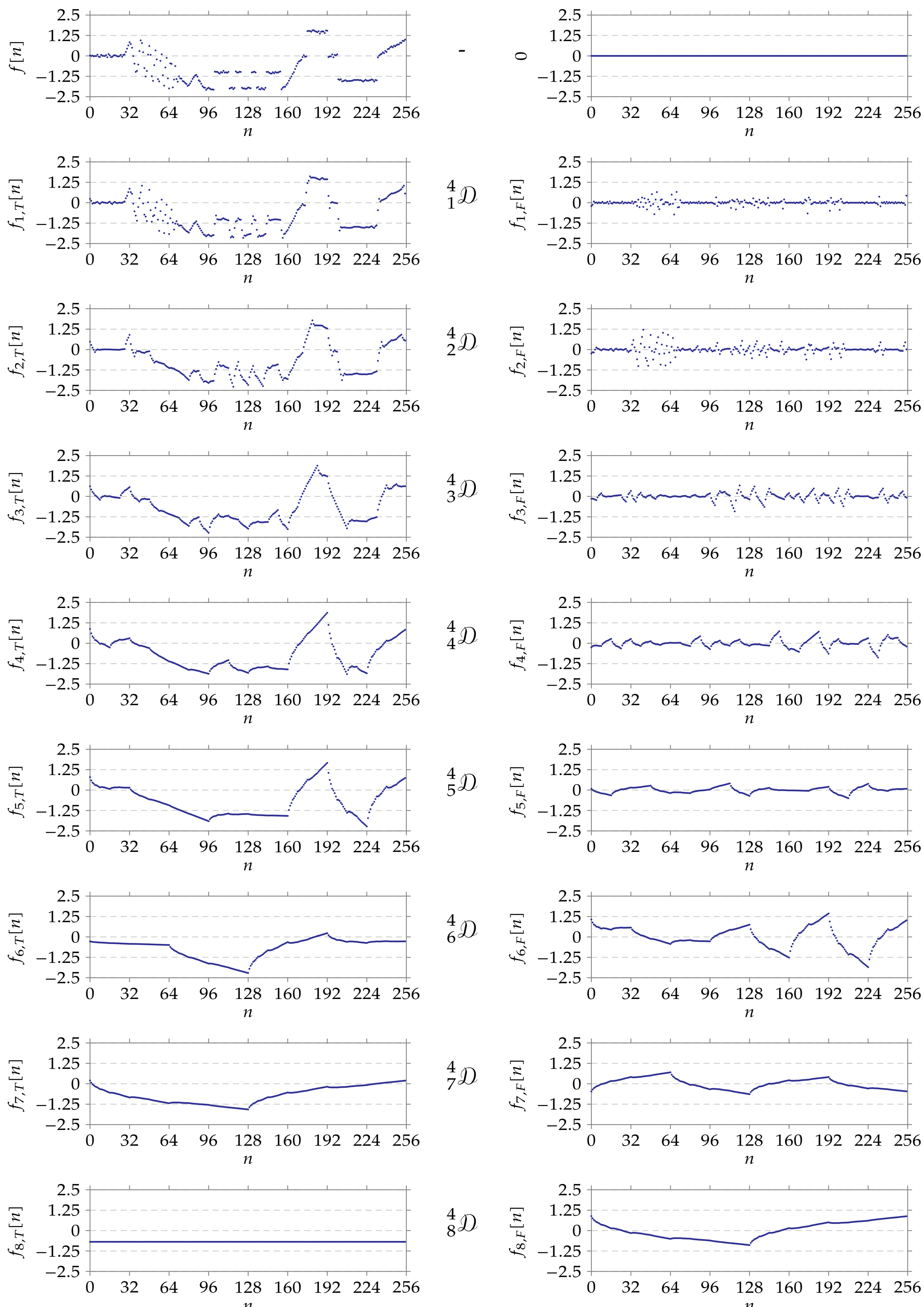


Figure 9.10: Example signal with its trend and fluctuation signals in the time domain corresponding to the Daub-4 transforms; the left column contains the trend signals in the time domain, the right column contains the fluctuation signals in the time domain.

	Energy fraction					
Level	Haar	Daub-4	Daub-6	Daub-16	Coif-6[10]	Coif-18[11]
0			100.00%			
1	95.98	98.29	98.18	98.86	97.77	98.11
2	93.39	93.45	94.73	94.30	93.79	94.34
3	89.90	90.61	90.33	90.76	89.86	90.46
4	83.19	86.73	86.99	88.36	86.87	87.91
5	66.02	84.92	69.69	68.52	71.30	69.27
6	46.64	55.68	53.50	51.90	54.67	50.63
7	32.38	48.38	49.45	35.09	48.16	46.33
8	31.05	31.05	31.05	31.05	31.05	31.05

Table 9.1: Energy fraction present in the trend subframe (or signal) of the example signal of page 239 for all the levels of various wavelet transforms.

	Energy fraction per sample					
Level	Haar	Daub-4	Daub-6	Daub-16	Coif-6	Coif-18
0			0.39%			
1	0.75	0.77	0.77	0.77	0.76	0.77
2	1.46	1.46	1.48	1.47	1.47	1.47
3	2.81	2.83	2.82	2.84	2.81	2.83
4	5.20	5.42	5.44	5.52	5.43	5.49
5	8.25	10.61	8.71	8.57	8.91	8.66
6	11.66	13.92	13.37	12.97	13.67	12.66
7	16.19	24.19	24.72	17.55	24.08	23.16
8	31.05	31.05	31.05	31.05	31.05	31.05

Table 9.2: Energy fraction *per sample* present in the trend subframe (or signal) of the example signal of page 239 for all the levels of various wavelet transforms.

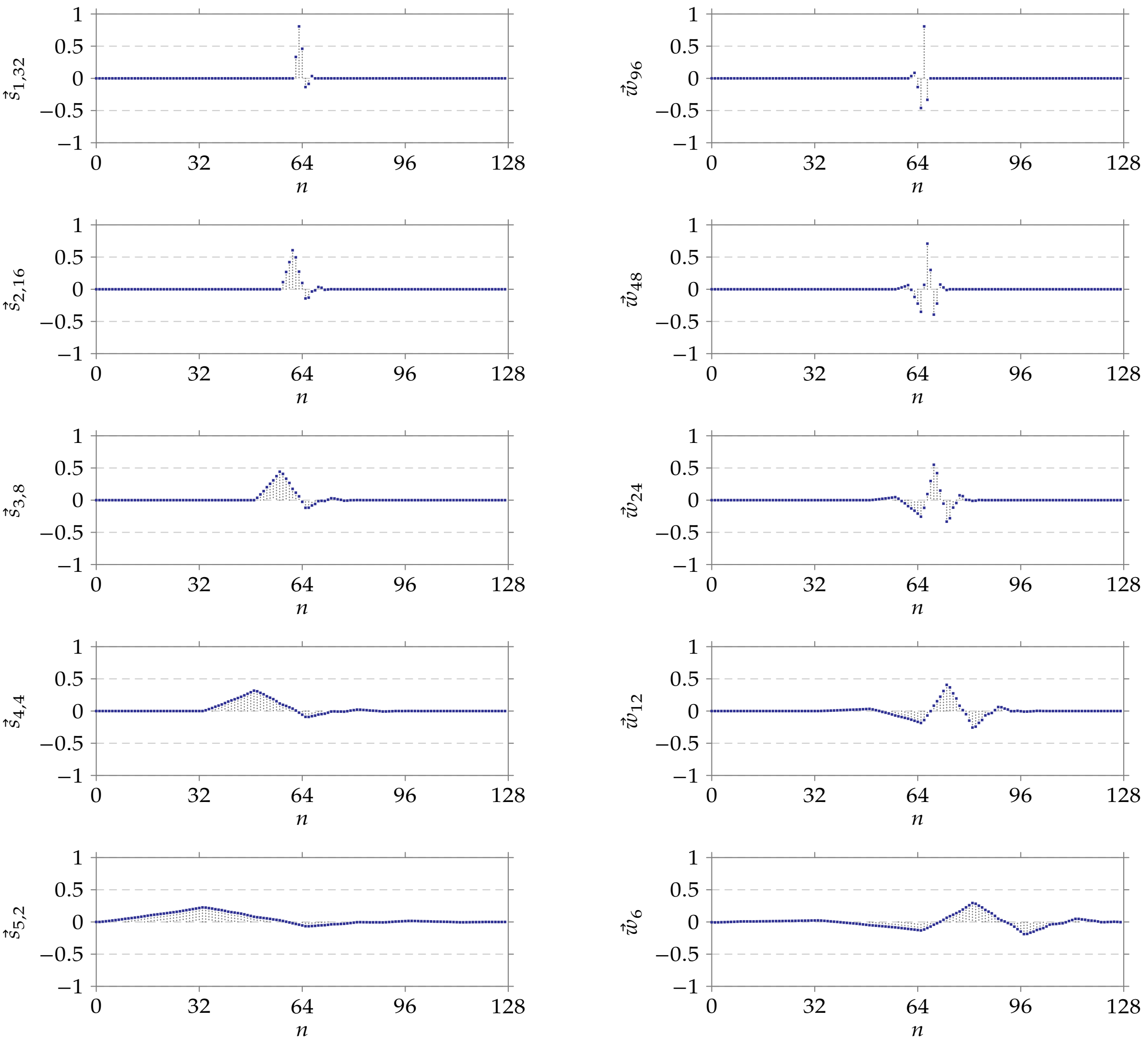


Figure 9.11: Illustration of the Daub-6 scaling signals (left column) and wavelets (right column)

We can extend this into a more generic theorem, that states that in general, for real-life signals, the fluctuations tend to be small.

Daub-6 small fluctuations theorem For a signal that originated from sampling a continuous-time signal with continuous derivatives up to third order, the Daub-6 transform generates small fluctuation signals. Assuming $M_k = N/2^k$:

$$c_k[M_k + i] \approx O\left(T_s^3\right), \qquad i = 0, 1, \ldots, M_k - 1$$

with T_s the sampling period.

Note: you can find more information on the Big-O notation $O(\bullet)$ in section B.3 on page 348.

The proof of this theorem is similar to the proof of the Daub-4 small fluctuations theorem and is left to the reader. Again, for higher transformation levels, the term $O\left(T_s^3\right)$ *will* become big.

A view of the result of applying the Daubechies-6 transform to our test signal can be found on page 260.

Daubechies-L wavelets (Daub-L) The idea of fluctuation zeroing can be continued to find Daublets with longer supports and increasing capability of finding higher-order trends. L-values of ten or more are not uncommon.

In fact, there is a common term for expressions of the following form:

$$\sum_i i^n \beta_i$$

These are called *moments*. For $n = 0$ it is a *zeroth order moment*, for $n = 1$ a *first order moment* and so on. When a moment becomes zero, we call it a *vanishing moment*.

Daubechies wavelets are based on creating as many vanishing wavelet moments as possible.

A view of the result of applying the Daubechies-16 transform to our test signal can be found on page 262.

9.5.3 Coifman wavelets

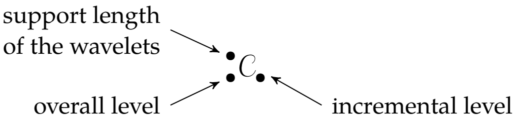
Daubechies at first focused on posing constraints on the wavelet numbers. Coifman suggested to her that posing constraints on the scaling numbers also makes sense. Elaborating on that idea, she developed (as a tribute to Coifman) the Coifman wavelets.

$$\begin{aligned}
\vec{s}_{1,0} &= \begin{bmatrix} \alpha_0, & \alpha_1, & \alpha_2, & \alpha_3, & 0, & 0, & \ldots, & 0, & 0, & 0, & 0, & \alpha_{-2}, & \alpha_{-1} \end{bmatrix} \\
\vec{s}_{1,1} &= \begin{bmatrix} \alpha_{-2}, & \alpha_{-1}, & \alpha_0, & \alpha_1, & \alpha_2, & \alpha_3, & \ldots, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix} \\
\vec{s}_{1,2} &= \begin{bmatrix} 0, & 0, & \alpha_{-2}, & \alpha_{-1}, & \alpha_0, & \alpha_1, & \ldots, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix} \\
&\vdots \\
\vec{s}_{1,M_1-2} &= \begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & \ldots, & \alpha_{-2}, & \alpha_{-1}, & \alpha_0, & \alpha_1, & \alpha_2, & \alpha_3 \end{bmatrix} \\
\vec{s}_{1,M_1-1} &= \begin{bmatrix} \alpha_2, & \alpha_3, & 0, & 0, & 0, & 0, & \ldots, & 0, & 0, & \alpha_{-2}, & \alpha_{-1}, & \alpha_0, & \alpha_1 \end{bmatrix}
\end{aligned}$$

and

$$\begin{aligned}
\vec{w}_{M_1} &= [\,\beta_0,\ \beta_1,\ \beta_2,\ \beta_3,\ 0,\ 0,\ \ldots,\ 0,\ 0,\ 0,\ 0,\ \beta_{-2},\ \beta_{-1}\,] \\
\vec{w}_{M_1+1} &= [\,\beta_{-2},\ \beta_{-1},\ \beta_0,\ \beta_1,\ \beta_2,\ \beta_3,\ \ldots,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0\,] \\
\vec{w}_{M_1+2} &= [\,0,\ 0,\ \beta_{-2},\ \beta_{-1},\ \beta_0,\ \beta_1,\ \ldots,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0\,] \\
&\vdots \\
\vec{w}_{N-2} &= [\,0,\ 0,\ 0,\ 0,\ 0,\ 0,\ \ldots,\ \beta_{-2},\ \beta_{-1},\ \beta_0,\ \beta_1,\ \beta_2,\ \beta_3\,] \\
\vec{w}_{N-1} &= [\,\beta_2,\ \beta_3,\ 0,\ 0,\ 0,\ 0,\ \ldots,\ 0,\ 0,\ \beta_{-2},\ \beta_{-1},\ \beta_0,\ \beta_1\,]
\end{aligned}$$

Notation Again the Coifman wavelet transform is a multi-level transform for which we'll use a notation that is very similar to the Haar and the Daubechies notation.



Coifman-6 wavelets (Coif-6)

The improvement of Daub-6 over Daub-4, was that a quadratic signal would yield a zero fluctuation. This was accomplished by adding the extra free constraint:

$$(-2)^2\beta_{-2} + (-1)^2\beta_{-1} + 0^2\beta_0 + 1^2\beta_1 + 2^2\beta_2 + 3^2\beta_3 = 0$$

The Coif-6 wavelets are based on the Daub-6 constraints but instead of (9.161), they add the following constraints on the scaling numbers:

$$(-2)\alpha_{-2} + (-1)\alpha_{-1} + 0\alpha_0 + 1\alpha_1 + 2\alpha_2 + 3\alpha_3 = 0 \tag{9.163}$$

$$(-2)^2\alpha_{-2} + (-1)^2\alpha_{-1} + 0^2\alpha_0 + 1^2\alpha_1 + 2^2\alpha_2 + 3^2\alpha_3 = 0 \tag{9.164}$$

This means that the Coifman-6 wavelets will not detect a quadratic trend, but only a linear trend (and therefore perform much like the Daub-4 wavelets).

Solving the combination of the mandatory boundary conditions, in combination with (9.159), (9.160) and the 'Coifman' conditions (9.163) and (9.164) yields:

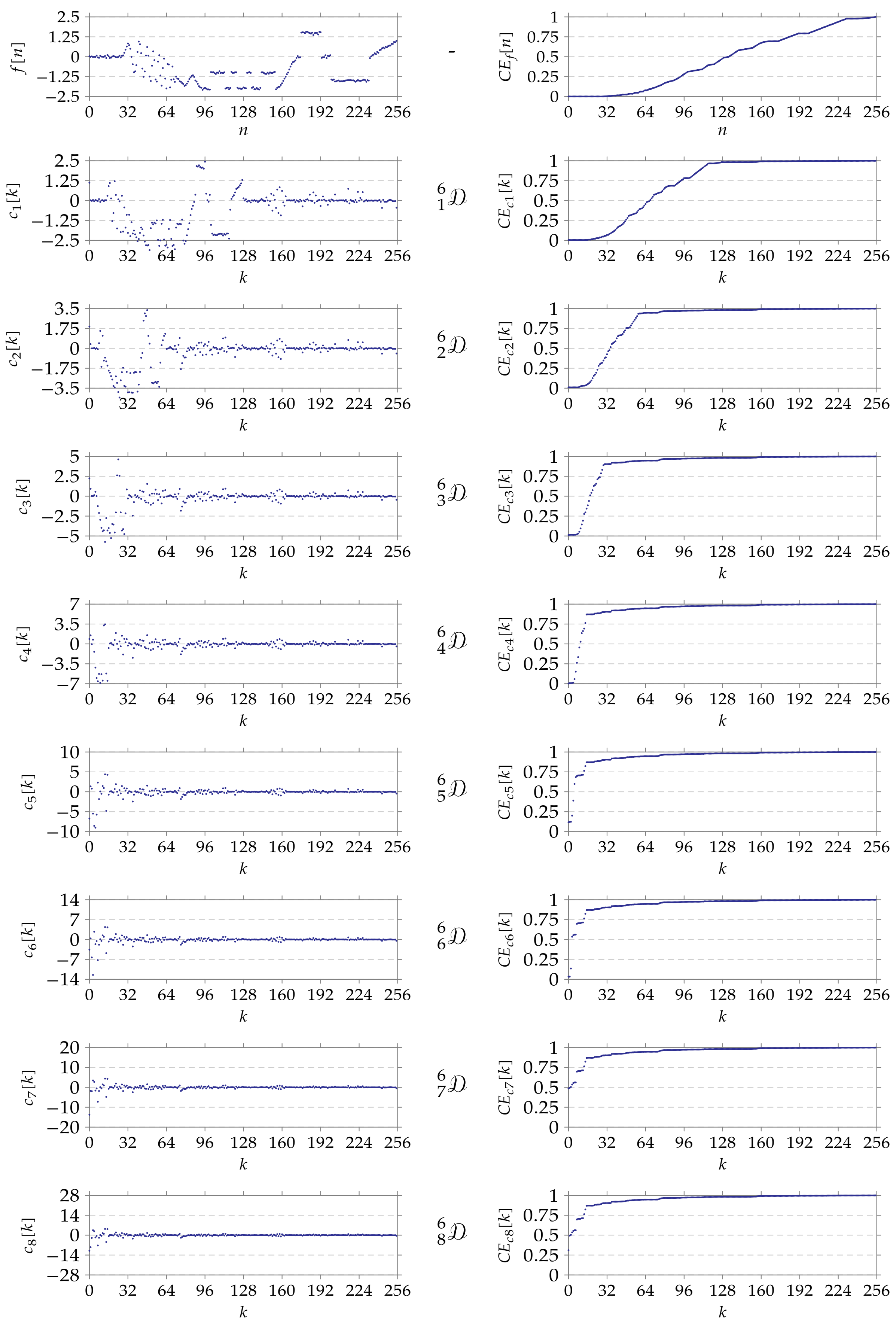


Figure 9.12: Example signal $f[n]$ with its Daub-6 transforms (${}^6_1\mathcal{D}$ to ${}^6_8\mathcal{D}$); the left column contains the (transformed) signal, the right column contains its cumulative energy distribution.

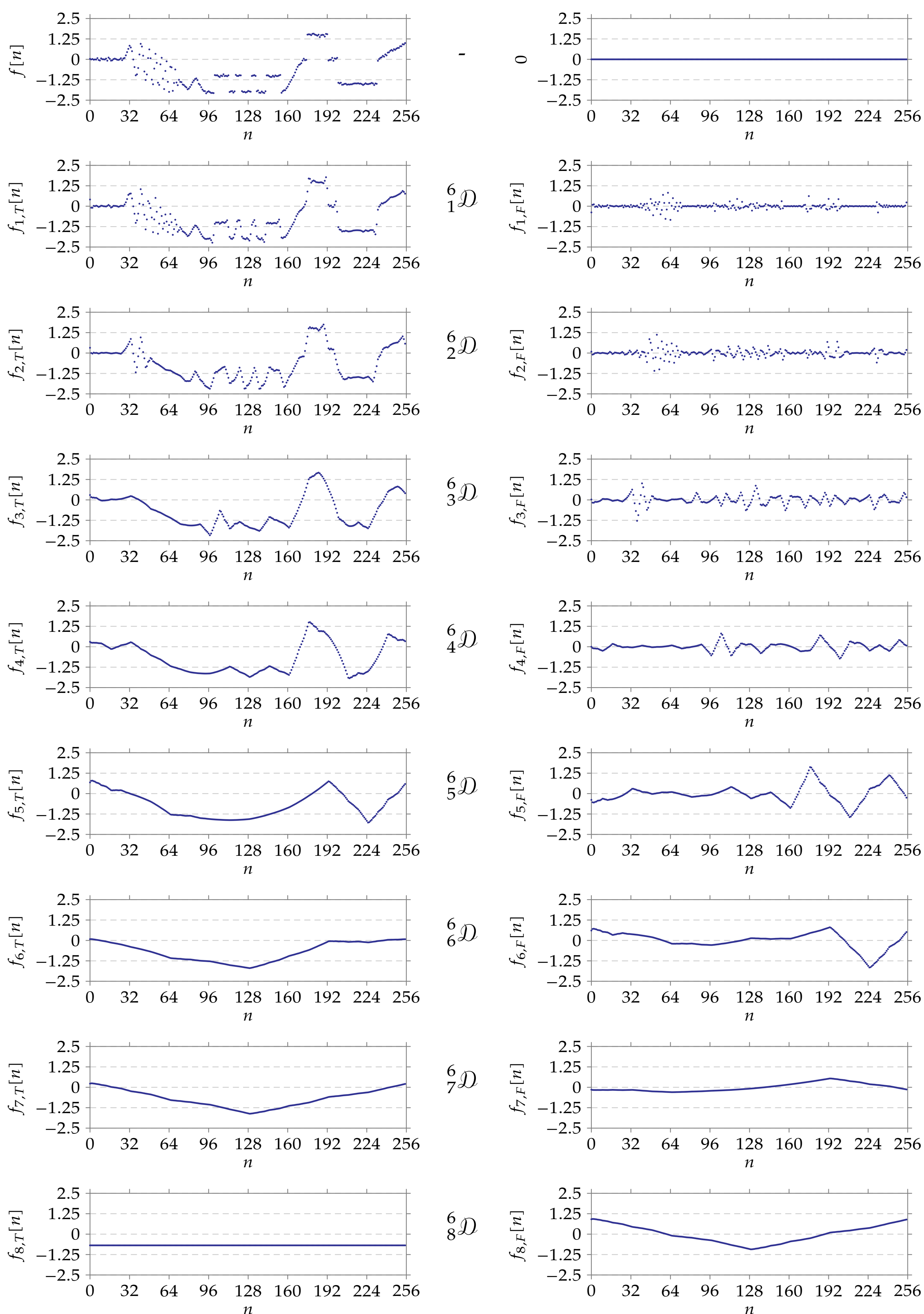


Figure 9.13: Example signal $f[n]$ with its trend and fluctuation signals in the time domain corresponding to the Daub-6 transforms; the left column contains the trend signals in the time domain, the right column contains the fluctuation signals in the time domain.

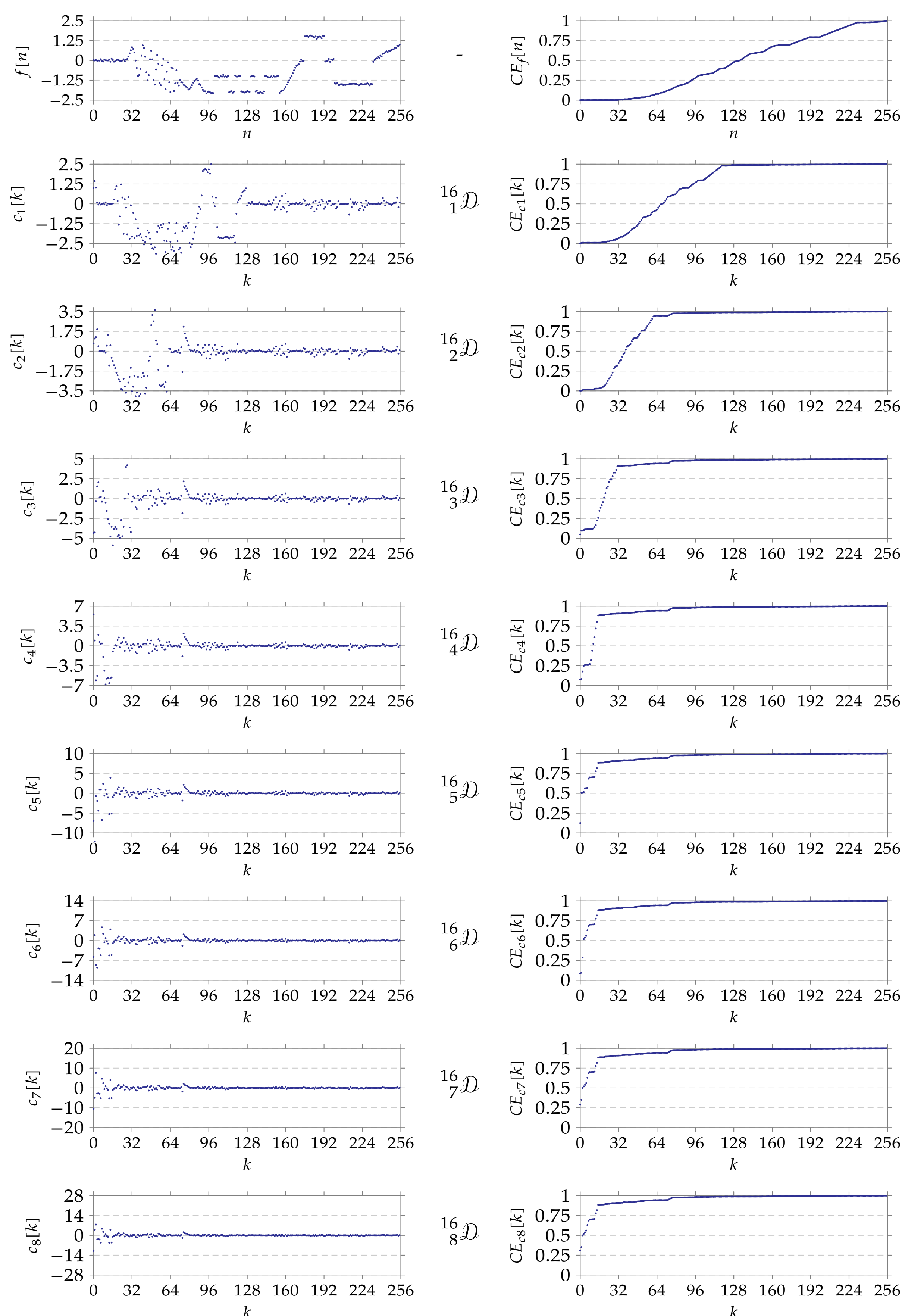


Figure 9.14: Example signal $f[n]$ with its Daub-16 transforms (${}^{16}_{1}\mathcal{D}$ to ${}^{16}_{8}\mathcal{D}$); the left column contains the (transformed) signal, the right column contains its cumulative energy distribution.

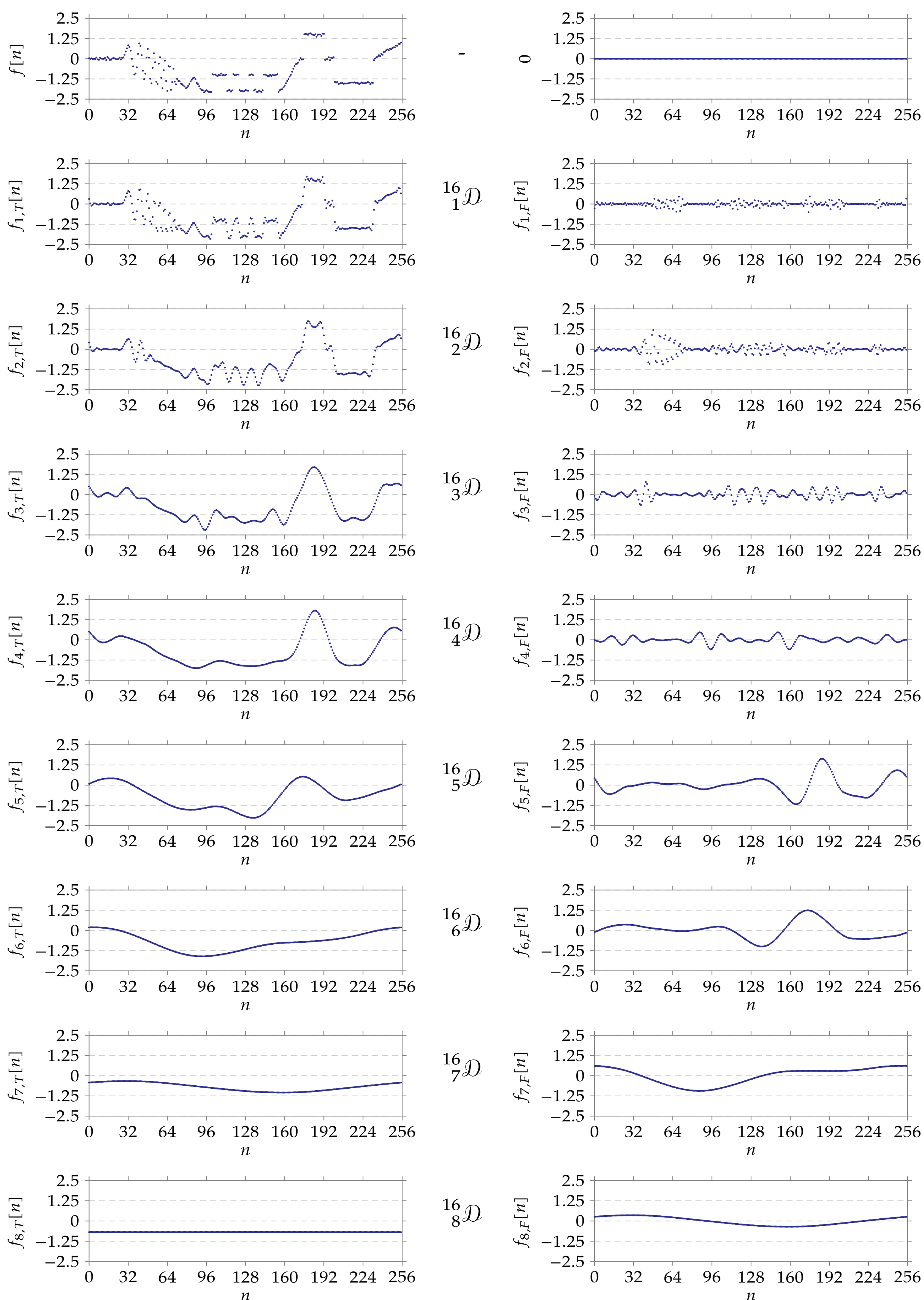


Figure 9.15: Example signal $f[n]$ with its trend and fluctuation signals in the time domain corresponding to the Daub-16 transforms; the left column contains the trend signals in the time domain, the right column contains the fluctuation signals in the time domain.

$$\beta_{-2} = -\frac{-3+\sqrt{7}}{16\sqrt{2}} \qquad \alpha_{-2} = \frac{1-\sqrt{7}}{16\sqrt{2}}$$
$$\beta_{-1} = \frac{1-\sqrt{7}}{16\sqrt{2}} \qquad \alpha_{-1} = \frac{5+\sqrt{7}}{16\sqrt{2}}$$
$$\beta_{0} = -\frac{14-2\sqrt{7}}{16\sqrt{2}} \qquad \alpha_{0} = \frac{14+2\sqrt{7}}{16\sqrt{2}}$$
$$\beta_{1} = \frac{14+2\sqrt{7}}{16\sqrt{2}} \qquad \alpha_{1} = \frac{14-2\sqrt{7}}{16\sqrt{2}}$$
$$\beta_{2} = -\frac{5+\sqrt{7}}{16\sqrt{2}} \qquad \alpha_{2} = \frac{1-\sqrt{7}}{16\sqrt{2}}$$
$$\beta_{3} = \frac{1-\sqrt{7}}{16\sqrt{2}} \qquad \alpha_{3} = \frac{-3+\sqrt{7}}{16\sqrt{2}}$$

The resulting scaling signals and wavelets have been illustrated in Figure 9.16.

One can also observe that:

$$\alpha_{-2} + \alpha_{-1} + \alpha_0 + \alpha_1 + \alpha_2 + \alpha_3 = \sqrt{2} \tag{9.165}$$

The benefit of all this is that the Coif-6 wavelets are better to create a trend signal at level n that (except for a scaling factor) is as close as can be to the trend signal of level $n-1$ in case the signal is a sampled version of a real-life continuous-times signal. This makes the trend signals more suited for amplitude measurements.

The proof of this claim is actually not that complicated. Let's state the exact theorem and prove it.

Coif-6 accurate trend theorem For a signal that originated from sampling a continuous-time signal with continuous derivatives up to third order, the Coif-6 transform generates trend signals that are as close to the original data points as $O\left(T_s^3\right)$.

$$c_n[i] \approx \sqrt{2}c_{n-1}[2i] + O\left(T_s^3\right)$$

with T_s the sampling period.

Note: you can find more information on the Big-O notation $O(\bullet)$ in section B.3 on page 348.

Proof:
Without loss of generality, let's prove the theorem for $n = 1$. Consider the continuous time signal $f(t)$ that has been sampled with a sample period T_s, yielding a discrete-time signal $f[n]$:

$$f[n] = f(nT_s)$$

Given the fact that $f(t)$ has continuous first, second and third-order derivatives, we can use Taylor's theorem (see section B.2 on page 348) and find a $c \in [0, \Delta t]$ such that:

$$f(t_0 + \Delta t) = f(t_0) + f'(t_0)(\Delta t) + \frac{f''(t_0)}{2}(\Delta t)^2 + \frac{f'''(t_0 + c)}{3!}(\Delta t)^3$$

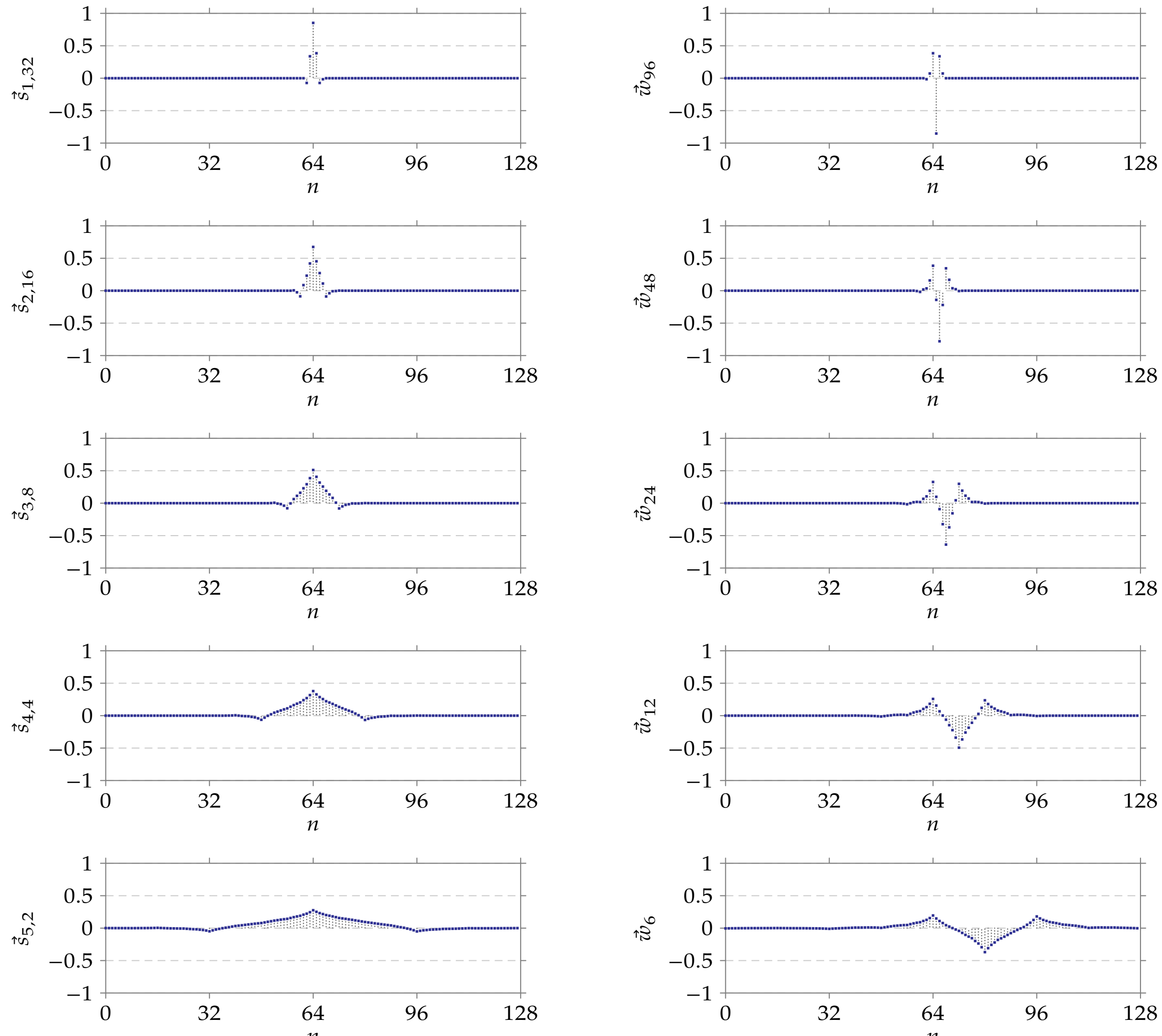


Figure 9.16: Illustration of the Coif-6 scaling signals (left column) and wavelets (right column)

Using the Big-O notation, we abbreviate this as:

$$f(t_0 + \Delta t) = f(t_0) + f'(t_0)(\Delta t) + \frac{1}{2} f''(t_0)(\Delta t)^2 + O\left((\Delta t)^3\right) \tag{9.166}$$

Now, let's project this function on an arbitrary scaling vector. This means calculating the scalar product:

$$c_1[i] = \left\langle \vec{f}, \vec{s}_{1,i} \right\rangle$$

If we explicitate the scalar product in terms of the scaling numbers, we obtain:

$$c_1[i] = \alpha_{-2} f[2i-2] + \alpha_{-1} f[2i-1] + \alpha_0 f[2i] + \alpha_1 f[2i+1] + \alpha_2 f[2i+2] + \alpha_3 f[2i+3]$$

However, given (9.166) and assuming that the discrete time point $2i$ corresponds $t = t_0$ we know that:

$$\begin{aligned}
\alpha_{-2} f[2i-2] &= \alpha_{-2}\left(f(t_0) + f'(t_0)(-2T_s) + \frac{1}{2} f''(t_0)(-2T_s)^2 + O\left(T_s^3\right)\right) \\
\alpha_{-1} f[2i-1] &= \alpha_{-1}\left(f(t_0) + f'(t_0)(-T_s) + \frac{1}{2} f''(t_0)(-T_s)^2 + O\left(T_s^3\right)\right) \\
\alpha_0 f[2i] &= \alpha_0\left(f(t_0)\right) \\
\alpha_1 f[2i+1] &= \alpha_1\left(f(t_0) + f'(t_0)(+T_s) + \frac{1}{2} f''(t_0)(+T_s)^2 + O\left(T_s^3\right)\right) \\
\alpha_2 f[2i+2] &= \alpha_2\left(f(t_0) + f'(t_0)(+2T_s) + \frac{1}{2} f''(t_0)(+2T_s)^2 + O\left(T_s^3\right)\right) \\
\alpha_3 f[2i+3] &= \alpha_3\left(f(t_0) + f'(t_0)(+3T_s) + \frac{1}{2} f''(t_0)(+3T_s)^2 + O\left(T_s^3\right)\right)
\end{aligned}$$

Summing the left-hand and right-hand sides and rearranging the terms yields:

$$\begin{aligned}
c_1[i] = &(\alpha_{-2} + \alpha_{-1} + \alpha_0 + \alpha_1 + \alpha_2 + \alpha_3) f(t_0) \\
&+ f'(t_0) T_s \left((-2)\alpha_{-2} + (-1)\alpha_{-1} + 0\alpha_0 + 1\alpha_1 + 2\alpha_2 + 3\alpha_3\right) \\
&+ \frac{f''(t_0)}{2} T_s^2 \left((-2)^2\alpha_{-2} + (-1)^2\alpha_{-1} + 0^2\alpha_0 + 1^2\alpha_1 + 2^2\alpha_2 + 3^2\alpha_3\right) \\
&+ O\left(T_s^3\right)
\end{aligned}$$

However, in view of (9.163), (9.164) and (9.165), this can be simplified to:

$$c_1[i] = \sqrt{2} f(t_0) + O\left(T_s^3\right)$$

This proves the theorem. ■

A view of the result of applying the Coif-6 transform to our test signal can be found on page 268.

Coifman-L wavelets (Coif-L) The idea of the mix of fluctuation zeroing and trend trueness can be continued to find Coiflets with longer supports and increasing capability of finding higher-order trends. L-values of ten or more are not uncommon.

In fact, again the concept of moments is a relevant one. Coifman wavelets try to make balanced amounts of vanishing wavelet moments and of vanishing scaling signal moments.

A view of the result of applying the Coif-18 transform to our test signal can be found on page 270.

9.5.4 Biorthonormal wavelets

The biorthonormal wavelet transform is a generalization of the orthonormal wavelet transform. It is often used in image processing for denoising and also for lossless and lossy image compression.

General principle Take your time to reread section 9.3.2 on page 225. One of its conclusions was that we are able to write a wavelet transform as a matrix product:

$$\vec{c}_1 = H_1\vec{f}$$

We just had to arrange the scaling signals and wavelets as rows in the matrix H_1. Equation (9.8) was stated for the Haar transform, but in general holds for any orthogonal wavelet transform, even for any arbitrary level.

On the same route, we found out that the inverse wavelet transform can also be written as a matrix product:

$$\vec{f} = H_1^{-1}\vec{c}_1$$

The essence of a reversible vector transformation is that the mapping realized by applying a forward transform first and then an inverse transform results in the same vector:

$$\vec{f} \xrightarrow{\mathcal{H}_1} \vec{c}_1 \xrightarrow{\mathcal{H}_1^{-1}} \vec{f}$$

or using operator notation:

$$\vec{f} = \mathcal{H}_1^{-1}\left(\mathcal{H}_1(\vec{f})\right)$$

If we write this using the matrix notation, it becomes:

$$\vec{f} = H^{-1}H\vec{f}$$

and because of the fact that H is row-filled with orthonormal base vectors, we also have:

$$H^{-1} = H^T$$

because of

$$H^TH = I = HH^T$$

The main conclusion we want to draw here, is that for an orthonormal wavelet transform the wavelets (and scaling signals) are localized on the rows of H, and on the columns of H^{-1} (†).

Now, what if we would loosen the tight relationship $H^{-1} = H^T$, and use separate forward and inverse transformation matrices?

Denoting the forward transformation matrix by A (instead of H) and the inverse transformation matrix by S (instead of H^T), the only constraint we'd impose is:

$$SA = I = AS \tag{9.173}$$

Two matrices obeying the equations above are called inverse matrices.

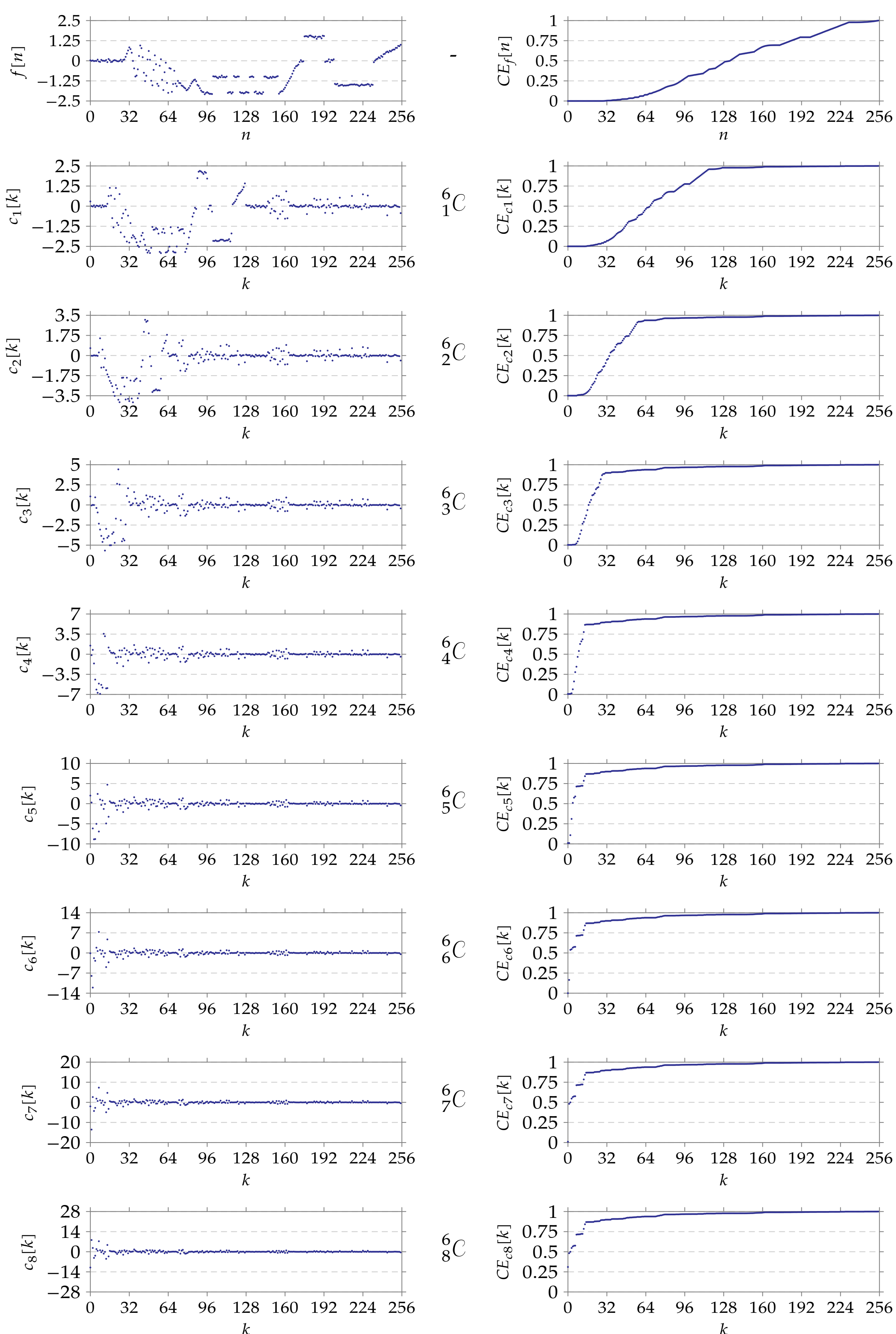


Figure 9.17: Example signal $f[n]$ with its Coif-6 transforms (${}^{6}_{1}C$ to ${}^{6}_{8}C$); the left column contains the (transformed) signal, the right column contains its cumulative energy distribution.

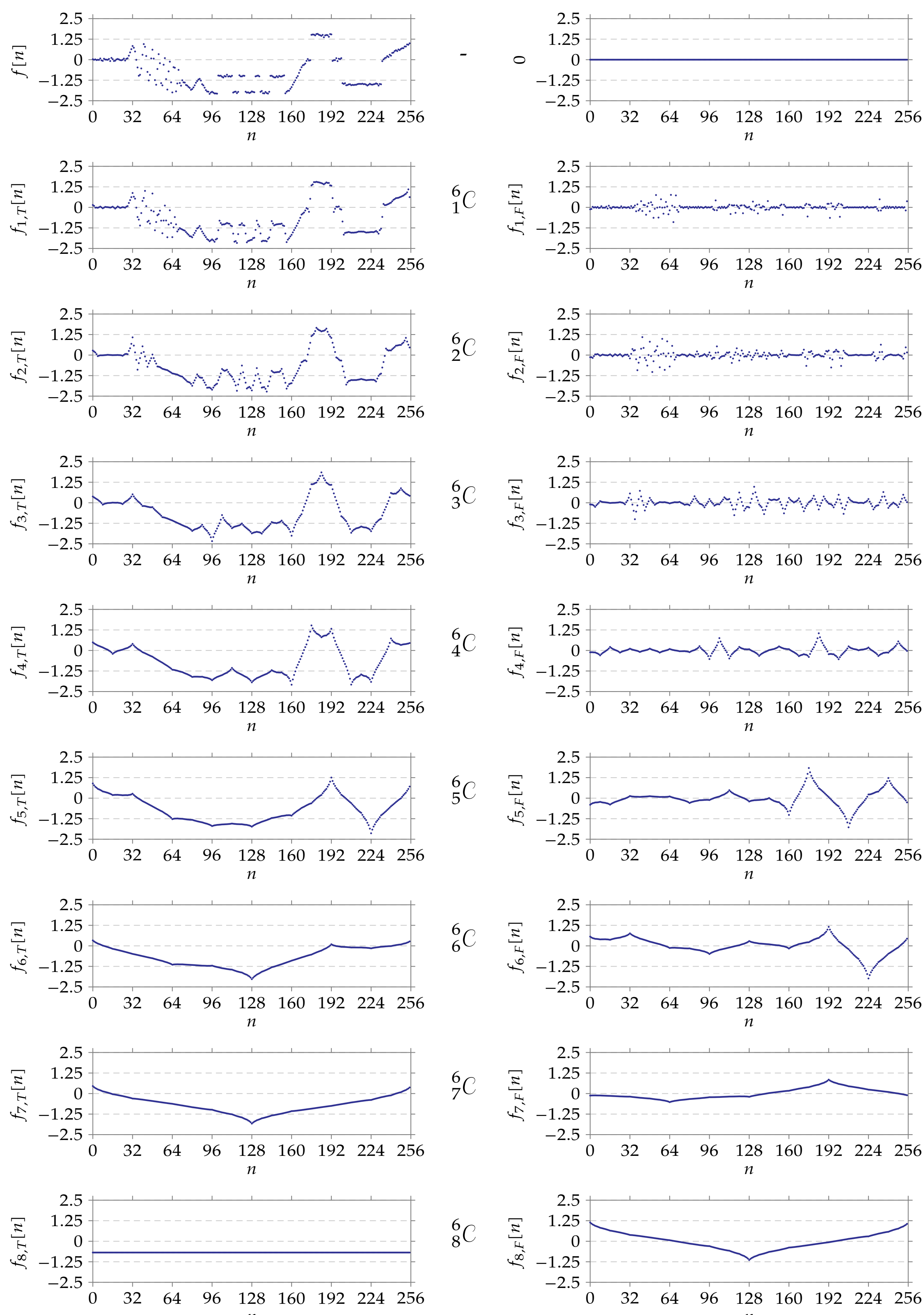


Figure 9.18: Example signal $f[n]$ with its trend and fluctuation signals in the time domain corresponding to the Coif-6 transforms; the left column contains the trend signals in the time domain, the right column contains the fluctuation signals in the time domain.

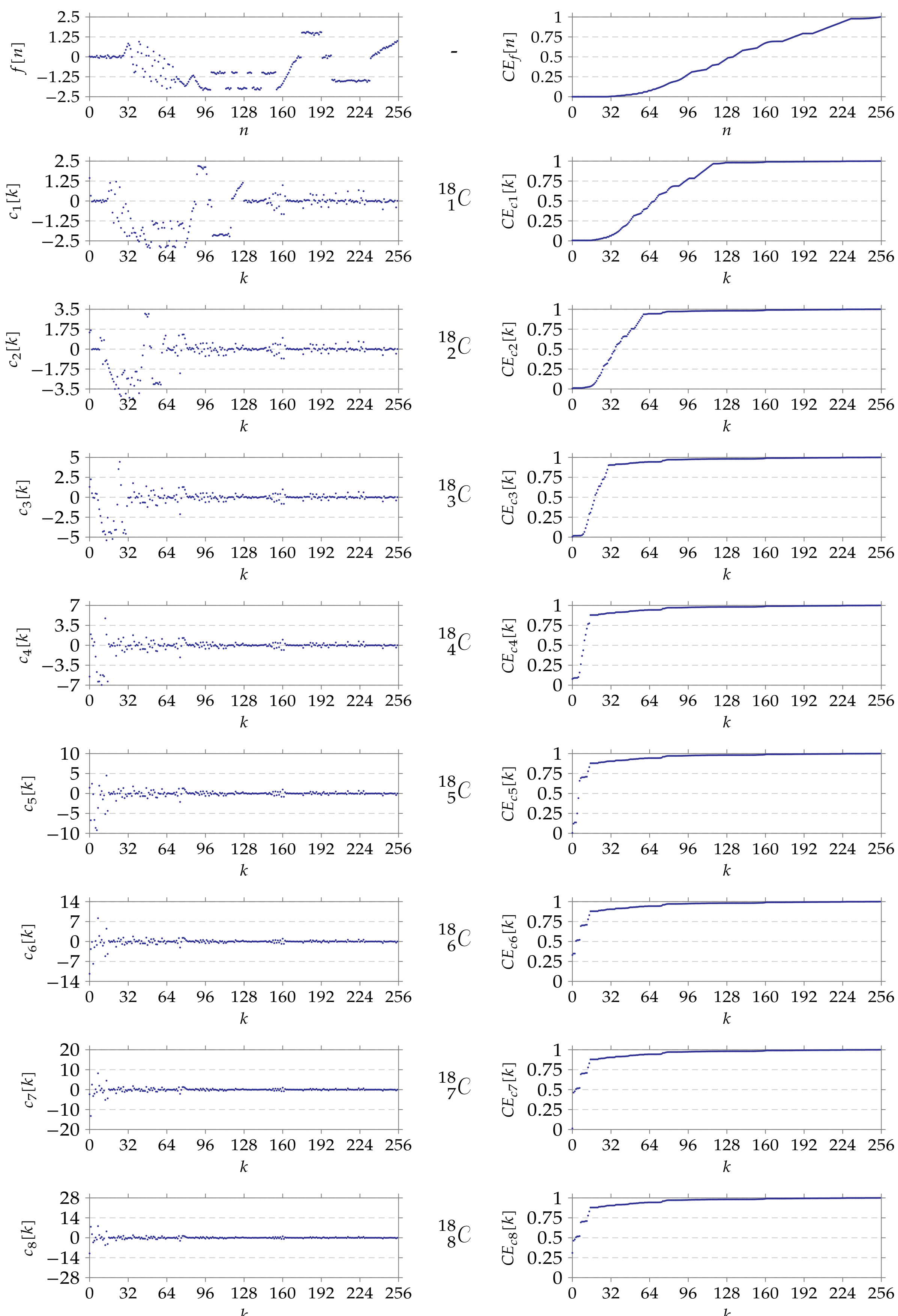


Figure 9.19: Example signal $f[n]$ with its Coif-18 transforms ($^{18}_{1}C$ to $^{18}_{8}C$); the left column contains the (transformed) signal, the right column contains its cumulative energy distribution.

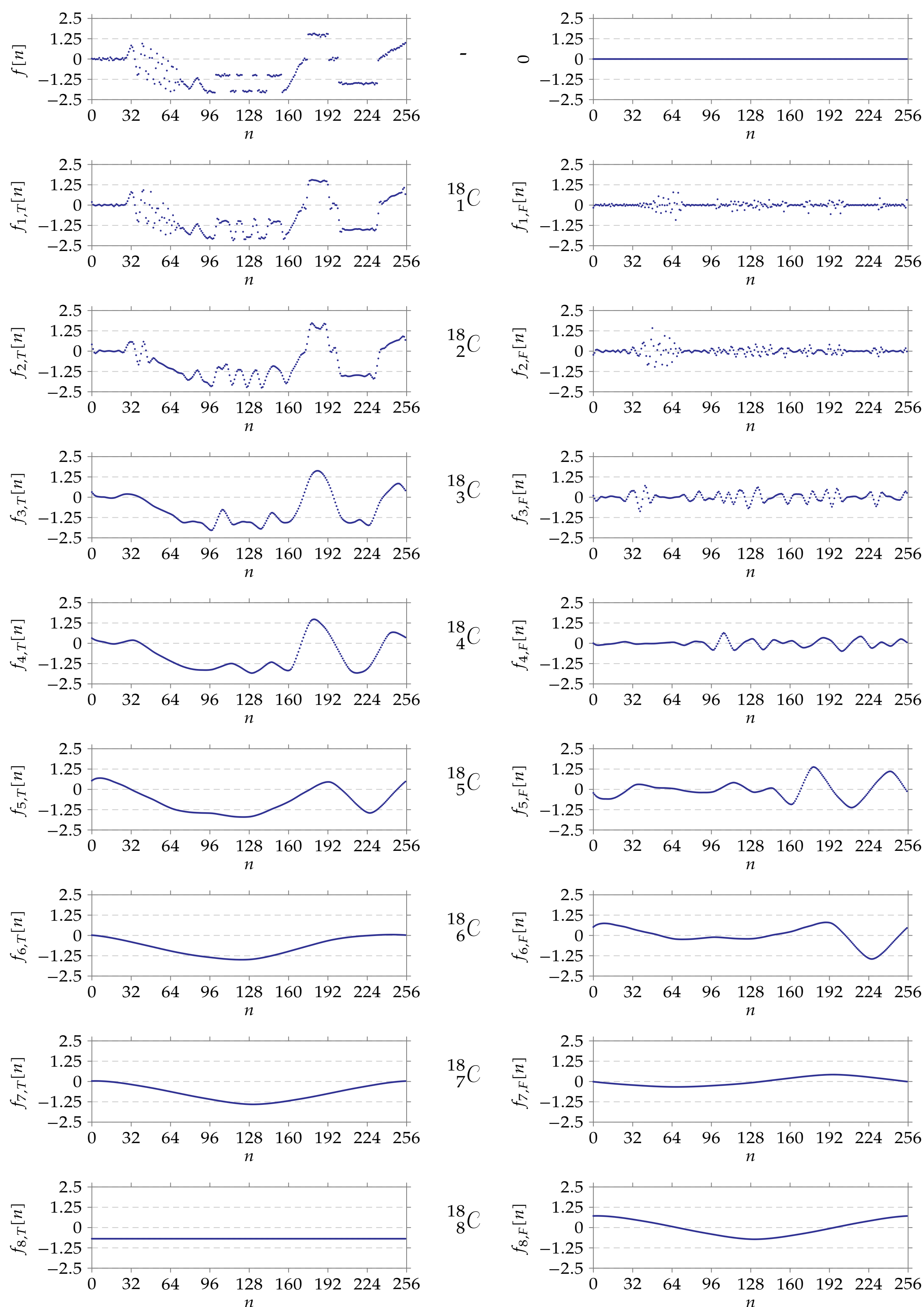


Figure 9.20: Example signal $f[n]$ with its trend and fluctuation signals in the time domain corresponding to the Coif-18 transforms; the left column contains the trend signals in the time domain, the right column contains the fluctuation signals in the time domain.

Now, let's call for reasons of analogy with (†) the rows of A the *analysis wavelets*[12] and the columns of S the *synthesis wavelets*. Given (9.5.4), the analysis wavelets are said to be biorthonormal with the synthesis wavelets (and vice versa) if they obey (9.5.4). In short, they are *biorthonormal*.

The only remaining problem is to find a set of appropriate wavelets to fill the rows of A and another set of wavelets to fill the columns of B, such that (9.5.4) is fulfilled.

Of course the transform obtained in this way is no longer orthogonal, no longer normalized and therefore also no longer energy preserving.

Those are drawbacks, but in exchange additional benefits arise, like the possibility to create symmetrical scaling signals and wavelets, and an even better discrimination between trend and fluctuation parts (offering higher degrees of compaction).

Finding appropriate biorthonormal bases The orthogonal wavelet transforms we have treated so far all obeyed a few standard rules:

- The supports of scaling vectors and wavelets were of equal length;
- They are generated using scaling and wavelet numbers α_i and β_j, that we chose to obey the following relationship:

  $$\alpha_i = (-1)^i \beta_{N-1-i}$$

  with N the wavelet's support length.

Biorthonormal base-pairs do not need to obey these requirements. They are still based on finding wavelet numbers for the analysis wavelets and synthesis wavelets attempting to create as many vanishing moments as possible (see section 9.5.2 on page 258 for an explanation of the concept of vanishing moments). Given the wavelet numbers, the base vectors can be found by a shifting of two time points for every consecutive wavelet or scaling signal.

We will *not* go through the elaborate mathematics of finding biorthonormal bases. Instead, we will just list two common ones:

- the Cohen-Daubechies-Feauveau 5/3 system
- the Cohen-Daubechies-Feauveau 9/7 system

The first number indicates the support length of the analysis scaling signals. The second number indicates the support length of the analysis wavelets. For the synthesis support lengths it is the other way around.

The *analysis scaling signals and wavelets* are defined by:

[12] we use the term wavelets here as a pars pro toto, i.e. to designate scaling signals and wavelets.

$$
\begin{aligned}
\vec{s}_{A1,0} &= [\, \alpha_0, \; \alpha_1, \; \alpha_2, \; 0, \; 0, \; 0, \; \dots, \; 0, \; 0, \; 0, \; 0, \; \alpha_{-2}, \; \alpha_{-1}] \\
\vec{s}_{A1,1} &= [\, \alpha_{-2}, \; \alpha_{-1}, \; \alpha_0, \; \alpha_1, \; \alpha_2, \; 0, \; \dots, \; 0, \; 0, \; 0, \; 0, \; 0, \; 0] \\
\vec{s}_{A1,2} &= [\, 0, \; 0, \; \alpha_{-2}, \; \alpha_{-1}, \; \alpha_0, \; \alpha_1, \; \dots, \; 0, \; 0, \; 0, \; 0, \; 0, \; 0] \\
&\vdots \\
\vec{s}_{A1,M_1-2} &= [\, 0, \; 0, \; 0, \; 0, \; 0, \; 0, \; \dots, \; \alpha_{-2}, \; \alpha_{-1}, \; \alpha_0, \; \alpha_1, \; \alpha_2, \; 0] \\
\vec{s}_{A1,M_1-1} &= [\, \alpha_2, \; 0, \; 0, \; 0, \; 0, \; 0, \; \dots, \; 0, \; 0, \; \alpha_{-2}, \; \alpha_{-1}, \; \alpha_0, \; \alpha_1]
\end{aligned}
$$

and

$$
\begin{aligned}
\vec{w}_{M_1} &= [\, \beta_0, \; \beta_1, \; \beta_2, \; 0, \; 0, \; 0, \; \dots, \; 0, \; 0, \; 0, \; 0, \; 0, \; 0] \\
\vec{w}_{AM_1+1} &= [\, 0, \; 0, \; \beta_0, \; \beta_1, \; \beta_2, \; 0, \; \dots, \; 0, \; 0, \; 0, \; 0, \; 0, \; 0] \\
\vec{w}_{AM_1+2} &= [\, 0, \; 0, \; 0, \; 0, \; \beta_0, \; \beta_1, \; \dots, \; 0, \; 0, \; 0, \; 0, \; 0, \; 0] \\
&\vdots \\
\vec{w}_{AN-2} &= [\, 0, \; 0, \; 0, \; 0, \; 0, \; 0, \; \dots, \; 0, \; 0, \; \beta_0, \; \beta_1, \; \beta_2, \; 0] \\
\vec{w}_{AN-1} &= [\, \beta_2, \; 0, \; 0, \; 0, \; 0, \; 0, \; \dots, \; 0, \; 0, \; 0, \; 0, \; \beta_0, \; \beta_1]
\end{aligned}
$$

The analysis scaling and wavelet numbers are:

$$
\begin{aligned}
\alpha_{-2} &= -\sqrt{2}\frac{1}{8} \\
\alpha_{-1} &= \sqrt{2}\frac{1}{4} & \beta_0 &= \sqrt{2}\frac{1}{4} \\
\alpha_0 &= \sqrt{2}\frac{3}{4} & \beta_1 &= -\sqrt{2}\frac{1}{2} \\
\alpha_1 &= \sqrt{2}\frac{1}{4} & \beta_2 &= \sqrt{2}\frac{1}{4} \\
\alpha_2 &= -\sqrt{2}\frac{1}{8}
\end{aligned}
$$

Note that the zeroth-order moment of the analysis scaling numbers equals $\sqrt{2}$, the first-order moment equals 0. Also note that the zero-th order and first-order moment of the wavelet numbers equal 0.

The *synthesis scaling signals and wavelets* are most easily obtained by

1. composing A with $\vec{s}_{A_i}$ and $\vec{w}_{A_i}$ on its rows
2. calculating its inverse $S = A^{-1}$
3. collecting them from the columns of S.

This results in:

$$\begin{aligned}
\vec{s}_{S1,0} &= \begin{bmatrix} \gamma_0, & \gamma_1, & 0, & 0, & 0, & 0, & \dots, & 0, & 0, & 0, & 0, & 0, & \gamma_{-1} \end{bmatrix} \\
\vec{s}_{S1,1} &= \begin{bmatrix} 0, & \gamma_{-1}, & \gamma_0, & \gamma_1, & 0, & 0, & \dots, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix} \\
\vec{s}_{S1,2} &= \begin{bmatrix} 0, & 0, & 0, & \gamma_{-1}, & \gamma_0, & \gamma_1, & \dots, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix} \\
&\vdots \\
\vec{s}_{S1,M_1-2} &= \begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & \dots, & 0, & \gamma_{-1}, & \gamma_0, & \gamma_1, & 0, & 0 \end{bmatrix} \\
\vec{s}_{S1,M_1-1} &= \begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & \dots, & 0, & 0, & 0, & \gamma_{-1}, & \gamma_0, & \gamma_1 \end{bmatrix}
\end{aligned}$$

and

$$\begin{aligned}
\vec{w}_{M_1} &= \begin{bmatrix} \epsilon_0, & \epsilon_1, & \epsilon_2, & \epsilon_3, & 0, & 0, & \dots, & 0, & 0, & 0, & 0, & 0, & \epsilon_{-1} \end{bmatrix} \\
\vec{w}_{SM_1+1} &= \begin{bmatrix} 0, & \epsilon_{-1}, & \epsilon_0, & \epsilon_1, & \epsilon_2, & \epsilon_3, & \dots, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix} \\
\vec{w}_{SM_1+2} &= \begin{bmatrix} 0, & 0, & 0, & \epsilon_{-1}, & \epsilon_0, & \epsilon_1, & \dots, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix} \\
&\vdots \\
\vec{w}_{SN-2} &= \begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & \dots, & 0, & \epsilon_{-1}, & \epsilon_0, & \epsilon_1, & \epsilon_2, & \epsilon_3 \end{bmatrix} \\
\vec{w}_{SN-1} &= \begin{bmatrix} \epsilon_2, & \epsilon_3, & 0, & 0, & 0, & 0, & \dots, & 0, & 0, & 0, & \epsilon_{-1}, & \epsilon_0, & \epsilon_1 \end{bmatrix}
\end{aligned}$$

with:

$$\begin{aligned}
\gamma_{-1} &= \frac{1}{\sqrt{2}}\frac{1}{2} \\
\gamma_0 &= \frac{1}{\sqrt{2}}1 \\
\gamma_1 &= \frac{1}{\sqrt{2}}\frac{1}{2}
\end{aligned}
\qquad
\begin{aligned}
\epsilon_{-1} &= \frac{1}{\sqrt{2}}\frac{1}{4} \\
\epsilon_0 &= \frac{1}{\sqrt{2}}\frac{1}{2} \\
\epsilon_1 &= -\frac{1}{\sqrt{2}}\frac{3}{2} \\
\epsilon_2 &= \frac{1}{\sqrt{2}}\frac{1}{2} \\
\epsilon_3 &= \frac{1}{\sqrt{2}}\frac{1}{4}
\end{aligned}$$

Remarks

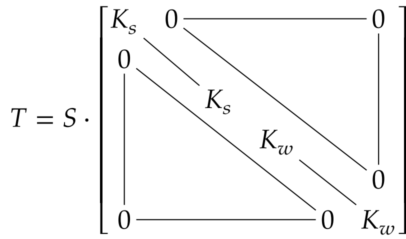
- A final remark on CDF-5/3: it can be reworked to an integer transform and as such yields very efficient implementation equations. Therefore, it is very usefull in lossless image compression.
- Consider the fact that the only requirement is that

  $$SA = I$$

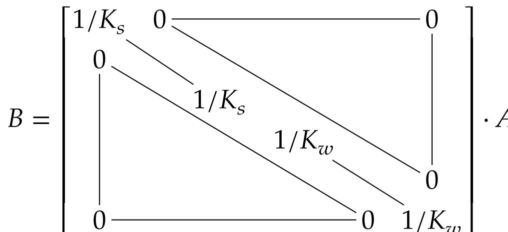
  Therefore, a logical consequence is that also:

  $$TB = I$$

  with



and



This means that one can scale the synthesis wavelets (by K_w) when one equally inversely scales the analysis wavelets (by $1/K_w$); likewise, one can scale the synthesis scaling signals (by K_s) when one equally inversely scales the analysis scaling signals (by $1/K_s$). This makes comparing literature quite cumbersome when it comes to wavelet and scaling numbers for biorthonormal transforms.

Cohen-Daubechies-Feauveau 9/7 (CDF 9/7)

We restrict ourselves to providing the analysis scaling signal and analysis wavelet numbers:

$$
\begin{aligned}
\alpha_{-4} &= 0.037828456 & & \\
\alpha_{-3} &= -0.023849465 & \beta_{-2} &= -0.064538887 \\
\alpha_{-2} &= -0.110624404 & \beta_{-1} &= 0.040689418 \\
\alpha_{-1} &= 0.377402856 & \beta_{0} &= 0.418092273 \\
\alpha_{0} &= 0.852698679 & \beta_{1} &= -0.788485616 \\
\alpha_{1} &= 0.377402856 & \beta_{2} &= 0.418092273 \\
\alpha_{2} &= -0.110624404 & \beta_{3} &= 0.040689418 \\
\alpha_{3} &= -0.023849465 & \beta_{4} &= -0.064538883 \\
\alpha_{4} &= 0.037828456 & &
\end{aligned}
$$

Note that the zeroth-order moment of the analysis scaling numbers equals $\sqrt{2}$, the first-order moment equals 0. Also note that the zero-th order, first-order, second-order and third-order moment of the wavelet numbers equal 0.

The *synthesis scaling signals and wavelets* are most easily obtained by

1. composing A with $\vec{s}_{Ai}$ and $\vec{w}_{Ai}$ on its rows
2. calculating its inverse $S = A^{-1}$
3. collecting them from the columns of S.

This results in:

$$\begin{aligned}
\vec{s}_{S1,0} &= [\ \gamma_0,\ \gamma_1,\ \gamma_2,\ \gamma_3,\ 0,\ 0,\ \ldots,\ 0,\ 0,\ 0,\ \gamma_{-3},\ \gamma_{-2},\ \gamma_{-1}] \\
\vec{s}_{S1,1} &= [\gamma_{-2},\ \gamma_{-1},\ \gamma_0,\ \gamma_1,\ \gamma_2,\ \gamma_3,\ \ldots,\ 0,\ 0,\ 0,\ 0,\ 0,\ \gamma_{-3}] \\
\vec{s}_{S1,2} &= [\ 0,\ \gamma_{-3},\ \gamma_{-2},\ \gamma_{-1},\ \gamma_0,\ \gamma_1,\ \ldots,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0] \\
&\vdots \\
\vec{s}_{S1,M_1-2} &= [\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ \ldots,\ \gamma_{-2},\ \gamma_{-1},\ \gamma_0,\ \gamma_1,\ \gamma_2,\ \gamma_3] \\
\vec{s}_{S1,M_1-1} &= [\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ \ldots,\ 0,\ \gamma_{-3},\ \gamma_{-2},\ \gamma_{-1},\ \gamma_0,\ \gamma_1]
\end{aligned}$$

and

$$\begin{aligned}
\vec{w}_{M_1} &= [\ \epsilon_0,\ \epsilon_1,\ \epsilon_2,\ \epsilon_3,\ \epsilon_4,\ \epsilon_5,\ \ldots,\ 0,\ 0,\ \epsilon_{-4},\ \epsilon_{-3},\ \epsilon_{-2},\ \epsilon_{-1}] \\
\vec{w}_{SM_1+1} &= [\epsilon_{-2},\ \epsilon_{-1},\ \epsilon_0,\ \epsilon_1,\ \epsilon_2,\ \epsilon_3,\ \ldots,\ 0,\ 0,\ 0,\ 0,\ \epsilon_{-4},\ \epsilon_{-3}] \\
\vec{w}_{SM_1+2} &= [\epsilon_{-4},\ \epsilon_{-3},\ \epsilon_{-2},\ \epsilon_{-1},\ \epsilon_0,\ \epsilon_1,\ \ldots,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0] \\
&\vdots \\
\vec{w}_{SN-2} &= [\ \epsilon_4,\ 0,\ 0,\ 0,\ 0,\ 0,\ \ldots,\ \epsilon_{-2},\ \epsilon_{-1},\ \epsilon_0,\ \epsilon_1,\ \epsilon_2,\ \epsilon_3] \\
\vec{w}_{SN-1} &= [\ \epsilon_2,\ \epsilon_3,\ \epsilon_{-4},\ 0,\ 0,\ 0,\ \ldots,\ \epsilon_{-4},\ \epsilon_{-3},\ \epsilon_{-2},\ \epsilon_{-1},\ \epsilon_0,\ \epsilon_1]
\end{aligned}$$

with:

$$\begin{aligned}
\gamma_{-3} &= -0.064538883 \\
\gamma_{-2} &= -0.040689418 \\
\gamma_{-1} &= 0.418092273 \\
\gamma_0 &= 0.788485616 \\
\gamma_1 &= 0.418092273 \\
\gamma_2 &= -0.040689418 \\
\gamma_3 &= -0.064538883
\end{aligned}$$

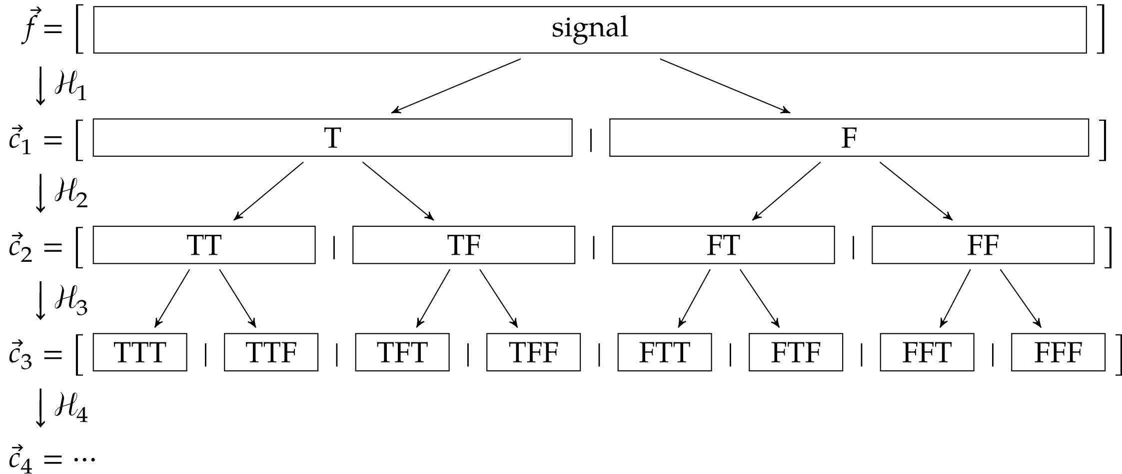
$$\begin{aligned}
\epsilon_{-3} &= -0.037828456 \\
\epsilon_{-2} &= -0.023849465 \\
\epsilon_{-1} &= 0.110624404 \\
\epsilon_0 &= 0.377402856 \\
\epsilon_1 &= -0.852698679 \\
\epsilon_2 &= 0.377402856 \\
\epsilon_3 &= 0.110624404 \\
\epsilon_4 &= -0.023849465 \\
\epsilon_5 &= -0.037828456
\end{aligned}$$

9.6 Wavelet packet transforms

Wavelet packet transforms are very related to ordinary wavelet transforms. The only difference is that the fluctuation signals at level n are also decomposed in a trend and a fluctuation signal.

The decomposition of the fluctuation signals may help in further compressing the information

into fewer significant transformed values.



9.7 Edge effects

The wavelets we've treated so far all assume that the signal under consideration is periodical. This can be concluded from the fact that the scaling signals and wavelets 'wrap around' when shifted.

We can avoid this wrapping around, by extending the signal to its left and to its right with extra data. We can:

- add zeros to the left and to the right (zero padding)
- perform a zeroth-order extension (replicating the edge sample multiple times)
- perform an N-th order polynomial extrapolation (fitting a polynomial through the N points closest to the edge) to find new values
- perform a symmetrical extension past the edge, considering the edge to be a symmetry axis
- come up with a new scheme to find values for the extra data

MATLAB facilitates this signal extension, by a convenience function `dwtmode`. You can check out the options, by issuing:

```
help dwtmode
```

You can inspect the currently active border extension mode, by issuing without any arguments:

```
dwtmode
```

The function applies to both one-dimensional and two-dimensional wavelet calculation functions from the Signal Processing toolbox.

9.8 Wavelets and Digital Filtering

So far, so good. We've seen quite a few wavelet transforms, even biorthonormal wavelet transforms and wavelet packet transforms. We've seen a number of applications, but the treatment was — admittedly — rather mathematical. By now, you must have the impression that wavelets are a mathematical analysis technique that can only be applied in batch mode, i.e. one takes a full image and treats it.

Probably, you're so prejudiced by now that you think you need a big chunky computer with loads of computing power, memory and a whopping 1TB solid-state drive to do the job.

However, reality is very different. Wavelet transformations are most apt to be implemented in the low-memory, low-power, high-throughput environment of DSPs. The reason for this is that every scaling signal and wavelet can be considered to be a small FIR filter. Even more: all the scaling signals and all the wavelets for a given level, can be implemented using only two filters:

- a scaling filter, a.k.a. the *low-pass* filter, and
- a wavelet filter, a.k.a. the *high-pass* filter.

For reasons that will become clear later, using this approach we enter the world of *subband filtering*.

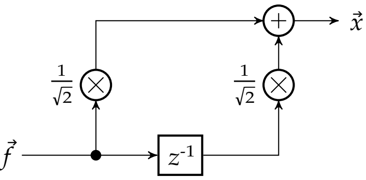
Let's use the Haar transform (chosen for its simplicity) as a vehicle to illustrate the principles.

9.8.1 The Haar transform as subband filter

The forward transform Let's reconsider the level-1 Haar transform. The first trend coefficients can be calculated as:

$$\begin{aligned} c_1[0] &= \langle \vec{s}_{1,0}, \vec{f} \rangle = \frac{f_0 + f_1}{\sqrt{2}} \\ c_1[1] &= \langle \vec{s}_{1,1}, \vec{f} \rangle = \frac{f_2 + f_3}{\sqrt{2}} \\ c_1[2] &= \langle \vec{s}_{1,2}, \vec{f} \rangle = \frac{f_4 + f_5}{\sqrt{2}} \\ &\vdots \end{aligned}$$

Now consider the filter setup below:



Given $f = [f_0, f_1, f_2, \ldots]$, one can easily verify that:

$$x[n] = \left[\frac{f_0}{\sqrt{2}}, \frac{f_0+f_1}{\sqrt{2}}, \frac{f_1+f_2}{\sqrt{2}}, \frac{f_2+f_3}{\sqrt{2}}, \frac{f_3+f_4}{\sqrt{2}}, \frac{f_4+f_5}{\sqrt{2}}, \frac{f_5+f_6}{\sqrt{2}}, \frac{f_6+f_7}{\sqrt{2}}, \ldots \right] \tag{9.174}$$

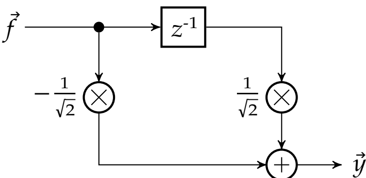
If you take a careful look at (9.174), you will find that the trend coefficients are present on the odd time-point positions, i.e. for $n = 1, 3, 5, 7, \ldots$ Decimation by a factor of 2 ($x_{decim}[n] = x[2n+1]$), is a simple way to get rid of the even time points. This completes the low-pass filter we referred to earlier.

Similarly, the first fluctuation coefficients of the level-1 Haar transform can be calculated as:

$$
\begin{aligned}
c_1[M_1] &= \left\langle \vec{w}_{M_1}, \vec{f} \right\rangle = \frac{f_0 - f_1}{\sqrt{2}} \\
c_1[M_1+1] &= \left\langle \vec{w}_{M_1+1}, \vec{f} \right\rangle = \frac{f_2 - f_3}{\sqrt{2}} \\
c_1[M_1+2] &= \left\langle \vec{w}_{M_1+2}, \vec{f} \right\rangle = \frac{f_4 - f_5}{\sqrt{2}} \\
&\vdots
\end{aligned}
$$

with $M_1 = N/2$.

Now consider the filter setup below:

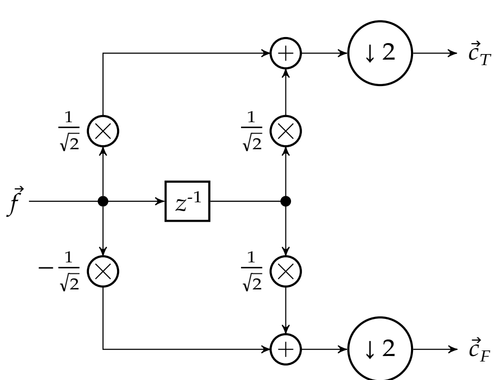


Given $f = [f_0, f_1, f_2, \ldots]$, one can easily verify that:

$$
y[n] = \left[-\frac{f_0}{\sqrt{2}},\ \frac{f_0 - f_1}{\sqrt{2}},\ \frac{f_1 - f_2}{\sqrt{2}},\ \frac{f_2 - f_3}{\sqrt{2}},\ \frac{f_3 - f_4}{\sqrt{2}},\ \frac{f_4 - f_5}{\sqrt{2}},\ \frac{f_5 - f_6}{\sqrt{2}},\ \frac{f_6 - f_7}{\sqrt{2}},\ \ldots \right] \tag{9.175}
$$

If you take a careful look at (9.175), you will find that the fluctuation coefficients are present on the odd time-point positions, i.e. for $n = 1, 3, 5, 7, \ldots$. Decimation by a factor of 2 ($y_{decim}[n] = y[2n+1]$), is a simple way to get rid of the even time points. This completes the high-pass filter we referred to earlier.

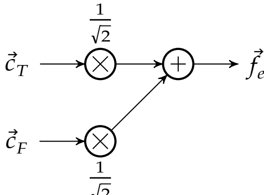
Combining the two filters, and adding the decimation blocks (that effectively reduce the sample rate by a factor of 2) results in the common *subband decomposition* setup for the Haar transform:



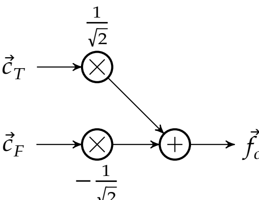
The inverse transform By analyzing the equations of the inverse Haar transform (see (9.10) to (9.11) on page 226), one can see that the even and odd time points originate from different equations, and every consecutive pair of time points is generated using the same wavelet coefficients (a scaling coefficient and a wavelet coefficient). Generically:

$$f[2i] = \frac{c_1[i] + c_1[M_1 + i]}{\sqrt{2}}$$
$$f[2i+1] = \frac{c_1[i] - c_1[M_1 + i]}{\sqrt{2}}$$

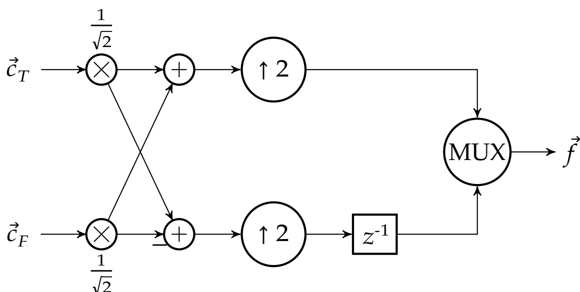
The first equation can be realized by the following filter setup:



The second equation can be realized by the following filter setup:



With a little optimization, the two can be combined into:



Note that we added two upsampling units (inserting zeros after every time point), an extra delay element and a synchronized clocked multiplexer.

9.8.2 Generic wavelet transform as subband filter

Forward transform The principle outlined for the Haar transform (in the previous section) can be generalized easily for arbitrary wavelet transforms.

This has been illustrated for a generic wavelet transform with a support length of 6 in Figure 9.21 on the following page.

Inverse transform The principle outlined for the Haar transform (in the previous section) can also easily be generalized for arbitrary wavelet transforms.

This has been illustrated for a generic wavelet transform with a support length of 6 in Figure 9.22 on the next page.

9.8.3 Recursive subband decomposition and recomposition

Of course, one can continue applying the decomposition to the trend data stream, in order to generate higher-level wavelet transforms.

Let's represent a single Wavelet decomposition step by the block diagram symbol on the left-hand below. The block takes a signal as its input and produces two half-rate signals as its output. Similarly, we represent a single Wavelet recomposition step by the block diagram symbol on the right-hand below. The latter block takes two half-rate signals as inputs and produces a full-rate output signal.

Given these conventions, the recursive subband decomposition and recomposition scheme corresponding to a 5-level wavelet transform can be found in Figure 9.23.

Given these conventions, the recursive subband decomposition and recomposition scheme corresponding to a 5-level *wavelet packet transform* can be found in Figure 9.24.

9.9 Two-dimensional wavelet transforms

When starting to read this section, you might be a little worried: the theory on one-dimensional wavelet transforms was pretty hefty. Will you be able to make yet another step up, towards the two-dimensional wavelet transform?

Luckily, the two-dimensional wavelet transform is nothing more than applying the one-dimensional wavelet transform twice. Therefore, we don't need to study new scaling signals or wavelets, we can just reuse all our knowledge of the one-dimensional transforms.

9.9.1 Principles

Consider an $M \times N$-pixel image, with M and N even. If M or N are not even, just extend the image with a single pixel row or column.

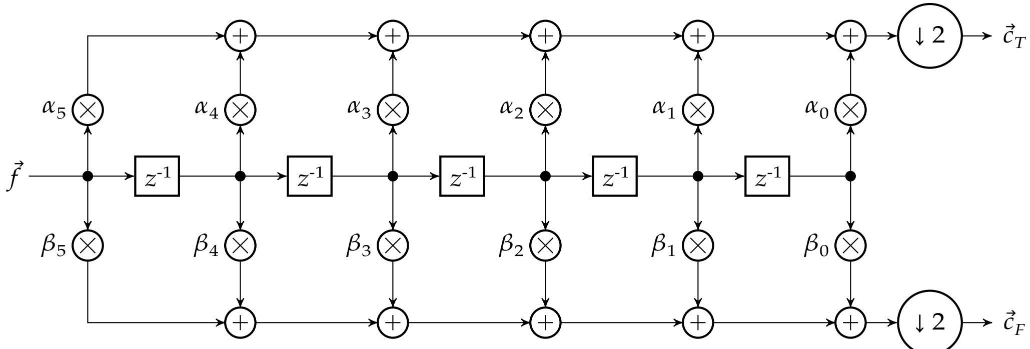


Figure 9.21: Generic scheme for a subband *decomposition* filter, illustrated for a wavelet transform with a support length of 6

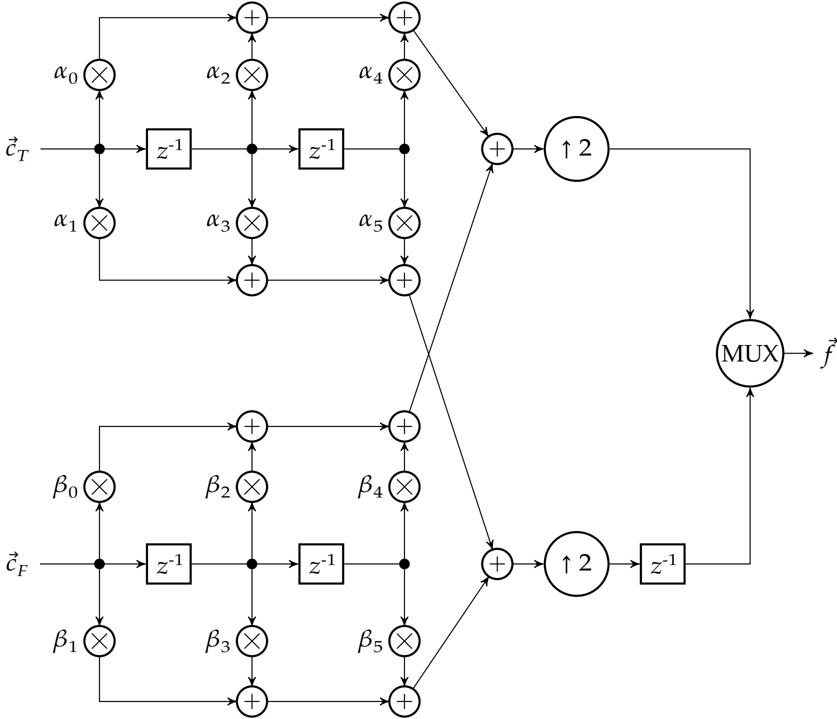


Figure 9.22: Generic scheme for a subband *(re)composition* filter, illustrated for a wavelet transform with a support length of 6

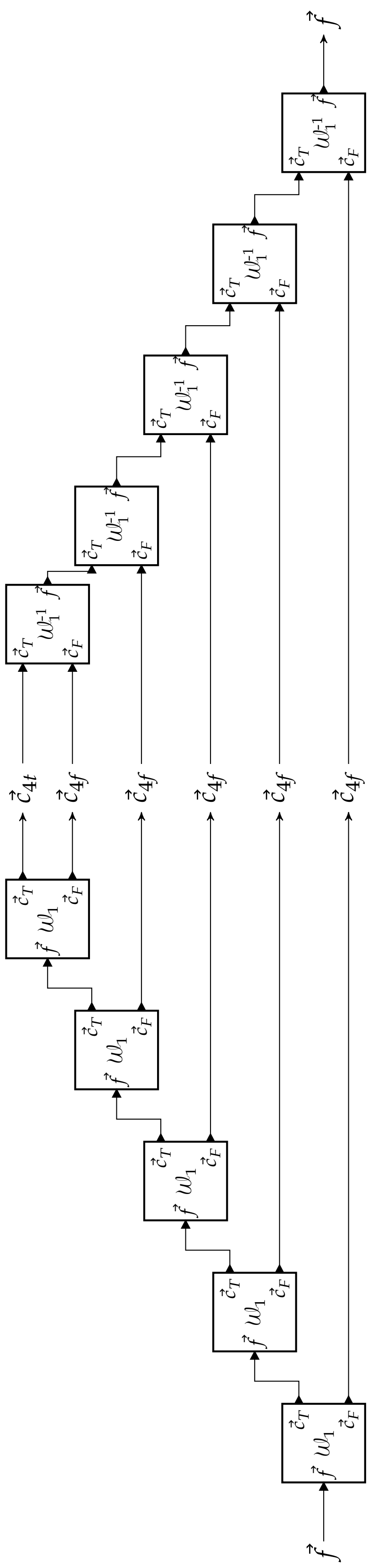


Figure 9.23: Block diagram of the recursive subband decomposition scheme (on the left-hand side), and the recusive subband recomposition scheme (on the right-hand side) for an arbitrary wavelet transform

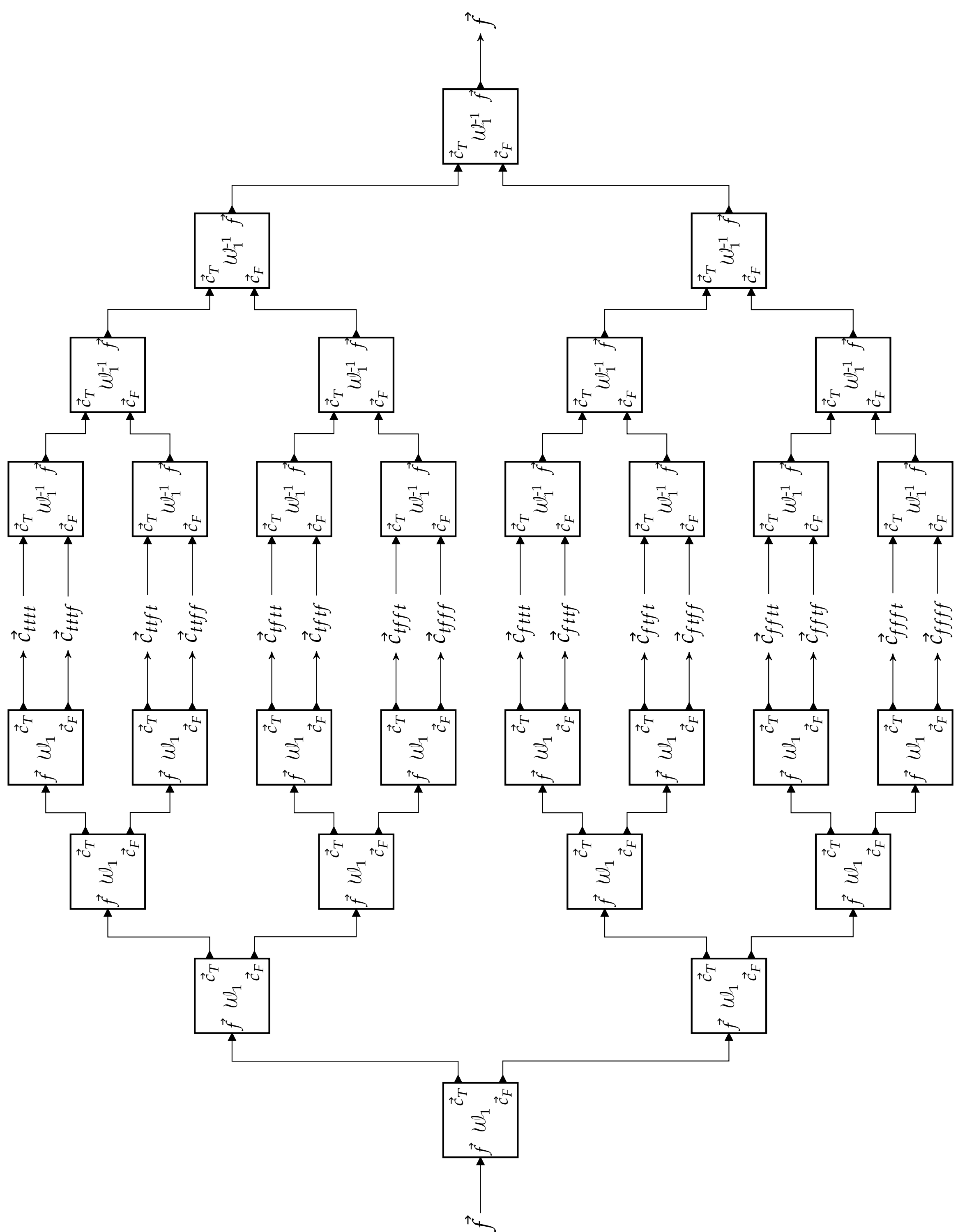


Figure 9.24: Block diagram of the recursive subband decomposition scheme (on the left-hand side), and the recusive subband recomposition scheme (on the right-hand side) for an arbitrary wavelet packet transform

The image can be represented by its intensity matrix:

$$\vec{f} = \begin{bmatrix} f_{1,1} & f_{1,2} & f_{1,3} & \cdots & f_{1,N} \\ f_{2,1} & f_{2,2} & f_{2,3} & \cdots & f_{2,N} \\ f_{3,1} & f_{3,2} & f_{3,3} & \cdots & f_{3,N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ f_{M,1} & f_{M,2} & f_{M,3} & \cdots & f_{M,N} \end{bmatrix}$$

The level-1 wavelet transform

We can follow two identical paths:

- Apply the wavelet transform for every column of the matrix, and then again for every row of the intermediate result:

$$\vec{f} = \left[\boxed{\text{signal}}\right] \xrightarrow{\mathcal{W}_{1C}} \vec{c}_{1C} = \left[\begin{array}{|c|}\hline \text{trend} \\ \hline \text{fluct} \\ \hline\end{array}\right] \xrightarrow{\mathcal{W}_{1R}} \vec{c}_1 = \left[\begin{array}{|c|c|}\hline \text{1-t} & \text{1-h} \\ \hline \text{1-v} & \text{1-d} \\ \hline\end{array}\right]$$

- Apply the wavelet transform for every row of the matrix, and then again for every column of the intermediate result:

$$\vec{f} = \left[\boxed{\text{signal}}\right] \xrightarrow{\mathcal{W}_{1R}} \vec{c}_{1R} = \left[\begin{array}{|c|c|}\hline \text{trend} & \text{fluct} \\ \hline\end{array}\right] \xrightarrow{\mathcal{W}_{1C}} \vec{c}_1 = \left[\begin{array}{|c|c|}\hline \text{1-t} & \text{1-h} \\ \hline \text{1-v} & \text{1-d} \\ \hline\end{array}\right]$$

However, the end result is exactly the same. We refer to 1-t, 1-h, 1-v and 1-d respectively as the *the first trend subframe*, the *first horizontal fluctuation subframe*, the *first vertical fluctuation subframe* and the *first diagonal fluctuation subframe*.

Remarks

- As the individual steps are energy preserving, so is the two-dimensional wavelet transform. Note that the energy of an image is calculated using the Frobenius norm, and not the matrix 2-norm. The Frobenious norm for an image $\vec{f} = \left[f_{i,j}\right]$ is defined by:

$$\left\|\vec{f}\right\|_F = \sqrt{\sum_{i,j} f_{i,j}^2}$$

- It's not very common, but one could choose to apply a different wavelet transform to the rows as one applies to the columns. The result is still a valid two-dimensional wavelet transform.
- Alas, the majority of all papers and books, and even MATLAB, use a crooked version of the fluctuation definitions stated above. They commit two errors:
  1. They interchange the notions of *horizontal fluctuation subframe* and *vertical fluctuation subframe*. The reason for this is that horizontal fluctuations along the rows are seen in an image as a vertical edge and vice versa.
  2. They organize the transformed intensity matrix differently. They interchange the positions of the horizontal fluctuation subframe and the vertical fluctuation subframe

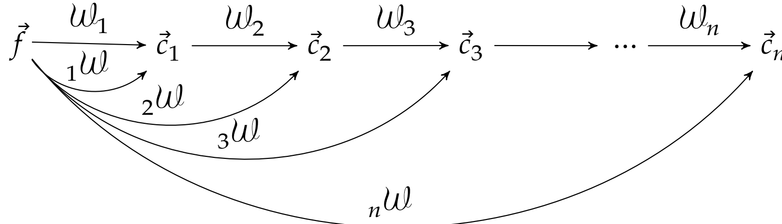
The combination of both errors cancels out, and therefore often remains unnoticed. The diagram below, labeling the first errors as (1) and the second error as (2) illustrates this:

$$\vec{f} \xrightarrow{\mathcal{W}_1} \vec{c}_1 = \left[\begin{array}{|c|c|} \hline \text{1-t} & \text{1-h} \\ \hline \text{1-v} & \text{1-d} \\ \hline \end{array} \right] \xrightarrow{(1)} \left[\begin{array}{|c|c|} \hline \text{1-t} & \text{1-v} \\ \hline \text{1-h} & \text{1-d} \\ \hline \end{array} \right] \xrightarrow{(2)} \left[\begin{array}{|c|c|} \hline \text{1-t} & \text{1-h} \\ \hline \text{1-v} & \text{1-d} \\ \hline \end{array} \right]$$

Make sure you know which reference framework you're using when setting up a meeting to discuss two-dimensional wavelet problems with your colleagues! In this text, we will systematically use the conceptually correct version.

The n-level wavelet transform The n-level wavelet transform is obtained by recursively applying the two-dimensional level-1 wavelet transform to the trend subframe. This has been illustrated in Figure 9.25 on the facing page.

Again, we use the same notational convention:



9.9.2 Example

Example - by hand Let's consider the simple intensity image below:

$$\vec{f} = \begin{bmatrix} 0.5 & 1 & 0.8 & 0.7 \\ 1 & 1 & 0.3 & 0.2 \\ 0.8 & 0.7 & 0.5 & 0.4 \\ 0.7 & 0.8 & 0.4 & 0.1 \end{bmatrix}$$

We can easily calculate the total energy in the signal by summing up the squares of the individual elements:

$$\begin{aligned} \left\|\vec{f}\right\|_F^2 &= 0.5^2 + 1^2 + 0.8^2 + 0.7^2 + 1^2 + 1^2 + 0.3^2 + 0.2^2 + 0.8^2 + 0.7^2 + 0.5^2 + 0.4^2 \\ &\quad + 0.7^2 + 0.8^2 + 0.4^2 + 0.1^2 = 7.35 \end{aligned}$$

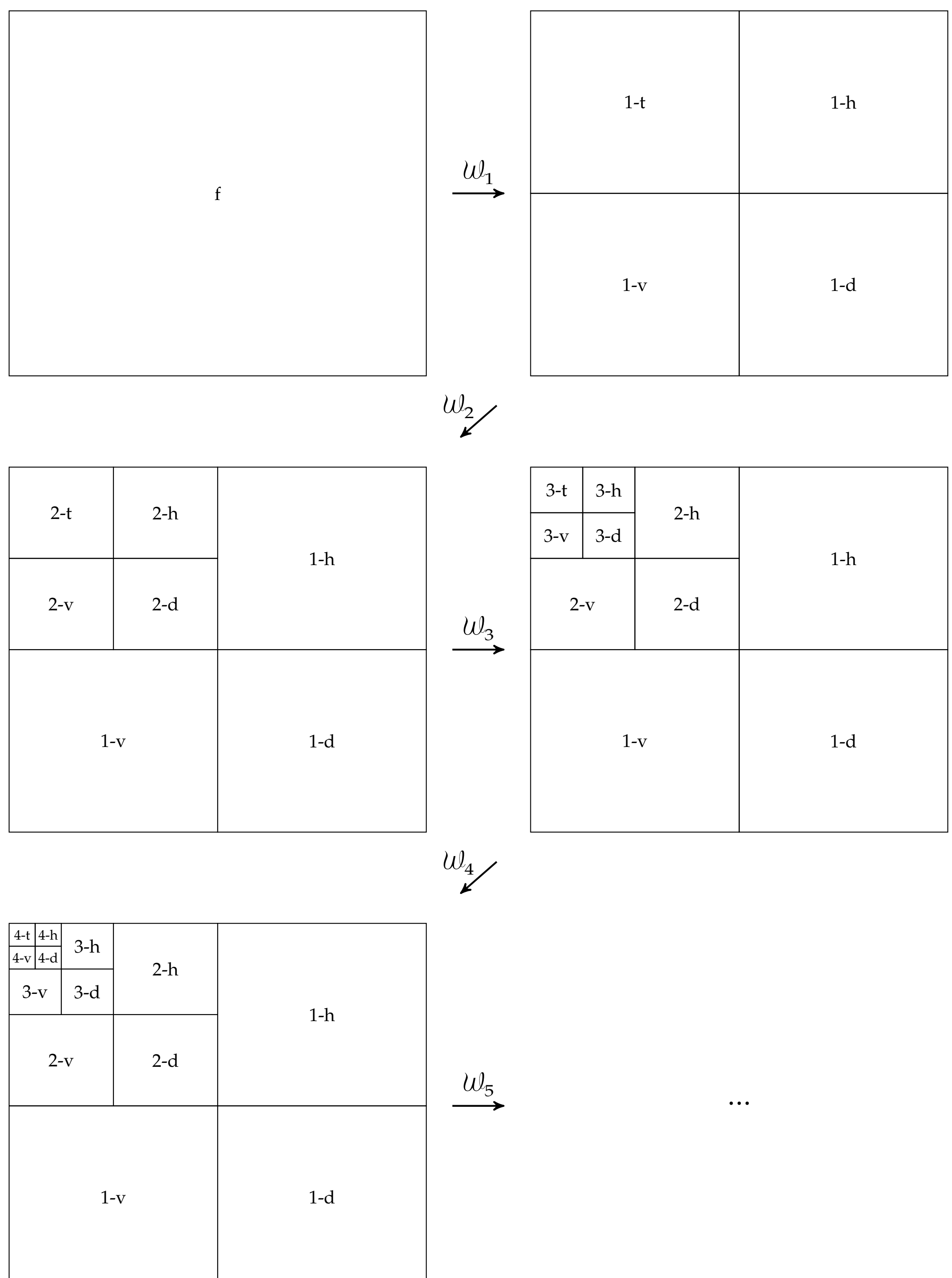


Figure 9.25: Illustration of the hierarchical nature of the two-dimensional wavelet decomposition

The first step is to apply the level-1 Haar transform to its columns. This results in $\vec{f} \xrightarrow{\mathcal{H}_{1C}} \vec{c}_{1C}$:

$$\vec{c}_{1C} = \begin{bmatrix} \frac{0.5+1}{\sqrt{2}} & \frac{1+1}{\sqrt{2}} & \frac{0.8+0.3}{\sqrt{2}} & \frac{0.7+0.2}{\sqrt{2}} \\ \frac{0.8+0.7}{\sqrt{2}} & \frac{0.7+0.8}{\sqrt{2}} & \frac{0.5+0.4}{\sqrt{2}} & \frac{0.4+0.1}{\sqrt{2}} \\ \frac{0.5-1}{\sqrt{2}} & \frac{1-1}{\sqrt{2}} & \frac{0.8-0.3}{\sqrt{2}} & \frac{0.7-0.2}{\sqrt{2}} \\ \frac{0.8-0.7}{\sqrt{2}} & \frac{0.7-0.8}{\sqrt{2}} & \frac{0.5-0.4}{\sqrt{2}} & \frac{0.4-0.1}{\sqrt{2}} \end{bmatrix}$$
$$= \begin{bmatrix} 1.0607 & 1.4142 & 0.7778 & 0.6364 \\ 1.0607 & 1.0607 & 0.6364 & 0.3536 \\ -0.3536 & 0 & 0.3536 & 0.3536 \\ 0.0707 & -0.0707 & 0.0707 & 0.2121 \end{bmatrix}$$

The second step is to apply the level-1 Haar transform to the obtained rows. This results in $\vec{c}_{1C} \xrightarrow{\mathcal{H}_{1R}} \vec{c}_1$:

$$\vec{c}_1 = \begin{bmatrix} \frac{1.0607+1.4142}{\sqrt{2}} & \frac{0.7778+0.6364}{\sqrt{2}} & \frac{1.0607-1.4142}{\sqrt{2}} & \frac{0.7778-0.6364}{\sqrt{2}} \\ \frac{1.0607+1.0607}{\sqrt{2}} & \frac{0.6364+0.3536}{\sqrt{2}} & \frac{1.0607-1.0607}{\sqrt{2}} & \frac{0.6364-0.3536}{\sqrt{2}} \\ \frac{-0.3536+0}{\sqrt{2}} & \frac{0.3536+0.3536}{\sqrt{2}} & \frac{-0.3536-0}{\sqrt{2}} & \frac{0.3536-0.3536}{\sqrt{2}} \\ \frac{0.0707-0.0707}{\sqrt{2}} & \frac{0.0707+0.2121}{\sqrt{2}} & \frac{0.0707+0.0707}{\sqrt{2}} & \frac{0.0707-0.2121}{\sqrt{2}} \end{bmatrix}$$
$$= \begin{bmatrix} 1.75 & 1 & -0.25 & 0.1 \\ 1.5 & 0.7 & 0 & 0.2 \\ -0.25 & 0.5 & -0.25 & 0 \\ 0 & 0.2 & 0.1 & -0.1 \end{bmatrix}$$

We can verify the energy preservation by calculating the Frobenius norm of the result:

$$\left\|\vec{f}\right\|_F^2 = 1.75^2 + 1^2 + (-0.25)^2 + 0.1^2 + 1.5^2 + 0.7^2 + 0.2^2 + (-0.25)^2 + 0.5^2 + (-0.25)^2$$
$$+ 0.2^2 + 0.1^2 + (-0.1)^2 - 7.35$$

The first trend signal is readily obtained by zeroing the h, v, and d quadrants of $\vec{c}_1$:

$$\vec{c}_{1-t} = \begin{bmatrix} 1.75 & 1 & 0 & 0 \\ 1.5 & 0.7 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The first trend signal in the image domain can easily be calculated by applying the inverse

Haar transform:

$$\vec{f}_{1-t} = \begin{bmatrix} \frac{1.75+0}{2} & \frac{1.75-0}{2} & \frac{1+0}{2} & \frac{1-0}{2} \\ \frac{1.75+0}{2} & \frac{1.75-0}{2} & \frac{1+0}{2} & \frac{1-0}{2} \\ \frac{1.5+0}{2} & \frac{1.5-0}{2} & \frac{0.7+0}{2} & \frac{0.7-0}{2} \\ \frac{1.5+0}{2} & \frac{1.5-0}{2} & \frac{0.7+0}{2} & \frac{0.7-0}{2} \end{bmatrix}$$

$$= \begin{bmatrix} 0.875 & 0.875 & 0.5 & 0.5 \\ 0.875 & 0.875 & 0.5 & 0.5 \\ 0.75 & 0.75 & 0.35 & 0.35 \\ 0.75 & 0.75 & 0.35 & 0.35 \end{bmatrix}$$

Note that this is again a reduced resolution image.

The total energy embedded in the transformed trend signal is easily calculated as:

$$\|\vec{c}_{1-t}\|_F^2 = 1.75^2 + 1^2 + 1.5^2 + 0.7^2 = 6.8025$$

and because of the energy preservation theorem it is equal to the total energy embedded in the trend signal:

$$\begin{aligned}\|\vec{f}_{1-t}\|_F^2 &= 0.875^2 + 0.875^2 + 0.5^2 + 0.5^2 + 0.875^2 + 0.875^2 + 0.5^2 + 0.5^2 \\ &+ 0.75^2 + 0.75^2 + 0.35^2 + 0.35^2 + 0.75^2 + 0.75^2 + 0.35^2 + 0.35^2 = 6.8025\end{aligned}$$

The first horizontal fluctuation signal is obtained by zeroing all the quadrants, except for the 1-h quadrant.

$$\vec{c}_{1-h} = \begin{bmatrix} 0 & 0 & -0.25 & 0.1 \\ 0 & 0 & 0 & 0.2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The first horizontal fluctuation signal in the image domain is:

$$\vec{f}_{1-h} = \begin{bmatrix} \frac{0+(-0.25)}{2}, & \frac{0-(-0.25)}{2}, & \frac{0+0.1}{2}, & \frac{1-0.1}{2} \\ \frac{0+(-0.25)}{2}, & \frac{0-(-0.25)}{2}, & \frac{0+0.1}{2}, & \frac{1-0.1}{2} \\ \frac{0+0}{2}, & \frac{0-0}{2}, & \frac{0+0.2}{2}, & \frac{0-0.2}{2} \\ \frac{0+0}{2}, & \frac{0-0}{2}, & \frac{0+0.2}{2}, & \frac{0-0.2}{2} \end{bmatrix}$$

$$= \begin{bmatrix} -0.125 & 0.125 & 0.05 & -0.05 \\ -0.125 & 0.125 & 0.05 & -0.05 \\ 0 & 0 & 0.1 & -0.1 \\ 0 & 0 & 0.1 & -0.1 \end{bmatrix}$$

The total energy embedded in the transformed horizontal fluctuation signal is equal to:

$$\|\vec{c}_{1-h}\|_F^2 = (-0.25)^2 + 0.1^2 + 0.2^2 = 0.1125$$

and because of the energy preservation theorem we find the same value for the total energy of the horizontal fluctuation in the image domain:

$$\left\|\vec{f}_{1-h}\right\|_F^2 = (-0.125)^2 + (0.125)^2 + 0.05^2 + (-0.05)^2 + (-0.125)^2 + 0.125^2 + 0.05^2 + (-0.05)^2 + 0.1^2 + (-0.1)^2 + 0.1^2 + (-0.1)^2 = 0.1125$$

Similarly, we can calculate the first vertical fluctuation signal by zeroing all the quadrants, except for the 1-v quadrant.

$$\vec{c}_{1-v} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -0.25 & 0.5 & 0 & 0 \\ 0 & 0.2 & 0 & 0 \end{bmatrix}$$

The first vertical fluctuation signal in the image domain can be calculated to be:

$$\vec{f}_{1-v} = \begin{bmatrix} -0.125 & -0.125 & 0.25 & 0.25 \\ 0.125 & 0.125 & -0.25 & -0.25 \\ 0 & 0 & 0.1 & 0.1 \\ 0 & 0 & -0.1 & -0.1 \end{bmatrix}$$

For the total energy embedded in the transformed vertical fluctuation signal we find:

$$\|\vec{c}_{1-v}\|_F^2 = (-0.25)^2 + 0.5^2 + 0.2^2 = 0.3525$$

Finally, we can calculate the diagonal fluctuation signal by zeroing all the quadrants, except for the 1-d quadrant.

$$\vec{c}_{1-d} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -0.25 & 0 \\ 0 & 0 & 0.1 & -0.1 \end{bmatrix}$$

The first diagonal fluctuation signal in the image domain is:

$$\vec{f}_{1-d} = \begin{bmatrix} -0.125 & 0.125 & 0 & 0 \\ 0.125 & -0.125 & 0 & 0 \\ 0.05 & -0.05 & -0.05 & 0.05 \\ -0.05 & 0.05 & 0.05 & -0.05 \end{bmatrix}$$

For the total energy embedded in the transformed diagonal fluctuation signal we find:

$$\|\vec{c}_{1-d}\|_F^2 = (-0.25)^2 + 0.1^2 + (-0.1)^2 = 0.0825$$

You can verify the statement below:

$$\left\|\vec{f}\right\|_F^2 = \left\|\vec{f}_{1-t}\right\|_F^2 + \left\|\vec{f}_{1-h}\right\|_F^2 + \left\|\vec{f}_{1-v}\right\|_F^2 + \left\|\vec{f}_{1-d}\right\|_F^2 = \|\vec{c}_{1-t}\|_F^2 + \|\vec{c}_{1-h}\|_F^2 + \|\vec{c}_{1-v}\|_F^2 + \|\vec{c}_{1-d}\|_F^2$$

Note however, that most of the energy of the signal (6.8025/7.35 = 92.55%) ends up in the trend signal, whereas only a small amount ends up in the fluctuation signals (0.5475/7.35 = 7.45%). Again, this illustrates the energy compaction toward the trend subframe.

We can continue by applying the Haar transform again to the trend signal. We leave this as an exercise for the reader.

Example - using Matlab The calculations above were tedious and time-consuming. Let's call Matlab to our rescue. The test signal is easily defined:

```
f = [ 0.5,   1, 0.8, 0.7 ;
        1,   1, 0.3, 0.2 ;
      0.8, 0.7, 0.5, 0.4 ;
      0.7, 0.8, 0.4, 0.1 ];
```

The two-dimensional level-1 Haar transform can be easily calculated using the `dwt2` function, but remember the awkward Matlab conventions, so we interchanged $\vec{c}_{1-v}$ and $\vec{c}_{1-h}$:

```
[ca1, cv1, ch1, cd1 ] = dwt2( f, 'db1' );
```

The full coefficient matrix can be composed as:

```
c = [ ca1, ch1; cv1, cd1 ];
```

Energy levels of signals are most easily calculated using the norm function. E.g., the energy in the image $\vec{f}$ is obtained by:

```
energy = norm( f, 'fro' )^2;
```

The inverse level1 Haar transformation is invoked using the `idwt2` function:

```
g = idwt2( ca1, cv1, ch1, cd1, 'db1' );
```

Note again the awkward order of `cv1` and `ch1`.

The `idwt2` function can also be used to obtain the trend and fluctuation signals in the image domain:

```
ft = idwt2( ca1, [], [], [], 'db1' );
fh = idwt2( [], [], ch1, [], 'db1' )
fv = idwt2( [], cv1, [], [], 'db1' );
fd = idwt2( [], [], [], cd1, 'db1' );
```

We can continue by applying the two-dimensional Haar transform once more to the first trend subframe:

```
[ca2, cv2, ch2, cd2 ] = dwt2( ca1, 'db1' );
```

The peculiar order of `cv2` and `ch1` should no longer be a suprise to you by now.

However, we can also perform the two-level two-dimensional Haar transform at once on the original image $\vec{f}$ using the wavedec2 function:

```
[c2,l2] = wavedec2( f, 2, 'db1' );
```

The resulting vector `c2` contains the result of the 2-level Haar transform in a single column vector. The `l2` vector contains information about the trend and fluctuation information present in `c2`. Check it out!

Again, convenience functions `appcoef2` and `detcoef2` exist to extract the trend and fluctuation coefficients from the `wavedec` results. One can use `wrcoef2` function to reconstruct the trend and the fluctuation subframes in the image domain. We leave it to the user to explore all the possibilities.

Example - somewhat more realistic Let's take a look at a somewhat more realistic example, but this time not focusing on the calculations, but on the resulting images.

Consider the image of an Italian monument below. It is an 800 × 512-pixel grayscale image.

We decomposed this image using Haar wavelets. You can find the results for increasing levels in Figure 9.26 on page 294. The multi-resolution analysis, obtained by recomposing the image only considering the trend coefficients for increasing levels, can be found in Figure 9.27 on the opposite page.

9.9.3 The base vectors of the two-dimensional wavelet transforms

Analogous to the one-dimensional case, every coefficient that results from a two-dimensional wavelet transform, must correspond to a scaling signal (in case it is a trend coefficient), or a wavelet (in case it is a horizontal, vertical or diagonal fluctuation coefficient). Often the term 'signal' is replaced by the term 'image', 'stamp' or 'kernel'. This leads to the terminology of scaling images and wavelet images, scaling stamps and wavelet stamps, or scaling kernels and wavelet kernels.

Decomposition The parallel goes even further. The wavelet transform is still a transformation, so every one of the scaling/wavelet images/stamps in fact is a base vector that can be used to decompose an arbitrary image.

Writing this decomposition in an analytical way leads to a rather complicated formula for an $M_0 \times N_0$ image:

$$\vec{f} = \underbrace{\sum_{i=0}^{M_1-1}\sum_{j=0}^{N_1-1} c_{i,j}\vec{s}_{i,j}}_{\text{1-t}} + \underbrace{\sum_{i=0}^{M_1-1}\sum_{j=N_1}^{N_0-1} c_{i,j}\vec{w}_{i,j}}_{\text{1-h}} + \underbrace{\sum_{i=M_1}^{M_0-1}\sum_{j=0}^{N_1-1} c_{i,j}\vec{w}_{i,j}}_{\text{1-v}} + \underbrace{\sum_{i=M_1}^{M_0-1}\sum_{j=N_1}^{N_0-1} c_{i,j}\vec{w}_{i,j}}_{\text{1-d}}$$

with the common definitions of $M_n = \frac{M_0}{2^n}$ and $N_n = \frac{N_0}{2^n}$.

In any case, the coefficients $c_{i,j}$ can be obtained by projecting the image onto the base vectors. This projection makes use of the scalar product, that is easily defined for matrices as the sum of the elementwise products.

Given: $A = [a_{i,j}]$ and $B = [b_{i,j}]$, then:

$$\langle A, B\rangle = \sum_i \sum_j a_{i,j} b_{i,j}$$

Projection In case the base vectors of the transform are orthonormal, the coefficients $c_{i,j}$ can be found using the classical projection formula:

$$\begin{aligned} c_{i,j} &= \left\langle \vec{f}, \vec{s}_{i,j}\right\rangle && \text{if } i < M_1 \text{ and } j < N_1 \text{(i.e. for the trend coefficients)} \\ c_{i,j} &= \left\langle \vec{f}, \vec{w}_{i,j}\right\rangle && \text{otherwise (i.e. for the wavelet coefficients)} \end{aligned} \tag{9.176}$$

The question is: how does one generate these base vectors, starting from the one-dimensional basevectors of the transform that we'd like to use?

We skip the tedious calculations and proceed immediately to the practical result.

Generation of scaling signal and wavelet stamps The starting point is arranging the base vectors of the one-dimensional column transform of choice as *columns* in an $M \times M$ base vector matrix B_C. Below, we illustrated this for a wavelet transform with a support length of 4:[13]

$$B_C = \begin{bmatrix} \alpha_0 & 0 & 0 & 0 & \cdots & \beta_0 & 0 & 0 & 0 & \cdots \\ \alpha_1 & 0 & 0 & 0 & \cdots & \beta_1 & 0 & 0 & 0 & \cdots \\ \alpha_2 & \alpha_0 & 0 & 0 & \cdots & \beta_2 & \beta_0 & 0 & 0 & \cdots \\ \alpha_3 & \alpha_1 & 0 & 0 & \cdots & \beta_3 & \beta_1 & 0 & 0 & \cdots \\ 0 & \alpha_2 & \alpha_0 & 0 & \cdots & 0 & \beta_2 & \beta_0 & 0 & \cdots \\ 0 & \alpha_3 & \alpha_1 & 0 & \cdots & 0 & \beta_3 & \beta_1 & 0 & \cdots \\ 0 & 0 & \alpha_2 & \alpha_0 & \cdots & 0 & 0 & \beta_2 & \beta_0 & \cdots \\ 0 & 0 & \alpha_3 & \alpha_1 & \cdots & 0 & 0 & \beta_3 & \beta_1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

[13] When discussing the one-dimensional wavelet transform, we denoted this matrix by H^T. To avoid a complicated notation, we will now label it B_C.

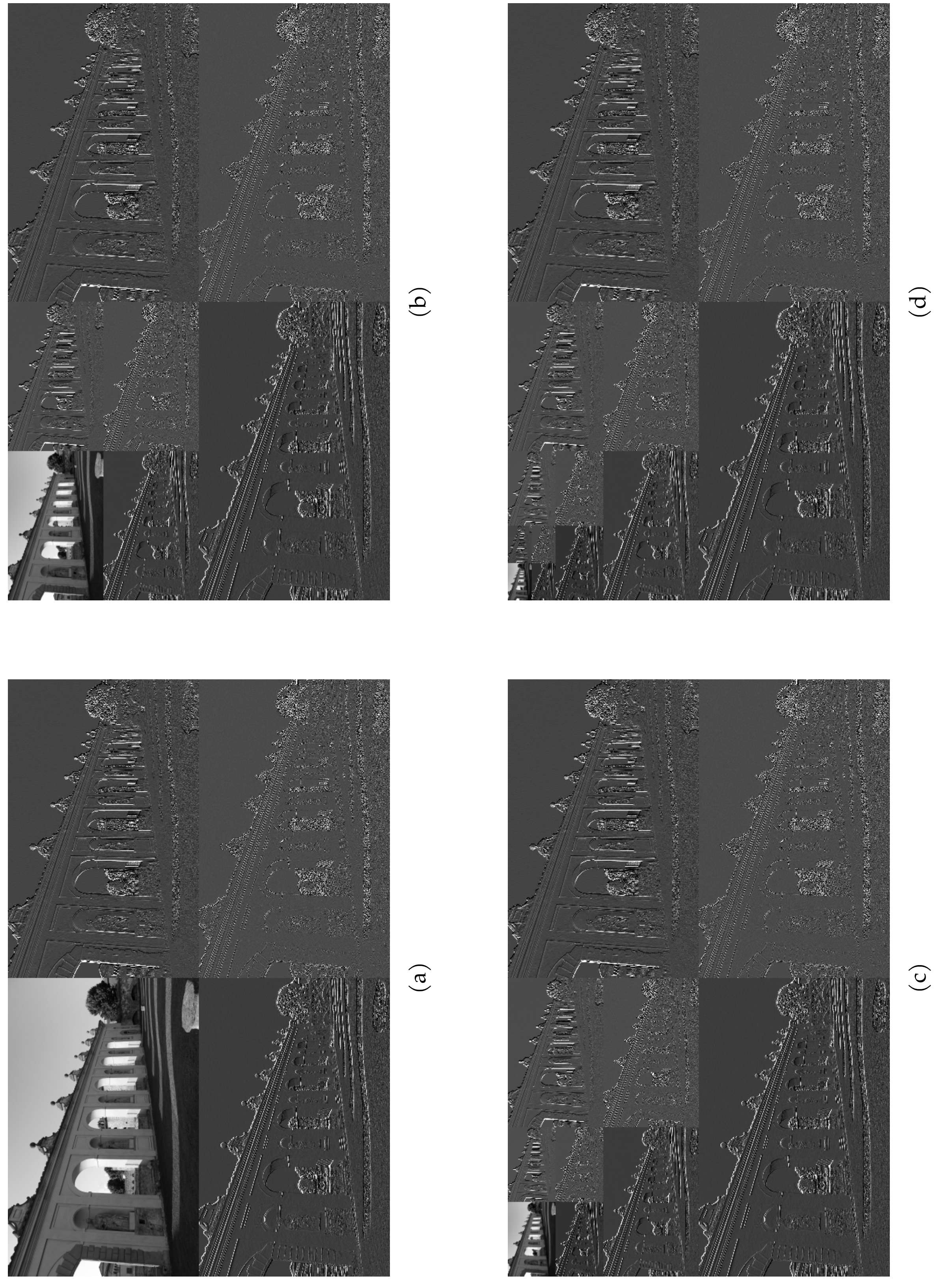


Figure 9.26: Haar decomposition of an Italian monumental wall: (a) level-1 decomposition, (b) 2-level decomposition, (c) 3-level decomposition, (d) 4-level composition

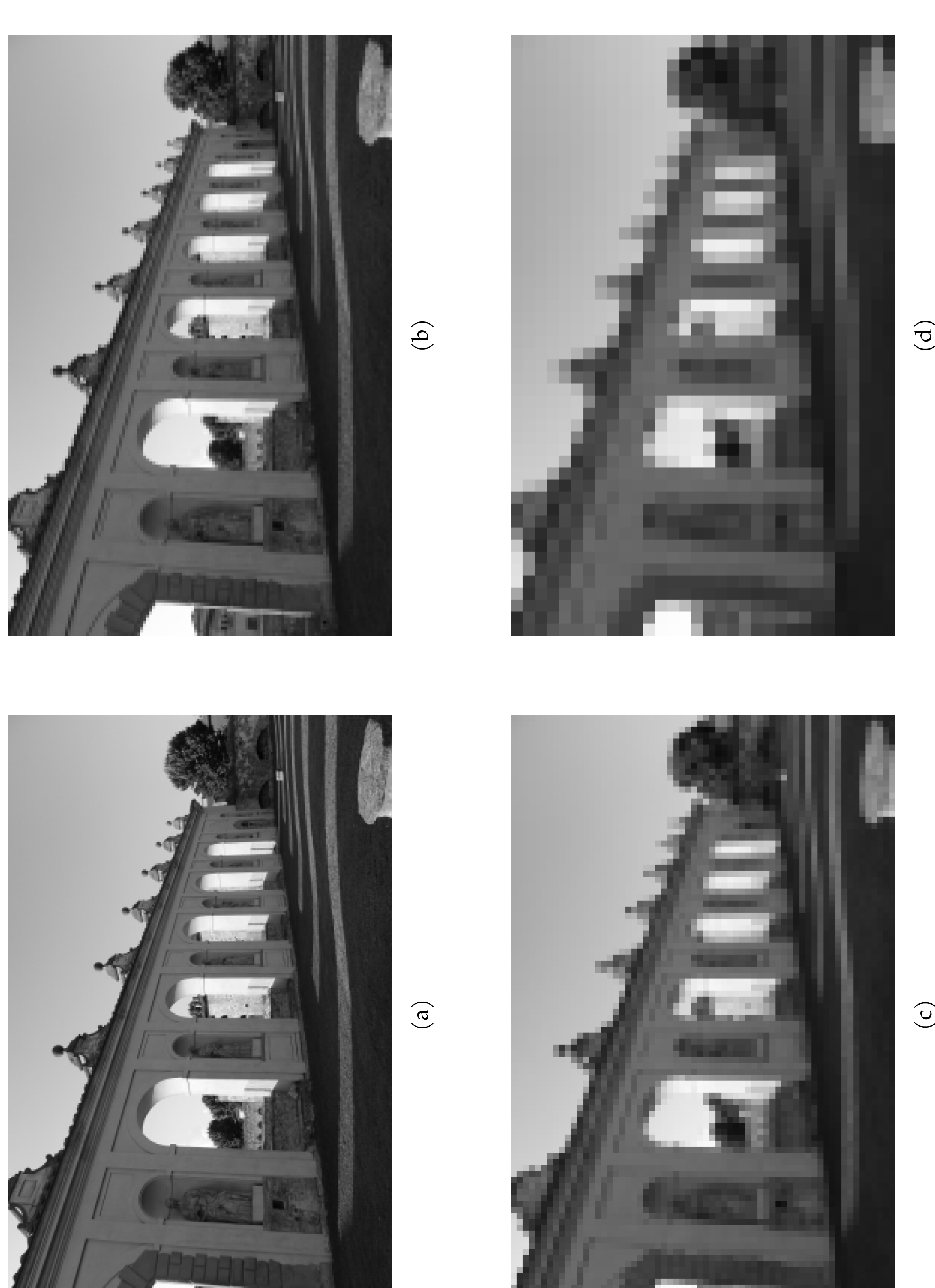


Figure 9.27: Haar recomposition of an Italian monumental wall, based on the trend coefficients of (a) the level-1 decomposition, (b) the 2-level decomposition, (c) the 3-level decomposition, (d) the 4-level composition

Likewise, we organize every base vector of the row transformation as columns in an $N \times N$ base vector matrix B_R.

$$B_R = \begin{bmatrix} \alpha_0 & 0 & 0 & 0 & \cdots & \beta_0 & 0 & 0 & 0 & \cdots \\ \alpha_1 & 0 & 0 & 0 & \cdots & \beta_1 & 0 & 0 & 0 & \cdots \\ \alpha_2 & \alpha_0 & 0 & 0 & \cdots & \beta_2 & \beta_0 & 0 & 0 & \cdots \\ \alpha_3 & \alpha_1 & 0 & 0 & \cdots & \beta_3 & \beta_1 & 0 & 0 & \cdots \\ 0 & \alpha_2 & \alpha_0 & 0 & \cdots & 0 & \beta_2 & \beta_0 & 0 & \cdots \\ 0 & \alpha_3 & \alpha_1 & 0 & \cdots & 0 & \beta_3 & \beta_1 & 0 & \cdots \\ 0 & 0 & \alpha_2 & \alpha_0 & \cdots & 0 & 0 & \beta_2 & \beta_0 & \cdots \\ 0 & 0 & \alpha_3 & \alpha_1 & \cdots & 0 & 0 & \beta_3 & \beta_1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Let's denote the i-th column of B_C by $B_{C-\bullet,i}$ and the j-th column of B_R as $B_{R-\bullet,j}$.[14] It can be proven that the scaling signal or wavelet image/stamp (denoted by $\vec{s}_{i,j}$ or $\vec{w}_{i,j}$) corresponding to coefficient $c_{i,j}$ is easily found as:

$$\vec{s}_{i,j} = B_{C-\bullet,i} B^T_{R-\bullet,j} \quad \text{if } i < M_1 \text{ and } j < N_1 \text{ (i.e. for the trend coefficients)} \tag{9.177}$$

$$\vec{w}_{i,j} = B_{C-\bullet,i} B^T_{R-\bullet,j} \quad \text{otherwise (i.e. for the wavelet coefficients)} \tag{9.178}$$

The product used above, is a so-called *outer product* of column vectors, as opposed to the *inner (scalar) product*. Another valid term is *tensor product*.

Definition: outer product of two vectors The outer product of an $M \times 1$ column vector $\vec{x} = [x_0, x_1, x_2, \ldots, x_{M-1}]^T$ and a $N \times 1$ column vector $\vec{y} = [y_0, y_1, y_2, \ldots, y_{N-1}]^T$ is denoted by $\vec{x} \otimes \vec{y}$, results in an $M \times N$ matrix and is defined by:

$$\vec{x} \otimes \vec{y} = \vec{x}\vec{y}^T$$

The $\otimes$-symbol used is the same as the one used for the thinning operation in image processing. The context almost always avoids confusion of the two.

Example Consider transforming a 8×6 image, using the Haar transform. We start by listing the base vectors of the column transform as columns in B_C:

$$B_C = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{bmatrix}$$

[14] Note that we use the *engineers' convention* to start numbering columns and rows from 0 instead of from 1. Therefore the leftmost column and top row are also referred to as the *zero-th* column and the *zero-th* row. The convention to start numbering from 1 is often used by mathematicians (and e.g., MATLAB). We use the former, because in almost any case, it leads to simpler equations.

The, we list the base vectors of the row transform as columns in B_R:

$$B_R = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & -1 \end{bmatrix}$$

Now, let's calculate some example scaling and wavelet stamps:

The scaling stamp $\vec{s}_{1,2}$ corresponding to coefficient $c_{1,2}$ is calculated according to (9.177) as:

$$\vec{s}_{1,2} = B_{C-\bullet,1} B_{R-\bullet,2}^T = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The wavelet $\vec{w}_{3,4}$ corresponding to coefficient $c_{3,4}$ of the first horizontal fluctuation is calculated according to (9.178) as:

$$\vec{w}_{3,4} = B_{C-\bullet,3} B_{R-\bullet,4}^T = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 0 & 1 & -1 & 0 & 0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \end{bmatrix}$$

The wavelet $\vec{w}_{5,1}$ corresponding to coefficient $c_{5,1}$ of the first vertical fluctuation is calculated according to (9.178) as:

$$\vec{w}_{5,1} = B_{C-\bullet,5} B_{R-\bullet,1}^T = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The wavelet $\vec{w}_{5,3}$ corresponding to coefficient $c_{5,3}$ of the first diagonal fluctuation is calculated according to (9.178) as:

$$\vec{w}_{5,3} = B_{C-\bullet,5} B^T_{R-\bullet,3} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The stamps obtained in this way can be used to calculate the corresponding individual wavelet coefficients. Consider the last one as an example. According to (9.176), we can easily determine $c_{5,3}$

$$c_{5,3} = \langle \vec{f}, \vec{w}_{5,3} \rangle = \left\langle \begin{bmatrix} f_{0,0} & f_{0,1} & f_{0,2} & f_{0,3} & f_{0,4} & f_{0,5} \\ f_{1,0} & f_{1,1} & f_{1,2} & f_{1,3} & f_{1,4} & f_{1,5} \\ f_{2,0} & f_{2,1} & f_{2,2} & f_{2,3} & f_{2,4} & f_{2,5} \\ f_{3,0} & f_{3,1} & f_{3,2} & f_{3,3} & f_{3,4} & f_{3,5} \\ f_{4,0} & f_{4,1} & f_{4,2} & f_{4,3} & f_{4,4} & f_{4,5} \\ f_{5,0} & f_{5,1} & f_{5,2} & f_{5,3} & f_{5,4} & f_{5,5} \\ f_{6,0} & f_{6,1} & f_{6,2} & f_{6,3} & f_{6,4} & f_{6,5} \\ f_{7,0} & f_{7,1} & f_{7,2} & f_{7,3} & f_{7,4} & f_{7,5} \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & -1/2 & 0 & 0 & 0 & 0 \\ -1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \right\rangle$$

Remember, that the latter product operation is not a matrix multiplication. The scalar product for matrices is defined as the sum of elementwise products.

Therefore:

$$c_{5,3} = \frac{1}{2} f_{2,0} - \frac{1}{2} f_{2,1} - \frac{1}{2} f_{3,0} + \frac{1}{2} f_{3,1}$$

9.10 Applications

Enough theory. Let's take a look at what we can do with wavelets. We'll consider several application examples.

- Signal compression
- Denoising

In any of these application areas, we will start with a one-dimensional example and then proceed to a two-dimensional example.

9.10.1 Signal compression

Before diving in the applications, we need you to remember a well known theorem from *information theory*.

> **Shannon's source coding theorem for finite-length discrete-time signals**
> A signal $\vec{f}$ consisting of N samples, quantized into levels x_i requires at least $N \cdot H(\vec{f})$ bits to be encoded without losing information.

In this theorem, $H(\vec{f})$ denotes the entropy of the signal. It is a measure for the average amount of information that is present in a sample of the signal.

In fact, the entropy is a lower boundary for the number of bits that we need to encode the signal without losing information. We can prove that we can find coding schemes that come arbitrarily close to that limit. However, we will employ a conservative estimate and assume we need half a bit more.

Maybe, you don't remember the definition of *signal entropy*. Let's activate your memory. Information theory defines the *entropy* of a quantized time-limited discrete-time signal. It is expressed in number of bits per sample (or bits per point). It is based on the relative occurrence frequency of any quantized value of a signal throughout the duration of the signal. If a quantized value x_i occurs n times on a total of N samples, than its relative occurrence frequency equals n/N. In fact, this is an estimate of the chance that one obtains the value x_i when one blindly picks a single sample from the entire signal. Let's denote this relative occurrence frequency by $p(x_i)$. Logically, it is a value between 0 and 1.

> **Definition: signal entropy**
> The entropy H of a discrete-time signal $\vec{f}$, quantized into levels x_i, with relative occurrence frequencies denoted by $p(x_i)$ equals:
>
> $$H(\vec{f}) = \sum_i p(x_i) \log_2 \left(\frac{1}{p(x_i)} \right)$$

In case the quantized value x_i does not occur, i.e. $p(x_i) = 0$, then we take $p(x_i) \log_2 \left(\frac{1}{p(x_i)} \right)$ to be 0. Equally good: let the index of the summation in the equation above only loop over values of x_i that are actually used in the signal.

Exercises

Exercise 9.10.1-1:

1. Calculate the entropy of the following signal:

$$\vec{f} = \begin{bmatrix} 0, & 1, & 2, & 3, & 4, & 5, & 6, & 7 \end{bmatrix}$$

   Is the result what you expected?
2. How many bits do you need to encode this signal?

Exercise 9.10.1-2:

1. Calculate the entropy of the following signal:

$$\vec{f} = \begin{bmatrix}0, & 0, & 0, & 0, & 0, & 0, & 0, & 0\end{bmatrix}$$

   Is the result what you expected?
2. How many bits do you need to encode this signal?

Exercise 9.10.1-3:

1. Calculate the entropy of the following signal:

$$\vec{f} = \begin{bmatrix}0, & 1, & 1, & 2, & 2, & 2, & 2, & 3\end{bmatrix}$$

2. How many bits do you need to encode this signal?

Exercise 9.10.1-4:

1. Calculate the entropy of the following signal:

$$\vec{f} = \begin{bmatrix}0, & -10, & -10, & 20, & 20, & 20, & 20, & -99\end{bmatrix}$$

   Compare your result to the result of the previous exercise. Did you expect this?
2. How many bits do you need to encode this signal?

One-dimensional: compressing Brother John's tune

We started this chapter with the well known tune of Brother John (see page 216).

The eight-second fragment has been recorded in mono using standard compact-disc quality, i.e. using a word length W of 16 bpp[15], and at a rate $f_s = 44.1\,\text{kHz}$.

Calculating the memory requirements S for the uncompressed fragment (of length $L = 8\,\text{s}$) should be a piece of cake to you:

$$S = L \cdot W \cdot f_s = 8\,\text{s} \cdot 16\,\text{bpp} \cdot 44.1\,\text{kHz} = 5.644\,8\,\text{Mbit}$$

corresponding to a bit-rate of 705.6 kbit/s. Compare this to the bit-rate of the songs you probably carry with you on your MP3 player? Probably these will be in the order of 128 kbit/s. That doesn't compare favorably for our Brother John fragment.

We will consider two options for compressing our fragment:

- lossless compression, using optimal coding[16]
- lossy compression, based on using wavelets

[15] The unit bpp indicates bit per point, and is a unit that can be used for one-dimensional signals, in which case the point is a sample, as well as for two-dimensional signals, in which case the point is most often called a pixel.

[16] There are more more options to perform lossless compression than using (optimal) coding schemes. There are even algorithms that use wavelets. We will not treat them here.

Lossless compression

The entropy of the audio fragment has been calculated to be $H(\vec{f}) = 10.93$ bit per sample. With a reasonable effort a coding scheme can be composed that realizes $10.93 + 0.5 = 11.43$ bit per sample. This allows for a significant reduction in the amount of data we need to store/transmit.

$$S_S = L \cdot W_{LLC} \cdot f_s = 8\,\text{s} \cdot 11.43\,\text{bpp} \cdot 44.1\,\text{kHz} = 4.032\,5\,\text{Mbit}$$

To be complete, we need to take into account that we need to transmit the chosen symbol alphabet of the coding scheme as well. This takes the amount of symbol codes times the number of bits per symbol used for the quantization. A conservative estimate assumes that all of the quantization codes are present in the alphabet. Therefore

$$S_A = 2^W \cdot W = 65536 \cdot 16\,\text{bpp} = 1.049\,\text{Mbit}$$

This is quite signficant when compared to the size of the music fragment itself. However, for practical reasons, we've chosen the music fragment (unrealistically) short. Because the alphabet size does not grow with the size of the music fragment, we'll assume this alphabet size to be neglectable. In short:

$$S_{S,LLC} = S_S + S_A \approx S_S = 4.032\,5\,\text{Mbit}$$

This corresponds to a compression ratio of 1.4:1. The compression ratio is often referred to as *the coding gain*:

$$G_C = \frac{S}{S_{S,LLC}}$$

Lossy compression

Let's apply an orthogonal Daubechies-15 wavelet transformation to the music fragment. The basic idea is that most of the energy will end up in the trend coefficients. Hence we can consider many of the fluctuation coefficients to be zero, without losing a lot of energy. Not having to store these coefficients will save space.

Indeed, performing a 1-level Daubechies-15 wavelet transform, pushes 99.9975% of the energy in to the trend coefficients. Therefore, an approximation based on just the trend-coefficients, might be worth considering. It offers a 2:1 compression ratio, without taking into account the possible gain by completing our technique by another lossless compression step using optimal coding. We will denote the compression ratio without coding gain by $R_{C,net}$ and refer to it as the *net compression ratio*, and denote the compression ratio with coding gain by $R_{C,brute}$ and refer to it as the *brute compression ratio*. It turns out that the difference for a 1-level Daubechies-15 transform is almost inaudible.

The table below shows the result of trying this idea for increased levels of the Daubechies-15 wavelet transform. The table lists the level employed, the energy retained in the trend coefficients, the number of trend coefficients, the compression ratio without coding gain ($R_{C,net}$), and the compression ratio taking the coding gain into account ($R_{C,brute}$). If you are reading an electronic hyperlinked version of this text, you can click on the speaker symbols to listen to the corresponding audio fragment.

Level	Preserved Energy	Preserved c_i	$R_{C,net}$	$R_{C,brute}$	Audio
0	100.0000%	352 800	1:1	1.4:1	🔊
1	99.9975%	176 500	2:1	2.8:1	🔊
2	99.9867%	88 200	4:1	5.6:1	🔊
3	99.9051%	44 100	8:1	11.2:1	🔊
4	98.5418%	22 050	16:1	22.4:1	🔊
5	89.8097%	11 025	32:1	44.8:1	🔊

As you probably can hear, the correspondence between the original and levels 1 and 2 is pretty good. One starts to hear a significant loss of higher frequencies on level 3. As a rule of thumb, we need to retain at least 99.99% of the energy to avoid too much audible loss.

A brute compression ratio of 5.6:1 for a 2-level wavelet trend signal approximation is not bad, but a careful listener will hear some defects.

Can we do better than this? Actually, we can. It is not such a good idea to reject all fluctuation coefficients. There might be valuable information in them. If there are large fluctuation values, then that is an indication of the inadequacy of our scaling signals: they were not able to capture the true nature of our signal.

In addition, the options on the achieved compression with the technique above are rather limited. We'd prefer a technique that allows controlling the amount of preserved energy to e.g., a guaranteed 99.99%, while not being stuck at the first wavelet level, only achieving a brute compression ratio of 2.8.

The technique that we propose, is to order all fluctuation values from small to large. Then we can incrementally zero the fluctuation values, starting with the smallest until 0.01% of the energy is lost. In this way, we hope to be able to increase the compression ratio further, while not ending up with a poor-quality approximation.

We only need to store/transmit the non-zero fluctuation values that we retained at the end in combination with the position they occur at. The position information is stored in a so-called *significance map*. This is a bit-map, that contains a one at the position where a non-zero fluctuation value occurs, and a zero otherwise. As we expect many subsequent fluctuation values to be zero, we expect large strings of zeros in the bit-map. This makes it an excellent choice for (lossless) run-length encoding. We can safely assume that the overhead imposed by the significance map is neglectable, when compared with the rest of the data.

We applied this technique to improve on the earlier results. The fruits of our effort can be found in the table below. The table lists the level employed, the energy retained in the coefficients, the number of retained coefficients, the compression ratio without coding gain ($R_{C,net}$), and the compression ratio taking the coding gain into account ($R_{C,brute}$). If you are reading an electronic hyperlinked version of this text, you can click on the speaker symbols to listen to the corresponding audio fragment.

Level	Preserved Energy	Preserved c_i	$R_{C,net}$	$R_{C,brute}$	Audio
0	100.0000%	352 768	1:1	1.4:1	🔊
1	99.9975%	176 356	2:1	2.8:1	🔊
2	99.9900%	88 432	3.99:1	5.59:1	🔊
3	99.9900%	61 833	5.71:1	7.99:1	🔊
4	99.9900%	56 742	6.22:1	8.70:1	🔊
5	99.9900%	54 814	6.44:1	9.01:1	🔊
6	99.9900%	53 914	6.54:1	9.16:1	🔊
7	99.9900%	53 447	6.60:1	9.24:1	🔊
8	99.9900%	52 130	6.77:1	9.49:1	🔊
9	99.9900%	51 265	6.88:1	9.63:1	🔊

The result is quite impressive. We achieved almost a 10:1 compression ratio! However, the approach is clearly running out of steam. One might consider using a wavelet packet transform to go one step further.

Remarks

- Many more strategies with better sophistication exist. E.g., one could choose to quantize the smaller fluctuation coefficients with less bits than the bigger trend coefficients. The best strategies even take into account psycho-acoustics science to be able to go beyond the 10:1 compression ratio. We will not treat these strategies here. This could be the subject of a course on itself, on data (or music) compression.
- If you analyze the tables above, you will observe that the ratio between $R_{C,brute}$ and $R_{C,net}$ seems to be independent of the wavelet compression level. Though it is not exactly a constant, it is true that wavelet compression hardly influences the entropy of the signal. Therefore the coding gain is (almost) constant.

Exercises

Exercise 9.10.1-5: Select a 10 s music fragment of your favorite song and try to compress it using Coifman wavelets. To this end, write a MATLAB function that takes the filename of the input file, the filename of the output file and the amount of error allowed as input parameters. Make the function print basic results, like:

- the amount of coefficients that remain
- the obtained net compression ratio

Exercise 9.10.1-6: Write a MATLAB function to calculate the entropy of a one-dimensional signal.
Try the `entropy` function of the MATLAB Image Processing toolbox. Does it produce correct results? Why not?

Exercise 9.10.1-7: Extend your algorithm of the first exercise to wavelet packet transforms.

Exercise 9.10.1-8: Extend your algorithm of the first exercise in such a way that also small trend coefficients can be zeroed. Is this effective? Can you obtain better results than without zeroing the trend coefficients?

Two-dimensional: compressing the image of an Italian monument

Consider the 800×512-pixel image of the Italian monument of page 292. We will again try two strategies after applying the wavelet transform:

1. Zeroing all fluctuation coefficients
2. Zeroing the fluctuation coefficients selectively

Again, as a measure for the error introduced by the approximation, we will use the relative amount of preserved energy. However, there is a second measure that is very common when considering approximation errors in images, i.e. the so-called *Peak Signal to Noise Ratio* (*PSNR*). It is defined by:

$$PSNR = 10 \log_{10} \left(\frac{(2^L - 1)^2}{MSE} \right)$$

with L the number of bits that has been used to quantize the image and MSE the so-called *Mean Square Error*. Given an $M \times N$ image $\vec{f}$ and its approximation $\vec{f}_a$ it is defined by:

$$MSE = \frac{\left\| \vec{f}_a - \vec{f} \right\|_F^2}{M \cdot N}$$

i.e., it is the squared distance between the two images, divided by the total number of pixels. It is easy to see that, if one neglects the effects of quantization, this corresponds to the average amount of energy lost per pixel because of the approximation.

It is commonly accepted that a compression that achieves a $PSNR$ of 40 dB, can be considered to be perceptually lossless.

Lossless compression

The entropy of the image has been calculated to be $H(\vec{f}) = 7.42$ bit per sample. It is fair to say that we need 8 bit to encode this image without losing detail. Therefore, no lossless image compression is possible. The coding gain equals 1.

We need to store/transmit S_s bit, with

$$S_{S,LLC} = L \cdot M \cdot N = 8\,\text{bit} \cdot 512 \cdot 800 = 3.276\,8\,\text{Mbit}$$

To be complete, we need to take into account that we need to transmit the chosen symbol alphabet of the coding scheme as well. As we won't use any special coding scheme, it does not need to be transmitted as well.

Lossy compression

Let's apply an orthogonal Daubechies-15 wavelet transformation to our test image. We start by compressing the image for increasing wavelet decomposition levels, zeroing all fluctuation coefficients. The results can be found in the table below.

Level	Preserved Energy	$PSNR$[dB]	Preserved c_i	$R_{C,net} = R_{C,brute}$	Refer to Figure 9.28
0	100.00%	$+\infty$	409 600	1:1	(a)
1	99.67%	26.90	204 800	2:1	(b)
2	99.22%	24.86	102 400	4:1	(c)
3	99.48%	22.11	51 200	8:1	(d)
4	97.34%	21.11	25 600	16:1	(e)
5	95.06%	17.33	12 800	32:1	(f)
6	91.83%	16.69	6 400	64:1	(g)
7	87.67%	14.69	3 200	128:1	(h)

If you check the resulting images, up until the 2 : 1 compression level, the results are almost not perceivable. From compression level 4 : 1 and up, undulations start to corrupt the image to a level that all detail in the image is lost.

Can we do better than this? Actually, we can. We're going to apply the very same technique as we did when compressing the music fragment of Brother John before. We will order all fluctuation coefficients and start zeroing them in order of increasing importance until a given error level is reached. Let's set the error level to preserving at least 99.9% of the energy. This is an error parameter that is easily controlled and usually corresponds to a *PSNR* of about 32 dB.

The results have been summarized in the table below.

Level	Preserved Energy	$PSNR$[dB]	Preserved c_i	$R_{C,net} = R_{C,brute}$	Refer to Figure 9.29
0	100.00%	$+\infty$	409 600	1:1	(a)
1	99.90%	32.26	116 489	3.52:1	(b)
2	99.90%	32.95	62 467	6.56:1	(c)
3	99.90%	32.12	53 992	7.59:1	(d)
4	99.90%	32.18	53 074	7.34:1	(e)
5	99.90%	32.14	53 017	7.34:1	(f)
6	99.90%	32.14	53 017	7.34:1	(g)
7	99.90%	32.14	53 017	7.34:1	(h)

It's obvious that the selected approach is running out of steam for levels higher than 4. However, the achieved compression ratio is a nice result.

Remarks

- Many more strategies with better sophistication exist. E.g., J2K (the new JPEG still image compression standard) achieves a compression ratio of about 16:1 for *PSNR*-levels of about 32 dB. It is based on wavelets.
- Even if there is margin for lossless compression, one will see that the ratio between $R_{C,brute}$ and $R_{C,net}$ is almost independent of the wavelet compression level. Though it is not exactly a

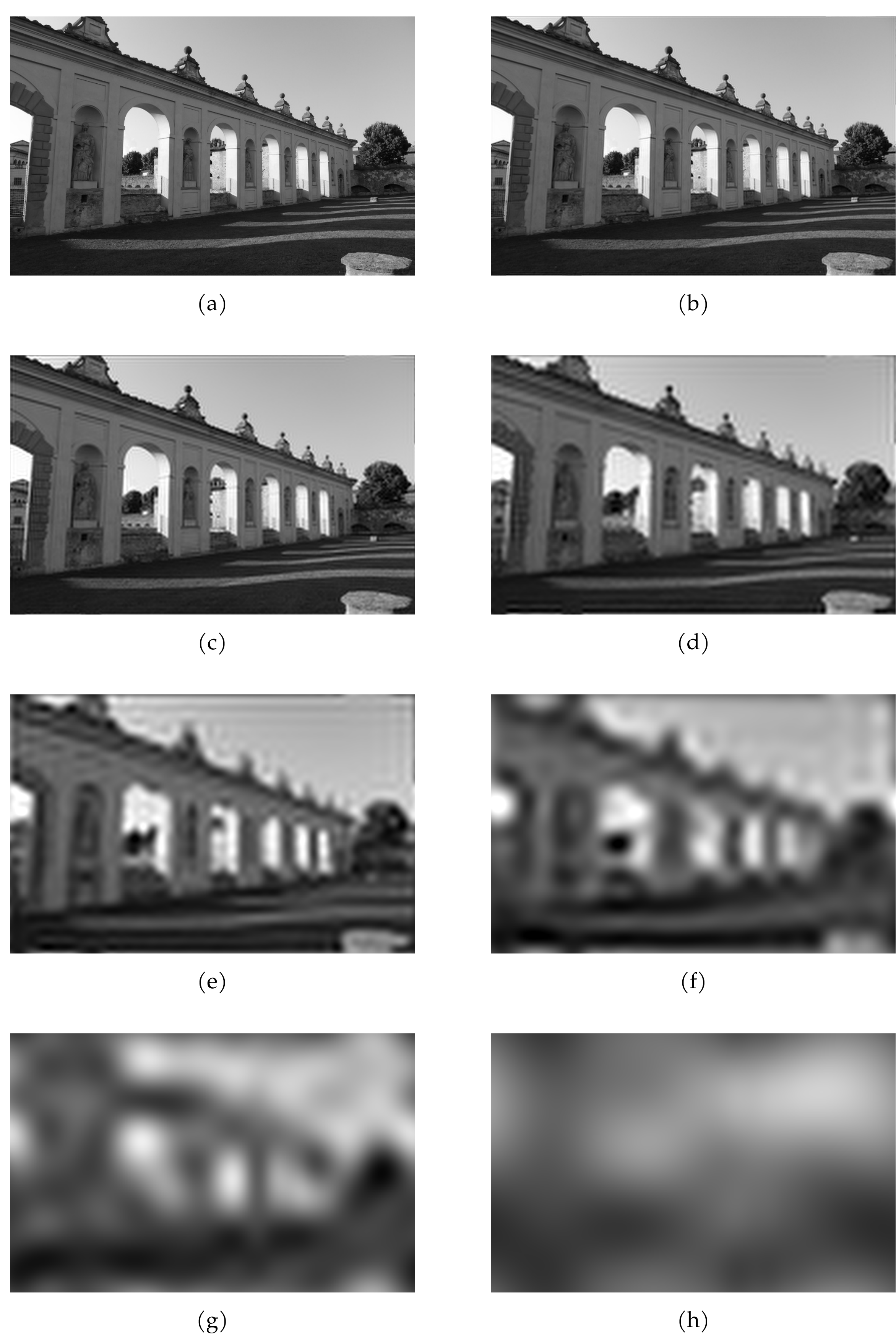
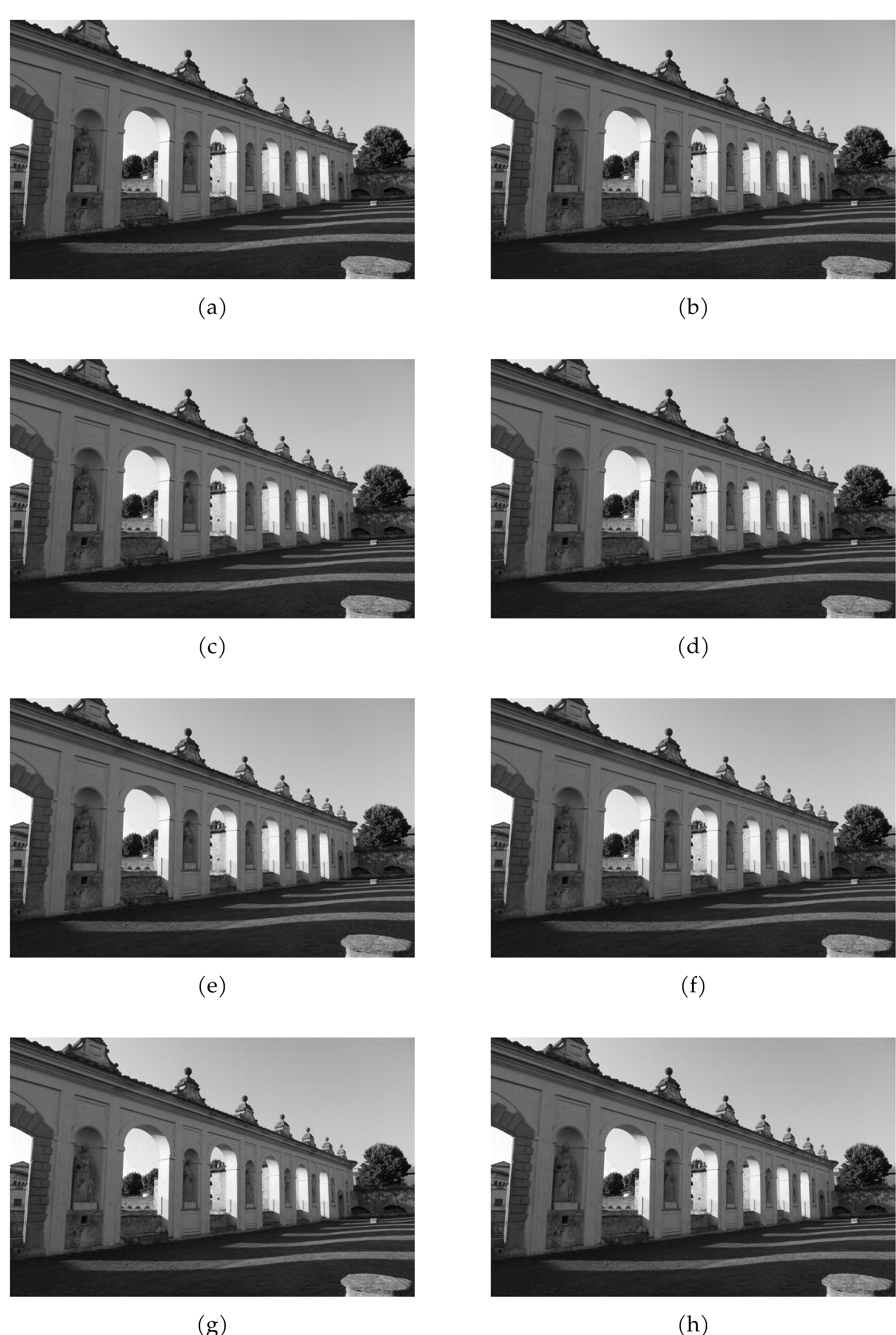


Figure 9.28: (a) Image (8-bit, 800 × 512 pixels) of an Italian momument, and its compressed versions, using the trend coefficients of a Daubechies-15 wavelet transform, for levels (b) 1, (c) 2, (d) 3, (e) 4, (f) 5, (g) 6, (h) 7



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 9.29: (a) Image (8-bit, 800 × 512 pixels) of an Italian momument, and its compressed versions, using the technique of zeroing fluctuation coefficients, starting from a Daubechies-15 wavelet transform, for levels (b) 1, (c) 2, (d) 3, (e) 4, (f) 5, (g) 6, (h) 7

constant, it is true that wavelet compression hardly influences the entropy of the signal. Therefore the coding gain is (almost) constant.

Exercises

Exercise 9.10.1-9: Select an image of your own personal photo collection and try to compress it using Coifman wavelets. To this end, write a MATLAB function that takes the filename of the input file, the filename of the output file and the amount of error allowed as input parameters. Make the function print basic results, like:

- the *PSNR*
- the amount of coefficients that remain
- the obtained net compression ratio

Exercise 9.10.1-10: Look for a MATLAB function to calculate the entropy of an image. Use the function to extend the MATLAB function you wrote for the previous exercise to calculate and print the entropy of the compressed images.

Exercise 9.10.1-11: Extend your algorithm of the first exercise towards wavelet packet transforms.

Exercise 9.10.1-12: Extend your algorithm of the first exercise in such a way that also small trend coefficients can be zeroed. Is this effective? Can you obtain better results than without zeroing the trend coefficients?

9.10.2 Denoising

Noise is omnipresent in our daily engineering life. Noise is present in the processes that we'd like to measure: small fluctuations that originate from the fact that we're not able to build systems that are totally isolated from their surroundings. These surroundings influence these processes. Noise is generated in sensors and by analog electronic components whether they are passive (resistors) or active (transistors). Noise is generated by sampling and quantizing signals, and by performing calculations followed by a requantization.

Luckily, in most cases the noise is differently correlated compared to the signals we're interested in.

Now, if there is one feature that wavelets are good at, it is decorrelating data. In fact, wavelets are conceived around the notions of *trend* and *fluctuation*.

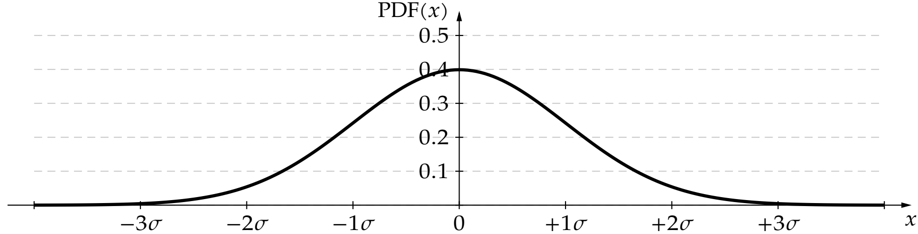
White noise and wavelets A most common form of noise, is white noise. The essence of white noise is that it is totally uncorrelated to itself, i.e. the autocorrelation is (almost) a delta function peaking at $n = 0$.

Very often white noise has a Gaussian magnitude distribution. One can prove that whenever a noise source originates from many different, independent contributions, its magnitude distribution PDF(x) is (near) Gaussian and has a zero mean:

$$\mathrm{PDF}(x) = \frac{1}{\sigma\sqrt{2\pi}}\,\mathrm{e}^{-\frac{x^2}{2\sigma^2}}$$

The parameter σ^2 is the so-called *variance* of the noise source, its square root σ is often referred to as the *standard deviation* or *RMS-value*.

For your convenience, we plotted this magnitude distribution below.



It is well known, that the area below the curve in the interval $[-\sigma, +\sigma]$ accounts for 68.2% of the total area. Enlarging the interval to $[-2\sigma, +2\sigma]$, accounts for 95.4% of the total area. Enlarging even further to $[-3\sigma, +3\sigma]$ covers 99.7% of the area, and to $[-4\sigma, +4\sigma]$ covers 99.99% of the area.

The nice thing about the combination of wavelets and white noise, is that the wavelet spectrum of white noise is flat. In addition, for Gaussian white noise, the wavelet coefficient magnitude distribution is also Gaussian and has the same variance.

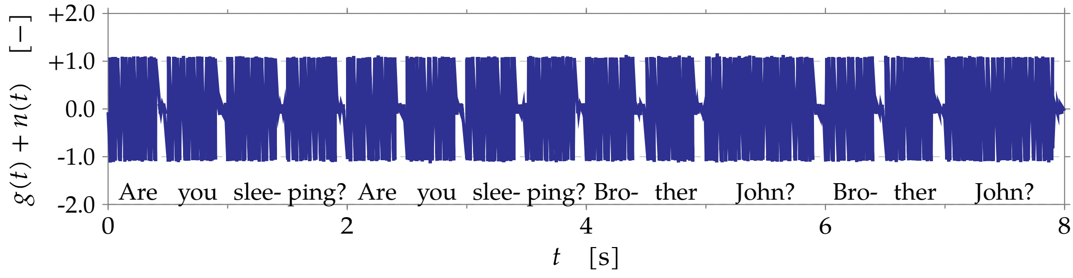
Add to this the ability of the wavelet transform to push most of the energy into the trend coefficients, and the discriminative power of the wavelet transform is very clear.

The idea is to set wavelet coefficients in the interval $[-4\sigma, +4\sigma]$ to zero. In this way we will have killed almost all of the noise.

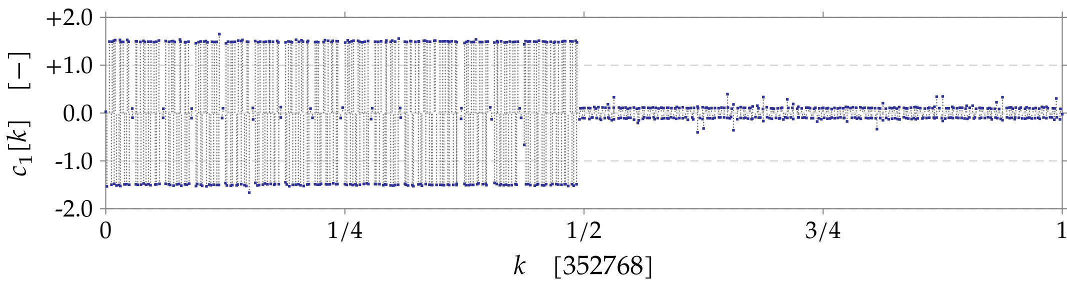
One-dimensional: denoising Brother John's tune

We will once again take the well known tune of Brother John (see page 216) as a test vehicle.

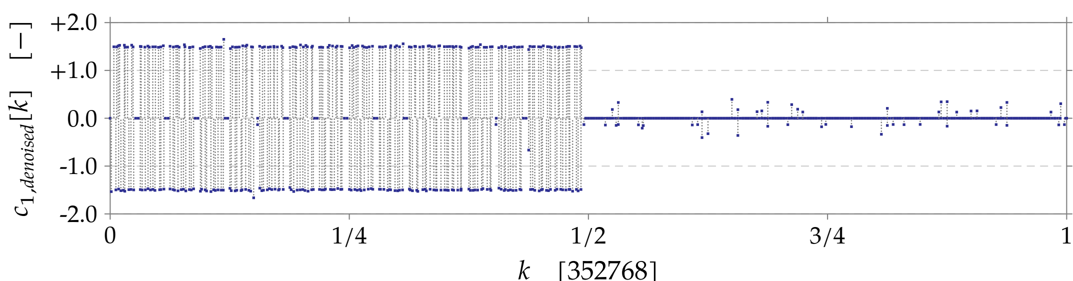
The eight-second fragment has been recorded in mono using standard compact-disc quality, but this time, we corrupted the fragment for test purposes with white Gaussian noise with a standard deviation of 5% of the RMS value of the fragment without noise.



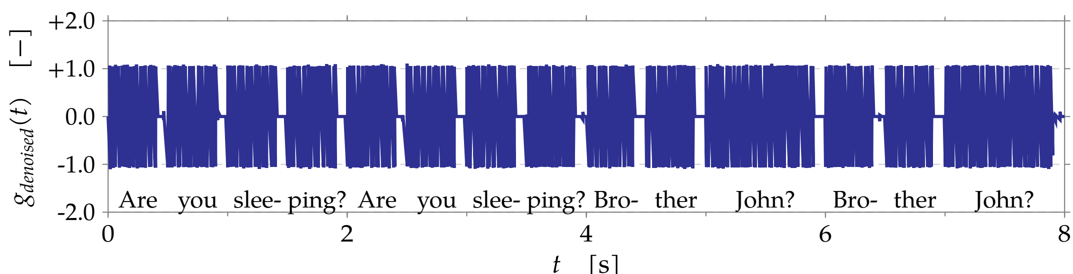
Then we run the fragment throuh a Daub-12 wavelet transform, with the following result:



Now, let's set any coefficient whose absolute value is smaller than 4σ to zero. This results in:



Then we recompose the signal. This results in the signal below.



As one can see, the noise in the pauses in between the different notes is almost totally gone. If you are reading an electronic hyperlinked version of this text, you can click on the graphs above to listen to the corresponding audio fragments.

Exercises

Exercise 9.10.2-1: Select a 10 s music fragment of your favorite song and add some white Gaussian noise to it (using the `randn` function of matlab). Try to denoise it by transforming your fragment using a Daub-15 wavelet transform, and zeroing all coefficients with an absolute value below 4σ. Are you pleased with the result?

Exercise 9.10.2-2: Parameterize your script above to generate a plot that displays the distance between the signal without noise and the denoised signal as a function of the zeroing treshold you use. Is your threshold choice a good one? Can you improve?

Exercise 9.10.2-3: Experiment with using higher-level transforms for denoising.

Two-dimensional: denoising the image of an Italian monument

We can apply the same principles for two-dimensional images. Consider Figure 9.30(a). The original picture of the Italian monument (subfigure (a)) has been corrupted with white Gaussian noise with a standard deviation value σ of 10% of the RMS value of the original image (see subfigure (b)), resulting in the image in subfigure (c). Denoising the image using a Daub-15 1-level wavelet transform and zeroing the coefficients smaller than 4σ, results in subfigure (d). The noise level is reduced considerably. However, the noise is still noticeable. Our vision is very susceptible for white noise on images. It is very hard to completely remove this kind of noise. An order-statistical filter might yield better results in this case.

Exercises

Exercise 9.10.2-4: Select an image of your own personal photo collection, add some noise to it and try to denoise it using Daubechies wavelets.

Exercise 9.10.2-5: Experiment with higher-level wavelet transforms to get better denoising results.

Exercise 9.10.2-6: Try scanning a photograph using a document scanner, or a slide scanner. Usually this results in quite some white noise. Denoise the scanned images.

Exercise 9.10.2-7: Try adding some salt-and-pepper noise to the image and check how the wavelet denoising performs.

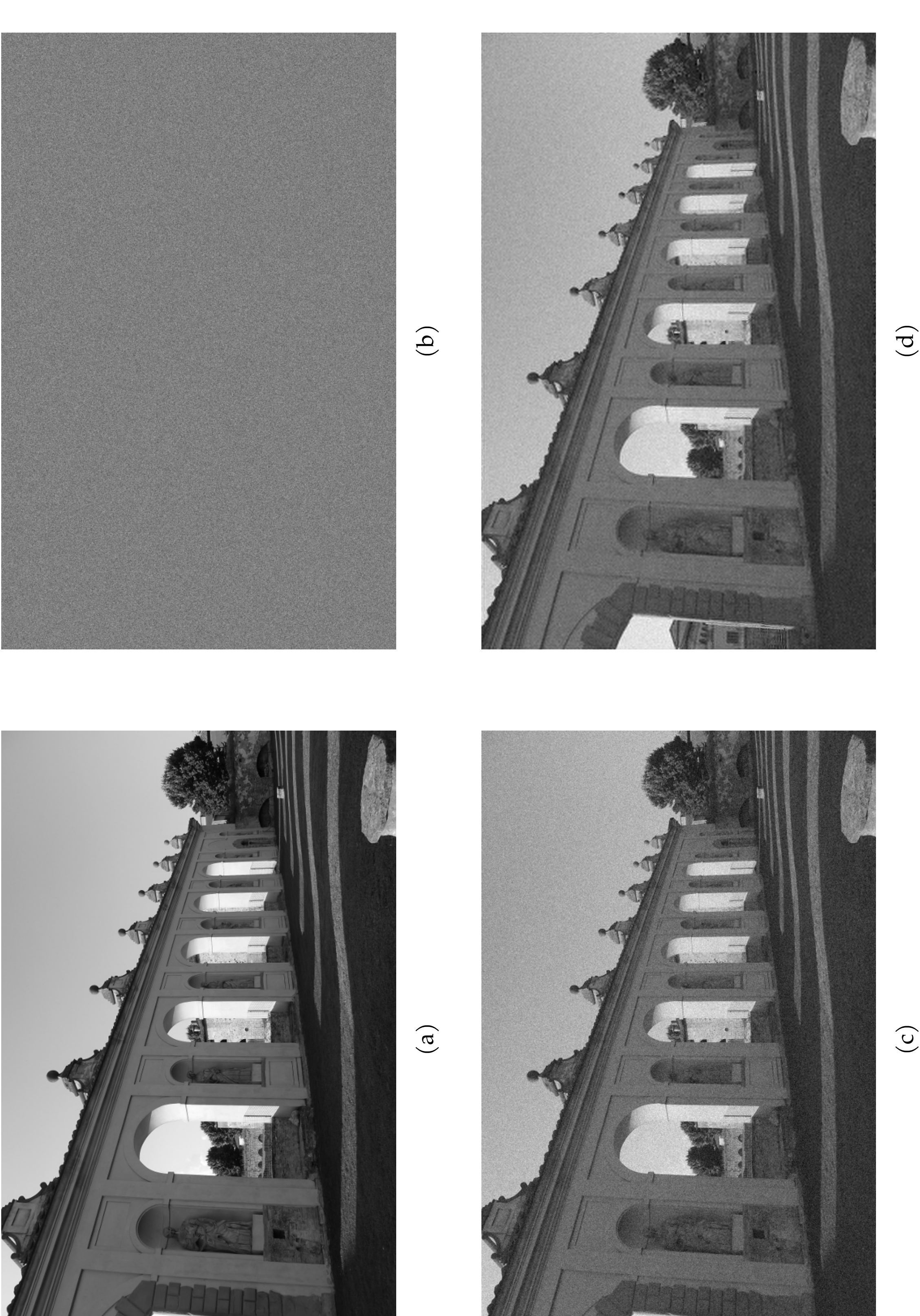


Figure 9.30: Image of (a) an Italian monument, (b) the white Gaussian noise that was added to obtain (c) a noisy picture. After denoising using the zeroing of wavelet coefficients the result of (d) was obtained.

Part III

Capita Selecta

CIC Filters

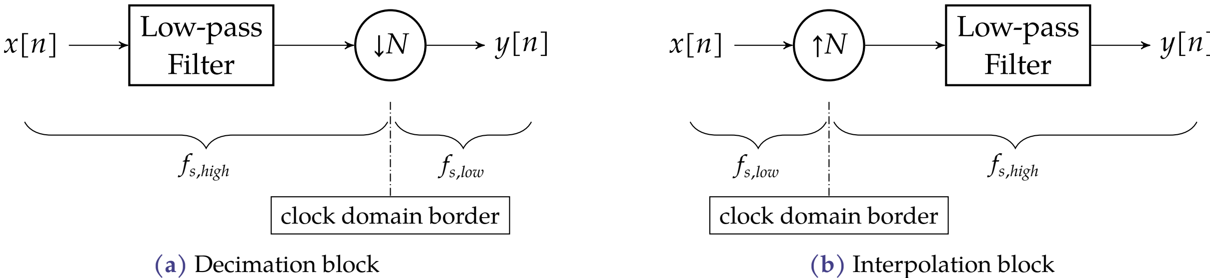
Systems in which the same sampling frequency is used throughout the entire application are rare. In some parts of the system a high sampling frequency is desirable. E.g. to relax (analog) sampling or reconstruction filter specifications, or to allow room for noise shaping in Delta-Sigma modulators. However, a high sampling frequency comes at a cost: the increased data rate requires extra compute power. This has a direct translation into an increased system power consumption and production cost.

The obvious solution to saving the environment[1] and our wallet is to only use high sampling rates in the crucial portions of the system. As soon as our signal has passed the point where a high rate is no longer required, we will decimate it, and only when an increased sampling frequency is crucial again, we will apply interpolation to get it 'up to speed'.

10.1 Decimation and interpolation

We have studied interpolation and decimation in the introductory course on signal processing. There you've learnt that low-pass filtering is crucial to prevent or remove aliases in the case of decimation and interpolation filters respectively. The typical setups for these rate change blocks can be found in Fig. 10.1.

The rate change operations in these blocks are cheap. They boil down to throwing away samples in the decimation case, and inserting zeros (so-called *zero stuffing*) in the interpolation case. The pitty of both rate-change blocks, is that the anti-alias filtering occurs at the expensive



(a) Decimation block

(b) Interpolation block

Figure 10.1: Typical setup for rate change blocks with sampling frequencies $f_{s,high} = N \cdot f_{s,low}$

[1]Remember: an increased system power means bigger power plants and larger batteries. Both are ruining the sustainability of our planet.

side of the rate-change element. Therefore, these are expensive filters that require sufficent attention in order to keep the required computation time within acceptable bounds.

Wouldn't it be nice if we could change the order of the low-pass filter and the rate change block? Considering the up- or downsampler as the border in between two clock domains, this could be seen as *letting the filter cross the border*.

10.2 Noble identities

Normally, people only cross borders if there is a benefit to it. Exploring new cultures (e.g. on a holiday), the quest for a better life, the opportunity of trade: they are all good reasons.

In the following sections, we will see that also on the continent of DSP, crossing borders offers great opportunities.

The border we are referring to is the border in between sample clock domains. The objective should be clear by now: if we can move signal operations from the high-rate clock domain to the low-rate clock domain, we can save significantly in terms of energy and cost.

The keys to the border fence are the *Noble identities*.

First Noble identity The two systems below are equivalent:

$$x[n] \longrightarrow \boxed{H(z^N)} \xrightarrow{v_h[n]} \textcircled{\downarrow N} \longrightarrow y[n] \quad \Leftrightarrow \quad x[n] \longrightarrow \textcircled{\downarrow N} \xrightarrow{v_l[n]} \boxed{H(z)} \longrightarrow y[n]$$

Again, this means you can move a block adjacent to the clock domain boundary from the low-rate part to the high-rate part, by replacing z by z^N or vice versa.

Proof:
Let's assume causal signals and let's use our standard notation: $x[n] \overset{Z}{\rightarrow} X(z)$.

Let's assume $h_N \overset{Z}{\rightarrow} H(z^N)$. If $h[n]$ is of length $M+1$, then $h_N[n]$ is of length $N \cdot M + 1$ and

$$h_N[n] = \begin{cases} h[n/N] & \text{if } n \in N\mathbb{Z} \\ 0 & \text{otherwise} \end{cases}$$

We can write for the left block diagram:

$$\begin{aligned} v_h[n] &= \sum_{i=0}^{+N\cdot M} h_N[i]x[n-i] \\ &= \sum_{m=0}^{M} h[m]x[n-m\cdot N] \end{aligned}$$

We also know that:

$$y[n] = v_h[n \cdot N]$$

Therefore:

$$y[n] = \sum_{m=0}^{M} h[m]x[n \cdot N - m \cdot N] = \sum_{m=0}^{M} h[m]x[(n-m) \cdot N] \tag{10.1}$$

Similarly, we can write for the right block diagram:

$$v_l[n] = x[n \cdot N]$$

and

$$y[n] = \sum_{m=0}^{M} h[m]v_l[n-m].$$

Therefore:

$$y[n] = \sum_{m=0}^{M} h[m]x[(n-m)N] \tag{10.2}$$

It is clear that (10.1) is identical to (10.2), leading to the conclusion that both systems are equivalent. ■

Second Noble identity The two systems below are equivalent:

$$x[n] \rightarrow \boxed{H(z)} \xrightarrow{v_l[n]} \uparrow N \rightarrow y[n] \quad \Leftrightarrow \quad x[n] \rightarrow \uparrow N \xrightarrow{v_h[n]} \boxed{H(z^N)} \rightarrow y[n]$$

This means you can move a block adjacent to the clock domain boundary from the low-rate part to the high-rate part, by replacing z by z^N or vice versa.

Proof:
Let's assume causal signals such that we can safely use the one-sided Z-transform and let's use our standard notation: $x[n] \xrightarrow{Z} X(z)$.

We can write for the left block diagram:

$$y[n] = \begin{cases} v_l[n/N] & \text{if } n \in N\mathbb{Z} \\ 0 & \text{otherwise} \end{cases}$$

Elaborating on the definition of the Z-transform, we easily obtain:

$$\begin{aligned} Y(z) &= \sum_{n=0}^{+\infty} y[n]z^{-n} \\ &= \sum_{m=0}^{+\infty} v_l[m]z^{-m \cdot N} = V_l(z^N) \end{aligned}$$

Given $V_l(z) = H(z) \cdot X(z)$, this leads to:

$$Y(z) = H(z^N)X(z^N) \tag{10.3}$$

Similarly, we can write for the right block diagram:

$$v_h[n] = \begin{cases} x[n/N] & \text{if } n \in N\mathbb{Z} \\ 0 & \text{otherwise} \end{cases}$$

Elaborating on the definition of the Z-transform, we easily obtain:

$$\begin{aligned} V_h(z) &= \sum_{n=0}^{+\infty} v_h[n] z^{-n} \\ &= \sum_{m=0}^{+\infty} x[m] z^{-m \cdot N} = X(z^N) \end{aligned}$$

Given $Y(z) = H(z^N) \cdot V_h(z)$, this leads to:

$$Y(z) = H(z^N) X(z^N) \tag{10.4}$$

It is clear that (10.3) is identical to (10.4), leading to the conclusion that both systems are equivalent. ■

10.3 Rate conversion and CIC filters

10.3.1 The attractive nature of moving average filters

Moving average filters turn out to be a very attractive starting structure to implement the low-pass filters decimation or interpolation filters. In fact, this will prove to be one of these occasions in which marrying two concepts, i.e. the concept of low-pass filters for rate change operations with the concept of moving average filters proves to be sweet like chocolate and strawberries. Even more: chocolate, strawberries and champagne!

Let's investigate this further.

The term *moving average filter* could not have been chosen in a better way, as it indeed calculates a moving average of the input samples you submit to it. Assuming we average N samples, the obvious filter equation is:

$$y[n] = \frac{1}{N} \sum_{i=0}^{N-1} x[n-i]$$

where $x[n]$ and $y[n]$ denote the input and output signal respectively. To keep the formulae below simple, let's omit the scaling with $1/N$, i.e.

$$y[n] = \sum_{i=0}^{N-1} x[n-i]$$

This simplified filter has a lot of beneficial properties:

- it only requires (N) additions, no scalers
- it is a finite impulse response (FIR) filter, and is therefore guaranteed to be stable
- it has a symmetrical impulse response and is therefore a linear phase filter!

Often, this filter is implemented in a smart way, based on the observation that the value of the current output sample can be obtained by taking the previous output value, subtracting the value of the input sample that went 'out of scope' and adding the value of the newly arrived input sample:

$$y[n] = y[n-1] - x[n-N] + x[n]$$

Transforming this filter equation to the Z-domain, leads to:

$$Y(z) = z^{-1}Y(z) - z^{-N}X(z) + X(z)$$

Solving this equation for the transfer function $H(z) = Y(z)/X(z)$ yields:

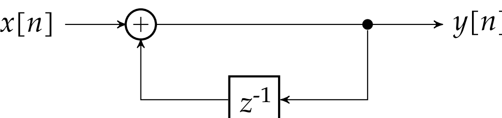
$$H(Z) = = \underbrace{\frac{1}{1 - z^{-1}}}_{\text{integrator}} \cdot \underbrace{\left(1 - z^{-N}\right)}_{\text{comb filter}}$$

The first factor of the transfer function corresponds to a *digital integrator*, the second one to a *feedforward comb filter*.

The integrator By inverse Z-transformation, the transfer function of the integrator can be easily found to be:

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

Indeed, this is the equation of a digital integrator. The current sample is added to the previous output of the filter. This filter function corresponds to the following filter block diagram:

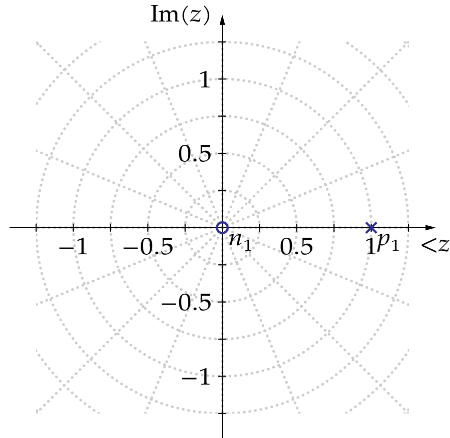


In terms of hardware requirements, the integrator is very cheap: it only requires one adder, one register and no scalers or multipliers.

In the Z-domain, the transfer function can also be written as:

$$H(z) = \frac{z}{z - 1} \tag{10.5}$$

The single pole $p_1 = 1$ and zero $n_1 = 0$ are easily determined and have been indicated on the pole-zero plot below.



By replacing $z = e^{j\omega}$ in (10.5) we can obtain the frequency response

$$\begin{aligned} H(e^{j\omega}) &= \frac{e^{j\omega}}{e^{j\omega} - 1} \\ &= \frac{e^{j\omega}}{\cos\omega - 1 + j\sin\omega} \end{aligned}$$

leading to closed form expressions for the amplitude and the phase of the transfer function in the Fourier domain:

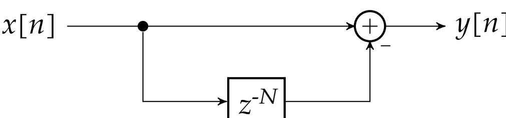
$$\begin{aligned} |H(e^{j\omega})| &= \frac{1}{\sqrt{(\cos\omega - 1)^2 + \sin^2\omega}} \\ &= \frac{1}{\sqrt{2(1 - \cos\omega)}} \end{aligned} \qquad \arg H(e^{j\omega}) = \omega - \operatorname{Arctan2}\frac{\sin\omega}{\cos\omega - 1}$$

This frequency response has been drawn in Figure 10.2a.

The comb filter By inverse Z-transformation, the transfer function of the comb filter can be easily found to be:

$$y[n] = x[n] - x[n - N]$$

This is the equation of a feed forward comb filter as can be seen on the corresponding block diagram below:



In terms of hardware requirements, the comb filter is more expensive than the integrator: instead of one register, it requires N registers.

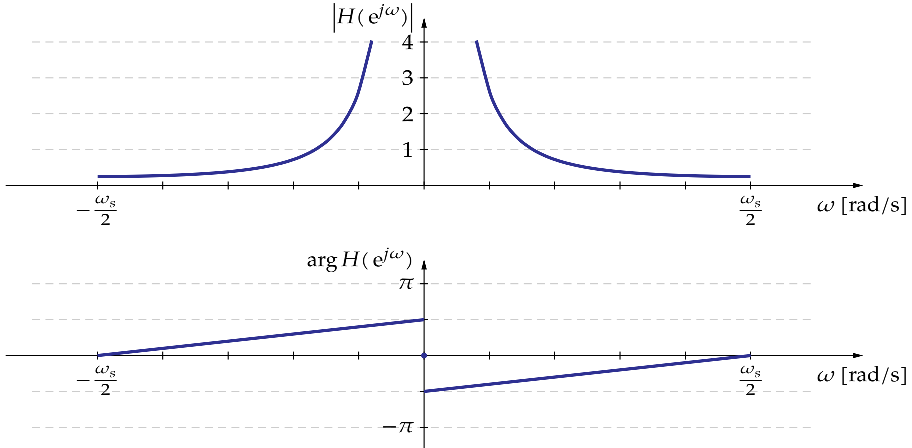
In the Z-domain, the transfer function can also be written as:

$$H(z) = \frac{z^N - 1}{z^N} \tag{10.6}$$

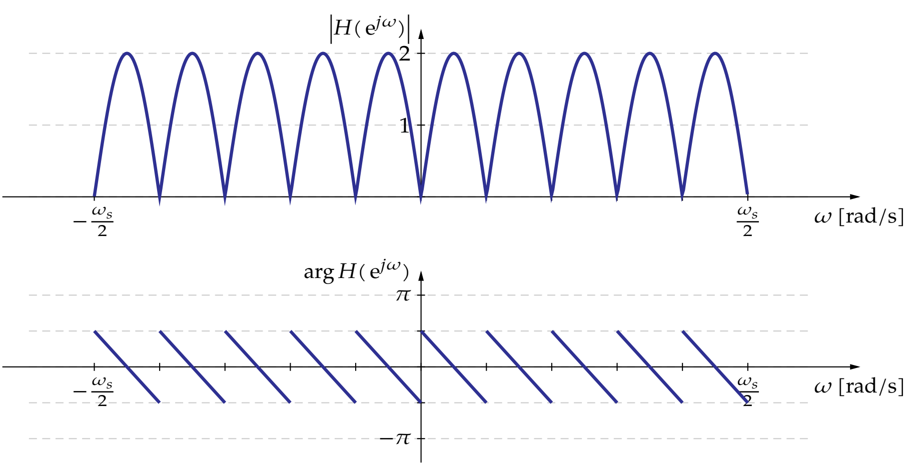
This transfer function has N zeros $n_i, i = 1 \dots N$ and as many poles $p_i, i = 1, 2, \dots, N$, with

$$\begin{aligned} n_i &= e^{j\frac{2\pi}{N}} \\ p_i &= 0 \end{aligned}$$

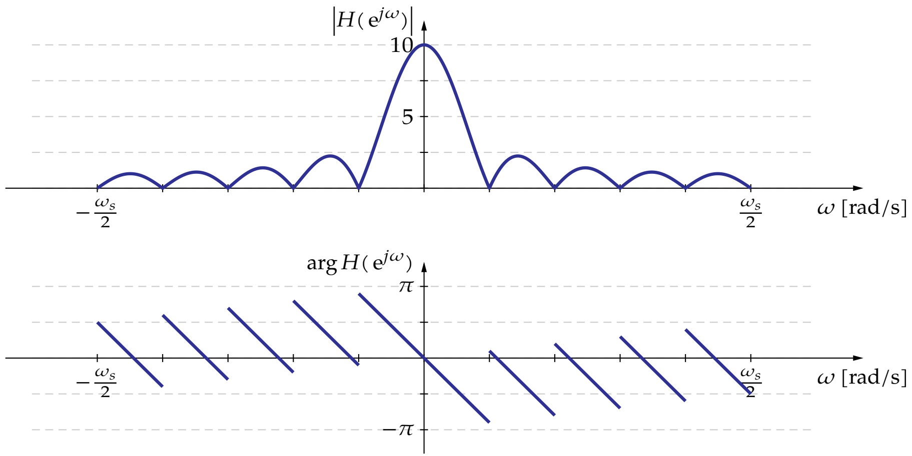
As can be seen on the pole-zero plot below (with $N = 10$), the zeros are equidistantly positioned on the unit circle.



(a) Frequency resonse of a digital integrator

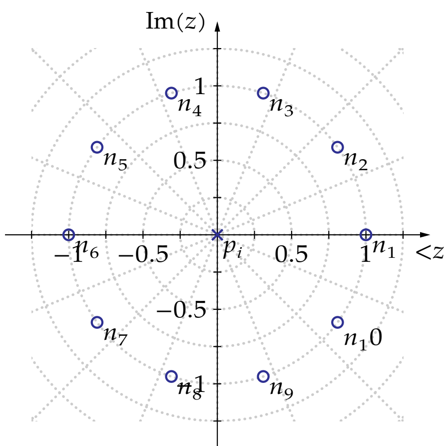


(b) Frequency resonse of the comb filter



(c) Frequency resonse of the CIC filter

Figure 10.2: Frequency response of the moving average filter (with $N = 10$ and its components



By replacing $z = \mathrm{e}^{j\omega}$ in (10.6) we can obtain the frequency response

$$\begin{aligned} H(\mathrm{e}^{j\omega}) &= \mathrm{e}^{-jN\omega}\left(\mathrm{e}^{jN\omega)} - 1\right) \\ &= \mathrm{e}^{-jN\omega}\left(\cos(N\omega) - 1 + j\sin(N\omega)\right) \end{aligned}$$

leading to closed form expressions for the amplitude and the phase of the transfer function in the Fourier domain:

$$\begin{aligned} \left|H(\mathrm{e}^{j\omega})\right| &= \sqrt{(\cos(N\omega) - 1)^2 + \sin^2(N\omega)} \\ &= \sqrt{2(1 - \cos(N\omega))} \end{aligned} \qquad \arg H(\mathrm{e}^{j\omega}) = -N\omega + \mathrm{Arctan2}\,\frac{\sin(N\omega)}{\cos(N\omega) - 1}$$

This frequency response has been drawn in Figure 10.2b.

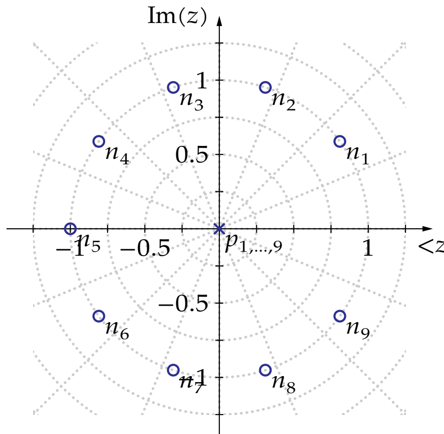
The moving average filter as an integrator-comb pair The combined transfer function equals:

$$H(z) = \frac{z}{z-1} \cdot \frac{z^N - 1}{z^N}$$

The corresponding block diagram is easily obtained as a concatenation of the integrator block and the comb filter. Note that due to the fact that this is an LTI system, we can interchange the order of the blocks.

$$x[n] \longrightarrow \boxed{\frac{z}{z-1}} \longrightarrow \boxed{\frac{z^N-1}{z^N}} \longrightarrow y[n] \quad \Leftrightarrow \quad x[n] \longrightarrow \boxed{\frac{z^N-1}{z^N}} \longrightarrow \boxed{\frac{z}{z-1}} \longrightarrow y[n]$$

The pole-zero plot is easily composed as the superposition of the pole-zero plots of its components. The pole at $z = 1$ of the integrator is canceled by a zero of the comb filter. Likewise at $z = 0$, there is also a canceling pole-zero pair. Only nine zeros remain.



The frequency resonnse can be most easily derived starting from the non-recursive transfer function:

$$
\begin{aligned}
H(z) &= \sum_{n=0}^{N-1} z^{-n} \\
&\Big\downarrow \; z = \mathrm{e}^{j\omega} \\
H(\mathrm{e}^{j\omega}) &= \sum_{n=0}^{N-1} \mathrm{e}^{-jn\omega} \\
&\Big\downarrow \; \textstyle\sum_{i=n}^{N-1} q^n = \frac{1-q^N}{1-q} \\
&= \frac{1-\mathrm{e}^{-jN\omega}}{1-\mathrm{e}^{-j\omega}} \\
&= \frac{\mathrm{e}^{-j\frac{N\omega}{2}}}{\mathrm{e}^{-j\frac{\omega}{2}}} \cdot \frac{\mathrm{e}^{j\frac{N\omega}{2}} - \mathrm{e}^{-j\frac{N\omega}{2}}}{\mathrm{e}^{j\frac{\omega}{2}} - \mathrm{e}^{-j\frac{\omega}{2}}} \\
&\Big\downarrow \; \sin\alpha = \frac{\mathrm{e}^{j\alpha} - \mathrm{e}^{-j\alpha}}{2j} \\
&= \mathrm{e}^{-j\frac{(N-1)\omega}{2}} \cdot \frac{\sin\left(\frac{N\omega}{2}\right)}{\sin\left(\frac{\omega}{2}\right)}
\end{aligned}
$$

The corresponding frequency response has been depicted in Figure 10.2c. Taking a look at the magnitude graph, you will agree that this filter is kind of a crude low-pass filter. The pass band (that we will assume to run from $\omega = 0 to \omega = \omega_S 2N$) is certainly not flat, there's quite a transition band and the stop band is all but blocking everything. However, it's definitely cheap. Moreover: it is a linear-phase filter!

The cascaded integrator comb (CIC) filter Now, let's apply this filter multiple times, i.e. let's make a cascade of moving average filters. In this way we hope to improve the ability of the filter to block frequencies in the stop band.

If we represent the integrators by I-blocks and the comb-filters by C_N-blocks, we obtain the following block diagram:

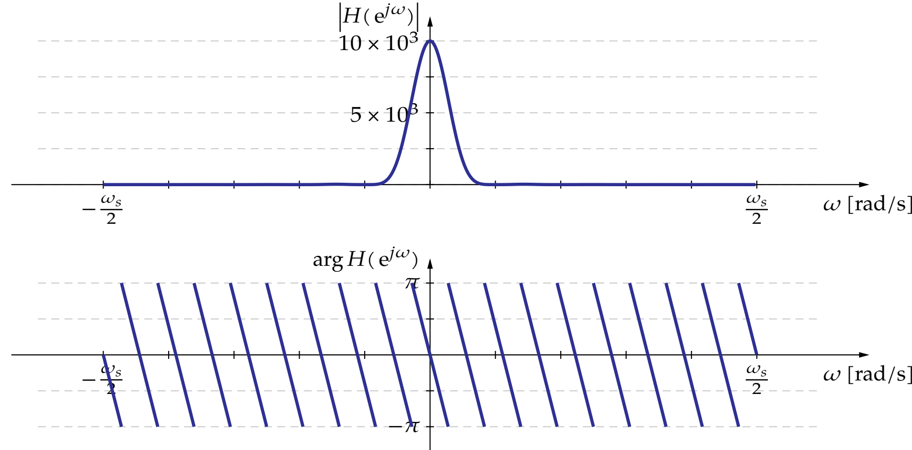
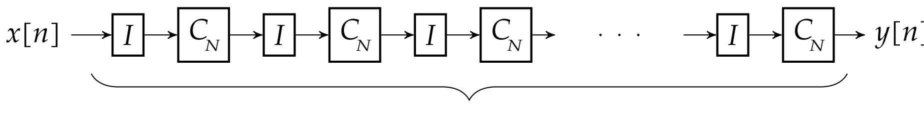
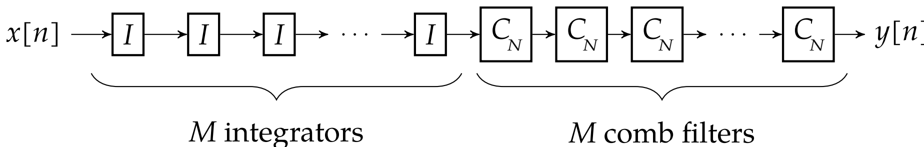


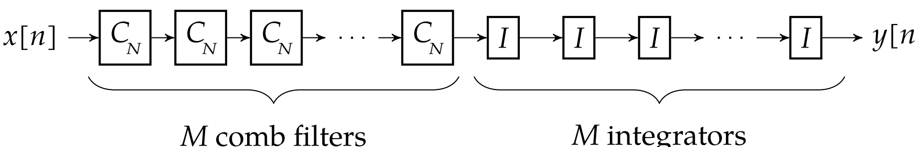
Figure 10.3: Frequency response of a cascaded integrator comb filter (with $M = 4$)



Because all components in the cascade are LTI systems, we can safely rearrange their order. E.g.,



or equally good:



For all these arrangements, the frequency response is identical. You can find a graph for the case $M = 4$ in Figure 10.3. As you can see, even a limited number of moving average filters gathered in the cascade yields a decent stop-band suppresion. However, there's also quite some pass-band distorsion. We will compensate for this afterwards using a *pass-band correction filter*.

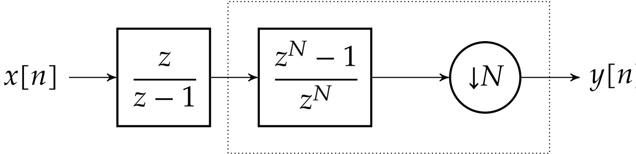
So far the strawberries and the cocolate. How about the champagne?

10.3.2 Crossing the rate-change border

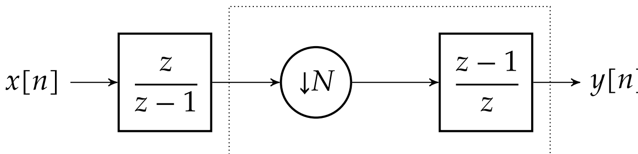
And now, for the champagne, let's get our things together to cross the rate-change border.

Decimation The starting point for decimation (taken from Figure 10.1a) was:

In this setup we are going to implement the low-pass filter as a (moving average) integrator-comb filter, i.e.

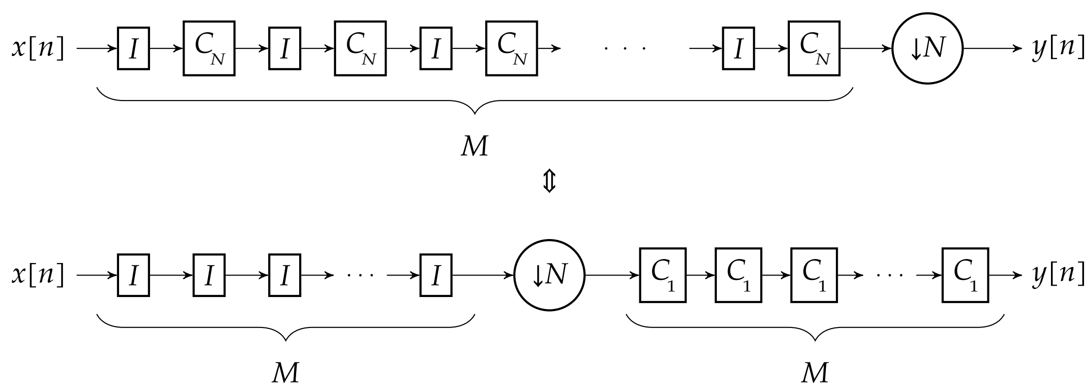


Now, uncork the bottle and apply the first Noble identity to the blocks in the dotted rectangle. This yields:



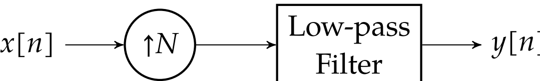
For short, we have proven the following equivalency:

or more generally:

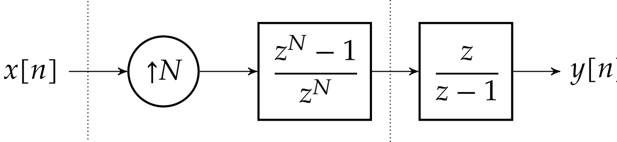


The savings in terms of required hardware are immense. The top implementation requires $M(N+1)$ registers. The bottom implementation only requires $2M$ registers. The relative savings amount to $2/(N+1)$!

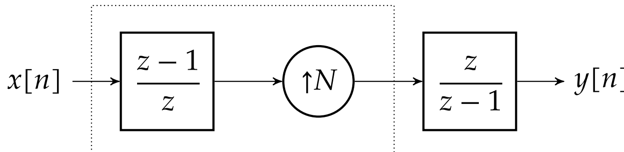
Interpolation The starting point for interpolation (taken from Figure 10.1b) was:



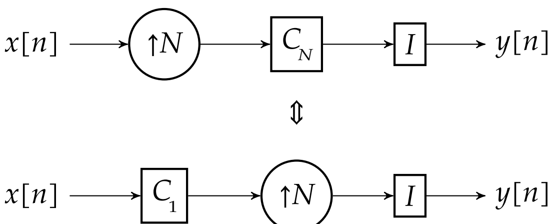
In this setup we are going to implement the low-pass filter as a (moving average) integrator-comb filter, i.e.



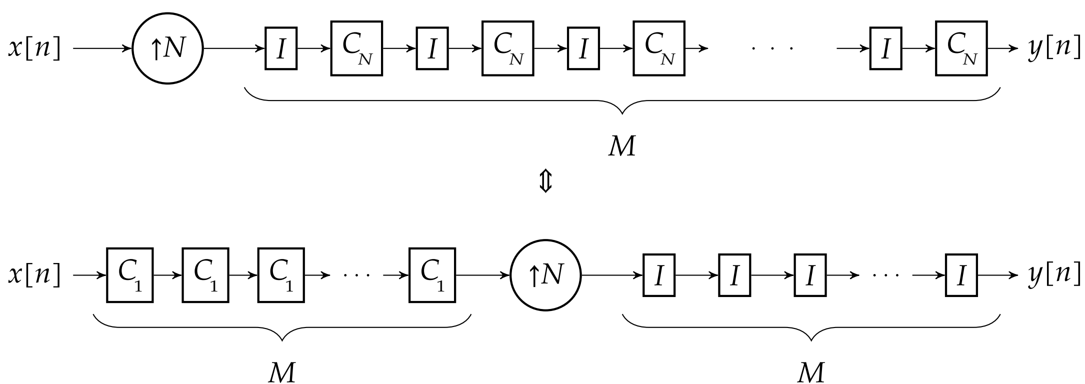
Now, uncork a second bottle and apply the second Noble identity to the blocks in the dotted rectangle. This yields:



For short, we have proven the following equivalency:



or more generally:



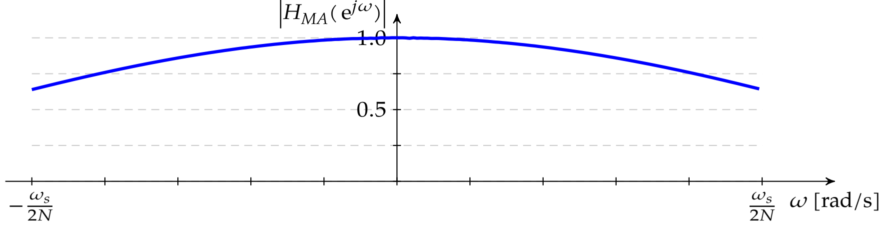
The savings in terms of required hardware are once again immense. The top implementation requires $M(N+1)$ registers. The bottom implementation only requires $2M$ registers. The relative savings amount to $2/(N+1)$!

10.4 Correcting the pass-band distorsion

The frequency characteristic of a moving-average is well known: It's mathematical description for an arbitrary N has been derived before:[2]

$$H_{MA}(\mathrm{e}^{j\omega}) = \frac{1}{N} \cdot \frac{\sin\left(\frac{N\omega}{2}\right)}{\sin\left(\frac{\omega}{2}\right)} \cdot \mathrm{e}^{-j\frac{(N-1)\omega}{2}}$$

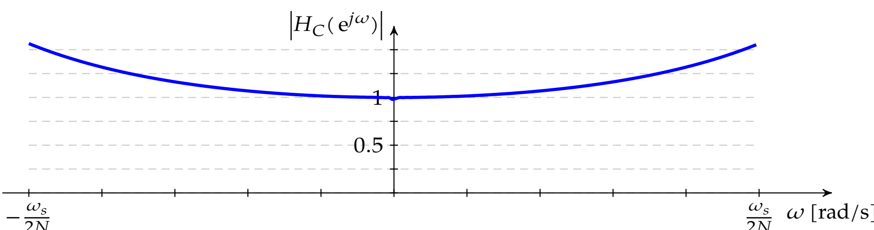
A graph of the frequency response (for $N = 10$ can be found in Figure 10.2c. If we zoom into the passband of the magnitude response (in between $-\frac{\omega_s}{2N}$ and $-\frac{\omega_s}{2N}$, we obtain:



Implementing a compensation filter means implementing a linear-phase FIR filter with the following magnitude response:

$$\left|H_C(\mathrm{e}^{j\omega})\right| = \frac{1}{\left|H_{MA}(\mathrm{e}^{j\omega})\right|} = N \cdot \frac{\sin\left(\frac{\omega}{2}\right)}{\sin\left(\frac{N\omega}{2}\right)}$$

Graphically:



One can use the FSDM or Parks-McClellan design techniques to implement it. To this end, the MATLAB/OCTAVE functions `fir2` and `remez` are very convenient.

[2]Note that we added the factor $1/N$ again.

Chapter 11

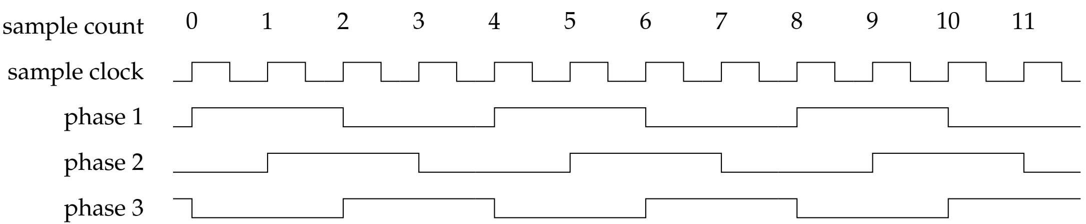
Polyphase filters

It was almost by accident that we discovered the perfect marriage between moving average filters and up- and down-sampling in the previous chapter. A fundamental question therefore pops up. Are there any other filters that exhibit similar properties, i.e. that allow parts of the filter to be moved accross the rate-change border?

The answer is as short as it is satisfying: yes, polyphase filters!

11.1 What's in a name?

In a three-phase power system, the sine waves of the phases are shifted over 120° w.r.t. to eachother. In a very similar way, we can define multi-phase clocks. This has been illustrated below for a 3-phase clock. On the top row, the sample counter has been shown, below it, the sample clock has been drawn, followed by the 3 phases.



Using these *polyphase clocks*, our intent is to treat the samples in a filter at a lower rate than the original sample clock. This explains the name *polyphase filters*. Let's first investigate a specific case for a 3-phase polyphase filter and let's generalize afterwards.

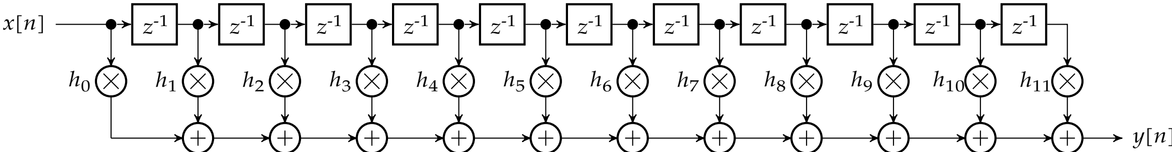
11.2 The maths - for 3 phases

Consider a FIR filter with an impulse response h_n of length 12. We'd like to split this filter into 3 polyphase filters.

The filter's transfer function can be written as:

$$\begin{aligned} H(z) = {} & h_0 + h_1 z^{-1} + h_2 z^{-2} + h_3 z^{-3} + h_4 z^{-4} + h_5 z^{-5} \\ & + h_6 z^{-6} + h_7 z^{-7} + h_8 z^{-8} + h_9 z^{-9} + h_{10} z^{-10} + h_{11} z^{-11} \end{aligned}$$

The corresponding block diagram is:



Implementing this filter requires 11 delay elements, 12 scalers, and 11 adders.

Let's rearrange this sum to gather the phase-corresponding terms:

$$\begin{aligned} H(z) &= h_0 + h_3\, z^{-3} + h_6\, z^{-6} + h_9\, z^{-9} \\ &\quad + h_1\, z^{-1} + h_4\, z^{-4} + h_7\, z^{-7} + h_{10}\, z^{-10} \\ &\quad + h_2\, z^{-2} + h_5\, z^{-5} + h_8\, z^{-8} + h_{11}\, z^{-11} \\ &= \underbrace{h_0 + h_3\, z^{-3} + h_6\, z^{-6} + h_9\, z^{-9}}_{H_0(z^3)} \\ &\quad + \; z^{-1}(\underbrace{h_1 + h_4\, z^{-3} + h_7\, z^{-6} + h_{10}\, z^{-9}}_{H_1(z^3)}) + \; z^{-2}(\underbrace{h_2 + h_5\, z^{-3} + h_8\, z^{-6} + h_{11}\, z^{-9}}_{H_2(z^3)}) \\ &= H_0(z) + z^{-1} H_1(z) + z^{-2} H_2(z) \end{aligned}$$

We assumed that $[h_i, h_{i+3}, h_{i+6}, h_{i+9}] \xrightarrow{Z} H_i(z)$ for $i = 0, \ldots M-1$.

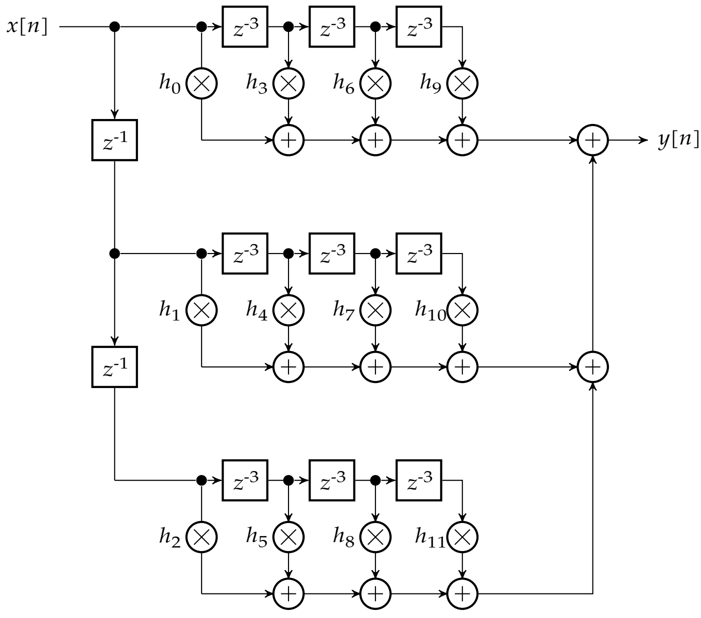
This factorization offers two routes. The first one is:

$$Y(z) = H(z)X(z) = H_0(z^3)X(z) + H_1(z^3)\Big(z^{-1}X(z)\Big) + H_2(z^3)\Big(z^{-2}X(z)\Big)$$

The second one:

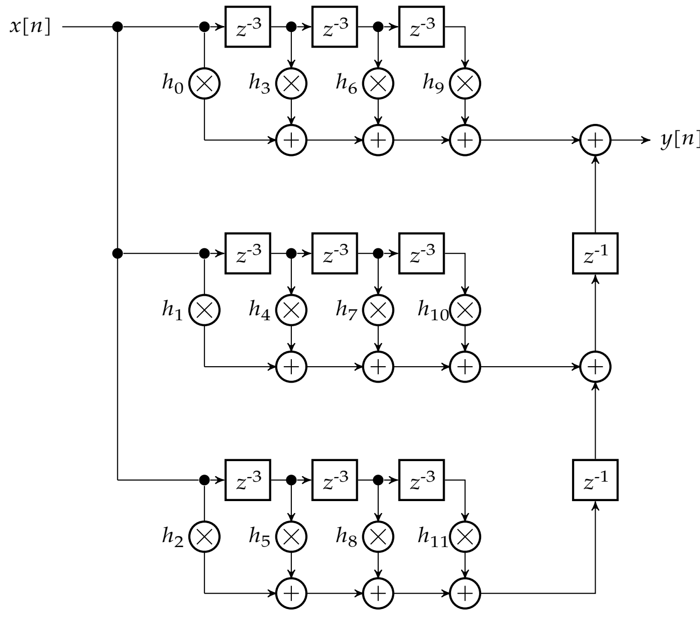
$$Y(z) = H(z)X(z) = H_0(z^3)X(z) + z^{-1}\Big(H_1(z^3)X(z)\Big) + z^{-2}\Big(H_2(z^3)X(Z)\Big)$$

The block diagram corresponding to the former is:



We will call it the *pre-delayed polyphase filter*.

The block diagram corresponding to the latter is:



We will call it the *post-delayed polyphase filter*.

Implementing these polyphase filters requires 29 delay elements 12 scalers and 11 adders. This is not really an improvement. The number of scalers and adders is unaltered, but we need approximately 3 times as many delay elements. On the positive side: the length of the adder chain is reduced, which may help keeping the timing in a hardware implementation.

11.3 The maths - for M phases

Let's start from a FIR filter, with an impulse response h_n of length $M \cdot N$. We'd like to split this impulse resonse into M polyphase filters.

The transfer function of our filter can be reworked as follows:

$$\begin{aligned} H(z) &= \sum_{n=0}^{M \cdot N-1} h_n \, z^{-n} \\ &= \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} h_{m+n \cdot M} \, z^{-m-n \cdot M} \\ &= \sum_{m=0}^{M-1} z^{-m} \underbrace{\sum_{n=0}^{N-1} h_{m+n \cdot M} \, z^{-n \cdot M}}_{H_m(z^M)} \end{aligned}$$

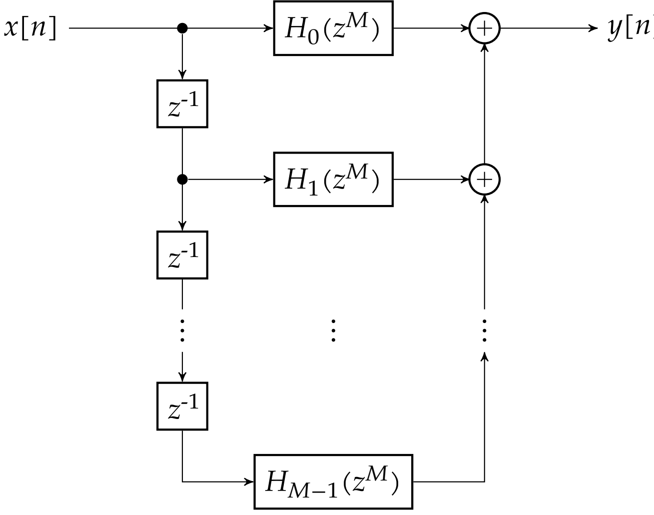
Again, we assumed that $[h_m, h_{m+M}, h_{m+2M}, \dots, h_{m+(N-1)M}] \xrightarrow{Z} H_m(z)$ for $m = 0, 1, \dots, M-1$.

This means that we can factorize an $M \cdot N$-length FIR filter into M single-phase filters at an M-times lower clock rate.

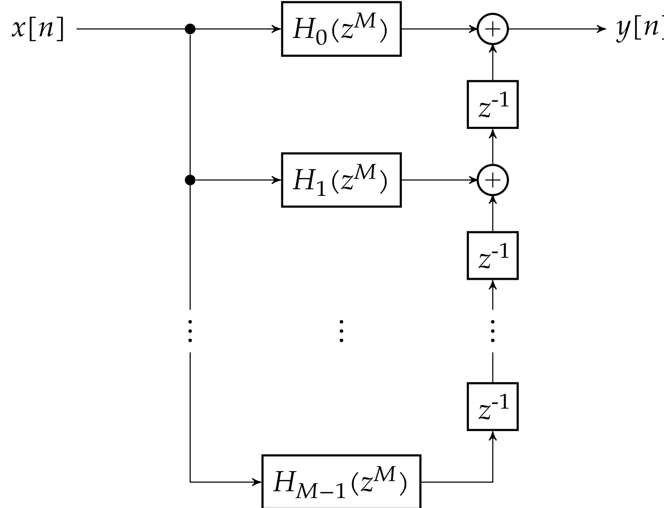
Graphically

$$x[n] \longrightarrow \boxed{H(z)} \longrightarrow y[n]$$

is equivalent with the *pre-delayed polyphase filter*:



and with the *post-delayed polyphase filter*:



Knowing that the length of the impulse response was MN, implementing the original filter required $MN - 1$ delay elements MN scalers and $MN - 1$ adders.

Determining the amount of hardware elements required for the polyphase versions is a bit more involved. Let's take a look at our earlier 3-phase example to determine them.

Number of scalers This is the easiest number to determine. It's obvious that there are as many scalers as there are elements in the original impulse response. Therefore: $M\dot{N}$ scalers are required.

Number of delay elements and adders The first observation is that there are an equal amount of adders and delay elements.
Let's start by considering the hardware not related to $H_i(z^3)$. The amount of adders and delay elements near the input and/our output equals $M-1$.
Then, there are M filter banks $H_i(z)$. Each of them requires $N-1$ adders and delay blocks. However, keep in mind that every delay block counts M delay elements.
Therefore, the total amounts to: $(M-1)+(M(N-1)) = MN-1$ adders and $(M-1)+M^2(N-1)$ delay elements.

Conclusion Implementing the filter as a polyphase filter requires

- $(N-1)M^2 + M - 1$
- MN scalers
- $MN-1$ adders

Again, this is not an improvement. The amount of scalers and addres is unaltered, but the amount of delay elements has gone up by almost a factor of M. On the positive side, the length of the adder chain is reduced.

11.4 Crossing the rate-change border

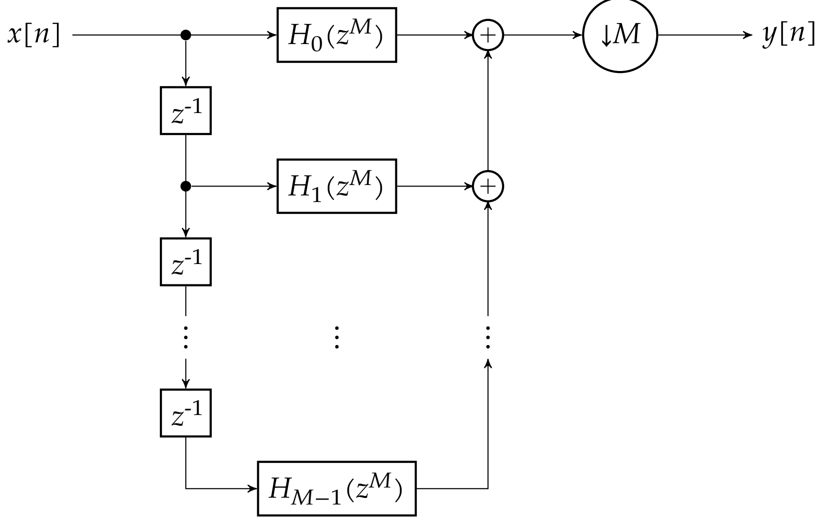
11.4.1 Decimation

Remember, the basic setup for decimation was

$$x[n] \longrightarrow \boxed{\mathrm{H(z)}} \longrightarrow (\downarrow M) \longrightarrow y[n]$$

with $H(z)$ a low-pass filter. Assuming we implement this as a FIR filter of length of MN. Implementing this filter requires MN delay elements, MN scalers and MN adders.

Now, let's plug in our pre-delayed polyphase filter:



Applying the first Noble identity this can be improved, yielding:

Implementing this filter requires exactly the same amount of hardware as the original filter:

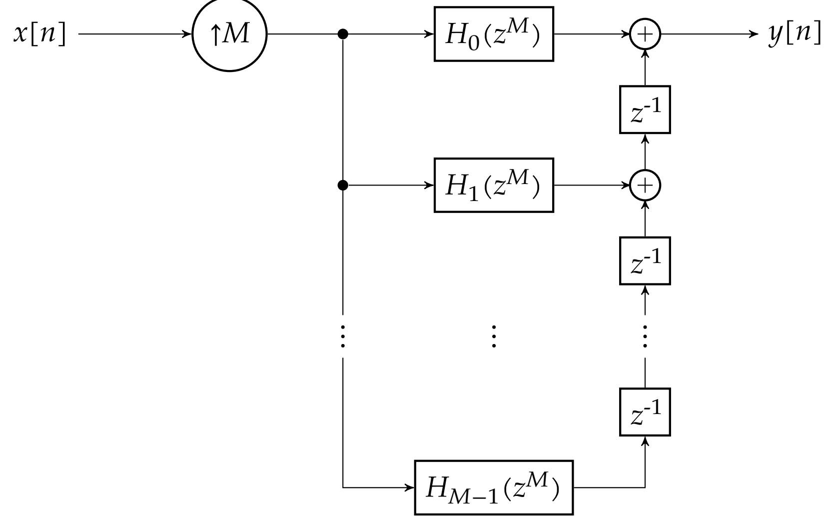
- $MN - 1$
- MN scalers
- $MN - 1$ adders

11.4.2 Interpolation

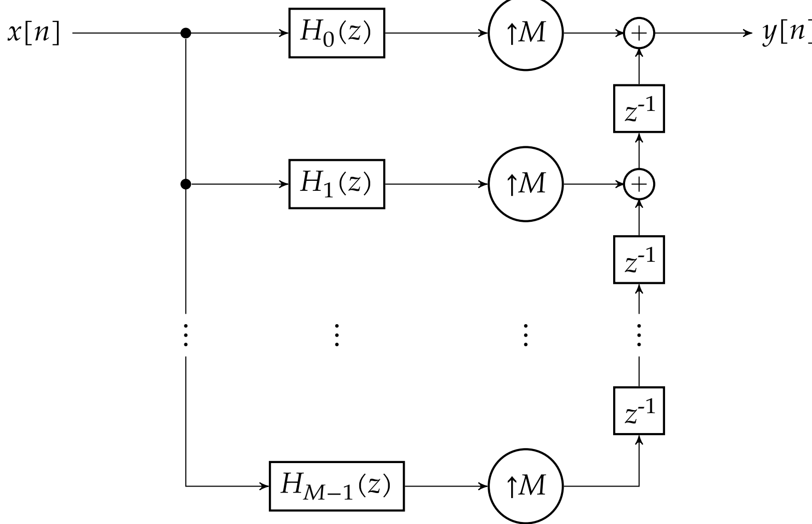
Remember, the basic setup for interpolation was

with $H(z)$ a low-pass filter. Assuming we implement this as a FIR filter of length of MN. Implementing this filter requires MN delay elements, MN scalers and MN adders.

Now, let's plug in our post-delayed polyphase filter:



Applying the second Noble identity this can be improved, yielding:



Implementing this filter requires

- $MN - 1$
- MN scalers
- $MN - 1$ adders

This is exactly the same amount as for the original filter.

11.5 Conclusion

Reconsidering the results of using polyphase filters for decimation and interpolation blocks, the conclusion so far is that we need exactly as much hardware as when using an ordinary filter. The only benefit seems the reduction of the adder chain length.

Then why were we so positive about polyphase filters in the introduction of this chapter?

Well, one should realize that though there is no gain in the amount of hardware, the hardware for the polyphase filter bank is running at a speed reduced by the decimation or interpolation factor (M). The power consumption of this block, therefore is reduced by a factor of M! Also in a software implementation this pays off in terms of an equal reduction in required processing power.

If we want to save the planet, this is another step in doing so.

Appendix A

Circular Library

Circular/cint.hh

```
/*
** CONFIDENTIAL AND PROPRIETARY (C) 2013 Universiteit Antwerpen
**
** Any use, compilation, modification, distribution, reproduction,
** performance, display, or disclosure ("Use") of this software code
** is subject to the terms and conditions of your written agreement
** with the Universiteit Antwerpen. If you do not have such an
** agreement, then any Use of this material is strictly prohibited.
**
** Unauthorized Use of this software code, or any portion of it, will
** result in civil liability and/or criminal penalties.  By modifying
** this software code, you agree to assign any and all intellectual
** property rights related in any way to your modification to the
** Universiteit Antwerpen.
**
** Students and former students of the Universiteit Antwerpen are
** allowed to use, compile and perform ( "Limited Use" ) this software
** code without an explicit written agreement.
*/

/*
** Name    : Circular/cint.hh
** Purpose: Circular buffer index
** Author : Walter Daems
*/

#ifndef CIRCULAR_CINT_HH
#define CIRCULAR_CINT_HH

#include <assert.h>

class cint {
public:
  // constructor
  cint( int i ) : _i ( i ) {}

  // copy constructor
  cint( const cint& ci ) : _i( ci._i ) {}

  // conversion operator
  operator int () const { return _i; }

  // modulo operator
  cint& operator %= ( int n )
  {
    assert( n > 0 );
    while ( _i >= n )
      _i -= n;
    while ( _i < 0 )
      _i += n;
  }
```

```
  // auto addition
  cint& operator += ( int i )
  {
    _i += i;
    return *this;
  }

  cint& operator += ( cint i )
  {
    _i += i._i;
    return *this;
  }

  // auto subtraction
  cint& operator -= ( int i )
  {
    _i -= i;
    return *this;
  }

  cint& operator -= ( cint i )
  {
    _i -= i._i;
    return *this;
  }

  // pre-increment operator
  cint& operator++ ()
  {
    ++_i;
    return *this;
  }

  // post-increment operator
  cint operator++ ( int )
  {
    cint i( *this );
    ++*this;
    return i;
  }

  // pre-decrement operator
  cint& operator-- ()
  {
    --_i;
    return *this;
  }

  // post-decrement operator
  cint operator-- ( int )
  {
    cint i( *this );
    --*this;
    return i;
  }

private:
  // representation
  int _i;
};

inline
cint operator % ( cint i, int n )
{
  i %= n;
  return i;
}

inline
```

```
cint operator + ( cint i, cint j )
{
  i += j;
  return i;
}

inline
cint operator + ( cint i, int j )
{
  i += j;
  return i;
}

inline
cint operator - ( cint i, cint j )
{
  i -= j;
  return i;
}

inline
cint operator - ( cint i, int j )
{
  i -= j;
  return i;
}

#endif /* CIRCULAR_CINT_HH */
```

Circular/cptr.hh

```
/*
** CONFIDENTIAL AND PROPRIETARY (C) 2008 Universiteit Antwerpen
**
** Any use, compilation, modification, distribution, reproduction,
** performance, display, or disclosure ("Use") of this software code
** is subject to the terms and conditions of your written agreement
** with the Universiteit Antwerpen. If you do not have such an
** agreement, then any Use of this material is strictly prohibited.
**
** Unauthorized Use of this software code, or any portion of it, will
** result in civil liability and/or criminal penalties.  By modifying
** this software code, you agree to assign any and all intellectual
** property rights related in any way to your modification to the
** Universiteit Antwerpen.
**
** Students and former students of the Universiteit Antwerpen are
** allowed to use, compile and perform ( "Limited Use" ) this software
** code without an explicit written agreement.
*/

/*
** Name   : Circular/cptr.hh
** Purpose: Circular buffer pointer
** Author : Walter Daems
*/

#ifndef CIRCULAR_CPTR_HH
#define CIRCULAR_CPTR_HH

#include <assert.h>

template <class T>
void modulo( T*& cur, T* bgn, T* end, int n )
{
  assert( n > 0 );
  assert( end - bgn == n );
  assert( end > bgn );

  while ( cur >= end )
    cur -= n;
  while ( cur < bgn )
    cur += n;
}


template <class T>
class cptr {
public:
  cptr( T* bgn, T* end, T* cur )
    : _bgn( bgn ), _end( end ), _cur( cur ), _size( end - bgn )
  {
    assert( end > bgn );
    assert( cur >= bgn );
    assert( cur < end );
  }

  cptr( T* bgn, T* end, int cur = 0 )
    : _bgn( bgn ), _end( end ), _cur( bgn + cur ), _size( end - bgn )
  {
    assert( end > bgn );
    assert( cur >= bgn );
    assert( cur < end );
  }

  cptr( T* bgn, int n, T* cur )
    : _bgn( bgn ), _end( bgn + n ), _cur( cur ), _size( n )
  {
  }
```

```
  cptr( T* bgn, int n, int cur = 0 )
    : _bgn( bgn ), _end( bgn + n ), _cur( bgn + cur ), _size( n )
  {
  }

  cptr( const cptr& ci )
    : _bgn( ci._bgn ), _end( ci._end ), _cur( ci._cur ), _size( ci._size )
  {
  }

  cptr& operator += ( int i )
  {
    modulo( _cur += i, _bgn, _end, _size );
    return *this;
  }

  cptr& operator -= ( int i )
  {
    return (*this) += -i;
  }

  cptr& operator++ ()
  {
    if( ++_cur == _end )
      _cur = _bgn;
    return *this;
  }

  cptr operator++ ( int )
  {
    cptr i( *this );
    ++*this;
    return i;
  }

  cptr& operator-- ()
  {
    if ( _cur == _bgn )
      _cur = _end;
    --_cur;
    return *this;
  }

  cptr operator-- ( int )
  {
    cptr i( *this );
    --*this;
    return i;
  }

  T& operator * () const
  {
    return *_cur;
  }

private:
  T*const    _bgn;
  T*const    _end;
  T*         _cur;
  const int  _size;
};

#endif /* CIRCULAR_CPTR_HH */
```

Appendix B

Mathematical Bits and Pieces

B.1 Orthogonal signal decompositions

Decomposing signals w.r.t. an orthogonal basis, is based on the observation that we can consider discrete-time signals to be vectors in a vector space.

We assume that the reader is familiar with the basics of vector spaces. The overview is only intended to refresh this knowledge and demonstrate how it can be applied to discrete-time signals. We start by investigating the elements of the set of signals.

B.1.1 Elements

The set under consideration is the set of all discrete-time signals S. These signals are sequences of numbers w.r.t. a time reference (e.g., $n = 0$).

Our signal f could be equidistantly sampled versions of a continuous-time function g:

$$f[n] = g(n\Delta), \qquad \forall n \in \mathbb{Z}$$

with Δ the *sampling period*.

However, f could as well be an abstract ordered set of numbers:

$$f[n] = [\dots, f_r, f_s, f_t, f_u, \underbrace{f_v}_{n=0}, f_w, f_x, f_y, \dots]$$

To make writing abstract signals easier, we will use integers as subscripts, with the subscript 0 referring to the time point $n = 0$. The abstract row above then becomes:

$$f[n] = \left[\dots, f_{-4}, \quad f_{-3}, \quad f_{-2}, \quad f_{-1}, \quad f_0, \quad f_1, \quad f_2, \quad f_3, \quad \dots\right]$$

We will concisely refer to the above equation as: $f[n] = [\dots, f_i, \dots]$

If a signal starts at $n = 0$ (i.e. it is zero for negative times, a so-called *causal signal*), we will omit the underbrace indicating the time point $n = 0$ and simply write

$$f[n] = \left[f_a, \quad f_b, \quad f_c, \quad f_d, \quad \dots\right]$$

silently assuming that f_a is the signal value for $n = 0$.

The signals we consider can be real or complex-valued ($f_i \in \mathbb{R}$ or $f_i \in \mathbb{C}$). Since $\mathbb{R} \subset \mathbb{C}$ we will only look at the more general case of complex-valued signals. In most occasions, the signals will have a limited time span (a.k.a. *limited support*), i.e., they are only nonzero in a limited range of time-indices. If these signals are only active for N time points, we will denote the set by S_N.

Finally, we also refer to the signal $f[n]$ in short as $\vec{f}$. When using $f[n]$ or $\vec{f}$ in the context of matrices, we will treat them as column vectors.

Remarks

- The arrow vector notation $\vec{f}$ is not so common in the wavelet and signal processing literature.
- However, an ambiguity arises from the fact that f_3 might denote the value of f for $n = 3$, but might as well denote a third version of f. In the former case it denotes a scalar, in the second case, it denotes a vector. In many (but not all cases) the correct meaning can be deduced from the context. However, in an introductory text, like this one, this puts an unacceptable burden on the novice reader. Therefore, we've chosen to adopt the vector notation, but in a limited fashion. Whenever a signal is not written as a function, we will use the arrow notation[1]. Whenever a signal is written as a function, we don't use the arrow symbol.

B.1.2 Operations

Addition Addition of two signals $\vec{f}$ and $\vec{g}$ is denoted by $\vec{f} + \vec{g}$ and obtained by adding the values for corresponding time points. If $f[n] = [\ldots, f_i, \ldots]$ and $g[n] = [\ldots, g_i, \ldots]$ then $\vec{h} = \vec{f} + \vec{g}$ is defined by:

$$h[n] = (f + g)[n] = [\ldots, f_i + g_i, \ldots]$$

Scaling Scaling a signal $\vec{f}$ with a scalar c out of a field F is denoted by $c\vec{f}$ and obtained by scaling all the individual values. If $f[n] = [\ldots, f_i, \ldots]$, then

$$cf[n] = [\ldots, cf_i, \ldots]$$

Most often, F is the set of real numbers $\mathbb{R}$ or complex numbers $\mathbb{C}$. In the former case, this leads to a *real vector space*; in the latter case to a *complex vector space*.

Scaling The scalar product of two signals $\vec{f}$ and $\vec{g}$ is denoted by $\langle \vec{f}, \vec{g} \rangle$. It is the basis for many new geometric notions like orthogonality and also length and distance.

If $f[n] = [\ldots, f_i, \ldots]$, and $g[n] = [\ldots, g_i, \ldots]$, then

$$\langle \vec{f}, \vec{g} \rangle = \sum_{i=-\infty}^{+\infty} f_i \overline{g_i} \tag{B.1}$$

where $\overline{g_i}$ denotes the complex conjugate of g_i.
For real signals $u[n] = [\ldots, u_i, \ldots]$ and $v[n] = [\ldots, v_i, \ldots]$, this reduces to:

$$\langle \vec{u}, \vec{v} \rangle = \sum_{i=-\infty}^{+\infty} u_i v_i$$

[1] A similar convention is to use a boldface font and is more common in signal processing literature. However, this is not compatible with hand writing. We therefore have chosen to use the arrow notation.

Note that for signals that are only active on time points ranging from r to t (i.e., they are of limited support), (D.1) reduces to:

$$\langle \vec{f}, \vec{g} \rangle = \sum_{i=r}^{t} f_i \overline{g_i}$$

B.1.3 Geometric notions

Two signals are orthogonal if their scalar product is zero. In symbols:

$$\vec{f} \perp \vec{g} \quad \Leftrightarrow \quad \langle \vec{f}, \vec{g} \rangle = 0$$

We can define the norm $\|\vec{f}\|$ of a signal $\vec{f}$ as:

$$\|\vec{f}\| = \sqrt{\langle \vec{f}, \vec{f} \rangle}$$

The norm of a signal $\vec{f}$ is a measure of its size. Very often the square of the norm is used, as it denotes the energy of the signal.

$$E = \|\vec{f}\|^2 = \langle \vec{f}, \vec{f} \rangle$$

We can define the distance $d(\vec{f}, \vec{g})$ between two signals $\vec{f}$ and $\vec{g}$ as:

$$d(\vec{f}, \vec{g}) = \|\vec{f} - \vec{g}\|$$

The distance is a measure of the similarity of the two signals. The smaller the distance, the more the signals are alike.

B.1.4 Vector space

The set of all complex-valued discrete-time signals S forms a vector space when it is equipped with the addition, scaling and scalar product as defined above.

Specifically, this means that the following properties hold. Let $\vec{x}$, $\vec{y}$ en $\vec{z}$ be arbitrary vectors in S, and a and b scalars.

1. the addition is commutative: $\vec{x} + \vec{y} = \vec{y} + \vec{x}$
2. the addition is associative: $(\vec{x} + \vec{y}) + \vec{z} = \vec{x} + (\vec{y} + \vec{z})$
3. the addition has an identity element $\vec{0}$: $\vec{x} + \vec{0} = \vec{x}$
4. every vector has an inverse element w.r.t. the addition $\forall \vec{x} \in S, \exists -\vec{x} \in S : \vec{x} + (-\vec{x}) = \vec{0}$
5. the scaling has an identity element, the scalar '1': $1\vec{x} = \vec{x}$

6. the scaling is distributive with respect to the vector addition: $a(\vec{x}+\vec{y}) = a\vec{x} + a\vec{y}$
7. the scaling is distributive with respect to the scalar addition: $(a+b)\vec{x} = a\vec{x} + b\vec{x}$

B.1.5 Set notions

Orthogonal set A (finite or infinite but countable) set of signals $V = \{\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots\}$ is a so-called *orthogonal set* of signals if and only if for every i and j holds:

$$\langle \vec{e}_i, \vec{e}_j \rangle = \begin{cases} 0 & \text{if} \quad i \neq j \\ \|e_i\|^2 \neq 0 & \text{if} \quad i = j \end{cases}$$

Independent set A (finite or infinite but countable) set of signals $V = \{\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots\}$ is an *independent set* of signals, if and only if the equation with scalars c_i

$$c_1\vec{e}_1 + c_2\vec{e}_2 + c_3\vec{e}_3 + \cdots = 0$$

only holds for $(c_1, c_2, c_3, \ldots) = (0, 0, 0, \ldots)$

Span The span S of a (finite or infinite but countable) set $V = \{\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots\}$ is defined by

$$S = \{\vec{g} \mid \exists (c_1, c_2, c_3, \ldots) \text{ with } c_i \in \mathbb{C} : \vec{g} = c_1\vec{e}_1 + c_2\vec{e}_2 + c_3\vec{e}_3 + \cdots\}$$

A commonly used notation is $S = \text{span}\{V\}$

B.1.6 Base and dimension of a vector space

Base A subset of vectors $B = \{\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots\}$ with $\vec{e}_i \in S$ is called a base of S if and only if

- B is an independent set
- B is complete, i.e. $S = \text{span}\{B\}$

If in addition B is an orthogonal set, then B is an *orthogonal base* of S.

If in addition $\|\vec{e}_i\| = 1, \forall i$, then B is an *orthonormal base* of S.[2]

Dimension The dimension of a vector space equals the size of the base, i.e. the number of base vectors.

For signals of unlimited support, the dimension of the vector space is infinite (but countable). When we restrict ourselves to signals of a specific limited support, the dimension of the vector space equals the support of the signals.

[2] In most literature, an orthonormal set of vectors is referred to as an *orthogonal set*. This is most confusing and for this reason, I chose not to use this very common misnomer.

B.1.7 Decomposition of vectors in terms of the base vectors

One can prove that for a signal $\vec{f}$ and an orthogonal base $B = \{\vec{e}_1, \vec{e}_2, \vec{e}_3, ...\}$ and the decomposition:

$$\vec{f} = c_1\vec{e}_1 + c_2\vec{e}_2 + c_3\vec{e}_3 + ...$$

the coefficients c_i can be determined as:

$$c_i = \frac{1}{\|\vec{e}_i\|^2}\left\langle \vec{f}, \vec{e}_i \right\rangle = \frac{\left\langle \vec{f}, \vec{e}_i \right\rangle}{\left\langle \vec{e}_i, \vec{e}_i \right\rangle}$$

This is the so-called *projection equation*. It calculates the projection of $\vec{f}$ onto a specific base vector.

If the base is orthonormal, we can simplify the above to:

$$c_i = \underbrace{\frac{1}{\|\vec{e}_i\|^2}}_{=1}\left\langle \vec{f}, \vec{e}_i \right\rangle = \left\langle \vec{f}, \vec{e}_i \right\rangle \tag{B.2}$$

B.1.8 Parseval's identity

Given a signal $\vec{f}$ and an orthogonal base $B = \{\vec{e}_1, \vec{e}_2, \vec{e}_3, ...\}$, such that $\vec{f} = \sum_i c_i\vec{e}_i$, then

$$\left\|\vec{f}\right\|^2 = |c_1|^2\|\vec{e}_1\|^2 + |c_2|^2\|\vec{e}_2\|^2 + |c_3|^2\|\vec{e}_3\|^2 + \cdots \tag{B.3}$$

In fact, Parseval's identity is a generalization of the Pythagorean theorem.

If the base B is orthonormal, (D.3) reduces to:

$$\left\|\vec{f}\right\|^2 = |c_1|^2 + |c_2|^2 + |c_3|^2 + \cdots \tag{B.4}$$

Very often this simplified equation is called the *energy preservation theorem* in relation to signal transforms, because of the following. When considering the sequence $[c_1, c_2, c_3, ...]$ to be a transformed signal $\vec{c}$ of the original signal $\vec{f}$, we can rewrite (D.4) as

$$\left\|\vec{f}\right\|^2 = \|\vec{c}\|^2$$

If we restrict ourselves to real signals, then (D.3) reduces to:

$$\left\|\vec{f}\right\|^2 = c_1^2\|\vec{e}_1\|^2 + c_2^2\|\vec{e}_2\|^2 + c_3^2\|\vec{e}_3\|^2 + \cdots$$

When, in addition the base is normalized, we can go one step further and write:

$$\left\|\vec{f}\right\|^2 = c_1^2 + c_2^2 + c_3^2 + \cdots$$

and again when $\vec{c} = [c_1, c_2, c_3, ...]$:

$$\left\|\vec{f}\right\|^2 = \|\vec{c}\|^2$$

B.2 Taylor's theorem

Preconditions Consider a function $f : \mathbb{R} \to \mathbb{R}$ on a closed interval $[a, b]$ with $a, b \in \mathbb{R}$. Let's assume that the function f is continuous and has continuous derivatives $f^{(i)}$ on the interval $[a, b]$ up to the $(n + 1)$-th derivative.

Theorem Under these circumstances, we can find a $c \in [a, b]$ for which the following holds:

$$
\begin{aligned}
f(b) &= f(a) + f^{(1)}(a) \cdot (b-a) + \frac{f^{(2)}(a)}{2!}(b-a)^2 + \frac{f^{(3)}(a)}{3!}(b-a)^3 + \cdots + \frac{f^{(n)}(a)}{n!}(b-a)^n \\
&\quad + \frac{f^{(n+1)}(c)}{(n+1)!}(b-a)^{(n+1)} \\
&= \sum_{i=0}^{n} \frac{f^{(i)}(a)}{i!}(b-a)^n + \frac{f^{(n+1)}(c)}{(n+1)!}(b-a)^{(n+1)}
\end{aligned}
$$

This is generally known as *Taylor's theorem* for approximating smooth continuous functions. It is also referred to as approximating functions by *Taylor expansion*. In case $a = 0$, the theorem is also referred to as *McLaurin's theorem*.

Note that the continuity conditions are a precondition to be able to apply this theorem!

Example As an example, consider approximating $\sin x$ for x in the interval $[0, 1]$. The sin-function is infinitely smooth (all its derivatives are continuous) and therefore Taylor's theorem can be applied. Applying the theorem for $n = 2$ yields:

$$
\begin{aligned}
\sin(x) &= \sin(0) + \cos(0) \cdot x + \frac{-\sin(0)}{2!}x^2 + \frac{\cos(c)}{3!}x^3 \\
&= x + \frac{\cos(c)}{3!}x^3
\end{aligned}
$$

with $c \in [0, 1]$.

B.3 The 'Big-O' notation

When calculating limits of a function $f(x)$, it is often not so interesting what the actual limit value is, but rather how quickly it approaches the limit value. This holds especially when the limiting value is infinity. Consider for example $f(x) = x$ and $f(x) = x^2$. The limit value for $x \to +\infty$ is in both cases $+\infty$. However, the speed at which they approach infinity is different. The function $f(x) = e^x$ even surpasses any x^n for arbitrary n in approaching infinity for $x \to \infty$.

To allow for a rough estimate of the approach speed, the Big-O concept was 'invented'.

B.3.1 Formal definition

Limiting case $x \to a$

We call $r(x) = O(f(x))$ for $x \to a$ if and only if we can find constants ϵ and C for which the following holds:

$$|r(x)| < C|f(x)|, \forall x \in [a, a + \varepsilon]$$

Limiting case $x \to \infty$
We call $r(x) = O(f(x))$ for $x \to \infty$ if and only if we can find constants x_0 and C for which the following holds:

$$|r(x)| < C|f(x)|, \forall x > x_0$$

B.3.2 Practical use

In asymptotic behavior of functions
If $r(x) = O(f(x))$ for $x \to a$ or $x \to \infty$ this means that both functions have the same *rate of attack* w.r.t. the asymptote. They may differ up to a multiplicative constant, but if you get close enough to the limiting value, they essentially approach their asymptotes with the same 'speed'. In words, we say 'r of x is big O of f of x when x approaches a'. Another frequently heard phrase is that $r(x)$ is of the same order of magnitude as $f(x)$ when x approaches a (or infinity).

In function approximation
In function approximation the Big-O notation is very used to identify an approximation error that is of a certain order of magnitude. Consider for example the Taylor series expansion of the previous section. The expression

$$f(b) = f(a) + f^{(1)}(a) \cdot (b-a) + \frac{f^{(2)}(a)}{2!}(b-a)^2 + \frac{f^{(3)}(a)}{3!}(b-a)^3 + \cdots + \frac{f^{(n)}(a)}{n!}(b-a)^n$$

is often written as:

$$f(b) = f(a) + f^{(1)}(a) \cdot (b-a) + \frac{f^{(2)}(a)}{2!}(b-a)^2 + \frac{f^{(3)}(a)}{3!}(b-a)^3 + \cdots + O((b-a)^n)$$

When $b \to a$, the value $(b-a)$ becomes infinitesimally small. in that case $(b-a)^n$ becomes very small. You probably have heard the phrase in case $n = 2$: "These are effects of second order, and therefore, we can neglect them". Of course, the phrase only holds if you think a linear approximation is good enough.

In algorithm development
The behavior of an algorithm w.r.t. the required computation time is usually dependent on the size of the problem. The relationship between size of the problem and the algorithm is called the *computational complexity* of the algorithm.

E.g., when writing a matrix multiplication algorithm to multiply $N \times N$-matrices, a naive implementation would lead you to the conclusion that we need about N^3 multiplications and additions. Of course, there are loop counters to be incremented and there is memory access to be taken care of. This may lead to the conclusion that one needs something like $1.05N^3 + 2N^2 - N + 5$ multiplications and additions.

In Big-O terms, we phrase this as 'matrix multiplication is $O\left(N^3\right)$'.[3]

The practical meaning is obvious. If you double the problem size N for an algorithm that is $O\left(N^3\right)$, the computation time will roughly increase by a factor of 2^3, i.e. 8. If a increase the problem size by a factor of 10, the computation time will roughly increase by a factor of 10^3, i.e. 1000!

Considering the computational complexity of an algorithm when attacking large problems is something you'd better do before writing any program code.

[3]Note that at this moment algorithms exist for matrix multiplication that are $O\left(N^{2.373}\right)$. This explains the adjective 'naive' that is used in the previous paragraph.

Appendix C

Amdahl's law

When using multiple concurrent computing resources (which we will denote as cores) to tackle a specific computing problem, the ideal case occurs when all these resources can operate in a parallel without interdependency.

This means that a system with Q equal and parallel cores offers a theoretical speedup S equaling

$$S = Q \tag{C.1}$$

It is because of this simplistic thinking, that you assume that your home computer with 8 cores, will be 8 times as fast as a similar single-core machine. However, it is not. The key reason is that not all tasks can be parallelized. If there is an interdependency between tasks that forces them to be executed sequentially, they cannot be parallelized (unless with other unrelated tasks). Therefore, we need to take this fact into account: only a fraction p (with $0 \leq p \leq 1$) of the total work to be done, can be parallelized. Under this assumption, the speedup is easily calculated to be:

$$S = \frac{1}{(1-p) + \frac{p}{Q}} \tag{C.2}$$

This is called *Amdahl's law* [Amd67]. It puts an effective limit on the speedup one can achieve. As such it is a true showstopper in the multi-core frenzy. The graph of Figure C.1 shows the speedup as a function of the number of cores with p as a parameter.

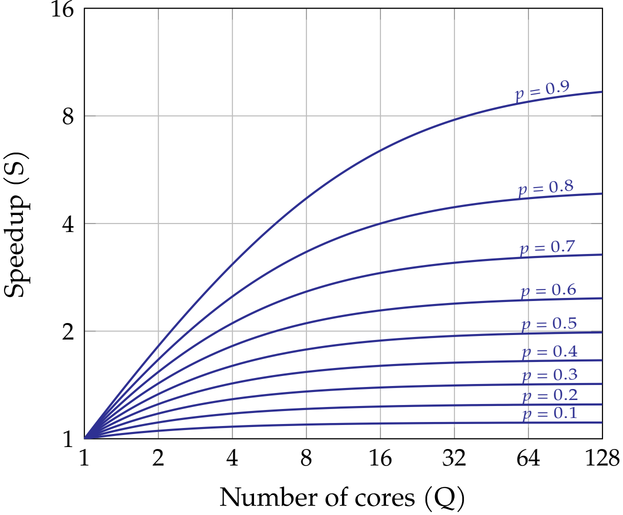


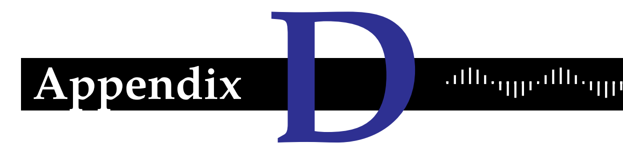
Figure C.1: Amdahl's law illustrated for various levels of the degree of parallelizability p

Note that (C.1) is a special case of (C.2) for $p = 1$.

The consequence of Amdahl's law is that adding more cores to a system will not pay-off linearly. It saturates with as upper speedup value the limit of (C.2) for $Q \to +\infty$:

$$\lim_{Q \to +\infty} S = \frac{1}{(1-p)}$$

Amdahl's law can be generalized to account for separate parts of the total work that can be parallized to a known degree, but this is outside of the scope of this appendix.



Stochastic Theory

In daily life, we often encounter phenomena that seem unpredictable. We often call them *random* phenomena. The term 'unpredictable' in fact is more correct than the term 'random'. In many cases the physics of of the phenomenon at hand is well understood and therefore not random at all. The only problem is that the physics governing the phenomenon is often very complex and therefore is influenced by a lot of factors. In addition, the sensitivity of the outcome on these factors is so big, that predicting that outcome becomes (almost) impossible.

As an example, consider throwing a dice. Throwing the dice can be accurately modeled using Newton's laws and fluid dynamics. However, the number of factors influencing the experiment of throwing a dice (the specific contraction of every muscle in your body when throwing, the humidity, the wind speed, the temperature, the pressure of the air, the smoothness of the table surface, the roughness of the dice edges, ...) is so big, that it is impossible to predict its outcome.

Where, in general, predicting the outcome of a single complex experiment is often impossible, luckily, in many cases we can make good predictions about executing a (big) number of experiments. In fact, the bigger the number of experiments, the more accurate the statements we can make about it.

Again, consider the example of a dice. When throwing a dice a thousand times, the number of cases in which the outcome will be a six, will be close to 1000/6 (in case of a fair dice). When summing all outcomes of throwing a thousand times, we expect the total value to be close to 3500.

In signal and image processing, stochastic theory is more and more gaining in importance. Therefore, in this appendix, we review the basic theory of stochastic phenomena.

D.1 Experiments and outcomes

The very basic concept of stochastic theory is the *experiment*. Consider it to be a particular action that

- one can take *repeatedly*, and
- leads to a specific *outcome* (a result).

In the process of throwing a dice, throwing a single time is the experiment, and the outcome is the number of eyes on the side facing up, i.e. one of 1, 2, 3, 4, 5 or 6.

The set of all outcomes is denoted by Ω, the outcome set. The outcome set can be discrete (e.g., when throwing a dice $\Omega = \{1, 2, 3, 4, 5, 6\}$), or can be continuous (e.g., the length of a person, when selecting a human being from the earth's population is the experiment).

D.2 Events

An event is a subset of the outcome set. If the outcome of an experiment is part of this subset, then we say *the event happened*. If it is not a part of the event subset, then we say *the event did not happen*.

In the process of throwing a dice, 'throwing an odd number', 'throwing a number less than or equal to two, would be events.

Note that the empty set $\emptyset$ and the outcome set Ω are both events themselves. The Ω-event always happens, the $\emptyset$-event never happens.

To keep things simple, we define the complement E_C of an event to be:

$$E^C = \Omega \setminus E$$

The set of all events is called the *event set* $\mathcal{E}$:

$$\mathcal{E} = \{E \mid E \subset \Omega\}$$

D.3 Probability function

We can define a probability function P that maps every event to a probability value:

$$P : \mathcal{E} \mapsto [0, 1]$$

The probability function expresses our estimate of the likelihood that an event occurs at the next experiment. If the probability function is dependent on time, we call the phenomenon under study a *stochastic process*. In the scope of this appendix, we will not consider this time dependence.

In order to be useful, a probability function should obey a number of rules:

1. $\forall E \in \mathcal{E} : 0 \leq P(E) \leq 1$
2. $P(\emptyset) = 0$
3. $P(\Omega) = 1$
4. $\forall E_1, E2 \in \mathcal{E} : E_1 \cap E_1 = \emptyset \quad \Rightarrow \quad P(E_1 \cup E_2) = P(E_1) + P(E_2)$

These *axiomae* enforce that our probability function makes sense.

A number of properties can be proven for a proper probability function:

1. $\forall E \in \mathcal{E} : P(E) + P(E^C) = 1$

2. $\forall E_1, E_2 \in \mathcal{E} : P(E_1 \cup E_2) = P(E_1) + P(E_2) - P(E_1 \cap E_2)$

We can also define a *conditional probability* $P(E \mid C)$ that expresses our estimate of an event E happening, knowing that event C happened for the experiment under study:

$$P(E \mid C) = \frac{P(E \cap C)}{P(C)}$$

in which we assumed that $C \neq \emptyset$. We call $P(E)$ the probability *a priori* and $P(E|C)$ the probability *a posteriori*. In plain English, we read $P(E|C)$ as the probability E happens, *given C happened* or short the probability of E *given* C.

This easily leads us to the well known *Bayes' rule*:

$$P(E|C) = \frac{P(C|E)P(E)}{P(C)} = \frac{P(C|E)P(E)}{P(C|E)P(E) + P(C|E^C)P(E^C)}$$

D.4 Stochastic or random variables

A stochastic variable is a mapping of the outcome set to a real number that is compatible with the event set.

$$X : \Omega \mapsto \mathbb{R}$$

The compatibility means that for every $a \in \mathbb{R}$ its corresponding event E_a, defined as:

$$E_a = \{\omega \mid X(\omega) \leq a\}$$

should be a proper event, i.e.:

$$E_a \in \mathcal{E}$$

When throwing a pair of dice, the sum of the eyes of a single throw would be an appropriate stochastic variable.

D.5 Describing stochastic variables

A stochastic variable can be described using its distribution. We distinguish two flavors of probability distributions.

D.5.1 Cumulative probability distribution

The *cumulative probability distribution* of a stochastic variable is a function $F_X(x)$ that is defined as:

$$F_X(x) = P(X \leq x)$$

with P the probability function corresponding to the experiments.

D.5.2 Ordinary probability distribution

Experiments with discrete outcome sets The *ordinary probability distribution* is defined as:

$$f_X(x) = P(X = x)$$

It is also known as the *probability mass function*.

Experiments with continuous outcome sets The *ordinary probability distribution* is defined as:

$$f_X(x) = \frac{\mathrm{d}}{\mathrm{d}x} F_X(x)$$

It is also known as the *probability density function*.

D.5.3 Properties

Again, a number of properties can be derived:

1. $F_X(x) = \int_{-\infty}^{x} f_X(u)\,\mathrm{d}u$
2. $0 \leq F_X(x) \leq 1$
3. $a \leq b \Rightarrow F_X(a) \leq F_X(b)$
4. $P(a < X \leq b) = F_X(b) - F_X(a)$

D.6 Describing distributions using moments

Distributions can have the most exotic shapes. Often the particular shape is not so important, but some basic properties are. This is the basis for describing distributions using *moments*. We distinguish *raw* moments and *centralized* moments.

D.6.1 Expected value

The basis for these moments is the *expectancy operator* E.

Experiments with discrete outcome sets

$$E\{X\} = \sum_i x_i f_X(x_i)$$

As the expectancy operator is an integral operator, it is linear (assuming the sum is related to the same random variable, and therefore governed by a single (joint) probability density function).

Experiments with continuous outcome sets

$$E\{X\} = \int_{-\infty}^{+\infty} x f_X(x)\,\mathrm{d}x$$

D.6.2 Raw moments

In general the i-th order raw moment $\alpha_{i,X}$ are defined as:

$$\alpha_{i,X} = E\left\{X^i\right\}$$

D.6.3 Centralized moments

In general the i-th order centralized moment $\mu_{i,X}$ is defined as:

$$\mu_{i,X} = E\left\{\left(X - \alpha_{1,X}\right)^i\right\}$$

It is easy to check that $\mu_{1,X} = 0$.

D.6.4 Characteristic values

Even more than the raw or centralized moments, the characteristic values are used. The symbols used below are quite typical and widely used:

The mean The mean indicates where the center of gravity of the distribution is located.

$$\mu_X = \alpha_{1,X} = E\left\{X\right\}$$

Note the very confusing symbol that is being used!

The variance The variance indicates how wide the distribution extends from its mean.

$$\sigma_X^2 = \mu_{2,X} = E\left\{\left(X - \mu_X\right)^2\right\}$$

Often, its square root is being used, the so-called *standard deviation* σ_X.

The skewness The skewness indicates if the distribution is assymetric.

$$\xi_X^3 = \frac{\mu_{3,X}}{\sigma_X^3}$$

If ξ_X^3 is positive, the distribution extends more to the right, if it is negative, it extends more to the left.

The kurtosis The kurtosis indicates the flatness of the distribution.

$$\kappa_X^4 = \frac{\mu_{4,x}}{\sigma_X^4} - 3$$

If $\kappa_X^4 > 0$ then it is flatter than the normal distribution. If it is smaller than zero, it is more peaked than the normal distribution.

D.7 Transformations of stochastic variables

Often, derived versions of stochastic variables are useful in some calculations. We consider two specific cases.

Affine transformation Given a stochastic variable X distributed as described by $F_X(x)$, we can derive a new stochastic variable Y by affine transformation:

$$Y = Ax + b \text{ with } a > 0.$$

This transformation is distributed according to $F_Y(y)$ with

$$F_Y(y) = F_X\left(\frac{y-b}{a}\right)$$

Nonlinear transformation Given a stochastic variable X distributed as described by $f_X(x)$, we can derive a new stochastic variable Y by nonlinear transformation g:

$$Y = g(X)$$

In case g is monotonically strictly increasing or decreasing, the following holds:

$$f_Y(y) = \frac{1}{\frac{\partial g(x)}{\partial x}} f_X(x)$$

D.8 Orthogonal signal decompositions

Decomposing signals w.r.t. an orthogonal basis, is based on the observation that we can consider discrete-time signals to be vectors in a vector space.

We assume that the reader is familiar with the basics of vector spaces. The overview is only intended to refresh this knowledge and demonstrate how it can be applied to discrete-time signals. We start by investigating the elements of the set of signals.

D.8.1 Elements

The set under consideration is the set of all discrete-time signals S. These signals are sequences of numbers w.r.t. a time reference (e.g., $n = 0$).

Our signal f could be equidistantly sampled versions of a continuous-time function g:

$$f[n] = g(n\Delta), \qquad \forall n \in \mathbb{Z}$$

with Δ the *sampling period*.

However, f could as well be an abstract ordered set of numbers:

$$f[n] = [\dots, f_r, f_s, f_t, f_u, \underbrace{f_v}_{n=0}, f_w, f_x, f_y, \dots]$$

To make writing abstract signals easier, we will use integers as subscripts, with the subscript 0 referring to the time point $n = 0$. The abstract row above then becomes:

$$f[n] = \left[\dots, f_{-4}, \quad f_{-3}, \quad f_{-2}, \quad f_{-1}, \quad f_0, \quad f_1, \quad f_2, \quad f_3, \quad \dots\right]$$

We will concisely refer to the above equation as: $f[n] = [\dots, f_i, \dots]$

If a signal starts at $n = 0$ (i.e. it is zero for negative times, a so-called *causal signal*), we will omit the underbrace indicating the time point $n = 0$ and simply write

$$f[n] = \left[f_a, \quad f_b, \quad f_c, \quad f_d, \quad \dots\right]$$

silently assuming that f_a is the signal value for $n = 0$.

The signals we consider can be real or complex-valued ($f_i \in \mathbb{R}$ or $f_i \in \mathbb{C}$). Since $\mathbb{R} \subset \mathbb{C}$ we will only look at the more general case of complex-valued signals. In most occasions, the signals will have a limited time span (a.k.a. *limited support*), i.e., they are only nonzero in a limited range of time-indices. If these signals are only active for N time points, we will denote the set by S_N.

Finally, we also refer to the signal $f[n]$ in short as $\vec{f}$. When using $f[n]$ or $\vec{f}$ in the context of matrices, we will treat them as column vectors.

Remarks

- The arrow vector notation $\vec{f}$ is not so common in the wavelet and signal processing literature.

- However, an ambiguity arises from the fact that f_3 might denote the value of f for $n = 3$, but might as well denote a third version of f. In the former case it denotes a scalar, in the second case, it denotes a vector. In many (but not all cases) the correct meaning can be deduced from the context. However, in an introductory text, like this one, this puts an unacceptable burden on the novice reader. Therefore, we've chosen to adopt the vector notation, but in a limited fashion. Whenever a signal is not written as a function, we will use the arrow notation[1]. Whenever a signal is written as a function, we don't use the arrow symbol.

D.8.2 Operations

Addition Addition of two signals $\vec{f}$ and $\vec{g}$ is denoted by $\vec{f} + \vec{g}$ and obtained by adding the values for corresponding time points. If $f[n] = [\dots, f_i, \dots]$ and $g[n] = [\dots, g_i, \dots]$ then $\vec{h} = \vec{f} + \vec{g}$ is defined by:

$$h[n] = (f + g)[n] = [\dots, f_i + g_i, \dots]$$

Scaling Scaling a signal $\vec{f}$ with a scalar c out of a field F is denoted by $c\vec{f}$ and obtained by scaling all the individual values. If $f[n] = [\dots, f_i, \dots]$, then

$$cf[n] = [\dots, cf_i, \dots]$$

Most often, F is the set of real numbers $\mathbb{R}$ or complex numbers $\mathbb{C}$. In the former case, this leads to a *real vector space*; in the latter case to a *complex vector space*.

Scaling The scalar product of two signals $\vec{f}$ and $\vec{g}$ is denoted by $\langle \vec{f}, \vec{g} \rangle$. It is the basis for many new geometric notions like orthogonality and also length and distance.

If $f[n] = [\dots, f_i, \dots]$, and $g[n] = [\dots, g_i, \dots]$, then

$$\langle \vec{f}, \vec{g} \rangle = \sum_{i=-\infty}^{+\infty} f_i \overline{g_i} \quad \text{(D.1)}$$

where $\overline{g_i}$ denotes the complex conjugate of g_i.
For real signals $u[n] = [\dots, u_i, \dots]$ and $v[n] = [\dots, v_i, \dots]$, this reduces to:

$$\langle \vec{u}, \vec{v} \rangle = \sum_{i=-\infty}^{+\infty} u_i v_i$$

Note that for signals that are only active on time points ranging from r to t (i.e., they are of limited support), (D.1) reduces to:

$$\langle \vec{f}, \vec{g} \rangle = \sum_{i=r}^{t} f_i \overline{g_i}$$

D.8.3 Geometric notions

Two signals are orthogonal if their scalar product is zero. In symbols:

$$\vec{f} \perp \vec{g} \quad \Leftrightarrow \quad \langle \vec{f}, \vec{g} \rangle = 0$$

[1]A similar convention is to use a boldface font and is more common in signal processing literature. However, this is not compatible with hand writing. We therefore have chosen to use the arrow notation.

We can define the norm $\|\vec{f}\|$ of a signal $\vec{f}$ as:

$$\|\vec{f}\| = \sqrt{\langle \vec{f}, \vec{f} \rangle}$$

The norm of a signal $\vec{f}$ is a measure of its size. Very often the square of the norm is used, as it denotes the energy of the signal.

$$E = \|\vec{f}\|^2 = \langle \vec{f}, \vec{f} \rangle$$

We can define the distance $d(\vec{f}, \vec{g})$ between two signals $\vec{f}$ and $\vec{g}$ as:

$$d(\vec{f}, \vec{g}) = \|\vec{f} - \vec{g}\|$$

The distance is a measure of the similarity of the two signals. The smaller the distance, the more the signals are alike.

D.8.4 Vector space

The set of all complex-valued discrete-time signals S forms a vector space when it is equipped with the addition, scaling and scalar product as defined above.

Specifically, this means that the following properties hold. Let $\vec{x}$, $\vec{y}$ en $\vec{z}$ be arbitrary vectors in S, and a and b scalars.

1. the addition is commutative: $\vec{x} + \vec{y} = \vec{y} + \vec{x}$
2. the addition is associative: $(\vec{x} + \vec{y}) + \vec{z} = \vec{x} + (\vec{y} + \vec{z})$
3. the addition has an identity element $\vec{0}$: $\vec{x} + \vec{0} = \vec{x}$
4. every vector has an inverse element w.r.t. the addition $\forall \vec{x} \in S, \exists -\vec{x} \in S : \vec{x} + (-\vec{x}) = \vec{0}$
5. the scaling has an identity element, the scalar '1': $1\vec{x} = \vec{x}$
6. the scaling is distributive with respect to the vector addition: $a(\vec{x} + \vec{y}) = a\vec{x} + a\vec{y}$
7. the scaling is distributive with respect to the scalar addition: $(a + b)\vec{x} = a\vec{x} + b\vec{x}$

D.8.5 Set notions

Orthogonal set A (finite or infinite but countable) set of signals $V = \{\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots\}$ is a so-called *orthogonal set* of signals if and only if for every i and j holds:

$$\langle \vec{e}_i, \vec{e}_j \rangle = \begin{cases} 0 & \text{if} \quad i \neq j \\ \|e_i\|^2 \neq 0 & \text{if} \quad i = j \end{cases}$$

Independent set A (finite or infinite but countable) set of signals $V = \{\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots\}$ is an *independent set* of signals, if and only if the equation with scalars c_i

$$c_1\vec{e}_1 + c_2\vec{e}_2 + c_3\vec{e}_3 + \cdots = 0$$

only holds for $(c_1, c_2, c_3, \ldots) = (0, 0, 0, \ldots)$

Span The span S of a (finite or infinite but countable) set $V = \{\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots\}$ is defined by

$$S = \{\vec{g} \mid \exists(c_1, c_2, c_3, \ldots) \text{ with } c_i \in \mathbb{C} : \vec{g} = c_1\vec{e}_1 + c_2\vec{e}_2 + c_3\vec{e}_3 + \cdots\}$$

A commonly used notation is $S = \text{span}\{V\}$

D.8.6 Base and dimension of a vector space

Base A subset of vectors $B = \{\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots\}$ with $\vec{e}_i \in S$ is called a base of S if and only if

- B is an independent set
- B is complete, i.e. $S = \text{span}\{B\}$

If in addition B is an orthogonal set, then B is an *orthogonal base* of S.

If in addition $\|\vec{e}_i\| = 1, \forall i$, then B is an *orthonormal base* of S.[2]

Dimension The dimension of a vector space equals the size of the base, i.e. the number of base vectors.

For signals of unlimited support, the dimension of the vector space is infinite (but countable). When we restrict ourselves to signals of a specific limited support, the dimension of the vector space equals the support of the signals.

D.8.7 Decomposition of vectors in terms of the base vectors

One can prove that for a signal $\vec{f}$ and an orthogonal base $B = \{\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots\}$ and the decomposition:

$$\vec{f} = c_1\vec{e}_1 + c_2\vec{e}_2 + c_3\vec{e}_3 + \ldots$$

the coefficients c_i can be determined as:

$$c_i = \frac{1}{\|\vec{e}_i\|^2}\left\langle \vec{f}, \vec{e}_i \right\rangle = \frac{\left\langle \vec{f}, \vec{e}_i \right\rangle}{\langle \vec{e}_i, \vec{e}_i \rangle}$$

This is the so-called *projection equation*. It calculates the projection of $\vec{f}$ onto a specific base vector.

[2]In most literature, an orthonormal set of vectors is referred to as an *orthogonal set*. This is most confusing and for this reason, I chose not to use this very common misnomer.

If the base is orthonormal, we can simplify the above to:

$$c_i = \frac{1}{\underbrace{\|\vec{e}_i\|^2}_{=1}} \left\langle \vec{f}, \vec{e}_i \right\rangle = \left\langle \vec{f}, \vec{e}_i \right\rangle \tag{D.2}$$

D.8.8 Parseval's identity

Given a signal $\vec{f}$ and an orthogonal base $B = \{\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots\}$, such that $\vec{f} = \sum_i c_i \vec{e}_i$, then

$$\left\|\vec{f}\right\|^2 = |c_1|^2 \left\|\vec{e}_1\right\|^2 + |c_2|^2 \left\|\vec{e}_2\right\|^2 + |c_3|^2 \left\|\vec{e}_3\right\|^2 + \cdots \tag{D.3}$$

In fact, Parseval's identity is a generalization of the Pythagorean theorem.

If the base B is orthonormal, (D.3) reduces to:

$$\left\|\vec{f}\right\|^2 = |c_1|^2 + |c_2|^2 + |c_3|^2 + \cdots \tag{D.4}$$

Very often this simplified equation is called the *energy preservation theorem* in relation to signal transforms, because of the following. When considering the sequence $[c_1, c_2, c_3, \ldots]$ to be a transformed signal $\vec{c}$ of the original signal $\vec{f}$, we can rewrite (D.4) as

$$\left\|\vec{f}\right\|^2 = \|\vec{c}\|^2$$

If we restrict ourselves to real signals, then (D.3) reduces to:

$$\left\|\vec{f}\right\|^2 = c_1^2 \|\vec{e}_1\|^2 + c_2^2 \|\vec{e}_2\|^2 + c_3^2 \|\vec{e}_3\|^2 + \cdots$$

When, in addition the base is normalized, we can go one step further and write:

$$\left\|\vec{f}\right\|^2 = c_1^2 + c_2^2 + c_3^2 + \cdots$$

and again when $\vec{c} = [c_1, c_2, c_3, \ldots]$:

$$\left\|\vec{f}\right\|^2 = \|\vec{c}\|^2$$

Bibliography

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[Boz94] S.M. Bozic. *Digital and Kalman Filtering*. Edward Arnold, 2nd edition, 1994.

[BtMvdBJvdV93] R.J. Beerends, H.G. ter Morsche, van den Berg J.C., and E.M. van de Vrie. *Fourier & Laplace transformaties* (*in Dutch*). Educaboek, 1993.

[DLW06] Gustaaf Deen, Paul Levrie, and Vanneste Wilfried. *Aanvullingen van Wiskunde* (*in Dutch*). Karel de Grote-Hogeschool, Katholieke Hogeschool Antwerpen, 2006.

[GW07] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Prentice Hall, 3rd edition, 2007.

[GWE09] Rafael C. Gonzalez, Richard E. Woods, and Steven L. Eddins. *Digital Image Processing Using MATLAB*. Prentice Hall, 2nd edition, 2009.

[Lyo04] Richard G. Lyons. *Understanding Digital Signal Processing, 2nd Edition*. Prentice Hall, 2004.

[Mal09] Stéphane Mallat. *A Wavelet Tour of Signal Processing* (*The Sparse Way*). Elsevier, Academic Press, 2009.

[MO94] H. Mannaert and A. Oosterlinck. *Stochastische Signaalanalyse en Filterontwerp* (*in Dutch*). Katholieke Universiteit Leuven, 1994.

[OSB99] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-time Signal Processing, 2nd Edition*. Prentice Hall, 1999.

[Sij04] Jan Sijbers. *Inleiding tot Digitale Signaalverwerking* (*in Dutch*). Universiteit Antwerpen, 2004.

[Smi03] Stephen W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing, 2nd Edition*. California Technical Publishing, 2003.

[Smi08a] Julius O. Smith. *Mathematics of the Discrete Fourier Transform* (*DFT*). http://ccrma.stanford.edu/~jos/mdft, accessed 2008. online book.

[Smi08b] Julius O. Smith. *Spectral Audio Signal Processing, March 2007 Draft*. http://ccrma.stanford.edu/~jos/sasp, accessed 2008. online book.

[Str97] Bjarne Stroustrup. *The C++ Programming Language, 3rd Edition*. Addison–Wesley, 1997.

[vdEV87] A.W.M. van den Enden and N.A.M. Verhoeckx. *Digitale signaalbewerking (in Dutch)*. Delta Press, 1987.

[vdW92] P.E.M van den Wyngaert. *Analoge en digitale ketens, een signaal- en systeembenadering (in Dutch)*. Katholieke Industriële Hogeschool Antwerpen, 1992.

[VK13] Martin Vetterli and Jelena Kovačević. *Wavelets and Subband Coding*. CreateSpace Independent Publishing Platform, 2013.

[Wal08] James S. Walker. *A primer on wavelets and their scientific applications*. Chapman & Hall/CRC, 2nd edition, 2008.

Index

adaptive systems, 31
all-pass filter, *see* digital filtering, basic filter types

band selection, *see* digital filtering
band-pass filter, *see* digital filtering, basic filter types
band-stop filter, *see* digital filtering, basic filter types
Base of a vector space, 346, 362
Bessel filters, 172
bilinear transformation, 178, 181
biquad filters, 166
bufer types
  circular buffers, 53
  linear buffers, 53
buffer types, 53
Butterworth filters, 171

Cauer filters, 174
causality, 15
Chebyshev filters, 172
Chebyshevv's Alternation Theorem, 151
CIC filters, 315–327
  correcting the pass-band distorsion, 327
  for decimation, 325
  for interpolation, 326
circular buffers, 53
comb filters
  feedback, 167
  feedforward, 116
Compression using wavelets, 299–308
convolution, 47–94
  basic implementation schemes, 52–53
    input-side algorithms, 52
    output-side algorithms, 53
  causal, 49
  computational complexity, 84
  generic discrete-time, 48
  latency, 84
  memory consumption, 84
  practical algorithms
    incremental, 57
    incremental — input-side, 57
    incremental — output-side, 59
    incremental), 60
  properties, 48–49
    associativity, 49
    commutativity, 48
    distributivity, 49
  selection of best algorithm, 84
    incremental vs. frame-based, 90
    overview, 85
  time-limited, 50
convolution vs. FFT-convolution
  coefficient of overperformance
    overlap–add, 92
    overlap–save, 92
correlation detection, *see* digital filtering

Decimation, 315
  CIC filters, 325
  polyphase filters, 333
deconvolution, *see* digital filtering
Denoising using wavelets, 308–311
digital filtering, 95–109
  band selection, 98
  basic filter types, 105
    all-pass filter, 107
    band-pass filter, 106
    band-stop filter, 106
    high-pass filter, 105
    low-pass filter, 105
    phase shaping filter, 107
  basic terminology and parameters, 107
  correlation detection, 102
  deconvolution, 101
    blind˜, 101
    limited˜, 102
  matched filtering, 102
  noise removal, 96
  parameters
    band-pass filter, 108
    band-stop filter, 108
    high-pass filter, 108
    low-pass filter, 108
Dimension of a vector space, 346, 362

Elliptical filters, 174
Euler transformation
  backward, 180
  forward, 179

feedback comb filters, 167
feedforward comb filters, 116
FFT-convolution, 72–94
  frame sequencing
    overlap–add, 76, 77, 80
    overlap–save, 78, 79, 82, 83
  overlap–add, 73
    double buffer, 75

single buffer, 74
overlap–save, 74
double buffer, 75
single buffer, 74
FFT-convolution vs. convolution
coefficient of overperformance
overlap–add, 92
overlap–save, 92
filter design
FIR filters, *see* FIR-filter design
IIR filters, *see* IIR-filter design
FIR-filter design, 113–161
bilinear transformation
warping and prewarping, 182
comparison, 160
frequency sampling method, 123
example, 128
truncation, 124
windowing, 127
impulse invariance, 120–123
example, 122
truncation, 121
windowing, 121
optimal methods, 133–161
least-squares, 137
least-squares — example, 145
least-squares — grid approach, 139
least-squares — integral approach, 142
Parks-McClellan, *see* Remez-exchange
Remez-exchange, 150
Remez-exchange — algorithm, 153
Remez-exchange — example, 155
overview, 160
zero placement, 114–120
arbitrary placement, 114
feedforward comb filters, 116
frequency sampling filter design method, 123

Gabor transform, *see* Short-time Fourier transform
Geometric notions related to signals, 345, 360

Haar transform, *see* Wavelets,Haar transform
high-pass filter, *see* digital filtering, basic filter types

IIR-filter design, 163–189
bilinear transformation method, 178–186
example, 183
fundamentals, 178
comparison, 189
impulse invariance, 175
design procedure, 176
example, 177
fundamentals, 175
overview, 189
pole-zero placement, 164–175
arbitrary placement, 164
band-selection filters, 171
biquad sections, 166
reverse filtering, 186–188
fundamentals, 187
principle, 186
zero placement
feedback comb filters, 167
impulse invariance filter design method
FIR, 120
IIR, 175
Interpolation, 315
CIC filters, 326
polyphase filters, 334
Inverse short-time discrete Fourier transform, 207
Inverse short-time Fourier transform
definition, 200

least-squares filter design method, 137
linear buffers, 53
Linear time-frequency transforms, 195–197
definition, 196
dictionary, 195
generation of, 197
scaling invariant, 196
translation and scaling invariant, 197
translation invariant, 196
energy, 196
Gabor transform, *see* Short-time Fourier transform
Short-time Fourier transform, *see* Short-time Fourier transform
linearity, 8
low-pass filter, *see* digital filtering, basic filter types
LTI system construction
continuous-time
nonrecursive, 20
recursive, 21
discrete-time
nonrecursive, 18
recursive, 19

matched filtering, *see* digital filtering
Moving average filters, 318
CIC filter, 323
comb filter, 320
IC-pair, 322
integrator, 319
moving-average filter, 96–98
filtering effect, 98
non-recursive implementation, 96
recursive implementation, 97
Multi-rate systems
decimation, *see* Decimation
interpolation, *see* Interpolation

Noble identities, 316
first, 316
second, 317
noise removal, *see* digital filtering
nonlinear systems, 31

optimal linear-phase filter design methods, 133
overlap–add
FFT-convolution, 73
overlap–save
FFT-convolution, 74

Parks-McClellan filter design method, *see* FIR-filter design, optimal methods, Remez-exchange
Parseval’s identity (vector space definition, 347, 363
pole-zero placement filter design method, 164
Polyphase filters, 329–335
3-phase example, 329
benefit of, 335
for decimation, 333
for interpolation, 334
M-phases, 331
naming explained, 329

Remez-exchange filter design method, *see* FIR-filter design, optimal methods, Remez-exchange

Set notions related to vector spaces, 346, 361
Short Time Fourier transform, 213
Short-time discrete Fourier transform, 201
computational complexity, 207
definition, 202
dictionary, 201
examples, 208
arbitrary signal analysis, 208
sound-level meter, 212
Heisenberg boxes, 203
position, 203
size, 205
Short-time Fourier transform, 193–201
definition, 198
dictionary, 197
Heisenberg boxes, 198
position, 199
size, 199
in discrete time, *see* Short-time discrete Fourier transform
Signal decompositions in terms of base vectors, 347, 362
Signal vector operations, 344, 360
Signals as elements of a vector space, 343, 359
system architectures, 33–45
comparison, 39–45
double-buffer architectures, 38–39
incremental vs frame based, 34–35
mixed-buffer architectures, 39
organizing frames, 35–39
single-buffer architectures, 35–38
systems, 5–31
classification, 6–10, 15–17, 22
definition, 6
homegeneous, 7
linear, 7
LTI, 10
basic forms, 31
benefits, 10
benefits — impulse decomposition, 12
benefits — multiple input reduction, 11
benefits — predicability, 11
causality, 15
FIR systems, 17
frequency-domain description — continuous-time, 14
frequency-domain description — discrete-time, 14
IIR systems, 17
nonrecursive, 22
recursive, 22
system construction, 17
system construction — continuous-time, 20
system construction — discrete-time, 17
system description, 12–15
system description — frequency-domain, 14–15
system description — time-domain, 12–14
time-domain description — continuous-time, 13
time-domain description — discrete-time, 12
MIMO, 6
MISO, 6
nonlinear, 7, 31
SIMO, 6
SISO, 6
stability, 7
superposition, 7
time invariance, 9
time-variant
adaptive, 31
variable, 31
systems (LTI)
basic forms, 23
nonrecursive direct form, 23
nonrecursive systems, 23
nonrecursive transposed form, 24
recursive direct form I, 26
recursive direct form II, 27
recursive systems, 25
recursive transposed form I, 26

recursive transposed form II, 27
selection — nonrecursive systems, 24
selection — recursive systems, 29

Time-frequency atoms, 193–195
Heisenberg boxes, 195
position and size in the frequency domain, 194
position and size in the time domain, 194

variable systems, 31
Vector space, 345, 361

Wavelets, 215–311
applications, 298–311
compression, 299–308
denoising, 308–311
approximation and detail, 220
base functions
Biorthonormal wavelets, 267
Cohen-Daubechies-Feauveau (CDF)wavelets, 272
Coifman wavelets, 258
Daubechies wavelets, 247
Haar wavelets, 223, 234
Brother John, 216–217, 300, 309
construction of, 240
digital filtering, 278
edge effects, 277
Haar transform, 219–240
level 1, 221
level-1 (from a xwavelet perspective), 223
level-n, 230
level-n (from a xwavelet perspective), 234
n-level, 230
trend and fluctuation, 227
Italian monument, 292, 304, 311
multiresolution analysis (MRA), 239
overview, 218
subband filtering, 278
trend and fluctuation, 220, 227
two-dimensional wavelets, 281
wavelet packet transform, 276
why?, 215
wavelets
base vectors
two-dimensional, 292
two-dimensional wavelets
level-1, 285
level-n, 286
n-level, 286
window filter design method, 123
windowed sinc filter design method, 123

zero placement filter design method, 114