

Handbook of Data Compression

Fifth Edition

www.it-ebooks.info

**David Salomon
Giovanni Motta**

With Contributions by David Bryant

Handbook of Data Compression

Fifth Edition

Previous editions published under the title
“Data Compression: The Complete Reference”

 **Springer**

www.it-ebooks.info

Prof. David Salomon (emeritus)
Computer Science Dept.
California State University, Northridge
Northridge, CA 91330-8281
USA
dsalomon@csun.edu

Dr. Giovanni Motta
Personal Systems Group, Mobility Solutions
Hewlett-Packard Corp.
10955 Tantau Ave.
Cupertino, California 95014-0770
gim@ieee.org

ISBN 978-1-84882-902-2 e-ISBN 978-1-84882-903-9
DOI 10.1007/10.1007/978-1-84882-903-9
Springer London Dordrecht Heidelberg New York

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2009936315

© Springer-Verlag London Limited 2010

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licenses issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

The use of registered names, trademarks, etc., in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

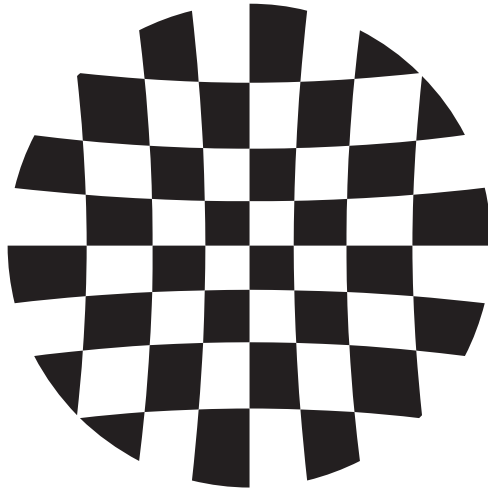
Cover design: eStudio Calamar S.L.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

www.it-ebooks.info

To users of data compression everywhere



I love being a writer. What I can't stand is the paperwork.
—Peter De Vries

www.it-ebooks.info

Preface to the New Handbook

GENTLE READER. The thick, heavy volume you are holding in your hands was intended to be the fifth edition of *Data Compression: The Complete Reference*. Instead, its title indicates that this is a handbook of data compression. What makes a book a handbook? What is the difference between a textbook and a handbook? It turns out that “handbook” is one of the many terms that elude precise definition. The many definitions found in dictionaries and reference books vary widely and do more to confuse than to illuminate the reader. Here are a few examples:

- A concise reference book providing specific information about a subject or location (but this book is not concise).
- A type of reference work that is intended to provide ready reference (but every reference work should provide ready reference).
- A pocket reference is intended to be carried at all times (but this book requires big pockets as well as deep ones).
- A small reference book; a manual (definitely does not apply to this book).
- General information source which provides quick reference for a given subject area. Handbooks are generally subject-specific (true for this book).

Confusing; but we will use the last of these definitions. The aim of this book is to provide a quick reference for the subject of data compression. Judging by the size of the book, the “reference” is certainly there, but what about “quick?” We believe that the following features make this book a quick reference:

- The detailed index which constitutes 3% of the book.
- The glossary. Most of the terms, concepts, and techniques discussed throughout the book appear also, albeit briefly, in the glossary.

www.it-ebooks.info

■ The particular organization of the book. Data is compressed by removing redundancies in its original representation, and these redundancies depend on the type of data. Text, images, video, and audio all have different types of redundancies and are best compressed by different algorithms which in turn are based on different approaches. Thus, the book is organized by different data types, with individual chapters devoted to image, video, and audio compression techniques. Some approaches to compression, however, are general and work well on many different types of data, which is why the book also has chapters on variable-length codes, statistical methods, dictionary-based methods, and wavelet methods.

The main body of this volume contains 11 chapters and one appendix, all organized in the following categories, basic methods of compression, variable-length codes, statistical methods, dictionary-based methods, methods for image compression, wavelet methods, video compression, audio compression, and other methods that do not conveniently fit into any of the above categories. The appendix discusses concepts of information theory, the theory that provides the foundation of the entire field of data compression.

In addition to its use as a quick reference, this book can be used as a starting point to learn more about approaches to and techniques of data compression as well as specific algorithms and their implementations and applications. The broad coverage makes the book as complete as practically possible. The extensive bibliography will be very helpful to those looking for more information on a specific topic. The liberal use of illustrations and tables of data helps to clarify the text.

This book is aimed at readers who have general knowledge of computer applications, binary data, and files and want to understand how different types of data can be compressed. The book is not for dummies, nor is it a guide to implementors. Someone who wants to implement a compression algorithm *A* should have coding experience and should rely on the original publication by the creator of *A*.

In spite of the growing popularity of Internet searching, which often locates quantities of information of questionable quality, we feel that there is still a need for a concise, reliable reference source spanning the full range of the important field of data compression.

New to the Handbook

The following is a list of the new material in this book (material not included in past editions of *Data Compression: The Complete Reference*).

- The topic of compression benchmarks has been added to the Introduction.
- The paragraphs titled “How to Hide Data” in the Introduction show how data compression can be utilized to quickly and efficiently hide data in plain sight in our computers.
- Several paragraphs on compression curiosities have also been added to the Introduction.
- The new Section 1.1.2 shows why irreversible compression may be useful in certain situations.
- Chapters 2 through 4 discuss the all-important topic of variable-length codes. These chapters discuss basic, advanced, and robust variable-length codes. Many types of VL

codes are known, they are used by many compression algorithms, have different properties, and are based on different principles. The most-important types of VL codes are prefix codes and codes that include their own length.

- Section 2.9 on phased-in codes was wrong and has been completely rewritten.
- An example of the start-step-stop code $(2, 2, \infty)$ has been added to Section 3.2.
- Section 3.5 is a description of two interesting variable-length codes dubbed recursive bottom-up coding (RBUC) and binary adaptive sequential coding (BASC). These codes represent compromises between the standard binary (β) code and the Elias gamma codes.
- Section 3.28 discusses the original method of interpolative coding whereby dynamic variable-length codes are assigned to a strictly monotonically increasing sequence of integers.
- Section 5.8 is devoted to the compression of PK (packed) fonts. These are older bitmaps fonts that were developed as part of the huge T_EX project. The compression algorithm is not especially efficient, but it provides a rare example of run-length encoding (RLE) without the use of Huffman codes.
- Section 5.13 is about the Hutter prize for text compression.
- PAQ (Section 5.15) is an open-source, high-performance compression algorithm and free software that features sophisticated prediction combined with adaptive arithmetic encoding. This free algorithm is especially interesting because of the great interest it has generated and because of the many versions, subversions, and derivatives that have been spun off it.
- Section 6.3.2 discusses LZR, a variant of the basic LZ77 method, where the lengths of both the search and look-ahead buffers are unbounded.
- Section 6.4.1 is a description of LZB, an extension of LZSS. It is the result of evaluating and comparing several data structures and variable-length codes with an eye to improving the performance of LZSS.
- SLH, the topic of Section 6.4.2, is another variant of LZSS. It is a two-pass algorithm where the first pass employs a hash table to locate the best match and to count frequencies, and the second pass encodes the offsets and the raw symbols with Huffman codes prepared from the frequencies counted by the first pass.
- Most LZ algorithms were developed during the 1980s, but LZPP, the topic of Section 6.5, is an exception. LZPP is a modern, sophisticated algorithm that extends LZSS in several directions and has been inspired by research done and experience gained by many workers in the 1990s. LZPP identifies several sources of redundancy in the various quantities generated and manipulated by LZSS and exploits these sources to obtain better overall compression.
- Section 6.14.1 is devoted to LZT, an extension of UNIX compress/LZC. The major innovation of LZT is the way it handles a full dictionary.

- LZJ (Section 6.17) is an interesting LZ variant. It stores in its dictionary, which can be viewed either as a multiway tree or as a forest, *every* phrase found in the input. If a phrase is found n times in the input, only one copy is stored in the dictionary. Such behavior tends to fill the dictionary up very quickly, so LZJ limits the length of phrases to a preset parameter h .
- The interesting, original concept of antidictionary is the topic of Section 6.31. A dictionary-based encoder maintains a list of bits and pieces of the data and employs this list to compress the data. An antidictionary method, on the other hand, maintains a list of strings that do not appear in the data. This generates negative knowledge that allows the encoder to predict with certainty the values of many bits and thus to drop those bits from the output, thereby achieving compression.
- The important term “pixel” is discussed in Section 7.1, where the reader will discover that a pixel is not a small square, as is commonly assumed, but a mathematical point.
- Section 7.10.8 discusses the new HD photo (also known as JPEG XR) compression method for continuous-tone still images.
- ALPC (*adaptive linear prediction and classification*), is a lossless image compression algorithm described in Section 7.12. ALPC is based on a linear predictor whose coefficients are computed for each pixel individually in a way that can be mimicked by the decoder.
- Grayscale Two-Dimensional Lempel-Ziv Encoding (GS-2D-LZ, Section 7.18) is an innovative dictionary-based method for the lossless compression of grayscale images.
- Section 7.19 has been partially rewritten.
- Section 7.40 is devoted to spatial prediction, a combination of JPEG and fractal-based image compression.
- A short historical overview of video compression is provided in Section 9.4.
- The all-important H.264/AVC video compression standard has been extended to allow for a compressed stream that supports temporal, spatial, and quality scalable video coding, while retaining a base layer that is still backward compatible with the original H.264/AVC. This extension is the topic of Section 9.10.
- The complex and promising VC-1 video codec is the topic of the new, long Section 9.11.
- The new Section 11.6.4 treats the topic of syllable-based compression, an approach to compression where the basic data symbols are syllables, a syntactic form between characters and words.
- The commercial compression software known as stuffit has been around since 1987. The methods and algorithms it employs are proprietary, but some information exists in various patents. The new Section 11.16 is an attempt to describe what is publicly known about this software and how it works.
- There is now a short appendix that presents and explains the basic concepts and terms of information theory.

We would like to acknowledge the help, encouragement, and cooperation provided by Yuriy Reznik, Matt Mahoney, Mahmoud El-Sakka, Pawel Pylak, Darryl Lovato, Raymond Lau, Cosmin Truța, Derong Bao, and Honggang Qi. They sent information, reviewed certain sections, made useful comments and suggestions, and corrected numerous errors.

A special mention goes to David Bryant who wrote Section 10.11.

Springer Verlag has created the Springer Handbook series on important scientific and technical subjects, and there can be no doubt that data compression should be included in this category. We are therefore indebted to our editor, Wayne Wheeler, for proposing this project and providing the encouragement and motivation to see it through.

The book's Web site is located at www.DavidSalomon.name. Our email addresses are dsalomon@csun.edu and gim@ieee.org and readers are encouraged to message us with questions, comments, and error corrections.

Those interested in data compression in general should consult the short section titled "Joining the Data Compression Community," at the end of the book, as well as the following resources:

- <http://compression.ca/>,
- <http://www-isl.stanford.edu/~gray/iii.html>,
- http://www.hn.is.uec.ac.jp/~arimura/compression_links.html, and
- <http://datacompression.info/>.

(URLs are notoriously short lived, so search the Internet.)

David Salomon

Giovanni Motta

The preface is usually that part of a book which can most safely be omitted.

—William Joyce, *Twilight Over England* (1940)



Preface to the Fourth Edition

(This is the Preface to the 4th edition of *Data Compression: The Complete Reference*, the predecessor of this volume.) I was pleasantly surprised when in November 2005 a message arrived from Wayne Wheeler, the new computer science editor of Springer Verlag, notifying me that he intends to qualify this book as a Springer major reference work (MRW), thereby releasing past restrictions on page counts, freeing me from the constraint of having to compress my style, and making it possible to include important and interesting data compression methods that were either ignored or mentioned in passing in previous editions.

These fascicles will represent my best attempt to write a comprehensive account, but computer science has grown to the point where I cannot hope to be an authority on all the material covered in these books. Therefore I'll need feedback from readers in order to prepare the official volumes later.

I try to learn certain areas of computer science exhaustively; then I try to digest that knowledge into a form that is accessible to people who don't have time for such study.

—Donald E. Knuth, <http://www-cs-faculty.stanford.edu/~knuth/> (2006)

Naturally, all the errors discovered by me and by readers in the third edition have been corrected. Many thanks to all those who bothered to send error corrections, questions, and comments. I also went over the entire book and made numerous additions, corrections, and improvements. In addition, the following new topics have been included in this edition:

- Tunstall codes (Section 2.6). The advantage of variable-size codes is well known to readers of this book, but these codes also have a downside; they are difficult to work with. The encoder has to accumulate and append several such codes in a short buffer, wait until n bytes of the buffer are full of code bits (where n must be at least 1), write the n bytes on the output, shift the buffer n bytes, and keep track of the location of the last bit placed in the buffer. The decoder has to go through the reverse process.

www.it-ebooks.info

The idea of Tunstall codes is to construct a set of fixed-size codes, each encoding a variable-size string of input symbols. As an aside, the “pod” code (Table 10.29) is also a new addition.

- Recursive range reduction (3R) (Section 1.7) is a simple coding algorithm due to Yann Guidon that offers decent compression, is easy to program, and its performance is independent of the amount of data to be compressed.
- LZARI, by Haruhiko Okumura (Section 6.4.3), is an improvement of LZSS.
- RAR (Section 6.22). The popular RAR software is the creation of Eugene Roshal. RAR has two compression modes, general and special. The general mode employs an LZSS-based algorithm similar to ZIP Deflate. The size of the sliding dictionary in RAR can be varied from 64 Kb to 4 Mb (with a 4 Mb default value) and the minimum match length is 2. Literals, offsets, and match lengths are compressed further by a Huffman coder. An important feature of RAR is an error-control code that increases the reliability of RAR archives while being transmitted or stored.
- 7-z and LZMA (Section 6.26). LZMA is the main (as well as the default) algorithm used in the popular 7z (or 7-Zip) compression software [7z 06]. Both 7z and LZMA are the creations of Igor Pavlov. The software runs on Windows and is free. Both LZMA and 7z were designed to provide high compression, fast decompression, and low memory requirements for decompression.
- Stephan Wolf made a contribution to Section 7.34.4.
- H.264 (Section 9.9). H.264 is an advanced video codec developed by the ISO and the ITU as a replacement for the existing video compression standards H.261, H.262, and H.263. H.264 has the main components of its predecessors, but they have been extended and improved. The only new component in H.264 is a (wavelet based) filter, developed specifically to reduce artifacts caused by the fact that individual macroblocks are compressed separately.
- Section 10.4 is devoted to the WAVE audio format. WAVE (or simply Wave) is the native file format employed by the Windows operating system for storing digital audio data.
- FLAC (Section 10.10). FLAC (free lossless audio compression) is the brainchild of Josh Coalson who developed it in 1999 based on ideas from Shorten. FLAC was especially designed for audio compression, and it also supports streaming and archival of audio data. Coalson started the FLAC project on the well-known sourceforge Web site [sourceforge.flac 06] by releasing his reference implementation. Since then many developers have contributed to improving the reference implementation and writing alternative implementations. The FLAC project, administered and coordinated by Josh Coalson, maintains the software and provides a reference codec and input plugins for several popular audio players.
- WavPack (Section 10.11, written by David Bryant). WavPack [WavPack 06] is a completely open, multiplatform audio compression algorithm and software that supports three compression modes, lossless, high-quality lossy, and a unique hybrid compression

mode. It handles integer audio samples up to 32 bits wide and also 32-bit IEEE floating-point data [IEEE754 85]. The input stream is partitioned by WavPack into blocks that can be either mono or stereo and are generally 0.5 seconds long (but the length is actually flexible). Blocks may be combined in sequence by the encoder to handle multichannel audio streams. All audio sampling rates are supported by WavPack in all its modes.

- Monkey's audio (Section 10.12). Monkey's audio is a fast, efficient, free, lossless audio compression algorithm and implementation that offers error detection, tagging, and external support.
- MPEG-4 ALS (Section 10.13). MPEG-4 Audio Lossless Coding (ALS) is the latest addition to the family of MPEG-4 audio codecs. ALS can input floating-point audio samples and is based on a combination of linear prediction (both short-term and long-term), multichannel coding, and efficient encoding of audio residues by means of Rice codes and block codes (the latter are also known as block Gilbert-Moore codes, or BGMC [Gilbert and Moore 59] and [Reznik 04]). Because of this organization, ALS is not restricted to the encoding of audio signals and can efficiently and losslessly compress other types of fixed-size, correlated signals, such as medical (ECG and EEG) and seismic data.
- AAC (Section 10.15). AAC (advanced audio coding) is an extension of the three layers of MPEG-1 and MPEG-2, which is why it is often called **mp4**. It started as part of the MPEG-2 project and was later augmented and extended as part of MPEG-4. Apple Computer has adopted AAC in 2003 for use in its well-known iPod, which is why many believe (wrongly) that the acronym AAC stands for apple audio coder.
- Dolby AC-3 (Section 10.16). AC-3, also known as Dolby Digital, stands for Dolby's third-generation audio coder. AC-3 is a perceptual audio codec based on the same principles as the three MPEG-1/2 layers and AAC. The new section included in this edition concentrates on the special features of AC-3 and what distinguishes it from other perceptual codecs.
- Portable Document Format (PDF, Section 11.13). PDF is a popular standard for creating, editing, and printing documents that are independent of any computing platform. Such a document may include text and images (graphics and photos), and its components are compressed by well-known compression algorithms.
- Section 11.14 (written by Giovanni Motta) covers a little-known but important aspect of data compression, namely how to compress the differences between two files.
- Hyperspectral data compression (Section 11.15, partly written by Giovanni Motta) is a relatively new and growing field. Hyperspectral data is a set of data items (called pixels) arranged in rows and columns where each pixel is a vector. A home digital camera focuses visible light on a sensor to create an image. In contrast, a camera mounted on a spy satellite (or a satellite searching for minerals and other resources) collects and measures radiation of many wavelengths. The intensity of each wavelength is converted into a number, and the numbers collected from one point on the ground form a vector that becomes a pixel of the hyperspectral data.

Another pleasant change is the great help I received from Giovanni Motta, David Bryant, and Cosmin Truța. Each proposed topics for this edition, went over some of

the new material, and came up with constructive criticism. In addition, David wrote Section 10.11 and Giovanni wrote Section 11.14 and part of Section 11.15.

I would like to thank the following individuals for information about certain topics and for clearing up certain points. Igor Pavlov for help with 7z and LZMA, Stephan Wolf for his contribution, Matt Ashland for help with Monkey's audio, Yann Guidon for his help with recursive range reduction (3R), Josh Coalson for help with FLAC, and Eugene Roshal for help with RAR.

In the first volume of this biography I expressed my gratitude to those individuals and corporate bodies without whose aid or encouragement it would not have been undertaken at all; and to those others whose help in one way or another advanced its progress. With the completion of this volume my obligations are further extended. I should like to express or repeat my thanks to the following for the help that they have given and the premissions they have granted.

Christabel Lady Aberconway; Lord Annan; Dr Igor Anrep; . . .

—Quentin Bell, *Virginia Woolf: A Biography* (1972)

Currently, the book's Web site is part of the author's Web site, which is located at <http://www.ecs.csun.edu/~dsalomon/>. Domain `DavidSalomon.name` has been reserved and will always point to any future location of the Web site. The author's email address is `dsalomon@csun.edu`, but email sent to `<anyname>@DavidSalomon.name` will be forwarded to the author.

Those interested in data compression in general should consult the short section titled "Joining the Data Compression Community," at the end of the book, as well as the following resources:

- <http://compression.ca/>,
- <http://www-isl.stanford.edu/~gray/iii.html>,
- http://www.hn.is.uec.ac.jp/~arimura/compression_links.html, and
- <http://datacompression.info/>.

(URLs are notoriously short lived, so search the Internet).

People err who think my art comes easily to me.

—Wolfgang Amadeus Mozart

Lakeside, California

David Salomon

Contents

	Preface to the New Handbook	vii
	Preface to the Fourth Edition	xiii
	Introduction	1
1	Basic Techniques	25
1.1	Intuitive Compression	25
1.2	Run-Length Encoding	31
1.3	RLE Text Compression	31
1.4	RLE Image Compression	36
1.5	Move-to-Front Coding	45
1.6	Scalar Quantization	49
1.7	Recursive Range Reduction	51
2	Basic VL Codes	55
2.1	Codes, Fixed- and Variable-Length	60
2.2	Prefix Codes	62
2.3	VLCs, Entropy, and Redundancy	63
2.4	Universal Codes	68
2.5	The Kraft–McMillan Inequality	69
2.6	Tunstall Code	72
2.7	Schalkwijk’s Coding	74
2.8	Tjalkens–Willems V-to-B Coding	79
2.9	Phased-In Codes	81
2.10	Redundancy Feedback (RF) Coding	85
2.11	Recursive Phased-In Codes	89
2.12	Self-Delimiting Codes	92

www.it-ebooks.info

3	Advanced VL Codes	95
3.1	VLCs for Integers	95
3.2	Start-Step-Stop Codes	97
3.3	Start/Stop Codes	99
3.4	Elias Codes	101
3.5	RBUC, Recursive Bottom-Up Coding	107
3.6	Levenstein Code	110
3.7	Even-Rodeh Code	111
3.8	Punctured Elias Codes	112
3.9	Other Prefix Codes	113
3.10	Ternary Comma Code	116
3.11	Location Based Encoding (LBE)	117
3.12	Stout Codes	119
3.13	Boldi-Vigna (ζ) Codes	122
3.14	Yamamoto's Recursive Code	125
3.15	VLCs and Search Trees	128
3.16	Taboo Codes	131
3.17	Wang's Flag Code	135
3.18	Yamamoto Flag Code	137
3.19	Number Bases	141
3.20	Fibonacci Code	143
3.21	Generalized Fibonacci Codes	147
3.22	Goldbach Codes	151
3.23	Additive Codes	157
3.24	Golomb Code	160
3.25	Rice Codes	166
3.26	Subexponential Code	170
3.27	Codes Ending with "1"	171
3.28	Interpolative Coding	172
4	Robust VL Codes	177
4.1	Codes For Error Control	177
4.2	The Free Distance	183
4.3	Synchronous Prefix Codes	184
4.4	Resynchronizing Huffman Codes	190
4.5	Bidirectional Codes	193
4.6	Symmetric Codes	202
4.7	VLEC Codes	204

5	Statistical Methods	211
5.1	Shannon-Fano Coding	211
5.2	Huffman Coding	214
5.3	Adaptive Huffman Coding	234
5.4	MNP5	240
5.5	MNP7	245
5.6	Reliability	247
5.7	Facsimile Compression	248
5.8	PK Font Compression	258
5.9	Arithmetic Coding	264
5.10	Adaptive Arithmetic Coding	276
5.11	The QM Coder	280
5.12	Text Compression	290
5.13	The Hutter Prize	290
5.14	PPM	292
5.15	PAQ	314
5.16	Context-Tree Weighting	320
6	Dictionary Methods	329
6.1	String Compression	331
6.2	Simple Dictionary Compression	333
6.3	LZ77 (Sliding Window)	334
6.4	LZSS	339
6.5	LZPP	344
6.6	Repetition Times	348
6.7	QIC-122	350
6.8	LZX	352
6.9	LZ78	354
6.10	LZFG	358
6.11	LZRW1	361
6.12	LZRW4	364
6.13	LZW	365
6.14	UNIX Compression (LZC)	375
6.15	LZMW	377
6.16	LZAP	378
6.17	LZJ	380
6.18	LZY	383
6.19	LZP	384
6.20	Repetition Finder	391
6.21	GIF Images	394
6.22	RAR and WinRAR	395
6.23	The V.42bis Protocol	398
6.24	Various LZ Applications	399
6.25	Deflate: Zip and Gzip	399
6.26	LZMA and 7-Zip	411
6.27	PNG	416
6.28	XML Compression: XMill	421

6.29	EXE Compressors	423
6.30	Off-Line Dictionary-Based Compression	424
6.31	DCA, Compression with Antidictionaries	430
6.32	CRC	434
6.33	Summary	437
6.34	Data Compression Patents	437
6.35	A Unification	439
7	Image Compression	443
7.1	Pixels	444
7.2	Image Types	446
7.3	Introduction	447
7.4	Approaches to Image Compression	453
7.5	Intuitive Methods	466
7.6	Image Transforms	467
7.7	Orthogonal Transforms	472
7.8	The Discrete Cosine Transform	480
7.9	Test Images	517
7.10	JPEG	520
7.11	JPEG-LS	541
7.12	Adaptive Linear Prediction and Classification	547
7.13	Progressive Image Compression	549
7.14	JBIG	557
7.15	JBIG2	567
7.16	Simple Images: EIDAC	577
7.17	Block Matching	579
7.18	Grayscale LZ Image Compression	582
7.19	Vector Quantization	588
7.20	Adaptive Vector Quantization	598
7.21	Block Truncation Coding	603
7.22	Context-Based Methods	609
7.23	FELICS	612
7.24	Progressive FELICS	615
7.25	MLP	619
7.26	Adaptive Golomb	633
7.27	PPPM	635
7.28	CALIC	636
7.29	Differential Lossless Compression	640
7.30	DPCM	641
7.31	Context-Tree Weighting	646
7.32	Block Decomposition	647
7.33	Binary Tree Predictive Coding	652
7.34	Quadrees	658
7.35	Quadrisection	676
7.36	Space-Filling Curves	683
7.37	Hilbert Scan and VQ	684
7.38	Finite Automata Methods	695
7.39	Iterated Function Systems	711
7.40	Spatial Prediction	725
7.41	Cell Encoding	729

8	Wavelet Methods	731
8.1	Fourier Transform	732
8.2	The Frequency Domain	734
8.3	The Uncertainty Principle	737
8.4	Fourier Image Compression	740
8.5	The CWT and Its Inverse	743
8.6	The Haar Transform	749
8.7	Filter Banks	767
8.8	The DWT	777
8.9	Multiresolution Decomposition	790
8.10	Various Image Decompositions	791
8.11	The Lifting Scheme	798
8.12	The IWT	809
8.13	The Laplacian Pyramid	811
8.14	SPIHT	815
8.15	CREW	827
8.16	EZW	827
8.17	DjVu	831
8.18	WSQ, Fingerprint Compression	834
8.19	JPEG 2000	840
9	Video Compression	855
9.1	Analog Video	855
9.2	Composite and Components Video	861
9.3	Digital Video	863
9.4	History of Video Compression	867
9.5	Video Compression	869
9.6	MPEG	880
9.7	MPEG-4	902
9.8	H.261	907
9.9	H.264	910
9.10	H.264/AVC Scalable Video Coding	922
9.11	VC-1	927
10	Audio Compression	953
10.1	Sound	954
10.2	Digital Audio	958
10.3	The Human Auditory System	961
10.4	WAVE Audio Format	969
10.5	μ -Law and A-Law Companding	971
10.6	ADPCM Audio Compression	977
10.7	MLP Audio	979
10.8	Speech Compression	984
10.9	Shorten	992
10.10	FLAC	996
10.11	WavPack	1007
10.12	Monkey's Audio	1017
10.13	MPEG-4 Audio Lossless Coding (ALS)	1018
10.14	MPEG-1/2 Audio Layers	1030
10.15	Advanced Audio Coding (AAC)	1055
10.16	Dolby AC-3	1082

11 Other Methods	1087
11.1 The Burrows-Wheeler Method	1089
11.2 Symbol Ranking	1094
11.3 ACB	1098
11.4 Sort-Based Context Similarity	1105
11.5 Sparse Strings	1110
11.6 Word-Based Text Compression	1121
11.7 Textual Image Compression	1128
11.8 Dynamic Markov Coding	1134
11.9 FHM Curve Compression	1142
11.10 Sequitur	1145
11.11 Triangle Mesh Compression: Edgebreaker	1150
11.12 SCSU: Unicode Compression	1161
11.13 Portable Document Format (PDF)	1167
11.14 File Differencing	1169
11.15 Hyperspectral Data Compression	1180
11.16 Stuffed	1191
A Information Theory	1199
A.1 Information Theory Concepts	1199
Answers to Exercises	1207
Bibliography	1271
Glossary	1303
Joining the Data Compression Community	1329
Index	1331

Content comes first...yet excellent design can catch
people's eyes and impress the contents on their memory.

—Hideki Nakajima



9.5 Video Compression

Video compression is based on two principles. The first is the spatial redundancy that exists in each frame because of pixel correlation. The second is the fact that most of the time, a video frame is very similar to its immediate neighbors. This is called *temporal redundancy*. A typical technique for video compression should therefore start by encoding the first frame using a still image compression method. It should then encode several successive frames by identifying the differences between a frame and its predecessor, and encoding these differences. If a frame is very different from its predecessor (as happens with the first frame of a shot), it should be coded independently of any other frame. In the video compression literature, a frame that is coded from its predecessor is called *inter frame* (or just *inter*), while a frame that is coded independently is called *intra frame* (or just *intra*).

Video compression is normally lossy. Encoding a frame F_i in terms of its predecessor F_{i-1} introduces some distortions. As a result, encoding the next frame F_{i+1} in terms of (the already distorted) F_i increases the distortion. Even in lossless video compression, a frame may lose some bits. This may happen during transmission or after a long shelf stay. If a frame F_i has lost some bits, then all the frames following it, up to the next intra frame, will be decoded improperly, possibly leading to accumulated errors. This is why intra frames should be used from time to time inside a sequence, not just at its beginning. An intra frame is labeled I , and an inter frame is labeled P (for *predictive*).

Once this idea is grasped, it is possible to generalize the concept of an inter frame. Such a frame can be coded based on one of its predecessors and also on one of its *successors*. We know that an encoder should not use any information that is not available to the decoder, but video compression is special because of the large quantities of data involved. We usually don't mind if the encoder is slow, but the decoder has to be fast. A typical case is video recorded on a disk or on a DVD, to be played back. The encoder can take minutes or hours to encode the data. The decoder, however, has to play it back at the correct frame rate (so many frames per second), so it has to be fast. This is why a typical video decoder works in parallel. It has several decoding circuits working simultaneously on several frames.

With this in mind it is easy to imagine a situation where the encoder encodes frame 2 based on both frames 1 and 3, and writes the frames on the compressed stream in the order 1, 3, 2. The decoder reads them in this order, decodes frames 1 and 3 in parallel, outputs frame 1, then decodes frame 2 based on frames 1 and 3. Naturally, the frames should be clearly tagged (or time stamped). A frame that is encoded based on both past and future frames is labeled B (for *bidirectional*).

Predicting a frame based on its successor makes sense in cases where the movement of an object in the picture gradually uncovers a background area. Such an area may be only partly known in the current frame but may be better known in the next frame. Thus, the next frame is a natural candidate for predicting this area in the current frame.

The idea of a B frame is so useful that most frames in a compressed video presentation may be of this type. We therefore end up with a sequence of compressed frames of the three types I , P , and B . An I frame is decoded independently of any other frame. A P frame is decoded using the preceding I or P frame. A B frame is decoded using the preceding *and* following I or P frames. Figure 9.9a shows a sequence of such frames

in the order in which they are generated by the encoder (and input by the decoder). Figure 9.9b shows the same sequence in the order in which the frames are output by the decoder and displayed. The frame labeled 2 should be displayed after frame 5, so each frame should have two time stamps, its coding time and its display time.

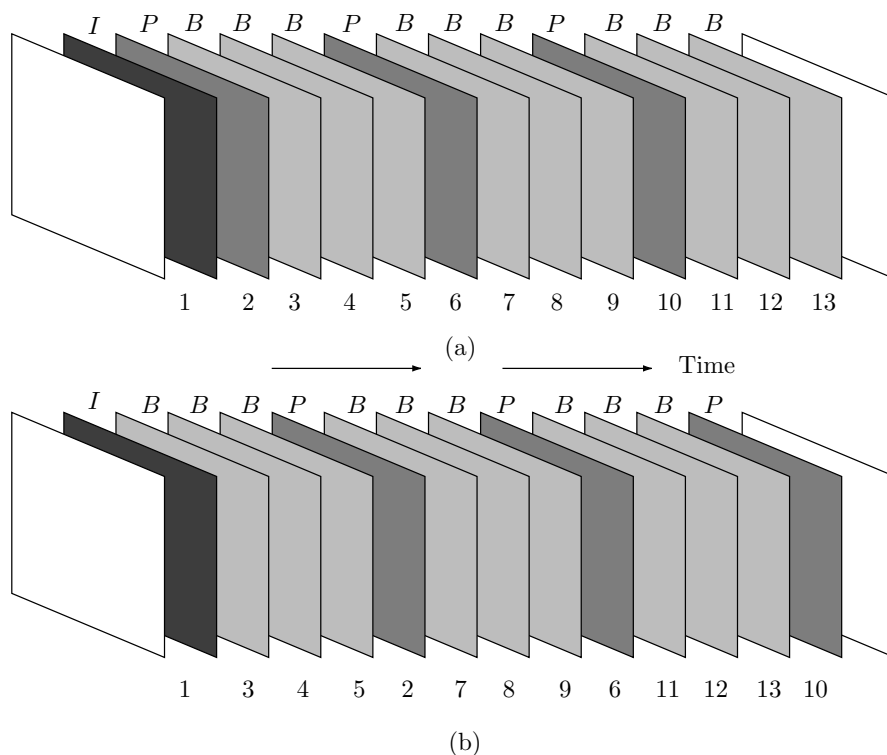


Figure 9.9: (a) Coding Order. (b) Display Order.

We start with a few intuitive video compression methods.

Subsampling: The encoder selects every other frame and writes it on the compressed stream. This yields a compression factor of 2. The decoder inputs a frame and duplicates it to create two frames.

Differencing: A frame is compared to its predecessor. If the difference between them is small (just a few pixels), the encoder encodes the pixels that are different by writing three numbers on the compressed stream for each pixel: its image coordinates, and the difference between the values of the pixel in the two frames. If the difference between the frames is large, the current frame is written on the output in raw format. Compare this method with relative encoding, Section 1.3.1.

A lossy version of differencing looks at the amount of change in a pixel. If the difference between the intensities of a pixel in the preceding frame and in the current

frame is smaller than a certain (user controlled) threshold, the pixel is not considered different.

Block Differencing: This is a further improvement of differencing. The image is divided into blocks of pixels, and each block B in the current frame is compared with the corresponding block P in the preceding frame. If the blocks differ by more than a certain amount, then B is compressed by writing its image coordinates, followed by the values of all its pixels (expressed as differences) on the compressed stream. The advantage is that the block coordinates are small numbers (smaller than a pixel's coordinates), and these coordinates have to be written just once for the entire block. On the downside, the values of all the pixels in the block, even those that haven't changed, have to be written on the output. However, since these values are expressed as differences, they are small numbers. Consequently, this method is sensitive to the block size.

Motion Compensation: Anyone who has watched movies knows that the difference between consecutive frames is small because it is the result of moving the scene, the camera, or both between frames. This feature can therefore be exploited to achieve better compression. If the encoder discovers that a part P of the preceding frame has been rigidly moved to a different location in the current frame, then P can be compressed by writing the following three items on the compressed stream: its previous location, its current location, and information identifying the boundaries of P . The following discussion of motion compensation is based on [Manning 98].

In principle, such a part can have any shape. In practice, we are limited to equal-size blocks (normally square but can also be rectangular). The encoder scans the current frame block by block. For each block B it searches the preceding frame for an identical block C (if compression is to be lossless) or for a similar one (if it can be lossy). Finding such a block, the encoder writes the difference between its past and present locations on the output. This difference is of the form

$$(C_x - B_x, C_y - B_y) = (\Delta x, \Delta y),$$

so it is called a *motion vector*. Figure 9.10a,b shows an example where the seagull sweeps to the right while the rest of the image slides slightly to the left because of camera movement.

Motion compensation is effective if objects are just translated, not scaled or rotated. Drastic changes in illumination from frame to frame also reduce the effectiveness of this method. In general, motion compensation is lossy. The following paragraphs discuss the main aspects of motion compensation in detail.

Frame Segmentation: The current frame is divided into equal-size nonoverlapping blocks. The blocks may be squares or rectangles. The latter choice assumes that motion in video is mostly horizontal, so horizontal blocks reduce the number of motion vectors without degrading the compression ratio. The block size is important, because large blocks reduce the chance of finding a match, and small blocks result in many motion vectors. In practice, block sizes that are integer powers of 2, such as 8 or 16, are used, since this simplifies the software.

Search Threshold: Each block B in the current frame is first compared to its counterpart C in the preceding frame. If they are identical, or if the difference between them is less than a preset threshold, the encoder assumes that the block hasn't been



Figure 9.10: Motion Compensation.

moved.

Block Search: This is a time-consuming process, and so has to be carefully designed. If B is the current block in the current frame, then the previous frame has to be searched for a block identical to or very close to B . The search is normally restricted to a small area (called the *search area*) around B , defined by the *maximum displacement* parameters dx and dy . These parameters specify the maximum horizontal and vertical distances, in pixels, between B and any matching block in the previous frame. If B is a square with side b , the search area will contain $(b + 2dx)(b + 2dy)$ pixels (Figure 9.11) and will consist of $(2dx + 1)(2dy + 1)$ distinct, overlapping $b \times b$ squares. The number of candidate blocks in this area is therefore proportional to $dx \cdot dy$.

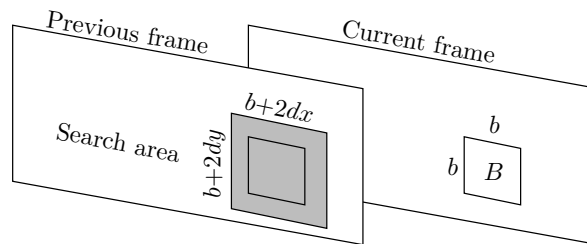


Figure 9.11: Search Area.

Distortion Measure: This is the most sensitive part of the encoder. The distortion measure selects the best match for block B . It has to be simple and fast, but also reliable. A few choices—similar to the ones of Section 7.19—are discussed here.

The *mean absolute difference* (or *mean absolute error*) calculates the average of the absolute differences between a pixel B_{ij} in B and its counterpart C_{ij} in a candidate block C :

$$\frac{1}{b^2} \sum_{i=1}^b \sum_{j=1}^b |B_{ij} - C_{ij}|.$$

This involves b^2 subtractions and absolute value operations, b^2 additions, and one division. This measure is calculated for each of the $(2dx + 1)(2dy + 1)$ distinct, overlapping $b \times b$ candidate blocks, and the smallest distortion (say, for block C_k) is examined. If it is smaller than the search threshold, then C_k is selected as the match for B . Otherwise, there is no match for B , and B has to be encoded without motion compensation.

- ◇ **Exercise 9.5:** How can such a thing happen? How can a block in the current frame match nothing in the preceding frame?

The *mean square difference* is a similar measure, where the square, rather than the absolute value, of a pixel difference is calculated:

$$\frac{1}{b^2} \sum_{i=1}^b \sum_{j=1}^b (B_{ij} - C_{ij})^2.$$

The *pel difference classification* (PDC) measure counts how many differences $|B_{ij} - C_{ij}|$ are smaller than the PDC parameter p .

The *integral projection* measure computes the sum of a row of B and subtracts it from the sum of the corresponding row of C . The absolute value of the difference is added to the absolute value of the difference of the columns sum:

$$\sum_{i=1}^b \left| \sum_{j=1}^b B_{ij} - \sum_{j=1}^b C_{ij} \right| + \sum_{j=1}^b \left| \sum_{i=1}^b B_{ij} - \sum_{i=1}^b C_{ij} \right|.$$

Suboptimal Search Methods: These methods search some, instead of all, the candidate blocks in the $(b + 2dx)(b + 2dy)$ area. They speed up the search for a matching block, at the expense of compression efficiency. Several such methods are discussed in detail in Section 9.5.1.

Motion Vector Correction: Once a block C has been selected as the best match for B , a motion vector is computed as the difference between the upper-left corner of C and the upper-left corner of B . Regardless of how the matching was determined, the motion vector may be wrong because of noise, local minima in the frame, or because the matching algorithm is not perfect. It is possible to apply smoothing techniques to the motion vectors after they have been calculated, in an attempt to improve the matching. Spatial correlations in the image suggest that the motion vectors should also be correlated. If certain vectors are found to violate this, they can be corrected.

This step is costly and may even backfire. A video presentation may involve slow, smooth motion of most objects, but also swift, jerky motion of some small objects. Correcting motion vectors may interfere with the motion vectors of such objects and cause distortions in the compressed frames.

Coding Motion Vectors: A large part of the current frame (perhaps close to half of it) may be converted to motion vectors, which is why the way these vectors are encoded is crucial; it must also be lossless. Two properties of motion vectors help in encoding them: (1) They are correlated and (2) their distribution is nonuniform. As we scan the frame block by block, adjacent blocks normally have motion vectors that don't

differ by much; they are correlated. The vectors also don't point in all directions. There are often one or two preferred directions in which all or most motion vectors point; the vectors are nonuniformly distributed.

No single method has proved ideal for encoding the motion vectors. Arithmetic coding, adaptive Huffman coding, and various prefix codes have been tried, and all seem to perform well. Here are two different methods that may perform better:

1. Predict a motion vector based on its predecessors in the same row and its predecessors in the same column of the current frame. Calculate the difference between the prediction and the actual vector, and Huffman encode it. This algorithm is important. It is used in MPEG and other compression methods.
2. Group the motion vectors in blocks. If all the vectors in a block are identical, the block is encoded by encoding this vector. Other blocks are encoded as in 1 above. Each encoded block starts with a code identifying its type.

Coding the Prediction Error: Motion compensation is lossy, since a block B is normally matched to a somewhat different block C . Compression can be improved by coding the difference between the current uncompressed and compressed frames on a block by block basis and only for blocks that differ much. This is usually done by transform coding. The difference is written on the output, following each frame, and is used by the decoder to improve the frame after it has been decoded.

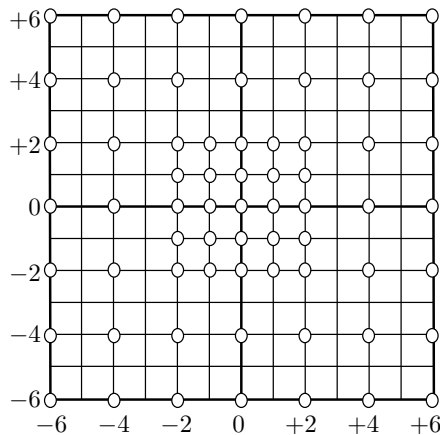
9.5.1 Suboptimal Search Methods

Video compression requires many steps and computations, so researchers have been looking for optimizations and faster algorithms, especially for steps that involve many calculations. One such step is the search for a block C in the previous frame to match a given block B in the current frame. An exhaustive search is time-consuming, so it pays to look for suboptimal search methods that search just some of the many overlapping candidate blocks. These methods do not always find the best match, but can generally speed up the entire compression process while incurring only a small loss of compression efficiency.

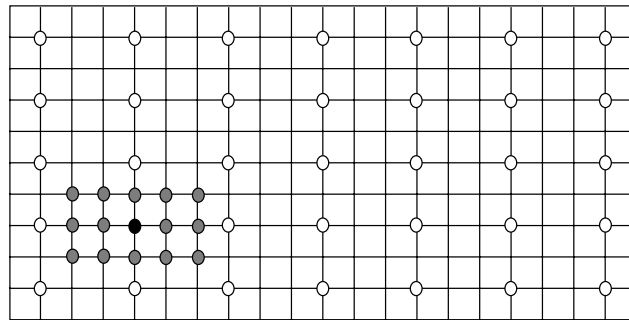
Signature-Based Methods: Such a method performs a number of steps, restricting the number of candidate blocks in each step. In the first step, all the candidate blocks are searched using a simple, fast distortion measure such as pel difference classification. Only the best matched blocks are included in the next step, where they are evaluated by a more restrictive distortion measure, or by the same measure but with a smaller parameter. A signature method may involve several steps, using different distortion measures in each.

Distance-Diluted Search: We know from experience that fast-moving objects look blurred in an animation, even if they are sharp in all the frames. This suggests a way to lose data. We may require a good block match for slow-moving objects, but allow for a worse match for fast-moving ones. The result is a block matching algorithm that searches all the blocks close to B , but fewer and fewer blocks as the search gets farther away from B . Figure 9.12a shows how such a method may work for maximum displacement parameters $dx = dy = 6$. The total number of blocks C being searched goes from $(2dx + 1) \cdot (2dy + 1) = 13 \times 13 = 169$ to just 65, less than 39%!

Locality-Based Search: This method is based on the assumption that once a good match has been found, even better matches are likely to be located near it (remember



(a)



O — first wave. ● — best match of first wave. ● — second wave.

(b)

Figure 9.12: (a) Distance-Diluted Search for $dx = dy = 6$. (b) A Locality Search.

that the blocks C searched for matches highly overlap). An obvious algorithm is to start searching for a match in a sparse set of blocks, then use the best-matched block C as the center of a second wave of searches, this time in a denser set of blocks. Figure 9.12b shows two waves of search, the first considers widely-spaced blocks, selecting one as the best match. The second wave searches every block in the vicinity of the best match.

Quadrant Monotonic Search: This is a variant of a locality-based search. It starts with a sparse set of blocks C that are searched for a match. The distortion measure is computed for each of those blocks, and the result is a set of distortion values. The idea is that the distortion values increase as we move away from the best match. By examining the set of distortion values obtained in the first step, the second step may predict where the best match is likely to be found. Figure 9.13 shows how a search of a

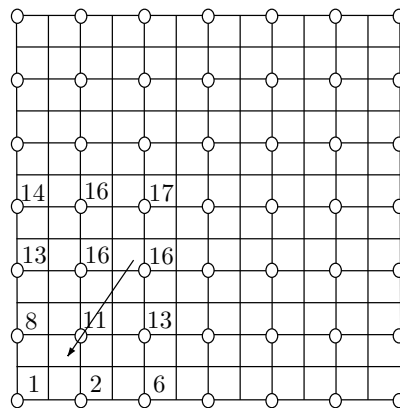


Figure 9.13: Quadrant Monotonic Search.

region of 4×3 blocks suggests a well-defined direction in which to continue searching.

This method is less reliable than the previous ones because the direction proposed by the set of distortion values may lead to a local best block, whereas *the* best block may be located elsewhere.

Dependent Algorithms: As mentioned earlier, motion in a frame is the result of either camera movement or object movement. If we assume that objects in the frame are bigger than a block, we conclude that it is reasonable to expect the motion vectors of adjacent blocks to be correlated. The search algorithm can therefore start by estimating the motion vector of a block B from the motion vectors that have already been found for its neighbors, then improve this estimate by comparing B to some candidate blocks C . This is the basis of several *dependent algorithms*, which can be spatial or temporal.

Spatial dependency: In a spatial dependent algorithm, the neighbors of a block B in the current frame are used to estimate the motion vector of B . These, of course, must be neighbors whose motion vectors have already been computed. Most blocks have eight neighbors each, but considering all eight may not be the best strategy (also, when a block B is considered, only some of its neighbors may have their motion vectors already computed). If blocks are matched in raster order, then it makes sense to use one, two, or three previously-matched neighbors, as shown in Figure 9.14a,b,c. Because of symmetry, however, it is better to use four symmetric neighbors, as in Figure 9.14d,e. This can be done by a three-pass method that scans blocks as in Figure 9.14f. The first pass scans all the blocks shown in black (one-quarter of the blocks in the frame). Motion vectors for those blocks are calculated by some other method. Pass two scans the blocks shown in gray (25% of the blocks) and estimates a motion vector for each using the motion vectors of its four corner neighbors. The white blocks (the remaining 50%) are scanned in the third pass, and the motion vector of each is estimated using the motion vectors of its neighbors on all four sides. If the motion vectors of the neighbors are very different, they should not be used, and a motion vector for block B is computed with a different method.

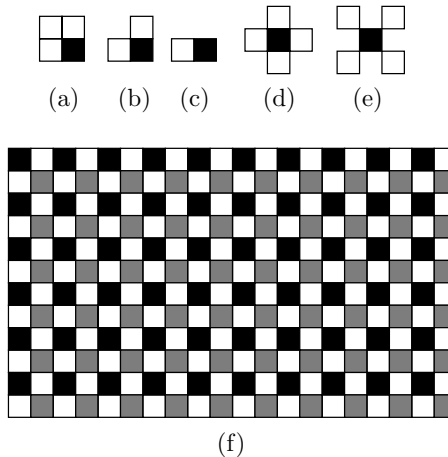


Figure 9.14: Strategies for Spatial Dependent Search Algorithms.

Temporal dependency: The motion vector of block B in the current frame can be estimated as the motion vector of the same block in the previous frame. This makes sense if we can assume uniform motion. After the motion vector of B is estimated this way, it should be improved and corrected using other methods.

More Quadrant Monotonic Search Methods: The following suboptimal block matching methods use the main assumption of the quadrant monotonic search method.

Two-Dimensional Logarithmic Search: This multistep method reduces the search area in each step until it shrinks to one block. We assume that the current block B is located at position (a, b) in the current frame. This position becomes the initial center of the search. The algorithm uses a distance parameter d that defines the search area. This parameter is user-controlled with a default value. The search area consists of the $(2d + 1) \times (2d + 1)$ blocks centered on the current block B .

Step 1: A step size s is computed by

$$s = 2^{\lfloor \log_2 d \rfloor - 1},$$

and the algorithm compares B with the five blocks at positions (a, b) , $(a, b + s)$, $(a, b - s)$, $(a + s, b)$, and $(a - s, b)$ in the previous frame. These five blocks form the pattern of a plus sign “+”.

Step 2: The best match among the five blocks is selected. We denote the position of this block by (x, y) . If $(x, y) = (a, b)$, then s is halved (this is the reason for the name *logarithmic*). Otherwise, s stays the same, and the center (a, b) of the search is moved to (x, y) .

Step 3: If $s = 1$, then the nine blocks around the center (a, b) of the search are searched, and the best match among them becomes the result of the algorithm. Otherwise the algorithm goes to Step 2.

Any blocks that need be searched but are outside the search area are ignored and are not used in the search. Figure 9.15 illustrates the case where $d = 8$. For simplicity

we assume that the current block B has frame coordinates $(0, 0)$. The search is limited to the (17×17) -block area centered on block B . Step 1 computes

$$s = 2^{\lceil \log_2 8 \rceil - 1} = 2^{3-1} = 4,$$

and searches the five blocks (labeled 1) at locations $(0, 0)$, $(4, 0)$, $(-4, 0)$, $(0, 4)$, and $(0, -4)$. We assume that the best match of these five is at $(0, 4)$, so this becomes the new center of the search, and the three blocks (labeled 2) at locations $(4, -4)$, $(4, 4)$, and $(8, 0)$ are searched in the second step.

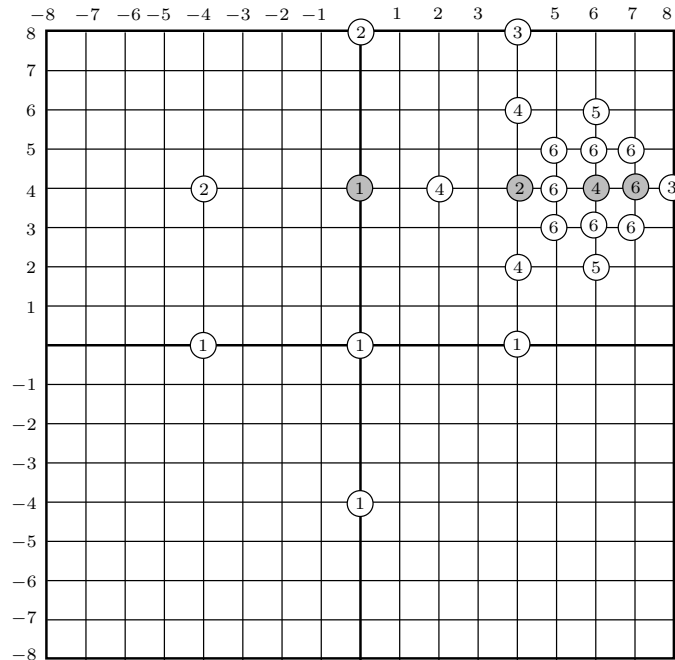


Figure 9.15: The Two-Dimensional Logarithmic Search Method.

Assuming that the best match among these three is at location $(4, 4)$, the next step searches the two blocks labeled 3 at locations $(8, 4)$ and $(4, 8)$, the block (labeled 2) at $(4, 4)$, and the “1” blocks at $(0, 4)$ and $(4, 0)$.

- ◇ **Exercise 9.6:** Assuming that $(4, 4)$ is again the best match, use the figure to describe the rest of the search.

Three-Step Search: This is somewhat similar to the two-dimensional logarithmic search. In each step it tests eight blocks, instead of four, around the center of search, then halves the step size. If $s = 3$ initially, the algorithm terminates after three steps, hence its name.

Orthogonal Search: This is a variation of both the two-dimensional logarithmic search and the three-step search. Each step of the orthogonal search involves a horizontal and a vertical search. The step size s is initialized to $\lfloor (d+1)/2 \rfloor$, and the block at the center of the search and two candidate blocks located on either side of it at a distance of s are searched. The location of smallest distortion becomes the center of the vertical search, where two candidate blocks above and below the center, at distances of s , are searched. The best of these locations becomes the center of the next search. If the step size s is 1, the algorithm terminates and returns the best block found in the current step. Otherwise, s is halved, and a new, similar set of horizontal and vertical searches is performed.

One-at-a-Time Search: In this type of search there are again two steps, a horizontal and a vertical. The horizontal step searches all the blocks in the search area whose y coordinates equal that of block B (i.e., that are located on the same horizontal axis as B). Assuming that block H has the minimum distortion among those, the vertical step searches all the blocks on the same vertical axis as H and returns the best of them. A variation repeats this on smaller and smaller search areas.

Cross Search: All the steps of this algorithm, except the last one, search the five blocks at the edges of a multiplication sign “ $\times$ ”. The step size is halved in each step until it gets down to 1. At the last step, the plus sign “ $+$ ” is used to search the areas located around the top-left and bottom-right corners of the preceding step.

This has been a survey of quadrant monotonic search methods. We follow with an outline of two advanced search methods.

Hierarchical Search Methods: Hierarchical methods take advantage of the fact that block matching is sensitive to the block size. A hierarchical search method starts with large blocks and uses their motion vectors as starting points for more searches with smaller blocks. Large blocks are less likely to stumble on a local maximum, while a small block generally produces a better motion vector. A hierarchical search method is therefore computationally intensive, and the main point is to speed it up by reducing the number of operations. This can be done in several ways as follows:

1. In the initial steps, when the blocks are still large, search just a sample of blocks. The resulting motion vectors are not the best, but they are only going to be used as starting points for better ones.
2. When searching large blocks, skip some of the pixels of a block. The algorithm may, for example, examine just one-quarter of the pixels of the large blocks, one-half of the pixels of smaller blocks, and so on.
3. Select the block sizes such that the block used in step i is divided into several (typically four or nine) blocks used in the following step. This way, a single motion vector computed in step i can be used as an estimate for several better motion vectors in step $i+1$.

Multidimensional Search Space Methods: These methods are more complex. When searching for a match for block B , such a method looks for matches that are rotations or zooms of B , not just translations.

A multidimensional search space method may also find a block C that matches B but has different lighting conditions. This is useful when an object moves among areas that are illuminated differently. All the methods discussed so far compare two blocks by comparing the luminance values of corresponding pixels. Two blocks B and C that

contain the same objects but differ in luminance would be declared different by such methods.

When a multidimensional search space method finds a block C that matches B but has different luminance, it may declare C the match of B and append a luminance value to the compressed frame B . This value (which may be negative) is added by the decoder to the pixels of the decompressed frame, to bring them back to their original values.

A multidimensional search space method may also compare a block B to rotated versions of the candidate blocks C . This is useful if objects in the video presentation may be rotated in addition to being moved. The algorithm may also try to match a block B to a block C containing a scaled version of the objects in B . If, for example, B is of size 8×8 pixels, the algorithm may consider blocks C of size 12×12 , shrink each to 8×8 , and compare it to B .

This kind of block search requires many extra operations and comparisons. We say that it increases the size of the *search space* significantly, hence the name *multidimensional search space*. It seems that at present there is no multidimensional search space method that can account for scaling, rotation, and changes in illumination and also be fast enough for practical use.

Television? No good will come of this device. The word is half Greek and half Latin.
—C. P. Scott

9.6 MPEG

Starting in 1988, the MPEG project was developed by a group of hundreds of experts under the auspices of the ISO (International Standardization Organization) and the IEC (International Electrotechnical Committee). The name MPEG is an acronym for Moving Pictures Experts Group. MPEG is a method for video compression, which involves the compression of digital images and sound, as well as synchronization of the two. There currently are several MPEG standards. MPEG-1 is intended for intermediate data rates, on the order of 1.5 Mbit/sec. MPEG-2 is intended for high data rates of at least 10 Mbit/sec. MPEG-3 was intended for HDTV compression but was found to be redundant and was merged with MPEG-2. MPEG-4 is intended for very low data rates of less than 64 Kbit/sec. A third international body, the ITU-T, has been involved in the design of both MPEG-2 and MPEG-4. This section concentrates on MPEG-1 and discusses only its image compression features.

The formal name of MPEG-1 is the international standard for moving picture video compression, IS11172-2. Like other standards developed by the ITU and ISO, the document describing MPEG-1 has *normative* and *informative* sections. A normative section is part of the specification of the standard. It is intended for implementers, is written in a precise language, and should be strictly adhered to when implementing the standard on actual computer platforms. An informative section, on the other hand, illustrates concepts discussed elsewhere, explains the reasons that led to certain choices and decisions, and contains background material. An example of a normative section is the various tables of variable codes used in MPEG. An example of an informative section is the algorithm used by MPEG to estimate motion and match blocks. MPEG does not

require any particular algorithm, and an MPEG encoder can use any method to match blocks. The section itself simply describes various alternatives.

The discussion of MPEG in this section is informal. The first subsection (main components) describes all the important terms, principles, and codes used in MPEG-1. The subsections that follow go into more details, especially in the description and listing of the various parameters and variable-length codes.

The importance of a widely accepted standard for video compression is apparent from the fact that many manufacturers (of computer games, DVD movies, digital television, and digital recorders, among others) implemented MPEG-1 and started using it even before it was finally approved by the MPEG committee. This also was one reason why MPEG-1 had to be frozen at an early stage and MPEG-2 had to be developed to accommodate video applications with high data rates.

There are many sources of information on MPEG. [Mitchell et al. 97] is one detailed source for MPEG-1, and the MPEG consortium [MPEG 98] contains lists of other resources. In addition, there are many Web pages with descriptions, explanations, and answers to frequently asked questions about MPEG.

To understand the meaning of the words “intermediate data rate” we consider a typical example of video with a resolution of 360×288 , a depth of 24 bits per pixel, and a refresh rate of 24 frames per second. The image part of this video requires $360 \times 288 \times 24 \times 24 = 59,719,680$ bits/s. For the audio part, we assume two sound tracks (stereo sound), each sampled at 44 kHz with 16-bit samples. The data rate is $2 \times 44,000 \times 16 = 1,408,000$ bits/s. The total is about 61.1 Mbit/s and this is supposed to be compressed by MPEG-1 to an intermediate data rate of about 1.5 Mbit/s (the size of the sound track alone), a compression factor of more than 40! Another aspect is the decoding speed. An MPEG-compressed movie may end up being stored on a CD-ROM or DVD and has to be decoded and played in real time.

MPEG uses its own vocabulary. An entire movie is considered a *video sequence*. It consists of *pictures*, each having three *components*, one luminance (*Y*) and two chrominance (*Cb* and *Cr*). The luminance component (Section 7.3) contains the black-and-white picture, and the chrominance components provide the color hue and saturation (see [Salomon 99] for a detailed discussion). Each component is a rectangular array of *samples*, and each row of the array is called a *raster line*. A *pel* is the set of three samples. The eye is sensitive to small spatial variations of luminance, but is less sensitive to similar changes in chrominance. As a result, MPEG-1 samples the chrominance components at half the resolution of the luminance component. The term *intra* is used, but *inter* and *nonintra* are used interchangeably.

The input to an MPEG encoder is called the *source data*, and the output of an MPEG decoder is the *reconstructed data*. The source data is organized in packs (Figure 9.16b), where each pack starts with a start code (32 bits) followed by a header, ends with a 32-bit end code, and contains a number of packets in between. A packet contains compressed data, either audio or video. The size of a packet is determined by the MPEG encoder according to the requirements of the storage or transmission medium, which is why a packet is not necessarily a complete video picture. It can be any part of a video picture or any part of the audio.

The MPEG decoder has three main parts, called *layers*, to decode the audio, the video, and the system data. The system layer reads and interprets the various codes

and headers in the source data, and routes the packets to either the audio or the video layers (Figure 9.16a) to be buffered and later decoded. Each of these two layers consists of several decoders that work simultaneously.

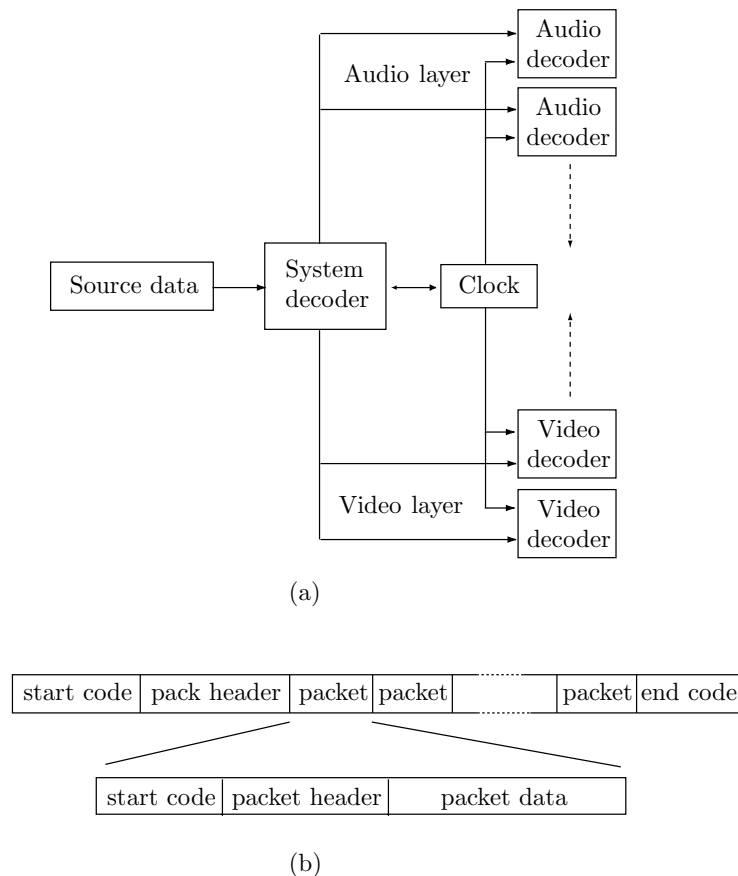


Figure 9.16: (a) MPEG Decoder Organization. (b) Source Format.

The MPEG home page is at <http://www.chiariglione.org/mpeg/index.htm>.

9.6.1 MPEG-1 Main Components

MPEG uses *I*, *P*, and *B* pictures, as discussed in Section 9.5. They are arranged in groups, where a group can be open or closed. The pictures are arranged in a certain order, called the *coding order*, but (after being decoded) they are output and displayed in a different order, called the *display order*. In a closed group, *P* and *B* pictures are decoded only from other pictures in the group. In an open group, they can be decoded from pictures outside the group. Different regions of a *B* picture may use different

pictures for their decoding. A region may be decoded from some preceding pictures, from some following pictures, from both types, or from none. Similarly, a region in a P picture may use several preceding pictures for its decoding, or use none at all, in which case it is decoded using MPEG's intra methods.

The basic building block of an MPEG picture is the *macroblock* (Figure 9.17a). It consists of a 16×16 block of luminance (grayscale) samples (divided into four 8×8 blocks) and two 8×8 blocks of the matching chrominance samples. The MPEG compression of a macroblock consists mainly in passing each of the six blocks through a discrete cosine transform, which creates decorrelated values, then quantizing and encoding the results. It is very similar to JPEG compression (Section 7.10), the main differences being that different quantization tables and different code tables are used in MPEG for intra and nonintra, and the rounding is done differently.

A picture in MPEG is organized in slices, where each slice is a contiguous set of macroblocks (in raster order) that have the same grayscale (i.e., luminance component). The concept of slices makes sense because a picture may often contain large uniform areas, causing many contiguous macroblocks to have the same grayscale. Figure 9.17b shows a hypothetical MPEG picture and how it is divided into slices. Each square in the picture is a macroblock. Notice that a slice can continue from scan line to scan line.

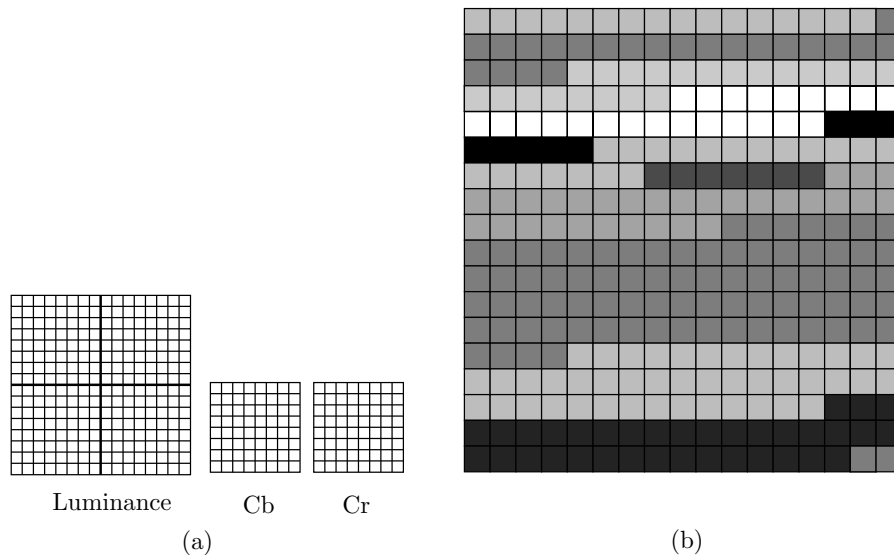


Figure 9.17: (a) A Macroblock. (b) A Possible Slice Structure.

- ◇ **Exercise 9.7:** How many samples are there in the hypothetical MPEG picture of Figure 9.17b?

When a picture is encoded in nonintra mode (i.e., it is encoded by means of another picture, normally its predecessor), the MPEG encoder generates the differences between

the pictures, then applies the DCT to the differences. In such a case, the DCT does not contribute much to the compression, because the differences are already decorrelated. Nevertheless, the DCT is useful even in this case, since it is followed by quantization, and the quantization in nonintra coding can be quite deep.

The precision of the numbers processed by the DCT in MPEG also depends on whether intra or nonintra coding is used. MPEG samples in intra coding are 8-bit unsigned integers, whereas in nonintra they are 9-bit signed integers. This is because a sample in nonintra is the difference of two unsigned integers, and may therefore be negative. The two summations of the two-dimensional DCT, Equation (7.16), can at most multiply a sample by $64 = 2^6$ and may therefore result in an $8 + 6 = 14$ -bit integer (see Exercise 7.19 for a similar case). In those summations, a sample is multiplied by cosine functions, which may result in a negative number. The result of the double sum is therefore a 15-bit signed integer. This integer is then multiplied by the factor $C_i C_j / 4$ which is at least $1/8$, thereby reducing the result to a 12-bit signed integer.

This 12-bit integer is then quantized by dividing it by a quantization coefficient (QC) taken from a quantization table. The result is, in general, a noninteger and has to be rounded. It is in quantization and rounding that information is irretrievably lost. MPEG specifies default quantization tables, but custom tables can also be used. In intra coding, rounding is done in the normal way, to the nearest integer, whereas in nonintra, rounding is done by truncating a noninteger to the nearest smaller integer. Figure 9.18a,b shows the results graphically. Notice the wide interval around zero in nonintra coding. This is the so-called *dead zone*.

The quantization and rounding steps are complex and involve more operations than just dividing a DCT coefficient by a quantization coefficient. They depend on a scale factor called `quantizer_scale`, an MPEG parameter that is an integer in the interval $[1, 31]$. The results of the quantization, and hence the compression performance, are sensitive to the value of `quantizer_scale`. The encoder can change this value from time to time and has to insert a special code in the compressed stream to indicate this.

We denote by DCT the DCT coefficient being quantized, by Q the QC from the quantization table, and by $QDCT$ the quantized value of DCT . The quantization rule for intra coding is

$$QDCT = \frac{(16 \times DCT) + \text{Sign}(DCT) \times \text{quantizer_scale} \times Q}{2 \times \text{quantizer_scale} \times Q}, \quad (9.2)$$

where the function $\text{Sign}(DCT)$ is the sign of DCT , defined by

$$\text{Sign}(DCT) = \begin{cases} +1, & \text{when } DCT > 0, \\ 0, & \text{when } DCT = 0, \\ -1, & \text{when } DCT < 0. \end{cases}$$

The second term of Equation (9.2) is called the *rounding term* and is responsible for the special form of rounding illustrated by Figure 9.18a. This is easy to see when we consider the case of a positive DCT . In this case, Equation (9.2) is reduced to the simpler expression

$$QDCT = \frac{(16 \times DCT)}{2 \times \text{quantizer_scale} \times Q} + \frac{1}{2}.$$

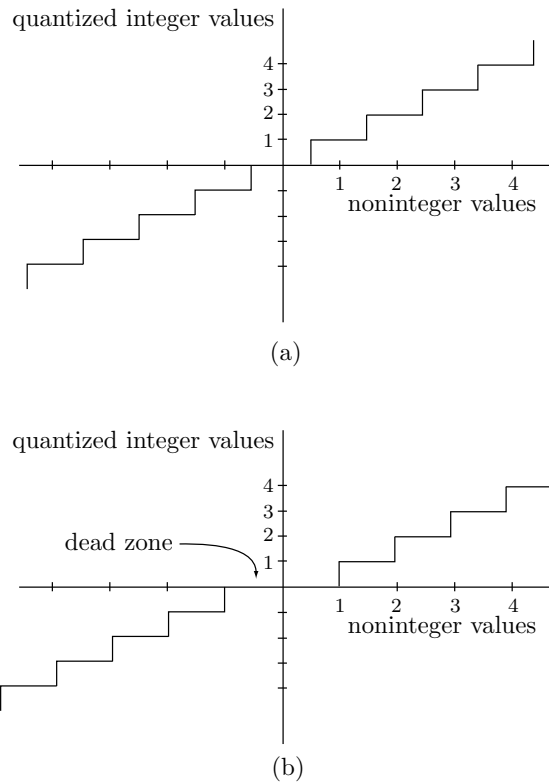


Figure 9.18: Rounding of Quantized DCT Coefficients.
(a) For Intra Coding. (b) For Nonintra Coding.

The rounding term is eliminated for nonintra coding, where quantization is done by

$$QDCT = \frac{(16 \times DCT)}{2 \times \text{quantizer_scale} \times Q}. \quad (9.3)$$

Dequantization, which is done by the decoder in preparation for the IDCT, is the inverse of quantization. For intra coding it is done by

$$DCT = \frac{(2 \times QDCT) \times \text{quantizer_scale} \times Q}{16}$$

(notice that there is no rounding term), and for nonintra it is the inverse of Equation (9.2)

$$DCT = \frac{((2 \times QDCT) + \text{Sign}(QDCT)) \times \text{quantizer_scale} \times Q}{16}.$$

The precise way to compute the IDCT is not specified by MPEG. This can lead to distortions in cases where a picture is encoded by one implementation and decoded by

another, where the IDCT is done differently. In a chain of inter pictures, where each picture is decoded by means of its neighbors, this can lead to accumulation of errors, a phenomenon known as *IDCT mismatch*. This is why MPEG requires periodic intra coding of every part of the picture. This *forced updating* has to be done at least once for every 132 *P* pictures in the sequence. In practice, forced updating is rare, since *I* pictures are fairly common, occurring every 10 to 15 pictures.

The quantized numbers *QDCT* are Huffman coded, using the nonadaptive Huffman method and Huffman code tables that were computed by gathering statistics from many training image sequences. The particular code table being used depends on the type of picture being encoded. To avoid the zero probability problem (Section 5.14), all the entries in the code tables were initialized to 1 before any statistics were collected.

Decorrelating the original pels by computing the DCT (or, in the case of inter coding, by calculating pel differences) is part of the statistical model of MPEG. The other part is the creation of a symbol set that takes advantage of the properties of Huffman coding. Section 5.2 explains that the Huffman method becomes inefficient when the data contains symbols with large probabilities. If the probability of a symbol is 0.5, it should ideally be assigned a 1-bit code. If the probability is higher, the symbol should be assigned a shorter code, but the Huffman codes are integers and hence cannot be shorter than one bit. To avoid symbols with high probability, MPEG uses an alphabet where several old symbols (i.e., several pel differences or quantized DCT coefficients) are combined to form one new symbol. An example is run lengths of zeros. After quantizing the 64 DCT coefficients of a block, many of the resulting numbers are zeros. The probability of a zero is therefore high and can easily exceed 0.5. The solution is to deal with runs of consecutive zeros. Each run becomes a new symbol and is assigned a Huffman code. This method creates a large number of new symbols, and many Huffman codes are needed as a result. Compression efficiency, however, is improved.

Table 9.19 is the default quantization coefficients table for the luminance samples in intra coding. The MPEG documentation “explains” this table by saying, “This table has a distribution of quantizing values that is roughly in accord with the frequency response of the human eye, given a viewing distance of approximately six times the screen width and a 360×240 pel picture.” Quantizing in nonintra coding is completely different, since the quantities being quantized are pel differences, and they do not have any spatial frequencies. This type of quantization is done by dividing the DCT coefficients of the differences by 16 (the default quantization table is thus flat), although custom quantization tables can also be specified.

In an *I* picture, the DC coefficients of the macroblocks are coded separately from the AC coefficients, similar to what is done in JPEG (Section 7.10.4). Figure 9.20 shows how three types of DC coefficients, for the *Y*, *Cb*, and *Cr* components of the *I* picture, are encoded separately in one stream. Each macroblock consists of four *Y* blocks, one *Cb*, and one *Cr* block, so it contributes four DC coefficients of the first type, and one coefficient of each of the other two types. A coefficient DC_i is first used to calculate a difference $\Delta DC = DC_i - P$ (where P is the previous DC coefficient of the same type), and then the difference is encoded by coding a size category followed by bits for the magnitude and sign of the difference. The size category is the number of bits required to encode the sign and magnitude of the difference ΔDC . Each size category is assigned a code. Three steps are needed to encode a DC difference ΔDC as follows (1) The

8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

Table 9.19: Default Luminance Quantization Table for Intra Coding.

size category is first determined and its code is emitted, (2) if ΔDC is negative, a 1 is subtracted from its 2's complement representation, and (3) the *size* least-significant bits of the difference are emitted. Table 9.21 summarizes the size categories, their codes, and the range of differences ΔDC for each size. Notice that the size category of zero is defined as 0. This table should be compared with Table 7.65, which lists the corresponding codes used by JPEG.

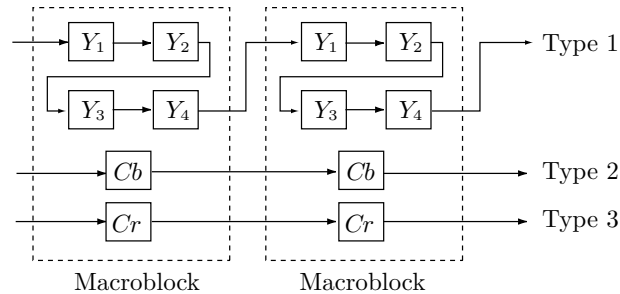


Figure 9.20: The Three Types of DC Coefficients.

Y codes	C codes	size	magnitude range
100	00	0	0
00	01	1	-1,1
01	10	2	-3...-2,2...3
101	110	3	-7...-4,4...7
110	1110	4	-15...-8,8...15
1110	11110	5	-31...-16,16...31
11110	111110	6	-63...-32,32...63
111110	1111110	7	-127...-64,64...127
1111110	11111110	8	-255...-128,128...255

Table 9.21: Codes for DC Coefficients (Luminance and Chrominance).

Examples: (1) A luminance ΔDC of 5. The number 5 can be expressed in three bits, so the size category is 3, and code 101 is emitted first. It is followed by the three least-significant bits of 5, which are 101. (2) A chrominance ΔDC of -3 . The number 3 can be expressed in 2 bits, so the size category is 2, and code 10 is first emitted. The difference -3 is represented in twos complement as $\dots 11101$. When 1 is subtracted from this number, the 2 least-significant bits are 00, and this code is emitted next.

- ◇ **Exercise 9.8:** Compute the code of luminance $\Delta DC = 0$ and the code of chrominance $\Delta DC = 4$.

The AC coefficients of an I picture (intra coding) are encoded by scanning them in the zigzag order shown in Figure 1.8b. The resulting sequence of AC coefficients consists of nonzero coefficients and run lengths of zero coefficients. A *run-level* code is output for each nonzero coefficient C , where *run* refers to the number of zero coefficients preceding C , and *level* refers to the absolute size of C . Each run-level code for a nonzero coefficient C is followed by the 1-bit sign of C (1 for negative and 0 for positive). The run-level code for the last nonzero coefficient is followed by a special 2-bit end-of-block (EOB) code. Table 9.23 lists the EOB code and the run-level codes for common values of runs and levels. Combinations of runs and levels that are not in the table are encoded by the escape code, followed by a 6-bit code for the run length and an 8- or 16-bit code for the level.

Figure 9.22a shows an example of an 8×8 block of quantized coefficients. The zigzag sequence of these coefficients is

127, 0, 0, -1 , 0, 2, 0, 0, 0, 1,

where 127 is the DC coefficient. Thus, the AC coefficients are encoded by the three run level codes (2, -1), (1, 2), (3, 1), followed by the EOB code. Table 9.23 shows that the codes are (notice the sign bits following the run-level codes)

0101 1|0001100|00111 0|10

(without the vertical bars).

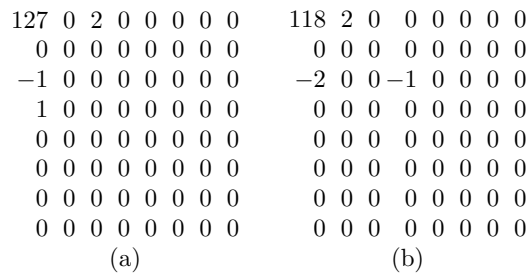


Figure 9.22: Two 8×8 Blocks of DCT Quantized Coefficients.

0/1	1s (first)	2	1/1	011s	4
0/1	11s (next)	3	1/2	0001 10s	7
0/2	0100 s	5	1/3	0010 0101 s	9
0/3	0010 1s	6	1/4	0000 0011 00s	11
0/4	0000 110s	8	1/5	0000 0001 1011 s	13
0/5	0010 0110 s	9	1/6	0000 0000 1011 0s	14
0/6	0010 0001 s	9	1/7	0000 0000 1010 1s	14
0/7	0000 0010 10s	11	1/8	0000 0000 0011 111s	16
0/8	0000 0001 1101 s	13	1/9	0000 0000 0011 110s	16
0/9	0000 0001 1000 s	13	1/10	0000 0000 0011 101s	16
0/10	0000 0001 0011 s	13	1/11	0000 0000 0011 100s	16
0/11	0000 0001 0000 s	13	1/12	0000 0000 0011 011s	16
0/12	0000 0000 1101 0s	14	1/13	0000 0000 0011 010s	16
0/13	0000 0000 1100 1s	14	1/14	0000 0000 0011 001s	16
0/14	0000 0000 1100 0s	14	1/15	0000 0000 0001 0011 s	17
0/15	0000 0000 1011 1s	14	1/16	0000 0000 0001 0010 s	17
0/16	0000 0000 0111 11s	15	1/17	0000 0000 0001 0001 s	17
0/17	0000 0000 0111 10s	15	1/18	0000 0000 0001 0000 s	17
0/18	0000 0000 0111 01s	15	2/1	0101 s	5
0/19	0000 0000 0111 00s	15	2/2	0000 100s	8
0/20	0000 0000 0110 11s	15	2/3	0000 0010 11s	11
0/21	0000 0000 0110 10s	15	2/4	0000 0001 0100 s	13
0/22	0000 0000 0110 01s	15	2/5	0000 0000 1010 0s	14
0/23	0000 0000 0110 00s	15	3/1	0011 1s	6
0/24	0000 0000 0101 11s	15	3/2	0010 0100 s	9
0/25	0000 0000 0101 10s	15	3/3	0000 0001 1100 s	13
0/26	0000 0000 0101 01s	15	3/4	0000 0000 1001 1s	14
0/27	0000 0000 0101 00s	15	4/1	0011 0s	6
0/28	0000 0000 0100 11s	15	4/2	0000 0011 11s	11
0/29	0000 0000 0100 10s	15	4/3	0000 0001 0010 s	13
0/30	0000 0000 0100 01s	15	5/1	0001 11s	7
0/31	0000 0000 0100 00s	15	5/2	0000 0010 01s	11
0/32	0000 0000 0011 000s	16	5/3	0000 0000 1001 0s	14
0/33	0000 0000 0010 111s	16	6/1	0001 01s	7
0/34	0000 0000 0010 110s	16	6/2	0000 0001 1110 s	13
0/35	0000 0000 0010 101s	16	6/3	0000 0000 0001 0100 s	17
0/36	0000 0000 0010 100s	16	7/1	0001 00s	7
0/37	0000 0000 0010 011s	16	7/2	0000 0001 0101 s	13
0/38	0000 0000 0010 010s	16	8/1	0000 111s	8
0/39	0000 0000 0010 001s	16	8/2	0000 0001 0001 s	13
0/40	0000 0000 0010 000s	16			

Table 9.23: Variable-Length Run-Level Codes (Part 1).

9/1	0000 101s	8	18/1	0000 0001 1010 s	13
9/2	0000 0000 1000 1s	14	19/1	0000 0001 1001 s	13
10/1	0010 0111 s	9	20/1	0000 0001 0111 s	13
10/2	0000 0000 1000 0s	14	21/1	0000 0001 0110 s	13
11/1	0010 0011 s	9	22/1	0000 0000 1111 1s	14
11/2	0000 0000 0001 1010 s	17	23/1	0000 0000 1111 0s	14
12/1	0010 0010 s	9	24/1	0000 0000 1110 1s	14
12/2	0000 0000 0001 1001 s	17	25/1	0000 0000 1110 0s	14
13/1	0010 0000 s	9	26/1	0000 0000 1101 1s	14
13/2	0000 0000 0001 1000 s	17	27/1	0000 0000 0001 1111 s	17
14/1	0000 0011 10s	11	28/1	0000 0000 0001 1110 s	17
14/2	0000 0000 0001 0111 s	17	29/1	0000 0000 0001 1101 s	17
15/1	0000 0011 01s	11	30/1	0000 0000 0001 1100 s	17
15/2	0000 0000 0001 0110 s	17	31/1	0000 0000 0001 1011 s	17
16/1	0000 0010 00s	11	EOB	10	2
16/2	0000 0000 0001 0101 s	17	ESC	0000 01	6
17/1	0000 0001 1111 s	13			

Table 9.23: Variable-Length Run-Level Codes (Part 2).

- ◇ **Exercise 9.9:** compute the zigzag sequence and run-level codes for the AC coefficients of Figure 9.22b.
- ◇ **Exercise 9.10:** Given a block with 63 zero AC coefficients how is it coded?

A peculiar feature of Table 9.23 is that it lists two codes for run-level (0,1). Also, the first of those codes (labeled “first”) is 1s, which may conflict with the EOB code. The explanation is that the second of those codes (labeled “next”), 11s, is normally used, and this causes no conflict. The first code, 1s, is used only in nonintra coding, where an all-zero DCT coefficients block is coded in a special way.

The discussion so far has concentrated on encoding the quantized DCT coefficients for intra coding (*I* pictures). For nonintra coding (i.e., *P* and *B* pictures) the situation is different. The process of predicting a picture from another picture already decorrelates the samples, and the main advantage of the DCT in nonintra coding is quantization. Deep quantization of the DCT coefficients increases compression, and even a flat default quantization table (that does not take advantage of the properties of human vision) is effective in this case. Another feature of the DCT in nonintra coding is that the DC and AC coefficients are not substantially different, since they are the DCT transforms of differences. There is, therefore, no need to separate the encoding of the DC and AC coefficients.

The encoding process starts by looking for runs of macroblocks that are completely zero. Such runs are encoded by a macroblock address increment. If a macroblock is not all zeros, some of its six component blocks may still be completely zero. For such macroblocks the encoder prepares a *coded block pattern* (cbp). This is a 6-bit variable where each bit specifies whether one of the six blocks is completely zero or not. A zero

block is identified as such by the corresponding cbp bit. A nonzero block is encoded using the codes of Table 9.23. When such a nonzero block is encoded, the encoder knows that it cannot be all zeros. There must be at least one nonzero coefficient among the 64 quantized coefficients in the block. If the first nonzero coefficient has a run-level code of (0, 1), it is coded as “1s” and there is no conflict with the EOB code since the EOB code cannot be the first code in such a block. Any other nonzero coefficients with a run-level code of (0, 1) are encoded using the “next” code, which is “11s”.

9.6.2 MPEG-1 Video Syntax

Some of the many parameters used by MPEG to specify and control the compression of a video sequence are described in this section in detail. Readers who are interested only in the general description of MPEG may skip this section. The concepts of video sequence, picture, slice, macroblock, and block have already been discussed. Figure 9.24 shows the format of the compressed MPEG stream and how it is organized in six layers. Optional parts are enclosed in dashed boxes. Notice that only the video sequence of the compressed stream is shown; the system parts are omitted.

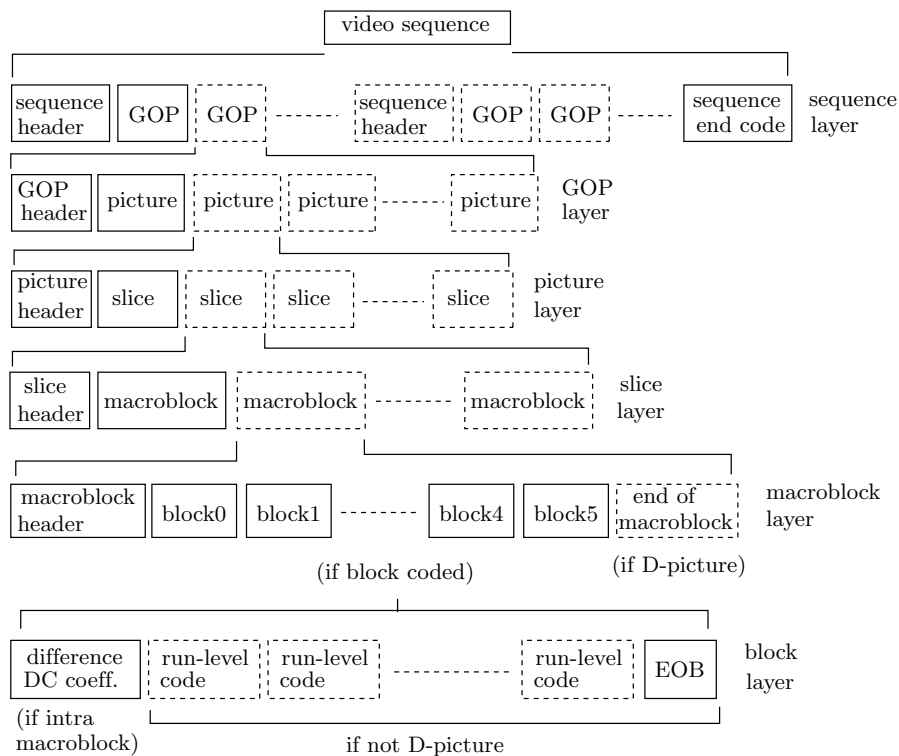


Figure 9.24: The Layers of a Video Stream.

The video sequence starts with a sequence header, followed by a group of pictures (GOP) and optionally by more GOPs. There may be other sequence headers followed

by more GOPs, and the sequence ends with a sequence-end-code. The extra sequence headers may be included to help in random access playback or video editing, but most of the parameters in the extra sequence headers must remain unchanged from the first header.

A group of pictures (GOP) starts with a GOP header, followed by one or more pictures. Each picture in a GOP starts with a picture header, followed by one or more slices. Each slice, in turn, consists of a slice header followed by one or more macroblocks of encoded, quantized DCT coefficients. A macroblock is a set of six 8×8 blocks, four blocks of luminance samples and two blocks of chrominance samples. Some blocks may be completely zero and may not be encoded. Each block is coded in intra or nonintra. An intra block starts with a difference between its DC coefficient and the previous DC coefficient (of the same type), followed by run-level codes for the nonzero AC coefficients and zero runs. The EOB code terminates the block. In a nonintra block, both DC and AC coefficients are run-level coded.

It should be mentioned that in addition to the *I*, *P*, and *B* picture types, there exists in MPEG a fourth type, a *D* picture (for *DC coded*). Such pictures contain only DC coefficient information; no run-level codes or EOB are included. However, *D* pictures are not allowed to be mixed with the other types of pictures, so they are rare and will not be discussed here.

The headers of a sequence, GOP, picture, and slice all start with a byte-aligned 32-bit start code. In addition to these video start codes there are other start codes for the system layer, user data, and error tagging. A start code starts with 23 zero bits, followed by a single bit of 1, followed by a unique byte. Table 9.25 lists all the video start codes. The “sequence.error” code is for cases where the encoder discovers unrecoverable errors in a video sequence and cannot encode it as a result. The run-level codes have variable lengths, so some 0 bits normally have to be appended to the video stream before a start code, to make sure the code starts on a byte boundary.

Start code	Hex	Binary
extension.start	000001B5	00000000 00000000 00000001 10110101
GOP.start	000001B8	00000000 00000000 00000001 10111000
picture.start	00000100	00000000 00000000 00000001 00000000
reserved	000001B0	00000000 00000000 00000001 10110000
reserved	000001B1	00000000 00000000 00000001 10110001
reserved	000001B6	00000000 00000000 00000001 10110110
sequence.end	000001B7	00000000 00000000 00000001 10110111
sequence.error	000001B4	00000000 00000000 00000001 10110100
sequence.header	000001B3	00000000 00000000 00000001 10110011
slice.start.1	00000101	00000000 00000000 00000001 00000001
...	...	...
slice.start.175	000001AF	00000000 00000000 00000001 10101111
user.data.start	000001B2	00000000 00000000 00000001 10110010

Table 9.25: MPEG Video Start Codes.

Code	height/width	Video source
0000	forbidden	
0001	1.0000	computers (VGA)
0010	0.6735	
0011	0.7031	16:9, 625 lines
0100	0.7615	
0101	0.8055	
0110	0.8437	16:9 525 lines
0111	0.8935	
1000	0.9157	CCIR Rec, 601, 625 lines
1001	0.9815	
1010	1.0255	
1011	1.0695	
1100	1.0950	CCIR Rec. 601, 525 lines
1101	1.1575	
1110	1.2015	
1111	reserved	

Table 9.26: MPEG Pel Aspect Ratio Codes.

Code	nominal picture rate	Typical applications
0000		forbidden
0001	23.976	Movies on NTSC broadcast monitors
0010	24	Movies, commercial clips, animation
0011	25	PAL, SECAM, generic 625/50Hz component video
0100	29.97	Broadcast rate NTSC
0101	30	NTSC profession studio, 525/60Hz component video
0110	50	noninterlaced PAL/SECAM/625 video
0111	59.94	Noninterlaced broadcast NTSC
1000	60	Noninterlaced studio 525 NTSC rate
1001		
...		
1111	reserved	

Table 9.27: MPEG Picture Rates and Typical Applications.

$\text{horizontal_size} \leq 768$ pels.
 $\text{vertical_size} \leq 576$ lines.
 number of macroblocks ≤ 396 .
 $(\text{number of macroblocks}) \times \text{picture_rate} \leq 396 \times 25$.
 $\text{picture_rate} \leq 30$ pictures per second.
 $\text{vbm_buffer_size} \leq 160$.
 $\text{bit_rate} \leq 4640$.
 $\text{forward_f_code} \leq 4$.
 $\text{backward_f_code} \leq 4$.

Table 9.28: Constrained Parameters Bounds.

Video Sequence Layer: This layer starts with start code 000001B3, followed by nine fixed-length data elements. The parameters `horizontal_size` and `vertical_size` are 12-bit parameters that define the width and height of the picture. Neither is allowed to be zero, and `vertical_size` must be even. Parameter `pel_aspect_ratio` is a 4-bit parameter that specifies the aspect ratio of a pel. Its 16 values are listed in Table 9.26. Parameter `picture_rate` is a 4-bit parameter that specifies one of 16 picture refresh rates. Its eight nonreserved values are listed in Table 9.27. The 18-bit `bit_rate` data element specifies the compressed data rate to the decoder (in units of 400 bits/s). This parameter has to be positive and is related to the true bit rate R by

$$\text{bit_rate} = \lceil R/400 \rceil.$$

Next comes a `marker_bit`. This single bit of 1 prevents the accidental generation of a start code in cases where some bits get corrupted. Marker bits are common in MPEG. The 10-bit `vbv_buffer_size` follows the marker bit. It specifies to the decoder the lower bound for the size of the compressed data buffer. The buffer size, in bits, is given by

$$B = 8 \times 2048 \times \text{vbv_buffer_size},$$

and is a multiple of 2K bytes. The `constrained_parameter_flag` is a 1-bit parameter that is normally 0. When set to 1, it signifies that some of the other parameters have the values listed in Table 9.28.

The last two data elements are 1-bit each and control the loading of the intra and nonintra quantization tables. When `load_intra_quantizer_matrix` is set to 1, it means that it is followed by the 64 8-bit QCs of the `intra_quantizer_matrix`. Similarly, `load_non_intra_quantizer_matrix` signals whether the quantization table `non_intra_quantizer_matrix` follows it or whether the default should be used.

GOP Layer: This layer starts with nine mandatory elements, optionally followed by extensions and user data, and by the (compressed) pictures themselves.

The 32-bit group start code 000001B8 is followed by the 25-bit `time_code`, which consists of the following six data elements: `drop_frame_flag` (one bit) is zero unless the picture rate is 29.97 Hz; `time_code_hours` (five bits, in the range [0,23]), data elements `time_code_minutes` (six bits, in the range [0,59]), and `time_code_seconds` (six bits, in the same range) indicate the hours, minutes, and seconds in the time interval from the start of the sequence to the display of the first picture in the GOP. The 6-bit `time_code_pictures` parameter indicates the number of pictures in a second. There is a `marker_bit` between `time_code_minutes` and `time_code_seconds`.

Following the `time_code` there are two 1-bit parameters. The flag `closed_gop` is set if the GOP is closed (i.e., its pictures can be decoded without reference to pictures from outside the group). The `broken_link` flag is set to 1 if editing has disrupted the original sequence of groups of pictures.

Picture Layer: Parameters in this layer specify the type of the picture (I , P , B , or D) and the motion vectors for the picture. The layer starts with the 32-bit `picture_start_code`, whose hexadecimal value is 00000100. It is followed by a 10-bit `temporal_reference` parameter, which is the picture number (modulo 1024) in the sequence. The next parameter is the 3-bit `picture_coding_type` (Table 9.29), and this

is followed by the 16-bit `vbv_delay` that tells the decoder how many bits must be in the compressed data buffer before the picture can be decoded. This parameter helps prevent buffer overflow and underflow.

Code	picture type
000	forbidden
001	<i>I</i>
010	<i>P</i>
011	<i>B</i>
100	<i>D</i>
101	reserved
...	...
111	reserved

Table 9.29: Picture Type Codes.

If the picture type is *P* or *B*, then this is followed by the forward motion vectors scale information, a 3-bit parameter called `forward_f_code` (see Table 9.34). For *B* pictures, there follows the backward motion vectors scale information, a 3-bit parameter called `backward_f_code`.

Slice Layer: There can be many slices in a picture, so the start code of a slice ends with a value in the range [1, 175]. This value defines the macroblock row where the slice starts (a picture can therefore have up to 175 rows of macroblocks). The horizontal position where the slice starts in that macroblock row is determined by other parameters.

The `quantizer_scale` (five bits) initializes the quantizer scale factor, discussed earlier in connection with the rounding of the quantized DCT coefficients. The `extra_bit_slice` flag following it is always 0 (the value 1 is reserved for future ISO standards). Following this, the encoded macroblocks are written.

Macroblock Layer: This layer identifies the position of the macroblock relative to the position of the current macroblock. It codes the motion vectors for the macroblock, and identifies the zero and nonzero blocks in the macroblock.

Each macroblock has an address, or index, in the picture. Index values start at 0 in the upper-left corner of the picture and continue in raster order. When the encoder starts encoding a new picture, it sets the macroblock address to -1. The `macroblock_address_increment` parameter contains the amount needed to increment the macroblock address in order to reach the macroblock being coded. This parameter is normally 1. If `macroblock_address_increment` is greater than 33, it is encoded as a sequence of `macroblock_escape` codes, each incrementing the macroblock address by 33, followed by the appropriate code from Table 9.30.

The `macroblock_type` is a variable-size parameter, between 1 and 6 bits long, whose values are listed in Table 9.31. Each value of this variable-size code is decoded into five bits that become the values of the following five flags:

1. `macroblock_quant`. If set, a new 5-bit quantization scale is sent.
2. `macroblock_motion_forward`. If set, a forward motion vector is sent.
3. `macroblock_motion_backward`. If set, a backward motion vector is sent.

Increment value	macroblock address increment	Increment value	macroblock address increment
1	1	19	0000 00
2	011	20	0000 0100 11
3	010	21	0000 0100 10
4	0011	22	0000 0100 011
5	0010	23	0000 0100 010
6	0001 1	24	0000 0100 001
7	0001 0	25	0000 0100 000
8	0000 111	26	0000 0011 111
9	0000 011	27	0000 0011 110
10	0000 1011	28	0000 0011 101
11	0000 1010	29	0000 0011 100
12	0000 1001	30	0000 0011 011
13	0000 1000	31	0000 0011 010
14	0000 0111	32	0000 0011 001
15	0000 0110	33	0000 0011 000
16	0000 0101 11	stuffing	0000 0001 111
17	0000 0101 10	escape	0000 0001 000
18	0000 0101 01		

Table 9.30: Codes for Macroblock Address Increment.

	Code	quant	forward	backward	pattern	intra
I	1	0	0	0	0	1
	01	1	0	0	0	1
	001	0	1	0	0	0
	01	0	0	0	1	0
P	00001	1	0	0	1	0
	1	0	1	0	1	0
	00010	1	1	0	1	0
	00011	0	0	0	0	1
	000001	1	0	0	0	1
	0010	0	1	0	0	0
	010	0	0	1	0	0
B	10	0	1	1	0	0
	0011	0	1	0	1	0
	000011	1	1	0	1	0
	011	0	0	1	1	0
	000010	1	0	1	1	0
	11	0	1	1	1	0
	00010	1	1	1	1	0
	00011	0	0	0	0	1
	000001	1	0	0	0	1
	1	0	0	0	0	1
	0	0	0	0	0	1
D	1	0	0	0	0	1

Table 9.31: Variable Length Codes for Macroblock Types.

cbp dec.	cbp binary	block # 012345	cbp code	cbp dec.	cbp binary	block # 012345	cbp code
0	000000		forbidden	32	100000	c.....	1010
1	000001	c	01011	33	100001	c....c	0010100
2	000010	c.	01001	34	100010	c...c.	0010000
3	000011	cc	001101	35	100011	c...cc	00011100
4	000100	...c..	1101	36	100100	c..c..	001110
5	000101	...c.c	0010111	37	100101	c..c.c	00001110
6	000110	...cc.	0010011	38	100110	c..cc.	00001100
7	000111	...ccc	00011111	39	100111	c..ccc	00000010
8	001000	.c....	1100	40	101000	c.c...	10000
9	001001	.c...c	0010110	41	101001	c.c...c	00011000
10	001010	.c.c..	0010010	42	101010	c.c.c.	00010100
11	001011	.c.c.c	00011110	43	101011	c.c.c.c	00010000
12	001100	.cc...	10011	44	101100	c.cc..	01110
13	001101	.cc.c.	00011011	45	101101	c.cc.c	00001010
14	001110	.ccc.	00010111	46	101110	c.ccc.	00000110
15	001111	.cccc	00010011	47	101111	c.cccc	000000110
16	010000	c.....	1011	48	110000	cc....	10010
17	010001	c...c	0010101	49	110001	cc...c	00011010
18	010010	c..c.	0010001	50	110010	cc...c	00010110
19	010011	c..cc	00011101	51	110011	cc...cc	00010010
20	010100	c.c..	10001	52	110100	cc.c..	01101
21	010101	c.c.c	00011001	53	110101	cc.c.c	00001001
22	010110	c.c.c.	00010101	54	110110	cc.cc.	00000101
23	010111	c.c.ccc	00010001	55	110111	cc.ccc	000000101
24	011000	.cc...	001111	56	111000	ccc...	01100
25	011001	.cc...c	00001111	57	111001	ccc...c	00001000
26	011010	.cc.c.	00001101	58	111010	ccc.c.	00000100
27	011011	.cc.c.c	000000011	59	111011	ccc.cc	000000100
28	011100	.ccc..	01111	60	111100	cccc..	111
29	011101	.ccc.c	00001011	61	111101	cccc.c	01010
30	011110	.cccc.	00000111	62	111110	ccccc.	01000
31	011111	.cccc	000000111	63	111111	cccccc	001100

Table 9.32: Codes for Macroblock Address Increment.

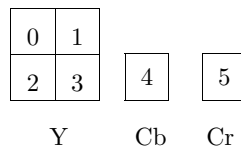


Figure 9.33: Indexes of Six Blocks in a Macroblock.

4. **macroblock_pattern**. If set, the **coded_block_pattern** code (variable length) listed in Table 9.32 follows to code the six **pattern_code** bits of variable **cbp** discussed earlier (blocks labeled “.” in the table are skipped, and blocks labeled **c** are coded). These six bits identify the six blocks of the macroblock as completely zero or not. The correspondence between the six bits and blocks is shown in Figure 9.33, where block 0 corresponds to the most-significant bit of the **pattern_code**.

5. **macroblock_intra**. If set, the six blocks of this macroblock are coded as intra.

These five flags determine the rest of the processing steps for the macroblock.

Once the **pattern_code** bits are known, blocks that correspond to 1 bits are encoded.

Block Layer: This layer is the lowest in the video sequence. It contains the encoded 8×8 blocks of quantized DCT coefficients. The coding depends on whether the block contains luminance or chrominance samples and on whether the macroblock is intra or nonintra. In nonintra coding, blocks that are completely zero are skipped; they don't have to be encoded.

The **macroblock_intra** flag gets its value from **macroblock_type**. If it is set, the DC coefficient of the block is coded separately from the AC coefficients.

9.6.3 Motion Compensation

An important element of MPEG is *motion compensation*, which is used in inter coding only. In this mode, the pels of the current picture are predicted by those of a previous reference picture (and, possibly, by those of a future reference picture). Pels are subtracted, and the differences (which should be small numbers) are DCT transformed, quantized, and encoded. The differences between the current picture and the reference picture are normally caused by motion (either camera motion or scene motion), so best prediction is obtained by matching a region in the current picture with a different region in the reference picture. MPEG does not require the use of any particular matching algorithm, and any implementation can use its own method for matching macroblocks (see Section 9.5 for examples of matching algorithms). The discussion here concentrates on the operations of the decoder.

Differences between consecutive pictures may also be caused by random noise in the video camera, or by variations of illumination, which may change brightness in a nonuniform way. In such cases, motion compensation is not used, and each region ends up being matched with the same spatial region in the reference picture.

If the difference between consecutive pictures is caused by camera motion, one motion vector is enough for the entire picture. Normally, however, there is also scene motion and movement of shadows, so a number of motion vectors are needed, to describe the motion of different regions in the picture. The size of those regions is critical. A large number of small regions improves prediction accuracy, whereas the opposite situation simplifies the algorithms used to find matching regions and also leads to fewer motion vectors and sometimes to better compression. Since a macroblock is such an important unit in MPEG, it was also selected as the elementary region for motion compensation.

Another important consideration is the precision of the motion vectors. A motion vector such as (15, -4) for a macroblock *M* typically means that *M* has been moved from the reference picture to the current picture by displacing it 15 pels to the right and four pels up (a positive vertical displacement is down). The components of the

vector are in units of pels. They may, however, be in units of half a pel, or even smaller. In MPEG-1, the precision of motion vectors may be either full-pel or half-pel, and the encoder signals this decision to the decoder by a parameter in the picture header (this parameter may be different from picture to picture).

It often happens that large areas of a picture move at identical or at similar speeds, and this implies that the motion vectors of adjacent macroblocks are correlated. This is the reason why the MPEG encoder encodes a motion vector by subtracting it from the motion vector of the preceding macroblock and encoding the difference.

A *P* picture uses an earlier *I* picture or *P* picture as a reference picture. We say that *P* pictures use forward motion-compensated prediction. When a motion vector *MD* for a macroblock is determined (*MD* stands for *motion displacement*, since the vector consists of two components, the horizontal and the vertical displacements), MPEG denotes the motion vector of the preceding macroblock in the slice by *PMD* and computes the difference $dMD = MD - PMD$. *PMD* is reset to zero at the start of a slice, after a macroblock is intra coded, when the macroblock is skipped, and when parameter `block_motion_forward` is zero.

The 1-bit parameter `full_pel_forward_vector` in the picture header defines the precision of the motion vectors (1=full-pel, 0=half-pel). The 3-bit parameter `forward_f_code` defines the range.

A *B* picture may use forward or backward motion compensation. If both are used, then two motion vectors are calculated by the encoder for the current macroblock, and each vector is encoded by first computing a difference, as in *P* pictures. Also, if both motion vectors are used, then the prediction is the average of both. Here is how the prediction is done in the case of both forward and backward prediction. Suppose that the encoder has determined that macroblock *M* in the current picture matches macroblock *MB* in the following picture and macroblock *MF* in the preceding picture. Each pel $M[i, j]$ in macroblock *M* in the current picture is predicted by computing the difference

$$M[i, j] - \frac{MF[i, j] + MB[i, j]}{2},$$

where the quotient of the division by 2 is rounded to the nearest integer.

Parameters `full_pel_forward_vector` and `forward_f_code` have the same meaning as for a *P* picture. In addition, there are the two corresponding parameters (for backward) `full_pel_backward_vector` and `backward_f_code`.

The following rules apply when coding motion vectors in *B* pictures:

1. The quantity *PMD* is reset to zero at the start of a slice and after a macroblock is skipped.
2. In a skipped macroblock, *MD* is predicted from the preceding macroblock in the slice.
3. When `motion_vector_forward` is zero, the forward *MDs* are predicted from the preceding macroblock in the slice.
4. When `motion_vector_backward` is zero, the backward *MDs* are predicted from the preceding macroblock in the slice.
5. A *B* picture is predicted by means of *I* and *P* pictures but not by another *B* picture.
6. The two components of a motion vector (the motion displacements) are computed to a precision of either full-pel or half-pel, as specified in the picture header.

Displacement Principal and Residual: The two components of a motion vector are motion displacements. Each motion displacement has two parts, a principal and a residual. The principal part is denoted by dMD_p . It is a signed integer that is given by the product

$$dMD_p = \text{motion_code} \times f,$$

where (1) **motion_code** is an integer parameter in the range $[-16, +16]$, included in the compressed stream by means of a variable-length code, and (2) f , the scaling factor, is a power of 2 given by

$$f = 2^{rsize}.$$

The integer $rsize$ is simply **f_code** $- 1$, where **f_code** is a 3-bit parameter with values in the range $[1, 7]$. This implies that $rsize$ has values $[0, 6]$ and f is a power of 2 in the range $[1, 2, 4, 8, 16, 32, 64]$.

The residual part is denoted by **r** and is a positive integer, defined by

$$\mathbf{r} = |dMD_p| - |dMD|.$$

After computing **r**, the encoder encodes it by concatenating the ones-complement of **r** and **motion_r** to the variable-length code for parameter **motion_code**. The parameter **motion_r** is related to **r** by

$$\text{motion_r} = (f - 1) - \mathbf{r},$$

and f should be chosen as the smallest value that satisfies the following inequalities for the largest (positive or negative) differential displacement in the entire picture

$$-(16 \times f) \leq \text{dMD} < (16 \times f).$$

Once f has been selected, the value of parameter **motion_code** for the differential displacement of a macroblock is given by

$$\text{motion_code} = \frac{\text{dMD} + \text{Sign}(\text{dMD}) \times (f - 1)}{f},$$

where the quotient is rounded such that $\text{motion_code} \times f \geq \text{dMD}$.

Table 9.34 lists the header parameters for motion vector computation (“p” stands for picture header and “mb” stands for macroblock header). Table 9.35 lists the generic and full names of the parameters mentioned here, and Table 9.36 lists the range of values of **motion_r** as a function of **f_code**.

9.6.4 Pel Reconstruction

The main task of the MPEG decoder is to reconstruct the pels of the entire video sequence. This is done by reading the codes of a block from the compressed stream, decoding them, dequantizing them, and calculating the IDCT. For nonintra blocks in P and B pictures, the decoder has to add the motion-compensated prediction to the results of the IDCT. This is repeated six times (or fewer, if some blocks are completely zero)

Header parameter	set in header	number of bits	displacement parameter
full_pel_forward_vector	p	1	precision
forward_f_code	p	3	range
full_pel_backward_vector	p	1	precision
backward_f_code	p	3	range
motion_horizontal_forward_code	mb	vlc	principal
motion_horizontal_forward_r	mb	forward_r_size	residual
motion_vertical_forward_code	mb	vlc	principal
motion_vertical_forward_r	mb	forward_r_size	residual

Table 9.34: Header Parameters for Motion Vector Computation.

Generic name	Full name	Range
full_pel_vector	full_pel_forward_vector full_pel_backward_vector	0,1
f_code	forward_f_code backward_f_code	1–7
r_size	forward_r_size backward_r_size	0–6
f	forward_f backward_f	1,2,4,8,16,32,64
motion_code	motion_horizontal_forward_code motion_vertical_forward_code motion_horizontal_backward_code motion_vertical_backward_code	$-16 \rightarrow +16$
motion_r	motion_horizontal_forward_r motion_vertical_forward_r motion_horizontal_backward_r motion_vertical_backward_r	$0 \rightarrow (f - 1)$
r	compliment_horizontal_forward_r compliment_vertical_forward_r compliment_horizontal_backward_r compliment_vertical_backward_r	$0 \rightarrow (f - 1)$

Table 9.35: Generic Names for Motion Displacement Parameters.

f_code	r_size	f	f - 1	r	motion_r
1	0	1	0		0
2	1	2	1	0,1	1,0
3	2	4	3	0–3	3,...,0
4	3	8	7	0–7	7,...,0
5	4	16	15	0–15	15,...,0
6	5	32	31	0–31	31,...,0
7	6	64	63	0–63	63,...,0

Table 9.36: Range of Values of motion_r as a Function of f_code.

for the six blocks of a macroblock. The entire sequence is decoded picture by picture, and within each picture, macroblock by macroblock.

It has already been mentioned that the IDCT is not rigidly defined in MPEG, which may lead to accumulation of errors, called *IDCT mismatch*, during decoding.

For intra-coded blocks, the decoder reads the differential code of the DC coefficient and uses the decoded value of the previous DC coefficient (of the same type) to decode the DC coefficient of the current block. It then reads the run-level codes until an EOB code is encountered, and decodes them, generating a sequence of 63 AC coefficients, normally with few nonzero coefficients and runs of zeros between them. The DC and 63 AC coefficients are then collected in zigzag order to create an 8×8 block. After dequantization and inverse DCT calculation, the resulting block becomes one of the six blocks that make up a macroblock (in intra coding all six blocks are always coded, even those that are completely zero).

For nonintra blocks, there is no distinction between DC and AC coefficients and between luminance and chrominance blocks. They are all decoded in the same way.

9.7 MPEG-4

MPEG-4 is a new standard for audiovisual data. Although video and audio compression is still a central feature of MPEG-4, this standard includes much more than just compression of the data. As a result, MPEG-4 is huge and this section can only describe its main features. No details are provided. We start with a bit of history (see also Section 9.9).

The MPEG-4 project started in May 1991 and initially aimed at finding ways to compress multimedia data to very low bitrates with minimal distortions. In July 1994, this goal was significantly altered in response to developments in audiovisual technologies. The MPEG-4 committee started thinking of future developments and tried to guess what features should be included in MPEG-4 to meet them. A call for proposals was issued in July 1995 and responses were received by October of that year. (The proposals were supposed to address the eight major functionalities of MPEG-4, listed below.) Tests of the proposals were conducted starting in late 1995. In January 1996, the first verification model was defined, and the cycle of calls for proposals—proposal implementation and verification was repeated several times in 1997 and 1998. Many proposals were accepted for the many facets of MPEG-4, and the first version of MPEG-4 was accepted and approved in late 1998. The formal description was published in 1999 with many amendments that keep coming out.

At present (mid-2006), the MPEG-4 standard is designated the ISO/IEC 14496 standard, and its formal description, which is available from [ISO 03], consists of 17 parts plus new amendments. More readable descriptions can be found in [Pereira and Ebrahimi 02] and [Symes 03].

MPEG-1 was originally developed as a compression standard for interactive video on CDs and for digital audio broadcasting. It turned out to be a technological triumph but a visionary failure. On the one hand, not a single design mistake was found during the implementation of this complex algorithm and it worked as expected. On the other hand, interactive CDs and digital audio broadcasting have had little commercial success,

so MPEG-1 is used today for general video compression. One aspect of MPEG-1 that was supposed to be minor, namely MP3, has grown out of proportion and is commonly used today for audio (Section 10.14). MPEG-2, on the other hand, was specifically designed for digital television and this standard has had tremendous commercial success.

The lessons learned from MPEG-1 and MPEG-2 were not lost on the MPEG committee members and helped shape their thinking for MPEG-4. The MPEG-4 project started as a standard for video compression at very low bitrates. It was supposed to deliver reasonable video data in only a few thousand bits per second. Such compression is important for video telephones, video conferences or for receiving video in a small, hand-held device, especially in a mobile environment, such as a moving car. After working on this project for two years, the committee members, realizing that the rapid development of multimedia applications and services will require more and more compression standards, have revised their approach. Instead of a compression standard, they decided to develop a set of tools (a toolbox) to deal with audiovisual products in general, at present and in the future. They hoped that such a set will encourage industry to invest in new ideas, technologies, and products in confidence, while making it possible for consumers to generate, distribute, and receive different types of multimedia data with ease and at a reasonable cost.

Traditionally, methods for compressing video have been based on pixels. Each video frame is a rectangular set of pixels, and the algorithm looks for correlations between pixels in a frame and between frames. The compression paradigm adopted for MPEG-4, on the other hand, is based on objects. (The name of the MPEG-4 project was also changed at this point to “coding of audiovisual objects.”) In addition to producing a movie in the traditional way with a camera or with the help of computer animation, an individual generating a piece of audiovisual data may start by defining objects, such as a flower, a face, or a vehicle, and then describing how each object should be moved and manipulated in successive frames. A flower may open slowly, a face may turn, smile, and fade, a vehicle may move toward the viewer and appear bigger. MPEG-4 includes an object description language that provides for a compact description of both objects and their movements and interactions.

Another important feature of MPEG-4 is *interoperability*. This term refers to the ability to exchange any type of data, be it text, graphics, video, or audio. Obviously, interoperability is possible only in the presence of standards. All devices that produce data, deliver it, and consume (play, display, or print) it must obey the same rules and read and write the same file structures.

During its important July 1994 meeting, the MPEG-4 committee decided to revise its original goal and also started thinking of future developments in the audiovisual field and of features that should be included in MPEG-4 to meet them. They came up with eight points that they considered important functionalities for MPEG-4.

1. **Content-based multimedia access tools.** The MPEG-4 standard should provide tools for accessing and organizing audiovisual data. Such tools may include indexing, linking, querying, browsing, delivering files, and deleting them. The main tools currently in existence are listed later in this section.

2. **Content-based manipulation and bitstream editing.** A syntax and a coding scheme should be part of MPEG-4. The idea is to enable users to manipulate and edit compressed files (bitstreams) without fully decompressing them. A user should

be able to select an object and modify it in the compressed file without decompressing the entire file.

3. **Hybrid natural and synthetic data coding.** A natural scene is normally produced by a video camera. A synthetic scene consists of text and graphics. MPEG-4 recognizes the need for tools to compress natural and synthetic scenes and mix them interactively.

4. **Improved temporal random access.** Users may often want to access part of the compressed file, so the MPEG-4 standard should include tags to make it easy to quickly reach any point in the file. This may be important when the file is stored in a central location and the user is trying to manipulate it remotely, over a slow communications channel.

5. **Improved coding efficiency.** This feature simply means improved compression. Imagine a case where audiovisual data has to be transmitted over a low-bandwidth channel (such as a telephone line) and stored in a low-capacity device such as a smart-card. This is possible only if the data is well compressed, and high compression rates (or equivalently, low bitrates) normally involve a trade-off in the form of smaller image size, reduced resolution (pixels per inch), and lower quality.

6. **Coding of multiple concurrent data streams.** It seems that future audiovisual applications will allow the user not just to watch and listen but also to interact with the image. As a result, the MPEG-4 compressed stream can include several views of the same scene, enabling the user to select any of them to watch and to change views at will. The point is that the different views may be similar, so any redundancy should be eliminated by means of efficient compression that takes into account identical patterns in the various views. The same is true for the audio part (the soundtracks).

7. **Robustness in error-prone environments.** MPEG-4 must provide error-correcting codes for cases where audiovisual data is transmitted through a noisy channel. This is especially important for low-bitrate streams, where even the smallest error may be noticeable and may propagate and affect large parts of the audiovisual presentation.

8. **Content-based scalability.** The compressed stream may include audiovisual data in fine resolution and high quality, but any MPEG-4 decoder should be able to decode it at low resolution and low quality. This feature is useful in cases where the data is decoded and displayed on a small, low-resolution screen, or in cases where the user is in a hurry and prefers to see a rough image rather than wait for a full decoding.

Once the eight fundamental functionalities above have been identified and listed, the MPEG-4 committee started the process of developing separate tools to satisfy these functionalities. This is an ongoing process that continues to this day and will continue in the future. An MPEG-4 author faced with an application has to identify the requirements of the application and select the right tools. It is now clear that compression is a central requirement in MPEG-4, but not the only requirement, as it was for MPEG-1 and MPEG-2.

An example may serve to illustrate the concept of natural and synthetic objects. In a news session on television, a few seconds may be devoted to the weather. The viewers see a weather map of their local geographic region (a computer-generated image) that may zoom in and out and pan. Graphic images of sun, clouds, rain drops, or a rainbow (synthetic scenes) appear, move, and disappear. A person is moving, pointing, and talking (a natural scene), and text (another synthetic scene) may also appear from time

to time. All those scenes are mixed by the producers into one audiovisual presentation that's compressed, transmitted (on television cable, on the air, or into the Internet), received by computers or television sets, decompressed, and displayed (consumed).

In general, audiovisual content goes through three stages: production, delivery, and consumption. Each of these stages is summarized below for the traditional approach and for the MPEG-4 approach.

Production. Traditionally, audiovisual data consists of two-dimensional scenes; it is produced with a camera and microphones and consists of natural objects. All the mixing of objects (composition of the image) is done during production. The MPEG-4 approach is to allow for both two-dimensional and three-dimensional objects and for natural and synthetic scenes. The composition of objects is explicitly specified by the producers during production by means of a special language. This allows later editing.

Delivery. The traditional approach is to transmit audiovisual data on a few networks, such as local-area networks and satellite transmissions. The MPEG-4 approach is to let practically any data network carry audiovisual data. Protocols exist to transmit audiovisual data over any type of network.

Consumption. Traditionally, a viewer can only watch video and listen to the accompanying audio. Everything is precomposed. The MPEG-4 approach is to allow the user as much freedom of composition as possible. The user should be able to interact with the audiovisual data, watch only parts of it, interactively modify the size, quality, and resolution of the parts being watched, and be as active in the consumption stage as possible.

Because of the wide goals and rich variety of tools available as part of MPEG-4, this standard is expected to have many applications. The ones listed here are just a few important examples.

1. Streaming multimedia data over the Internet or over local-area networks. This is important for entertainment and education.
2. Communications, both visual and audio, between vehicles and/or individuals. This has military and law enforcement applications.
3. Broadcasting digital multimedia. This, again, has many entertainment and educational applications.
4. Context-based storage and retrieval. Audiovisual data can be stored in compressed form and retrieved for delivery or consumption.
5. Studio and television postproduction. A movie originally produced in English may be translated to another language by dubbing or subtitling.
6. Surveillance. Low-quality video and audio data can be compressed and transmitted from a surveillance camera to a central monitoring location over an inexpensive, slow communications channel. Control signals may be sent back to the camera through the same channel to rotate or zoom it in order to follow the movements of a suspect.
7. Virtual conferencing. This time-saving application is the favorite of busy executives.

Our short description of MPEG-4 concludes with a list of the main tools specified by the MPEG-4 standard.

Object descriptor framework. Imagine an individual participating in a video conference. There is an MPEG-4 object representing this individual and there are video

and audio streams associated with this object. The object descriptor (OD) provides information on elementary streams available to represent a given MPEG-4 object. The OD also has information on the source location of the streams (perhaps a URL) and on various MPEG-4 decoders available to consume (i.e., display and play sound) the streams. Certain objects place restrictions on their consumption, and these are also included in the OD of the object. A common example of a restriction is the need to pay before an object can be consumed. A video, for example, may be watched only if it has been paid for, and the consumption may be restricted to streaming only, so that the consumer cannot copy the original movie.

Systems decoder model. All the basic synchronization and streaming features of the MPEG-4 standard are included in this tool. It specifies how the buffers of the receiver should be initialized and managed during transmission and consumption. It also includes specifications for timing identification and mechanisms for recovery from errors.

Binary format for scenes. An MPEG-4 scene consists of objects, but for the scene to make sense, the objects must be placed at the right locations and moved and manipulated at the right times. This important tool (BIFS for short) is responsible for describing a scene, both spatially and temporally. It contains functions that are used to describe two-dimensional and three-dimensional objects and their movements. It also provides ways to describe and manipulate synthetic scenes, such as text and graphics.

MPEG-J. A user may want to use the Java programming language to implement certain parts of an MPEG-4 content. MPEG-J allows the user to write such MPEGlets and it also includes useful Java APIs that help the user interface with the output device and with the networks used to deliver the content. In addition, MPEG-J also defines a delivery mechanism that allows MPEGlets and other Java classes to be streamed to the output separately.

Extensible MPEG-4 textual format. This tool is a format, abbreviated XMT, that allows authors to exchange MPEG-4 content with other authors. XMT can be described as a framework that uses a textual syntax to represent MPEG-4 scene descriptions.

Transport tools. Two such tools, MP4 and FlexMux, are defined to help users transport multimedia content. The former writes MPEG-4 content on a file, whereas the latter is used to interleave multiple streams into a single stream, including timing information.

Video compression. It has already been mentioned that compression is only one of the many goals of MPEG-4. The video compression tools consist of various algorithms that can compress video data to bitrates between 5 kbits/sec (very low bitrate, implying low-resolution and low-quality video) and 1 Gbit/sec. Compression methods vary from very lossy to nearly lossless, and some also support progressive and interlaced video. Many MPEG-4 objects consist of polygon meshes, so most of the video compression tools are designed to compress such meshes. Section 11.11 is an example of such a method.

Robustness tools. Data compression is based on removing redundancies from the original data, but this also renders the data more vulnerable to errors. All methods for error detection and correction are based on increasing the redundancy of the data. MPEG-4 includes tools to add robustness, in the form of error-correcting codes, to

the compressed content. Such tools are important in applications where data has to be transmitted through unreliable lines. Robustness also has to be added to very low bitrate MPEG-4 streams because these suffer most from errors.

Fine-grain scalability. When MPEG-4 content is streamed, it is sometimes desirable to first send a rough image, and then improve its visual quality by adding layers of extra information. This is the function of the fine-grain scalability (FGS) tools.

Face and body animation. Often, an MPEG-4 file contains human faces and bodies, and they have to be animated. The MPEG-4 standard therefore provides tools for constructing and animating such surfaces.

Speech coding. Speech may often be part of MPEG-4 content, and special tools are provided to compress it efficiently at bitrates from 2 Kbit/sec up to 24 Kbit/sec. The main algorithm for speech compression is CELP, but there is also a parametric coder.

Audio coding. Several algorithms are available as MPEG-4 tools for audio compression. Examples are (1) advanced audio coding (AAC, Section 10.15), (2) transform-domain weighted interleaved vector quantization (Twin VQ, can produce low bitrates such as 6 kbit/sec/channel), and (3) harmonic and individual lines plus noise (HILN, a parametric audio coder).

Synthetic audio coding. Algorithms are provided to generate the sound of familiar musical instruments. They can be used to generate synthetic music in compressed format. The MIDI format, popular with computer music users, is also included among these tools. Text-to-speech tools allow authors to write text that will be pronounced when the MPEG-4 content is consumed. This text may include parameters such as pitch contour and phoneme duration that improve the speech quality.

9.8 H.261

In late 1984, the CCITT (currently the ITU-T) organized an expert group to develop a standard for visual telephony for ISDN services. The idea was to send images and sound between special terminals, so that users could talk and see each other. This type of application requires sending large amounts of data, so compression became an important consideration. The group eventually came up with a number of standards, known as the H series (for video) and the G series (for audio) recommendations, all operating at speeds of $p \times 64$ Kbit/sec for $1 \leq p \leq 30$. These standards are known today under the umbrella name of $p \times 64$ and are summarized in Table 9.37. This section provides a short summary of the H.261 standard [Liou 91].

Members of the $p \times 64$ also participated in the development of MPEG, so the two methods have many common elements. There is, however, an important difference between them. In MPEG, the decoder must be fast, since it may have to operate in real time, but the encoder can be slow. This leads to very asymmetric compression, and the encoder can be hundreds of times more complex than the decoder. In H.261, both encoder and decoder operate in real time, so both have to be fast. Still, the H.261 standard defines only the data stream and the decoder. The encoder can use any method as long as it creates a valid compressed stream. The compressed stream is organized in layers, and macroblocks are used as in MPEG. Also, the same 8×8 DCT and the same zigzag order as in MPEG are used. The intra DC coefficient is quantized by always

Standard	Purpose
H.261	Video
H.221	Communications
H.230	Initial handshake
H.320	Terminal systems
H.242	Control protocol
G.711	Companded audio (64 Kbits/s)
G.722	High quality audio (64 Kbits/s)
G.728	Speech (LD-CELP @16kbits/s)

Table 9.37: The $p \times 64$ Standards.

dividing it by 8, and it has no dead zone. The inter DC and all AC coefficients are quantized with a dead zone.

Motion compensation is used when pictures are predicted from other pictures, and motion vectors are coded as differences. Blocks that are completely zero can be skipped within a macroblock, and variable-length codes that are very similar to those of MPEG (such as run-level codes), or are even identical (such as motion vector codes) are used. In all these aspects, H.261 and MPEG are very similar.

There are, however, important differences between them. H.261 uses a single quantization coefficient instead of an 8×8 table of QCs, and this coefficient can be changed only after 11 macroblocks. AC coefficients that are intra coded have a dead zone. The compressed stream has just four layers, instead of MPEG's six. The motion vectors are always full-pel and are limited to a range of just ± 15 pels. There are no B pictures, and only the immediately preceding picture can be used to predict a P picture.

9.8.1 H.261 Compressed Stream

H.261 limits the image to just two sizes, the common intermediate format (CIF), which is optional, and the quarter CIF (QCIF). These are shown in Figure 9.38a,b. The CIF format has dimensions of 288×360 for luminance, but only 352 of the 360 columns of pels are actually coded, creating an active area of 288×352 pels. For chrominance, the dimensions are 144×180 , but only 176 columns are coded, for an active area of 144×176 pels. The QCIF format is one-fourth of CIF, so the luminance component is 144×180 with an active area of 144×176 , and the chrominance components are 72×90 , with an active area of 72×88 pels.

The macroblocks are organized in groups (known as groups of blocks or GOB) of 33 macroblocks each. Each GOB is a 3×11 array (48×176 pels), as shown in Figure 9.38c. A CIF picture consists of 12 GPBs, and a QCIF picture is three GOBs, numbered as in Figure 9.38d.

Figure 9.39 shows the four H.261 video sequence layers. The main layer is the picture layer. It start with a picture header, followed by the GOBs (12 or 3, depending on the image size). The compressed stream may contain as many pictures as necessary. The next layer is the GOB, which consists of a header, followed by the macroblocks. If the motion compensation is good, some macroblocks may be completely zero and may be skipped. The macroblock layer is next, with a header that may contain a motion vector, followed by the blocks. There are six blocks per macroblock, but blocks that are

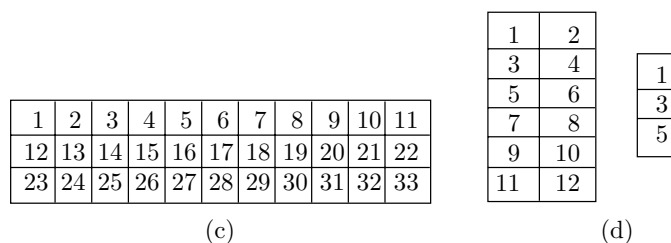
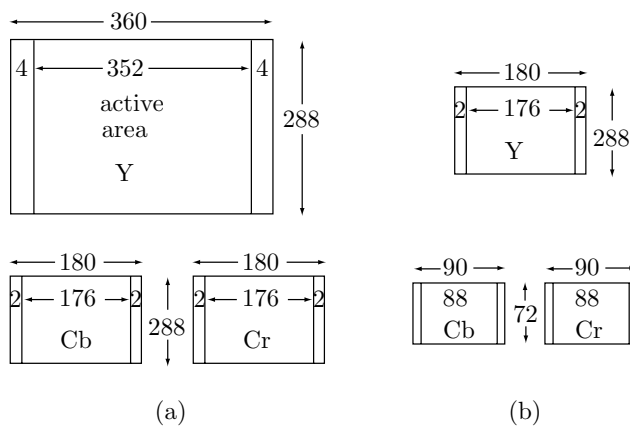


Figure 9.38: (a) CIF. (b) QCIF. (c) GOB.

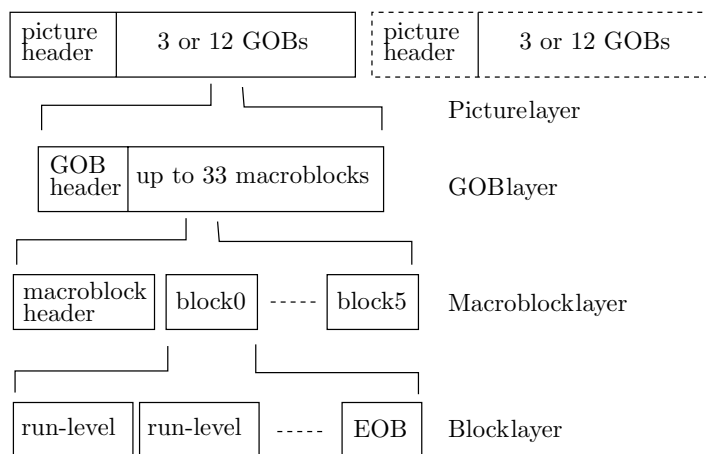


Figure 9.39: H.261 Video Sequence Layers.

zero are skipped. The block layer contains the blocks that are coded. If a block is coded, its nonzero coefficients are encoded with a run-level code. A 2-bit EOB code terminates each block.

9.9 H.264

The last years of the 20th century witnessed an unprecedented and unexpected progress in computing power, video applications, and network support for video data. Both producers and consumers of video felt the need for an advanced video codec to replace the existing video compression standards H.261, H.262, and H.263. The last of these standards, H.263, was developed around 1995 and in 2001 was already outdated. The two groups responsible for developing video compression standards, ISO-MPEG and ITU-VCEG (motion pictures experts group and video coding experts group), felt that the new standard should offer (1) increased compression efficiency, (2) support for special video applications such as videoconferencing, DVD storage, video broadcasting, and streaming over the Internet, and (3) greater reliability. In 2001, in response to this demand, the ITU started two projects. The first, short term, project was an attempt to add extra features to H.263. This project resulted in version 2 of this standard. The second, long term, project was an effort to develop a new standard for efficient (i.e., low bitrate) video compression. This project received the codename H.26L. In the same year, the ITU established a new study group, SG 16, whose members came from MPEG and VCEG, and whose mission was to study new methods, select promising ones, and implement, test, and approve the new standard. The new standard, H.264, was approved in May 2003 [ITU-T264 02], and a corrigendum was added and approved in May 2004. This standard, which is now part of the huge MPEG-4 project, is known by several names. The ITU calls it ITU-T Recommendation H.264, advanced video coding for generic audiovisual services. Its official title is advanced video coding (AVC). The ISO considers it part 10 of the MPEG-4 standard. Regardless of its title, this standard (referred to in this section as H.264) is the best video codec available at the time of writing (mid 2006).

The following is a list of some of the many references currently available for H.264. The official standard is [H.264Draft 06]. As usual with such standards, this document describes only the format of the compressed stream and the operation of the decoder. As such, it is difficult to follow. The main text is [Richardson 03], a detailed book on H.264 and MPEG-4. The integer transform of H.264 is surveyed in [H.264Transform 06]. Many papers and articles are available online at [H.264Standards 06]. Two survey papers by Iain Richardson [H.264PaperIR 06] and Rico Malvar [H.264PaperRM 06] are also widely available and serve as introductions to this topic. The discussion in this section follows both [H.264Draft 06] and [Richardson 03].

Principles of operation

The H.264 standard does not specify the operations of the encoder, but it makes sense to assume that practical implementations of the encoder and decoder will consist of the building blocks shown in Figures 9.40 and 9.41, respectively. Those familiar with older video codecs will notice that the main components of H.264 (prediction, transform, quantization, and entropy encoding) are not much different from those of its predecessors MPEG-1, MPEG-2, MPEG-4, H.261, and H.263. What makes H.264 new and original are the details of those components and the presence of the only new component, the filter. Because of this similarity, the discussion here concentrates on those features of H.264 that distinguish it from its predecessors. Readers who are interested in more details should study the principles of MPEG-1 (Section 9.6) before reading any further.

The input to a video encoder is a set of video frames (where a frame may consist of progressive or interlaced video). Each frame is encoded separately, and the encoded frame is referred to as a coded picture. There are several types of frames, mostly *I*, *P*, and *B* as discussed at the start of Section 9.5, and the order in which the frames are encoded may be different from the order in which they have to be displayed (Figure 9.9). A frame is broken up into slices, and each slice is further partitioned into macroblocks as illustrated in Figure 9.17.

In H.264, slices and macroblocks also have types. An *I* slice may have only *I*-type macroblocks, a *P* slice may have *P* and *I* macroblocks, and a *B* slice may have *B* and *I* macroblocks (there are also slices of types *SI* and *SP*). Figure 9.40 shows the main steps of the H.264 encoder. A frame F_n is predicted (each macroblock in the frame is predicted by other macroblocks in the same frame or by macroblocks from other frames). The predicted frame P is subtracted from the original F_n to produce a difference D_n , which is transformed (in the box labeled T), filtered (in Q), reordered, and entropy encoded. What is new in the H.264 encoder is the reconstruction path.

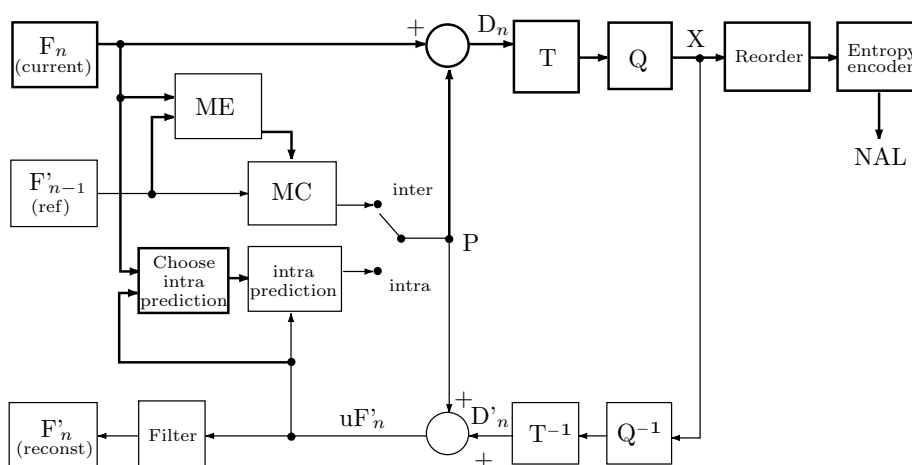


Figure 9.40: H.264 Encoder.

Even a cursory glance at Figure 9.40 shows that the encoder consists of two data paths, a “forward” path (mostly from left to right, shown in thick lines) and a “reconstruction” path (mostly from right to left, shown in thin lines). The decoder (Figure 9.41) is very similar to the encoder’s reconstruction path and has most of the latter’s components.

The main part of the encoder is its forward path. The next video frame to be compressed is denoted by F_n . The frame is partitioned into macroblocks of 16×16 pixels each, and each macroblock is encoded in intra or inter mode. In either mode, a prediction macroblock P is constructed based on a reconstructed video frame. In the intra mode, P is constructed from previously-encoded samples in the current frame n . These samples are decoded and reconstructed, becoming uF'_n in the figure. In the

inter mode, P is constructed by motion-compensated prediction from one or several reference frames (F'_{n-1} in the Figure). Notice that the prediction for each macroblock may be based on one or two frames that have already been encoded and reconstructed (these may be past or future frames). The prediction macroblock P is then subtracted from the current macroblock to produce a residual or difference macroblock D_n . This macroblock is transformed and quantized to produce a set X of quantized transform coefficients. The coefficients are reordered in zigzag and entropy encoded into a short bit string. This string, together with side information for the decoder (also entropy encoded, it includes the macroblock prediction mode, quantizer step size, motion vector information specifying the motion compensation, as well as other items), become the compressed stream which is passed to a network abstraction layer (NAL) for transmission outside the computer or proper storage. NAL consists of units, each with a header and a raw byte sequence payload (RBSP) that can be sent as packets over a network or stored as records that constitute the compressed file.

Each slice starts with a header that contains the slice type, the number of the coded picture that includes the slice, and other information. The slice data consists mostly of coded macroblocks, but may also have skip indicators, indicating macroblocks that have been skipped (not encoded).

Slices of type I (intra) contain only I macroblocks (macroblocks that are predicted from previous macroblocks in the same slice). This type of slice is used in all three profiles (profiles are discussed below).

Slices of type P (predicted) contain P and/or I macroblocks. The former type is predicted from one reference picture. This type of slice is used in all three profiles.

Slices of type B (bipredictive) contain B and/or I macroblocks. The former type is predicted from a list of reference pictures. This type of slice is used in the main and extended profiles.

Slices of type SP (switching P) are special. They facilitate switching between coding streams and they contain extended P and/or I macroblocks. This type of slice is used only in the extended profile.

Slices of type SI (switching I) are also special and contain extended SI macroblocks. This type of slice is used only in the extended profile.

The encoder has a special reconstruction path whose purpose is to reconstruct a frame for encoding of further macroblocks. The main step in this path is to decode the quantized macroblock coefficients X . They are rescaled in box Q^{-1} and inverse transformed in T^{-1} , resulting in a difference macroblock D'_n . Notice that this macroblock is different from the original difference macroblock D_n because of losses introduced by the quantization. We can consider D'_n a distorted version of D_n . The next step creates a reconstructed macroblock uF'_n (a distorted version of the original macroblock) by adding the prediction macroblock P to D'_n . Finally, a filter is applied to a series of macroblocks uF'_n in order to soften the effects of blocking. This creates a reconstructed reference frame F'_n .

The decoder (Figure 9.41) inputs a compressed bitstream from the NAL. The first two steps (entropy decoding and reordering) produce a set of quantized coefficients X . Once these are rescaled and inverse transformed they result in a D'_n identical to the D'_n of the encoder). Using the header side information from the bitstream, the decoder constructs a prediction macroblock P , identical to the original prediction P created by

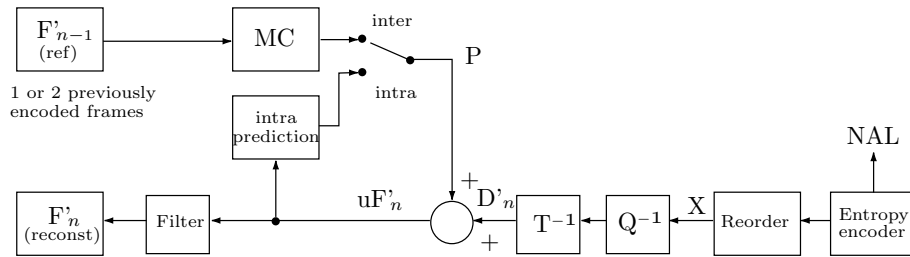


Figure 9.41: H.264 Decoder.

the encoder. In the next step, P is added to D'_n to produce uF'_n . In the final step, uF'_n is filtered to create the decoded macroblock F'_n .

Figures 9.40 and 9.41 and the paragraphs above make it clear that the reconstruction path in the encoder has an important task. It ensures that both encoder and decoder use identical reference frames to create the prediction P . It is important for the predictions P in encoder and decoder to be identical, because any changes between them tend to accumulate and lead to an increasing error or “drift” between the encoder and decoder.

The remainder of this section discusses several important features of H.264. We start with the profiles and levels. H.264 defines three profiles, baseline, main, and extended. Each profile consists of a certain set of encoding functions, and each is intended for different video applications. The baseline profile is designed for videotelephony, video conferencing, and wireless communication. It has inter and intra coding with I and P slices and performs the entropy encoding with context-adaptive variable-length codes (CAVLC). The main profile is designed to handle television broadcasting and video storage. It can deal with interlaced video, perform inter coding with B slices, and encode macroblocks with context-based arithmetic coding (CABAC). The extended profile is intended for streaming video and audio. It supports only progressive video and has modes to encode SP and SI slices. It also employs data partitioning for greater error control. Figure 9.42 shows the main components of each profile and makes it clear that the baseline profile is a subset of the extended profile.

This section continues with the main components of the baseline profile. Those who are interested in the details of the main and extended profiles are referred to [Richardson 03].

The baseline profile

This profile handles I and P slices. The macroblocks in an I slice are intra coded, meaning that each macroblock is predicted from previous macroblocks in the same slice. The macroblocks in a P slice can either be skipped, intra coded, or inter coded. The latter term means that a macroblock is predicted from the same macroblock in previously-coded pictures. H.264 uses a tree-structured motion compensation algorithm based on motion vectors as described in Section 9.5.

Once a macroblock F_n has been predicted, the prediction is subtracted from F_n to obtain a macroblock D_n of residuals. The residuals are transformed with a 4×4 integer transform, similar to the DCT. The transform coefficients are reordered (collected

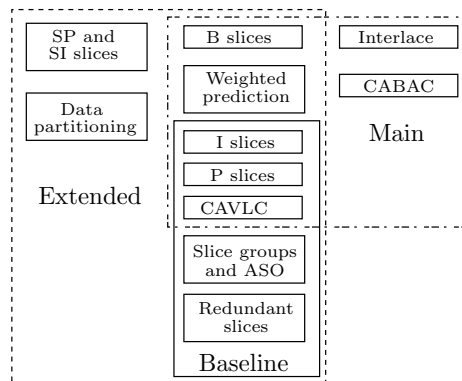


Figure 9.42: H.264 Profiles.

in zigzag order), quantized, and encoded by a context-adaptive variable-length code (CAVLC). Other information that has to go on the compressed stream is encoded with a Golomb code.

Inter prediction. The model for inter prediction is based on several frames that have already been encoded. The model is similar to that employed by older MPEG video standards, with the main differences being (1) it can deal with block sizes from 16×16 video samples down to 4×4 samples and (2) it uses fine-resolution motion vectors.

The tree-structured motion compensation algorithm starts by splitting the luma component of a 16×16 macroblock into partitions and computing a separate motion vector for each partition. Thus, each partition is motion compensated separately. A macroblock can be split in four different ways as illustrated in Figure 9.43a. The macroblock is either left as a single 16×16 block, or is split in one of three ways into (1) two 8×16 partitions, (2) two 16×8 partitions, or (3) four 8×8 partitions. If the latter split is chosen, then each partition may be further split in one of four ways as shown in Figure 9.43b.

The partition sizes listed here are for the luma color component of the macroblock. Each of the two chroma components (Cb and Cr) is partitioned in the same way as the luma component, but with half the vertical and horizontal sizes. As a result, when a motion vector is applied to the chroma partitions, its vertical and horizontal components have to be halved.

It is obvious that the partitioning scheme and the individual motion vectors have to be sent to the decoder as side information. There is therefore an important trade-off between the better compression that results from small partitions and the larger side information that small partitions require. The choice of partition size is therefore important, and the general rule is that large partitions should be selected by the encoder for uniform (or close to uniform) regions of a video frame while small partitions perform better for regions with much video noise.

Intra prediction. In this mode, the luma component of a macroblock either remains a single 16×16 block or is split into small 4×4 partitions. In the former case, the standard specifies four methods (summarized later) for predicting the macroblock. In

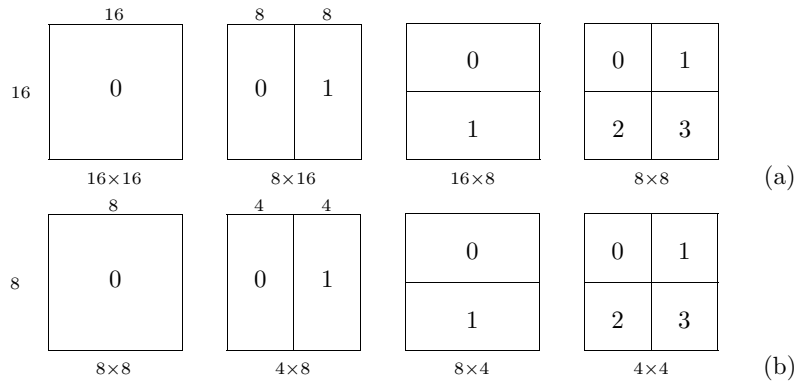


Figure 9.43: H.264 Macroblock Partitions.

the latter case, the 16 elements of a 4×4 partition are predicted in one of nine modes from 12 elements of three previously-encoded partitions to its left and above it. These 12 elements are shown in the top-left part of Figure 9.44 where they are labeled A through H, I through L, and M. Either the x or the y coordinate of these elements is -1 . Notice that these elements have already been encoded and reconstructed, which implies that they are also available to the decoder, not just to the encoder. The 16 elements of the partition to be predicted are labeled a–p, and their x and y coordinates are in the interval $[0, 3]$. The remaining nine parts of Figure 9.44 show the nine prediction modes available to the encoder. A sophisticated encoder should try all nine modes, predict the 4×4 partition nine times, and select the best prediction. The mode number for each partition should be written on the compressed stream for the use of the decoder. Following are the details of each of the nine modes.

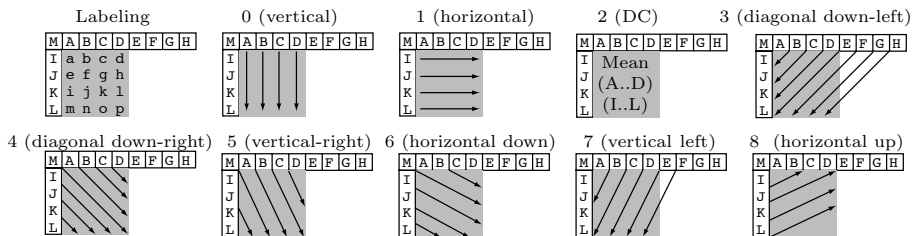


Figure 9.44: Nine Luma Prediction Modes.

Mode 0 (vertical) is simple. Elements a, e, i, and m are set to A. In general $p(x, y) = p(x, -1)$ for $x, y = 0, 1, 2$, and 3.

Mode 1 (horizontal) is similar. Elements a, b, c, and d are set to I. In general $p(x, y) = p(-1, y)$ for $x, y = 0, 1, 2$, and 3.

Mode 2 (DC). Each of the 16 elements is set to the average $(I + J + K + L + A + B + C + D + 4)/8$. If any of A, B, C, and D haven't been previously encoded, then each of the

16 elements is set to the average $(I + J + K + L + 2)/4$. Similarly, if any of I, J, K, and L haven't been previously encoded, then each of the 16 elements is set to the average $(A + B + C + D + 2)/4$. In the case where some of A, B, C, and D and some of I, J, K, and L haven't been previously encoded, each of the 16 elements is set to 128. This mode can always be selected and used by the encoder.

Mode 3 (diagonal down-left). This mode can be used only if all eight elements A through H have been encoded. Element p is set to $(G + 3H + 2)/4$, and elements $p(x, y)$ where $x \neq 3$ or $y \neq 3$ are set to $[p(x + y, -1) + 2p(x + y + 1, -1) + p(x + y + 2, -1) + 2]/4$. As an example, element g , whose coordinates are (2, 1), is set to the average $[p(3, -1) + 2p(4, -1) + p(5, -1) + 2]/4 = [D + 2E + F + 2]/4$.

Mode 4 (diagonal down-right). This mode can be used only if elements A through D, M, and I through L have been encoded. The four diagonal elements a , f , k , and p are set to $[A + 2M + I + 2]/4$. Elements $p(x, y)$ where $x > y$ are set to $[p(x - y - 2, -1) + 2p(x - y - 1, -1) + p(x - y, -1) + 2]/4$. Elements where $x < y$ are set to a similar average but with $y - x$ instead of $x - y$. Example. Element h , whose coordinates are (3, 1), is set to $[p(0, -1) + 2p(1, -1) + p(2, -1) + 2]/4 = [A + 2B + C + 2]/4$.

Mode 5 (vertical right). This mode can be used only if elements A through D, M, and I through L have been encoded. We set variable z to $2x - y$. If z is even (equals 0, 2, 4, or 6), element $p(x, y)$ is set to $[p(x - y/2 - 1, -1) + p(x - y/2, -1) + 1]/2$. If z equals 1, 3, or 5, element $p(x, y)$ is set to $[p((x - y/2 - 2, -1) + 2p(x - y/2 - 1, -1) + p(x - y/2, -1) + 2)]/4$. If $z = -1$, element $p(x, y)$ is set to $[A + 2M + I + 2]/4$. Finally, if $z = -2$ or -3 , element $p(x, y)$ is set to $[p(-1, y - 1) + 2p(-1, y - 2) + p(-1, y - 3) + 2]/4$.

Mode 6 (horizontal down). This mode can be used only if elements A through D, M, and I through L have been encoded. We set variable z to $2y - x$. If z is even (equals 0, 2, 4, or 6), element $p(x, y)$ is set to $[p(-1, y - x/2 - 1) + p(-1, y - x/2) + 1]/2$. If z equals 1, 3, or 5, element $p(x, y)$ is set to $[p(-1, y - x/2 - 2) + 2p(-1, y - x/2 - 1) + p(-1, y - x/2) + 2]/4$. If $z = -1$, element $p(x, y)$ is set to $[A + 2M + I + 2]/4$. Finally, if $z = -2$ or -3 , element $p(x, y)$ is set to $[(p(x - 1, -1) + 2p(x - 2, -1) + p(x - 3, -1) + 2)]/4$.

Mode 7 (vertical left). This mode can be used only if elements A through H have been encoded. If y equals 0 or 2, element $p(x, y)$ is set to $[p(x + y/2, -1) + p(x + y/2 + 1, -1) + 1]/2$. If y equals 1 or 3, element $p(x, y)$ is set to $[p(x + y/2, -1) + 2p(x + y/2 + 1, -1) + p(x + y/2 + 2, -1) + 2]/4$.

Mode 8 (horizontal up). This mode can be used only if elements I through L have been encoded. We set z to $x + 2y$. If z is even (equals 0, 2, or 4), element $p(x, y)$ is set to $[p(-1, y + x/2 - 1) + p(-1, y + x/2) + 1]/2$. If z equals 1 or 3, element $p(x, y)$ is set to $[p(-1, y + x/2) + 2p(-1, y + x/2 + 1) + p(-1, y + x/2 + 2) + 2]/4$. If z equals 5, $p(x, y)$ is set to $[K + 3L + 2]/4$. If $z > 5$, $p(x, y)$ is set to L.

If the encoder decides to predict an entire 16×16 macroblock of luma components without splitting it, the standard specifies four prediction modes where modes 0, 1, and 2 are identical to the corresponding modes for 4×4 partitions described above and mode 3 specifies a special linear function of the elements above and to the left of the 16×16 macroblock.

There are also four prediction modes for the 4×4 blocks of chroma components.

Deblocking filter. Individual macroblocks are part of a video frame. The fact that each macroblock is predicted separately introduces blocking distortions, which is why H.264 has a new component, the deblocking filter, designed to reduce these distortions. The

filter is applied after the inverse transform. Naturally, the inverse transform is computed by the decoder, but the reader should recall, with the help of Figure 9.40, that in H.264 the inverse transform is also performed by the reconstruction path of the encoder (box T^{-1} , before the macroblock is reconstructed and stored for later predictions). The filter improves the appearance of decoded video frames by smoothing the edges of blocks. Without the filter, decompressed blocks often feature artifacts due to small differences between the edges of adjacent blocks.

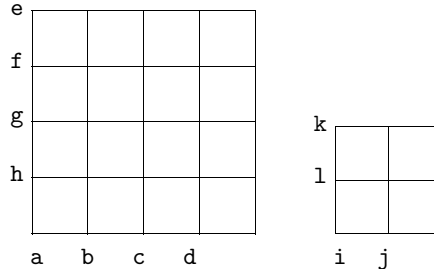


Figure 9.45: Edge Designation in a Macroblock.

The filter is applied to the horizontal or vertical edges of 4×4 blocks of a macroblock, except for edges located on the boundary of a slice, in the following order (Figure 9.45):

1. Filter the four vertical boundaries **a**, **b**, **c**, and **d** of the 16×16 luma component of the macroblock.
2. Filter the four horizontal boundaries **e**, **f**, **g**, and **h** of the 16×16 luma component of the macroblock.
3. Filter the two vertical boundaries **i** and **j** of each 8×8 luma component of the macroblock.
4. Filter the two horizontal boundaries **k** and **l** of each 8×8 luma component of the macroblock.

The filtering operation is an n -tap discrete wavelet transform (Section 8.7) where n can be 3, 4, or 5. Figure 9.46 shows eight samples **p0** through **p3** and **q0** through **q3** being filtered, four on each side of the boundary (vertical and horizontal) between macroblocks. Samples **p0** and **q0** are on the boundary. The filtering operation can have different strengths, and it affects up to three samples on each side of the boundary. The strength of the filtering is an integer between 0 (no filtering) and 4 (maximum filtering), and it depends on the current quantizer, on the decoding modes (inter or intra) of adjacent macroblocks, and on the gradient of video samples across the boundary.

The H.264 standard defines two threshold parameters α and β whose values depend on the average quantizer parameters of the two blocks **p** and **q** in question. These parameters help to determine whether filtering should take place. A set of eight video samples **pi qi** is filtered only if $|p0 - q0| < \alpha$, $|p1 - p0| < \beta$, and $|q1 - q0| \leq \beta$.

When the filtering strength is between 1 and 3, the filtering process proceeds as follows: Samples **p1**, **p0**, **q0**, and **q1** are processed by a 4-tap filter to produce **p'0** and **q'0**. If $|p2 - p0| < \beta$, another 4-tap filter is applied to **p2**, **p1**, **p0**, and **q0** to produce **p'1**. If $|q2 - q0| < \beta$, a third 4-tap filter is applied to **q2**, **q1**, **q0**, and **p0** to produce

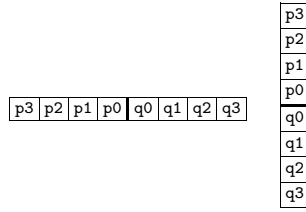


Figure 9.46: Horizontal and Vertical Boundaries.

$q'1$ (the last two steps are performed for the luma component only). When the filtering strength is 4, the filtering process is similar, but longer. It has more steps and employs 3-, 4-, and 5-tap filters.

Transform. Figures 9.40 and 9.41 show that blocks D_n of residuals are transformed (in box T) by the encoder and reverse transformed (box T^{-1}) by both decoder and the reconstruction path of the encoder. Most of the residuals are transformed in H.264 by a special 4×4 version of the discrete cosine transform (DCT, Section 7.8). Recall that an $n \times n$ block results, after being transformed by the DCT, in an $n \times n$ block of transform coefficients where the top-left coefficient is DC and the remaining $n^2 - 1$ coefficients are AC. In H.264, certain DC coefficients are transformed by the Walsh-Hadamard transform (WHT, Section 7.7.2). The DCT and WHT transforms employed by H.264 are special and have the following features:

1. They use only integers, so there is no loss of accuracy.
2. The core parts of the transforms use only additions and subtractions.
3. Part of the DCT is scaling multiplication and this is integrated into the quantizer, thereby reducing the number of multiplications.

Given a 4×4 block $\mathbf{X}$ of residuals, the special DCT employed by H.264 can be written in the form

$$\mathbf{Y} = \mathbf{A} \mathbf{X} \mathbf{A}^T = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{bmatrix} \mathbf{X} \begin{bmatrix} a & b & a & c \\ a & c & -a & -b \\ a & -c & -a & b \\ a & -b & a & -c \end{bmatrix},$$

where

$$a = \frac{1}{2}, \quad b = \sqrt{\frac{1}{2}} \cos \frac{\pi}{8}, \quad \text{and} \quad c = \sqrt{\frac{1}{2}} \cos \frac{3\pi}{8}.$$

This can also be written in the form

$$\mathbf{Y} = (\mathbf{C} \mathbf{X} \mathbf{C}^T) \otimes \mathbf{E} = \left\{ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & d & -d & -1 \\ 1 & -1 & -1 & 1 \\ d & -1 & 1 & -d \end{bmatrix} \mathbf{X} \begin{bmatrix} 1 & 1 & 1 & d \\ 1 & d & -1 & -1 \\ 1 & -d & -1 & 1 \\ 1 & -1 & 1 & -d \end{bmatrix} \right\} \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix},$$

where $(\mathbf{C} \mathbf{X} \mathbf{C}^T)$ is a core transform (where $d = c/b \approx 0.414$), matrix $\mathbf{E}$ consists of

scaling factors, and the $\otimes$ operation indicates scalar matrix multiplication (pairs of corresponding matrix elements are multiplied).

In order to simplify the computations and ensure that the transform remains orthogonal, the following approximate values are actually used

$$a = \frac{1}{2}, \quad b = \sqrt{\frac{2}{5}}, \quad d = \frac{1}{2},$$

bringing this special forward transform to the form

$$\mathbf{Y} = \left\{ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \mathbf{X} \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \right\} \otimes \begin{bmatrix} a^2 & ab/2 & a^2 & ab/2 \\ ab/2 & b^2/4 & ab/2 & b^2/4 \\ a^2 & ab/2 & a^2 & ab/2 \\ ab/2 & b^2/4 & ab/2 & b^2/4 \end{bmatrix}.$$

Following the transform, each transform coefficient Y_{ij} is quantized by the simple operation $Z_{ij} = \text{round}(Y_{ij}/Q_{\text{step}})$ where Q_{step} is the quantization step. The rounding operation is also special. The encoder may decide to use “floor” or “ceiling” instead of rounding. The H.264 standard specifies 52 quantization steps Q_{step} that range from 0.625 up to 224 depending on the quantization parameter QP whose values range from 0 to 51. Table 9.47 lists these values and shows that every increment of 6 in QP doubles Q_{step} . The wide range of values of the latter quantity allows a sophisticated encoder to precisely control the trade-off between low bitrate and high quality. The values of QP and Q_{step} may be different for the luma and chroma transformed blocks.

QP	0	1	2	3	4	5	6	7	8	9	10	11	12	...
Q_{step}	0.625	0.6875	0.8125	0.875	1	1.125	1.25	1.375	1.625	1.75	2	2.25	2.5	...
QP	...	18	...	24	...	30	...	36	...	42	...	48	...	51
Q_{step}		5		10		20		40		80		160		224

Table 9.47: Quantization Step Sizes.

If a macroblock is encoded in 16×16 intra prediction mode, then each 4×4 block of residuals in the macroblock is first transformed by the core transform described above, and then the 4×4 Walsh-Hadamard transform is applied to the resulting DC coefficients as follows

$$\mathbf{Y}_D = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \mathbf{W}_D \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix},$$

where $\mathbf{W}_D$ is a 4×4 block of DC coefficients.

After a macroblock is predicted and the residuals are transformed and quantized, they are reordered. Each 4×4 block of quantized transform coefficients is scanned in zigzag order (Figure 1.8b). If the macroblock is encoded in intra mode, there will be 16 DC coefficients (the top-left coefficient of each 4×4 luma block). These 16 coefficients

are arranged in their own 4×4 block which is scanned in zigzag. The remaining 15 AC coefficients in each 4×4 luma block are also scanned in the same zigzag order. The same reordering is applied to the smaller chroma blocks.

Entropy coding

H.264 involves many items—counts, parameters, and flags—that have to be encoded. The baseline profile employs variable-length codes and a special context adaptive arithmetic coding (CABAC) scheme to encode parameters such as macroblock type, reference frame index, motion vectors differences, and differences of quantization parameters. The discussion in this section concentrates on the encoding of the zigzag sequence of quantized transform coefficients. This sequence is encoded in a special context-adaptive variable-length coding scheme (CAVLC) based on Golomb codes (Section 3.24).

CAVLC is used to encode reordered zigzag sequences of either 4×4 or 2×2 blocks of quantized transform coefficients. The method is designed to take advantage of the following features of this special type of data.

1. Such a sequence is sparse, has runs of zeros, and normally ends with such a run.
2. Among the nonzero coefficients, such a sequence has many +1s and -1s and these are referred to as trailing ± 1 s.
3. The number of nonzero coefficients in such a sequence is often similar to the corresponding numbers in neighboring blocks. Thus, these numbers are correlated across adjacent blocks.
4. The first (DC) coefficient tends to be the largest one in the sequence, and subsequent nonzero coefficients tend to get smaller.

CAVLC starts by scanning and analyzing the sequence of 16 coefficients. This step assigns values to the six encoding parameters `coeff_token`, `trailing_ones_sign_flag`, `level_prefix`, `level_suffix`, `total_zeros`, and `run_before`. These parameters are then used in five steps to select tables of variable-length codes and codes within those tables. The codes are concatenated to form a binary string that is later formatted by the NAL stage and is finally written on the compressed stream. The five encoding steps are as follows:

Step 1. Parameter `coeff_token` is the number of nonzero coefficients and trailing ± 1 s among the 16 coefficients in the sequence. The number of nonzero coefficients can be from 0 to 16. The number of trailing ± 1 s can be up to the number of nonzero coefficients, but `coeff_token` takes into account (i.e., it signals) up to three trailing ± 1 s. Any trailing ± 1 s beyond the first three are encoded separately as other nonzero coefficients. If the number of nonzero coefficients is zero, there can be no trailing ± 1 s. If the number of nonzero coefficients is one, there can be zero or one trailing ± 1 s. If the number of nonzero coefficients is two, the number of trailing ± 1 s can be 0, 1, or 2. If the number of nonzero coefficients is greater than two, there can be between zero and three trailing ± 1 s that are signaled by `coeff_token`. There can be between zero and 16 nonzero coefficients, so the total number of values of parameter `coeff_token` is $1 + 2 + 3 + 14 \times 4 = 62$. Thus, the H.264 standard should specify a table of 62 variable-length codes to encode the value of `coeff_token`. However, because the number of nonzero coefficients is correlated across neighboring blocks, the standard specifies four such tables (table 9–5 on page 159 of [H.264Draft 06]).

To select one of the four tables, the encoder counts the numbers nA and nB of nonzero coefficients in the blocks to the left of and above the current block (if such blocks exist, they have already been encoded). A value nC is computed from nA and nB as follows. If both blocks exist, then $nC = \text{round}[(nA + nB)/2]$. If only the left block exists, then $nC = nA$. If only the block above exists, then $nC = nB$. Otherwise $nC = 0$. The resulting value of nC is therefore in the interval $[0, 16]$ and it is used to select one of four tables as listed in Table 9.48.

nC:	0	1	2	3	4	5	6	7	8	9	...
Table:	1	1	2	2	3	3	3	3	4	4	...

Table 9.48: nC Values to Select a Table.

Step 2. Parameter `trailing_ones_sign_flag` is encoded as a single bit for each trailing ± 1 signaled by `coeff_token`. The trailing ± 1 s are scanned in reverse zigzag order, and the (up to three) sign bits that are generated are appended to the binary string generated so far.

Step 3. The next set of bits appended to the binary string encodes the values of the remaining nonzero coefficients (i.e., the nonzero coefficients except those trailing ± 1 s that were signaled by `coeff_token`). These nonzero coefficients are scanned in reverse zigzag order, and each is encoded in two parts, as a prefix (`level_prefix`) and suffix (`level_suffix`). The prefix is an integer between 0 and 15 and is encoded with a variable-length code specified by the standard (table 9–6 on page 162 of [H.264Draft 06]). The length of the suffix part is determined by parameter `suffixLength`. This parameter can have values between 0 and 6 and is adapted while the nonzero coefficients are being located and encoded. The principle of adapting `suffixLength` is to match the suffix of a coefficient to the magnitudes of the recently-encoded nonzero coefficients. The following rules govern the way `suffixLength` is adapted:

1. Parameter `suffixlength` is initialized to 0 (but if there are more than 10 nonzero coefficients and fewer than three trailing ± 1 s, it is initialized to 1).
2. The next coefficient in reverse zigzag order is encoded.
3. If the magnitude of that coefficient is greater than the current threshold, increment `suffixLength` by 1. This also selects another threshold. The values of the thresholds are listed in Table 9.49.

suffixLength:	0	1	2	3	4	5	6
Threshold:	0	3	6	12	24	48	na

Table 9.49: Thresholds to Increment `suffixLength`.

Step 4. Parameter `total_zeros` is the number of zeros preceding the last nonzero coefficient. (This number does not include the last run of zeros, the run that follows the last nonzero coefficient.) `total_zeros` is an integer in the interval $[0, 15]$ and it is encoded with a variable-length code specified by the standard (tables 9–7 and 9–8 on page 163 of [H.264Draft 06]). This parameter and the next one (`run_before`) together are sufficient to encode all the runs of zeros up to the last nonzero coefficient.

Step 5. Parameter `run_before` is the length of a run of zeros preceding a nonzero coefficient. This parameter is determined and encoded for each nonzero coefficient (except, of course, the first one) in reverse zigzag order. The length of such a run can be zero, but its largest value is 14 (this happens in the rare case where the DC coefficient is followed by a run of 14 zeros, which is followed in turn by a nonzero coefficient). While the encoder encodes these run lengths, it keeps track of their total length and terminates this step when the total reaches `total_zeros` (i.e., when no more zeros remain to be encoded). The standard specifies a table of variable-length codes (table 9–10 on page 164 of [H.264Draft 06]) to encode these run lengths.

Once the five steps have encoded all the nonzero coefficients and all the runs of zeros between them, there is no need to encode the last run of zeros, the one (if any) that follows the last nonzero coefficient, because the length of this run will be known to the decoder once it has decoded the entire bit string.

The conscientious reader should consult [Richardson 03] for examples of encoded sequences and also compare the CAVLC method of encoding a sequence of DCT transform coefficients with the (much simpler) way similar sequences are encoded by JPEG (Section 7.10.4).

9.10 H.264/AVC Scalable Video Coding

The Joint Video Team, an organization that combines members of the ITU-T Video coding Group (VCEG) and members of the ISO/IEC Moving Picture expert Group (MPEG), has extended H.264/AVC with an Amendment defining a scalable video encoder called *Scalable Video Coding* or *SVC*. The purpose of this extension is to specify a compressed bitstream that supports temporal, spatial, and quality scalable video coding, while retaining a base layer that is still backward compatible with H.264/AVC.

A bitstream is called *multi-layer* or *scalable* if parts of the bitstream can be discarded to generate a substream that is still valid and decodable by a target decoder, but with a reconstruction quality that is lower than the quality obtained by decoding the entire bitstream. A bitstream that does not have this property is called *single-layer*.

A scalable video bitstream can be easily adapted to the capacity of a transmission channel, to the computational power of a decoder, or to the resolution of a display device. Furthermore, since the layers in a scalable bitstream can be easily prioritized, assigning unequal error protection to the layers can improve error resilience and gracefully degrade the quality of the reconstructed signal in the presence of channel error and packet loss.

Scalable video compression has not been introduced by SVC. In fact, almost all the recent international coding standards such as MPEG-2 Video, H.263, and MPEG-4 Visual already support the most useful forms of scalability. Unfortunately, the lack of coding efficiency and the increased complexity of the scalable tools offered by these standards has limited the deployment of this technology.

In order to be useful, a scalable encoder should generate a bitstream substantially more compact than the independent compression of all its component layers, so to provide a convenient alternative to *simulcast*, the simultaneous broadcasting of all the layers independently compressed. Furthermore, the increase in decoder complexity should be lower than the complexity of *transcoding* the bitstream, the conversion of a layer into

another directly in the digital domain. It is obvious that the scalable compression of a layer other than the base layer will, in general, be slightly worse than its non-scalable compression. However, it is important to design the system so that performance loss is kept to a minimum.

Designed according to these principles, the Scalable Video Coding extension to H.264/AVC provides temporal, spatial, and quality scalability with the mechanisms outlined in the following paragraphs.

Temporal Scalability: Video coding standards are traditionally “decoder” or “bitstream” standards. The format of the bitstream and the behavior of a reference decoder are specified by the standard, while leaving the implementors complete freedom in the design of the encoder. For this reason, a standard will specify a tool only when strictly necessary to achieve a given functionality. Following this basic guideline, SVC supports temporal scalability without adding any new tool to H.264/AVC.

Syntactically, the only extension is the signalling of a *temporal layer identification* T . T is an integer that starts from 0 (to signal the pictures belonging to the base layer) and grows as the temporal resolution of a layer increases. The temporal layer identification is necessary because a decoder that wants to decode a temporal layer $T = t$ should be able to isolate and decode all pictures with $T \leq t$ and discard any remaining picture. The temporal layer identification is contained in a header that is peculiar to SVC, the *SVC NAL unit header*. This additional header is stored in the so-called *prefix NAL units*. Prefix NAL units can be discarded in order to generate a base layer with NALs that are fully H.264/AVC compliant.

H.264/AVC already allows a pretty flexible temporal scalability with the use of hierarchical prediction. Figures 9.50, 9.51, and 9.52 show three possible alternatives that implement dyadic and non-dyadic temporal scalability with different delays. The arrows indicate the reference pictures used in the encoding of each picture.

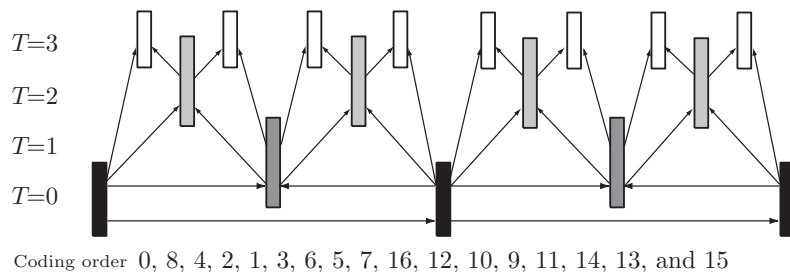


Figure 9.50: Dyadic Hierarchical Prediction with Four Temporal Layers.

The hierarchical structure of Figure 9.50 uses a GOP (group of pictures) of eight frames to specify a four-layer dyadic scheme. In a dyadic scheme, each layer doubles the temporal resolution of the layer preceding it. The gray and white frames in the figure are B-pictures, respectively, used and unused as reference in the motion compensated prediction. Figure 9.51 depicts a three-layer non-dyadic scheme.

The hierarchical structures depicted in Figures 9.50 and 9.51 are capable of generating a temporally scalable bitstream with a compression of the upper layers that is

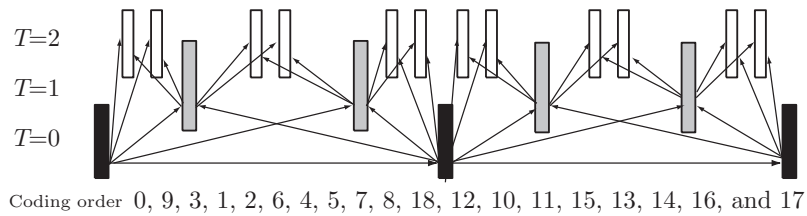


Figure 9.51: Non-dyadic Hierarchical Prediction with Three Temporal Layers.

substantially better than what could be achieved with a single layer “IPPP” coding scheme. This is apparently counterintuitive, since we have said before that scalability slightly penalizes compression efficiency. The reason for this apparent contradiction is found in the delay introduced by these schemes. The hierarchical structures depicted in Figure 9.50 and Figure 9.51 introduce delays of seven and eight pictures, respectively. These delays are the “price” paid for the better compression.

It is important to notice though that temporal scalability can be implemented with arbitrary delays, all we have to do is to appropriately constrain the prediction structure. Figure 9.52 shows a zero-delay dyadic scheme that does not use B-pictures. A temporally scalable prediction that is implemented via a low-delay scheme will eventually exhibit a reduced coding efficiency.

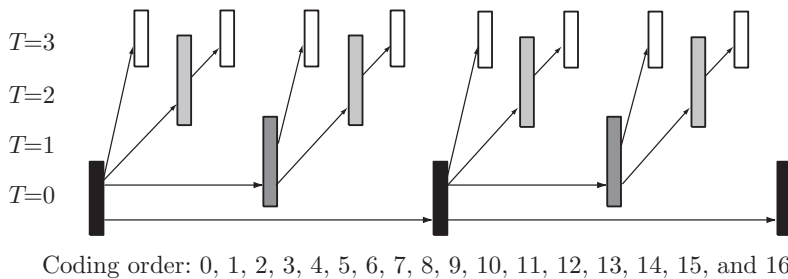


Figure 9.52: Low-delay Dyadic Hierarchical Prediction with Four Temporal Layers.

Spatial Scalability: With a mechanism similar to the temporal layer identification described above, SVC introduces the concept of *dependency identifier*, D . The dependency identifier describes the relationship between spatial layers and is also stored in a prefix NAL unit. The value of D is zero for the base layer ($T = 0$) and increases by 1 with each spatial layer.

In SVC, each layer can use an independent motion compensation. Additionally, SVC exploits the correlation of layers with the following three inter-layer prediction mechanisms:

- *Inter-Layer Motion Prediction:* The partitioning used in a block in the reference layer is upsampled and used to infer the partitioning of the corresponding macroblock

in the enhancement layer. Motion vectors from the block in the reference layer are eventually scaled before being used in the enhancement layer.

- *Inter-Layer Residual Prediction:* It can be used on all inter-coded macroblocks and uses the upsampled residual of the corresponding reference block. Upsampling is performed with a bilinear filter.
- *Inter-Layer Intra Prediction:* The prediction signal of intra coded blocks in the reference layer is upsampled and used to predict the corresponding macroblock in the enhancement layer.

In general, the coding loss introduced by spatial scalability is observable only in the enhancement layers, since the base layer is often encoded independently by a regular H.264/AVC encoder. However, since enhancement layers are predicted starting from the base layer, joint optimization of all layers is known to reduce coding loss at the cost of more complex encoding. With the use of joint optimization, the bit rate increase due to spatial scalability can be as low as 10%.

The main restriction imposed by spatial scalability is that horizontal and vertical resolutions of the video sequence cannot decrease from one layer to the next. The pictures in an enhancement layer are allowed to have the same resolution as the pictures in the reference layer. This is a feature that enables scalable encoding of video formats with dimensions that are not necessarily multiples of each other. For example, SVC can encode a base layer representing a video sequence in QVGA (Quarter VGA) format, with pictures having 320×240 pixels. The first enhancement layer can encode the same sequence, but at a higher resolution, for example, a VGA version of the video with pictures of 640×480 pixels. A second enhancement layer can then be used to represent the original video in WVGA (Wide VGA) format, at 800×480 pixels (Figure 9.53). In this case, the second enhancement layer will only need to encode columns that have been cropped out to obtain the VGA sequence from the WVGA original.

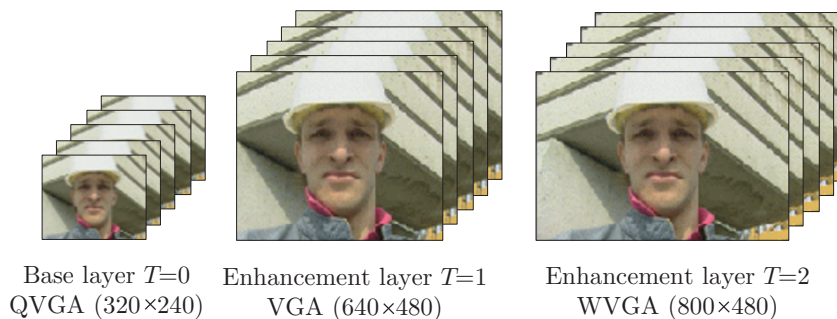


Figure 9.53: Spatial Layers Encoding QVGA, VGA, and WVGA Video Sequences.

In SVC, the coding order of a picture cannot change from one layer to another. However, lower layers are not required to contain information on all pictures. This option allows combination of spatial and temporal scalability.

Quality Scalability: A coarse form of quality scalability can be easily implemented as a special case of spatial scalability, where the reference and the enhancement layers have identical size and resolution but different quantization parameters. While simple, this solution has the major drawback that only a prefixed number of quality levels can be reconstructed, one for each layer present in the bitstream. SVC adds to this possibility a more refined form of quality scalability in which a part of the packets that constitute the enhancement layer can be discarded in order to match a target bit rate that is between the rate of the reference layer and the rate of the enhancement layer. Enhancement layer's packets can be randomly discarded until the target bit rate is reached. A better alternative, which is also supported by the SVC syntax, is assigning priority level to packets, so that packets can be selectively discarded in order of increasing priority.

A critical issue in the design of a quality scalable video encoder is the decision of which layer should be used in the motion estimation. The highest coding efficiency is obtained when motion estimation is performed on the pictures with the best quality, and so, on the highest quality enhancement layer. However, discarding packets from the enhancement layer compromises the quality of the reference frames used by the decoder in the motion compensation. The difference between the reference frames used by the encoder in the motion estimation and the reference frames used by the decoder in the motion compensation generates an error that is known as *drift*. Using only the base layer in the motion estimation prevents drift while reducing coding effectiveness.

SVC compromises between these two solutions by introducing the concept of *key pictures*. A key picture is one in which only the base layer is used in the motion compensated prediction. All non-key pictures will instead use the enhancement layer. Key pictures control drift by periodically resynchronizing encoder and decoder. Figure 9.54 shows an example where key pictures (marked in gray) are periodically inserted in the base and in the enhancement layer. The arrows represent the pictures used in the motion compensated prediction. The bit rate increase due to the introduction of quality scalability is typically less than 10%.

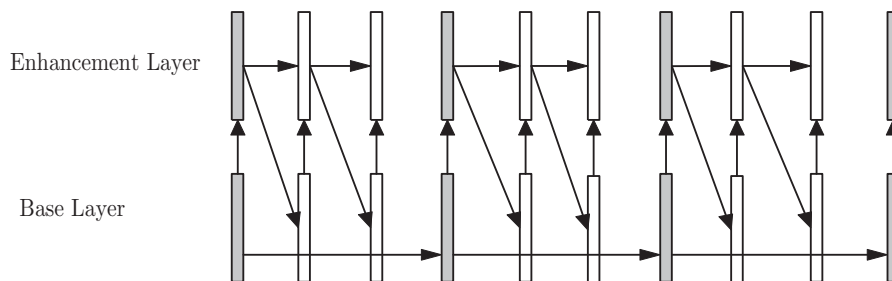


Figure 9.54: Key Pictures (Gray) Used to Control Drift in Quality Scalable Video Coding.

A detailed description of the H.264/AVC scalable video coding extension can be found in [Schwarz et. al. 07] and [Segall and Sullivan 07], both published in a special issue of the *IEEE Transaction on Circuits and Systems for Video Technology* dedicated to SVC.

9.11 VC-1

Political and social revolutions are common and some consider them the engine that drives human history, but scientific and technical progress is mostly evolutionary. Among the very few scientific revolutions that immediately spring to mind are relativity, quantum mechanics, and Newton's theory of gravity (in physics), Natural selection and the genetic code (in biology), the atomic hypothesis (in chemistry), and information theory. As in other scientific fields, most progress in data compression is evolutionary and VC-1, the video codec described here, is no exception. It represents an evolution of the successful DCT-based video codec design that is found in popular video codecs such as H.261, H.263, MPEG-1, MPEG-2, and MPEG-4 Part 2.

Like most video codecs that preceded it, VC-1 is a hybrid codec that is based on the two pillars of video compression, namely a transform (often the DCT, Section 7.8) combined with motion compensated prediction (Section 9.5). The former is used to reduce spatial redundancy, while the latter is used to reduce temporal redundancy. Like other video codecs, it extends these techniques and adds other "bells and whistles." One of the chief goals of the VC-1 project was to develop an algorithm that can encode interlaced video without first converting it to progressive, and this feature, among others, has made VC-1 attractive to video professionals.

VC-1 (the name stands for video codec) is a video codec specification that was adopted in 2006 as a standard by the Society of Motion Picture and Television Engineers (SMPTE). SMPTE (located at [SMPTE 08]) is the pre-eminent society of film and video experts, with members in 85 countries worldwide. Similar to the ITU, the standards that SMPTE publishes are voluntary, but are nevertheless widely adopted, implemented, and employed by companies doing business in video, motion pictures, and digital cinema. Within SMPTE, the VC-1 standard was developed by C42, the Video Compression Technology Committee, which is responsible for all video technologies in SMPTE.

VC-1 has been implemented by Microsoft as Microsoft Windows Media Video (WMV) 9 [WMV 08]. The formal title of VC-1/WMV standard is SMPTE 421M [SMPTE-421M 08] and there are two documents (SMPTE RP227 and SMPTE RP228) that accompany it. They describe VC-1 transport (details about carrying VC-1 elementary streams in MPEG-2 Program and Transport Streams) and conformance (the test procedures and criteria for determining conformance to the 421M specifications). As usual, the formal standard specifies only the syntax of the compressed stream and the operations of the decoder; the precise steps of the encoder are left unspecified, in the hope that innovative programmers will come up with clever algorithms and shortcuts to improve the encoder's performance and speed it up.

References. [SMPTE-421M 08] and [Lee and Kalva 08] are the two main references. The former is the formal specification of this standard, which many people may find cryptic, while the latter may include too much detail for most readers. Those interested in short, easy-to-understand overviews may want to consult [WMV review 08] and [WikiVC-1 08].

VC-1 adoption. Notwithstanding its tender age, VC-1 has been favorably received and adopted by the digital video industry. The main reasons for its popularity are its performance combined with the fact that it is well documented, extremely stable, and easily licensable. Of special notice are the following adopters:

- High-definition DVD. The DVD forum (dvdforum.org) has mandated VC-1, H.264, and MPEG-2 for the HD DVD format.
- The Blu-ray Disc Association (blu-raydisc.com) has mandated the same three codecs for their Blu-ray Disc format (so named because of its use of blue laser).
- The FVD (forward versatile disc) standard from Taiwan has adopted VC-1 as its only mandated video codec. (FVD is the Taiwanese alternative for the high-definition storing optical media based on red laser technology.)
- Several DSP and chip manufacturers have plans (or even devices) to implement VC-1 in hardware.
- VC-1 is already used for professional video broadcast. Several companies are already offering products, ranging from encoders and decoders to professional video test equipment, that use VC-1.
- The Digital Living Network Alliance (DLNA, dlna.org) is a group of 245 companies that agreed to adopt compatible standards and make compatible equipment in the areas of consumer electronics, computer, and mobile devices. VC-1 is one of several compression formats adopted by the DLNA.
- The acronym DVB-H stands for Digital Video Broadcasting - Handheld (dvb-h.org). This is a set of specifications for bringing broadcast services to battery-powered handheld receivers. Again, VC-1 is one of the compression formats adopted by this standard.
- VC-1 has been designated by Microsoft as the Xbox 360 video game console's official video codec. Thus, game developers may utilize VC-1 for any video included with games.
- The FFmpeg project (<http://ffmpeg.org/>) includes a free VC-1 decoder (FFmpeg is a computer program that can record, convert, and stream digital audio and video in numerous formats.)

9.11.1 Overview

VC-1 is complex. It includes many parameters, features, and processes. The description here attempts to cover the general design and the main parts of this algorithm. The only items that are described in detail are the pixel interpolation and the transform. The formal definition of VC-1 simply lists the steps and parameters used by these items, but the discussion here tries to explain why these specific steps and constants were chosen by the codec's developers.

Most video codecs are designed for a specific application and a particular bitrate, but VC-1 is intended for use in a wide variety of video applications, and must therefore perform well at bitrates that may vary from very low (low-resolution, 160×120 pixels video, requiring about 10 Kbps) to very high ($2,048 \times 1,536$ pixels, or about 135 Mbps, for high-definition video). It is expected that a careful implementation of the VC-1 encoder would be up to three times as efficient as MPEG-2 and up to twice as good as MPEG-4 Part 2.

As has already been mentioned, the main encoding techniques used by VC-1 are a (16-bit integer) transform and block-based motion compensation. These are optimized in several ways and, with the help of several innovations, result in a fast, efficient video

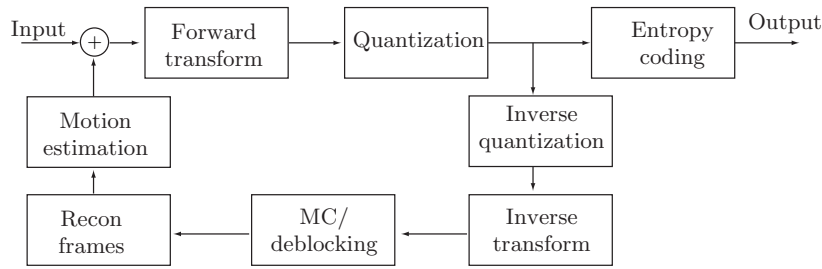


Figure 9.55: VC-1 Main Block Diagram.

codec. Figure 9.55 is a block diagram of the main components of the VC-1 encoder (where MC stands for motion compensation).

The figure illustrates the main steps in encoding a frame. Blocks of prediction residuals are transformed, the transformation coefficients are quantized, entropy encoded, and output. The motion compensation part involves inverse quantization, deblocking, reconstructing each frame, predicting the motion (the differences between this frame and its predecessor/seccessor), and using the prediction to improve the encoding of blocks in the next frame.

The experienced reader will notice that the various MPEG methods feature similar block diagrams. Thus, the superior performance of VC-1 stems from its innovations, the main ones of which are listed here:

- **Adaptive Block-Size Transform.** The JPEG standard (Section 7.10) applies the DCT to blocks of 8×8 pixels, and these dimensions have become a tradition in image and video compression. Users of JPEG know from long experience that setting the JPEG parameters to maximum compression often results in compression artifacts (known as blocking or ringing, see Figure 7.57) along block boundaries. This phenomenon is caused by differences between pixel blocks due to the short range of pixel correlation. Thus, image regions with low pixel correlation compress better when the DCT (or any other transform) is applied to smaller blocks.

VC-1 can transform a block of 8×8 pixels in one of the following ways (Figure 9.56): As a single 8×8 block, as two 8×4 blocks, as two 4×8 blocks, or as four 4×4 blocks. The encoder decides which transform size to use for any given block (although the definition of the standard does not specify how this decision should be made).

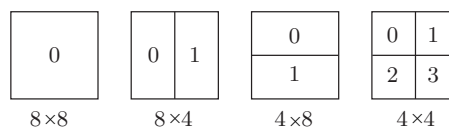


Figure 9.56: Variable-Size Transformed Blocks.

- A specially-developed integer transform is employed by VC-1. The integer nature of the transform makes it easy to implement the VC-1 decoder in 16-bit architectures. This features makes VC-1 attractive to hardware implementors, especially for use in a variety of DSP (digital signal processing) devices. The transform matrices were derived in a special way to allow for fast, easy decoding.

- Section 9.5 discusses motion compensation. The encoder computes a prediction of a video frame and uses it to predict the next frame. Naturally, the decoder should be able to mimic this operation, which is why Figure 9.55 shows that the encoder computes the prediction by performing the same operations (inverse quantization, inverse transform, deblocking, and reconstructing the frame from its constituent blocks) as the decoder. Typically, each 8×8 block or each 16×16 macroblock of pixels is motion compensated individually to generate a motion vector. The vector specifies how much the encoder predicts the image of the block (or macroblock) to be shifted in the x and y directions in the corresponding (macro)block in the reference frame.

The motion compensation component employs two sets of filters, an approximate bicubic filter with four taps and a bilinear filter with two taps. Filtering the pixels of a block sometimes allows the decoder to isolate the color of a part of a pixel (a sub-pixel) as small as $1/4$ of a pixel. The encoder uses the block size, sub-pixel data, and filter type to determine one of four motion compensation modes for each block. These modes improve compression as well as simplify the decoder's implementation.

- Blocking artifacts have already been mentioned. They are the result of discontinuities along block boundaries, caused by the differences between quantization and dequantization. These artifacts are especially noticeable (and annoying) when the codec's parameters are set for maximum compression (i.e., coarse quantization). The VC-1 encoder has an in-loop deblocking filter that tries to remove these discontinuities. This filter needs to know the block size of the transform. Also, the VC-1 encoder can overlap the transform between blocks in order to reduce annoying blocking artifacts. This technique is known as overlapped transform (OLT).

- Many video transmissions are interlaced (see Section 9.1.1 for interlacing). VC-1 includes an innovative algorithm that uses data from both sets of scan lines to improve the prediction of the encoder's motion compensation component. This algorithm is referred to as interlace coding.

- VC-1 supports five types of video frames. Three of these are the standard I frame (a key frame or intra; a frame that's encoded independently of its neighbors), P frame (predictive, a frame encoded based on its predecessor), and B frame (bidirectional, a frame encoded based on its predecessors and successors). These types are covered in more detail in Section 9.5. One of the innovations included in VC-1 is advanced B-frame encoding. This feature consists of several optimizations that make B-frame encoding more efficient.

The other two frame types are S (skipped) and BI. The former indicates a frame that is so similar to its reference frame that there is no need to encode it. When the decoder identifies an S frame, it simply repeats its reference frame. The latter type is a B frame with only I macroblocks. When there is a complete change of scene in a video stream, a B frame tends to result in a large number of I macroblocks. In such a case, the special BI frame type occupies fewer bits than a standard B frame.

It should be noted that video codecs normally collect several consecutive frames into a group of frames (GOP). In VC-1, the number of frames in a GOP is variable.

- It has been known for a while that motion compensation has its drawbacks, one of which is that it cannot efficiently model fading. Often, a succession of video frames contains regions that fade to or from black, and these are encoded very inefficiently because of the nature of motion compensation. The fading compensation feature of VC-1 detects fades and employs methods other than motion compensation to improve encoding in such cases. This feature also handles other global illumination changes.

- The coefficients produced by the transform are quantized by dividing them by a quantization parameter (QP) which itself is transmitted to the decoder in the compressed stream. Quantization is the step where information is irretrievably lost but much compression is gained.

Many compression algorithms based on quantization employ a single QP for the entire image (in case of image compression) or for an entire video frame (for video compression). VC-1 supports differential quantization (or dquant), where several quantization steps are applied to a frame. This innovation is based on the fact that different macroblocks require different grades of quantization, depending on how many nonzero AC coefficients remain after the macroblock is transformed. The simplest form of differential quantization uses two quantization levels and is termed bi-level dquant.

- The audio component of VC-1 can produce bitrates of 128–768 Kbps and is hybrid; it employs different algorithms for speech and music.

- From its inception, VC-1 was designed to perform well on a wide variety of video data. To this end, the designers included three profiles, each with several levels. Each level is geared toward a range of video applications and each determines the features a VC-1 encoder would need (its mathematical complexity) to process a video stream generated by those applications. Table 9.57 lists the profiles and their levels and Table 9.58 lists the features and innovations included in each profile.

9.11.2 The Encoder

This section is a high-level summary of the operations of the encoder. It should again be stressed that the formal definition of the VC-1 standard does not specify how each step is to be performed and leaves the details to the ingenuity of implementors. The following is a list of the main steps performed by the encoder:

- A video frame is read from the input and is decomposed into macroblocks and blocks (this is common in video encoders).
- For an intra-coded block:
 1. Transform the block with an 8×8 transform.
 2. The transform can overlap neighboring intra-coded blocks.
 3. Quantize the 64 transform coefficients. DC/AC prediction can be applied to the quantized coefficients to reduce redundancy even more.
 4. Encode the DC coefficient with a variable-length code. Scan the AC coefficients in zigzag and encode the run-lengths and nonzero coefficients with another variable-length code.

Profile	Level	Max bitrate	Representative resolutions by frame rate
Simple	Low	96 Kbps	176 × 144 @ 15 Hz (QCIF)
	Medium	384 Kbps	240 × 176 @ 30 Hz 352 × 288 @ 15 Hz (CIF)
	High	20 Mbps	1920 × 1080 @ 30 Hz (1080p)
Main	Low	2 Mbps	320 × 240 @ 24 Hz (QVGA)
	Medium	10 Mbps	720 × 480 @ 30 Hz (480p) 720 × 576 @ 25 Hz (576p)
	High	20 Mbps	1920 × 1080 @ 30 Hz (1080p)
Advanced	L0	2 Mbps	352 × 288 @ 30 Hz (CIF)
	L1	10 Mbps	720 × 480 @ 30 Hz (NTSC-SD) 720 × 576 @ 25 Hz (PAL-SD)
	L2	20 Mbps	720 × 480 @ 60 Hz (480p) 1280 × 720 @ 30 Hz (720p)
	L3	45 Mbps	1920 × 1080 @ 24 Hz (1080p) 1920 × 1080 @ 30 Hz (1080i) 1280 × 720 @ 60 Hz (720p)
	L4	135 Mbps	1920 × 1080 @ 60 Hz (1080p) 2048 × 1536 @ 24 Hz

Table 9.57: VC-1 Profiles and Levels.

Feature	Simple	Main	Adv.
Baseline intra frame compression	Yes	Yes	Yes
Variable-sized transform	Yes	Yes	Yes
16-bit transform	Yes	Yes	Yes
Overlapped transform	Yes	Yes	Yes
4 motion vector per macroblock	Yes	Yes	Yes
1/4 pixel luminance motion compensation	Yes	Yes	Yes
1/4 pixel chrominance motion compensation	No	Yes	Yes
Start codes	No	Yes	Yes
Extended motion vectors	No	Yes	Yes
Loop filter	No	Yes	Yes
Dynamic resolution change	No	Yes	Yes
Adaptive macroblock quantisation	No	Yes	Yes
B frames	No	Yes	Yes
Intensity compensation	No	Yes	Yes
Range adjustment	No	Yes	Yes
Field and frame coding modes	No	No	Yes
GOP Layer	No	No	Yes
Display metadata	No	No	Yes

Table 9.58: VC-1 Profiles and Included Features.

- For an inter-coded block:
 1. Predict the block from the corresponding motion-compensated block in the previous frame. If the frame is bidirectional, the block can also be predicted from the motion-compensated block in the next frame.
 2. Predict the motion vectors themselves from the motion vectors of neighboring blocks and encode the motion vector differences with a variable-length code.
 3. Apply one of the four sizes of the transform to the prediction error of the block.
 4. Quantize the resulting transform coefficients, scan the results in a zigzag pattern, and encode the run-lengths and nonzero coefficients with a variable-length code.

Figure 9.59 shows how an 8×8 intra block is encoded after an overlap operation. Figure 9.60 shows the main steps in the encoding of an inter block.

9.11.3 Comparison of Features

Today (early 2009) there are quite a few competing algorithms, implementations, and standards for video compression. Many experts may agree that the most important participants in this field are MPEG-2 (Section 9.6), H.264/AVC (Section 9.9), and the newcomer VC-1. This short section discusses ways to compare the main features of these standards.

What important features of a codec should be included in such a comparison? One way to compare codecs is to list the features each has or is missing. A more useful comparison should try to compare (1) the quality of the decompressed data, (2) the compression ratios (or equivalently, bitrates) achieved, and (3) the time requirements for both encoding and decoding of each implementation.

Quality comparison. The algorithms being compared include quantization as an important encoding step and are therefore lossy. The decompressed data is not identical to the original and may even contain visible distortions and compression artifacts. It is therefore important to have a well-defined process that can measure the quality of a given compression. Notice that the quality depends on the specific implementation, and not just on the algorithm. A poor implementation may result in poor compression quality regardless of the sophistication of the underlying algorithm.

The chief aim of a lossy video encoder is to lose only data that is perceptually irrelevant; data whose absence would not be noticed by a viewer. However, viewers (and people in general) rarely agree on the quality of video (or on the quality of anything else for that matter). Thus, any test for compression quality that relies on the subjective judgment of people has to be carefully planned, has to involve several judges, and should be repeated several times, with several different video streams screened under different conditions.

Because of these difficulties, some testers rely mostly on objective measures of quality, measures that yield a single number that depends on the differences between the original and decompressed video streams. One such measure is the peak signal to noise ratio (PSNR). The PSNR for two $N \times M$ images (original and reconstructed) with n -bit pixels is defined by

$$\text{PSNR}_{\text{dB}} = 10 \log_{10} \frac{(2^n - 1)N \times M}{\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} (P_{i,j} - Q_{i,j})^2}, \quad (9.4)$$

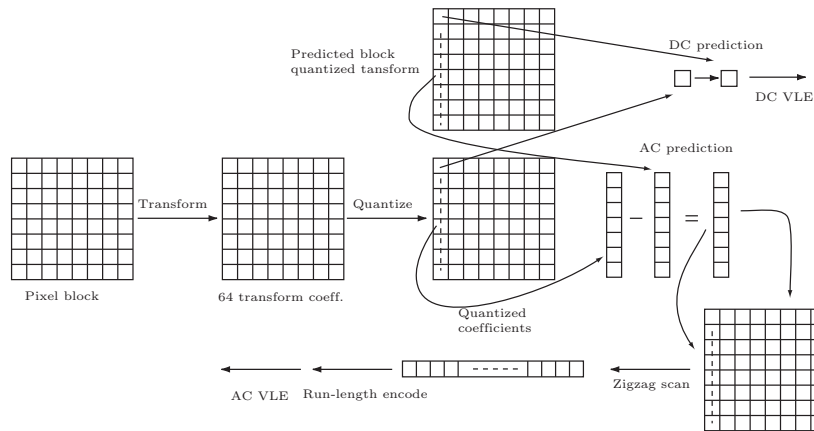


Figure 9.59: Encoding an Intra Block.

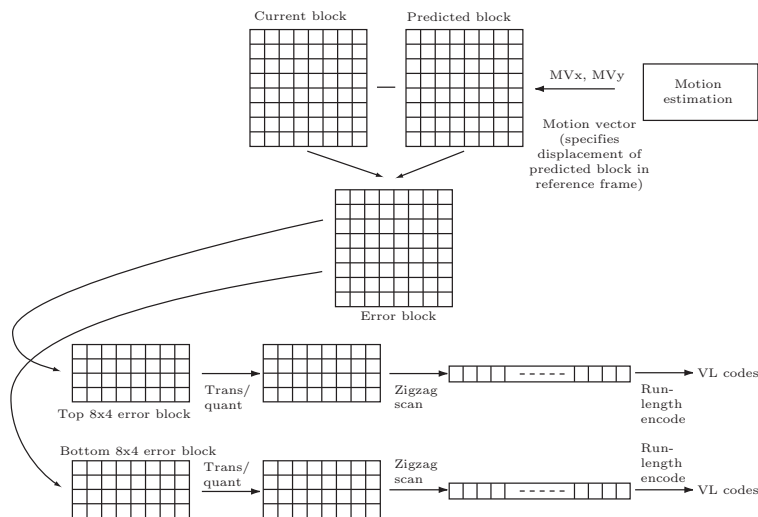


Figure 9.60: Encoding an Inter Block.

where $P_{i,j}$ and $Q_{i,j}$ are the original and reconstructed values of pixel (i, j) , respectively. This expression is slightly different from Equation (7.3), but it is also dimensionless.

The PSNR is the ratio of the maximum value of a signal (255 in the case of 8-bit pixels) to the quantization noise. Thus, the lower the PSNR, the noisier the signal. Lower compression ratios (i.e, higher bitrates) generally correspond to less noise and therefore higher PSNR. For the purposes of measuring compression quality, it therefore makes sense to plot the PSNRs for many bitrates. Such a graph (which is a rate-distortion curve) would illustrate the behavior of the codec over a wide range of bitrates

(or equivalently, a range of encoder settings).

When objective measures, such as the PSNR, alone are deemed insufficient, the testers may resort to subjective measures of quality that involve human judgment. One such test, comparing codecs for the next-generation optical disc standard, was conducted in 2002 by the DVD forum. Viewers from several interested organizations were asked to rate each video clip for (1) resolution, (2) noise, and (3) overall impression, on a scale of 1 to 5 for each measure. The aim was to select an algorithm and an implementation, so several video clips—each compressed and decompressed by MPEG-2, MPEG-4, H.264, and WMV9—were anonymously screened to the participants. The results confirmed VC-1 (in its WMV implementation) as the leader in all three categories.

Other independent subjective quality tests, performed by *DV Magazine* (located at dv.com/magazine), Tandberg television (tandberg.tv), *C'T Magazine* (German, heise.de/ct), and European Broadcasting Union (EBU, www.ebu.ch) also confirmed VC-1 as a leader.

Complexity comparison. Currently, there are many simple devices that can decompress video and send it to a screen in real time. The fastest way to do this is to implement the decoder in hardware, which is why a video codec whose decoder is simple to implement would be a winner in any complexity comparison. Here, VC-1 (and also H.264) are often losers, because they are more complex than MPEG-2.

The remainder of this section provides more details on the most important building blocks and features of VC-1.

9.11.4 The Input and Output Streams

- VC-1 expects its input stream to be in the YUV 4:2:0 format. YUV is a color space (Section 9.1.1) that is related to the familiar RGB by the simple expressions

$$\begin{aligned} Y &= 0.299R' + 0.587G' + 0.114B', \\ U &= -0.147R' - 0.289G' + 0.436B' = 0.492(B' - Y), \\ V &= 0.615R' - 0.515G' - 0.100B' = 0.877(R' - Y), \end{aligned}$$

whose inverse transform is

$$\begin{aligned} R' &= Y + 1.140V, \\ G' &= Y - 0.394U - 0.580V, \\ B' &= Y - 2.030U. \end{aligned}$$

The Y component of YUV is called luma (brightness) and the U and V components are color differences or chroma (they specify the color). The PAL television standard also uses YUV.

The three components of the RGB color space designate the intensities of the primary colors. However, lossy image and video codecs should lose information to which the eye is not sensitive, which is why such codecs prefer a color space that has luma (or luminance, Section 7.10.1) as a component. The human visual system is much more sensitive to variations in brightness than in color, which is why a video codec can lose more information (by quantization) in the chroma components, while keeping the luma

component intact or almost so. Losing information is done either by quantization or by subsampling (deleting one or two color components in a certain percentage of the pixels).

Notations such as 4:2:0 refer to subsampling pixels (or decimating the chroma components, see for example [Poynton 08]). This is illustrated in Figure 9.61. The 4:4:4 notation means no subsampling. Every pixel contains all three color components. 4:2:2 is the case where the chroma components are subsampled horizontally by a factor of 2. Similarly, a horizontal subsampling by a factor of 4 is indicated by 4:1:1. Finally, 4:2:0 is the case where pixels are subsampled in both directions by a factor of 2. This is equivalent to saying that the pixels retain only their luma component and a virtual chroma pixel is positioned between the rows and columns of the real pixels. There are actually three versions of 4:2:0 that differ in the precise positioning of the virtual chroma components.

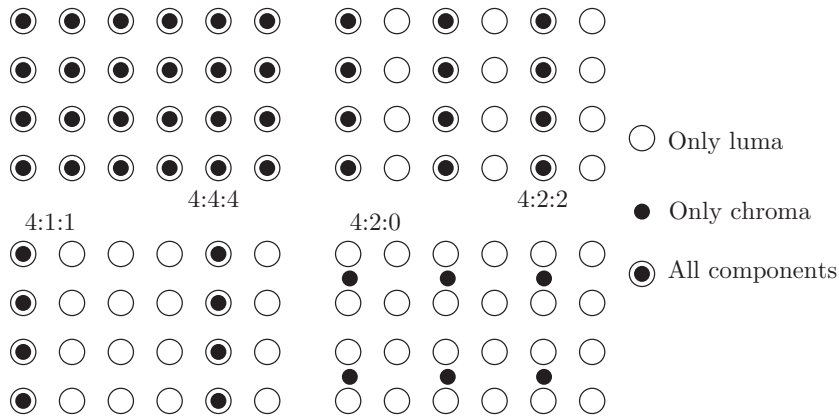


Figure 9.61: Subsampling Pixels.

The compressed stream output by VC-1 is also in YUV 4:2:0 format. It features a hierarchy of the following layers: sequence, picture, entry point, slice, macroblock, and block (the slice and entry point layers exist only in the advanced profile). A sequence is a set of pictures, a picture is a set of slices, and a slice consists of rows of macroblocks. The entry point layer is very useful. It makes it possible to enter the compressed stream at any point and also to skip parts of the video.

9.11.5 Profiles and Levels

As is the norm with important compression standards, the operations of the VC-1 encoder are not described in the official publications. This allows implementors to come up with original, sophisticated, and fast algorithms for the encoder's basic building blocks—the prediction (including both intra and inter prediction, where inter prediction means the motion estimation and compensation), the transform, and the entropy coding. A general VC-1 encoder can therefore be quite complex, which is one reason why the designers incorporated the concepts of profiles and levels in VC-1.

A profile is a precisely defined subset of a compression standard. It includes some of the tools, algorithms, parameters, and syntax of the overall standard, and it restricts the values that parameters may take. A profile may further be divided into levels which are subsubsets of the standard.

The VC-1 definition specifies three profiles, simple, main, and advanced. The simple profile has two levels and is intended for low bitrate applications, such as mobile video communication devices, security cameras, and software for video playback on personal computers. The main profile is intended for higher bitrate internet applications, such as video delivery over IP. This profile has three levels. The advanced profile is aimed for broadcast applications, such as digital television, DVDs, or HDTV. This profile supports interlaced video and contains five levels. Thus, someone trying to implement VC-1 for a mobile telephone or a portable video player may decide to include only the features of the simple profile, while engineers designing a new DVD format might consider to go all the way with a VC-1 encoder operating in the advanced profile.

Each profile has levels that place constraints on the values of the parameters used in the profile. In this way, a level limits the video resolution, frame rate, bitrate, buffer sizes, and the range of motion vectors.

9.11.6 Motion Compensation

As with other video encoders, the motion compensation step is the most elaborate and time consuming part of the encoder. The principle of motion compensation is to locate, for each block B in the current frame F, the most similar block S among the reference frames of F. Once found, S is subtracted from B and the difference, which is small, is transformed, quantized, and entropy encoded.

Motion compensation is beneficial because a video frame tends to be similar to its immediate predecessors and successors. Often, the only difference between a frame and the preceding frame is the location of certain objects that have moved slightly between the frames. Thus, the chance of finding blocks similar to the current block increases when the blocks are small. Similarly, a large number of reference frames also increases the chance of finding good matches. As usual, however, there is a tradeoff. Smaller blocks (and therefore a larger number of blocks) and many reference frames imply more work and increased complexity for the encoder. VC-1 allows for only two reference frames and a maximum of two block sizes.

When a frame of type inter is encoded, its macroblocks are motion compensated with either 16×16 blocks and one motion vector (MV) for each block, or with four 8×8 blocks, each with its own MV. Notice that the MVDs (MV differences) have to be transmitted to the decoder. An MV specifies the xy coordinates of the matched block S relative to the current block B. Normally, the coordinates are measured in pixels, but VC-1 allows for coordinates whose units are $1/2$ pixels or even $1/4$ pixels (and this choice has also to be transmitted to the decoder).

Intensity Compensation is one of the many innovations and complexities introduced in VC-1. This is a special mode that the encoder enters when it decides that the effectiveness of motion compensation can be improved by scaling the original color. In this mode, the encoder scales the luma and chroma color components of the reference frame before using them in motion compensation.

9.11.7 Pixel Interpolation

The principles of motion compensation are discussed in Section 9.5. The idea is to predict a block from the corresponding blocks in previous (and sometimes also future) frames. Modern video codecs extend this idea by artificially increasing the resolution of each block before it is predicted. This is done by interpolating the values of pixels. In its simplest form, interpolation starts with a group of 2×2 pixels, computes their average, and considers this average a new, fractional, pixel at the center of the group. In VC-1, pixels can be interpolated to half-pixel or even quarter-pixel, as illustrated by Figure 9.62a,b. The black squares in the figure are pixels, the white squares are half pixels, and the circles are quarter pixels.

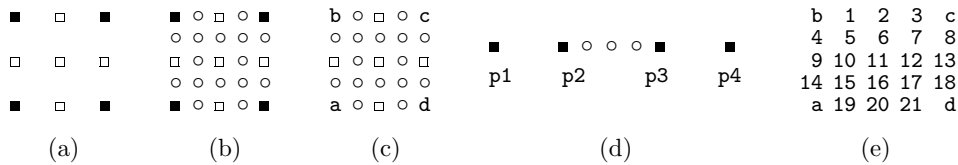


Figure 9.62: Fractional Pixels.

VC-1 uses bilinear and bicubic pixel interpolation methods. Instead of listing the complex rules and interpolation formulas employed by VC-1, this short section tries to explain how the formulas and the many constants they include were derived.

We start with bilinear interpolation (see [Salomon 06] for more information and figures). Figure 9.62c shows a group of 2×2 pixels labeled **a** through **d**, with five half pixels and 16 quarter pixels to be linearly interpolated. We imagine that the four original pixels are equally-spaced three-dimensional points (where the values of the pixels are the z coordinates of the points) and we use linear operations to construct a rectangular surface patch whose four corners are these points. We connect points **a** and **d** with the straight line $L_1(u) = \mathbf{a}(1 - u) + \mathbf{d}u$ and then connect points **b** and **c** with the straight line $L_2(u) = \mathbf{b}(1 - u) + \mathbf{c}u$. Notice that $L_1(0) = \mathbf{a}$, $L_1(1) = \mathbf{d}$, and similarly for L_2 . The entire bilinear surface $P(u, w)$ is obtained by selecting a value for parameter w , selecting the two points $P_1 = L_1(w)$ and $P_2 = L_2(w)$, and connecting P_1 and P_2 with a straight line $P(u, w) = P_1(1 - w) + P_2w$. The entire surface (Figure 9.63 is an example) is obtained when u and w vary independently in the interval $[0, 1]$.

We now consider the pixel grid of Figure 9.62c a two-dimensional coordinate system with point **a** as its origin, parameter u as the x axis, and parameter w as the y axis. With this notation, it is easy to see that the half pixels (from top to bottom and left to right) are the five points $P(.5, 1)$, $P(0, .5)$, $P(.5, .5)$, $P(1, .5)$, and $P(.5, 0)$. The two quarter pixels on the top row are the surface points $P(.25, 1)$ and $P(.75, 1)$, and the remaining 14 quarter pixels have similar coordinates.

Figure 9.64 lists Mathematica code for bilinear interpolation, as well as a few results for half- and quarter pixels. It is easy to see how some fractional pixels depend on only two of the four original pixels, while others depend on all four. Notice that the formal definition of the VC-1 standard employs integers (followed by a right shift) instead of the real numbers listed in our figure. For point $P(0.5, 0.25)$, for example, a VC-1

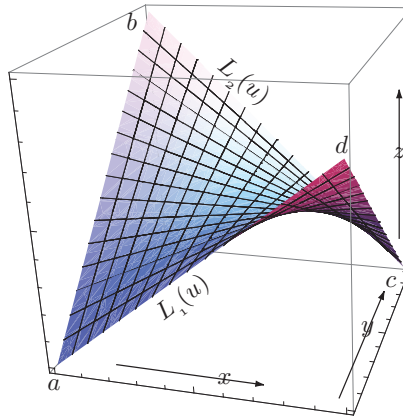


Figure 9.63: A Bilinear Surface.

implementation computes the integer expression $(6a + 2b + 2c + 6d + 2) \gg 4$ instead of the equivalent real expression $0.375a + 0.125b + 0.125c + 0.375d$.

```

bil[u_,w_]:=a(1-u)(1-w)+b(1-u)w+d(1-w)u+c u w;
bil[.25, .25]
bil[.75, .5]
bil[.25, 1]
bil[.5, .25]

0.5625a+0.1875b+0.0625c+0.1875d
0.125a+0.125b+0.375c+0.375d
0.75b+0.25c
0.375a+0.125b+0.125c+0.375d

```

Figure 9.64: Code For Bilinear Interpolation.

The principles of bicubic interpolation (we are interested in the one-dimensional case) are discussed in Section 7.25.5. The idea is to start with the values of four consecutive pixels, consider them the y coordinates of four equally-spaced two-dimensional points $p1$ through $p4$, and construct a polynomial $P(t)$ that passes through the points when t varies from 0 to 1. This polynomial is given by Equation (7.45), duplicated here

$$\mathbf{G}(t) = (t^3, t^2, t, 1) \begin{pmatrix} -4.5 & 13.5 & -13.5 & 4.5 \\ 9.0 & -22.5 & 18 & -4.5 \\ -5.5 & 9.0 & -4.5 & 1.0 \\ 1.0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} p1 \\ p2 \\ p3 \\ p4 \end{pmatrix}. \quad (7.45)$$

The four original points are obtained for t values 0, $1/3$, $2/3$, and 1 and Figure 9.62d makes it clear that we are interested in the three pixels that are equally spaced between

$p2$ and $p3$. These correspond to t values 0.44444, 0.5, and 0.55555, respectively. The Mathematica code of Figure 9.65 shows how this computation is carried out. In order to speed up the decoder, the definition of VC-1 employs integers and a right shift instead of the real numbers listed in the figure.

```
g[t_]:= {t^3, t^2, t, 1}.
{{{-4.5,13.5,-13.5,4.5},{9.,-22.5,18.,-4.5},
{-5.5,9.,-4.5,1.},{1.,0,0,0}}.{p1,p2,p3,p4};
g[0.44444]
g[0.5]
g[0.55555]

-0.0617277p1+0.740754p2+0.370355p3-0.0493812 p4
-0.0625p1+0.5625p2+0.5625p3-0.0625p4
-0.0493846p1+0.37039p2+0.740724p3-0.0617293p4
```

Figure 9.65: Code For Bicubic Interpolation.

Section 7.25.7 shows how cubic interpolation can be extended to two dimensions, but VC-1 specifies this process differently. The VC-1 encoder computes the bicubic interpolation of half- and quarter pixels in several steps. First, this type of interpolation is performed horizontally to compute the three pixels 1, 2, 3 between **b** and **c** (Figure 9.62e), using, in addition to **b** and **c**, the values of the pixel to the left of **b** (as our $p1$) and the pixel to the right of **c** (as our $p4$). Next, the three pixels 19, 20, and 21 between **a** and **d** are computed in a similar way. Using the values of 1 (and the corresponding pixel in the 5×5 grid above) and 19 (and the corresponding pixel in the grid below), it is possible to interpolate for pixels 5, 10, and 15. The values of 6, 11, and 16, and 7, 12, and 17 are similarly computed. Finally, the values of 4, 9, and 14 are computed from pixels **b** (and the one above it) and **a** (and the one below it), and similarly for 8, 13, and 18. Another tough job for the VC-1 implementor!

9.11.8 Deblocking Filters

The VC-1 encoder transforms blocks of pixels and it has long been known that this introduces boundary artifacts into the decompressed frame. Imagine two adjacent identical pixels with a value of, say, 100. Assume further that a block boundary passes between the pixels. After being transformed and quantized by the encoder and later reverse-transformed and dequantized by the decoder, one pixel may end up as 102, while the other may become 98. The difference between an original and a reconstructed pixel is 2, but the difference between the two reconstructed pixels is double that. Blocking artifact can also be caused by a poor quantization of the DC coefficient, causing the intensity of a decoded block to become lower or higher than the intensities of its neighbors.

VC-1 introduces two innovative techniques to reduce the effect of blocking artifacts. They are referred to as overlapped transform (OLT) and in-loop deblocking filter (ILF).

OLT acts as a smoothing filter. It is designed to perform best when two adjacent 8×8 blocks differ much in their quality. Imagine a block **P** on the left and a block **Q** on the right with a vertical boundary between them (alternatively, **P** may be directly above

Q, with a horizontal boundary between them). If P and Q differ in their qualities (one is high quality and the other is low quality), then the encoder applies the OLT filter to smooth their boundary. This operation is carried out only on intra-coded blocks, which are always 8×8 . The formal definition places many restrictions on the OLT filter, so the encoder has to signal to the decoder the presence of this operation.

Once the encoder decides to apply OLT to two adjacent blocks (the codec definition, as usual, does not specify how the encoder should make this decision), it starts with the four edge pixels P1, P0, Q0, and Q1 shown in Figure 9.66 (for two blocks with a horizontal boundary, the figure should be rotated 90°). For each of the eight rows of the blocks, the encoder computes new values for these pixels, based on their original values and on certain rounding parameters, and substitutes the new values for the old ones. The decoder performs the reverse operations. Specifically, the computation carried out by the decoder is

$$\begin{bmatrix} P1' \\ P0' \\ Q0' \\ Q1' \end{bmatrix} = \left\{ \begin{bmatrix} 7 & 0 & 0 & 1 \\ -1 & 7 & 1 & 1 \\ 1 & 1 & 7 & -1 \\ 1 & 0 & 0 & 7 \end{bmatrix} \begin{bmatrix} P1 \\ P0 \\ Q0 \\ Q1 \end{bmatrix} + \begin{bmatrix} r0 \\ r1 \\ r0 \\ r1 \end{bmatrix} \right\} \gg 3,$$

where the notation $\gg$ means a right shift and the two rounding parameters $r0$ and $r1$ take the values (3, 4) for even-numbered rows of pixels and (4, 3) for odd-numbered rows.

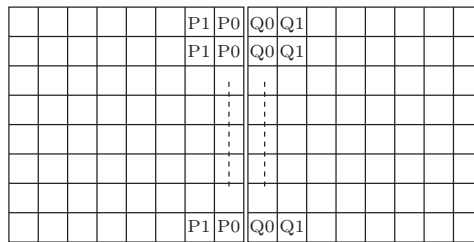


Figure 9.66: Edge Pixels For OLT.

When two adjacent blocks have similar qualities, the VC-1 standard recommends the use of ILF. Notice that an in-loop filter improves prediction since it filters frames that will be used for reference (this is different than a post-filter). For blocks in I and B frames, the ILF filter should be applied to all four boundaries of the 8×8 blocks in the following way. First, ILF should be applied to every 8th and 9th horizontal rows of the frame, and then to every 8th and 9th vertical columns. If the rows are numbered 0 through $N - 1$, then ILF should be applied to rows (7, 8), (15, 16), ..., $((8(N - 1) - 1, 8(N - 1))$, and similarly for the columns (Figure 9.67). Those rows and columns form the block boundaries in the frame.

For blocks in P frames, the situation is more complex, because blocks can be of four different sizes, as illustrated by Figure 9.69a. The encoder therefore has to work harder to identify the pixels that constitute block boundaries, as shown in part (b) of

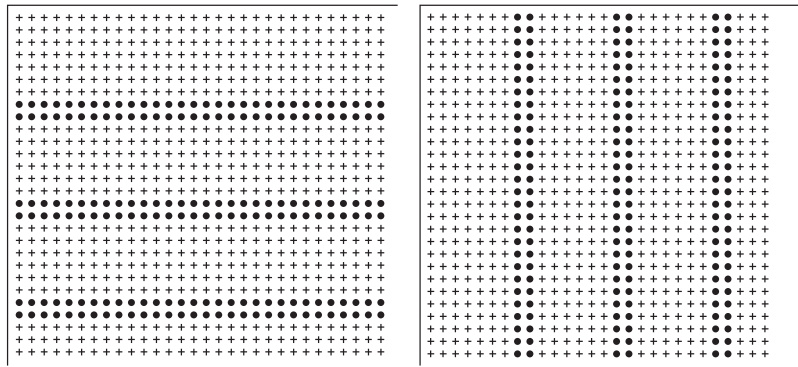


Figure 9.67: ILF Boundary Rows and Columns For I and B Frames.

the figure. The figure shows examples where filtering between neighboring blocks (or subblocks) does or does not occur. If the motion vectors of adjacent blocks are different, then the boundary between them is always filtered. Thus, our example assumes that the motion vectors of the blocks are identical. The shaded blocks or subblocks in the figure are those that have at least one nonzero coefficient, while the clear blocks or subblocks have no transform coefficients. The thick lines designate boundaries that should be filtered. The order of filtering is important and is specified in detail in the VC-1 definition. The figure illustrates only horizontal block neighbors. The handling of vertical block neighbors is similar.

Once the encoder identifies the block boundaries that should be filtered, the filtering operation can proceed. Figure 9.68 illustrates this process. Filtering is always done on a segment of 2×4 (horizontal) or 4×2 (vertical) pixels. Thus, if a boundary is eight pixels long, its filtering becomes a two-step process. In each four-pixel segment, the third pair of pixels is filtered first and the result of this operation determines whether the other three pairs should also be filtered.

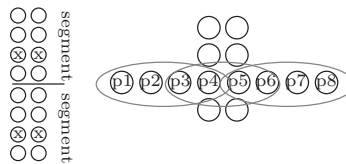


Figure 9.68: ILF Filtering.

The rest of the filtering operation differs slightly for P frames and for the other frame types. It is described here in essence but not in detail. The third pair and its six near neighbors are split into three groups as shown in the figure and several expressions involving differences of pairs of pixels in each group are computed. The idea is to discover any blocking effects caused by differences between pixels on both sides of the boundary. These differences are examined to see whether the remaining three pairs should also

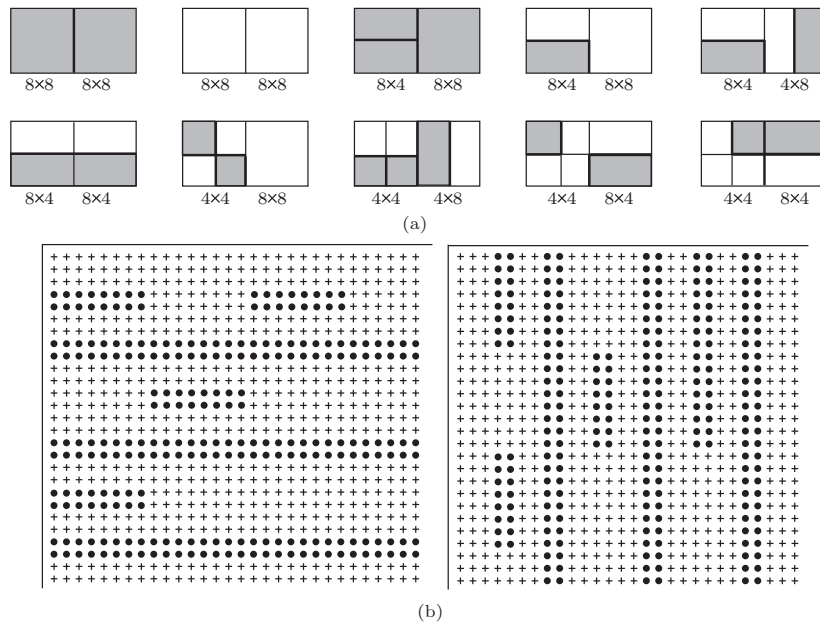


Figure 9.69: ILF Boundary Rows and Columns For P Frames.

be filtered. The two pixels p_4 and p_5 that are closest to the boundary are updated according to

$$d = (p_4 - p_5)/2, \quad p_4 = p_4 - d, \quad p_5 = p_5 + d.$$

If the decision is made to filter the other three pairs, they are updated in a similar way. In each pair, pixel differences are computed and the two pixels closest to the boundary are updated.

9.11.9 The Transform

The transform is a central component of VC-1. It is an integer transform that is based on the discrete cosine transform (DCT). Clearly, an integer transform is easier to implement and is faster to execute. Its precision, however, depends on the size of the integers in question, but each color component of a pixel is typically only eight bits long, so a 16-bit transform provides enough precision to generate intermediate results that are much bigger than an original pixel.

It has already been mentioned that the transform operates on four different block sizes, as shown in Figure 9.56. This is one of the many innovations of VC-1 and it is an option of the encoder (that is signaled at the sequence layer). Blocks are first motion compensated and then transformed. The intra blocks (blocks in macroblocks in I frames) always use the 8×8 transform, while the inter blocks (in the other types of frames) may use any of the four transform sizes (and this information is stored in the picture, macroblock, or block layers). When the transform size appears in the picture

layer, all the blocks in the picture use the same transform size. When the size appear at the macroblock layer, all the blocks of the macroblock use the same transform size.

The details of this transform are described here, based on [Srinivasan and Regunathan 05], but the interested reader should first become familiar with the concept of orthogonal transforms (Section 7.7) and especially with the matrix of Equation (7.6) and with the DCT matrix in one dimension for $n = 8$, Table 7.38, shown here as matrix (9.5) (unnormalized) and (9.6) (normalized). Notice that the odd-numbered rows of this matrix have only three distinct elements and the even-numbered rows consist of another set of four distinct elements.

$$\begin{bmatrix} 1. & 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 0.981 & 0.831 & 0.556 & 0.195 & -0.195 & -0.556 & -0.831 & -0.981 \\ 0.924 & 0.383 & -0.383 & -0.924 & -0.924 & -0.383 & 0.383 & 0.924 \\ 0.831 & -0.195 & -0.981 & -0.556 & 0.556 & 0.981 & 0.195 & -0.831 \\ 0.707 & -0.707 & -0.707 & 0.707 & 0.707 & -0.707 & -0.707 & 0.707 \\ 0.556 & -0.981 & 0.195 & 0.831 & -0.831 & -0.195 & 0.981 & -0.556 \\ 0.383 & -0.924 & 0.924 & -0.383 & -0.383 & 0.924 & -0.924 & 0.383 \\ 0.195 & -0.556 & 0.831 & -0.981 & 0.981 & -0.831 & 0.556 & -0.195 \end{bmatrix} \cdot \quad (9.5)$$

$$\begin{bmatrix} 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 \\ 0.4904 & 0.4157 & 0.2778 & 0.0975 & -0.0975 & -0.2778 & -0.4157 & -0.4904 \\ 0.4619 & 0.1913 & -0.1913 & -0.4619 & -0.4619 & -0.1913 & 0.1913 & 0.4619 \\ 0.4157 & -0.0975 & -0.4904 & -0.2778 & 0.2778 & 0.4904 & 0.0975 & -0.4157 \\ 0.3536 & -0.3536 & -0.3536 & 0.3536 & 0.3536 & -0.3536 & -0.3536 & 0.3536 \\ 0.2778 & -0.4904 & 0.0975 & 0.4157 & -0.4157 & -0.0975 & 0.4904 & -0.2778 \\ 0.1913 & -0.4619 & 0.4619 & -0.1913 & -0.1913 & 0.4619 & -0.4619 & 0.1913 \\ 0.0975 & -0.2778 & 0.4157 & -0.4904 & 0.4904 & -0.4157 & 0.2778 & -0.0975 \end{bmatrix} \cdot \quad (9.6)$$

The first thing that comes to mind, when trying to derive an integer version of the DCT, is to round off the elements of Equation (9.5) to the nearest integer, but it is easy to see that this would simply result in a matrix of zeros and 1's, similar to the basic orthogonal transform matrix of Equation (7.6). It would still be an orthogonal transform, but without the useful energy compaction properties of the DCT (however, note that energy compaction is not a critical issue here, because the DCT operates on the residuals of the prediction). Thus, a different approach is needed. The designers of the VC-1 transform started with the following considerations:

- Video encoding can sometimes be slow, but decoding must be done at real time. Video decoding is often done by small, specialized processors that are embedded in DVD players, digital televisions, home theaters, handheld mp3 and video devices, and other devices and instruments designed for video playback. Currently, most such devices are based on 16-bit architectures.
- It is known from long experience with video compression that the inverse transform is generally responsible for 10–30% of the CPU time used by a video decoder. A single video frame of 1920×1080 pixel (the so-called HDTV 1080p) consists of $32,400 \times 8$ blocks that have to be inverse transformed. At 30 frames per second, the decoder has to inverse-transform 972,000 blocks each second; a tough job for even fast processors.

These two considerations imply that a 16-bit integer transform can be very useful if it also offers energy compaction equivalent to that of the DCT. The integer transform developed for VC-1 allows for fast decoding. The decoder performs the inverse transform using just 16-bit integer arithmetic. The encoder, on the other hand, can compute the forward transform with fixed-point (scaled integer) or even floating-point arithmetic. (The encoder can also benefit from a simple transform, because it has to try all the valid block sizes, such as 8×8 and 4×8 , in order to choose the best.)

A full understanding of this transform starts with the concept of *norm*. In mathematics, the term “norm” is defined for certain mathematical objects that are more complex than a single number. Intuitively, the norm of an object is a quantity that describes the length, magnitude, or extent of the object. Norms are defined for complex numbers, quaternions, vectors, matrices, and other objects that are not just numbers. Often, several different norms are defined for the same type of object.

The norm of a matrix can be defined in several ways, but perhaps the most common definition is the Frobenius norm (older texts refer to it as the Hilbert-Schmidt norm or the Schur norm). Given an $n \times m$ matrix A with elements a_{ij} , its Frobenius norm is

$$\|A\| = \left[\sum_{i=1}^n \sum_{j=1}^m |a_{ij}|^2 \right]^{1/2}.$$

Similarly, the norm of a vector is the square root of the sum of the squares of its elements. These norms are very similar to the concept of energy, defined in Section 7.6.

Given a vector $V = (2, 2, 2)$ and a 3×3 matrix A all whose elements are 10, the result of the product $V \cdot A$ is the vector $W = (60, 60, 60)$, and it is easy to verify that $\|W\| = \|V\| \cdot \|A\| = 60\sqrt{3}$. We therefore conclude that multiplying a vector by a matrix A scales its norm by (approximately) a factor of $\|A\|$.

The two-dimensional DCT (Equation (7.16), duplicated here) can be written in matrix notation as the product of a row of m functions $\cos[(2y+1)j\pi/2m]$, an $n \times m$ matrix of pixels p_{xy} , and a column of n functions $\cos[(2x+1)i\pi/2n]$. Thus

$$\begin{aligned} G_{ij} &= \sqrt{\frac{2}{m}} \sqrt{\frac{2}{n}} C_i C_j \sum_{x=0}^{n-1} \sum_{y=0}^{m-1} p_{xy} \cos \left[\frac{(2y+1)j\pi}{2m} \right] \cos \left[\frac{(2x+1)i\pi}{2n} \right] \\ &= \left[\text{row of } \cos \left[\frac{(2y+1)j\pi}{2m} \right] \right] \begin{bmatrix} p_{00} \\ \vdots \\ p_{nm} \end{bmatrix} \begin{bmatrix} \text{column} \\ \text{of} \\ \cos \left[\frac{(2x+1)i\pi}{2n} \right] \end{bmatrix}, \end{aligned} \quad (9.7)$$

Each coefficient G_{ij} is therefore the product of a row of cosines, a block of pixels, and a column of cosines. An entire block of transform coefficients G_{ij} is therefore the product of the one-dimensional DCT matrix (Equation (7.14)), the corresponding block of pixels, and the transpose of the same DCT matrix. Our task is to replace the DCT matrix with a matrix T of integers, and the earlier discussion of norms makes it clear that the resulting transform coefficients G_{ij} will be bigger than the original pixels by a factor of approximately $\|T\|^2$.

We denote $L \stackrel{\text{def}}{=} \log_2 \|T\|$. We assume that the prediction residuals are 9-bit signed integers, i.e., in the interval $[-256, 255]$. We require that the transform coefficients be 12-bit signed integers. Transforming a block of pixels therefore increases the magnitude of a pixel by three bits. Because transforming is done by two matrix multiplications (by the transform matrix T and its transpose), we conclude that each matrix multiplication increases the magnitude of pixels by 1.5 bits on average.

To a large extent, decoding is the reverse of encoding, but the integer nature of the transform creates a problem. Decoding starts with 12-bit signed transform coefficients and has to end up with 9-bit signed integer pixels, but decoding is done by two matrix multiplications which *increase* the size of the numbers involve. Thus, each matrix multiplication should be followed by a right shift (denoted by $\gg$) which shrinks the results from 12 bits to 10.5 bits and then to nine bits. Assuming that decoding starts with a matrix A of 12-bit transform coefficients, the first step is to multiply $B = T \cdot A$ and shift $C = B \gg n$, where the value of n should be such that the elements of C will be 10.5 bits long on average. The second step is another multiplication $D = C \cdot T'$ followed by another right shift $E = D \gg m$, where m should be chosen such that the elements of E are 9-bit integers.

This sequence of transformations and magnitude changes is achieved if we require that $\|T\|^2 = 2^{n+m}$, which implies $2L = n + m$. Also, the dynamic range (the range of values) of matrix B should be $10.5 + L$ and the range of D should be $9 + 2L - n$. These conditions should be achieved within the constraints of 16-bit arithmetic, which results in the following relations

$$2L = n + m, \quad 10.5 + L \leq 16, \quad 9 + 2L - n \leq 16.$$

One solution of these relations is $L \leq 5.5$ (which implies $\|T\|^2 = 2^{11} = 2,048$) and $m = 6$.

These values are enough to place a limit on the DC coefficient of the transform. Equation (9.7) implies that the DC coefficient G_{00} is obtained by multiplying the pixels by the top row of the transform matrix, a row whose elements are identical. If we denote the (yet unknown) element on the top row of our 8×8 integer transform matrix T by d_8 , then the norm of that row is $\sqrt{8d_8^2}$. This norm must be less than or equal the norm of T , so it must satisfy $8d_8^2 \leq 2,048$ or $d_8 \leq 16$. For the 4×4 transform matrix, where each row is half the size of the 8×8 matrix, the element of the top row is denoted by d_4 and the norm requirement is $d_4 = \sqrt{2}d_8$. It is now easy to check every pair of integers (d_4, d_8) for the constraints $d_8 \leq 16$ and $d_4 = \sqrt{2}d_8$, and a quick search verifies that the only pairs that come within 5% of the constraints are (7, 5), (10, 7), (17, 12) and (20, 14).

The next task is to determine the remaining seven rows of the 8×8 integer transform matrix. We first examine the even-numbered rows and it is easy to see that they contain only four distinct elements that we denote by $C1$ through $C4$. These rows form the 4×8 matrix

$$\begin{bmatrix} C1 & C2 & C3 & C4 & -C4 & -C3 & -C2 & -C1 \\ C2 & -C4 & -C1 & -C3 & C3 & C1 & C4 & -C2 \\ C3 & -C1 & C4 & C2 & -C2 & -C4 & C1 & -C3 \\ C4 & -C3 & C2 & -C1 & C1 & -C2 & C3 & -C4 \end{bmatrix}. \quad (9.8)$$

The following conditions were set by experts in order to determine the four elements

(the curious reader might want to glance at Section 8.7.1, for a list of similar conditions used to determine the values of wavelet filter coefficients):

1. The rows of T must be orthogonal. We know that each even-numbered row is orthogonal to the odd-numbered rows, because of the properties of the cosine function. Thus, we have to choose the four elements such that the four even-numbered rows will be orthogonal.

2. The vector $(C1, C2, C3, C4)$ should correlate well with the special DCT vector $(0.4904, 0.4157, 0.2778, 0.0975)$. The term “correlate” refers to the cosine of the angle between the vectors, as measured by their dot product. The DCT vector above was obtained in the following way. The DCT matrix of Equation (9.6) is normalized. Normalization is done by dividing each row by its length, so it becomes a vector of length 1. When this is done to the second row, it becomes the unit vector $(0.4904, 0.4157, 0.2778, 0.0975, -0.0975, -0.2778, -0.4157, -0.4904)$ and the DCT vector above is the first half of this (its length is therefore 0.5).

3. The norm $C1^2 + C2^2 + C3^2 + C4^2$ of the four elements should match that of d_8 , i.e., it should be as close as possible to $d_8^2 + d_8^2 + d_8^2 + d_8^2 = 4d_8^2$. The four choices of d_8 listed earlier yield the four values 100, 196, 576, and 784 for $4d_8^2$.

How can we derive the set of four integers that come closest to satisfying the three conditions above? The simplest approach is to use brute force (i.e., the high speed of our computers). We loop over all the possible sets of four positive integers and select the sets that satisfy $C1^2 + C2^2 + C3^2 + C4^2 \leq 800$. (There is a very large number of such sets, but it is finite.) For each set, we check to see if the four rows of matrix (9.8) are orthogonal. If so, we compute the dot product of condition 2 above and save it. After all possible sets of four integers have been examined in this way, we select the set whose dot product is the maximal. This turns out to be the set $(16, 15, 9, 4)$, whose norm is 578 (implying that $d_4 = 17$ and $d_8 = 12$), very close to the ideal 576. The correlation of condition 2 is 0.9984, also very encouraging.

We next examine the odd-numbered rows of Equation (9.6). We already know that $d_8 = 12$, so these four rows become the 4×8 matrix

$$\begin{bmatrix} 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \\ C6 & C6 & -C6 & -C5 & -C5 & -C6 & C6 & C5 \\ 12 & -12 & -12 & 12 & 12 & -12 & -12 & 12 \\ C6 & -C5 & C5 & -C6 & -C6 & C5 & -C5 & C6 \end{bmatrix}$$

and require the derivation of only two elements, denoted by $C5$ and $C6$. The norm $C5^2 + C6^2$ of these elements should match (similar to condition 3 above) that of d_4 , i.e., it should be as close as possible to $d_8^2 + d_8^2 = 288$, and this is satisfied, to within 1%, by the pair $(16, 6)$.

The 8×8 transform matrix T_8 has now been fully determined and is given by

$$T_8 = \begin{bmatrix} 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \\ 16 & 15 & 9 & 4 & -4 & -9 & -15 & -16 \\ 16 & 6 & -6 & -16 & -16 & -6 & 6 & 16 \\ 15 & -4 & -16 & -9 & 9 & 16 & 4 & -15 \\ 12 & -12 & -12 & 12 & 12 & -12 & -12 & 12 \\ 9 & -16 & 4 & 15 & -15 & -4 & 16 & -9 \\ 6 & -16 & 16 & -6 & -6 & 16 & -16 & 6 \\ 4 & -9 & 15 & -16 & 16 & -15 & 9 & -4 \end{bmatrix}.$$

The 4×4 integer transform is derived in a similar way, but is much simpler. We already know that the transform matrix should look like the 4×4 matrix of Equation (7.6) and that d_4 equals 17. Thus, we are looking for a matrix of the form

$$\begin{bmatrix} 17 & 17 & 17 & 17 \\ D1 & D2 & -D2 & -D1 \\ 17 & -17 & -17 & 17 \\ D2 & -D1 & D1 & -D2 \end{bmatrix}.$$

This matrix is inherently orthogonal for any choice of elements $D1$ and $D2$, but the best choices are those that satisfy the norm condition $D1^2 + D2^2 = d_4^2 + d_4^2 = 578$, and a direct search yields the pair $D1 = 22$ and $D2 = 10$, that satisfy $D1^2 + D2^2 = 22^2 + 10^2 = 584$. Thus, the complete 4×4 integer transform used by VC-1 is

$$T_4 = \begin{bmatrix} 17 & 17 & 17 & 17 \\ 22 & 10 & -10 & -22 \\ 17 & -17 & -17 & 17 \\ 10 & -22 & 22 & -10 \end{bmatrix}.$$

Once the two integer transform matrices have been constructed, we are ready for the transform itself. As has already been mentioned, the forward transform is computed by the encoder, which doesn't always have to work at real time. Thus, the forward transform can be performed with fixed-point (scaled integer) or even floating-point arithmetic. We start with two normalization vectors

$$c_4 = \left(\frac{8}{289}, \frac{8}{292}, \frac{8}{289}, \frac{8}{292} \right), \quad c_8 = \left(\frac{8}{288}, \frac{8}{289}, \frac{8}{292}, \frac{8}{289}, \frac{8}{288}, \frac{8}{289}, \frac{8}{292}, \frac{8}{289} \right).$$

These are used to construct the four normalization matrices $N_{4 \times 4}$, $N_{8 \times 4}$, $N_{4 \times 8}$, and $N_{8 \times 8}$ by means of outer products (also known as tensor products) of the vectors c_4 and c_8 . Thus, for example, $N_{8 \times 4} = c_8 \otimes c_4$.

The outer product is a vector operation whereby a matrix is constructed from the elements of two vectors. Given the two vectors $a = (a_1, a_2, \dots, a_m)$ and $b = (b_1, b_2, \dots, b_n)$,

their outer product $a \otimes b$ is the $m \times n$ matrix

$$\begin{bmatrix} a_1b_1 & a_1b_2 & \dots & a_1b_n \\ a_2b_1 & a_2b_2 & \dots & a_2b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_mb_1 & a_mb_2 & \dots & a_mb_n \end{bmatrix}.$$

In contrast, the more familiar inner product (or dot product) operates on the elements of two equal-length vectors to produce a single number.

The forward transform starts with a block D of pixels and produces a block $\hat{D}$ of transform coefficients. The transform is defined by $\hat{D} = (T_i D T_j') \circ N_{i \times j}$ where i and j take the values 4 or 8, and the notation “ $\circ$ ” means an entry-wise multiplication of two matrices. Thus, depending on the transform block size, there are the following four cases:

$$\hat{D} = (T_4 D T_4') \circ N_{4 \times 4}, \quad \hat{D} = (T_8 D T_4') \circ N_{8 \times 4}, \quad \hat{D} = (T_4 D T_8') \circ N_{4 \times 8}, \quad \hat{D} = (T_8 D T_8') \circ N_{8 \times 8}.$$

The inverse transform starts with a block of 12-bit signed (i.e., in the interval $[-2,048, +2,047]$), quantized transform coefficients $D_{i \times j}$ (where i and j are 4 or 8) and results in a block $R_{i \times j}$ of 10-bit signed integers (i.e., in the interval $[-512, 511]$) that become the reconstructed pixels. The transform consists of the two steps

$$E_{i \times j} = (D_{i \times j} \cdot T_j + \mathbf{4}) \gg 3, \quad R_{i \times j} = (T_i' \cdot E_{i \times j} + C_i \otimes \mathbf{1}_j + \mathbf{64}) \gg 7,$$

where $E_{i \times j}$ are intermediate matrices whose entries are 13-bit signed integers (i.e., in the interval $[-4,096, +4,095]$), $\mathbf{1}_i$ is a row of i 1's, C_j are the columns $C_4 = (0, 0, 0, 0)'$ and $C_8 = (0, 0, 0, 0, 1, 1, 1, 1)'$, and the notation “ $\gg$ ” means an arithmetic (sign-preserving) right shift, performed on all the entries of a matrix. The boldface **4** and **64** refer to adding these constants to all the elements of a matrix.

9.11.10 Quantization and Scan

VC-1 uses both uniform and non-uniform quantization schemes. The uniform quantization method treats the DC coefficients of intra blocks differently from the other transform coefficients. These DC coefficients are quantized by dividing them by a parameter DCStepSize, that is computed as follows

$$\text{DCStepSize} = \begin{cases} \text{MQUANT} = 1, 2, & 2 \cdot \text{MQUANT}, \\ \text{MQUANT} = 3, 4, & 8, \\ \text{MQUANT} \geq 5, & 6 + \text{MQUANT}/2, \end{cases}$$

where MQUANT is a parameter derived in a complex way from the syntax elements DQBILEVEL, MQDIFF, PQQUANT (the latter parameter is the picture quantizer scale parameter, which itself is derived in a complex way from the Picture Quantizer Index, PQINDEX), ALTPQUANT and ABSMQ. All other transform coefficients are quantized by dividing them by $2 \cdot \text{MQUANT}$ (except in some cases, not listed here, where the quantization parameter is $2 \cdot \text{MQUANT} + \text{HALFQP}$).

The non-uniform quantization method also treats the DC coefficients of intra blocks differently from the other transform coefficients. This coefficient is quantized as in the uniform method, while all other transform coefficients are quantized by dividing them by first subtracting MQUANT and then dividing by $2 \cdot \text{MQUANT}$ (except in some cases, not listed here, where the transform parameter is divided by $2 \cdot \text{MQUANT} + \text{HALFQP}$).

The decoder performs the reverse operations, then rounds the results to the nearest integer and, if necessary, also truncates them to 12 bits, signed, so they lie in the interval $[-2,048, +2,047]$.

Once the transform coefficients have been quantized, they are collected in a zigzag scan of the block. As most other features of VC-1, the zigzag scan depends on syntax parameters and can be done in different ways.

Intra-coded blocks are scanned in normal, horizontal, or vertical mode depending on parameter ACPRED and on the DC prediction direction (which can be top or left) as follows. If $\text{ACPRED} = 0$, perform normal scan. If $\text{ACPRED} = 1$ and prediction direction is top, perform horizontal scan. If $\text{ACPRED} = 1$ and prediction direction is left, perform vertical scan. Figure 9.70 shows the normal and horizontal zigzag scans and cell numbering within an 8×8 block.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Normal scan: 0, 8, 1, 2, 9, 16, 24, 17, 10, 3, 4, 11, 18, 25, 32, 40, 33, 48, 26, 19, 12, 5, 6, 13, 20, 27, 34, 41, 56, 49, 57, 42, 35, 28, 21, 14, 7, 15, 22, 29, 36, 43, 50, 58, 51, 59, 44, 37, 30, 23, 31, 38, 45, 52, 60, 53, 61, 46, 39, 47, 54, 62, 55, 63.

Horizontal scan: 0, 1, 8, 2, 3, 9, 16, 24, 17, 10, 4, 5, 11, 18, 25, 32, 40, 48, 33, 26, 19, 12, 6, 7, 31, 20, 27, 34, 41, 56, 49, 57, 42, 35, 28, 21, 14, 15, 22, 29, 36, 43, 50, 58, 51, 44, 37, 30, 23, 31, 38, 45, 52, 59, 60, 53, 46, 39, 47, 54, 61, 62, 55, 63.

Figure 9.70: VC-1 Normal and Horizontal Zigzag Scans.

For inter-coded blocks, there are four block sizes and VC-1 defines two scan patterns for each, one for the simple and main profiles, and the other for the advanced profile; an implementor's nightmare!

9.11.11 Entropy Coding

The step following quantization is the entropy coding of the quantized transform coefficients. This is also very complex because many different tables of indexes and of variable-length codes are used, depending on the profile/level and the type of block being coded (intra or inter). Instead of listing the many rules, syntax parameters, and tables, we show a typical example that illustrates the basic process. Figure 9.71 shows an 8×8 block of quantized transform coefficients, where only eight of the 64 coefficients are nonzero. The block is scanned in basic zigzag pattern that brings the nonzero coefficients to the front, creates several short runs of zeros between them, and ends with a long run of zeros. Each nonzero coefficient C is first converted to a triplet (**run**, **level**, **flag**), where **run** is the length of the run of zeros preceding C , **level** is the (signed) value of C , and **flag** is a single bit, that is zero except for the last triplet.

Thus, the eight triplets for our example are (1, 1, 0), (0, -2, 0), (2, 5, 0), (1, -1, 0), (1, 1, 0), (0, -3, 0), (0, 4, 0), and (1, 1, 1).

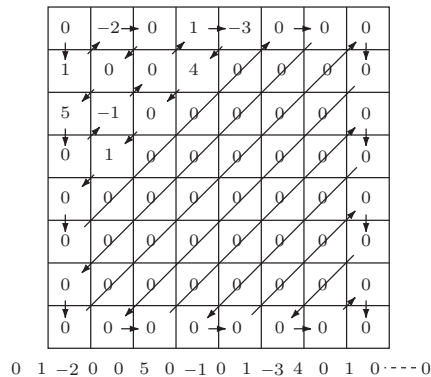


Figure 9.71: Example of Variable-Length Coding of a Block.

VC-1 employs 65 tables of variable-length codes. In the formal specification of this codec [SMPTE-421M 08], these tables are numbered from 168 to 232 and most are used to encode transform coefficients. Tables 175–190 and 219–232 contain variable-length codes for high-level inter and intra-coded blocks. Tables 191 through 204 are similarly used to encode low-level transform coefficients, and tables 205 through 218 specify codes for mid-level. Each set consists of tables with indexes and tables with the actual codes. For our example, we use the tables for low-motion, specifically table 199 for indexes of triplets of the form $(\text{run}, \text{level}, 0)$, table 200 for an index for the last triplet, whose format is $(\text{run}, \text{level}, 1)$, and table 198 for the actual variable-length codes. Table 9.72 lists parts of these three code tables.

The first triplet is $(1, 1, 0)$. We search table 199 for run 1 and level 1, and find index 14. Entry 14 of table 198 specifies the 4-bit code $11_{10} = 1011$. Thus, the encoded stream starts with $1011|0$, where the single zero implies a positive nonzero coefficient. The second triplet is $(0, -2, 0)$. We search table 199 for run 0 and level 2, and find index 1. Entry 1 of table 198 specifies the 5-bit code $20_{10} = 10100$. The encoded stream becomes $1011|0|10100|1$, where the last 1 implies a negative coefficient. The third triplet is $(2, 5, 0)$. We search table 199 for run 2 and level 5, and find index 27. Entry 27 of table 198 specifies the 14-bit code $1522_{10} = 00010111110010$. The encoded stream becomes $1011|0|10100|1|00010111110010|0$. This process continues until the last triplet $(1, 1, 1)$. Since the flag is 1 (indicating the last triplet), we search table 200 for an index. The table yields index 86 for run 1 and level 1, so we look at entry 86 of table 198 and find the 4-bit code $5 = 0101$. The complete encoded bitstream is therefore

$1011|0|10100|1|00010111110010|0|1011|1|1011|0|0010111|1|01111111|0|0101|0,$

and this encodes eight 12-bit signed nonzero quantized transform coefficients.

VLC								
Index	Code	Size	Index	Run	Level	Index	Run	Level
0	4	3	0	0	1	81	0	1
1	20	5	1	0	2	82	0	2
2	23	7	2	0	3	83	0	3
3	127	8	3	0	4	84	0	4
4	340	9	4	0	5	85	0	5
5	498	10	5	0	6	86	1	1
...	...		...	...		...	...	
14	11	4	14	1	1	103	7	2
...	...		...	...				
27	1522	14	27	2	5			
...	...		...	...				
86	5	4	80	29	1			
...	...							
147	5524	13						

Table 198

Table 199

Table 200

Table 9.72: Tables of Indexes and Variable-Length Codes.

In seven to ten years video traffic on the Internet
will exceed data and voice traffic combined.

—Bob Metcalfe

