

Computer Vision

Algorithms and Applications



Richard Szeliski



Springer

Texts in Computer Science

Editors

David Gries
Fred B. Schneider

For further volumes:
www.springer.com/series/3191

Richard Szeliski

Computer Vision

Algorithms and Applications



Dr. Richard Szeliski
Microsoft Research
One Microsoft Way
98052-6399 Redmond
Washington
USA
szeliski@microsoft.com

Series Editors

David Gries
Department of Computer Science
Upson Hall
Cornell University
Ithaca, NY 14853-7501, USA

Fred B. Schneider
Department of Computer Science
Upson Hall
Cornell University
Ithaca, NY 14853-7501, USA

ISSN 1868-0941 e-ISSN 1868-095X
ISBN 978-1-84882-934-3 e-ISBN 978-1-84882-935-0
DOI 10.1007/978-1-84882-935-0
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: 2010936817

© Springer-Verlag London Limited 2011

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.

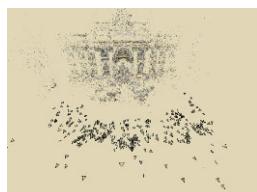
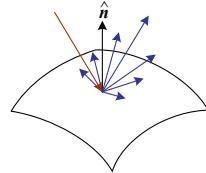
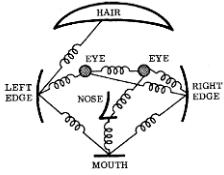
The use of registered names, trademarks, etc., in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

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.

Printed on acid-free paper

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

*This book is dedicated to my parents,
Zdzisław and Jadwiga,
and my family,
Lyn, Anne, and Stephen.*



1 Introduction

1

What is computer vision? • A brief history •
Book overview • Sample syllabus • Notation

2 Image formation

27

Geometric primitives and transformations •
Photometric image formation •
The digital camera

3 Image processing

87

Point operators • Linear filtering •
More neighborhood operators • Fourier transforms •
Pyramids and wavelets • Geometric transformations •
Global optimization

4 Feature detection and matching

181

Points and patches •
Edges • Lines

5 Segmentation

235

Active contours • Split and merge •
Mean shift and mode finding • Normalized cuts •
Graph cuts and energy-based methods

6 Feature-based alignment

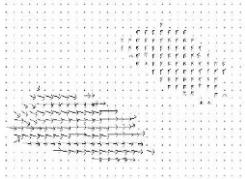
273

2D and 3D feature-based alignment •
Pose estimation •
Geometric intrinsic calibration

7 Structure from motion

303

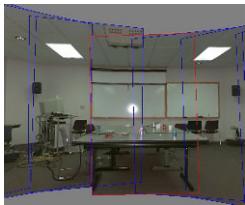
Triangulation • Two-frame structure from motion •
Factorization • Bundle adjustment •
Constrained structure and motion



8 Dense motion estimation

335

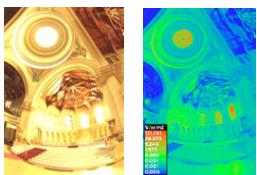
- Translational alignment • Parametric motion •
- Spline-based motion • Optical flow •
- Layered motion



9 Image stitching

375

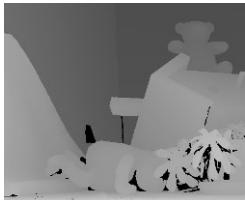
- Motion models • Global alignment •
- Compositing



10 Computational photography

409

- Photometric calibration • High dynamic range imaging •
- Super-resolution and blur removal •
- Image matting and compositing •
- Texture analysis and synthesis



11 Stereo correspondence

467

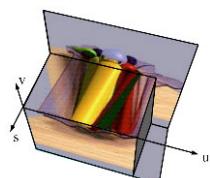
- Epipolar geometry • Sparse correspondence •
- Dense correspondence • Local methods •
- Global optimization • Multi-view stereo



12 3D reconstruction

505

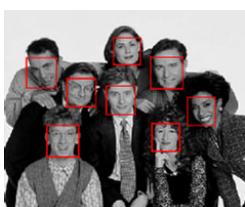
- Shape from X • Active rangefinding •
- Surface representations • Point-based representations •
- Volumetric representations • Model-based reconstruction •
- Recovering texture maps and albedos



13 Image-based rendering

543

- View interpolation • Layered depth images •
- Light fields and Lumigraphs • Environment mattes •
- Video-based rendering



14 Recognition

575

- Object detection • Face recognition •
- Instance recognition • Category recognition •
- Context and scene understanding •
- Recognition databases and test sets

Preface

The seeds for this book were first planted in 2001 when Steve Seitz at the University of Washington invited me to co-teach a course called “Computer Vision for Computer Graphics”. At that time, computer vision techniques were increasingly being used in computer graphics to create image-based models of real-world objects, to create visual effects, and to merge real-world imagery using computational photography techniques. Our decision to focus on the applications of computer vision to fun problems such as image stitching and photo-based 3D modeling from personal photos seemed to resonate well with our students.

Since that time, a similar syllabus and project-oriented course structure has been used to teach general computer vision courses both at the University of Washington and at Stanford. (The latter was a course I co-taught with David Fleet in 2003.) Similar curricula have been adopted at a number of other universities and also incorporated into more specialized courses on computational photography. (For ideas on how to use this book in your own course, please see Table 1.1 in Section 1.4.)

This book also reflects my 20 years’ experience doing computer vision research in corporate research labs, mostly at Digital Equipment Corporation’s Cambridge Research Lab and at Microsoft Research. In pursuing my work, I have mostly focused on problems and solution techniques (algorithms) that have practical real-world applications and that work well in practice. Thus, this book has more emphasis on basic techniques that work under real-world conditions and less on more esoteric mathematics that has intrinsic elegance but less practical applicability.

This book is suitable for teaching a senior-level undergraduate course in computer vision to students in both computer science and electrical engineering. I prefer students to have either an image processing or a computer graphics course as a prerequisite so that they can spend less time learning general background mathematics and more time studying computer vision techniques. The book is also suitable for teaching graduate-level courses in computer vision (by delving into the more demanding application and algorithmic areas) and as a general reference to fundamental techniques and the recent research literature. To this end, I have attempted wherever possible to at least cite the newest research in each sub-field, even if the technical details are too complex to cover in the book itself.

In teaching our courses, we have found it useful for the students to attempt a number of small implementation projects, which often build on one another, in order to get them used to working with real-world images and the challenges that these present. The students are then asked to choose an individual topic for each of their small-group, final projects. (Sometimes these projects even turn into conference papers!) The exercises at the end of each chapter contain numerous suggestions for smaller mid-term projects, as well as more open-ended

problems whose solutions are still active research topics. Wherever possible, I encourage students to try their algorithms on their own personal photographs, since this better motivates them, often leads to creative variants on the problems, and better acquaints them with the variety and complexity of real-world imagery.

In formulating and solving computer vision problems, I have often found it useful to draw inspiration from three high-level approaches:

- **Scientific:** build detailed models of the image formation process and develop mathematical techniques to invert these in order to recover the quantities of interest (where necessary, making simplifying assumption to make the mathematics more tractable).
- **Statistical:** use probabilistic models to quantify the prior likelihood of your unknowns and the noisy measurement processes that produce the input images, then infer the best possible estimates of your desired quantities and analyze their resulting uncertainties. The inference algorithms used are often closely related to the optimization techniques used to invert the (scientific) image formation processes.
- **Engineering:** develop techniques that are simple to describe and implement but that are also known to work well in practice. Test these techniques to understand their limitation and failure modes, as well as their expected computational costs (run-time performance).

These three approaches build on each other and are used throughout the book.

My personal research and development philosophy (and hence the exercises in the book) have a strong emphasis on *testing* algorithms. It's too easy in computer vision to develop an algorithm that does something *plausible* on a few images rather than something *correct*. The best way to validate your algorithms is to use a three-part strategy.

First, test your algorithm on clean synthetic data, for which the exact results are known. Second, add noise to the data and evaluate how the performance degrades as a function of noise level. Finally, test the algorithm on real-world data, preferably drawn from a wide variety of sources, such as photos found on the Web. Only then can you truly know if your algorithm can deal with real-world complexity, i.e., images that do not fit some simplified model or assumptions.

In order to help students in this process, this books comes with a large amount of supplementary material, which can be found on the book's Web site <http://szeliski.org/Book>. This material, which is described in Appendix C, includes:

- pointers to commonly used data sets for the problems, which can be found on the Web
- pointers to software libraries, which can help students get started with basic tasks such as reading/writing images or creating and manipulating images
- slide sets corresponding to the material covered in this book
- a BibTeX bibliography of the papers cited in this book.

The latter two resources may be of more interest to instructors and researchers publishing new papers in this field, but they will probably come in handy even with regular students. Some of the software libraries contain implementations of a wide variety of computer vision algorithms, which can enable you to tackle more ambitious projects (with your instructor's consent).

Acknowledgements

I would like to gratefully acknowledge all of the people whose passion for research and inquiry as well as encouragement have helped me write this book.

Steve Zucker at McGill University first introduced me to computer vision, taught all of his students to question and debate research results and techniques, and encouraged me to pursue a graduate career in this area.

Takeo Kanade and Geoff Hinton, my Ph. D. thesis advisors at Carnegie Mellon University, taught me the fundamentals of good research, writing, and presentation. They fired up my interest in visual processing, 3D modeling, and statistical methods, while Larry Matthies introduced me to Kalman filtering and stereo matching.

Demetri Terzopoulos was my mentor at my first industrial research job and taught me the ropes of successful publishing. Yvan Leclerc and Pascal Fua, colleagues from my brief interlude at SRI International, gave me new perspectives on alternative approaches to computer vision.

During my six years of research at Digital Equipment Corporation's Cambridge Research Lab, I was fortunate to work with a great set of colleagues, including Ingrid Carlom, Gudrun Klinker, Keith Waters, Richard Weiss, Stéphane Lavallée, and Sing Bing Kang, as well as to supervise the first of a long string of outstanding summer interns, including David Tonnesen, Sing Bing Kang, James Coughlan, and Harry Shum. This is also where I began my long-term collaboration with Daniel Scharstein, now at Middlebury College.

At Microsoft Research, I've had the outstanding fortune to work with some of the world's best researchers in computer vision and computer graphics, including Michael Cohen, Hugues Hoppe, Stephen Gortler, Steve Shafer, Matthew Turk, Harry Shum, Anandan, Phil Torr, Antonio Criminisi, Georg Petschnigg, Kentaro Toyama, Ramin Zabih, Shai Avidan, Sing Bing Kang, Matt Uyttendaele, Patrice Simard, Larry Zitnick, Richard Hartley, Simon Winder, Drew Steedly, Chris Pal, Nebojsa Jojic, Patrick Baudisch, Dani Lischinski, Matthew Brown, Simon Baker, Michael Goesele, Eric Stollnitz, David Nistér, Blaise Aguera y Arcas, Sudipta Sinha, Johannes Kopf, Neel Joshi, and Krishnan Ramnath. I was also lucky to have as interns such great students as Polina Golland, Simon Baker, Mei Han, Arno Schödl, Ron Dror, Ashley Eden, Jinxiang Chai, Rahul Swaminathan, Yanghai Tsin, Sam Hasinoff, Anat Levin, Matthew Brown, Eric Bennett, Vaibhav Vaish, Jan-Michael Frahm, James Diebel, Ce Liu, Josef Sivic, Grant Schindler, Colin Zheng, Neel Joshi, Sudipta Sinha, Zeev Farbman, Rahul Garg, Tim Cho, Yekeun Jeong, Richard Roberts, Varsha Hedau, and Dilip Krishnan.

While working at Microsoft, I've also had the opportunity to collaborate with wonderful colleagues at the University of Washington, where I hold an Affiliate Professor appointment. I'm indebted to Tony DeRose and David Salesin, who first encouraged me to get involved with the research going on at UW, my long-time collaborators Brian Curless, Steve Seitz, Maneesh Agrawala, Sameer Agarwal, and Yasu Furukawa, as well as the students I have had the privilege to supervise and interact with, including Frédéric Pighin, Yung-Yu Chuang, Doug Zongker, Colin Zheng, Aseem Agarwala, Dan Goldman, Noah Snavely, Rahul Garg, and Ryan Kaminsky. As I mentioned at the beginning of this preface, this book owes its inception to the vision course that Steve Seitz invited me to co-teach, as well as to Steve's encouragement, course notes, and editorial input.

I'm also grateful to the many other computer vision researchers who have given me so many constructive suggestions about the book, including Sing Bing Kang, who was my infor-

mal book editor, Vladimir Kolmogorov, who contributed Appendix B.5.5 on linear programming techniques for MRF inference, Daniel Scharstein, Richard Hartley, Simon Baker, Noah Snavely, Bill Freeman, Svetlana Lazebnik, Matthew Turk, Jitendra Malik, Alyosha Efros, Michael Black, Brian Curless, Sameer Agarwal, Li Zhang, Deva Ramanan, Olga Veksler, Yuri Boykov, Carsten Rother, Phil Torr, Bill Triggs, Bruce Maxwell, Jana Košecká, Eero Simoncelli, Aaron Hertzmann, Antonio Torralba, Tomaso Poggio, Theo Pavlidis, Baba Vemuri, Nando de Freitas, Chuck Dyer, Song Yi, Falk Schubert, Roman Pflugfelder, Marshall Tappen, James Coughlan, Sammy Rogmans, Klaus Strobel, Shanmuganathan, Andreas Siebert, Yongjun Wu, Fred Pighin, Juan Cockburn, Ronald Mallet, Tim Soper, Georgios Evangelidis, Dwight Fowler, Itzik Bayaz, Daniel O'Connor, and Srikrishna Bhat. Shena Deuchers did a fantastic job copy-editing the book and suggesting many useful improvements and Wayne Wheeler and Simon Rees at Springer were most helpful throughout the whole book publishing process. Keith Price's Annotated Computer Vision Bibliography was invaluable in tracking down references and finding related work.

If you have any suggestions for improving the book, please send me an e-mail, as I would like to keep the book as accurate, informative, and timely as possible.

Lastly, this book would not have been possible or worthwhile without the incredible support and encouragement of my family. I dedicate this book to my parents, Zdzisław and Jadwiga, whose love, generosity, and accomplishments have always inspired me; to my sister Basia for her lifelong friendship; and especially to Lyn, Anne, and Stephen, whose daily encouragement in all matters (including this book project) makes it all worthwhile.

*Lake Wenatchee
August, 2010*

Contents

Preface	vii
1 Introduction	1
1.1 What is computer vision?	3
1.2 A brief history	10
1.3 Book overview	17
1.4 Sample syllabus	23
1.5 A note on notation	25
1.6 Additional reading	25
2 Image formation	27
2.1 Geometric primitives and transformations	29
2.1.1 Geometric primitives	29
2.1.2 2D transformations	33
2.1.3 3D transformations	36
2.1.4 3D rotations	37
2.1.5 3D to 2D projections	42
2.1.6 Lens distortions	52
2.2 Photometric image formation	54
2.2.1 Lighting	54
2.2.2 Reflectance and shading	55
2.2.3 Optics	61
2.3 The digital camera	65
2.3.1 Sampling and aliasing	69
2.3.2 Color	71
2.3.3 Compression	80
2.4 Additional reading	82
2.5 Exercises	82
3 Image processing	87
3.1 Point operators	89
3.1.1 Pixel transforms	91
3.1.2 Color transforms	92
3.1.3 Compositing and matting	92
3.1.4 Histogram equalization	94

3.1.5 <i>Application: Tonal adjustment</i>	97
3.2 Linear filtering	98
3.2.1 Separable filtering	102
3.2.2 Examples of linear filtering	103
3.2.3 Band-pass and steerable filters	104
3.3 More neighborhood operators	108
3.3.1 Non-linear filtering	108
3.3.2 Morphology	112
3.3.3 Distance transforms	113
3.3.4 Connected components	115
3.4 Fourier transforms	116
3.4.1 Fourier transform pairs	119
3.4.2 Two-dimensional Fourier transforms	123
3.4.3 Wiener filtering	123
3.4.4 <i>Application: Sharpening, blur, and noise removal</i>	126
3.5 Pyramids and wavelets	127
3.5.1 Interpolation	127
3.5.2 Decimation	130
3.5.3 Multi-resolution representations	132
3.5.4 Wavelets	136
3.5.5 <i>Application: Image blending</i>	140
3.6 Geometric transformations	143
3.6.1 Parametric transformations	145
3.6.2 Mesh-based warping	149
3.6.3 <i>Application: Feature-based morphing</i>	152
3.7 Global optimization	153
3.7.1 Regularization	154
3.7.2 Markov random fields	158
3.7.3 <i>Application: Image restoration</i>	169
3.8 Additional reading	169
3.9 Exercises	171
4 Feature detection and matching	181
4.1 Points and patches	183
4.1.1 Feature detectors	185
4.1.2 Feature descriptors	196
4.1.3 Feature matching	200
4.1.4 Feature tracking	207
4.1.5 <i>Application: Performance-driven animation</i>	209
4.2 Edges	210
4.2.1 Edge detection	210
4.2.2 Edge linking	215
4.2.3 <i>Application: Edge editing and enhancement</i>	219
4.3 Lines	220
4.3.1 Successive approximation	220
4.3.2 Hough transforms	221

4.3.3	Vanishing points	224
4.3.4	<i>Application:</i> Rectangle detection	226
4.4	Additional reading	227
4.5	Exercises	228
5	Segmentation	235
5.1	Active contours	237
5.1.1	Snakes	238
5.1.2	Dynamic snakes and CONDENSATION	243
5.1.3	Scissors	246
5.1.4	Level Sets	248
5.1.5	<i>Application:</i> Contour tracking and rotoscoping	249
5.2	Split and merge	250
5.2.1	Watershed	251
5.2.2	Region splitting (divisive clustering)	251
5.2.3	Region merging (agglomerative clustering)	251
5.2.4	Graph-based segmentation	252
5.2.5	Probabilistic aggregation	253
5.3	Mean shift and mode finding	254
5.3.1	K-means and mixtures of Gaussians	256
5.3.2	Mean shift	257
5.4	Normalized cuts	260
5.5	Graph cuts and energy-based methods	264
5.5.1	<i>Application:</i> Medical image segmentation	268
5.6	Additional reading	268
5.7	Exercises	270
6	Feature-based alignment	273
6.1	2D and 3D feature-based alignment	275
6.1.1	2D alignment using least squares	275
6.1.2	<i>Application:</i> Panography	277
6.1.3	Iterative algorithms	278
6.1.4	Robust least squares and RANSAC	281
6.1.5	3D alignment	283
6.2	Pose estimation	284
6.2.1	Linear algorithms	284
6.2.2	Iterative algorithms	286
6.2.3	<i>Application:</i> Augmented reality	287
6.3	Geometric intrinsic calibration	288
6.3.1	Calibration patterns	289
6.3.2	Vanishing points	290
6.3.3	<i>Application:</i> Single view metrology	292
6.3.4	Rotational motion	293
6.3.5	Radial distortion	295
6.4	Additional reading	296
6.5	Exercises	296

7 Structure from motion	303
7.1 Triangulation	305
7.2 Two-frame structure from motion	307
7.2.1 Projective (uncalibrated) reconstruction	312
7.2.2 Self-calibration	313
7.2.3 <i>Application:</i> View morphing	315
7.3 Factorization	315
7.3.1 Perspective and projective factorization	318
7.3.2 <i>Application:</i> Sparse 3D model extraction	319
7.4 Bundle adjustment	320
7.4.1 Exploiting sparsity	322
7.4.2 <i>Application:</i> Match move and augmented reality	324
7.4.3 Uncertainty and ambiguities	326
7.4.4 <i>Application:</i> Reconstruction from Internet photos	327
7.5 Constrained structure and motion	329
7.5.1 Line-based techniques	330
7.5.2 Plane-based techniques	331
7.6 Additional reading	332
7.7 Exercises	332
8 Dense motion estimation	335
8.1 Translational alignment	337
8.1.1 Hierarchical motion estimation	341
8.1.2 Fourier-based alignment	341
8.1.3 Incremental refinement	345
8.2 Parametric motion	350
8.2.1 <i>Application:</i> Video stabilization	354
8.2.2 Learned motion models	354
8.3 Spline-based motion	355
8.3.1 <i>Application:</i> Medical image registration	358
8.4 Optical flow	360
8.4.1 Multi-frame motion estimation	363
8.4.2 <i>Application:</i> Video denoising	364
8.4.3 <i>Application:</i> De-interlacing	364
8.5 Layered motion	365
8.5.1 <i>Application:</i> Frame interpolation	368
8.5.2 Transparent layers and reflections	368
8.6 Additional reading	370
8.7 Exercises	371
9 Image stitching	375
9.1 Motion models	378
9.1.1 Planar perspective motion	379
9.1.2 <i>Application:</i> Whiteboard and document scanning	379
9.1.3 Rotational panoramas	380
9.1.4 Gap closing	382

9.1.5 <i>Application:</i> Video summarization and compression	383
9.1.6 Cylindrical and spherical coordinates	385
9.2 Global alignment	387
9.2.1 Bundle adjustment	388
9.2.2 Parallax removal	391
9.2.3 Recognizing panoramas	392
9.2.4 Direct vs. feature-based alignment	393
9.3 Compositing	396
9.3.1 Choosing a compositing surface	396
9.3.2 Pixel selection and weighting (de-ghosting)	398
9.3.3 <i>Application:</i> Photomontage	403
9.3.4 Blending	403
9.4 Additional reading	406
9.5 Exercises	407
10 Computational photography	409
10.1 Photometric calibration	412
10.1.1 Radiometric response function	412
10.1.2 Noise level estimation	415
10.1.3 Vignetting	416
10.1.4 Optical blur (spatial response) estimation	416
10.2 High dynamic range imaging	419
10.2.1 Tone mapping	427
10.2.2 <i>Application:</i> Flash photography	434
10.3 Super-resolution and blur removal	436
10.3.1 Color image demosaicing	440
10.3.2 <i>Application:</i> Colorization	442
10.4 Image matting and compositing	443
10.4.1 Blue screen matting	445
10.4.2 Natural image matting	446
10.4.3 Optimization-based matting	450
10.4.4 Smoke, shadow, and flash matting	452
10.4.5 Video matting	454
10.5 Texture analysis and synthesis	455
10.5.1 <i>Application:</i> Hole filling and inpainting	457
10.5.2 <i>Application:</i> Non-photorealistic rendering	458
10.6 Additional reading	460
10.7 Exercises	461
11 Stereo correspondence	467
11.1 Epipolar geometry	471
11.1.1 Rectification	472
11.1.2 Plane sweep	474
11.2 Sparse correspondence	475
11.2.1 3D curves and profiles	476
11.3 Dense correspondence	477

11.3.1	Similarity measures	479
11.4	Local methods	480
11.4.1	Sub-pixel estimation and uncertainty	482
11.4.2	<i>Application:</i> Stereo-based head tracking	483
11.5	Global optimization	484
11.5.1	Dynamic programming	485
11.5.2	Segmentation-based techniques	487
11.5.3	<i>Application:</i> Z-keying and background replacement	489
11.6	Multi-view stereo	489
11.6.1	Volumetric and 3D surface reconstruction	492
11.6.2	Shape from silhouettes	497
11.7	Additional reading	499
11.8	Exercises	500
12	3D reconstruction	505
12.1	Shape from X	508
12.1.1	Shape from shading and photometric stereo	508
12.1.2	Shape from texture	510
12.1.3	Shape from focus	511
12.2	Active rangefinding	512
12.2.1	Range data merging	515
12.2.2	<i>Application:</i> Digital heritage	517
12.3	Surface representations	518
12.3.1	Surface interpolation	518
12.3.2	Surface simplification	520
12.3.3	Geometry images	520
12.4	Point-based representations	521
12.5	Volumetric representations	522
12.5.1	Implicit surfaces and level sets	522
12.6	Model-based reconstruction	523
12.6.1	Architecture	524
12.6.2	Heads and faces	526
12.6.3	<i>Application:</i> Facial animation	528
12.6.4	Whole body modeling and tracking	530
12.7	Recovering texture maps and albedos	534
12.7.1	Estimating BRDFs	536
12.7.2	<i>Application:</i> 3D photography	537
12.8	Additional reading	538
12.9	Exercises	539
13	Image-based rendering	543
13.1	View interpolation	545
13.1.1	View-dependent texture maps	547
13.1.2	<i>Application:</i> Photo Tourism	548
13.2	Layered depth images	549
13.2.1	Impostors, sprites, and layers	549

13.3	Light fields and Lumigraphs	551
13.3.1	Unstructured Lumigraph	554
13.3.2	Surface light fields	555
13.3.3	<i>Application:</i> Concentric mosaics	556
13.4	Environment mattes	556
13.4.1	Higher-dimensional light fields	558
13.4.2	The modeling to rendering continuum	559
13.5	Video-based rendering	560
13.5.1	Video-based animation	560
13.5.2	Video textures	561
13.5.3	<i>Application:</i> Animating pictures	564
13.5.4	3D Video	564
13.5.5	<i>Application:</i> Video-based walkthroughs	566
13.6	Additional reading	569
13.7	Exercises	570
14	Recognition	575
14.1	Object detection	578
14.1.1	Face detection	578
14.1.2	Pedestrian detection	585
14.2	Face recognition	588
14.2.1	Eigenfaces	589
14.2.2	Active appearance and 3D shape models	596
14.2.3	<i>Application:</i> Personal photo collections	601
14.3	Instance recognition	602
14.3.1	Geometric alignment	603
14.3.2	Large databases	604
14.3.3	<i>Application:</i> Location recognition	609
14.4	Category recognition	611
14.4.1	Bag of words	612
14.4.2	Part-based models	615
14.4.3	Recognition with segmentation	620
14.4.4	<i>Application:</i> Intelligent photo editing	621
14.5	Context and scene understanding	625
14.5.1	Learning and large image collections	627
14.5.2	<i>Application:</i> Image search	630
14.6	Recognition databases and test sets	631
14.7	Additional reading	631
14.8	Exercises	637
15	Conclusion	641
A	Linear algebra and numerical techniques	645
A.1	Matrix decompositions	646
A.1.1	Singular value decomposition	646
A.1.2	Eigenvalue decomposition	647

A.1.3	QR factorization	649
A.1.4	Cholesky factorization	650
A.2	Linear least squares	651
A.2.1	Total least squares	653
A.3	Non-linear least squares	654
A.4	Direct sparse matrix techniques	655
A.4.1	Variable reordering	656
A.5	Iterative techniques	656
A.5.1	Conjugate gradient	657
A.5.2	Preconditioning	659
A.5.3	Multigrid	660
B	Bayesian modeling and inference	661
B.1	Estimation theory	662
B.1.1	Likelihood for multivariate Gaussian noise	663
B.2	Maximum likelihood estimation and least squares	665
B.3	Robust statistics	666
B.4	Prior models and Bayesian inference	667
B.5	Markov random fields	668
B.5.1	Gradient descent and simulated annealing	670
B.5.2	Dynamic programming	670
B.5.3	Belief propagation	672
B.5.4	Graph cuts	674
B.5.5	Linear programming	676
B.6	Uncertainty estimation (error analysis)	678
C	Supplementary material	679
C.1	Data sets	680
C.2	Software	682
C.3	Slides and lectures	689
C.4	Bibliography	690
References		691
Index		793

Chapter 1

Introduction

1.1	What is computer vision?	3
1.2	A brief history	10
1.3	Book overview	17
1.4	Sample syllabus	23
1.5	A note on notation	25
1.6	Additional reading	25



Figure 1.1 The human visual system has no problem interpreting the subtle variations in translucency and shading in this photograph and correctly segmenting the object from its background.

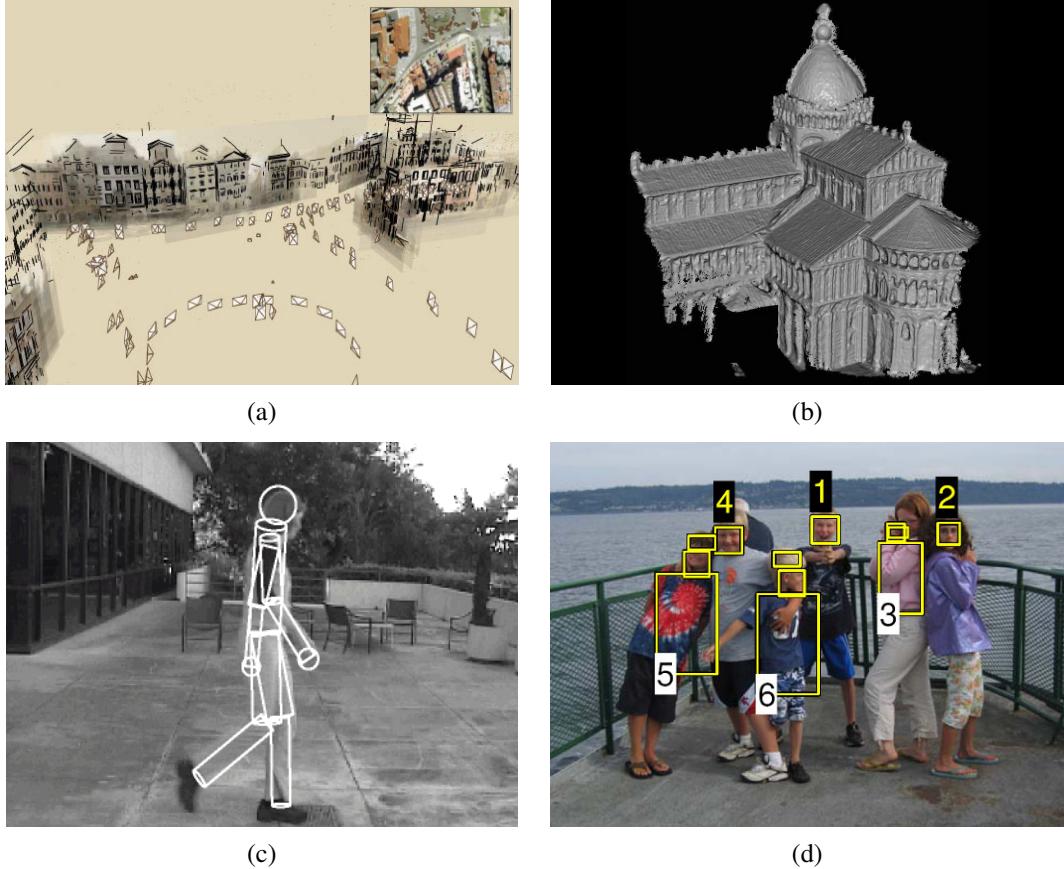


Figure 1.2 Some examples of computer vision algorithms and applications. (a) *Structure from motion* algorithms can reconstruct a sparse 3D point model of a large complex scene from hundreds of partially overlapping photographs (Snavely, Seitz, and Szeliski 2006) © 2006 ACM. (b) *Stereo matching* algorithms can build a detailed 3D model of a building façade from hundreds of differently exposed photographs taken from the Internet (Goesele, Snavely, Curless *et al.* 2007) © 2007 IEEE. (c) *Person tracking* algorithms can track a person walking in front of a cluttered background (Sidenbladh, Black, and Fleet 2000) © 2000 Springer. (d) *Face detection* algorithms, coupled with color-based clothing and hair detection algorithms, can locate and recognize the individuals in this image (Sivic, Zitnick, and Szeliski 2006) © 2006 Springer.

1.1 What is computer vision?

As humans, we perceive the three-dimensional structure of the world around us with apparent ease. Think of how vivid the three-dimensional percept is when you look at a vase of flowers sitting on the table next to you. You can tell the shape and translucency of each petal through the subtle patterns of light and shading that play across its surface and effortlessly segment each flower from the background of the scene (Figure 1.1). Looking at a framed group portrait, you can easily count (and name) all of the people in the picture and even guess at their emotions from their facial appearance. Perceptual psychologists have spent decades trying to understand how the visual system works and, even though they can devise optical illusions¹ to tease apart some of its principles (Figure 1.3), a complete solution to this puzzle remains elusive (Marr 1982; Palmer 1999; Livingstone 2008).

Researchers in computer vision have been developing, in parallel, mathematical techniques for recovering the three-dimensional shape and appearance of objects in imagery. We now have reliable techniques for accurately computing a partial 3D model of an environment from thousands of partially overlapping photographs (Figure 1.2a). Given a large enough set of views of a particular object or façade, we can create accurate dense 3D surface models using stereo matching (Figure 1.2b). We can track a person moving against a complex background (Figure 1.2c). We can even, with moderate success, attempt to find and name all of the people in a photograph using a combination of face, clothing, and hair detection and recognition (Figure 1.2d). However, despite all of these advances, the dream of having a computer interpret an image at the same level as a two-year old (for example, counting all of the animals in a picture) remains elusive. Why is vision so difficult? In part, it is because vision is an *inverse problem*, in which we seek to recover some unknowns given insufficient information to fully specify the solution. We must therefore resort to physics-based and probabilistic *models* to disambiguate between potential solutions. However, modeling the visual world in all of its rich complexity is far more difficult than, say, modeling the vocal tract that produces spoken sounds.

The *forward* models that we use in computer vision are usually developed in physics (radiometry, optics, and sensor design) and in computer graphics. Both of these fields model how objects move and animate, how light reflects off their surfaces, is scattered by the atmosphere, refracted through camera lenses (or human eyes), and finally projected onto a flat (or curved) image plane. While computer graphics are not yet perfect (no fully computer-animated movie with human characters has yet succeeded at crossing the *uncanny valley*² that separates real humans from android robots and computer-animated humans), in limited domains, such as rendering a still scene composed of everyday objects or animating extinct creatures such as dinosaurs, the illusion of reality *is* perfect.

In computer vision, we are trying to do the inverse, i.e., to describe the world that we see in one or more images and to reconstruct its properties, such as shape, illumination, and color distributions. It is amazing that humans and animals do this so effortlessly, while computer vision algorithms are so error prone. People who have not worked in the field often underestimate the difficulty of the problem. (Colleagues at work often ask me for software to find and name all the people in photos, so they can get on with the more “interesting” work.) This

¹ http://www.michaelbach.de/ot/sze_muelue

² The term *uncanny valley* was originally coined by roboticist Masahiro Mori as applied to robotics (Mori 1970). It is also commonly applied to computer-animated films such as *Final Fantasy* and *Polar Express* (Geller 2008).

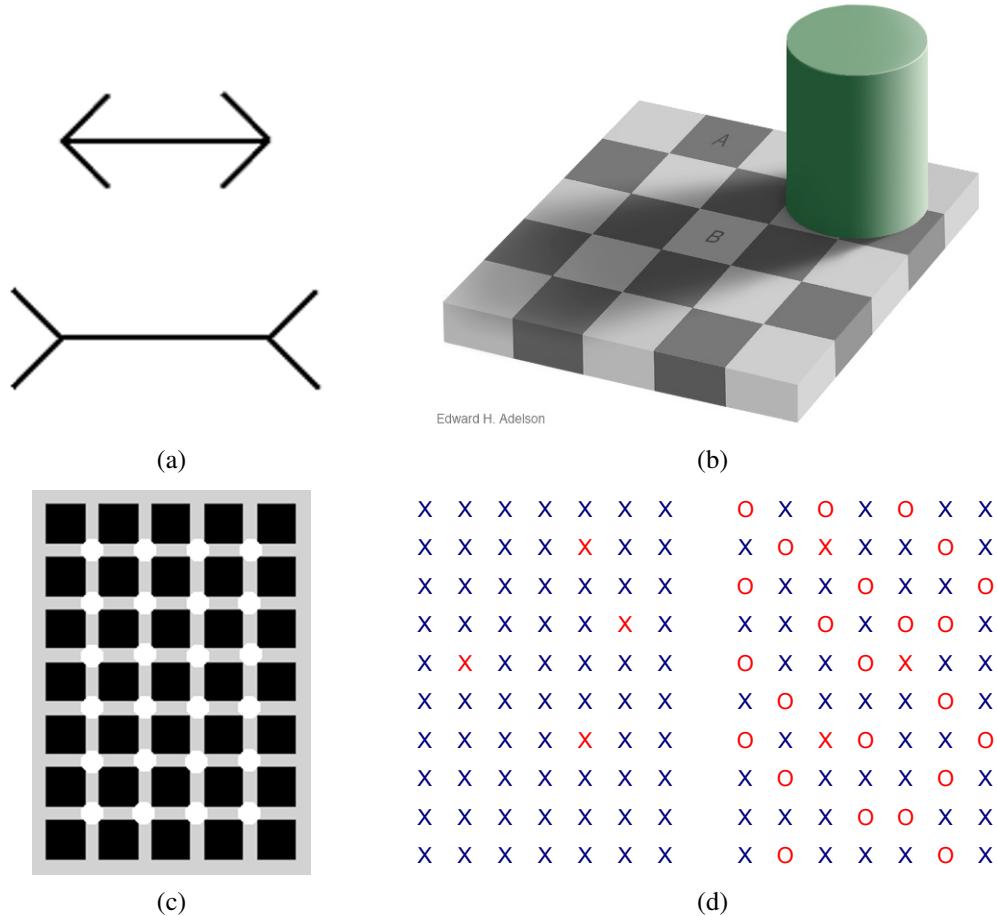


Figure 1.3 Some common optical illusions and what they might tell us about the visual system: (a) The classic Müller-Lyer illusion, where the length of the two horizontal lines appear different, probably due to the imagined perspective effects. (b) The “white” square B in the shadow and the “black” square A in the light actually have the same absolute intensity value. The percept is due to *brightness constancy*, the visual system’s attempt to discount illumination when interpreting colors. Image courtesy of Ted Adelson, http://web.mit.edu/persci/people/adelson/checkershadow_illusion.html. (c) A variation of the Hermann grid illusion, courtesy of Hany Farid, <http://www.cs.dartmouth.edu/~farid/illusions/hermann.html>. As you move your eyes over the figure, gray spots appear at the intersections. (d) Count the red Xs in the left half of the figure. Now count them in the right half. Is it significantly harder? The explanation has to do with a *pop-out* effect (Treisman 1985), which tells us about the operations of parallel perception and integration pathways in the brain.

misperception that vision should be easy dates back to the early days of artificial intelligence (see Section 1.2), when it was initially believed that the *cognitive* (logic proving and planning) parts of intelligence were intrinsically more difficult than the *perceptual* components (Boden 2006).

The good news is that computer vision *is* being used today in a wide variety of real-world applications, which include:

- **Optical character recognition (OCR):** reading handwritten postal codes on letters (Figure 1.4a) and automatic number plate recognition (ANPR);
- **Machine inspection:** rapid parts inspection for quality assurance using stereo vision with specialized illumination to measure tolerances on aircraft wings or auto body parts (Figure 1.4b) or looking for defects in steel castings using X-ray vision;
- **Retail:** object recognition for automated checkout lanes (Figure 1.4c);
- **3D model building (photogrammetry):** fully automated construction of 3D models from aerial photographs used in systems such as Bing Maps;
- **Medical imaging:** registering pre-operative and intra-operative imagery (Figure 1.4d) or performing long-term studies of people's brain morphology as they age;
- **Automotive safety:** detecting unexpected obstacles such as pedestrians on the street, under conditions where active vision techniques such as radar or lidar do not work well (Figure 1.4e; see also Miller, Campbell, Huttenlocher *et al.* (2008); Montemerlo, Becker, Bhat *et al.* (2008); Urmson, Anhalt, Bagnell *et al.* (2008) for examples of fully automated driving);
- **Match move:** merging computer-generated imagery (CGI) with live action footage by tracking feature points in the source video to estimate the 3D camera motion and shape of the environment. Such techniques are widely used in Hollywood (e.g., in movies such as Jurassic Park) (Roble 1999; Roble and Zafar 2009); they also require the use of precise *matting* to insert new elements between foreground and background elements (Chuang, Agarwala, Curless *et al.* 2002).
- **Motion capture (mocap):** using retro-reflective markers viewed from multiple cameras or other vision-based techniques to capture actors for computer animation;
- **Surveillance:** monitoring for intruders, analyzing highway traffic (Figure 1.4f), and monitoring pools for drowning victims;
- **Fingerprint recognition and biometrics:** for automatic access authentication as well as forensic applications.

David Lowe's Web site of industrial vision applications (<http://www.cs.ubc.ca/spider/lowe/vision.html>) lists many other interesting industrial applications of computer vision. While the above applications are all extremely important, they mostly pertain to fairly specialized kinds of imagery and narrow domains.

In this book, we focus more on broader *consumer-level* applications, such as fun things you can do with your own personal photographs and video. These include:

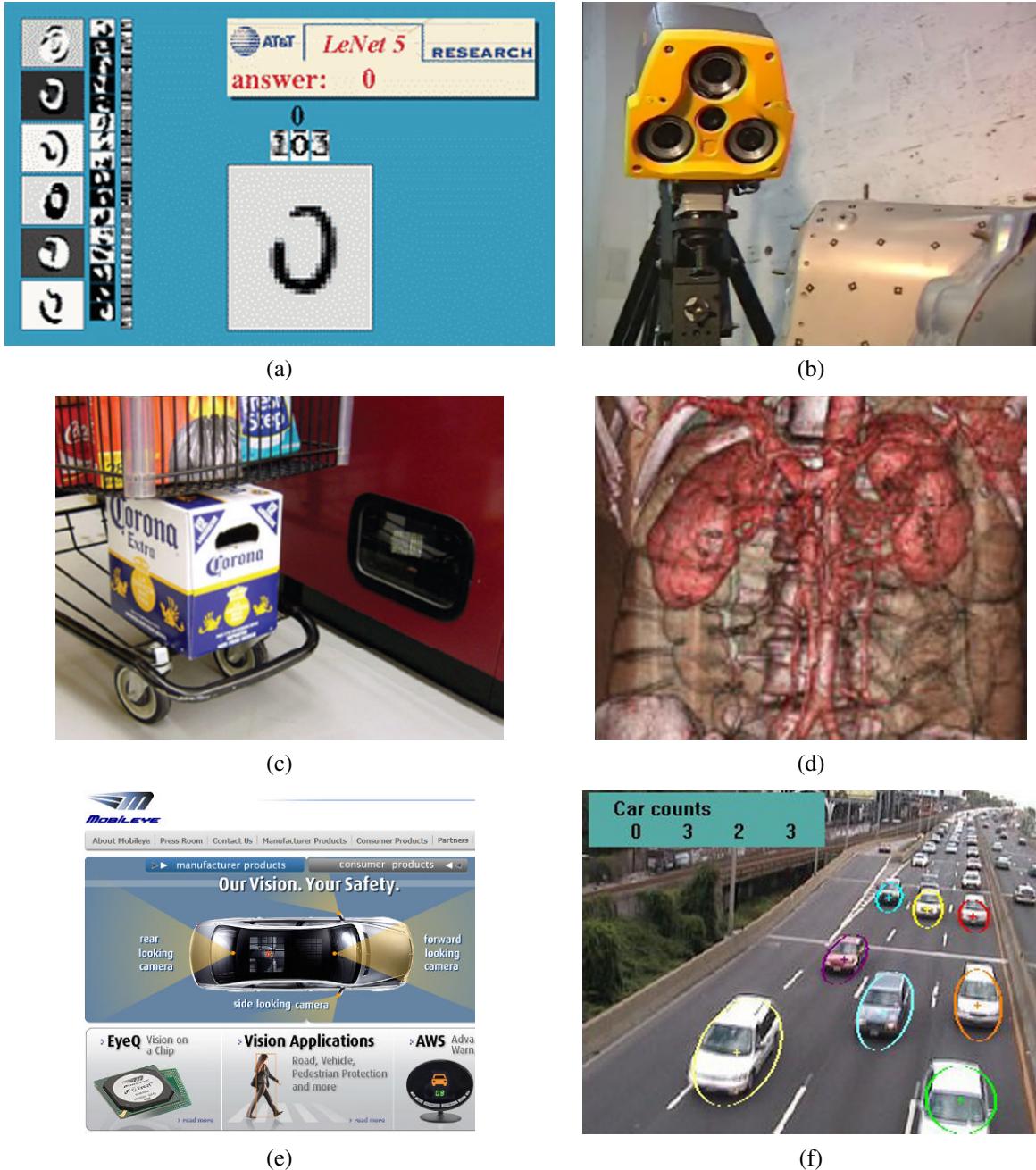


Figure 1.4 Some industrial applications of computer vision: (a) optical character recognition (OCR) <http://yann.lecun.com/exdb/lenet/>; (b) mechanical inspection <http://www.cognitens.com/>; (c) retail <http://www.evoretail.com/>; (d) medical imaging <http://www.clarontech.com/>; (e) automotive safety <http://www.mobileye.com/>; (f) surveillance and traffic monitoring <http://www.honeywellvideo.com/>, courtesy of Honeywell International Inc.

- **Stitching:** turning overlapping photos into a single seamlessly stitched panorama (Figure 1.5a), as described in Chapter 9;
- **Exposure bracketing:** merging multiple exposures taken under challenging lighting conditions (strong sunlight and shadows) into a single perfectly exposed image (Figure 1.5b), as described in Section 10.2;
- **Morphing:** turning a picture of one of your friends into another, using a seamless *morph* transition (Figure 1.5c);
- **3D modeling:** converting one or more snapshots into a 3D model of the object or person you are photographing (Figure 1.5d), as described in Section 12.6
- **Video match move and stabilization:** inserting 2D pictures or 3D models into your videos by automatically tracking nearby reference points (see Section 7.4.2)³ or using motion estimates to remove shake from your videos (see Section 8.2.1);
- **Photo-based walkthroughs:** navigating a large collection of photographs, such as the interior of your house, by flying between different photos in 3D (see Sections 13.1.2 and 13.5.5)
- **Face detection:** for improved camera focusing as well as more relevant image searching (see Section 14.1.1);
- **Visual authentication:** automatically logging family members onto your home computer as they sit down in front of the webcam (see Section 14.2).

The great thing about these applications is that they are already familiar to most students; they are, at least, technologies that students can immediately appreciate and use with their own personal media. Since computer vision is a challenging topic, given the wide range of mathematics being covered⁴ and the intrinsically difficult nature of the problems being solved, having fun and relevant problems to work on can be highly motivating and inspiring.

The other major reason why this book has a strong focus on applications is that they can be used to *formulate* and *constrain* the potentially open-ended problems endemic in vision. For example, if someone comes to me and asks for a good edge detector, my first question is usually to ask *why*? What kind of problem are they trying to solve and why do they believe that edge detection is an important component? If they are trying to locate faces, I usually point out that most successful face detectors use a combination of skin color detection (Exercise 2.8) and simple blob features Section 14.1.1; they do not rely on edge detection. If they are trying to match door and window edges in a building for the purpose of 3D reconstruction, I tell them that edges are a fine idea but it is better to tune the edge detector for long edges (see Sections 3.2.3 and 4.2) and link them together into straight lines with common vanishing points before matching (see Section 4.3).

Thus, it is better to think back from the problem at hand to suitable techniques, rather than to grab the first technique that you may have heard of. This kind of working back from

³ For a fun student project on this topic, see the “PhotoBook” project at <http://www.cc.gatech.edu/dvfx/videos/dvfx2005.html>.

⁴ These techniques include physics, Euclidean and projective geometry, statistics, and optimization. They make computer vision a fascinating field to study and a great way to learn techniques widely applicable in other fields.

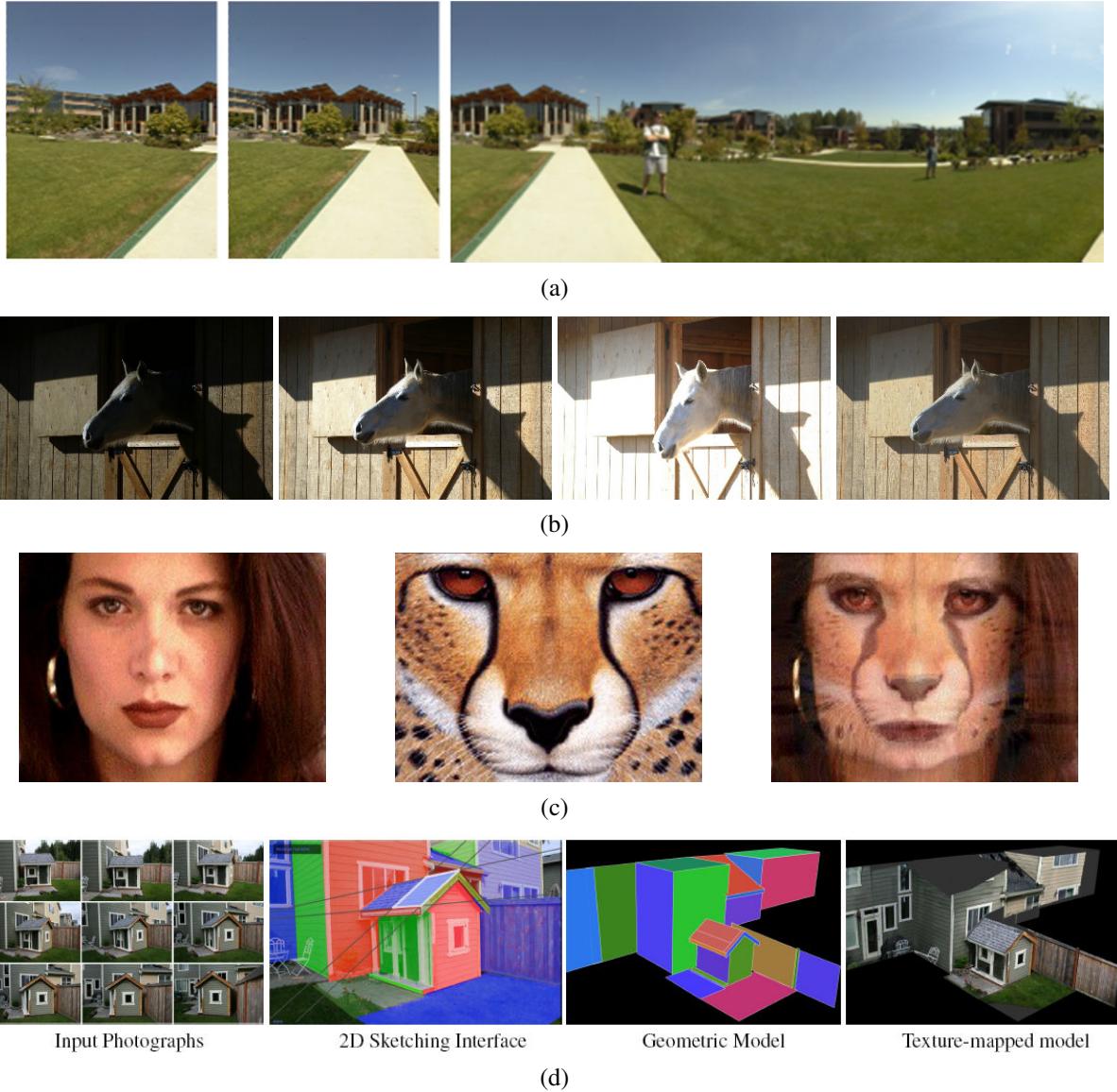


Figure 1.5 Some consumer applications of computer vision: (a) image stitching: merging different views (Szeliski and Shum 1997) © 1997 ACM; (b) exposure bracketing: merging different exposures; (c) morphing: blending between two photographs (Gomes, Darsa, Costa *et al.* 1999) © 1999 Morgan Kaufmann; (d) turning a collection of photographs into a 3D model (Sinha, Steedly, Szeliski *et al.* 2008) © 2008 ACM.

problems to solutions is typical of an **engineering** approach to the study of vision and reflects my own background in the field. First, I come up with a detailed problem definition and decide on the constraints and specifications for the problem. Then, I try to find out which techniques are known to work, implement a few of these, evaluate their performance, and finally make a selection. In order for this process to work, it is important to have realistic **test data**, both synthetic, which can be used to verify correctness and analyze noise sensitivity, and real-world data typical of the way the system will finally be used.

However, this book is not just an engineering text (a source of recipes). It also takes a **scientific** approach to basic vision problems. Here, I try to come up with the best possible models of the physics of the system at hand: how the scene is created, how light interacts with the scene and atmospheric effects, and how the sensors work, including sources of noise and uncertainty. The task is then to try to invert the acquisition process to come up with the best possible description of the scene.

The book often uses a **statistical** approach to formulating and solving computer vision problems. Where appropriate, probability distributions are used to model the scene and the noisy image acquisition process. The association of prior distributions with unknowns is often called *Bayesian modeling* (Appendix B). It is possible to associate a risk or loss function with mis-estimating the answer (Section B.2) and to set up your inference algorithm to minimize the expected risk. (Consider a robot trying to estimate the distance to an obstacle: it is usually safer to underestimate than to overestimate.) With statistical techniques, it often helps to gather lots of training data from which to learn probabilistic models. Finally, statistical approaches enable you to use proven inference techniques to estimate the best answer (or distribution of answers) and to quantify the uncertainty in the resulting estimates.

Because so much of computer vision involves the solution of inverse problems or the estimation of unknown quantities, my book also has a heavy emphasis on **algorithms**, especially those that are known to work well in practice. For many vision problems, it is all too easy to come up with a mathematical description of the problem that either does not match realistic real-world conditions or does not lend itself to the stable estimation of the unknowns. What we need are algorithms that are both **robust** to noise and deviation from our models and reasonably **efficient** in terms of run-time resources and space. In this book, I go into these issues in detail, using Bayesian techniques, where applicable, to ensure robustness, and efficient search, minimization, and linear system solving algorithms to ensure efficiency. Most of the algorithms described in this book are at a high level, being mostly a list of steps that have to be filled in by students or by reading more detailed descriptions elsewhere. In fact, many of the algorithms are sketched out in the exercises.

Now that I've described the goals of this book and the frameworks that I use, I devote the rest of this chapter to two additional topics. Section 1.2 is a brief synopsis of the history of computer vision. It can easily be skipped by those who want to get to "the meat" of the new material in this book and do not care as much about who invented what when.

The second is an overview of the book's contents, Section 1.3, which is useful reading for everyone who intends to make a study of this topic (or to jump in partway, since it describes chapter inter-dependencies). This outline is also useful for instructors looking to structure one or more courses around this topic, as it provides sample curricula based on the book's contents.

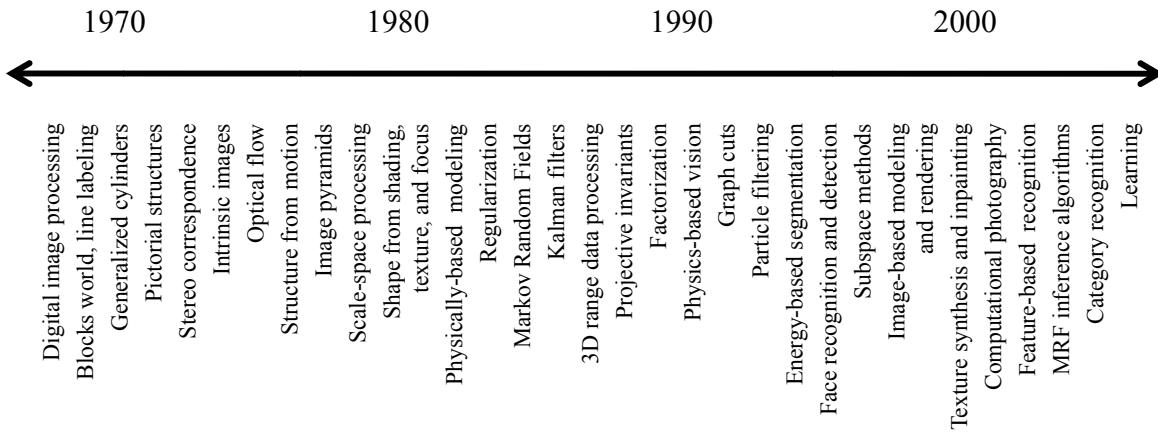


Figure 1.6 A rough timeline of some of the most active topics of research in computer vision.

1.2 A brief history

In this section, I provide a brief personal synopsis of the main developments in computer vision over the last 30 years (Figure 1.6); at least, those that I find personally interesting and which appear to have stood the test of time. Readers not interested in the provenance of various ideas and the evolution of this field should skip ahead to the book overview in Section 1.3.

1970s. When computer vision first started out in the early 1970s, it was viewed as the visual perception component of an ambitious agenda to mimic human intelligence and to endow robots with intelligent behavior. At the time, it was believed by some of the early pioneers of artificial intelligence and robotics (at places such as MIT, Stanford, and CMU) that solving the “visual input” problem would be an easy step along the path to solving more difficult problems such as higher-level reasoning and planning. According to one well-known story, in 1966, Marvin Minsky at MIT asked his undergraduate student Gerald Jay Sussman to “spend the summer linking a camera to a computer and getting the computer to describe what it saw” (Boden 2006, p. 781).⁵ We now know that the problem is slightly more difficult than that.⁶

What distinguished computer vision from the already existing field of digital image processing (Rosenfeld and Pfaltz 1966; Rosenfeld and Kak 1976) was a desire to recover the three-dimensional structure of the world from images and to use this as a stepping stone towards full scene understanding. Winston (1975) and Hanson and Riseman (1978) provide two nice collections of classic papers from this early period.

Early attempts at scene understanding involved extracting edges and then inferring the 3D structure of an object or a “blocks world” from the topological structure of the 2D lines

⁵ Boden (2006) cites (Crevier 1993) as the original source. The actual Vision Memo was authored by Seymour Papert (1966) and involved a whole cohort of students.

⁶ To see how far robotic vision has come in the last four decades, have a look at the towel-folding robot at <http://rll.eecs.berkeley.edu/pr/icra10/> (Maitin-Shepard, Cusumano-Towner, Lei *et al.* 2010).

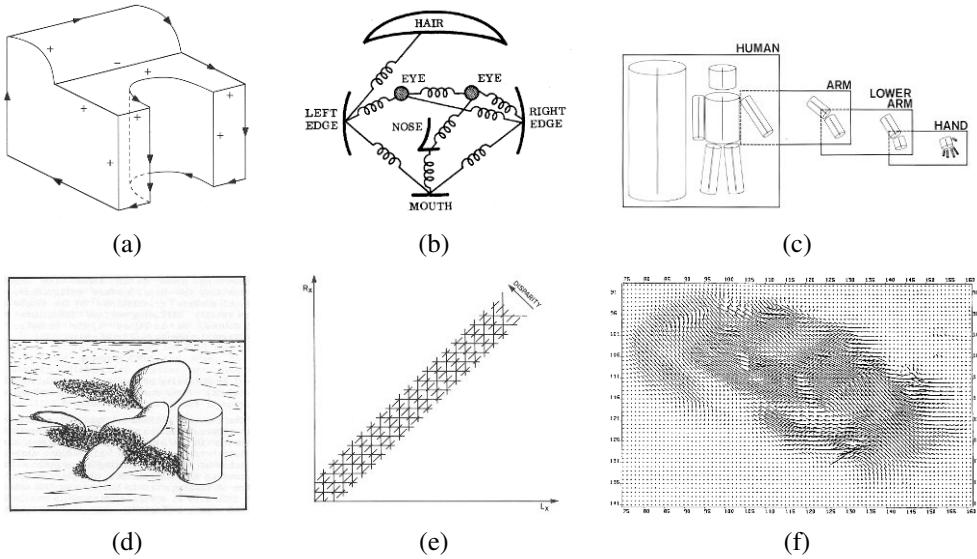


Figure 1.7 Some early (1970s) examples of computer vision algorithms: (a) line labeling (Nalwa 1993) © 1993 Addison-Wesley, (b) pictorial structures (Fischler and Elschlager 1973) © 1973 IEEE, (c) articulated body model (Marr 1982) © 1982 David Marr, (d) intrinsic images (Barrow and Tenenbaum 1981) © 1973 IEEE, (e) stereo correspondence (Marr 1982) © 1982 David Marr, (f) optical flow (Nagel and Enkelmann 1986) © 1986 IEEE.

(Roberts 1965). Several *line labeling* algorithms (Figure 1.7a) were developed at that time (Huffman 1971; Clowes 1971; Waltz 1975; Rosenfeld, Hummel, and Zucker 1976; Kanade 1980). Nalwa (1993) gives a nice review of this area. The topic of edge detection was also an active area of research; a nice survey of contemporaneous work can be found in (Davis 1975).

Three-dimensional modeling of non-polyhedral objects was also being studied (Baumgart 1974; Baker 1977). One popular approach used *generalized cylinders*, i.e., solids of revolution and swept closed curves (Agin and Binford 1976; Nevatia and Binford 1977), often arranged into parts relationships⁷ (Hinton 1977; Marr 1982) (Figure 1.7c). Fischler and Elschlager (1973) called such *elastic* arrangements of parts *pictorial structures* (Figure 1.7b). This is currently one of the favored approaches being used in object recognition (see Section 14.4 and Felzenszwalb and Huttenlocher 2005).

A qualitative approach to understanding intensities and shading variations and explaining them by the effects of image formation phenomena, such as surface orientation and shadows, was championed by Barrow and Tenenbaum (1981) in their paper on *intrinsic images* (Figure 1.7d), along with the related *2½-D sketch* ideas of Marr (1982). This approach is again seeing a bit of a revival in the work of Tappen, Freeman, and Adelson (2005).

More quantitative approaches to computer vision were also developed at the time, including the first of many feature-based stereo correspondence algorithms (Figure 1.7e) (Dev 1974; Marr and Poggio 1976; Moravec 1977; Marr and Poggio 1979; Mayhew and Frisby 1981; Baker 1982; Barnard and Fischler 1982; Ohta and Kanade 1985; Grimson 1985; Pollard, Mayhew, and Frisby 1985; Prazdny 1985) and intensity-based optical flow algorithms

⁷ In robotics and computer animation, these linked-part graphs are often called *kinematic chains*.

(Figure 1.7f) (Horn and Schunck 1981; Huang 1981; Lucas and Kanade 1981; Nagel 1986). The early work in simultaneously recovering 3D structure and camera motion (see Chapter 7) also began around this time (Ullman 1979; Longuet-Higgins 1981).

A lot of the philosophy of how vision was believed to work at the time is summarized in David Marr's (1982) book.⁸ In particular, Marr introduced his notion of the three levels of description of a (visual) information processing system. These three levels, very loosely paraphrased according to my own interpretation, are:

- **Computational theory:** What is the goal of the computation (task) and what are the constraints that are known or can be brought to bear on the problem?
- **Representations and algorithms:** How are the input, output, and intermediate information represented and which algorithms are used to calculate the desired result?
- **Hardware implementation:** How are the representations and algorithms mapped onto actual hardware, e.g., a biological vision system or a specialized piece of silicon? Conversely, how can hardware constraints be used to guide the choice of representation and algorithm? With the increasing use of graphics chips (GPUs) and many-core architectures for computer vision (see Section C.2), this question is again becoming quite relevant.

As I mentioned earlier in this introduction, it is my conviction that a careful analysis of the problem specification and known constraints from image formation and priors (the scientific and statistical approaches) must be married with efficient and robust algorithms (the engineering approach) to design successful vision algorithms. Thus, it seems that Marr's philosophy is as good a guide to framing and solving problems in our field today as it was 25 years ago.

1980s. In the 1980s, a lot of attention was focused on more sophisticated mathematical techniques for performing quantitative image and scene analysis.

Image pyramids (see Section 3.5) started being widely used to perform tasks such as image blending (Figure 1.8a) and coarse-to-fine correspondence search (Rosenfeld 1980; Burt and Adelson 1983a,b; Rosenfeld 1984; Quam 1984; Anandan 1989). Continuous versions of pyramids using the concept of *scale-space* processing were also developed (Witkin 1983; Witkin, Terzopoulos, and Kass 1986; Lindeberg 1990). In the late 1980s, wavelets (see Section 3.5.4) started displacing or augmenting regular image pyramids in some applications (Adelson, Simoncelli, and Hingorani 1987; Mallat 1989; Simoncelli and Adelson 1990a,b; Simoncelli, Freeman, Adelson *et al.* 1992).

The use of stereo as a quantitative shape cue was extended by a wide variety of *shape-from-X* techniques, including shape from shading (Figure 1.8b) (see Section 12.1.1 and Horn 1975; Pentland 1984; Blake, Zimmerman, and Knowles 1985; Horn and Brooks 1986, 1989), photometric stereo (see Section 12.1.1 and Woodham 1981), shape from texture (see Section 12.1.2 and Witkin 1981; Pentland 1984; Malik and Rosenholtz 1997), and shape from focus (see Section 12.1.3 and Nayar, Watanabe, and Noguchi 1995). Horn (1986) has a nice discussion of most of these techniques.

⁸ More recent developments in visual perception theory are covered in (Palmer 1999; Livingstone 2008).

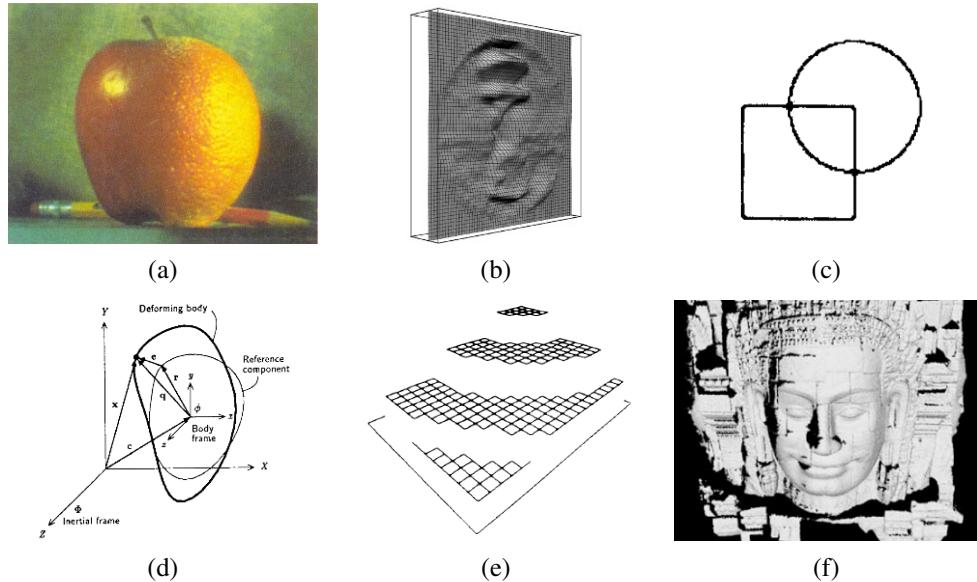


Figure 1.8 Examples of computer vision algorithms from the 1980s: (a) pyramid blending (Burt and Adelson 1983b) © 1983 ACM, (b) shape from shading (Freeman and Adelson 1991) © 1991 IEEE, (c) edge detection (Freeman and Adelson 1991) © 1991 IEEE, (d) physically based models (Terzopoulos and Witkin 1988) © 1988 IEEE, (e) regularization-based surface reconstruction (Terzopoulos 1988) © 1988 IEEE, (f) range data acquisition and merging (Banno, Masuda, Oishi *et al.* 2008) © 2008 Springer.

Research into better edge and contour detection (Figure 1.8c) (see Section 4.2) was also active during this period (Canny 1986; Nalwa and Binford 1986), including the introduction of dynamically evolving contour trackers (Section 5.1.1) such as *snakes* (Kass, Witkin, and Terzopoulos 1988), as well as three-dimensional *physically based models* (Figure 1.8d) (Terzopoulos, Witkin, and Kass 1987; Kass, Witkin, and Terzopoulos 1988; Terzopoulos and Fleischer 1988; Terzopoulos, Witkin, and Kass 1988).

Researchers noticed that a lot of the stereo, flow, shape-from-X, and edge detection algorithms could be unified, or at least described, using the same mathematical framework if they were posed as variational optimization problems (see Section 3.7) and made more robust (well-posed) using regularization (Figure 1.8e) (see Section 3.7.1 and Terzopoulos 1983; Poggio, Torre, and Koch 1985; Terzopoulos 1986b; Blake and Zisserman 1987; Bertero, Poggio, and Torre 1988; Terzopoulos 1988). Around the same time, Geman and Geman (1984) pointed out that such problems could equally well be formulated using discrete *Markov Random Field* (MRF) models (see Section 3.7.2), which enabled the use of better (global) search and optimization algorithms, such as simulated annealing.

Online variants of MRF algorithms that modeled and updated uncertainties using the Kalman filter were introduced a little later (Dickmanns and Graefe 1988; Matthies, Kanade, and Szeliski 1989; Szeliski 1989). Attempts were also made to map both regularized and MRF algorithms onto parallel hardware (Poggio and Koch 1985; Poggio, Little, Gamble *et al.* 1988; Fischler, Firschein, Barnard *et al.* 1989). The book by Fischler and Firschein (1987) contains a nice collection of articles focusing on all of these topics (stereo, flow,

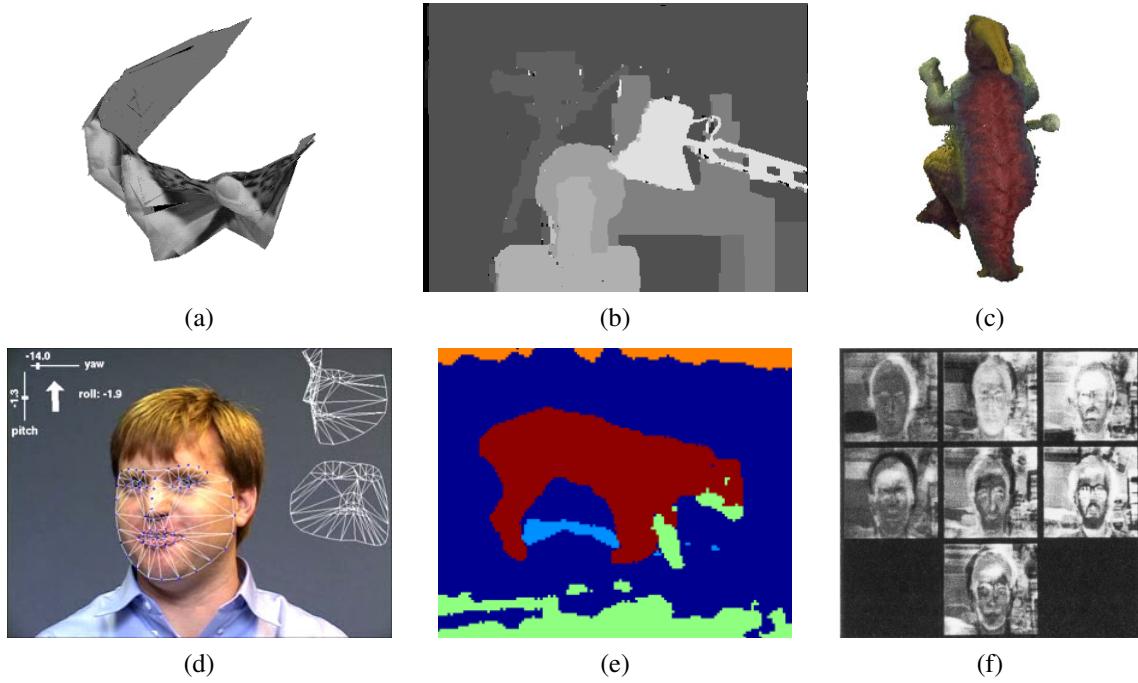


Figure 1.9 Examples of computer vision algorithms from the 1990s: (a) factorization-based structure from motion (Tomasi and Kanade 1992) © 1992 Springer, (b) dense stereo matching (Boykov, Veksler, and Zabih 2001), (c) multi-view reconstruction (Seitz and Dyer 1999) © 1999 Springer, (d) face tracking (Matthews, Xiao, and Baker 2007), (e) image segmentation (Belongie, Fowlkes, Chung *et al.* 2002) © 2002 Springer, (f) face recognition (Turk and Pentland 1991a).

regularization, MRFs, and even higher-level vision).

Three-dimensional range data processing (acquisition, merging, modeling, and recognition; see Figure 1.8f) continued being actively explored during this decade (Agin and Binford 1976; Besl and Jain 1985; Faugeras and Hebert 1987; Curless and Levoy 1996). The compilation by Kanade (1987) contains a lot of the interesting papers in this area.

1990s. While a lot of the previously mentioned topics continued to be explored, a few of them became significantly more active.

A burst of activity in using projective invariants for recognition (Mundy and Zisserman 1992) evolved into a concerted effort to solve the structure from motion problem (see Chapter 7). A lot of the initial activity was directed at *projective reconstructions*, which did not require knowledge of camera calibration (Faugeras 1992; Hartley, Gupta, and Chang 1992; Hartley 1994a; Faugeras and Luong 2001; Hartley and Zisserman 2004). Simultaneously, *factorization* techniques (Section 7.3) were developed to solve efficiently problems for which orthographic camera approximations were applicable (Figure 1.9a) (Tomasi and Kanade 1992; Poelman and Kanade 1997; Anandan and Irani 2002) and then later extended to the perspective case (Christy and Horaud 1996; Triggs 1996). Eventually, the field started using full global optimization (see Section 7.4 and Taylor, Kriegman, and Anandan 1991; Szeliski and

Kang 1994; Azarbayejani and Pentland 1995), which was later recognized as being the same as the *bundle adjustment* techniques traditionally used in photogrammetry (Triggs, McLauchlan, Hartley *et al.* 1999). Fully automated (sparse) 3D modeling systems were built using such techniques (Beardsley, Torr, and Zisserman 1996; Schaffalitzky and Zisserman 2002; Brown and Lowe 2003; Snavely, Seitz, and Szeliski 2006).

Work begun in the 1980s on using detailed measurements of color and intensity combined with accurate physical models of radiance transport and color image formation created its own subfield known as *physics-based vision*. A good survey of the field can be found in the three-volume collection on this topic (Wolff, Shafer, and Healey 1992a; Healey and Shafer 1992; Shafer, Healey, and Wolff 1992).

Optical flow methods (see Chapter 8) continued to be improved (Nagel and Enkelmann 1986; Bolles, Baker, and Marimont 1987; Horn and Weldon Jr. 1988; Anandan 1989; Bergen, Anandan, Hanna *et al.* 1992; Black and Anandan 1996; Bruhn, Weickert, and Schnörr 2005; Papenberg, Bruhn, Brox *et al.* 2006), with (Nagel 1986; Barron, Fleet, and Beauchemin 1994; Baker, Black, Lewis *et al.* 2007) being good surveys. Similarly, a lot of progress was made on dense stereo correspondence algorithms (see Chapter 11, Okutomi and Kanade (1993, 1994); Boykov, Veksler, and Zabih (1998); Birchfield and Tomasi (1999); Boykov, Veksler, and Zabih (2001), and the survey and comparison in Scharstein and Szeliski (2002)), with the biggest breakthrough being perhaps global optimization using *graph cut* techniques (Figure 1.9b) (Boykov, Veksler, and Zabih 2001).

Multi-view stereo algorithms (Figure 1.9c) that produce complete 3D surfaces (see Section 11.6) were also an active topic of research (Seitz and Dyer 1999; Kutulakos and Seitz 2000) that continues to be active today (Seitz, Curless, Diebel *et al.* 2006). Techniques for producing 3D volumetric descriptions from binary silhouettes (see Section 11.6.2) continued to be developed (Potmesil 1987; Srivasan, Liang, and Hackwood 1990; Szeliski 1993; Laurentini 1994), along with techniques based on tracking and reconstructing smooth occluding contours (see Section 11.2.1 and Cipolla and Blake 1992; Vaillant and Faugeras 1992; Zheng 1994; Boyer and Berger 1997; Szeliski and Weiss 1998; Cipolla and Giblin 2000).

Tracking algorithms also improved a lot, including contour tracking using *active contours* (see Section 5.1), such as *snakes* (Kass, Witkin, and Terzopoulos 1988), *particle filters* (Blake and Isard 1998), and *level sets* (Malladi, Sethian, and Vemuri 1995), as well as intensity-based (*direct*) techniques (Lucas and Kanade 1981; Shi and Tomasi 1994; Rehg and Kanade 1994), often applied to tracking faces (Figure 1.9d) (Lanitis, Taylor, and Cootes 1997; Matthews and Baker 2004; Matthews, Xiao, and Baker 2007) and whole bodies (Sidenbladh, Black, and Fleet 2000; Hilton, Fua, and Ronfard 2006; Moeslund, Hilton, and Krüger 2006).

Image segmentation (see Chapter 5) (Figure 1.9e), a topic which has been active since the earliest days of computer vision (Brice and Fennema 1970; Horowitz and Pavlidis 1976; Riseman and Arbib 1977; Rosenfeld and Davis 1979; Haralick and Shapiro 1985; Pavlidis and Liow 1990), was also an active topic of research, producing techniques based on minimum energy (Mumford and Shah 1989) and minimum description length (Leclerc 1989), *normalized cuts* (Shi and Malik 2000), and *mean shift* (Comaniciu and Meer 2002).

Statistical learning techniques started appearing, first in the application of principal component *eigenface* analysis to face recognition (Figure 1.9f) (see Section 14.2.1 and Turk and Pentland 1991a) and linear dynamical systems for curve tracking (see Section 5.1.1 and Blake and Isard 1998).

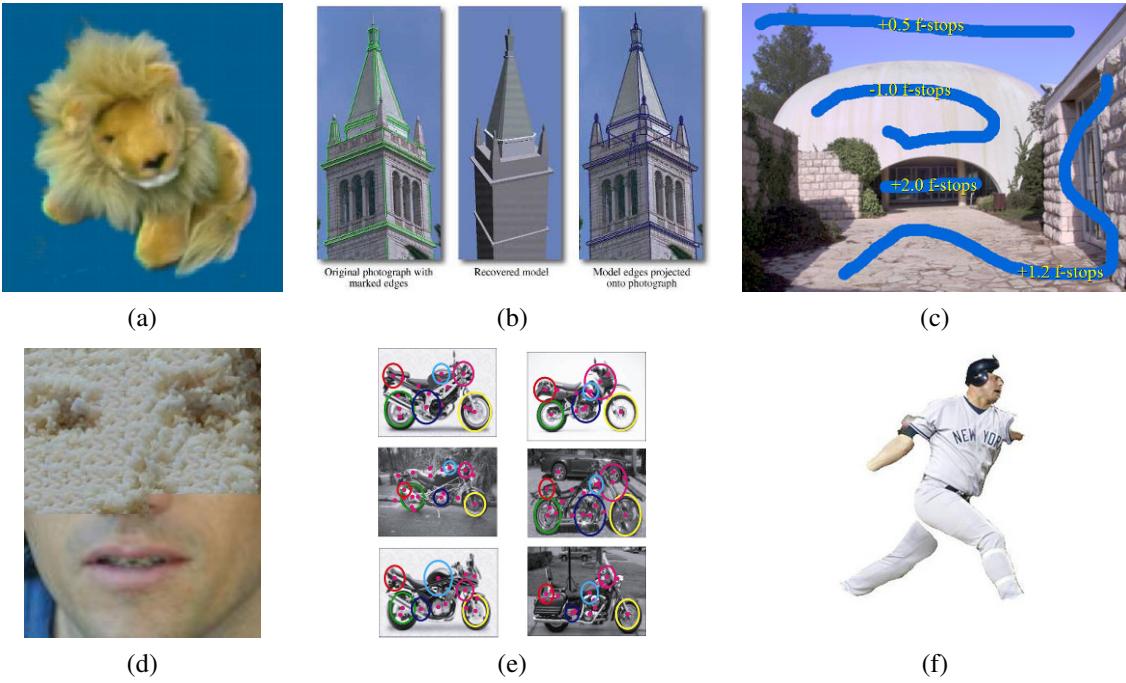


Figure 1.10 Recent examples of computer vision algorithms: (a) image-based rendering (Gortler, Grzeszczuk, Szeliski *et al.* 1996), (b) image-based modeling (Debevec, Taylor, and Malik 1996) © 1996 ACM, (c) interactive tone mapping (Lischinski, Farbman, Uyttendaele *et al.* 2006a) (d) texture synthesis (Efros and Freeman 2001), (e) feature-based recognition (Fergus, Perona, and Zisserman 2007), (f) region-based recognition (Mori, Ren, Efros *et al.* 2004) © 2004 IEEE.

Perhaps the most notable development in computer vision during this decade was the increased interaction with computer graphics (Seitz and Szeliski 1999), especially in the cross-disciplinary area of *image-based modeling and rendering* (see Chapter 13). The idea of manipulating real-world imagery directly to create new animations first came to prominence with *image morphing* techniques (Figure 1.5c) (see Section 3.6.3 and Beier and Neely 1992) and was later applied to *view interpolation* (Chen and Williams 1993; Seitz and Dyer 1996), panoramic image stitching (Figure 1.5a) (see Chapter 9 and Mann and Picard 1994; Chen 1995; Szeliski 1996; Szeliski and Shum 1997; Szeliski 2006a), and full light-field rendering (Figure 1.10a) (see Section 13.3 and Gortler, Grzeszczuk, Szeliski *et al.* 1996; Levoy and Hanrahan 1996; Shade, Gortler, He *et al.* 1998). At the same time, image-based modeling techniques (Figure 1.10b) for automatically creating realistic 3D models from collections of images were also being introduced (Beardsley, Torr, and Zisserman 1996; Debevec, Taylor, and Malik 1996; Taylor, Debevec, and Malik 1996).

2000s. This past decade has continued to see a deepening interplay between the vision and graphics fields. In particular, many of the topics introduced under the rubric of image-based rendering, such as image stitching (see Chapter 9), light-field capture and rendering (see Section 13.3), and *high dynamic range* (HDR) image capture through exposure bracketing (Figure 1.5b) (see Section 10.2 and Mann and Picard 1995; Debevec and Malik 1997), were

rechristened as *computational photography* (see Chapter 10) to acknowledge the increased use of such techniques in everyday digital photography. For example, the rapid adoption of exposure bracketing to create high dynamic range images necessitated the development of *tone mapping* algorithms (Figure 1.10c) (see Section 10.2.1) to convert such images back to displayable results (Fattal, Lischinski, and Werman 2002; Durand and Dorsey 2002; Reinhard, Stark, Shirley *et al.* 2002; Lischinski, Farbman, Uyttendaele *et al.* 2006a). In addition to merging multiple exposures, techniques were developed to merge flash images with non-flash counterparts (Eisemann and Durand 2004; Petschnigg, Agrawala, Hoppe *et al.* 2004) and to interactively or automatically select different regions from overlapping images (Agarwala, Dontcheva, Agrawala *et al.* 2004).

Texture synthesis (Figure 1.10d) (see Section 10.5), quilting (Efros and Leung 1999; Efros and Freeman 2001; Kwatra, Schödl, Essa *et al.* 2003) and inpainting (Bertalmio, Sapiro, Caselles *et al.* 2000; Bertalmio, Vese, Sapiro *et al.* 2003; Criminisi, Pérez, and Toyama 2004) are additional topics that can be classified as computational photography techniques, since they re-combine input image samples to produce new photographs.

A second notable trend during this past decade has been the emergence of feature-based techniques (combined with learning) for object recognition (see Section 14.3 and Ponce, Hebert, Schmid *et al.* 2006). Some of the notable papers in this area include the *constellation model* of Fergus, Perona, and Zisserman (2007) (Figure 1.10e) and the *pictorial structures* of Felzenszwalb and Huttenlocher (2005). Feature-based techniques also dominate other recognition tasks, such as scene recognition (Zhang, Marszalek, Lazebnik *et al.* 2007) and panorama and location recognition (Brown and Lowe 2007; Schindler, Brown, and Szeliski 2007). And while *interest point* (patch-based) features tend to dominate current research, some groups are pursuing recognition based on contours (Belongie, Malik, and Puzicha 2002) and region segmentation (Figure 1.10f) (Mori, Ren, Efros *et al.* 2004).

Another significant trend from this past decade has been the development of more efficient algorithms for complex global optimization problems (see Sections 3.7 and B.5 and Szeliski, Zabih, Scharstein *et al.* 2008; Blake, Kohli, and Rother 2010). While this trend began with work on graph cuts (Boykov, Veksler, and Zabih 2001; Kohli and Torr 2007), a lot of progress has also been made in message passing algorithms, such as *loopy belief propagation* (LBP) (Yedidia, Freeman, and Weiss 2001; Kumar and Torr 2006).

The final trend, which now dominates a lot of the visual recognition research in our community, is the application of sophisticated machine learning techniques to computer vision problems (see Section 14.5.1 and Freeman, Perona, and Schölkopf 2008). This trend coincides with the increased availability of immense quantities of partially labelled data on the Internet, which makes it more feasible to learn object categories without the use of careful human supervision.

1.3 Book overview

In the final part of this introduction, I give a brief tour of the material in this book, as well as a few notes on notation and some additional general references. Since computer vision is such a broad field, it is possible to study certain aspects of it, e.g., geometric image formation and 3D structure recovery, without engaging other parts, e.g., the modeling of reflectance and shading. Some of the chapters in this book are only loosely coupled with others, and it is not

strictly necessary to read all of the material in sequence.

Figure 1.11 shows a rough layout of the contents of this book. Since computer vision involves going from images to a structural description of the scene (and computer graphics the converse), I have positioned the chapters horizontally in terms of which major component they address, in addition to vertically according to their dependence.

Going from left to right, we see the major column headings as Images (which are 2D in nature), Geometry (which encompasses 3D descriptions), and Photometry (which encompasses object appearance). (An alternative labeling for these latter two could also be *shape* and *appearance*—see, e.g., Chapter 13 and Kang, Szeliski, and Anandan (2000).) Going from top to bottom, we see increasing levels of modeling and abstraction, as well as techniques that build on previously developed algorithms. Of course, this taxonomy should be taken with a large grain of salt, as the processing and dependencies in this diagram are not strictly sequential and subtle additional dependencies and relationships also exist (e.g., some recognition techniques make use of 3D information). The placement of topics along the horizontal axis should also be taken lightly, as most vision algorithms involve mapping between at least two different representations.⁹

Interspersed throughout the book are sample **applications**, which relate the algorithms and mathematical material being presented in various chapters to useful, real-world applications. Many of these applications are also presented in the exercises sections, so that students can write their own.

At the end of each section, I provide a set of **exercises** that the students can use to implement, test, and refine the algorithms and techniques presented in each section. Some of the exercises are suitable as written homework assignments, others as shorter one-week projects, and still others as open-ended research problems that make for challenging final projects. Motivated students who implement a reasonable subset of these exercises will, by the end of the book, have a computer vision software library that can be used for a variety of interesting tasks and projects.

As a reference book, I try wherever possible to discuss which techniques and algorithms work well in practice, as well as providing up-to-date pointers to the latest research results in the areas that I cover. The exercises can be used to build up your own personal library of self-tested and validated vision algorithms, which is more worthwhile in the long term (assuming you have the time) than simply pulling algorithms out of a library whose performance you do not really understand.

The book begins in Chapter 2 with a review of the image formation processes that create the images that we see and capture. Understanding this process is fundamental if you want to take a scientific (model-based) approach to computer vision. Students who are eager to just start implementing algorithms (or courses that have limited time) can skip ahead to the next chapter and dip into this material later. In Chapter 2, we break down image formation into three major components. Geometric image formation (Section 2.1) deals with points, lines, and planes, and how these are mapped onto images using *projective geometry* and other models (including radial lens distortion). Photometric image formation (Section 2.2) covers *radiometry*, which describes how light interacts with surfaces in the world, and *optics*, which projects light onto the sensor plane. Finally, Section 2.3 covers how sensors work, including

⁹ For an interesting comparison with what is known about the human visual system, e.g., the largely parallel *what* and *where* pathways, see some textbooks on human perception (Palmer 1999; Livingstone 2008).

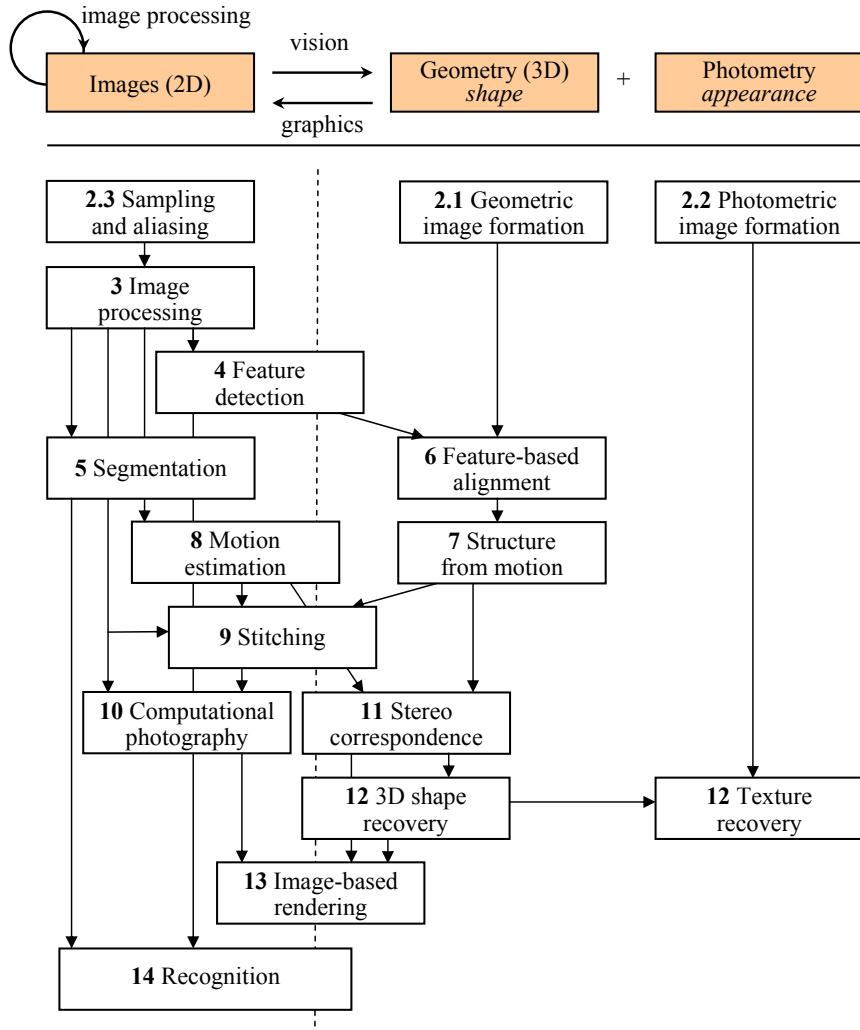


Figure 1.11 Relationship between images, geometry, and photometry, as well as a taxonomy of the topics covered in this book. Topics are roughly positioned along the left-right axis depending on whether they are more closely related to image-based (left), geometry-based (middle) or appearance-based (right) representations, and on the vertical axis by increasing level of abstraction. The whole figure should be taken with a large grain of salt, as there are many additional subtle connections between topics not illustrated here.

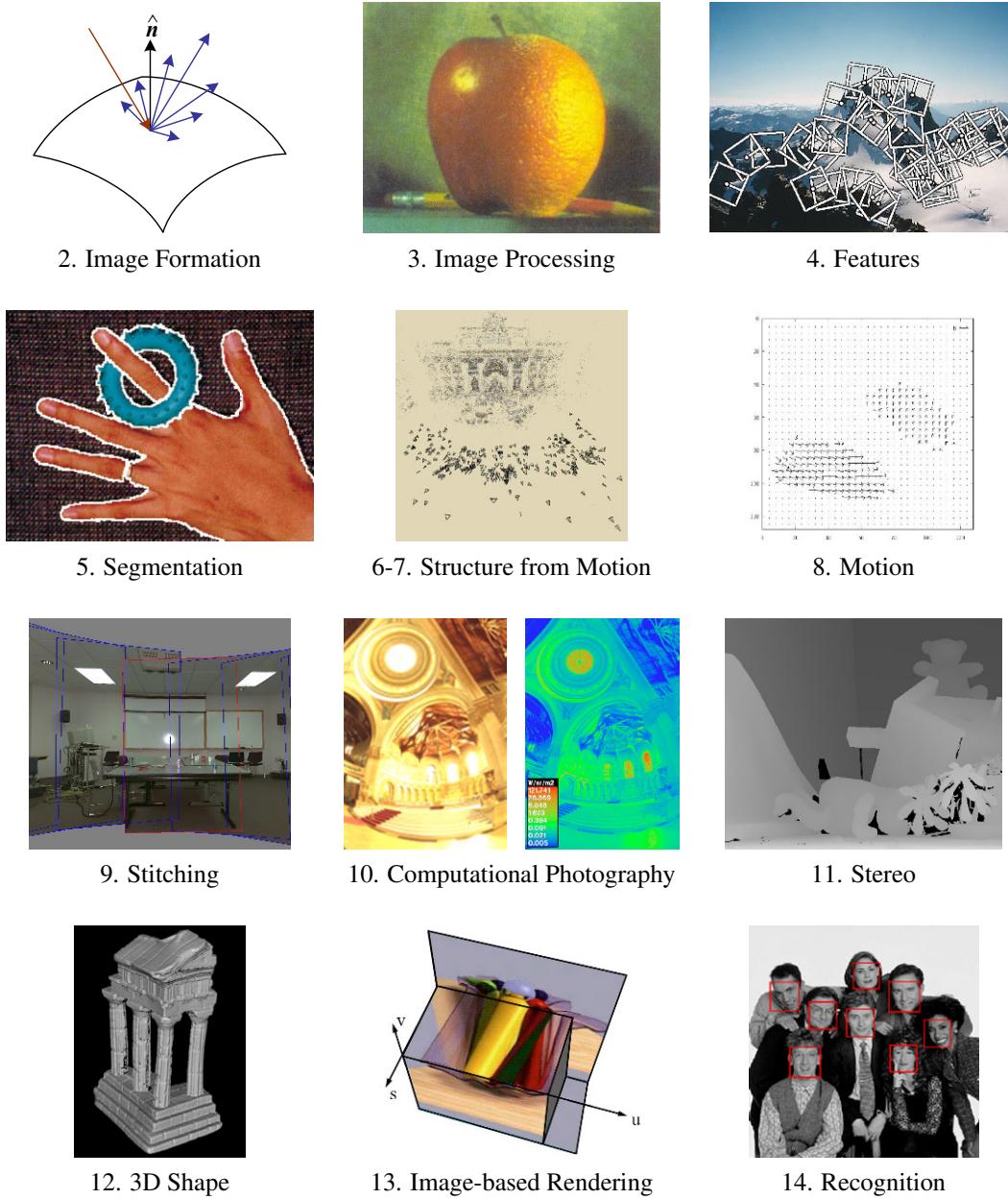


Figure 1.12 A pictorial summary of the chapter contents. Sources: Brown, Szeliski, and Winder (2005); Comaniciu and Meer (2002); Snavely, Seitz, and Szeliski (2006); Nagel and Enkelmann (1986); Szeliski and Shum (1997); Debevec and Malik (1997); Gortler, Grzeszczuk, Szeliski *et al.* (1996); Viola and Jones (2004)—see the figures in the respective chapters for copyright information.

topics such as sampling and aliasing, color sensing, and in-camera compression.

Chapter 3 covers image processing, which is needed in almost all computer vision applications. This includes topics such as linear and non-linear filtering (Section 3.3), the Fourier transform (Section 3.4), image pyramids and wavelets (Section 3.5), geometric transformations such as image warping (Section 3.6), and global optimization techniques such as *regularization* and *Markov Random Fields* (MRFs) (Section 3.7). While most of this material is covered in courses and textbooks on image processing, the use of optimization techniques is more typically associated with computer vision (although MRFs are now being widely used in image processing as well). The section on MRFs is also the first introduction to the use of Bayesian inference techniques, which are covered at a more abstract level in Appendix B. Chapter 3 also presents applications such as seamless image blending and image restoration.

In Chapter 4, we cover feature detection and matching. A lot of current 3D reconstruction and recognition techniques are built on extracting and matching *feature points* (Section 4.1), so this is a fundamental technique required by many subsequent chapters (Chapters 6, 7, 9 and 14). We also cover edge and straight line detection in Sections 4.2 and 4.3.

Chapter 5 covers region segmentation techniques, including active contour detection and tracking (Section 5.1). Segmentation techniques include top-down (split) and bottom-up (merge) techniques, mean shift techniques that find modes of clusters, and various graph-based segmentation approaches. All of these techniques are essential building blocks that are widely used in a variety of applications, including performance-driven animation, interactive image editing, and recognition.

In Chapter 6, we cover geometric alignment and camera calibration. We introduce the basic techniques of feature-based alignment in Section 6.1 and show how this problem can be solved using either linear or non-linear least squares, depending on the motion involved. We also introduce additional concepts, such as uncertainty weighting and robust regression, which are essential to making real-world systems work. Feature-based alignment is then used as a building block for 3D pose estimation (*extrinsic calibration*) in Section 6.2 and camera (*intrinsic*) calibration in Section 6.3. Chapter 6 also describes applications of these techniques to photo alignment for flip-book animations, 3D pose estimation from a hand-held camera, and single-view reconstruction of building models.

Chapter 7 covers the topic of *structure from motion*, which involves the simultaneous recovery of 3D camera motion and 3D scene structure from a collection of tracked 2D features. This chapter begins with the easier problem of 3D point *triangulation* (Section 7.1), which is the 3D reconstruction of points from matched features when the camera positions are known. It then describes two-frame structure from motion (Section 7.2), for which algebraic techniques exist, as well as robust sampling techniques such as RANSAC that can discount erroneous feature matches. The second half of Chapter 7 describes techniques for multi-frame structure from motion, including factorization (Section 7.3), bundle adjustment (Section 7.4), and constrained motion and structure models (Section 7.5). It also presents applications in view morphing, sparse 3D model construction, and match move.

In Chapter 8, we go back to a topic that deals directly with image intensities (as opposed to feature tracks), namely dense intensity-based motion estimation (*optical flow*). We start with the simplest possible motion models, translational motion (Section 8.1), and cover topics such as hierarchical (coarse-to-fine) motion estimation, Fourier-based techniques, and iterative refinement. We then present parametric motion models, which can be used to com-

pensate for camera rotation and zooming, as well as affine or planar perspective motion (Section 8.2). This is then generalized to spline-based motion models (Section 8.3) and finally to general per-pixel optical flow (Section 8.4), including layered and learned motion models (Section 8.5). Applications of these techniques include automated morphing, frame interpolation (slow motion), and motion-based user interfaces.

Chapter 9 is devoted to *image stitching*, i.e., the construction of large panoramas and composites. While stitching is just one example of *computation photography* (see Chapter 10), there is enough depth here to warrant a separate chapter. We start by discussing various possible motion models (Section 9.1), including planar motion and pure camera rotation. We then discuss global alignment (Section 9.2), which is a special (simplified) case of general bundle adjustment, and then present *panorama recognition*, i.e., techniques for automatically discovering which images actually form overlapping panoramas. Finally, we cover the topics of *image compositing* and *blending* (Section 9.3), which involve both selecting which pixels from which images to use and blending them together so as to disguise exposure differences.

Image stitching is a wonderful application that ties together most of the material covered in earlier parts of this book. It also makes for a good mid-term course project that can build on previously developed techniques such as image warping and feature detection and matching. Chapter 9 also presents more specialized variants of stitching such as whiteboard and document scanning, video summarization, *panography*, full 360° spherical panoramas, and interactive photomontage for blending repeated action shots together.

Chapter 10 presents additional examples of *computational photography*, which is the process of creating new images from one or more input photographs, often based on the careful modeling and calibration of the image formation process (Section 10.1). Computational photography techniques include merging multiple exposures to create *high dynamic range* images (Section 10.2), increasing image resolution through blur removal and *super-resolution* (Section 10.3), and image editing and compositing operations (Section 10.4). We also cover the topics of texture analysis, synthesis and *inpainting* (hole filling) in Section 10.5, as well as non-photorealistic rendering (Section 10.5.2).

In Chapter 11, we turn to the issue of stereo correspondence, which can be thought of as a special case of motion estimation where the camera positions are already known (Section 11.1). This additional knowledge enables stereo algorithms to search over a much smaller space of correspondences and, in many cases, to produce dense depth estimates that can be converted into visible surface models (Section 11.3). We also cover multi-view stereo algorithms that build a true 3D surface representation instead of just a single depth map (Section 11.6). Applications of stereo matching include head and gaze tracking, as well as depth-based background replacement (*Z-keying*).

Chapter 12 covers additional 3D shape and appearance modeling techniques. These include classic *shape-from-X* techniques such as shape from shading, shape from texture, and shape from focus (Section 12.1), as well as shape from smooth occluding contours (Section 11.2.1) and silhouettes (Section 12.5). An alternative to all of these *passive* computer vision techniques is to use *active rangefinding* (Section 12.2), i.e., to project patterned light onto scenes and recover the 3D geometry through triangulation. Processing all of these 3D representations often involves interpolating or simplifying the geometry (Section 12.3), or using alternative representations such as surface point sets (Section 12.4).

The collection of techniques for going from one or more images to partial or full 3D

models is often called *image-based modeling* or *3D photography*. Section 12.6 examines three more specialized application areas (architecture, faces, and human bodies), which can use *model-based reconstruction* to fit parameterized models to the sensed data. Section 12.7 examines the topic of *appearance modeling*, i.e., techniques for estimating the texture maps, albedos, or even sometimes complete *bi-directional reflectance distribution functions* (BRDFs) that describe the appearance of 3D surfaces.

In Chapter 13, we discuss the large number of image-based rendering techniques that have been developed in the last two decades, including simpler techniques such as view interpolation (Section 13.1), layered depth images (Section 13.2), and sprites and layers (Section 13.2.1), as well as the more general framework of light fields and Lumigraphs (Section 13.3) and higher-order fields such as environment mattes (Section 13.4). Applications of these techniques include navigating 3D collections of photographs using *photo tourism* and viewing 3D models as *object movies*.

In Chapter 13, we also discuss video-based rendering, which is the temporal extension of image-based rendering. The topics we cover include video-based animation (Section 13.5.1), periodic video turned into *video textures* (Section 13.5.2), and 3D video constructed from multiple video streams (Section 13.5.4). Applications of these techniques include video denoising, morphing, and tours based on 360° video.

Chapter 14 describes different approaches to recognition. It begins with techniques for detecting and recognizing faces (Sections 14.1 and 14.2), then looks at techniques for finding and recognizing particular objects (*instance recognition*) in Section 14.3. Next, we cover the most difficult variant of recognition, namely the recognition of broad *categories*, such as cars, motorcycles, horses and other animals (Section 14.4), and the role that scene context plays in recognition (Section 14.5).

To support the book's use as a textbook, the appendices and associated Web site contain more detailed mathematical topics and additional material. Appendix A covers linear algebra and numerical techniques, including matrix algebra, least squares, and iterative techniques. Appendix B covers Bayesian estimation theory, including maximum likelihood estimation, robust statistics, Markov random fields, and uncertainty modeling. Appendix C describes the supplementary material available to complement this book, including images and data sets, pointers to software, course slides, and an on-line bibliography.

1.4 Sample syllabus

Teaching all of the material covered in this book in a single quarter or semester course is a Herculean task and likely one not worth attempting. It is better to simply pick and choose topics related to the lecturer's preferred emphasis and tailored to the set of mini-projects envisioned for the students.

Steve Seitz and I have successfully used a 10-week syllabus similar to the one shown in Table 1.1 (omitting the parenthesized weeks) as both an undergraduate and a graduate-level course in computer vision. The undergraduate course¹⁰ tends to go lighter on the mathematics and takes more time reviewing basics, while the graduate-level course¹¹ dives more deeply into techniques and assumes the students already have a decent grounding in either vision

¹⁰ <http://www.cs.washington.edu/education/courses/455/>

¹¹ <http://www.cs.washington.edu/education/courses/576/>

Week	Material	Project
(1.)	Chapter 2 Image formation	
2.	Chapter 3 Image processing	
3.	Chapter 4 Feature detection and matching	P1
4.	Chapter 6 Feature-based alignment	
5.	Chapter 9 Image stitching	P2
6.	Chapter 8 Dense motion estimation	
7.	Chapter 7 Structure from motion	PP
8.	Chapter 14 Recognition	
(9.)	Chapter 10 Computational photography	
10.	Chapter 11 Stereo correspondence	
(11.)	Chapter 12 3D reconstruction	
12.	Chapter 13 Image-based rendering	
13.	Final project presentations	FP

Table 1.1 Sample syllabi for 10-week and 13-week courses. The weeks in parentheses are not used in the shorter version. P1 and P2 are two early-term mini-projects, PP is when the (student-selected) final project proposals are due, and FP is the final project presentations.

or related mathematical techniques. (See also the *Introduction to Computer Vision* course at Stanford,¹² which uses a similar curriculum.) Related courses have also been taught on the topics of 3D photography¹³ and computational photography.¹⁴

When Steve and I teach the course, we prefer to give the students several small programming projects early in the course rather than focusing on written homework or quizzes. With a suitable choice of topics, it is possible for these projects to build on each other. For example, introducing feature matching early on can be used in a second assignment to do image alignment and stitching. Alternatively, direct (optical flow) techniques can be used to do the alignment and more focus can be put on either graph cut seam selection or multi-resolution blending techniques.

We also ask the students to propose a final project (we provide a set of suggested topics for those who need ideas) by the middle of the course and reserve the last week of the class for student presentations. With any luck, some of these final projects can actually turn into conference submissions!

No matter how you decide to structure the course or how you choose to use this book, I encourage you to try at least a few small programming tasks to get a good feel for how vision techniques work, and when they do not. Better yet, pick topics that are fun and can be used on your own photographs, and try to push your creative boundaries to come up with surprising results.

¹² <http://vision.stanford.edu/teaching/cs223b/>

¹³ <http://www.cs.washington.edu/education/courses/558/06sp/>

¹⁴ <http://graphics.cs.cmu.edu/courses/15-463/>

1.5 A note on notation

For better or worse, the notation found in computer vision and multi-view geometry textbooks tends to vary all over the map (Faugeras 1993; Hartley and Zisserman 2004; Girod, Greiner, and Niemann 2000; Faugeras and Luong 2001; Forsyth and Ponce 2003). In this book, I use the convention I first learned in my high school physics class (and later multi-variate calculus and computer graphics courses), which is that vectors v are lower case bold, matrices M are upper case bold, and scalars (T, s) are mixed case italic. Unless otherwise noted, vectors operate as column vectors, i.e., they post-multiply matrices, Mv , although they are sometimes written as comma-separated parenthesized lists $\mathbf{x} = (x, y)$ instead of bracketed column vectors $\mathbf{x} = [x \ y]^T$. Some commonly used matrices are R for rotations, K for calibration matrices, and I for the identity matrix. Homogeneous coordinates (Section 2.1) are denoted with a tilde over the vector, e.g., $\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{w}) = \tilde{w}(x, y, 1) = \tilde{w}\bar{\mathbf{x}}$ in \mathcal{P}^2 . The cross product operator in matrix form is denoted by $[\]_\times$.

1.6 Additional reading

This book attempts to be self-contained, so that students can implement the basic assignments and algorithms described here without the need for outside references. However, it does presuppose a general familiarity with basic concepts in linear algebra and numerical techniques, which are reviewed in Appendix A, and image processing, which is reviewed in Chapter 3.

Students who want to delve more deeply into these topics can look in (Golub and Van Loan 1996) for matrix algebra and (Strang 1988) for linear algebra. In image processing, there are a number of popular textbooks, including (Crane 1997; Gomes and Velho 1997; Jähne 1997; Pratt 2007; Russ 2007; Burger and Burge 2008; Gonzales and Woods 2008). For computer graphics, popular texts include (Foley, van Dam, Feiner *et al.* 1995; Watt 1995), with (Glassner 1995) providing a more in-depth look at image formation and rendering. For statistics and machine learning, Chris Bishop's (2006) book is a wonderful and comprehensive introduction with a wealth of exercises. Students may also want to look in other textbooks on computer vision for material that we do not cover here, as well as for additional project ideas (Ballard and Brown 1982; Faugeras 1993; Nalwa 1993; Trucco and Verri 1998; Forsyth and Ponce 2003).

There is, however, no substitute for reading the latest research literature, both for the latest ideas and techniques and for the most up-to-date references to related literature.¹⁵ In this book, I have attempted to cite the most recent work in each field so that students can read them directly and use them as inspiration for their own work. Browsing the last few years' conference proceedings from the major vision and graphics conferences, such as CVPR, ECCV, ICCV, and SIGGRAPH, will provide a wealth of new ideas. The tutorials offered at these conferences, for which slides or notes are often available on-line, are also an invaluable resource.

¹⁵ For a comprehensive bibliography and taxonomy of computer vision research, Keith Price's Annotated Computer Vision Bibliography <http://www.visionbib.com/bibliography/contents.html> is an invaluable resource.

Chapter 2

Image formation

2.1	Geometric primitives and transformations	29
2.1.1	Geometric primitives	29
2.1.2	2D transformations	33
2.1.3	3D transformations	36
2.1.4	3D rotations	37
2.1.5	3D to 2D projections	42
2.1.6	Lens distortions	52
2.2	Photometric image formation	54
2.2.1	Lighting	54
2.2.2	Reflectance and shading	55
2.2.3	Optics	61
2.3	The digital camera	65
2.3.1	Sampling and aliasing	69
2.3.2	Color	71
2.3.3	Compression	80
2.4	Additional reading	82
2.5	Exercises	82

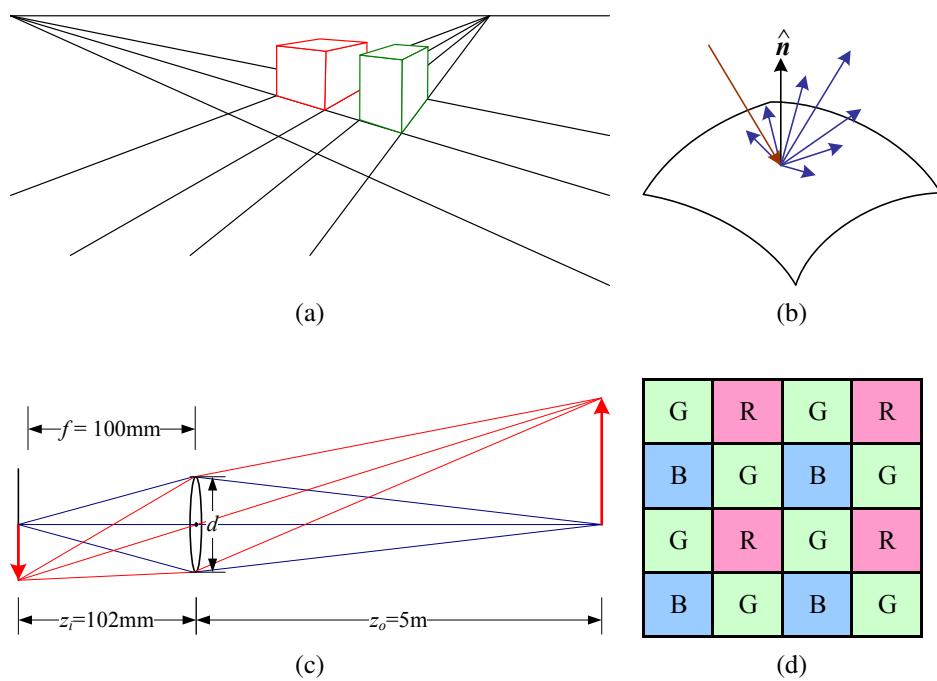


Figure 2.1 A few components of the image formation process: (a) perspective projection; (b) light scattering when hitting a surface; (c) lens optics; (d) Bayer color filter array.

Before we can intelligently analyze and manipulate images, we need to establish a vocabulary for describing the geometry of a scene. We also need to understand the image formation process that produced a particular image given a set of lighting conditions, scene geometry, surface properties, and camera optics. In this chapter, we present a simplified model of such an image formation process.

Section 2.1 introduces the basic geometric primitives used throughout the book (points, lines, and planes) and the *geometric* transformations that project these 3D quantities into 2D image features (Figure 2.1a). Section 2.2 describes how lighting, surface properties (Figure 2.1b), and camera *optics* (Figure 2.1c) interact in order to produce the color values that fall onto the image sensor. Section 2.3 describes how continuous color images are turned into discrete digital *samples* inside the image sensor (Figure 2.1d) and how to avoid (or at least characterize) sampling deficiencies, such as aliasing.

The material covered in this chapter is but a brief summary of a very rich and deep set of topics, traditionally covered in a number of separate fields. A more thorough introduction to the geometry of points, lines, planes, and projections can be found in textbooks on multi-view geometry (Hartley and Zisserman 2004; Faugeras and Luong 2001) and computer graphics (Foley, van Dam, Feiner *et al.* 1995). The image formation (synthesis) process is traditionally taught as part of a computer graphics curriculum (Foley, van Dam, Feiner *et al.* 1995; Glassner 1995; Watt 1995; Shirley 2005) but it is also studied in physics-based computer vision (Wolff, Shafer, and Healey 1992a). The behavior of camera lens systems is studied in optics (Möller 1988; Hecht 2001; Ray 2002). Two good books on color theory are (Wyszecki and Stiles 2000; Healey and Shafer 1992), with (Livingstone 2008) providing a more fun and informal introduction to the topic of color perception. Topics relating to sampling and aliasing are covered in textbooks on signal and image processing (Crane 1997; Jähne 1997; Oppenheim and Schafer 1996; Oppenheim, Schafer, and Buck 1999; Pratt 2007; Russ 2007; Burger and Burge 2008; Gonzales and Woods 2008).

A note to students: If you have already studied computer graphics, you may want to skim the material in Section 2.1, although the sections on projective depth and object-centered projection near the end of Section 2.1.5 may be new to you. Similarly, physics students (as well as computer graphics students) will mostly be familiar with Section 2.2. Finally, students with a good background in image processing will already be familiar with sampling issues (Section 2.3) as well as some of the material in Chapter 3.

2.1 Geometric primitives and transformations

In this section, we introduce the basic 2D and 3D primitives used in this textbook, namely points, lines, and planes. We also describe how 3D features are projected into 2D features. More detailed descriptions of these topics (along with a gentler and more intuitive introduction) can be found in textbooks on multiple-view geometry (Hartley and Zisserman 2004; Faugeras and Luong 2001).

2.1.1 Geometric primitives

Geometric primitives form the basic building blocks used to describe three-dimensional shapes. In this section, we introduce points, lines, and planes. Later sections of the book discuss

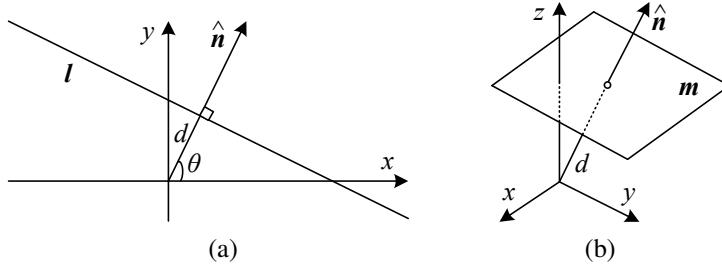


Figure 2.2 (a) 2D line equation and (b) 3D plane equation, expressed in terms of the normal $\hat{\mathbf{n}}$ and distance to the origin d .

curves (Sections 5.1 and 11.2), surfaces (Section 12.3), and volumes (Section 12.5).

2D points. 2D points (pixel coordinates in an image) can be denoted using a pair of values, $\mathbf{x} = (x, y) \in \mathcal{R}^2$, or alternatively,

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}. \quad (2.1)$$

(As stated in the introduction, we use the (x_1, x_2, \dots) notation to denote column vectors.)

2D points can also be represented using *homogeneous coordinates*, $\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{w}) \in \mathcal{P}^2$, where vectors that differ only by scale are considered to be equivalent. $\mathcal{P}^2 = \mathcal{R}^3 - (0, 0, 0)$ is called the 2D *projective space*.

A homogeneous vector $\tilde{\mathbf{x}}$ can be converted back into an *inhomogeneous* vector \mathbf{x} by dividing through by the last element \tilde{w} , i.e.,

$$\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{w}) = \tilde{w}(x, y, 1) = \tilde{w}\bar{\mathbf{x}}, \quad (2.2)$$

where $\bar{\mathbf{x}} = (x, y, 1)$ is the *augmented vector*. Homogeneous points whose last element is $\tilde{w} = 0$ are called *ideal points* or *points at infinity* and do not have an equivalent inhomogeneous representation.

2D lines. 2D lines can also be represented using homogeneous coordinates $\tilde{\mathbf{l}} = (a, b, c)$. The corresponding *line equation* is

$$\bar{\mathbf{x}} \cdot \tilde{\mathbf{l}} = ax + by + c = 0. \quad (2.3)$$

We can normalize the line equation vector so that $\mathbf{l} = (\hat{n}_x, \hat{n}_y, d) = (\hat{\mathbf{n}}, d)$ with $\|\hat{\mathbf{n}}\| = 1$. In this case, $\hat{\mathbf{n}}$ is the *normal vector* perpendicular to the line and d is its distance to the origin (Figure 2.2). (The one exception to this normalization is the *line at infinity* $\tilde{\mathbf{l}} = (0, 0, 1)$, which includes all (ideal) points at infinity.)

We can also express $\hat{\mathbf{n}}$ as a function of rotation angle θ , $\hat{\mathbf{n}} = (\hat{n}_x, \hat{n}_y) = (\cos \theta, \sin \theta)$ (Figure 2.2a). This representation is commonly used in the *Hough transform* line-finding algorithm, which is discussed in Section 4.3.2. The combination (θ, d) is also known as *polar coordinates*.

When using homogeneous coordinates, we can compute the intersection of two lines as

$$\tilde{\mathbf{x}} = \tilde{\mathbf{l}}_1 \times \tilde{\mathbf{l}}_2, \quad (2.4)$$

where \times is the cross product operator. Similarly, the line joining two points can be written as

$$\tilde{l} = \tilde{x}_1 \times \tilde{x}_2. \quad (2.5)$$

When trying to fit an intersection point to multiple lines or, conversely, a line to multiple points, least squares techniques (Section 6.1.1 and Appendix A.2) can be used, as discussed in Exercise 2.1.

2D conics. There are other algebraic curves that can be expressed with simple polynomial homogeneous equations. For example, the *conic sections* (so called because they arise as the intersection of a plane and a 3D cone) can be written using a *quadric* equation

$$\tilde{x}^T Q \tilde{x} = 0. \quad (2.6)$$

Quadric equations play useful roles in the study of multi-view geometry and camera calibration (Hartley and Zisserman 2004; Faugeras and Luong 2001) but are not used extensively in this book.

3D points. Point coordinates in three dimensions can be written using inhomogeneous coordinates $x = (x, y, z) \in \mathcal{R}^3$ or homogeneous coordinates $\tilde{x} = (\tilde{x}, \tilde{y}, \tilde{z}, \tilde{w}) \in \mathcal{P}^3$. As before, it is sometimes useful to denote a 3D point using the augmented vector $\bar{x} = (x, y, z, 1)$ with $\tilde{x} = \tilde{w}\bar{x}$.

3D planes. 3D planes can also be represented as homogeneous coordinates $\tilde{m} = (a, b, c, d)$ with a corresponding plane equation

$$\bar{x} \cdot \tilde{m} = ax + by + cz + d = 0. \quad (2.7)$$

We can also normalize the plane equation as $m = (\hat{n}_x, \hat{n}_y, \hat{n}_z, d) = (\hat{n}, d)$ with $\|\hat{n}\| = 1$. In this case, \hat{n} is the *normal vector* perpendicular to the plane and d is its distance to the origin (Figure 2.2b). As with the case of 2D lines, the *plane at infinity* $\tilde{m} = (0, 0, 0, 1)$, which contains all the points at infinity, cannot be normalized (i.e., it does not have a unique normal or a finite distance).

We can express \hat{n} as a function of two angles (θ, ϕ) ,

$$\hat{n} = (\cos \theta \cos \phi, \sin \theta \cos \phi, \sin \phi), \quad (2.8)$$

i.e., using *spherical coordinates*, but these are less commonly used than polar coordinates since they do not uniformly sample the space of possible normal vectors.

3D lines. Lines in 3D are less elegant than either lines in 2D or planes in 3D. One possible representation is to use two points on the line, (p, q) . Any other point on the line can be expressed as a linear combination of these two points

$$r = (1 - \lambda)p + \lambda q, \quad (2.9)$$

as shown in Figure 2.3. If we restrict $0 \leq \lambda \leq 1$, we get the *line segment* joining p and q .

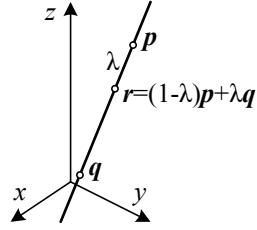


Figure 2.3 3D line equation, $r = (1 - \lambda)\mathbf{p} + \lambda\mathbf{q}$.

If we use homogeneous coordinates, we can write the line as

$$\tilde{\mathbf{r}} = \mu\tilde{\mathbf{p}} + \lambda\tilde{\mathbf{q}}. \quad (2.10)$$

A special case of this is when the second point is at infinity, i.e., $\tilde{\mathbf{q}} = (\hat{d}_x, \hat{d}_y, \hat{d}_z, 0) = (\hat{\mathbf{d}}, 0)$. Here, we see that $\hat{\mathbf{d}}$ is the *direction* of the line. We can then re-write the inhomogeneous 3D line equation as

$$\mathbf{r} = \mathbf{p} + \lambda\hat{\mathbf{d}}. \quad (2.11)$$

A disadvantage of the endpoint representation for 3D lines is that it has too many degrees of freedom, i.e., six (three for each endpoint) instead of the four degrees that a 3D line truly has. However, if we fix the two points on the line to lie in specific planes, we obtain a representation with four degrees of freedom. For example, if we are representing nearly vertical lines, then $z = 0$ and $z = 1$ form two suitable planes, i.e., the (x, y) coordinates in both planes provide the four coordinates describing the line. This kind of two-plane parameterization is used in the *light field* and *Lumigraph* image-based rendering systems described in Chapter 13 to represent the collection of rays seen by a camera as it moves in front of an object. The two-endpoint representation is also useful for representing line segments, even when their exact endpoints cannot be seen (only guessed at).

If we wish to represent all possible lines without bias towards any particular orientation, we can use *Plücker coordinates* (Hartley and Zisserman 2004, Chapter 2; Faugeras and Luong 2001, Chapter 3). These coordinates are the six independent non-zero entries in the 4×4 skew symmetric matrix

$$\mathbf{L} = \tilde{\mathbf{p}}\tilde{\mathbf{q}}^T - \tilde{\mathbf{q}}\tilde{\mathbf{p}}^T, \quad (2.12)$$

where $\tilde{\mathbf{p}}$ and $\tilde{\mathbf{q}}$ are *any* two (non-identical) points on the line. This representation has only four degrees of freedom, since \mathbf{L} is homogeneous and also satisfies $\det(\mathbf{L}) = 0$, which results in a quadratic constraint on the Plücker coordinates.

In practice, the minimal representation is not essential for most applications. An adequate model of 3D lines can be obtained by estimating their direction (which may be known ahead of time, e.g., for architecture) and some point within the visible portion of the line (see Section 7.5.1) or by using the two endpoints, since lines are most often visible as finite line segments. However, if you are interested in more details about the topic of minimal line parameterizations, Förstner (2005) discusses various ways to infer and model 3D lines in projective geometry, as well as how to estimate the uncertainty in such fitted models.

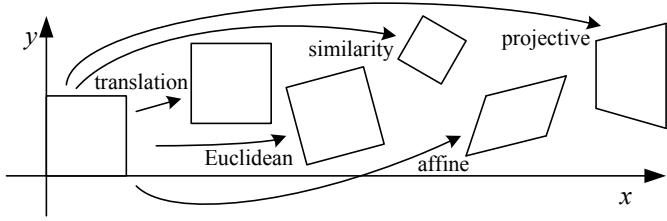


Figure 2.4 Basic set of 2D planar transformations.

3D quadrics. The 3D analog of a conic section is a quadric surface

$$\bar{x}^T Q \bar{x} = 0 \quad (2.13)$$

(Hartley and Zisserman 2004, Chapter 2). Again, while quadric surfaces are useful in the study of multi-view geometry and can also serve as useful modeling primitives (spheres, ellipsoids, cylinders), we do not study them in great detail in this book.

2.1.2 2D transformations

Having defined our basic primitives, we can now turn our attention to how they can be transformed. The simplest transformations occur in the 2D plane and are illustrated in Figure 2.4.

Translation. 2D translations can be written as $x' = x + t$ or

$$x' = [\ I \ t] \bar{x} \quad (2.14)$$

where I is the (2×2) identity matrix or

$$\bar{x}' = \begin{bmatrix} I & t \\ 0^T & 1 \end{bmatrix} \bar{x} \quad (2.15)$$

where 0 is the zero vector. Using a 2×3 matrix results in a more compact notation, whereas using a full-rank 3×3 matrix (which can be obtained from the 2×3 matrix by appending a $[0^T \ 1]$ row) makes it possible to chain transformations using matrix multiplication. Note that in any equation where an augmented vector such as \bar{x} appears on both sides, it can always be replaced with a full homogeneous vector \tilde{x} .

Rotation + translation. This transformation is also known as *2D rigid body motion* or the *2D Euclidean transformation* (since Euclidean distances are preserved). It can be written as $x' = Rx + t$ or

$$x' = [\ R \ t] \bar{x} \quad (2.16)$$

where

$$R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (2.17)$$

is an orthonormal rotation matrix with $RR^T = I$ and $|R| = 1$.

Scaled rotation. Also known as the *similarity transform*, this transformation can be expressed as $\mathbf{x}' = s\mathbf{R}\mathbf{x} + \mathbf{t}$ where s is an arbitrary scale factor. It can also be written as

$$\mathbf{x}' = [\mathbf{sR} \quad \mathbf{t}] \bar{\mathbf{x}} = \begin{bmatrix} a & -b & t_x \\ b & a & t_y \end{bmatrix} \bar{\mathbf{x}}, \quad (2.18)$$

where we no longer require that $a^2 + b^2 = 1$. The similarity transform preserves angles between lines.

Affine. The affine transformation is written as $\mathbf{x}' = \mathbf{A}\bar{\mathbf{x}}$, where \mathbf{A} is an arbitrary 2×3 matrix, i.e.,

$$\mathbf{x}' = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} \bar{\mathbf{x}}. \quad (2.19)$$

Parallel lines remain parallel under affine transformations.

Projective. This transformation, also known as a *perspective transform* or *homography*, operates on homogeneous coordinates,

$$\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}\tilde{\mathbf{x}}, \quad (2.20)$$

where $\tilde{\mathbf{H}}$ is an arbitrary 3×3 matrix. Note that $\tilde{\mathbf{H}}$ is homogeneous, i.e., it is only defined up to a scale, and that two $\tilde{\mathbf{H}}$ matrices that differ only by scale are equivalent. The resulting homogeneous coordinate $\tilde{\mathbf{x}}'$ must be normalized in order to obtain an inhomogeneous result \mathbf{x} , i.e.,

$$x' = \frac{h_{00}x + h_{01}y + h_{02}}{h_{20}x + h_{21}y + h_{22}} \text{ and } y' = \frac{h_{10}x + h_{11}y + h_{12}}{h_{20}x + h_{21}y + h_{22}}. \quad (2.21)$$

Perspective transformations preserve straight lines (i.e., they remain straight after the transformation).

Hierarchy of 2D transformations. The preceding set of transformations are illustrated in Figure 2.4 and summarized in Table 2.1. The easiest way to think of them is as a set of (potentially restricted) 3×3 matrices operating on 2D homogeneous coordinate vectors. Hartley and Zisserman (2004) contains a more detailed description of the hierarchy of 2D planar transformations.

The above transformations form a nested set of *groups*, i.e., they are closed under composition and have an inverse that is a member of the same group. (This will be important later when applying these transformations to images in Section 3.6.) Each (simpler) group is a subset of the more complex group below it.

Co-vectors. While the above transformations can be used to transform points in a 2D plane, can they also be used directly to transform a line equation? Consider the homogeneous equation $\tilde{\mathbf{l}} \cdot \tilde{\mathbf{x}} = 0$. If we transform $\mathbf{x}' = \tilde{\mathbf{H}}\mathbf{x}$, we obtain

$$\tilde{\mathbf{l}} \cdot \tilde{\mathbf{x}}' = \tilde{\mathbf{l}}^T \tilde{\mathbf{H}} \tilde{\mathbf{x}} = (\tilde{\mathbf{H}}^T \tilde{\mathbf{l}})^T \tilde{\mathbf{x}} = \tilde{\mathbf{l}} \cdot \tilde{\mathbf{x}} = 0, \quad (2.22)$$

i.e., $\tilde{\mathbf{l}}' = \tilde{\mathbf{H}}^{-T} \tilde{\mathbf{l}}$. Thus, the action of a projective transformation on a *co-vector* such as a 2D line or 3D normal can be represented by the transposed inverse of the matrix, which is equivalent to the *adjoint* of $\tilde{\mathbf{H}}$, since projective transformation matrices are homogeneous. Jim

Transformation	Matrix	# DoF	Preserves	Icon
translation	$[\mathbf{I} \mathbf{t}]_{2 \times 3}$	2	orientation	
rigid (Euclidean)	$[\mathbf{R} \mathbf{t}]_{2 \times 3}$	3	lengths	
similarity	$[s\mathbf{R} \mathbf{t}]_{2 \times 3}$	4	angles	
affine	$[\mathbf{A}]_{2 \times 3}$	6	parallelism	
projective	$[\tilde{\mathbf{H}}]_{3 \times 3}$	8	straight lines	

Table 2.1 Hierarchy of 2D coordinate transformations. Each transformation also preserves the properties listed in the rows below it, i.e., similarity preserves not only angles but also parallelism and straight lines. The 2×3 matrices are extended with a third $[\mathbf{0}^T \ 1]$ row to form a full 3×3 matrix for homogeneous coordinate transformations.

Blinn (1998) describes (in Chapters 9 and 10) the ins and outs of notating and manipulating co-vectors.

While the above transformations are the ones we use most extensively, a number of additional transformations are sometimes used.

Stretch/squash. This transformation changes the aspect ratio of an image,

$$\begin{aligned} x' &= s_x x + t_x \\ y' &= s_y y + t_y, \end{aligned}$$

and is a restricted form of an affine transformation. Unfortunately, it does not nest cleanly with the groups listed in Table 2.1.

Planar surface flow. This eight-parameter transformation (Horn 1986; Bergen, Anandan, Hanna *et al.* 1992; Girod, Greiner, and Niemann 2000),

$$\begin{aligned} x' &= a_0 + a_1 x + a_2 y + a_6 x^2 + a_7 x y \\ y' &= a_3 + a_4 x + a_5 y + a_7 x^2 + a_6 x y, \end{aligned}$$

arises when a planar surface undergoes a small 3D motion. It can thus be thought of as a small motion approximation to a full homography. Its main attraction is that it is *linear* in the motion parameters, a_k , which are often the quantities being estimated.

Bilinear interpolant. This eight-parameter transform (Wolberg 1990),

$$\begin{aligned} x' &= a_0 + a_1 x + a_2 y + a_6 x y \\ y' &= a_3 + a_4 x + a_5 y + a_7 x y, \end{aligned}$$

can be used to interpolate the deformation due to the motion of the four corner points of a square. (In fact, it can interpolate the motion of any four non-collinear points.) While

Transformation	Matrix	# DoF	Preserves	Icon
translation	$[\mathbf{I} \mid \mathbf{t}]_{3 \times 4}$	3	orientation	
rigid (Euclidean)	$[\mathbf{R} \mid \mathbf{t}]_{3 \times 4}$	6	lengths	
similarity	$[s\mathbf{R} \mid \mathbf{t}]_{3 \times 4}$	7	angles	
affine	$[\mathbf{A}]_{3 \times 4}$	12	parallelism	
projective	$[\tilde{\mathbf{H}}]_{4 \times 4}$	15	straight lines	

Table 2.2 Hierarchy of 3D coordinate transformations. Each transformation also preserves the properties listed in the rows below it, i.e., similarity preserves not only angles but also parallelism and straight lines. The 3×4 matrices are extended with a fourth $[0^T \ 1]$ row to form a full 4×4 matrix for homogeneous coordinate transformations. The mnemonic icons are drawn in 2D but are meant to suggest transformations occurring in a full 3D cube.

the deformation is linear in the motion parameters, it does not generally preserve straight lines (only lines parallel to the square axes). However, it is often quite useful, e.g., in the interpolation of sparse grids using splines (Section 8.3).

2.1.3 3D transformations

The set of three-dimensional coordinate transformations is very similar to that available for 2D transformations and is summarized in Table 2.2. As in 2D, these transformations form a nested set of groups. Hartley and Zisserman (2004, Section 2.4) give a more detailed description of this hierarchy.

Translation. 3D translations can be written as $\mathbf{x}' = \mathbf{x} + \mathbf{t}$ or

$$\mathbf{x}' = [\mathbf{I} \ \mathbf{t}] \bar{\mathbf{x}} \quad (2.23)$$

where \mathbf{I} is the (3×3) identity matrix and $\mathbf{0}$ is the zero vector.

Rotation + translation. Also known as 3D *rigid body motion* or the 3D *Euclidean transformation*, it can be written as $\mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t}$ or

$$\mathbf{x}' = [\mathbf{R} \ \mathbf{t}] \bar{\mathbf{x}} \quad (2.24)$$

where \mathbf{R} is a 3×3 orthonormal rotation matrix with $\mathbf{R}\mathbf{R}^T = \mathbf{I}$ and $|\mathbf{R}| = 1$. Note that sometimes it is more convenient to describe a rigid motion using

$$\mathbf{x}' = \mathbf{R}(\mathbf{x} - \mathbf{c}) = \mathbf{R}\mathbf{x} - \mathbf{R}\mathbf{c}, \quad (2.25)$$

where \mathbf{c} is the center of rotation (often the camera center).

Compactly parameterizing a 3D rotation is a non-trivial task, which we describe in more detail below.

Scaled rotation. The 3D *similarity transform* can be expressed as $\mathbf{x}' = s\mathbf{R}\mathbf{x} + \mathbf{t}$ where s is an arbitrary scale factor. It can also be written as

$$\mathbf{x}' = [s\mathbf{R} \quad \mathbf{t}] \bar{\mathbf{x}}. \quad (2.26)$$

This transformation preserves angles between lines and planes.

Affine. The affine transform is written as $\mathbf{x}' = \mathbf{A}\bar{\mathbf{x}}$, where \mathbf{A} is an arbitrary 3×4 matrix, i.e.,

$$\mathbf{x}' = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{bmatrix} \bar{\mathbf{x}}. \quad (2.27)$$

Parallel lines and planes remain parallel under affine transformations.

Projective. This transformation, variously known as a *3D perspective transform*, *homography*, or *collineation*, operates on homogeneous coordinates,

$$\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}\tilde{\mathbf{x}}, \quad (2.28)$$

where $\tilde{\mathbf{H}}$ is an arbitrary 4×4 homogeneous matrix. As in 2D, the resulting homogeneous coordinate $\tilde{\mathbf{x}}'$ must be normalized in order to obtain an inhomogeneous result \mathbf{x} . Perspective transformations preserve straight lines (i.e., they remain straight after the transformation).

2.1.4 3D rotations

The biggest difference between 2D and 3D coordinate transformations is that the parameterization of the 3D rotation matrix \mathbf{R} is not as straightforward but several possibilities exist.

Euler angles

A rotation matrix can be formed as the product of three rotations around three cardinal axes, e.g., x , y , and z , or x , y , and x . This is generally a bad idea, as the result depends on the order in which the transforms are applied. What is worse, it is not always possible to move smoothly in the parameter space, i.e., sometimes one or more of the Euler angles change dramatically in response to a small change in rotation.¹ For these reasons, we do not even give the formula for Euler angles in this book—interested readers can look in other textbooks or technical reports (Faugeras 1993; Diebel 2006). Note that, in some applications, if the rotations are known to be a set of uni-axial transforms, they can always be represented using an explicit set of rigid transformations.

Axis/angle (exponential twist)

A rotation can be represented by a rotation axis $\hat{\mathbf{n}}$ and an angle θ , or equivalently by a 3D vector $\boldsymbol{\omega} = \theta\hat{\mathbf{n}}$. Figure 2.5 shows how we can compute the equivalent rotation. First, we project the vector \mathbf{v} onto the axis $\hat{\mathbf{n}}$ to obtain

$$\mathbf{v}_{\parallel} = \hat{\mathbf{n}}(\hat{\mathbf{n}} \cdot \mathbf{v}) = (\hat{\mathbf{n}}\hat{\mathbf{n}}^T)\mathbf{v}, \quad (2.29)$$

¹ In robotics, this is sometimes referred to as *gimbal lock*.

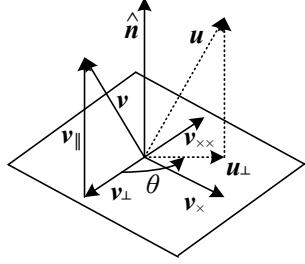


Figure 2.5 Rotation around an axis \hat{n} by an angle θ .

which is the component of v that is not affected by the rotation. Next, we compute the perpendicular residual of v from \hat{n} ,

$$\mathbf{v}_\perp = \mathbf{v} - \mathbf{v}_\parallel = (\mathbf{I} - \hat{\mathbf{n}}\hat{\mathbf{n}}^T)\mathbf{v}. \quad (2.30)$$

We can rotate this vector by 90° using the cross product,

$$\mathbf{v}_\times = \hat{\mathbf{n}} \times \mathbf{v} = [\hat{\mathbf{n}}]_\times \mathbf{v}, \quad (2.31)$$

where $[\hat{\mathbf{n}}]_\times$ is the matrix form of the cross product operator with the vector $\hat{\mathbf{n}} = (\hat{n}_x, \hat{n}_y, \hat{n}_z)$,

$$[\hat{\mathbf{n}}]_\times = \begin{bmatrix} 0 & -\hat{n}_z & \hat{n}_y \\ \hat{n}_z & 0 & -\hat{n}_x \\ -\hat{n}_y & \hat{n}_x & 0 \end{bmatrix}. \quad (2.32)$$

Note that rotating this vector by another 90° is equivalent to taking the cross product again,

$$\mathbf{v}_{\times\times} = \hat{\mathbf{n}} \times \mathbf{v}_\times = [\hat{\mathbf{n}}]_\times^2 \mathbf{v} = -\mathbf{v}_\perp,$$

and hence

$$\mathbf{v}_\parallel = \mathbf{v} - \mathbf{v}_\perp = \mathbf{v} + \mathbf{v}_{\times\times} = (\mathbf{I} + [\hat{\mathbf{n}}]_\times^2)\mathbf{v}.$$

We can now compute the in-plane component of the rotated vector \mathbf{u} as

$$\mathbf{u}_\perp = \cos \theta \mathbf{v}_\perp + \sin \theta \mathbf{v}_\times = (\sin \theta [\hat{\mathbf{n}}]_\times - \cos \theta [\hat{\mathbf{n}}]_\times^2)\mathbf{v}.$$

Putting all these terms together, we obtain the final rotated vector as

$$\mathbf{u} = \mathbf{u}_\perp + \mathbf{v}_\parallel = (\mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_\times + (1 - \cos \theta)[\hat{\mathbf{n}}]_\times^2)\mathbf{v}. \quad (2.33)$$

We can therefore write the rotation matrix corresponding to a rotation by θ around an axis \hat{n} as

$$\mathbf{R}(\hat{\mathbf{n}}, \theta) = \mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_\times + (1 - \cos \theta)[\hat{\mathbf{n}}]_\times^2, \quad (2.34)$$

which is known as *Rodriguez's formula* (Ayache 1989).

The product of the axis \hat{n} and angle θ , $\boldsymbol{\omega} = \theta \hat{n} = (\omega_x, \omega_y, \omega_z)$, is a minimal representation for a 3D rotation. Rotations through common angles such as multiples of 90° can be represented exactly (and converted to exact matrices) if θ is stored in degrees. Unfortunately,

this representation is not unique, since we can always add a multiple of 360° (2π radians) to θ and get the same rotation matrix. As well, $(\hat{\mathbf{n}}, \theta)$ and $(-\hat{\mathbf{n}}, -\theta)$ represent the same rotation.

However, for small rotations (e.g., corrections to rotations), this is an excellent choice. In particular, for small (infinitesimal or instantaneous) rotations and θ expressed in radians, Rodriguez's formula simplifies to

$$\mathbf{R}(\boldsymbol{\omega}) \approx \mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_{\times} \approx \mathbf{I} + [\theta \hat{\mathbf{n}}]_{\times} = \begin{bmatrix} 1 & -\omega_z & \omega_y \\ \omega_z & 1 & -\omega_x \\ -\omega_y & \omega_x & 1 \end{bmatrix}, \quad (2.35)$$

which gives a nice linearized relationship between the rotation parameters $\boldsymbol{\omega}$ and \mathbf{R} . We can also write $\mathbf{R}(\boldsymbol{\omega})\mathbf{v} \approx \mathbf{v} + \boldsymbol{\omega} \times \mathbf{v}$, which is handy when we want to compute the derivative of $\mathbf{R}\mathbf{v}$ with respect to $\boldsymbol{\omega}$,

$$\frac{\partial \mathbf{R}\mathbf{v}}{\partial \boldsymbol{\omega}^T} = -[\mathbf{v}]_{\times} = \begin{bmatrix} 0 & z & -y \\ -z & 0 & x \\ y & -x & 0 \end{bmatrix}. \quad (2.36)$$

Another way to derive a rotation through a finite angle is called the *exponential twist* (Murray, Li, and Sastry 1994). A rotation by an angle θ is equivalent to k rotations through θ/k . In the limit as $k \rightarrow \infty$, we obtain

$$\mathbf{R}(\hat{\mathbf{n}}, \theta) = \lim_{k \rightarrow \infty} (\mathbf{I} + \frac{1}{k} [\theta \hat{\mathbf{n}}]_{\times})^k = \exp [\boldsymbol{\omega}]_{\times}. \quad (2.37)$$

If we expand the matrix exponential as a Taylor series (using the identity $[\hat{\mathbf{n}}]_{\times}^{k+2} = -[\hat{\mathbf{n}}]_{\times}^k$, $k > 0$, and again assuming θ is in radians),

$$\begin{aligned} \exp [\boldsymbol{\omega}]_{\times} &= \mathbf{I} + \theta [\hat{\mathbf{n}}]_{\times} + \frac{\theta^2}{2} [\hat{\mathbf{n}}]_{\times}^2 + \frac{\theta^3}{3!} [\hat{\mathbf{n}}]_{\times}^3 + \dots \\ &= \mathbf{I} + (\theta - \frac{\theta^3}{3!} + \dots) [\hat{\mathbf{n}}]_{\times} + (\frac{\theta^2}{2} - \frac{\theta^3}{4!} + \dots) [\hat{\mathbf{n}}]_{\times}^2 \\ &= \mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_{\times} + (1 - \cos \theta) [\hat{\mathbf{n}}]_{\times}^2, \end{aligned} \quad (2.38)$$

which yields the familiar Rodriguez's formula.

Unit quaternions

The unit quaternion representation is closely related to the angle/axis representation. A unit quaternion is a unit length 4-vector whose components can be written as $\mathbf{q} = (q_x, q_y, q_z, q_w)$ or $\mathbf{q} = (x, y, z, w)$ for short. Unit quaternions live on the unit sphere $\|\mathbf{q}\| = 1$ and *antipodal* (opposite sign) quaternions, \mathbf{q} and $-\mathbf{q}$, represent the same rotation (Figure 2.6). Other than this ambiguity (dual covering), the unit quaternion representation of a rotation is unique. Furthermore, the representation is *continuous*, i.e., as rotation matrices vary continuously, one can find a continuous quaternion representation, although the path on the quaternion sphere may wrap all the way around before returning to the “origin” $\mathbf{q}_o = (0, 0, 0, 1)$. For these and other reasons given below, quaternions are a very popular representation for pose and for pose interpolation in computer graphics (Shoemake 1985).

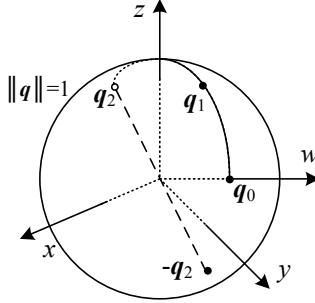


Figure 2.6 Unit quaternions live on the unit sphere $\|q\| = 1$. This figure shows a smooth trajectory through the three quaternions q_0 , q_1 , and q_2 . The *antipodal* point to q_2 , namely $-q_2$, represents the same rotation as q_2 .

Quaternions can be derived from the axis/angle representation through the formula

$$\mathbf{q} = (\mathbf{v}, w) = \left(\sin \frac{\theta}{2} \hat{\mathbf{n}}, \cos \frac{\theta}{2} \right), \quad (2.39)$$

where $\hat{\mathbf{n}}$ and θ are the rotation axis and angle. Using the trigonometric identities $\sin \theta = 2 \sin \frac{\theta}{2} \cos \frac{\theta}{2}$ and $(1 - \cos \theta) = 2 \sin^2 \frac{\theta}{2}$, Rodriguez's formula can be converted to

$$\begin{aligned} \mathbf{R}(\hat{\mathbf{n}}, \theta) &= \mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_{\times} + (1 - \cos \theta) [\hat{\mathbf{n}}]_{\times}^2 \\ &= \mathbf{I} + 2w[\mathbf{v}]_{\times} + 2[\mathbf{v}]_{\times}^2. \end{aligned} \quad (2.40)$$

This suggests a quick way to rotate a vector \mathbf{v} by a quaternion using a series of cross products, scalings, and additions. To obtain a formula for $\mathbf{R}(\mathbf{q})$ as a function of (x, y, z, w) , recall that

$$[\mathbf{v}]_{\times} = \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix} \text{ and } [\mathbf{v}]_{\times}^2 = \begin{bmatrix} -y^2 - z^2 & xy & xz \\ xy & -x^2 - z^2 & yz \\ xz & yz & -x^2 - y^2 \end{bmatrix}.$$

We thus obtain

$$\mathbf{R}(\mathbf{q}) = \begin{bmatrix} 1 - 2(y^2 + z^2) & 2(xy - zw) & 2(xz + yw) \\ 2(xy + zw) & 1 - 2(x^2 + z^2) & 2(yz - xw) \\ 2(xz - yw) & 2(yz + xw) & 1 - 2(x^2 + y^2) \end{bmatrix}. \quad (2.41)$$

The diagonal terms can be made more symmetrical by replacing $1 - 2(y^2 + z^2)$ with $(x^2 + w^2 - y^2 - z^2)$, etc.

The nicest aspect of unit quaternions is that there is a simple algebra for composing rotations expressed as unit quaternions. Given two quaternions $\mathbf{q}_0 = (\mathbf{v}_0, w_0)$ and $\mathbf{q}_1 = (\mathbf{v}_1, w_1)$, the *quaternion multiply* operator is defined as

$$\mathbf{q}_2 = \mathbf{q}_0 \mathbf{q}_1 = (\mathbf{v}_0 \times \mathbf{v}_1 + w_0 \mathbf{v}_1 + w_1 \mathbf{v}_0, w_0 w_1 - \mathbf{v}_0 \cdot \mathbf{v}_1), \quad (2.42)$$

with the property that $\mathbf{R}(\mathbf{q}_2) = \mathbf{R}(\mathbf{q}_0)\mathbf{R}(\mathbf{q}_1)$. Note that quaternion multiplication is *not* commutative, just as 3D rotations and matrix multiplications are not.

```

procedure slerp( $\mathbf{q}_0, \mathbf{q}_1, \alpha$ ):
    1.  $\mathbf{q}_r = \mathbf{q}_1 / \mathbf{q}_0 = (\mathbf{v}_r, w_r)$ 
    2. if  $w_r < 0$  then  $\mathbf{q}_r \leftarrow -\mathbf{q}_r$ 
    3.  $\theta_r = 2 \tan^{-1}(\|\mathbf{v}_r\|/w_r)$ 
    4.  $\hat{\mathbf{n}}_r = \mathcal{N}(\mathbf{v}_r) = \mathbf{v}_r / \|\mathbf{v}_r\|$ 
    5.  $\theta_\alpha = \alpha \theta_r$ 
    6.  $\mathbf{q}_\alpha = (\sin \frac{\theta_\alpha}{2} \hat{\mathbf{n}}_r, \cos \frac{\theta_\alpha}{2})$ 
    7. return  $\mathbf{q}_2 = \mathbf{q}_\alpha \mathbf{q}_0$ 

```

Algorithm 2.1 Spherical linear interpolation (slerp). The axis and total angle are first computed from the quaternion ratio. (This computation can be lifted outside an inner loop that generates a set of interpolated position for animation.) An incremental quaternion is then computed and multiplied by the starting rotation quaternion.

Taking the inverse of a quaternion is easy: Just flip the sign of v or w (but not both!). (You can verify this has the desired effect of transposing the R matrix in (2.41).) Thus, we can also define *quaternion division* as

$$\mathbf{q}_2 = \mathbf{q}_0 / \mathbf{q}_1 = \mathbf{q}_0 \mathbf{q}_1^{-1} = (\mathbf{v}_0 \times \mathbf{v}_1 + w_0 \mathbf{v}_1 - w_1 \mathbf{v}_0, -w_0 w_1 - \mathbf{v}_0 \cdot \mathbf{v}_1). \quad (2.43)$$

This is useful when the *incremental rotation* between two rotations is desired.

In particular, if we want to determine a rotation that is partway between two given rotations, we can compute the incremental rotation, take a fraction of the angle, and compute the new rotation. This procedure is called *spherical linear interpolation* or *slerp* for short (Shoemake 1985) and is given in Algorithm 2.1. Note that Shoemake presents two formulas other than the one given here. The first exponentiates \mathbf{q}_r by alpha before multiplying the original quaternion,

$$\mathbf{q}_2 = \mathbf{q}_r^\alpha \mathbf{q}_0, \quad (2.44)$$

while the second treats the quaternions as 4-vectors on a sphere and uses

$$\mathbf{q}_2 = \frac{\sin(1-\alpha)\theta}{\sin \theta} \mathbf{q}_0 + \frac{\sin \alpha \theta}{\sin \theta} \mathbf{q}_1, \quad (2.45)$$

where $\theta = \cos^{-1}(\mathbf{q}_0 \cdot \mathbf{q}_1)$ and the dot product is directly between the quaternion 4-vectors. All of these formulas give comparable results, although care should be taken when \mathbf{q}_0 and \mathbf{q}_1 are close together, which is why I prefer to use an arctangent to establish the rotation angle.

Which rotation representation is better?

The choice of representation for 3D rotations depends partly on the application.

The axis/angle representation is minimal, and hence does not require any additional constraints on the parameters (no need to re-normalize after each update). If the angle is expressed in degrees, it is easier to understand the pose (say, 90° twist around x -axis), and also

easier to express exact rotations. When the angle is in radians, the derivatives of \mathbf{R} with respect to $\boldsymbol{\omega}$ can easily be computed (2.36).

Quaternions, on the other hand, are better if you want to keep track of a smoothly moving camera, since there are no discontinuities in the representation. It is also easier to interpolate between rotations and to chain rigid transformations (Murray, Li, and Sastry 1994; Bregler and Malik 1998).

My usual preference is to use quaternions, but to update their estimates using an incremental rotation, as described in Section 6.2.2.

2.1.5 3D to 2D projections

Now that we know how to represent 2D and 3D geometric primitives and how to transform them spatially, we need to specify how 3D primitives are projected onto the image plane. We can do this using a linear 3D to 2D projection matrix. The simplest model is orthography, which requires no division to get the final (inhomogeneous) result. The more commonly used model is perspective, since this more accurately models the behavior of real cameras.

Orthography and para-perspective

An orthographic projection simply drops the z component of the three-dimensional coordinate \mathbf{p} to obtain the 2D point \mathbf{x} . (In this section, we use \mathbf{p} to denote 3D points and \mathbf{x} to denote 2D points.) This can be written as

$$\mathbf{x} = [\mathbf{I}_{2 \times 2} | \mathbf{0}] \mathbf{p}. \quad (2.46)$$

If we are using homogeneous (projective) coordinates, we can write

$$\tilde{\mathbf{x}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{p}}, \quad (2.47)$$

i.e., we drop the z component but keep the w component. Orthography is an approximate model for long focal length (telephoto) lenses and objects whose depth is *shallow* relative to their distance to the camera (Sawhney and Hanson 1991). It is exact only for *telecentric* lenses (Baker and Nayar 1999, 2001).

In practice, world coordinates (which may measure dimensions in meters) need to be scaled to fit onto an image sensor (physically measured in millimeters, but ultimately measured in pixels). For this reason, *scaled orthography* is actually more commonly used,

$$\mathbf{x} = [s\mathbf{I}_{2 \times 2} | \mathbf{0}] \mathbf{p}. \quad (2.48)$$

This model is equivalent to first projecting the world points onto a local fronto-parallel image plane and then scaling this image using regular perspective projection. The scaling can be the same for all parts of the scene (Figure 2.7b) or it can be different for objects that are being modeled independently (Figure 2.7c). More importantly, the scaling can vary from frame to frame when estimating *structure from motion*, which can better model the scale change that occurs as an object approaches the camera.

Scaled orthography is a popular model for reconstructing the 3D shape of objects far away from the camera, since it greatly simplifies certain computations. For example, *pose* (camera

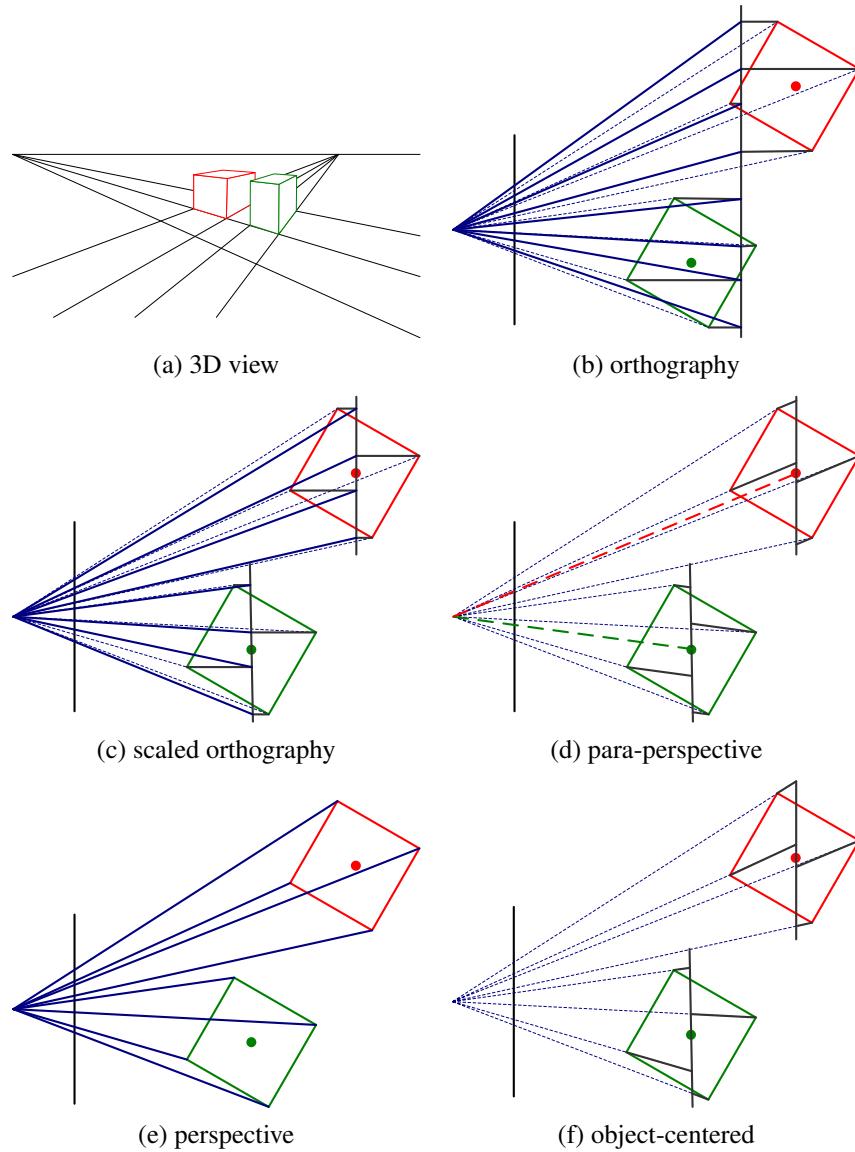


Figure 2.7 Commonly used projection models: (a) 3D view of world, (b) orthography, (c) scaled orthography, (d) para-perspective, (e) perspective, (f) object-centered. Each diagram shows a top-down view of the projection. Note how parallel lines on the ground plane and box sides remain parallel in the non-perspective projections.

orientation) can be estimated using simple least squares (Section 6.2.1). Under orthography, structure and motion can simultaneously be estimated using *factorization* (singular value decomposition), as discussed in Section 7.3 (Tomasi and Kanade 1992).

A closely related projection model is *para-perspective* (Aloimonos 1990; Poelman and Kanade 1997). In this model, object points are again first projected onto a local reference parallel to the image plane. However, rather than being projected orthogonally to this plane, they are projected *parallel* to the line of sight to the object center (Figure 2.7d). This is followed by the usual projection onto the final image plane, which again amounts to a scaling. The combination of these two projections is therefore *affine* and can be written as

$$\tilde{\mathbf{x}} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{p}}. \quad (2.49)$$

Note how parallel lines in 3D remain parallel after projection in Figure 2.7b–d. Para-perspective provides a more accurate projection model than scaled orthography, without incurring the added complexity of per-pixel perspective division, which invalidates traditional factorization methods (Poelman and Kanade 1997).

Perspective

The most commonly used projection in computer graphics and computer vision is true 3D *perspective* (Figure 2.7e). Here, points are projected onto the image plane by dividing them by their z component. Using inhomogeneous coordinates, this can be written as

$$\bar{\mathbf{x}} = \mathcal{P}_z(\mathbf{p}) = \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix}. \quad (2.50)$$

In homogeneous coordinates, the projection has a simple linear form,

$$\tilde{\mathbf{x}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tilde{\mathbf{p}}, \quad (2.51)$$

i.e., we drop the w component of \mathbf{p} . Thus, after projection, it is not possible to recover the *distance* of the 3D point from the image, which makes sense for a 2D imaging sensor.

A form often seen in computer graphics systems is a two-step projection that first projects 3D coordinates into *normalized device coordinates* in the range $(x, y, z) \in [-1, -1] \times [-1, 1] \times [0, 1]$, and then rescales these coordinates to integer pixel coordinates using a *viewport* transformation (Watt 1995; OpenGL-ARB 1997). The (initial) perspective projection is then represented using a 4×4 matrix

$$\tilde{\mathbf{x}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -z_{\text{far}}/z_{\text{range}} & z_{\text{near}}z_{\text{far}}/z_{\text{range}} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tilde{\mathbf{p}}, \quad (2.52)$$

where z_{near} and z_{far} are the near and far z *clipping planes* and $z_{\text{range}} = z_{\text{far}} - z_{\text{near}}$. Note that the first two rows are actually scaled by the focal length and the aspect ratio so that

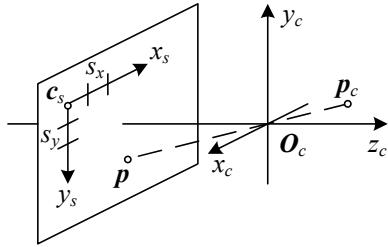


Figure 2.8 Projection of a 3D camera-centered point p_c onto the sensor planes at location p . O_c is the camera center (nodal point), c_s is the 3D origin of the sensor plane coordinate system, and s_x and s_y are the pixel spacings.

visible rays are mapped to $(x, y, z) \in [-1, -1]^2$. The reason for keeping the third row, rather than dropping it, is that visibility operations, such as *z-buffering*, require a depth for every graphical element that is being rendered.

If we set $z_{\text{near}} = 1$, $z_{\text{far}} \rightarrow \infty$, and switch the sign of the third row, the third element of the normalized screen vector becomes the inverse depth, i.e., the *disparity* (Okutomi and Kanade 1993). This can be quite convenient in many cases since, for cameras moving around outdoors, the inverse depth to the camera is often a more well-conditioned parameterization than direct 3D distance.

While a regular 2D image sensor has no way of measuring distance to a surface point, *range sensors* (Section 12.2) and stereo matching algorithms (Chapter 11) can compute such values. It is then convenient to be able to map from a sensor-based depth or disparity value d directly back to a 3D location using the inverse of a 4×4 matrix (Section 2.1.5). We can do this if we represent perspective projection using a full-rank 4×4 matrix, as in (2.64).

Camera intrinsics

Once we have projected a 3D point through an ideal pinhole using a projection matrix, we must still transform the resulting coordinates according to the pixel sensor spacing and the relative position of the sensor plane to the origin. Figure 2.8 shows an illustration of the geometry involved. In this section, we first present a mapping from 2D pixel coordinates to 3D rays using a sensor homography M_s , since this is easier to explain in terms of physically measurable quantities. We then relate these quantities to the more commonly used camera intrinsic matrix K , which is used to map 3D camera-centered points p_c to 2D pixel coordinates \tilde{x}_s .

Image sensors return pixel values indexed by integer *pixel coordinates* (x_s, y_s) , often with the coordinates starting at the upper-left corner of the image and moving down and to the right. (This convention is not obeyed by all imaging libraries, but the adjustment for other coordinate systems is straightforward.) To map pixel centers to 3D coordinates, we first scale the (x_s, y_s) values by the pixel spacings (s_x, s_y) (sometimes expressed in microns for solid-state sensors) and then describe the orientation of the sensor array relative to the camera projection center O_c with an origin c_s and a 3D rotation R_s (Figure 2.8).

The combined 2D to 3D projection can then be written as

$$\mathbf{p} = [\mathbf{R}_s \mid \mathbf{c}_s] \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix} = \mathbf{M}_s \bar{\mathbf{x}}_s. \quad (2.53)$$

The first two columns of the 3×3 matrix \mathbf{M}_s are the 3D vectors corresponding to unit steps in the image pixel array along the x_s and y_s directions, while the third column is the 3D image array origin \mathbf{c}_s .

The matrix \mathbf{M}_s is parameterized by eight unknowns: the three parameters describing the rotation \mathbf{R}_s , the three parameters describing the translation \mathbf{c}_s , and the two scale factors (s_x, s_y) . Note that we ignore here the possibility of *skew* between the two axes on the image plane, since solid-state manufacturing techniques render this negligible. In practice, unless we have accurate external knowledge of the sensor spacing or sensor orientation, there are only seven degrees of freedom, since the distance of the sensor from the origin cannot be teased apart from the sensor spacing, based on external image measurement alone.

However, estimating a camera model \mathbf{M}_s with the required seven degrees of freedom (i.e., where the first two columns are orthogonal after an appropriate re-scaling) is impractical, so most practitioners assume a general 3×3 homogeneous matrix form.

The relationship between the 3D pixel center \mathbf{p} and the 3D camera-centered point \mathbf{p}_c is given by an unknown scaling s , $\mathbf{p} = s\mathbf{p}_c$. We can therefore write the complete projection between \mathbf{p}_c and a homogeneous version of the pixel address $\tilde{\mathbf{x}}_s$ as

$$\tilde{\mathbf{x}}_s = \alpha \mathbf{M}_s^{-1} \mathbf{p}_c = \mathbf{K} \mathbf{p}_c. \quad (2.54)$$

The 3×3 matrix \mathbf{K} is called the *calibration matrix* and describes the camera *intrinsics* (as opposed to the camera's orientation in space, which are called the *extrinsics*).

From the above discussion, we see that \mathbf{K} has seven degrees of freedom in theory and eight degrees of freedom (the full dimensionality of a 3×3 homogeneous matrix) in practice. Why, then, do most textbooks on 3D computer vision and multi-view geometry (Faugeras 1993; Hartley and Zisserman 2004; Faugeras and Luong 2001) treat \mathbf{K} as an upper-triangular matrix with five degrees of freedom?

While this is usually not made explicit in these books, it is because we cannot recover the full \mathbf{K} matrix based on external measurement alone. When calibrating a camera (Chapter 6) based on external 3D points or other measurements (Tsai 1987), we end up estimating the intrinsic (\mathbf{K}) and extrinsic (\mathbf{R}, \mathbf{t}) camera parameters simultaneously using a series of measurements,

$$\tilde{\mathbf{x}}_s = \mathbf{K} [\mathbf{R} \mid \mathbf{t}] \mathbf{p}_w = \mathbf{P} \mathbf{p}_w, \quad (2.55)$$

where \mathbf{p}_w are known 3D world coordinates and

$$\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}] \quad (2.56)$$

is known as the *camera matrix*. Inspecting this equation, we see that we can post-multiply \mathbf{K} by \mathbf{R}_1 and pre-multiply $[\mathbf{R}|\mathbf{t}]$ by \mathbf{R}_1^T , and still end up with a valid calibration. Thus, it is impossible based on image measurements alone to know the true orientation of the sensor and the true camera intrinsics.

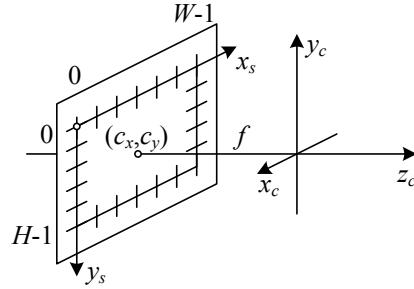


Figure 2.9 Simplified camera intrinsics showing the focal length f and the optical center (c_x, c_y) . The image width and height are W and H .

The choice of an upper-triangular form for \mathbf{K} seems to be conventional. Given a full 3×4 camera matrix $\mathbf{P} = \mathbf{K}[\mathbf{R}|t]$, we can compute an upper-triangular \mathbf{K} matrix using QR factorization (Golub and Van Loan 1996). (Note the unfortunate clash of terminologies: In matrix algebra textbooks, \mathbf{R} represents an upper-triangular (right of the diagonal) matrix; in computer vision, \mathbf{R} is an orthogonal rotation.)

There are several ways to write the upper-triangular form of \mathbf{K} . One possibility is

$$\mathbf{K} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad (2.57)$$

which uses independent *focal lengths* f_x and f_y for the sensor x and y dimensions. The entry s encodes any possible *skew* between the sensor axes due to the sensor not being mounted perpendicular to the optical axis and (c_x, c_y) denotes the *optical center* expressed in pixel coordinates. Another possibility is

$$\mathbf{K} = \begin{bmatrix} f & s & c_x \\ 0 & af & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad (2.58)$$

where the *aspect ratio* a has been made explicit and a common focal length f is used.

In practice, for many applications an even simpler form can be obtained by setting $a = 1$ and $s = 0$,

$$\mathbf{K} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}. \quad (2.59)$$

Often, setting the origin at roughly the center of the image, e.g., $(c_x, c_y) = (W/2, H/2)$, where W and H are the image height and width, can result in a perfectly usable camera model with a single unknown, i.e., the focal length f .

Figure 2.9 shows how these quantities can be visualized as part of a simplified imaging model. Note that now we have placed the image plane *in front* of the nodal point (projection center of the lens). The sense of the y axis has also been flipped to get a coordinate system compatible with the way that most imaging libraries treat the vertical (row) coordinate. Certain graphics libraries, such as Direct3D, use a left-handed coordinate system, which can lead to some confusion.

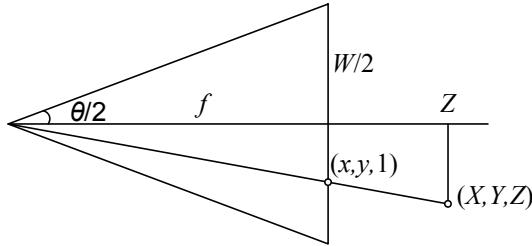


Figure 2.10 Central projection, showing the relationship between the 3D and 2D coordinates, p and x , as well as the relationship between the focal length f , image width W , and the field of view θ .

A note on focal lengths

The issue of how to express focal lengths is one that often causes confusion in implementing computer vision algorithms and discussing their results. This is because the focal length depends on the units used to measure pixels.

If we number pixel coordinates using integer values, say $[0, W) \times [0, H)$, the focal length f and camera center (c_x, c_y) in (2.59) can be expressed as pixel values. How do these quantities relate to the more familiar focal lengths used by photographers?

Figure 2.10 illustrates the relationship between the focal length f , the sensor width W , and the field of view θ , which obey the formula

$$\tan \frac{\theta}{2} = \frac{W}{2f} \quad \text{or} \quad f = \frac{W}{2} \left[\tan \frac{\theta}{2} \right]^{-1}. \quad (2.60)$$

For conventional film cameras, $W = 35\text{mm}$, and hence f is also expressed in millimeters. Since we work with digital images, it is more convenient to express W in pixels so that the focal length f can be used directly in the calibration matrix \mathbf{K} as in (2.59).

Another possibility is to scale the pixel coordinates so that they go from $[-1, 1]$ along the longer image dimension and $[-a^{-1}, a^{-1}]$ along the shorter axis, where $a \geq 1$ is the *image aspect ratio* (as opposed to the *sensor cell aspect ratio* introduced earlier). This can be accomplished using *modified normalized device coordinates*,

$$x'_s = (2x_s - W)/S \quad \text{and} \quad y'_s = (2y_s - H)/S, \quad \text{where} \quad S = \max(W, H). \quad (2.61)$$

This has the advantage that the focal length f and optical center (c_x, c_y) become independent of the image resolution, which can be useful when using multi-resolution, image-processing algorithms, such as image pyramids (Section 3.5).² The use of S instead of W also makes the focal length the same for landscape (horizontal) and portrait (vertical) pictures, as is the case in 35mm photography. (In some computer graphics textbooks and systems, normalized device coordinates go from $[-1, 1] \times [-1, 1]$, which requires the use of two different focal lengths to describe the camera intrinsics (Watt 1995; OpenGL-ARB 1997).) Setting $S = W = 2$ in (2.60), we obtain the simpler (unitless) relationship

$$f^{-1} = \tan \frac{\theta}{2}. \quad (2.62)$$

² To make the conversion truly accurate after a downsampling step in a pyramid, floating point values of W and H would have to be maintained since they can become non-integral if they are ever odd at a larger resolution in the pyramid.

The conversion between the various focal length representations is straightforward, e.g., to go from a unitless f to one expressed in pixels, multiply by $W/2$, while to convert from an f expressed in pixels to the equivalent 35mm focal length, multiply by $35/W$.

Camera matrix

Now that we have shown how to parameterize the calibration matrix \mathbf{K} , we can put the camera intrinsics and extrinsics together to obtain a single 3×4 *camera matrix*

$$\mathbf{P} = \mathbf{K} [\mathbf{R} \mid \mathbf{t}]. \quad (2.63)$$

It is sometimes preferable to use an invertible 4×4 matrix, which can be obtained by not dropping the last row in the \mathbf{P} matrix,

$$\tilde{\mathbf{P}} = \begin{bmatrix} \mathbf{K} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} = \tilde{\mathbf{K}}\mathbf{E}, \quad (2.64)$$

where \mathbf{E} is a 3D rigid-body (Euclidean) transformation and $\tilde{\mathbf{K}}$ is the full-rank calibration matrix. The 4×4 camera matrix $\tilde{\mathbf{P}}$ can be used to map directly from 3D world coordinates $\bar{\mathbf{p}}_w = (x_w, y_w, z_w, 1)$ to screen coordinates (plus disparity), $\mathbf{x}_s = (x_s, y_s, 1, d)$,

$$\mathbf{x}_s \sim \tilde{\mathbf{P}}\bar{\mathbf{p}}_w, \quad (2.65)$$

where \sim indicates equality up to scale. Note that after multiplication by $\tilde{\mathbf{P}}$, the vector is divided by the *third* element of the vector to obtain the normalized form $\mathbf{x}_s = (x_s, y_s, 1, d)$.

Plane plus parallax (projective depth)

In general, when using the 4×4 matrix $\tilde{\mathbf{P}}$, we have the freedom to remap the last row to whatever suits our purpose (rather than just being the “standard” interpretation of disparity as inverse depth). Let us re-write the last row of $\tilde{\mathbf{P}}$ as $\mathbf{p}_3 = s_3[\hat{\mathbf{n}}_0|c_0]$, where $\|\hat{\mathbf{n}}_0\| = 1$. We then have the equation

$$d = \frac{s_3}{z}(\hat{\mathbf{n}}_0 \cdot \mathbf{p}_w + c_0), \quad (2.66)$$

where $z = \mathbf{p}_2 \cdot \bar{\mathbf{p}}_w = r_z \cdot (\mathbf{p}_w - \mathbf{c})$ is the distance of \mathbf{p}_w from the camera center C (2.25) along the optical axis Z (Figure 2.11). Thus, we can interpret d as the *projective disparity* or *projective depth* of a 3D scene point \mathbf{p}_w from the *reference plane* $\hat{\mathbf{n}}_0 \cdot \mathbf{p}_w + c_0 = 0$ (Szeliski and Coughlan 1997; Szeliski and Golland 1999; Shade, Gortler, He *et al.* 1998; Baker, Szeliski, and Anandan 1998). (The projective depth is also sometimes called *parallax* in reconstruction algorithms that use the term *plane plus parallax* (Kumar, Anandan, and Hanna 1994; Sawhney 1994).) Setting $\hat{\mathbf{n}}_0 = \mathbf{0}$ and $c_0 = 1$, i.e., putting the reference plane at infinity, results in the more standard $d = 1/z$ version of disparity (Okutomi and Kanade 1993).

Another way to see this is to invert the $\tilde{\mathbf{P}}$ matrix so that we can map pixels plus disparity directly back to 3D points,

$$\tilde{\mathbf{p}}_w = \tilde{\mathbf{P}}^{-1}\mathbf{x}_s. \quad (2.67)$$

In general, we can choose $\tilde{\mathbf{P}}$ to have whatever form is convenient, i.e., to sample space using an arbitrary projection. This can come in particularly handy when setting up multi-view

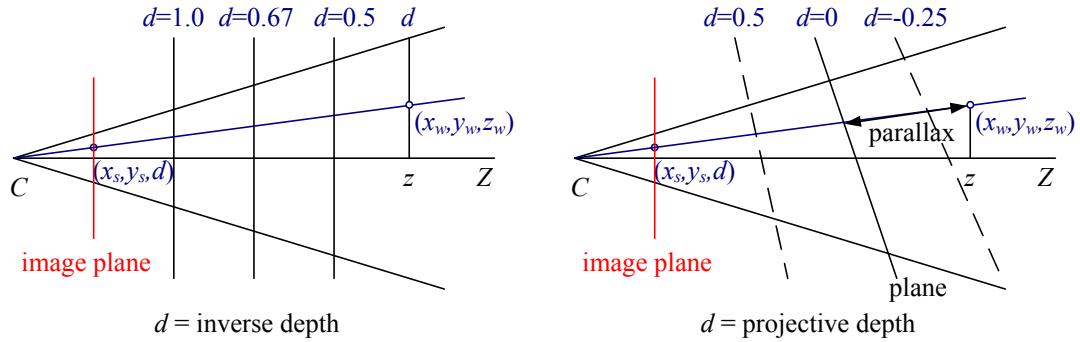


Figure 2.11 Regular disparity (inverse depth) and projective depth (parallax from a reference plane).

stereo reconstruction algorithms, since it allows us to sweep a series of planes (Section 11.1.2) through space with a variable (projective) sampling that best matches the sensed image motions (Collins 1996; Szeliski and Golland 1999; Saito and Kanade 1999).

Mapping from one camera to another

What happens when we take two images of a 3D scene from different camera positions or orientations (Figure 2.12a)? Using the full rank 4×4 camera matrix $\tilde{\mathbf{P}} = \tilde{\mathbf{K}}\mathbf{E}$ from (2.64), we can write the projection from world to screen coordinates as

$$\tilde{\mathbf{x}}_0 \sim \tilde{\mathbf{K}}_0 \mathbf{E}_0 \mathbf{p} = \tilde{\mathbf{P}}_0 \mathbf{p}. \quad (2.68)$$

Assuming that we know the z-buffer or disparity value d_0 for a pixel in one image, we can compute the 3D point location \mathbf{p} using

$$\mathbf{p} \sim \mathbf{E}_0^{-1} \tilde{\mathbf{K}}_0^{-1} \tilde{\mathbf{x}}_0 \quad (2.69)$$

and then project it into another image yielding

$$\tilde{\mathbf{x}}_1 \sim \tilde{\mathbf{K}}_1 \mathbf{E}_1 \mathbf{p} = \tilde{\mathbf{K}}_1 \mathbf{E}_1 \mathbf{E}_0^{-1} \tilde{\mathbf{K}}_0^{-1} \tilde{\mathbf{x}}_0 = \tilde{\mathbf{P}}_1 \tilde{\mathbf{P}}_0^{-1} \tilde{\mathbf{x}}_0 = \mathbf{M}_{10} \tilde{\mathbf{x}}_0. \quad (2.70)$$

Unfortunately, we do not usually have access to the depth coordinates of pixels in a regular photographic image. However, for a *planar scene*, as discussed above in (2.66), we can replace the last row of \mathbf{P}_0 in (2.64) with a general *plane equation*, $\hat{\mathbf{n}}_0 \cdot \mathbf{p} + c_0$ that maps points on the plane to $d_0 = 0$ values (Figure 2.12b). Thus, if we set $d_0 = 0$, we can ignore the last column of \mathbf{M}_{10} in (2.70) and also its last row, since we do not care about the final z-buffer depth. The mapping equation (2.70) thus reduces to

$$\tilde{\mathbf{x}}_1 \sim \tilde{\mathbf{H}}_{10} \tilde{\mathbf{x}}_0, \quad (2.71)$$

where $\tilde{\mathbf{H}}_{10}$ is a general 3×3 homography matrix and $\tilde{\mathbf{x}}_1$ and $\tilde{\mathbf{x}}_0$ are now 2D homogeneous coordinates (i.e., 3-vectors) (Szeliski 1996). This justifies the use of the 8-parameter homography as a general alignment model for mosaics of planar scenes (Mann and Picard 1994; Szeliski 1996).

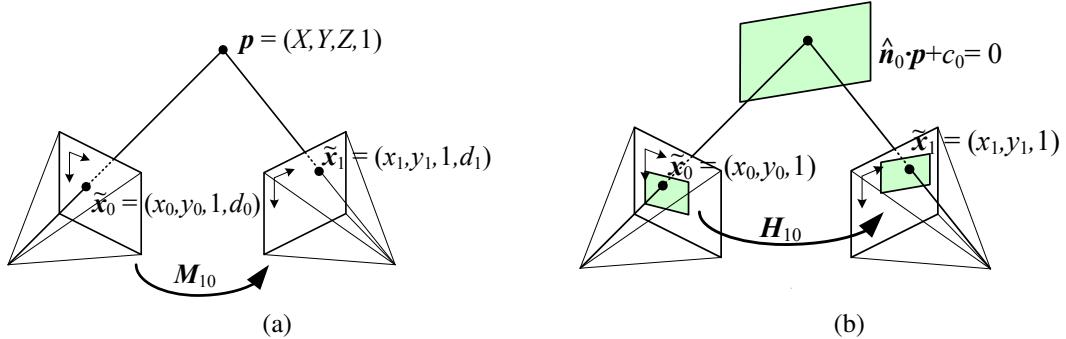


Figure 2.12 A point is projected into two images: (a) relationship between the 3D point coordinate $(X, Y, Z, 1)$ and the 2D projected point $(x, y, 1, d)$; (b) planar homography induced by points all lying on a common plane $\hat{\mathbf{n}}_0 \cdot \mathbf{p} + c_0 = 0$.

The other special case where we do not need to know depth to perform inter-camera mapping is when the camera is undergoing pure rotation (Section 9.1.3), i.e., when $t_0 = t_1$. In this case, we can write

$$\tilde{\mathbf{x}}_1 \sim \mathbf{K}_1 \mathbf{R}_1 \mathbf{R}_0^{-1} \mathbf{K}_0^{-1} \tilde{\mathbf{x}}_0 = \mathbf{K}_1 \mathbf{R}_{10} \mathbf{K}_0^{-1} \tilde{\mathbf{x}}_0, \quad (2.72)$$

which again can be represented with a 3×3 homography. If we assume that the calibration matrices have known aspect ratios and centers of projection (2.59), this homography can be parameterized by the rotation amount and the two unknown focal lengths. This particular formulation is commonly used in image-stitching applications (Section 9.1.3).

Object-centered projection

When working with long focal length lenses, it often becomes difficult to reliably estimate the focal length from image measurements alone. This is because the focal length and the distance to the object are highly correlated and it becomes difficult to tease these two effects apart. For example, the change in scale of an object viewed through a zoom telephoto lens can either be due to a zoom change or a motion towards the user. (This effect was put to dramatic use in some of Alfred Hitchcock's film *Vertigo*, where the simultaneous change of zoom and camera motion produces a disquieting effect.)

This ambiguity becomes clearer if we write out the projection equation corresponding to the simple calibration matrix \mathbf{K} (2.59),

$$x_s = f \frac{\mathbf{r}_x \cdot \mathbf{p} + t_x}{\mathbf{r}_z \cdot \mathbf{p} + t_z} + c_x \quad (2.73)$$

$$y_s = f \frac{\mathbf{r}_y \cdot \mathbf{p} + t_y}{\mathbf{r}_z \cdot \mathbf{p} + t_z} + c_y, \quad (2.74)$$

where \mathbf{r}_x , \mathbf{r}_y , and \mathbf{r}_z are the three rows of \mathbf{R} . If the distance to the object center $t_z \gg \|\mathbf{p}\|$ (the size of the object), the denominator is approximately t_z and the overall scale of the projected object depends on the ratio of f to t_z . It therefore becomes difficult to disentangle these two quantities.

To see this more clearly, let $\eta_z = t_z^{-1}$ and $s = \eta_z f$. We can then re-write the above equations as

$$x_s = s \frac{\mathbf{r}_x \cdot \mathbf{p} + t_x}{1 + \eta_z \mathbf{r}_z \cdot \mathbf{p}} + c_x \quad (2.75)$$

$$y_s = s \frac{\mathbf{r}_y \cdot \mathbf{p} + t_y}{1 + \eta_z \mathbf{r}_z \cdot \mathbf{p}} + c_y \quad (2.76)$$

(Szeliski and Kang 1994; Pighin, Hecker, Lischinski *et al.* 1998). The scale of the projection s can be reliably estimated if we are looking at a known object (i.e., the 3D coordinates \mathbf{p} are known). The inverse distance η_z is now mostly decoupled from the estimates of s and can be estimated from the amount of *foreshortening* as the object rotates. Furthermore, as the lens becomes longer, i.e., the projection model becomes orthographic, there is no need to replace a perspective imaging model with an orthographic one, since the same equation can be used, with $\eta_z \rightarrow 0$ (as opposed to f and t_z both going to infinity). This allows us to form a natural link between orthographic reconstruction techniques such as factorization and their projective/perspective counterparts (Section 7.3).

2.1.6 Lens distortions

The above imaging models all assume that cameras obey a *linear* projection model where straight lines in the world result in straight lines in the image. (This follows as a natural consequence of linear matrix operations being applied to homogeneous coordinates.) Unfortunately, many wide-angle lenses have noticeable *radial distortion*, which manifests itself as a visible curvature in the projection of straight lines. (See Section 2.2.3 for a more detailed discussion of lens optics, including chromatic aberration.) Unless this distortion is taken into account, it becomes impossible to create highly accurate photorealistic reconstructions. For example, image mosaics constructed without taking radial distortion into account will often exhibit blurring due to the mis-registration of corresponding features before pixel blending (Chapter 9).

Fortunately, compensating for radial distortion is not that difficult in practice. For most lenses, a simple quartic model of distortion can produce good results. Let (x_c, y_c) be the pixel coordinates obtained *after* perspective division but *before* scaling by focal length f and shifting by the optical center (c_x, c_y) , i.e.,

$$\begin{aligned} x_c &= \frac{\mathbf{r}_x \cdot \mathbf{p} + t_x}{\mathbf{r}_z \cdot \mathbf{p} + t_z} \\ y_c &= \frac{\mathbf{r}_y \cdot \mathbf{p} + t_y}{\mathbf{r}_z \cdot \mathbf{p} + t_z}. \end{aligned} \quad (2.77)$$

The radial distortion model says that coordinates in the observed images are displaced away (*barrel* distortion) or towards (*pincushion* distortion) the image center by an amount proportional to their radial distance (Figure 2.13a–b).³ The simplest radial distortion models use low-order polynomials, e.g.,

$$\begin{aligned} \hat{x}_c &= x_c(1 + \kappa_1 r_c^2 + \kappa_2 r_c^4) \\ \hat{y}_c &= y_c(1 + \kappa_1 r_c^2 + \kappa_2 r_c^4), \end{aligned} \quad (2.78)$$

³ Anamorphic lenses, which are widely used in feature film production, do not follow this radial distortion model. Instead, they can be thought of, to a first approximation, as inducing different vertical and horizontal scalings, i.e., non-square pixels.



Figure 2.13 Radial lens distortions: (a) barrel, (b) pincushion, and (c) fisheye. The fisheye image spans almost 180° from side-to-side.

where $r_c^2 = x_c^2 + y_c^2$ and κ_1 and κ_2 are called the *radial distortion parameters*.⁴ After the radial distortion step, the final pixel coordinates can be computed using

$$\begin{aligned} x_s &= fx'_c + c_x \\ y_s &= fy'_c + c_y. \end{aligned} \quad (2.79)$$

A variety of techniques can be used to estimate the radial distortion parameters for a given lens, as discussed in Section 6.3.5.

Sometimes the above simplified model does not model the true distortions produced by complex lenses accurately enough (especially at very wide angles). A more complete analytic model also includes *tangential distortions* and *decentering distortions* (Slama 1980), but these distortions are not covered in this book.

Fisheye lenses (Figure 2.13c) require a model that differs from traditional polynomial models of radial distortion. Fisheye lenses behave, to a first approximation, as *equi-distance* projectors of angles away from the optical axis (Xiong and Turkowski 1997), which is the same as the *polar projection* described by Equations (9.22–9.24). Xiong and Turkowski (1997) describe how this model can be extended with the addition of an extra quadratic correction in ϕ and how the unknown parameters (center of projection, scaling factor s , etc.) can be estimated from a set of overlapping fisheye images using a direct (intensity-based) non-linear minimization algorithm.

For even larger, less regular distortions, a parametric distortion model using splines may be necessary (Goshtasby 1989). If the lens does not have a single center of projection, it may become necessary to model the 3D *line* (as opposed to *direction*) corresponding to each pixel separately (Gremban, Thorpe, and Kanade 1988; Champleboux, Lavallée, Sautot *et al.* 1992; Grossberg and Nayar 2001; Sturm and Ramalingam 2004; Tardif, Sturm, Trudeau *et al.* 2009). Some of these techniques are described in more detail in Section 6.3.5, which discusses how to calibrate lens distortions.

⁴ Sometimes the relationship between x_c and \hat{x}_c is expressed the other way around, i.e., $x_c = \hat{x}_c(1 + \kappa_1 \hat{r}_c^2 + \kappa_2 \hat{r}_c^4)$. This is convenient if we map image pixels into (warped) rays by dividing through by f . We can then undistort the rays and have true 3D rays in space.

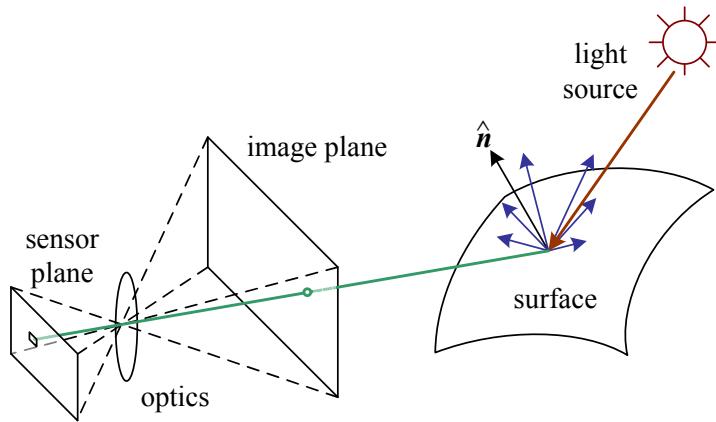


Figure 2.14 A simplified model of photometric image formation. Light is emitted by one or more light sources and is then reflected from an object’s surface. A portion of this light is directed towards the camera. This simplified model ignores multiple reflections, which often occur in real-world scenes.

There is one subtle issue associated with the simple radial distortion model that is often glossed over. We have introduced a non-linearity between the perspective projection and final sensor array projection steps. Therefore, we cannot, in general, post-multiply an arbitrary 3×3 matrix K with a rotation to put it into upper-triangular form and absorb this into the global rotation. However, this situation is not as bad as it may at first appear. For many applications, keeping the simplified diagonal form of (2.59) is still an adequate model. Furthermore, if we correct radial and other distortions to an accuracy where straight lines are preserved, we have essentially converted the sensor back into a linear imager and the previous decomposition still applies.

2.2 Photometric image formation

In modeling the image formation process, we have described how 3D geometric features in the world are projected into 2D features in an image. However, images are not composed of 2D features. Instead, they are made up of discrete color or intensity values. Where do these values come from? How do they relate to the lighting in the environment, surface properties and geometry, camera optics, and sensor properties (Figure 2.14)? In this section, we develop a set of models to describe these interactions and formulate a generative process of image formation. A more detailed treatment of these topics can be found in other textbooks on computer graphics and image synthesis (Glassner 1995; Weyrich, Lawrence, Lensch *et al.* 2008; Foley, van Dam, Feiner *et al.* 1995; Watt 1995; Cohen and Wallace 1993; Sillion and Puech 1994).

2.2.1 Lighting

Images cannot exist without light. To produce an image, the scene must be illuminated with one or more light sources. (Certain modalities such as fluorescent microscopy and X-ray

tomography do not fit this model, but we do not deal with them in this book.) Light sources can generally be divided into point and area light sources.

A point light source originates at a single location in space (e.g., a small light bulb), potentially at infinity (e.g., the sun). (Note that for some applications such as modeling soft shadows (*penumbras*), the sun may have to be treated as an area light source.) In addition to its location, a point light source has an intensity and a color spectrum, i.e., a distribution over wavelengths $L(\lambda)$. The intensity of a light source falls off with the square of the distance between the source and the object being lit, because the same light is being spread over a larger (spherical) area. A light source may also have a directional falloff (dependence), but we ignore this in our simplified model.

Area light sources are more complicated. A simple area light source such as a fluorescent ceiling light fixture with a diffuser can be modeled as a finite rectangular area emitting light equally in all directions (Cohen and Wallace 1993; Sillion and Puech 1994; Glassner 1995). When the distribution is strongly directional, a four-dimensional lightfield can be used instead (Ashdown 1993).

A more complex light distribution that approximates, say, the incident illumination on an object sitting in an outdoor courtyard, can often be represented using an *environment map* (Greene 1986) (originally called a *reflection map* (Blinn and Newell 1976)). This representation maps incident light directions \hat{v} to color values (or wavelengths, λ),

$$L(\hat{v}; \lambda), \quad (2.80)$$

and is equivalent to assuming that all light sources are at infinity. Environment maps can be represented as a collection of cubical faces (Greene 1986), as a single longitude–latitude map (Blinn and Newell 1976), or as the image of a reflecting sphere (Watt 1995). A convenient way to get a rough model of a real-world environment map is to take an image of a reflective mirrored sphere and to unwrap this image onto the desired environment map (Debevec 1998). Watt (1995) gives a nice discussion of environment mapping, including the formulas needed to map directions to pixels for the three most commonly used representations.

2.2.2 Reflectance and shading

When light hits an object’s surface, it is scattered and reflected (Figure 2.15a). Many different models have been developed to describe this interaction. In this section, we first describe the most general form, the bidirectional reflectance distribution function, and then look at some more specialized models, including the diffuse, specular, and Phong shading models. We also discuss how these models can be used to compute the *global illumination* corresponding to a scene.

The Bidirectional Reflectance Distribution Function (BRDF)

The most general model of light scattering is the *bidirectional reflectance distribution function* (BRDF).⁵ Relative to some local coordinate frame on the surface, the BRDF is a four-dimensional function that describes how much of each wavelength arriving at an *incident*

⁵ Actually, even more general models of light transport exist, including some that model spatial variation along the surface, sub-surface scattering, and atmospheric effects—see Section 12.7.1—(Dorsey, Rushmeier, and Sillion 2007; Weyrich, Lawrence, Lensch *et al.* 2008).

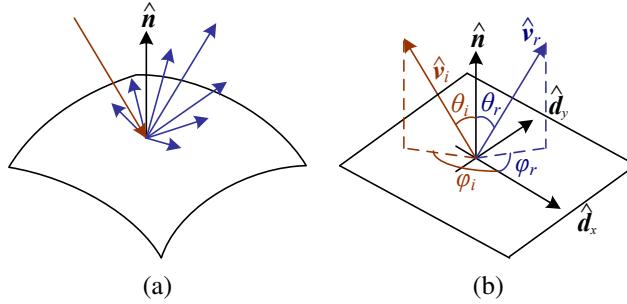


Figure 2.15 (a) Light scatters when it hits a surface. (b) The bidirectional reflectance distribution function (BRDF) $f_r(\theta_i, \phi_i, \theta_r, \phi_r)$ is parameterized by the angles that the incident, \hat{v}_i , and reflected, \hat{v}_r , light ray directions make with the local surface coordinate frame (\hat{d}_x , \hat{d}_y , \hat{n}).

direction \hat{v}_i is emitted in a *reflected* direction \hat{v}_r (Figure 2.15b). The function can be written in terms of the angles of the incident and reflected directions relative to the surface frame as

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r; \lambda). \quad (2.81)$$

The BRDF is *reciprocal*, i.e., because of the physics of light transport, you can interchange the roles of \hat{v}_i and \hat{v}_r and still get the same answer (this is sometimes called *Helmholtz reciprocity*).

Most surfaces are *isotropic*, i.e., there are no preferred directions on the surface as far as light transport is concerned. (The exceptions are *anisotropic* surfaces such as brushed (scratched) aluminum, where the reflectance depends on the light orientation relative to the direction of the scratches.) For an isotropic material, we can simplify the BRDF to

$$f_r(\theta_i, \theta_r, |\phi_r - \phi_i|; \lambda) \text{ or } f_r(\hat{v}_i, \hat{v}_r, \hat{n}; \lambda), \quad (2.82)$$

since the quantities θ_i , θ_r and $\phi_r - \phi_i$ can be computed from the directions \hat{v}_i , \hat{v}_r , and \hat{n} .

To calculate the amount of light exiting a surface point p in a direction \hat{v}_r under a given lighting condition, we integrate the product of the incoming light $L_i(\hat{v}_i; \lambda)$ with the BRDF (some authors call this step a *convolution*). Taking into account the *foreshortening* factor $\cos^+ \theta_i$, we obtain

$$L_r(\hat{v}_r; \lambda) = \int L_i(\hat{v}_i; \lambda) f_r(\hat{v}_i, \hat{v}_r, \hat{n}; \lambda) \cos^+ \theta_i d\hat{v}_i, \quad (2.83)$$

where

$$\cos^+ \theta_i = \max(0, \cos \theta_i). \quad (2.84)$$

If the light sources are discrete (a finite number of point light sources), we can replace the integral with a summation,

$$L_r(\hat{v}_r; \lambda) = \sum_i L_i(\lambda) f_r(\hat{v}_i, \hat{v}_r, \hat{n}; \lambda) \cos^+ \theta_i. \quad (2.85)$$

BRDFs for a given surface can be obtained through physical modeling (Torrance and Sparrow 1967; Cook and Torrance 1982; Glassner 1995), heuristic modeling (Phong 1975), or



Figure 2.16 This close-up of a statue shows both diffuse (smooth shading) and specular (shiny highlight) reflection, as well as darkening in the grooves and creases due to reduced light visibility and interreflections. (Photo courtesy of the Caltech Vision Lab, <http://www.vision.caltech.edu/archive.html>.)

through empirical observation (Ward 1992; Westin, Arvo, and Torrance 1992; Dana, van Ginneken, Nayar *et al.* 1999; Dorsey, Rushmeier, and Sillion 2007; Weyrich, Lawrence, Lensch *et al.* 2008).⁶ Typical BRDFs can often be split into their *diffuse* and *specular* components, as described below.

Diffuse reflection

The diffuse component (also known as *Lambertian* or *matte* reflection) scatters light uniformly in all directions and is the phenomenon we most normally associate with *shading*, e.g., the smooth (non-shiny) variation of intensity with surface normal that is seen when observing a statue (Figure 2.16). Diffuse reflection also often imparts a strong *body color* to the light since it is caused by selective absorption and re-emission of light inside the object’s material (Shafer 1985; Glassner 1995).

While light is scattered uniformly in all directions, i.e., the BRDF is constant,

$$f_d(\hat{\mathbf{v}}_i, \hat{\mathbf{v}}_r, \hat{\mathbf{n}}; \lambda) = f_d(\lambda), \quad (2.86)$$

the amount of light depends on the angle between the incident light direction and the surface normal θ_i . This is because the surface area exposed to a given amount of light becomes larger at oblique angles, becoming completely self-shadowed as the outgoing surface normal points away from the light (Figure 2.17a). (Think about how you orient yourself towards the sun or fireplace to get maximum warmth and how a flashlight projected obliquely against a wall is less bright than one pointing directly at it.) The *shading equation* for diffuse reflection can thus be written as

$$L_d(\hat{\mathbf{v}}_r; \lambda) = \sum_i L_i(\lambda) f_d(\lambda) \cos^+ \theta_i = \sum_i L_i(\lambda) f_d(\lambda) [\hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}]^+, \quad (2.87)$$

⁶ See <http://www1.cs.columbia.edu/CAVE/software/curet/> for a database of some empirically sampled BRDFs.

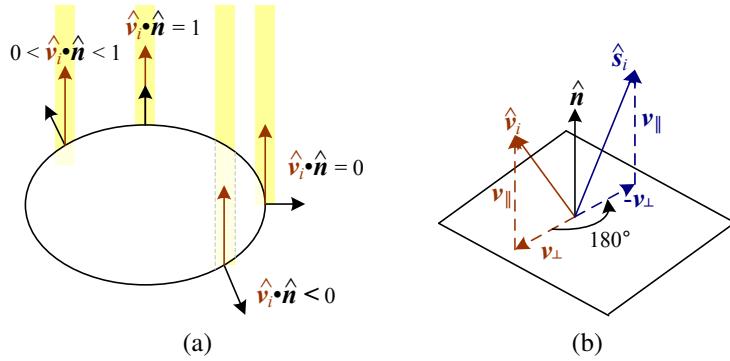


Figure 2.17 (a) The diminution of returned light caused by *foreshortening* depends on $\hat{v}_i \cdot \hat{n}$, the cosine of the angle between the incident light direction \hat{v}_i and the surface normal \hat{n} . (b) Mirror (specular) reflection: The incident light ray direction \hat{v}_i is reflected onto the specular direction \hat{s}_i around the surface normal \hat{n} .

where

$$[\hat{v}_i \cdot \hat{n}]^+ = \max(0, \hat{v}_i \cdot \hat{n}). \quad (2.88)$$

Specular reflection

The second major component of a typical BRDF is *specular* (gloss or highlight) reflection, which depends strongly on the direction of the outgoing light. Consider light reflecting off a mirrored surface (Figure 2.17b). Incident light rays are reflected in a direction that is rotated by 180° around the surface normal \hat{n} . Using the same notation as in Equations (2.29–2.30), we can compute the *specular reflection* direction \hat{s}_i as

$$\hat{s}_i = \mathbf{v}_{\parallel} - \mathbf{v}_{\perp} = (2\hat{n}\hat{n}^T - \mathbf{I})\mathbf{v}_i. \quad (2.89)$$

The amount of light reflected in a given direction \hat{v}_r thus depends on the angle $\theta_s = \cos^{-1}(\hat{v}_r \cdot \hat{s}_i)$ between the view direction \hat{v}_r and the specular direction \hat{s}_i . For example, the Phong (1975) model uses a power of the cosine of the angle,

$$f_s(\theta_s; \lambda) = k_s(\lambda) \cos^{k_e} \theta_s, \quad (2.90)$$

while the Torrance and Sparrow (1967) micro-facet model uses a Gaussian,

$$f_s(\theta_s; \lambda) = k_s(\lambda) \exp(-c_s^2 \theta_s^2). \quad (2.91)$$

Larger exponents k_e (or inverse Gaussian widths c_s) correspond to more specular surfaces with distinct highlights, while smaller exponents better model materials with softer gloss.

Phong shading

Phong (1975) combined the diffuse and specular components of reflection with another term, which he called the *ambient illumination*. This term accounts for the fact that objects are generally illuminated not only by point light sources but also by a general diffuse illumination corresponding to inter-reflection (e.g., the walls in a room) or distant sources, such as the

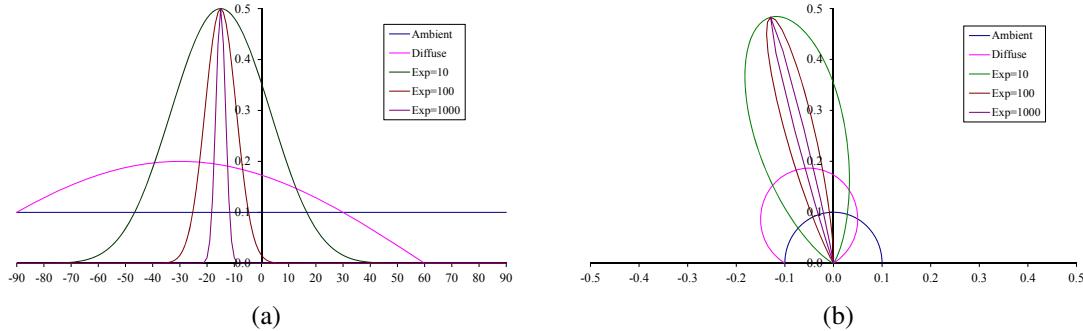


Figure 2.18 Cross-section through a Phong shading model BRDF for a fixed incident illumination direction: (a) component values as a function of angle away from surface normal; (b) polar plot. The value of the Phong exponent k_e is indicated by the “Exp” labels and the light source is at an angle of 30° away from the normal.

blue sky. In the Phong model, the ambient term does not depend on surface orientation, but depends on the color of both the ambient illumination $L_a(\lambda)$ and the object $k_a(\lambda)$,

$$f_a(\lambda) = k_a(\lambda)L_a(\lambda). \quad (2.92)$$

Putting all of these terms together, we arrive at the *Phong shading* model,

$$L_r(\hat{v}_r; \lambda) = k_a(\lambda)L_a(\lambda) + k_d(\lambda) \sum_i L_i(\lambda)[\hat{v}_i \cdot \hat{n}]^+ + k_s(\lambda) \sum_i L_i(\lambda)(\hat{v}_r \cdot \hat{s}_i)^{k_e}. \quad (2.93)$$

Figure 2.18 shows a typical set of Phong shading model components as a function of the angle away from the surface normal (in a plane containing both the lighting direction and the viewer).

Typically, the ambient and diffuse reflection color distributions $k_a(\lambda)$ and $k_d(\lambda)$ are the same, since they are both due to sub-surface scattering (body reflection) inside the surface material (Shafer 1985). The specular reflection distribution $k_s(\lambda)$ is often uniform (white), since it is caused by interface reflections that do not change the light color. (The exception to this are *metallic* materials, such as copper, as opposed to the more common *dielectric* materials, such as plastics.)

The ambient illumination $L_a(\lambda)$ often has a different color cast from the direct light sources $L_i(\lambda)$, e.g., it may be blue for a sunny outdoor scene or yellow for an interior lit with candles or incandescent lights. (The presence of ambient sky illumination in shadowed areas is what often causes shadows to appear bluer than the corresponding lit portions of a scene). Note also that the diffuse component of the Phong model (or of any shading model) depends on the angle of the *incoming* light source \hat{v}_i , while the specular component depends on the relative angle between the viewer v_r and the specular reflection direction \hat{s}_i (which itself depends on the incoming light direction \hat{v}_i and the surface normal \hat{n}).

The Phong shading model has been superseded in terms of physical accuracy by a number of more recently developed models in computer graphics, including the model developed by Cook and Torrance (1982) based on the original micro-facet model of Torrance and Sparrow (1967). Until recently, most computer graphics hardware implemented the Phong model but the recent advent of programmable pixel shaders makes the use of more complex models feasible.

Di-chromatic reflection model

The Torrance and Sparrow (1967) model of reflection also forms the basis of Shafer's (1985) *di-chromatic reflection model*, which states that the apparent color of a uniform material lit from a single source depends on the sum of two terms,

$$L_r(\hat{v}_r; \lambda) = L_i(\hat{v}_r, \hat{v}_i, \hat{n}; \lambda) + L_b(\hat{v}_r, \hat{v}_i, \hat{n}; \lambda) \quad (2.94)$$

$$= c_i(\lambda)m_i(\hat{v}_r, \hat{v}_i, \hat{n}) + c_b(\lambda)m_b(\hat{v}_r, \hat{v}_i, \hat{n}), \quad (2.95)$$

i.e., the radiance of the light reflected at the *interface*, L_i , and the radiance reflected at the *surface body*, L_b . Each of these, in turn, is a simple product between a relative power spectrum $c(\lambda)$, which depends only on wavelength, and a magnitude $m(\hat{v}_r, \hat{v}_i, \hat{n})$, which depends only on geometry. (This model can easily be derived from a generalized version of Phong's model by assuming a single light source and no ambient illumination, and re-arranging terms.) The di-chromatic model has been successfully used in computer vision to segment specular colored objects with large variations in shading (Klinker 1993) and more recently has inspired local two-color models for applications such Bayer pattern demosaicing (Bennett, Uyttendaele, Zitnick *et al.* 2006).

Global illumination (ray tracing and radiosity)

The simple shading model presented thus far assumes that light rays leave the light sources, bounce off surfaces visible to the camera, thereby changing in intensity or color, and arrive at the camera. In reality, light sources can be shadowed by occluders and rays can bounce multiple times around a scene while making their trip from a light source to the camera.

Two methods have traditionally been used to model such effects. If the scene is mostly specular (the classic example being scenes made of glass objects and mirrored or highly polished balls), the preferred approach is *ray tracing* or *path tracing* (Glassner 1995; Akenine-Möller and Haines 2002; Shirley 2005), which follows individual rays from the camera across multiple bounces towards the light sources (or vice versa). If the scene is composed mostly of uniform albedo simple geometry illuminators and surfaces, *radiosity* (*global illumination*) techniques are preferred (Cohen and Wallace 1993; Sillion and Puech 1994; Glassner 1995). Combinations of the two techniques have also been developed (Wallace, Cohen, and Greenberg 1987), as well as more general *light transport* techniques for simulating effects such as the *caustics* cast by rippling water.

The basic ray tracing algorithm associates a light ray with each pixel in the camera image and finds its intersection with the nearest surface. A *primary* contribution can then be computed using the simple shading equations presented previously (e.g., Equation (2.93)) for all light sources that are visible for that surface element. (An alternative technique for computing which surfaces are illuminated by a light source is to compute a *shadow map*, or *shadow buffer*, i.e., a rendering of the scene from the light source's perspective, and then compare the depth of pixels being rendered with the map (Williams 1983; Akenine-Möller and Haines 2002).) Additional *secondary* rays can then be cast along the specular direction towards other objects in the scene, keeping track of any attenuation or color change that the specular reflection induces.

Radiosity works by associating lightness values with rectangular surface areas in the scene (including area light sources). The amount of light interchanged between any two (mutually

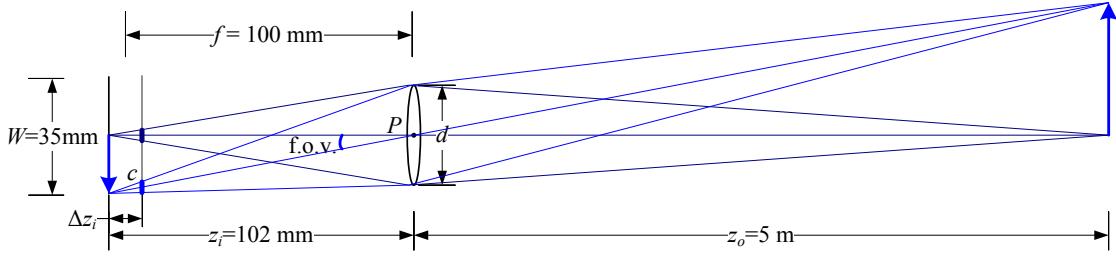


Figure 2.19 A thin lens of focal length f focuses the light from a plane a distance z_o in front of the lens at a distance z_i behind the lens, where $\frac{1}{z_o} + \frac{1}{z_i} = \frac{1}{f}$. If the focal plane (vertical gray line next to c) is moved forward, the images are no longer in focus and the *circle of confusion* c (small thick line segments) depends on the distance of the image plane motion Δz_i relative to the lens aperture diameter d . The field of view (f.o.v.) depends on the ratio between the sensor width W and the focal length f (or, more precisely, the focusing distance z_i , which is usually quite close to f).

visible) areas in the scene can be captured as a *form factor*, which depends on their relative orientation and surface reflectance properties, as well as the $1/r^2$ fall-off as light is distributed over a larger effective sphere the further away it is (Cohen and Wallace 1993; Sillion and Puech 1994; Glassner 1995). A large linear system can then be set up to solve for the final lightness of each area patch, using the light sources as the forcing function (right hand side). Once the system has been solved, the scene can be rendered from any desired point of view. Under certain circumstances, it is possible to recover the global illumination in a scene from photographs using computer vision techniques (Yu, Debevec, Malik *et al.* 1999).

The basic radiosity algorithm does not take into account certain *near field* effects, such as the darkening inside corners and scratches, or the limited ambient illumination caused by partial shadowing from other surfaces. Such effects have been exploited in a number of computer vision algorithms (Nayar, Ikeuchi, and Kanade 1991; Langer and Zucker 1994).

While all of these global illumination effects can have a strong effect on the appearance of a scene, and hence its 3D interpretation, they are not covered in more detail in this book. (But see Section 12.7.1 for a discussion of recovering BRDFs from real scenes and objects.)

2.2.3 Optics

Once the light from a scene reaches the camera, it must still pass through the lens before reaching the sensor (analog film or digital silicon). For many applications, it suffices to treat the lens as an ideal pinhole that simply projects all rays through a common center of projection (Figures 2.8 and 2.9).

However, if we want to deal with issues such as focus, exposure, vignetting, and aberration, we need to develop a more sophisticated model, which is where the study of *optics* comes in (Möller 1988; Hecht 2001; Ray 2002).

Figure 2.19 shows a diagram of the most basic lens model, i.e., the *thin lens* composed of a single piece of glass with very low, equal curvature on both sides. According to the *lens law* (which can be derived using simple geometric arguments on light ray refraction), the relationship between the distance to an object z_o and the distance behind the lens at which a

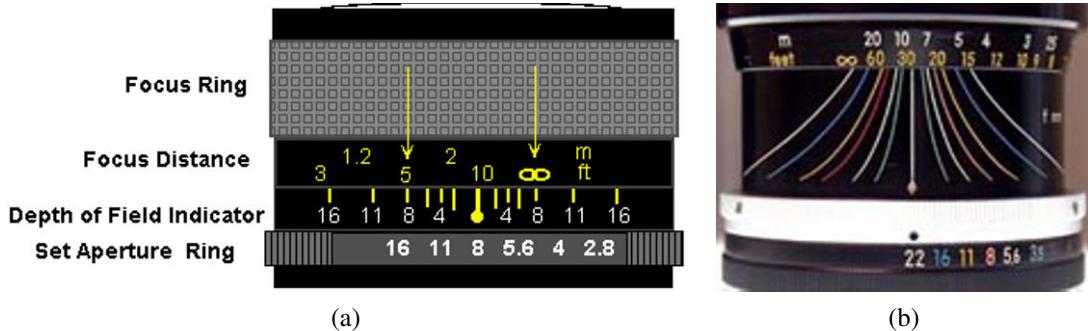


Figure 2.20 Regular and zoom lens depth of field indicators.

focused image is formed z_i can be expressed as

$$\frac{1}{z_o} + \frac{1}{z_i} = \frac{1}{f}, \quad (2.96)$$

where f is called the *focal length* of the lens. If we let $z_o \rightarrow \infty$, i.e., we adjust the lens (move the image plane) so that objects at infinity are in focus, we get $z_i = f$, which is why we can think of a lens of focal length f as being equivalent (to a first approximation) to a pinhole a distance f from the focal plane (Figure 2.10), whose field of view is given by (2.60).

If the focal plane is moved away from its proper in-focus setting of z_i (e.g., by twisting the focus ring on the lens), objects at z_o are no longer in focus, as shown by the gray plane in Figure 2.19. The amount of mis-focus is measured by the *circle of confusion* c (shown as short thick blue line segments on the gray plane).⁷ The equation for the circle of confusion can be derived using similar triangles; it depends on the distance of travel in the focal plane Δz_i relative to the original focus distance z_i and the diameter of the aperture d (see Exercise 2.4).

The allowable depth variation in the scene that limits the circle of confusion to an acceptable number is commonly called the *depth of field* and is a function of both the focus distance and the aperture, as shown diagrammatically by many lens markings (Figure 2.20). Since this depth of field depends on the aperture diameter d , we also have to know how this varies with the commonly displayed *f-number*, which is usually denoted as $f/\#$ or N and is defined as

$$f/\# = N = \frac{f}{d}, \quad (2.97)$$

where the focal length f and the aperture diameter d are measured in the same unit (say, millimeters).

The usual way to write the f-number is to replace the $\#$ in $f/\#$ with the actual number, i.e., $f/1.4, f/2, f/2.8, \dots, f/22$. (Alternatively, we can say $N = 1.4$, etc.) An easy way to interpret these numbers is to notice that dividing the focal length by the f-number gives us the diameter d , so these are just formulas for the aperture diameter.⁸

⁷ If the aperture is not completely circular, e.g., if it is caused by a hexagonal diaphragm, it is sometimes possible to see this effect in the actual blur function (Levin, Fergus, Durand *et al.* 2007; Joshi, Szeliski, and Kriegman 2008) or in the “glints” that are seen when shooting into the sun.

⁸ This also explains why, with zoom lenses, the f-number varies with the current zoom (focal length) setting.

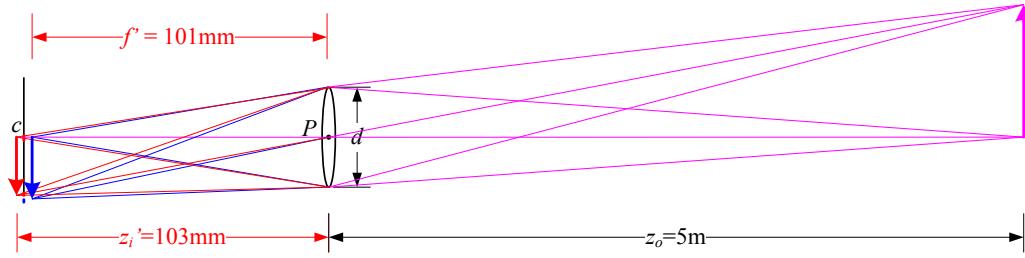


Figure 2.21 In a lens subject to *chromatic aberration*, light at different wavelengths (e.g., the red and blur arrows) is focused with a different focal length f' and hence a different depth z'_i , resulting in both a geometric (in-plane) displacement and a loss of focus.

Notice that the usual progression for f-numbers is in *full stops*, which are multiples of $\sqrt{2}$, since this corresponds to doubling the area of the entrance pupil each time a smaller f-number is selected. (This doubling is also called changing the exposure by one *exposure value* or EV. It has the same effect on the amount of light reaching the sensor as doubling the exposure duration, e.g., from $1/125$ to $1/250$, see Exercise 2.5.)

Now that you know how to convert between f-numbers and aperture diameters, you can construct your own plots for the depth of field as a function of focal length f , circle of confusion c , and focus distance z_o , as explained in Exercise 2.4 and see how well these match what you observe on actual lenses, such as those shown in Figure 2.20.

Of course, real lenses are not infinitely thin and therefore suffer from geometric aberrations, unless compound elements are used to correct for them. The classic five *Seidel aberrations*, which arise when using *third-order optics*, include spherical aberration, coma, astigmatism, curvature of field, and distortion (Möller 1988; Hecht 2001; Ray 2002).

Chromatic aberration

Because the index of refraction for glass varies slightly as a function of wavelength, simple lenses suffer from *chromatic aberration*, which is the tendency for light of different colors to focus at slightly different distances (and hence also with slightly different magnification factors), as shown in Figure 2.21. The wavelength-dependent magnification factor, i.e., the *transverse chromatic aberration*, can be modeled as a per-color radial distortion (Section 2.1.6) and, hence, calibrated using the techniques described in Section 6.3.5. The wavelength-dependent blur caused by *longitudinal chromatic aberration* can be calibrated using techniques described in Section 10.1.4. Unfortunately, the blur induced by longitudinal aberration can be harder to undo, as higher frequencies can get strongly attenuated and hence hard to recover.

In order to reduce chromatic and other kinds of aberrations, most photographic lenses today are *compound lenses* made of different glass elements (with different coatings). Such lenses can no longer be modeled as having a single *nodal point* P through which all of the rays must pass (when approximating the lens with a pinhole model). Instead, these lenses have both a *front nodal point*, through which the rays enter the lens, and a *rear nodal point*, through which they leave on their way to the sensor. In practice, only the location of the front

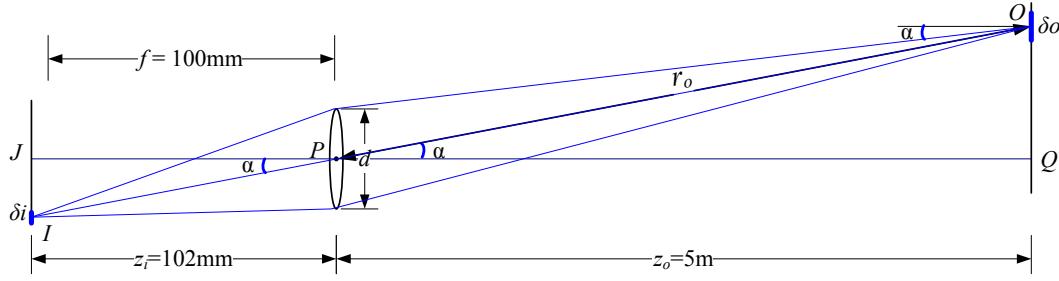


Figure 2.22 The amount of light hitting a pixel of surface area δi depends on the square of the ratio of the aperture diameter d to the focal length f , as well as the fourth power of the off-axis angle α cosine, $\cos^4 \alpha$.

nodal point is of interest when performing careful camera calibration, e.g., when determining the point around which to rotate to capture a parallax-free panorama (see Section 9.1.3).

Not all lenses, however, can be modeled as having a single nodal point. In particular, very wide-angle lenses such as fisheye lenses (Section 2.1.6) and certain *catadioptric* imaging systems consisting of lenses and curved mirrors (Baker and Nayar 1999) do not have a single point through which all of the acquired light rays pass. In such cases, it is preferable to explicitly construct a mapping function (look-up table) between pixel coordinates and 3D rays in space (Gremban, Thorpe, and Kanade 1988; Champleboux, Lavallée, Sautot *et al.* 1992; Grossberg and Nayar 2001; Sturm and Ramalingam 2004; Tardif, Sturm, Trudeau *et al.* 2009), as mentioned in Section 2.1.6.

Vignetting

Another property of real-world lenses is *vignetting*, which is the tendency for the brightness of the image to fall off towards the edge of the image.

Two kinds of phenomena usually contribute to this effect (Ray 2002). The first is called *natural vignetting* and is due to the foreshortening in the object surface, projected pixel, and lens aperture, as shown in Figure 2.22. Consider the light leaving the object surface patch of size δo located at an *off-axis angle* α . Because this patch is foreshortened with respect to the camera lens, the amount of light reaching the lens is reduced by a factor $\cos \alpha$. The amount of light reaching the lens is also subject to the usual $1/r^2$ fall-off; in this case, the distance $r_o = z_o / \cos \alpha$. The actual area of the aperture through which the light passes is foreshortened by an additional factor $\cos \alpha$, i.e., the aperture as seen from point O is an ellipse of dimensions $d \times d \cos \alpha$. Putting all of these factors together, we see that the amount of light leaving O and passing through the aperture on its way to the image pixel located at I is proportional to

$$\frac{\delta o \cos \alpha}{r_o^2} \pi \left(\frac{d}{2} \right)^2 \cos \alpha = \delta o \frac{\pi}{4} \frac{d^2}{z_o^2} \cos^4 \alpha. \quad (2.98)$$

Since triangles ΔOPQ and ΔIPJ are similar, the projected areas of the object surface δo and image pixel δi are in the same (squared) ratio as $z_o : z_i$,

$$\frac{\delta o}{\delta i} = \frac{z_o^2}{z_i^2}. \quad (2.99)$$

Putting these together, we obtain the final relationship between the amount of light reaching pixel i and the aperture diameter d , the focusing distance $z_i \approx f$, and the off-axis angle α ,

$$\delta o \frac{\pi}{4} \frac{d^2}{z_o^2} \cos^4 \alpha = \delta i \frac{\pi}{4} \frac{d^2}{z_i^2} \cos^4 \alpha \approx \delta i \frac{\pi}{4} \left(\frac{d}{f} \right)^2 \cos^4 \alpha, \quad (2.100)$$

which is called the *fundamental radiometric relation* between the scene radiance L and the light (irradiance) E reaching the pixel sensor,

$$E = L \frac{\pi}{4} \left(\frac{d}{f} \right)^2 \cos^4 \alpha, \quad (2.101)$$

(Horn 1986; Nalwa 1993; Hecht 2001; Ray 2002). Notice in this equation how the amount of light depends on the pixel surface area (which is why the smaller sensors in point-and-shoot cameras are so much noisier than digital single lens reflex (SLR) cameras), the inverse square of the f-stop $N = f/d$ (2.97), and the fourth power of the $\cos^4 \alpha$ off-axis fall-off, which is the natural vignetting term.

The other major kind of vignetting, called *mechanical vignetting*, is caused by the internal occlusion of rays near the periphery of lens elements in a compound lens, and cannot easily be described mathematically without performing a full ray-tracing of the actual lens design.⁹ However, unlike natural vignetting, mechanical vignetting can be decreased by reducing the camera aperture (increasing the f-number). It can also be calibrated (along with natural vignetting) using special devices such as integrating spheres, uniformly illuminated targets, or camera rotation, as discussed in Section 10.1.3.

2.3 The digital camera

After starting from one or more light sources, reflecting off one or more surfaces in the world, and passing through the camera's optics (lenses), light finally reaches the imaging sensor. How are the photons arriving at this sensor converted into the digital (R, G, B) values that we observe when we look at a digital image? In this section, we develop a simple model that accounts for the most important effects such as exposure (gain and shutter speed), non-linear mappings, sampling and aliasing, and noise. Figure 2.23, which is based on camera models developed by Healey and Kondepudy (1994); Tsin, Ramesh, and Kanade (2001); Liu, Szeliski, Kang *et al.* (2008), shows a simple version of the processing stages that occur in modern digital cameras. Chakrabarti, Scharstein, and Zickler (2009) developed a sophisticated 24-parameter model that is an even better match to the processing performed in today's cameras.

Light falling on an imaging sensor is usually picked up by an *active sensing area*, integrated for the duration of the exposure (usually expressed as the shutter speed in a fraction of a second, e.g., $\frac{1}{125}$, $\frac{1}{60}$, $\frac{1}{30}$), and then passed to a set of *sense amplifiers*. The two main kinds of sensor used in digital still and video cameras today are charge-coupled device (CCD) and complementary metal oxide on silicon (CMOS).

In a CCD, photons are accumulated in each active *well* during the exposure time. Then, in a *transfer* phase, the charges are transferred from well to well in a kind of "bucket brigade"

⁹ There are some empirical models that work well in practice (Kang and Weiss 2000; Zheng, Lin, and Kang 2006).

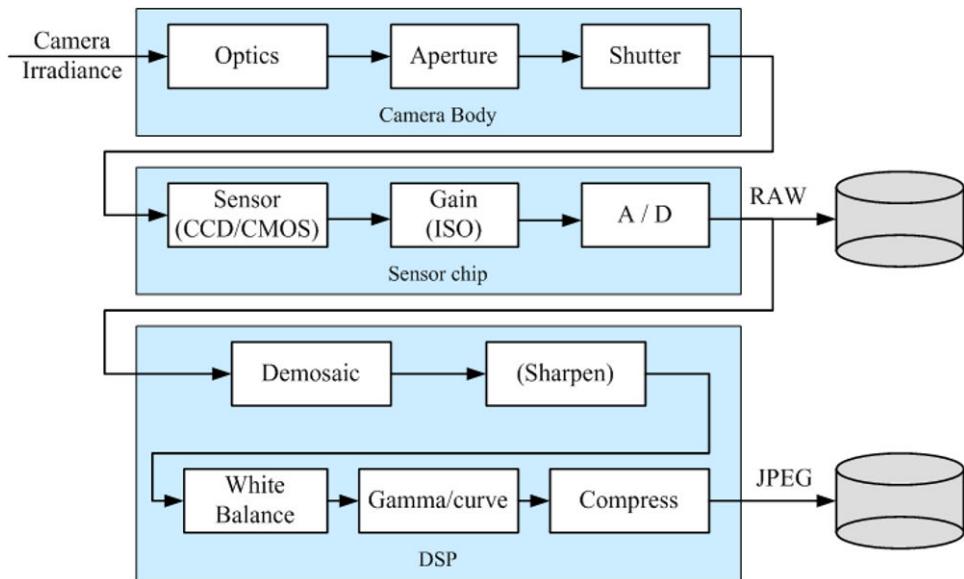


Figure 2.23 Image sensing pipeline, showing the various sources of noise as well as typical digital post-processing steps.

until they are deposited at the sense amplifiers, which amplify the signal and pass it to an analog-to-digital converter (ADC).¹⁰ Older CCD sensors were prone to *blooming*, when charges from one over-exposed pixel spilled into adjacent ones, but most newer CCDs have anti-blooming technology (“troughs” into which the excess charge can spill).

In CMOS, the photons hitting the sensor directly affect the conductivity (or gain) of a photodetector, which can be selectively gated to control exposure duration, and locally amplified before being read out using a multiplexing scheme. Traditionally, CCD sensors outperformed CMOS in quality sensitive applications, such as digital SLRs, while CMOS was better for low-power applications, but today CMOS is used in most digital cameras.

The main factors affecting the performance of a digital image sensor are the shutter speed, sampling pitch, fill factor, chip size, analog gain, sensor noise, and the resolution (and quality) of the analog-to-digital converter. Many of the actual values for these parameters can be read from the EXIF tags embedded with digital images, while others can be obtained from the camera manufacturers’ specification sheets or from camera review or calibration Web sites.¹¹

Shutter speed. The shutter speed (exposure time) directly controls the amount of light reaching the sensor and, hence, determines if images are under- or over-exposed. (For bright scenes, where a large aperture or slow shutter speed are desired to get a shallow depth of field or motion blur, *neutral density filters* are sometimes used by photographers.) For dynamic scenes, the shutter speed also determines the amount of *motion blur* in the resulting picture.

¹⁰ In digital still cameras, a complete frame is captured and then read out sequentially at once. However, if video is being captured, a *rolling shutter*, which exposes and transfers each line separately, is often used. In older video cameras, the even fields (lines) were scanned first, followed by the odd fields, in a process that is called *interlacing*.

¹¹ <http://www.clarkvision.com/imagedetail/digital.sensor.performance.summary/>.

Usually, a higher shutter speed (less motion blur) makes subsequent analysis easier (see Section 10.3 for techniques to remove such blur). However, when video is being captured for display, some motion blur may be desirable to avoid stroboscopic effects.

Sampling pitch. The sampling pitch is the physical spacing between adjacent sensor cells on the imaging chip. A sensor with a smaller sampling pitch has a higher *sampling density* and hence provides a higher *resolution* (in terms of pixels) for a given active chip area. However, a smaller pitch also means that each sensor has a smaller area and cannot accumulate as many photons; this makes it not as *light sensitive* and more prone to noise.

Fill factor. The fill factor is the active sensing area size as a fraction of the theoretically available sensing area (the product of the horizontal and vertical sampling pitches). Higher fill factors are usually preferable, as they result in more light capture and less *aliasing* (see Section 2.3.1). However, this must be balanced with the need to place additional electronics between the active sense areas. The fill factor of a camera can be determined empirically using a photometric camera calibration process (see Section 10.1.4).

Chip size. Video and point-and-shoot cameras have traditionally used small chip areas ($\frac{1}{4}$ -inch to $\frac{1}{2}$ -inch sensors¹²), while digital SLR cameras try to come closer to the traditional size of a 35mm film frame.¹³ When overall device size is not important, having a larger chip size is preferable, since each sensor cell can be more photo-sensitive. (For compact cameras, a smaller chip means that all of the optics can be shrunk down proportionately.) However, larger chips are more expensive to produce, not only because fewer chips can be packed into each wafer, but also because the probability of a chip defect goes up linearly with the chip area.

Analog gain. Before analog-to-digital conversion, the sensed signal is usually boosted by a *sense amplifier*. In video cameras, the gain on these amplifiers was traditionally controlled by *automatic gain control* (AGC) logic, which would adjust these values to obtain a good overall exposure. In newer digital still cameras, the user now has some additional control over this gain through the *ISO setting*, which is typically expressed in ISO standard units such as 100, 200, or 400. Since the automated exposure control in most cameras also adjusts the aperture and shutter speed, setting the ISO manually removes one degree of freedom from the camera's control, just as manually specifying aperture and shutter speed does. In theory, a higher gain allows the camera to perform better under low light conditions (less motion blur due to long exposure times when the aperture is already maxed out). In practice, however, higher ISO settings usually amplify the *sensor noise*.

¹² These numbers refer to the “tube diameter” of the old vidicon tubes used in video cameras (http://www.dpreview.com/learn/?/Glossary/Camera_System/sensor_sizes_01.htm). The 1/2.5” sensor on the Canon SD800 camera actually measures 5.76mm × 4.29mm, i.e., a sixth of the size (on side) of a 35mm full-frame (36mm × 24mm) DSLR sensor.

¹³ When a DSLR chip does not fill the 35mm full frame, it results in a *multiplier effect* on the lens focal length. For example, a chip that is only 0.6 the dimension of a 35mm frame will make a 50mm lens image the same angular extent as a $50/0.6 = 50 \times 1.6 = 80$ mm lens, as demonstrated in (2.60).

Sensor noise. Throughout the whole sensing process, noise is added from various sources, which may include *fixed pattern noise*, *dark current noise*, *shot noise*, *amplifier noise* and *quantization noise* (Healey and Kondepudy 1994; Tsin, Ramesh, and Kanade 2001). The final amount of noise present in a sampled image depends on all of these quantities, as well as the incoming light (controlled by the scene radiance and aperture), the exposure time, and the sensor gain. Also, for low light conditions where the noise is due to low photon counts, a Poisson model of noise may be more appropriate than a Gaussian model.

As discussed in more detail in Section 10.1.1, Liu, Szeliski, Kang *et al.* (2008) use this model, along with an empirical database of camera response functions (CRFs) obtained by Grossberg and Nayar (2004), to estimate the *noise level function* (NLF) for a given image, which predicts the overall noise variance at a given pixel as a function of its brightness (a separate NLF is estimated for each color channel). An alternative approach, when you have access to the camera before taking pictures, is to pre-calibrate the NLF by taking repeated shots of a scene containing a variety of colors and luminances, such as the Macbeth Color Chart shown in Figure 10.3b (McCamy, Marcus, and Davidson 1976). (When estimating the variance, be sure to throw away or downweight pixels with large gradients, as small shifts between exposures will affect the sensed values at such pixels.) Unfortunately, the pre-calibration process may have to be repeated for different exposure times and gain settings because of the complex interactions occurring within the sensing system.

In practice, most computer vision algorithms, such as image denoising, edge detection, and stereo matching, all benefit from at least a rudimentary estimate of the noise level. Barring the ability to pre-calibrate the camera or to take repeated shots of the same scene, the simplest approach is to look for regions of near-constant value and to estimate the noise variance in such regions (Liu, Szeliski, Kang *et al.* 2008).

ADC resolution. The final step in the analog processing chain occurring within an imaging sensor is the *analog to digital conversion* (ADC). While a variety of techniques can be used to implement this process, the two quantities of interest are the *resolution* of this process (how many bits it yields) and its noise level (how many of these bits are useful in practice). For most cameras, the number of bits quoted (eight bits for compressed JPEG images and a nominal 16 bits for the RAW formats provided by some DSLRs) exceeds the actual number of usable bits. The best way to tell is to simply calibrate the noise of a given sensor, e.g., by taking repeated shots of the same scene and plotting the estimated noise as a function of brightness (Exercise 2.6).

Digital post-processing. Once the irradiance values arriving at the sensor have been converted to digital bits, most cameras perform a variety of *digital signal processing* (DSP) operations to enhance the image before compressing and storing the pixel values. These include color filter array (CFA) demosaicing, white point setting, and mapping of the luminance values through a *gamma function* to increase the perceived dynamic range of the signal. We cover these topics in Section 2.3.2 but, before we do, we return to the topic of aliasing, which was mentioned in connection with sensor array fill factors.

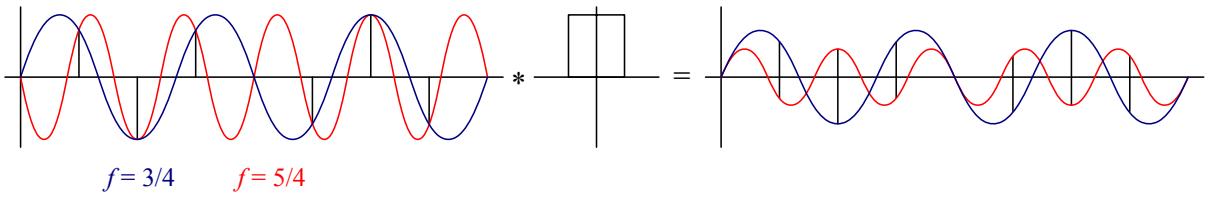


Figure 2.24 Aliasing of a one-dimensional signal: The blue sine wave at $f = 3/4$ and the red sine wave at $f = 5/4$ have the same digital samples, when sampled at $f = 2$. Even after convolution with a 100% fill factor box filter, the two signals, while no longer of the same magnitude, are still aliased in the sense that the sampled red signal looks like an inverted lower magnitude version of the blue signal. (The image on the right is scaled up for better visibility. The actual sine magnitudes are 30% and -18% of their original values.)

2.3.1 Sampling and aliasing

What happens when a field of light impinging on the image sensor falls onto the active sense areas in the imaging chip? The photons arriving at each active cell are integrated and then digitized. However, if the fill factor on the chip is small and the signal is not otherwise *band-limited*, visually unpleasing aliasing can occur.

To explore the phenomenon of aliasing, let us first look at a one-dimensional signal (Figure 2.24), in which we have two sine waves, one at a frequency of $f = 3/4$ and the other at $f = 5/4$. If we sample these two signals at a frequency of $f = 2$, we see that they produce the same samples (shown in black), and so we say that they are *aliased*.¹⁴ Why is this a bad effect? In essence, we can no longer reconstruct the original signal, since we do not know which of the two original frequencies was present.

In fact, Shannon's Sampling Theorem shows that the minimum sampling (Oppenheim and Schafer 1996; Oppenheim, Schafer, and Buck 1999) rate required to reconstruct a signal from its instantaneous samples must be at least twice the highest frequency,¹⁵

$$f_s \geq 2f_{\max}. \quad (2.102)$$

The maximum frequency in a signal is known as the *Nyquist frequency* and the inverse of the minimum sampling frequency $r_s = 1/f_s$ is known as the *Nyquist rate*.

However, you may ask, since an imaging chip actually *averages* the light field over a finite area, are the results on point sampling still applicable? Averaging over the sensor area does tend to attenuate some of the higher frequencies. However, even if the fill factor is 100%, as in the right image of Figure 2.24, frequencies above the Nyquist limit (half the sampling frequency) still produce an aliased signal, although with a smaller magnitude than the corresponding band-limited signals.

A more convincing argument as to why aliasing is bad can be seen by downsampling a signal using a poor quality filter such as a box (square) filter. Figure 2.25 shows a high-frequency *chirp* image (so called because the frequencies increase over time), along with the results of sampling it with a 25% fill-factor area sensor, a 100% fill-factor sensor, and a high-

¹⁴ An alias is an alternate name for someone, so the sampled signal corresponds to two different *aliases*.

¹⁵ The actual theorem states that f_s must be at least twice the signal *bandwidth* but, since we are not dealing with modulated signals such as radio waves during image capture, the maximum frequency suffices.

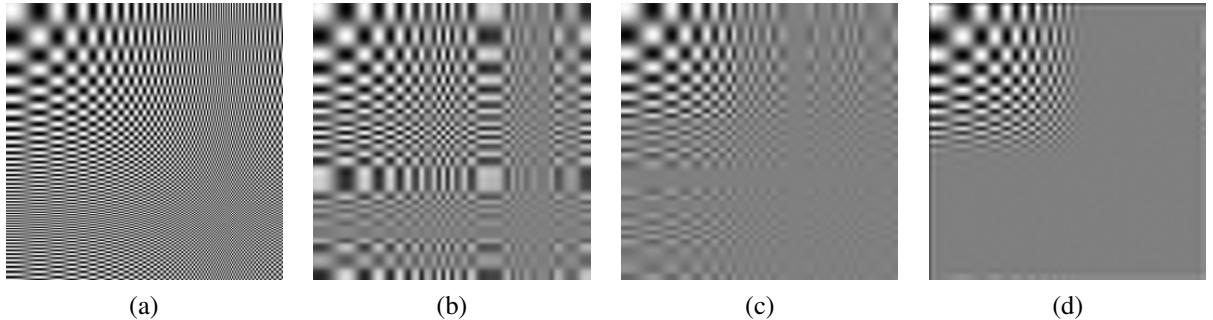


Figure 2.25 Aliasing of a two-dimensional signal: (a) original full-resolution image; (b) downsampled $4\times$ with a 25% fill factor box filter; (c) downsampled $4\times$ with a 100% fill factor box filter; (d) downsampled $4\times$ with a high-quality 9-tap filter. Notice how the higher frequencies are aliased into visible frequencies with the lower quality filters, while the 9-tap filter completely removes these higher frequencies.

quality 9-tap filter. Additional examples of downsampling (*decimation*) filters can be found in Section 3.5.2 and Figure 3.30.

The best way to predict the amount of aliasing that an imaging system (or even an image processing algorithm) will produce is to estimate the *point spread function* (PSF), which represents the response of a particular pixel sensor to an ideal point light source. The PSF is a combination (convolution) of the blur induced by the optical system (lens) and the finite integration area of a chip sensor.¹⁶

If we know the blur function of the lens and the fill factor (sensor area shape and spacing) for the imaging chip (plus, optionally, the response of the anti-aliasing filter), we can convolve these (as described in Section 3.2) to obtain the PSF. Figure 2.26a shows the one-dimensional cross-section of a PSF for a lens whose blur function is assumed to be a disc of a radius equal to the pixel spacing s plus a sensing chip whose horizontal fill factor is 80%. Taking the Fourier transform of this PSF (Section 3.4), we obtain the *modulation transfer function* (MTF), from which we can estimate the amount of aliasing as the area of the Fourier magnitude outside the $f \leq f_s$ Nyquist frequency.¹⁷ If we de-focus the lens so that the blur function has a radius of $2s$ (Figure 2.26c), we see that the amount of aliasing decreases significantly, but so does the amount of image detail (frequencies closer to $f = f_s$).

Under laboratory conditions, the PSF can be estimated (to pixel precision) by looking at a point light source such as a pin hole in a black piece of cardboard lit from behind. However, this PSF (the actual image of the pin hole) is only accurate to a pixel resolution and, while it can model larger blur (such as blur caused by defocus), it cannot model the sub-pixel shape of the PSF and predict the amount of aliasing. An alternative technique, described in Section 10.1.4, is to look at a calibration pattern (e.g., one consisting of slanted step edges (Reichenbach, Park, and Narayanswamy 1991; Williams and Burns 2001; Joshi, Szeliski, and Kriegman 2008)) whose ideal appearance can be re-synthesized to sub-pixel precision.

In addition to occurring during image acquisition, aliasing can also be introduced in var-

¹⁶ Imaging chips usually interpose an optical *anti-aliasing filter* just before the imaging chip to reduce or control the amount of aliasing.

¹⁷ The complex Fourier transform of the PSF is actually called the *optical transfer function* (OTF) (Williams 1999). Its magnitude is called the *modulation transfer function* (MTF) and its phase is called the *phase transfer function* (PTF).

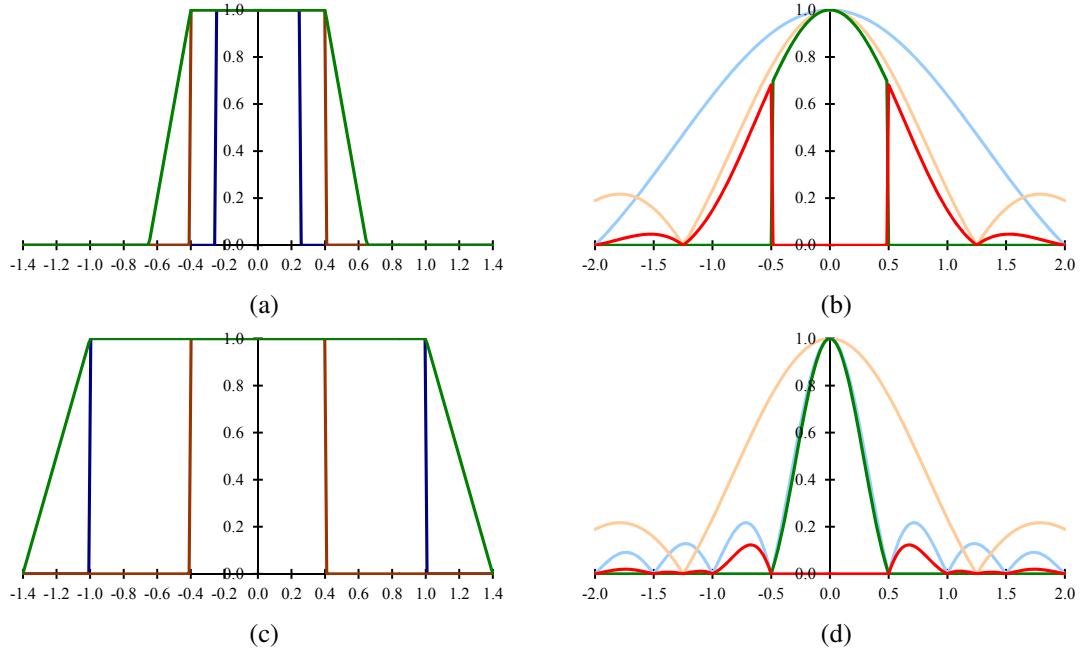


Figure 2.26 Sample point spread functions (PSF): The diameter of the blur disc (blue) in (a) is equal to half the pixel spacing, while the diameter in (c) is twice the pixel spacing. The horizontal fill factor of the sensing chip is 80% and is shown in brown. The convolution of these two kernels gives the point spread function, shown in green. The Fourier response of the PSF (the MTF) is plotted in (b) and (d). The area above the Nyquist frequency where aliasing occurs is shown in red.

ious image processing operations, such as resampling, upsampling, and downsampling. Sections 3.4 and 3.5.2 discuss these issues and show how careful selection of filters can reduce the amount of aliasing that operations inject.

2.3.2 Color

In Section 2.2, we saw how lighting and surface reflections are functions of wavelength. When the incoming light hits the imaging sensor, light from different parts of the spectrum is somehow integrated into the discrete red, green, and blue (RGB) color values that we see in a digital image. How does this process work and how can we analyze and manipulate color values?

You probably recall from your childhood days the magical process of mixing paint colors to obtain new ones. You may recall that blue+yellow makes green, red+blue makes purple, and red+green makes brown. If you revisited this topic at a later age, you may have learned that the proper *subtractive* primaries are actually cyan (a light blue-green), magenta (pink), and yellow (Figure 2.27b), although black is also often used in four-color printing (CMYK). (If you ever subsequently took any painting classes, you learned that colors can have even more fanciful names, such as alizarin crimson, cerulean blue, and chartreuse.) The subtractive colors are called subtractive because pigments in the paint absorb certain wavelengths in the color spectrum.

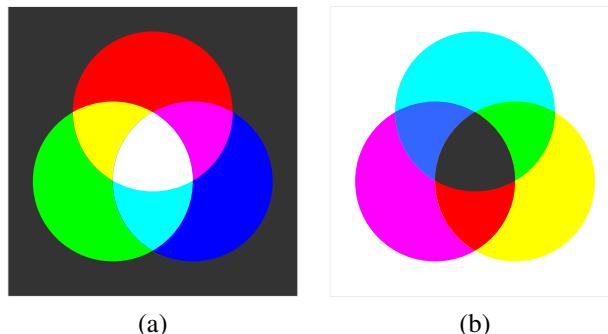


Figure 2.27 Primary and secondary colors: (a) additive colors red, green, and blue can be mixed to produce cyan, magenta, yellow, and white; (b) subtractive colors cyan, magenta, and yellow can be mixed to produce red, green, blue, and black.

Later on, you may have learned about the *additive* primary colors (red, green, and blue) and how they can be added (with a slide projector or on a computer monitor) to produce cyan, magenta, yellow, white, and all the other colors we typically see on our TV sets and monitors (Figure 2.27a).

Through what process is it possible for two different colors, such as red and green, to interact to produce a third color like yellow? Are the wavelengths somehow mixed up to produce a new wavelength?

You probably know that the correct answer has nothing to do with physically mixing wavelengths. Instead, the existence of three primaries is a result of the *tri-stimulus* (or *trichromatic*) nature of the human visual system, since we have three different kinds of cone, each of which responds selectively to a different portion of the color spectrum (Glassner 1995; Wyszecki and Stiles 2000; Fairchild 2005; Reinhard, Ward, Pattanaik *et al.* 2005; Livingstone 2008).¹⁸ Note that for machine vision applications, such as remote sensing and terrain classification, it is preferable to use many more wavelengths. Similarly, surveillance applications can often benefit from sensing in the near-infrared (NIR) range.

CIE RGB and XYZ

To test and quantify the tri-chromatic theory of perception, we can attempt to reproduce all *monochromatic* (single wavelength) colors as a mixture of three suitably chosen primaries. (Pure wavelength light can be obtained using either a prism or specially manufactured color filters.) In the 1930s, the Commission Internationale d'Eclairage (CIE) standardized the RGB representation by performing such *color matching* experiments using the primary colors of red (700.0nm wavelength), green (546.1nm), and blue (435.8nm).

Figure 2.28 shows the results of performing these experiments with a *standard observer*, i.e., averaging perceptual results over a large number of subjects. You will notice that for certain pure spectra in the blue–green range, a *negative* amount of red light has to be added, i.e., a certain amount of red has to be added to the color being matched in order to get a color match. These results also provided a simple explanation for the existence of *metamers*, which are colors with different spectra that are perceptually indistinguishable. Note that two fabrics

¹⁸ See also Mark Fairchild's Web page, http://www.cis.rit.edu/fairchild/WhyIsColor/books_links.html.

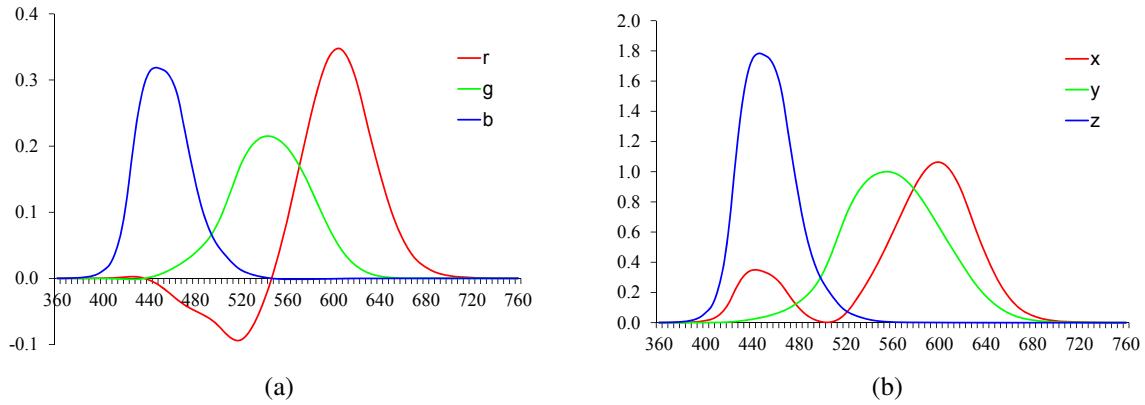


Figure 2.28 Standard CIE color matching functions: (a) $\bar{r}(\lambda)$, $\bar{g}(\lambda)$, $\bar{b}(\lambda)$ color spectra obtained from matching pure colors to the R=700.0nm, G=546.1nm, and B=435.8nm primaries; (b) $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, $\bar{z}(\lambda)$ color matching functions, which are linear combinations of the $(\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda))$ spectra.

or paint colors that are metamers under one light may no longer be so under different lighting.

Because of the problem associated with mixing negative light, the CIE also developed a new color space called XYZ, which contains all of the pure spectral colors within its positive octant. (It also maps the Y axis to the *luminance*, i.e., perceived relative brightness, and maps pure white to a diagonal (equal-valued) vector.) The transformation from RGB to XYZ is given by

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \frac{1}{0.17697} \begin{bmatrix} 0.49 & 0.31 & 0.20 \\ 0.17697 & 0.81240 & 0.01063 \\ 0.00 & 0.01 & 0.99 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \quad (2.103)$$

While the official definition of the CIE XYZ standard has the matrix normalized so that the Y value corresponding to pure red is 1, a more commonly used form is to omit the leading fraction, so that the second row adds up to one, i.e., the RGB triplet (1, 1, 1) maps to a Y value of 1. Linearly blending the $(\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda))$ curves in Figure 2.28a according to (2.103), we obtain the resulting $(\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda))$ curves shown in Figure 2.28b. Notice how all three spectra (color matching functions) now have only positive values and how the $\bar{y}(\lambda)$ curve matches that of the luminance perceived by humans.

If we divide the XYZ values by the sum of X+Y+Z, we obtain the *chromaticity coordinates*

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z}, \quad (2.104)$$

which sum up to 1. The chromaticity coordinates discard the absolute intensity of a given color sample and just represent its pure color. If we sweep the monochromatic color λ parameter in Figure 2.28b from $\lambda = 380$ nm to $\lambda = 800$ nm, we obtain the familiar *chromaticity diagram* shown in Figure 2.29. This figure shows the (x, y) value for every color value perceivable by most humans. (Of course, the CMYK reproduction process in this book does not actually span the whole gamut of perceivable colors.) The outer curved rim represents where

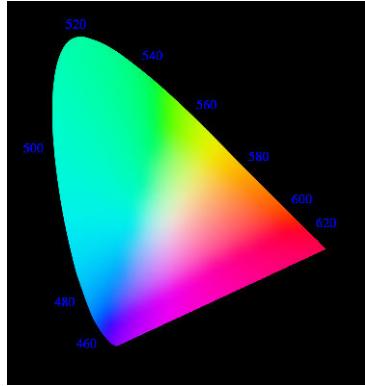


Figure 2.29 CIE chromaticity diagram, showing colors and their corresponding (x, y) values. Pure spectral colors are arranged around the outside of the curve.

all of the pure monochromatic color values map in (x, y) space, while the lower straight line, which connects the two endpoints, is known as the *purple line*.

A convenient representation for color values, when we want to tease apart luminance and chromaticity, is therefore Yxy (luminance plus the two most distinctive chrominance components).

L*a*b* color space

While the XYZ color space has many convenient properties, including the ability to separate luminance from chrominance, it does not actually predict how well humans perceive *differences* in color or luminance.

Because the response of the human visual system is roughly logarithmic (we can perceive *relative* luminance differences of about 1%), the CIE defined a non-linear re-mapping of the XYZ space called L*a*b* (also sometimes called CIELAB), where differences in luminance or chrominance are more perceptually uniform.¹⁹

The L* component of *lightness* is defined as

$$L^* = 116f\left(\frac{Y}{Y_n}\right), \quad (2.105)$$

where Y_n is the luminance value for nominal white (Fairchild 2005) and

$$f(t) = \begin{cases} t^{1/3} & t > \delta^3 \\ t/(3\delta^2) + 2\delta/3 & \text{else,} \end{cases} \quad (2.106)$$

is a finite-slope approximation to the cube root with $\delta = 6/29$. The resulting 0 ... 100 scale roughly measures equal amounts of lightness perceptibility.

In a similar fashion, the a* and b* components are defined as

$$a^* = 500 \left[f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right) \right] \quad \text{and} \quad b^* = 200 \left[f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right) \right], \quad (2.107)$$

¹⁹ Another perceptually motivated color space called L*u*v* was developed and standardized simultaneously (Fairchild 2005).

where again, (X_n, Y_n, Z_n) is the measured white point. Figure 2.32i–k show the L*a*b* representation for a sample color image.

Color cameras

While the preceding discussion tells us how we can uniquely describe the perceived tri-stimulus description of any color (spectral distribution), it does not tell us how RGB still and video cameras actually work. Do they just measure the amount of light at the nominal wavelengths of red (700.0nm), green (546.1nm), and blue (435.8nm)? Do color monitors just emit exactly these wavelengths and, if so, how can they emit negative red light to reproduce colors in the cyan range?

In fact, the design of RGB video cameras has historically been based around the availability of colored phosphors that go into television sets. When standard-definition color television was invented (NTSC), a mapping was defined between the RGB values that would drive the three color guns in the cathode ray tube (CRT) and the XYZ values that unambiguously define perceived color (this standard was called ITU-R BT.601). With the advent of HDTV and newer monitors, a new standard called ITU-R BT.709 was created, which specifies the XYZ values of each of the color primaries,

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \begin{bmatrix} R_{709} \\ G_{709} \\ B_{709} \end{bmatrix}. \quad (2.108)$$

In practice, each color camera integrates light according to the *spectral response function* of its red, green, and blue sensors,

$$\begin{aligned} R &= \int L(\lambda)S_R(\lambda)d\lambda, \\ G &= \int L(\lambda)S_G(\lambda)d\lambda, \\ B &= \int L(\lambda)S_B(\lambda)d\lambda, \end{aligned} \quad (2.109)$$

where $L(\lambda)$ is the incoming spectrum of light at a given pixel and $\{S_R(\lambda), S_G(\lambda), S_B(\lambda)\}$ are the red, green, and blue *spectral sensitivities* of the corresponding sensors.

Can we tell what spectral sensitivities the cameras actually have? Unless the camera manufacturer provides us with this data or we observe the response of the camera to a whole spectrum of monochromatic lights, these sensitivities are *not* specified by a standard such as BT.709. Instead, all that matters is that the tri-stimulus values for a given color produce the specified RGB values. The manufacturer is free to use sensors with sensitivities that do not match the standard XYZ definitions, so long as they can later be converted (through a linear transform) to the standard colors.

Similarly, while TV and computer monitors are supposed to produce RGB values as specified by Equation (2.108), there is no reason that they cannot use digital logic to transform the incoming RGB values into different signals to drive each of the color channels. Properly calibrated monitors make this information available to software applications that perform *color management*, so that colors in real life, on the screen, and on the printer all match as closely as possible.

G	R	G	R
B	G	B	G
G	R	G	R
B	G	B	G

(a)

rGb	Rgb	rGb	Rgb
rgB	rGb	rgB	rGb
rGb	Rgb	rGb	Rgb
rgB	rGb	rgB	rGb

(b)

Figure 2.30 Bayer RGB pattern: (a) color filter array layout; (b) interpolated pixel values, with unknown (guessed) values shown as lower case.

Color filter arrays

While early color TV cameras used three *vidicons* (tubes) to perform their sensing and later cameras used three separate RGB sensing chips, most of today's digital still and video cameras use a *color filter array* (CFA), where alternating sensors are covered by different colored filters.²⁰

The most commonly used pattern in color cameras today is the *Bayer pattern* (Bayer 1976), which places green filters over half of the sensors (in a checkerboard pattern), and red and blue filters over the remaining ones (Figure 2.30). The reason that there are twice as many green filters as red and blue is because the luminance signal is mostly determined by green values and the visual system is much more sensitive to high frequency detail in luminance than in chrominance (a fact that is exploited in color image compression—see Section 2.3.3). The process of *interpolating* the missing color values so that we have valid RGB values for all the pixels is known as *demosaicing* and is covered in detail in Section 10.3.1.

Similarly, color LCD monitors typically use alternating stripes of red, green, and blue filters placed in front of each liquid crystal active area to simulate the experience of a full color display. As before, because the visual system has higher resolution (acuity) in luminance than chrominance, it is possible to digitally pre-filter RGB (and monochrome) images to enhance the perception of crispness (Betrisey, Blinn, Dresevic *et al.* 2000; Platt 2000).

Color balance

Before encoding the sensed RGB values, most cameras perform some kind of *color balancing* operation in an attempt to move the white point of a given image closer to pure white (equal RGB values). If the color system and the illumination are the same (the BT.709 system uses the daylight illuminant D₆₅ as its reference white), the change may be minimal. However, if the illuminant is strongly colored, such as incandescent indoor lighting (which generally results in a yellow or orange hue), the compensation can be quite significant.

A simple way to perform color correction is to multiply each of the RGB values by a different factor (i.e., to apply a diagonal matrix transform to the RGB color space). More

²⁰ A newer chip design by Foveon (<http://www.foveon.com>) stacks the red, green, and blue sensors beneath each other, but it has not yet gained widespread adoption.

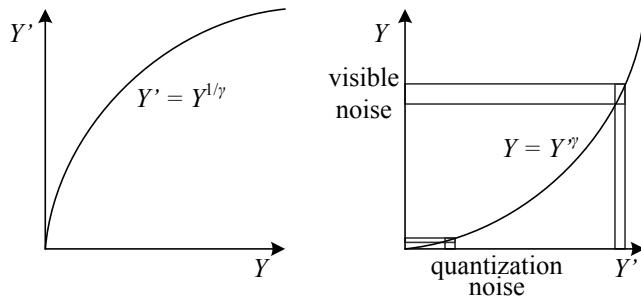


Figure 2.31 Gamma compression: (a) The relationship between the input signal luminance Y and the transmitted signal Y' is given by $Y' = Y^{1/\gamma}$. (b) At the receiver, the signal Y' is exponentiated by the factor γ , $\hat{Y} = Y'^\gamma$. Noise introduced during transmission is squashed in the dark regions, which corresponds to the more noise-sensitive region of the visual system.

complicated transforms, which are sometimes the result of mapping to XYZ space and back, actually perform a *color twist*, i.e., they use a general 3×3 color transform matrix.²¹ Exercise 2.9 has you explore some of these issues.

Gamma

In the early days of black and white television, the phosphors in the CRT used to display the TV signal responded non-linearly to their input voltage. The relationship between the voltage and the resulting brightness was characterized by a number called *gamma* (γ), since the formula was roughly

$$B = V^\gamma, \quad (2.110)$$

with a γ of about 2.2. To compensate for this effect, the electronics in the TV camera would pre-map the sensed luminance Y through an inverse gamma,

$$Y' = Y^{\frac{1}{\gamma}}, \quad (2.111)$$

with a typical value of $\frac{1}{\gamma} = 0.45$.

The mapping of the signal through this non-linearity before transmission had a beneficial side effect: noise added during transmission (remember, these were analog days!) would be reduced (after applying the gamma at the receiver) in the darker regions of the signal where it was more visible (Figure 2.31).²² (Remember that our visual system is roughly sensitive to relative differences in luminance.)

When color television was invented, it was decided to separately pass the red, green, and blue signals through the same gamma non-linearity before combining them for encoding. Today, even though we no longer have analog noise in our transmission systems, signals are still quantized during compression (see Section 2.3.3), so applying inverse gamma to sensed values is still useful.

²¹ Those of you old enough to remember the early days of color television will naturally think of the *hue* adjustment knob on the television set, which could produce truly bizarre results.

²² A related technique called *companding* was the basis of the Dolby noise reduction systems used with audio tapes.

Unfortunately, for both computer vision and computer graphics, the presence of gamma in images is often problematic. For example, the proper simulation of radiometric phenomena such as shading (see Section 2.2 and Equation (2.87)) occurs in a linear radiance space. Once all of the computations have been performed, the appropriate gamma should be applied before display. Unfortunately, many computer graphics systems (such as shading models) operate directly on RGB values and display these values directly. (Fortunately, newer color imaging standards such as the 16-bit scRGB use a linear space, which makes this less of a problem (Glassner 1995).)

In computer vision, the situation can be even more daunting. The accurate determination of surface normals, using a technique such as photometric stereo (Section 12.1.1) or even a simpler operation such as accurate image deblurring, require that the measurements be in a linear space of intensities. Therefore, it is imperative when performing detailed quantitative computations such as these to first undo the gamma and the per-image color re-balancing in the sensed color values. Chakrabarti, Scharstein, and Zickler (2009) develop a sophisticated 24-parameter model that is a good match to the processing performed by today’s digital cameras; they also provide a database of color images you can use for your own testing.²³

For other vision applications, however, such as feature detection or the matching of signals in stereo and motion estimation, this linearization step is often not necessary. In fact, determining whether it is necessary to undo gamma can take some careful thinking, e.g., in the case of compensating for exposure variations in image stitching (see Exercise 2.7).

If all of these processing steps sound confusing to model, they are. Exercise 2.10 has you try to tease apart some of these phenomena using empirical investigation, i.e., taking pictures of color charts and comparing the RAW and JPEG compressed color values.

Other color spaces

While RGB and XYZ are the primary color spaces used to describe the spectral content (and hence tri-stimulus response) of color signals, a variety of other representations have been developed both in video and still image coding and in computer graphics.

The earliest color representation developed for video transmission was the YIQ standard developed for NTSC video in North America and the closely related YUV standard developed for PAL in Europe. In both of these cases, it was desired to have a *luma* channel Y (so called since it only roughly mimics true luminance) that would be comparable to the regular black-and-white TV signal, along with two lower frequency *chroma* channels.

In both systems, the Y signal (or more appropriately, the Y’ luma signal since it is gamma compressed) is obtained from

$$Y'_{601} = 0.299R' + 0.587G' + 0.114B', \quad (2.112)$$

where R’G’B’ is the triplet of gamma-compressed color components. When using the newer color definitions for HDTV in BT.709, the formula is

$$Y'_{709} = 0.2125R' + 0.7154G' + 0.0721B'. \quad (2.113)$$

The UV components are derived from scaled versions of (B’ – Y’) and (R’ – Y’), namely,

$$U = 0.492111(B' - Y') \text{ and } V = 0.877283(R' - Y'), \quad (2.114)$$

²³ <http://vision.middlebury.edu/color/>.

whereas the IQ components are the UV components rotated through an angle of 33° . In composite (NTSC and PAL) video, the chroma signals were then low-pass filtered horizontally before being modulated and superimposed on top of the Y' luma signal. Backward compatibility was achieved by having older black-and-white TV sets effectively ignore the high-frequency chroma signal (because of slow electronics) or, at worst, superimposing it as a high-frequency pattern on top of the main signal.

While these conversions were important in the early days of computer vision, when frame grabbers would directly digitize the composite TV signal, today all digital video and still image compression standards are based on the newer YCbCr conversion. YCbCr is closely related to YUV (the C_b and C_r signals carry the blue and red color difference signals and have more useful mnemonics than UV) but uses different scale factors to fit within the eight-bit range available with digital signals.

For video, the Y' signal is re-scaled to fit within the [16 . . . 235] range of values, while the C_b and C_r signals are scaled to fit within [16 . . . 240] (Gomes and Velho 1997; Fairchild 2005). For still images, the JPEG standard uses the full eight-bit range with no reserved values,

$$\begin{bmatrix} Y' \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}, \quad (2.115)$$

where the $R'G'B'$ values are the eight-bit gamma-compressed color components (i.e., the actual RGB values we obtain when we open up or display a JPEG image). For most applications, this formula is not that important, since your image reading software will directly provide you with the eight-bit gamma-compressed $R'G'B'$ values. However, if you are trying to do careful image deblocking (Exercise 3.30), this information may be useful.

Another color space you may come across is *hue, saturation, value* (HSV), which is a projection of the RGB color cube onto a non-linear chroma angle, a radial saturation percentage, and a luminance-inspired value. In more detail, value is defined as either the mean or maximum color value, saturation is defined as scaled distance from the diagonal, and hue is defined as the direction around a color wheel (the exact formulas are described by Hall (1989); Foley, van Dam, Feiner *et al.* (1995)). Such a decomposition is quite natural in graphics applications such as color picking (it approximates the Munsell chart for color description). Figure 2.321–n shows an HSV representation of a sample color image, where saturation is encoded using a gray scale (saturated = darker) and hue is depicted as a color.

If you want your computer vision algorithm to only affect the value (luminance) of an image and not its saturation or hue, a simpler solution is to use either the Yxy (luminance + chromaticity) coordinates defined in (2.104) or the even simpler *color ratios*,

$$r = \frac{R}{R+G+B}, \quad g = \frac{G}{R+G+B}, \quad b = \frac{B}{R+G+B} \quad (2.116)$$

(Figure 2.32e–h). After manipulating the luma (2.112), e.g., through the process of histogram equalization (Section 3.1.4), you can multiply each color ratio by the ratio of the new to old luma to obtain an adjusted RGB triplet.

While all of these color systems may sound confusing, in the end, it often may not matter that much which one you use. Poynton, in his *Color FAQ*, <http://www.poynton.com/ColorFAQ.html>, notes that the perceptually motivated L*a*b* system is qualitatively similar

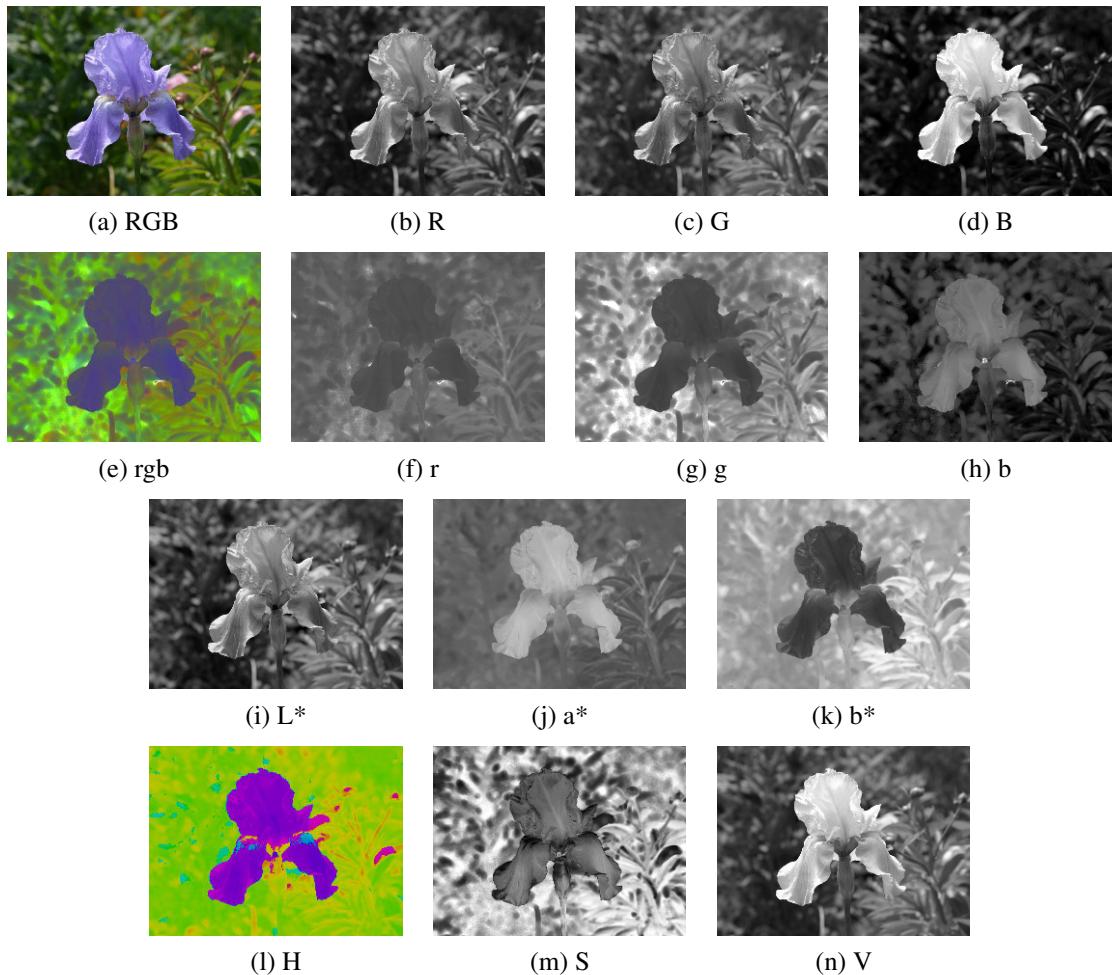


Figure 2.32 Color space transformations: (a–d) RGB; (e–h) rgb. (i–k) $L^*a^*b^*$; (l–n) HSV. Note that the rgb, $L^*a^*b^*$, and HSV values are all re-scaled to fit the dynamic range of the printed page.

to the gamma-compressed $R'G'B'$ system we mostly deal with, since both have a fractional power scaling (which approximates a logarithmic response) between the actual intensity values and the numbers being manipulated. As in all cases, think carefully about what you are trying to accomplish before deciding on a technique to use.²⁴

2.3.3 Compression

The last stage in a camera’s processing pipeline is usually some form of image compression (unless you are using a lossless compression scheme such as camera RAW or PNG).

All color video and image compression algorithms start by converting the signal into YCbCr (or some closely related variant), so that they can compress the luminance signal with

²⁴ If you are at a loss for questions at a conference, you can always ask why the speaker did not use a perceptual color space, such as $L^*a^*b^*$. Conversely, if they did use $L^*a^*b^*$, you can ask if they have any concrete evidence that this works better than regular colors.



Figure 2.33 Image compressed with JPEG at three quality settings. Note how the amount of block artifact and high-frequency aliasing (“mosquito noise”) increases from left to right.

higher fidelity than the chrominance signal. (Recall that the human visual system has poorer frequency response to color than to luminance changes.) In video, it is common to subsample Cb and Cr by a factor of two horizontally; with still images (JPEG), the subsampling (averaging) occurs both horizontally and vertically.

Once the luminance and chrominance images have been appropriately subsampled and separated into individual images, they are then passed to a *block transform* stage. The most common technique used here is the *discrete cosine transform* (DCT), which is a real-valued variant of the discrete Fourier transform (DFT) (see Section 3.4.3). The DCT is a reasonable approximation to the Karhunen–Loëve or eigenvalue decomposition of natural image patches, i.e., the decomposition that simultaneously packs the most energy into the first coefficients and diagonalizes the joint covariance matrix among the pixels (makes transform coefficients statistically independent). Both MPEG and JPEG use 8×8 DCT transforms (Wallace 1991; Le Gall 1991), although newer variants use smaller 4×4 blocks or alternative transformations, such as wavelets (Taubman and Marcellin 2002) and lapped transforms (Malvar 1990, 1998, 2000) are now used.

After transform coding, the coefficient values are quantized into a set of small integer values that can be coded using a variable bit length scheme such as a Huffman code or an arithmetic code (Wallace 1991). (The DC (lowest frequency) coefficients are also adaptively predicted from the previous block’s DC values. The term “DC” comes from “direct current”, i.e., the non-sinusoidal or non-alternating part of a signal.) The step size in the quantization is the main variable controlled by the *quality* setting on the JPEG file (Figure 2.33).

With video, it is also usual to perform block-based *motion compensation*, i.e., to encode the difference between each block and a *predicted* set of pixel values obtained from a shifted block in the previous frame. (The exception is the *motion-JPEG* scheme used in older DV camcorders, which is nothing more than a series of individually JPEG compressed image frames.) While basic MPEG uses 16×16 motion compensation blocks with integer motion values (Le Gall 1991), newer standards use adaptively sized block, sub-pixel motions, and the ability to reference blocks from older frames. In order to recover more gracefully from failures and to allow for random access to the video stream, predicted P frames are interleaved among independently coded I frames. (Bi-directional B frames are also sometimes used.)

The quality of a compression algorithm is usually reported using its *peak signal-to-noise*

ratio (PSNR), which is derived from the average *mean square error*,

$$MSE = \frac{1}{n} \sum_{\mathbf{x}} [I(\mathbf{x}) - \hat{I}(\mathbf{x})]^2, \quad (2.117)$$

where $I(\mathbf{x})$ is the original uncompressed image and $\hat{I}(\mathbf{x})$ is its compressed counterpart, or equivalently, the *root mean square error* (RMS error), which is defined as

$$RMS = \sqrt{MSE}. \quad (2.118)$$

The PSNR is defined as

$$PSNR = 10 \log_{10} \frac{I_{\max}^2}{MSE} = 20 \log_{10} \frac{I_{\max}}{RMS}, \quad (2.119)$$

where I_{\max} is the maximum signal extent, e.g., 255 for eight-bit images.

While this is just a high-level sketch of how image compression works, it is useful to understand so that the artifacts introduced by such techniques can be compensated for in various computer vision applications.

2.4 Additional reading

As we mentioned at the beginning of this chapter, it provides but a brief summary of a very rich and deep set of topics, traditionally covered in a number of separate fields.

A more thorough introduction to the geometry of points, lines, planes, and projections can be found in textbooks on multi-view geometry (Hartley and Zisserman 2004; Faugeras and Luong 2001) and computer graphics (Foley, van Dam, Feiner *et al.* 1995; Watt 1995; OpenGL-ARB 1997). Topics covered in more depth include higher-order primitives such as quadrics, conics, and cubics, as well as three-view and multi-view geometry.

The image formation (synthesis) process is traditionally taught as part of a computer graphics curriculum (Foley, van Dam, Feiner *et al.* 1995; Glassner 1995; Watt 1995; Shirley 2005) but it is also studied in physics-based computer vision (Wolff, Shafer, and Healey 1992a).

The behavior of camera lens systems is studied in optics (Möller 1988; Hecht 2001; Ray 2002).

Some good books on color theory have been written by Healey and Shafer (1992); Wyszecki and Stiles (2000); Fairchild (2005), with Livingstone (2008) providing a more fun and informal introduction to the topic of color perception. Mark Fairchild's page of color books and links²⁵ lists many other sources.

Topics relating to sampling and aliasing are covered in textbooks on signal and image processing (Crane 1997; Jähne 1997; Oppenheim and Schafer 1996; Oppenheim, Schafer, and Buck 1999; Pratt 2007; Russ 2007; Burger and Burge 2008; Gonzales and Woods 2008).

2.5 Exercises

A note to students: This chapter is relatively light on exercises since it contains mostly background material and not that many usable techniques. If you really want to understand

²⁵ http://www.cis.rit.edu/fairchild/WhyIsColor/books_links.html.

multi-view geometry in a thorough way, I encourage you to read and do the exercises provided by Hartley and Zisserman (2004). Similarly, if you want some exercises related to the image formation process, Glassner's (1995) book is full of challenging problems.

Ex 2.1: Least squares intersection point and line fitting—advanced Equation (2.4) shows how the intersection of two 2D lines can be expressed as their cross product, assuming the lines are expressed as homogeneous coordinates.

1. If you are given more than two lines and want to find a point \tilde{x} that minimizes the sum of squared distances to each line,

$$D = \sum_i (\tilde{x} \cdot \tilde{l}_i)^2, \quad (2.120)$$

how can you compute this quantity? (Hint: Write the dot product as $\tilde{x}^T \tilde{l}_i$ and turn the squared quantity into a *quadratic form*, $\tilde{x}^T A \tilde{x}$.)

2. To fit a line to a bunch of points, you can compute the *centroid* (mean) of the points as well as the *covariance matrix* of the points around this mean. Show that the line passing through the centroid along the major axis of the covariance ellipsoid (largest eigenvector) minimizes the sum of squared distances to the points.
3. These two approaches are fundamentally different, even though projective duality tells us that points and lines are interchangeable. Why are these two algorithms so apparently different? Are they actually minimizing different objectives?

Ex 2.2: 2D transform editor Write a program that lets you interactively create a set of rectangles and then modify their “pose” (2D transform). You should implement the following steps:

1. Open an empty window (“canvas”).
2. Shift drag (rubber-band) to create a new rectangle.
3. Select the deformation mode (motion model): translation, rigid, similarity, affine, or perspective.
4. Drag any corner of the outline to change its transformation.

This exercise should be built on a set of pixel coordinate and transformation classes, either implemented by yourself or from a software library. Persistence of the created representation (save and load) should also be supported (for each rectangle, save its transformation).

Ex 2.3: 3D viewer Write a simple viewer for 3D points, lines, and polygons. Import a set of point and line commands (primitives) as well as a viewing transform. Interactively modify the object or camera transform. This viewer can be an extension of the one you created in (Exercise 2.2). Simply replace the viewing transformations with their 3D equivalents.

(Optional) Add a z-buffer to do hidden surface removal for polygons.

(Optional) Use a 3D drawing package and just write the viewer control.

Ex 2.4: Focus distance and depth of field Figure out how the focus distance and depth of field indicators on a lens are determined.

1. Compute and plot the focus distance z_o as a function of the distance traveled from the focal length $\Delta z_i = f - z_i$ for a lens of focal length f (say, 100mm). Does this explain the hyperbolic progression of focus distances you see on a typical lens (Figure 2.20)?
2. Compute the depth of field (minimum and maximum focus distances) for a given focus setting z_o as a function of the circle of confusion diameter c (make it a fraction of the sensor width), the focal length f , and the f-stop number N (which relates to the aperture diameter d). Does this explain the usual depth of field markings on a lens that bracket the in-focus marker, as in Figure 2.20a?
3. Now consider a zoom lens with a varying focal length f . Assume that as you zoom, the lens stays in focus, i.e., the distance from the rear nodal point to the sensor plane z_i adjusts itself automatically for a fixed focus distance z_o . How do the depth of field indicators vary as a function of focal length? Can you reproduce a two-dimensional plot that mimics the curved depth of field lines seen on the lens in Figure 2.20b?

Ex 2.5: F-numbers and shutter speeds List the common f-numbers and shutter speeds that your camera provides. On older model SLRs, they are visible on the lens and shutter speed dials. On newer cameras, you have to look at the electronic viewfinder (or LCD screen/indicator) as you manually adjust exposures.

1. Do these form geometric progressions; if so, what are the ratios? How do these relate to exposure values (EVs)?
2. If your camera has shutter speeds of $\frac{1}{60}$ and $\frac{1}{125}$, do you think that these two speeds are exactly a factor of two apart or a factor of $125/60 = 2.083$ apart?
3. How accurate do you think these numbers are? Can you devise some way to measure exactly how the aperture affects how much light reaches the sensor and what the exact exposure times actually are?

Ex 2.6: Noise level calibration Estimate the amount of noise in your camera by taking repeated shots of a scene with the camera mounted on a tripod. (Purchasing a remote shutter release is a good investment if you own a DSLR.) Alternatively, take a scene with constant color regions (such as a color checker chart) and estimate the variance by fitting a smooth function to each color region and then taking differences from the predicted function.

1. Plot your estimated variance as a function of level for each of your color channels separately.
2. Change the ISO setting on your camera; if you cannot do that, reduce the overall light in your scene (turn off lights, draw the curtains, wait until dusk). Does the amount of noise vary a lot with ISO/gain?
3. Compare your camera to another one at a different price point or year of make. Is there evidence to suggest that “you get what you pay for”? Does the quality of digital cameras seem to be improving over time?

Ex 2.7: Gamma correction in image stitching Here’s a relatively simple puzzle. Assume you are given two images that are part of a panorama that you want to stitch (see Chapter 9). The two images were taken with different exposures, so you want to adjust the RGB values so that they match along the seam line. Is it necessary to undo the gamma in the color values in order to achieve this?

Ex 2.8: Skin color detection Devise a simple skin color detector (Forsyth and Fleck 1999; Jones and Rehg 2001; Vezhnevets, Sazonov, and Andreeva 2003; Kakumanu, Makrogiannis, and Bourbakis 2007) based on chromaticity or other color properties.

1. Take a variety of photographs of people and calculate the *xy chromaticity values* for each pixel.
2. Crop the photos or otherwise indicate with a painting tool which pixels are likely to be skin (e.g. face and arms).
3. Calculate a color (chromaticity) distribution for these pixels. You can use something as simple as a mean and covariance measure or as complicated as a mean-shift segmentation algorithm (see Section 5.3.2). You can optionally use non-skin pixels to model the *background distribution*.
4. Use your computed distribution to find the skin regions in an image. One easy way to visualize this is to paint all non-skin pixels a given color, such as white or black.
5. How sensitive is your algorithm to color balance (scene lighting)?
6. Does a simpler chromaticity measurement, such as a color ratio (2.116), work just as well?

Ex 2.9: White point balancing—tricky A common (in-camera or post-processing) technique for performing white point adjustment is to take a picture of a white piece of paper and to adjust the RGB values of an image to make this a neutral color.

1. Describe how you would adjust the RGB values in an image given a sample “white color” of (R_w, G_w, B_w) to make this color neutral (without changing the exposure too much).
2. Does your transformation involve a simple (per-channel) scaling of the RGB values or do you need a full 3×3 color twist matrix (or something else)?
3. Convert your RGB values to XYZ. Does the appropriate correction now only depend on the XY (or xy) values? If so, when you convert back to RGB space, do you need a full 3×3 color twist matrix to achieve the same effect?
4. If you used pure diagonal scaling in the direct RGB mode but end up with a twist if you work in XYZ space, how do you explain this apparent dichotomy? Which approach is correct? (Or is it possible that neither approach is actually correct?)

If you want to find out what your camera *actually* does, continue on to the next exercise.

Ex 2.10: In-camera color processing—challenging If your camera supports a RAW pixel mode, take a pair of RAW and JPEG images, and see if you can infer what the camera is doing when it converts the RAW pixel values to the final color-corrected and gamma-compressed eight-bit JPEG pixel values.

1. Deduce the pattern in your color filter array from the correspondence between co-located RAW and color-mapped pixel values. Use a color checker chart at this stage if it makes your life easier. You may find it helpful to split the RAW image into four separate images (subsampling even and odd columns and rows) and to treat each of these new images as a “virtual” sensor.
2. Evaluate the quality of the demosaicing algorithm by taking pictures of challenging scenes which contain strong color edges (such as those shown in in Section 10.3.1).
3. If you can take the same exact picture after changing the color balance values in your camera, compare how these settings affect this processing.
4. Compare your results against those presented by Chakrabarti, Scharstein, and Zickler (2009) or use the data available in their database of color images.²⁶

²⁶ <http://vision.middlebury.edu/color/>.

Chapter 3

Image processing

3.1	Point operators	89
3.1.1	Pixel transforms	91
3.1.2	Color transforms	92
3.1.3	Compositing and matting	92
3.1.4	Histogram equalization	94
3.1.5	<i>Application:</i> Tonal adjustment	97
3.2	Linear filtering	98
3.2.1	Separable filtering	102
3.2.2	Examples of linear filtering	103
3.2.3	Band-pass and steerable filters	104
3.3	More neighborhood operators	108
3.3.1	Non-linear filtering	108
3.3.2	Morphology	112
3.3.3	Distance transforms	113
3.3.4	Connected components	115
3.4	Fourier transforms	116
3.4.1	Fourier transform pairs	119
3.4.2	Two-dimensional Fourier transforms	123
3.4.3	Wiener filtering	123
3.4.4	<i>Application:</i> Sharpening, blur, and noise removal	126
3.5	Pyramids and wavelets	127
3.5.1	Interpolation	127
3.5.2	Decimation	130
3.5.3	Multi-resolution representations	132
3.5.4	Wavelets	136
3.5.5	<i>Application:</i> Image blending	140
3.6	Geometric transformations	143
3.6.1	Parametric transformations	145
3.6.2	Mesh-based warping	149
3.6.3	<i>Application:</i> Feature-based morphing	152
3.7	Global optimization	153
3.7.1	Regularization	154
3.7.2	Markov random fields	158
3.7.3	<i>Application:</i> Image restoration	169
3.8	Additional reading	169
3.9	Exercises	171



Figure 3.1 Some common image processing operations: (a) original image; (b) increased contrast; (c) change in hue; (d) “posterized” (quantized colors); (e) blurred; (f) rotated.

Now that we have seen how images are formed through the interaction of 3D scene elements, lighting, and camera optics and sensors, let us look at the first stage in most computer vision applications, namely the use of image processing to preprocess the image and convert it into a form suitable for further analysis. Examples of such operations include exposure correction and color balancing, the reduction of image noise, increasing sharpness, or straightening the image by rotating it (Figure 3.1). While some may consider image processing to be outside the purview of computer vision, most computer vision applications, such as computational photography and even recognition, require care in designing the image processing stages in order to achieve acceptable results.

In this chapter, we review standard image processing operators that map pixel values from one image to another. Image processing is often taught in electrical engineering departments as a follow-on course to an introductory course in signal processing (Oppenheim and Schafer 1996; Oppenheim, Schafer, and Buck 1999). There are several popular textbooks for image processing (Crane 1997; Gomes and Velho 1997; Jähne 1997; Pratt 2007; Russ 2007; Burger and Burge 2008; Gonzales and Woods 2008).

We begin this chapter with the simplest kind of image transforms, namely those that manipulate each pixel independently of its neighbors (Section 3.1). Such transforms are often called *point operators* or *point processes*. Next, we examine *neighborhood* (area-based) operators, where each new pixel's value depends on a small number of neighboring input values (Sections 3.2 and 3.3). A convenient tool to analyze (and sometimes accelerate) such neighborhood operations is the *Fourier Transform*, which we cover in Section 3.4. Neighborhood operators can be cascaded to form *image pyramids* and *wavelets*, which are useful for analyzing images at a variety of resolutions (scales) and for accelerating certain operations (Section 3.5). Another important class of global operators are *geometric transformations*, such as rotations, shears, and perspective deformations (Section 3.6). Finally, we introduce *global optimization* approaches to image processing, which involve the minimization of an energy functional or, equivalently, optimal estimation using Bayesian *Markov random field* models (Section 3.7).

3.1 Point operators

The simplest kinds of image processing transforms are *point operators*, where each output pixel's value depends on only the corresponding input pixel value (plus, potentially, some globally collected information or parameters). Examples of such operators include brightness and contrast adjustments (Figure 3.2) as well as color correction and transformations. In the image processing literature, such operations are also known as *point processes* (Crane 1997).

We begin this section with a quick review of simple point operators such as brightness scaling and image addition. Next, we discuss how colors in images can be manipulated. We then present *image compositing* and *matting* operations, which play an important role in computational photography (Chapter 10) and computer graphics applications. Finally, we describe the more global process of *histogram equalization*. We close with an example application that manipulates *tonal values* (exposure and contrast) to improve image appearance.

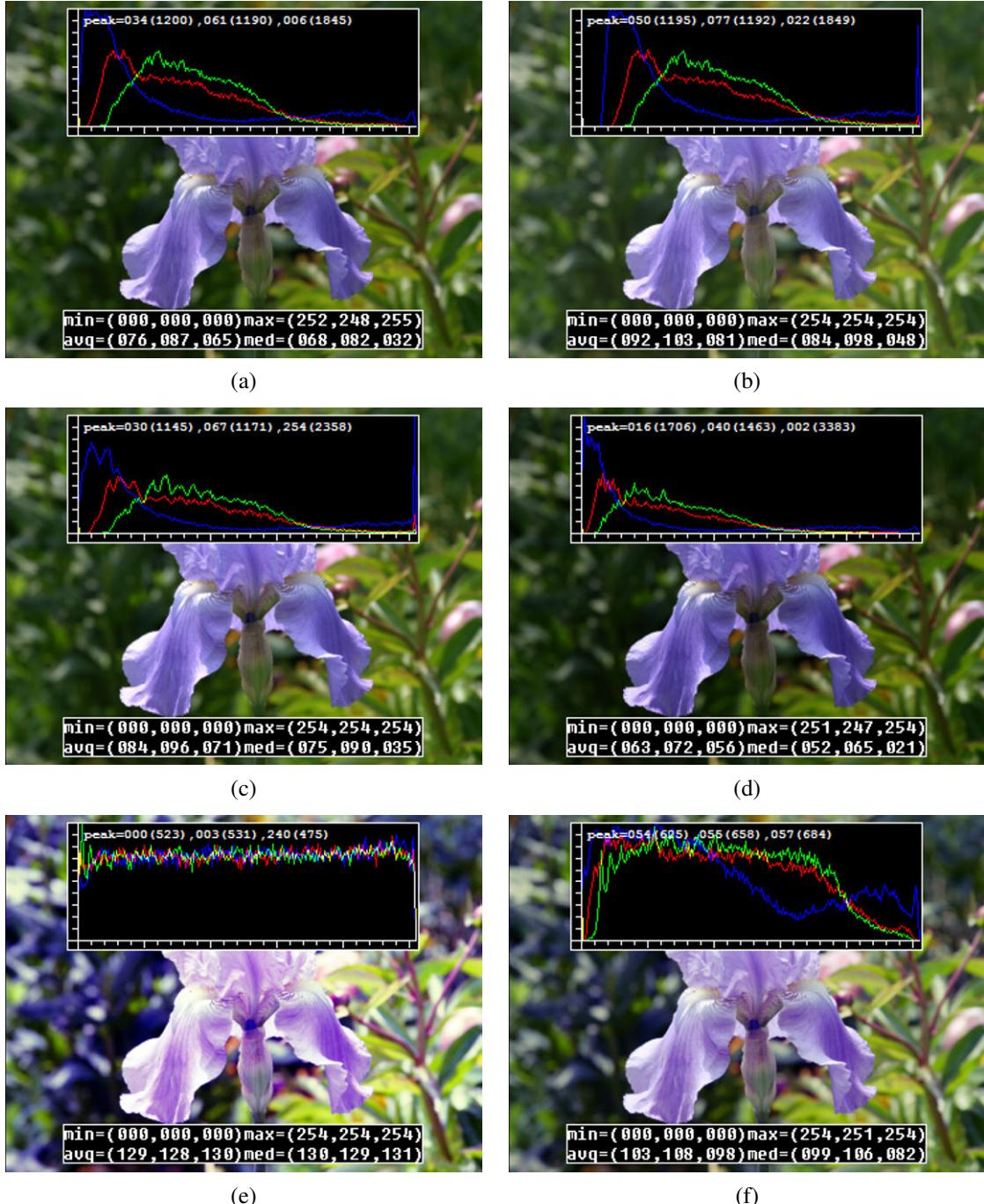


Figure 3.2 Some local image processing operations: (a) original image along with its three color (per-channel) histograms; (b) brightness increased (additive offset, $b = 16$); (c) contrast increased (multiplicative gain, $a = 1.1$); (d) gamma (partially) linearized ($\gamma = 1.2$); (e) full histogram equalization; (f) partial histogram equalization.

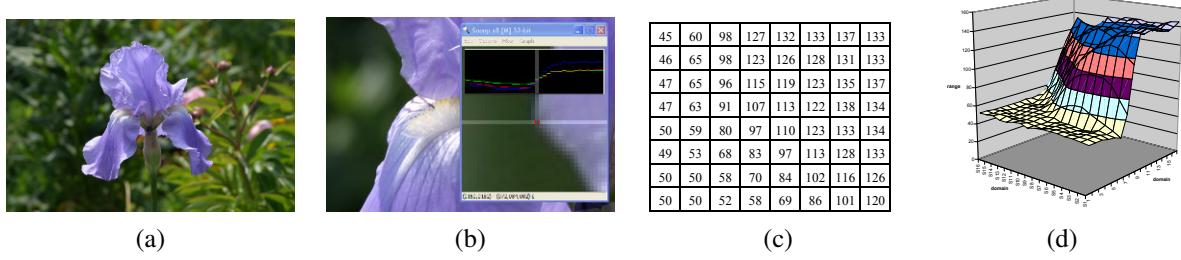


Figure 3.3 Visualizing image data: (a) original image; (b) cropped portion and scanline plot using an image inspection tool; (c) grid of numbers; (d) surface plot. For figures (c)–(d), the image was first converted to grayscale.

3.1.1 Pixel transforms

A general image processing *operator* is a function that takes one or more input images and produces an output image. In the continuous domain, this can be denoted as

$$g(\mathbf{x}) = h(f(\mathbf{x})) \text{ or } g(\mathbf{x}) = h(f_0(\mathbf{x}), \dots, f_n(\mathbf{x})), \quad (3.1)$$

where \mathbf{x} is in the D -dimensional *domain* of the functions (usually $D = 2$ for images) and the functions f and g operate over some *range*, which can either be scalar or vector-valued, e.g., for color images or 2D motion. For discrete (sampled) images, the domain consists of a finite number of *pixel locations*, $\mathbf{x} = (i, j)$, and we can write

$$g(i, j) = h(f(i, j)). \quad (3.2)$$

Figure 3.3 shows how an image can be represented either by its color (appearance), as a grid of numbers, or as a two-dimensional function (surface plot).

Two commonly used point processes are multiplication and addition with a constant,

$$g(\mathbf{x}) = af(\mathbf{x}) + b. \quad (3.3)$$

The parameters $a > 0$ and b are often called the *gain* and *bias* parameters; sometimes these parameters are said to control *contrast* and *brightness*, respectively (Figures 3.2b–c).¹ The bias and gain parameters can also be spatially varying,

$$g(\mathbf{x}) = a(\mathbf{x})f(\mathbf{x}) + b(\mathbf{x}), \quad (3.4)$$

e.g., when simulating the *graded density filter* used by photographers to selectively darken the sky or when modeling vignetting in an optical system.

Multiplicative gain (both global and spatially varying) is a *linear* operation, since it obeys the *superposition principle*,

$$h(f_0 + f_1) = h(f_0) + h(f_1). \quad (3.5)$$

(We will have more to say about linear shift invariant operators in Section 3.2.) Operators such as image squaring (which is often used to get a local estimate of the *energy* in a band-pass filtered signal, see Section 3.5) are not linear.

¹ An image's luminance characteristics can also be summarized by its *key* (average luminance) and *range* (Kopf, Uyttendaele, Deussen *et al.* 2007).

Another commonly used *dyadic* (two-input) operator is the *linear blend* operator,

$$g(\mathbf{x}) = (1 - \alpha)f_0(\mathbf{x}) + \alpha f_1(\mathbf{x}). \quad (3.6)$$

By varying α from $0 \rightarrow 1$, this operator can be used to perform a temporal *cross-dissolve* between two images or videos, as seen in slide shows and film production, or as a component of image *morphing* algorithms (Section 3.6.3).

One highly used non-linear transform that is often applied to images before further processing is *gamma correction*, which is used to remove the non-linear mapping between input radiance and quantized pixel values (Section 2.3.2). To invert the gamma mapping applied by the sensor, we can use

$$g(\mathbf{x}) = [f(\mathbf{x})]^{1/\gamma}, \quad (3.7)$$

where a gamma value of $\gamma \approx 2.2$ is a reasonable fit for most digital cameras.

3.1.2 Color transforms

While color images can be treated as arbitrary vector-valued functions or collections of independent bands, it usually makes sense to think about them as highly correlated signals with strong connections to the image formation process (Section 2.2), sensor design (Section 2.3), and human perception (Section 2.3.2). Consider, for example, brightening a picture by adding a constant value to all three channels, as shown in Figure 3.2b. Can you tell if this achieves the desired effect of making the image look brighter? Can you see any undesirable side-effects or artifacts?

In fact, adding the same value to each color channel not only increases the apparent *intensity* of each pixel, it can also affect the pixel's *hue* and *saturation*. How can we define and manipulate such quantities in order to achieve the desired perceptual effects?

As discussed in Section 2.3.2, chromaticity coordinates (2.104) or even simpler color ratios (2.116) can first be computed and then used after manipulating (e.g., brightening) the luminance Y to re-compute a valid RGB image with the same hue and saturation. Figure 2.32g–i shows some color ratio images multiplied by the middle gray value for better visualization.

Similarly, color balancing (e.g., to compensate for incandescent lighting) can be performed either by multiplying each channel with a different scale factor or by the more complex process of mapping to XYZ color space, changing the nominal white point, and mapping back to RGB, which can be written down using a linear 3×3 *color twist* transform matrix. Exercises 2.9 and 3.1 have you explore some of these issues.

Another fun project, best attempted after you have mastered the rest of the material in this chapter, is to take a picture with a rainbow in it and enhance the strength of the rainbow (Exercise 3.29).

3.1.3 Compositing and matting

In many photo editing and visual effects applications, it is often desirable to cut a *foreground* object out of one scene and put it on top of a different *background* (Figure 3.4). The process of extracting the object from the original image is often called *matting* (Smith and Blinn



Figure 3.4 Image matting and compositing (Chuang, Curless, Salesin *et al.* 2001) © 2001 IEEE: (a) source image; (b) extracted foreground object F ; (c) alpha matte α shown in grayscale; (d) new composite C .

$$\begin{array}{ccccc}
 \text{(a)} & \times & \text{(b)} & + & \text{(c)} \\
 B & & \alpha & & \alpha F \\
 & & & & = \\
 & & & & \text{(d)} \\
 & & & & C
 \end{array}$$

Figure 3.5 Compositing equation $C = (1 - \alpha)B + \alpha F$. The images are taken from a close-up of the region of the hair in the upper right part of the lion in Figure 3.4.

1996), while the process of inserting it into another image (without visible artifacts) is called *compositing* (Porter and Duff 1984; Blinn 1994a).

The intermediate representation used for the foreground object between these two stages is called an *alpha-matted color image* (Figure 3.4b–c). In addition to the three color RGB channels, an alpha-matted image contains a fourth *alpha* channel α (or A) that describes the relative amount of *opacity* or *fractional coverage* at each pixel (Figures 3.4c and 3.5b). The opacity is the opposite of the *transparency*. Pixels within the object are fully opaque ($\alpha = 1$), while pixels fully outside the object are transparent ($\alpha = 0$). Pixels on the boundary of the object vary smoothly between these two extremes, which hides the perceptual visible *jaggies* that occur if only binary opacities are used.

To composite a new (or foreground) image on top of an old (background) image, the *over operator*, first proposed by Porter and Duff (1984) and then studied extensively by Blinn (1994a; 1994b), is used,

$$C = (1 - \alpha)B + \alpha F. \quad (3.8)$$

This operator *attenuates* the influence of the background image B by a factor $(1 - \alpha)$ and then adds in the color (and opacity) values corresponding to the foreground layer F , as shown in Figure 3.5.

In many situations, it is convenient to represent the foreground colors in *pre-multiplied* form, i.e., to store (and manipulate) the αF values directly. As Blinn (1994b) shows, the pre-multiplied RGBA representation is preferred for several reasons, including the ability to blur or resample (e.g., rotate) alpha-matted images without any additional complications (just treating each RGBA band independently). However, when matting using local color consistency (Ruzon and Tomasi 2000; Chuang, Curless, Salesin *et al.* 2001), the pure un-



Figure 3.6 An example of light reflecting off the transparent glass of a picture frame (Black and Anandan 1996) © 1996 Elsevier. You can clearly see the woman's portrait inside the picture frame superimposed with the reflection of a man's face off the glass.

multiplied foreground colors F are used, since these remain constant (or vary slowly) in the vicinity of the object edge.

The over operation is not the only kind of compositing operation that can be used. Porter and Duff (1984) describe a number of additional operations that can be useful in photo editing and visual effects applications. In this book, we concern ourselves with only one additional, commonly occurring case (but see Exercise 3.2).

When light reflects off clean transparent glass, the light passing through the glass and the light reflecting off the glass are simply added together (Figure 3.6). This model is useful in the analysis of *transparent motion* (Black and Anandan 1996; Szeliski, Avidan, and Anandan 2000), which occurs when such scenes are observed from a moving camera (see Section 8.5.2).

The actual process of *matting*, i.e., recovering the foreground, background, and alpha matte values from one or more images, has a rich history, which we study in Section 10.4. Smith and Blinn (1996) have a nice survey of traditional *blue-screen matting* techniques, while Toyama, Krumm, Brumitt *et al.* (1999) review *difference matting*. More recently, there has been a lot of activity in computational photography relating to *natural image matting* (Ruzon and Tomasi 2000; Chuang, Curless, Salesin *et al.* 2001; Wang and Cohen 2007a), which attempts to extract the mattes from a single natural image (Figure 3.4a) or from extended video sequences (Chuang, Agarwala, Curless *et al.* 2002). All of these techniques are described in more detail in Section 10.4.

3.1.4 Histogram equalization

While the brightness and gain controls described in Section 3.1.1 can improve the appearance of an image, how can we automatically determine their best values? One approach might be to look at the darkest and brightest pixel values in an image and map them to pure black and pure white. Another approach might be to find the *average* value in the image, push it towards middle gray, and expand the *range* so that it more closely fills the displayable values (Kopf, Uyttendaele, Deussen *et al.* 2007).

How can we visualize the set of lightness values in an image in order to test some of

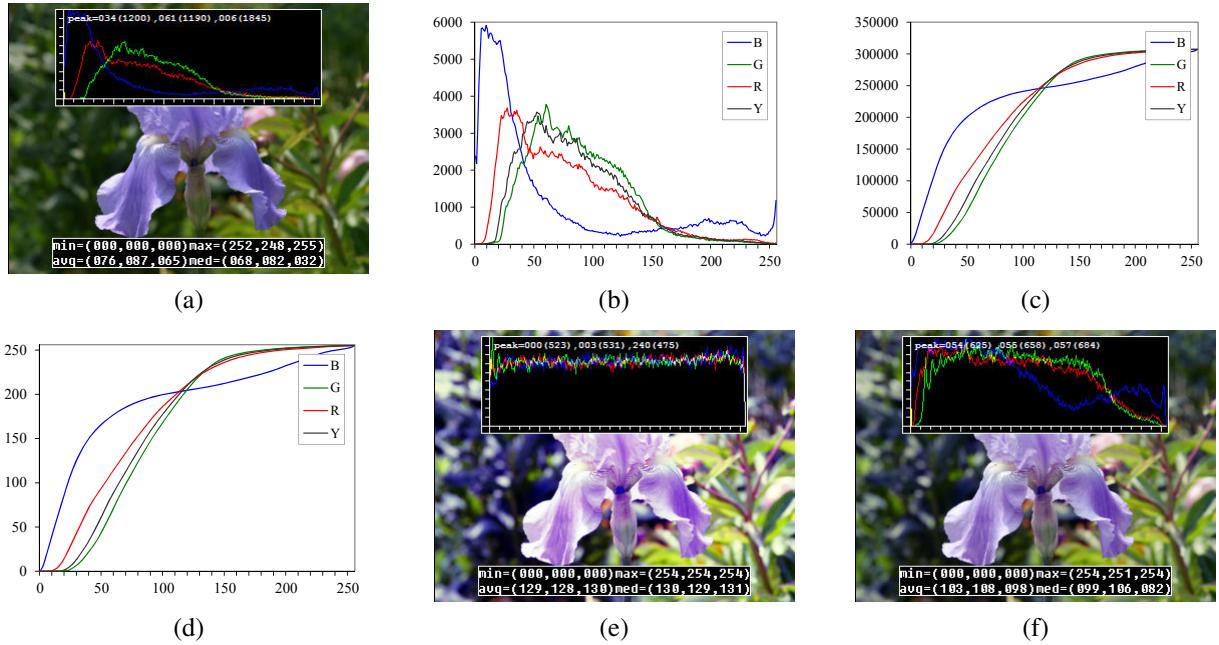


Figure 3.7 Histogram analysis and equalization: (a) original image (b) color channel and intensity (luminance) histograms; (c) cumulative distribution functions; (d) equalization (transfer) functions; (e) full histogram equalization; (f) partial histogram equalization.

these heuristics? The answer is to plot the *histogram* of the individual color channels and luminance values, as shown in Figure 3.7b.² From this distribution, we can compute relevant statistics such as the minimum, maximum, and average intensity values. Notice that the image in Figure 3.7a has both an excess of dark values and light values, but that the mid-range values are largely under-populated. Would it not be better if we could simultaneously brighten some dark values and darken some light values, while still using the full extent of the available dynamic range? Can you think of a mapping that might do this?

One popular answer to this question is to perform *histogram equalization*, i.e., to find an intensity mapping function $f(I)$ such that the resulting histogram is flat. The trick to finding such a mapping is the same one that people use to generate random samples from a *probability density function*, which is to first compute the *cumulative distribution function* shown in Figure 3.7c.

Think of the original histogram $h(I)$ as the distribution of grades in a class after some exam. How can we map a particular grade to its corresponding *percentile*, so that students at the 75% percentile range scored better than 3/4 of their classmates? The answer is to integrate the distribution $h(I)$ to obtain the cumulative distribution $c(I)$,

$$c(I) = \frac{1}{N} \sum_{i=0}^I h(i) = c(I-1) + \frac{1}{N} h(I), \quad (3.9)$$

² The histogram is simply the *count* of the number of pixels at each gray level value. For an eight-bit image, an accumulation table with 256 entries is needed. For higher bit depths, a table with the appropriate number of entries (probably fewer than the full number of gray levels) should be used.



Figure 3.8 Locally adaptive histogram equalization: (a) original image; (b) block histogram equalization; (c) full locally adaptive equalization.

where N is the number of pixels in the image or students in the class. For any given grade or intensity, we can look up its corresponding percentile $c(I)$ and determine the final value that pixel should take. When working with eight-bit pixel values, the I and c axes are rescaled from $[0, 255]$.

Figure 3.7d shows the result of applying $f(I) = c(I)$ to the original image. As we can see, the resulting histogram is flat; so is the resulting image (it is “flat” in the sense of a lack of contrast and being muddy looking). One way to compensate for this is to only *partially* compensate for the histogram unevenness, e.g., by using a mapping function $f(I) = \alpha c(I) + (1 - \alpha)I$, which is a linear blend between the cumulative distribution function and the identity transform (a straight line). As you can see in Figure 3.7e, the resulting image maintains more of its original grayscale distribution while having a more appealing balance.

Another potential problem with histogram equalization (or, in general, image brightening) is that noise in dark regions can be amplified and become more visible. Exercise 3.6 suggests some possible ways to mitigate this, as well as alternative techniques to maintain contrast and “punch” in the original images (Larson, Rushmeier, and Piatko 1997; Stark 2000).

Locally adaptive histogram equalization

While global histogram equalization can be useful, for some images it might be preferable to apply different kinds of equalization in different regions. Consider for example the image in Figure 3.8a, which has a wide range of luminance values. Instead of computing a single curve, what if we were to subdivide the image into $M \times M$ pixel blocks and perform separate histogram equalization in each sub-block? As you can see in Figure 3.8b, the resulting image exhibits a lot of blocking artifacts, i.e., intensity discontinuities at block boundaries.

One way to eliminate blocking artifacts is to use a *moving window*, i.e., to recompute the histogram for every $M \times M$ block centered at each pixel. This process can be quite slow (M^2 operations per pixel), although with clever programming only the histogram entries corresponding to the pixels entering and leaving the block (in a raster scan across the image) need to be updated (M operations per pixel). Note that this operation is an example of the *non-linear neighborhood operations* we study in more detail in Section 3.3.1.

A more efficient approach is to compute non-overlapped block-based equalization functions as before, but to then smoothly interpolate the transfer functions as we move between blocks. This technique is known as *adaptive histogram equalization* (AHE) and its contrast-

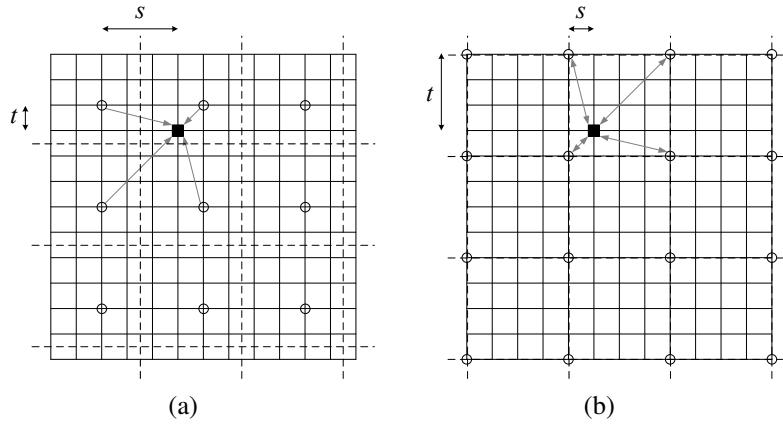


Figure 3.9 Local histogram interpolation using relative (s, t) coordinates: (a) block-based histograms, with block centers shown as circles; (b) corner-based “spline” histograms. Pixels are located on grid intersections. The black square pixel’s transfer function is interpolated from the four adjacent lookup tables (gray arrows) using the computed (s, t) values. Block boundaries are shown as dashed lines.

limited (gain-limited) version is known as CLAHE (Pizer, Amburn, Austin *et al.* 1987).³ The weighting function for a given pixel (i, j) can be computed as a function of its horizontal and vertical position (s, t) within a block, as shown in Figure 3.9a. To blend the four lookup functions $\{f_{00}, \dots, f_{11}\}$, a *bilinear* blending function,

$$f_{s,t}(I) = (1-s)(1-t)f_{00}(I) + s(1-t)f_{10}(I) + (1-s)tf_{01}(I) + stf_{11}(I) \quad (3.10)$$

can be used. (See Section 3.5.2 for higher-order generalizations of such *spline* functions.) Note that instead of blending the four lookup tables for each output pixel (which would be quite slow), we can instead blend the results of mapping a given pixel through the four neighboring lookups.

A variant on this algorithm is to place the lookup tables at the *corners* of each $M \times M$ block (see Figure 3.9b and Exercise 3.7). In addition to blending four lookups to compute the final value, we can also *distribute* each input pixel into four adjacent lookup tables during the histogram accumulation phase (notice that the gray arrows in Figure 3.9b point both ways), i.e.,

$$h_{k,l}(I(i, j)) += w(i, j, k, l), \quad (3.11)$$

where $w(i, j, k, l)$ is the bilinear weighting function between pixel (i, j) and lookup table (k, l) . This is an example of *soft histogramming*, which is used in a variety of other applications, including the construction of SIFT feature descriptors (Section 4.1.3) and vocabulary trees (Section 14.3.2).

3.1.5 Application: Tonal adjustment

One of the most widely used applications of point-wise image processing operators is the manipulation of contrast or *tone* in photographs, to make them look either more attractive or

³This algorithm is implemented in the MATLAB `adaphist` function.

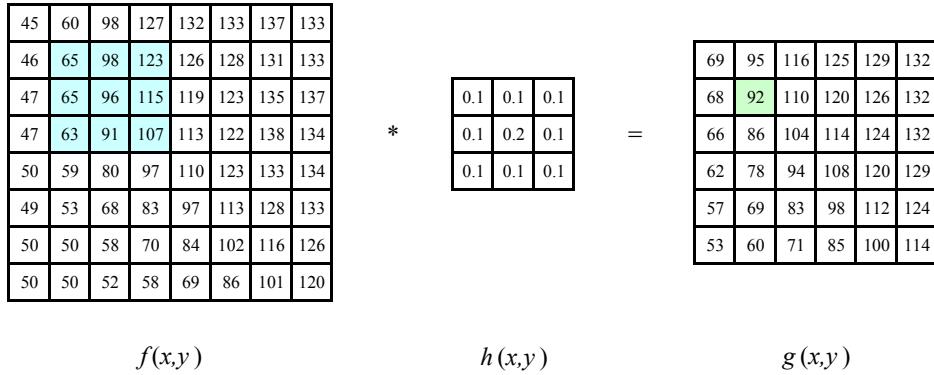


Figure 3.10 Neighborhood filtering (convolution): The image on the left is convolved with the filter in the middle to yield the image on the right. The light blue pixels indicate the source neighborhood for the light green destination pixel.

more interpretable. You can get a good sense of the range of operations possible by opening up any photo manipulation tool and trying out a variety of contrast, brightness, and color manipulation options, as shown in Figures 3.2 and 3.7.

Exercises 3.1, 3.5, and 3.6 have you implement some of these operations, in order to become familiar with basic image processing operators. More sophisticated techniques for tonal adjustment (Reinhard, Ward, Pattanaik *et al.* 2005; Bae, Paris, and Durand 2006) are described in the section on high dynamic range tone mapping (Section 10.2.1).

3.2 Linear filtering

Locally adaptive histogram equalization is an example of a *neighborhood operator* or *local operator*, which uses a collection of pixel values in the vicinity of a given pixel to determine its final output value (Figure 3.10). In addition to performing local tone adjustment, neighborhood operators can be used to *filter* images in order to add soft blur, sharpen details, accentuate edges, or remove noise (Figure 3.11b–d). In this section, we look at *linear* filtering operators, which involve weighted combinations of pixels in small neighborhoods. In Section 3.3, we look at non-linear operators such as morphological filters and distance transforms.

The most commonly used type of neighborhood operator is a *linear filter*, in which an output pixel's value is determined as a weighted sum of input pixel values (Figure 3.10),

$$g(i, j) = \sum_{k,l} f(i + k, j + l)h(k, l). \quad (3.12)$$

The entries in the weight *kernel* or *mask* $h(k, l)$ are often called the *filter coefficients*. The above *correlation* operator can be more compactly notated as

$$g = f \otimes h. \quad (3.13)$$

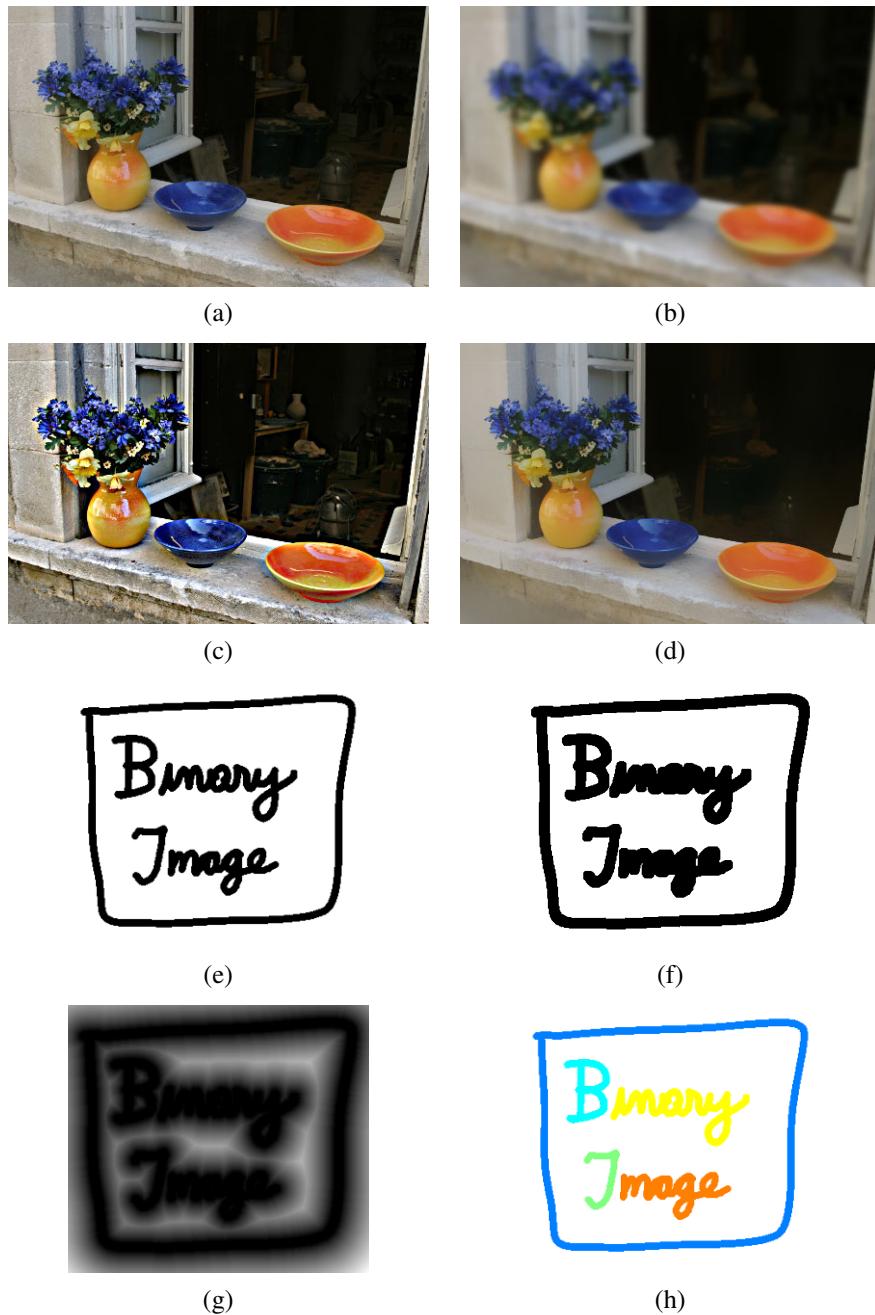


Figure 3.11 Some neighborhood operations: (a) original image; (b) blurred; (c) sharpened; (d) smoothed with edge-preserving filter; (e) binary image; (f) dilated; (g) distance transform; (h) connected components. For the dilation and connected components, black (ink) pixels are assumed to be active, i.e., to have a value of 1 in Equations (3.41–3.45).

$$\begin{bmatrix} 72 & 88 & 62 & 52 & 37 \end{bmatrix} * \begin{bmatrix} 1/4 & 1/2 & 1/4 \end{bmatrix} \Leftrightarrow \frac{1}{4} \begin{bmatrix} 2 & 1 & . & . & . \\ 1 & 2 & 1 & . & . \\ . & 1 & 2 & 1 & . \\ . & . & 1 & 2 & 1 \\ . & . & . & 1 & 2 \end{bmatrix} \begin{bmatrix} 72 \\ 88 \\ 62 \\ 52 \\ 37 \end{bmatrix}$$

Figure 3.12 One-dimensional signal convolution as a sparse matrix-vector multiply, $\mathbf{g} = \mathbf{H}\mathbf{f}$.

A common variant on this formula is

$$g(i, j) = \sum_{k, l} f(i - k, j - l)h(k, l) = \sum_{k, l} f(k, l)h(i - k, j - l), \quad (3.14)$$

where the sign of the offsets in f has been reversed. This is called the *convolution* operator,

$$g = f * h, \quad (3.15)$$

and h is then called the *impulse response function*.⁴ The reason for this name is that the kernel function, h , convolved with an impulse signal, $\delta(i, j)$ (an image that is 0 everywhere except at the origin) reproduces itself, $h * \delta = h$, whereas correlation produces the reflected signal. (Try this yourself to verify that it is so.)

In fact, Equation (3.14) can be interpreted as the superposition (addition) of shifted impulse response functions $h(i - k, j - l)$ multiplied by the input pixel values $f(k, l)$. Convolution has additional nice properties, e.g., it is both commutative and associative. As well, the Fourier transform of two convolved images is the product of their individual Fourier transforms (Section 3.4).

Both correlation and convolution are *linear shift-invariant* (LSI) operators, which obey both the superposition principle (3.5),

$$h \circ (f_0 + f_1) = h \circ f_0 + h \circ f_1, \quad (3.16)$$

and the *shift invariance* principle,

$$g(i, j) = f(i + k, j + l) \Leftrightarrow (h \circ g)(i, j) = (h \circ f)(i + k, j + l), \quad (3.17)$$

which means that shifting a signal commutes with applying the operator (\circ stands for the LSI operator). Another way to think of shift invariance is that the operator “behaves the same everywhere”.

Occasionally, a shift-variant version of correlation or convolution may be used, e.g.,

$$g(i, j) = \sum_{k, l} f(i - k, j - l)h(k, l; i, j), \quad (3.18)$$

where $h(k, l; i, j)$ is the convolution kernel at pixel (i, j) . For example, such a spatially varying kernel can be used to model blur in an image due to variable depth-dependent defocus.

Correlation and convolution can both be written as a matrix-vector multiply, if we first convert the two-dimensional images $f(i, j)$ and $g(i, j)$ into raster-ordered vectors \mathbf{f} and \mathbf{g} ,

$$\mathbf{g} = \mathbf{H}\mathbf{f}, \quad (3.19)$$

⁴ The continuous version of convolution can be written as $g(\mathbf{x}) = \int f(\mathbf{x} - \mathbf{u})h(\mathbf{u})d\mathbf{u}$.

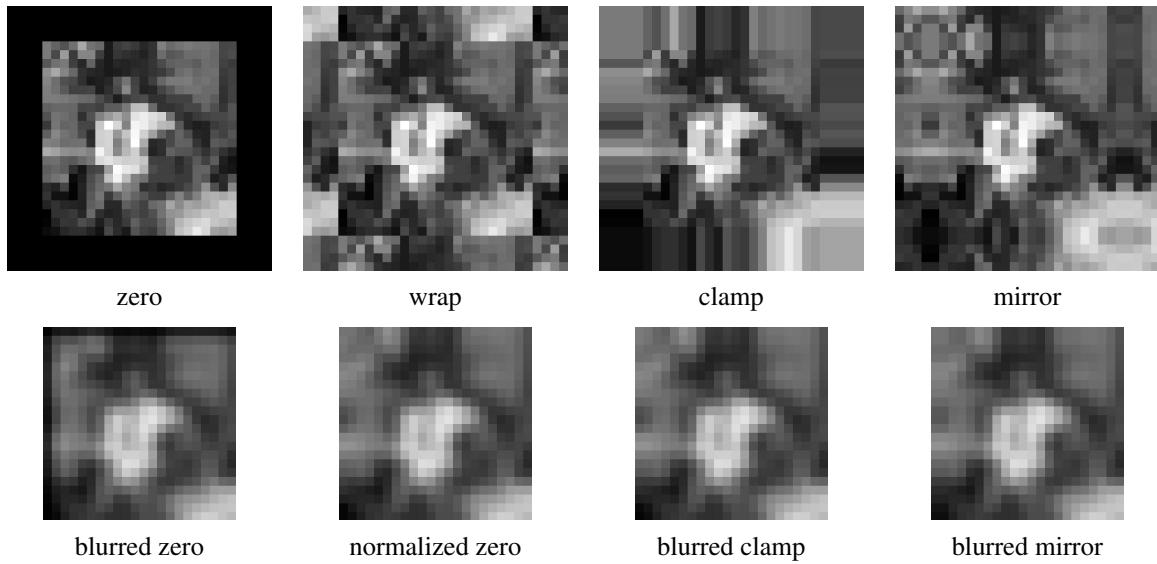


Figure 3.13 Border padding (top row) and the results of blurring the padded image (bottom row). The normalized zero image is the result of dividing (normalizing) the blurred zero-padded RGBA image by its corresponding soft alpha value.

where the (sparse) \mathbf{H} matrix contains the convolution kernels. Figure 3.12 shows how a one-dimensional convolution can be represented in matrix-vector form.

Padding (border effects)

The astute reader will notice that the matrix multiply shown in Figure 3.12 suffers from *boundary effects*, i.e., the results of filtering the image in this form will lead to a *darkening* of the corner pixels. This is because the original image is effectively being padded with 0 values wherever the convolution kernel extends beyond the original image boundaries.

To compensate for this, a number of alternative *padding* or extension modes have been developed (Figure 3.13):

- *zero*: set all pixels outside the source image to 0 (a good choice for alpha-matted cutout images);
- *constant (border color)*: set all pixels outside the source image to a specified *border* value;
- *clamp (replicate or clamp to edge)*: repeat edge pixels indefinitely;
- *(cyclic) wrap (repeat or tile)*: loop “around” the image in a “toroidal” configuration;
- *mirror*: reflect pixels across the image edge;
- *extend*: extend the signal by subtracting the mirrored version of the signal from the edge pixel value.

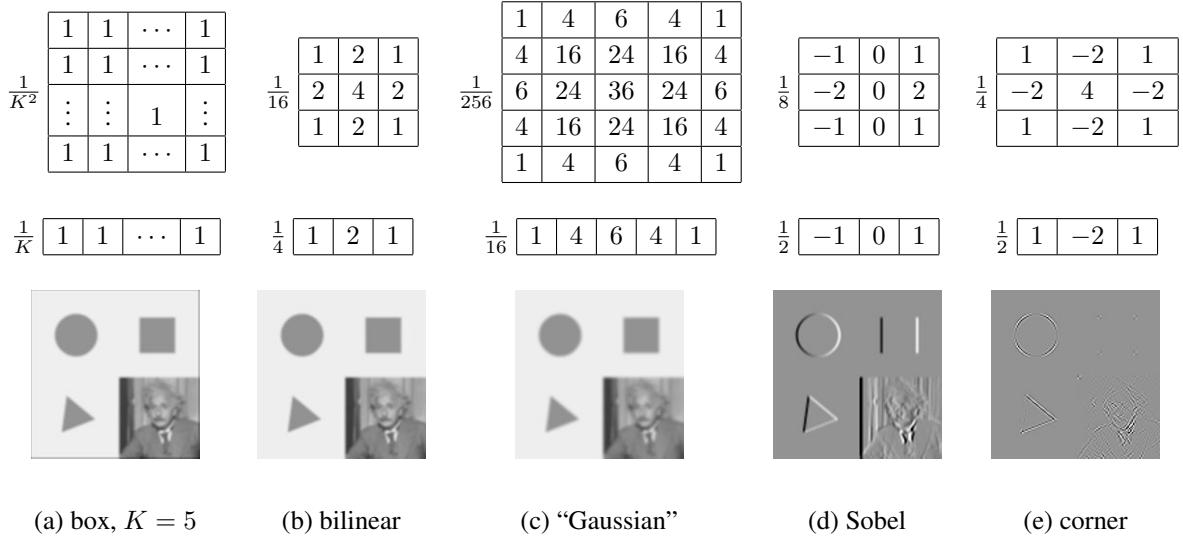


Figure 3.14 Separable linear filters: For each image (a)–(e), we show the 2D filter kernel (top), the corresponding horizontal 1D kernel (middle), and the filtered image (bottom). The filtered Sobel and corner images are signed, scaled up by $2\times$ and $4\times$, respectively, and added to a gray offset before display.

In the computer graphics literature (Akenine-Möller and Haines 2002, p. 124), these mechanisms are known as the *wrapping mode* (OpenGL) or *texture addressing mode* (Direct3D). The formulas for each of these modes are left to the reader (Exercise 3.8).

Figure 3.13 shows the effects of padding an image with each of the above mechanisms and then blurring the resulting padded image. As you can see, zero padding darkens the edges, clamp (replication) padding propagates border values inward, mirror (reflection) padding preserves colors near the borders. Extension padding (not shown) keeps the border pixels fixed (during blur).

An alternative to padding is to blur the zero-padded RGBA image and to then divide the resulting image by its alpha value to remove the darkening effect. The results can be quite good, as seen in the normalized zero image in Figure 3.13.

3.2.1 Separable filtering

The process of performing a convolution requires K^2 (multiply-add) operations per pixel, where K is the size (width or height) of the convolution kernel, e.g., the box filter in Figure 3.14a. In many cases, this operation can be significantly sped up by first performing a one-dimensional horizontal convolution followed by a one-dimensional vertical convolution (which requires a total of $2K$ operations per pixel). A convolution kernel for which this is possible is said to be *separable*.

It is easy to show that the two-dimensional kernel \mathbf{K} corresponding to successive convolution with a horizontal kernel \mathbf{h} and a vertical kernel \mathbf{v} is the *outer product* of the two kernels,

$$\mathbf{K} = \mathbf{v}\mathbf{h}^T \quad (3.20)$$

(see Figure 3.14 for some examples). Because of the increased efficiency, the design of

convolution kernels for computer vision applications is often influenced by their separability.

How can we tell if a given kernel \mathbf{K} is indeed separable? This can often be done by inspection or by looking at the analytic form of the kernel (Freeman and Adelson 1991). A more direct method is to treat the 2D kernel as a 2D matrix \mathbf{K} and to take its singular value decomposition (SVD),

$$\mathbf{K} = \sum_i \sigma_i \mathbf{u}_i \mathbf{v}_i^T \quad (3.21)$$

(see Appendix A.1.1 for the definition of the SVD). If only the first singular value σ_0 is non-zero, the kernel is separable and $\sqrt{\sigma_0} \mathbf{u}_0$ and $\sqrt{\sigma_0} \mathbf{v}_0^T$ provide the vertical and horizontal kernels (Perona 1995). For example, the Laplacian of Gaussian kernel (3.26 and 4.23) can be implemented as the sum of two separable filters (4.24) (Wiejak, Buxton, and Buxton 1985).

What if your kernel is not separable and yet you still want a faster way to implement it? Perona (1995), who first made the link between kernel separability and SVD, suggests using more terms in the (3.21) series, i.e., summing up a number of separable convolutions. Whether this is worth doing or not depends on the relative sizes of K and the number of significant singular values, as well as other considerations, such as cache coherency and memory locality.

3.2.2 Examples of linear filtering

Now that we have described the process for performing linear filtering, let us examine a number of frequently used filters.

The simplest filter to implement is the *moving average* or *box* filter, which simply averages the pixel values in a $K \times K$ window. This is equivalent to convolving the image with a kernel of all ones and then scaling (Figure 3.14a). For large kernels, a more efficient implementation is to slide a moving window across each scanline (in a separable filter) while adding the newest pixel and subtracting the oldest pixel from the running sum. This is related to the concept of *summed area tables*, which we describe shortly.

A smoother image can be obtained by separably convolving the image with a piecewise linear “tent” function (also known as a *Bartlett* filter). Figure 3.14b shows a 3×3 version of this filter, which is called the *bilinear* kernel, since it is the outer product of two linear (first-order) splines (see Section 3.5.2).

Convoluting the linear tent function with itself yields the cubic approximating spline, which is called the “Gaussian” kernel (Figure 3.14c) in Burt and Adelson’s (1983a) *Laplacian pyramid* representation (Section 3.5). Note that approximate Gaussian kernels can also be obtained by iterated convolution with box filters (Wells 1986). In applications where the filters really need to be rotationally symmetric, carefully tuned versions of sampled Gaussians should be used (Freeman and Adelson 1991) (Exercise 3.10).

The kernels we just discussed are all examples of blurring (smoothing) or *low-pass* kernels (since they pass through the lower frequencies while attenuating higher frequencies). How good are they at doing this? In Section 3.4, we use frequency-space Fourier analysis to examine the exact frequency response of these filters. We also introduce the *sinc* ($(\sin x)/x$) filter, which performs *ideal* low-pass filtering.

In practice, smoothing kernels are often used to reduce high-frequency noise. We have much more to say about using variants on smoothing to remove noise later (see Sections 3.3.1, 3.4, and 3.7).

Surprisingly, smoothing kernels can also be used to *sharpen* images using a process called *unsharp masking*. Since blurring the image reduces high frequencies, adding some of the difference between the original and the blurred image makes it sharper,

$$g_{\text{sharp}} = f + \gamma(f - h_{\text{blur}} * f). \quad (3.22)$$

In fact, before the advent of digital photography, this was the standard way to sharpen images in the darkroom: create a blurred (“positive”) negative from the original negative by misfocusing, then overlay the two negatives before printing the final image, which corresponds to

$$g_{\text{unsharp}} = f(1 - \gamma h_{\text{blur}} * f). \quad (3.23)$$

This is no longer a linear filter but it still works well.

Linear filtering can also be used as a pre-processing stage to edge extraction (Section 4.2) and interest point detection (Section 4.1) algorithms. Figure 3.14d shows a simple 3×3 edge extractor called the Sobel operator, which is a separable combination of a horizontal *central difference* (so called because the horizontal derivative is centered on the pixel) and a vertical tent filter (to smooth the results). As you can see in the image below the kernel, this filter effectively emphasizes horizontal edges.

The simple corner detector (Figure 3.14e) looks for simultaneous horizontal and vertical second derivatives. As you can see however, it responds not only to the corners of the square, but also along diagonal edges. Better corner detectors, or at least interest point detectors that are more rotationally invariant, are described in Section 4.1.

3.2.3 Band-pass and steerable filters

The Sobel and corner operators are simple examples of band-pass and oriented filters. More sophisticated kernels can be created by first smoothing the image with a (unit area) Gaussian filter,

$$G(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \quad (3.24)$$

and then taking the first or second derivatives (Marr 1982; Witkin 1983; Freeman and Adelson 1991). Such filters are known collectively as *band-pass filters*, since they filter out both low and high frequencies.

The (undirected) second derivative of a two-dimensional image,

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}, \quad (3.25)$$

is known as the *Laplacian* operator. Blurring an image with a Gaussian and then taking its Laplacian is equivalent to convolving directly with the *Laplacian of Gaussian* (LoG) filter,

$$\nabla^2 G(x, y; \sigma) = \left(\frac{x^2 + y^2}{\sigma^4} - \frac{2}{\sigma^2} \right) G(x, y; \sigma), \quad (3.26)$$

which has certain nice *scale-space properties* (Witkin 1983; Witkin, Terzopoulos, and Kass 1986). The five-point Laplacian is just a compact approximation to this more sophisticated filter.

Likewise, the Sobel operator is a simple approximation to a *directional* or *oriented* filter, which can be obtained by smoothing with a Gaussian (or some other filter) and then taking a

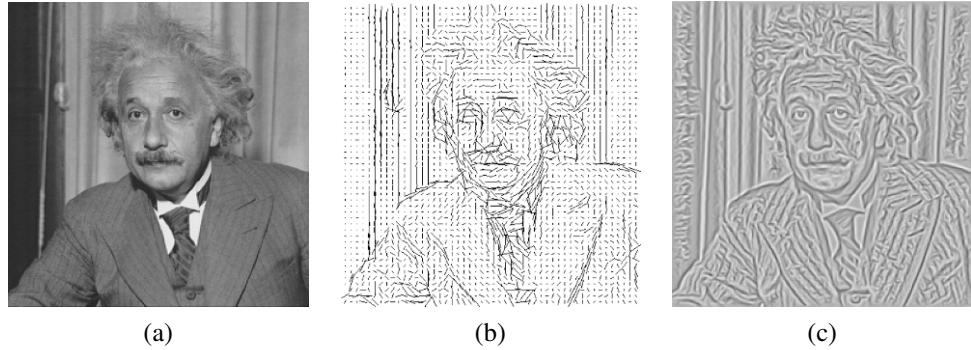


Figure 3.15 Second-order steerable filter (Freeman 1992) © 1992 IEEE: (a) original image of Einstein; (b) orientation map computed from the second-order oriented energy; (c) original image with oriented structures enhanced.

directional derivative $\nabla \hat{\mathbf{u}} = \frac{\partial}{\partial \hat{\mathbf{u}}}$, which is obtained by taking the dot product between the gradient field ∇ and a unit direction $\hat{\mathbf{u}} = (\cos \theta, \sin \theta)$,

$$\hat{\mathbf{u}} \cdot \nabla(G * f) = \nabla_{\hat{\mathbf{u}}}(G * f) = (\nabla_{\hat{\mathbf{u}}} G) * f. \quad (3.27)$$

The smoothed directional derivative filter,

$$G_{\hat{\mathbf{u}}} = uG_x + vG_y = u\frac{\partial G}{\partial x} + v\frac{\partial G}{\partial y}, \quad (3.28)$$

where $\hat{\mathbf{u}} = (u, v)$, is an example of a *steerable* filter, since the value of an image convolved with $G_{\hat{\mathbf{u}}}$ can be computed by first convolving with the pair of filters (G_x, G_y) and then *steering* the filter (potentially locally) by multiplying this gradient field with a unit vector $\hat{\mathbf{u}}$ (Freeman and Adelson 1991). The advantage of this approach is that a whole *family* of filters can be evaluated with very little cost.

How about steering a directional second derivative filter $\nabla_{\hat{\mathbf{u}}} \cdot \nabla_{\hat{\mathbf{u}}} G_{\hat{\mathbf{u}}}$, which is the result of taking a (smoothed) directional derivative and then taking the directional derivative again? For example, G_{xx} is the second directional derivative in the x direction.

At first glance, it would appear that the steering trick will not work, since for every direction $\hat{\mathbf{u}}$, we need to compute a different first directional derivative. Somewhat surprisingly, Freeman and Adelson (1991) showed that, for directional Gaussian derivatives, it is possible to steer *any* order of derivative with a relatively small number of basis functions. For example, only three basis functions are required for the second-order directional derivative,

$$G_{\hat{\mathbf{u}}\hat{\mathbf{u}}} = u^2 G_{xx} + 2uv G_{xy} + v^2 G_{yy}. \quad (3.29)$$

Furthermore, each of the basis filters, while not itself necessarily separable, can be computed using a linear combination of a small number of separable filters (Freeman and Adelson 1991).

This remarkable result makes it possible to construct directional derivative filters of increasingly greater *directional selectivity*, i.e., filters that only respond to edges that have strong local consistency in orientation (Figure 3.15). Furthermore, higher order steerable

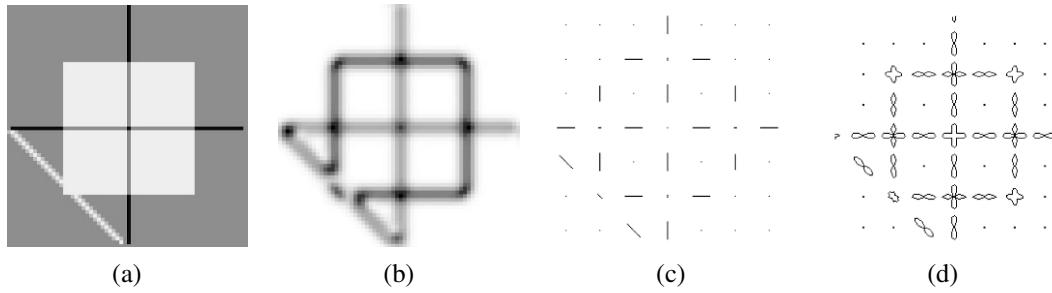


Figure 3.16 Fourth-order steerable filter (Freeman and Adelson 1991) © 1991 IEEE: (a) test image containing bars (lines) and step edges at different orientations; (b) average oriented energy; (c) dominant orientation; (d) oriented energy as a function of angle (polar plot).

filters can respond to potentially more than a single edge orientation at a given location, and they can respond to both *bar* edges (thin lines) and the classic step edges (Figure 3.16). In order to do this, however, full *Hilbert transform pairs* need to be used for second-order and higher filters, as described in (Freeman and Adelson 1991).

Steerable filters are often used to construct both feature descriptors (Section 4.1.3) and edge detectors (Section 4.2). While the filters developed by Freeman and Adelson (1991) are best suited for detecting linear (edge-like) structures, more recent work by Koethe (2003) shows how a combined 2×2 boundary tensor can be used to encode both edge and junction (“corner”) features. Exercise 3.12 has you implement such steerable filters and apply them to finding both edge and corner features.

Summed area table (integral image)

If an image is going to be repeatedly convolved with different box filters (and especially filters of different sizes at different locations), you can precompute the *summed area table* (Crow 1984), which is just the running sum of all the pixel values from the origin,

$$s(i, j) = \sum_{k=0}^i \sum_{l=0}^j f(k, l). \quad (3.30)$$

This can be efficiently computed using a recursive (raster-scan) algorithm,

$$s(i, j) = s(i - 1, j) + s(i, j - 1) - s(i - 1, j - 1) + f(i, j). \quad (3.31)$$

The image $s(i, j)$ is also often called an *integral image* (see Figure 3.17) and can actually be computed using only two additions per pixel if separate row sums are used (Viola and Jones 2004). To find the summed area (integral) inside a rectangle $[i_0, i_1] \times [j_0, j_1]$, we simply combine four samples from the summed area table,

$$S(i_0 \dots i_1, j_0 \dots j_1) = \sum_{i=i_0}^{i_1} \sum_{j=j_0}^{j_1} s(i_1, j_1) - s(i_1, j_0 - 1) - s(i_0 - 1, j_1) + s(i_0 - 1, j_0 - 1). \quad (3.32)$$

A potential disadvantage of summed area tables is that they require $\log M + \log N$ extra bits in the accumulation image compared to the original image, where M and N are the image

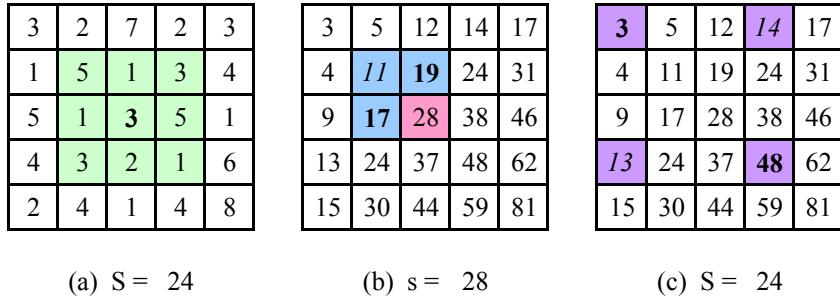


Figure 3.17 Summed area tables: (a) original image; (b) summed area table; (c) computation of area sum. Each value in the summed area table $s(i, j)$ (red) is computed recursively from its three adjacent (blue) neighbors (3.31). Area sums S (green) are computed by combining the four values at the rectangle corners (purple) (3.32). Positive values are shown in **bold** and negative values in *italics*.

width and height. Extensions of summed area tables can also be used to approximate other convolution kernels (Wolberg (1990, Section 6.5.2) contains a review).

In computer vision, summed area tables have been used in face detection (Viola and Jones 2004) to compute simple multi-scale low-level features. Such features, which consist of adjacent rectangles of positive and negative values, are also known as *boxlets* (Simard, Bottou, Haffner *et al.* 1998). In principle, summed area tables could also be used to compute the sums in the sum of squared differences (SSD) stereo and motion algorithms (Section 11.4). In practice, separable moving average filters are usually preferred (Kanade, Yoshida, Oda *et al.* 1996), unless many different window shapes and sizes are being considered (Veksler 2003).

Recursive filtering

The incremental formula (3.31) for the summed area is an example of a *recursive filter*, i.e., one whose values depends on previous filter outputs. In the signal processing literature, such filters are known as *infinite impulse response* (IIR), since the output of the filter to an impulse (single non-zero value) goes on forever. For example, for a summed area table, an impulse generates an infinite rectangle of 1s below and to the right of the impulse. The filters we have previously studied in this chapter, which involve the image with a finite extent kernel, are known as *finite impulse response* (FIR).

Two-dimensional IIR filters and recursive formulas are sometimes used to compute quantities that involve large area interactions, such as two-dimensional distance functions (Section 3.3.3) and connected components (Section 3.3.4).

More commonly, however, IIR filters are used inside one-dimensional separable filtering stages to compute large-extent smoothing kernels, such as efficient approximations to Gaussians and edge filters (Deriche 1990; Nielsen, Florack, and Deriche 1997). Pyramid-based algorithms (Section 3.5) can also be used to perform such large-area smoothing computations.

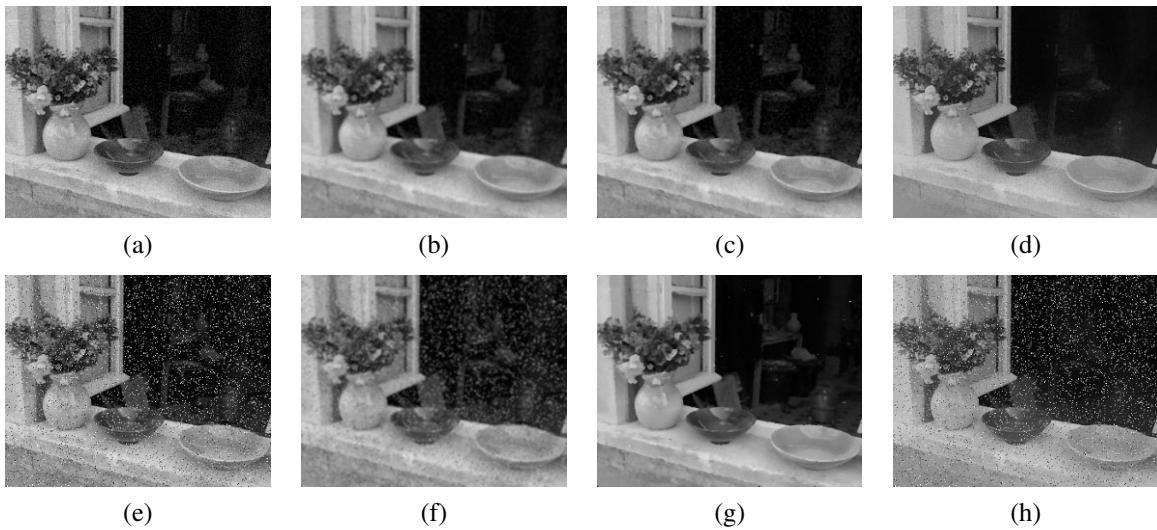


Figure 3.18 Median and bilateral filtering: (a) original image with Gaussian noise; (b) Gaussian filtered; (c) median filtered; (d) bilaterally filtered; (e) original image with shot noise; (f) Gaussian filtered; (g) median filtered; (h) bilaterally filtered. Note that the bilateral filter fails to remove the shot noise because the noisy pixels are too different from their neighbors.

3.3 More neighborhood operators

As we have just seen, linear filters can perform a wide variety of image transformations. However non-linear filters, such as edge-preserving median or bilateral filters, can sometimes perform even better. Other examples of neighborhood operators include *morphological* operators that operate on binary images, as well as *semi-global* operators that compute *distance transforms* and find *connected components* in binary images (Figure 3.11f–h).

3.3.1 Non-linear filtering

The filters we have looked at so far have all been *linear*, i.e., their response to a sum of two signals is the same as the sum of the individual responses. This is equivalent to saying that each output pixel is a weighted summation of some number of input pixels (3.19). Linear filters are easier to compose and are amenable to frequency response analysis (Section 3.4).

In many cases, however, better performance can be obtained by using a *non-linear* combination of neighboring pixels. Consider for example the image in Figure 3.18e, where the noise, rather than being Gaussian, is *shot noise*, i.e., it occasionally has very large values. In this case, regular blurring with a Gaussian filter fails to remove the noisy pixels and instead turns them into softer (but still visible) spots (Figure 3.18f).

Median filtering

A better filter to use in this case is the *median* filter, which selects the median value from each pixel's neighborhood (Figure 3.19a). Median values can be computed in expected linear time using a randomized select algorithm (Cormen 2001) and incremental variants have also been

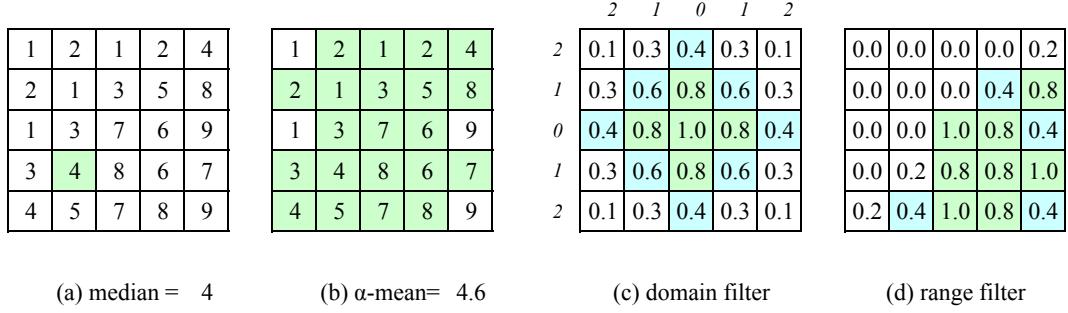


Figure 3.19 Median and bilateral filtering: (a) median pixel (green); (b) selected α -trimmed mean pixels; (c) domain filter (numbers along edge are pixel distances); (d) range filter.

developed by Tomasi and Manduchi (1998) and Bovik (2000, Section 3.2). Since the shot noise value usually lies well outside the true values in the neighborhood, the median filter is able to filter away such bad pixels (Figure 3.18c).

One downside of the median filter, in addition to its moderate computational cost, is that since it selects only one input pixel value to replace each output pixel, it is not as *efficient* at averaging away regular Gaussian noise (Huber 1981; Hampel, Ronchetti, Rousseeuw *et al.* 1986; Stewart 1999). A better choice may be the α -trimmed mean (Lee and Redner 1990) (Crane 1997, p. 109), which averages together all of the pixels except for the α fraction that are the smallest and the largest (Figure 3.19b).

Another possibility is to compute a *weighted median*, in which each pixel is used a number of times depending on its distance from the center. This turns out to be equivalent to minimizing the weighted objective function

$$\sum_{k,l} w(k,l) |f(i+k, j+l) - g(i, j)|^p, \quad (3.33)$$

where $g(i, j)$ is the desired output value and $p = 1$ for the weighted median. The value $p = 2$ is the usual *weighted mean*, which is equivalent to correlation (3.12) after normalizing by the sum of the weights (Bovik 2000, Section 3.2) (Haralick and Shapiro 1992, Section 7.2.6). The weighted mean also has deep connections to other methods in robust statistics (see Appendix B.3), such as influence functions (Huber 1981; Hampel, Ronchetti, Rousseeuw *et al.* 1986).

Non-linear smoothing has another, perhaps even more important property, especially since shot noise is rare in today's cameras. Such filtering is more *edge preserving*, i.e., it has less tendency to soften edges while filtering away high-frequency noise.

Consider the noisy image in Figure 3.18a. In order to remove most of the noise, the Gaussian filter is forced to smooth away high-frequency detail, which is most noticeable near strong edges. Median filtering does better but, as mentioned before, does not do as good a job at smoothing away discontinuities. See (Tomasi and Manduchi 1998) for some additional references to edge-preserving smoothing techniques.

While we could try to use the α -trimmed mean or weighted median, these techniques still have a tendency to round sharp corners, since the majority of pixels in the smoothing area come from the background distribution.

Bilateral filtering

What if we were to combine the idea of a weighted filter kernel with a better version of outlier rejection? What if instead of rejecting a fixed percentage α , we simply reject (in a soft way) pixels whose *values* differ too much from the central pixel value? This is the essential idea in *bilateral filtering*, which was first popularized in the computer vision community by Tomasi and Manduchi (1998). Chen, Paris, and Durand (2007) and Paris, Kornprobst, Tumblin *et al.* (2008) cite similar earlier work (Aurich and Weule 1995; Smith and Brady 1997) as well as the wealth of subsequent applications in computer vision and computational photography.

In the bilateral filter, the output pixel value depends on a weighted combination of neighboring pixel values

$$g(i, j) = \frac{\sum_{k,l} f(k, l) w(i, j, k, l)}{\sum_{k,l} w(i, j, k, l)}. \quad (3.34)$$

The weighting coefficient $w(i, j, k, l)$ depends on the product of a *domain kernel* (Figure 3.19c),

$$d(i, j, k, l) = \exp\left(-\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2}\right), \quad (3.35)$$

and a data-dependent *range kernel* (Figure 3.19d),

$$r(i, j, k, l) = \exp\left(-\frac{\|f(i, j) - f(k, l)\|^2}{2\sigma_r^2}\right). \quad (3.36)$$

When multiplied together, these yield the data-dependent *bilateral weight function*

$$w(i, j, k, l) = \exp\left(-\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2} - \frac{\|f(i, j) - f(k, l)\|^2}{2\sigma_r^2}\right). \quad (3.37)$$

Figure 3.20 shows an example of the bilateral filtering of a noisy step edge. Note how the domain kernel is the usual Gaussian, the range kernel measures appearance (intensity) similarity to the center pixel, and the bilateral filter kernel is a product of these two.

Notice that the range filter (3.36) uses the *vector distance* between the center and the neighboring pixel. This is important in color images, since an edge in any *one* of the color bands signals a change in material and hence the need to downweight a pixel's influence.⁵

Since bilateral filtering is quite slow compared to regular separable filtering, a number of acceleration techniques have been developed (Durand and Dorsey 2002; Paris and Durand 2006; Chen, Paris, and Durand 2007; Paris, Kornprobst, Tumblin *et al.* 2008). Unfortunately, these techniques tend to use more memory than regular filtering and are hence not directly applicable to filtering full-color images.

Iterated adaptive smoothing and anisotropic diffusion

Bilateral (and other) filters can also be applied in an iterative fashion, especially if an appearance more like a “cartoon” is desired (Tomasi and Manduchi 1998). When iterated filtering is applied, a much smaller neighborhood can often be used.

⁵ Tomasi and Manduchi (1998) show that using the vector distance (as opposed to filtering each color band separately) reduces color fringing effects. They also recommend taking the color difference in the more perceptually uniform CIELAB color space (see Section 2.3.2).

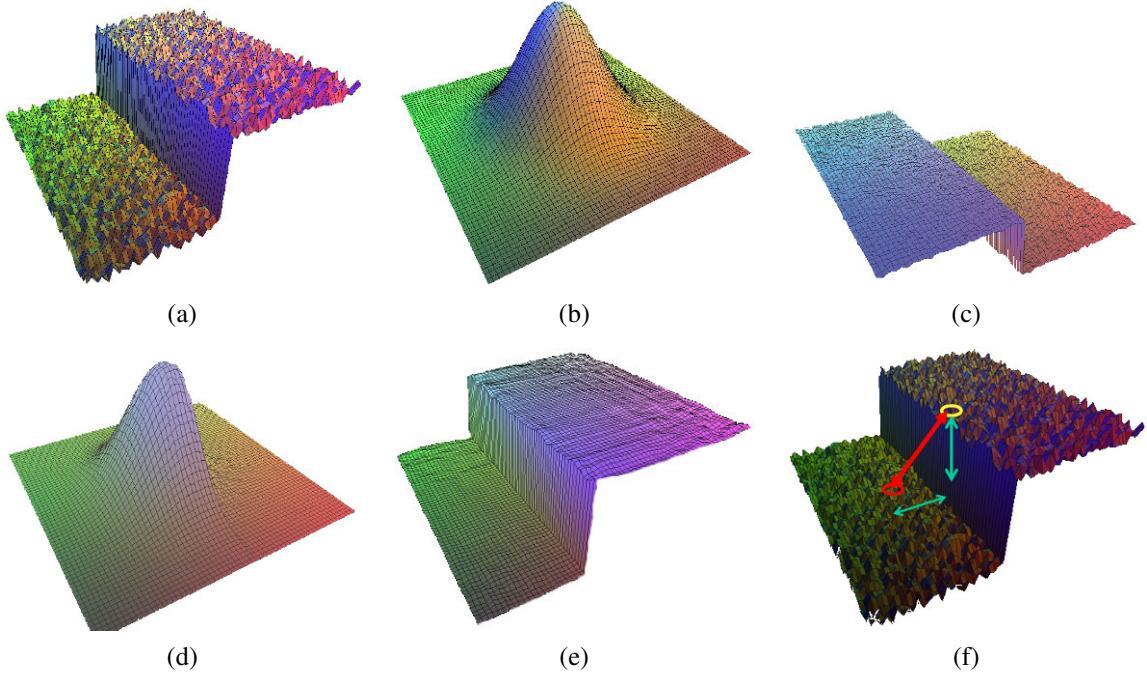


Figure 3.20 Bilateral filtering (Durand and Dorsey 2002) © 2002 ACM: (a) noisy step edge input; (b) domain filter (Gaussian); (c) range filter (similarity to center pixel value); (d) bilateral filter; (e) filtered step edge output; (f) 3D distance between pixels.

Consider, for example, using only the four nearest neighbors, i.e., restricting $|k - i| + |l - j| \leq 1$ in (3.34). Observe that

$$d(i, j, k, l) = \exp\left(-\frac{(i - k)^2 + (j - l)^2}{2\sigma_d^2}\right) \quad (3.38)$$

$$= \begin{cases} 1, & |k - i| + |l - j| = 0, \\ \lambda = e^{-1/2\sigma_d^2}, & |k - i| + |l - j| = 1. \end{cases} \quad (3.39)$$

We can thus re-write (3.34) as

$$\begin{aligned} f^{(t+1)}(i, j) &= \frac{f^{(t)}(i, j) + \eta \sum_{k, l} f^{(t)}(k, l) r(i, j, k, l)}{1 + \eta \sum_{k, l} r(i, j, k, l)} \\ &= f^{(t)}(i, j) + \frac{\eta}{1 + \eta R} \sum_{k, l} r(i, j, k, l) [f^{(t)}(k, l) - f^{(t)}(i, j)], \end{aligned} \quad (3.40)$$

where $R = \sum_{(k, l)} r(i, j, k, l)$, (k, l) are the \mathcal{N}_4 neighbors of (i, j) , and we have made the iterative nature of the filtering explicit.

As Barash (2002) notes, (3.40) is the same as the discrete *anisotropic diffusion* equation first proposed by Perona and Malik (1990b).⁶ Since its original introduction, anisotropic diffusion has been extended and applied to a wide range of problems (Nielsen, Florack, and Deriche 1997; Black, Sapiro, Marimont *et al.* 1998; Weickert, ter Haar Romeny, and Viergever

⁶ The $1/(1 + \eta R)$ factor is not present in anisotropic diffusion but becomes negligible as $\eta \rightarrow 0$.

1998; Weickert 1998). It has also been shown to be closely related to other *adaptive smoothing* techniques (Saint-Marc, Chen, and Medioni 1991; Barash 2002; Barash and Comaniciu 2004) as well as Bayesian regularization with a non-linear smoothness term that can be derived from image statistics (Scharr, Black, and Haussecker 2003).

In its general form, the range kernel $r(i, j, k, l) = r(\|f(i, j) - f(k, l)\|)$, which is usually called the *gain* or *edge-stopping* function, or diffusion coefficient, can be any monotonically increasing function with $r'(x) \rightarrow 0$ as $x \rightarrow \infty$. Black, Sapiro, Marimont *et al.* (1998) show how anisotropic diffusion is equivalent to minimizing a robust penalty function on the image gradients, which we discuss in Sections 3.7.1 and 3.7.2). Scharr, Black, and Haussecker (2003) show how the edge-stopping function can be derived in a principled manner from local image statistics. They also extend the diffusion neighborhood from \mathcal{N}_4 to \mathcal{N}_8 , which allows them to create a diffusion operator that is both rotationally invariant and incorporates information about the eigenvalues of the local structure tensor.

Note that, without a bias term towards the original image, anisotropic diffusion and iterative adaptive smoothing converge to a constant image. Unless a small number of iterations is used (e.g., for speed), it is usually preferable to formulate the smoothing problem as a joint minimization of a smoothness term and a data fidelity term, as discussed in Sections 3.7.1 and 3.7.2 and by Scharr, Black, and Haussecker (2003), which introduce such a bias in a principled manner.

3.3.2 Morphology

While non-linear filters are often used to enhance grayscale and color images, they are also used extensively to process binary images. Such images often occur after a *thresholding* operation,

$$\theta(f, t) = \begin{cases} 1 & \text{if } f \geq t, \\ 0 & \text{else,} \end{cases} \quad (3.41)$$

e.g., converting a scanned grayscale document into a binary image for further processing such as *optical character recognition*.

The most common binary image operations are called *morphological operations*, since they change the *shape* of the underlying binary objects (Ritter and Wilson 2000, Chapter 7). To perform such an operation, we first convolve the binary image with a binary *structuring element* and then select a binary output value depending on the thresholded result of the convolution. (This is not the usual way in which these operations are described, but I find it a nice simple way to unify the processes.) The structuring element can be any shape, from a simple 3×3 box filter, to more complicated disc structures. It can even correspond to a particular shape that is being sought for in the image.

Figure 3.21 shows a close-up of the convolution of a binary image f with a 3×3 structuring element s and the resulting images for the operations described below. Let

$$c = f \otimes s \quad (3.42)$$

be the integer-valued *count* of the number of 1s inside each structuring element as it is scanned over the image and S be the size of the structuring element (number of pixels). The standard operations used in binary morphology include:

- **dilation:** $\text{dilate}(f, s) = \theta(c, 1);$

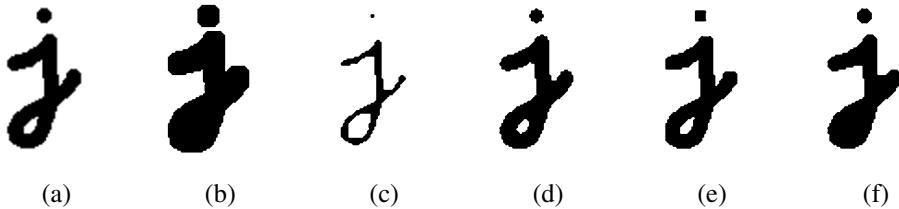


Figure 3.21 Binary image morphology: (a) original image; (b) dilation; (c) erosion; (d) majority; (e) opening; (f) closing. The structuring element for all examples is a 5×5 square. The effects of majority are a subtle rounding of sharp corners. Opening fails to eliminate the dot, since it is not wide enough.

- **erosion:** $\text{erode}(f, s) = \theta(c, S)$;
- **majority:** $\text{maj}(f, s) = \theta(c, S/2)$;
- **opening:** $\text{open}(f, s) = \text{dilate}(\text{erode}(f, s), s)$;
- **closing:** $\text{close}(f, s) = \text{erode}(\text{dilate}(f, s), s)$.

As we can see from Figure 3.21, dilation grows (thickens) objects consisting of 1s, while erosion shrinks (thins) them. The opening and closing operations tend to leave large regions and smooth boundaries unaffected, while removing small objects or holes and smoothing boundaries.

While we will not use mathematical morphology much in the rest of this book, it is a handy tool to have around whenever you need to clean up some thresholded images. You can find additional details on morphology in other textbooks on computer vision and image processing (Haralick and Shapiro 1992, Section 5.2) (Bovik 2000, Section 2.2) (Ritter and Wilson 2000, Section 7) as well as articles and books specifically on this topic (Serra 1982; Serra and Vincent 1992; Yuille, Vincent, and Geiger 1992; Soille 2006).

3.3.3 Distance transforms

The distance transform is useful in quickly precomputing the distance to a curve or set of points using a two-pass raster algorithm (Rosenfeld and Pfaltz 1966; Danielsson 1980; Borgefors 1986; Paglieroni 1992; Breu, Gil, Kirkpatrick *et al.* 1995; Felzenszwalb and Huttenlocher 2004a; Fabbri, Costa, Torelli *et al.* 2008). It has many applications, including level sets (Section 5.1.4), fast *chamfer matching* (binary image alignment) (Huttenlocher, Klanderman, and Ruckridge 1993), feathering in image stitching and blending (Section 9.3.2), and nearest point alignment (Section 12.2.1).

The distance transform $D(i, j)$ of a binary image $b(i, j)$ is defined as follows. Let $d(k, l)$ be some *distance metric* between pixel offsets. Two commonly used metrics include the *city block* or *Manhattan* distance

$$d_1(k, l) = |k| + |l| \quad (3.43)$$

and the *Euclidean* distance

$$d_2(k, l) = \sqrt{k^2 + l^2}. \quad (3.44)$$

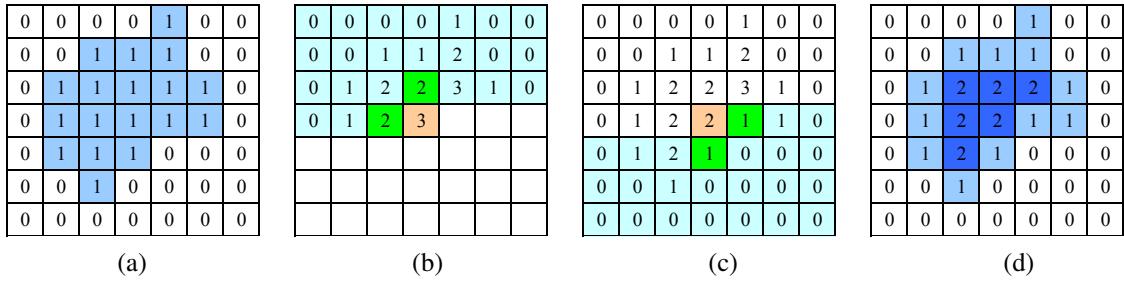


Figure 3.22 City block distance transform: (a) original binary image; (b) top to bottom (forward) raster sweep: green values are used to compute the orange value; (c) bottom to top (backward) raster sweep: green values are merged with old orange value; (d) final distance transform.

The distance transform is then defined as

$$D(i, j) = \min_{k, l: b(k, l)=0} d(i - k, j - l), \quad (3.45)$$

i.e., it is the distance to the *nearest* background pixel whose value is 0.

The D_1 city block distance transform can be efficiently computed using a forward and backward pass of a simple raster-scan algorithm, as shown in Figure 3.22. During the forward pass, each non-zero pixel in b is replaced by the minimum of $1 +$ the distance of its north or west neighbor. During the backward pass, the same occurs, except that the minimum is both over the current value D and $1 +$ the distance of the south and east neighbors (Figure 3.22).

Efficiently computing the Euclidean distance transform is more complicated. Here, just keeping the minimum scalar distance to the boundary during the two passes is not sufficient. Instead, a *vector-valued* distance consisting of both the x and y coordinates of the distance to the boundary must be kept and compared using the squared distance (hypotenuse) rule. As well, larger search regions need to be used to obtain reasonable results. Rather than explaining the algorithm (Danielsson 1980; Borgefors 1986) in more detail, we leave it as an exercise for the motivated reader (Exercise 3.13).

Figure 3.11g shows a distance transform computed from a binary image. Notice how the values grow away from the black (ink) regions and form ridges in the white area of the original image. Because of this linear growth from the starting boundary pixels, the distance transform is also sometimes known as the *grassfire transform*, since it describes the time at which a fire starting inside the black region would consume any given pixel, or a *chamfer*, because it resembles similar shapes used in woodworking and industrial design. The ridges in the distance transform become the *skeleton* (or *medial axis transform (MAT)*) of the region where the transform is computed, and consist of pixels that are of equal distance to two (or more) boundaries (Tek and Kimia 2003; Sebastian and Kimia 2005).

A useful extension of the basic distance transform is the *signed distance transform*, which computes distances to boundary pixels for *all* the pixels (Lavallée and Szeliski 1995). The simplest way to create this is to compute the distance transforms for both the original binary image and its complement and to negate one of them before combining. Because such distance fields tend to be smooth, it is possible to store them more compactly (with minimal loss in *relative* accuracy) using a spline defined over a quadtree or octree data structure

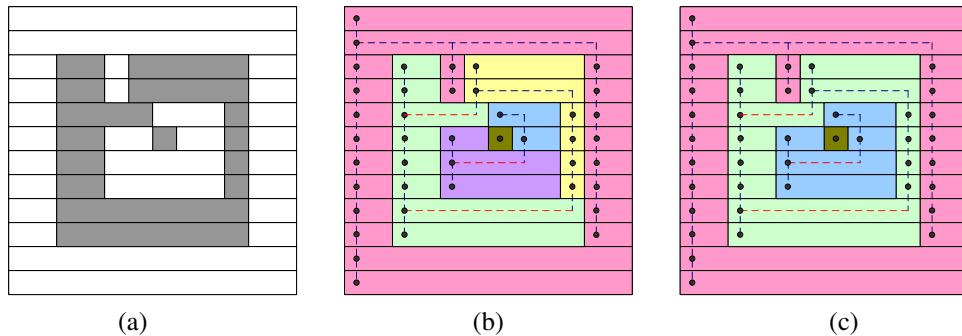


Figure 3.23 Connected component computation: (a) original grayscale image; (b) horizontal runs (nodes) connected by vertical (graph) edges (dashed blue)—runs are pseudocolored with unique colors inherited from parent nodes; (c) re-coloring after merging adjacent segments.

(Lavallée and Szeliski 1995; Szeliski and Lavallée 1996; Friskin, Perry, Rockwood *et al.* 2000). Such precomputed signed distance transforms can be extremely useful in efficiently aligning and merging 2D curves and 3D surfaces (Huttenlocher, Klanderman, and Ruckridge 1993; Szeliski and Lavallée 1996; Curless and Levoy 1996), especially if the *vectorial* version of the distance transform, i.e., a pointer from each pixel or voxel to the nearest boundary or surface element, is stored and interpolated. Signed distance fields are also an essential component of level set evolution (Section 5.1.4), where they are called *characteristic functions*.

3.3.4 Connected components

Another useful semi-global image operation is finding *connected components*, which are defined as regions of adjacent pixels that have the same input value (or label). (In the remainder of this section, consider pixels to be *adjacent* if they are immediate \mathcal{N}_4 neighbors and they have the same input value.) Connected components can be used in a variety of applications, such as finding individual letters in a scanned document or finding objects (say, cells) in a thresholded image and computing their area statistics.

Consider the grayscale image in Figure 3.23a. There are four connected components in this figure: the outermost set of white pixels, the large ring of gray pixels, the white enclosed region, and the single gray pixel. These are shown pseudocolored in Figure 3.23c as pink, green, blue, and brown.

To compute the connected components of an image, we first (conceptually) split the image into horizontal *runs* of adjacent pixels, and then color the runs with unique labels, re-using the labels of vertically adjacent runs whenever possible. In a second phase, adjacent runs of different colors are then merged.

While this description is a little sketchy, it should be enough to enable a motivated student to implement this algorithm (Exercise 3.14). Haralick and Shapiro (1992, Section 2.3) give a much longer description of various connected component algorithms, including ones that avoid the creation of a potentially large re-coloring (equivalence) table. Well-debugged connected component algorithms are also available in most image processing libraries.

Once a binary or multi-valued image has been segmented into its connected components,

it is often useful to compute the area statistics for each individual region \mathcal{R} . Such statistics include:

- the area (number of pixels);
- the perimeter (number of boundary pixels);
- the centroid (average x and y values);
- the second moments,

$$\mathbf{M} = \sum_{(x,y) \in \mathcal{R}} \begin{bmatrix} x - \bar{x} \\ y - \bar{y} \end{bmatrix} \begin{bmatrix} x - \bar{x} & y - \bar{y} \end{bmatrix}, \quad (3.46)$$

from which the major and minor axis orientation and lengths can be computed using eigenvalue analysis.⁷

These statistics can then be used for further processing, e.g., for sorting the regions by the area size (to consider the largest regions first) or for preliminary matching of regions in different images.

3.4 Fourier transforms

In Section 3.2, we mentioned that Fourier analysis could be used to analyze the frequency characteristics of various filters. In this section, we explain both how Fourier analysis lets us determine these characteristics (or equivalently, the frequency *content* of an image) and how using the Fast Fourier Transform (FFT) lets us perform large-kernel convolutions in time that is independent of the kernel's size. More comprehensive introductions to Fourier transforms are provided by Bracewell (1986); Glassner (1995); Oppenheim and Schafer (1996); Oppenheim, Schafer, and Buck (1999).

How can we analyze what a given filter does to high, medium, and low frequencies? The answer is to simply pass a sinusoid of known frequency through the filter and to observe by how much it is attenuated. Let

$$s(x) = \sin(2\pi f x + \phi_i) = \sin(\omega x + \phi_i) \quad (3.47)$$

be the input sinusoid whose *frequency* is f , *angular frequency* is $\omega = 2\pi f$, and *phase* is ϕ_i . Note that in this section, we use the variables x and y to denote the spatial coordinates of an image, rather than i and j as in the previous sections. This is both because the letters i and j are used for the *imaginary* number (the usage depends on whether you are reading complex variables or electrical engineering literature) and because it is clearer how to distinguish the horizontal (x) and vertical (y) components in frequency space. In this section, we use the letter j for the imaginary number, since that is the form more commonly found in the signal processing literature (Bracewell 1986; Oppenheim and Schafer 1996; Oppenheim, Schafer, and Buck 1999).

⁷ Moments can also be computed using Green's theorem applied to the boundary pixels (Yang and Albregtsen 1996).

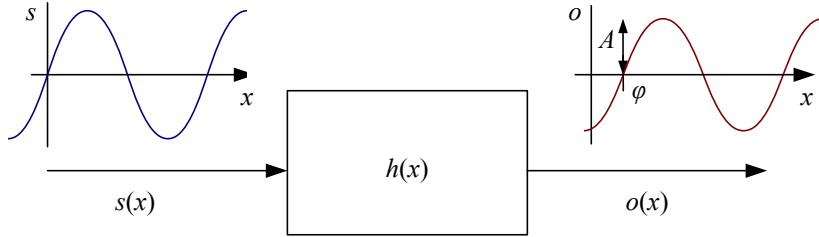


Figure 3.24 The Fourier Transform as the response of a filter $h(x)$ to an input sinusoid $s(x) = e^{j\omega x}$ yielding an output sinusoid $o(x) = h(x) * s(x) = Ae^{j\omega x + \phi}$.

If we convolve the sinusoidal signal $s(x)$ with a filter whose impulse response is $h(x)$, we get another sinusoid of the same frequency but different magnitude A and phase ϕ_o ,

$$o(x) = h(x) * s(x) = A \sin(\omega x + \phi_o), \quad (3.48)$$

as shown in Figure 3.24. To see that this is the case, remember that a convolution can be expressed as a weighted summation of shifted input signals (3.14) and that the summation of a bunch of shifted sinusoids of the same frequency is just a single sinusoid at that frequency.⁸ The new magnitude A is called the *gain* or *magnitude* of the filter, while the phase difference $\Delta\phi = \phi_o - \phi_i$ is called the *shift* or *phase*.

In fact, a more compact notation is to use the complex-valued sinusoid

$$s(x) = e^{j\omega x} = \cos \omega x + j \sin \omega x. \quad (3.49)$$

In that case, we can simply write,

$$o(x) = h(x) * s(x) = Ae^{j\omega x + \phi}. \quad (3.50)$$

The *Fourier transform* is simply a tabulation of the magnitude and phase response at each frequency,

$$H(\omega) = \mathcal{F}\{h(x)\} = Ae^{j\phi}, \quad (3.51)$$

i.e., it is the response to a complex sinusoid of frequency ω passed through the filter $h(x)$. The Fourier transform pair is also often written as

$$h(x) \xleftrightarrow{\mathcal{F}} H(\omega). \quad (3.52)$$

Unfortunately, (3.51) does not give an actual *formula* for computing the Fourier transform. Instead, it gives a *recipe*, i.e., convolve the filter with a sinusoid, observe the magnitude and phase shift, repeat. Fortunately, closed form equations for the Fourier transform exist both in the continuous domain,

$$H(\omega) = \int_{-\infty}^{\infty} h(x)e^{-j\omega x} dx, \quad (3.53)$$

⁸ If h is a general (non-linear) transform, additional *harmonic* frequencies are introduced. This was traditionally the bane of audiophiles, who insisted on equipment with no *harmonic distortion*. Now that digital audio has introduced pure distortion-free sound, some audiophiles are buying retro tube amplifiers or digital signal processors that simulate such distortions because of their “warmer sound”.

Property	Signal	Transform
superposition	$f_1(x) + f_2(x)$	$F_1(\omega) + F_2(\omega)$
shift	$f(x - x_0)$	$F(\omega)e^{-j\omega x_0}$
reversal	$f(-x)$	$F^*(\omega)$
convolution	$f(x) * h(x)$	$F(\omega)H(\omega)$
correlation	$f(x) \otimes h(x)$	$F(\omega)H^*(\omega)$
multiplication	$f(x)h(x)$	$F(\omega) * H(\omega)$
differentiation	$f'(x)$	$j\omega F(\omega)$
domain scaling	$f(ax)$	$1/aF(\omega/a)$
real images	$f(x) = f^*(x) \Leftrightarrow F(\omega) = F(-\omega)$	
Parseval's Theorem	$\sum_x [f(x)]^2$	$= \sum_\omega [F(\omega)]^2$

Table 3.1 Some useful properties of Fourier transforms. The original transform pair is $F(\omega) = \mathcal{F}\{f(x)\}$.

and in the discrete domain,

$$H(k) = \frac{1}{N} \sum_{x=0}^{N-1} h(x) e^{-j \frac{2\pi k x}{N}}, \quad (3.54)$$

where N is the length of the signal or region of analysis. These formulas apply both to filters, such as $h(x)$, and to signals or images, such as $s(x)$ or $g(x)$.

The discrete form of the Fourier transform (3.54) is known as the *Discrete Fourier Transform* (DFT). Note that while (3.54) can be evaluated for any value of k , it only makes sense for values in the range $k \in [-\frac{N}{2}, \frac{N}{2}]$. This is because larger values of k alias with lower frequencies and hence provide no additional information, as explained in the discussion on aliasing in Section 2.3.1.

At face value, the DFT takes $O(N^2)$ operations (multiply-adds) to evaluate. Fortunately, there exists a faster algorithm called the *Fast Fourier Transform* (FFT), which requires only $O(N \log_2 N)$ operations (Bracewell 1986; Oppenheim, Schafer, and Buck 1999). We do not explain the details of the algorithm here, except to say that it involves a series of $\log_2 N$ stages, where each stage performs small 2×2 transforms (matrix multiplications with known coefficients) followed by some semi-global permutations. (You will often see the term *butterfly* applied to these stages because of the pictorial shape of the signal processing graphs involved.) Implementations for the FFT can be found in most numerical and signal processing libraries.

Now that we have defined the Fourier transform, what are some of its properties and how can they be used? Table 3.1 lists a number of useful properties, which we describe in a little more detail below:

- **Superposition:** The Fourier transform of a sum of signals is the sum of their Fourier transforms. Thus, the Fourier transform is a linear operator.
- **Shift:** The Fourier transform of a shifted signal is the transform of the original signal multiplied by a *linear phase shift* (complex sinusoid).

- **Reversal:** The Fourier transform of a reversed signal is the complex conjugate of the signal's transform.
- **Convolution:** The Fourier transform of a pair of convolved signals is the product of their transforms.
- **Correlation:** The Fourier transform of a correlation is the product of the first transform times the complex conjugate of the second one.
- **Multiplication:** The Fourier transform of the product of two signals is the convolution of their transforms.
- **Differentiation:** The Fourier transform of the derivative of a signal is that signal's transform multiplied by the frequency. In other words, differentiation linearly emphasizes (magnifies) higher frequencies.
- **Domain scaling:** The Fourier transform of a stretched signal is the equivalently compressed (and scaled) version of the original transform and *vice versa*.
- **Real images:** The Fourier transform of a real-valued signal is symmetric around the origin. This fact can be used to save space and to double the speed of image FFTs by packing alternating scanlines into the real and imaginary parts of the signal being transformed.
- **Parseval's Theorem:** The energy (sum of squared values) of a signal is the same as the energy of its Fourier transform.

All of these properties are relatively straightforward to prove (see Exercise 3.15) and they will come in handy later in the book, e.g., when designing optimum Wiener filters (Section 3.4.3) or performing fast image correlations (Section 8.1.2).

3.4.1 Fourier transform pairs

Now that we have these properties in place, let us look at the Fourier transform pairs of some commonly occurring filters and signals, as listed in Table 3.2. In more detail, these pairs are as follows:

- **Impulse:** The impulse response has a constant (all frequency) transform.
- **Shifted impulse:** The shifted impulse has unit magnitude and linear phase.
- **Box filter:** The box (moving average) filter

$$\text{box}(x) = \begin{cases} 1 & \text{if } |x| \leq 1 \\ 0 & \text{else} \end{cases} \quad (3.55)$$

has a sinc Fourier transform,

$$\text{sinc}(\omega) = \frac{\sin \omega}{\omega}, \quad (3.56)$$

which has an infinite number of side lobes. Conversely, the sinc filter is an ideal low-pass filter. For a non-unit box, the width of the box a and the spacing of the zero crossings in the sinc $1/a$ are inversely proportional.

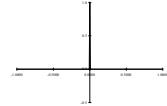
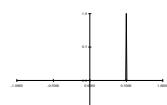
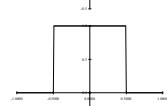
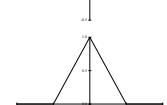
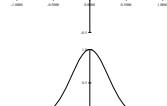
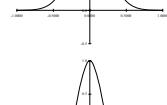
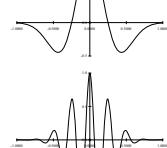
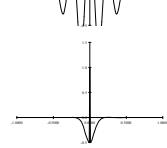
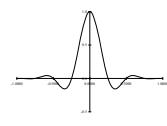
Name	Signal	\Leftrightarrow	Transform
impulse		$\delta(x)$	1
shifted impulse		$\delta(x - u)$	$e^{-j\omega u}$
box filter		\Leftrightarrow	$a \text{sinc}(a\omega)$
tent		\Leftrightarrow	$a \text{sinc}^2(a\omega)$
Gaussian		\Leftrightarrow	$\frac{\sqrt{2\pi}}{\sigma} G(\omega; \sigma^{-1})$
Laplacian of Gaussian		\Leftrightarrow	$-\frac{\sqrt{2\pi}}{\sigma} \omega^2 G(\omega; \sigma^{-1})$
Gabor		\Leftrightarrow	$\frac{\sqrt{2\pi}}{\sigma} G(\omega \pm \omega_0; \sigma^{-1})$
unsharp mask		\Leftrightarrow	$\frac{(1+\gamma)-}{\sqrt{2\pi}\gamma} G(\omega; \sigma^{-1})$
windowed sinc		\Leftrightarrow	(see Figure 3.29)

Table 3.2 Some useful (continuous) Fourier transform pairs: The dashed line in the Fourier transform of the shifted impulse indicates its (linear) phase. All other transforms have zero phase (they are real-valued). Note that the figures are not necessarily drawn to scale but are drawn to illustrate the general shape and characteristics of the filter or its response. In particular, the Laplacian of Gaussian is drawn inverted because it resembles more a “Mexican hat”, as it is sometimes called.

- **Tent:** The piecewise linear tent function,

$$\text{tent}(x) = \max(0, 1 - |x|), \quad (3.57)$$

has a sinc^2 Fourier transform.

- **Gaussian:** The (unit area) Gaussian of width σ ,

$$G(x; \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}, \quad (3.58)$$

has a (unit height) Gaussian of width σ^{-1} as its Fourier transform.

- **Laplacian of Gaussian:** The second derivative of a Gaussian of width σ ,

$$\text{LoG}(x; \sigma) = \left(\frac{x^2}{\sigma^4} - \frac{1}{\sigma^2}\right) G(x; \sigma) \quad (3.59)$$

has a band-pass response of

$$-\frac{\sqrt{2\pi}}{\sigma} \omega^2 G(\omega; \sigma^{-1}) \quad (3.60)$$

as its Fourier transform.

- **Gabor:** The even Gabor function, which is the product of a cosine of frequency ω_0 and a Gaussian of width σ , has as its transform the sum of the two Gaussians of width σ^{-1} centered at $\omega = \pm\omega_0$. The odd Gabor function, which uses a sine, is the difference of two such Gaussians. Gabor functions are often used for oriented and band-pass filtering, since they can be more frequency selective than Gaussian derivatives.
- **Unsharp mask:** The unsharp mask introduced in (3.22) has as its transform a unit response with a slight boost at higher frequencies.

- **Windowed sinc:** The windowed (masked) sinc function shown in Table 3.2 has a response function that approximates an ideal low-pass filter better and better as additional side lobes are added (W is increased). Figure 3.29 shows the shapes of these such filters along with their Fourier transforms. For these examples, we use a one-lobe raised cosine,

$$\text{rcos}(x) = \frac{1}{2}(1 + \cos \pi x)\text{box}(x), \quad (3.61)$$

also known as the *Hann window*, as the windowing function. Wolberg (1990) and Oppenheim, Schafer, and Buck (1999) discuss additional windowing functions, which include the *Lanczos* window, the positive first lobe of a sinc function.

We can also compute the Fourier transforms for the small discrete kernels shown in Figure 3.14 (see Table 3.3). Notice how the moving average filters do not uniformly dampen higher frequencies and hence can lead to ringing artifacts. The binomial filter (Gomes and Velho 1997) used as the “Gaussian” in Burt and Adelson’s (1983a) Laplacian pyramid (see Section 3.5), does a decent job of separating the high and low frequencies, but still leaves a fair amount of high-frequency detail, which can lead to aliasing after downsampling. The Sobel edge detector at first linearly accentuates frequencies, but then decays at higher frequencies, and hence has trouble detecting fine-scale edges, e.g., adjacent black and white columns. We look at additional examples of small kernel Fourier transforms in Section 3.5.2, where we study better kernels for pre-filtering before decimation (size reduction).

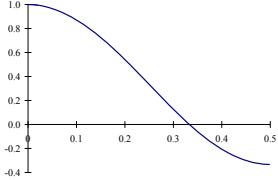
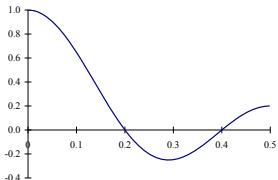
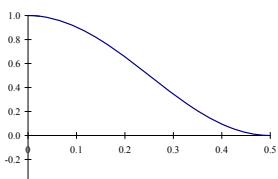
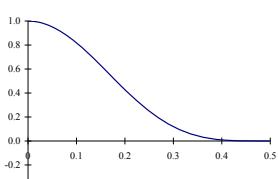
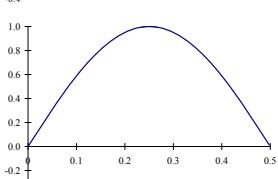
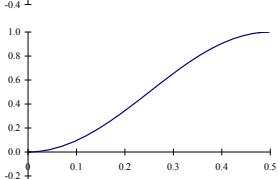
Name	Kernel	Transform	Plot
box-3	$\frac{1}{3} [1 \ 1 \ 1]$	$\frac{1}{3}(1 + 2 \cos \omega)$	
box-5	$\frac{1}{5} [1 \ 1 \ 1 \ 1 \ 1]$	$\frac{1}{5}(1 + 2 \cos \omega + 2 \cos 2\omega)$	
linear	$\frac{1}{4} [1 \ 2 \ 1]$	$\frac{1}{2}(1 + \cos \omega)$	
binomial	$\frac{1}{16} [1 \ 4 \ 6 \ 4 \ 1]$	$\frac{1}{4}(1 + \cos \omega)^2$	
Sobel	$\frac{1}{2} [-1 \ 0 \ 1]$	$\sin \omega$	
corner	$\frac{1}{2} [-1 \ 2 \ -1]$	$\frac{1}{2}(1 - \cos \omega)$	

Table 3.3 Fourier transforms of the separable kernels shown in Figure 3.14.

3.4.2 Two-dimensional Fourier transforms

The formulas and insights we have developed for one-dimensional signals and their transforms translate directly to two-dimensional images. Here, instead of just specifying a horizontal or vertical frequency ω_x or ω_y , we can create an oriented sinusoid of frequency (ω_x, ω_y) ,

$$s(x, y) = \sin(\omega_x x + \omega_y y). \quad (3.62)$$

The corresponding two-dimensional Fourier transforms are then

$$H(\omega_x, \omega_y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(x, y) e^{-j(\omega_x x + \omega_y y)} dx dy, \quad (3.63)$$

and in the discrete domain,

$$H(k_x, k_y) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h(x, y) e^{-j2\pi \frac{k_x x + k_y y}{MN}}, \quad (3.64)$$

where M and N are the width and height of the image.

All of the Fourier transform properties from Table 3.1 carry over to two dimensions if we replace the scalar variables x , ω , x_0 and a with their 2D vector counterparts $\mathbf{x} = (x, y)$, $\boldsymbol{\omega} = (\omega_x, \omega_y)$, $\mathbf{x}_0 = (x_0, y_0)$, and $\mathbf{a} = (a_x, a_y)$, and use vector inner products instead of multiplications.

3.4.3 Wiener filtering

While the Fourier transform is a useful tool for analyzing the frequency characteristics of a filter kernel or image, it can also be used to analyze the frequency spectrum of a whole *class* of images.

A simple model for images is to assume that they are random noise fields whose expected magnitude at each frequency is given by this *power spectrum* $P_s(\omega_x, \omega_y)$, i.e.,

$$\langle [S(\omega_x, \omega_y)]^2 \rangle = P_s(\omega_x, \omega_y), \quad (3.65)$$

where the angle brackets $\langle \cdot \rangle$ denote the expected (mean) value of a random variable.⁹ To generate such an image, we simply create a random Gaussian noise image $S(\omega_x, \omega_y)$ where each “pixel” is a zero-mean Gaussian¹⁰ of variance $P_s(\omega_x, \omega_y)$ and then take its inverse FFT.

The observation that signal spectra capture a first-order description of spatial statistics is widely used in signal and image processing. In particular, assuming that an image is a sample from a correlated Gaussian random noise field combined with a statistical model of the measurement process yields an optimum restoration filter known as the *Wiener filter*.¹¹

To derive the Wiener filter, we analyze each frequency component of a signal’s Fourier transform independently. The noisy image formation process can be written as

$$o(x, y) = s(x, y) + n(x, y), \quad (3.66)$$

⁹ The notation $E[\cdot]$ is also commonly used.

¹⁰ We set the DC (i.e., constant) component at $S(0, 0)$ to the mean grey level. See Algorithm C.1 in Appendix C.2 for code to generate Gaussian noise.

¹¹ Wiener is pronounced “veener” since, in German, the “w” is pronounced “v”. Remember that next time you order “Wiener schnitzel”.

where $s(x, y)$ is the (unknown) image we are trying to recover, $n(x, y)$ is the additive noise signal, and $o(x, y)$ is the *observed* noisy image. Because of the linearity of the Fourier transform, we can write

$$O(\omega_x, \omega_y) = S(\omega_x, \omega_y) + N(\omega_x, \omega_y), \quad (3.67)$$

where each quantity in the above equation is the Fourier transform of the corresponding image.

At each frequency (ω_x, ω_y) , we know from our image spectrum that the unknown transform component $S(\omega_x, \omega_y)$ has a *prior* distribution which is a zero-mean Gaussian with variance $P_s(\omega_x, \omega_y)$. We also have noisy measurement $O(\omega_x, \omega_y)$ whose variance is $P_n(\omega_x, \omega_y)$, i.e., the power spectrum of the noise, which is usually assumed to be constant (white), $P_n(\omega_x, \omega_y) = \sigma_n^2$.

According to Bayes' Rule (Appendix B.4), the *posterior estimate* of S can be written as

$$p(S|O) = \frac{p(O|S)p(S)}{p(O)}, \quad (3.68)$$

where $p(O) = \int_S p(O|S)p(S)$ is a normalizing constant used to make the $p(S|O)$ distribution *proper* (integrate to 1). The prior distribution $p(S)$ is given by

$$p(S) = e^{-\frac{(S-\mu)^2}{2P_s}}, \quad (3.69)$$

where μ is the expected mean at that frequency (0 everywhere except at the origin) and the measurement distribution $P(O|S)$ is given by

$$p(O) = e^{-\frac{(S-O)^2}{2P_n}}. \quad (3.70)$$

Taking the negative logarithm of both sides of (3.68) and setting $\mu = 0$ for simplicity, we get

$$-\log p(S|O) = -\log p(O|S) - \log p(S) + C \quad (3.71)$$

$$= 1/2P_n^{-1}(S-O)^2 + 1/2P_s^{-1}S^2 + C, \quad (3.72)$$

which is the *negative posterior log likelihood*. The minimum of this quantity is easy to compute,

$$S_{\text{opt}} = \frac{P_n^{-1}}{P_n^{-1} + P_s^{-1}}O = \frac{P_s}{P_s + P_n}O = \frac{1}{1 + P_n/P_s}O. \quad (3.73)$$

The quantity

$$W(\omega_x, \omega_y) = \frac{1}{1 + \sigma_n^2/P_s(\omega_x, \omega_y)} \quad (3.74)$$

is the Fourier transform of the optimum *Wiener filter* needed to remove the noise from an image whose power spectrum is $P_s(\omega_x, \omega_y)$.

Notice that this filter has the right qualitative properties, i.e., for low frequencies where $P_s \gg \sigma_n^2$, it has unit gain, whereas for high frequencies, it attenuates the noise by a factor P_s/σ_n^2 . Figure 3.25 shows the one-dimensional transform $W(f)$ and the corresponding filter kernel $w(x)$ for the commonly assumed case of $P(f) = f^{-2}$ (Field 1987). Exercise 3.16 has you compare the Wiener filter as a denoising algorithm to hand-tuned Gaussian smoothing.

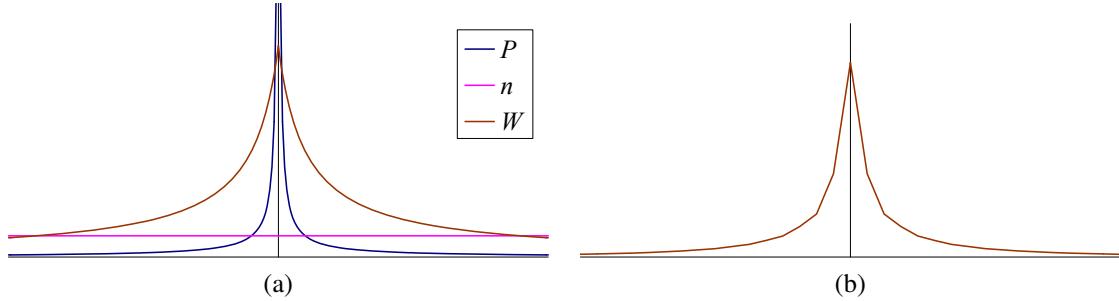


Figure 3.25 One-dimensional Wiener filter: (a) power spectrum of signal $P_s(f)$, noise level σ^2 , and Wiener filter transform $W(f)$; (b) Wiener filter spatial kernel.

The methodology given above for deriving the Wiener filter can easily be extended to the case where the observed image is a noisy blurred version of the original image,

$$o(x, y) = b(x, y) * s(x, y) + n(x, y), \quad (3.75)$$

where $b(x, y)$ is the known blur kernel. Rather than deriving the corresponding Wiener filter, we leave it as an exercise (Exercise 3.17), which also encourages you to compare your de-blurring results with unsharp masking and naïve inverse filtering. More sophisticated algorithms for blur removal are discussed in Sections 3.7 and 10.3.

Discrete cosine transform

The *discrete cosine transform* (DCT) is a variant of the Fourier transform particularly well-suited to compressing images in a block-wise fashion. The one-dimensional DCT is computed by taking the dot product of each N -wide block of pixels with a set of cosines of different frequencies,

$$F(k) = \sum_{i=0}^{N-1} \cos\left(\frac{\pi}{N}(i + \frac{1}{2})k\right) f(i), \quad (3.76)$$

where k is the coefficient (frequency) index, and the $1/2$ -pixel offset is used to make the basis coefficients symmetric (Wallace 1991). Some of the discrete cosine basis functions are shown in Figure 3.26. As you can see, the first basis function (the straight blue line) encodes the average DC value in the block of pixels, while the second encodes a slightly curvy version of the slope.

It turns out that the DCT is a good approximation to the optimal Karhunen–Loëve decomposition of natural image statistics over small patches, which can be obtained by performing a principal component analysis (PCA) of images, as described in Section 14.2.1. The KL-transform de-correlates the signal optimally (assuming the signal is described by its spectrum) and thus, theoretically, leads to optimal compression.

The two-dimensional version of the DCT is defined similarly,

$$F(k, l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \cos\left(\frac{\pi}{N}(i + \frac{1}{2})k\right) \cos\left(\frac{\pi}{N}(j + \frac{1}{2})l\right) f(i, j). \quad (3.77)$$

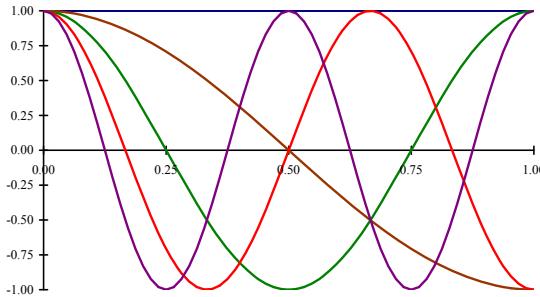


Figure 3.26 Discrete cosine transform (DCT) basis functions: The first DC (i.e., constant) basis is the horizontal blue line, the second is the brown half-cycle waveform, etc. These bases are widely used in image and video compression standards such as JPEG.

Like the 2D Fast Fourier Transform, the 2D DCT can be implemented separably, i.e., first computing the DCT of each line in the block and then computing the DCT of each resulting column. Like the FFT, each of the DCTs can also be computed in $O(N \log N)$ time.

As we mentioned in Section 2.3.3, the DCT is widely used in today's image and video compression algorithms, although it is slowly being supplanted by wavelet algorithms (Simoncelli and Adelson 1990b), as discussed in Section 3.5.4, and overlapped variants of the DCT (Malvar 1990, 1998, 2000), which are used in the new JPEG XR standard.¹² These newer algorithms suffer less from the *blocking artifacts* (visible edge-aligned discontinuities) that result from the pixels in each block (typically 8×8) being transformed and quantized independently. See Exercise 3.30 for ideas on how to remove blocking artifacts from compressed JPEG images.

3.4.4 Application: Sharpening, blur, and noise removal

Another common application of image processing is the enhancement of images through the use of sharpening and noise removal operations, which require some kind of neighborhood processing. Traditionally, these kinds of operation were performed using linear filtering (see Sections 3.2 and Section 3.4.3). Today, it is more common to use non-linear filters (Section 3.3.1), such as the weighted median or bilateral filter (3.34–3.37), anisotropic diffusion (3.39–3.40), or non-local means (Buades, Coll, and Morel 2008). Variational methods (Section 3.7.1), especially those using non-quadratic (robust) norms such as the L_1 norm (which is called *total variation*), are also often used. Figure 3.19 shows some examples of linear and non-linear filters being used to remove noise.

When measuring the effectiveness of image denoising algorithms, it is common to report the results as a *peak signal-to-noise ratio (PSNR)* measurement (2.119), where $I(\mathbf{x})$ is the original (noise-free) image and $\hat{I}(\mathbf{x})$ is the image after denoising; this is for the case where the noisy image has been synthetically generated, so that the clean image is known. A better way to measure the quality is to use a perceptually based similarity metric, such as the structural similarity (SSIM) index (Wang, Bovik, Sheikh *et al.* 2004; Wang, Bovik, and Simoncelli 2005).

¹² <http://www.itu.int/rec/T-REC-T.832-200903-I/en>.

Exercises 3.11, 3.16, 3.17, 3.21, and 3.28 have you implement some of these operations and compare their effectiveness. More sophisticated techniques for blur removal and the related task of super-resolution are discussed in Section 10.3.

3.5 Pyramids and wavelets

So far in this chapter, all of the image transformations we have studied produce output images of the same size as the inputs. Often, however, we may wish to change the resolution of an image before proceeding further. For example, we may need to interpolate a small image to make its resolution match that of the output printer or computer screen. Alternatively, we may want to reduce the size of an image to speed up the execution of an algorithm or to save on storage space or transmission time.

Sometimes, we do not even know what the appropriate resolution for the image should be. Consider, for example, the task of finding a face in an image (Section 14.1.1). Since we do not know the scale at which the face will appear, we need to generate a whole *pyramid* of differently sized images and scan each one for possible faces. (Biological visual systems also operate on a hierarchy of scales (Marr 1982).) Such a pyramid can also be very helpful in accelerating the search for an object by first finding a smaller instance of that object at a coarser level of the pyramid and then looking for the full resolution object only in the vicinity of coarse-level detections (Section 8.1.1). Finally, image pyramids are extremely useful for performing multi-scale editing operations such as blending images while maintaining details.

In this section, we first discuss good filters for changing image resolution, i.e., upsampling (*interpolation*, Section 3.5.1) and downsampling (*decimation*, Section 3.5.2). We then present the concept of multi-resolution pyramids, which can be used to create a complete hierarchy of differently sized images and to enable a variety of applications (Section 3.5.3). A closely related concept is that of *wavelets*, which are a special kind of pyramid with higher frequency selectivity and other useful properties (Section 3.5.4). Finally, we present a useful application of pyramids, namely the blending of different images in a way that hides the seams between the image boundaries (Section 3.5.5).

3.5.1 Interpolation

In order to *interpolate* (or *upsample*) an image to a higher resolution, we need to select some interpolation kernel with which to convolve the image,

$$g(i, j) = \sum_{k,l} f(k, l)h(i - rk, j - rl). \quad (3.78)$$

This formula is related to the discrete convolution formula (3.14), except that we replace k and l in $h()$ with rk and rl , where r is the upsampling rate. Figure 3.27a shows how to think of this process as the superposition of sample weighted interpolation kernels, one centered at each input sample k . An alternative mental model is shown in Figure 3.27b, where the kernel is centered at the output pixel value i (the two forms are equivalent). The latter form is sometimes called the *polyphase filter* form, since the kernel values $h(i)$ can be stored as r separate kernels, each of which is selected for convolution with the input samples depending on the *phase* of i relative to the upsampled grid.

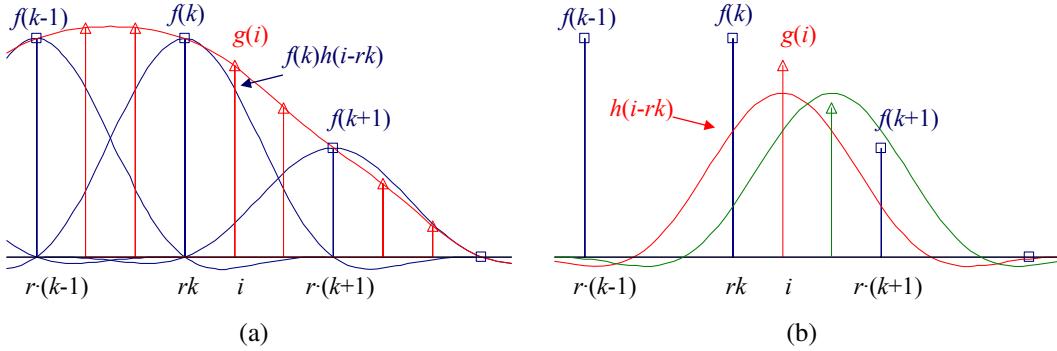


Figure 3.27 Signal interpolation, $g(i) = \sum_k f(k)h(i - rk)$: (a) weighted summation of input values; (b) polyphase filter interpretation.

What kinds of kernel make good interpolators? The answer depends on the application and the computation time involved. Any of the smoothing kernels shown in Tables 3.2 and 3.3 can be used after appropriate re-scaling.¹³ The *linear* interpolator (corresponding to the tent kernel) produces interpolating piecewise linear curves, which result in unappealing *creases* when applied to images (Figure 3.28a). The cubic B-spline, whose discrete $1/2$ -pixel sampling appears as the *binomial kernel* in Table 3.3, is an *approximating* kernel (the interpolated image does not pass through the input data points) that produces soft images with reduced high-frequency detail. The equation for the cubic B-spline is easiest to derive by convolving the tent function (linear B-spline) with itself.

While most graphics cards use the bilinear kernel (optionally combined with a MIP-map—see Section 3.5.3), most photo editing packages use *bicubic* interpolation. The cubic interpolant is a C^1 (derivative-continuous) piecewise-cubic *spline* (the term “spline” is synonymous with “piecewise-polynomial”)¹⁴ whose equation is

$$h(x) = \begin{cases} 1 - (a + 3)x^2 + (a + 2)|x|^3 & \text{if } |x| < 1 \\ a(|x| - 1)(|x| - 2)^2 & \text{if } 1 \leq |x| < 2 \\ 0 & \text{otherwise,} \end{cases} \quad (3.79)$$

where a specifies the derivative at $x = 1$ (Parker, Kenyon, and Troxel 1983). The value of a is often set to -1 , since this best matches the frequency characteristics of a sinc function (Figure 3.29). It also introduces a small amount of sharpening, which can be visually appealing. Unfortunately, this choice does not linearly interpolate straight lines (intensity ramps), so some visible ringing may occur. A better choice for large amounts of interpolation is probably $a = -0.5$, which produces a *quadratic reproducing* spline; it interpolates linear and quadratic functions exactly (Wolberg 1990, Section 5.4.3). Figure 3.29 shows the $a = -1$ and $a = -0.5$ cubic interpolating kernel along with their Fourier transforms; Figure 3.28b and c shows them being applied to two-dimensional interpolation.

Splines have long been used for function and data value interpolation because of the abil-

¹³ The smoothing kernels in Table 3.3 have a unit area. To turn them into interpolating kernels, we simply scale them up by the interpolation rate r .

¹⁴ The term “spline” comes from the draughtsman’s workshop, where it was the name of a flexible piece of wood or metal used to draw smooth curves.

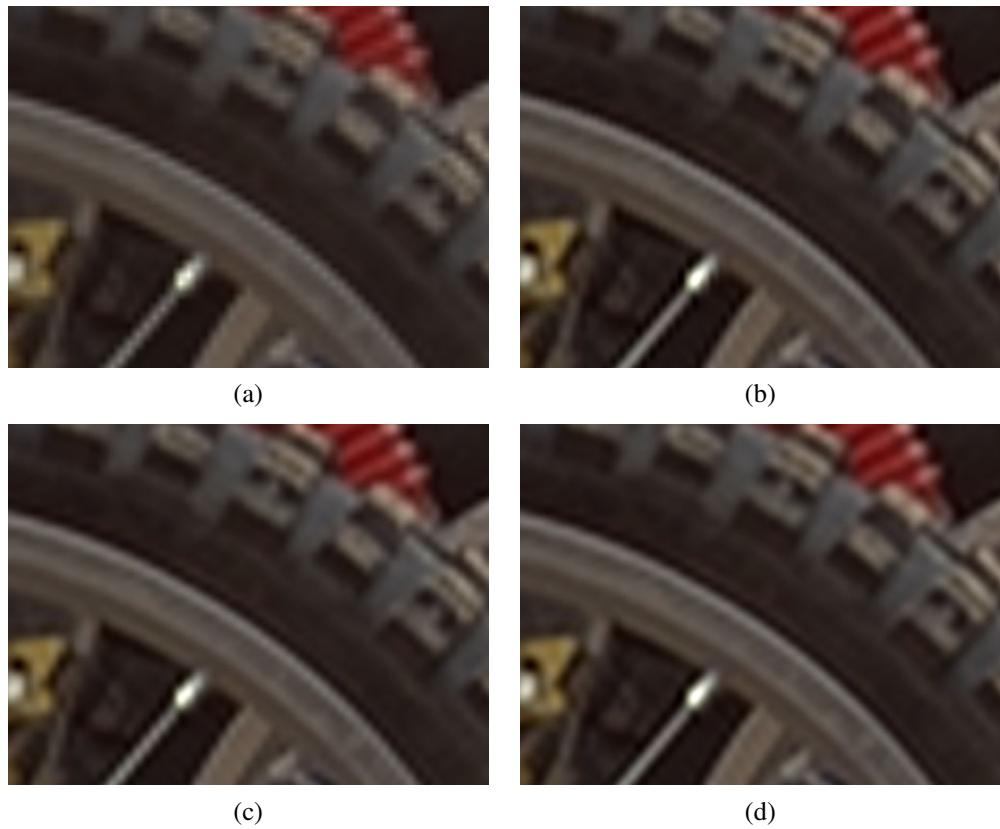


Figure 3.28 Two-dimensional image interpolation: (a) bilinear; (b) bicubic ($a = -1$); (c) bicubic ($a = -0.5$); (d) windowed sinc (nine taps).

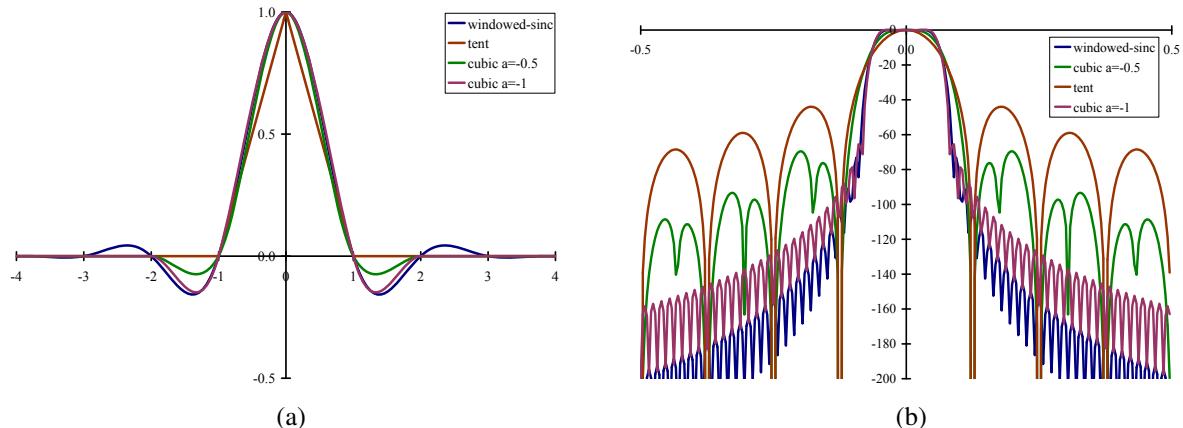


Figure 3.29 (a) Some windowed sinc functions and (b) their log Fourier transforms: raised-cosine windowed sinc in blue, cubic interpolators ($a = -1$ and $a = -0.5$) in green and purple, and tent function in brown. They are often used to perform high-accuracy low-pass filtering operations.

ity to precisely specify derivatives at control points and efficient *incremental* algorithms for their evaluation (Bartels, Beatty, and Barsky 1987; Farin 1992, 1996). Splines are widely used in geometric modeling and computer-aided design (CAD) applications, although they have started being displaced by subdivision surfaces (Zorin, Schröder, and Sweldens 1996; Peters and Reif 2008). In computer vision, splines are often used for elastic image deformations (Section 3.6.2), motion estimation (Section 8.3), and surface interpolation (Section 12.3). In fact, it is possible to carry out most image processing operations by representing images as splines and manipulating them in a multi-resolution framework (Unser 1999).

The highest quality interpolator is generally believed to be the windowed sinc function because it both preserves details in the lower resolution image and avoids aliasing. (It is also possible to construct a C^1 piecewise-cubic approximation to the windowed sinc by matching its derivatives at zero crossing (Szeliski and Ito 1986).) However, some people object to the excessive *ringing* that can be introduced by the windowed sinc and to the repetitive nature of the ringing frequencies (see Figure 3.28d). For this reason, some photographers prefer to repeatedly interpolate images by a small fractional amount (this tends to de-correlate the original pixel grid with the final image). Additional possibilities include using the bilateral filter as an interpolator (Kopf, Cohen, Lischinski *et al.* 2007), using global optimization (Section 3.6) or hallucinating details (Section 10.3).

3.5.2 Decimation

While interpolation can be used to increase the resolution of an image, decimation (downsampling) is required to reduce the resolution.¹⁵ To perform decimation, we first (conceptually) convolve the image with a low-pass filter (to avoid aliasing) and then keep every r th sample. In practice, we usually only evaluate the convolution at every r th sample,

$$g(i, j) = \sum_{k,l} f(k, l)h(ri - k, rj - l), \quad (3.80)$$

as shown in Figure 3.30. Note that the smoothing kernel $h(k, l)$, in this case, is often a stretched and re-scaled version of an interpolation kernel. Alternatively, we can write

$$g(i, j) = \frac{1}{r} \sum_{k,l} f(k, l)h(i - k/r, j - l/r) \quad (3.81)$$

and keep the same kernel $h(k, l)$ for both interpolation and decimation.

One commonly used ($r = 2$) decimation filter is the *binomial* filter introduced by Burt and Adelson (1983a). As shown in Table 3.3, this kernel does a decent job of separating the high and low frequencies, but still leaves a fair amount of high-frequency detail, which can lead to aliasing after downsampling. However, for applications such as image blending (discussed later in this section), this aliasing is of little concern.

If, however, the downsampled images will be displayed directly to the user or, perhaps, blended with other resolutions (as in MIP-mapping, Section 3.5.3), a higher-quality filter is

¹⁵ The term “decimation” has a gruesome etymology relating to the practice of killing every tenth soldier in a Roman unit guilty of cowardice. It is generally used in signal processing to mean any downsampling or rate reduction operation.

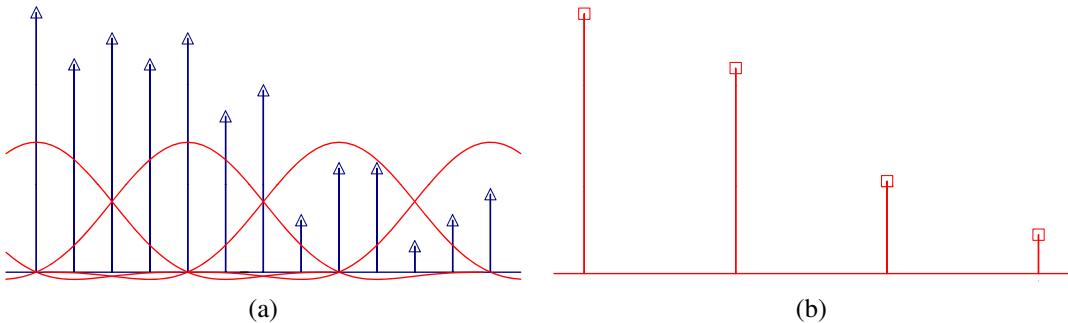


Figure 3.30 Signal decimation: (a) the original samples are (b) convolved with a low-pass filter before being downsampled.

desired. For high downsampling rates, the windowed sinc pre-filter is a good choice (Figure 3.29). However, for small downsampling rates, e.g., $r = 2$, more careful filter design is required.

Table 3.4 shows a number of commonly used $r = 2$ downsampling filters, while Figure 3.31 shows their corresponding frequency responses. These filters include:

- the linear $[1, 2, 1]$ filter gives a relatively poor response;
- the binomial $[1, 4, 6, 4, 1]$ filter cuts off a lot of frequencies but is useful for computer vision analysis pyramids;
- the cubic filters from (3.79); the $a = -1$ filter has a sharper fall-off than the $a = -0.5$ filter (Figure 3.31);
- a cosine-windowed sinc function (Table 3.2);
- the QMF-9 filter of Simoncelli and Adelson (1990b) is used for wavelet denoising and aliases a fair amount (note that the original filter coefficients are normalized to $\sqrt{2}$ gain so they can be “self-inverting”);
- the 9/7 analysis filter from JPEG 2000 (Taubman and Marcellin 2002).

Please see the original papers for the full-precision values of some of these coefficients.

$ n $	Linear	Binomial	Cubic $a = -1$	Cubic $a = -0.5$	Windowed sinc	QMF-9	JPEG 2000
0	0.50	0.3750	0.5000	0.50000	0.4939	0.5638	0.6029
1	0.25	0.2500	0.3125	0.28125	0.2684	0.2932	0.2669
2		0.0625	0.0000	0.00000	0.0000	-0.0519	-0.0782
3			-0.0625	-0.03125	-0.0153	-0.0431	-0.0169
4					0.0000	0.0198	0.0267

Table 3.4 Filter coefficients for $2 \times$ decimation. These filters are of odd length, are symmetric, and are normalized to have unit DC gain (sum up to 1). See Figure 3.31 for their associated frequency responses.

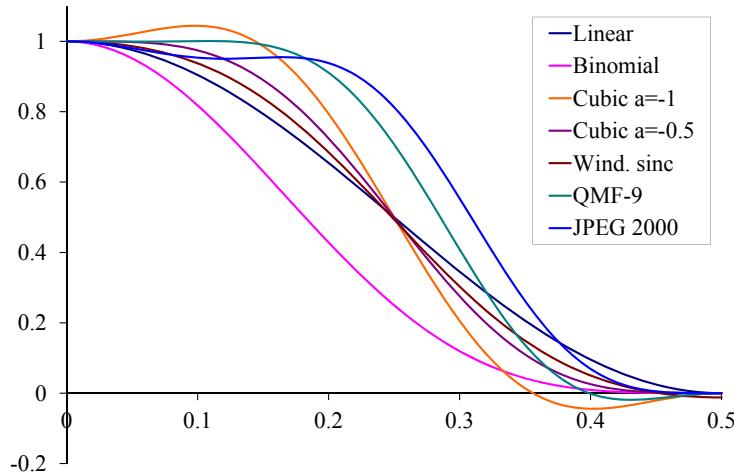


Figure 3.31 Frequency response for some $2 \times$ decimation filters. The cubic $a = -1$ filter has the sharpest fall-off but also a bit of ringing; the wavelet analysis filters (QMF-9 and JPEG 2000), while useful for compression, have more aliasing.

3.5.3 Multi-resolution representations

Now that we have described interpolation and decimation algorithms, we can build a complete image pyramid (Figure 3.32). As we mentioned before, pyramids can be used to accelerate coarse-to-fine search algorithms, to look for objects or patterns at different scales, and to perform multi-resolution blending operations. They are also widely used in computer graphics hardware and software to perform fractional-level decimation using the MIP-map, which we cover in Section 3.6.

The best known (and probably most widely used) pyramid in computer vision is Burt and Adelson’s (1983a) Laplacian pyramid. To construct the pyramid, we first blur and subsample the original image by a factor of two and store this in the next level of the pyramid (Figure 3.33). Because adjacent levels in the pyramid are related by a sampling rate $r = 2$, this kind of pyramid is known as an *octave pyramid*. Burt and Adelson originally proposed a five-tap kernel of the form

$$\boxed{c \ b \ a \ b \ c}, \quad (3.82)$$

with $b = 1/4$ and $c = 1/4 - a/2$. In practice, $a = 3/8$, which results in the familiar binomial kernel,

$$\frac{1}{16} \boxed{1 \ 4 \ 6 \ 4 \ 1}, \quad (3.83)$$

which is particularly easy to implement using shifts and adds. (This was important in the days when multipliers were expensive.) The reason they call their resulting pyramid a *Gaussian* pyramid is that repeated convolutions of the binomial kernel converge to a Gaussian.¹⁶

To compute the *Laplacian* pyramid, Burt and Adelson first interpolate a lower resolution image to obtain a *reconstructed* low-pass version of the original image (Figure 3.34b). They then subtract this low-pass version from the original to yield the band-pass “Laplacian”

¹⁶ Then again, this is true for any smoothing kernel (Wells 1986).

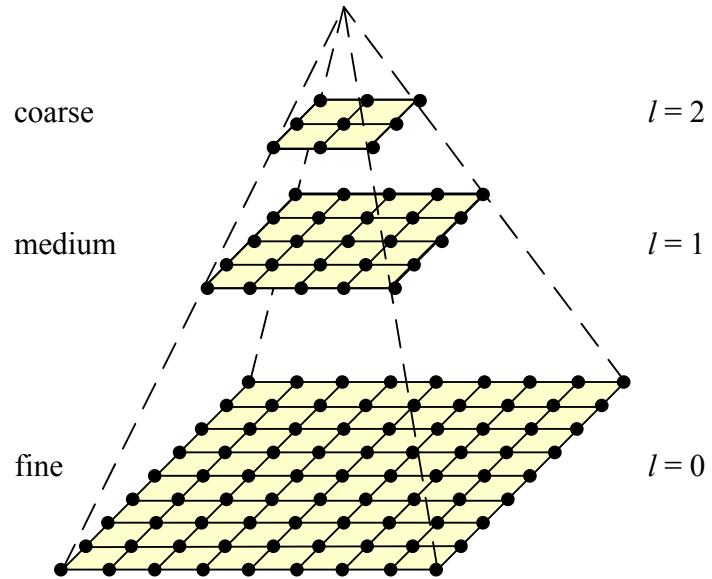


Figure 3.32 A traditional image pyramid: each level has half the resolution (width and height), and hence a quarter of the pixels, of its parent level.

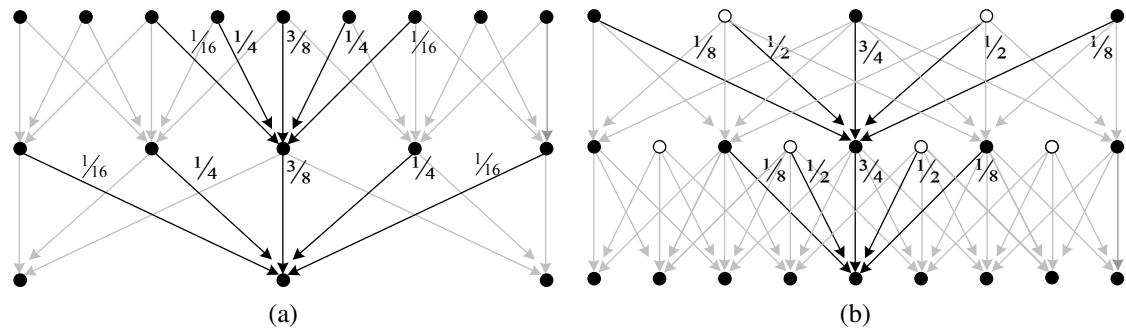


Figure 3.33 The Gaussian pyramid shown as a signal processing diagram: The (a) analysis and (b) re-synthesis stages are shown as using similar computations. The white circles indicate zero values inserted by the $\uparrow 2$ upsampling operation. Notice how the reconstruction filter coefficients are twice the analysis coefficients. The computation is shown as flowing down the page, regardless of whether we are going from coarse to fine or *vice versa*.

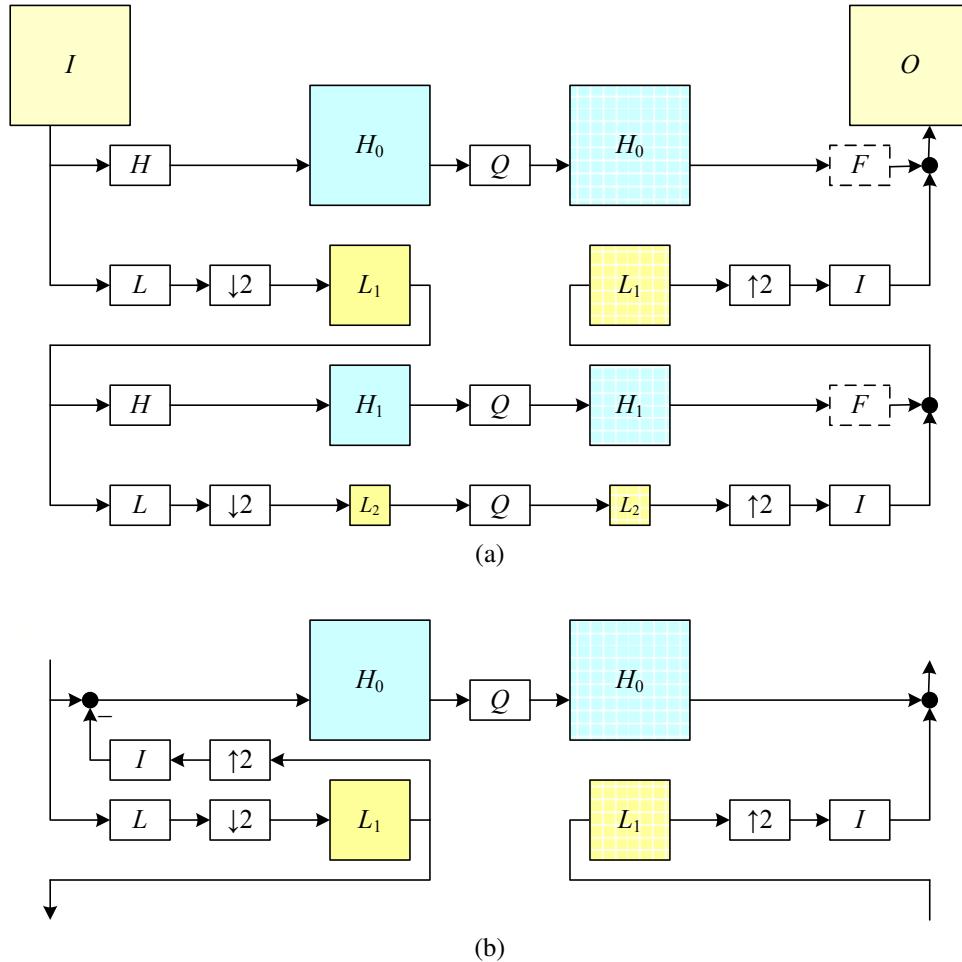


Figure 3.34 The Laplacian pyramid: (a) The conceptual flow of images through processing stages: images are high-pass and low-pass filtered, and the low-pass filtered images are processed in the next stage of the pyramid. During reconstruction, the interpolated image and the (optionally filtered) high-pass image are added back together. The Q box indicates quantization or some other pyramid processing, e.g., noise removal by *coring* (setting small wavelet values to 0). (b) The actual computation of the high-pass filter involves first interpolating the down-sampled low-pass image and then subtracting it. This results in perfect reconstruction when Q is the identity. The high-pass (or band-pass) images are typically called *Laplacian* images, while the low-pass images are called *Gaussian* images.

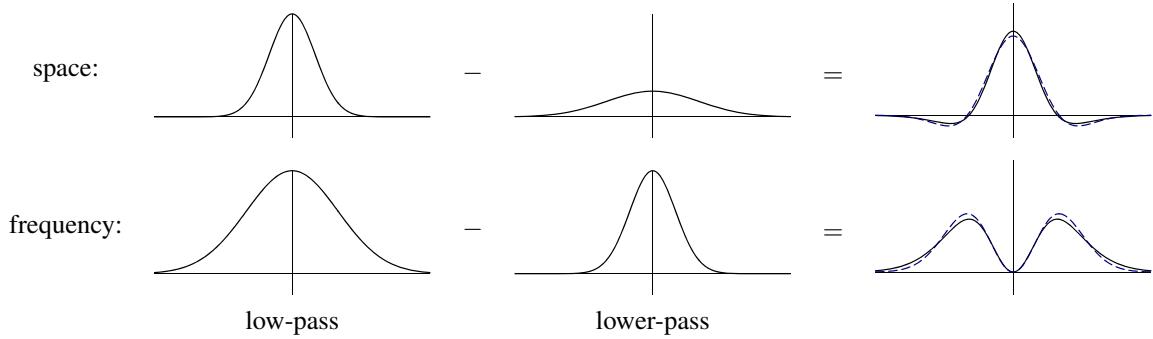


Figure 3.35 The difference of two low-pass filters results in a band-pass filter. The dashed blue lines show the close fit to a half-octave Laplacian of Gaussian.

image, which can be stored away for further processing. The resulting pyramid has *perfect reconstruction*, i.e., the Laplacian images plus the base-level Gaussian (L_2 in Figure 3.34b) are sufficient to exactly reconstruct the original image. Figure 3.33 shows the same computation in one dimension as a signal processing diagram, which completely captures the computations being performed during the analysis and re-synthesis stages.

Burt and Adelson also describe a variant on the Laplacian pyramid, where the low-pass image is taken from the original blurred image rather than the reconstructed pyramid (piping the output of the L box directly to the subtraction in Figure 3.34b). This variant has less aliasing, since it avoids one downsampling and upsampling round-trip, but it is not self-inverting, since the Laplacian images are no longer adequate to reproduce the original image.

As with the Gaussian pyramid, the term Laplacian is a bit of a misnomer, since their band-pass images are really differences of (approximate) Gaussians, or DoGs,

$$\text{DoG}\{I; \sigma_1, \sigma_2\} = G_{\sigma_1} * I - G_{\sigma_2} * I = (G_{\sigma_1} - G_{\sigma_2}) * I. \quad (3.84)$$

A Laplacian of Gaussian (which we saw in (3.26)) is actually its second derivative,

$$\text{LoG}\{I; \sigma\} = \nabla^2(G_{\sigma} * I) = (\nabla^2 G_{\sigma}) * I, \quad (3.85)$$

where

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \quad (3.86)$$

is the Laplacian (operator) of a function. Figure 3.35 shows how the Differences of Gaussian and Laplacians of Gaussian look in both space and frequency.

Laplacians of Gaussian have elegant mathematical properties, which have been widely studied in the *scale-space* community (Witkin 1983; Witkin, Terzopoulos, and Kass 1986; Lindeberg 1990; Nielsen, Florack, and Deriche 1997) and can be used for a variety of applications including edge detection (Marr and Hildreth 1980; Perona and Malik 1990b), stereo matching (Witkin, Terzopoulos, and Kass 1987), and image enhancement (Nielsen, Florack, and Deriche 1997).

A less widely used variant is *half-octave pyramids*, shown in Figure 3.36a. These were first introduced to the vision community by Crowley and Stern (1984), who call them *Difference of Low-Pass* (DOLP) transforms. Because of the small scale change between adja-

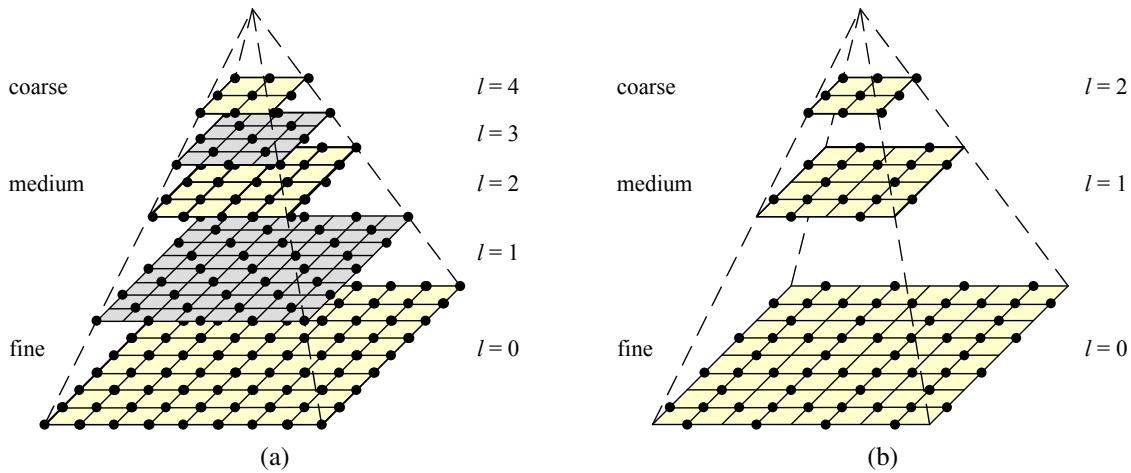


Figure 3.36 Multiresolution pyramids: (a) pyramid with half-octave (*quincunx*) sampling (odd levels are colored gray for clarity). (b) wavelet pyramid—each wavelet level stores $3/4$ of the original pixels (usually the horizontal, vertical, and mixed gradients), so that the total number of wavelet coefficients and original pixels is the same.

cent levels, the authors claim that coarse-to-fine algorithms perform better. In the image-processing community, half-octave pyramids combined with checkerboard sampling grids are known as *quincunx* sampling (Feilner, Van De Ville, and Unser 2005). In detecting multi-scale features (Section 4.1.1), it is often common to use half-octave or even quarter-octave pyramids (Lowe 2004; Triggs 2004). However, in this case, the subsampling only occurs at every octave level, i.e., the image is repeatedly blurred with wider Gaussians until a full octave of resolution change has been achieved (Figure 4.11).

3.5.4 Wavelets

While pyramids are used extensively in computer vision applications, some people use *wavelet* decompositions as an alternative. Wavelets are filters that localize a signal in both space and frequency (like the Gabor filter in Table 3.2) and are defined over a hierarchy of scales. Wavelets provide a smooth way to decompose a signal into frequency components without blocking and are closely related to pyramids.

Wavelets were originally developed in the applied math and signal processing communities and were introduced to the computer vision community by Mallat (1989). Strang (1989); Simoncelli and Adelson (1990b); Rioul and Vetterli (1991); Chui (1992); Meyer (1993) all provide nice introductions to the subject along with historical reviews, while Chui (1992) provides a more comprehensive review and survey of applications. Sweldens (1997) describes the more recent *lifting* approach to wavelets that we discuss shortly.

Wavelets are widely used in the computer graphics community to perform multi-resolution geometric processing (Stollnitz, DeRose, and Salesin 1996) and have also been used in computer vision for similar applications (Szeliski 1990b; Pentland 1994; Gortler and Cohen 1995; Yaou and Chang 1994; Lai and Vemuri 1997; Szeliski 2006b), as well as for multi-scale oriented filtering (Simoncelli, Freeman, Adelson *et al.* 1992) and denoising (Portilla, Strela, Wainwright *et al.* 2003).

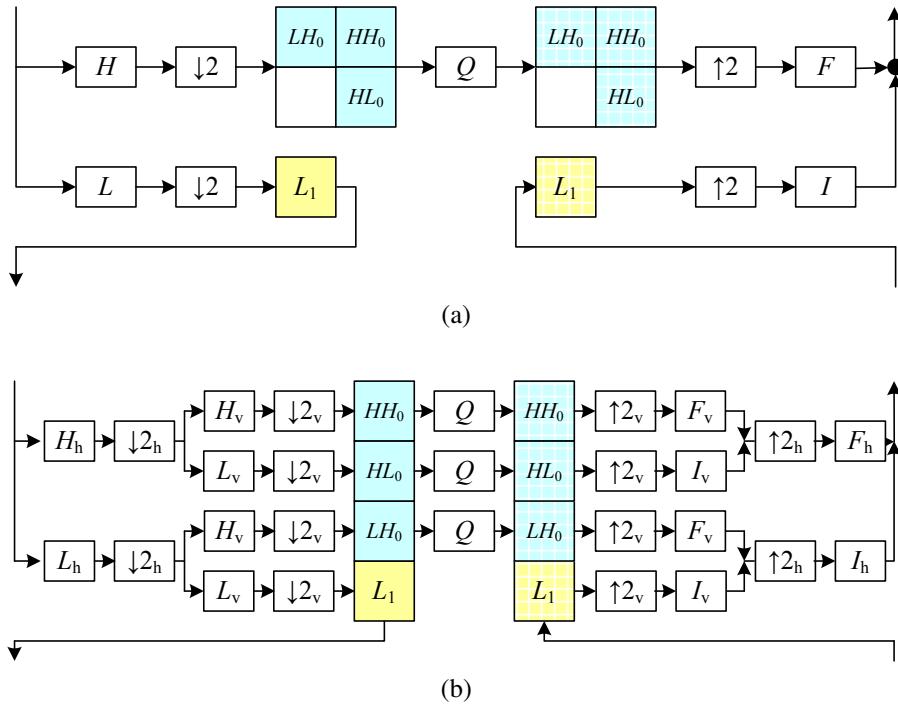


Figure 3.37 Two-dimensional wavelet decomposition: (a) high-level diagram showing the low-pass and high-pass transforms as single boxes; (b) separable implementation, which involves first performing the wavelet transform horizontally and then vertically. The I and F boxes are the interpolation and filtering boxes required to re-synthesize the image from its wavelet components.

Since both image pyramids and wavelets decompose an image into multi-resolution descriptions that are localized in both space and frequency, how do they differ? The usual answer is that traditional pyramids are *overcomplete*, i.e., they use more pixels than the original image to represent the decomposition, whereas wavelets provide a *tight frame*, i.e., they keep the size of the decomposition the same as the image (Figure 3.36b). However, some wavelet families *are*, in fact, overcomplete in order to provide better shiftability or steering in orientation (Simoncelli, Freeman, Adelson *et al.* 1992). A better distinction, therefore, might be that wavelets are more orientation selective than regular band-pass pyramids.

How are two-dimensional wavelets constructed? Figure 3.37a shows a high-level diagram of one stage of the (recursive) coarse-to-fine construction (analysis) pipeline alongside the complementary re-construction (synthesis) stage. In this diagram, the high-pass filter followed by decimation keeps $3/4$ of the original pixels, while $1/4$ of the low-frequency coefficients are passed on to the next stage for further analysis. In practice, the filtering is usually broken down into two separable sub-stages, as shown in Figure 3.37b. The resulting three wavelet images are sometimes called the high–high (HH), high–low (HL), and low–high (LH) images. The high–low and low–high images accentuate the horizontal and vertical edges and gradients, while the high–high image contains the less frequently occurring mixed derivatives.

How are the high-pass H and low-pass L filters shown in Figure 3.37b chosen and how

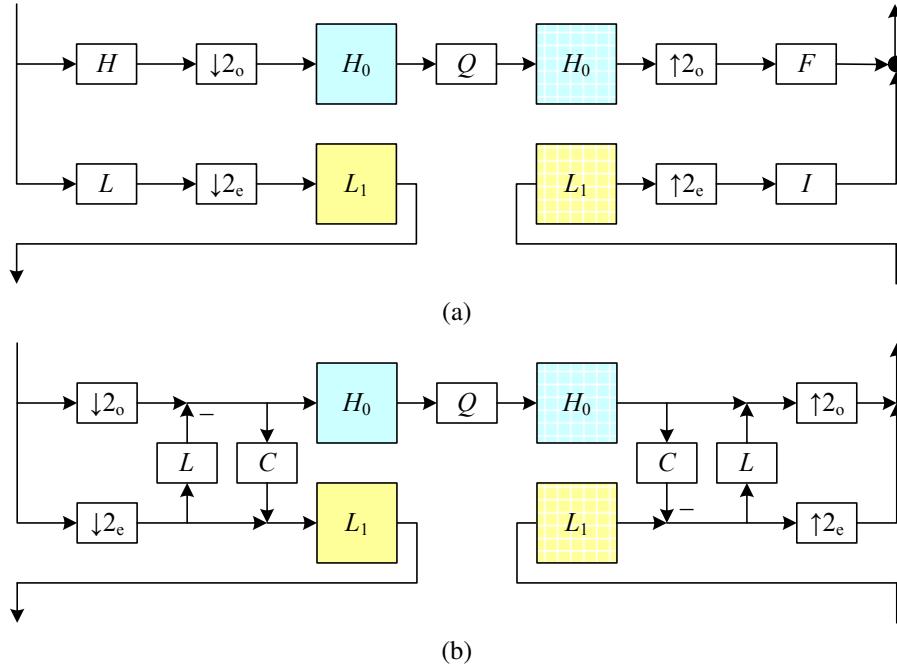


Figure 3.38 One-dimensional wavelet transform: (a) usual high-pass + low-pass filters followed by odd ($\downarrow 2_o$) and even ($\downarrow 2_e$) downsampling; (b) lifted version, which first selects the odd and even subsequences and then applies a low-pass prediction stage L and a high-pass correction stage C in an easily reversible manner.

can the corresponding reconstruction filters I and F be computed? Can filters be designed that all have finite impulse responses? This topic has been the main subject of study in the wavelet community for over two decades. The answer depends largely on the intended application, e.g., whether the wavelets are being used for compression, image analysis (feature finding), or denoising. Simoncelli and Adelson (1990b) show (in Table 4.1) some good odd-length quadrature mirror filter (QMF) coefficients that seem to work well in practice.

Since the design of wavelet filters is such a tricky art, is there perhaps a better way? Indeed, a simpler procedure is to split the signal into its even and odd components and then perform trivially reversible filtering operations on each sequence to produce what are called *lifted wavelets* (Figures 3.38 and 3.39). Sweldens (1996) gives a wonderfully understandable introduction to the *lifting scheme* for *second-generation wavelets*, followed by a comprehensive review (Sweldens 1997).

As Figure 3.38 demonstrates, rather than first filtering the whole input sequence (image) with high-pass and low-pass filters and then keeping the odd and even sub-sequences, the lifting scheme first splits the sequence into its even and odd sub-components. Filtering the even sequence with a low-pass filter L and subtracting the result from the even sequence is trivially reversible: simply perform the same filtering and then add the result back in. Furthermore, this operation can be performed in place, resulting in significant space savings. The same applies to filtering the even sequence with the correction filter C , which is used to ensure that the even sequence is low-pass. A series of such *lifting* steps can be used to create more complex filter responses with low computational cost and guaranteed reversibility.

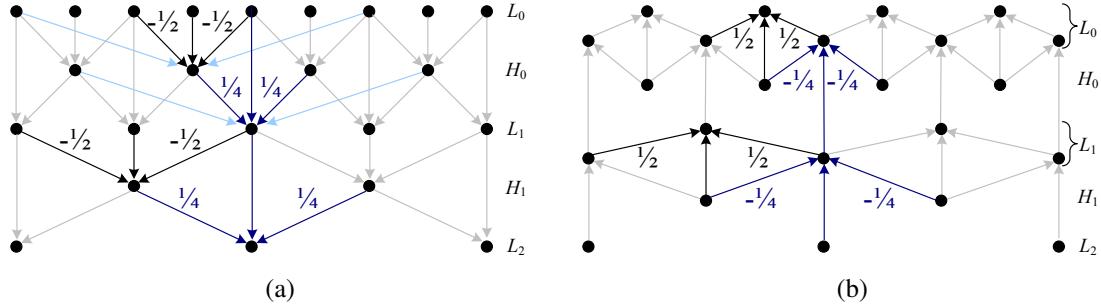


Figure 3.39 Lifted transform shown as a signal processing diagram: (a) The analysis stage first predicts the odd value from its even neighbors, stores the difference wavelet, and then compensates the coarser even value by adding in a fraction of the wavelet. (b) The synthesis stage simply reverses the flow of computation and the signs of some of the filters and operations. The light blue lines show what happens if we use four taps for the prediction and correction instead of just two.

This process can perhaps be more easily understood by considering the signal processing diagram in Figure 3.39. During analysis, the average of the even values is subtracted from the odd value to obtain a high-pass wavelet coefficient. However, the even samples still contain an aliased sample of the low-frequency signal. To compensate for this, a small amount of the high-pass wavelet is added back to the even sequence so that it is properly low-pass filtered. (It is easy to show that the effective low-pass filter is $[-1/8, 1/4, 3/4, 1/4, -1/8]$, which is indeed a low-pass filter.) During synthesis, the same operations are reversed with a judicious change in sign.

Of course, we need not restrict ourselves to two-tap filters. Figure 3.39 shows as light blue arrows additional filter coefficients that could optionally be added to the lifting scheme without affecting its reversibility. In fact, the low-pass and high-pass filtering operations can be interchanged, e.g., we could use a five-tap cubic low-pass filter on the odd sequence (plus center value) first, followed by a four-tap cubic low-pass predictor to estimate the wavelet, although I have not seen this scheme written down.

Lifted wavelets are called *second-generation wavelets* because they can easily adapt to non-regular sampling topologies, e.g., those that arise in computer graphics applications such as multi-resolution surface manipulation (Schröder and Sweldens 1995). It also turns out that lifted *weighted wavelets*, i.e., wavelets whose coefficients adapt to the underlying problem being solved (Fattal 2009), can be extremely effective for low-level image manipulation tasks and also for preconditioning the kinds of sparse linear systems that arise in the optimization-based approaches to vision algorithms that we discuss in Section 3.7 (Szeliski 2006b).

An alternative to the widely used “separable” approach to wavelet construction, which decomposes each level into horizontal, vertical, and “cross” sub-bands, is to use a representation that is more rotationally symmetric and orientationally selective and also avoids the aliasing inherent in sampling signals below their Nyquist frequency.¹⁷ Simoncelli, Freeman, Adelson *et al.* (1992) introduce such a representation, which they call a *pyramidal radial frequency implementation of shiftable multi-scale transforms* or, more succinctly, *steerable pyramids*.

¹⁷ Such aliasing can often be seen as the signal content moving between bands as the original signal is slowly shifted.

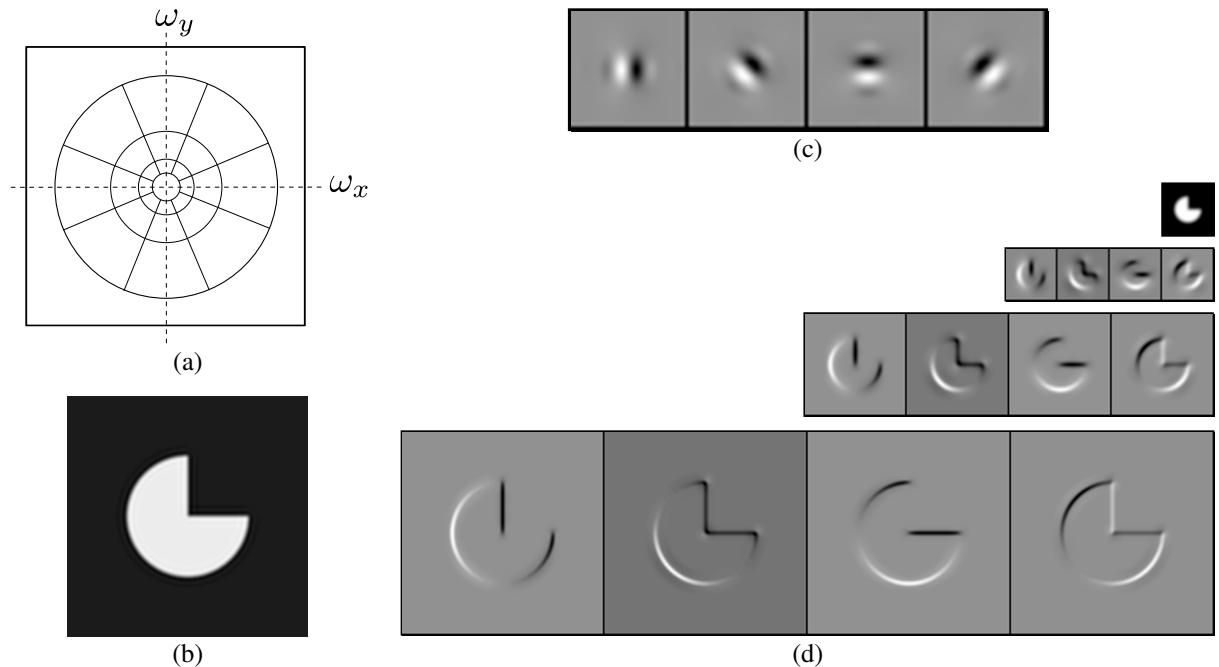


Figure 3.40 Steerable shiftable multiscale transforms (Simoncelli, Freeman, Adelson *et al.* 1992) © 1992 IEEE: (a) radial multi-scale frequency domain decomposition; (b) original image; (c) a set of four steerable filters; (d) the radial multi-scale wavelet decomposition.

Their representation is not only overcomplete (which eliminates the aliasing problem) but is also orientationally selective and has identical analysis and synthesis basis functions, i.e., it is *self-inverting*, just like “regular” wavelets. As a result, this makes steerable pyramids a much more useful basis for the structural analysis and matching tasks commonly used in computer vision.

Figure 3.40a shows how such a decomposition looks in frequency space. Instead of recursively dividing the frequency domain into 2×2 squares, which results in checkerboard high frequencies, radial arcs are used instead. Figure 3.40b illustrates the resulting pyramid sub-bands. Even though the representation is *overcomplete*, i.e., there are more wavelet coefficients than input pixels, the additional frequency and orientation selectivity makes this representation preferable for tasks such as texture analysis and synthesis (Portilla and Simoncelli 2000) and image denoising (Portilla, Strela, Wainwright *et al.* 2003; Lyu and Simoncelli 2009).

3.5.5 Application: Image blending

One of the most engaging and fun applications of the Laplacian pyramid presented in Section 3.5.3 is the creation of blended composite images, as shown in Figure 3.41 (Burt and Adelson 1983b). While splicing the apple and orange images together along the midline produces a noticeable cut, *splicing* them together (as Burt and Adelson (1983b) called their procedure) creates a beautiful illusion of a truly hybrid fruit. The key to their approach is that the low-frequency color variations between the red apple and the orange are smoothly

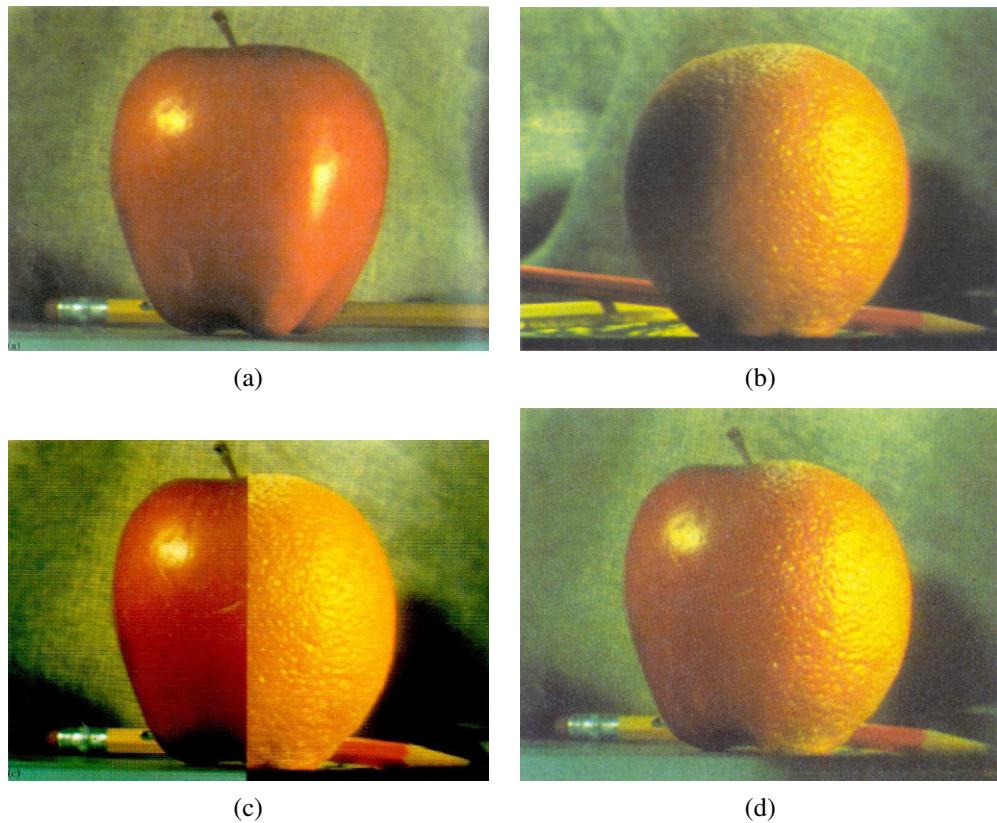


Figure 3.41 Laplacian pyramid blending (Burt and Adelson 1983b) © 1983 ACM: (a) original image of apple, (b) original image of orange, (c) regular splice, (d) pyramid blend.

blended, while the higher-frequency textures on each fruit are blended more quickly to avoid “ghosting” effects when two textures are overlaid.

To create the blended image, each source image is first decomposed into its own Laplacian pyramid (Figure 3.42, left and middle columns). Each band is then multiplied by a smooth weighting function whose extent is proportional to the pyramid level. The simplest and most general way to create these weights is to take a binary mask image (Figure 3.43c) and to construct a *Gaussian* pyramid from this mask. Each Laplacian pyramid image is then multiplied by its corresponding Gaussian mask and the sum of these two weighted pyramids is then used to construct the final image (Figure 3.42, right column).

Figure 3.43 shows that this process can be applied to arbitrary mask images with surprising results. It is also straightforward to extend the pyramid blend to an arbitrary number of images whose pixel provenance is indicated by an integer-valued label image (see Exercise 3.20). This is particularly useful in image stitching and compositing applications, where the exposures may vary between different images, as described in Section 9.3.4.

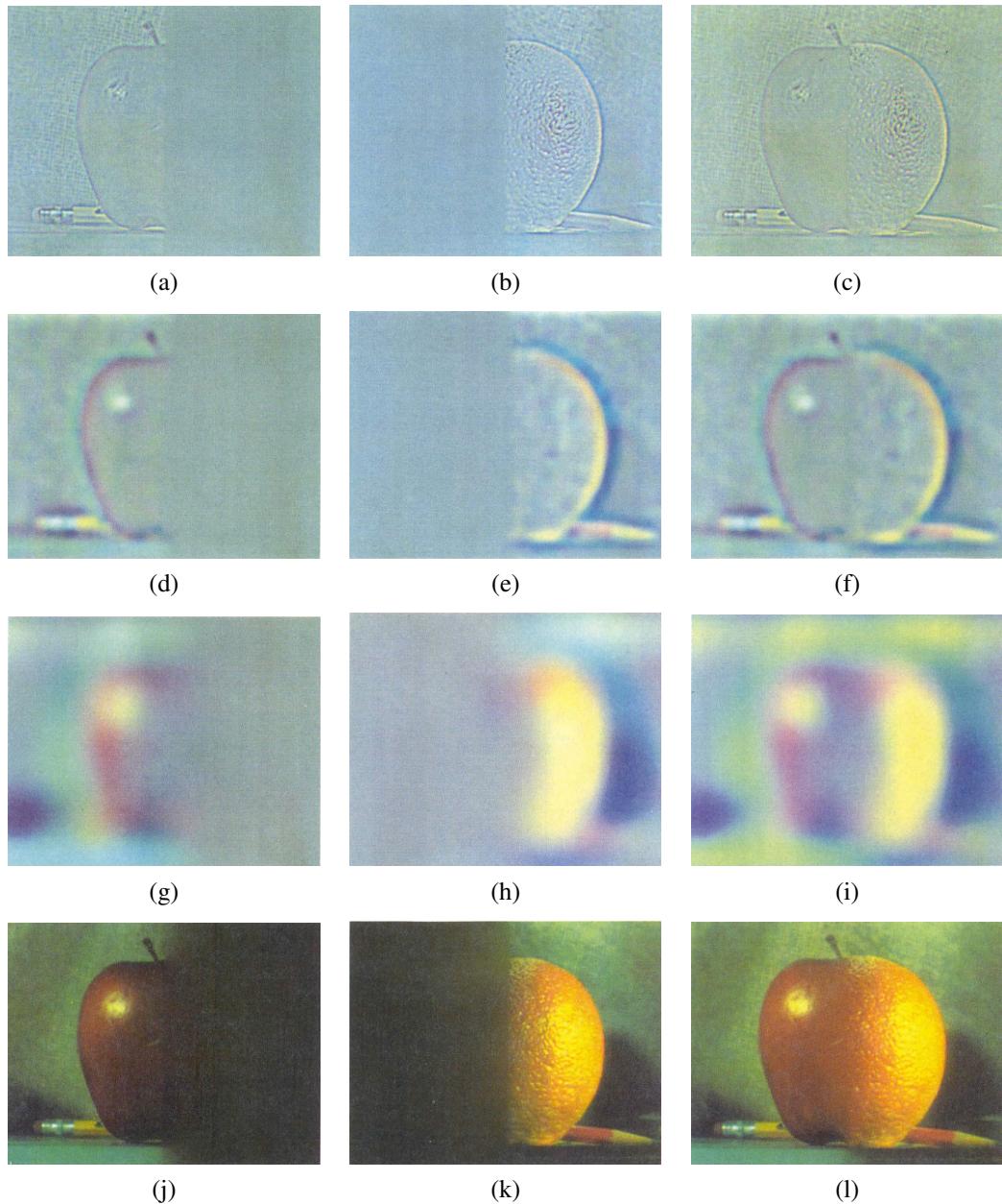


Figure 3.42 Laplacian pyramid blending details (Burt and Adelson 1983b) © 1983 ACM. The first three rows show the high, medium, and low frequency parts of the Laplacian pyramid (taken from levels 0, 2, and 4). The left and middle columns show the original apple and orange images weighted by the smooth interpolation functions, while the right column shows the averaged contributions.

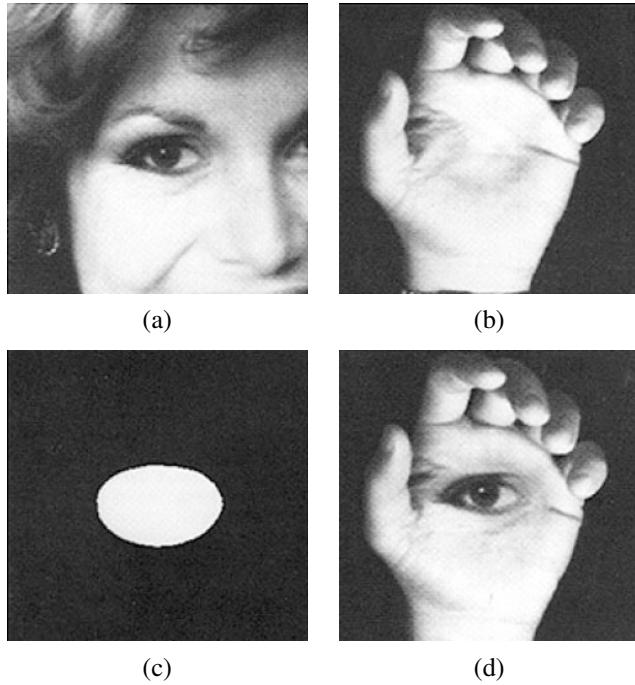


Figure 3.43 Laplacian pyramid blend of two images of arbitrary shape (Burt and Adelson 1983b) © 1983 ACM:
 (a) first input image; (b) second input image; (c) region mask; (d) blended image.

3.6 Geometric transformations

In the previous sections, we saw how interpolation and decimation could be used to change the *resolution* of an image. In this section, we look at how to perform more general transformations, such as image rotations or general warps. In contrast to the point processes we saw in Section 3.1, where the function applied to an image transforms the *range* of the image,

$$g(\mathbf{x}) = h(f(\mathbf{x})), \quad (3.87)$$

here we look at functions that transform the *domain*,

$$g(\mathbf{x}) = f(\mathbf{h}(\mathbf{x})) \quad (3.88)$$

(see Figure 3.44).

We begin by studying the global *parametric* 2D transformation first introduced in Section 2.1.2. (Such a transformation is called parametric because it is controlled by a small number of parameters.) We then turn our attention to more local general deformations such as those defined on meshes (Section 3.6.2). Finally, we show how image warps can be combined with cross-dissolves to create interesting *morphs* (in-between animations) in Section 3.6.3. For readers interested in more details on these topics, there is an excellent survey by Heckbert (1986) as well as very accessible textbooks by Wolberg (1990), Gomes, Darsa, Costa *et al.* (1999) and Akenine-Möller and Haines (2002). Note that Heckbert's survey is on *texture mapping*, which is how the computer graphics community refers to the topic of warping images onto surfaces.

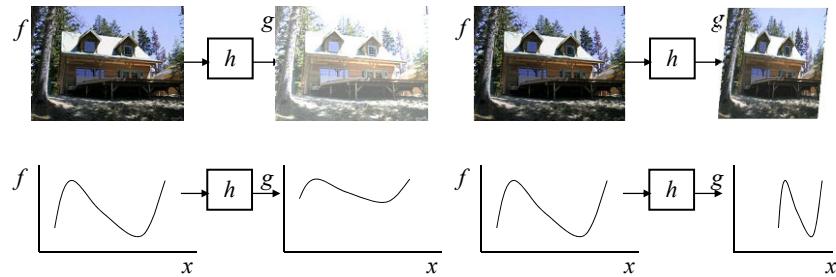


Figure 3.44 Image warping involves modifying the *domain* of an image function rather than its *range*.

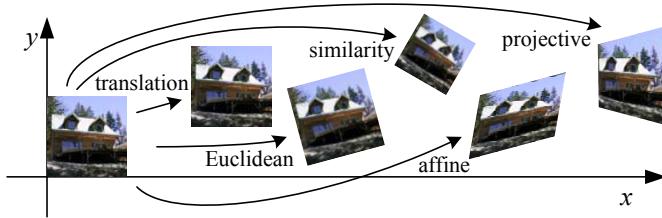


Figure 3.45 Basic set of 2D geometric image transformations.

Transformation	Matrix	# DoF	Preserves	Icon
translation	$[I \mid t]_{2 \times 3}$	2	orientation	
rigid (Euclidean)	$[R \mid t]_{2 \times 3}$	3	lengths	
similarity	$[sR \mid t]_{2 \times 3}$	4	angles	
affine	$[A]_{2 \times 3}$	6	parallelism	
projective	$[\tilde{H}]_{3 \times 3}$	8	straight lines	

Table 3.5 Hierarchy of 2D coordinate transformations. Each transformation also preserves the properties listed in the rows below it, i.e., similarity preserves not only angles but also parallelism and straight lines. The 2×3 matrices are extended with a third $[0^T \ 1]$ row to form a full 3×3 matrix for homogeneous coordinate transformations.

```

procedure forwardWarp( $f, h, \text{out } g$ ):
    For every pixel  $x$  in  $f(x)$ 
        1. Compute the destination location  $x' = h(x)$ .
        2. Copy the pixel  $f(x)$  to  $g(x')$ .

```

Algorithm 3.1 Forward warping algorithm for transforming an image $f(x)$ into an image $g(x')$ through the parametric transform $x' = h(x)$.

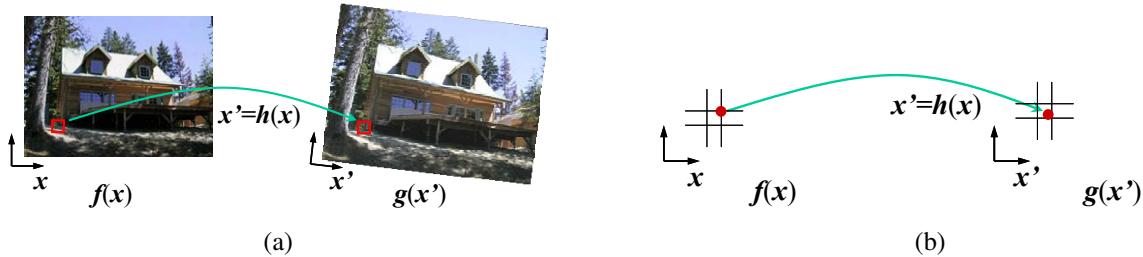


Figure 3.46 Forward warping algorithm: (a) a pixel $f(x)$ is copied to its corresponding location $x' = h(x)$ in image $g(x')$; (b) detail of the source and destination pixel locations.

3.6.1 Parametric transformations

Parametric transformations apply a global deformation to an image, where the behavior of the transformation is controlled by a small number of parameters. Figure 3.45 shows a few examples of such transformations, which are based on the 2D geometric transformations shown in Figure 2.4. The formulas for these transformations were originally given in Table 2.1 and are reproduced here in Table 3.5 for ease of reference.

In general, given a transformation specified by a formula $x' = h(x)$ and a source image $f(x)$, how do we compute the values of the pixels in the new image $g(x)$, as given in (3.88)? Think about this for a minute before proceeding and see if you can figure it out.

If you are like most people, you will come up with an algorithm that looks something like Algorithm 3.1. This process is called *forward warping* or *forward mapping* and is shown in Figure 3.46a. Can you think of any problems with this approach?

In fact, this approach suffers from several limitations. The process of copying a pixel $f(x)$ to a location x' in g is not well defined when x' has a non-integer value. What do we do in such a case? What would you do?

You can round the value of x' to the nearest integer coordinate and copy the pixel there, but the resulting image has severe aliasing and pixels that jump around a lot when animating the transformation. You can also “distribute” the value among its four nearest neighbors in a weighted (bilinear) fashion, keeping track of the per-pixel weights and normalizing at the end. This technique is called *splatting* and is sometimes used for volume rendering in the graphics community (Levoy and Whitted 1985; Levoy 1988; Westover 1989; Rusinkiewicz and Levoy 2000). Unfortunately, it suffers from both moderate amounts of aliasing and a fair amount of blur (loss of high-resolution detail).

```
procedure inverseWarp( $f, h, \text{out } g$ ):
    For every pixel  $x'$  in  $g(x')$ 
        1. Compute the source location  $x = \hat{h}(x')$ 
        2. Resample  $f(x)$  at location  $x$  and copy to  $g(x')$ 
```

Algorithm 3.2 Inverse warping algorithm for creating an image $g(x')$ from an image $f(x)$ using the parametric transform $x' = h(x)$.



Figure 3.47 Inverse warping algorithm: (a) a pixel $g(x')$ is sampled from its corresponding location $x = \hat{h}(x')$ in image $f(x)$; (b) detail of the source and destination pixel locations.

The second major problem with forward warping is the appearance of cracks and holes, especially when magnifying an image. Filling such holes with their nearby neighbors can lead to further aliasing and blurring.

What can we do instead? A preferable solution is to use *inverse warping* (Algorithm 3.2), where each pixel in the destination image $g(x')$ is sampled from the original image $f(x)$ (Figure 3.47).

How does this differ from the forward warping algorithm? For one thing, since $\hat{h}(x')$ is (presumably) defined for all pixels in $g(x')$, we no longer have holes. More importantly, resampling an image at non-integer locations is a well-studied problem (general image interpolation, see Section 3.5.2) and high-quality filters that control aliasing can be used.

Where does the function $\hat{h}(x')$ come from? Quite often, it can simply be computed as the inverse of $h(x)$. In fact, all of the parametric transforms listed in Table 3.5 have closed form solutions for the inverse transform: simply take the inverse of the 3×3 matrix specifying the transform.

In other cases, it is preferable to formulate the problem of image warping as that of resampling a source image $f(x)$ given a mapping $x = \hat{h}(x')$ from destination pixels x' to source pixels x . For example, in optical flow (Section 8.4), we estimate the flow field as the location of the *source* pixel which produced the current pixel whose flow is being estimated, as opposed to computing the *destination* pixel to which it is going. Similarly, when correcting for radial distortion (Section 2.1.6), we calibrate the lens by computing for each pixel in the final (undistorted) image the corresponding pixel location in the original (distorted) image.

What kinds of interpolation filter are suitable for the resampling process? Any of the filters we studied in Section 3.5.2 can be used, including nearest neighbor, bilinear, bicubic, and

windowed sinc functions. While bilinear is often used for speed (e.g., inside the inner loop of a patch-tracking algorithm, see Section 8.1.3), bicubic, and windowed sinc are preferable where visual quality is important.

To compute the value of $f(\mathbf{x})$ at a non-integer location \mathbf{x} , we simply apply our usual FIR resampling filter,

$$g(x, y) = \sum_{k,l} f(k, l)h(x - k, y - l), \quad (3.89)$$

where (x, y) are the sub-pixel coordinate values and $h(x, y)$ is some interpolating or smoothing kernel. Recall from Section 3.5.2 that when decimation is being performed, the smoothing kernel is stretched and re-scaled according to the downsampling rate r .

Unfortunately, for a general (non-zoom) image transformation, the resampling rate r is not well defined. Consider a transformation that stretches the x dimensions while squashing the y dimensions. The resampling kernel should be performing regular interpolation along the x dimension and smoothing (to anti-alias the blurred image) in the y direction. This gets even more complicated for the case of general affine or perspective transforms.

What can we do? Fortunately, Fourier analysis can help. The two-dimensional generalization of the one-dimensional *domain scaling* law given in Table 3.1 is

$$g(\mathbf{Ax}) \Leftrightarrow |\mathbf{A}|^{-1}G(\mathbf{A}^{-T}\mathbf{f}). \quad (3.90)$$

For all of the transforms in Table 3.5 except perspective, the matrix \mathbf{A} is already defined. For perspective transformations, the matrix \mathbf{A} is the linearized *derivative* of the perspective transformation (Figure 3.48a), i.e., the local affine approximation to the stretching induced by the projection (Heckbert 1986; Wolberg 1990; Gomes, Darsa, Costa *et al.* 1999; Akenine-Möller and Haines 2002).

To prevent aliasing, we need to pre-filter the image $f(\mathbf{x})$ with a filter whose frequency response is the projection of the final desired spectrum through the \mathbf{A}^{-T} transform (Szeliski, Winder, and Uyttendaele 2010). In general (for non-zoom transforms), this filter is non-separable and hence is very slow to compute. Therefore, a number of approximations to this filter are used in practice, include MIP-mapping, elliptically weighted Gaussian averaging, and anisotropic filtering (Akenine-Möller and Haines 2002).

MIP-mapping

MIP-mapping was first proposed by Williams (1983) as a means to rapidly pre-filter images being used for *texture mapping* in computer graphics. A MIP-map¹⁸ is a standard image pyramid (Figure 3.32), where each level is pre-filtered with a high-quality filter rather than a poorer quality approximation, such as Burt and Adelson's (1983b) five-tap binomial. To resample an image from a MIP-map, a scalar estimate of the resampling rate r is first computed. For example, r can be the maximum of the absolute values in \mathbf{A} (which suppresses aliasing) or it can be the minimum (which reduces blurring). Akenine-Möller and Haines (2002) discuss these issues in more detail.

Once a resampling rate has been specified, a *fractional* pyramid level is computed using the base 2 logarithm,

$$l = \log_2 r. \quad (3.91)$$

¹⁸ The term ‘MIP’ stands for *multi in parvo*, meaning ‘many in one’.

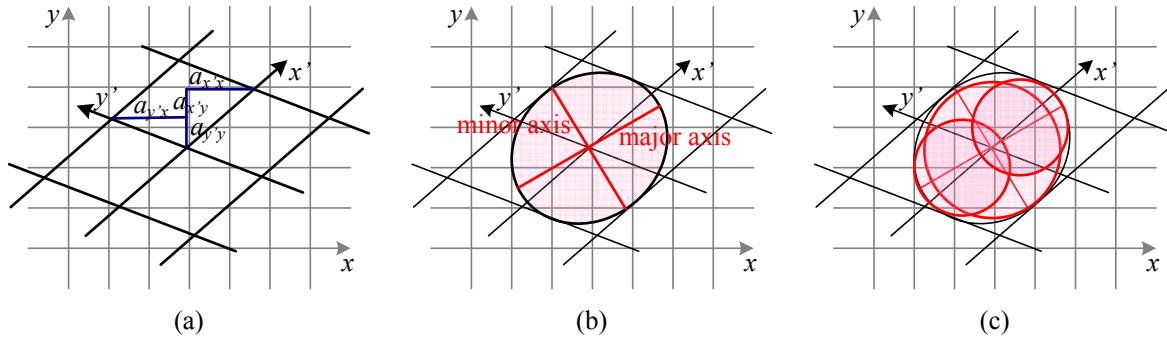


Figure 3.48 Anisotropic texture filtering: (a) Jacobian of transform A and the induced horizontal and vertical resampling rates $\{a_{x'x}, a_{x'y}, a_{y'x}, a_{y'y}\}$; (b) elliptical footprint of an EWA smoothing kernel; (c) anisotropic filtering using multiple samples along the major axis. Image pixels lie at line intersections.

One simple solution is to resample the texture from the next higher or lower pyramid level, depending on whether it is preferable to reduce aliasing or blur. A better solution is to resample *both* images and blend them linearly using the fractional component of l . Since most MIP-map implementations use bilinear resampling within each level, this approach is usually called *trilinear MIP-mapping*. Computer graphics rendering APIs, such as OpenGL and Direct3D, have parameters that can be used to select which variant of MIP-mapping (and of the sampling rate r computation) should be used, depending on the desired tradeoff between speed and quality. Exercise 3.22 has you examine some of these tradeoffs in more detail.

Elliptical Weighted Average

The Elliptical Weighted Average (EWA) filter invented by Greene and Heckbert (1986) is based on the observation that the affine mapping $\mathbf{x} = \mathbf{A}\mathbf{x}'$ defines a skewed two-dimensional coordinate system in the vicinity of each source pixel \mathbf{x} (Figure 3.48a). For every destination pixel \mathbf{x}' , the ellipsoidal projection of a small pixel grid in \mathbf{x}' onto \mathbf{x} is computed (Figure 3.48b). This is then used to filter the source image $g(\mathbf{x})$ with a Gaussian whose inverse covariance matrix is this ellipsoid.

Despite its reputation as a high-quality filter (Akenine-Möller and Haines 2002), we have found in our work (Szeliski, Winder, and Uyttendaele 2010) that because a Gaussian kernel is used, the technique suffers simultaneously from both blurring and aliasing, compared to higher-quality filters. The EWA is also quite slow, although faster variants based on MIP-mapping have been proposed (Szeliski, Winder, and Uyttendaele (2010) provide some additional references).

Anisotropic filtering

An alternative approach to filtering oriented textures, which is sometimes implemented in graphics hardware (GPUs), is to use anisotropic filtering (Barkans 1997; Akenine-Möller and Haines 2002). In this approach, several samples at different resolutions (fractional levels in the MIP-map) are combined along the major axis of the EWA Gaussian (Figure 3.48c).

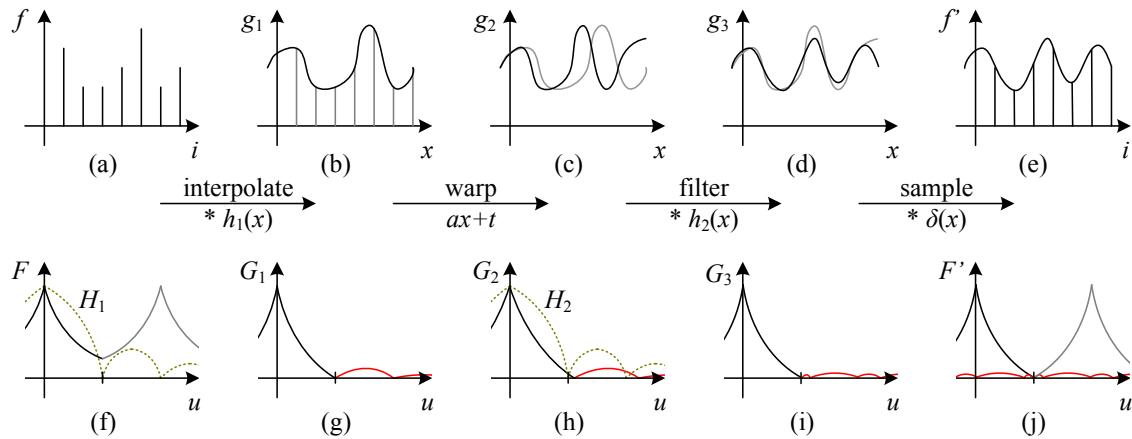


Figure 3.49 One-dimensional signal resampling (Szeliski, Winder, and Uyttendaele 2010): (a) original sampled signal $f(i)$; (b) interpolated signal $g_1(x)$; (c) warped signal $g_2(x)$; (d) filtered signal $g_3(x)$; (e) sampled signal $f'(i)$. The corresponding spectra are shown below the signals, with the aliased portions shown in red.

Multi-pass transforms

The optimal approach to warping images without excessive blurring or aliasing is to adaptively pre-filter the source image at each pixel using an ideal low-pass filter, i.e., an oriented skewed sinc or low-order (e.g., cubic) approximation (Figure 3.48a). Figure 3.49 shows how this works in one dimension. The signal is first (theoretically) interpolated to a continuous waveform, (ideally) low-pass filtered to below the new Nyquist rate, and then re-sampled to the final desired resolution. In practice, the interpolation and decimation steps are concatenated into a single *polyphase* digital filtering operation (Szeliski, Winder, and Uyttendaele 2010).

For parametric transforms, the oriented two-dimensional filtering and resampling operations can be approximated using a series of one-dimensional resampling and shearing transforms (Catmull and Smith 1980; Heckbert 1989; Wolberg 1990; Gomes, Darsa, Costa *et al.* 1999; Szeliski, Winder, and Uyttendaele 2010). The advantage of using a series of one-dimensional transforms is that they are much more efficient (in terms of basic arithmetic operations) than large, non-separable, two-dimensional filter kernels.

In order to prevent aliasing, however, it may be necessary to upsample in the opposite direction before applying a shearing transformation (Szeliski, Winder, and Uyttendaele 2010). Figure 3.50 shows this process for a rotation, where a vertical upsampling stage is added before the horizontal shearing (and upsampling) stage. The upper image shows the appearance of the letter being rotated, while the lower image shows its corresponding Fourier transform.

3.6.2 Mesh-based warping

While parametric transforms specified by a small number of global parameters have many uses, *local* deformations with more degrees of freedom are often required.

Consider, for example, changing the appearance of a face from a frown to a smile (Figure 3.51a). What is needed in this case is to curve the corners of the mouth upwards while

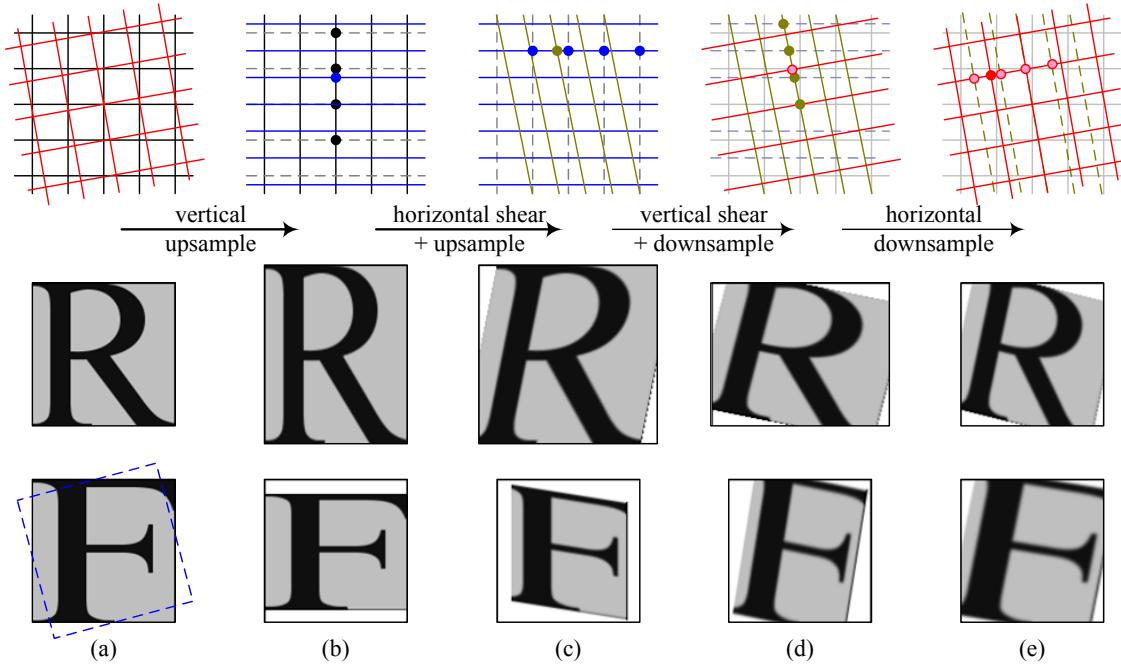


Figure 3.50 Four-pass rotation (Szeliski, Winder, and Uyttendaele 2010): (a) original pixel grid, image, and its Fourier transform; (b) vertical upsampling; (c) horizontal shear and upsampling; (d) vertical shear and downsampling; (e) horizontal downsampling. The general affine case looks similar except that the first two stages perform general resampling.

leaving the rest of the face intact.¹⁹ To perform such a transformation, different amounts of motion are required in different parts of the image. Figure 3.51 shows some of the commonly used approaches.

The first approach, shown in Figure 3.51a–b, is to specify a *sparse* set of corresponding points. The displacement of these points can then be interpolated to a dense *displacement field* (Chapter 8) using a variety of techniques (Nielson 1993). One possibility is to *triangulate* the set of points in one image (de Berg, Cheong, van Kreveld *et al.* 2006; Litwinowicz and Williams 1994; Buck, Finkelstein, Jacobs *et al.* 2000) and to use an *affine* motion model (Table 3.5), specified by the three triangle vertices, inside each triangle. If the destination image is triangulated according to the new vertex locations, an inverse warping algorithm (Figure 3.47) can be used. If the source image is triangulated and used as a *texture map*, computer graphics rendering algorithms can be used to draw the new image (but care must be taken along triangle edges to avoid potential aliasing).

Alternative methods for interpolating a sparse set of displacements include moving nearby quadrilateral mesh vertices, as shown in Figure 3.51a, using *variational* (energy minimizing) interpolants such as regularization (Litwinowicz and Williams 1994), see Section 3.7.1, or using locally weighted (*radial basis function*) combinations of displacements (Nielson 1993). (See (Section 12.3.1) for additional *scattered data interpolation* techniques.) If quadrilateral

¹⁹ Rowland and Perrett (1995); Pighin, Hecker, Lischinski *et al.* (1998); Blanz and Vetter (1999); Leyvand, Cohen-Or, Dror *et al.* (2008) show more sophisticated examples of changing facial expression and appearance.

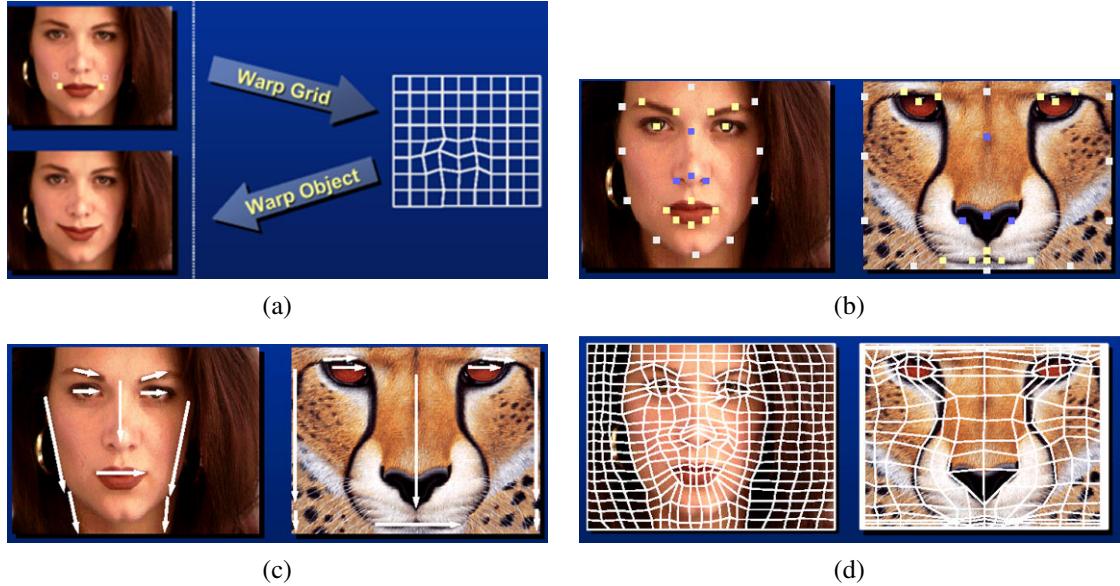


Figure 3.51 Image warping alternatives (Gomes, Darsa, Costa *et al.* 1999) © 1999 Morgan Kaufmann: (a) sparse control points —→ deformation grid; (b) denser set of control point correspondences; (c) oriented line correspondences; (d) uniform quadrilateral grid.

meshes are used, it may be desirable to interpolate displacements down to individual pixel values using a smooth interpolant such as a quadratic B-spline (Farin 1996; Lee, Wolberg, Chwa *et al.* 1996).²⁰

In some cases, e.g., if a dense depth map has been estimated for an image (Shade, Gortler, He *et al.* 1998), we only know the forward displacement for each pixel. As mentioned before, drawing source pixels at their destination location, i.e., forward warping (Figure 3.46), suffers from several potential problems, including aliasing and the appearance of small cracks. An alternative technique in this case is to forward warp the *displacement field* (or depth map) to its new location, fill small holes in the resulting map, and then use inverse warping to perform the resampling (Shade, Gortler, He *et al.* 1998). The reason that this generally works better than forward warping is that displacement fields tend to be much smoother than images, so the aliasing introduced during the forward warping of the displacement field is much less noticeable.

A second approach to specifying displacements for local deformations is to use corresponding *oriented line segments* (Beier and Neely 1992), as shown in Figures 3.51c and 3.52. Pixels along each line segment are transferred from source to destination exactly as specified, and other pixels are warped using a smooth interpolation of these displacements. Each line segment correspondence specifies a translation, rotation, and scaling, i.e., a *similarity transform* (Table 3.5), for pixels in its vicinity, as shown in Figure 3.52a. Line segments influence the overall displacement of the image using a weighting function that depends on the minimum distance to the line segment (v in Figure 3.52a if $u \in [0, 1]$, else the shorter of the two

²⁰ Note that the *block-based* motion models used by many video compression standards (Le Gall 1991) can be thought of as a 0th-order (piecewise-constant) displacement field.

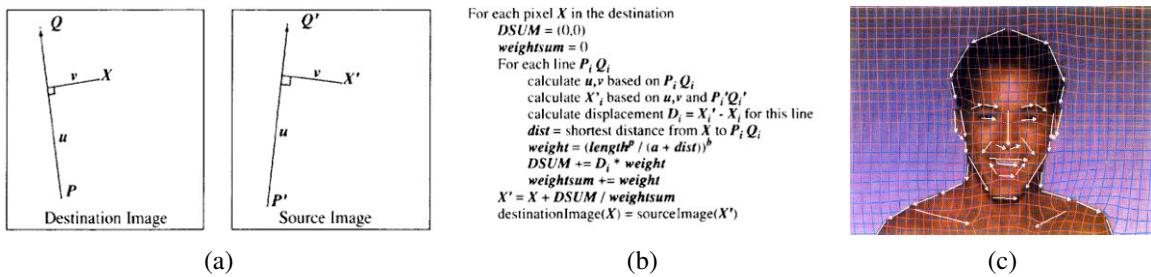


Figure 3.52 Line-based image warping (Beier and Neely 1992) © 1992 ACM: (a) distance computation and position transfer; (b) rendering algorithm; (c) two intermediate warps used for morphing.

distances to P and Q).

For each pixel X , the target location X' for each line correspondence is computed along with a weight that depends on the distance and the line segment length (Figure 3.52b). The weighted average of all target locations X'_i then becomes the final destination location. Note that while Beier and Neely describe this algorithm as a forward warp, an equivalent algorithm can be written by sequencing through the destination pixels. The resulting warps are not identical because line lengths or distances to lines may be different. Exercise 3.23 has you implement the Beier–Neely (line-based) warp and compare it to a number of other local deformation methods.

Yet another way of specifying correspondences in order to create image warps is to use snakes (Section 5.1.1) combined with B-splines (Lee, Wolberg, Chwa *et al.* 1996). This technique is used in Apple’s Shake software and is popular in the medical imaging community.

One final possibility for specifying displacement fields is to use a mesh specifically *adapted* to the underlying image content, as shown in Figure 3.51d. Specifying such meshes by hand can involve a fair amount of work; Gomes, Darsa, Costa *et al.* (1999) describe an interactive system for doing this. Once the two meshes have been specified, intermediate warps can be generated using linear interpolation and the displacements at mesh nodes can be interpolated using splines.

3.6.3 Application: Feature-based morphing

While warps can be used to change the appearance of or to animate a *single* image, even more powerful effects can be obtained by warping and blending two or more images using a process now commonly known as *morphing* (Beier and Neely 1992; Lee, Wolberg, Chwa *et al.* 1996; Gomes, Darsa, Costa *et al.* 1999).

Figure 3.53 shows the essence of image morphing. Instead of simply cross-dissolving between two images, which leads to ghosting as shown in the top row, each image is warped toward the other image before blending, as shown in the bottom row. If the correspondences have been set up well (using any of the techniques shown in Figure 3.51), corresponding features are aligned and no ghosting results.

The above process is repeated for each intermediate frame being generated during a morph, using different blends (and amounts of deformation) at each interval. Let $t \in [0, 1]$ be

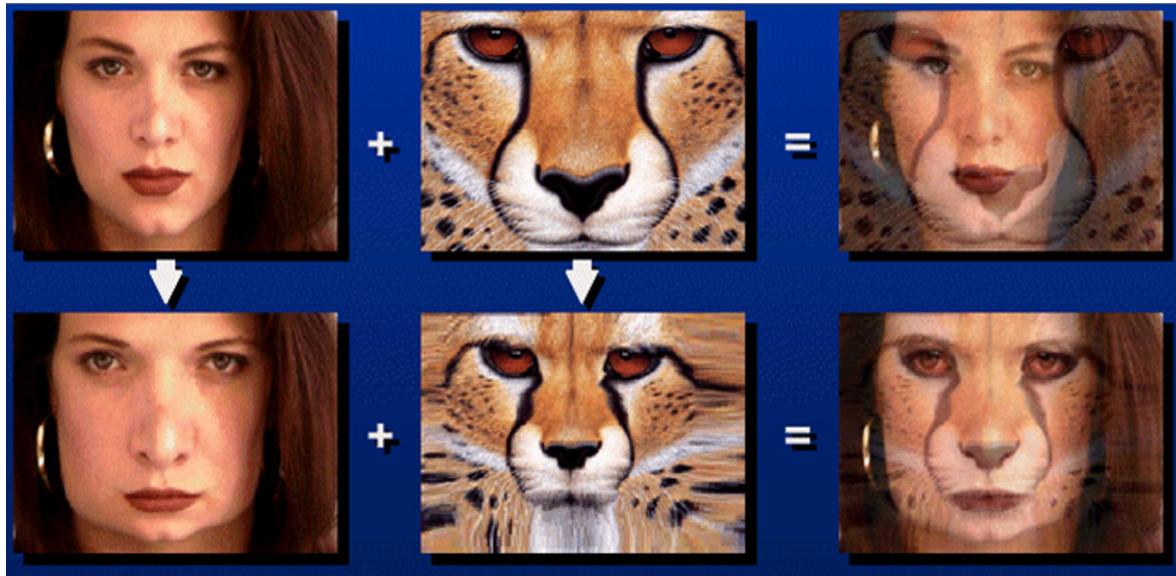


Figure 3.53 Image morphing (Gomes, Darsa, Costa *et al.* 1999) © 1999 Morgan Kaufmann. Top row: if the two images are just blended, visible ghosting results. Bottom row: both images are first warped to the same intermediate location (e.g., halfway towards the other image) and the resulting warped images are then blended resulting in a seamless morph.

the time parameter that describes the sequence of interpolated frames. The weighting functions for the two warped images in the blend go as $(1 - t)$ and t . Conversely, the amount of motion that image 0 undergoes at time t is t of the total amount of motion that is specified by the correspondences. However, some care must be taken in defining what it means to partially warp an image towards a destination, especially if the desired motion is far from linear (Sederberg, Gao, Wang *et al.* 1993). Exercise 3.25 has you implement a morphing algorithm and test it out under such challenging conditions.

3.7 Global optimization

So far in this chapter, we have covered a large number of image processing operators that take as input one or more images and produce some filtered or transformed version of these images. In many applications, it is more useful to first *formulate* the goals of the desired transformation using some optimization criterion and then find or infer the solution that best meets this criterion.

In this final section, we present two different (but closely related) variants on this idea. The first, which is often called *regularization* or *variational methods* (Section 3.7.1), constructs a continuous global energy function that describes the desired characteristics of the solution and then finds a minimum energy solution using sparse linear systems or related iterative techniques. The second formulates the problem using Bayesian statistics, modeling both the noisy measurement process that produced the input images as well as *prior assumptions* about the solution space, which are often encoded using a *Markov random field*

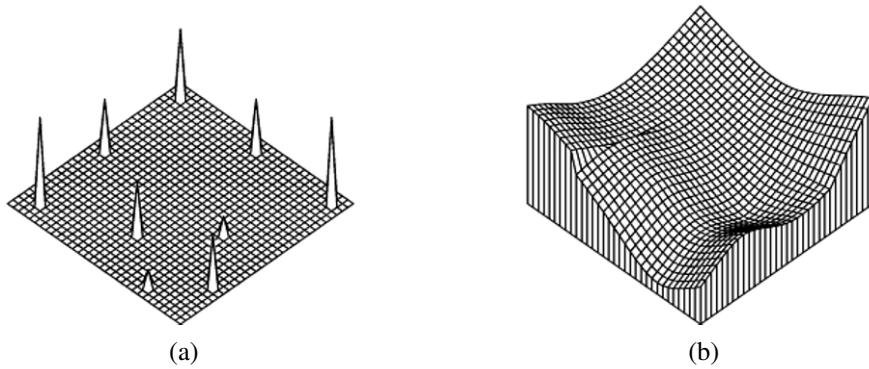


Figure 3.54 A simple surface interpolation problem: (a) nine data points of various height scattered on a grid; (b) second-order, controlled-continuity, thin-plate spline interpolator, with a tear along its left edge and a crease along its right (Szeliski 1989) © 1989 Springer.

(Section 3.7.2).

Examples of such problems include surface interpolation from scattered data (Figure 3.54), image denoising and the restoration of missing regions (Figure 3.57), and the segmentation of images into foreground and background regions (Figure 3.61).

3.7.1 Regularization

The theory of regularization was first developed by statisticians trying to fit models to data that severely underconstrained the solution space (Tikhonov and Arsenin 1977; Engl, Hanke, and Neubauer 1996). Consider, for example, finding a smooth surface that passes through (or near) a set of measured data points (Figure 3.54). Such a problem is described as *ill-posed* because many possible surfaces can fit this data. Since small changes in the input can sometimes lead to large changes in the fit (e.g., if we use polynomial interpolation), such problems are also often *ill-conditioned*. Since we are trying to recover the unknown function $f(x, y)$ from which the data point $d(x_i, y_i)$ were sampled, such problems are also often called *inverse problems*. Many computer vision tasks can be viewed as inverse problems, since we are trying to recover a full description of the 3D world from a limited set of images.

In order to quantify what it means to find a *smooth* solution, we can define a norm on the solution space. For one-dimensional functions $f(x)$, we can integrate the squared first derivative of the function,

$$\mathcal{E}_1 = \int f_x^2(x) dx \quad (3.92)$$

or perhaps integrate the squared second derivative,

$$\mathcal{E}_2 = \int f_{xx}^2(x) dx. \quad (3.93)$$

(Here, we use subscripts to denote differentiation.) Such energy measures are examples of *functionals*, which are operators that map functions to scalar values. They are also often called *variational methods*, because they measure the variation (non-smoothness) in a function.

In two dimensions (e.g., for images, flow fields, or surfaces), the corresponding smoothness functionals are

$$\mathcal{E}_1 = \int f_x^2(x, y) + f_y^2(x, y) dx dy = \int \|\nabla f(x, y)\|^2 dx dy \quad (3.94)$$

and

$$\mathcal{E}_2 = \int f_{xx}^2(x, y) + 2f_{xy}^2(x, y) + f_{yy}^2(x, y) dx dy, \quad (3.95)$$

where the mixed $2f_{xy}^2$ term is needed to make the measure rotationally invariant (Grimson 1983).

The first derivative norm is often called the *membrane*, since interpolating a set of data points using this measure results in a tent-like structure. (In fact, this formula is a small-deflection approximation to the surface area, which is what soap bubbles minimize.) The second-order norm is called the *thin-plate spline*, since it approximates the behavior of thin plates (e.g., flexible steel) under small deformations. A blend of the two is called the *thin-plate spline under tension*; versions of these formulas where each derivative term is multiplied by a local weighting function are called *controlled-continuity splines* (Terzopoulos 1988). Figure 3.54 shows a simple example of a controlled-continuity interpolator fit to nine scattered data points. In practice, it is more common to find first-order smoothness terms used with images and flow fields (Section 8.4) and second-order smoothness associated with surfaces (Section 12.3.1).

In addition to the smoothness term, regularization also requires a data term (or *data penalty*). For scattered data interpolation (Nielson 1993), the data term measures the distance between the function $f(x, y)$ and a set of data points $d_i = d(x_i, y_i)$,

$$\mathcal{E}_d = \sum_i [f(x_i, y_i) - d_i]^2. \quad (3.96)$$

For a problem like noise removal, a continuous version of this measure can be used,

$$\mathcal{E}_d = \int [f(x, y) - d(x, y)]^2 dx dy. \quad (3.97)$$

To obtain a global energy that can be minimized, the two energy terms are usually added together,

$$\mathcal{E} = \mathcal{E}_d + \lambda \mathcal{E}_s, \quad (3.98)$$

where \mathcal{E}_s is the *smoothness penalty* (\mathcal{E}_1 , \mathcal{E}_2 or some weighted blend) and λ is the *regularization parameter*, which controls how smooth the solution should be.

In order to find the minimum of this continuous problem, the function $f(x, y)$ is usually first discretized on a regular grid.²¹ The most principled way to perform this discretization is to use *finite element analysis*, i.e., to approximate the function with a piecewise continuous spline, and then perform the analytic integration (Bathe 2007).

Fortunately, for both the first-order and second-order smoothness functionals, the judicious selection of appropriate finite elements results in particularly simple discrete forms

²¹ The alternative of using *kernel basis functions* centered on the data points (Boult and Kender 1986; Nielson 1993) is discussed in more detail in Section 12.3.1.

(Terzopoulos 1983). The corresponding *discrete* smoothness energy functions become

$$\begin{aligned} E_1 &= \sum_{i,j} s_x(i,j)[f(i+1,j) - f(i,j) - g_x(i,j)]^2 \\ &\quad + s_y(i,j)[f(i,j+1) - f(i,j) - g_y(i,j)]^2 \end{aligned} \quad (3.99)$$

and

$$\begin{aligned} E_2 &= h^{-2} \sum_{i,j} c_x(i,j)[f(i+1,j) - 2f(i,j) + f(i-1,j)]^2 \\ &\quad + 2c_m(i,j)[f(i+1,j+1) - f(i+1,j) - f(i,j+1) + f(i,j)]^2 \\ &\quad + c_y(i,j)[f(i,j+1) - 2f(i,j) + f(i,j-1)]^2, \end{aligned} \quad (3.100)$$

where h is the size of the finite element grid. The h factor is only important if the energy is being discretized at a variety of resolutions, as in coarse-to-fine or multigrid techniques.

The optional smoothness weights $s_x(i,j)$ and $s_y(i,j)$ control the location of horizontal and vertical tears (or weaknesses) in the surface. For other problems, such as colorization (Levin, Lischinski, and Weiss 2004) and interactive tone mapping (Lischinski, Farbman, Uyttendaele *et al.* 2006a), they control the smoothness in the interpolated chroma or exposure field and are often set inversely proportional to the local luminance gradient strength. For second-order problems, the crease variables $c_x(i,j)$, $c_m(i,j)$, and $c_y(i,j)$ control the locations of creases in the surface (Terzopoulos 1988; Szeliski 1990a).

The data values $g_x(i,j)$ and $g_y(i,j)$ are gradient data terms (constraints) used by algorithms, such as photometric stereo (Section 12.1.1), HDR tone mapping (Section 10.2.1) (Fattal, Lischinski, and Werman 2002), Poisson blending (Section 9.3.4) (Pérez, Gangnet, and Blake 2003), and gradient-domain blending (Section 9.3.4) (Levin, Zomet, Peleg *et al.* 2004). They are set to zero when just discretizing the conventional first-order smoothness functional (3.94).

The two-dimensional discrete data energy is written as

$$E_d = \sum_{i,j} w(i,j)[f(i,j) - d(i,j)]^2, \quad (3.101)$$

where the local weights $w(i,j)$ control how strongly the data constraint is enforced. These values are set to zero where there is no data and can be set to the inverse variance of the data measurements when there is data (as discussed by Szeliski (1989) and in Section 3.7.2).

The total energy of the discretized problem can now be written as a *quadratic form*

$$E = E_d + \lambda E_s = \mathbf{x}^T \mathbf{A} \mathbf{x} - 2\mathbf{x}^T \mathbf{b} + c, \quad (3.102)$$

where $\mathbf{x} = [f(0,0) \dots f(m-1,n-1)]$ is called the *state vector*.²²

The sparse symmetric positive-definite matrix \mathbf{A} is called the *Hessian* since it encodes the second derivative of the energy function.²³ For the one-dimensional, first-order problem, \mathbf{A}

²² We use \mathbf{x} instead of \mathbf{f} because this is the more common form in the numerical analysis literature (Golub and Van Loan 1996).

²³ In numerical analysis, \mathbf{A} is called the *coefficient* matrix (Saad 2003); in finite element analysis (Bathe 2007), it is called the *stiffness* matrix.

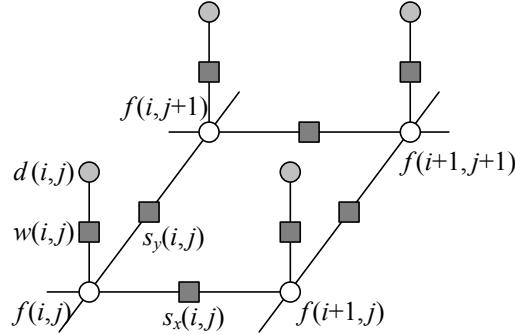


Figure 3.55 Graphical model interpretation of first-order regularization. The white circles are the unknowns $f(i, j)$ while the dark circles are the input data $d(i, j)$. In the resistive grid interpretation, the d and f values encode input and output voltages and the black squares denote resistors whose *conductance* is set to $s_x(i, j)$, $s_y(i, j)$, and $w(i, j)$. In the spring-mass system analogy, the circles denote elevations and the black squares denote springs. The same graphical model can be used to depict a first-order Markov random field (Figure 3.56).

is tridiagonal; for the two-dimensional, first-order problem, it is multi-banded with five non-zero entries per row. We call \mathbf{b} the *weighted data vector*. Minimizing the above quadratic form is equivalent to solving the sparse linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \quad (3.103)$$

which can be done using a variety of sparse matrix techniques, such as multigrid (Briggs, Henson, and McCormick 2000) and hierarchical preconditioners (Szeliski 2006b), as described in Appendix A.5.

While regularization was first introduced to the vision community by Poggio, Torre, and Koch (1985) and Terzopoulos (1986b) for problems such as surface interpolation, it was quickly adopted by other vision researchers for such varied problems as edge detection (Section 4.2), optical flow (Section 8.4), and shape from shading (Section 12.1) (Poggio, Torre, and Koch 1985; Horn and Brooks 1986; Terzopoulos 1986b; Bertero, Poggio, and Torre 1988; Brox, Bruhn, Papenberg *et al.* 2004). Poggio, Torre, and Koch (1985) also showed how the discrete energy defined by Equations (3.100–3.101) could be implemented in a resistive grid, as shown in Figure 3.55. In computational photography (Chapter 10), regularization and its variants are commonly used to solve problems such as high-dynamic range tone mapping (Fattal, Lischinski, and Werman 2002; Lischinski, Farbman, Uyttendaele *et al.* 2006a), Poisson and gradient-domain blending (Pérez, Gangnet, and Blake 2003; Levin, Zomet, Peleg *et al.* 2004; Agarwala, Dontcheva, Agrawala *et al.* 2004), colorization (Levin, Lischinski, and Weiss 2004), and natural image matting (Levin, Lischinski, and Weiss 2008).

Robust regularization

While regularization is most commonly formulated using quadratic (L_2) norms (compare with the squared derivatives in (3.92–3.95) and squared differences in (3.100–3.101)), it can also be formulated using non-quadratic *robust* penalty functions (Appendix B.3). For exam-

ple, (3.100) can be generalized to

$$\begin{aligned} E_{1r} &= \sum_{i,j} s_x(i,j) \rho(f(i+1,j) - f(i,j)) \\ &\quad + s_y(i,j) \rho(f(i,j+1) - f(i,j)), \end{aligned} \quad (3.104)$$

where $\rho(x)$ is some monotonically increasing penalty function. For example, the family of norms $\rho(x) = |x|^p$ is called p -norms. When $p < 2$, the resulting smoothness terms become more piecewise continuous than totally smooth, which can better model the discontinuous nature of images, flow fields, and 3D surfaces.

An early example of robust regularization is the *graduated non-convexity* (GNC) algorithm introduced by Blake and Zisserman (1987). Here, the norms on the data and derivatives are clamped to a maximum value

$$\rho(x) = \min(x^2, V). \quad (3.105)$$

Because the resulting problem is highly non-convex (it has many local minima), a *continuation* method is proposed, where a quadratic norm (which is convex) is gradually replaced by the non-convex robust norm (Allgower and Georg 2003). (Around the same time, Terzopoulos (1988) was also using continuation to infer the tear and crease variables in his surface interpolation problems.)

Today, it is more common to use the L_1 ($p = 1$) norm, which is often called *total variation* (Chan, Osher, and Shen 2001; Tschumperlé and Deriche 2005; Tschumperlé 2006; Kaftory, Schechner, and Zeevi 2007). Other norms, for which the *influence* (derivative) more quickly decays to zero, are presented by Black and Rangarajan (1996); Black, Sapiro, Marimont *et al.* (1998) and discussed in Appendix B.3.

Even more recently, *hyper-Laplacian* norms with $p < 1$ have gained popularity, based on the observation that the log-likelihood distribution of image derivatives follows a $p \approx 0.5 - 0.8$ slope and is therefore a hyper-Laplacian distribution (Simoncelli 1999; Levin and Weiss 2007; Weiss and Freeman 2007; Krishnan and Fergus 2009). Such norms have an even stronger tendency to prefer large discontinuities over small ones. See the related discussion in Section 3.7.2 (3.114).

While least squares regularized problems using L_2 norms can be solved using linear systems, other p -norms require different iterative techniques, such as iteratively reweighted least squares (IRLS), Levenberg–Marquardt, or alternation between local non-linear subproblems and global quadratic regularization (Krishnan and Fergus 2009). Such techniques are discussed in Section 6.1.3 and Appendices A.3 and B.3.

3.7.2 Markov random fields

As we have just seen, regularization, which involves the minimization of energy functionals defined over (piecewise) continuous functions, can be used to formulate and solve a variety of low-level computer vision problems. An alternative technique is to formulate a *Bayesian* model, which separately models the noisy image formation (*measurement*) process, as well as assuming a statistical *prior* model over the solution space. In this section, we look at priors based on Markov random fields, whose log-likelihood can be described using local neighborhood interaction (or penalty) terms (Kindermann and Snell 1980; Geman and Geman 1984; Marroquin, Mitter, and Poggio 1987; Li 1995; Szelski, Zabih, Scharstein *et al.* 2008).

The use of Bayesian modeling has several potential advantages over regularization (see also Appendix B). The ability to model measurement processes statistically enables us to extract the maximum information possible from each measurement, rather than just guessing what weighting to give the data. Similarly, the parameters of the prior distribution can often be *learned* by observing samples from the class we are modeling (Roth and Black 2007a; Tappen 2007; Li and Huttenlocher 2008). Furthermore, because our model is probabilistic, it is possible to estimate (in principle) complete probability *distributions* over the unknowns being recovered and, in particular, to model the *uncertainty* in the solution, which can be useful in latter processing stages. Finally, Markov random field models can be defined over *discrete* variables, such as image labels (where the variables have no proper ordering), for which regularization does not apply.

Recall from (3.68) in Section 3.4.3 (or see Appendix B.4) that, according to Bayes' Rule, the *posterior* distribution for a given set of measurements \mathbf{y} , $p(\mathbf{y}|\mathbf{x})$, combined with a prior $p(\mathbf{x})$ over the unknowns \mathbf{x} , is given by

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{y})}, \quad (3.106)$$

where $p(\mathbf{y}) = \int_{\mathbf{x}} p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$ is a normalizing constant used to make the $p(\mathbf{x}|\mathbf{y})$ distribution *proper* (integrate to 1). Taking the negative logarithm of both sides of (3.106), we get

$$-\log p(\mathbf{x}|\mathbf{y}) = -\log p(\mathbf{y}|\mathbf{x}) - \log p(\mathbf{x}) + C, \quad (3.107)$$

which is the *negative posterior log likelihood*.

To find the most likely (*maximum a posteriori* or MAP) solution \mathbf{x} given some measurements \mathbf{y} , we simply minimize this negative log likelihood, which can also be thought of as an *energy*,

$$E(\mathbf{x}, \mathbf{y}) = E_d(\mathbf{x}, \mathbf{y}) + E_p(\mathbf{x}). \quad (3.108)$$

(We drop the constant C because its value does not matter during energy minimization.) The first term $E_d(\mathbf{x}, \mathbf{y})$ is the *data energy* or *data penalty*; it measures the negative log likelihood that the data were observed given the unknown state \mathbf{x} . The second term $E_p(\mathbf{x})$ is the *prior energy*; it plays a role analogous to the smoothness energy in regularization. Note that the MAP estimate may not always be desirable, since it selects the “peak” in the posterior distribution rather than some more stable statistic—see the discussion in Appendix B.2 and by Levin, Weiss, Durand *et al.* (2009).

For image processing applications, the unknowns \mathbf{x} are the set of output pixels

$$\mathbf{x} = [f(0, 0) \dots f(m-1, n-1)],$$

and the data are (in the simplest case) the input pixels

$$\mathbf{y} = [d(0, 0) \dots d(m-1, n-1)]$$

as shown in Figure 3.56.

For a Markov random field, the probability $p(\mathbf{x})$ is a *Gibbs* or *Boltzmann distribution*, whose negative log likelihood (according to the Hammersley–Clifford theorem) can be written as a sum of pairwise interaction potentials,

$$E_p(\mathbf{x}) = \sum_{\{(i,j),(k,l)\} \in \mathcal{N}} V_{i,j,k,l}(f(i,j), f(k,l)), \quad (3.109)$$

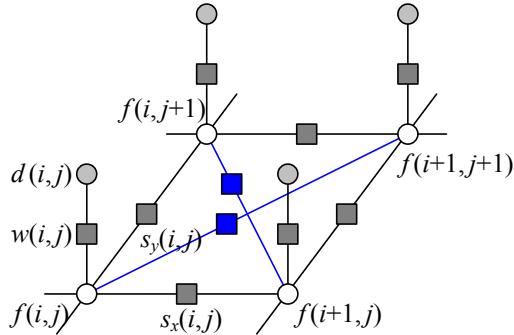


Figure 3.56 Graphical model for an \mathcal{N}_4 neighborhood Markov random field. (The blue edges are added for an \mathcal{N}_8 neighborhood.) The white circles are the unknowns $f(i, j)$, while the dark circles are the input data $d(i, j)$. The $s_x(i, j)$ and $s_y(i, j)$ black boxes denote arbitrary *interaction potentials* between adjacent nodes in the random field, and the $w(i, j)$ denote the *data penalty* functions. The same graphical model can be used to depict a discrete version of a first-order regularization problem (Figure 3.55).

where $\mathcal{N}(i, j)$ denotes the *neighbors* of pixel (i, j) . In fact, the general version of the theorem says that the energy may have to be evaluated over a larger set of *cliques*, which depend on the *order* of the Markov random field (Kindermann and Snell 1980; Geman and Geman 1984; Bishop 2006; Kohli, Ladický, and Torr 2009; Kohli, Kumar, and Torr 2009).

The most commonly used neighborhood in Markov random field modeling is the \mathcal{N}_4 neighborhood, where each pixel in the field $f(i, j)$ interacts only with its immediate neighbors. The model in Figure 3.56, which we previously used in Figure 3.55 to illustrate the discrete version of first-order regularization, shows an \mathcal{N}_4 MRF. The $s_x(i, j)$ and $s_y(i, j)$ black boxes denote arbitrary *interaction potentials* between adjacent nodes in the random field and the $w(i, j)$ denote the data penalty functions. These square nodes can also be interpreted as *factors* in a *factor graph* version of the (undirected) graphical model (Bishop 2006), which is another name for interaction potentials. (Strictly speaking, the factors are (improper) probability functions whose product is the (un-normalized) posterior distribution.)

As we will see in (3.112–3.113), there is a close relationship between these interaction potentials and the discretized versions of regularized image restoration problems. Thus, to a first approximation, we can view energy minimization being performed when solving a regularized problem and the maximum a posteriori inference being performed in an MRF as equivalent.

While \mathcal{N}_4 neighborhoods are most commonly used, in some applications \mathcal{N}_8 (or even higher order) neighborhoods perform better at tasks such as image segmentation because they can better model discontinuities at different orientations (Boykov and Kolmogorov 2003; Rother, Kohli, Feng *et al.* 2009; Kohli, Ladický, and Torr 2009; Kohli, Kumar, and Torr 2009).

Binary MRFs

The simplest possible example of a Markov random field is a binary field. Examples of such fields include 1-bit (black and white) scanned document images as well as images segmented into foreground and background regions.

To denoise a scanned image, we set the data penalty to reflect the agreement between the

scanned and final images,

$$E_d(i, j) = w\delta(f(i, j), d(i, j)) \quad (3.110)$$

and the smoothness penalty to reflect the agreement between neighboring pixels

$$E_p(i, j) = E_x(i, j) + E_y(i, j) = s\delta(f(i, j), f(i+1, j)) + s\delta(f(i, j), f(i, j+1)). \quad (3.111)$$

Once we have formulated the energy, how do we minimize it? The simplest approach is to perform gradient descent, flipping one state at a time if it produces a lower energy. This approach is known as *contextual classification* (Kittler and Föglein 1984), *iterated conditional modes* (ICM) (Besag 1986), or *highest confidence first* (HCF) (Chou and Brown 1990) if the pixel with the largest energy decrease is selected first.

Unfortunately, these downhill methods tend to get easily stuck in local minima. An alternative approach is to add some randomness to the process, which is known as *stochastic gradient descent* (Metropolis, Rosenbluth, Rosenbluth *et al.* 1953; Geman and Geman 1984). When the amount of noise is decreased over time, this technique is known as *simulated annealing* (Kirkpatrick, Gelatt, and Vecchi 1983; Carnevali, Coletti, and Patarnello 1985; Wolberg and Pavlidis 1985; Swendsen and Wang 1987) and was first popularized in computer vision by Geman and Geman (1984) and later applied to stereo matching by Barnard (1989), among others.

Even this technique, however, does not perform that well (Boykov, Veksler, and Zabih 2001). For binary images, a much better technique, introduced to the computer vision community by Boykov, Veksler, and Zabih (2001) is to re-formulate the energy minimization as a *max-flow/min-cut* graph optimization problem (Greig, Porteous, and Seheult 1989). This technique has informally come to be known as *graph cuts* in the computer vision community (Boykov and Kolmogorov 2010). For simple energy functions, e.g., those where the penalty for non-identical neighboring pixels is a constant, this algorithm is guaranteed to produce the *global minimum*. Kolmogorov and Zabih (2004) formally characterize the class of binary energy potentials (*regularity conditions*) for which these results hold, while newer work by Komodakis, Tziritas, and Paragios (2008) and Rother, Kolmogorov, Lempitsky *et al.* (2007) provide good algorithms for the cases when they do not.

In addition to the above mentioned techniques, a number of other optimization approaches have been developed for MRF energy minimization, such as (loopy) belief propagation and dynamic programming (for one-dimensional problems). These are discussed in more detail in Appendix B.5 as well as the comparative survey paper by Szeliski, Zabih, Scharstein *et al.* (2008).

Ordinal-valued MRFs

In addition to binary images, Markov random fields can be applied to ordinal-valued labels such as grayscale images or depth maps. The term "ordinal" indicates that the labels have an implied ordering, e.g., that higher values are lighter pixels. In the next section, we look at unordered labels, such as source image labels for image compositing.

In many cases, it is common to extend the binary data and smoothness prior terms as

$$E_d(i, j) = w(i, j)\rho_d(f(i, j) - d(i, j)) \quad (3.112)$$

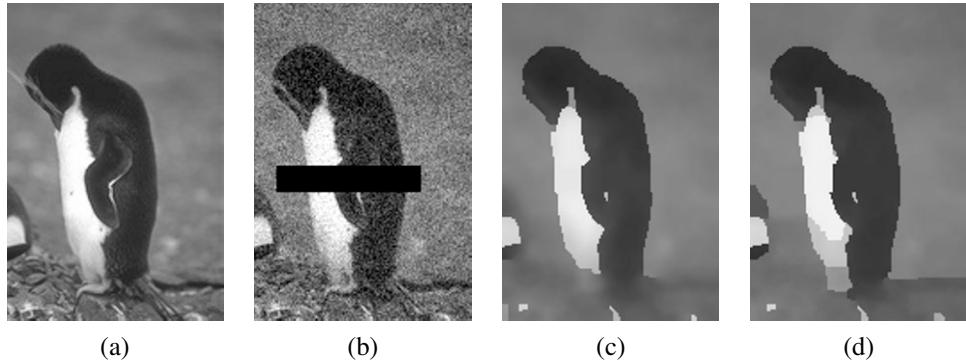


Figure 3.57 Grayscale image denoising and inpainting: (a) original image; (b) image corrupted by noise and with missing data (black bar); (c) image restored using loopy belief propagation; (d) image restored using expansion move graph cuts. Images are from <http://vision.middlebury.edu/MRF/results/> (Szeliski, Zabih, Scharstein *et al.* 2008).

and

$$E_p(i, j) = s_x(i, j)\rho_p(f(i, j) - f(i + 1, j)) + s_y(i, j)\rho_p(f(i, j) - f(i, j + 1)), \quad (3.113)$$

which are robust generalizations of the quadratic penalty terms (3.101) and (3.100), first introduced in (3.105). As before, the $w(i, j)$, $s_x(i, j)$ and $s_y(i, j)$ weights can be used to locally control the data weighting and the horizontal and vertical smoothness. Instead of using a quadratic penalty, however, a general monotonically increasing penalty function $\rho()$ is used. (Different functions can be used for the data and smoothness terms.) For example, ρ_p can be a hyper-Laplacian penalty

$$\rho_p(d) = |d|^p, \quad p < 1, \quad (3.114)$$

which better encodes the distribution of gradients (mainly edges) in an image than either a quadratic or linear (total variation) penalty.²⁴ Levin and Weiss (2007) use such a penalty to separate a transmitted and reflected image (Figure 8.17) by encouraging gradients to lie in one or the other image, but not both. More recently, Levin, Fergus, Durand *et al.* (2007) use the hyper-Laplacian as a prior for image deconvolution (deblurring) and Krishnan and Fergus (2009) develop a faster algorithm for solving such problems. For the data penalty, ρ_d can be quadratic (to model Gaussian noise) or the log of a *contaminated Gaussian* (Appendix B.3).

When ρ_p is a quadratic function, the resulting Markov random field is called a Gaussian Markov random field (GMRF) and its minimum can be found by sparse linear system solving (3.103). When the weighting functions are uniform, the GMRF becomes a special case of Wiener filtering (Section 3.4.3). Allowing the weighting functions to depend on the input image (a special kind of conditional random field, which we describe below) enables quite sophisticated image processing algorithms to be performed, including colorization (Levin, Lischinski, and Weiss 2004), interactive tone mapping (Lischinski, Farbman, Uyttendaele *et*

²⁴ Note that, unlike a quadratic penalty, the sum of the horizontal and vertical derivative p -norms is not rotationally invariant. A better approach may be to locally estimate the gradient direction and to impose different norms on the perpendicular and parallel components, which Roth and Black (2007b) call a *steerable random field*.

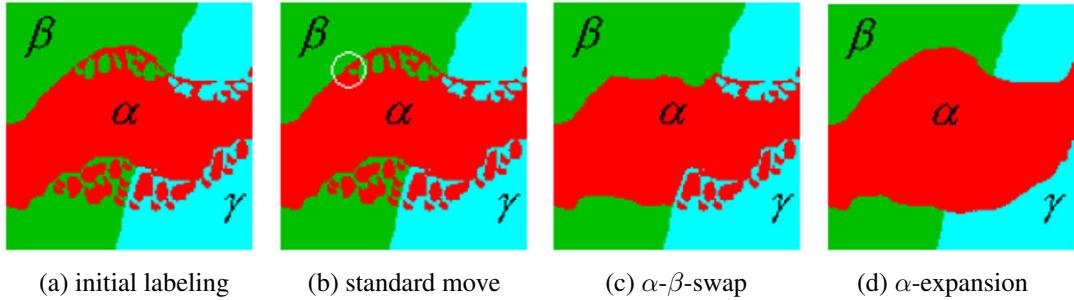


Figure 3.58 Multi-level graph optimization from (Boykov, Veksler, and Zabih 2001) © 2001 IEEE: (a) initial problem configuration; (b) the standard move only changes one pixel; (c) the α - β -swap optimally exchanges all α and β -labeled pixels; (d) the α -expansion move optimally selects among current pixel values and the α label.

al. 2006a), natural image matting (Levin, Lischinski, and Weiss 2008), and image restoration (Tappen, Liu, Freeman *et al.* 2007).

When ρ_d or ρ_p are non-quadratic functions, gradient descent techniques such as non-linear least squares or iteratively re-weighted least squares can sometimes be used (Appendix A.3). However, if the search space has lots of local minima, as is the case for stereo matching (Barnard 1989; Boykov, Veksler, and Zabih 2001), more sophisticated techniques are required.

The extension of graph cut techniques to multi-valued problems was first proposed by Boykov, Veksler, and Zabih (2001). In their paper, they develop two different algorithms, called the *swap move* and the *expansion move*, which iterate among a series of binary labeling sub-problems to find a good solution (Figure 3.58). Note that a global solution is generally not achievable, as the problem is provably NP-hard for general energy functions. Because both these algorithms use a binary MRF optimization inside their inner loop, they are subject to the kind of constraints on the energy functions that occur in the binary labeling case (Kolmogorov and Zabih 2004). Appendix B.5.4 discusses these algorithms in more detail, along with some more recently developed approaches to this problem.

Another MRF inference technique is *belief propagation* (BP). While belief propagation was originally developed for inference over trees, where it is exact (Pearl 1988), it has more recently been applied to graphs with loops such as Markov random fields (Freeman, Pasztor, and Carmichael 2000; Yedidia, Freeman, and Weiss 2001). In fact, some of the better performing stereo-matching algorithms use loopy belief propagation (LBP) to perform their inference (Sun, Zheng, and Shum 2003). LBP is discussed in more detail in Appendix B.5.3 as well as the comparative survey paper on MRF optimization (Szeliski, Zabih, Scharstein *et al.* 2008).

Figure 3.57 shows an example of image denoising and inpainting (hole filling) using a non-quadratic energy function (non-Gaussian MRF). The original image has been corrupted by noise and a portion of the data has been removed (the black bar). In this case, the loopy belief propagation algorithm computes a slightly lower energy and also a smoother image than the alpha-expansion graph cut algorithm.

Of course, the above formula (3.113) for the smoothness term $E_p(i, j)$ just shows the simplest case. In more recent work, Roth and Black (2009) propose a *Field of Experts* (FoE) model, which sums up a large number of exponentiated local filter outputs to arrive at the

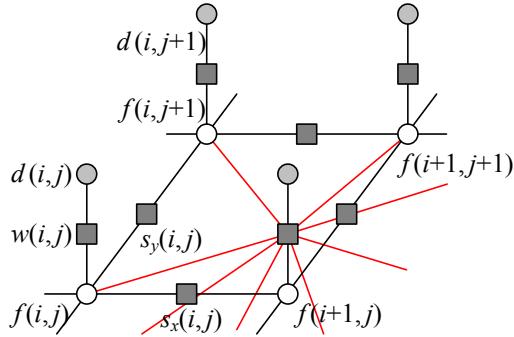


Figure 3.59 Graphical model for a Markov random field with a more complex measurement model. The additional colored edges show how combinations of unknown values (say, in a sharp image) produce the measured values (a noisy blurred image). The resulting graphical model is still a classic MRF and is just as easy to sample from, but some inference algorithms (e.g., those based on graph cuts) may not be applicable because of the increased network complexity, since state changes during the inference become more entangled and the posterior MRF has much larger cliques.

smoothness penalty. Weiss and Freeman (2007) analyze this approach and compare it to the simpler hyper-Laplacian model of natural image statistics. Lyu and Simoncelli (2009) use *Gaussian Scale Mixtures* (GSMs) to construct an inhomogeneous multi-scale MRF, with one (positive exponential) GMRF modulating the variance (amplitude) of another Gaussian MRF.

It is also possible to extend the *measurement* model to make the sampled (noise-corrupted) input pixels correspond to blends of unknown (latent) image pixels, as in Figure 3.59. This is the commonly occurring case when trying to de-blur an image. While this kind of a model is still a traditional generative Markov random field, finding an optimal solution can be difficult because the clique sizes get larger. In such situations, gradient descent techniques, such as iteratively reweighted least squares, can be used (Joshi, Zitnick, Szeliski *et al.* 2009). Exercise 3.31 has you explore some of these issues.

Unordered labels

Another case with multi-valued labels where Markov random fields are often applied are *unordered labels*, i.e., labels where there is no semantic meaning to the numerical difference between the values of two labels. For example, if we are classifying terrain from aerial imagery, it makes no sense to take the numeric difference between the labels assigned to forest, field, water, and pavement. In fact, the adjacencies of these various kinds of terrain each have different likelihoods, so it makes more sense to use a prior of the form

$$E_p(i,j) = s_x(i,j)V(l(i,j), l(i+1,j)) + s_y(i,j)V(l(i,j), l(i,j+1)), \quad (3.115)$$

where $V(l_0, l_1)$ is a general *compatibility* or *potential* function. (Note that we have also replaced $f(i,j)$ with $l(i,j)$ to make it clearer that these are labels rather than discrete function samples.) An alternative way to write this prior energy (Boykov, Veksler, and Zabih 2001;



Figure 3.60 An unordered label MRF (Agarwala, Dontcheva, Agrawala *et al.* 2004) © 2004 ACM: Strokes in each of the source images on the left are used as constraints on an MRF optimization, which is solved using graph cuts. The resulting multi-valued label field is shown as a color overlay in the middle image, and the final composite is shown on the right.

Szeliski, Zabih, Scharstein *et al.* 2008) is

$$E_p = \sum_{(p,q) \in \mathcal{N}} V_{p,q}(l_p, l_q), \quad (3.116)$$

where the (p, q) are neighboring pixels and a spatially varying potential function $V_{p,q}$ is evaluated for each neighboring pair.

An important application of unordered MRF labeling is seam finding in image compositing (Davis 1998; Agarwala, Dontcheva, Agrawala *et al.* 2004) (see Figure 3.60, which is explained in more detail in Section 9.3.2). Here, the compatibility $V_{p,q}(l_p, l_q)$ measures the quality of the visual appearance that would result from placing a pixel p from image I_p next to a pixel q from image I_q . As with most MRFs, we assume that $V_{p,q}(l, l) = 0$, i.e., it is perfectly fine to choose contiguous pixels from the same image. For different labels, however, the compatibility $V_{p,q}(l_p, l_q)$ may depend on the values of the underlying pixels $I_{l_p}(p)$ and $I_{l_q}(q)$.

Consider, for example, where one image I_0 is all sky blue, i.e., $I_0(p) = I_0(q) = B$, while the other image I_1 has a transition from sky blue, $I_1(p) = B$, to forest green, $I_1(q) = G$.

$$I_0 : \begin{array}{|c|c|} \hline p & q \\ \hline \end{array} \quad \begin{array}{|c|c|} \hline p & q \\ \hline \end{array} : I_1$$

In this case, $V_{p,q}(1, 0) = 0$ (the colors agree), while $V_{p,q}(0, 1) > 0$ (the colors disagree).

Conditional random fields

In a classic Bayesian model (3.106–3.108),

$$p(\mathbf{x}|\mathbf{y}) \propto p(\mathbf{y}|\mathbf{x})p(\mathbf{x}), \quad (3.117)$$

the prior distribution $p(\mathbf{x})$ is independent of the observations \mathbf{y} . Sometimes, however, it is useful to modify our prior assumptions, say about the smoothness of the field we are trying to estimate, in response to the sensed data. Whether this makes sense from a probability viewpoint is something we discuss once we have explained the new model.



Figure 3.61 Image segmentation (Boykov and Funka-Lea 2006) © 2006 Springer: The user draws a few red strokes in the foreground object and a few blue ones in the background. The system computes color distributions for the foreground and background and solves a binary MRF. The smoothness weights are modulated by the intensity gradients (edges), which makes this a conditional random field (CRF).

Consider the interactive image segmentation problem shown in Figure 3.61 (Boykov and Funka-Lea 2006). In this application, the user draws foreground (red) and background (blue) strokes, and the system then solves a binary MRF labeling problem to estimate the extent of the foreground object. In addition to minimizing a data term, which measures the pointwise similarity between pixel colors and the inferred region distributions (Section 5.5), the MRF is modified so that the smoothness terms $s_x(x, y)$ and $s_y(x, y)$ in Figure 3.56 and (3.113) depend on the magnitude of the gradient between adjacent pixels.²⁵

Since the smoothness term now depends on the data, Bayes' Rule (3.117) no longer applies. Instead, we use a direct model for the posterior distribution $p(\mathbf{x}|\mathbf{y})$, whose negative log likelihood can be written as

$$\begin{aligned} E(\mathbf{x}|\mathbf{y}) &= E_d(\mathbf{x}, \mathbf{y}) + E_s(\mathbf{x}, \mathbf{y}) \\ &= \sum_p V_p(x_p, \mathbf{y}) + \sum_{(p,q) \in \mathcal{N}} V_{p,q}(x_p, x_q, \mathbf{y}), \end{aligned} \quad (3.118)$$

using the notation introduced in (3.116). The resulting probability distribution is called a *conditional random field* (CRF) and was first introduced to the computer vision field by Kumar and Hebert (2003), based on earlier work in text modeling by Lafferty, McCallum, and Pereira (2001).

Figure 3.62 shows a graphical model where the smoothness terms depend on the data values. In this particular model, each smoothness term depends only on its adjacent pair of data values, i.e., terms are of the form $V_{p,q}(x_p, x_q, y_p, y_q)$ in (3.118).

The idea of modifying smoothness terms in response to input data is not new. For example, Boykov and Jolly (2001) used this idea for interactive segmentation, as shown in Figure 3.61, and it is now widely used in image segmentation (Section 5.5) (Blake, Rother, Brown *et al.* 2004; Rother, Kolmogorov, and Blake 2004), denoising (Tappen, Liu, Freeman *et al.* 2007), and object recognition (Section 14.4.3) (Winn and Shotton 2006; Shotton, Winn, Rother *et al.* 2009).

²⁵ An alternative formulation that also uses detected edges to modulate the smoothness of a depth or motion field and hence to integrate multiple lower level vision modules is presented by Poggio, Gamble, and Little (1988).

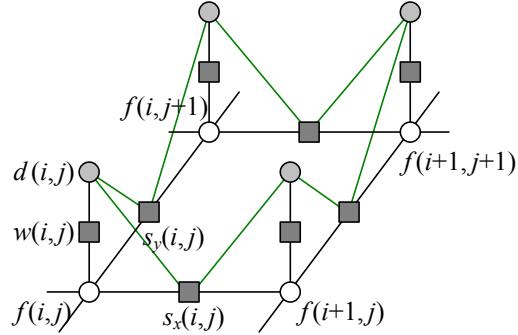


Figure 3.62 Graphical model for a conditional random field (CRF). The additional green edges show how combinations of sensed data influence the smoothness in the underlying MRF prior model, i.e., $s_x(i, j)$ and $s_y(i, j)$ in (3.113) depend on adjacent $d(i, j)$ values. These additional links (factors) enable the smoothness to depend on the input data. However, they make sampling from this MRF more complex.

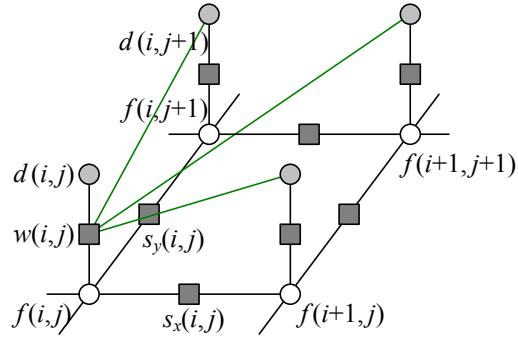


Figure 3.63 Graphical model for a discriminative random field (DRF). The additional green edges show how combinations of sensed data, e.g., $d(i, j + 1)$, influence the data term for $f(i, j)$. The generative model is therefore more complex, i.e., we cannot just apply a simple function to the unknown variables and add noise.

In stereo matching, the idea of encouraging disparity discontinuities to coincide with intensity edges goes back even further to the early days of optimization and MRF-based algorithms (Poggio, Gamble, and Little 1988; Fua 1993; Bobick and Intille 1999; Boykov, Veksler, and Zabih 2001) and is discussed in more detail in (Section 11.5).

In addition to using smoothness terms that adapt to the input data, Kumar and Hebert (2003) also compute a neighborhood function over the input data for each $V_p(x_p, \mathbf{y})$ term, as illustrated in Figure 3.63, instead of using the classic unary MRF data term $V_p(x_p, y_p)$ shown in Figure 3.56.²⁶ Because such neighborhood functions can be thought of as *discriminative* functions (a term widely used in machine learning (Bishop 2006)), they call the resulting graphical model a *discriminative random field* (DRF). In their paper, Kumar and Hebert (2006) show that DRFs outperform similar CRFs on a number of applications, such as structure detection (Figure 3.64) and binary image denoising.

Here again, one could argue that previous stereo correspondence algorithms also look at

²⁶ Kumar and Hebert (2006) call the unary potentials $V_p(x_p, \mathbf{y})$ *association potentials* and the pairwise potentials $V_{p,q}(x_p, y_q, \mathbf{y})$ *interaction potentials*.

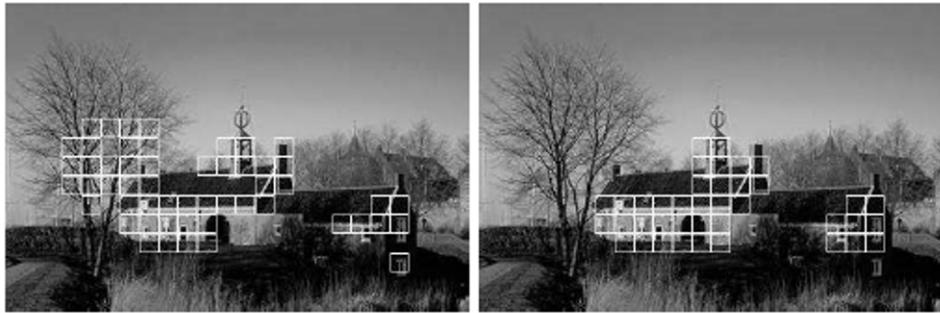


Figure 3.64 Structure detection results using an MRF (left) and a DRF (right) (Kumar and Hebert 2006) © 2006 Springer.

a neighborhood of input data, either explicitly, because they compute correlation measures (Criminisi, Cross, Blake *et al.* 2006) as data terms, or implicitly, because even pixel-wise disparity costs look at several pixels in either the left or right image (Barnard 1989; Boykov, Veksler, and Zabih 2001).

What, then are the advantages and disadvantages of using conditional or discriminative random fields instead of MRFs?

Classic Bayesian inference (MRF) assumes that the prior distribution of the data is independent of the measurements. This makes a lot of sense: if you see a pair of sixes when you first throw a pair of dice, it would be unwise to assume that they will always show up thereafter. However, if after playing for a long time you detect a statistically significant bias, you may want to adjust your prior. What CRFs do, in essence, is to select or modify the prior model based on observed data. This can be viewed as making a partial inference over additional hidden variables or correlations between the unknowns (say, a label, depth, or clean image) and the knowns (observed images).

In some cases, the CRF approach makes a lot of sense and is, in fact, the only plausible way to proceed. For example, in grayscale image colorization (Section 10.3.2) (Levin, Lischinski, and Weiss 2004), the best way to transfer the continuity information from the input grayscale image to the unknown color image is to modify local smoothness constraints. Similarly, for simultaneous segmentation and recognition (Winn and Shotton 2006; Shotton, Winn, Rother *et al.* 2009), it makes a lot of sense to permit strong color edges to influence the semantic image label continuities.

In other cases, such as image denoising, the situation is more subtle. Using a non-quadratic (robust) smoothness term as in (3.113) plays a qualitatively similar role to setting the smoothness based on local gradient information in a Gaussian MRF (GMRF) (Tappen, Liu, Freeman *et al.* 2007). (In more recent work, Tanaka and Okutomi (2008) use a larger neighborhood and full covariance matrix on a related Gaussian MRF.) The advantage of Gaussian MRFs, when the smoothness can be correctly inferred, is that the resulting quadratic energy can be minimized in a single step. However, for situations where the discontinuities are not self-evident in the input data, such as for piecewise-smooth sparse data interpolation (Blake and Zisserman 1987; Terzopoulos 1988), classic robust smoothness energy minimization may be preferable. Thus, as with most computer vision algorithms, a careful analysis of

the problem at hand and desired robustness and computation constraints may be required to choose the best technique.

Perhaps the biggest advantage of CRFs and DRFs, as argued by Kumar and Hebert (2006), Tappen, Liu, Freeman *et al.* (2007) and Blake, Rother, Brown *et al.* (2004), is that learning the model parameters is sometimes easier. While learning parameters in MRFs and their variants is not a topic that we cover in this book, interested readers can find more details in recently published articles (Kumar and Hebert 2006; Roth and Black 2007a; Tappen, Liu, Freeman *et al.* 2007; Tappen 2007; Li and Huttenlocher 2008).

3.7.3 Application: Image restoration

In Section 3.4.4, we saw how two-dimensional linear and non-linear filters can be used to remove noise or enhance sharpness in images. Sometimes, however, images are degraded by larger problems, such as scratches and blotches (Kokaram 2004). In this case, Bayesian methods such as MRFs, which can model spatially varying per-pixel measurement noise, can be used instead. An alternative is to use hole filling or inpainting techniques (Bertalmio, Sapiro, Caselles *et al.* 2000; Bertalmio, Vese, Sapiro *et al.* 2003; Criminisi, Pérez, and Toyama 2004), as discussed in Sections 5.1.4 and 10.5.1.

Figure 3.57 shows an example of image denoising and inpainting (hole filling) using a Markov random field. The original image has been corrupted by noise and a portion of the data has been removed. In this case, the loopy belief propagation algorithm computes a slightly lower energy and also a smoother image than the alpha-expansion graph cut algorithm.

3.8 Additional reading

If you are interested in exploring the topic of image processing in more depth, some popular textbooks have been written by Lim (1990); Crane (1997); Gomes and Velho (1997); Jähne (1997); Pratt (2007); Russ (2007); Burger and Burge (2008); Gonzales and Woods (2008). The pre-eminent conference and journal in this field are the IEEE Conference on Image Processing and the IEEE Transactions on Image Processing.

For image compositing operators, the seminal reference is by Porter and Duff (1984) while Blinn (1994a,b) provides a more detailed tutorial. For image compositing, Smith and Blinn (1996) were the first to bring this topic to the attention of the graphics community, while Wang and Cohen (2007a) provide a recent in-depth survey.

In the realm of linear filtering, Freeman and Adelson (1991) provide a great introduction to separable and steerable oriented band-pass filters, while Perona (1995) shows how to approximate any filter as a sum of separable components.

The literature on non-linear filtering is quite wide and varied; it includes such topics as bilateral filtering (Tomasi and Manduchi 1998; Durand and Dorsey 2002; Paris and Durand 2006; Chen, Paris, and Durand 2007; Paris, Kornprobst, Tumblin *et al.* 2008), related iterative algorithms (Saint-Marc, Chen, and Medioni 1991; Nielsen, Florack, and Deriche 1997; Black, Sapiro, Marimont *et al.* 1998; Weickert, ter Haar Romeny, and Viergever 1998; Weickert 1998; Barash 2002; Scharr, Black, and Haussecker 2003; Barash and Comaniciu 2004),

and variational approaches (Chan, Osher, and Shen 2001; Tschumperlé and Deriche 2005; Tschumperlé 2006; Kaftory, Schechner, and Zeevi 2007).

Good references to image morphology include (Haralick and Shapiro 1992, Section 5.2; Bovik 2000, Section 2.2; Ritter and Wilson 2000, Section 7; Serra 1982; Serra and Vincent 1992; Yuille, Vincent, and Geiger 1992; Soille 2006).

The classic papers for image pyramids and pyramid blending are by Burt and Adelson (1983a,b). Wavelets were first introduced to the computer vision community by Mallat (1989) and good tutorial and review papers and books are available (Strang 1989; Simoncelli and Adelson 1990b; Rioul and Vetterli 1991; Chui 1992; Meyer 1993; Sweldens 1997). Wavelets are widely used in the computer graphics community to perform multi-resolution geometric processing (Stollnitz, DeRose, and Salesin 1996) and have been used in computer vision for similar applications (Szeliski 1990b; Pentland 1994; Gortler and Cohen 1995; Yaou and Chang 1994; Lai and Vemuri 1997; Szeliski 2006b), as well as for multi-scale oriented filtering (Simoncelli, Freeman, Adelson *et al.* 1992) and denoising (Portilla, Strela, Wainwright *et al.* 2003).

While image pyramids (Section 3.5.3) are usually constructed using linear filtering operators, some recent work has started investigating non-linear filters, since these can better preserve details and other salient features. Some representative papers in the computer vision literature are by Gluckman (2006a,b); Lyu and Simoncelli (2008) and in computational photography by Bae, Paris, and Durand (2006); Farbman, Fattal, Lischinski *et al.* (2008); Fattal (2009).

High-quality algorithms for image warping and resampling are covered both in the image processing literature (Wolberg 1990; Dodgson 1992; Gomes, Darsa, Costa *et al.* 1999; Szeliski, Winder, and Uyttendaele 2010) and in computer graphics (Williams 1983; Heckbert 1986; Barkans 1997; Akenine-Möller and Haines 2002), where they go under the name of *texture mapping*. Combination of image warping and image blending techniques are used to enable *morphing* between images, which is covered in a series of seminal papers and books (Beier and Neely 1992; Gomes, Darsa, Costa *et al.* 1999).

The regularization approach to computer vision problems was first introduced to the vision community by Poggio, Torre, and Koch (1985) and Terzopoulos (1986a,b, 1988) and continues to be a popular framework for formulating and solving low-level vision problems (Ju, Black, and Jepson 1996; Nielsen, Florack, and Deriche 1997; Nordström 1990; Brox, Bruhn, Papenberg *et al.* 2004; Levin, Lischinski, and Weiss 2008). More detailed mathematical treatment and additional applications can be found in the applied mathematics and statistics literature (Tikhonov and Arsenin 1977; Engl, Hanke, and Neubauer 1996).

The literature on Markov random fields is truly immense, with publications in related fields such as optimization and control theory of which few vision practitioners are even aware. A good guide to the latest techniques is the book edited by Blake, Kohli, and Rother (2010). Other recent articles that contain nice literature reviews or experimental comparisons include (Boykov and Funka-Lea 2006; Szeliski, Zabih, Scharstein *et al.* 2008; Kumar, Veksler, and Torr 2010).

The seminal paper on Markov random fields is the work of Geman and Geman (1984), who introduced this formalism to computer vision researchers and also introduced the notion of *line processes*, additional binary variables that control whether smoothness penalties are enforced or not. Black and Rangarajan (1996) showed how independent line processes

could be replaced with robust pairwise potentials; Boykov, Veksler, and Zabih (2001) developed iterative binary, graph cut algorithms for optimizing multi-label MRFs; Kolmogorov and Zabih (2004) characterized the class of binary energy potentials required for these techniques to work; and Freeman, Pasztor, and Carmichael (2000) popularized the use of loopy belief propagation for MRF inference. Many more additional references can be found in Sections 3.7.2 and 5.5, and Appendix B.5.

3.9 Exercises

Ex 3.1: Color balance Write a simple application to change the color balance of an image by multiplying each color value by a different user-specified constant. If you want to get fancy, you can make this application interactive, with sliders.

1. Do you get different results if you take out the gamma transformation before or after doing the multiplication? Why or why not?
2. Take the same picture with your digital camera using different color balance settings (most cameras control the color balance from one of the menus). Can you recover what the color balance ratios are between the different settings? You may need to put your camera on a tripod and align the images manually or automatically to make this work. Alternatively, use a color checker chart (Figure 10.3b), as discussed in Sections 2.3 and 10.1.1.
3. If you have access to the RAW image for the camera, perform the demosaicing yourself (Section 10.3.1) or downsample the image resolution to get a “true” RGB image. Does your camera perform a simple linear mapping between RAW values and the color-balanced values in a JPEG? Some high-end cameras have a RAW+JPEG mode, which makes this comparison much easier.
4. Can you think of any reason why you might want to perform a color twist (Section 3.1.2) on the images? See also Exercise 2.9 for some related ideas.

Ex 3.2: Compositing and reflections Section 3.1.3 describes the process of compositing an alpha-matted image on top of another. Answer the following questions and optionally validate them experimentally:

1. Most captured images have gamma correction applied to them. Does this invalidate the basic compositing equation (3.8); if so, how should it be fixed?
2. The additive (pure reflection) model may have limitations. What happens if the glass is tinted, especially to a non-gray hue? How about if the glass is dirty or smudged? How could you model wavy glass or other kinds of refractive objects?

Ex 3.3: Blue screen matting Set up a blue or green background, e.g., by buying a large piece of colored posterboard. Take a picture of the empty background, and then of the background with a new object in front of it. *Pull the matte* using the difference between each colored pixel and its assumed corresponding background pixel, using one of the techniques described in Section 3.1.3) or by Smith and Blinn (1996).

Ex 3.4: Difference keying Implement a difference keying algorithm (see Section 3.1.3) (Toyama, Krumm, Brumitt *et al.* 1999), consisting of the following steps:

1. Compute the mean and variance (or median and robust variance) at each pixel in an “empty” video sequence.
2. For each new frame, classify each pixel as foreground or background (set the background pixels to $\text{RGBA}=0$).
3. (Optional) Compute the alpha channel and composite over a new background.
4. (Optional) Clean up the image using morphology (Section 3.3.1), label the connected components (Section 3.3.4), compute their centroids, and track them from frame to frame. Use this to build a “people counter”.

Ex 3.5: Photo effects Write a variety of photo enhancement or effects filters: contrast, solarization (quantization), etc. Which ones are useful (perform sensible corrections) and which ones are more creative (create unusual images)?

Ex 3.6: Histogram equalization Compute the gray level (luminance) histogram for an image and equalize it so that the tones look better (and the image is less sensitive to exposure settings). You may want to use the following steps:

1. Convert the color image to luminance (Section 3.1.2).
2. Compute the histogram, the cumulative distribution, and the compensation transfer function (Section 3.1.4).
3. (Optional) Try to increase the “punch” in the image by ensuring that a certain fraction of pixels (say, 5%) are mapped to pure black and white.
4. (Optional) Limit the local $gain f'(I)$ in the transfer function. One way to do this is to limit $f(I) < \gamma I$ or $f'(I) < \gamma$ while performing the accumulation (3.9), keeping any unaccumulated values “in reserve”. (I’ll let you figure out the exact details.)
5. Compensate the luminance channel through the lookup table and re-generate the color image using color ratios (2.116).
6. (Optional) Color values that are *clipped* in the original image, i.e., have one or more saturated color channels, may appear unnatural when remapped to a non-clipped value. Extend your algorithm to handle this case in some useful way.

Ex 3.7: Local histogram equalization Compute the gray level (luminance) histograms for each patch, but add to vertices based on distance (a spline).

1. Build on Exercise 3.6 (luminance computation).
2. Distribute values (counts) to adjacent vertices (bilinear).
3. Convert to CDF (look-up functions).
4. (Optional) Use low-pass filtering of CDFs.

5. Interpolate adjacent CDFs for final lookup.

Ex 3.8: Padding for neighborhood operations Write down the formulas for computing the padded pixel values $\tilde{f}(i, j)$ as a function of the original pixel values $f(k, l)$ and the image width and height (M, N) for *each* of the padding modes shown in Figure 3.13. For example, for replication (clamping),

$$\begin{aligned}\tilde{f}(i, j) = f(k, l), & \quad k = \max(0, \min(M - 1, i)), \\ & \quad l = \max(0, \min(N - 1, j)),\end{aligned}$$

(Hint: you may want to use the min, max, mod, and absolute value operators in addition to the regular arithmetic operators.)

- Describe in more detail the advantages and disadvantages of these various modes.
- (Optional) Check what your graphics card does by drawing a texture-mapped rectangle where the texture coordinates lie beyond the $[0.0, 1.0]$ range and using different texture clamping modes.

Ex 3.9: Separable filters Implement convolution with a separable kernel. The input should be a grayscale or color image along with the horizontal and vertical kernels. Make sure you support the padding mechanisms developed in the previous exercise. You will need this functionality for some of the later exercises. If you already have access to separable filtering in an image processing package you are using (such as IPL), skip this exercise.

- (Optional) Use Pietro Perona's (1995) technique to approximate convolution as a sum of a number of separable kernels. Let the user specify the number of kernels and report back some sensible metric of the approximation fidelity.

Ex 3.10: Discrete Gaussian filters Discuss the following issues with implementing a discrete Gaussian filter:

- If you just sample the equation of a continuous Gaussian filter at discrete locations, will you get the desired properties, e.g., will the coefficients sum up to 0? Similarly, if you sample a derivative of a Gaussian, do the samples sum up to 0 or have vanishing higher-order moments?
- Would it be preferable to take the original signal, interpolate it with a sinc, blur with a continuous Gaussian, then pre-filter with a sinc before re-sampling? Is there a simpler way to do this in the frequency domain?
- Would it make more sense to produce a Gaussian frequency response in the Fourier domain and to then take an inverse FFT to obtain a discrete filter?
- How does truncation of the filter change its frequency response? Does it introduce any additional artifacts?
- Are the resulting two-dimensional filters as rotationally invariant as their continuous analogs? Is there some way to improve this? In fact, can any 2D discrete (separable or non-separable) filter be truly rotationally invariant?

Ex 3.11: Sharpening, blur, and noise removal Implement some softening, sharpening, and non-linear diffusion (selective sharpening or noise removal) filters, such as Gaussian, median, and bilateral (Section 3.3.1), as discussed in Section 3.4.4.

Take blurry or noisy images (shooting in low light is a good way to get both) and try to improve their appearance and legibility.

Ex 3.12: Steerable filters Implement Freeman and Adelson's (1991) steerable filter algorithm. The input should be a grayscale or color image and the output should be a multi-banded image consisting of $G_1^{0^\circ}$ and $G_1^{90^\circ}$. The coefficients for the filters can be found in the paper by Freeman and Adelson (1991).

Test the various order filters on a number of images of your choice and see if you can reliably find corner and intersection features. These filters will be quite useful later to detect elongated structures, such as lines (Section 4.3).

Ex 3.13: Distance transform Implement some (raster-scan) algorithms for city block and Euclidean distance transforms. Can you do it without peeking at the literature (Danielsson 1980; Borgefors 1986)? If so, what problems did you come across and resolve?

Later on, you can use the distance functions you compute to perform *feathering* during image stitching (Section 9.3.2).

Ex 3.14: Connected components Implement one of the connected component algorithms from Section 3.3.4 or Section 2.3 from Haralick and Shapiro's book (1992) and discuss its computational complexity.

- Threshold or quantize an image to obtain a variety of input labels and then compute the area statistics for the regions that you find.
- Use the connected components that you have found to track or match regions in different images or video frames.

Ex 3.15: Fourier transform Prove the properties of the Fourier transform listed in Table 3.1 and derive the formulas for the Fourier transforms listed in Tables 3.2 and 3.3. These exercises are very useful if you want to become comfortable working with Fourier transforms, which is a very useful skill when analyzing and designing the behavior and efficiency of many computer vision algorithms.

Ex 3.16: Wiener filtering Estimate the frequency spectrum of your personal photo collection and use it to perform Wiener filtering on a few images with varying degrees of noise.

1. Collect a few hundred of your images by re-scaling them to fit within a 512×512 window and cropping them.
2. Take their Fourier transforms, throw away the phase information, and average together all of the spectra.
3. Pick two of your favorite images and add varying amounts of Gaussian noise, $\sigma_n \in \{1, 2, 5, 10, 20\}$ gray levels.
4. For each combination of image and noise, determine by eye which width of a Gaussian blurring filter σ_s gives the best denoised result. You will have to make a subjective decision between sharpness and noise.

5. Compute the Wiener filtered version of all the noised images and compare them against your hand-tuned Gaussian-smoothed images.
6. (Optional) Do your image spectra have a lot of energy concentrated along the horizontal and vertical axes ($f_x = 0$ and $f_y = 0$)? Can you think of an explanation for this? Does rotating your image samples by 45° move this energy to the diagonals? If not, could it be due to edge effects in the Fourier transform? Can you suggest some techniques for reducing such effects?

Ex 3.17: Deblurring using Wiener filtering Use Wiener filtering to deblur some images.

1. Modify the Wiener filter derivation (3.66–3.74) to incorporate blur (3.75).
2. Discuss the resulting Wiener filter in terms of its noise suppression and frequency boosting characteristics.
3. Assuming that the blur kernel is Gaussian and the image spectrum follows an inverse frequency law, compute the frequency response of the Wiener filter, and compare it to the unsharp mask.
4. Synthetically blur two of your sample images with Gaussian blur kernels of different radii, add noise, and then perform Wiener filtering.
5. Repeat the above experiment with a “pillbox” (disc) blurring kernel, which is characteristic of a finite aperture lens (Section 2.2.3). Compare these results to Gaussian blur kernels (be sure to inspect your frequency plots).
6. It has been suggested that regular apertures are anathema to de-blurring because they introduce zeros in the sensed frequency spectrum (Veeraraghavan, Raskar, Agrawal *et al.* 2007). Show that this is indeed an issue if no prior model is assumed for the signal, i.e., $P_s^{-1}l1$. If a reasonable power spectrum is assumed, is this still a problem (do we still get banding or ringing artifacts)?

Ex 3.18: High-quality image resampling Implement several of the low-pass filters presented in Section 3.5.2 and also the discussion of the windowed sinc shown in Table 3.2 and Figure 3.29. Feel free to implement other filters (Wolberg 1990; Unser 1999).

Apply your filters to continuously resize an image, both magnifying (interpolating) and minifying (decimating) it; compare the resulting animations for several filters. Use both a synthetic chirp image (Figure 3.65a) and natural images with lots of high-frequency detail (Figure 3.65b-c).²⁷

You may find it helpful to write a simple visualization program that continuously plays the animations for two or more filters at once and that let you “blink” between different results.

Discuss the merits and deficiencies of each filter, as well as its tradeoff between speed and quality.

Ex 3.19: Pyramids Construct an image pyramid. The inputs should be a grayscale or color image, a separable filter kernel, and the number of desired levels. Implement at least the following kernels:

²⁷ These particular images are available on the book’s Web site.

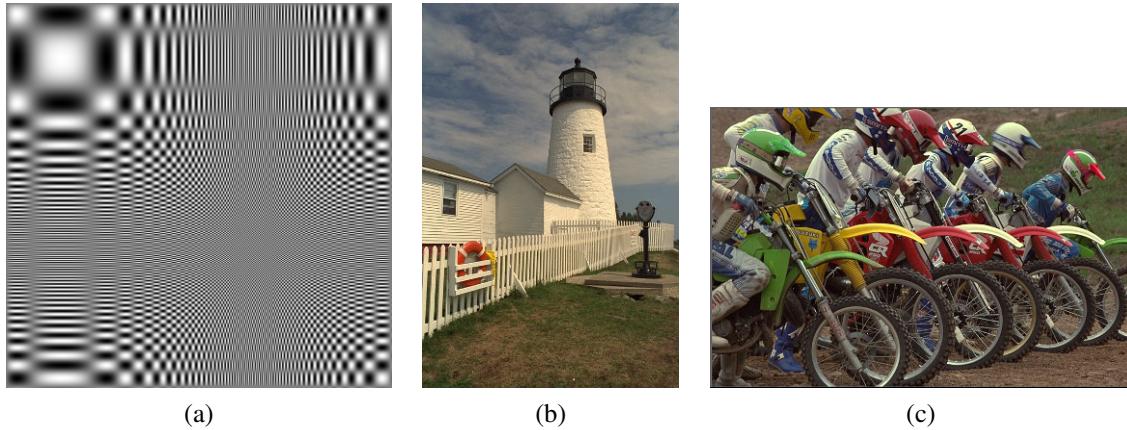


Figure 3.65 Sample images for testing the quality of resampling algorithms: (a) a synthetic chirp; (b) and (c) some high-frequency images from the image compression community.

- 2×2 block filtering;
- Burt and Adelson's binomial kernel $\frac{1}{16}(1, 4, 6, 4, 1)$ (Burt and Adelson 1983a);
- a high-quality seven- or nine-tap filter.

Compare the visual quality of the various decimation filters. Also, shift your input image by 1 to 4 pixels and compare the resulting decimated (quarter size) image sequence.

Ex 3.20: Pyramid blending Write a program that takes as input two color images and a binary mask image and produces the Laplacian pyramid blend of the two images.

1. Construct the Laplacian pyramid for each image.
2. Construct the Gaussian pyramid for the two mask images (the input image and its complement).
3. Multiply each Laplacian image by its corresponding mask and sum the images (see Figure 3.43).
4. Reconstruct the final image from the blended Laplacian pyramid.

Generalize your algorithm to input n images and a label image with values $1 \dots n$ (the value 0 can be reserved for “no input”). Discuss whether the weighted summation stage (step 3) needs to keep track of the total weight for renormalization, or whether the math just works out. Use your algorithm either to blend two differently exposed image (to avoid under- and over-exposed regions) or to make a creative blend of two different scenes.

Ex 3.21: Wavelet construction and applications Implement one of the wavelet families described in Section 3.5.4 or by Simoncelli and Adelson (1990b), as well as the basic Laplacian pyramid (Exercise 3.19). Apply the resulting representations to one of the following two tasks:

- **Compression:** Compute the entropy in each band for the different wavelet implementations, assuming a given quantization level (say, $\frac{1}{4}$ gray level, to keep the rounding error acceptable). Quantize the wavelet coefficients and reconstruct the original images. Which technique performs better? (See (Simoncelli and Adelson 1990b) or any of the multitude of wavelet compression papers for some typical results.)
- **Denoising.** After computing the wavelets, suppress small values using *coring*, i.e., set small values to zero using a piecewise linear or other C^0 function. Compare the results of your denoising using different wavelet and pyramid representations.

Ex 3.22: Parametric image warping Write the code to do affine and perspective image warps (optionally bilinear as well). Try a variety of interpolants and report on their visual quality. In particular, discuss the following:

- In a MIP-map, selecting only the coarser level adjacent to the computed fractional level will produce a blurrier image, while selecting the finer level will lead to aliasing. Explain why this is so and discuss whether blending an aliased and a blurred image (tri-linear MIP-mapping) is a good idea.
- When the ratio of the horizontal and vertical resampling rates becomes very different (anisotropic), the MIP-map performs even worse. Suggest some approaches to reduce such problems.

Ex 3.23: Local image warping Open an image and deform its appearance in one of the following ways:

1. Click on a number of pixels and move (drag) them to new locations. Interpolate the resulting sparse displacement field to obtain a dense motion field (Sections 3.6.2 and 3.5.1).
2. Draw a number of lines in the image. Move the endpoints of the lines to specify their new positions and use the Beier–Neely interpolation algorithm (Beier and Neely 1992), discussed in Section 3.6.2, to get a dense motion field.
3. Overlay a spline control grid and move one grid point at a time (optionally select the level of the deformation).
4. Have a dense per-pixel flow field and use a soft “paintbrush” to design a horizontal and vertical velocity field.
5. (Optional): Prove whether the Beier–Neely warp does or does not reduce to a sparse point-based deformation as the line segments become shorter (reduce to points).

Ex 3.24: Forward warping Given a displacement field from the previous exercise, write a forward warping algorithm:

1. Write a forward warper using splatting, either nearest neighbor or soft accumulation (Section 3.6.1).
2. Write a two-pass algorithm, which forward warps the displacement field, fills in small holes, and then uses inverse warping (Shade, Gortler, He *et al.* 1998).

3. Compare the quality of these two algorithms.

Ex 3.25: Feature-based morphing Extend the warping code you wrote in Exercise 3.23 to import two different images and specify correspondences (point, line, or mesh-based) between the two images.

1. Create a morph by partially warping the images towards each other and cross-dissolving (Section 3.6.3).
2. Try using your morphing algorithm to perform an image rotation and discuss whether it behaves the way you want it to.

Ex 3.26: 2D image editor Extend the program you wrote in Exercise 2.2 to import images and let you create a “collage” of pictures. You should implement the following steps:

1. Open up a new image (in a separate window).
2. Shift drag (rubber-band) to crop a subregion (or select whole image).
3. Paste into the current canvas.
4. Select the deformation mode (motion model): translation, rigid, similarity, affine, or perspective.
5. Drag any corner of the outline to change its transformation.
6. (Optional) Change the relative ordering of the images and which image is currently being manipulated.

The user should see the composition of the various images’ pieces on top of each other.

This exercise should be built on the image transformation classes supported in the software library. Persistence of the created representation (save and load) should also be supported (for each image, save its transformation).

Ex 3.27: 3D texture-mapped viewer Extend the viewer you created in Exercise 2.3 to include texture-mapped polygon rendering. Augment each polygon with (u, v, w) coordinates into an image.

Ex 3.28: Image denoising Implement at least two of the various image denoising techniques described in this chapter and compare them on both synthetically noised image sequences and real-world (low-light) sequences. Does the performance of the algorithm depend on the correct choice of noise level estimate? Can you draw any conclusions as to which techniques work better?

Ex 3.29: Rainbow enhancer—challenging Take a picture containing a rainbow, such as Figure 3.66, and enhance the strength (saturation) of the rainbow.

1. Draw an arc in the image delineating the extent of the rainbow.

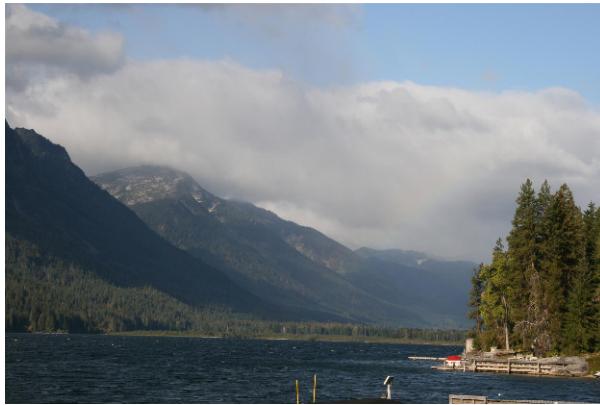


Figure 3.66 There is a faint image of a rainbow visible in the right hand side of this picture. Can you think of a way to enhance it (Exercise 3.29)?

2. Fit an *additive* rainbow function (explain why it is additive) to this arc (it is best to work with linearized pixel values), using the spectrum as the cross section, and estimating the width of the arc and the amount of color being added. This is the trickiest part of the problem, as you need to tease apart the (low-frequency) rainbow pattern and the natural image hiding behind it.
3. Amplify the rainbow signal and add it back into the image, re-applying the gamma function if necessary to produce the final image.

Ex 3.30: Image deblocking—challenging Now that you have some good techniques to distinguish signal from noise, develop a technique to remove the *blocking artifacts* that occur with JPEG at high compression settings (Section 2.3.3). Your technique can be as simple as looking for unexpected edges along block boundaries, to looking at the quantization step as a projection of a convex region of the transform coefficient space onto the corresponding quantized values.

1. Does the knowledge of the compression factor, which is available in the JPEG header information, help you perform better deblocking?
2. Because the quantization occurs in the DCT transformed YCbCr space (2.115), it may be preferable to perform the analysis in this space. On the other hand, image priors make more sense in an RGB space (or do they?). Decide how you will approach this dichotomy and discuss your choice.
3. While you are at it, since the YCbCr conversion is followed by a chrominance subsampling stage (before the DCT), see if you can restore some of the lost high-frequency chrominance signal using one of the better restoration techniques discussed in this chapter.
4. If your camera has a RAW + JPEG mode, how close can you come to the noise-free true pixel values? (This suggestion may not be that useful, since cameras generally use reasonably high quality settings for their RAW + JPEG models.)

Ex 3.31: Inference in de-blurring—challenging Write down the graphical model corresponding to Figure 3.59 for a non-blind image deblurring problem, i.e., one where the blur kernel is known ahead of time.

What kind of efficient inference (optimization) algorithms can you think of for solving such problems?

Chapter 4

Feature detection and matching

4.1	Points and patches	183
4.1.1	Feature detectors	185
4.1.2	Feature descriptors	196
4.1.3	Feature matching	200
4.1.4	Feature tracking	207
4.1.5	<i>Application:</i> Performance-driven animation	209
4.2	Edges	210
4.2.1	Edge detection	210
4.2.2	Edge linking	215
4.2.3	<i>Application:</i> Edge editing and enhancement	219
4.3	Lines	220
4.3.1	Successive approximation	220
4.3.2	Hough transforms	221
4.3.3	Vanishing points	224
4.3.4	<i>Application:</i> Rectangle detection	226
4.4	Additional reading	227
4.5	Exercises	228

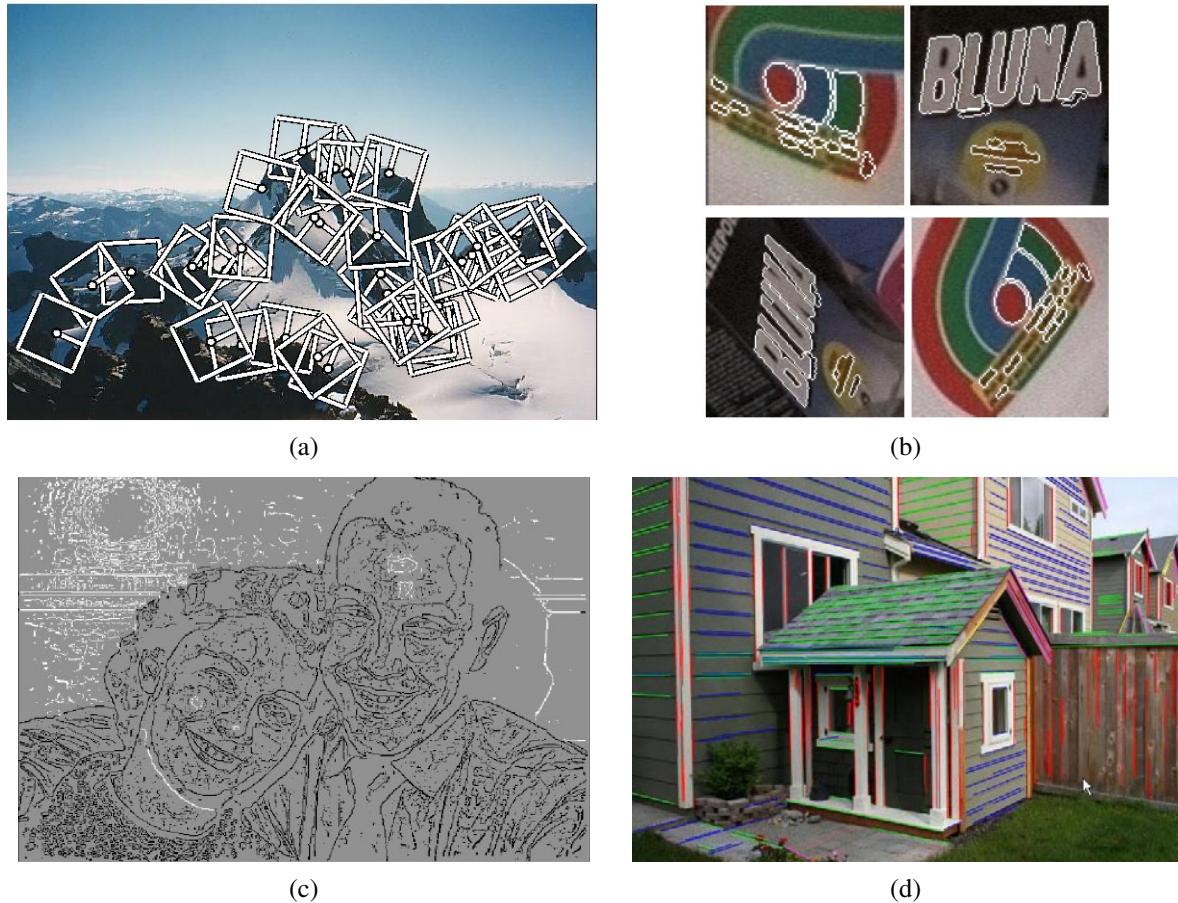


Figure 4.1 A variety of feature detectors and descriptors can be used to analyze, describe and match images: (a) point-like interest operators (Brown, Szeliski, and Winder 2005) © 2005 IEEE; (b) region-like interest operators (Matas, Chum, Urban *et al.* 2004) © 2004 Elsevier; (c) edges (Elder and Goldberg 2001) © 2001 IEEE; (d) straight lines (Sinha, Steedly, Szeliski *et al.* 2008) © 2008 ACM.

Feature detection and matching are an essential component of many computer vision applications. Consider the two pairs of images shown in Figure 4.2. For the first pair, we may wish to *align* the two images so that they can be seamlessly stitched into a composite mosaic (Chapter 9). For the second pair, we may wish to establish a dense set of *correspondences* so that a 3D model can be constructed or an in-between view can be generated (Chapter 11). In either case, what kinds of *features* should you detect and then match in order to establish such an alignment or set of correspondences? Think about this for a few moments before reading on.

The first kind of feature that you may notice are specific locations in the images, such as mountain peaks, building corners, doorways, or interestingly shaped patches of snow. These kinds of localized feature are often called *keypoint features* or *interest points* (or even *corners*) and are often described by the appearance of patches of pixels surrounding the point location (Section 4.1). Another class of important features are *edges*, e.g., the profile of mountains against the sky, (Section 4.2). These kinds of features can be matched based on their orientation and local appearance (edge profiles) and can also be good indicators of object boundaries and *occlusion* events in image sequences. Edges can be grouped into longer *curves* and *straight line segments*, which can be directly matched or analyzed to find *vanishing points* and hence internal and external camera parameters (Section 4.3).

In this chapter, we describe some practical approaches to detecting such features and also discuss how feature correspondences can be established across different images. Point features are now used in such a wide variety of applications that it is good practice to read and implement some of the algorithms from (Section 4.1). Edges and lines provide information that is complementary to both keypoint and region-based descriptors and are well-suited to describing object boundaries and man-made objects. These alternative descriptors, while extremely useful, can be skipped in a short introductory course.

4.1 Points and patches

Point features can be used to find a sparse set of corresponding locations in different images, often as a pre-cursor to computing camera pose (Chapter 7), which is a prerequisite for computing a denser set of correspondences using stereo matching (Chapter 11). Such correspondences can also be used to align different images, e.g., when stitching image mosaics or performing video stabilization (Chapter 9). They are also used extensively to perform object instance and category recognition (Sections 14.3 and 14.4). A key advantage of keypoints is that they permit matching even in the presence of clutter (occlusion) and large scale and orientation changes.

Feature-based correspondence techniques have been used since the early days of stereo matching (Hannah 1974; Moravec 1983; Hannah 1988) and have more recently gained popularity for image-stitching applications (Zoghliami, Faugeras, and Deriche 1997; Brown and Lowe 2007) as well as fully automated 3D modeling (Beardsley, Torr, and Zisserman 1996; Schaffalitzky and Zisserman 2002; Brown and Lowe 2003; Snavely, Seitz, and Szeliski 2006).

There are two main approaches to finding feature points and their correspondences. The first is to find features in one image that can be accurately *tracked* using a local search technique, such as correlation or least squares (Section 4.1.4). The second is to independently



Figure 4.2 Two pairs of images to be matched. What kinds of feature might one use to establish a set of *correspondences* between these images?

detect features in all the images under consideration and then *match* features based on their local appearance (Section 4.1.3). The former approach is more suitable when images are taken from nearby viewpoints or in rapid succession (e.g., video sequences), while the latter is more suitable when a large amount of motion or appearance change is expected, e.g., in stitching together panoramas (Brown and Lowe 2007), establishing correspondences in *wide baseline stereo* (Schaffalitzky and Zisserman 2002), or performing object recognition (Fergus, Perona, and Zisserman 2007).

In this section, we split the keypoint detection and matching pipeline into four separate stages. During the *feature detection* (extraction) stage (Section 4.1.1), each image is searched for locations that are likely to match well in other images. At the *feature description* stage (Section 4.1.2), each region around detected keypoint locations is converted into a more compact and stable (invariant) *descriptor* that can be matched against other descriptors. The *feature matching* stage (Section 4.1.3) efficiently searches for likely matching candidates in other images. The *feature tracking* stage (Section 4.1.4) is an alternative to the third stage that only searches a small neighborhood around each detected feature and is therefore more suitable for video processing.

A wonderful example of all of these stages can be found in David Lowe's (2004) paper, which describes the development and refinement of his *Scale Invariant Feature Transform* (SIFT). Comprehensive descriptions of alternative techniques can be found in a series of survey and evaluation papers covering both feature detection (Schmid, Mohr, and Bauckhage 2000; Mikolajczyk, Tuytelaars, Schmid *et al.* 2005; Tuytelaars and Mikolajczyk 2007) and feature descriptors (Mikolajczyk and Schmid 2005). Shi and Tomasi (1994) and Triggs (2004) also provide nice reviews of feature detection techniques.

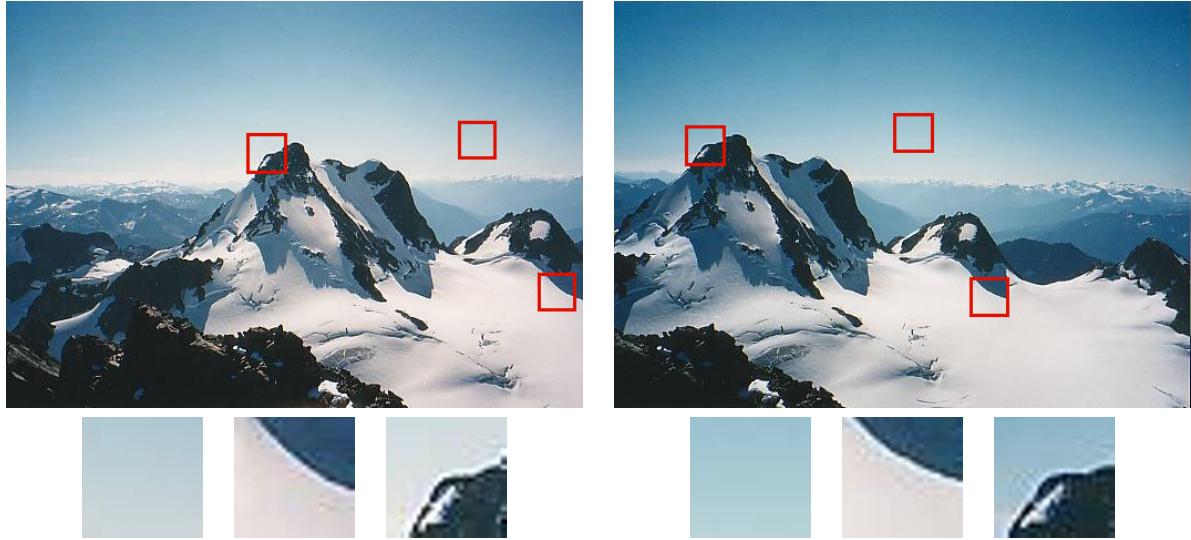


Figure 4.3 Image pairs with extracted patches below. Notice how some patches can be localized or matched with higher accuracy than others.

4.1.1 Feature detectors

How can we find image locations where we can reliably find correspondences with other images, i.e., what are good features to track (Shi and Tomasi 1994; Triggs 2004)? Look again at the image pair shown in Figure 4.3 and at the three sample *patches* to see how well they might be matched or tracked. As you may notice, textureless patches are nearly impossible to localize. Patches with large contrast changes (gradients) are easier to localize, although straight line segments at a single orientation suffer from the *aperture problem* (Horn and Schunck 1981; Lucas and Kanade 1981; Anandan 1989), i.e., it is only possible to align the patches along the direction *normal* to the edge direction (Figure 4.4b). Patches with gradients in at least two (significantly) different orientations are the easiest to localize, as shown schematically in Figure 4.4a.

These intuitions can be formalized by looking at the simplest possible matching criterion for comparing two image patches, i.e., their (weighted) summed square difference,

$$EWSSD(\mathbf{u}) = \sum_i w(\mathbf{x}_i)[I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)]^2, \quad (4.1)$$

where I_0 and I_1 are the two images being compared, $\mathbf{u} = (u, v)$ is the *displacement* vector, $w(\mathbf{x})$ is a spatially varying weighting (or window) function, and the summation i is over all the pixels in the patch. Note that this is the same formulation we later use to estimate motion between complete images (Section 8.1).

When performing feature detection, we do not know which other image locations the feature will end up being matched against. Therefore, we can only compute how stable this metric is with respect to small variations in position $\Delta\mathbf{u}$ by comparing an image patch against

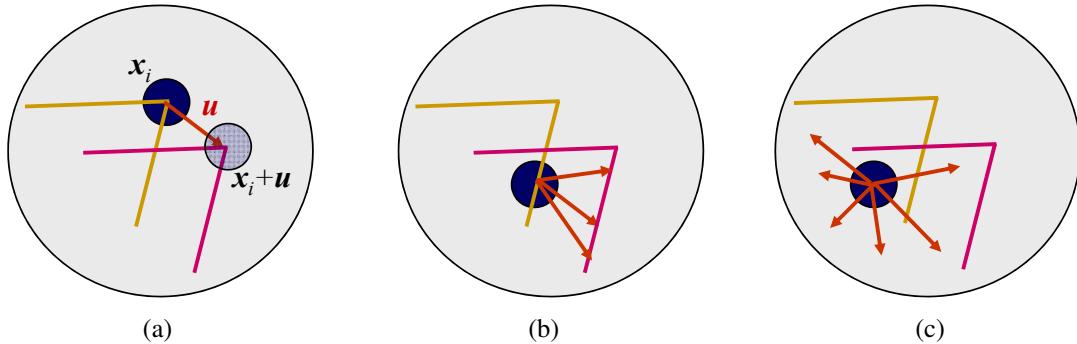


Figure 4.4 Aperture problems for different image patches: (a) stable (“corner-like”) flow; (b) classic aperture problem (barber-pole illusion); (c) textureless region. The two images I_0 (yellow) and I_1 (red) are overlaid. The red vector \mathbf{u} indicates the displacement between the patch centers and the $w(\mathbf{x}_i)$ weighting function (patch window) is shown as a dark circle.

itself, which is known as an *auto-correlation function or surface*

$$E_{AC}(\Delta\mathbf{u}) = \sum_i w(\mathbf{x}_i)[I_0(\mathbf{x}_i + \Delta\mathbf{u}) - I_0(\mathbf{x}_i)]^2 \quad (4.2)$$

(Figure 4.5).¹ Note how the auto-correlation surface for the textured flower bed (Figure 4.5b and the red cross in the lower right quadrant of Figure 4.5a) exhibits a strong minimum, indicating that it can be well localized. The correlation surface corresponding to the roof edge (Figure 4.5c) has a strong ambiguity along one direction, while the correlation surface corresponding to the cloud region (Figure 4.5d) has no stable minimum.

Using a Taylor Series expansion of the image function $I_0(\mathbf{x}_i + \Delta\mathbf{u}) \approx I_0(\mathbf{x}_i) + \nabla I_0(\mathbf{x}_i) \cdot \Delta\mathbf{u}$ (Lucas and Kanade 1981; Shi and Tomasi 1994), we can approximate the auto-correlation surface as

$$E_{AC}(\Delta\mathbf{u}) = \sum_i w(\mathbf{x}_i)[I_0(\mathbf{x}_i + \Delta\mathbf{u}) - I_0(\mathbf{x}_i)]^2 \quad (4.3)$$

$$\approx \sum_i w(\mathbf{x}_i)[I_0(\mathbf{x}_i) + \nabla I_0(\mathbf{x}_i) \cdot \Delta\mathbf{u} - I_0(\mathbf{x}_i)]^2 \quad (4.4)$$

$$= \sum_i w(\mathbf{x}_i)[\nabla I_0(\mathbf{x}_i) \cdot \Delta\mathbf{u}]^2 \quad (4.5)$$

$$= \Delta\mathbf{u}^T \mathbf{A} \Delta\mathbf{u}, \quad (4.6)$$

where

$$\nabla I_0(\mathbf{x}_i) = \left(\frac{\partial I_0}{\partial x}, \frac{\partial I_0}{\partial y} \right)(\mathbf{x}_i) \quad (4.7)$$

is the *image gradient* at \mathbf{x}_i . This gradient can be computed using a variety of techniques (Schmid, Mohr, and Bauckhage 2000). The classic “Harris” detector (Harris and Stephens 1988) uses a [-2 -1 0 1 2] filter, but more modern variants (Schmid, Mohr, and Bauckhage 2000; Triggs 2004) convolve the image with horizontal and vertical derivatives of a Gaussian (typically with $\sigma = 1$).

¹ Strictly speaking, a correlation is the *product* of two patches (3.12); I’m using the term here in a more qualitative sense. The weighted sum of squared differences is often called an *SSD surface* (Section 8.1).

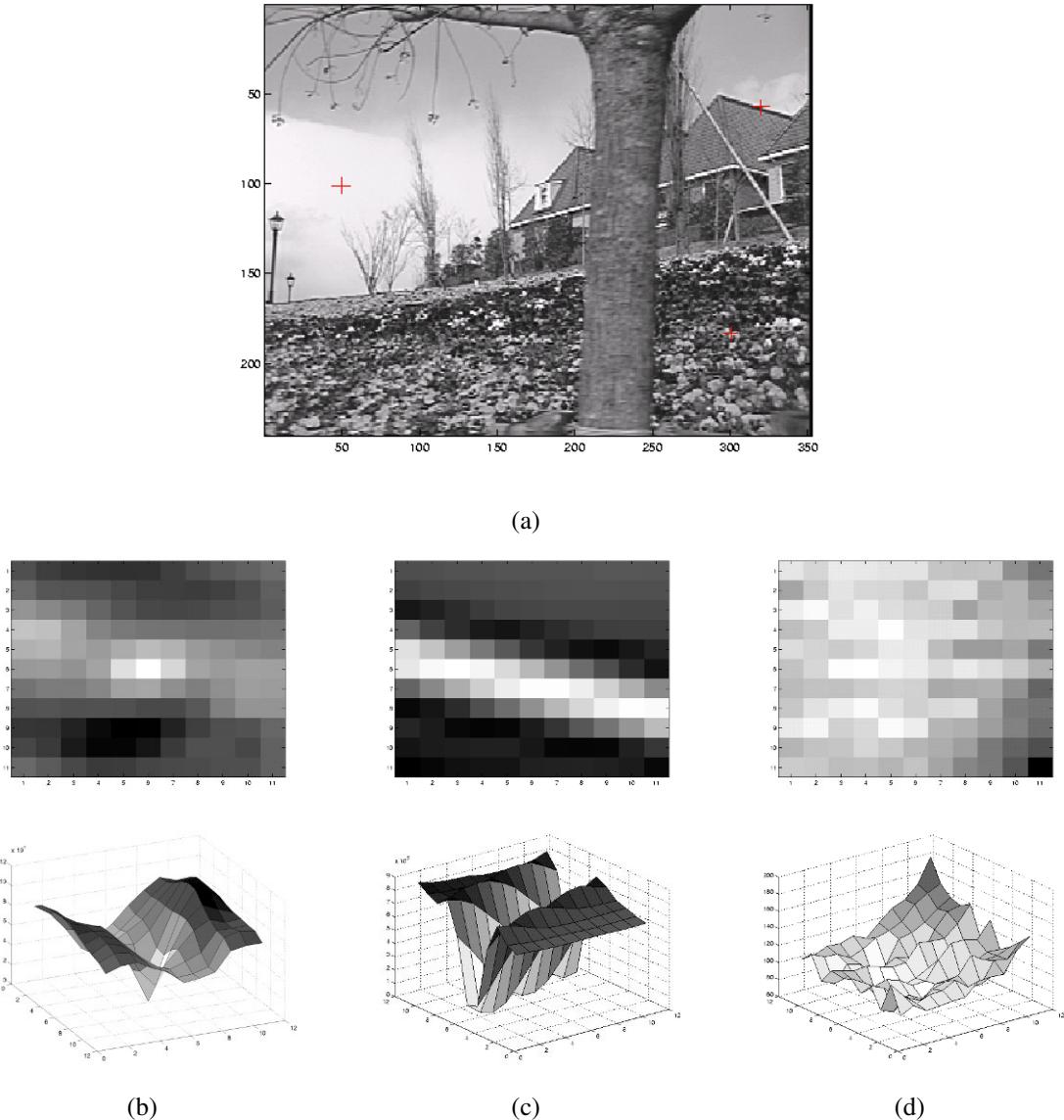


Figure 4.5 Three auto-correlation surfaces $E_{AC}(\Delta u)$ shown as both grayscale images and surface plots: (a) The original image is marked with three red crosses to denote where the auto-correlation surfaces were computed; (b) this patch is from the flower bed (good unique minimum); (c) this patch is from the roof edge (one-dimensional aperture problem); and (d) this patch is from the cloud (no good peak). Each grid point in figures b–d is one value of Δu .

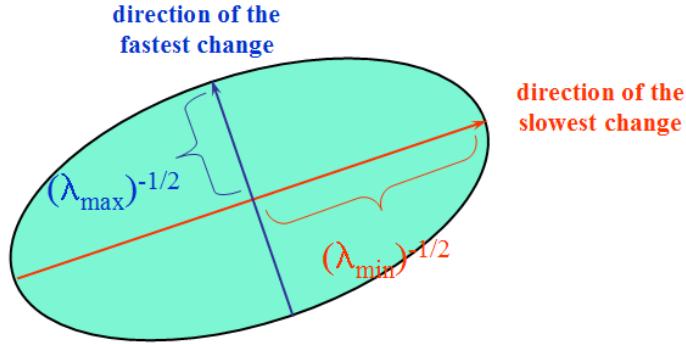


Figure 4.6 Uncertainty ellipse corresponding to an eigenvalue analysis of the auto-correlation matrix \mathbf{A} .

The auto-correlation matrix \mathbf{A} can be written as

$$\mathbf{A} = w * \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \quad (4.8)$$

where we have replaced the weighted summations with discrete convolutions with the weighting kernel w . This matrix can be interpreted as a tensor (multiband) image, where the outer products of the gradients ∇I are convolved with a weighting function w to provide a per-pixel estimate of the local (quadratic) shape of the auto-correlation function.

As first shown by Anandan (1984; 1989) and further discussed in Section 8.1.3 and (8.44), the inverse of the matrix \mathbf{A} provides a lower bound on the uncertainty in the location of a matching patch. It is therefore a useful indicator of which patches can be reliably matched. The easiest way to visualize and reason about this uncertainty is to perform an eigenvalue analysis of the auto-correlation matrix \mathbf{A} , which produces two eigenvalues (λ_0, λ_1) and two eigenvector directions (Figure 4.6). Since the larger uncertainty depends on the smaller eigenvalue, i.e., $\lambda_0^{-1/2}$, it makes sense to find maxima in the smaller eigenvalue to locate good features to track (Shi and Tomasi 1994).

Förstner–Harris. While Anandan and Lucas and Kanade (1981) were the first to analyze the uncertainty structure of the auto-correlation matrix, they did so in the context of associating certainties with optic flow measurements. Förstner (1986) and Harris and Stephens (1988) were the first to propose using local maxima in rotationally invariant scalar measures derived from the auto-correlation matrix to locate keypoints for the purpose of sparse feature matching. (Schmid, Mohr, and Bauckhage (2000); Triggs (2004) give more detailed historical reviews of feature detection algorithms.) Both of these techniques also proposed using a Gaussian weighting window instead of the previously used square patches, which makes the detector response insensitive to in-plane image rotations.

The minimum eigenvalue λ_0 (Shi and Tomasi 1994) is not the only quantity that can be used to find keypoints. A simpler quantity, proposed by Harris and Stephens (1988), is

$$\det(\mathbf{A}) - \alpha \operatorname{trace}(\mathbf{A})^2 = \lambda_0 \lambda_1 - \alpha(\lambda_0 + \lambda_1)^2 \quad (4.9)$$

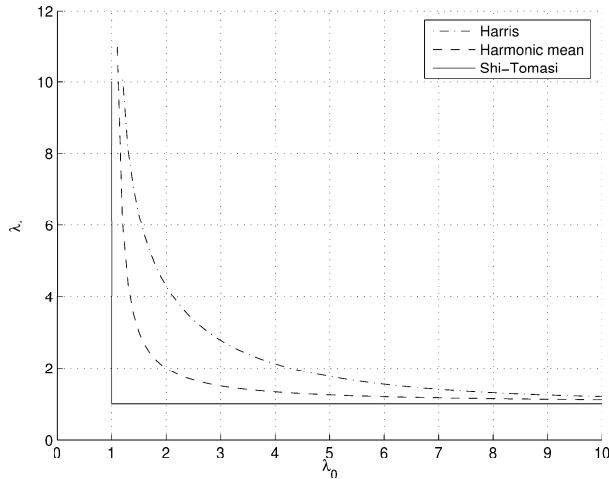


Figure 4.7 Isocontours of popular keypoint detection functions (Brown, Szeliski, and Winder 2004). Each detector looks for points where the eigenvalues λ_0, λ_1 of $\mathbf{A} = w * \nabla I \nabla I^T$ are both large.

with $\alpha = 0.06$. Unlike eigenvalue analysis, this quantity does not require the use of square roots and yet is still rotationally invariant and also downweights edge-like features where $\lambda_1 \gg \lambda_0$. Triggs (2004) suggests using the quantity

$$\lambda_0 - \alpha \lambda_1 \quad (4.10)$$

(say, with $\alpha = 0.05$), which also reduces the response at 1D edges, where aliasing errors sometimes inflate the smaller eigenvalue. He also shows how the basic 2×2 Hessian can be extended to parametric motions to detect points that are also accurately localizable in scale and rotation. Brown, Szeliski, and Winder (2005), on the other hand, use the harmonic mean,

$$\frac{\det \mathbf{A}}{\text{tr } \mathbf{A}} = \frac{\lambda_0 \lambda_1}{\lambda_0 + \lambda_1}, \quad (4.11)$$

which is a smoother function in the region where $\lambda_0 \approx \lambda_1$. Figure 4.7 shows isocontours of the various interest point operators, from which we can see how the two eigenvalues are blended to determine the final interest value.

The steps in the basic auto-correlation-based keypoint detector are summarized in Algorithm 4.1. Figure 4.8 shows the resulting interest operator responses for the classic Harris detector as well as the difference of Gaussian (DoG) detector discussed below.

Adaptive non-maximal suppression (ANMS). While most feature detectors simply look for local maxima in the interest function, this can lead to an uneven distribution of feature points across the image, e.g., points will be denser in regions of higher contrast. To mitigate this problem, Brown, Szeliski, and Winder (2005) only detect features that are both local maxima and whose response value is significantly (10%) greater than that of all of its neighbors within a radius r (Figure 4.9c-d). They devise an efficient way to associate suppression radii with all local maxima by first sorting them by their response strength and then creating a second list sorted by decreasing suppression radius (Brown, Szeliski, and

1. Compute the horizontal and vertical derivatives of the image I_x and I_y by convolving the original image with derivatives of Gaussians (Section 3.2.3).
2. Compute the three images corresponding to the outer products of these gradients. (The matrix A is symmetric, so only three entries are needed.)
3. Convolve each of these images with a larger Gaussian.
4. Compute a scalar interest measure using one of the formulas discussed above.
5. Find local maxima above a certain threshold and report them as detected feature point locations.

Algorithm 4.1 Outline of a basic feature detection algorithm.

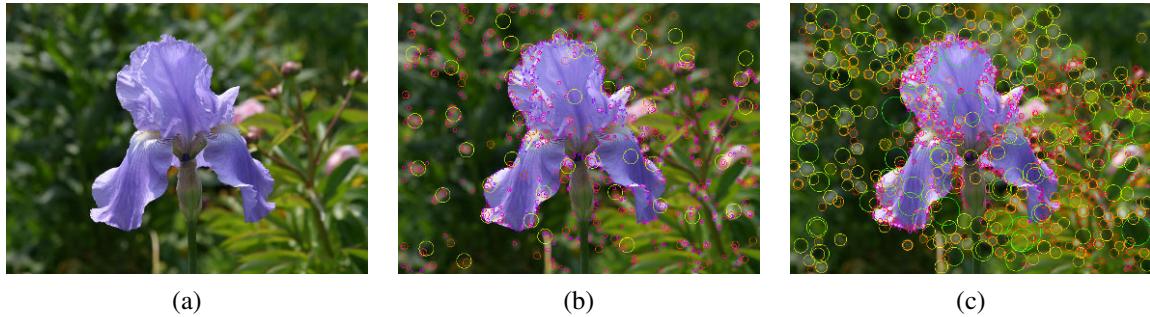


Figure 4.8 Interest operator responses: (a) Sample image, (b) Harris response, and (c) DoG response. The circle sizes and colors indicate the scale at which each interest point was detected. Notice how the two detectors tend to respond at complementary locations.

Winder 2005). Figure 4.9 shows a qualitative comparison of selecting the top n features and using ANMS.

Measuring repeatability. Given the large number of feature detectors that have been developed in computer vision, how can we decide which ones to use? Schmid, Mohr, and Bauckhage (2000) were the first to propose measuring the *repeatability* of feature detectors, which they define as the frequency with which keypoints detected in one image are found within ϵ (say, $\epsilon = 1.5$) pixels of the corresponding location in a transformed image. In their paper, they transform their planar images by applying rotations, scale changes, illumination changes, viewpoint changes, and adding noise. They also measure the *information content* available at each detected feature point, which they define as the entropy of a set of rotationally invariant local grayscale descriptors. Among the techniques they survey, they find that the improved (Gaussian derivative) version of the Harris operator with $\sigma_d = 1$ (scale of the derivative Gaussian) and $\sigma_i = 2$ (scale of the integration Gaussian) works best.

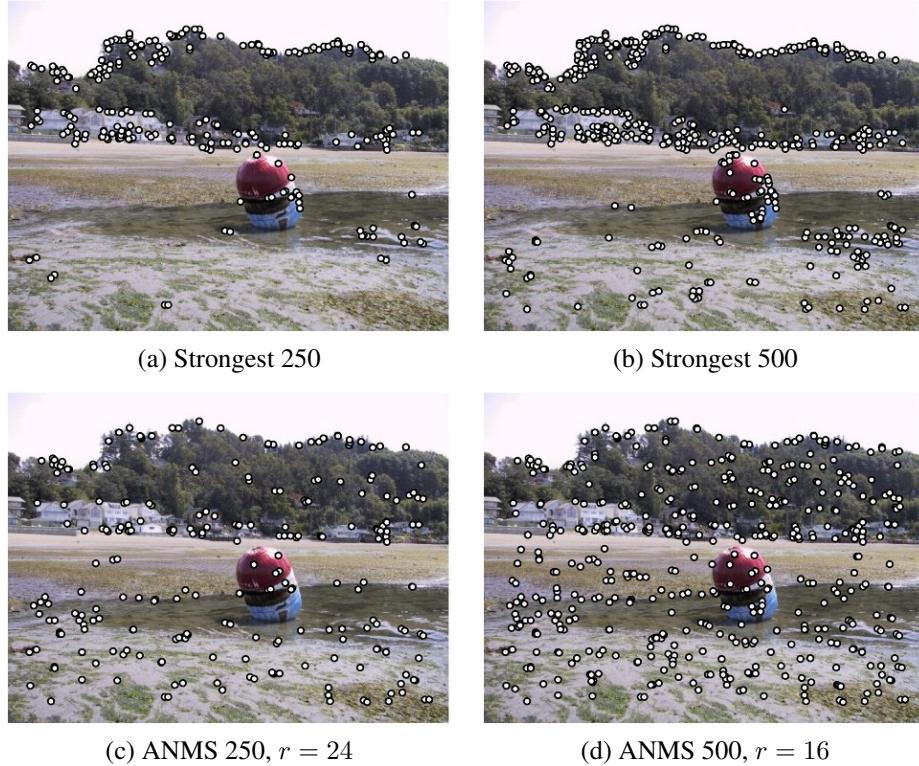


Figure 4.9 Adaptive non-maximal suppression (ANMS) (Brown, Szeliski, and Winder 2005) © 2005 IEEE: The upper two images show the strongest 250 and 500 interest points, while the lower two images show the interest points selected with adaptive non-maximal suppression, along with the corresponding suppression radius r . Note how the latter features have a much more uniform spatial distribution across the image.

Scale invariance

In many situations, detecting features at the finest stable scale possible may not be appropriate. For example, when matching images with little high frequency detail (e.g., clouds), fine-scale features may not exist.

One solution to the problem is to extract features at a variety of scales, e.g., by performing the same operations at multiple resolutions in a pyramid and then matching features at the same level. This kind of approach is suitable when the images being matched do not undergo large scale changes, e.g., when matching successive aerial images taken from an airplane or stitching panoramas taken with a fixed-focal-length camera. Figure 4.10 shows the output of one such approach, the multi-scale, oriented patch detector of Brown, Szeliski, and Winder (2005), for which responses at five different scales are shown.

However, for most object recognition applications, the scale of the object in the image is unknown. Instead of extracting features at many different scales and then matching all of them, it is more efficient to extract features that are stable in both location *and* scale (Lowe 2004; Mikolajczyk and Schmid 2004).

Early investigations into scale selection were performed by Lindeberg (1993; 1998b), who first proposed using extrema in the Laplacian of Gaussian (LoG) function as interest

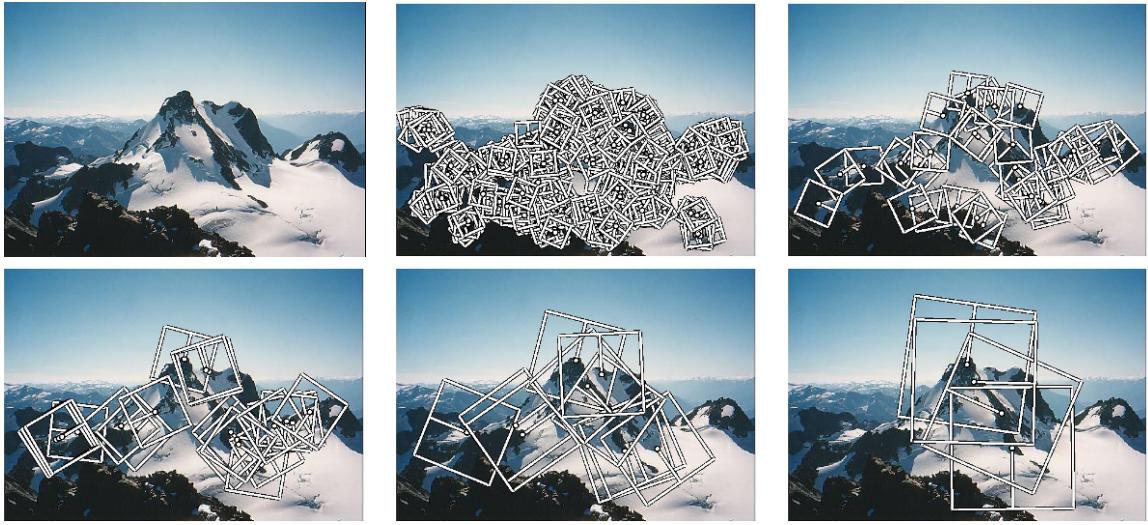


Figure 4.10 Multi-scale oriented patches (MOPS) extracted at five pyramid levels (Brown, Szeliski, and Winder 2005) © 2005 IEEE. The boxes show the feature orientation and the region from which the descriptor vectors are sampled.

point locations. Based on this work, Lowe (2004) proposed computing a set of sub-octave Difference of Gaussian filters (Figure 4.11a), looking for 3D (space+scale) maxima in the resulting structure (Figure 4.11b), and then computing a sub-pixel space+scale location using a quadratic fit (Brown and Lowe 2002). The number of sub-octave levels was determined, after careful empirical investigation, to be three, which corresponds to a quarter-octave pyramid, which is the same as used by Triggs (2004).

As with the Harris operator, pixels where there is strong asymmetry in the local curvature of the indicator function (in this case, the DoG) are rejected. This is implemented by first computing the local Hessian of the difference image D ,

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}, \quad (4.12)$$

and then rejecting keypoints for which

$$\frac{\text{Tr}(\mathbf{H})^2}{\text{Det}(\mathbf{H})} > 10. \quad (4.13)$$

While Lowe's Scale Invariant Feature Transform (SIFT) performs well in practice, it is not based on the same theoretical foundation of maximum spatial stability as the auto-correlation-based detectors. (In fact, its detection locations are often complementary to those produced by such techniques and can therefore be used in conjunction with these other approaches.) In order to add a scale selection mechanism to the Harris corner detector, Mikolajczyk and Schmid (2004) evaluate the Laplacian of Gaussian function at each detected Harris point (in a multi-scale pyramid) and keep only those points for which the Laplacian is extremal (larger or smaller than both its coarser and finer-level values). An optional iterative refinement for both scale and position is also proposed and evaluated. Additional examples of scale invariant

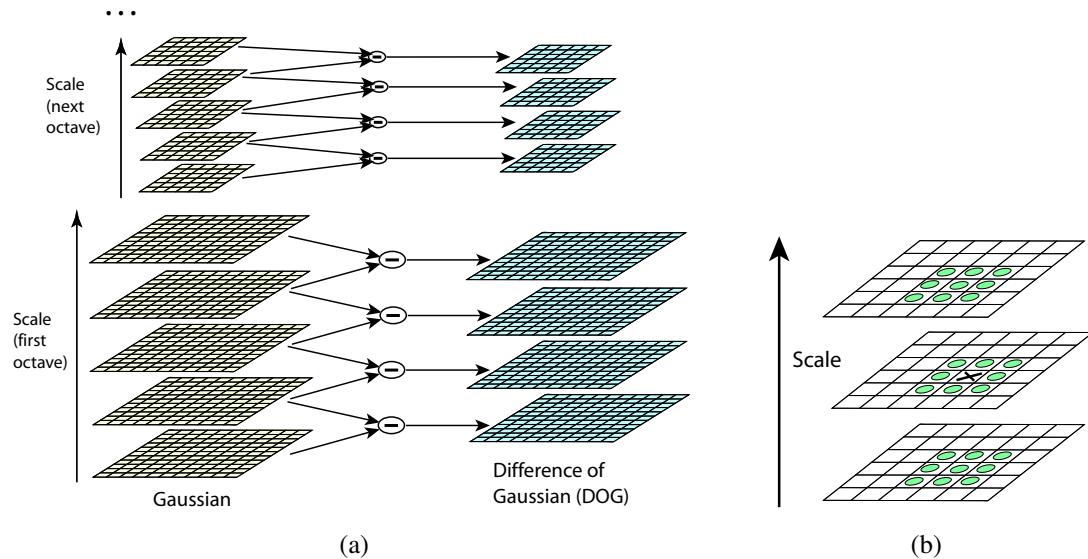


Figure 4.11 Scale-space feature detection using a sub-octave Difference of Gaussian pyramid (Lowe 2004) © 2004 Springer: (a) Adjacent levels of a sub-octave Gaussian pyramid are subtracted to produce Difference of Gaussian images; (b) extrema (maxima and minima) in the resulting 3D volume are detected by comparing a pixel to its 26 neighbors.

region detectors are discussed by Mikolajczyk, Tuytelaars, Schmid *et al.* (2005); Tuytelaars and Mikolajczyk (2007).

Rotational invariance and orientation estimation

In addition to dealing with scale changes, most image matching and object recognition algorithms need to deal with (at least) in-plane image rotation. One way to deal with this problem is to design descriptors that are rotationally invariant (Schmid and Mohr 1997), but such descriptors have poor discriminability, i.e. they map different looking patches to the same descriptor.

A better method is to estimate a *dominant orientation* at each detected keypoint. Once the local orientation and scale of a keypoint have been estimated, a scaled and oriented patch around the detected point can be extracted and used to form a feature descriptor (Figures 4.10 and 4.17).

The simplest possible orientation estimate is the average gradient within a region around the keypoint. If a Gaussian weighting function is used (Brown, Szeliski, and Winder 2005), this average gradient is equivalent to a first-order steerable filter (Section 3.2.3), i.e., it can be computed using an image convolution with the horizontal and vertical derivatives of Gaussian filter (Freeman and Adelson 1991). In order to make this estimate more reliable, it is usually preferable to use a larger aggregation window (Gaussian kernel size) than detection window (Brown, Szeliski, and Winder 2005). The orientations of the square boxes shown in Figure 4.10 were computed using this technique.

Sometimes, however, the averaged (signed) gradient in a region can be small and therefore

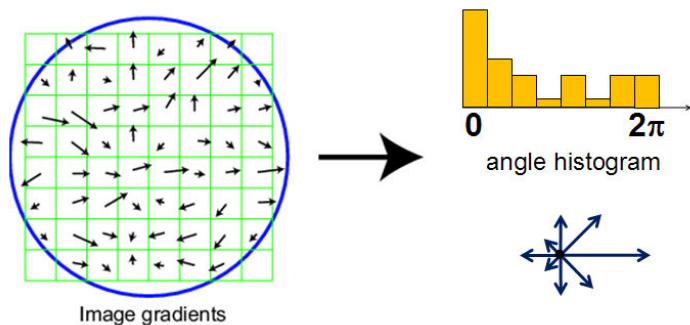


Figure 4.12 A dominant orientation estimate can be computed by creating a histogram of all the gradient orientations (weighted by their magnitudes or after thresholding out small gradients) and then finding the significant peaks in this distribution (Lowe 2004) © 2004 Springer.

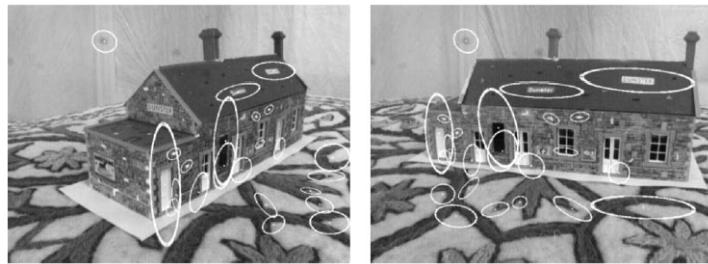


Figure 4.13 Affine region detectors used to match two images taken from dramatically different viewpoints (Mikolajczyk and Schmid 2004) © 2004 Springer.

an unreliable indicator of orientation. A more reliable technique is to look at the *histogram* of orientations computed around the keypoint. Lowe (2004) computes a 36-bin histogram of edge orientations weighted by both gradient magnitude and Gaussian distance to the center, finds all peaks within 80% of the global maximum, and then computes a more accurate orientation estimate using a three-bin parabolic fit (Figure 4.12).

Affine invariance

While scale and rotation invariance are highly desirable, for many applications such as *wide baseline stereo matching* (Pritchett and Zisserman 1998; Schaffalitzky and Zisserman 2002) or location recognition (Chum, Philbin, Sivic *et al.* 2007), full affine invariance is preferred. Affine-invariant detectors not only respond at consistent locations after scale and orientation changes, they also respond consistently across affine deformations such as (local) perspective foreshortening (Figure 4.13). In fact, for a small enough patch, any continuous image warping can be well approximated by an affine deformation.

To introduce affine invariance, several authors have proposed fitting an ellipse to the auto-correlation or Hessian matrix (using eigenvalue analysis) and then using the principal axes and ratios of this fit as the affine coordinate frame (Lindeberg and Garding 1997; Baumberg

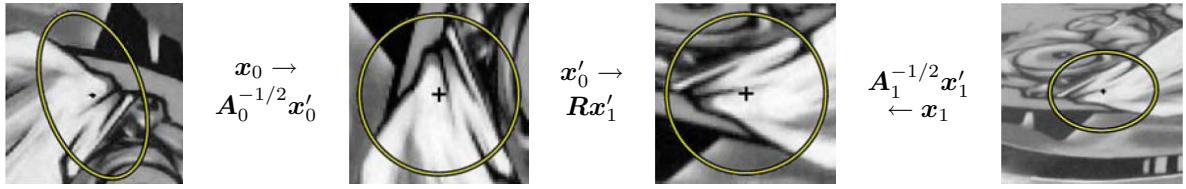


Figure 4.14 Affine normalization using the second moment matrices, as described by Mikolajczyk, Tuytelaars, Schmid *et al.* (2005) © 2005 Springer. After image coordinates are transformed using the matrices $A_0^{-1/2}$ and $A_1^{-1/2}$, they are related by a pure rotation R , which can be estimated using a dominant orientation technique.



Figure 4.15 Maximally stable extremal regions (MSERs) extracted and matched from a number of images (Matas, Chum, Urban *et al.* 2004) © 2004 Elsevier.

2000; Mikolajczyk and Schmid 2004; Mikolajczyk, Tuytelaars, Schmid *et al.* 2005; Tuytelaars and Mikolajczyk 2007). Figure 4.14 shows how the square root of the moment matrix can be used to transform local patches into a frame which is similar up to rotation.

Another important affine invariant region detector is the maximally stable extremal region (MSER) detector developed by Matas, Chum, Urban *et al.* (2004). To detect MSERs, binary regions are computed by thresholding the image at all possible gray levels (the technique therefore only works for grayscale images). This operation can be performed efficiently by first sorting all pixels by gray value and then incrementally adding pixels to each connected component as the threshold is changed (Nistér and Stewénius 2008). As the threshold is changed, the area of each component (region) is monitored; regions whose rate of change of area with respect to the threshold is minimal are defined as *maximally stable*. Such regions are therefore invariant to both affine geometric and photometric (linear bias-gain or smooth monotonic) transformations (Figure 4.15). If desired, an affine coordinate frame can be fit to each detected region using its moment matrix.

The area of feature point detectors continues to be very active, with papers appearing every year at major computer vision conferences (Xiao and Shah 2003; Koethe 2003; Carneiro and Jepson 2005; Kenney, Zuliani, and Manjunath 2005; Bay, Tuytelaars, and Van Gool 2006; Platel, Balmachnova, Florack *et al.* 2006; Rosten and Drummond 2006). Mikolajczyk, Tuytelaars, Schmid *et al.* (2005) survey a number of popular affine region detectors and provide experimental comparisons of their invariance to common image transformations such as scaling, rotations, noise, and blur. These experimental results, code, and pointers to the surveyed papers can be found on their Web site at <http://www.robots.ox.ac.uk/~vgg/research/affine/>.

Of course, keypoints are not the only features that can be used for registering images. Zoghlaoui, Faugeras, and Deriche (1997) use line segments as well as point-like features to estimate homographies between pairs of images, whereas Bartoli, Coquerelle, and Sturm (2004) use line segments with local correspondences along the edges to extract 3D structure

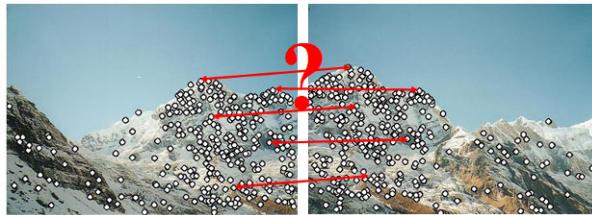


Figure 4.16 Feature matching: how can we extract local descriptors that are invariant to inter-image variations and yet still discriminative enough to establish correct correspondences?

and motion. Tuytelaars and Van Gool (2004) use affine invariant regions to detect correspondences for wide baseline stereo matching, whereas Kadir, Zisserman, and Brady (2004) detect salient regions where patch entropy and its rate of change with scale are locally maximal. Corso and Hager (2005) use a related technique to fit 2D oriented Gaussian kernels to homogeneous regions. More details on techniques for finding and matching curves, lines, and regions can be found later in this chapter.

4.1.2 Feature descriptors

After detecting features (keypoints), we must *match* them, i.e., we must determine which features come from corresponding locations in different images. In some situations, e.g., for video sequences (Shi and Tomasi 1994) or for stereo pairs that have been *rectified* (Zhang, Deriche, Faugeras *et al.* 1995; Loop and Zhang 1999; Scharstein and Szeliski 2002), the local motion around each feature point may be mostly translational. In this case, simple error metrics, such as the *sum of squared differences* or *normalized cross-correlation*, described in Section 8.1 can be used to directly compare the intensities in small patches around each feature point. (The comparative study by Mikolajczyk and Schmid (2005), discussed below, uses cross-correlation.) Because feature points may not be exactly located, a more accurate matching score can be computed by performing incremental motion refinement as described in Section 8.1.3 but this can be time consuming and can sometimes even decrease performance (Brown, Szeliski, and Winder 2005).

In most cases, however, the local appearance of features will change in orientation and scale, and sometimes even undergo affine deformations. Extracting a local scale, orientation, or affine frame estimate and then using this to resample the patch before forming the feature descriptor is thus usually preferable (Figure 4.17).

Even after compensating for these changes, the local appearance of image patches will usually still vary from image to image. How can we make image descriptors more invariant to such changes, while still preserving discriminability between different (non-corresponding) patches (Figure 4.16)? Mikolajczyk and Schmid (2005) review some recently developed view-invariant local image descriptors and experimentally compare their performance. Below, we describe a few of these descriptors in more detail.

Bias and gain normalization (MOPS). For tasks that do not exhibit large amounts of foreshortening, such as image stitching, simple normalized intensity patches perform reasonably well and are simple to implement (Brown, Szeliski, and Winder 2005) (Figure 4.17). In

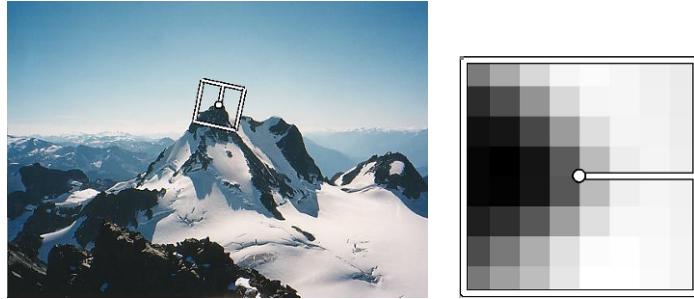


Figure 4.17 MOPS descriptors are formed using an 8×8 sampling of bias and gain normalized intensity values, with a sample spacing of five pixels relative to the detection scale (Brown, Szeliski, and Winder 2005) © 2005 IEEE. This low frequency sampling gives the features some robustness to interest point location error and is achieved by sampling at a higher pyramid level than the detection scale.

order to compensate for slight inaccuracies in the feature point detector (location, orientation, and scale), these multi-scale oriented patches (MOPS) are sampled at a spacing of five pixels relative to the detection scale, using a coarser level of the image pyramid to avoid aliasing. To compensate for affine photometric variations (linear exposure changes or bias and gain, (3.3)), patch intensities are re-scaled so that their mean is zero and their variance is one.

Scale invariant feature transform (SIFT). SIFT features are formed by computing the gradient at each pixel in a 16×16 window around the detected keypoint, using the appropriate level of the Gaussian pyramid at which the keypoint was detected. The gradient magnitudes are downweighted by a Gaussian fall-off function (shown as a blue circle in (Figure 4.18a) in order to reduce the influence of gradients far from the center, as these are more affected by small misregistrations.

In each 4×4 quadrant, a gradient orientation histogram is formed by (conceptually) adding the weighted gradient value to one of eight orientation histogram bins. To reduce the effects of location and dominant orientation misestimation, each of the original 256 weighted gradient magnitudes is softly added to $2 \times 2 \times 2$ histogram bins using trilinear interpolation. Softly distributing values to adjacent histogram bins is generally a good idea in any application where histograms are being computed, e.g., for Hough transforms (Section 4.3.2) or local histogram equalization (Section 3.1.4).

The resulting 128 non-negative values form a raw version of the SIFT descriptor vector. To reduce the effects of contrast or gain (additive variations are already removed by the gradient), the 128-D vector is normalized to unit length. To further make the descriptor robust to other photometric variations, values are clipped to 0.2 and the resulting vector is once again renormalized to unit length.

PCA-SIFT. Ke and Sukthankar (2004) propose a simpler way to compute descriptors inspired by SIFT; it computes the x and y (gradient) derivatives over a 39×39 patch and then reduces the resulting 3042-dimensional vector to 36 using principal component analysis (PCA) (Section 14.2.1 and Appendix A.1.2). Another popular variant of SIFT is SURF (Bay, Tuytelaars, and Van Gool 2006), which uses box filters to approximate the derivatives and

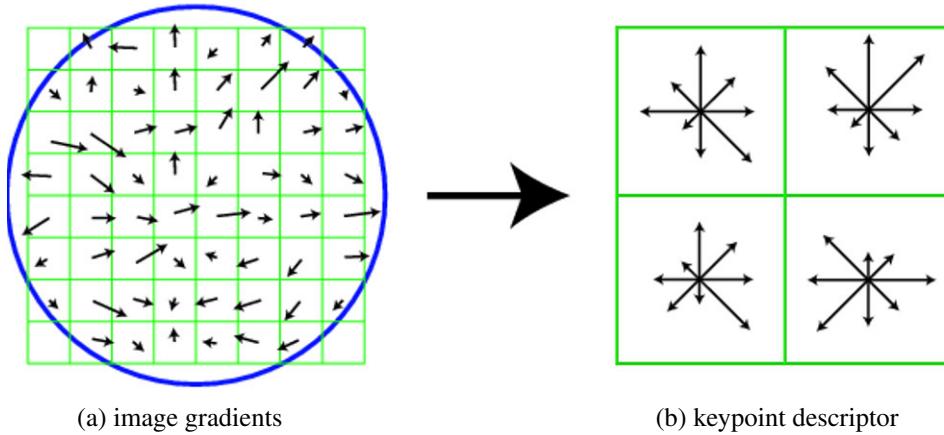


Figure 4.18 A schematic representation of Lowe’s (2004) scale invariant feature transform (SIFT): (a) Gradient orientations and magnitudes are computed at each pixel and weighted by a Gaussian fall-off function (blue circle). (b) A weighted gradient orientation histogram is then computed in each subregion, using trilinear interpolation. While this figure shows an 8×8 pixel patch and a 2×2 descriptor array, Lowe’s actual implementation uses 16×16 patches and a 4×4 array of eight-bin histograms.

integrals used in SIFT.

Gradient location-orientation histogram (GLOH). This descriptor, developed by Mikolajczyk and Schmid (2005), is a variant on SIFT that uses a log-polar binning structure instead of the four quadrants used by Lowe (2004) (Figure 4.19). The spatial bins are of radius 6, 11, and 15, with eight angular bins (except for the central region), for a total of 17 spatial bins and 16 orientation bins. The 272-dimensional histogram is then projected onto a 128-dimensional descriptor using PCA trained on a large database. In their evaluation, Mikolajczyk and Schmid (2005) found that GLOH, which has the best performance overall, outperforms SIFT by a small margin.

Steerable filters. Steerable filters (Section 3.2.3) are combinations of derivative of Gaussian filters that permit the rapid computation of even and odd (symmetric and anti-symmetric) edge-like and corner-like features at all possible orientations (Freeman and Adelson 1991). Because they use reasonably broad Gaussians, they too are somewhat insensitive to localization and orientation errors.

Performance of local descriptors. Among the local descriptors that Mikolajczyk and Schmid (2005) compared, they found that GLOH performed best, followed closely by SIFT (see Figure 4.25). They also present results for many other descriptors not covered in this book.

The field of feature descriptors continues to evolve rapidly, with some of the newer techniques looking at local color information (van de Weijer and Schmid 2006; Abdel-Hakim and Farag 2006). Winder and Brown (2007) develop a multi-stage framework for feature descriptor computation that subsumes both SIFT and GLOH (Figure 4.20a) and also allows

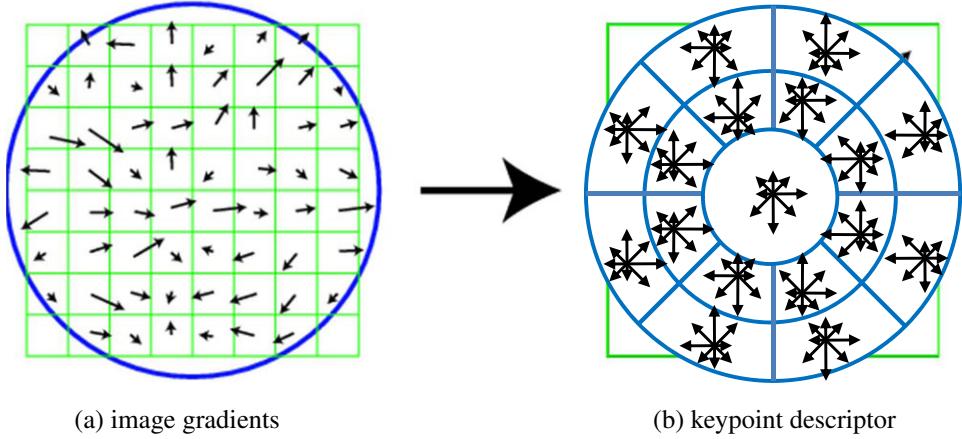


Figure 4.19 The gradient location-orientation histogram (GLOH) descriptor uses log-polar bins instead of square bins to compute orientation histograms (Mikolajczyk and Schmid 2005).

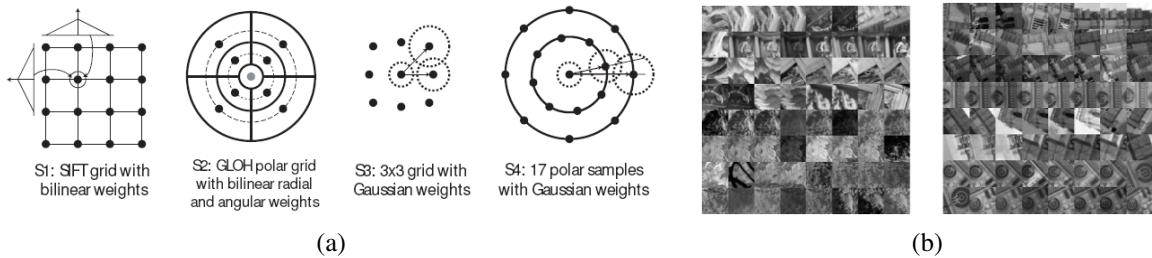


Figure 4.20 Spatial summation blocks for SIFT, GLOH, and some newly developed feature descriptors (Winder and Brown 2007) © 2007 IEEE: (a) The parameters for the new features, e.g., their Gaussian weights, are learned from a training database of (b) matched real-world image patches obtained from robust structure-from-motion applied to Internet photo collections (Hua, Brown, and Winder 2007).

them to learn optimal parameters for newer descriptors that outperform previous hand-tuned descriptors. Hua, Brown, and Winder (2007) extend this work by learning lower-dimensional projections of higher-dimensional descriptors that have the best discriminative power. Both of these papers use a database of real-world image patches (Figure 4.20b) obtained by sampling images at locations that were reliably matched using a robust structure-from-motion algorithm applied to Internet photo collections (Snavely, Seitz, and Szeliski 2006; Goesele, Snavely, Curless *et al.* 2007). In concurrent work, Tola, Lepetit, and Fua (2010) developed a similar DAISY descriptor for dense stereo matching and optimized its parameters based on ground truth stereo data.

While these techniques construct feature detectors that optimize for repeatability across *all* object classes, it is also possible to develop class- or instance-specific feature detectors that maximize *discriminability* from other classes (Ferencz, Learned-Miller, and Malik 2008).



Figure 4.21 Recognizing objects in a cluttered scene (Lowe 2004) © 2004 Springer. Two of the training images in the database are shown on the left. These are matched to the cluttered scene in the middle using SIFT features, shown as small squares in the right image. The affine warp of each recognized database image onto the scene is shown as a larger parallelogram in the right image.

4.1.3 Feature matching

Once we have extracted features and their descriptors from two or more images, the next step is to establish some preliminary feature matches between these images. In this section, we divide this problem into two separate components. The first is to select a *matching strategy*, which determines which correspondences are passed on to the next stage for further processing. The second is to devise efficient *data structures* and *algorithms* to perform this matching as quickly as possible. (See the discussion of related techniques in Section 14.3.2.)

Matching strategy and error rates

Determining which feature matches are reasonable to process further depends on the context in which the matching is being performed. Say we are given two images that overlap to a fair amount (e.g., for image stitching, as in Figure 4.16, or for tracking objects in a video). We know that most features in one image are likely to match the other image, although some may not match because they are occluded or their appearance has changed too much.

On the other hand, if we are trying to recognize how many known objects appear in a cluttered scene (Figure 4.21), most of the features may not match. Furthermore, a large number of potentially matching objects must be searched, which requires more efficient strategies, as described below.

To begin with, we assume that the feature descriptors have been designed so that Euclidean (vector magnitude) distances in feature space can be directly used for ranking potential matches. If it turns out that certain parameters (axes) in a descriptor are more reliable than others, it is usually preferable to re-scale these axes ahead of time, e.g., by determining how much they vary when compared against other known good matches (Hua, Brown, and Winder 2007). A more general process, which involves transforming feature vectors into a new scaled basis, is called *whitening* and is discussed in more detail in the context of eigenface-based face recognition (Section 14.2.1).

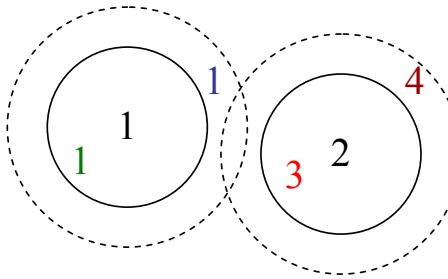


Figure 4.22 False positives and negatives: The black digits 1 and 2 are features being matched against a database of features in other images. At the current threshold setting (the solid circles), the green 1 is a *true positive* (good match), the blue 1 is a *false negative* (failure to match), and the red 3 is a *false positive* (incorrect match). If we set the threshold higher (the dashed circles), the blue 1 becomes a true positive but the brown 4 becomes an additional false positive.

	True matches	True non-matches	
Predicted matches	TP = 18	FP = 4	P' = 22
Predicted non-matches	FN = 2	TN = 76	N' = 78
P = 20	N = 80	Total = 100	
TPR = 0.90		FPR = 0.05	ACC = 0.94
			PPV = 0.82

Table 4.1 The number of matches correctly and incorrectly estimated by a feature matching algorithm, showing the number of true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN). The columns sum up to the actual number of positives (P) and negatives (N), while the rows sum up to the predicted number of positives (P') and negatives (N'). The formulas for the true positive rate (TPR), the false positive rate (FPR), the positive predictive value (PPV), and the accuracy (ACC) are given in the text.

Given a Euclidean distance metric, the simplest matching strategy is to set a threshold (maximum distance) and to return all matches from other images within this threshold. Setting the threshold too high results in too many *false positives*, i.e., incorrect matches being returned. Setting the threshold too low results in too many *false negatives*, i.e., too many correct matches being missed (Figure 4.22).

We can quantify the performance of a matching algorithm at a particular threshold by first counting the number of true and false matches and match failures, using the following definitions (Fawcett 2006):

- **TP:** true positives, i.e., number of correct matches;
- **FN:** false negatives, matches that were not correctly detected;
- **FP:** false positives, proposed matches that are incorrect;
- **TN:** true negatives, non-matches that were correctly rejected.

Table 4.1 shows a sample *confusion matrix* (contingency table) containing such numbers.

We can convert these numbers into *unit rates* by defining the following quantities (Fawcett 2006):

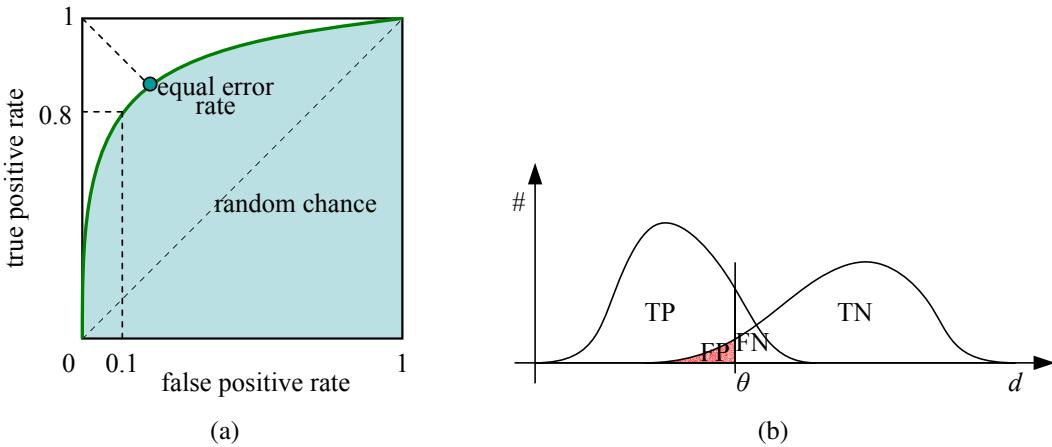


Figure 4.23 ROC curve and its related rates: (a) The ROC curve plots the true positive rate against the false positive rate for a particular combination of feature extraction and matching algorithms. Ideally, the true positive rate should be close to 1, while the false positive rate is close to 0. The area under the ROC curve (AUC) is often used as a single (scalar) measure of algorithm performance. Alternatively, the equal error rate is sometimes used. (b) The distribution of positives (matches) and negatives (non-matches) as a function of inter-feature distance d . As the threshold θ is increased, the number of true positives (TP) and false positives (FP) increases.

- true positive rate (TPR),

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{TP}}{\text{P}}; \quad (4.14)$$

- false positive rate (FPR),

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} = \frac{\text{FP}}{\text{N}}; \quad (4.15)$$

- positive predictive value (PPV),

$$\text{PPV} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\text{TP}}{\text{P}'}; \quad (4.16)$$

- accuracy (ACC),

$$\text{ACC} = \frac{\text{TP} + \text{TN}}{\text{P} + \text{N}}. \quad (4.17)$$

In the *information retrieval* (or document retrieval) literature (Baeza-Yates and Ribeiro-Neto 1999; Manning, Raghavan, and Schütze 2008), the term *precision* (how many returned documents are relevant) is used instead of PPV and *recall* (what fraction of relevant documents was found) is used instead of TPR.

Any particular matching strategy (at a particular threshold or parameter setting) can be rated by the TPR and FPR numbers; ideally, the true positive rate will be close to 1 and the false positive rate close to 0. As we vary the matching threshold, we obtain a family of such points, which are collectively known as the *receiver operating characteristic (ROC curve)* (Fawcett 2006) (Figure 4.23a). The closer this curve lies to the upper left corner, i.e., the larger the area under the curve (AUC), the better its performance. Figure 4.23b shows how we can plot the number of matches and non-matches as a function of inter-feature distance d .

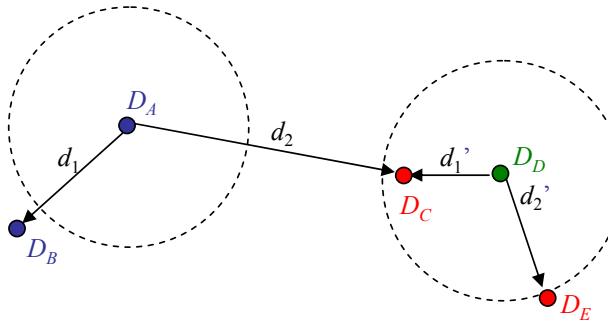


Figure 4.24 Fixed threshold, nearest neighbor, and nearest neighbor distance ratio matching. At a fixed distance threshold (dashed circles), descriptor D_A fails to match D_B and D_D incorrectly matches D_C and D_E . If we pick the nearest neighbor, D_A correctly matches D_B but D_D incorrectly matches D_C . Using nearest neighbor distance ratio (NNDR) matching, the small NNDR d_1/d_2 correctly matches D_A with D_B , and the large NNDR d'_1/d'_2 correctly rejects matches for D_D .

These curves can then be used to plot an ROC curve (Exercise 4.3). The ROC curve can also be used to calculate the *mean average precision*, which is the average precision (PPV) as you vary the threshold to select the best results, then the two top results, etc.

The problem with using a fixed threshold is that it is difficult to set; the useful range of thresholds can vary a lot as we move to different parts of the feature space (Lowe 2004; Mikolajczyk and Schmid 2005). A better strategy in such cases is to simply match the *nearest neighbor* in feature space. Since some features may have no matches (e.g., they may be part of background clutter in object recognition or they may be occluded in the other image), a threshold is still used to reduce the number of false positives.

Ideally, this threshold itself will adapt to different regions of the feature space. If sufficient training data is available (Hua, Brown, and Winder 2007), it is sometimes possible to learn different thresholds for different features. Often, however, we are simply given a collection of images to match, e.g., when stitching images or constructing 3D models from unordered photo collections (Brown and Lowe 2007, 2003; Snavely, Seitz, and Szeliski 2006). In this case, a useful heuristic can be to compare the nearest neighbor distance to that of the second nearest neighbor, preferably taken from an image that is known not to match the target (e.g., a different object in the database) (Brown and Lowe 2002; Lowe 2004). We can define this *nearest neighbor distance ratio* (Mikolajczyk and Schmid 2005) as

$$\text{NNDR} = \frac{d_1}{d_2} = \frac{\|D_A - D_B\|}{\|D_A - D_C\|}, \quad (4.18)$$

where d_1 and d_2 are the nearest and second nearest neighbor distances, D_A is the target descriptor, and D_B and D_C are its closest two neighbors (Figure 4.24).

The effects of using these three different matching strategies for the feature descriptors evaluated by Mikolajczyk and Schmid (2005) are shown in Figure 4.25. As you can see, the nearest neighbor and NNDR strategies produce improved ROC curves.

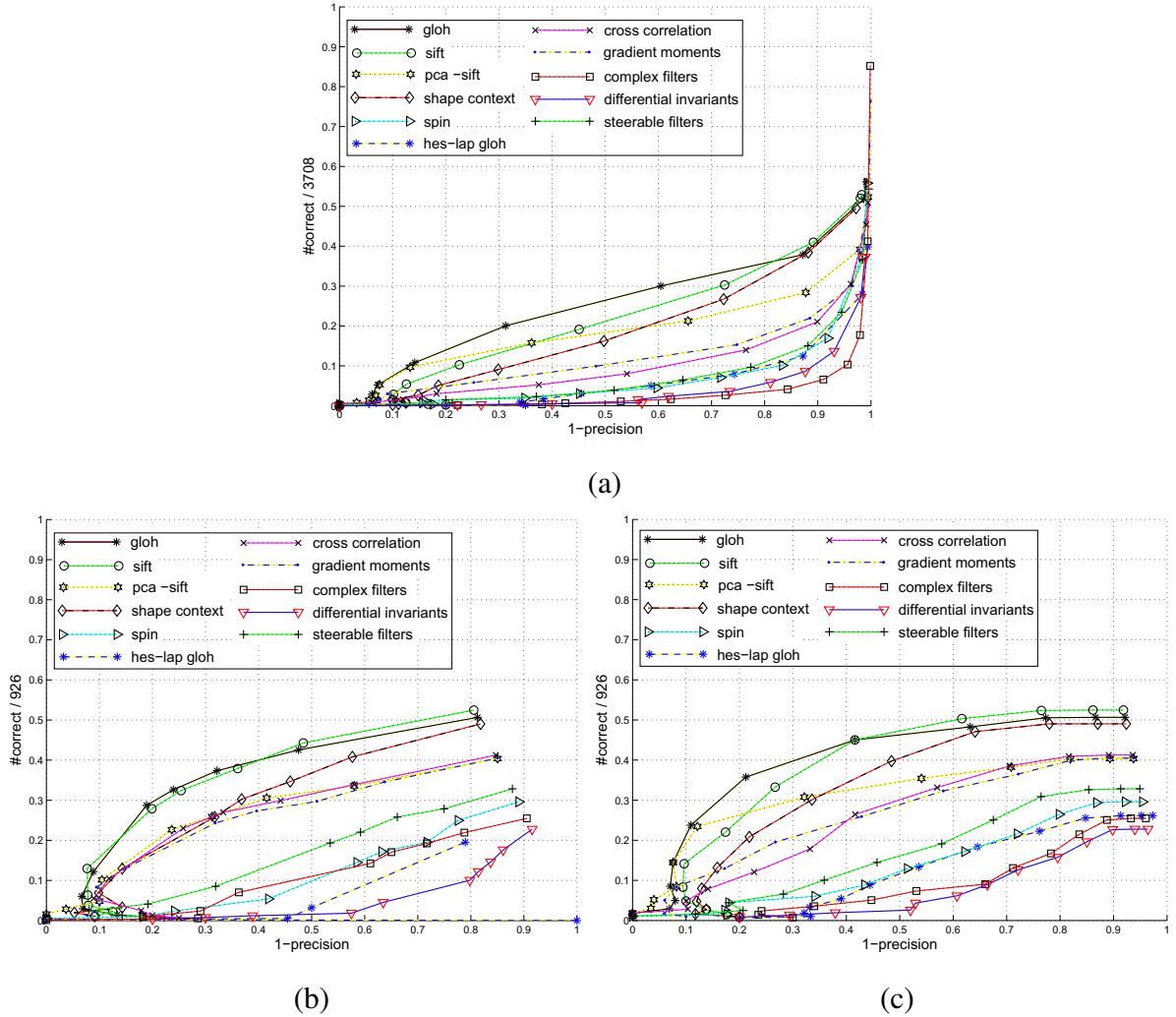


Figure 4.25 Performance of the feature descriptors evaluated by Mikolajczyk and Schmid (2005) © 2005 IEEE, shown for three matching strategies: (a) fixed threshold; (b) nearest neighbor; (c) nearest neighbor distance ratio (NNDR). Note how the ordering of the algorithms does not change that much, but the overall performance varies significantly between the different matching strategies.

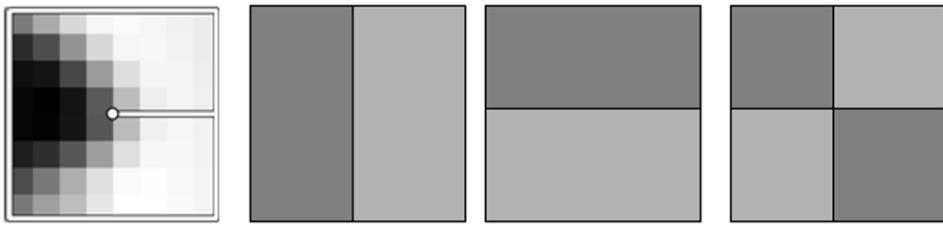


Figure 4.26 The three Haar wavelet coefficients used for hashing the MOPS descriptor devised by Brown, Szeliski, and Winder (2005) are computed by summing each 8×8 normalized patch over the light and dark gray regions and taking their difference.

Efficient matching

Once we have decided on a matching strategy, we still need to search efficiently for potential candidates. The simplest way to find all corresponding feature points is to compare all features against all other features in each pair of potentially matching images. Unfortunately, this is quadratic in the number of extracted features, which makes it impractical for most applications.

A better approach is to devise an *indexing structure*, such as a multi-dimensional search tree or a hash table, to rapidly search for features near a given feature. Such indexing structures can either be built for each image independently (which is useful if we want to only consider certain potential matches, e.g., searching for a particular object) or globally for all the images in a given database, which can potentially be faster, since it removes the need to iterate over each image. For extremely large databases (millions of images or more), even more efficient structures based on ideas from document retrieval (e.g., *vocabulary trees*, (Nistér and Stewénius 2006)) can be used (Section 14.3.2).

One of the simpler techniques to implement is multi-dimensional hashing, which maps descriptors into fixed size buckets based on some function applied to each descriptor vector. At matching time, each new feature is hashed into a bucket, and a search of nearby buckets is used to return potential candidates, which can then be sorted or graded to determine which are valid matches.

A simple example of hashing is the Haar wavelets used by Brown, Szeliski, and Winder (2005) in their MOPS paper. During the matching structure construction, each 8×8 scaled, oriented, and normalized MOPS patch is converted into a three-element index by performing sums over different quadrants of the patch (Figure 4.26). The resulting three values are normalized by their expected standard deviations and then mapped to the two (of $b = 10$) nearest 1D bins. The three-dimensional indices formed by concatenating the three quantized values are used to index the $2^3 = 8$ bins where the feature is stored (added). At query time, only the primary (closest) indices are used, so only a single three-dimensional bin needs to be examined. The coefficients in the bin can then be used to select k approximate nearest neighbors for further processing (such as computing the NNDR).

A more complex, but more widely applicable, version of hashing is called *locality sensitive hashing*, which uses unions of independently computed hashing functions to index the features (Gionis, Indyk, and Motwani 1999; Shakhnarovich, Darrell, and Indyk 2006). Shakhnarovich, Viola, and Darrell (2003) extend this technique to be more sensitive to the

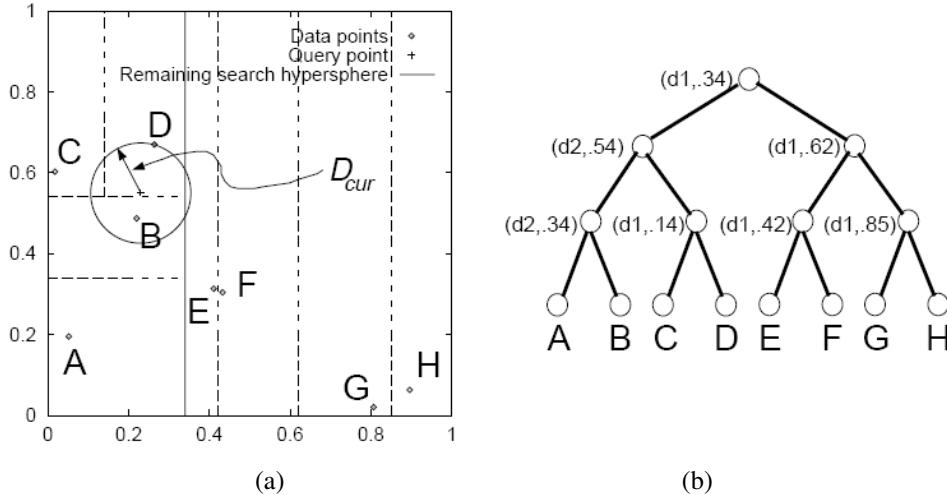


Figure 4.27 K-d tree and best bin first (BBF) search (Beis and Lowe 1999) © 1999 IEEE: (a) The spatial arrangement of the axis-aligned cutting planes is shown using dashed lines. Individual data points are shown as small diamonds. (b) The same subdivision can be represented as a tree, where each interior node represents an axis-aligned cutting plane (e.g., the top node cuts along dimension d_1 at value .34) and each leaf node is a data point. During a BBF search, a query point (denoted by “+”) first looks in its containing bin (D) and then in its nearest adjacent bin (B), rather than its closest neighbor in the tree (C).

distribution of points in parameter space, which they call *parameter-sensitive hashing*. Even more recent work converts high-dimensional descriptor vectors into binary codes that can be compared using Hamming distances (Torralba, Weiss, and Fergus 2008; Weiss, Torralba, and Fergus 2008) or that can accommodate arbitrary kernel functions (Kulis and Grauman 2009; Raginsky and Lazebnik 2009).

Another widely used class of indexing structures are multi-dimensional search trees. The best known of these are *k-d trees*, also often written as *kd-trees*, which divide the multi-dimensional feature space along alternating axis-aligned hyperplanes, choosing the threshold along each axis so as to maximize some criterion, such as the search tree balance (Samet 1989). Figure 4.27 shows an example of a two-dimensional k-d tree. Here, eight different data points A–H are shown as small diamonds arranged on a two-dimensional plane. The k-d tree recursively splits this plane along axis-aligned (horizontal or vertical) cutting planes. Each split can be denoted using the dimension number and split value (Figure 4.27b). The splits are arranged so as to try to balance the tree, i.e., to keep its maximum depth as small as possible. At query time, a classic k-d tree search first locates the query point (+) in its appropriate bin (D), and then searches nearby leaves in the tree (C, B, ...) until it can guarantee that the nearest neighbor has been found. The best bin first (BBF) search (Beis and Lowe 1999) searches bins in order of their spatial proximity to the query point and is therefore usually more efficient.

Many additional data structures have been developed over the years for solving nearest neighbor problems (Arya, Mount, Netanyahu *et al.* 1998; Liang, Liu, Xu *et al.* 2001; Hjaltason and Samet 2003). For example, Nene and Nayar (1997) developed a technique they call

slicing that uses a series of 1D binary searches on the point list sorted along different dimensions to efficiently cull down a list of candidate points that lie within a hypercube of the query point. Grauman and Darrell (2005) reweight the matches at different levels of an indexing tree, which allows their technique to be less sensitive to discretization errors in the tree construction. Nistér and Stewénius (2006) use a *metric tree*, which compares feature descriptors to a small number of prototypes at each level in a hierarchy. The resulting quantized *visual words* can then be used with classical information retrieval (document relevance) techniques to quickly winnow down a set of potential candidates from a database of millions of images (Section 14.3.2). Muja and Lowe (2009) compare a number of these approaches, introduce a new one of their own (priority search on hierarchical k-means trees), and conclude that multiple randomized k-d trees often provide the best performance. Despite all of this promising work, the rapid computation of image feature correspondences remains a challenging open research problem.

Feature match verification and densification

Once we have some hypothetical (putative) matches, we can often use geometric alignment (Section 6.1) to verify which matches are *inliers* and which ones are *outliers*. For example, if we expect the whole image to be translated or rotated in the matching view, we can fit a global geometric transform and keep only those feature matches that are sufficiently close to this estimated transformation. The process of selecting a small set of seed matches and then verifying a larger set is often called *random sampling* or RANSAC (Section 6.1.4). Once an initial set of correspondences has been established, some systems look for additional matches, e.g., by looking for additional correspondences along epipolar lines (Section 11.1) or in the vicinity of estimated locations based on the global transform. These topics are discussed further in Sections 6.1, 11.2, and 14.3.1.

4.1.4 Feature tracking

An alternative to independently finding features in all candidate images and then matching them is to find a set of likely feature locations in a first image and to then *search* for their corresponding locations in subsequent images. This kind of *detect then track* approach is more widely used for video tracking applications, where the expected amount of motion and appearance deformation between adjacent frames is expected to be small.

The process of selecting good features to track is closely related to selecting good features for more general recognition applications. In practice, regions containing high gradients in both directions, i.e., which have high eigenvalues in the auto-correlation matrix (4.8), provide stable locations at which to find correspondences (Shi and Tomasi 1994).

In subsequent frames, searching for locations where the corresponding patch has low squared difference (4.1) often works well enough. However, if the images are undergoing brightness change, explicitly compensating for such variations (8.9) or using *normalized cross-correlation* (8.11) may be preferable. If the search range is large, it is also often more efficient to use a *hierarchical* search strategy, which uses matches in lower-resolution images to provide better initial guesses and hence speed up the search (Section 8.1.1). Alternatives to this strategy involve learning what the appearance of the patch being tracked should be and then searching for it in the vicinity of its predicted position (Avidan 2001; Jurie and Dhome



Figure 4.28 Feature tracking using an affine motion model (Shi and Tomasi 1994) © 1994 IEEE, Top row: image patch around the tracked feature location. Bottom row: image patch after warping back toward the first frame using an affine deformation. Even though the speed sign gets larger from frame to frame, the affine transformation maintains a good resemblance between the original and subsequent tracked frames.

2002; Williams, Blake, and Cipolla 2003). These topics are all covered in more detail in Section 8.1.3.

If features are being tracked over longer image sequences, their appearance can undergo larger changes. You then have to decide whether to continue matching against the originally detected patch (feature) or to re-sample each subsequent frame at the matching location. The former strategy is prone to failure as the original patch can undergo appearance changes such as foreshortening. The latter runs the risk of the feature drifting from its original location to some other location in the image (Shi and Tomasi 1994). (Mathematically, small mis-registration errors compound to create a *Markov Random Walk*, which leads to larger drift over time.)

A preferable solution is to compare the original patch to later image locations using an *affine* motion model (Section 8.2). Shi and Tomasi (1994) first compare patches in neighboring frames using a translational model and then use the location estimates produced by this step to initialize an affine registration between the patch in the current frame and the base frame where a feature was first detected (Figure 4.28). In their system, features are only detected infrequently, i.e., only in regions where tracking has failed. In the usual case, an area around the current *predicted* location of the feature is searched with an incremental registration algorithm (Section 8.1.3). The resulting tracker is often called the Kanade–Lucas–Tomasi (KLT) tracker.

Since their original work on feature tracking, Shi and Tomasi's approach has generated a string of interesting follow-on papers and applications. Beardsley, Torr, and Zisserman (1996) use extended feature tracking combined with structure from motion (Chapter 7) to incrementally build up sparse 3D models from video sequences. Kang, Szeliski, and Shum (1997) tie together the corners of adjacent (regularly gridded) patches to provide some additional stability to the tracking, at the cost of poorer handling of occlusions. Tommasini, Fusello, Trucco *et al.* (1998) provide a better spurious match rejection criterion for the basic Shi and Tomasi algorithm, Collins and Liu (2003) provide improved mechanisms for feature selection and dealing with larger appearance changes over time, and Shafique and Shah (2005) develop algorithms for feature matching (data association) for videos with large numbers of



Figure 4.29 Real-time head tracking using the fast trained classifiers of Lepetit, Pilet, and Fua (2004) © 2004 IEEE.

moving objects or points. Yilmaz, Javed, and Shah (2006) and Lepetit and Fua (2005) survey the larger field of object tracking, which includes not only feature-based techniques but also alternative techniques based on contour and region (Section 5.1).

One of the newest developments in feature tracking is the use of learning algorithms to build special-purpose recognizers to rapidly search for matching features anywhere in an image (Lepetit, Pilet, and Fua 2006; Hinterstoisser, Benhimane, Navab *et al.* 2008; Rogez, Rihan, Ramalingam *et al.* 2008; Özysal, Calonder, Lepetit *et al.* 2010).² By taking the time to train classifiers on sample patches and their affine deformations, extremely fast and reliable feature detectors can be constructed, which enables much faster motions to be supported (Figure 4.29). Coupling such features to deformable models (Pilet, Lepetit, and Fua 2008) or structure-from-motion algorithms (Klein and Murray 2008) can result in even higher stability.

4.1.5 Application: Performance-driven animation

One of the most compelling applications of fast feature tracking is *performance-driven animation*, i.e., the interactive deformation of a 3D graphics model based on tracking a user's motions (Williams 1990; Litwinowicz and Williams 1994; Lepetit, Pilet, and Fua 2004).

Buck, Finkelstein, Jacobs *et al.* (2000) present a system that tracks a user's facial expressions and head motions and then uses them to morph among a series of hand-drawn sketches. An animator first extracts the eye and mouth regions of each sketch and draws control lines over each image (Figure 4.30a). At run time, a face-tracking system (Toyama 1998) determines the current location of these features (Figure 4.30b). The animation system decides

² See also my previous comment on earlier work in learning-based tracking (Avidan 2001; Jurie and Dhome 2002; Williams, Blake, and Cipolla 2003).



Figure 4.30 Performance-driven, hand-drawn animation (Buck, Finkelstein, Jacobs *et al.* 2000) © 2000 ACM: (a) eye and mouth portions of hand-drawn sketch with their overlaid control lines; (b) an input video frame with the tracked features overlaid; (c) a different input video frame along with its (d) corresponding hand-drawn animation.

which input images to morph based on nearest neighbor feature appearance matching and triangular barycentric interpolation. It also computes the global location and orientation of the head from the tracked features. The resulting morphed eye and mouth regions are then composited back into the overall head model to yield a frame of hand-drawn animation (Figure 4.30d).

In more recent work, Barnes, Jacobs, Sanders *et al.* (2008) watch users animate paper cutouts on a desk and then turn the resulting motions and drawings into seamless 2D animations.

4.2 Edges

While interest points are useful for finding image locations that can be accurately matched in 2D, edge points are far more plentiful and often carry important semantic associations. For example, the boundaries of objects, which also correspond to occlusion events in 3D, are usually delineated by visible contours. Other kinds of edges correspond to shadow boundaries or crease edges, where surface orientation changes rapidly. Isolated edge points can also be grouped into longer *curves* or *contours*, as well as *straight line segments* (Section 4.3). It is interesting that even young children have no difficulty in recognizing familiar objects or animals from such simple line drawings.

4.2.1 Edge detection

Given an image, how can we find the salient edges? Consider the color images in Figure 4.31. If someone asked you to point out the most “salient” or “strongest” edges or the object boundaries (Martin, Fowlkes, and Malik 2004; Arbeláez, Maire, Fowlkes *et al.* 2010), which ones would you trace? How closely do your perceptions match the edge images shown in Figure 4.31?

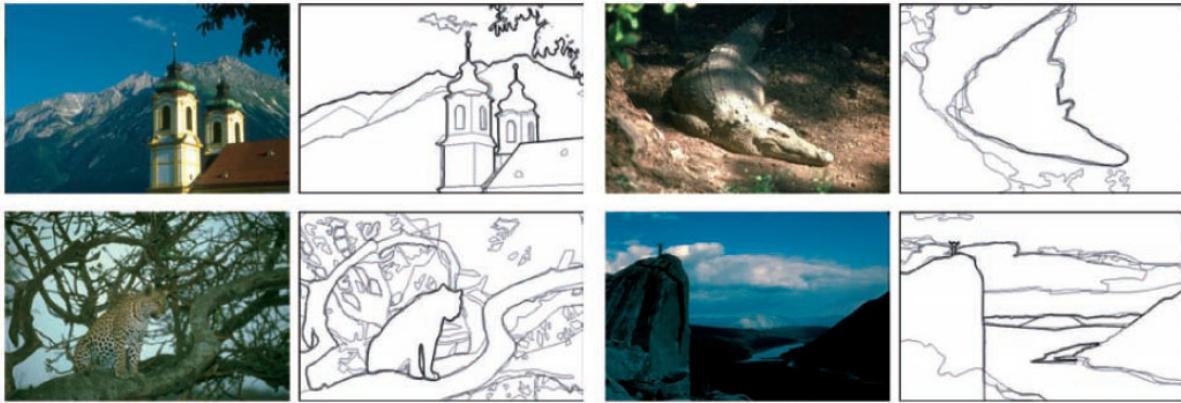


Figure 4.31 Human boundary detection (Martin, Fowlkes, and Malik 2004) © 2004 IEEE. The darkness of the edges corresponds to how many human subjects marked an object boundary at that location.

Qualitatively, edges occur at boundaries between regions of different color, intensity, or texture. Unfortunately, segmenting an image into coherent regions is a difficult task, which we address in Chapter 5. Often, it is preferable to detect edges using only purely local information.

Under such conditions, a reasonable approach is to define an edge as a location of *rapid intensity variation*.³ Think of an image as a height field. On such a surface, edges occur at locations of *steep slopes*, or equivalently, in regions of closely packed contour lines (on a topographic map).

A mathematical way to define the slope and direction of a surface is through its gradient,

$$\mathbf{J}(\mathbf{x}) = \nabla I(\mathbf{x}) = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right)(\mathbf{x}). \quad (4.19)$$

The local gradient vector \mathbf{J} points in the direction of *steepest ascent* in the intensity function. Its magnitude is an indication of the slope or strength of the variation, while its orientation points in a direction *perpendicular* to the local contour.

Unfortunately, taking image derivatives accentuates high frequencies and hence amplifies noise, since the proportion of noise to signal is larger at high frequencies. It is therefore prudent to smooth the image with a low-pass filter prior to computing the gradient. Because we would like the response of our edge detector to be independent of orientation, a circularly symmetric smoothing filter is desirable. As we saw in Section 3.2, the Gaussian is the only separable circularly symmetric filter and so it is used in most edge detection algorithms. Canny (1986) discusses alternative filters and a number of researcher review alternative edge detection algorithms and compare their performance (Davis 1975; Nalwa and Binford 1986; Nalwa 1987; Deriche 1987; Freeman and Adelson 1991; Nalwa 1993; Heath, Sarkar, Sanocki *et al.* 1998; Crane 1997; Ritter and Wilson 2000; Bowyer, Kranenburg, and Dougherty 2001; Arbeláez, Maire, Fowlkes *et al.* 2010).

Because differentiation is a linear operation, it commutes with other linear filtering oper-

³ We defer the topic of edge detection in color images.

ations. The gradient of the smoothed image can therefore be written as

$$\mathbf{J}_\sigma(\mathbf{x}) = \nabla[G_\sigma(\mathbf{x}) * I(\mathbf{x})] = [\nabla G_\sigma](\mathbf{x}) * I(\mathbf{x}), \quad (4.20)$$

i.e., we can convolve the image with the horizontal and vertical derivatives of the Gaussian kernel function,

$$\nabla G_\sigma(\mathbf{x}) = \left(\frac{\partial G_\sigma}{\partial x}, \frac{\partial G_\sigma}{\partial y} \right)(\mathbf{x}) = [-x \ -y] \frac{1}{\sigma^3} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \quad (4.21)$$

(The parameter σ indicates the width of the Gaussian.) This is the same computation that is performed by Freeman and Adelson's (1991) first-order steerable filter, which we already covered in Section 3.2.3.

For many applications, however, we wish to thin such a continuous gradient image to only return isolated edges, i.e., as single pixels at discrete locations along the edge contours. This can be achieved by looking for *maxima* in the edge strength (gradient magnitude) in a direction *perpendicular* to the edge orientation, i.e., along the gradient direction.

Finding this maximum corresponds to taking a directional derivative of the strength field in the direction of the gradient and then looking for zero crossings. The desired directional derivative is equivalent to the dot product between a second gradient operator and the results of the first,

$$S_\sigma(\mathbf{x}) = \nabla \cdot \mathbf{J}_\sigma(\mathbf{x}) = [\nabla^2 G_\sigma](\mathbf{x}) * I(\mathbf{x}). \quad (4.22)$$

The gradient operator dot product with the gradient is called the *Laplacian*. The convolution kernel

$$\nabla^2 G_\sigma(\mathbf{x}) = \frac{1}{\sigma^3} \left(2 - \frac{x^2 + y^2}{2\sigma^2} \right) \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \quad (4.23)$$

is therefore called the *Laplacian of Gaussian* (LoG) kernel (Marr and Hildreth 1980). This kernel can be split into two separable parts,

$$\nabla^2 G_\sigma(\mathbf{x}) = \frac{1}{\sigma^3} \left(1 - \frac{x^2}{2\sigma^2} \right) G_\sigma(x) G_\sigma(y) + \frac{1}{\sigma^3} \left(1 - \frac{y^2}{2\sigma^2} \right) G_\sigma(y) G_\sigma(x) \quad (4.24)$$

(Wiejak, Buxton, and Buxton 1985), which allows for a much more efficient implementation using separable filtering (Section 3.2.1).

In practice, it is quite common to replace the Laplacian of Gaussian convolution with a Difference of Gaussian (DoG) computation, since the kernel shapes are qualitatively similar (Figure 3.35). This is especially convenient if a “Laplacian pyramid” (Section 3.5) has already been computed.⁴

In fact, it is not strictly necessary to take differences between adjacent levels when computing the edge field. Think about what a zero crossing in a “generalized” difference of Gaussians image represents. The finer (smaller kernel) Gaussian is a noise-reduced version of the original image. The coarser (larger kernel) Gaussian is an estimate of the average intensity over a larger region. Thus, whenever the DoG image changes sign, this corresponds to the (slightly blurred) image going from relatively darker to relatively lighter, as compared to the average intensity in that neighborhood.

⁴ Recall that Burt and Adelson's (1983a) “Laplacian pyramid” actually computed differences of Gaussian-filtered levels.

Once we have computed the sign function $S(\mathbf{x})$, we must find its *zero crossings* and convert these into edge elements (*edgels*). An easy way to detect and represent zero crossings is to look for adjacent pixel locations \mathbf{x}_i and \mathbf{x}_j where the sign changes value, i.e., $[S(\mathbf{x}_i) > 0] \neq [S(\mathbf{x}_j) > 0]$.

The sub-pixel location of this crossing can be obtained by computing the “ x -intercept” of the “line” connecting $S(\mathbf{x}_i)$ and $S(\mathbf{x}_j)$,

$$\mathbf{x}_z = \frac{\mathbf{x}_i S(\mathbf{x}_j) - \mathbf{x}_j S(\mathbf{x}_i)}{S(\mathbf{x}_j) - S(\mathbf{x}_i)}. \quad (4.25)$$

The orientation and strength of such edgels can be obtained by linearly interpolating the gradient values computed on the original pixel grid.

An alternative edgel representation can be obtained by linking adjacent edgels on the dual grid to form edgels that live *inside* each square formed by four adjacent pixels in the original pixel grid.⁵ The (potential) advantage of this representation is that the edgels now live on a grid offset by half a pixel from the original pixel grid and are thus easier to store and access. As before, the orientations and strengths of the edges can be computed by interpolating the gradient field or estimating these values from the difference of Gaussian image (see Exercise 4.7).

In applications where the accuracy of the edge orientation is more important, higher-order steerable filters can be used (Freeman and Adelson 1991) (see Section 3.2.3). Such filters are more selective for more elongated edges and also have the possibility of better modeling curve intersections because they can represent multiple orientations at the same pixel (Figure 3.16). Their disadvantage is that they are more expensive to compute and the directional derivative of the edge strength does not have a simple closed form solution.⁶

Scale selection and blur estimation

As we mentioned before, the derivative, Laplacian, and Difference of Gaussian filters (4.20–4.23) all require the selection of a spatial scale parameter σ . If we are only interested in detecting sharp edges, the width of the filter can be determined from image noise characteristics (Canny 1986; Elder and Zucker 1998). However, if we want to detect edges that occur at different resolutions (Figures 4.32b–c), a *scale-space* approach that detects and then selects edges at different scales may be necessary (Witkin 1983; Lindeberg 1994, 1998a; Nielsen, Florack, and Deriche 1997).

Elder and Zucker (1998) present a principled approach to solving this problem. Given a known image noise level, their technique computes, for every pixel, the minimum scale at which an edge can be reliably detected (Figure 4.32d). Their approach first computes gradients densely over an image by selecting among gradient estimates computed at different scales, based on their gradient magnitudes. It then performs a similar estimate of minimum scale for directed second derivatives and uses zero crossings of this latter quantity to robustly select edges (Figures 4.32e–f). As an optional final step, the blur width of each edge can be computed from the distance between extrema in the second derivative response minus the width of the Gaussian filter.

⁵ This algorithm is a 2D version of the 3D *marching cubes* isosurface extraction algorithm (Lorensen and Cline 1987).

⁶ In fact, the edge orientation can have a 180° ambiguity for “bar edges”, which makes the computation of zero crossings in the derivative more tricky.

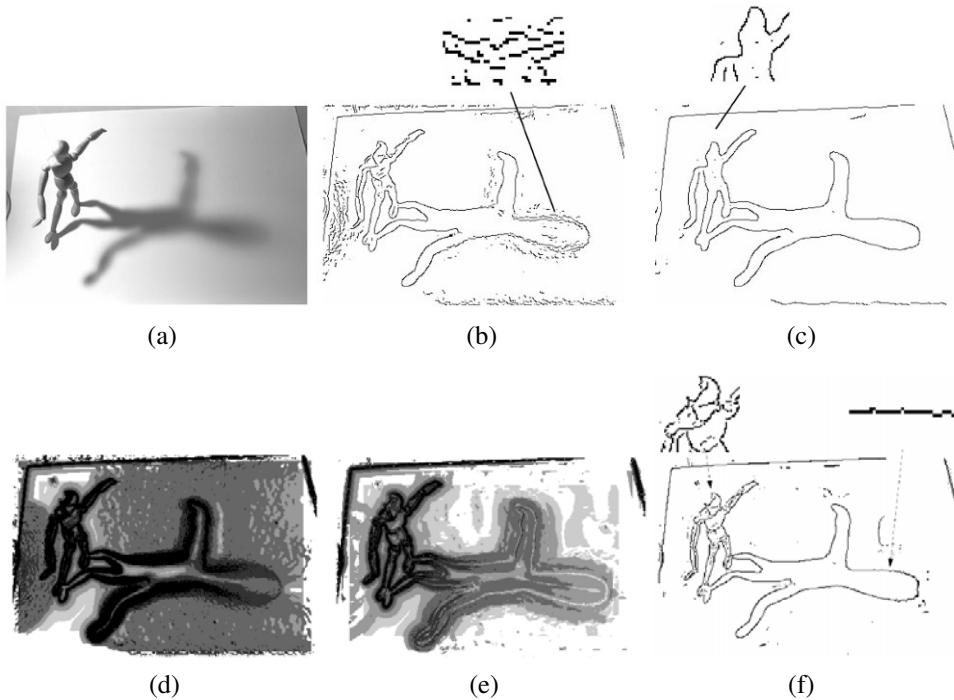


Figure 4.32 Scale selection for edge detection (Elder and Zucker 1998) © 1998 IEEE: (a) original image; (b–c) Canny/Deriche edge detector tuned to the finer (mannequin) and coarser (shadow) scales; (d) minimum reliable scale for gradient estimation; (e) minimum reliable scale for second derivative estimation; (f) final detected edges.

Color edge detection

While most edge detection techniques have been developed for grayscale images, color images can provide additional information. For example, noticeable edges between *iso-luminant* colors (colors that have the same luminance) are useful cues but fail to be detected by grayscale edge operators.

One simple approach is to combine the outputs of grayscale detectors run on each color band separately.⁷ However, some care must be taken. For example, if we simply sum up the gradients in each of the color bands, the signed gradients may actually cancel each other! (Consider, for example a pure red-to-green edge.) We could also detect edges independently in each band and then take the union of these, but this might lead to thickened or doubled edges that are hard to link.

A better approach is to compute the *oriented energy* in each band (Morrone and Burr 1988; Perona and Malik 1990a), e.g., using a second-order steerable filter (Section 3.2.3) (Freeman and Adelson 1991), and then sum up the orientation-weighted energies and find their joint best orientation. Unfortunately, the directional derivative of this energy may not have a closed form solution (as in the case of signed first-order steerable filters), so a simple zero crossing-based strategy cannot be used. However, the technique described by Elder and

⁷ Instead of using the raw RGB space, a more perceptually uniform color space such as L*a*b* (see Section 2.3.2) can be used instead. When trying to match human performance (Martin, Fowlkes, and Malik 2004), this makes sense. However, in terms of the physics of the underlying image formation and sensing, it may be a questionable strategy.

Zucker (1998) can be used to compute these zero crossings numerically instead.

An alternative approach is to estimate local color statistics in regions around each pixel (Ruzon and Tomasi 2001; Martin, Fowlkes, and Malik 2004). This has the advantage that more sophisticated techniques (e.g., 3D color histograms) can be used to compare regional statistics and that additional measures, such as texture, can also be considered. Figure 4.33 shows the output of such detectors.

Of course, many other approaches have been developed for detecting color edges, dating back to early work by Nevatia (1977). Ruzon and Tomasi (2001) and Gevers, van de Weijer, and Stokman (2006) provide good reviews of these approaches, which include ideas such as fusing outputs from multiple channels, using multidimensional gradients, and vector-based methods.

Combining edge feature cues

If the goal of edge detection is to match human *boundary detection* performance (Bowyer, Kranenburg, and Dougherty 2001; Martin, Fowlkes, and Malik 2004; Arbeláez, Maire, Fowlkes *et al.* 2010), as opposed to simply finding stable features for matching, even better detectors can be constructed by combining multiple low-level cues such as brightness, color, and texture.

Martin, Fowlkes, and Malik (2004) describe a system that combines brightness, color, and texture edges to produce state-of-the-art performance on a database of hand-segmented natural color images (Martin, Fowlkes, Tal *et al.* 2001). First, they construct and train⁸ separate oriented half-disc detectors for measuring significant differences in brightness (luminance), color (a* and b* channels, summed responses), and texture (un-normalized filter bank responses from the work of Malik, Belongie, Leung *et al.* (2001)). Some of the responses are then sharpened using a soft non-maximal suppression technique. Finally, the outputs of the three detectors are combined using a variety of machine-learning techniques, from which logistic regression is found to have the best tradeoff between speed, space and accuracy . The resulting system (see Figure 4.33 for some examples) is shown to outperform previously developed techniques. Maire, Arbelaez, Fowlkes *et al.* (2008) improve on these results by combining the detector based on local appearance with a *spectral* (segmentation-based) detector (Belongie and Malik 1998). In more recent work, Arbeláez, Maire, Fowlkes *et al.* (2010) build a hierarchical segmentation on top of this edge detector using a variant of the watershed algorithm.

4.2.2 Edge linking

While isolated edges can be useful for a variety of applications, such as line detection (Section 4.3) and sparse stereo matching (Section 11.2), they become even more useful when linked into continuous contours.

If the edges have been detected using zero crossings of some function, linking them up is straightforward, since adjacent edgels share common endpoints. Linking the edgels into chains involves picking up an unlinked edgel and following its neighbors in both directions. Either a sorted list of edgels (sorted first by x coordinates and then by y coordinates, for example) or a 2D array can be used to accelerate the neighbor finding. If edges were not

⁸ The training uses 200 labeled images and testing is performed on a different set of 100 images.



Figure 4.33 Combined brightness, color, texture boundary detector (Martin, Fowlkes, and Malik 2004) © 2004 IEEE. Successive rows show the outputs of the brightness gradient (BG), color gradient (CG), texture gradient (TG), and combined (BG+CG+TG) detectors. The final row shows human-labeled boundaries derived from a database of hand-segmented images (Martin, Fowlkes, Tal *et al.* 2001).

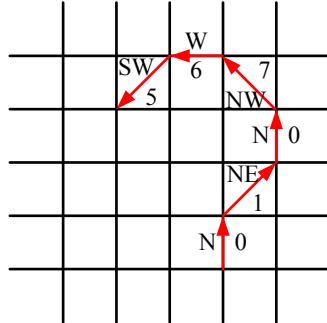


Figure 4.34 Chain code representation of a grid-aligned linked edge chain. The code is represented as a series of direction codes, e.g., 0 1 0 7 6 5, which can further be compressed using predictive and run-length coding.

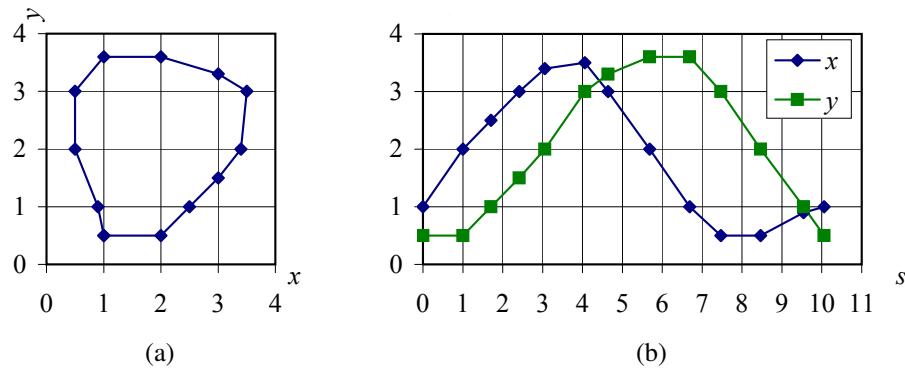


Figure 4.35 Arc-length parameterization of a contour: (a) discrete points along the contour are first transcribed as (b) (x, y) pairs along the arc length s . This curve can then be regularly re-sampled or converted into alternative (e.g., Fourier) representations.

detected using zero crossings, finding the continuation of an edgel can be tricky. In this case, comparing the orientation (and, optionally, phase) of adjacent edgels can be used for disambiguation. Ideas from connected component computation can also sometimes be used to make the edge linking process even faster (see Exercise 4.8).

Once the edgels have been linked into chains, we can apply an optional thresholding with hysteresis to remove low-strength contour segments (Canny 1986). The basic idea of hysteresis is to set two different thresholds and allow a curve being tracked above the higher threshold to dip in strength down to the lower threshold.

Linked edgel lists can be encoded more compactly using a variety of alternative representations. A *chain code* encodes a list of connected points lying on an \mathcal{N}_8 grid using a three-bit code corresponding to the eight cardinal directions (N, NE, E, SE, S, SW, W, NW) between a point and its successor (Figure 4.34). While this representation is more compact than the original edgel list (especially if predictive variable-length coding is used), it is not very suitable for further processing.

A more useful representation is the *arc length parameterization* of a contour, $x(s)$, where s denotes the arc length along a curve. Consider the linked set of edgels shown in Fig-

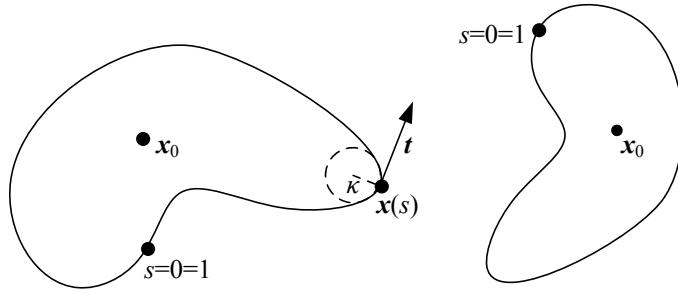


Figure 4.36 Matching two contours using their arc-length parameterization. If both curves are normalized to unit length, $s \in [0, 1]$ and centered around their centroid x_0 , they will have the same descriptor up to an overall “temporal” shift (due to different starting points for $s = 0$) and a phase (x - y) shift (due to rotation).

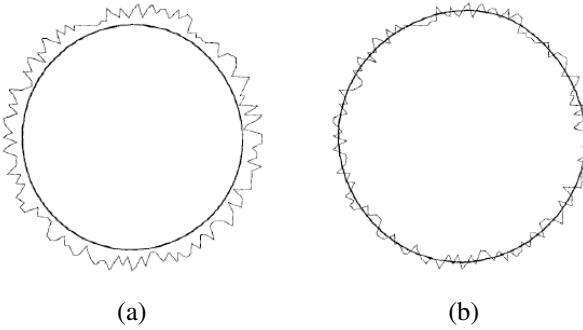


Figure 4.37 Curve smoothing with a Gaussian kernel (Lowe 1988) © 1998 IEEE: (a) without a shrinkage correction term; (b) with a shrinkage correction term.

ure 4.35a. We start at one point (the dot at $(1.0, 0.5)$ in Figure 4.35a) and plot it at coordinate $s = 0$ (Figure 4.35b). The next point at $(2.0, 0.5)$ gets plotted at $s = 1$, and the next point at $(2.5, 1.0)$ gets plotted at $s = 1.7071$, i.e., we increment s by the length of each edge segment. The resulting plot can be resampled on a regular (say, integral) s grid before further processing.

The advantage of the arc-length parameterization is that it makes matching and processing (e.g., smoothing) operations much easier. Consider the two curves describing similar shapes shown in Figure 4.36. To compare the curves, we first subtract the average values $\bar{x}_0 = \int_s x(s)$ from each descriptor. Next, we rescale each descriptor so that s goes from 0 to 1 instead of 0 to S , i.e., we divide $x(s)$ by S . Finally, we take the Fourier transform of each normalized descriptor, treating each $x = (x, y)$ value as a complex number. If the original curves are the same (up to an unknown scale and rotation), the resulting Fourier transforms should differ only by a scale change in magnitude plus a constant complex phase shift, due to rotation, and a linear phase shift in the domain, due to different starting points for s (see Exercise 4.9).

Arc-length parameterization can also be used to smooth curves in order to remove digitization noise. However, if we just apply a regular smoothing filter, the curve tends to shrink

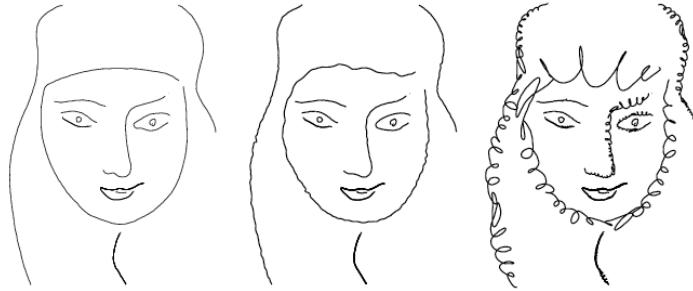


Figure 4.38 Changing the character of a curve without affecting its sweep (Finkelstein and Salesin 1994) © 1994 ACM: higher frequency wavelets can be replaced with exemplars from a style library to effect different local appearances.

on itself (Figure 4.37a). Lowe (1989) and Taubin (1995) describe techniques that compensate for this shrinkage by adding an offset term based on second derivative estimates or a larger smoothing kernel (Figure 4.37b). An alternative approach, based on selectively modifying different frequencies in a wavelet decomposition, is presented by Finkelstein and Salesin (1994). In addition to controlling shrinkage without affecting its “sweep”, wavelets allow the “character” of a curve to be interactively modified, as shown in Figure 4.38.

The evolution of curves as they are smoothed and simplified is related to “grassfire” (distance) transforms and region skeletons (Section 3.3.3) (Tek and Kimia 2003), and can be used to recognize objects based on their contour shape (Sebastian and Kimia 2005). More local descriptors of curve shape such as *shape contexts* (Belongie, Malik, and Puzicha 2002) can also be used for recognition and are potentially more robust to missing parts due to occlusions.

The field of contour detection and linking continues to evolve rapidly and now includes techniques for global contour grouping, boundary completion, and junction detection (Maire, Arbelaez, Fowlkes *et al.* 2008), as well as grouping contours into likely regions (Arbeláez, Maire, Fowlkes *et al.* 2010) and wide-baseline correspondence (Meltzer and Soatto 2008).

4.2.3 Application: Edge editing and enhancement

While edges can serve as components for object recognition or features for matching, they can also be used directly for image editing.

In fact, if the edge magnitude and blur estimate are kept along with each edge, a visually similar image can be reconstructed from this information (Elder 1999). Based on this principle, Elder and Goldberg (2001) propose a system for “image editing in the contour domain”. Their system allows users to selectively remove edges corresponding to unwanted features such as specularities, shadows, or distracting visual elements. After reconstructing the image from the remaining edges, the undesirable visual features have been removed (Figure 4.39).

Another potential application is to enhance perceptually salient edges while simplifying the underlying image to produce a cartoon-like or “pen-and-ink” stylized image (DeCarlo and Santella 2002). This application is discussed in more detail in Section 10.5.2.

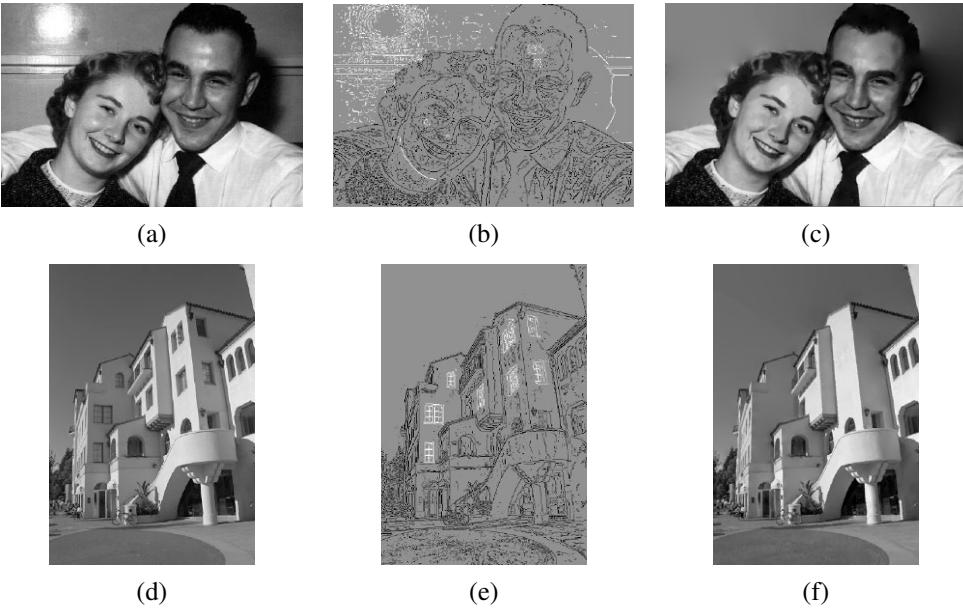


Figure 4.39 Image editing in the contour domain (Elder and Goldberg 2001) © 2001 IEEE: (a) and (d) original images; (b) and (e) extracted edges (edges to be deleted are marked in white); (c) and (f) reconstructed edited images.

4.3 Lines

While edges and general curves are suitable for describing the contours of natural objects, the man-made world is full of straight lines. Detecting and matching these lines can be useful in a variety of applications, including architectural modeling, pose estimation in urban environments, and the analysis of printed document layouts.

In this section, we present some techniques for extracting *piecewise linear* descriptions from the curves computed in the previous section. We begin with some algorithms for approximating a curve as a piecewise-linear polyline. We then describe the *Hough transform*, which can be used to group edgels into line segments even across gaps and occlusions. Finally, we describe how 3D lines with common *vanishing points* can be grouped together. These vanishing points can be used to calibrate a camera and to determine its orientation relative to a rectahedral scene, as described in Section 6.3.2.

4.3.1 Successive approximation

As we saw in Section 4.2.2, describing a curve as a series of 2D locations $\mathbf{x}_i = \mathbf{x}(s_i)$ provides a general representation suitable for matching and further processing. In many applications, however, it is preferable to approximate such a curve with a simpler representation, e.g., as a piecewise-linear polyline or as a B-spline curve (Farin 1996), as shown in Figure 4.40.

Many techniques have been developed over the years to perform this approximation, which is also known as *line simplification*. One of the oldest, and simplest, is the one proposed by Ramer (1972) and Douglas and Peucker (1973), who recursively subdivide the curve at

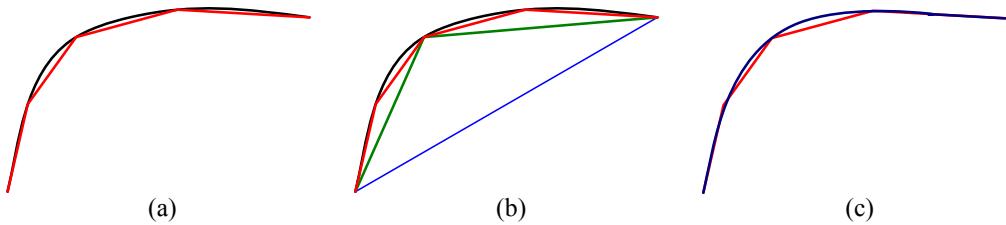


Figure 4.40 Approximating a curve (shown in black) as a polyline or B-spline: (a) original curve and a polyline approximation shown in red; (b) successive approximation by recursively finding points furthest away from the current approximation; (c) smooth interpolating spline, shown in dark blue, fit to the polyline vertices.

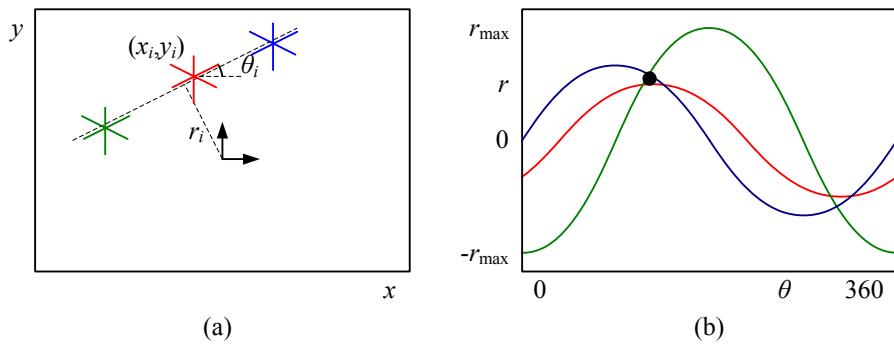


Figure 4.41 Original Hough transform: (a) each point votes for a complete family of potential lines $r_i(\theta) = x_i \cos \theta + y_i \sin \theta$; (b) each pencil of lines sweeps out a sinusoid in (r, θ) ; their intersection provides the desired line equation.

the point furthest away from the line joining the two endpoints (or the current coarse polyline approximation), as shown in Figure 4.40. Hershberger and Snoeyink (1992) provide a more efficient implementation and also cite some of the other related work in this area.

Once the line simplification has been computed, it can be used to approximate the original curve. If a smoother representation or visualization is desired, either approximating or interpolating splines or curves can be used (Sections 3.5.1 and 5.1.1) (Szeliski and Ito 1986; Bartels, Beatty, and Barsky 1987; Farin 1996), as shown in Figure 4.40c.

4.3.2 Hough transforms

While curve approximation with polylines can often lead to successful line extraction, lines in the real world are sometimes broken up into disconnected components or made up of many collinear line segments. In many cases, it is desirable to group such collinear segments into extended lines. At a further processing stage (described in Section 4.3.3), we can then group such lines into collections with common vanishing points.

The Hough transform, named after its original inventor (Hough 1962), is a well-known technique for having edges “vote” for plausible line locations (Duda and Hart 1972; Ballard 1981; Illingworth and Kittler 1988). In its original formulation (Figure 4.41), each edge point votes for *all* possible lines passing through it, and lines corresponding to high *accumulator* or

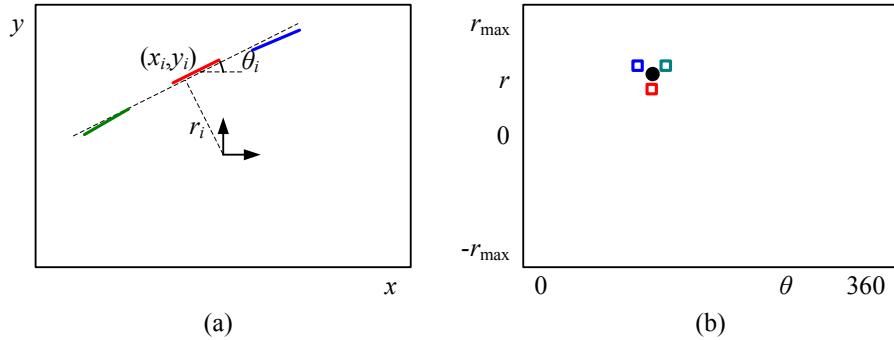


Figure 4.42 Oriented Hough transform: (a) an edgel re-parameterized in polar (r, θ) coordinates, with $\hat{\mathbf{n}}_i = (\cos \theta_i, \sin \theta_i)$ and $r_i = \hat{\mathbf{n}}_i \cdot \mathbf{x}_i$; (b) (r, θ) accumulator array, showing the votes for the three edgels marked in red, green, and blue.

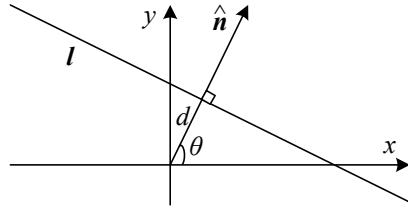


Figure 4.43 2D line equation expressed in terms of the normal $\hat{\mathbf{n}}$ and distance to the origin d .

bin values are examined for potential line fits.⁹ Unless the points on a line are truly punctate, a better approach (in my experience) is to use the local orientation information at each edgel to vote for a *single* accumulator cell (Figure 4.42), as described below. A hybrid strategy, where each edgel votes for a number of possible orientation or location pairs centered around the estimate orientation, may be desirable in some cases.

Before we can vote for line hypotheses, we must first choose a suitable representation. Figure 4.43 (copied from Figure 2.2a) shows the normal-distance ($\hat{\mathbf{n}}, d$) parameterization for a line. Since lines are made up of edge segments, we adopt the convention that the line normal $\hat{\mathbf{n}}$ points in the same direction (i.e., has the same sign) as the image gradient $\mathbf{J}(\mathbf{x}) = \nabla I(\mathbf{x})$ (4.19). To obtain a minimal two-parameter representation for lines, we convert the normal vector into an angle

$$\theta = \tan^{-1} n_y / n_x, \quad (4.26)$$

as shown in Figure 4.43. The range of possible (θ, d) values is $[-180^\circ, 180^\circ] \times [-\sqrt{2}, \sqrt{2}]$, assuming that we are using normalized pixel coordinates (2.61) that lie in $[-1, 1]$. The number of bins to use along each axis depends on the accuracy of the position and orientation estimate available at each edgel and the expected line density, and is best set experimentally with some test runs on sample imagery.

Given the line parameterization, the Hough transform proceeds as shown in Algorithm 4.2.

⁹ The Hough transform can also be *generalized* to look for other geometric features such as circles (Ballard 1981), but we do not cover such extensions in this book.

procedure $Hough(\{(x, y, \theta)\})$:

1. Clear the accumulator array.
2. For each detected edgel at location (x, y) and orientation $\theta = \tan^{-1} n_y/n_x$, compute the value of
$$d = x n_x + y n_y$$
and increment the accumulator corresponding to (θ, d) .
3. Find the peaks in the accumulator corresponding to lines.
4. Optionally re-fit the lines to the constituent edgels.

Algorithm 4.2 Outline of a Hough transform algorithm based on oriented edge segments.

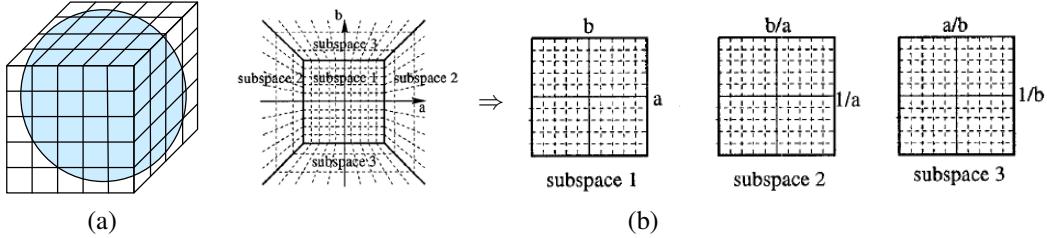


Figure 4.44 Cube map representation for line equations and vanishing points: (a) a cube map surrounding the unit sphere; (b) projecting the half-cube onto three subspaces (Tuytelaars, Van Gool, and Proesmans 1997) © 1997 IEEE.

Note that the original formulation of the Hough transform, which assumed no knowledge of the edgel orientation θ , has an additional loop inside Step 2 that iterates over all possible values of θ and increments a whole series of accumulators.

There are a lot of details in getting the Hough transform to work well, but these are best worked out by writing an implementation and testing it out on sample data. Exercise 4.12 describes some of these steps in more detail, including using edge segment lengths or strengths during the voting process, keeping a list of constituent edgels in the accumulator array for easier post-processing, and optionally combining edges of different “polarity” into the same line segments.

An alternative to the 2D polar (θ, d) representation for lines is to use the full 3D $\mathbf{m} = (\hat{\mathbf{n}}, d)$ line equation, projected onto the unit sphere. While the sphere can be parameterized using spherical coordinates (2.8),

$$\hat{\mathbf{m}} = (\cos \theta \cos \phi, \sin \theta \cos \phi, \sin \phi), \quad (4.27)$$

this does not uniformly sample the sphere and still requires the use of trigonometry.

An alternative representation can be obtained by using a *cube map*, i.e., projecting \mathbf{m} onto the face of a unit cube (Figure 4.44a). To compute the cube map coordinate of a 3D vector \mathbf{m} , first find the largest (absolute value) component of \mathbf{m} , i.e., $m = \pm \max(|n_x|, |n_y|, |d|)$,

and use this to select one of the six cube faces. Divide the remaining two coordinates by m and use these as indices into the cube face. While this avoids the use of trigonometry, it does require some decision logic.

One advantage of using the cube map, first pointed out by Tuytelaars, Van Gool, and Proesmans (1997), is that all of the lines passing through a point correspond to line segments on the cube faces, which is useful if the original (full voting) variant of the Hough transform is being used. In their work, they represent the line equation as $ax + b + y = 0$, which does not treat the x and y axes symmetrically. Note that if we restrict $d \geq 0$ by ignoring the polarity of the edge orientation (gradient sign), we can use a half-cube instead, which can be represented using only three cube faces, as shown in Figure 4.44b (Tuytelaars, Van Gool, and Proesmans 1997).

RANSAC-based line detection. Another alternative to the Hough transform is the RANdom SAmple Consensus (RANSAC) algorithm described in more detail in Section 6.1.4. In brief, RANSAC randomly chooses pairs of edgels to form a line hypothesis and then tests how many other edgels fall onto this line. (If the edge orientations are accurate enough, a single edgel can produce this hypothesis.) Lines with sufficiently large numbers of *inliers* (matching edgels) are then selected as the desired line segments.

An advantage of RANSAC is that no accumulator array is needed and so the algorithm can be more space efficient and potentially less prone to the choice of bin size. The disadvantage is that many more hypotheses may need to be generated and tested than those obtained by finding peaks in the accumulator array.

In general, there is no clear consensus on which line estimation technique performs best. It is therefore a good idea to think carefully about the problem at hand and to implement several approaches (successive approximation, Hough, and RANSAC) to determine the one that works best for your application.

4.3.3 Vanishing points

In many scenes, structurally important lines have the same vanishing point because they are parallel in 3D. Examples of such lines are horizontal and vertical building edges, zebra crossings, railway tracks, the edges of furniture such as tables and dressers, and of course, the ubiquitous calibration pattern (Figure 4.45). Finding the vanishing points common to such line sets can help refine their position in the image and, in certain cases, help determine the intrinsic and extrinsic orientation of the camera (Section 6.3.2).

Over the years, a large number of techniques have been developed for finding vanishing points, including (Quan and Mohr 1989; Collins and Weiss 1990; Brillaut-O’Mahoney 1991; McLean and Kotturi 1995; Becker and Bove 1995; Shufelt 1999; Tuytelaars, Van Gool, and Proesmans 1997; Schaffalitzky and Zisserman 2000; Antone and Teller 2002; Rother 2002; Košeká and Zhang 2005; Pflugfelder 2008; Tardif 2009)—see some of the more recent papers for additional references. In this section, we present a simple Hough technique based on having line pairs vote for potential vanishing point locations, followed by a robust least squares fitting stage. For alternative approaches, please see some of the more recent papers listed above.

The first stage in my vanishing point detection algorithm uses a Hough transform to accumulate votes for likely vanishing point candidates. As with line fitting, one possible approach

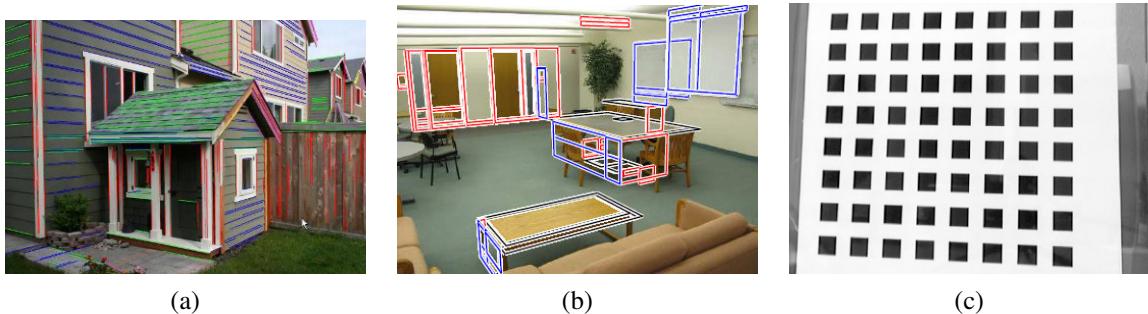


Figure 4.45 Real-world vanishing points: (a) architecture (Sinha, Steedly, Szeliski *et al.* 2008), (b) furniture (Mičušík, Wildenauer, and Košecká 2008) © 2008 IEEE, and (c) calibration patterns (Zhang 2000).

is to have each line vote for *all* possible vanishing point directions, either using a cube map (Tuytelaars, Van Gool, and Proesmans 1997; Antone and Teller 2002) or a Gaussian sphere (Collins and Weiss 1990), optionally using knowledge about the uncertainty in the vanishing point location to perform a weighted vote (Collins and Weiss 1990; Brillaut-O’Mahoney 1991; Shufelt 1999). My preferred approach is to use pairs of detected line segments to form candidate vanishing point locations. Let $\hat{\mathbf{m}}_i$ and $\hat{\mathbf{m}}_j$ be the (unit norm) line equations for a pair of line segments and l_i and l_j be their corresponding segment lengths. The location of the corresponding vanishing point hypothesis can be computed as

$$\mathbf{v}_{ij} = \hat{\mathbf{m}}_i \times \hat{\mathbf{m}}_j \quad (4.28)$$

and the corresponding weight set to

$$w_{ij} = \|\mathbf{v}_{ij}\| l_i l_j. \quad (4.29)$$

This has the desirable effect of downweighting (near-)collinear line segments and short line segments. The Hough space itself can either be represented using spherical coordinates (4.27) or as a cube map (Figure 4.44a).

Once the Hough accumulator space has been populated, peaks can be detected in a manner similar to that previously discussed for line detection. Given a set of candidate line segments that voted for a vanishing point, which can optionally be kept as a list at each Hough accumulator cell, I then use a robust least squares fit to estimate a more accurate location for each vanishing point.

Consider the relationship between the two line segment endpoints $\{\mathbf{p}_{i0}, \mathbf{p}_{i1}\}$ and the vanishing point \mathbf{v} , as shown in Figure 4.46. The area A of the triangle given by these three points, which is the magnitude of their triple product

$$A_i = |(\mathbf{p}_{i0} \times \mathbf{p}_{i1}) \cdot \mathbf{v}|, \quad (4.30)$$

is proportional to the perpendicular distance d_1 between each endpoint and the line through \mathbf{v} and the other endpoint, as well as the distance between \mathbf{p}_{i0} and \mathbf{v} . Assuming that the accuracy of a fitted line segment is proportional to its endpoint accuracy (Exercise 4.13), this therefore serves as an optimal metric for how well a vanishing point fits a set of extracted lines (Leibowitz (2001, Section 3.6.1) and Pflugfelder (2008, Section 2.1.1.3)). A robustified

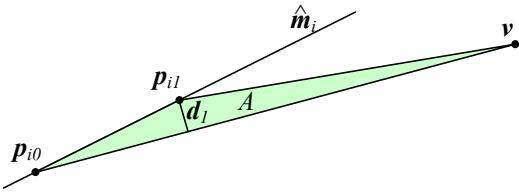


Figure 4.46 Triple product of the line segments endpoints p_{i0} and p_{i1} and the vanishing point v . The area A is proportional to the perpendicular distance d_1 and the distance between the other endpoint p_{i0} and the vanishing point.

least squares estimate (Appendix B.3) for the vanishing point can therefore be written as

$$\mathcal{E} = \sum_i \rho(A_i) = \mathbf{v}^T \left(\sum_i w_i(A_i) \mathbf{m}_i \mathbf{m}_i^T \right) \mathbf{v} = \mathbf{v}^T \mathbf{M} \mathbf{v}, \quad (4.31)$$

where $\mathbf{m}_i = \mathbf{p}_{i0} \times \mathbf{p}_{i1}$ is the segment line equation weighted by its length l_i , and $w_i = \rho'(A_i)/A_i$ is the *influence* of each robustified (reweighted) measurement on the final error (Appendix B.3). Notice how this metric is closely related to the original formula for the pairwise weighted Hough transform accumulation step. The final desired value for \mathbf{v} is computed as the least eigenvector of \mathbf{M} .

While the technique described above proceeds in two discrete stages, better results may be obtained by alternating between assigning lines to vanishing points and refitting the vanishing point locations (Antone and Teller 2002; Košecká and Zhang 2005; Pflugfelder 2008). The results of detecting individual vanishing points can also be made more robust by simultaneously searching for pairs or triplets of mutually orthogonal vanishing points (Shufelt 1999; Antone and Teller 2002; Rother 2002; Sinha, Steedly, Szeliski *et al.* 2008). Some results of such vanishing point detection algorithms can be seen in Figure 4.45.

4.3.4 Application: Rectangle detection

Once sets of mutually orthogonal vanishing points have been detected, it now becomes possible to search for 3D rectangular structures in the image (Figure 4.47). Over the last decade, a variety of techniques have been developed to find such rectangles, primarily focused on architectural scenes (Košecká and Zhang 2005; Han and Zhu 2005; Shaw and Barnes 2006; Mičušík, Wildenauer, and Košecká 2008; Schindler, Krishnamurthy, Lublinerman *et al.* 2008).

After detecting orthogonal vanishing directions, Košecká and Zhang (2005) refine the fitted line equations, search for corners near line intersections, and then verify rectangle hypotheses by rectifying the corresponding patches and looking for a preponderance of horizontal and vertical edges (Figures 4.47a–b). In follow-on work, Mičušík, Wildenauer, and Košecká (2008) use a Markov random field (MRF) to disambiguate between potentially overlapping rectangle hypotheses. They also use a plane sweep algorithm to match rectangles between different views (Figures 4.47d–f).

A different approach is proposed by Han and Zhu (2005), who use a grammar of potential rectangle shapes and nesting structures (between rectangles and vanishing points) to infer the most likely assignment of line segments to rectangles (Figure 4.47c).

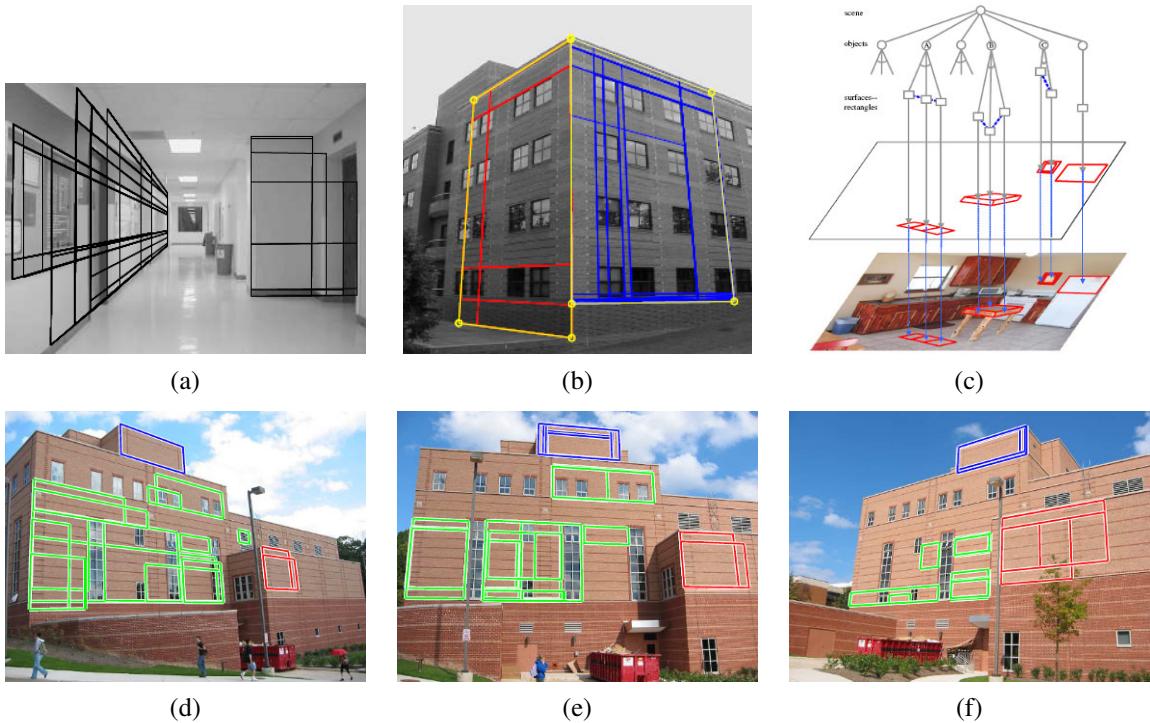


Figure 4.47 Rectangle detection: (a) indoor corridor and (b) building exterior with grouped facades (Košecká and Zhang 2005) © 2005 Elsevier; (c) grammar-based recognition (Han and Zhu 2005) © 2005 IEEE; (d–f) rectangle matching using a plane sweep algorithm (Mičušík, Wildenauer, and Košecká 2008) © 2008 IEEE.

4.4 Additional reading

One of the seminal papers on feature detection, description, and matching is by Lowe (2004). Comprehensive surveys and evaluations of such techniques have been made by Schmid, Mohr, and Bauckhage (2000); Mikolajczyk and Schmid (2005); Mikolajczyk, Tuytelaars, Schmid *et al.* (2005); Tuytelaars and Mikolajczyk (2007) while Shi and Tomasi (1994) and Triggs (2004) also provide nice reviews.

In the area of feature detectors (Mikolajczyk, Tuytelaars, Schmid *et al.* 2005), in addition to such classic approaches as Förstner–Harris (Förstner 1986; Harris and Stephens 1988) and difference of Gaussians (Lindeberg 1993, 1998b; Lowe 2004), maximally stable extremal regions (MSERs) are widely used for applications that require affine invariance (Matas, Chum, Urban *et al.* 2004; Nistér and Stewénius 2008). More recent interest point detectors are discussed by Xiao and Shah (2003); Kothe (2003); Carneiro and Jepson (2005); Kenney, Zuliani, and Manjunath (2005); Bay, Tuytelaars, and Van Gool (2006); Platel, Balmachnova, Florack *et al.* (2006); Rosten and Drummond (2006), as well as techniques based on line matching (Zoghliami, Faugeras, and Deriche 1997; Bartoli, Coquerelle, and Sturm 2004) and region detection (Kadir, Zisserman, and Brady 2004; Matas, Chum, Urban *et al.* 2004; Tuytelaars and Van Gool 2004; Corso and Hager 2005).

A variety of local feature descriptors (and matching heuristics) are surveyed and compared by Mikolajczyk and Schmid (2005). More recent publications in this area include

those by van de Weijer and Schmid (2006); Abdel-Hakim and Farag (2006); Winder and Brown (2007); Hua, Brown, and Winder (2007). Techniques for efficiently matching features include k-d trees (Beis and Lowe 1999; Lowe 2004; Muja and Lowe 2009), pyramid matching kernels (Grauman and Darrell 2005), metric (vocabulary) trees (Nistér and Stewénius 2006), and a variety of multi-dimensional hashing techniques (Shakhnarovich, Viola, and Darrell 2003; Torralba, Weiss, and Fergus 2008; Weiss, Torralba, and Fergus 2008; Kulis and Grauman 2009; Raginsky and Lazebnik 2009).

The classic reference on feature detection and tracking is (Shi and Tomasi 1994). More recent work in this field has focused on learning better matching functions for specific features (Avidan 2001; Jurie and Dhome 2002; Williams, Blake, and Cipolla 2003; Lepetit and Fua 2005; Lepetit, Pilet, and Fua 2006; Hinterstoisser, Benhimane, Navab *et al.* 2008; Rogez, Rihan, Ramalingam *et al.* 2008; Özuyusal, Calonder, Lepetit *et al.* 2010).

A highly cited and widely used edge detector is the one developed by Canny (1986). Alternative edge detectors as well as experimental comparisons can be found in publications by Nalwa and Binford (1986); Nalwa (1987); Deriche (1987); Freeman and Adelson (1991); Nalwa (1993); Heath, Sarkar, Sanocki *et al.* (1998); Crane (1997); Ritter and Wilson (2000); Bowyer, Kranenburg, and Dougherty (2001); Arbeláez, Maire, Fowlkes *et al.* (2010). The topic of scale selection in edge detection is nicely treated by Elder and Zucker (1998), while approaches to color and texture edge detection can be found in (Ruzon and Tomasi 2001; Martin, Fowlkes, and Malik 2004; Gevers, van de Weijer, and Stokman 2006). Edge detectors have also recently been combined with region segmentation techniques to further improve the detection of semantically salient boundaries (Maire, Arbelaez, Fowlkes *et al.* 2008; Arbeláez, Maire, Fowlkes *et al.* 2010). Edges linked into contours can be smoothed and manipulated for artistic effect (Lowe 1989; Finkelstein and Salesin 1994; Taubin 1995) and used for recognition (Belongie, Malik, and Puzicha 2002; Tek and Kimia 2003; Sebastian and Kimia 2005).

An early, well-regarded paper on straight line extraction in images was written by Burns, Hanson, and Riseman (1986). More recent techniques often combine line detection with vanishing point detection (Quan and Mohr 1989; Collins and Weiss 1990; Brillaut-O’Mahoney 1991; McLean and Kotturi 1995; Becker and Bove 1995; Shufelt 1999; Tuytelaars, Van Gool, and Proesmans 1997; Schaffalitzky and Zisserman 2000; Antone and Teller 2002; Rother 2002; Košecká and Zhang 2005; Pflugfelder 2008; Sinha, Steedly, Szeliski *et al.* 2008; Tardif 2009).

4.5 Exercises

Ex 4.1: Interest point detector Implement one or more keypoint detectors and compare their performance (with your own or with a classmate’s detector).

Possible detectors:

- Laplacian or Difference of Gaussian;
- Förstner–Harris Hessian (try different formula variants given in (4.9–4.11));
- oriented/steerable filter, looking for either second-order high second response or two edges in a window (Koethe 2003), as discussed in Section 4.1.1.

Other detectors are described by Mikolajczyk, Tuytelaars, Schmid *et al.* (2005); Tuytelaars and Mikolajczyk (2007). Additional optional steps could include:

1. Compute the detections on a sub-octave pyramid and find 3D maxima.
2. Find local orientation estimates using steerable filter responses or a gradient histogramming method.
3. Implement non-maximal suppression, such as the adaptive technique of Brown, Szeliski, and Winder (2005).
4. Vary the window shape and size (pre-filter and aggregation).

To test for repeatability, download the code from <http://www.robots.ox.ac.uk/~vgg/research/affine/> (Mikolajczyk, Tuytelaars, Schmid *et al.* 2005; Tuytelaars and Mikolajczyk 2007) or simply rotate or shear your own test images. (Pick a domain you may want to use later, e.g., for outdoor stitching.)

Be sure to measure and report the stability of your scale and orientation estimates.

Ex 4.2: Interest point descriptor Implement one or more descriptors (steered to local scale and orientation) and compare their performance (with your own or with a classmate's detector).

Some possible descriptors include

- contrast-normalized patches (Brown, Szeliski, and Winder 2005);
- SIFT (Lowe 2004);
- GLOH (Mikolajczyk and Schmid 2005);
- DAISY (Winder and Brown 2007; Tola, Lepetit, and Fua 2010).

Other detectors are described by Mikolajczyk and Schmid (2005).

Ex 4.3: ROC curve computation Given a pair of curves (histograms) plotting the number of matching and non-matching features as a function of Euclidean distance d as shown in Figure 4.23b, derive an algorithm for plotting a ROC curve (Figure 4.23a). In particular, let $t(d)$ be the distribution of true matches and $f(d)$ be the distribution of (false) non-matches. Write down the equations for the ROC, i.e., TPR(FPR), and the AUC.

(Hint: Plot the cumulative distributions $T(d) = \int t(d)$ and $F(d) = \int f(d)$ and see if these help you derive the TPR and FPR at a given threshold θ .)

Ex 4.4: Feature matcher After extracting features from a collection of overlapping or distorted images,¹⁰ match them up by their descriptors either using nearest neighbor matching or a more efficient matching strategy such as a k-d tree.

See whether you can improve the accuracy of your matches using techniques such as the nearest neighbor distance ratio.

¹⁰ <http://www.robots.ox.ac.uk/~vgg/research/affine/>.

Ex 4.5: Feature tracker Instead of finding feature points independently in multiple images and then matching them, find features in the first image of a video or image sequence and then re-locate the corresponding points in the next frames using either search and gradient descent (Shi and Tomasi 1994) or learned feature detectors (Lepetit, Pilet, and Fua 2006; Fossati, Dimitrijevic, Lepetit *et al.* 2007). When the number of tracked points drops below a threshold or new regions in the image become visible, find additional points to track.

(Optional) Winnow out incorrect matches by estimating a homography (6.19–6.23) or fundamental matrix (Section 7.2.1).

(Optional) Refine the accuracy of your matches using the iterative registration algorithm described in Section 8.2 and Exercise 8.2.

Ex 4.6: Facial feature tracker Apply your feature tracker to tracking points on a person's face, either manually initialized to interesting locations such as eye corners or automatically initialized at interest points.

(Optional) Match features between two people and use these features to perform image morphing (Exercise 3.25).

Ex 4.7: Edge detector Implement an edge detector of your choice. Compare its performance to that of your classmates' detectors or code downloaded from the Internet.

A simple but well-performing sub-pixel edge detector can be created as follows:

1. Blur the input image a little,

$$B_\sigma(\mathbf{x}) = G_\sigma(\mathbf{x}) * I(\mathbf{x}).$$

2. Construct a Gaussian pyramid (Exercise 3.19),

$$P = \text{Pyramid}\{B_\sigma(\mathbf{x})\}$$

3. Subtract an interpolated coarser-level pyramid image from the original resolution blurred image,

$$S(\mathbf{x}) = B_\sigma(\mathbf{x}) - P.\text{InterpolatedLevel}(L).$$

4. For each quad of pixels, $\{(i, j), (i + 1, j), (i, j + 1), (i + 1, j + 1)\}$, count the number of zero crossings along the four edges.
5. When there are exactly two zero crossings, compute their locations using (4.25) and store these edgel endpoints along with the midpoint in the edgel structure (Figure 4.48).
6. For each edgel, compute the local gradient by taking the horizontal and vertical differences between the values of S along the zero crossing edges.
7. Store the magnitude of this gradient as the edge strength and either its orientation or that of the segment joining the edgel endpoints as the edge orientation.
8. Add the edgel to a list of edgelets or store it in a 2D array of edgelets (addressed by pixel coordinates).

Figure 4.48 shows a possible representation for each computed edgel.

```

struct SEdgel {
    float e[2][2];      // edgel endpoints (zero crossing)
    float x, y;         // sub-pixel edge position (midpoint)
    float n_x, n_y;     // orientation, as normal vector
    float theta;         // orientation, as angle (degrees)
    float length;        // length of edgel
    float strength;      // strength of edgel (gradient magnitude)
};

struct SLine : public SEdgel {
    float line_length;   // length of line (est. from ellipsoid)
    float sigma;          // estimated std. dev. of edgel noise
    float r;              // line equation: x * n_y - y * n_x = r
};

```

Figure 4.48 A potential C++ structure for edgel and line elements.

Ex 4.8: Edge linking and thresholding Link up the edges computed in the previous exercise into chains and optionally perform thresholding with hysteresis.

The steps may include:

1. Store the edgels either in a 2D array (say, an integer image with indices into the edgel list) or pre-sort the edgel list first by (integer) x coordinates and then y coordinates, for faster neighbor finding.
2. Pick up an edgel from the list of unlinked edgels and find its neighbors in both directions until no neighbor is found or a closed contour is obtained. Flag edgels as linked as you visit them and push them onto your list of linked edgels.
3. Alternatively, generalize a previously developed connected component algorithm (Exercise 3.14) to perform the linking in just two raster passes.
4. (Optional) Perform hysteresis-based thresholding (Canny 1986). Use two thresholds "hi" and "lo" for the edge strength. A candidate edgel is considered an edge if either its strength is above the "hi" threshold or its strength is above the "lo" threshold and it is (recursively) connected to a previously detected edge.
5. (Optional) Link together contours that have small gaps but whose endpoints have similar orientations.
6. (Optional) Find junctions between adjacent contours, e.g., using some of the ideas (or references) from Maire, Arbelaez, Fowlkes *et al.* (2008).

Ex 4.9: Contour matching Convert a closed contour (linked edgel list) into its arc-length parameterization and use this to match object outlines.

The steps may include:

1. Walk along the contour and create a list of (x_i, y_i, s_i) triplets, using the arc-length formula

$$s_{i+1} = s_i + \|\mathbf{x}_{i+1} - \mathbf{x}_i\|. \quad (4.32)$$

2. Resample this list onto a regular set of (x_j, y_j, j) samples using linear interpolation of each segment.
3. Compute the average values of x and y , i.e., \bar{x} and \bar{y} and subtract them from your sampled curve points.
4. Resample the original (x_i, y_i, s_i) piecewise-linear function onto a length-independent set of samples, say $j \in [0, 1023]$. (Using a length which is a power of two makes subsequent Fourier transforms more convenient.)
5. Compute the Fourier transform of the curve, treating each (x, y) pair as a complex number.
6. To compare two curves, fit a linear equation to the phase difference between the two curves. (Careful: phase wraps around at 360° . Also, you may wish to weight samples by their Fourier spectrum magnitude—see Section 8.1.2.)
7. (Optional) Prove that the constant phase component corresponds to the temporal shift in s , while the linear component corresponds to rotation.

Of course, feel free to try any other curve descriptor and matching technique from the computer vision literature (Tek and Kimia 2003; Sebastian and Kimia 2005).

Ex 4.10: Jigsaw puzzle solver—challenging Write a program to automatically solve a jigsaw puzzle from a set of scanned puzzle pieces. Your software may include the following components:

1. Scan the pieces (either face up or face down) on a flatbed scanner with a distinctively colored background.
2. (Optional) Scan in the box top to use as a low-resolution reference image.
3. Use color-based thresholding to isolate the pieces.
4. Extract the contour of each piece using edge finding and linking.
5. (Optional) Re-represent each contour using an arc-length or some other re-parameterization. Break up the contours into meaningful matchable pieces. (Is this hard?)
6. (Optional) Associate color values with each contour to help in the matching.
7. (Optional) Match pieces to the reference image using some rotationally invariant feature descriptors.
8. Solve a global optimization or (backtracking) search problem to snap pieces together and place them in the correct location relative to the reference image.
9. Test your algorithm on a succession of more difficult puzzles and compare your results with those of others.

Ex 4.11: Successive approximation line detector Implement a line simplification algorithm (Section 4.3.1) (Ramer 1972; Douglas and Peucker 1973) to convert a hand-drawn curve (or linked edge image) into a small set of polylines.

(Optional) Re-render this curve using either an approximating or interpolating spline or Bezier curve (Szeliski and Ito 1986; Bartels, Beatty, and Barsky 1987; Farin 1996).

Ex 4.12: Hough transform line detector Implement a Hough transform for finding lines in images:

1. Create an accumulator array of the appropriate user-specified size and clear it. The user can specify the spacing in degrees between orientation bins and in pixels between distance bins. The array can be allocated as integer (for simple counts), floating point (for weighted counts), or as an array of vectors for keeping back pointers to the constituent edges.
2. For each detected edgel at location (x, y) and orientation $\theta = \tan^{-1} n_y/n_x$, compute the value of

$$d = xn_x + yn_y \quad (4.33)$$

and increment the accumulator corresponding to (θ, d) .

(Optional) Weight the vote of each edge by its length (see Exercise 4.7) or the strength of its gradient.

3. (Optional) Smooth the scalar accumulator array by adding in values from its immediate neighbors. This can help counteract the *discretization* effect of voting for only a single bin—see Exercise 3.7.
4. Find the largest peaks (local maxima) in the accumulator corresponding to lines.
5. (Optional) For each peak, re-fit the lines to the constituent edgels, using *total least squares* (Appendix A.2). Use the original edgel lengths or strength weights to weight the least squares fit, as well as the agreement between the hypothesized line orientation and the edgel orientation. Determine whether these heuristics help increase the accuracy of the fit.
6. After fitting each peak, zero-out or eliminate that peak and its adjacent bins in the array, and move on to the next largest peak.

Test out your Hough transform on a variety of images taken indoors and outdoors, as well as checkerboard calibration patterns.

For checkerboard patterns, you can modify your Hough transform by collapsing *antipodal* bins ($\theta \pm 180^\circ, -d$) with (θ, d) to find lines that do not care about polarity changes. Can you think of examples in real-world images where this might be desirable as well?

Ex 4.13: Line fitting uncertainty Estimate the uncertainty (covariance) in your line fit using uncertainty analysis.

1. After determining which edgels belong to the line segment (using either successive approximation or Hough transform), re-fit the line segment using total least squares (Van Huffel and Vandewalle 1991; Van Huffel and Lemmerling 2002), i.e., find the

mean or centroid of the edgels and then use eigenvalue analysis to find the dominant orientation.

2. Compute the perpendicular errors (deviations) to the line and robustly estimate the variance of the fitting noise using an estimator such as MAD (Appendix B.3).
3. (Optional) re-fit the line parameters by throwing away outliers or using a robust norm or influence function.
4. Estimate the error in the perpendicular location of the line segment and its orientation.

Ex 4.14: Vanishing points Compute the vanishing points in an image using one of the techniques described in Section 4.3.3 and optionally refine the original line equations associated with each vanishing point. Your results can be used later to track a target (Exercise 6.5) or reconstruct architecture (Section 12.6.1).

Ex 4.15: Vanishing point uncertainty Perform an uncertainty analysis on your estimated vanishing points. You will need to decide how to represent your vanishing point, e.g., homogeneous coordinates on a sphere, to handle vanishing points near infinity.

See the discussion of Bingham distributions by Collins and Weiss (1990) for some ideas.

Chapter 5

Segmentation

5.1	Active contours	237
5.1.1	Snakes	238
5.1.2	Dynamic snakes and CONDENSATION	243
5.1.3	Scissors	246
5.1.4	Level Sets	248
5.1.5	<i>Application:</i> Contour tracking and rotoscoping	249
5.2	Split and merge	250
5.2.1	Watershed	251
5.2.2	Region splitting (divisive clustering)	251
5.2.3	Region merging (agglomerative clustering)	251
5.2.4	Graph-based segmentation	252
5.2.5	Probabilistic aggregation	253
5.3	Mean shift and mode finding	254
5.3.1	K-means and mixtures of Gaussians	256
5.3.2	Mean shift	257
5.4	Normalized cuts	260
5.5	Graph cuts and energy-based methods	264
5.5.1	<i>Application:</i> Medical image segmentation	268
5.6	Additional reading	268
5.7	Exercises	270



Figure 5.1 Some popular image segmentation techniques: (a) active contours (Isard and Blake 1998) © 1998 Springer; (b) level sets (Cremers, Rousson, and Deriche 2007) © 2007 Springer; (c) graph-based merging (Felzenszwalb and Huttenlocher 2004b) © 2004 Springer; (d) mean shift (Comaniciu and Meer 2002) © 2002 IEEE; (e) texture and intervening contour-based normalized cuts (Malik, Belongie, Leung *et al.* 2001) © 2001 Springer; (f) binary MRF solved using graph cuts (Boykov and Funka-Lea 2006) © 2006 Springer.

Image segmentation is the task of finding groups of pixels that “go together”. In statistics, this problem is known as *cluster analysis* and is a widely studied area with hundreds of different algorithms (Jain and Dubes 1988; Kaufman and Rousseeuw 1990; Jain, Duin, and Mao 2000; Jain, Topchy, Law *et al.* 2004).

In computer vision, image segmentation is one of the oldest and most widely studied problems (Brice and Fennema 1970; Pavlidis 1977; Riseman and Arbib 1977; Ohlander, Price, and Reddy 1978; Rosenfeld and Davis 1979; Haralick and Shapiro 1985). Early techniques tend to use region splitting or merging (Brice and Fennema 1970; Horowitz and Pavlidis 1976; Ohlander, Price, and Reddy 1978; Pavlidis and Liow 1990), which correspond to *divisive* and *agglomerative* algorithms in the clustering literature (Jain, Topchy, Law *et al.* 2004). More recent algorithms often optimize some global criterion, such as intra-region consistency and inter-region boundary lengths or dissimilarity (Leclerc 1989; Mumford and Shah 1989; Shi and Malik 2000; Comaniciu and Meer 2002; Felzenszwalb and Huttenlocher 2004b; Cremers, Rousson, and Deriche 2007).

We have already seen examples of image segmentation in Sections 3.3.2 and 3.7.2. In this chapter, we review some additional techniques that have been developed for image segmentation. These include algorithms based on active contours (Section 5.1) and level sets (Section 5.1.4), region splitting and merging (Section 5.2), *mean shift* (mode finding) (Section 5.3), *normalized cuts* (splitting based on pixel similarity metrics) (Section 5.4), and binary Markov random fields solved using graph cuts (Section 5.5). Figure 5.1 shows some examples of these techniques applied to different images.

Since the literature on image segmentation is so vast, a good way to get a handle on some of the better performing algorithms is to look at experimental comparisons on human-labeled databases (Arbeláez, Maire, Fowlkes *et al.* 2010). The best known of these is the Berkeley Segmentation Dataset and Benchmark¹ (Martin, Fowlkes, Tal *et al.* 2001), which consists of 1000 images from a Corel image dataset that were hand-labeled by 30 human subjects. Many of the more recent image segmentation algorithms report comparative results on this database. For example, Unnikrishnan, Pantofaru, and Hebert (2007) propose new metrics for comparing such algorithms. Estrada and Jepson (2009) compare four well-known segmentation algorithms on the Berkeley data set and conclude that while their own SE-MinCut algorithm (Estrada, Jepson, and Chennubhotla 2004) algorithm outperforms the others by a small margin, there still exists a wide gap between automated and human segmentation performance.² A new database of foreground and background segmentations, used by Alpert, Galun, Basri *et al.* (2007), is also available.³

5.1 Active contours

While lines, vanishing points, and rectangles are commonplace in the man-made world, curves corresponding to object boundaries are even more common, especially in the natural environment. In this section, we describe three related approaches to locating such boundary curves in images.

¹ <http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/>

² An interesting observation about their ROC plots is that automated techniques cluster tightly along similar curves, but human performance is all over the map.

³ http://www.wisdom.weizmann.ac.il/~vision/Seg_Evaluation_DB/index.html

The first, originally called *snakes* by its inventors (Kass, Witkin, and Terzopoulos 1988) (Section 5.1.1), is an energy-minimizing, two-dimensional spline curve that evolves (moves) towards image features such as strong edges. The second, *intelligent scissors* (Mortensen and Barrett 1995) (Section 5.1.3), allow the user to sketch in real time a curve that clings to object boundaries. Finally, *level set* techniques (Section 5.1.4) evolve the curve as the zero-set of a *characteristic function*, which allows them to easily change topology and incorporate region-based statistics.

All three of these are examples of *active contours* (Blake and Isard 1998; Mortensen 1999), since these boundary detectors iteratively move towards their final solution under the combination of image and optional user-guidance forces.

5.1.1 Snakes

Snakes are a two-dimensional generalization of the 1D energy-minimizing splines first introduced in Section 3.7.1,

$$\mathcal{E}_{\text{int}} = \int \alpha(s) \|\mathbf{f}_s(s)\|^2 + \beta(s) \|\mathbf{f}_{ss}(s)\|^2 ds, \quad (5.1)$$

where s is the arc-length along the curve $\mathbf{f}(s) = (x(s), y(s))$ and $\alpha(s)$ and $\beta(s)$ are first- and second-order continuity weighting functions analogous to the $s(x, y)$ and $c(x, y)$ terms introduced in (3.100–3.101). We can discretize this energy by sampling the initial curve position evenly along its length (Figure 4.35) to obtain

$$\begin{aligned} E_{\text{int}} &= \sum_i \alpha(i) \|f(i+1) - f(i)\|^2 / h^2 \\ &\quad + \beta(i) \|f(i+1) - 2f(i) + f(i-1)\|^2 / h^4, \end{aligned} \quad (5.2)$$

where h is the step size, which can be neglected if we resample the curve along its arc-length after each iteration.

In addition to this *internal* spline energy, a snake simultaneously minimizes external image-based and constraint-based potentials. The image-based potentials are the sum of several terms

$$\mathcal{E}_{\text{image}} = w_{\text{line}} \mathcal{E}_{\text{line}} + w_{\text{edge}} \mathcal{E}_{\text{edge}} + w_{\text{term}} \mathcal{E}_{\text{term}}, \quad (5.3)$$

where the *line* term attracts the snake to dark ridges, the *edge* term attracts it to strong gradients (edges), and the *term* term attracts it to line terminations. In practice, most systems only use the edge term, which can either be directly proportional to the image gradients,

$$E_{\text{edge}} = \sum_i -\|\nabla I(\mathbf{f}(i))\|^2, \quad (5.4)$$

or to a smoothed version of the image Laplacian,

$$E_{\text{edge}} = \sum_i -|(G_\sigma * \nabla^2 I)(\mathbf{f}(i))|^2. \quad (5.5)$$

People also sometimes extract edges and then use a distance map to the edges as an alternative to these two originally proposed potentials.

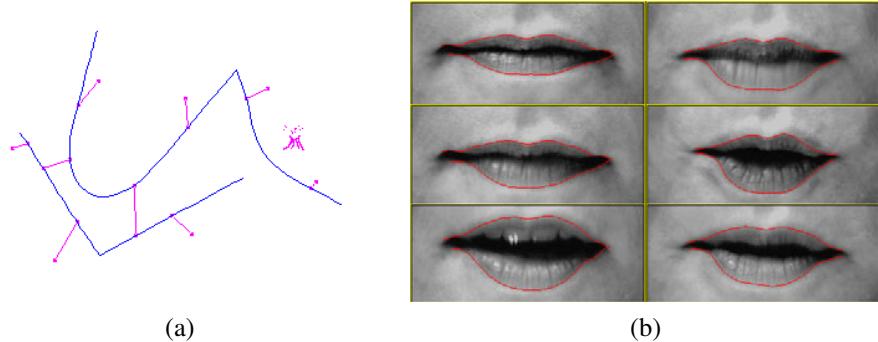


Figure 5.2 Snakes (Kass, Witkin, and Terzopoulos 1988) © 1988 Springer: (a) the “snake pit” for interactively controlling shape; (b) lip tracking.

In interactive applications, a variety of user-placed constraints can also be added, e.g., attractive (spring) forces towards anchor points $d(i)$,

$$E_{\text{spring}} = k_i \|\mathbf{f}(i) - \mathbf{d}(i)\|^2, \quad (5.6)$$

as well as repulsive $1/r$ (“volcano”) forces (Figure 5.2a). As the snakes evolve by minimizing their energy, they often “wiggle” and “slither”, which accounts for their popular name. Figure 5.2b shows snakes being used to track a person’s lips.

Because regular snakes have a tendency to shrink (Exercise 5.1), it is usually better to initialize them by drawing the snake outside the object of interest to be tracked. Alternatively, an expansion *ballooning* force can be added to the dynamics (Cohen and Cohen 1993), essentially moving each point outwards along its normal.

To efficiently solve the sparse linear system arising from snake energy minimization, a sparse direct solver (Appendix A.4) can be used, since the linear system is essentially pentadiagonal.⁴ Snake evolution is usually implemented as an alternation between this linear system solution and the linearization of non-linear constraints such as edge energy. A more direct way to find a global energy minimum is to use dynamic programming (Amini, Weymouth, and Jain 1990; Williams and Shah 1992), but this is not often used in practice, since it has been superseded by even more efficient or interactive algorithms such as intelligent scissors (Section 5.1.3) and GrabCut (Section 5.5).

Elastic nets and slippery springs

An interesting variant on snakes, first proposed by Durbin and Willshaw (1987) and later re-formulated in an energy-minimizing framework by Durbin, Szeliski, and Yuille (1989), is the *elastic net* formulation of the Traveling Salesman Problem (TSP). Recall that in a TSP, the salesman must visit each city once while minimizing the total distance traversed. A snake that is constrained to pass through each city could solve this problem (without any optimality guarantees) but it is impossible to tell ahead of time which snake control point should be associated with each city.

⁴ A closed snake has a Toeplitz matrix form, which can still be factored and solved in $O(N)$ time.

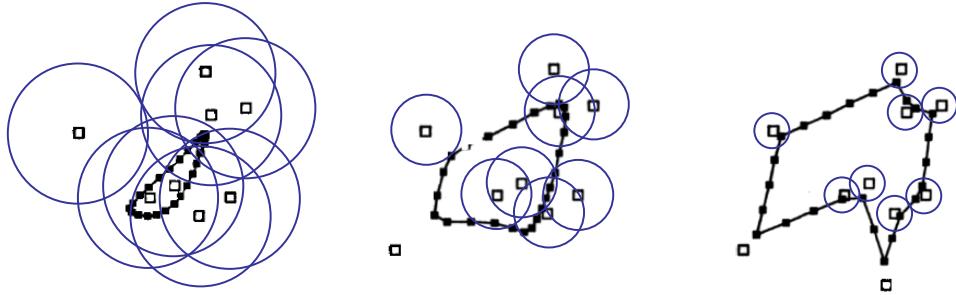


Figure 5.3 Elastic net: The open squares indicate the cities and the closed squares linked by straight line segments are the tour points. The blue circles indicate the approximate extent of the attraction force of each city, which is reduced over time. Under the Bayesian interpretation of the elastic net, the blue circles correspond to one standard deviation of the circular Gaussian that generates each city from some unknown tour point.

Instead of having a fixed constraint between snake nodes and cities, as in (5.6), a city is assumed to pass near *some* point along the tour (Figure 5.3). In a probabilistic interpretation, each city is generated as a *mixture* of Gaussians centered at each tour point,

$$p(\mathbf{d}(j)) = \sum_i p_{ij} \text{ with } p_{ij} = e^{-d_{ij}^2/(2\sigma^2)} \quad (5.7)$$

where σ is the standard deviation of the Gaussian and

$$d_{ij} = \|\mathbf{f}(i) - \mathbf{d}(j)\| \quad (5.8)$$

is the Euclidean distance between a tour point $\mathbf{f}(i)$ and a city location $\mathbf{d}(j)$. The corresponding data fitting energy (negative log likelihood) is

$$E_{\text{slippery}} = - \sum_j \log p(\mathbf{d}(j)) = - \sum_j \log \left[\sum_i e^{-\|\mathbf{f}(i) - \mathbf{d}(j)\|^2 / 2\sigma^2} \right]. \quad (5.9)$$

This energy derives its name from the fact that, unlike a regular spring, which couples a given snake point to a given constraint (5.6), this alternative energy defines a *slippery spring* that allows the association between constraints (cities) and curve (tour) points to evolve over time (Szeliski 1989). Note that this is a soft variant of the popular *iterated closest point* data constraint that is often used in fitting or aligning surfaces to data points or to each other (Section 12.2.1) (Besl and McKay 1992; Zhang 1994).

To compute a good solution to the TSP, the slippery spring data association energy is combined with a regular first-order internal smoothness energy (5.3) to define the cost of a tour. The tour $\mathbf{f}(s)$ is initialized as a small circle around the mean of the city points and σ is progressively lowered (Figure 5.3). For large σ values, the tour tries to stay near the centroid of the points but as σ decreases each city pulls more and more strongly on its closest tour points (Durbin, Szeliski, and Yuille 1989). In the limit as $\sigma \rightarrow 0$, each city is guaranteed to capture at least one tour point and the tours between subsequent cities become straight lines.

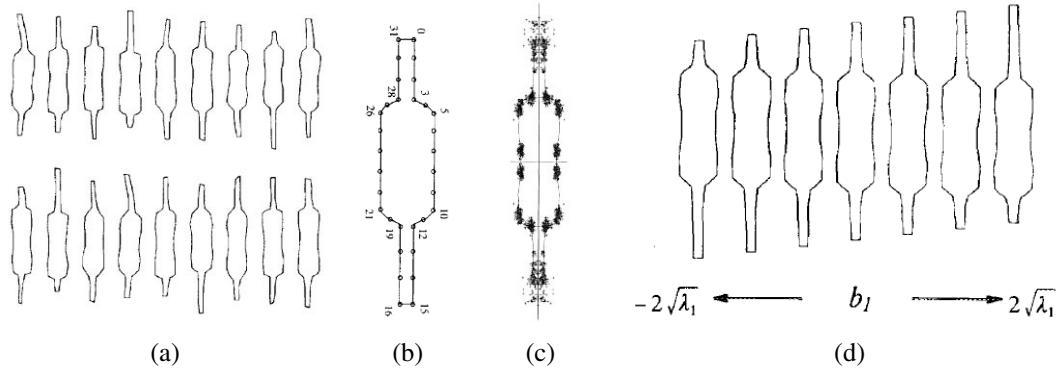


Figure 5.4 Point distribution model for a set of resistors (Cootes, Cooper, Taylor *et al.* 1995) © 1995 Elsevier:
(a) set of input resistor shapes; (b) assignment of control points to the boundary; (c) distribution (scatter plot) of point locations; (d) first (largest) mode of variation in the ensemble shapes.

Splines and shape priors

While snakes can be very good at capturing the fine and irregular detail in many real-world contours, they sometimes exhibit too many degrees of freedom, making it more likely that they can get trapped in local minima during their evolution.

One solution to this problem is to control the snake with fewer degrees of freedom through the use of B-spline approximations (Menet, Saint-Marc, and Medioni 1990b,a; Cipolla and Blake 1990). The resulting *B-snake* can be written as

$$\mathbf{f}(s) = \sum_k B_k(s) \mathbf{x}_k \quad (5.10)$$

or in discrete form as

$$\mathbf{F} = \mathbf{B} \mathbf{X} \quad (5.11)$$

with

$$\mathbf{F} = \begin{bmatrix} \mathbf{f}^T(0) \\ \vdots \\ \mathbf{f}^T(N) \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} B_0(s_0) & \dots & B_K(s_0) \\ \vdots & \ddots & \vdots \\ B_0(s_N) & \dots & B_K(s_N) \end{bmatrix}, \quad \text{and} \quad \mathbf{X} = \begin{bmatrix} \mathbf{x}^T(0) \\ \vdots \\ \mathbf{x}^T(K) \end{bmatrix}. \quad (5.12)$$

If the object being tracked or recognized has large variations in location, scale, or orientation, these can be modeled as an additional transformation on the control points, e.g., $\mathbf{x}'_k = s \mathbf{R} \mathbf{x}_k + \mathbf{t}$ (2.18), which can be estimated at the same time as the values of the control points. Alternatively, separate *detection* and *alignment* stages can be run to first localize and orient the objects of interest (Cootes, Cooper, Taylor *et al.* 1995).

In a B-snake, because the snake is controlled by fewer degrees of freedom, there is less need for the internal smoothness forces used with the original snakes, although these can still be derived and implemented using finite element analysis, i.e., taking derivatives and integrals of the B-spline basis functions (Terzopoulos 1983; Bathe 2007).

In practice, it is more common to estimate a set of *shape priors* on the typical distribution of the control points $\{\mathbf{x}_k\}$ (Cootes, Cooper, Taylor *et al.* 1995). Consider the set of resistor

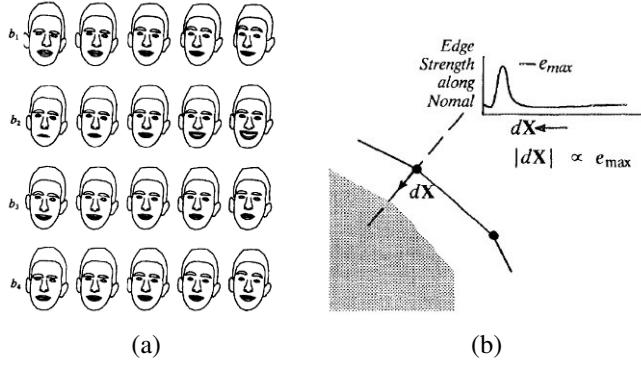


Figure 5.5 Active Shape Model (ASM): (a) the effect of varying the first four shape parameters for a set of faces (Cootes, Taylor, Lanitis *et al.* 1993) © 1993 IEEE; (b) searching for the strongest gradient along the normal to each control point (Cootes, Cooper, Taylor *et al.* 1995) © 1995 Elsevier.

shapes shown in Figure 5.4a. If we describe each contour with the set of control points shown in Figure 5.4b, we can plot the distribution of each point in a scatter plot, as shown in Figure 5.4c.

One potential way of describing this distribution would be by the location \bar{x}_k and 2D covariance C_k of each individual point x_k . These could then be turned into a quadratic penalty (prior energy) on the point location,

$$E_{\text{loc}}(\mathbf{x}_k) = \frac{1}{2}(\mathbf{x}_k - \bar{\mathbf{x}}_k)^T C_k^{-1}(\mathbf{x}_k - \bar{\mathbf{x}}_k). \quad (5.13)$$

In practice, however, the variation in point locations is usually highly correlated.

A preferable approach is to estimate the joint covariance of all the points simultaneously. First, concatenate all of the point locations $\{\mathbf{x}_k\}$ into a single vector \mathbf{x} , e.g., by interleaving the x and y locations of each point. The distribution of these vectors across all training examples (Figure 5.4a) can be described with a mean $\bar{\mathbf{x}}$ and a covariance

$$\mathbf{C} = \frac{1}{P} \sum_p (\mathbf{x}_p - \bar{\mathbf{x}})(\mathbf{x}_p - \bar{\mathbf{x}})^T, \quad (5.14)$$

where \mathbf{x}_p are the P training examples. Using *eigenvalue analysis* (Appendix A.1.2), which is also known as *Principal Component Analysis* (PCA) (Appendix B.1.1), the covariance matrix can be written as,

$$\mathbf{C} = \Phi \text{ diag}(\lambda_0 \dots \lambda_{K-1}) \Phi^T. \quad (5.15)$$

In most cases, the likely appearance of the points can be modeled using only a few eigenvectors with the largest eigenvalues. The resulting *point distribution model* (Cootes, Taylor, Lanitis *et al.* 1993; Cootes, Cooper, Taylor *et al.* 1995) can be written as

$$\mathbf{x} = \bar{\mathbf{x}} + \hat{\Phi} \mathbf{b}, \quad (5.16)$$

where \mathbf{b} is an $M \ll K$ element *shape parameter* vector and $\hat{\Phi}$ are the first m columns of Φ . To constrain the shape parameters to reasonable values, we can use a quadratic penalty of the

form

$$E_{\text{shape}} = \frac{1}{2} \mathbf{b}^T \text{diag}(\lambda_0 \dots \lambda_{M-1}) \mathbf{b} = \sum_m b_m^2 / 2\lambda_m. \quad (5.17)$$

Alternatively, the range of allowable b_m values can be limited to some range, e.g., $|b_m| \leq 3\sqrt{\lambda_m}$ (Cootes, Cooper, Taylor *et al.* 1995). Alternative approaches for deriving a set of shape vectors are reviewed by Isard and Blake (1998).

Varying the individual shape parameters b_m over the range $-2\sqrt{\lambda_m} \leq 2\sqrt{\lambda_m}$ can give a good indication of the expected variation in appearance, as shown in Figure 5.4d. Another example, this time related to face contours, is shown in Figure 5.5a.

In order to align a point distribution model with an image, each control point searches in a direction normal to the contour to find the most likely corresponding image edge point (Figure 5.5b). These individual measurements can be combined with priors on the shape parameters (and, if desired, position, scale, and orientation parameters) to estimate a new set of parameters. The resulting *Active Shape Model* (ASM) can be iteratively minimized to fit images to non-rigidly deforming objects such as medical images or body parts such as hands (Cootes, Cooper, Taylor *et al.* 1995). The ASM can also be combined with a PCA analysis of the underlying gray-level distribution to create an *Active Appearance Model* (AAM) (Cootes, Edwards, and Taylor 2001), which we discuss in more detail in Section 14.2.2.

5.1.2 Dynamic snakes and CONDENSATION

In many applications of active contours, the object of interest is being tracked from frame to frame as it deforms and evolves. In this case, it makes sense to use estimates from the previous frame to predict and constrain the new estimates.

One way to do this is to use Kalman filtering, which results in a formulation called *Kalman snakes* (Terzopoulos and Szeliski 1992; Blake, Curwen, and Zisserman 1993). The Kalman filter is based on a linear dynamic model of shape parameter evolution,

$$\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{w}_t, \quad (5.18)$$

where \mathbf{x}_t and \mathbf{x}_{t-1} are the current and previous state variables, \mathbf{A} is the linear *transition matrix*, and \mathbf{w} is a noise (perturbation) vector, which is often modeled as a Gaussian (Gelb 1974). The matrices \mathbf{A} and the noise covariance can be learned ahead of time by observing typical sequences of the object being tracked (Blake and Isard 1998).

The qualitative behavior of the Kalman filter can be seen in Figure 5.6a. The linear dynamic model causes a deterministic change (drift) in the previous estimate, while the process noise (perturbation) causes a stochastic diffusion that increases the system entropy (lack of certainty). New measurements from the current frame restore some of the certainty (peakedness) in the updated estimate.

In many situations, however, such as when tracking in clutter, a better estimate for the contour can be obtained if we remove the assumptions that the distribution are Gaussian, which is what the Kalman filter requires. In this case, a general multi-modal distribution is propagated, as shown in Figure 5.6b. In order to model such multi-modal distributions, Isard and Blake (1998) introduced the use of *particle filtering* to the computer vision community.⁵

⁵ Alternatives to modeling multi-modal distributions include *mixtures of Gaussians* (Bishop 2006) and *multiple hypothesis tracking* (Bar-Shalom and Fortmann 1988; Cham and Rehg 1999).

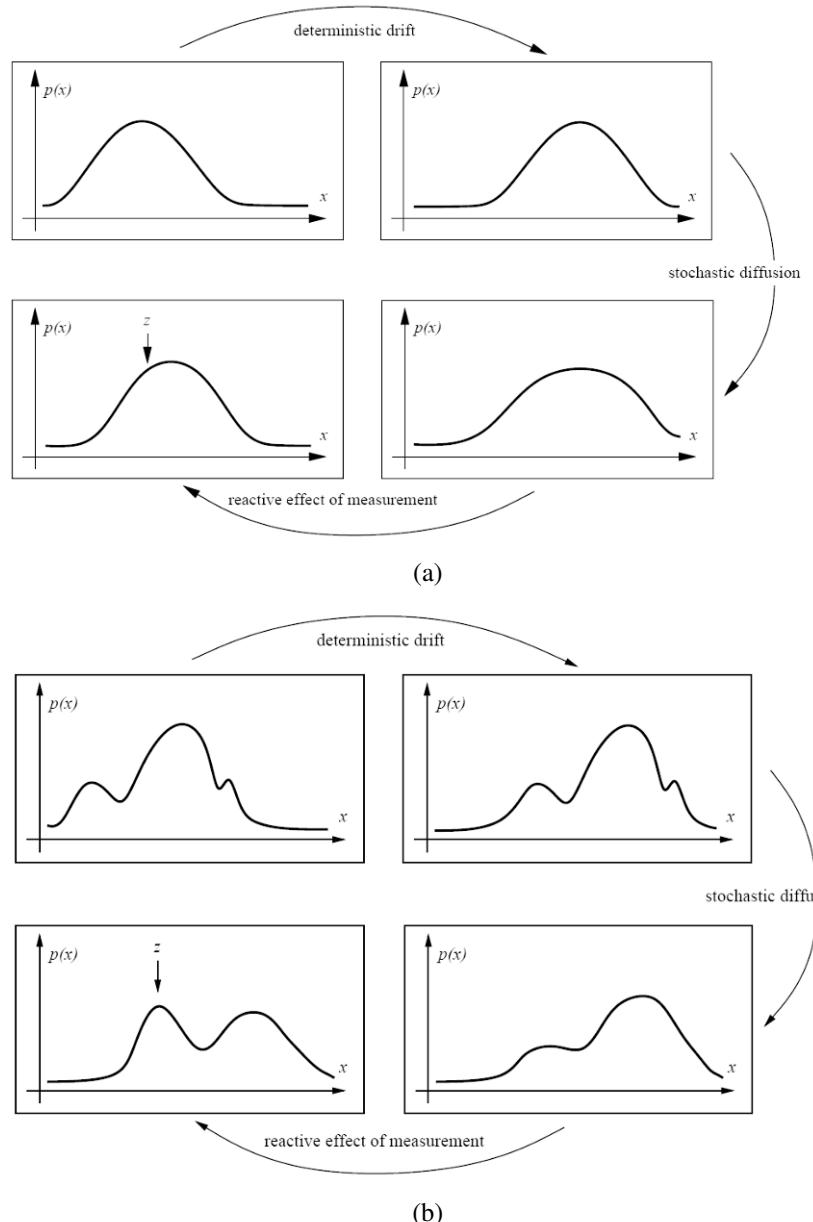


Figure 5.6 Probability density propagation (Isard and Blake 1998) © 1998 Springer. At the beginning of each estimation step, the probability density is updated according to the linear dynamic model (deterministic drift) and its certainty is reduced due to process noise (stochastic diffusion). New measurements introduce additional information that helps refine the current estimate. (a) The Kalman filter models the distributions as uni-modal, i.e., using a mean and covariance. (b) Some applications require more general multi-modal distributions.

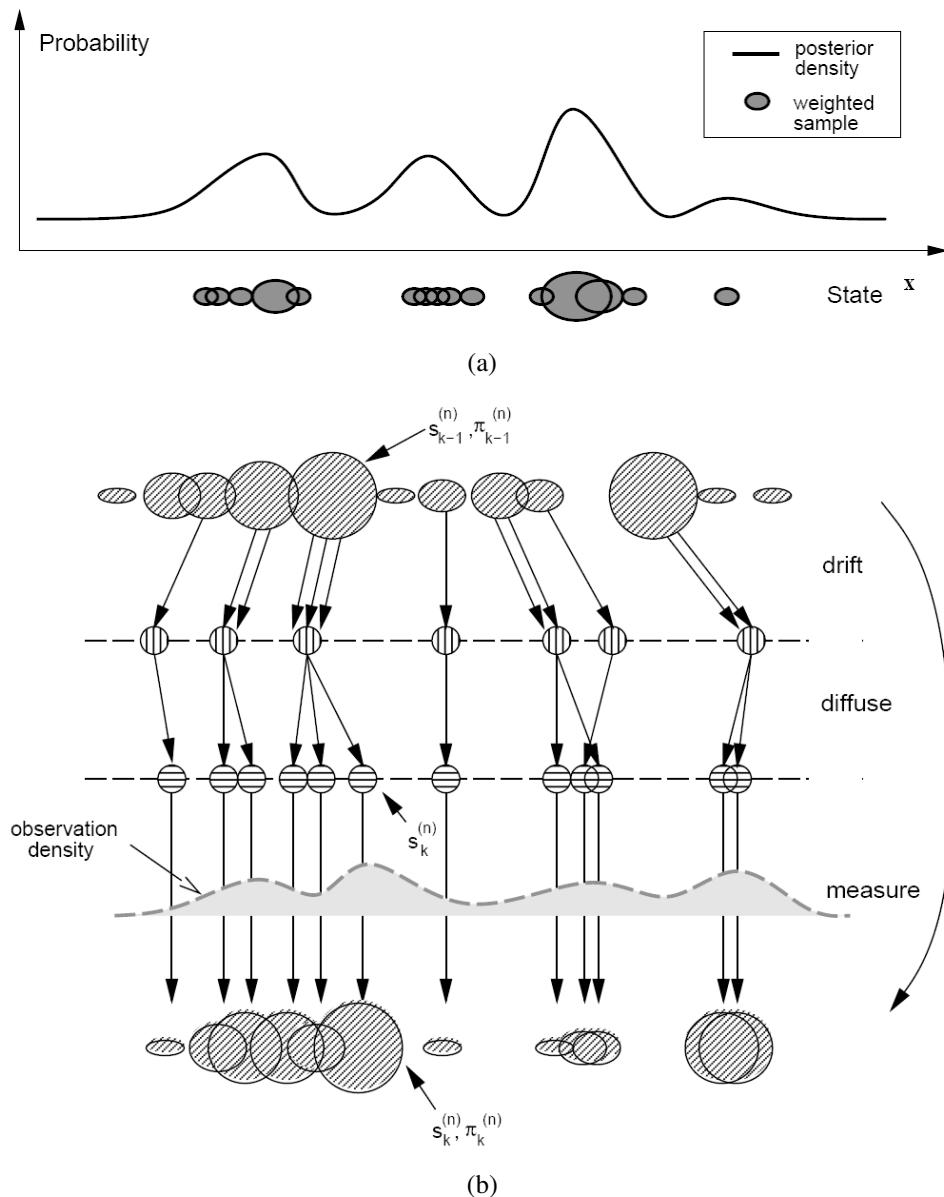


Figure 5.7 Factored sampling using particle filter in the CONDENSATION algorithm (Isard and Blake 1998) © 1998 Springer: (a) each density distribution is represented using a superposition of weighted *particles*; (b) the drift-diffusion-measurement cycle implemented using random sampling, perturbation, and re-weighting stages.

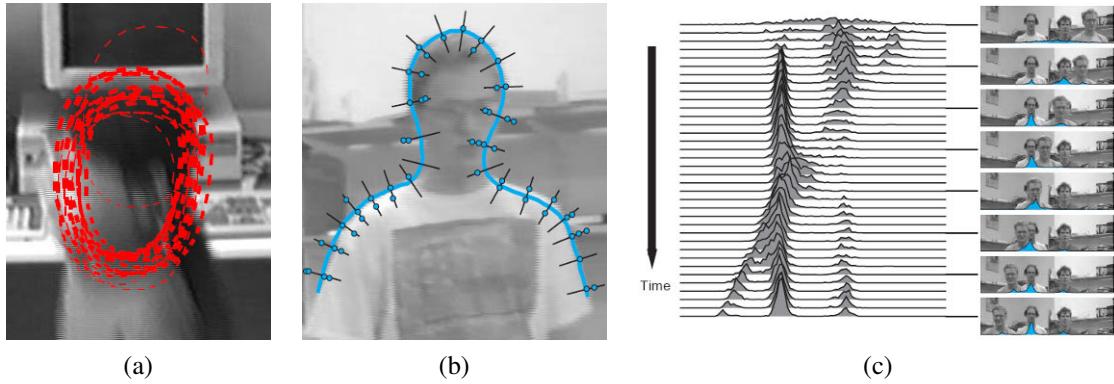


Figure 5.8 Head tracking using CONDENSATION (Isard and Blake 1998) © 1998 Springer: (a) sample set representation of head estimate distribution; (b) multiple measurements at each control vertex location; (c) multi-hypothesis tracking over time.

Particle filtering techniques represent a probability distribution using a collection of weighted point samples (Figure 5.7a) (Andrieu, de Freitas, Doucet *et al.* 2003; Bishop 2006; Koller and Friedman 2009). To update the locations of the samples according to the linear dynamics (deterministic drift), the centers of the samples are updated according to (5.18) and multiple samples are generated for each point (Figure 5.7b). These are then perturbed to account for the stochastic diffusion, i.e., their locations are moved by random vectors taken from the distribution of w .⁶ Finally, the weights of these samples are multiplied by the measurement probability density, i.e., we take each sample and measure its likelihood given the current (new) measurements. Because the point samples represent and propagate conditional estimates of the multi-modal density, Isard and Blake (1998) dubbed their algorithm CONditional DENsity propagATION or CONDENSATION.

Figure 5.8a shows what a factored sample of a head tracker might look like, drawing a red B-spline contour for each of (a subset of) the particles being tracked. Figure 5.8b shows why the measurement density itself is often multi-modal: the locations of the edges perpendicular to the spline curve can have multiple local maxima due to background clutter. Finally, Figure 5.8c shows the temporal evolution of the conditional density (x coordinate of the head and shoulder tracker centroid) as it tracks several people over time.

5.1.3 Scissors

Active contours allow a user to roughly specify a boundary of interest and have the system evolve the contour towards a more accurate location as well as track it over time. The results of this curve evolution, however, may be unpredictable and may require additional user-based hints to achieve the desired result.

An alternative approach is to have the system optimize the contour in real time as the user is drawing (Mortensen 1999). The *intelligent scissors* system developed by Mortensen and Barrett (1995) does just that. As the user draws a rough outline (the white curve in Figure 5.9a), the system computes and draws a better curve that clings to high-contrast edges

⁶ Note that because of the structure of these steps, non-linear dynamics and non-Gaussian noise can be used.

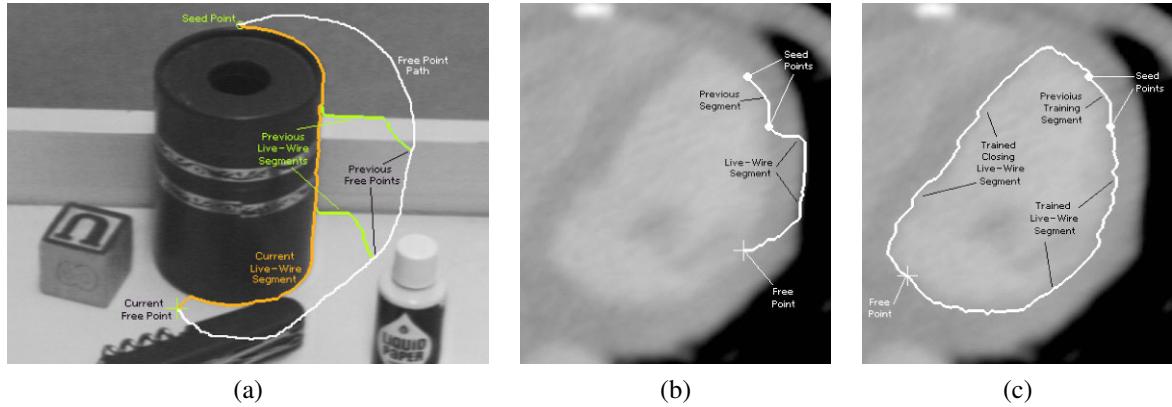


Figure 5.9 Intelligent scissors: (a) as the mouse traces the white path, the scissors follow the orange path along the object boundary (the green curves show intermediate positions) (Mortensen and Barrett 1995) © 1995 ACM; (b) regular scissors can sometimes jump to a stronger (incorrect) boundary; (c) after training to the previous segment, similar edge profiles are preferred (Mortensen and Barrett 1998) © 1995 Elsevier.

(the orange curve).

To compute the optimal curve path (*live-wire*), the image is first pre-processed to associate low costs with edges (links between neighboring horizontal, vertical, and diagonal, i.e., \mathcal{N}_8 neighbors) that are likely to be boundary elements. Their system uses a combination of zero-crossing, gradient magnitudes, and gradient orientations to compute these costs.

Next, as the user traces a rough curve, the system continuously recomputes the lowest-cost path between the starting *seed point* and the current mouse location using Dijkstra's algorithm, a breadth-first dynamic programming algorithm that terminates at the current target location.

In order to keep the system from jumping around unpredictably, the system will “freeze” the curve to date (reset the seed point) after a period of inactivity. To prevent the live wire from jumping onto adjacent higher-contrast contours, the system also “learns” the intensity profile under the current optimized curve, and uses this to preferentially keep the wire moving along the same (or a similar looking) boundary (Figure 5.9b–c).

Several extensions have been proposed to the basic algorithm, which works remarkably well even in its original form. Mortensen and Barrett (1999) use *tobogganing*, which is a simple form of watershed region segmentation, to pre-segment the image into regions whose boundaries become candidates for optimized curve paths. The resulting region boundaries are turned into a much smaller graph, where nodes are located wherever three or four regions meet. The Dijkstra algorithm is then run on this reduced graph, resulting in much faster (and often more stable) performance. Another extension to intelligent scissors is to use a probabilistic framework that takes into account the current trajectory of the boundary, resulting in a system called JetStream (Pérez, Blake, and Gangnet 2001).

Instead of re-computing an optimal curve at each time instant, a simpler system can be developed by simply “snapping” the current mouse position to the nearest likely boundary point (Gleicher 1995). Applications of these boundary extraction techniques to image cutting and pasting are presented in Section 10.4.

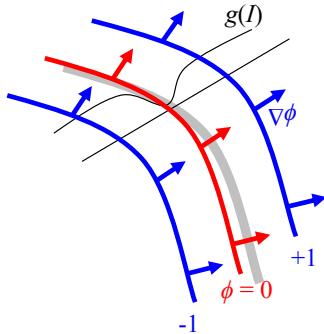


Figure 5.10 Level set evolution for a geodesic active contour. The embedding function ϕ is updated based on the curvature of the underlying surface modulated by the edge/speed function $g(I)$, as well as the gradient of $g(I)$, thereby attracting it to strong edges.

5.1.4 Level Sets

A limitation of active contours based on parametric curves of the form $f(s)$, e.g., snakes, B-snakes, and CONDENSATION, is that it is challenging to change the topology of the curve as it evolves. (McInerney and Terzopoulos (1999, 2000) describe one approach to doing this.) Furthermore, if the shape changes dramatically, curve reparameterization may also be required.

An alternative representation for such closed contours is to use a *level set*, where the *zero-crossing(s)* of a *characteristic* (or signed distance (Section 3.3.3)) function define the curve. Level sets evolve to fit and track objects of interest by modifying the underlying *embedding function* (another name for this 2D function) $\phi(x, y)$ instead of the curve $f(s)$ (Malladi, Sethian, and Vemuri 1995; Sethian 1999; Sapiro 2001; Osher and Paragios 2003). To reduce the amount of computation required, only a small strip (frontier) around the locations of the current zero-crossing needs to be updated at each step, which results in what are called *fast marching methods* (Sethian 1999).

An example of an evolution equation is the *geodesic active contour* proposed by Caselles, Kimmel, and Sapiro (1997) and Yezzi, Kichenassamy, Kumar *et al.* (1997),

$$\begin{aligned} \frac{d\phi}{dt} &= |\nabla\phi| \operatorname{div} \left(g(I) \frac{\nabla\phi}{|\nabla\phi|} \right) \\ &= g(I) |\nabla\phi| \operatorname{div} \left(\frac{\nabla\phi}{|\nabla\phi|} \right) + \nabla g(I) \cdot \nabla\phi, \end{aligned} \quad (5.19)$$

where $g(I)$ is a generalized version of the snake edge potential (5.5). To get an intuitive sense of the curve's behavior, assume that the embedding function ϕ is a signed distance function away from the curve (Figure 5.10), in which case $|\phi| = 1$. The first term in Equation (5.19) moves the curve in the direction of its curvature, i.e., it acts to straighten the curve, under the influence of the modulation function $g(I)$. The second term moves the curve down the gradient of $g(I)$, encouraging the curve to migrate towards minima of $g(I)$.

While this level-set formulation can readily change topology, it is still susceptible to local minima, since it is based on local measurements such as image gradients. An alternative

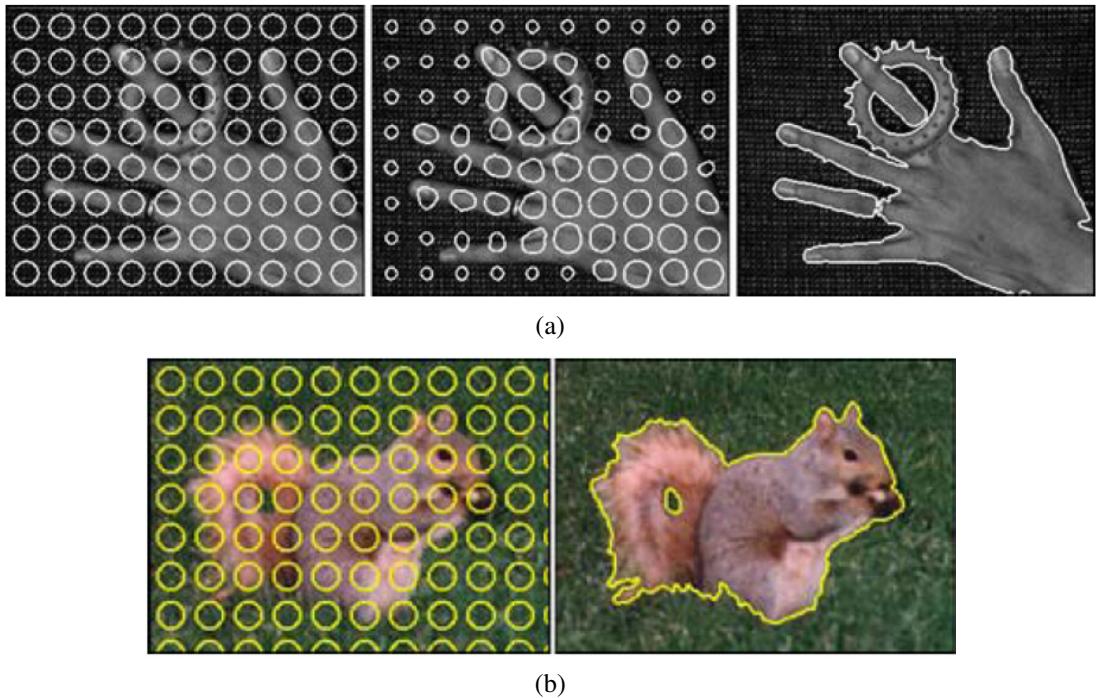


Figure 5.11 Level set segmentation (Cremers, Rousson, and Deriche 2007) © 2007 Springer: (a) grayscale image segmentation and (b) color image segmentation. Uni-variate and multi-variate Gaussians are used to model the foreground and background pixel distributions. The initial circles evolve towards an accurate segmentation of foreground and background, adapting their topology as they evolve.

approach is to re-cast the problem in a segmentation framework, where the energy measures the consistency of the image statistics (e.g., color, texture, motion) inside and outside the segmented regions (Cremers, Rousson, and Deriche 2007; Rousson and Paragios 2008; Houhou, Thiran, and Bresson 2008). These approaches build on earlier energy-based segmentation frameworks introduced by Leclerc (1989), Mumford and Shah (1989), and Chan and Vese (1992), which are discussed in more detail in Section 5.5. Examples of such level-set segmentations are shown in Figure 5.11, which shows the evolution of the level sets from a series of distributed circles towards the final binary segmentation.

For more information on level sets and their applications, please see the collection of papers edited by Osher and Paragios (2003) as well as the series of Workshops on Variational and Level Set Methods in Computer Vision (Paragios, Faugeras, Chan *et al.* 2005) and Special Issues on Scale Space and Variational Methods in Computer Vision (Paragios and Sgallari 2009).

5.1.5 Application: Contour tracking and rotoscoping

Active contours can be used in a wide variety of object-tracking applications (Blake and Isard 1998; Yilmaz, Javed, and Shah 2006). For example, they can be used to track facial features for performance-driven animation (Terzopoulos and Waters 1990; Lee, Terzopoulos, and Wa-

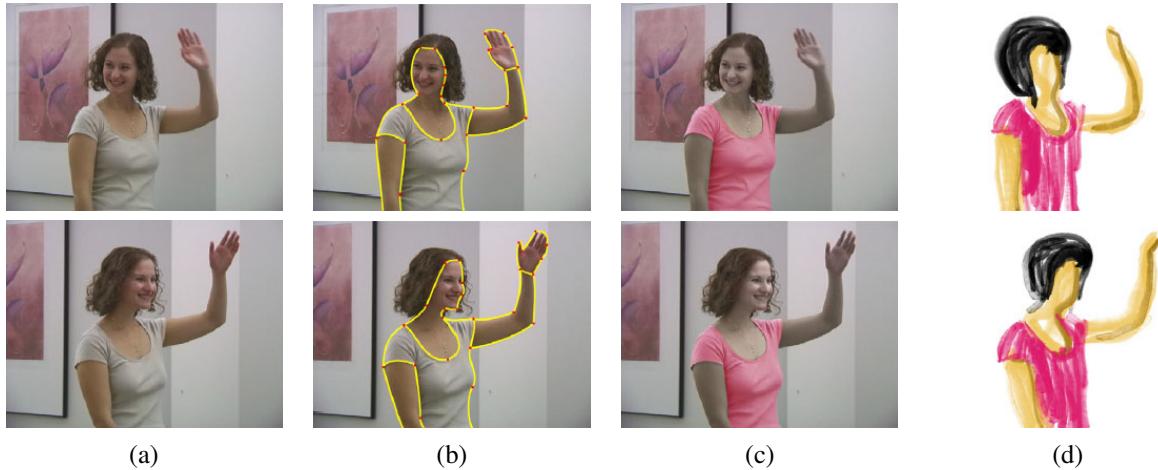


Figure 5.12 Keyframe-based rotoscoping (Agarwala, Hertzmann, Seitz *et al.* 2004) © 2004 ACM: (a) original frames; (b) rotoscoped contours; (c) re-colored blouse; (d) rotoscoped hand-drawn animation.

ters 1995; Parke and Waters 1996; Bregler, Covell, and Slaney 1997) (Figure 5.2b). They can also be used to track heads and people, as shown in Figure 5.8, as well as moving vehicles (Paragios and Deriche 2000). Additional applications include medical image segmentation, where contours can be tracked from slice to slice in computerized tomography (3D medical imagery) (Cootes and Taylor 2001) or over time, as in ultrasound scans.

An interesting application that is closer to computer animation and visual effects is *rotoscoping*, which uses the tracked contours to deform a set of hand-drawn animations (or to modify or replace the original video frames).⁷ Agarwala, Hertzmann, Seitz *et al.* (2004) present a system based on tracking hand-drawn B-spline contours drawn at selected keyframes, using a combination of geometric and appearance-based criteria (Figure 5.12). They also provide an excellent review of previous rotoscoping and image-based, contour-tracking systems.

Additional applications of rotoscoping (object contour detection and segmentation), such as cutting and pasting objects from one photograph into another, are presented in Section 10.4.

5.2 Split and merge

As mentioned in the introduction to this chapter, the simplest possible technique for segmenting a grayscale image is to select a threshold and then compute connected components (Section 3.3.2). Unfortunately, a single threshold is rarely sufficient for the whole image because of lighting and intra-object statistical variations.

In this section, we describe a number of algorithms that proceed either by recursively splitting the whole image into pieces based on region statistics or, conversely, merging pixels and regions together in a hierarchical fashion. It is also possible to combine both splitting and merging by starting with a medium-grain segmentation (in a quadtree representation) and

⁷ The term comes from a device (a rotoscope) that projected frames of a live-action film underneath an acetate so that artists could draw animations directly over the actors' shapes.

then allowing both merging and splitting operations (Horowitz and Pavlidis 1976; Pavlidis and Liow 1990).

5.2.1 Watershed

A technique related to thresholding, since it operates on a grayscale image, is *watershed* computation (Vincent and Soille 1991). This technique segments an image into several *catchment basins*, which are the regions of an image (interpreted as a height field or landscape) where rain would flow into the same lake. An efficient way to compute such regions is to start flooding the landscape at all of the local minima and to label ridges wherever differently evolving components meet. The whole algorithm can be implemented using a priority queue of pixels and breadth-first search (Vincent and Soille 1991).⁸

Since images rarely have dark regions separated by lighter ridges, watershed segmentation is usually applied to a smoothed version of the gradient magnitude image, which also makes it usable with color images. As an alternative, the maximum oriented energy in a steerable filter (3.28–3.29) (Freeman and Adelson 1991) can be used as the basis of the *oriented watershed transform* developed by Arbeláez, Maire, Fowlkes *et al.* (2010). Such techniques end up finding smooth regions separated by visible (higher gradient) boundaries. Since such boundaries are what active contours usually follow, active contour algorithms (Mortensen and Barrett 1999; Li, Sun, Tang *et al.* 2004) often precompute such a segmentation using either the watershed or the related *tobogganing* technique (Section 5.1.3).

Unfortunately, watershed segmentation associates a unique region with each local minimum, which can lead to over-segmentation. Watershed segmentation is therefore often used as part of an interactive system, where the user first marks seed locations (with a click or a short stroke) that correspond to the centers of different desired components. Figure 5.13 shows the results of running the watershed algorithm with some manually placed markers on a confocal microscopy image. It also shows the result for an improved version of watershed that uses local morphology to smooth out and optimize the boundaries separating the regions (Beare 2006).

5.2.2 Region splitting (divisive clustering)

Splitting the image into successively finer regions is one of the oldest techniques in computer vision. Ohlander, Price, and Reddy (1978) present such a technique, which first computes a histogram for the whole image and then finds a threshold that best separates the large peaks in the histogram. This process is repeated until regions are either fairly uniform or below a certain size.

More recent splitting algorithms often optimize some metric of intra-region similarity and inter-region dissimilarity. These are covered in Sections 5.4 and 5.5.

5.2.3 Region merging (agglomerative clustering)

Region merging techniques also date back to the beginnings of computer vision. Brice and Fennema (1970) use a dual grid for representing boundaries between pixels and merge re-

⁸ A related algorithm can be used to compute maximally stable extremal regions (MSERs) efficiently (Section 4.1.1) (Nistér and Stewénius 2008).

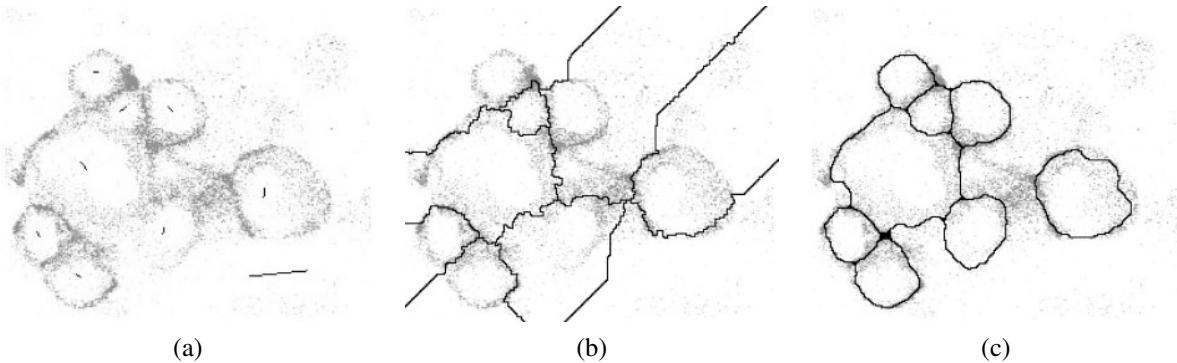


Figure 5.13 Locally constrained watershed segmentation (Beare 2006) © 2006 IEEE: (a) original confocal microscopy image with marked seeds (line segments); (b) standard watershed segmentation; (c) locally constrained watershed segmentation.

gions based on their relative boundary lengths and the strength of the visible edges at these boundaries.

In data clustering, algorithms can link clusters together based on the distance between their closest points (single-link clustering), their farthest points (complete-link clustering), or something in between (Jain, Topchy, Law *et al.* 2004). Kamvar, Klein, and Manning (2002) provide a probabilistic interpretation of these algorithms and show how additional models can be incorporated within this framework.

A very simple version of pixel-based merging combines adjacent regions whose average color difference is below a threshold or whose regions are too small. Segmenting the image into such *superpixels* (Mori, Ren, Efros *et al.* 2004), which are not semantically meaningful, can be a useful pre-processing stage to make higher-level algorithms such as stereo matching (Zitnick, Kang, Uyttendaele *et al.* 2004; Taguchi, Wilburn, and Zitnick 2008), optic flow (Zitnick, Jovic, and Kang 2005; Brox, Bregler, and Malik 2009), and recognition (Mori, Ren, Efros *et al.* 2004; Mori 2005; Gu, Lim, Arbelaez *et al.* 2009; Lim, Arbelaez, Gu *et al.* 2009) both faster and more robust.

5.2.4 Graph-based segmentation

While many merging algorithms simply apply a fixed rule that groups pixels and regions together, Felzenszwalb and Huttenlocher (2004b) present a merging algorithm that uses *relative dissimilarities* between regions to determine which ones should be merged; it produces an algorithm that provably optimizes a global grouping metric. They start with a pixel-to-pixel dissimilarity measure $w(e)$ that measures, for example, intensity differences between \mathcal{N}_8 neighbors. (Alternatively, they can use the *joint feature space* distances (5.42) introduced by Comaniciu and Meer (2002), which we discuss in Section 5.3.2.)

For any region R , its *internal difference* is defined as the largest edge weight in the region's minimum spanning tree,

$$Int(R) = \min_{e \in MST(R)} w(e). \quad (5.20)$$

For any two adjacent regions with at least one edge connecting their vertices, the difference

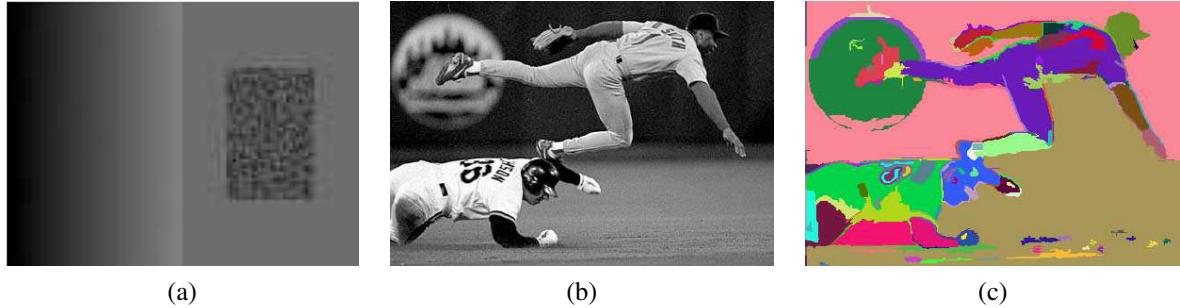


Figure 5.14 Graph-based merging segmentation (Felzenszwalb and Huttenlocher 2004b) © 2004 Springer: (a) input grayscale image that is successfully segmented into three regions even though the variation inside the smaller rectangle is larger than the variation across the middle edge; (b) input grayscale image; (c) resulting segmentation using an \mathcal{N}_8 pixel neighborhood.

between these regions is defined as the minimum weight edge connecting the two regions,

$$Dif(R_1, R_2) = \min_{e=(v_1, v_2) | v_1 \in R_1, v_2 \in R_2} w(e). \quad (5.21)$$

Their algorithm merges any two adjacent regions whose difference is smaller than the minimum internal difference of these two regions,

$$MInt(R_1, R_2) = \min(Int(R_1) + \tau(R_1), Int(R_2) + \tau(R_2)), \quad (5.22)$$

where $\tau(R)$ is a heuristic region penalty that Felzenszwalb and Huttenlocher (2004b) set to $k/|R|$, but which can be set to any application-specific measure of region goodness.

By merging regions in decreasing order of the edges separating them (which can be efficiently evaluated using a variant of Kruskal's minimum spanning tree algorithm), they provably produce segmentations that are neither too fine (there exist regions that could have been merged) nor too coarse (there are regions that could be split without being mergeable). For fixed-size pixel neighborhoods, the running time for this algorithm is $O(N \log N)$, where N is the number of image pixels, which makes it one of the fastest segmentation algorithms (Paris and Durand 2007). Figure 5.14 shows two examples of images segmented using their technique.

5.2.5 Probabilistic aggregation

Alpert, Galun, Basri *et al.* (2007) develop a probabilistic merging algorithm based on two cues, namely gray-level similarity and texture similarity. The gray-level similarity between regions R_i and R_j is based on the *minimal external difference* from other neighboring regions,

$$\sigma_{local}^+ = \min(\Delta_i^+, \Delta_j^+), \quad (5.23)$$

where $\Delta_i^+ = \min_k |\Delta_{ik}|$ and Δ_{ik} is the difference in average intensities between regions R_i and R_k . This is compared to the *average intensity difference*,

$$\sigma_{local}^- = \frac{\Delta_i^- + \Delta_j^-}{2}, \quad (5.24)$$

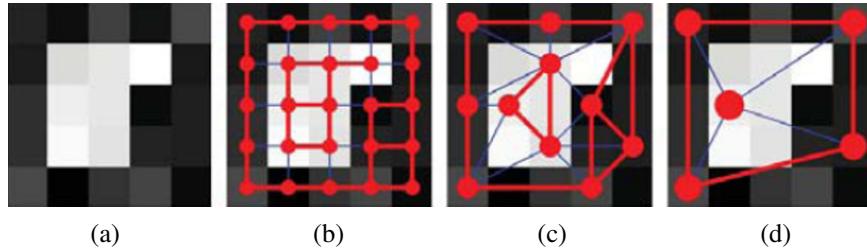


Figure 5.15 Coarse to fine node aggregation in segmentation by weighted aggregation (SWA) (Sharon, Galun, Sharon *et al.* 2006) © 2006 Macmillan Publishers Ltd [Nature]: (a) original gray-level pixel grid; (b) inter-pixel couplings, where thicker lines indicate stronger couplings; (c) after one level of coarsening, where each original pixel is strongly coupled to one of the coarse-level nodes; (d) after two levels of coarsening.

where $\Delta_i^- = \sum_k (\tau_{ik} \Delta_{ik}) / \sum_k (\tau_{ik})$ and τ_{ik} is the boundary length between regions R_i and R_k . The texture similarity is defined using relative differences between histogram bins of simple oriented Sobel filter responses. The pairwise statistics σ_{local}^+ and σ_{local}^- are used to compute the likelihoods p_{ij} that two regions should be merged. (See the paper by Alpert, Galun, Basri *et al.* (2007) for more details.)

Merging proceeds in a hierarchical fashion inspired by algebraic multigrid techniques (Brandt 1986; Briggs, Henson, and McCormick 2000) and previously used by Alpert, Galun, Basri *et al.* (2007) in their segmentation by weighted aggregation (SWA) algorithm (Sharon, Galun, Sharon *et al.* 2006), which we discuss in Section 5.4. A subset of the nodes $C \subset V$ that are (collectively) *strongly coupled* to all of the original nodes (regions) are used to define the problem at a coarser scale (Figure 5.15), where strong coupling is defined as

$$\frac{\sum_{j \in C} p_{ij}}{\sum_{j \in V} p_{ij}} > \phi, \quad (5.25)$$

with ϕ usually set to 0.2. The intensity and texture similarity statistics for the coarser nodes are recursively computed using weighted averaging, where the relative strengths (couplings) between coarse- and fine-level nodes are based on their merge probabilities p_{ij} . This allows the algorithm to run in essentially $O(N)$ time, using the same kind of hierarchical aggregation operations that are used in pyramid-based filtering or preconditioning algorithms. After a segmentation has been identified at a coarser level, the exact memberships of each pixel are computed by propagating coarse-level assignments to their finer-level “children” (Sharon, Galun, Sharon *et al.* 2006; Alpert, Galun, Basri *et al.* 2007). Figure 5.22 shows the segmentations produced by this algorithm compared to other popular segmentation algorithms.

5.3 Mean shift and mode finding

Mean-shift and mode finding techniques, such as k-means and mixtures of Gaussians, model the feature vectors associated with each pixel (e.g., color and position) as samples from an unknown probability density function and then try to find clusters (modes) in this distribution.

Consider the color image shown in Figure 5.16a. How would you segment this image based on color alone? Figure 5.16b shows the distribution of pixels in $L^*u^*v^*$ space, which is equivalent to what a vision algorithm that ignores spatial location would see. To make the

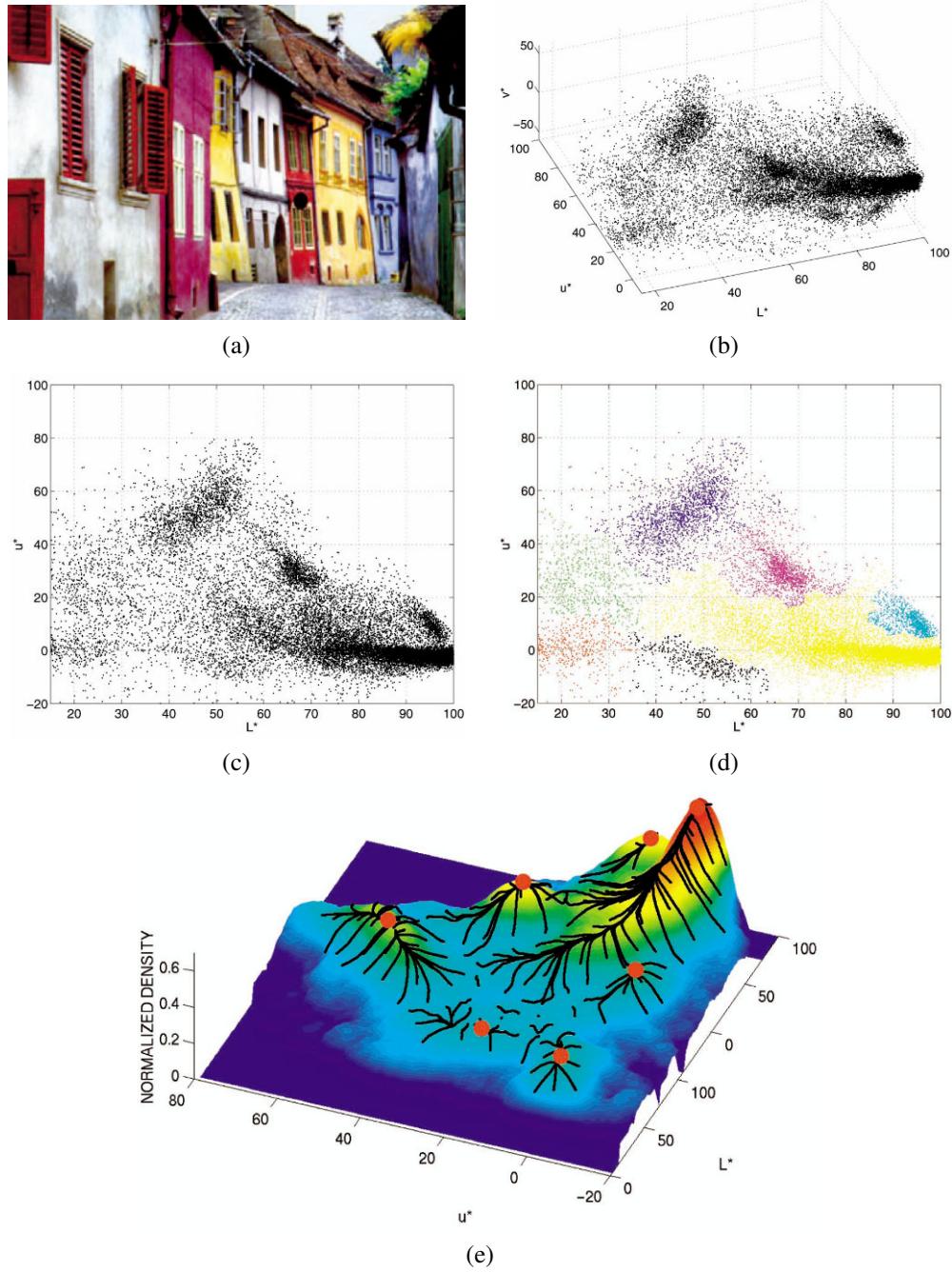


Figure 5.16 Mean-shift image segmentation (Comaniciu and Meer 2002) © 2002 IEEE: (a) input color image; (b) pixels plotted in $L^*u^*v^*$ space; (c) L^*u^* space distribution; (d) clustered results after 159 mean-shift procedures; (e) corresponding trajectories with peaks marked as red dots.

visualization simpler, let us only consider the L^*u^* coordinates, as shown in Figure 5.16c. How many obvious (elongated) clusters do you see? How would you go about finding these clusters?

The k-means and mixtures of Gaussians techniques use a *parametric* model of the density function to answer this question, i.e., they assume the density is the superposition of a small number of simpler distributions (e.g., Gaussians) whose locations (centers) and shape (covariance) can be estimated. Mean shift, on the other hand, smoothes the distribution and finds its peaks as well as the regions of feature space that correspond to each peak. Since a complete density is being modeled, this approach is called *non-parametric* (Bishop 2006). Let us look at these techniques in more detail.

5.3.1 K-means and mixtures of Gaussians

While k-means implicitly models the probability density as a superposition of spherically symmetric distributions, it does not require any probabilistic reasoning or modeling (Bishop 2006). Instead, the algorithm is given the number of clusters k it is supposed to find; it then iteratively updates the cluster center location based on the samples that are closest to each center. The algorithm can be initialized by randomly sampling k centers from the input feature vectors. Techniques have also been developed for splitting or merging cluster centers based on their statistics, and for accelerating the process of finding the nearest mean center (Bishop 2006).

In mixtures of Gaussians, each cluster center is augmented by a covariance matrix whose values are re-estimated from the corresponding samples. Instead of using nearest neighbors to associate input samples with cluster centers, a *Mahalanobis distance* (Appendix B.1.1) is used:

$$d(\mathbf{x}_i, \boldsymbol{\mu}_k; \boldsymbol{\Sigma}_k) = \|\mathbf{x}_i - \boldsymbol{\mu}_k\|_{\boldsymbol{\Sigma}_k^{-1}} = (\mathbf{x}_i - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_k) \quad (5.26)$$

where \mathbf{x}_i are the input samples, $\boldsymbol{\mu}_k$ are the cluster centers, and $\boldsymbol{\Sigma}_k$ are their covariance estimates. Samples can be associated with the nearest cluster center (a *hard assignment* of membership) or can be *softly assigned* to several nearby clusters.

This latter, more commonly used, approach corresponds to iteratively re-estimating the parameters for a mixture of Gaussians density function,

$$p(\mathbf{x} | \{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}) = \sum_k \pi_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \quad (5.27)$$

where π_k are the *mixing coefficients*, $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ are the Gaussian means and covariances, and

$$\mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{|\boldsymbol{\Sigma}_k|} e^{-d(\mathbf{x}, \boldsymbol{\mu}_k; \boldsymbol{\Sigma}_k)} \quad (5.28)$$

is the *normal* (Gaussian) distribution (Bishop 2006).

To iteratively compute (a local) maximum likely estimate for the unknown mixture parameters $\{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}$, the *expectation maximization* (EM) algorithm (Dempster, Laird, and Rubin 1977) proceeds in two alternating stages:

1. The *expectation* stage (E step) estimates the *responsibilities*

$$z_{ik} = \frac{1}{Z_i} \pi_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \quad \text{with} \quad \sum_k z_{ik} = 1, \quad (5.29)$$

which are the estimates of how likely a sample \mathbf{x}_i was generated from the k th Gaussian cluster.

2. The *maximization* stage (M step) updates the parameter values

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_i z_{ik} \mathbf{x}_i, \quad (5.30)$$

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_i z_{ik} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T, \quad (5.31)$$

$$\pi_k = \frac{N_k}{N}, \quad (5.32)$$

where

$$N_k = \sum_i z_{ik}. \quad (5.33)$$

is an estimate of the number of sample points assigned to each cluster.

Bishop (2006) has a wonderful exposition of both mixture of Gaussians estimation and the more general topic of expectation maximization.

In the context of image segmentation, Ma, Derksen, Hong *et al.* (2007) present a nice review of segmentation using mixtures of Gaussians and develop their own extension based on Minimum Description Length (MDL) coding, which they show produces good results on the Berkeley segmentation database.

5.3.2 Mean shift

While k-means and mixtures of Gaussians use a parametric form to model the probability density function being segmented, mean shift implicitly models this distribution using a smooth continuous *non-parametric model*. The key to mean shift is a technique for efficiently finding peaks in this high-dimensional data distribution without ever computing the complete function explicitly (Fukunaga and Hostetler 1975; Cheng 1995; Comaniciu and Meer 2002).

Consider once again the data points shown in Figure 5.16c, which can be thought of as having been drawn from some probability density function. If we could compute this density function, as visualized in Figure 5.16e, we could find its major peaks (*modes*) and identify regions of the input space that climb to the same peak as being part of the same region. This is the inverse of the *watershed* algorithm described in Section 5.2.1, which climbs downhill to find *basins of attraction*.

The first question, then, is how to estimate the density function given a sparse set of samples. One of the simplest approaches is to just smooth the data, e.g., by convolving it with a fixed kernel of width h ,

$$f(\mathbf{x}) = \sum_i K(\mathbf{x} - \mathbf{x}_i) = \sum_i k \left(\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{h^2} \right), \quad (5.34)$$

where \mathbf{x}_i are the input samples and $k(r)$ is the kernel function (or *Parzen window*).⁹ This approach is known as *kernel density estimation* or the *Parzen window technique* (Duda, Hart,

⁹ In this simplified formula, a Euclidean metric is used. We discuss a little later (5.42) how to generalize this to non-uniform (scaled or oriented) metrics. Note also that this distribution may not be *proper*, i.e., integrate to 1. Since we are looking for maxima in the density, this does not matter.

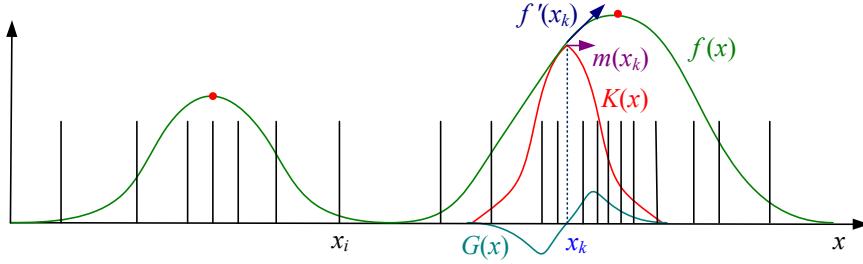


Figure 5.17 One-dimensional visualization of the kernel density estimate, its derivative, and a mean shift. The kernel density estimate $f(x)$ is obtained by convolving the sparse set of input samples x_i with the kernel function $K(x)$. The derivative of this function, $f'(x)$, can be obtained by convolving the inputs with the derivative kernel $G(x)$. Estimating the local displacement vectors around a current estimate x_k results in the mean-shift vector $m(x_k)$, which, in a multi-dimensional setting, point in the same direction as the function gradient $\nabla f(x_k)$. The red dots indicate local maxima in $f(x)$ to which the mean shifts converge.

and Stork 2001, Section 4.3; Bishop 2006, Section 2.5.1). Once we have computed $f(\mathbf{x})$, as shown in Figures 5.16e and 5.17, we can find its local maxima using gradient ascent or some other optimization technique.

The problem with this “brute force” approach is that, for higher dimensions, it becomes computationally prohibitive to evaluate $f(\mathbf{x})$ over the complete search space.¹⁰ Instead, mean shift uses a variant of what is known in the optimization literature as *multiple restart gradient descent*. Starting at some guess for a local maximum, \mathbf{y}_k , which can be a random input data point \mathbf{x}_i , mean shift computes the gradient of the density estimate $f(\mathbf{x})$ at \mathbf{y}_k and takes an uphill step in that direction (Figure 5.17). The gradient of $f(\mathbf{x})$ is given by

$$\nabla f(\mathbf{x}) = \sum_i (\mathbf{x}_i - \mathbf{x}) G(\mathbf{x} - \mathbf{x}_i) = \sum_i (\mathbf{x}_i - \mathbf{x}) g\left(\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{h^2}\right), \quad (5.35)$$

where

$$g(r) = -k'(r), \quad (5.36)$$

and $k'(r)$ is the first derivative of $k(r)$. We can re-write the gradient of the density function as

$$\nabla f(\mathbf{x}) = \left[\sum_i G(\mathbf{x} - \mathbf{x}_i) \right] \mathbf{m}(\mathbf{x}), \quad (5.37)$$

where the vector

$$\mathbf{m}(\mathbf{x}) = \frac{\sum_i \mathbf{x}_i G(\mathbf{x} - \mathbf{x}_i)}{\sum_i G(\mathbf{x} - \mathbf{x}_i)} - \mathbf{x} \quad (5.38)$$

is called the *mean shift*, since it is the difference between the weighted mean of the neighbors \mathbf{x}_i around \mathbf{x} and the current value of \mathbf{x} .

In the mean-shift procedure, the current estimate of the mode \mathbf{y}_k at iteration k is replaced by its locally weighted mean,

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \mathbf{m}(\mathbf{y}_k) = \frac{\sum_i \mathbf{x}_i G(\mathbf{y}_k - \mathbf{x}_i)}{\sum_i G(\mathbf{y}_k - \mathbf{x}_i)}. \quad (5.39)$$

¹⁰ Even for one dimension, if the space is extremely sparse, it may be inefficient.

Comaniciu and Meer (2002) prove that this algorithm converges to a local maximum of $f(\mathbf{x})$ under reasonably weak conditions on the kernel $k(r)$, i.e., that it is monotonically decreasing. This convergence is not guaranteed for regular gradient descent unless appropriate step size control is used.

The two kernels that Comaniciu and Meer (2002) studied are the Epanechnikov kernel,

$$k_E(r) = \max(0, 1 - r), \quad (5.40)$$

which is a radial generalization of a bilinear kernel, and the Gaussian (normal) kernel,

$$k_N(r) = \exp\left(-\frac{1}{2}r^2\right). \quad (5.41)$$

The corresponding derivative kernels $g(r)$ are a unit ball and another Gaussian, respectively. Using the Epanechnikov kernel converges in a finite number of steps, while the Gaussian kernel has a smoother trajectory (and produces better results), but converges very slowly near a mode (Exercise 5.5).

The simplest way to apply mean shift is to start a separate mean-shift mode estimate \mathbf{y} at every input point \mathbf{x}_i and to iterate for a fixed number of steps or until the mean-shift magnitude is below a threshold. A faster approach is to randomly subsample the input points \mathbf{x}_i and to keep track of each point's temporal evolution. The remaining points can then be classified based on the nearest evolution path (Comaniciu and Meer 2002). Paris and Durand (2007) review a number of other more efficient implementations of mean shift, including their own approach, which is based on using an efficient low-resolution estimate of the complete multi-dimensional space of $f(\mathbf{x})$ along with its stationary points.

The color-based segmentation shown in Figure 5.16 only looks at pixel colors when determining the best clustering. It may therefore cluster together small isolated pixels that happen to have the same color, which may not correspond to a semantically meaningful segmentation of the image.

Better results can usually be obtained by clustering in the *joint domain* of color and location. In this approach, the spatial coordinates of the image $\mathbf{x}_s = (x, y)$, which are called the *spatial domain*, are concatenated with the color values \mathbf{x}_r , which are known as the *range domain*, and mean-shift clustering is applied in this five-dimensional space \mathbf{x}_j . Since location and color may have different scales, the kernels are adjusted accordingly, i.e., we use a kernel of the form

$$K(\mathbf{x}_j) = k\left(\frac{\|\mathbf{x}_r\|^2}{h_r^2}\right) k\left(\frac{\|\mathbf{x}_s\|^2}{h_s^2}\right), \quad (5.42)$$

where separate parameters h_s and h_r are used to control the spatial and range bandwidths of the filter kernels. Figure 5.18 shows an example of mean-shift clustering in the joint domain, with parameters $(h_s, h_r, M) = (16, 19, 40)$, where spatial regions containing less than M pixels are eliminated.

The form of the joint domain filter kernel (5.42) is reminiscent of the bilateral filter kernel (3.34–3.37) discussed in Section 3.3.1. The difference between mean shift and bilateral filtering, however, is that in mean shift the spatial coordinates of each pixel are adjusted along with its color values, so that the pixel migrates more quickly towards other pixels with similar colors, and can therefore later be used for clustering and segmentation.

Determining the best bandwidth parameters h to use with mean shift remains something of an art, although a number of approaches have been explored. These include optimizing

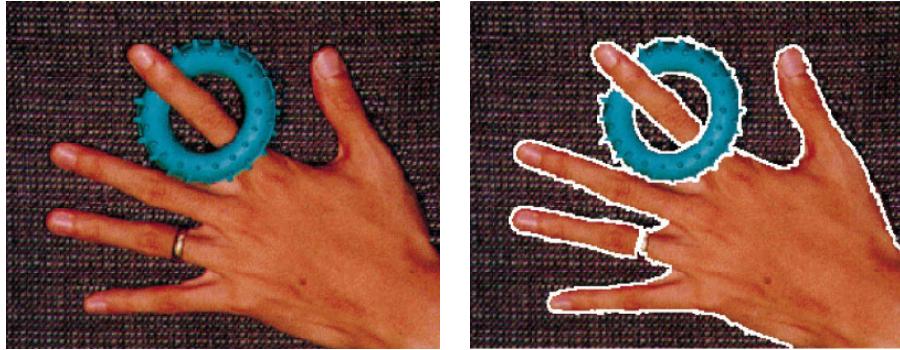


Figure 5.18 Mean-shift color image segmentation with parameters $(h_s, h_r, M) = (16, 19, 40)$ (Comaniciu and Meer 2002) © 2002 IEEE.

the bias–variance tradeoff, looking for parameter ranges where the number of clusters varies slowly, optimizing some external clustering criterion, or using top-down (application domain) knowledge (Comaniciu and Meer 2003). It is also possible to change the orientation of the kernel in joint parameter space for applications such as spatio-temporal (video) segmentations (Wang, Thiesson, Xu *et al.* 2004).

Mean shift has been applied to a number of different problems in computer vision, including face tracking, 2D shape extraction, and texture segmentation (Comaniciu and Meer 2002), and more recently in stereo matching (Chapter 11) (Wei and Quan 2004), non-photorealistic rendering (Section 10.5.2) (DeCarlo and Santella 2002), and video editing (Section 10.4.5) (Wang, Bhat, Colburn *et al.* 2005). Paris and Durand (2007) provide a nice review of such applications, as well as techniques for more efficiently solving the mean-shift equations and producing hierarchical segmentations.

5.4 Normalized cuts

While bottom-up merging techniques aggregate regions into coherent wholes and mean-shift techniques try to find clusters of similar pixels using mode finding, the normalized cuts technique introduced by Shi and Malik (2000) examines the *affinities* (similarities) between nearby pixels and tries to separate groups that are connected by weak affinities.

Consider the simple graph shown in Figure 5.19a. The pixels in group A are all strongly connected with high affinities, shown as thick red lines, as are the pixels in group B . The connections between these two groups, shown as thinner blue lines, are much weaker. A *normalized cut* between the two groups, shown as a dashed line, separates them into two clusters.

The cut between two groups A and B is defined as the sum of all the weights being cut,

$$\text{cut}(A, B) = \sum_{i \in A, j \in B} w_{ij}, \quad (5.43)$$

where the weights between two pixels (or regions) i and j measure their similarity. Using a minimum cut as a segmentation criterion, however, does not result in reasonable clusters, since the smallest cuts usually involve isolating a single pixel.

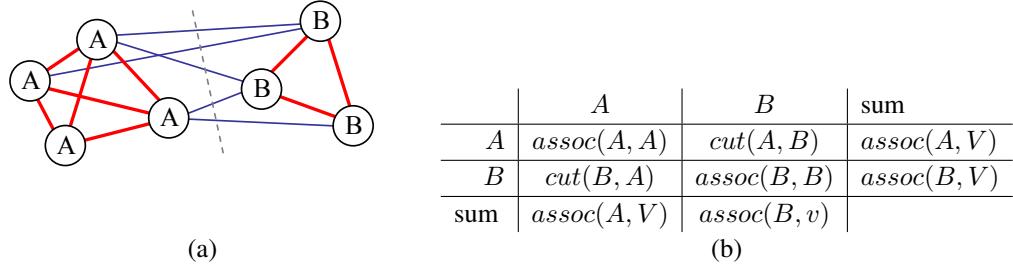


Figure 5.19 Sample weighted graph and its normalized cut: (a) a small sample graph and its smallest normalized cut; (b) tabular form of the associations and cuts for this graph. The *assoc* and *cut* entries are computed as area sums of the associated weight matrix \mathbf{W} (Figure 5.20). Normalizing the table entries by the row or column sums produces normalized associations and cuts N_{assoc} and N_{cut} .

A better measure of segmentation is the normalized cut, which is defined as

$$N_{cut}(A, B) = \frac{cut(A, B)}{assoc(A, V)} + \frac{cut(A, B)}{assoc(B, V)}, \quad (5.44)$$

where $assoc(A, A) = \sum_{i \in A, j \in A} w_{ij}$ is the *association* (sum of all the weights) within a cluster and $assoc(A, V) = assoc(A, A) + cut(A, B)$ is the sum of *all* the weights associated with nodes in A . Figure 5.19b shows how the cuts and associations can be thought of as area sums in the weight matrix $\mathbf{W} = [w_{ij}]$, where the entries of the matrix have been arranged so that the nodes in A come first and the nodes in B come second. Figure 5.20 shows an actual weight matrix for which these area sums can be computed. Dividing each of these areas by the corresponding row sum (the rightmost column of Figure 5.19b) results in the normalized cut and association values. These normalized values better reflect the fitness of a particular segmentation, since they look for collections of edges that are weak relative to all of the edges both inside and emanating from a particular region.

Unfortunately, computing the optimal normalized cut is NP-complete. Instead, Shi and Malik (2000) suggest computing a real-valued assignment of nodes to groups. Let \mathbf{x} be the *indicator vector* where $x_i = +1$ iff $i \in A$ and $x_i = -1$ iff $i \in B$. Let $\mathbf{d} = \mathbf{W}\mathbf{1}$ be the row sums of the symmetric matrix \mathbf{W} and $\mathbf{D} = \text{diag}(\mathbf{d})$ be the corresponding diagonal matrix. Shi and Malik (2000) show that minimizing the normalized cut over all possible indicator vectors \mathbf{x} is equivalent to minimizing

$$\min_{\mathbf{y}} \frac{\mathbf{y}^T (\mathbf{D} - \mathbf{W}) \mathbf{y}}{\mathbf{y}^T \mathbf{D} \mathbf{y}}, \quad (5.45)$$

where $\mathbf{y} = ((\mathbf{1} + \mathbf{x}) - b(\mathbf{1} - \mathbf{x}))/2$ is a vector consisting of all 1s and $-bs$ such that $\mathbf{y} \cdot \mathbf{d} = 0$. Minimizing this *Rayleigh quotient* is equivalent to solving the generalized eigenvalue system

$$(\mathbf{D} - \mathbf{W})\mathbf{y} = \lambda \mathbf{D}\mathbf{y}, \quad (5.46)$$

which can be turned into a regular eigenvalue problem

$$(\mathbf{I} - \mathbf{N})\mathbf{z} = \lambda \mathbf{z}, \quad (5.47)$$

where $\mathbf{N} = \mathbf{D}^{-1/2} \mathbf{W} \mathbf{D}^{-1/2}$ is the *normalized* affinity matrix (Weiss 1999) and $\mathbf{z} = \mathbf{D}^{1/2} \mathbf{y}$. Because these eigenvectors can be interpreted as the large modes of vibration in

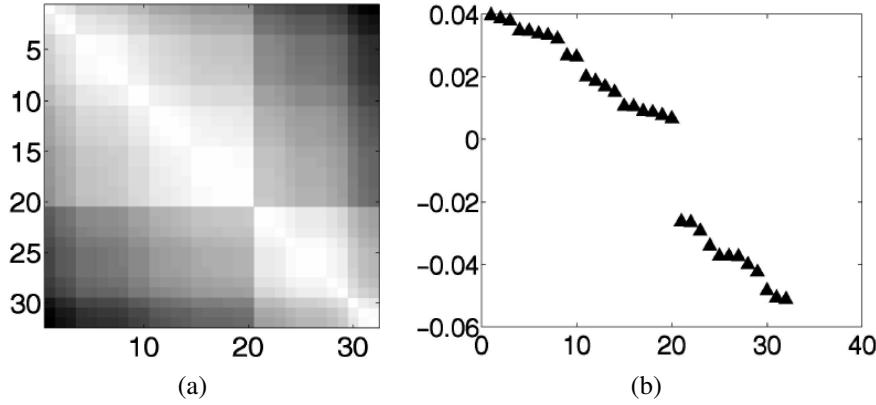


Figure 5.20 Sample weight table and its second smallest eigenvector (Shi and Malik 2000) © 2000 IEEE: (a) sample \$32 \times 32\$ weight matrix \$\mathbf{W}\$; (b) eigenvector corresponding to the second smallest eigenvalue of the generalized eigenvalue problem \$(\mathbf{D} - \mathbf{W})\mathbf{y} = \lambda\mathbf{D}\mathbf{y}\$.

a spring-mass system, normalized cuts is an example of a *spectral method* for image segmentation.

Extending an idea originally proposed by Scott and Longuet-Higgins (1990), Weiss (1999) suggests normalizing the affinity matrix and then using the top k eigenvectors to reconstitute a \mathbf{Q} matrix. Other papers have extended the basic normalized cuts framework by modifying the affinity matrix in different ways, finding better discrete solutions to the minimization problem, or applying multi-scale techniques (Meilă and Shi 2000, 2001; Ng, Jordan, and Weiss 2001; Yu and Shi 2003; Cour, Bénézit, and Shi 2005; Tolliver and Miller 2006).

Figure 5.20b shows the second smallest (real-valued) eigenvector corresponding to the weight matrix shown in Figure 5.20a. (Here, the rows have been permuted to separate the two groups of variables that belong to the different components of this eigenvector.) After this real-valued vector is computed, the variables corresponding to positive and negative eigenvector values are associated with the two cut components. This process can be further repeated to hierarchically subdivide an image, as shown in Figure 5.21.

The original algorithm proposed by Shi and Malik (2000) used spatial position and image feature differences to compute the pixel-wise affinities,

$$w_{ij} = \exp\left(-\frac{\|\mathbf{F}_i - \mathbf{F}_j\|^2}{\sigma_F^2} - \frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{\sigma_s^2}\right), \quad (5.48)$$

for pixels within a radius $\|\mathbf{x}_i - \mathbf{x}_j\| < r$, where \mathbf{F} is a feature vector that consists of intensities, colors, or oriented filter histograms. (Note how (5.48) is the negative exponential of the joint feature space distance (5.42).)

In subsequent work, Malik, Belongie, Leung *et al.* (2001) look for *intervening contours* between pixels i and j and define an intervening contour weight

$$w_{ij}^{IC} = 1 - \max_{\mathbf{x} \in l_{ij}} p_{con}(\mathbf{x}), \quad (5.49)$$

where l_{ij} is the image line joining pixels i and j and $p_{con}(\mathbf{x})$ is the probability of an intervening contour perpendicular to this line, which is defined as the negative exponential of the

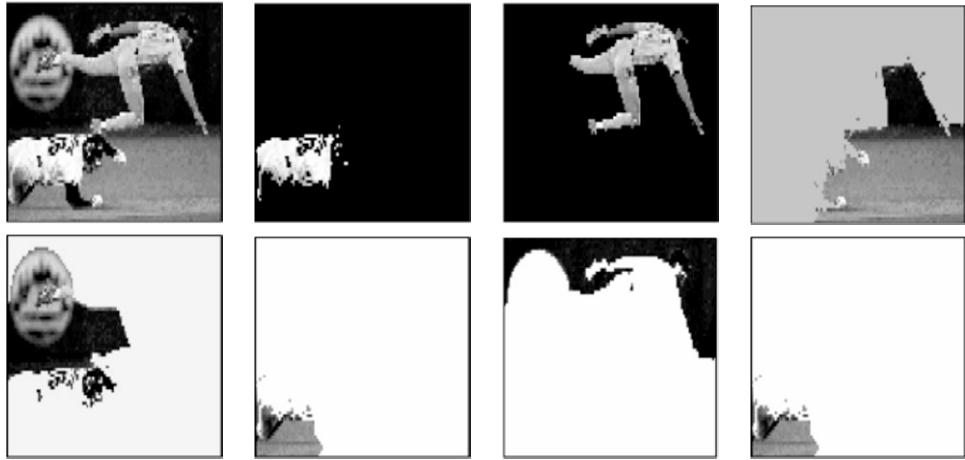


Figure 5.21 Normalized cuts segmentation (Shi and Malik 2000) © 2000 IEEE: The input image and the components returned by the normalized cuts algorithm.

oriented energy in the perpendicular direction. They multiply these weights with a texton-based texture similarity metric and use an initial over-segmentation based purely on local pixel-wise features to re-estimate intervening contours and texture statistics in a region-based manner. Figure 5.22 shows the results of running this improved algorithm on a number of test images.

Because it requires the solution of large sparse eigenvalue problems, normalized cuts can be quite slow. Sharon, Galun, Sharon *et al.* (2006) present a way to accelerate the computation of the normalized cuts using an approach inspired by algebraic multigrid (Brandt 1986; Briggs, Henson, and McCormick 2000). To coarsen the original problem, they select a smaller number of variables such that the remaining fine-level variables are *strongly coupled* to at least one coarse-level variable. Figure 5.15 shows this process schematically, while (5.25) gives the definition for strong coupling except that, in this case, the original weights w_{ij} in the normalized cut are used instead of merge probabilities p_{ij} .

Once a set of coarse variables has been selected, an inter-level interpolation matrix with elements similar to the left hand side of (5.25) is used to define a reduced version of the normalized cuts problem. In addition to computing the weight matrix using interpolation-based coarsening, additional region statistics are used to modulate the weights. After a normalized cut has been computed at the coarsest level of analysis, the membership values of finer-level nodes are computed by interpolating parent values and mapping values within $\epsilon = 0.1$ of 0 and 1 to pure Boolean values.

An example of the segmentation produced by weighted aggregation (SWA) is shown in Figure 5.22, along with the most recent probabilistic bottom-up merging algorithm by Alpert, Galun, Basri *et al.* (2007), which was described in Section 5.2. In even more recent work, Wang and Oliensis (2010) show how to estimate statistics over segmentations (e.g., mean region size) directly from the affinity graph. They use this to produce segmentations that are more *central* with respect to other possible segmentations.

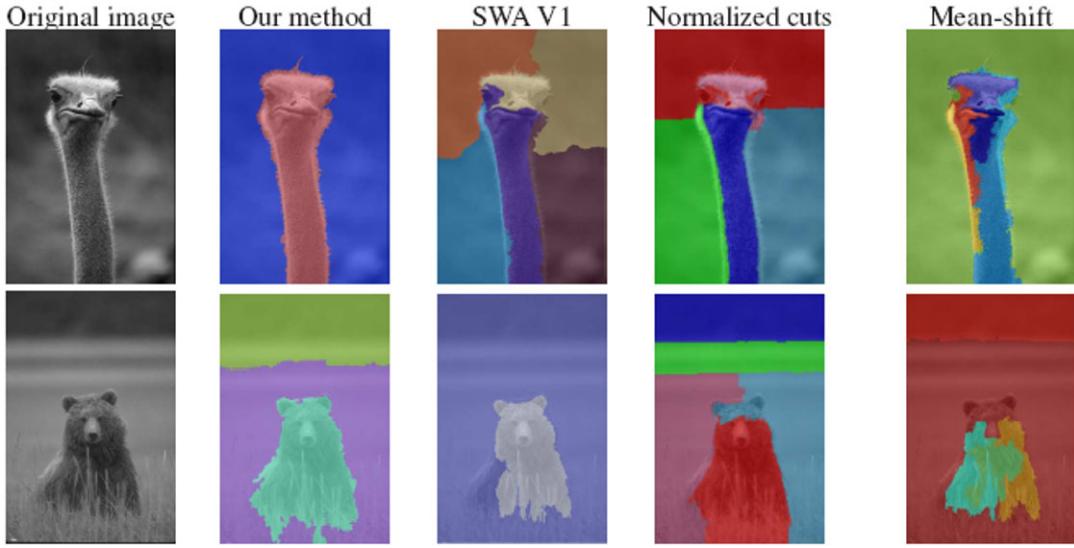


Figure 5.22 Comparative segmentation results (Alpert, Galun, Basri *et al.* 2007) © 2007 IEEE. “Our method” refers to the probabilistic bottom-up merging algorithm developed by Alpert *et al.*

5.5 Graph cuts and energy-based methods

A common theme in image segmentation algorithms is the desire to group pixels that have similar appearance (statistics) and to have the boundaries between pixels in different regions be of short length and across visible discontinuities. If we restrict the boundary measurements to be between immediate neighbors and compute region membership statistics by summing over pixels, we can formulate this as a classic pixel-based energy function using either a *variational formulation* (regularization, see Section 3.7.1) or as a binary Markov random field (Section 3.7.2).

Examples of the continuous approach include (Mumford and Shah 1989; Chan and Vese 1992; Zhu and Yuille 1996; Tabb and Ahuja 1997) along with the level set approaches discussed in Section 5.1.4. An early example of a discrete labeling problem that combines both region-based and boundary-based energy terms is the work of Leclerc (1989), who used minimum description length (MDL) coding to derive the energy function being minimized. Boykov and Funka-Lea (2006) present a wonderful survey of various energy-based techniques for binary object segmentation, some of which we discuss below.

As we saw in Section 3.7.2, the energy corresponding to a segmentation problem can be written (c.f. Equations (3.100) and (3.108–3.113)) as

$$E(f) = \sum_{i,j} E_r(i,j) + E_b(i,j), \quad (5.50)$$

where the region term

$$E_r(i,j) = E_S(I(i,j); R(f(i,j))) \quad (5.51)$$

is the negative log likelihood that pixel intensity (or color) $I(i,j)$ is consistent with the statis-

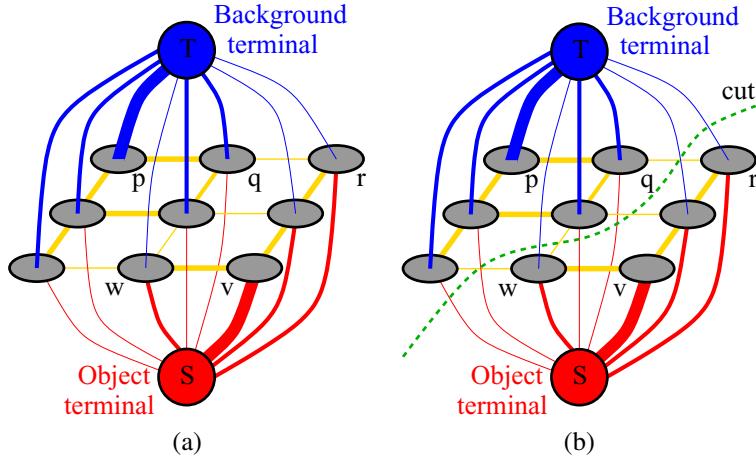


Figure 5.23 Graph cuts for region segmentation (Boykov and Jolly 2001) © 2001 IEEE: (a) the energy function is encoded as a maximum flow problem; (b) the minimum cut determines the region boundary.

tics of region $R(f(i, j))$ and the boundary term

$$E_b(i, j) = s_x(i, j)\delta(f(i, j) - f(i + 1, j)) + s_y(i, j)\delta(f(i, j) - f(i, j + 1)) \quad (5.52)$$

measures the inconsistency between \mathcal{N}_4 neighbors modulated by local horizontal and vertical smoothness terms $s_x(i, j)$ and $s_y(i, j)$.

Region statistics can be something as simple as the mean gray level or color (Leclerc 1989), in which case

$$E_S(I; \mu_k) = \|I - \mu_k\|^2. \quad (5.53)$$

Alternatively, they can be more complex, such as region intensity histograms (Boykov and Jolly 2001) or color Gaussian mixture models (Rother, Kolmogorov, and Blake 2004). For smoothness (boundary) terms, it is common to make the strength of the smoothness $s_x(i, j)$ inversely proportional to the local edge strength (Boykov, Veksler, and Zabih 2001).

Originally, energy-based segmentation problems were optimized using iterative gradient descent techniques, which were slow and prone to getting trapped in local minima. Boykov and Jolly (2001) were the first to apply the binary MRF optimization algorithm developed by Greig, Porteous, and Seheult (1989) to binary object segmentation.

In this approach, the user first delineates pixels in the background and foreground regions using a few strokes of an image brush (Figure 3.61). These pixels then become the *seeds* that tie nodes in the *S-T graph* to the source and sink labels *S* and *T* (Figure 5.23a). Seed pixels can also be used to estimate foreground and background region statistics (intensity or color histograms).

The capacities of the other edges in the graph are derived from the region and boundary energy terms, i.e., pixels that are more compatible with the foreground or background region get stronger connections to the respective source or sink; adjacent pixels with greater smoothness also get stronger links. Once the minimum-cut/maximum-flow problem has been solved using a polynomial time algorithm (Goldberg and Tarjan 1988; Boykov and Kolmogorov 2004), pixels on either side of the computed cut are labeled according to the source or sink to

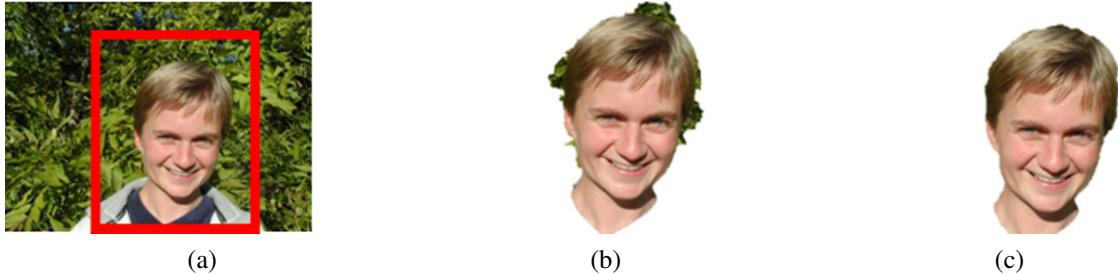


Figure 5.24 GrabCut image segmentation (Rother, Kolmogorov, and Blake 2004) © 2004 ACM: (a) the user draws a bounding box in red; (b) the algorithm guesses color distributions for the object and background and performs a binary segmentation; (c) the process is repeated with better region statistics.

which they remain connected (Figure 5.23b). While graph cuts is just one of several known techniques for MRF energy minimization (Appendix B.5.4), it is still the one most commonly used for solving binary MRF problems.

The basic binary segmentation algorithm of Boykov and Jolly (2001) has been extended in a number of directions. The *GrabCut* system of Rother, Kolmogorov, and Blake (2004) iteratively re-estimates the region statistics, which are modeled as a mixtures of Gaussians in color space. This allows their system to operate given minimal user input, such as a single bounding box (Figure 5.24a)—the background color model is initialized from a strip of pixels around the box outline. (The foreground color model is initialized from the interior pixels, but quickly converges to a better estimate of the object.) The user can also place additional strokes to refine the segmentation as the solution progresses. In more recent work, Cui, Yang, Wen *et al.* (2008) use color and edge models derived from previous segmentations of similar objects to improve the local models used in GrabCut.

Another major extension to the original binary segmentation formulation is the addition of *directed edges*, which allows boundary regions to be oriented, e.g., to prefer light to dark transitions or *vice versa* (Kolmogorov and Boykov 2005). Figure 5.25 shows an example where the directed graph cut correctly segments the light gray liver from its dark gray surround. The same approach can be used to measure the *flux* exiting a region, i.e., the signed gradient projected normal to the region boundary. Combining oriented graphs with larger neighborhoods enables approximating continuous problems such as those traditionally solved using level sets in the globally optimal graph cut framework (Boykov and Kolmogorov 2003; Kolmogorov and Boykov 2005).

Even more recent developments in graph cut-based segmentation techniques include the addition of connectivity priors to force the foreground to be in a single piece (Vicente, Kolmogorov, and Rother 2008) and shape priors to use knowledge about an object’s shape during the segmentation process (Lempitsky and Boykov 2007; Lempitsky, Blake, and Rother 2008).

While optimizing the binary MRF energy (5.50) requires the use of combinatorial optimization techniques, such as maximum flow, an approximate solution can be obtained by converting the binary energy terms into quadratic energy terms defined over a continuous $[0, 1]$ random field, which then becomes a classical membrane-based regularization problem (3.100–3.102). The resulting quadratic energy function can then be solved using standard linear system solvers (3.102–3.103), although if speed is an issue, you should use multigrid

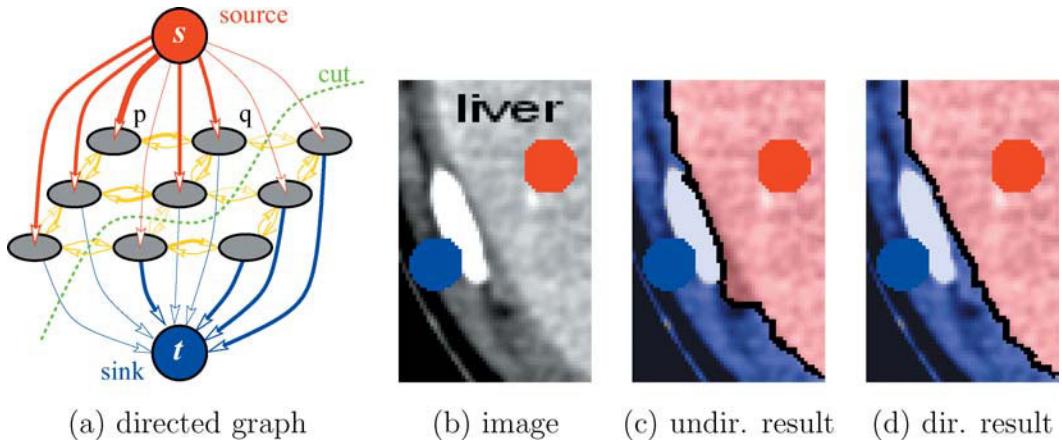


Figure 5.25 Segmentation with a directed graph cut (Boykov and Funka-Lea 2006) © 2006 Springer: (a) directed graph; (b) image with seed points; (c) the undirected graph incorrectly continues the boundary along the bright object; (d) the directed graph correctly segments the light gray region from its darker surround.

or one of its variants (Appendix A.5). Once the continuous solution has been computed, it can be thresholded at 0.5 to yield a binary segmentation.

The $[0, 1]$ continuous optimization problem can also be interpreted as computing the probability at each pixel that a *random walker* starting at that pixel ends up at one of the labeled seed pixels, which is also equivalent to computing the potential in a resistive grid where the resistors are equal to the edge weights (Grady 2006; Sinop and Grady 2007). K -way segmentations can also be computed by iterating through the seed labels, using a binary problem with one label set to 1 and all the others set to 0 to compute the relative membership probabilities for each pixel. In follow-on work, Grady and Ali (2008) use a precomputation of the eigenvectors of the linear system to make the solution with a novel set of seeds faster, which is related to the Laplacian matting problem presented in Section 10.4.3 (Levin, Acha, and Lischinski 2008). Couplie, Grady, Najman *et al.* (2009) relate the random walker to watersheds and other segmentation techniques. Singaraju, Grady, and Vidal (2008) add directed-edge constraints in order to support flux, which makes the energy piecewise quadratic and hence not solvable as a single linear system. The random walker algorithm can also be used to solve the Mumford–Shah segmentation problem (Grady and Alvino 2008) and to compute fast multigrid solutions (Grady 2008). A nice review of these techniques is given by Singaraju, Grady, Sinop *et al.* (2010).

An even faster way to compute a continuous $[0, 1]$ approximate segmentation is to compute *weighted geodesic distances* between the 0 and 1 seed regions (Bai and Sapiro 2009), which can also be used to estimate soft alpha mattes (Section 10.4.3). A related approach by Criminisi, Sharp, and Blake (2008) can be used to find fast approximate solutions to general binary Markov random field optimization problems.

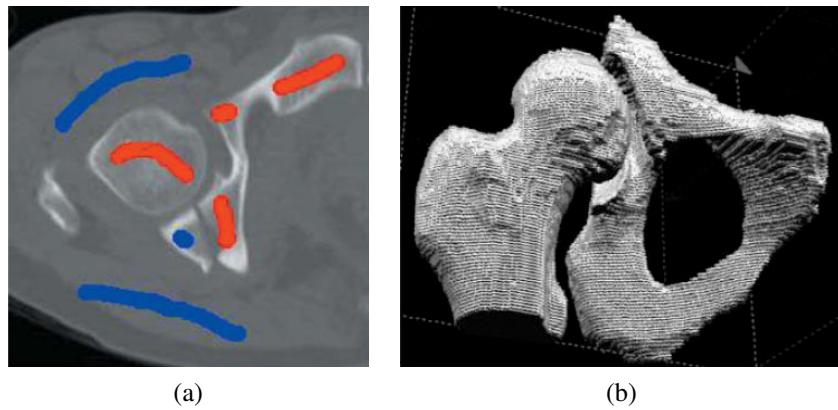


Figure 5.26 3D volumetric medical image segmentation using graph cuts (Boykov and Funka-Lea 2006) © 2006 Springer: (a) computed tomography (CT) slice with some seeds; (b) recovered 3D volumetric bone model (on a $256 \times 256 \times 119$ voxel grid).

5.5.1 Application: Medical image segmentation

One of the most promising applications of image segmentation is in the medical imaging domain, where it can be used to segment anatomical tissues for later quantitative analysis. Figure 5.25 shows a binary graph cut with directed edges being used to segment the liver tissue (light gray) from its surrounding bone (white) and muscle (dark gray) tissue. Figure 5.26 shows the segmentation of bones in a $256 \times 256 \times 119$ computed X-ray tomography (CT) volume. Without the powerful optimization techniques available in today's image segmentation algorithms, such processing used to require much more laborious manual tracing of individual X-ray slices.

The fields of medical image segmentation (McInerney and Terzopoulos 1996) and medical image registration (Kybic and Unser 2003) (Section 8.3.1) are rich research fields with their own specialized conferences, such as *Medical Imaging Computing and Computer Assisted Intervention (MICCAI)*,¹¹ and journals, such as *Medical Image Analysis* and *IEEE Transactions on Medical Imaging*. These can be great sources of references and ideas for research in this area.

5.6 Additional reading

The topic of image segmentation is closely related to clustering techniques, which are treated in a number of monographs and review articles (Jain and Dubes 1988; Kaufman and Rousseeuw 1990; Jain, Duin, and Mao 2000; Jain, Topchy, Law *et al.* 2004). Some early segmentation techniques include those described by Brice and Fennema (1970); Pavlidis (1977); Riseman and Arbib (1977); Ohlander, Price, and Reddy (1978); Rosenfeld and Davis (1979); Haralick and Shapiro (1985), while examples of newer techniques are developed by Leclerc (1989); Mumford and Shah (1989); Shi and Malik (2000); Felzenszwalb and Huttenlocher (2004b).

¹¹<http://www.miccai.org/>.

Arbeláez, Maire, Fowlkes *et al.* (2010) provide a good review of automatic segmentation techniques and also compare their performance on the Berkeley Segmentation Dataset and Benchmark (Martin, Fowlkes, Tal *et al.* 2001).¹² Additional comparison papers and databases include those by Unnikrishnan, Pantofaru, and Hebert (2007); Alpert, Galun, Basri *et al.* (2007); Estrada and Jepson (2009).

The topic of active contours has a long history, beginning with the seminal work on snakes and other energy-minimizing variational methods (Kass, Witkin, and Terzopoulos 1988; Cootes, Cooper, Taylor *et al.* 1995; Blake and Isard 1998), continuing through techniques such as intelligent scissors (Mortensen and Barrett 1995, 1999; Pérez, Blake, and Gangnet 2001), and culminating in level sets (Malladi, Sethian, and Vemuri 1995; Caselles, Kimmel, and Sapiro 1997; Sethian 1999; Paragios and Deriche 2000; Sapiro 2001; Osher and Paragios 2003; Paragios, Faugeras, Chan *et al.* 2005; Cremers, Rousson, and Deriche 2007; Rousson and Paragios 2008; Paragios and Sgallari 2009), which are currently the most widely used active contour methods.

Techniques for segmenting images based on local pixel similarities combined with aggregation or splitting methods include watersheds (Vincent and Soille 1991; Beare 2006; Arbeláez, Maire, Fowlkes *et al.* 2010), region splitting (Ohlander, Price, and Reddy 1978), region merging (Brice and Fennema 1970; Pavlidis and Liow 1990; Jain, Topchy, Law *et al.* 2004), as well as graph-based and probabilistic multi-scale approaches (Felzenszwalb and Huttenlocher 2004b; Alpert, Galun, Basri *et al.* 2007).

Mean-shift algorithms, which find modes (peaks) in a density function representation of the pixels, are presented by Comaniciu and Meer (2002); Paris and Durand (2007). Parametric mixtures of Gaussians can also be used to represent and segment such pixel densities (Bishop 2006; Ma, Derksen, Hong *et al.* 2007).

The seminal work on spectral (eigenvalue) methods for image segmentation is the *normalized cut* algorithm of Shi and Malik (2000). Related work includes that by Weiss (1999); Meilă and Shi (2000, 2001); Malik, Belongie, Leung *et al.* (2001); Ng, Jordan, and Weiss (2001); Yu and Shi (2003); Cour, Bénézit, and Shi (2005); Sharon, Galun, Sharon *et al.* (2006); Tolliver and Miller (2006); Wang and Oliensis (2010).

Continuous-energy-based (variational) approaches to interactive segmentation include Leclerc (1989); Mumford and Shah (1989); Chan and Vese (1992); Zhu and Yuille (1996); Tabb and Ahuja (1997). Discrete variants of such problems are usually optimized using binary graph cuts or other combinatorial energy minimization methods (Boykov and Jolly 2001; Boykov and Kolmogorov 2003; Rother, Kolmogorov, and Blake 2004; Kolmogorov and Boykov 2005; Cui, Yang, Wen *et al.* 2008; Vicente, Kolmogorov, and Rother 2008; Lempitsky and Boykov 2007; Lempitsky, Blake, and Rother 2008), although continuous optimization techniques followed by thresholding can also be used (Grady 2006; Grady and Ali 2008; Singaraju, Grady, and Vidal 2008; Criminisi, Sharp, and Blake 2008; Grady 2008; Bai and Sapiro 2009; Couplie, Grady, Najman *et al.* 2009). Boykov and Funka-Lea (2006) present a good survey of various energy-based techniques for binary object segmentation.

¹² <http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/>.

5.7 Exercises

Ex 5.1: Snake evolution Prove that, in the absence of external forces, a snake will always shrink to a small circle and eventually a single point, regardless of whether first- or second-order smoothness (or some combination) is used.

(Hint: If you can show that the evolution of the $x(s)$ and $y(s)$ components are independent, you can analyze the 1D case more easily.)

Ex 5.2: Snake tracker Implement a snake-based contour tracker:

1. Decide whether to use a large number of contour points or a smaller number interpolated with a B-spline.
2. Define your internal smoothness energy function and decide what image-based attractive forces to use.
3. At each iteration, set up the banded linear system of equations (quadratic energy function) and solve it using banded Cholesky factorization (Appendix A.4).

Ex 5.3: Intelligent scissors Implement the intelligent scissors (live-wire) interactive segmentation algorithm (Mortensen and Barrett 1995) and design a graphical user interface (GUI) to let you draw such curves over an image and use them for segmentation.

Ex 5.4: Region segmentation Implement one of the region segmentation algorithms described in this chapter. Some popular segmentation algorithms include:

- k-means (Section 5.3.1);
- mixtures of Gaussians (Section 5.3.1);
- mean shift (Section 5.3.2) and Exercise 5.5;
- normalized cuts (Section 5.4);
- similarity graph-based segmentation (Section 5.2.4);
- binary Markov random fields solved using graph cuts (Section 5.5).

Apply your region segmentation to a video sequence and use it to track moving regions from frame to frame.

Alternatively, test out your segmentation algorithm on the Berkeley segmentation database (Martin, Fowlkes, Tal *et al.* 2001).

Ex 5.5: Mean shift Develop a mean-shift segmentation algorithm for color images (Comaniciu and Meer 2002).

1. Convert your image to L*a*b* space, or keep the original RGB colors, and augment them with the pixel (x, y) locations.
2. For every pixel (L, a, b, x, y) , compute the weighted mean of its neighbors using either a unit ball (Epanechnikov kernel) or finite-radius Gaussian, or some other kernel of your choosing. Weight the color and spatial scales differently, e.g., using values of $(h_s, h_r, M) = (16, 19, 40)$ as shown in Figure 5.18.

3. Replace the current value with this weighted mean and iterate until either the motion is below a threshold or a finite number of steps has been taken.
4. Cluster all final values (modes) that are within a threshold, i.e., find the connected components. Since each pixel is associated with a final mean-shift (mode) value, this results in an image segmentation, i.e., each pixel is labeled with its final component.
5. (Optional) Use a random subset of the pixels as starting points and find which component each unlabeled pixel belongs to, either by finding its nearest neighbor or by iterating the mean shift until it finds a neighboring track of mean-shift values. Describe the data structures you use to make this efficient.
6. (Optional) Mean shift divides the kernel density function estimate by the local weighting to obtain a step size that is guaranteed to converge but may be slow. Use an alternative step size estimation algorithm from the optimization literature to see if you can make the algorithm converge faster.

Chapter 6

Feature-based alignment

6.1	2D and 3D feature-based alignment	275
6.1.1	2D alignment using least squares	275
6.1.2	<i>Application:</i> Panography	277
6.1.3	Iterative algorithms	278
6.1.4	Robust least squares and RANSAC	281
6.1.5	3D alignment	283
6.2	Pose estimation	284
6.2.1	Linear algorithms	284
6.2.2	Iterative algorithms	286
6.2.3	<i>Application:</i> Augmented reality	287
6.3	Geometric intrinsic calibration	288
6.3.1	Calibration patterns	289
6.3.2	Vanishing points	290
6.3.3	<i>Application:</i> Single view metrology	292
6.3.4	Rotational motion	293
6.3.5	Radial distortion	295
6.4	Additional reading	296
6.5	Exercises	296

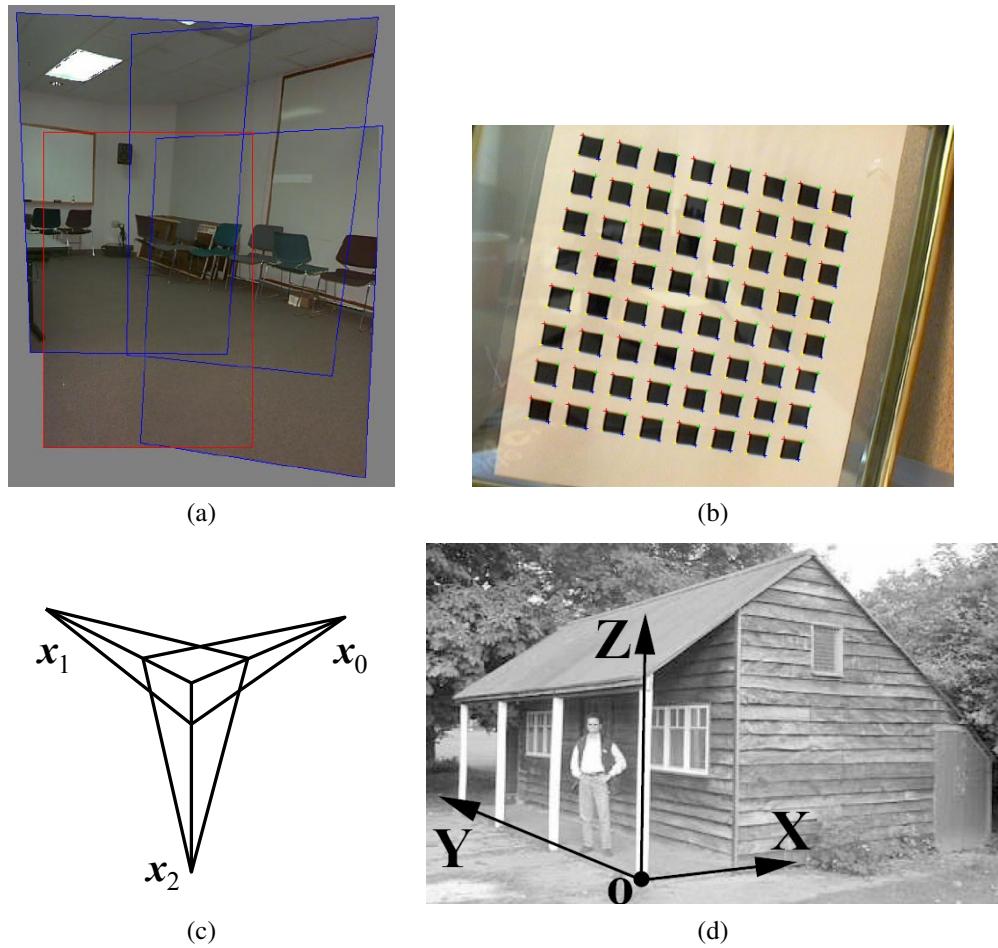


Figure 6.1 Geometric alignment and calibration: (a) geometric alignment of 2D images for stitching (Szeliski and Shum 1997) © 1997 ACM; (b) a two-dimensional calibration target (Zhang 2000) © 2000 IEEE; (c) calibration from vanishing points; (d) scene with easy-to-find lines and vanishing directions (Criminisi, Reid, and Zisserman 2000) © 2000 Springer.

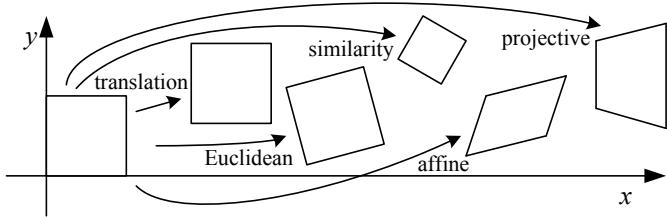


Figure 6.2 Basic set of 2D planar transformations

Once we have extracted features from images, the next stage in many vision algorithms is to match these features across different images (Section 4.1.3). An important component of this matching is to verify whether the set of matching features is geometrically consistent, e.g., whether the feature displacements can be described by a simple 2D or 3D geometric transformation. The computed motions can then be used in other applications such as image stitching (Chapter 9) or augmented reality (Section 6.2.3).

In this chapter, we look at the topic of geometric image registration, i.e., the computation of 2D and 3D transformations that map features in one image to another (Section 6.1). One special case of this problem is *pose estimation*, which is determining a camera's position relative to a known 3D object or scene (Section 6.2). Another case is the computation of a camera's *intrinsic calibration*, which consists of the internal parameters such as focal length and radial distortion (Section 6.3). In Chapter 7, we look at the related problems of how to estimate 3D point structure from 2D matches (*triangulation*) and how to simultaneously estimate 3D geometry and camera motion (*structure from motion*).

6.1 2D and 3D feature-based alignment

Feature-based alignment is the problem of estimating the motion between two or more sets of matched 2D or 3D points. In this section, we restrict ourselves to global *parametric* transformations, such as those described in Section 2.1.2 and shown in Table 2.1 and Figure 6.2, or higher order transformation for curved surfaces (Shashua and Toelg 1997; Can, Stewart, Roysam *et al.* 2002). Applications to non-rigid or elastic deformations (Bookstein 1989; Szeliski and Lavallée 1996; Torresani, Hertzmann, and Bregler 2008) are examined in Sections 8.3 and 12.6.4.

6.1.1 2D alignment using least squares

Given a set of matched feature points $\{(x_i, x'_i)\}$ and a planar parametric transformation¹ of the form

$$x' = f(x; p), \quad (6.1)$$

¹ For examples of non-planar parametric models, such as quadrics, see the work of Shashua and Toelg (1997); Shashua and Wexler (2001).

Transform	Matrix	Parameters \mathbf{p}	Jacobian \mathbf{J}
translation	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$	(t_x, t_y)	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
Euclidean	$\begin{bmatrix} c_\theta & -s_\theta & t_x \\ s_\theta & c_\theta & t_y \end{bmatrix}$	(t_x, t_y, θ)	$\begin{bmatrix} 1 & 0 & -s_\theta x - c_\theta y \\ 0 & 1 & c_\theta x - s_\theta y \end{bmatrix}$
similarity	$\begin{bmatrix} 1+a & -b & t_x \\ b & 1+a & t_y \end{bmatrix}$	(t_x, t_y, a, b)	$\begin{bmatrix} 1 & 0 & x & -y \\ 0 & 1 & y & x \end{bmatrix}$
affine	$\begin{bmatrix} 1+a_{00} & a_{01} & t_x \\ a_{10} & 1+a_{11} & t_y \end{bmatrix}$	$(t_x, t_y, a_{00}, a_{01}, a_{10}, a_{11})$	$\begin{bmatrix} 1 & 0 & x & y & 0 & 0 \\ 0 & 1 & 0 & 0 & x & y \end{bmatrix}$
projective	$\begin{bmatrix} 1+h_{00} & h_{01} & h_{02} \\ h_{10} & 1+h_{11} & h_{12} \\ h_{20} & h_{21} & 1 \end{bmatrix}$	$(h_{00}, h_{01}, \dots, h_{21})$	(see Section 6.1.3)

Table 6.1 Jacobians of the 2D coordinate transformations $\mathbf{x}' = \mathbf{f}(\mathbf{x}; \mathbf{p})$ shown in Table 2.1, where we have re-parameterized the motions so that they are identity for $\mathbf{p} = 0$.

how can we produce the best estimate of the motion parameters \mathbf{p} ? The usual way to do this is to use least squares, i.e., to minimize the sum of squared residuals

$$E_{\text{LS}} = \sum_i \|\mathbf{r}_i\|^2 = \sum_i \|\mathbf{f}(\mathbf{x}_i; \mathbf{p}) - \mathbf{x}'_i\|^2, \quad (6.2)$$

where

$$\mathbf{r}_i = \mathbf{f}(\mathbf{x}_i; \mathbf{p}) - \mathbf{x}'_i = \hat{\mathbf{x}}'_i - \tilde{\mathbf{x}}'_i \quad (6.3)$$

is the *residual* between the measured location $\hat{\mathbf{x}}'_i$ and its corresponding current location $\tilde{\mathbf{x}}'_i = \mathbf{f}(\mathbf{x}_i; \mathbf{p})$. (See Appendix A.2 for more on least squares and Appendix B.2 for a statistical justification.)

Many of the motion models presented in Section 2.1.2 and Table 2.1, i.e., translation, similarity, and affine, have a *linear* relationship between the amount of motion $\Delta\mathbf{x} = \mathbf{x}' - \mathbf{x}$ and the unknown parameters \mathbf{p} ,

$$\Delta\mathbf{x} = \mathbf{x}' - \mathbf{x} = \mathbf{J}(\mathbf{x})\mathbf{p}, \quad (6.4)$$

where $\mathbf{J} = \partial\mathbf{f}/\partial\mathbf{p}$ is the *Jacobian* of the transformation \mathbf{f} with respect to the motion parameters \mathbf{p} (see Table 6.1). In this case, a simple *linear* regression (linear least squares problem) can be formulated as

$$E_{\text{LLS}} = \sum_i \|\mathbf{J}(\mathbf{x}_i)\mathbf{p} - \Delta\mathbf{x}_i\|^2 \quad (6.5)$$

$$= \mathbf{p}^T \left[\sum_i \mathbf{J}^T(\mathbf{x}_i) \mathbf{J}(\mathbf{x}_i) \right] \mathbf{p} - 2\mathbf{p}^T \left[\sum_i \mathbf{J}^T(\mathbf{x}_i) \Delta\mathbf{x}_i \right] + \sum_i \|\Delta\mathbf{x}_i\|^2 \quad (6.6)$$

$$= \mathbf{p}^T \mathbf{A}\mathbf{p} - 2\mathbf{p}^T \mathbf{b} + c. \quad (6.7)$$

The minimum can be found by solving the symmetric positive definite (SPD) system of *normal equations*²

$$\mathbf{A}\mathbf{p} = \mathbf{b}, \quad (6.8)$$

where

$$\mathbf{A} = \sum_i \mathbf{J}^T(\mathbf{x}_i) \mathbf{J}(\mathbf{x}_i) \quad (6.9)$$

is called the *Hessian* and $\mathbf{b} = \sum_i \mathbf{J}^T(\mathbf{x}_i) \Delta \mathbf{x}_i$. For the case of pure translation, the resulting equations have a particularly simple form, i.e., the translation is the average translation between corresponding points or, equivalently, the translation of the point centroids.

Uncertainty weighting. The above least squares formulation assumes that all feature points are matched with the same accuracy. This is often not the case, since certain points may fall into more textured regions than others. If we associate a scalar variance estimate σ_i^2 with each correspondence, we can minimize the *weighted least squares* problem instead,³

$$E_{\text{WLS}} = \sum_i \sigma_i^{-2} \|\mathbf{r}_i\|^2. \quad (6.10)$$

As shown in Section 8.1.3, a covariance estimate for patch-based matching can be obtained by multiplying the inverse of the *patch Hessian* \mathbf{A}_i (8.55) with the per-pixel noise covariance σ_n^2 (8.44). Weighting each squared residual by its inverse covariance $\Sigma_i^{-1} = \sigma_n^{-2} \mathbf{A}_i$ (which is called the *information matrix*), we obtain

$$E_{\text{CWLS}} = \sum_i \|\mathbf{r}_i\|_{\Sigma_i^{-1}}^2 = \sum_i \mathbf{r}_i^T \Sigma_i^{-1} \mathbf{r}_i = \sum_i \sigma_n^{-2} \mathbf{r}_i^T \mathbf{A}_i \mathbf{r}_i. \quad (6.11)$$

6.1.2 Application: Panography

One of the simplest (and most fun) applications of image alignment is a special form of image stitching called *panography*. In a panograph, images are translated and optionally rotated and scaled before being blended with simple averaging (Figure 6.3). This process mimics the photographic collages created by artist David Hockney, although his compositions use an opaque overlay model, being created out of regular photographs.

In most of the examples seen on the Web, the images are aligned by hand for best artistic effect.⁴ However, it is also possible to use feature matching and alignment techniques to perform the registration automatically (Nomura, Zhang, and Nayar 2007; Zelnik-Manor and Perona 2007).

Consider a simple translational model. We want all the corresponding features in different images to line up as best as possible. Let \mathbf{t}_j be the location of the j th image coordinate frame in the global composite frame and \mathbf{x}_{ij} be the location of the i th matched feature in the j th image. In order to align the images, we wish to minimize the least squares error

$$E_{\text{PLS}} = \sum_{ij} \|(\mathbf{t}_j + \mathbf{x}_{ij}) - \mathbf{x}_i\|^2, \quad (6.12)$$

² For poorly conditioned problems, it is better to use QR decomposition on the set of linear equations $\mathbf{J}(\mathbf{x}_i)\mathbf{p} = \Delta \mathbf{x}_i$ instead of the normal equations (Björck 1996; Golub and Van Loan 1996). However, such conditions rarely arise in image registration.

³ Problems where each measurement can have a different variance or certainty are called *heteroscedastic models*.

⁴ <http://www.flickr.com/groups/panography/>.

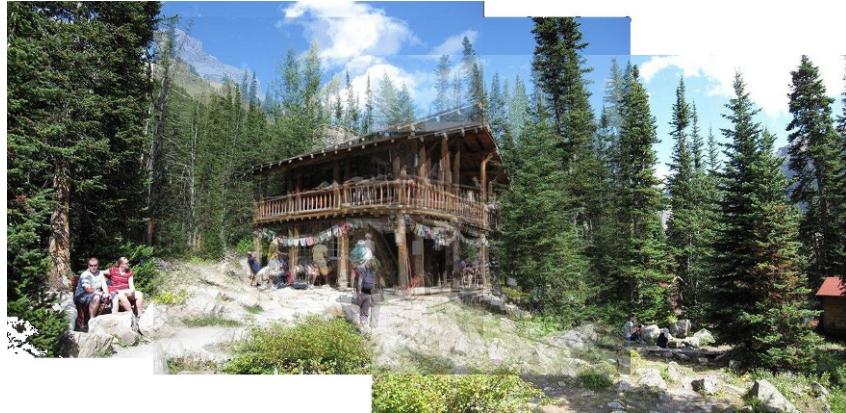


Figure 6.3 A simple panograph consisting of three images automatically aligned with a translational model and then averaged together.

where \mathbf{x}_i is the consensus (average) position of feature i in the global coordinate frame. (An alternative approach is to register each pair of overlapping images separately and then compute a consensus location for each frame—see Exercise 6.2.)

The above least squares problem is indeterminate (you can add a constant offset to all the frame and point locations \mathbf{t}_j and \mathbf{x}_i). To fix this, either pick one frame as being at the origin or add a constraint to make the average frame offsets be 0.

The formulas for adding rotation and scale transformations are straightforward and are left as an exercise (Exercise 6.2). See if you can create some collages that you would be happy to share with others on the Web.

6.1.3 Iterative algorithms

While linear least squares is the simplest method for estimating parameters, most problems in computer vision do not have a simple linear relationship between the measurements and the unknowns. In this case, the resulting problem is called *non-linear least squares* or *non-linear regression*.

Consider, for example, the problem of estimating a rigid Euclidean 2D transformation (translation plus rotation) between two sets of points. If we parameterize this transformation by the translation amount (t_x, t_y) and the rotation angle θ , as in Table 2.1, the Jacobian of this transformation, given in Table 6.1, depends on the current value of θ . Notice how in Table 6.1, we have re-parameterized the motion matrices so that they are always the identity at the origin $\mathbf{p} = 0$, which makes it easier to initialize the motion parameters.

To minimize the non-linear least squares problem, we iteratively find an update $\Delta\mathbf{p}$ to the current parameter estimate \mathbf{p} by minimizing

$$E_{\text{NLS}}(\Delta\mathbf{p}) = \sum_i \|\mathbf{f}(\mathbf{x}_i; \mathbf{p} + \Delta\mathbf{p}) - \mathbf{x}'_i\|^2 \quad (6.13)$$

$$\approx \sum_i \|\mathbf{J}(\mathbf{x}_i; \mathbf{p})\Delta\mathbf{p} - \mathbf{r}_i\|^2 \quad (6.14)$$

$$= \Delta\mathbf{p}^T \left[\sum_i \mathbf{J}^T \mathbf{J} \right] \Delta\mathbf{p} - 2\Delta\mathbf{p}^T \left[\sum_i \mathbf{J}^T \mathbf{r}_i \right] + \sum_i \|\mathbf{r}_i\|^2 \quad (6.15)$$

$$= \Delta\mathbf{p}^T \mathbf{A} \Delta\mathbf{p} - 2\Delta\mathbf{p}^T \mathbf{b} + c, \quad (6.16)$$

where the “Hessian”⁵ \mathbf{A} is the same as Equation (6.9) and the right hand side vector

$$\mathbf{b} = \sum_i \mathbf{J}^T(\mathbf{x}_i) \mathbf{r}_i \quad (6.17)$$

is now a Jacobian-weighted sum of residual vectors. This makes intuitive sense, as the parameters are pulled in the direction of the prediction error with a strength proportional to the Jacobian.

Once \mathbf{A} and \mathbf{b} have been computed, we solve for $\Delta\mathbf{p}$ using

$$(\mathbf{A} + \lambda \text{diag}(\mathbf{A})) \Delta\mathbf{p} = \mathbf{b}, \quad (6.18)$$

and update the parameter vector $\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p}$ accordingly. The parameter λ is an additional damping parameter used to ensure that the system takes a “downhill” step in energy (squared error) and is an essential component of the Levenberg–Marquardt algorithm (described in more detail in Appendix A.3). In many applications, it can be set to 0 if the system is successfully converging.

For the case of our 2D translation+rotation, we end up with a 3×3 set of normal equations in the unknowns $(\delta t_x, \delta t_y, \delta\theta)$. An initial guess for (t_x, t_y, θ) can be obtained by fitting a four-parameter similarity transform in (t_x, t_y, c, s) and then setting $\theta = \tan^{-1}(s/c)$. An alternative approach is to estimate the translation parameters using the centroids of the 2D points and to then estimate the rotation angle using polar coordinates (Exercise 6.3).

For the other 2D motion models, the derivatives in Table 6.1 are all fairly straightforward, except for the projective 2D motion (homography), which arises in image-stitching applications (Chapter 9). These equations can be re-written from (2.21) in their new parametric form as

$$x' = \frac{(1 + h_{00})x + h_{01}y + h_{02}}{h_{20}x + h_{21}y + 1} \quad \text{and} \quad y' = \frac{h_{10}x + (1 + h_{11})y + h_{12}}{h_{20}x + h_{21}y + 1}. \quad (6.19)$$

The Jacobian is therefore

$$\mathbf{J} = \frac{\partial \mathbf{f}}{\partial \mathbf{p}} = \frac{1}{D} \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -x'x & -x'y \\ 0 & 0 & 0 & x & y & 1 & -y'x & -y'y \end{bmatrix}, \quad (6.20)$$

where $D = h_{20}x + h_{21}y + 1$ is the denominator in (6.19), which depends on the current parameter settings (as do x' and y').

An initial guess for the eight unknowns $\{h_{00}, h_{01}, \dots, h_{21}\}$ can be obtained by multiplying both sides of the equations in (6.19) through by the denominator, which yields the linear set of equations,

$$\begin{bmatrix} \hat{x}' - x \\ \hat{y}' - y \end{bmatrix} = \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -\hat{x}'x & -\hat{x}'y \\ 0 & 0 & 0 & x & y & 1 & -\hat{y}'x & -\hat{y}'y \end{bmatrix} \begin{bmatrix} h_{00} \\ \vdots \\ h_{21} \end{bmatrix}. \quad (6.21)$$

⁵ The “Hessian” \mathbf{A} is not the true Hessian (second derivative) of the non-linear least squares problem (6.13). Instead, it is the approximate Hessian, which neglects second (and higher) order derivatives of $\mathbf{f}(\mathbf{x}_i; \mathbf{p} + \Delta\mathbf{p})$.

However, this is not optimal from a statistical point of view, since the denominator D , which was used to multiply each equation, can vary quite a bit from point to point.⁶

One way to compensate for this is to *reweight* each equation by the inverse of the current estimate of the denominator, D ,

$$\frac{1}{D} \begin{bmatrix} \hat{x}' - x \\ \hat{y}' - y \end{bmatrix} = \frac{1}{D} \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -\hat{x}'x & -\hat{x}'y \\ 0 & 0 & 0 & x & y & 1 & -\hat{y}'x & -\hat{y}'y \end{bmatrix} \begin{bmatrix} h_{00} \\ \vdots \\ h_{21} \end{bmatrix}. \quad (6.22)$$

While this may at first seem to be the exact same set of equations as (6.21), because least squares is being used to solve the over-determined set of equations, the weightings *do* matter and produce a different set of normal equations that performs better in practice.

The most principled way to do the estimation, however, is to directly minimize the squared residual equations (6.13) using the Gauss–Newton approximation, i.e., performing a first-order Taylor series expansion in \mathbf{p} , as shown in (6.14), which yields the set of equations

$$\begin{bmatrix} \hat{x}' - \tilde{x}' \\ \hat{y}' - \tilde{y}' \end{bmatrix} = \frac{1}{D} \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -\tilde{x}'x & -\tilde{x}'y \\ 0 & 0 & 0 & x & y & 1 & -\tilde{y}'x & -\tilde{y}'y \end{bmatrix} \begin{bmatrix} \Delta h_{00} \\ \vdots \\ \Delta h_{21} \end{bmatrix}. \quad (6.23)$$

While these look similar to (6.22), they differ in two important respects. First, the left hand side consists of unweighted *prediction errors* rather than point displacements and the solution vector is a *perturbation* to the parameter vector \mathbf{p} . Second, the quantities inside \mathbf{J} involve *predicted* feature locations (\tilde{x}', \tilde{y}') instead of *sensed* feature locations (\hat{x}', \hat{y}') . Both of these differences are subtle and yet they lead to an algorithm that, when combined with proper checking for downhill steps (as in the Levenberg–Marquardt algorithm), will converge to a local minimum. Note that iterating Equations (6.22) is not guaranteed to converge, since it is not minimizing a well-defined energy function.

Equation (6.23) is analogous to the *additive* algorithm for direct intensity-based registration (Section 8.2), since the change to the full transformation is being computed. If we prepend an incremental homography to the current homography instead, i.e., we use a *compositional* algorithm (described in Section 8.2), we get $D = 1$ (since $\mathbf{p} = 0$) and the above formula simplifies to

$$\begin{bmatrix} \hat{x}' - x \\ \hat{y}' - y \end{bmatrix} = \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -x^2 & -xy \\ 0 & 0 & 0 & x & y & 1 & -xy & -y^2 \end{bmatrix} \begin{bmatrix} \Delta h_{00} \\ \vdots \\ \Delta h_{21} \end{bmatrix}, \quad (6.24)$$

where we have replaced (\tilde{x}', \tilde{y}') with (x, y) for conciseness. (Notice how this results in the same Jacobian as (8.63).)

⁶ Hartley and Zisserman (2004) call this strategy of forming linear equations from rational equations the *direct linear transform*, but that term is more commonly associated with pose estimation (Section 6.2). Note also that our definition of the h_{ij} parameters differs from that used in their book, since we define h_{ii} to be the *difference* from unity and we do not leave h_{22} as a free parameter, which means that we cannot handle certain extreme homographies.

6.1.4 Robust least squares and RANSAC

While regular least squares is the method of choice for measurements where the noise follows a normal (Gaussian) distribution, more robust versions of least squares are required when there are outliers among the correspondences (as there almost always are). In this case, it is preferable to use an *M-estimator* (Huber 1981; Hampel, Ronchetti, Rousseeuw *et al.* 1986; Black and Rangarajan 1996; Stewart 1999), which involves applying a robust penalty function $\rho(r)$ to the residuals

$$E_{\text{RLS}}(\Delta \mathbf{p}) = \sum_i \rho(\|\mathbf{r}_i\|) \quad (6.25)$$

instead of squaring them.

We can take the derivative of this function with respect to \mathbf{p} and set it to 0,

$$\sum_i \psi(\|\mathbf{r}_i\|) \frac{\partial \|\mathbf{r}_i\|}{\partial \mathbf{p}} = \sum_i \frac{\psi(\|\mathbf{r}_i\|)}{\|\mathbf{r}_i\|} \mathbf{r}_i^T \frac{\partial \mathbf{r}_i}{\partial \mathbf{p}} = 0, \quad (6.26)$$

where $\psi(r) = \rho'(r)$ is the derivative of ρ and is called the *influence function*. If we introduce a *weight function*, $w(r) = \Psi(r)/r$, we observe that finding the stationary point of (6.25) using (6.26) is equivalent to minimizing the *iteratively reweighted least squares* (IRLS) problem

$$E_{\text{IRLS}} = \sum_i w(\|\mathbf{r}_i\|) \|\mathbf{r}_i\|^2, \quad (6.27)$$

where the $w(\|\mathbf{r}_i\|)$ play the same local weighting role as σ_i^{-2} in (6.10). The IRLS algorithm alternates between computing the influence functions $w(\|\mathbf{r}_i\|)$ and solving the resulting weighted least squares problem (with fixed w values). Other incremental robust least squares algorithms can be found in the work of Sawhney and Ayer (1996); Black and Anandan (1996); Black and Rangarajan (1996); Baker, Gross, Ishikawa *et al.* (2003) and textbooks and tutorials on robust statistics (Huber 1981; Hampel, Ronchetti, Rousseeuw *et al.* 1986; Rousseeuw and Leroy 1987; Stewart 1999).

While M-estimators can definitely help reduce the influence of outliers, in some cases, starting with too many outliers will prevent IRLS (or other gradient descent algorithms) from converging to the global optimum. A better approach is often to find a starting set of *inlier* correspondences, i.e., points that are consistent with a dominant motion estimate.⁷

Two widely used approaches to this problem are called RANDom SAmple Consensus, or RANSAC for short (Fischler and Bolles 1981), and *least median of squares* (LMS) (Rousseeuw 1984). Both techniques start by selecting (at random) a subset of k correspondences, which is then used to compute an initial estimate for \mathbf{p} . The *residuals* of the full set of correspondences are then computed as

$$\mathbf{r}_i = \tilde{\mathbf{x}}'_i(\mathbf{x}_i; \mathbf{p}) - \hat{\mathbf{x}}'_i, \quad (6.28)$$

where $\tilde{\mathbf{x}}'_i$ are the *estimated* (mapped) locations and $\hat{\mathbf{x}}'_i$ are the sensed (detected) feature point locations.

The RANSAC technique then counts the number of *inliers* that are within ϵ of their predicted location, i.e., whose $\|\mathbf{r}_i\| \leq \epsilon$. (The ϵ value is application dependent but is often around 1–3 pixels.) Least median of squares finds the median value of the $\|\mathbf{r}_i\|^2$ values. The

⁷ For pixel-based alignment methods (Section 8.1.1), hierarchical (coarse-to-fine) techniques are often used to lock onto the *dominant motion* in a scene.

k	p	S
3	0.5	35
6	0.6	97
6	0.5	293

Table 6.2 Number of trials S to attain a 99% probability of success (Stewart 1999).

random selection process is repeated S times and the sample set with the largest number of inliers (or with the smallest median residual) is kept as the final solution. Either the initial parameter guess p or the full set of computed inliers is then passed on to the next data fitting stage.

When the number of measurements is quite large, it may be preferable to only score a subset of the measurements in an initial round that selects the most plausible hypotheses for additional scoring and selection. This modification of RANSAC, which can significantly speed up its performance, is called *Preemptive RANSAC* (Nistér 2003). In another variant on RANSAC called PROSAC (PROgressive SAmple Consensus), random samples are initially added from the most “confident” matches, thereby speeding up the process of finding a (statistically) likely good set of inliers (Chum and Matas 2005).

To ensure that the random sampling has a good chance of finding a true set of inliers, a sufficient number of trials S must be tried. Let p be the probability that any given correspondence is valid and P be the total probability of success after S trials. The likelihood in one trial that all k random samples are inliers is p^k . Therefore, the likelihood that S such trials will all fail is

$$1 - P = (1 - p^k)^S \quad (6.29)$$

and the required minimum number of trials is

$$S = \frac{\log(1 - P)}{\log(1 - p^k)}. \quad (6.30)$$

Stewart (1999) gives examples of the required number of trials S to attain a 99% probability of success. As you can see from Table 6.2, the number of trials grows quickly with the number of sample points used. This provides a strong incentive to use the *minimum* number of sample points k possible for any given trial, which is how RANSAC is normally used in practice.

Uncertainty modeling

In addition to robustly computing a good alignment, some applications require the computation of uncertainty (see Appendix B.6). For linear problems, this estimate can be obtained by inverting the Hessian matrix (6.9) and multiplying it by the feature position noise (if these have not already been used to weight the individual measurements, as in Equations (6.10) and 6.11)). In statistics, the Hessian, which is the inverse covariance, is sometimes called the (Fisher) *information matrix* (Appendix B.1.1).

When the problem involves non-linear least squares, the inverse of the Hessian matrix provides the *Cramer–Rao lower bound* on the covariance matrix, i.e., it provides the *minimum*

amount of covariance in a given solution, which can actually have a wider spread (“longer tails”) if the energy flattens out away from the local minimum where the optimal solution is found.

6.1.5 3D alignment

Instead of aligning 2D sets of image features, many computer vision applications require the alignment of 3D points. In the case where the 3D transformations are linear in the motion parameters, e.g., for translation, similarity, and affine, regular least squares (6.5) can be used.

The case of rigid (Euclidean) motion,

$$E_{\text{R3D}} = \sum_i \|x'_i - Rx_i - t\|^2, \quad (6.31)$$

which arises more frequently and is often called the *absolute orientation* problem (Horn 1987), requires slightly different techniques. If only scalar weightings are being used (as opposed to full 3D per-point anisotropic covariance estimates), the weighted centroids of the two point clouds c and c' can be used to estimate the translation $t = c' - Rc$.⁸ We are then left with the problem of estimating the rotation between two sets of points $\{\hat{x}_i = x_i - c\}$ and $\{\hat{x}'_i = x'_i - c'\}$ that are both centered at the origin.

One commonly used technique is called the *orthogonal Procrustes algorithm* (Golub and Van Loan 1996, p. 601) and involves computing the singular value decomposition (SVD) of the 3×3 correlation matrix

$$C = \sum_i \hat{x}' \hat{x}^T = U \Sigma V^T. \quad (6.32)$$

The rotation matrix is then obtained as $R = UV^T$. (Verify this for yourself when $\hat{x}' = R\hat{x}$.)

Another technique is the absolute orientation algorithm (Horn 1987) for estimating the unit quaternion corresponding to the rotation matrix R , which involves forming a 4×4 matrix from the entries in C and then finding the eigenvector associated with its largest positive eigenvalue.

Lorusso, Eggert, and Fisher (1995) experimentally compare these two techniques to two additional techniques proposed in the literature, but find that the difference in accuracy is negligible (well below the effects of measurement noise).

In situations where these closed-form algorithms are not applicable, e.g., when full 3D covariances are being used or when the 3D alignment is part of some larger optimization, the incremental rotation update introduced in Section 2.1.4 (2.35–2.36), which is parameterized by an instantaneous rotation vector ω , can be used (See Section 9.1.3 for an application to image stitching.)

In some situations, e.g., when merging range data maps, the correspondence between data points is not known *a priori*. In this case, iterative algorithms that start by matching nearby points and then update the most likely correspondence can be used (Besl and McKay 1992; Zhang 1994; Szeliski and Lavallée 1996; Gold, Rangarajan, Lu *et al.* 1998; David, DeMenthon, Duraiswami *et al.* 2004; Li and Hartley 2007; Enqvist, Josephson, and Kahl 2009). These techniques are discussed in more detail in Section 12.2.1.

⁸ When full covariances are used, they are transformed by the rotation and so a closed-form solution for translation is not possible.

6.2 Pose estimation

A particular instance of feature-based alignment, which occurs very often, is estimating an object’s 3D pose from a set of 2D point projections. This *pose estimation* problem is also known as *extrinsic* calibration, as opposed to the *intrinsic* calibration of internal camera parameters such as focal length, which we discuss in Section 6.3. The problem of recovering pose from three correspondences, which is the minimal amount of information necessary, is known as the *perspective-3-point-problem* (P3P), with extensions to larger numbers of points collectively known as PnP (Haralick, Lee, Ottenberg *et al.* 1994; Quan and Lan 1999; Moreno-Noguer, Lepetit, and Fua 2007).

In this section, we look at some of the techniques that have been developed to solve such problems, starting with the *direct linear transform* (DLT), which recovers a 3×4 camera matrix, followed by other “linear” algorithms, and then looking at statistically optimal iterative algorithms.

6.2.1 Linear algorithms

The simplest way to recover the pose of the camera is to form a set of linear equations analogous to those used for 2D motion estimation (6.19) from the camera matrix form of perspective projection (2.55–2.56),

$$x_i = \frac{p_{00}X_i + p_{01}Y_i + p_{02}Z_i + p_{03}}{p_{20}X_i + p_{21}Y_i + p_{22}Z_i + p_{23}} \quad (6.33)$$

$$y_i = \frac{p_{10}X_i + p_{11}Y_i + p_{12}Z_i + p_{13}}{p_{20}X_i + p_{21}Y_i + p_{22}Z_i + p_{23}}, \quad (6.34)$$

where (x_i, y_i) are the measured 2D feature locations and (X_i, Y_i, Z_i) are the known 3D feature locations (Figure 6.4). As with (6.21), this system of equations can be solved in a linear fashion for the unknowns in the camera matrix \mathbf{P} by multiplying the denominator on both sides of the equation.⁹ The resulting algorithm is called the *direct linear transform* (DLT) and is commonly attributed to Sutherland (1974). (For a more in-depth discussion, refer to the work of Hartley and Zisserman (2004).) In order to compute the 12 (or 11) unknowns in \mathbf{P} , at least six correspondences between 3D and 2D locations must be known.

As with the case of estimating homographies (6.21–6.23), more accurate results for the entries in \mathbf{P} can be obtained by directly minimizing the set of Equations (6.33–6.34) using non-linear least squares with a small number of iterations.

Once the entries in \mathbf{P} have been recovered, it is possible to recover both the intrinsic calibration matrix \mathbf{K} and the rigid transformation (\mathbf{R}, \mathbf{t}) by observing from Equation (2.56) that

$$\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}]. \quad (6.35)$$

Since \mathbf{K} is by convention upper-triangular (see the discussion in Section 2.1.5), both \mathbf{K} and \mathbf{R} can be obtained from the front 3×3 sub-matrix of \mathbf{P} using RQ factorization (Golub and Van Loan 1996).¹⁰

⁹ Because \mathbf{P} is unknown up to a scale, we can either fix one of the entries, e.g., $p_{23} = 1$, or find the smallest singular vector of the set of linear equations.

¹⁰ Note the unfortunate clash of terminologies: In matrix algebra textbooks, \mathbf{R} represents an upper-triangular matrix; in computer vision, \mathbf{R} is an orthogonal rotation.

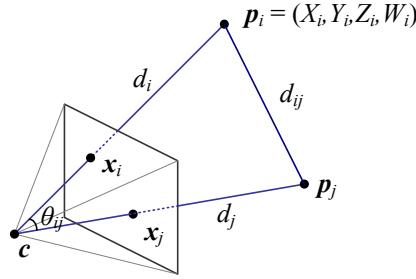


Figure 6.4 Pose estimation by the direct linear transform and by measuring visual angles and distances between pairs of points.

In most applications, however, we have some prior knowledge about the intrinsic calibration matrix \mathbf{K} , e.g., that the pixels are square, the skew is very small, and the optical center is near the center of the image (2.57–2.59). Such constraints can be incorporated into a non-linear minimization of the parameters in \mathbf{K} and (\mathbf{R}, \mathbf{t}) , as described in Section 6.2.2.

In the case where the camera is already calibrated, i.e., the matrix \mathbf{K} is known (Section 6.3), we can perform pose estimation using as few as three points (Fischler and Bolles 1981; Haralick, Lee, Ottenberg *et al.* 1994; Quan and Lan 1999). The basic observation that these *linear PnP* (*perspective n-point*) algorithms employ is that the visual angle between any pair of 2D points \hat{x}_i and \hat{x}_j must be the same as the angle between their corresponding 3D points p_i and p_j (Figure 6.4).

Given a set of corresponding 2D and 3D points $\{(\hat{x}_i, p_i)\}$, where the \hat{x}_i are unit directions obtained by transforming 2D pixel measurements x_i to unit norm 3D directions \hat{x}_i through the inverse calibration matrix \mathbf{K} ,

$$\hat{x}_i = \mathcal{N}(\mathbf{K}^{-1}x_i) = \mathbf{K}^{-1}x_i / \|\mathbf{K}^{-1}x_i\|, \quad (6.36)$$

the unknowns are the distances d_i from the camera origin c to the 3D points p_i , where

$$p_i = d_i \hat{x}_i + c \quad (6.37)$$

(Figure 6.4). The cosine law for triangle $\Delta(c, p_i, p_j)$ gives us

$$f_{ij}(d_i, d_j) = d_i^2 + d_j^2 - 2d_i d_j c_{ij} - d_{ij}^2 = 0, \quad (6.38)$$

where

$$c_{ij} = \cos \theta_{ij} = \hat{x}_i \cdot \hat{x}_j \quad (6.39)$$

and

$$d_{ij}^2 = \|p_i - p_j\|^2. \quad (6.40)$$

We can take any triplet of constraints (f_{ij}, f_{ik}, f_{jk}) and eliminate the d_j and d_k using Sylvester resultants (Cox, Little, and O’Shea 2007) to obtain a quartic equation in d_i^2 ,

$$g_{ijk}(d_i^2) = a_4 d_i^8 + a_3 d_i^6 + a_2 d_i^4 + a_1 d_i^2 + a_0 = 0. \quad (6.41)$$

Given five or more correspondences, we can generate $\frac{(n-1)(n-2)}{2}$ triplets to obtain a linear estimate (using SVD) for the values of $(d_i^8, d_i^6, d_i^4, d_i^2)$ (Quan and Lan 1999). Estimates for

d_i^2 can be computed as ratios of successive d_i^{2n+2}/d_i^{2n} estimates and these can be averaged to obtain a final estimate of d_i^2 (and hence d_i).

Once the individual estimates of the d_i distances have been computed, we can generate a 3D structure consisting of the scaled point directions $d_i\hat{x}_i$, which can then be aligned with the 3D point cloud $\{\mathbf{p}_i\}$ using absolute orientation (Section 6.1.5) to obtain the desired pose estimate. Quan and Lan (1999) give accuracy results for this and other techniques, which use fewer points but require more complicated algebraic manipulations. The paper by Moreno-Noguer, Lepetit, and Fua (2007) reviews more recent alternatives and also gives a lower complexity algorithm that typically produces more accurate results.

Unfortunately, because minimal PnP solutions can be quite noise sensitive and also suffer from *bas-relief ambiguities* (e.g., depth reversals) (Section 7.4.3), it is often preferable to use the linear six-point algorithm to guess an initial pose and then optimize this estimate using the iterative technique described in Section 6.2.2.

An alternative pose estimation algorithm involves starting with a scaled orthographic projection model and then iteratively refining this initial estimate using a more accurate perspective projection model (DeMenthon and Davis 1995). The attraction of this model, as stated in the paper's title, is that it can be implemented "in 25 lines of [Mathematica] code".

6.2.2 Iterative algorithms

The most accurate (and flexible) way to estimate pose is to directly minimize the squared (or robust) reprojection error for the 2D points as a function of the unknown pose parameters in (\mathbf{R}, \mathbf{t}) and optionally \mathbf{K} using non-linear least squares (Tsai 1987; Bogart 1991; Gleicher and Witkin 1992). We can write the projection equations as

$$\mathbf{x}_i = \mathbf{f}(\mathbf{p}_i; \mathbf{R}, \mathbf{t}, \mathbf{K}) \quad (6.42)$$

and iteratively minimize the robustified linearized reprojection errors

$$E_{\text{NLP}} = \sum_i \rho \left(\frac{\partial \mathbf{f}}{\partial \mathbf{R}} \Delta \mathbf{R} + \frac{\partial \mathbf{f}}{\partial \mathbf{t}} \Delta \mathbf{t} + \frac{\partial \mathbf{f}}{\partial \mathbf{K}} \Delta \mathbf{K} - \mathbf{r}_i \right), \quad (6.43)$$

where $\mathbf{r}_i = \tilde{\mathbf{x}}_i - \hat{\mathbf{x}}_i$ is the current residual vector (2D error in predicted position) and the partial derivatives are with respect to the unknown pose parameters (rotation, translation, and optionally calibration). Note that if full 2D covariance estimates are available for the 2D feature locations, the above squared norm can be weighted by the inverse point covariance matrix, as in Equation (6.11).

An easier to understand (and implement) version of the above non-linear regression problem can be constructed by re-writing the projection equations as a concatenation of simpler steps, each of which transforms a 4D homogeneous coordinate \mathbf{p}_i by a simple transformation such as translation, rotation, or perspective division (Figure 6.5). The resulting projection equations can be written as

$$\mathbf{y}^{(1)} = \mathbf{f}_{\text{T}}(\mathbf{p}_i; \mathbf{c}_j) = \mathbf{p}_i - \mathbf{c}_j, \quad (6.44)$$

$$\mathbf{y}^{(2)} = \mathbf{f}_{\text{R}}(\mathbf{y}^{(1)}; \mathbf{q}_j) = \mathbf{R}(\mathbf{q}_j) \mathbf{y}^{(1)}, \quad (6.45)$$

$$\mathbf{y}^{(3)} = \mathbf{f}_{\text{P}}(\mathbf{y}^{(2)}) = \frac{\mathbf{y}^{(2)}}{z^{(2)}}, \quad (6.46)$$

$$\mathbf{x}_i = \mathbf{f}_{\text{C}}(\mathbf{y}^{(3)}; \mathbf{k}) = \mathbf{K}(\mathbf{k}) \mathbf{y}^{(3)}. \quad (6.47)$$

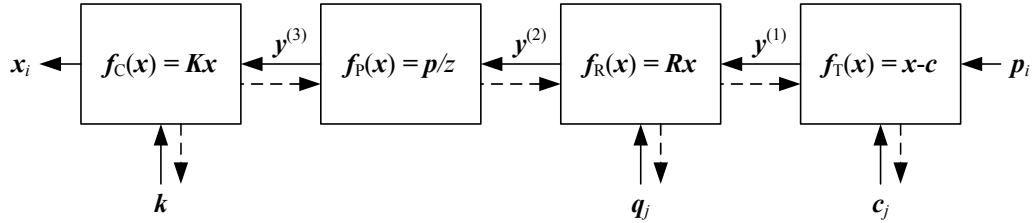


Figure 6.5 A set of chained transforms for projecting a 3D point p_i to a 2D measurement x_i through a series of transformations $f^{(k)}$, each of which is controlled by its own set of parameters. The dashed lines indicate the flow of information as partial derivatives are computed during a backward pass.

Note that in these equations, we have indexed the camera centers c_j and camera rotation quaternions q_j by an index j , in case more than one pose of the calibration object is being used (see also Section 7.4.) We are also using the camera center c_j instead of the world translation t_j , since this is a more natural parameter to estimate.

The advantage of this chained set of transformations is that each one has a simple partial derivative with respect both to its parameters and to its input. Thus, once the predicted value of \tilde{x}_i has been computed based on the 3D point location p_i and the current values of the pose parameters (c_j, q_j, k) , we can obtain all of the required partial derivatives using the chain rule

$$\frac{\partial r_i}{\partial p^{(k)}} = \frac{\partial r_i}{\partial y^{(k)}} \frac{\partial y^{(k)}}{\partial p^{(k)}}, \quad (6.48)$$

where $p^{(k)}$ indicates one of the parameter vectors that is being optimized. (This same “trick” is used in neural networks as part of the *backpropagation* algorithm (Bishop 2006).)

The one special case in this formulation that can be considerably simplified is the computation of the rotation update. Instead of directly computing the derivatives of the 3×3 rotation matrix $R(q)$ as a function of the unit quaternion entries, you can prepend the incremental rotation matrix $\Delta R(\omega)$ given in Equation (2.35) to the current rotation matrix and compute the partial derivative of the transform with respect to these parameters, which results in a simple cross product of the backward chaining partial derivative and the outgoing 3D vector (2.36).

6.2.3 Application: Augmented reality

A widely used application of pose estimation is *augmented reality*, where virtual 3D images or annotations are superimposed on top of a live video feed, either through the use of see-through glasses (a head-mounted display) or on a regular computer or mobile device screen (Azuma, Baillot, Behringer *et al.* 2001; Haller, Billinghurst, and Thomas 2007). In some applications, a special pattern printed on cards or in a book is tracked to perform the augmentation (Kato, Billinghurst, Poupyrev *et al.* 2000; Billinghurst, Kato, and Poupyrev 2001). For a desktop application, a grid of dots printed on a mouse pad can be tracked by a camera embedded in an augmented mouse to give the user control of a full six degrees of freedom over their position and orientation in a 3D space (Hinckley, Sinclair, Hanson *et al.* 1999), as shown in Figure 6.6.

Sometimes, the scene itself provides a convenient object to track, such as the rectangle defining a desktop used in *through-the-lens camera control* (Gleicher and Witkin 1992). In

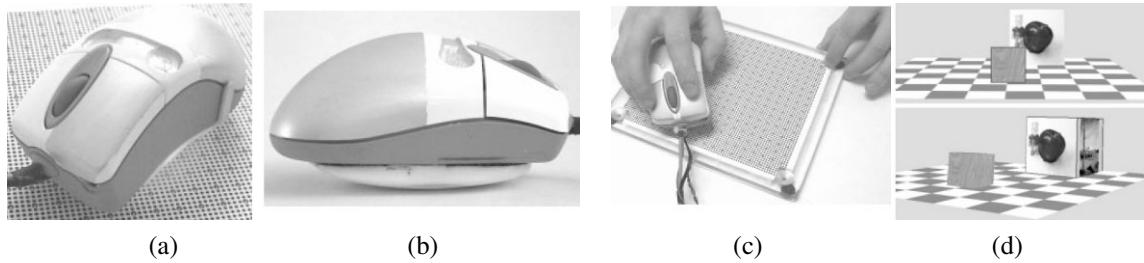


Figure 6.6 The VideoMouse can sense six degrees of freedom relative to a specially printed mouse pad using its embedded camera (Hinckley, Sinclair, Hanson *et al.* 1999) © 1999 ACM: (a) top view of the mouse; (b) view of the mouse showing the curved base for rocking; (c) moving the mouse pad with the other hand extends the interaction capabilities; (d) the resulting movement seen on the screen.

outdoor locations, such as film sets, it is more common to place special markers such as brightly colored balls in the scene to make it easier to find and track them (Bogart 1991). In older applications, surveying techniques were used to determine the locations of these balls before filming. Today, it is more common to apply structure-from-motion directly to the film footage itself (Section 7.4.2).

Rapid pose estimation is also central to tracking the position and orientation of the hand-held remote controls used in Nintendo’s Wii game systems. A high-speed camera embedded in the remote control is used to track the locations of the infrared (IR) LEDs in the bar that is mounted on the TV monitor. Pose estimation is then used to infer the remote control’s location and orientation at very high frame rates. The Wii system can be extended to a variety of other user interaction applications by mounting the bar on a hand-held device, as described by Johnny Lee.¹¹

Exercises 6.4 and 6.5 have you implement two different tracking and pose estimation systems for augmented-reality applications. The first system tracks the outline of a rectangular object, such as a book cover or magazine page, and the second has you track the pose of a hand-held Rubik’s cube.

6.3 Geometric intrinsic calibration

As described above in Equations (6.42–6.43), the computation of the internal (intrinsic) camera calibration parameters can occur simultaneously with the estimation of the (extrinsic) pose of the camera with respect to a known calibration target. This, indeed, is the “classic” approach to camera calibration used in both the photogrammetry (Slama 1980) and the computer vision (Tsai 1987) communities. In this section, we look at alternative formulations (which may not involve the full solution of a non-linear regression problem), the use of alternative calibration targets, and the estimation of the non-linear part of camera optics such as radial distortion.¹²

¹¹ <http://johnnylee.net/projects/wii/>.

¹² In some applications, you can use the EXIF tags associated with a JPEG image to obtain a rough estimate of a camera’s focal length but this technique should be used with caution as the results are often inaccurate.

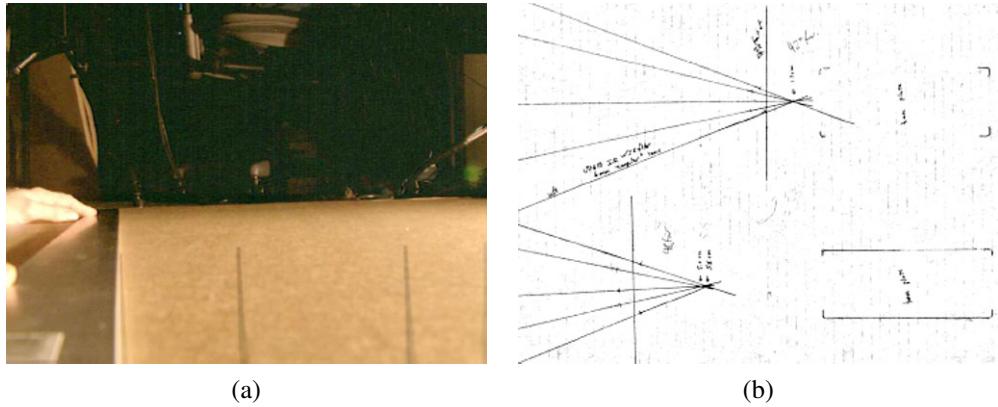


Figure 6.7 Calibrating a lens by drawing straight lines on cardboard (Debevec, Wenger, Tchou *et al.* 2002) © 2002 ACM: (a) an image taken by the video camera showing a hand holding a metal ruler whose right edge appears vertical in the image; (b) the set of lines drawn on the cardboard converging on the front nodal point (center of projection) of the lens and indicating the horizontal field of view.

6.3.1 Calibration patterns

The use of a calibration pattern or set of markers is one of the more reliable ways to estimate a camera’s intrinsic parameters. In photogrammetry, it is common to set up a camera in a large field looking at distant calibration targets whose exact location has been precomputed using surveying equipment (Slama 1980; Atkinson 1996; Kraus 1997). In this case, the translational component of the pose becomes irrelevant and only the camera rotation and intrinsic parameters need to be recovered.

If a smaller calibration rig needs to be used, e.g., for indoor robotics applications or for mobile robots that carry their own calibration target, it is best if the calibration object can span as much of the workspace as possible (Figure 6.8a), as planar targets often fail to accurately predict the components of the pose that lie far away from the plane. A good way to determine if the calibration has been successfully performed is to estimate the covariance in the parameters (Section 6.1.4) and then project 3D points from various points in the workspace into the image in order to estimate their 2D positional uncertainty.

An alternative method for estimating the focal length and center of projection of a lens is to place the camera on a large flat piece of cardboard and use a long metal ruler to draw lines on the cardboard that appear vertical in the image, as shown in Figure 6.7a (Debevec, Wenger, Tchou *et al.* 2002). Such lines lie on planes that are parallel to the vertical axis of the camera sensor and also pass through the lens’ front nodal point. The location of the nodal point (projected vertically onto the cardboard plane) and the horizontal field of view (determined from lines that graze the left and right edges of the visible image) can be recovered by intersecting these lines and measuring their angular extent (Figure 6.7b).

If no calibration pattern is available, it is also possible to perform calibration simultaneously with structure and pose recovery (Sections 6.3.4 and 7.4), which is known as *self-calibration* (Faugeras, Luong, and Maybank 1992; Hartley and Zisserman 2004; Moons, Van Gool, and Vergauwen 2010). However, such an approach requires a large amount of imagery to be accurate.

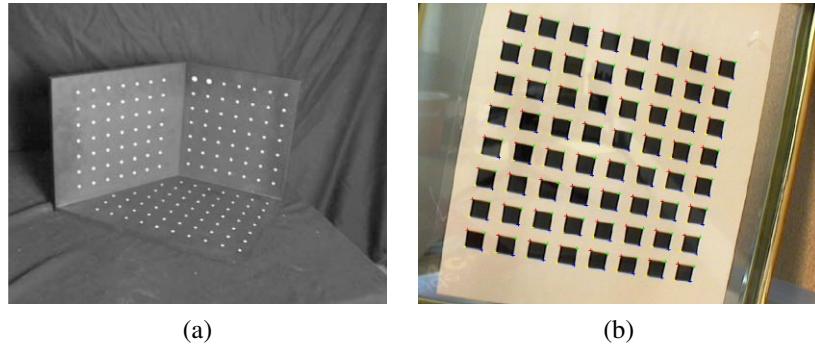


Figure 6.8 Calibration patterns: (a) a three-dimensional target (Quan and Lan 1999) © 1999 IEEE; (b) a two-dimensional target (Zhang 2000) © 2000 IEEE. Note that radial distortion needs to be removed from such images before the feature points can be used for calibration.

Planar calibration patterns

When a finite workspace is being used and accurate machining and motion control platforms are available, a good way to perform calibration is to move a planar calibration target in a controlled fashion through the workspace volume. This approach is sometimes called the *N-planes* calibration approach (Gremban, Thorpe, and Kanade 1988; Champleboux, Lavallée, Szeliski *et al.* 1992; Grossberg and Nayar 2001) and has the advantage that each camera pixel can be mapped to a unique 3D ray in space, which takes care of both linear effects modeled by the calibration matrix \mathbf{K} and non-linear effects such as radial distortion (Section 6.3.5).

A less cumbersome but also less accurate calibration can be obtained by waving a planar calibration pattern in front of a camera (Figure 6.8b). In this case, the pattern's pose has (in principle) to be recovered in conjunction with the intrinsics. In this technique, each input image is used to compute a separate homography (6.19–6.23) $\tilde{\mathbf{H}}$ mapping the plane's calibration points $(X_i, Y_i, 0)$ into image coordinates (x_i, y_i) ,

$$\mathbf{x}_i = \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \sim \mathbf{K} \begin{bmatrix} \mathbf{r}_0 & \mathbf{r}_1 & \mathbf{t} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ 1 \end{bmatrix} \sim \tilde{\mathbf{H}} \mathbf{p}_i, \quad (6.49)$$

where the \mathbf{r}_i are the first two columns of \mathbf{R} and \sim indicates equality up to scale. From these, Zhang (2000) shows how to form linear constraints on the nine entries in the $\mathbf{B} = \mathbf{K}^{-T} \mathbf{K}^{-1}$ matrix, from which the calibration matrix \mathbf{K} can be recovered using a matrix square root and inversion. (The matrix \mathbf{B} is known as the *image of the absolute conic* (IAC) in projective geometry and is commonly used for camera calibration (Hartley and Zisserman 2004, Section 7.5).) If only the focal length is being recovered, the even simpler approach of using vanishing points can be used instead.

6.3.2 Vanishing points

A common case for calibration that occurs often in practice is when the camera is looking at a man-made scene with strong extended rectahedral objects such as boxes or room walls. In this case, we can intersect the 2D lines corresponding to 3D parallel lines to compute their

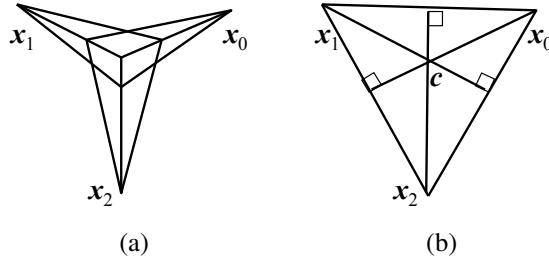


Figure 6.9 Calibration from vanishing points: (a) any pair of finite vanishing points (\hat{x}_i, \hat{x}_j) can be used to estimate the focal length; (b) the orthocenter of the vanishing point triangle gives the optical center of the image c .

vanishing points, as described in Section 4.3.3, and use these to determine the intrinsic and extrinsic calibration parameters (Caprile and Torre 1990; Becker and Bove 1995; Liebowitz and Zisserman 1998; Cipolla, Drummond, and Robertson 1999; Antone and Teller 2002; Criminisi, Reid, and Zisserman 2000; Hartley and Zisserman 2004; Pflugfelder 2008).

Let us assume that we have detected two or more orthogonal vanishing points, all of which are *finite*, i.e., they are not obtained from lines that appear to be parallel in the image plane (Figure 6.9a). Let us also assume a simplified form for the calibration matrix \mathbf{K} where only the focal length is unknown (2.59). (It is often safe for rough 3D modeling to assume that the optical center is at the center of the image, that the aspect ratio is 1, and that there is no skew.) In this case, the projection equation for the vanishing points can be written as

$$\hat{\mathbf{x}}_i = \begin{bmatrix} x_i - c_x \\ y_i - c_y \\ f \end{bmatrix} \sim \mathbf{R}\mathbf{p}_i = \mathbf{r}_i, \quad (6.50)$$

where \mathbf{p}_i corresponds to one of the cardinal directions $(1, 0, 0)$, $(0, 1, 0)$, or $(0, 0, 1)$, and \mathbf{r}_i is the i th column of the rotation matrix \mathbf{R} .

From the orthogonality between columns of the rotation matrix, we have

$$\mathbf{r}_i \cdot \mathbf{r}_j \sim (x_i - c_x)(x_j - c_x) + (y_i - c_y)(y_j - c_y) + f^2 = 0 \quad (6.51)$$

from which we can obtain an estimate for f^2 . Note that the accuracy of this estimate increases as the vanishing points move closer to the center of the image. In other words, it is best to tilt the calibration pattern a decent amount around the 45° axis, as in Figure 6.9a. Once the focal length f has been determined, the individual columns of \mathbf{R} can be estimated by normalizing the left hand side of (6.50) and taking cross products. Alternatively, an SVD of the initial \mathbf{R} estimate, which is a variant on orthogonal Procrustes (6.32), can be used.

If all three vanishing points are visible and finite in the same image, it is also possible to estimate the optical center as the orthocenter of the triangle formed by the three vanishing points (Caprile and Torre 1990; Hartley and Zisserman 2004, Section 7.6) (Figure 6.9b). In practice, however, it is more accurate to re-estimate any unknown intrinsic calibration parameters using non-linear least squares (6.42).

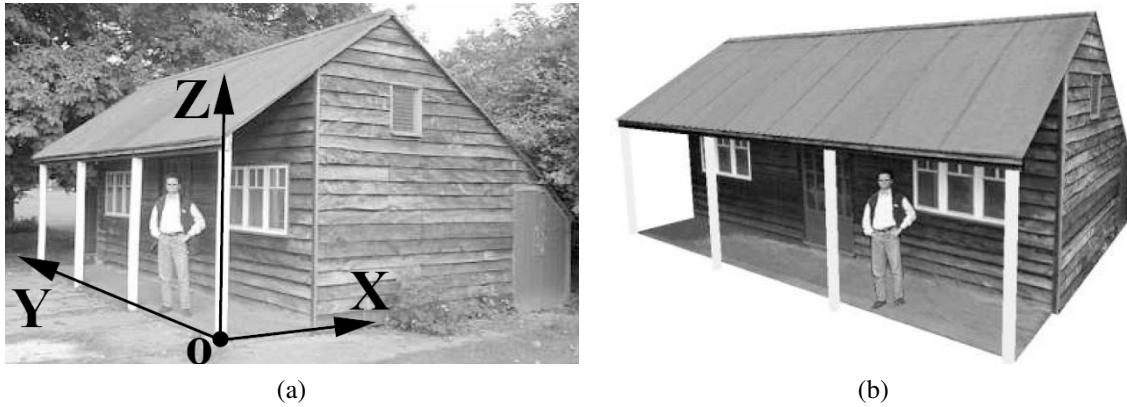


Figure 6.10 Single view metrology (Criminisi, Reid, and Zisserman 2000) © 2000 Springer: (a) input image showing the three coordinate axes computed from the two horizontal vanishing points (which can be determined from the sidings on the shed); (b) a new view of the 3D reconstruction.

6.3.3 Application: Single view metrology

A fun application of vanishing point estimation and camera calibration is the *single view metrology* system developed by Criminisi, Reid, and Zisserman (2000). Their system allows people to interactively measure heights and other dimensions as well as to build piecewise-planar 3D models, as shown in Figure 6.10.

The first step in their system is to identify two orthogonal vanishing points on the ground plane and the vanishing point for the vertical direction, which can be done by drawing some parallel sets of lines in the image. (Alternatively, automated techniques such as those discussed in Section 4.3.3 or by Schaffalitzky and Zisserman (2000) could be used.) The user then marks a few dimensions in the image, such as the height of a reference object, and the system can automatically compute the height of another object. Walls and other planar impostors (geometry) can also be sketched and reconstructed.

In the formulation originally developed by Criminisi, Reid, and Zisserman (2000), the system produces an *affine* reconstruction, i.e., one that is only known up to a set of independent scaling factors along each axis. A potentially more useful system can be constructed by assuming that the camera is calibrated up to an unknown focal length, which can be recovered from orthogonal (finite) vanishing directions, as we just described in Section 6.3.2. Once this is done, the user can indicate an origin on the ground plane and another point a known distance away. From this, points on the ground plane can be directly projected into 3D and points above the ground plane, when paired with their ground plane projections, can also be recovered. A fully metric reconstruction of the scene then becomes possible.

Exercise 6.9 has you implement such a system and then use it to model some simple 3D scenes. Section 12.6.1 describes other, potentially multi-view, approaches to architectural reconstruction, including an interactive piecewise-planar modeling system that uses vanishing points to establish 3D line directions and plane normals (Sinha, Steedly, Szeliski *et al.* 2008).

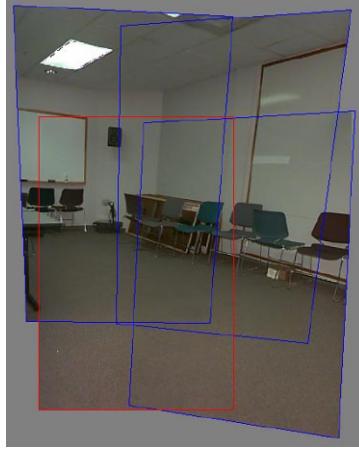


Figure 6.11 Four images taken with a hand-held camera registered using a 3D rotation motion model, which can be used to estimate the focal length of the camera (Szeliski and Shum 1997) © 2000 ACM.

6.3.4 Rotational motion

When no calibration targets or known structures are available but you can rotate the camera around its front nodal point (or, equivalently, work in a large open environment where all objects are distant), the camera can be calibrated from a set of overlapping images by assuming that it is undergoing pure rotational motion, as shown in Figure 6.11 (Stein 1995; Hartley 1997b; Hartley, Hayman, de Agapito *et al.* 2000; de Agapito, Hayman, and Reid 2001; Kang and Weiss 1999; Shum and Szeliski 2000; Frahm and Koch 2003). When a full 360° motion is used to perform this calibration, a very accurate estimate of the focal length f can be obtained, as the accuracy in this estimate is proportional to the total number of pixels in the resulting cylindrical panorama (Section 9.1.6) (Stein 1995; Shum and Szeliski 2000).

To use this technique, we first compute the homographies $\tilde{\mathbf{H}}_{ij}$ between all overlapping pairs of images, as explained in Equations (6.19–6.23). Then, we use the observation, first made in Equation (2.72) and explored in more detail in Section 9.1.3 (9.5), that each homography is related to the inter-camera rotation \mathbf{R}_{ij} through the (unknown) calibration matrices \mathbf{K}_i and \mathbf{K}_j ,

$$\tilde{\mathbf{H}}_{ij} = \mathbf{K}_i \mathbf{R}_{ij} \mathbf{R}_{ji}^{-1} \mathbf{K}_j^{-1} = \mathbf{K}_i \mathbf{R}_{ij} \mathbf{K}_j^{-1}. \quad (6.52)$$

The simplest way to obtain the calibration is to use the simplified form of the calibration matrix (2.59), where we assume that the pixels are square and the optical center lies at the center of the image, i.e., $\mathbf{K}_k = \text{diag}(f_k, f_k, 1)$. (We number the pixel coordinates accordingly, i.e., place pixel $(x, y) = (0, 0)$ at the center of the image.) We can then rewrite Equation (6.52) as

$$\mathbf{R}_{10} \sim \mathbf{K}_1^{-1} \tilde{\mathbf{H}}_{10} \mathbf{K}_0 \sim \begin{bmatrix} h_{00} & h_{01} & f_0^{-1} h_{02} \\ h_{10} & h_{11} & f_0^{-1} h_{12} \\ f_1 h_{20} & f_1 h_{21} & f_0^{-1} f_1 h_{22} \end{bmatrix}, \quad (6.53)$$

where h_{ij} are the elements of $\tilde{\mathbf{H}}_{10}$.

Using the orthonormality properties of the rotation matrix \mathbf{R}_{10} and the fact that the right hand side of (6.53) is known only up to a scale, we obtain

$$h_{00}^2 + h_{01}^2 + f_0^{-2}h_{02}^2 = h_{10}^2 + h_{11}^2 + f_0^{-2}h_{12}^2 \quad (6.54)$$

and

$$h_{00}h_{10} + h_{01}h_{11} + f_0^{-2}h_{02}h_{12} = 0. \quad (6.55)$$

From this, we can compute estimates for f_0 of

$$f_0^2 = \frac{h_{12}^2 - h_{02}^2}{h_{00}^2 + h_{01}^2 - h_{10}^2 - h_{11}^2} \text{ if } h_{00}^2 + h_{01}^2 \neq h_{10}^2 + h_{11}^2 \quad (6.56)$$

or

$$f_0^2 = -\frac{h_{02}h_{12}}{h_{00}h_{10} + h_{01}h_{11}} \text{ if } h_{00}h_{10} \neq -h_{01}h_{11}. \quad (6.57)$$

(Note that the equations originally given by Szeliski and Shum (1997) are erroneous; the correct equations are given by Shum and Szeliski (2000).) If neither of these conditions holds, we can also take the dot products between the first (or second) row and the third one. Similar results can be obtained for f_1 as well, by analyzing the columns of $\tilde{\mathbf{H}}_{10}$. If the focal length is the same for both images, we can take the geometric mean of f_0 and f_1 as the estimated focal length $f = \sqrt{f_1f_0}$. When multiple estimates of f are available, e.g., from different homographies, the median value can be used as the final estimate.

A more general (upper-triangular) estimate of \mathbf{K} can be obtained in the case of a fixed-parameter camera $\mathbf{K}_i = \mathbf{K}$ using the technique of Hartley (1997b). Observe from (6.52) that $\mathbf{R}_{ij} \sim \mathbf{K}^{-1}\tilde{\mathbf{H}}_{ij}\mathbf{K}$ and $\mathbf{R}_{ij}^{-T} \sim \mathbf{K}^T\tilde{\mathbf{H}}_{ij}^{-T}\mathbf{K}^{-T}$. Equating $\mathbf{R}_{ij} = \mathbf{R}_{ij}^{-T}$ we obtain $\mathbf{K}^{-1}\tilde{\mathbf{H}}_{ij}\mathbf{K} \sim \mathbf{K}^T\tilde{\mathbf{H}}_{ij}^{-T}\mathbf{K}^{-T}$, from which we get

$$\tilde{\mathbf{H}}_{ij}(\mathbf{K}\mathbf{K}^T) \sim (\mathbf{K}\mathbf{K}^T)\tilde{\mathbf{H}}_{ij}^{-T}. \quad (6.58)$$

This provides us with some homogeneous linear constraints on the entries in $\mathbf{A} = \mathbf{K}\mathbf{K}^T$, which is known as the *dual of the image of the absolute conic* (Hartley 1997b; Hartley and Zisserman 2004). (Recall that when we estimate a homography, we can only recover it up to an unknown scale.) Given a sufficient number of independent homography estimates $\tilde{\mathbf{H}}_{ij}$, we can recover \mathbf{A} (up to a scale) using either SVD or eigenvalue analysis and then recover \mathbf{K} through Cholesky decomposition (Appendix A.1.4). Extensions to the cases of temporally varying calibration parameters and non-stationary cameras are discussed by Hartley, Hayman, de Agapito *et al.* (2000) and de Agapito, Hayman, and Reid (2001).

The quality of the intrinsic camera parameters can be greatly increased by constructing a full 360° panorama, since mis-estimating the focal length will result in a gap (or excessive overlap) when the first image in the sequence is stitched to itself (Figure 9.5). The resulting mis-alignment can be used to improve the estimate of the focal length and to re-adjust the rotation estimates, as described in Section 9.1.4. Rotating the camera by 90° around its optic axis and re-shooting the panorama is a good way to check for aspect ratio and skew pixel problems, as is generating a full hemi-spherical panorama when there is sufficient texture.

Ultimately, however, the most accurate estimate of the calibration parameters (including radial distortion) can be obtained using a full simultaneous non-linear minimization of the intrinsic and extrinsic (rotation) parameters, as described in Section 9.2.

6.3.5 Radial distortion

When images are taken with wide-angle lenses, it is often necessary to model *lens distortions* such as *radial distortion*. As discussed in Section 2.1.6, the radial distortion model says that coordinates in the observed images are displaced away from (*barrel* distortion) or towards (*pincushion* distortion) the image center by an amount proportional to their radial distance (Figure 2.13a–b). The simplest radial distortion models use low-order polynomials (c.f. Equation (2.78)),

$$\begin{aligned}\hat{x} &= x(1 + \kappa_1 r^2 + \kappa_2 r^4) \\ \hat{y} &= y(1 + \kappa_1 r^2 + \kappa_2 r^4),\end{aligned}\quad (6.59)$$

where $r^2 = x^2 + y^2$ and κ_1 and κ_2 are called the *radial distortion parameters* (Brown 1971; Slama 1980).¹³

A variety of techniques can be used to estimate the radial distortion parameters for a given lens.¹⁴ One of the simplest and most useful is to take an image of a scene with a lot of straight lines, especially lines aligned with and near the edges of the image. The radial distortion parameters can then be adjusted until all of the lines in the image are straight, which is commonly called the *plumb-line method* (Brown 1971; Kang 2001; El-Melegy and Farag 2003). Exercise 6.10 gives some more details on how to implement such a technique.

Another approach is to use several overlapping images and to combine the estimation of the radial distortion parameters with the image alignment process, i.e., by extending the pipeline used for stitching in Section 9.2.1. Sawhney and Kumar (1999) use a hierarchy of motion models (translation, affine, projective) in a coarse-to-fine strategy coupled with a quadratic radial distortion correction term. They use direct (intensity-based) minimization to compute the alignment. Stein (1997) uses a feature-based approach combined with a general 3D motion model (and quadratic radial distortion), which requires more matches than a parallax-free rotational panorama but is potentially more general. More recent approaches sometimes simultaneously compute both the unknown intrinsic parameters and the radial distortion coefficients, which may include higher-order terms or more complex rational or non-parametric forms (Claus and Fitzgibbon 2005; Sturm 2005; Thirthala and Pollefeys 2005; Barreto and Daniilidis 2005; Hartley and Kang 2005; Steele and Jaynes 2006; Tardif, Sturm, Trudeau *et al.* 2009).

When a known calibration target is being used (Figure 6.8), the radial distortion estimation can be folded into the estimation of the other intrinsic and extrinsic parameters (Zhang 2000; Hartley and Kang 2007; Tardif, Sturm, Trudeau *et al.* 2009). This can be viewed as adding another stage to the general non-linear minimization pipeline shown in Figure 6.5 between the intrinsic parameter multiplication box f_C and the perspective division box f_P . (See Exercise 6.11 on more details for the case of a planar calibration target.)

Of course, as discussed in Section 2.1.6, more general models of lens distortion, such as fisheye and non-central projection, may sometimes be required. While the parameterization of such lenses may be more complicated (Section 2.1.6), the general approach of either using calibration rigs with known 3D positions or self-calibration through the use of multiple

¹³ Sometimes the relationship between x and \hat{x} is expressed the other way around, i.e., using primed (final) coordinates on the right-hand side, $x = \hat{x}(1 + \kappa_1 \hat{r}^2 + \kappa_2 \hat{r}^4)$. This is convenient if we map image pixels into (warped) rays and then undistort the rays to obtain 3D rays in space, i.e., if we are using inverse warping.

¹⁴ Some of today's digital cameras are starting to remove radial distortion using software in the camera itself.

overlapping images of a scene can both be used (Hartley and Kang 2007; Tardif, Sturm, and Roy 2007). The same techniques used to calibrate for radial distortion can also be used to reduce the amount of chromatic aberration by separately calibrating each color channel and then warping the channels to put them back into alignment (Exercise 6.12).

6.4 Additional reading

Hartley and Zisserman (2004) provide a wonderful introduction to the topics of feature-based alignment and optimal motion estimation, as well as an in-depth discussion of camera calibration and pose estimation techniques.

Techniques for robust estimation are discussed in more detail in Appendix B.3 and in monographs and review articles on this topic (Huber 1981; Hampel, Ronchetti, Rousseeuw *et al.* 1986; Rousseeuw and Leroy 1987; Black and Rangarajan 1996; Stewart 1999). The most commonly used robust initialization technique in computer vision is RANdom SAmple Consensus (RANSAC) (Fischler and Bolles 1981), which has spawned a series of more efficient variants (Nistér 2003; Chum and Matas 2005).

The topic of registering 3D point data sets is called *absolute orientation* (Horn 1987) and *3D pose estimation* (Lorusso, Eggert, and Fisher 1995). A variety of techniques has been developed for simultaneously computing 3D point correspondences and their corresponding rigid transformations (Besl and McKay 1992; Zhang 1994; Szeliski and Lavallée 1996; Gold, Rangarajan, Lu *et al.* 1998; David, DeMenthon, Duraiswami *et al.* 2004; Li and Hartley 2007; Enqvist, Josephson, and Kahl 2009).

Camera calibration was first studied in photogrammetry (Brown 1971; Slama 1980; Atkinson 1996; Kraus 1997) but it has also been widely studied in computer vision (Tsai 1987; Gremban, Thorpe, and Kanade 1988; Chappleboux, Lavallée, Szeliski *et al.* 1992; Zhang 2000; Grossberg and Nayar 2001). Vanishing points observed either from rectahedral calibration objects or man-made architecture are often used to perform rudimentary calibration (Caprile and Torre 1990; Becker and Bove 1995; Liebowitz and Zisserman 1998; Cipolla, Drummond, and Robertson 1999; Antone and Teller 2002; Criminisi, Reid, and Zisserman 2000; Hartley and Zisserman 2004; Pflugfelder 2008). Performing camera calibration without using known targets is known as *self-calibration* and is discussed in textbooks and surveys on structure from motion (Faugeras, Luong, and Maybank 1992; Hartley and Zisserman 2004; Moons, Van Gool, and Vergauwen 2010). One popular subset of such techniques uses pure rotational motion (Stein 1995; Hartley 1997b; Hartley, Hayman, de Agapito *et al.* 2000; de Agapito, Hayman, and Reid 2001; Kang and Weiss 1999; Shum and Szeliski 2000; Frahm and Koch 2003).

6.5 Exercises

Ex 6.1: Feature-based image alignment for flip-book animations Take a set of photos of an action scene or portrait (preferably in motor-drive—continuous shooting—mode) and align them to make a composite or flip-book animation.

1. Extract features and feature descriptors using some of the techniques described in Sections 4.1.1–4.1.2.

2. Match your features using nearest neighbor matching with a nearest neighbor distance ratio test (4.18).
3. Compute an optimal 2D translation and rotation between the first image and all subsequent images, using least squares (Section 6.1.1) with optional RANSAC for robustness (Section 6.1.4).
4. Resample all of the images onto the first image's coordinate frame (Section 3.6.1) using either bilinear or bicubic resampling and optionally crop them to their common area.
5. Convert the resulting images into an animated GIF (using software available from the Web) or optionally implement cross-dissolves to turn them into a "slo-mo" video.
6. (Optional) Combine this technique with feature-based (Exercise 3.25) morphing.

Ex 6.2: Panography Create the kind of panograph discussed in Section 6.1.2 and commonly found on the Web.

1. Take a series of interesting overlapping photos.
2. Use the feature detector, descriptor, and matcher developed in Exercises 4.1–4.4 (or existing software) to match features among the images.
3. Turn each connected component of matching features into a *track*, i.e., assign a unique index i to each track, discarding any tracks that are inconsistent (contain two different features in the same image).
4. Compute a global translation for each image using Equation (6.12).
5. Since your matches probably contain errors, turn the above least square metric into a robust metric (6.25) and re-solve your system using iteratively reweighted least squares.
6. Compute the size of the resulting composite canvas and resample each image into its final position on the canvas. (Keeping track of bounding boxes will make this more efficient.)
7. Average all of the images, or choose some kind of ordering and implement translucent *over* compositing (3.8).
8. (Optional) Extend your parametric motion model to include rotations and scale, i.e., the similarity transform given in Table 6.1. Discuss how you could handle the case of translations and rotations only (no scale).
9. (Optional) Write a simple tool to let the user adjust the ordering and opacity, and add or remove images.
10. (Optional) Write down a different least squares problem that involves pairwise matching of images. Discuss why this might be better or worse than the global matching formula given in (6.12).

Ex 6.3: 2D rigid/Euclidean matching Several alternative approaches are given in Section 6.1.3 for estimating a 2D rigid (Euclidean) alignment.

1. Implement the various alternatives and compare their accuracy on synthetic data, i.e., random 2D point clouds with noisy feature positions.
2. One approach is to estimate the translations from the centroids and then estimate rotation in polar coordinates. Do you need to weight the angles obtained from a polar decomposition in some way to get the statistically correct estimate?
3. How can you modify your techniques to take into account either scalar (6.10) or full two-dimensional point covariance weightings (6.11)? Do all of the previously developed “shortcuts” still work or does full weighting require iterative optimization?

Ex 6.4: 2D match move/augmented reality Replace a picture in a magazine or a book with a different image or video.

1. With a webcam, take a picture of a magazine or book page.
2. Outline a figure or picture on the page with a rectangle, i.e., draw over the four sides as they appear in the image.
3. Match features in this area with each new image frame.
4. Replace the original image with an “advertising” insert, warping the new image with the appropriate homography.
5. Try your approach on a clip from a sporting event (e.g., indoor or outdoor soccer) to implement a billboard replacement.

Ex 6.5: 3D joystick Track a Rubik’s cube to implement a 3D joystick/mouse control.

1. Get out an old Rubik’s cube (or get one from your parents).
2. Write a program to detect the center of each colored square.
3. Group these centers into lines and then find the vanishing points for each face.
4. Estimate the rotation angle and focal length from the vanishing points.
5. Estimate the full 3D pose (including translation) by finding one or more 3×3 grids and recovering the plane’s full equation from this known homography using the technique developed by Zhang (2000).
6. Alternatively, since you already know the rotation, simply estimate the unknown translation from the known 3D corner points on the cube and their measured 2D locations using either linear or non-linear least squares.
7. Use the 3D rotation and position to control a VRML or 3D game viewer.

Ex 6.6: Rotation-based calibration Take an outdoor or indoor sequence from a rotating camera with very little parallax and use it to calibrate the focal length of your camera using the techniques described in Section 6.3.4 or Sections 9.1.3–9.2.1.

1. Take out any radial distortion in the images using one of the techniques from Exercises 6.10–6.11 or using parameters supplied for a given camera by your instructor.

2. Detect and match feature points across neighboring frames and chain them into feature tracks.
3. Compute homographies between overlapping frames and use Equations (6.56–6.57) to get an estimate of the focal length.
4. Compute a full 360° panorama and update your focal length estimate to close the gap (Section 9.1.4).
5. (Optional) Perform a complete bundle adjustment in the rotation matrices and focal length to obtain the highest quality estimate (Section 9.2.1).

Ex 6.7: Target-based calibration Use a three-dimensional target to calibrate your camera.

1. Construct a three-dimensional calibration pattern with known 3D locations. It is not easy to get high accuracy unless you use a machine shop, but you can get close using heavy plywood and printed patterns.
2. Find the corners, e.g., using a line finder and intersecting the lines.
3. Implement one of the iterative calibration and pose estimation algorithms described in Tsai (1987); Bogart (1991); Gleicher and Witkin (1992) or the system described in Section 6.2.2.
4. Take many pictures at different distances and orientations relative to the calibration target and report on both your re-projection errors and accuracy. (To do the latter, you may need to use simulated data.)

Ex 6.8: Calibration accuracy Compare the three calibration techniques (plane-based, rotation-based, and 3D-target-based).

One approach is to have a different student implement each one and to compare the results. Another approach is to use synthetic data, potentially re-using the software you developed for Exercise 2.3. The advantage of using synthetic data is that you know the ground truth for the calibration and pose parameters, you can easily run lots of experiments, and you can synthetically vary the noise in your measurements.

Here are some possible guidelines for constructing your test sets:

1. Assume a medium-wide focal length (say, 50° field of view).
2. For the plane-based technique, generate a 2D grid target and project it at different inclinations.
3. For a 3D target, create an inner cube corner and position it so that it fills most of field of view.
4. For the rotation technique, scatter points uniformly on a sphere until you get a similar number of points as for other techniques.

Before comparing your techniques, predict which one will be the most accurate (normalize your results by the square root of the number of points used).

Add varying amounts of noise to your measurements and describe the noise sensitivity of your various techniques.

Ex 6.9: Single view metrology Implement a system to measure dimensions and reconstruct a 3D model from a single image of a man-made scene using visible vanishing directions (Section 6.3.3) (Criminisi, Reid, and Zisserman 2000).

1. Find the three orthogonal vanishing points from parallel lines and use them to establish the three coordinate axes (rotation matrix \mathbf{R} of the camera relative to the scene). If two of the vanishing points are finite (not at infinity), use them to compute the focal length, assuming a known optical center. Otherwise, find some other way to calibrate your camera; you could use some of the techniques described by Schaffalitzky and Zisserman (2000).
2. Click on a ground plane point to establish your origin and click on a point a known distance away to establish the scene scale. This lets you compute the translation \mathbf{t} between the camera and the scene. As an alternative, click on a pair of points, one on the ground plane and one above it, and use the known height to establish the scene scale.
3. Write a user interface that lets you click on ground plane points to recover their 3D locations. (Hint: you already know the camera matrix, so knowledge of a point's z value is sufficient to recover its 3D location.) Click on pairs of points (one on the ground plane, one above it) to measure vertical heights.
4. Extend your system to let you draw quadrilaterals in the scene that correspond to axis-aligned rectangles in the world, using some of the techniques described by Sinha, Steedly, Szeliski *et al.* (2008). Export your 3D rectangles to a VRML or PLY¹⁵ file.
5. (Optional) Warp the pixels enclosed by the quadrilateral using the correct homography to produce a texture map for each planar polygon.

Ex 6.10: Radial distortion with plumb lines Implement a plumb-line algorithm to determine the radial distortion parameters.

1. Take some images of scenes with lots of straight lines, e.g., hallways in your home or office, and try to get some of the lines as close to the edges of the image as possible.
2. Extract the edges and link them into curves, as described in Section 4.2.2 and Exercise 4.8.
3. Fit quadratic or elliptic curves to the linked edges using a generalization of the successive line approximation algorithm described in Section 4.3.1 and Exercise 4.11 and keep the curves that fit this form well.
4. For each curved segment, fit a straight line and minimize the perpendicular distance between the curve and the line while adjusting the radial distortion parameters.
5. Alternate between re-fitting the straight line and adjusting the radial distortion parameters until convergence.

¹⁵ <http://meshlab.sf.net>.

Ex 6.11: Radial distortion with a calibration target Use a grid calibration target to determine the radial distortion parameters.

1. Print out a planar calibration target, mount it on a stiff board, and get it to fill your field of view.
2. Detect the squares, lines, or dots in your calibration target.
3. Estimate the homography mapping the target to the camera from the central portion of the image that does not have any radial distortion.
4. Predict the positions of the remaining targets and use the differences between the observed and predicted positions to estimate the radial distortion.
5. (Optional) Fit a general spline model (for severe distortion) instead of the quartic distortion model.
6. (Optional) Extend your technique to calibrate a fisheye lens.

Ex 6.12: Chromatic aberration Use the radial distortion estimates for each color channel computed in the previous exercise to clean up wide-angle lens images by warping all of the channels into alignment. (Optional) Straighten out the images at the same time.

Can you think of any reasons why this warping strategy may not always work?

Chapter 7

Structure from motion

7.1	Triangulation	305
7.2	Two-frame structure from motion	307
7.2.1	Projective (uncalibrated) reconstruction	312
7.2.2	Self-calibration	313
7.2.3	<i>Application:</i> View morphing	315
7.3	Factorization	315
7.3.1	Perspective and projective factorization	318
7.3.2	<i>Application:</i> Sparse 3D model extraction	319
7.4	Bundle adjustment	320
7.4.1	Exploiting sparsity	322
7.4.2	<i>Application:</i> Match move and augmented reality	324
7.4.3	Uncertainty and ambiguities	326
7.4.4	<i>Application:</i> Reconstruction from Internet photos	327
7.5	Constrained structure and motion	329
7.5.1	Line-based techniques	330
7.5.2	Plane-based techniques	331
7.6	Additional reading	332
7.7	Exercises	332

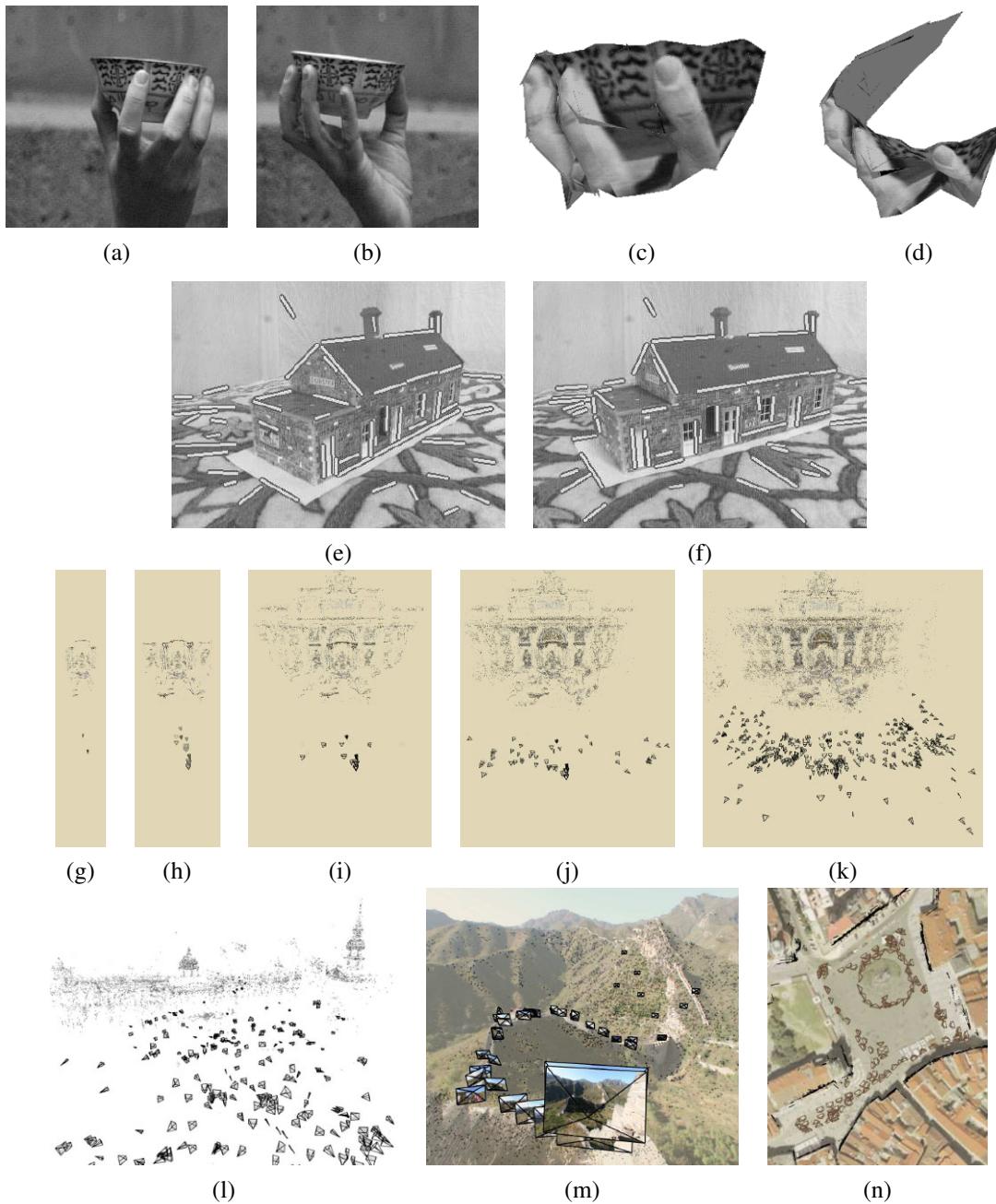


Figure 7.1 Structure from motion systems: (a–d) orthographic factorization (Tomasi and Kanade 1992) © 1992 Springer; (e–f) line matching (Schmid and Zisserman 1997) © 1997 IEEE; (g–k) incremental structure from motion (Snavely, Seitz, and Szeliski 2006); (l) 3D reconstruction of Trafalgar Square (Snavely, Seitz, and Szeliski 2006); (m) 3D reconstruction of the Great Wall of China (Snavely, Seitz, and Szeliski 2006); (n) 3D reconstruction of the Old Town Square, Prague (Snavely, Seitz, and Szeliski 2006) © 2006 ACM.

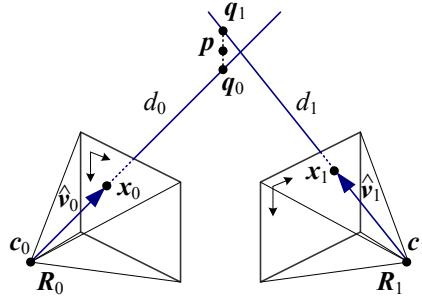


Figure 7.2 3D point triangulation by finding the point p that lies nearest to all of the optical rays $c_j + d_j \hat{v}_j$.

In the previous chapter, we saw how 2D and 3D point sets could be aligned and how such alignments could be used to estimate both a camera’s pose and its internal calibration parameters. In this chapter, we look at the converse problem of estimating the locations of 3D points from multiple images given only a sparse set of correspondences between image features. While this process often involves simultaneously estimating both 3D geometry (structure) and camera pose (motion), it is commonly known as *structure from motion* (Ullman 1979).

The topics of projective geometry and structure from motion are extremely rich and some excellent textbooks and surveys have been written on them (Faugeras and Luong 2001; Hartley and Zisserman 2004; Moons, Van Gool, and Vergauwen 2010). This chapter skips over a lot of the richer material available in these books, such as the trifocal tensor and algebraic techniques for full self-calibration, and concentrates instead on the basics that we have found useful in large-scale, image-based reconstruction problems (Snavely, Seitz, and Szeliski 2006).

We begin with a brief discussion of *triangulation* (Section 7.1), which is the problem of estimating a point’s 3D location when it is seen from multiple cameras. Next, we look at the two-frame structure from motion problem (Section 7.2), which involves the determination of the *epipolar geometry* between two cameras and which can also be used to recover certain information about the camera intrinsics using self-calibration (Section 7.2.2). Section 7.3 looks at *factorization* approaches to simultaneously estimating structure and motion from large numbers of point tracks using orthographic approximations to the projection model. We then develop a more general and useful approach to structure from motion, namely the simultaneous *bundle adjustment* of all the camera and 3D structure parameters (Section 7.4). We also look at special cases that arise when there are higher-level structures, such as lines and planes, in the scene (Section 7.5).

7.1 Triangulation

The problem of determining a point’s 3D position from a set of corresponding image locations and known camera positions is known as *triangulation*. This problem is the converse of the pose estimation problem we studied in Section 6.2.

One of the simplest ways to solve this problem is to find the 3D point p that lies closest to all of the 3D rays corresponding to the 2D matching feature locations $\{x_j\}$ observed by cam-

eras $\{\mathbf{P}_j = \mathbf{K}_j[\mathbf{R}_j|\mathbf{t}_j]\}$, where $\mathbf{t}_j = -\mathbf{R}_j\mathbf{c}_j$ and \mathbf{c}_j is the j th camera center (2.55–2.56). As you can see in Figure 7.2, these rays originate at \mathbf{c}_j in a direction $\hat{\mathbf{v}}_j = \mathcal{N}(\mathbf{R}_j^{-1}\mathbf{K}_j^{-1}\mathbf{x}_j)$. The nearest point to \mathbf{p} on this ray, which we denote as \mathbf{q}_j , minimizes the distance

$$\|\mathbf{c}_j + d_j \hat{\mathbf{v}}_j - \mathbf{p}\|^2, \quad (7.1)$$

which has a minimum at $d_j = \hat{\mathbf{v}}_j \cdot (\mathbf{p} - \mathbf{c}_j)$. Hence,

$$\mathbf{q}_j = \mathbf{c}_j + (\hat{\mathbf{v}}_j \hat{\mathbf{v}}_j^T)(\mathbf{p} - \mathbf{c}_j) = \mathbf{c}_j + (\mathbf{p} - \mathbf{c}_j)_\parallel, \quad (7.2)$$

in the notation of Equation (2.29), and the squared distance between \mathbf{p} and \mathbf{q}_j is

$$r_j^2 = \|(\mathbf{I} - \hat{\mathbf{v}}_j \hat{\mathbf{v}}_j^T)(\mathbf{p} - \mathbf{c}_j)\|^2 = \|(\mathbf{p} - \mathbf{c}_j)_\perp\|^2. \quad (7.3)$$

The optimal value for \mathbf{p} , which lies closest to all of the rays, can be computed as a regular least squares problem by summing over all the r_j^2 and finding the optimal value of \mathbf{p} ,

$$\mathbf{p} = \left[\sum_j (\mathbf{I} - \hat{\mathbf{v}}_j \hat{\mathbf{v}}_j^T) \right]^{-1} \left[\sum_j (\mathbf{I} - \hat{\mathbf{v}}_j \hat{\mathbf{v}}_j^T) \mathbf{c}_j \right]. \quad (7.4)$$

An alternative formulation, which is more statistically optimal and which can produce significantly better estimates if some of the cameras are closer to the 3D point than others, is to minimize the residual in the measurement equations

$$x_j = \frac{p_{00}^{(j)} X + p_{01}^{(j)} Y + p_{02}^{(j)} Z + p_{03}^{(j)} W}{p_{20}^{(j)} X + p_{21}^{(j)} Y + p_{22}^{(j)} Z + p_{23}^{(j)} W} \quad (7.5)$$

$$y_j = \frac{p_{10}^{(j)} X + p_{11}^{(j)} Y + p_{12}^{(j)} Z + p_{13}^{(j)} W}{p_{20}^{(j)} X + p_{21}^{(j)} Y + p_{22}^{(j)} Z + p_{23}^{(j)} W}, \quad (7.6)$$

where (x_j, y_j) are the measured 2D feature locations and $\{p_{00}^{(j)} \dots p_{23}^{(j)}\}$ are the known entries in camera matrix \mathbf{P}_j (Sutherland 1974).

As with Equations (6.21, 6.33, and 6.34), this set of non-linear equations can be converted into a linear least squares problem by multiplying both sides of the denominator. Note that if we use homogeneous coordinates $\mathbf{p} = (X, Y, Z, W)$, the resulting set of equations is homogeneous and is best solved as a singular value decomposition (SVD) or eigenvalue problem (looking for the smallest singular vector or eigenvector). If we set $W = 1$, we can use regular linear least squares, but the resulting system may be singular or poorly conditioned, i.e., if all of the viewing rays are parallel, as occurs for points far away from the camera.

For this reason, it is generally preferable to parameterize 3D points using homogeneous coordinates, especially if we know that there are likely to be points at greatly varying distances from the cameras. Of course, minimizing the set of observations (7.5–7.6) using non-linear least squares, as described in (6.14 and 6.23), is preferable to using linear least squares, regardless of the representation chosen.

For the case of two observations, it turns out that the location of the point \mathbf{p} that exactly minimizes the true reprojection error (7.5–7.6) can be computed using the solution of degree six equations (Hartley and Sturm 1997). Another problem to watch out for with triangulation is the issue of *chirality*, i.e., ensuring that the reconstructed points lie in front of all the

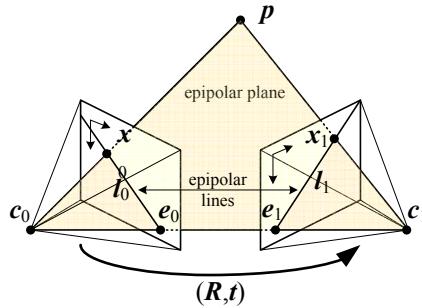


Figure 7.3 Epipolar geometry: The vectors $t = c_1 - c_0$, $p - c_0$ and $p - c_1$ are co-planar and define the basic epipolar constraint expressed in terms of the pixel measurements x_0 and x_1 .

cameras (Hartley 1998). While this cannot always be guaranteed, a useful heuristic is to take the points that lie behind the cameras because their rays are diverging (imagine Figure 7.2 where the rays were pointing *away* from each other) and to place them on the plane at infinity by setting their W values to 0.

7.2 Two-frame structure from motion

So far in our study of 3D reconstruction, we have always assumed that either the 3D point positions or the 3D camera poses are known in advance. In this section, we take our first look at *structure from motion*, which is the simultaneous recovery of 3D structure and pose from image correspondences.

Consider Figure 7.3, which shows a 3D point p being viewed from two cameras whose relative position can be encoded by a rotation R and a translation t . Since we do not know anything about the camera positions, without loss of generality, we can set the first camera at the origin $c_0 = 0$ and at a canonical orientation $R_0 = I$.

Now notice that the observed location of point p in the first image, $p_0 = d_0 \hat{x}_0$ is mapped into the second image by the transformation

$$d_1 \hat{x}_1 = p_1 = R p_0 + t = R(d_0 \hat{x}_0) + t, \quad (7.7)$$

where $\hat{x}_j = K_j^{-1} x_j$ are the (local) ray direction vectors. Taking the cross product of both sides with t in order to annihilate it on the right hand side yields¹

$$d_1 [t]_{\times} \hat{x}_1 = d_0 [t]_{\times} R \hat{x}_0. \quad (7.8)$$

Taking the dot product of both sides with \hat{x}_1 yields

$$d_0 \hat{x}_1^T ([t]_{\times} R) \hat{x}_0 = d_1 \hat{x}_1^T [t]_{\times} \hat{x}_1 = 0, \quad (7.9)$$

since the right hand side is a triple product with two identical entries. (Another way to say this is that the cross product matrix $[t]_{\times}$ is skew symmetric and returns 0 when pre- and post-multiplied by the same vector.)

¹ The cross-product operator $[\cdot]_{\times}$ was introduced in (2.32).

We therefore arrive at the basic *epipolar constraint*

$$\hat{\mathbf{x}}_1^T \mathbf{E} \hat{\mathbf{x}}_0 = 0, \quad (7.10)$$

where

$$\mathbf{E} = [\mathbf{t}]_{\times} \mathbf{R} \quad (7.11)$$

is called the *essential matrix* (Longuet-Higgins 1981).

An alternative way to derive the epipolar constraint is to notice that in order for the cameras to be oriented so that the rays $\hat{\mathbf{x}}_0$ and $\hat{\mathbf{x}}_1$ intersect in 3D at point \mathbf{p} , the vectors connecting the two camera centers $\mathbf{c}_1 - \mathbf{c}_0 = -\mathbf{R}_1^{-1}\mathbf{t}$ and the rays corresponding to pixels \mathbf{x}_0 and \mathbf{x}_1 , namely $\mathbf{R}_j^{-1}\hat{\mathbf{x}}_j$, must be co-planar. This requires that the triple product

$$(\hat{\mathbf{x}}_0, \mathbf{R}^{-1}\hat{\mathbf{x}}_1, -\mathbf{R}^{-1}\mathbf{t}) = (\mathbf{R}\hat{\mathbf{x}}_0, \hat{\mathbf{x}}_1, -\mathbf{t}) = \hat{\mathbf{x}}_1 \cdot (\mathbf{t} \times \mathbf{R}\hat{\mathbf{x}}_0) = \hat{\mathbf{x}}_1^T ([\mathbf{t}]_{\times} \mathbf{R}) \hat{\mathbf{x}}_0 = 0. \quad (7.12)$$

Notice that the essential matrix \mathbf{E} maps a point $\hat{\mathbf{x}}_0$ in image 0 into a line $\mathbf{l}_1 = \mathbf{E}\hat{\mathbf{x}}_0$ in image 1, since $\hat{\mathbf{x}}_1^T \mathbf{l}_1 = 0$ (Figure 7.3). All such lines must pass through the second *epipole* \mathbf{e}_1 , which is therefore defined as the left singular vector of \mathbf{E} with a 0 singular value, or, equivalently, the projection of the vector \mathbf{t} into image 1. The dual (transpose) of these relationships gives us the epipolar line in the first image as $\mathbf{l}_0 = \mathbf{E}^T \hat{\mathbf{x}}_1$ and \mathbf{e}_0 as the zero-value right singular vector of \mathbf{E} .

Given this fundamental relationship (7.10), how can we use it to recover the camera motion encoded in the essential matrix \mathbf{E} ? If we have N corresponding measurements $\{(\mathbf{x}_{i0}, \mathbf{x}_{i1})\}$, we can form N homogeneous equations in the nine elements of $\mathbf{E} = \{e_{00} \dots e_{22}\}$,

$$\begin{aligned} x_{i0}x_{i1}e_{00} &+ y_{i0}x_{i1}e_{01} &+ x_{i1}e_{02} &+ \\ x_{i0}y_{i1}e_{00} &+ y_{i0}y_{i1}e_{11} &+ y_{i1}e_{12} &+ \\ x_{i0}e_{20} &+ y_{i0}e_{21} &+ e_{22} &= 0 \end{aligned} \quad (7.13)$$

where $\mathbf{x}_{ij} = (x_{ij}, y_{ij}, 1)$. This can be written more compactly as

$$[\mathbf{x}_{i1} \mathbf{x}_{i0}^T] \otimes \mathbf{E} = \mathbf{Z}_i \otimes \mathbf{E} = \mathbf{z}_i \cdot \mathbf{f} = 0, \quad (7.14)$$

where \otimes indicates an element-wise multiplication and summation of matrix elements, and \mathbf{z}_i and \mathbf{f} are the rasterized (vector) forms of the $\mathbf{Z}_i = \hat{\mathbf{x}}_{i1}\hat{\mathbf{x}}_{i0}^T$ and \mathbf{E} matrices.² Given $N \geq 8$ such equations, we can compute an estimate (up to scale) for the entries in \mathbf{E} using an SVD.

In the presence of noisy measurements, how close is this estimate to being statistically optimal? If you look at the entries in (7.13), you can see that some entries are the products of image measurements such as $x_{i0}y_{i1}$ and others are direct image measurements (or even the identity). If the measurements have comparable noise, the terms that are products of measurements have their noise amplified by the other element in the product, which can lead to very poor scaling, e.g., an inordinately large influence of points with large coordinates (far away from the image center).

In order to counteract this trend, Hartley (1997a) suggests that the point coordinates should be translated and scaled so that their centroid lies at the origin and their variance is unity, i.e.,

$$\tilde{x}_i = s(x_i - \mu_x) \quad (7.15)$$

$$\tilde{y}_i = s(y_i - \mu_y) \quad (7.16)$$

² We use \mathbf{f} instead of \mathbf{e} to denote the rasterized form of \mathbf{E} to avoid confusion with the epipoles \mathbf{e}_j .

such that $\sum_i \tilde{x}_i = \sum_i \tilde{y}_i = 0$ and $\sum_i \tilde{x}_i^2 + \sum_i \tilde{y}_i^2 = 2n$, where n is the number of points.³

Once the essential matrix $\tilde{\mathbf{E}}$ has been computed from the transformed coordinates $\{(\tilde{\mathbf{x}}_{i0}, \tilde{\mathbf{x}}_{i1})\}$, where $\tilde{\mathbf{x}}_{ij} = \mathbf{T}_j \hat{\mathbf{x}}_{ij}$, the original essential matrix \mathbf{E} can be recovered as

$$\mathbf{E} = \mathbf{T}_1 \tilde{\mathbf{E}} \mathbf{T}_0. \quad (7.17)$$

In his paper, Hartley (1997a) compares the improvement due to his re-normalization strategy to alternative distance measures proposed by others such as Zhang (1998a,b) and concludes that his simple re-normalization in most cases is as effective as (or better than) alternative techniques. Torr and Fitzgibbon (2004) recommend a variant on this algorithm where the norm of the upper 2×2 sub-matrix of \mathbf{E} is set to 1 and show that it has even better stability with respect to 2D coordinate transformations.

Once an estimate for the essential matrix \mathbf{E} has been recovered, the direction of the translation vector \mathbf{t} can be estimated. Note that the absolute distance between the two cameras can never be recovered from pure image measurements alone, regardless of how many cameras or points are used. Knowledge about absolute camera and point positions or distances, often called *ground control points* in photogrammetry, is always required to establish the final scale, position, and orientation.

To estimate this direction $\hat{\mathbf{t}}$, observe that under ideal noise-free conditions, the essential matrix \mathbf{E} is singular, i.e., $\hat{\mathbf{t}}^T \mathbf{E} = 0$. This singularity shows up as a singular value of 0 when an SVD of \mathbf{E} is performed,

$$\mathbf{E} = [\hat{\mathbf{t}}]_\times \mathbf{R} = \mathbf{U} \Sigma \mathbf{V}^T = \begin{bmatrix} \mathbf{u}_0 & \mathbf{u}_1 & \hat{\mathbf{t}} \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ & & 0 \end{bmatrix} \begin{bmatrix} \mathbf{v}_0^T \\ \mathbf{v}_1^T \\ \mathbf{v}_2^T \end{bmatrix} \quad (7.18)$$

When \mathbf{E} is computed from noisy measurements, the singular vector associated with the smallest singular value gives us $\hat{\mathbf{t}}$. (The other two singular values should be similar but are not, in general, equal to 1 because \mathbf{E} is only computed up to an unknown scale.)

Because \mathbf{E} is rank-deficient, it turns out that we actually only need seven correspondences of the form of Equation (7.14) instead of eight to estimate this matrix (Hartley 1994a; Torr and Murray 1997; Hartley and Zisserman 2004). (The advantage of using fewer correspondences inside a RANSAC robust fitting stage is that fewer random samples need to be generated.) From this set of seven homogeneous equations (which we can stack into a 7×9 matrix for SVD analysis), we can find two independent vectors, say \mathbf{f}_0 and \mathbf{f}_1 such that $\mathbf{z}_i \cdot \mathbf{f}_j = 0$. These two vectors can be converted back into 3×3 matrices \mathbf{E}_0 and \mathbf{E}_1 , which span the solution space for

$$\mathbf{E} = \alpha \mathbf{E}_0 + (1 - \alpha) \mathbf{E}_1. \quad (7.19)$$

To find the correct value of α , we observe that \mathbf{E} has a zero determinant, since it is rank deficient, and hence

$$\det |\alpha \mathbf{E}_0 + (1 - \alpha) \mathbf{E}_1| = 0. \quad (7.20)$$

This gives us a cubic equation in α , which has either one or three solutions (roots). Substituting these values into (7.19) to obtain \mathbf{E} , we can test this essential matrix against other unused feature correspondences to select the correct one.

³ More precisely, Hartley (1997a) suggests scaling the points “so that the average distance from the origin is equal to $\sqrt{2}$ ” but the heuristic of unit variance is faster to compute (does not require per-point square roots) and should yield comparable improvements.

Once $\hat{\mathbf{t}}$ has been recovered, how can we estimate the corresponding rotation matrix \mathbf{R} ? Recall that the cross-product operator $[\hat{\mathbf{t}}]_{\times}$ (2.32) projects a vector onto a set of orthogonal basis vectors that include $\hat{\mathbf{t}}$, zeros out the $\hat{\mathbf{t}}$ component, and rotates the other two by 90° ,

$$[\hat{\mathbf{t}}]_{\times} = \mathbf{S} \mathbf{Z} \mathbf{R}_{90^\circ} \mathbf{S}^T = \begin{bmatrix} s_0 & s_1 & \hat{\mathbf{t}} \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ & & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 & \\ 1 & 0 & \\ & & 1 \end{bmatrix} \begin{bmatrix} \mathbf{s}_0^T \\ \mathbf{s}_1^T \\ \hat{\mathbf{t}}^T \end{bmatrix}, \quad (7.21)$$

where $\hat{\mathbf{t}} = \mathbf{s}_0 \times \mathbf{s}_1$. From Equations (7.18 and 7.21), we get

$$\mathbf{E} = [\hat{\mathbf{t}}]_{\times} \mathbf{R} = \mathbf{S} \mathbf{Z} \mathbf{R}_{90^\circ} \mathbf{S}^T \mathbf{R} = \mathbf{U} \Sigma \mathbf{V}^T, \quad (7.22)$$

from which we can conclude that $\mathbf{S} = \mathbf{U}$. Recall that for a noise-free essential matrix, ($\Sigma = \mathbf{Z}$), and hence

$$\mathbf{R}_{90^\circ} \mathbf{U}^T \mathbf{R} = \mathbf{V}^T \quad (7.23)$$

and

$$\mathbf{R} = \mathbf{U} \mathbf{R}_{90^\circ}^T \mathbf{V}^T. \quad (7.24)$$

Unfortunately, we only know both \mathbf{E} and $\hat{\mathbf{t}}$ up to a sign. Furthermore, the matrices \mathbf{U} and \mathbf{V} are not guaranteed to be rotations (you can flip both their signs and still get a valid SVD). For this reason, we have to generate all four possible rotation matrices

$$\mathbf{R} = \pm \mathbf{U} \mathbf{R}_{\pm 90^\circ}^T \mathbf{V}^T \quad (7.25)$$

and keep the two whose determinant $|\mathbf{R}| = 1$. To disambiguate between the remaining pair of potential rotations, which form a *twisted pair* (Hartley and Zisserman 2004, p. 240), we need to pair them with both possible signs of the translation direction $\pm \hat{\mathbf{t}}$ and select the combination for which the largest number of points is seen in front of both cameras.⁴

The property that points must lie in front of the camera, i.e., at a positive distance along the viewing rays emanating from the camera, is known as *chirality* (Hartley 1998). In addition to determining the signs of the rotation and translation, as described above, the chirality (sign of the distances) of the points in a reconstruction can be used inside a RANSAC procedure (along with the reprojection errors) to distinguish between likely and unlikely configurations.⁵ Chirality can also be used to transform projective reconstructions (Sections 7.2.1 and 7.2.2) into *quasi-affine* reconstructions (Hartley 1998).

The normalized “eight-point algorithm” (Hartley 1997a) described above is not the only way to estimate the camera motion from correspondences. Variants include using seven points while enforcing the rank two constraint in \mathbf{E} (7.19–7.20) and a five-point algorithm that requires finding the roots of a 10th degree polynomial (Nistér 2004). Since such algorithms use fewer points to compute their estimates, they are less sensitive to outliers when used as part of a random sampling (RANSAC) strategy.

⁴ In the noise-free case, a single point suffices. It is safer, however, to test all or a sufficient subset of points, downweighting the ones that lie close to the plane at infinity, for which it is easy to get depth reversals.

⁵ Note that as points get further away from a camera, i.e., closer toward the plane at infinity, errors in chirality become more likely.

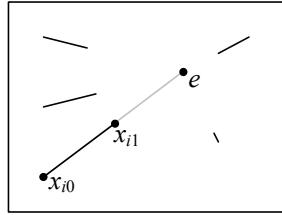


Figure 7.4 Pure translational camera motion results in visual motion where all the points move towards (or away from) a common *focus of expansion* (FOE) e . They therefore satisfy the triple product condition $(\mathbf{x}_0, \mathbf{x}_1, \mathbf{e}) = \mathbf{e} \cdot (\mathbf{x}_0 \times \mathbf{x}_1) = 0$.

Pure translation (known rotation)

In the case where we know the rotation, we can pre-rotate the points in the second image to match the viewing direction of the first. The resulting set of 3D points all move towards (or away from) the *focus of expansion* (FOE), as shown in Figure 7.4.⁶ The resulting essential matrix \mathbf{E} is (in the noise-free case) skew symmetric and so can be estimated more directly by setting $e_{ij} = -e_{ji}$ and $e_{ii} = 0$ in (7.13). Two points with non-zero parallax now suffice to estimate the FOE.

A more direct derivation of the FOE estimate can be obtained by minimizing the triple product

$$\sum_i (\mathbf{x}_{i0}, \mathbf{x}_{i1}, \mathbf{e})^2 = \sum_i ((\mathbf{x}_{i0} \times \mathbf{x}_{i1}) \cdot \mathbf{e})^2, \quad (7.26)$$

which is equivalent to finding the null space for the set of equations

$$(y_{i0} - y_{i1})e_0 + (x_{i1} - x_{i0})e_1 + (x_{i0}y_{i1} - y_{i0}x_{i1})e_2 = 0. \quad (7.27)$$

Note that, as in the eight-point algorithm, it is advisable to normalize the 2D points to have unit variance before computing this estimate.

In situations where a large number of points at infinity are available, e.g., when shooting outdoor scenes or when the camera motion is small compared to distant objects, this suggests an alternative RANSAC strategy for estimating the camera motion. First, pick a pair of points to estimate a rotation, hoping that both of the points lie at infinity (very far from the camera). Then, compute the FOE and check whether the residual error is small (indicating agreement with this rotation hypothesis) and whether the motions towards or away from the epipole (FOE) are all in the same direction (ignoring very small motions, which may be noise-contaminated).

Pure rotation

The case of pure rotation results in a degenerate estimate of the essential matrix \mathbf{E} and of the translation direction $\hat{\mathbf{t}}$. Consider first the case of the rotation matrix being known. The estimates for the FOE will be degenerate, since $\mathbf{x}_{i0} \approx \mathbf{x}_{i1}$, and hence (7.27), is degenerate. A similar argument shows that the equations for the essential matrix (7.13) are also rank-deficient.

⁶ Fans of *Star Trek* and *Star Wars* will recognize this as the “jump to hyperdrive” visual effect.

This suggests that it might be prudent before computing a full essential matrix to first compute a rotation estimate \mathbf{R} using (6.32), potentially with just a small number of points, and then compute the residuals after rotating the points before proceeding with a full \mathbf{E} computation.

7.2.1 Projective (uncalibrated) reconstruction

In many cases, such as when trying to build a 3D model from Internet or legacy photos taken by unknown cameras without any EXIF tags, we do not know ahead of time the intrinsic calibration parameters associated with the input images. In such situations, we can still estimate a two-frame reconstruction, although the true metric structure may not be available, e.g., orthogonal lines or planes in the world may not end up being reconstructed as orthogonal.

Consider the derivations we used to estimate the essential matrix \mathbf{E} (7.10–7.12). In the uncalibrated case, we do not know the calibration matrices \mathbf{K}_j , so we cannot use the normalized ray directions $\hat{\mathbf{x}}_j = \mathbf{K}_j^{-1} \mathbf{x}_j$. Instead, we have access only to the image coordinates \mathbf{x}_j , and so the essential matrix (7.10) becomes

$$\hat{\mathbf{x}}_1^T \mathbf{E} \hat{\mathbf{x}}_1 = \mathbf{x}_1^T \mathbf{K}_1^{-T} \mathbf{E} \mathbf{K}_0^{-1} \mathbf{x}_0 = \mathbf{x}_1^T \mathbf{F} \mathbf{x}_0 = 0, \quad (7.28)$$

where

$$\mathbf{F} = \mathbf{K}_1^{-T} \mathbf{E} \mathbf{K}_0^{-1} = [\mathbf{e}]_{\times} \tilde{\mathbf{H}} \quad (7.29)$$

is called the *fundamental matrix* (Faugeras 1992; Hartley, Gupta, and Chang 1992; Hartley and Zisserman 2004).

Like the essential matrix, the fundamental matrix is (in principle) rank two,

$$\mathbf{F} = [\mathbf{e}]_{\times} \tilde{\mathbf{H}} = \mathbf{U} \boldsymbol{\Sigma} \mathbf{V}^T = \begin{bmatrix} \mathbf{u}_0 & \mathbf{u}_1 & \mathbf{e}_1 \end{bmatrix} \begin{bmatrix} \sigma_0 & & \\ & \sigma_1 & \\ & & 0 \end{bmatrix} \begin{bmatrix} \mathbf{v}_0^T \\ \mathbf{v}_1^T \\ \mathbf{e}_0^T \end{bmatrix}. \quad (7.30)$$

Its smallest left singular vector indicates the epipole \mathbf{e}_1 in the image 1 and its smallest right singular vector is \mathbf{e}_0 (Figure 7.3). The homography $\tilde{\mathbf{H}}$ in (7.29), which in principle should equal

$$\tilde{\mathbf{H}} = \mathbf{K}_1^{-T} \mathbf{R} \mathbf{K}_0^{-1}, \quad (7.31)$$

cannot be uniquely recovered from \mathbf{F} , since any homography of the form $\tilde{\mathbf{H}}' = \tilde{\mathbf{H}} + \mathbf{e} \mathbf{v}^T$ results in the same \mathbf{F} matrix. (Note that $[\mathbf{e}]_{\times}$ annihilates any multiple of \mathbf{e} .)

Any one of these valid homographies $\tilde{\mathbf{H}}$ maps some plane in the scene from one image to the other. It is not possible to tell in advance which one it is without either selecting four or more co-planar correspondences to compute $\tilde{\mathbf{H}}$ as part of the \mathbf{F} estimation process (in a manner analogous to guessing a rotation for \mathbf{E}) or mapping all points in one image through $\tilde{\mathbf{H}}$ and seeing which ones line up with their corresponding locations in the other.⁷

In order to create a *projective* reconstruction of the scene, we can pick any valid homography $\tilde{\mathbf{H}}$ that satisfies Equation (7.29). For example, following a technique analogous to Equations (7.18–7.24), we get

$$\mathbf{F} = [\mathbf{e}]_{\times} \tilde{\mathbf{H}} = \mathbf{S} \mathbf{Z} \mathbf{R}_{90^\circ} \mathbf{S}^T \tilde{\mathbf{H}} = \mathbf{U} \boldsymbol{\Sigma} \mathbf{V}^T \quad (7.32)$$

⁷ This process is sometimes referred to as *plane plus parallax* (Section 2.1.5) (Kumar, Anandan, and Hanna 1994; Sawhney 1994).

and hence

$$\tilde{\mathbf{H}} = \mathbf{U} \mathbf{R}_{90^\circ}^T \hat{\Sigma} \mathbf{V}^T, \quad (7.33)$$

where $\hat{\Sigma}$ is the singular value matrix with the smallest value replaced by a reasonable alternative (say, the middle value).⁸ We can then form a pair of camera matrices

$$\mathbf{P}_0 = [\mathbf{I}|\mathbf{0}] \quad \text{and} \quad \mathbf{P}_0 = [\tilde{\mathbf{H}}|\mathbf{e}], \quad (7.34)$$

from which a projective reconstruction of the scene can be computed using triangulation (Section 7.1).

While the projective reconstruction may not be useful in practice, it can often be *upgraded* to an affine or metric reconstruction, as detailed below. Even without this step, however, the fundamental matrix \mathbf{F} can be very useful in finding additional correspondences, as they must all lie on corresponding epipolar lines, i.e., any feature \mathbf{x}_0 in image 0 must have its correspondence lying on the associated epipolar line $\mathbf{l}_1 = \mathbf{F}\mathbf{x}_0$ in image 1, assuming that the point motions are due to a rigid transformation.

7.2.2 Self-calibration

The results of structure from motion computation are much more useful (and intelligible) if a *metric* reconstruction is obtained, i.e., one in which parallel lines are parallel, orthogonal walls are at right angles, and the reconstructed model is a scaled version of reality. Over the years, a large number of *self-calibration* (or *auto-calibration*) techniques have been developed for converting a projective reconstruction into a metric one, which is equivalent to recovering the unknown calibration matrices \mathbf{K}_j associated with each image (Hartley and Zisserman 2004; Moons, Van Gool, and Vergauwen 2010).

In situations where certain additional information is known about the scene, different methods may be employed. For example, if there are parallel lines in the scene (usually, having several lines converge on the same vanishing point is good evidence), three or more vanishing points, which are the images of points at infinity, can be used to establish the homography for the plane at infinity, from which focal lengths and rotations can be recovered. If two or more finite *orthogonal* vanishing points have been observed, the single-image calibration method based on vanishing points (Section 6.3.2) can be used instead.

In the absence of such external information, it is not possible to recover a fully parameterized independent calibration matrix \mathbf{K}_j for each image from correspondences alone. To see this, consider the set of all camera matrices $\mathbf{P}_j = \mathbf{K}_j[\mathbf{R}_j|\mathbf{t}_j]$ projecting world coordinates $\mathbf{p}_i = (X_i, Y_i, Z_i, W_i)$ into screen coordinates $\mathbf{x}_{ij} \sim \mathbf{P}_j \mathbf{p}_i$. Now consider transforming the 3D scene $\{\mathbf{p}_i\}$ through an arbitrary 4×4 projective transformation $\tilde{\mathbf{H}}$, yielding a new model consisting of points $\mathbf{p}'_i = \tilde{\mathbf{H}} \mathbf{p}_i$. Post-multiplying each \mathbf{P}_j matrix by $\tilde{\mathbf{H}}^{-1}$ still produces the same screen coordinates and a new set calibration matrices can be computed by applying RQ decomposition to the new camera matrix $\mathbf{P}'_j = \mathbf{P}_j \tilde{\mathbf{H}}^{-1}$.

For this reason, all self-calibration methods assume some restricted form of the calibration matrix, either by setting or equating some of their elements or by assuming that they do not vary over time. While most of the techniques discussed by Hartley and Zisserman (2004);

⁸ Hartley and Zisserman (2004, p. 237) recommend using $\tilde{\mathbf{H}} = [\mathbf{e}] \times \mathbf{F}$ (Luong and Viéville 1996), which places the camera on the plane at infinity.

Moons, Van Gool, and Vergauwen (2010) require three or more frames, in this section we present a simple technique that can recover the focal lengths (f_0, f_1) of both images from the fundamental matrix \mathbf{F} in a two-frame reconstruction (Hartley and Zisserman 2004, p. 456).

To accomplish this, we assume that the camera has zero skew, a known aspect ratio (usually set to 1), and a known optical center, as in Equation (2.59). How reasonable is this assumption in practice? The answer, as with many questions, is “it depends”.

If absolute metric accuracy is required, as in photogrammetry applications, it is imperative to pre-calibrate the cameras using one of the techniques from Section 6.3 and to use ground control points to pin down the reconstruction. If instead, we simply wish to reconstruct the world for visualization or image-based rendering applications, as in the Photo Tourism system of Snavely, Seitz, and Szeliski (2006), this assumption is quite reasonable in practice.

Most cameras today have square pixels and an optical center near the middle of the image, and are much more likely to deviate from a simple camera model due to radial distortion (Section 6.3.5), which should be compensated for whenever possible. The biggest problems occur when images have been cropped off-center, in which case the optical center will no longer be in the middle, or when perspective pictures have been taken of a different picture, in which case a general camera matrix becomes necessary.⁹

Given these caveats, the two-frame focal length estimation algorithm based on the Kruppa equations developed by Hartley and Zisserman (2004, p. 456) proceeds as follows. Take the left and right singular vectors $\{\mathbf{u}_0, \mathbf{u}_1, \mathbf{v}_0, \mathbf{v}_1\}$ of the fundamental matrix \mathbf{F} (7.30) and their associated singular values $\{\sigma_0, \sigma_1\}$ and form the following set of equations:

$$\frac{\mathbf{u}_1^T \mathbf{D}_0 \mathbf{u}_1}{\sigma_0^2 \mathbf{v}_0^T \mathbf{D}_1 \mathbf{v}_0} = -\frac{\mathbf{u}_0^T \mathbf{D}_0 \mathbf{u}_1}{\sigma_0 \sigma_1 \mathbf{v}_0^T \mathbf{D}_1 \mathbf{v}_1} = \frac{\mathbf{u}_0^T \mathbf{D}_0 \mathbf{u}_0}{\sigma_1^2 \mathbf{v}_1^T \mathbf{D}_1 \mathbf{v}_1}, \quad (7.35)$$

where the two matrices

$$\mathbf{D}_j = \mathbf{K}_j \mathbf{K}_j^T = \text{diag}(f_j^2, f_j^2, 1) = \begin{bmatrix} f_j^2 & & \\ & f_j^2 & \\ & & 1 \end{bmatrix} \quad (7.36)$$

encode the unknown focal lengths. For simplicity, let us rewrite each of the numerators and denominators in (7.35) as

$$e_{ij0}(f_0^2) = \mathbf{u}_i^T \mathbf{D}_0 \mathbf{u}_j = a_{ij} + b_{ij} f_0^2, \quad (7.37)$$

$$e_{ij1}(f_1^2) = \sigma_i \sigma_j \mathbf{v}_i^T \mathbf{D}_1 \mathbf{v}_j = c_{ij} + d_{ij} f_1^2. \quad (7.38)$$

Notice that each of these is affine (linear plus constant) in either f_0^2 or f_1^2 . Hence, we can cross-multiply these equations to obtain quadratic equations in f_j^2 , which can readily be solved. (See also the work by Bougnoux (1998) for some alternative formulations.)

An alternative solution technique is to observe that we have a set of three equations related by an unknown scalar λ , i.e.,

$$e_{ij0}(f_0^2) = \lambda e_{ij1}(f_1^2) \quad (7.39)$$

(Richard Hartley, personal communication, July 2009). These can readily be solved to yield $(f_0^2, \lambda f_1^2, \lambda)$ and hence (f_0, f_1) .

⁹ In Photo Tourism, our system registered photographs of an information sign outside Notre Dame with real pictures of the cathedral.

How well does this approach work in practice? There are certain degenerate configurations, such as when there is no rotation or when the optical axes intersect, when it does not work at all. (In such a situation, you can vary the focal lengths of the cameras and obtain a deeper or shallower reconstruction, which is an example of a *bas-relief ambiguity* (Section 7.4.3).) Hartley and Zisserman (2004) recommend using techniques based on three or more frames. However, if you find two images for which the estimates of $(f_0^2, \lambda f_1^2, \lambda)$ are well conditioned, they can be used to initialize a more complete bundle adjustment of all the parameters (Section 7.4). An alternative, which is often used in systems such as Photo Tourism, is to use camera EXIF tags or generic default values to initialize focal length estimates and refine them as part of bundle adjustment.

7.2.3 Application: View morphing

An interesting application of basic two-frame structure from motion is *view morphing* (also known as *view interpolation*, see Section 13.1), which can be used to generate a smooth 3D animation from one view of a 3D scene to another (Chen and Williams 1993; Seitz and Dyer 1996).

To create such a transition, you must first smoothly interpolate the camera matrices, i.e., the camera positions, orientations, and focal lengths. While simple linear interpolation can be used (representing rotations as quaternions (Section 2.1.4)), a more pleasing effect is obtained by *easing in* and *easing out* the camera parameters, e.g., using a raised cosine, as well as moving the camera along a more circular trajectory (Snavely, Seitz, and Szeliski 2006).

To generate in-between frames, either a full set of 3D correspondences needs to be established (Section 11.3) or 3D models (proxies) must be created for each reference view. Section 13.1 describes several widely used approaches to this problem. One of the simplest is to just triangulate the set of matched feature points in each image, e.g., using Delaunay triangulation. As the 3D points are re-projected into their intermediate views, pixels can be mapped from their original source images to their new views using affine or projective mapping (Szeliski and Shum 1997). The final image is then composited using a linear blend of the two reference images, as with usual morphing (Section 3.6.3).

7.3 Factorization

When processing video sequences, we often get extended *feature tracks* (Section 4.1.4) from which it is possible to recover the structure and motion using a process called *factorization*. Consider the tracks generated by a rotating ping pong ball, which has been marked with dots to make its shape and motion more discernable (Figure 7.5). We can readily see from the shape of the tracks that the moving object must be a sphere, but how can we infer this mathematically?

It turns out that, under orthography or related models we discuss below, the shape and motion can be recovered simultaneously using a singular value decomposition (Tomasi and Kanade 1992). Consider the orthographic and weak perspective projection models introduced in Equations (2.47–2.49). Since the last row is always $[0 \ 0 \ 0 \ 1]$, there is no perspective division and we can write

$$\mathbf{x}_{ji} = \tilde{\mathbf{P}}_j \bar{\mathbf{p}}_i, \quad (7.40)$$

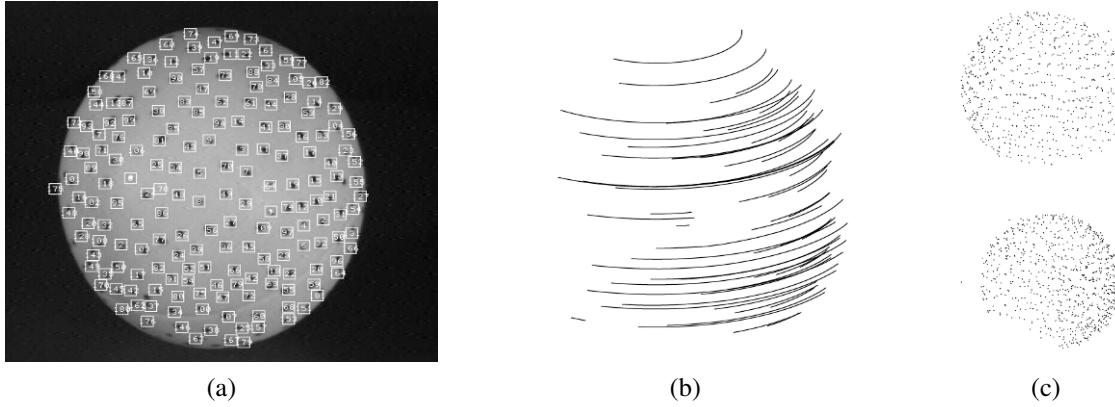


Figure 7.5 3D reconstruction of a rotating ping pong ball using factorization (Tomasi and Kanade 1992) © 1992 Springer: (a) sample image with tracked features overlaid; (b) subsampled feature motion stream; (c) two views of the reconstructed 3D model.

where \mathbf{x}_{ji} is the location of the i th point in the j th frame, $\tilde{\mathbf{P}}_j$ is the upper 2×4 portion of the projection matrix \mathbf{P}_j , and $\bar{\mathbf{p}}_i = (X_i, Y_i, Z_i, 1)$ is the augmented 3D point position.¹⁰

Let us assume (for now) that every point i is visible in every frame j . We can take the *centroid* (average) of the projected point locations \mathbf{x}_{ji} in frame j ,

$$\bar{\mathbf{x}}_j = \frac{1}{N} \sum_i \mathbf{x}_{ji} = \tilde{\mathbf{P}}_j \frac{1}{N} \sum_i \bar{\mathbf{p}}_i = \tilde{\mathbf{P}}_j \bar{\mathbf{c}}, \quad (7.41)$$

where $\bar{\mathbf{c}} = (\bar{X}, \bar{Y}, \bar{Z}, 1)$ is the augmented 3D centroid of the point cloud.

Since world coordinate frames in structure from motion are always arbitrary, i.e., we cannot recover true 3D locations without ground control points (known measurements), we can place the origin of the world at the centroid of the points, i.e, $\bar{X} = \bar{Y} = \bar{Z} = 0$, so that $\bar{\mathbf{c}} = (0, 0, 0, 1)$. We see from this that the centroid of the 2D points in each frame $\bar{\mathbf{x}}_j$ directly gives us the last element of $\tilde{\mathbf{P}}_j$.

Let $\tilde{\mathbf{x}}_{ji} = \mathbf{x}_{ji} - \bar{\mathbf{x}}_j$ be the 2D point locations after their image centroid has been subtracted. We can now write

$$\tilde{\mathbf{x}}_{ji} = \mathbf{M}_j \mathbf{p}_i, \quad (7.42)$$

where \mathbf{M}_j is the upper 2×3 portion of the projection matrix \mathbf{P}_j and $\mathbf{p}_i = (X_i, Y_i, Z_i)$. We can concatenate all of these measurement equations into one large matrix

$$\hat{\mathbf{X}} = \begin{bmatrix} \tilde{\mathbf{x}}_{11} & \cdots & \tilde{\mathbf{x}}_{1i} & \cdots & \tilde{\mathbf{x}}_{1N} \\ \vdots & & \vdots & & \vdots \\ \tilde{\mathbf{x}}_{j1} & \cdots & \tilde{\mathbf{x}}_{ji} & \cdots & \tilde{\mathbf{x}}_{jN} \\ \vdots & & \vdots & & \vdots \\ \tilde{\mathbf{x}}_{M1} & \cdots & \tilde{\mathbf{x}}_{Mi} & \cdots & \tilde{\mathbf{x}}_{MN} \end{bmatrix} = \begin{bmatrix} \mathbf{M}_1 \\ \vdots \\ \mathbf{M}_j \\ \vdots \\ \mathbf{M}_M \end{bmatrix} [\mathbf{p}_1 \cdots \mathbf{p}_i \cdots \mathbf{p}_N] = \hat{\mathbf{M}} \hat{\mathbf{S}}. \quad (7.43)$$

$\hat{\mathbf{X}}$ is called the *measurement* matrix and $\hat{\mathbf{M}}$ and $(\hat{\mathbf{S}}$ are the *motion*) and *structure* matrices, respectively (Tomasi and Kanade 1992).

¹⁰ In this section, we index the 2D point positions as \mathbf{x}_{ji} instead of \mathbf{x}_{ij} , since this is the convention adopted by factorization papers (Tomasi and Kanade 1992) and is consistent with the factorization given in (7.43).

Because the motion matrix $\hat{\mathbf{M}}$ is $2M \times 3$ and the structure matrix $\hat{\mathbf{S}}$ is $3 \times N$, an SVD applied to $\hat{\mathbf{X}}$ has only three non-zero singular values. In the case where the measurements in $\hat{\mathbf{X}}$ are noisy, SVD returns the rank-three factorization of $\hat{\mathbf{X}}$ that is the closest to $\hat{\mathbf{X}}$ in a least squares sense (Tomasi and Kanade 1992; Golub and Van Loan 1996; Hartley and Zisserman 2004).

It would be nice if the SVD of $\hat{\mathbf{X}} = \mathbf{U}\Sigma\mathbf{V}^T$ directly returned the matrices $\hat{\mathbf{M}}$ and $\hat{\mathbf{S}}$, but it does not. Instead, we can write the relationship

$$\hat{\mathbf{X}} = \mathbf{U}\Sigma\mathbf{V}^T = [\mathbf{U}\mathbf{Q}][\mathbf{Q}^{-1}\Sigma\mathbf{V}^T] \quad (7.44)$$

and set $\hat{\mathbf{M}} = \mathbf{U}\mathbf{Q}$ and $\hat{\mathbf{S}} = \mathbf{Q}^{-1}\Sigma\mathbf{V}^T$.¹¹

How can we recover the values of the 3×3 matrix \mathbf{Q} ? This depends on the motion model being used. In the case of orthographic projection (2.47), the entries in \mathbf{M}_j are the first two rows of rotation matrices \mathbf{R}_j , so we have

$$\begin{aligned} \mathbf{m}_{j0} \cdot \mathbf{m}_{j0} &= \mathbf{u}_{2j}\mathbf{Q}\mathbf{Q}^T\mathbf{u}_{2j}^T = 1, \\ \mathbf{m}_{j0} \cdot \mathbf{m}_{j1} &= \mathbf{u}_{2j}\mathbf{Q}\mathbf{Q}^T\mathbf{u}_{2j+1}^T = 0, \\ \mathbf{m}_{j1} \cdot \mathbf{m}_{j1} &= \mathbf{u}_{2j+1}\mathbf{Q}\mathbf{Q}^T\mathbf{u}_{2j+1}^T = 1, \end{aligned} \quad (7.45)$$

where \mathbf{u}_k are the 3×1 rows of the matrix \mathbf{U} . This gives us a large set of equations for the entries in the matrix $\mathbf{Q}\mathbf{Q}^T$, from which the matrix \mathbf{Q} can be recovered using a matrix square root (Appendix A.1.4). If we have scaled orthography (2.48), i.e., $\mathbf{M}_j = s_j\mathbf{R}_j$, the first and third equations are equal to s_j and can be set equal to each other.

Note that even once \mathbf{Q} has been recovered, there still exists a bas-relief ambiguity, i.e., we can never be sure if the object is rotating left to right or if its depth reversed version is moving the other way. (This can be seen in the classic rotating Necker Cube visual illusion.) Additional cues, such as the appearance and disappearance of points, or perspective effects, both of which are discussed below, can be used to remove this ambiguity.

For motion models other than pure orthography, e.g., for scaled orthography or paraperspective, the approach above must be extended in the appropriate manner. Such techniques are relatively straightforward to derive from first principles; more details can be found in papers that extend the basic factorization approach to these more flexible models (Poelman and Kanade 1997). Additional extensions of the original factorization algorithm include multi-body rigid motion (Costeira and Kanade 1995), sequential updates to the factorization (Morita and Kanade 1997), the addition of lines and planes (Morris and Kanade 1998), and re-scaling the measurements to incorporate individual location uncertainties (Anandan and Irani 2002).

A disadvantage of factorization approaches is that they require a complete set of tracks, i.e., each point must be visible in each frame, in order for the factorization approach to work. Tomasi and Kanade (1992) deal with this problem by first applying factorization to smaller denser subsets and then using known camera (motion) or point (structure) estimates to *hallucinate* additional missing values, which allows them to incrementally incorporate more features and cameras. Huynh, Hartley, and Heyden (2003) extend this approach to view missing data as special cases of outliers. Buchanan and Fitzgibbon (2005) develop fast iterative algorithms for performing large matrix factorizations with missing data. The general topic of

¹¹ Tomasi and Kanade (1992) first take the square root of Σ and distribute this to \mathbf{U} and \mathbf{V} , but there is no particular reason to do this.

principal component analysis (PCA) with missing data also appears in other computer vision problems (Shum, Ikeuchi, and Reddy 1995; De la Torre and Black 2003; Gross, Matthews, and Baker 2006; Torresani, Hertzmann, and Bregler 2008; Vidal, Ma, and Sastry 2010).

7.3.1 Perspective and projective factorization

Another disadvantage of regular factorization is that it cannot deal with perspective cameras. One way to get around this problem is to perform an initial affine (e.g., orthographic) reconstruction and to then correct for the perspective effects in an iterative manner (Christy and Horaud 1996).

Observe that the object-centered projection model (2.76)

$$x_{ji} = s_j \frac{\mathbf{r}_{xj} \cdot \mathbf{p}_i + t_{xj}}{1 + \eta_j \mathbf{r}_{zj} \cdot \mathbf{p}_i} \quad (7.46)$$

$$y_{ji} = s_j \frac{\mathbf{r}_{yj} \cdot \mathbf{p}_i + t_{yj}}{1 + \eta_j \mathbf{r}_{zj} \cdot \mathbf{p}_i} \quad (7.47)$$

differs from the scaled orthographic projection model (7.40) by the inclusion of the denominator terms $(1 + \eta_j \mathbf{r}_{zj} \cdot \mathbf{p}_i)$.¹²

If we knew the correct values of $\eta_j = t_{zj}^{-1}$ and the structure and motion parameters \mathbf{R}_j and \mathbf{p}_i , we could cross-multiply the left hand side (visible point measurements x_{ji} and y_{ji}) by the denominator and get corrected values, for which the bilinear projection model (7.40) is exact. In practice, after an initial reconstruction, the values of η_j can be estimated independently for each frame by comparing reconstructed and sensed point positions. (The third row of the rotation matrix \mathbf{r}_{zj} is always available as the cross-product of the first two rows.) Note that since the η_j are determined from the image measurements, the cameras do not have to be pre-calibrated, i.e., their focal lengths can be recovered from $f_j = s_j / \eta_j$.

Once the η_j have been estimated, the feature locations can then be corrected before applying another round of factorization. Note that because of the initial depth reversal ambiguity, both reconstructions have to be tried while calculating η_j . (The incorrect reconstruction will result in a negative η_j , which is not physically meaningful.) Christy and Horaud (1996) report that their algorithm usually converges in three to five iterations, with the majority of the time spent in the SVD computation.

An alternative approach, which does not assume partially calibrated cameras (known optical center, square pixels, and zero skew) is to perform a fully *projective* factorization (Sturm and Triggs 1996; Triggs 1996). In this case, the inclusion of the third row of the camera matrix in (7.40) is equivalent to multiplying each reconstructed measurement $\mathbf{x}_{ji} = \mathbf{M}_j \mathbf{p}_i$ by its inverse (projective) depth $\eta_{ji} = d_{ji}^{-1} = 1/(\mathbf{P}_{j2} \mathbf{p}_i)$ or, equivalently, multiplying each measured position by its projective depth d_{ji} ,

$$\hat{\mathbf{X}} = \begin{bmatrix} d_{11} \tilde{\mathbf{x}}_{11} & \cdots & d_{1i} \tilde{\mathbf{x}}_{1i} & \cdots & d_{1N} \tilde{\mathbf{x}}_{1N} \\ \vdots & & \vdots & & \vdots \\ d_{j1} \tilde{\mathbf{x}}_{j1} & \cdots & d_{ji} \tilde{\mathbf{x}}_{ji} & \cdots & d_{jN} \tilde{\mathbf{x}}_{jN} \\ \vdots & & \vdots & & \vdots \\ d_{M1} \tilde{\mathbf{x}}_{M1} & \cdots & d_{Mi} \tilde{\mathbf{x}}_{Mi} & \cdots & d_{MN} \tilde{\mathbf{x}}_{MN} \end{bmatrix} = \hat{\mathbf{M}} \hat{\mathbf{S}}. \quad (7.48)$$

¹² Assuming that the optical center (c_x, c_y) lies at $(0, 0)$ and that pixels are square.

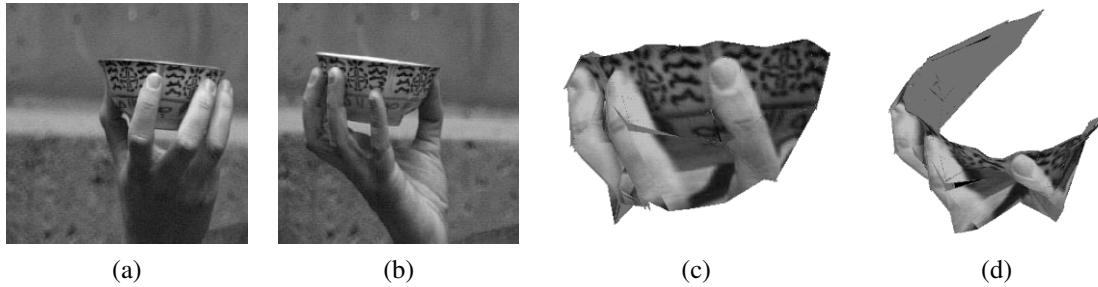


Figure 7.6 3D teacup model reconstructed from a 240-frame video sequence (Tomasi and Kanade 1992) © 1992 Springer: (a) first frame of video; (b) last frame of video; (c) side view of 3D model; (d) top view of 3D model.

In the original paper by Sturm and Triggs (1996), the projective depths d_{ji} are obtained from two-frame reconstructions, while in later work (Triggs 1996; Oliensis and Hartley 2007), they are initialized to $d_{ji} = 1$ and updated after each iteration. Oliensis and Hartley (2007) present an update formula that is guaranteed to converge to a fixed point. None of these authors suggest actually estimating the third row of \mathbf{P}_j as part of the projective depth computations. In any case, it is unclear when a fully projective reconstruction would be preferable to a partially calibrated one, especially if they are being used to initialize a full bundle adjustment of all the parameters.

One of the attractions of factorization methods is that they provide a “closed form” (sometimes called a “linear”) method to initialize iterative techniques such as bundle adjustment. An alternative initialization technique is to estimate the homographies corresponding to some common plane seen by all the cameras (Rother and Carlsson 2002). In a calibrated camera setting, this can correspond to estimating consistent rotations for all of the cameras, for example, using matched vanishing points (Antone and Teller 2002). Once these have been recovered, the camera positions can then be obtained by solving a linear system (Antone and Teller 2002; Rother and Carlsson 2002; Rother 2003).

7.3.2 Application: Sparse 3D model extraction

Once a multi-view 3D reconstruction of the scene has been estimated, it then becomes possible to create a texture-mapped 3D model of the object and to look at it from new directions.

The first step is to create a denser 3D model than the sparse point cloud that structure from motion produces. One alternative is to run dense multi-view stereo (Sections 11.3–11.6). Alternatively, a simpler technique such as 3D triangulation can be used, as shown in Figure 7.6, in which 207 reconstructed 3D points are triangulated to produce a surface mesh.

In order to create a more realistic model, a *texture map* can be extracted for each triangle face. The equations to map points on the surface of a 3D triangle to a 2D image are straightforward: just pass the local 2D coordinates on the triangle through the 3×4 camera projection matrix to obtain a 3×3 homography (planar perspective projection). When multiple source images are available, as is usually the case in multi-view reconstruction, either the closest and most fronto-parallel image can be used or multiple images can be blended in to deal with view-dependent foreshortening (Wang, Kang, Szeliski *et al.* 2001) or to obtain super-resolved results (Goldluecke and Cremers 2009). Another alternative is to create a sep-

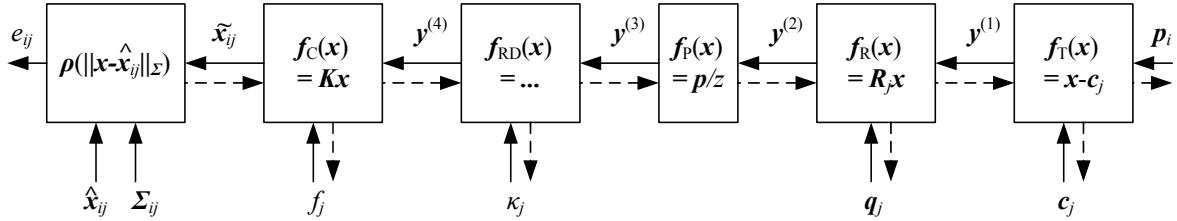


Figure 7.7 A set of chained transforms for projecting a 3D point p_i into a 2D measurement x_{ij} through a series of transformations $f^{(k)}$, each of which is controlled by its own set of parameters. The dashed lines indicate the flow of information as partial derivatives are computed during a backward pass. The formula for the radial distortion function is $f_{\text{RD}}(\mathbf{x}) = (1 + \kappa_1 r^2 + \kappa_2 r^4)\mathbf{x}$.

arate texture map from each reference camera and to blend between them during rendering, which is known as *view-dependent texture mapping* (Section 13.1.1) (Debevec, Taylor, and Malik 1996; Debevec, Yu, and Borshukov 1998).

7.4 Bundle adjustment

As we have mentioned several times before, the most accurate way to recover structure and motion is to perform robust non-linear minimization of the measurement (re-projection) errors, which is commonly known in the photogrammetry (and now computer vision) communities as *bundle adjustment*.¹³ Triggs, McLauchlan, Hartley *et al.* (1999) provide an excellent overview of this topic, including its historical development, pointers to the photogrammetry literature (Slama 1980; Atkinson 1996; Kraus 1997), and subtle issues with gauge ambiguities. The topic is also treated in depth in textbooks and surveys on multi-view geometry (Faugeras and Luong 2001; Hartley and Zisserman 2004; Moons, Van Gool, and Vergauwen 2010).

We have already introduced the elements of bundle adjustment in our discussion on iterative pose estimation (Section 6.2.2), i.e., Equations (6.42–6.48) and Figure 6.5. The biggest difference between these formulas and full bundle adjustment is that our feature location measurements \mathbf{x}_{ij} now depend not only on the point (track index) i but also on the camera pose index j ,

$$\mathbf{x}_{ij} = \mathbf{f}(\mathbf{p}_i, \mathbf{R}_j, \mathbf{c}_j, \mathbf{K}_j), \quad (7.49)$$

and that the 3D point positions \mathbf{p}_i are also being simultaneously updated. In addition, it is common to add a stage for radial distortion parameter estimation (2.78),

$$\mathbf{f}_{\text{RD}}(\mathbf{x}) = (1 + \kappa_1 r^2 + \kappa_2 r^4)\mathbf{x}, \quad (7.50)$$

if the cameras being used have not been pre-calibrated, as shown in Figure 7.7.

While most of the boxes (transforms) in Figure 7.7 have previously been explained (6.47), the leftmost box has not. This box performs a robust comparison of the predicted and mea-

¹³ The term "bundle" refers to the bundles of rays connecting camera centers to 3D points and the term "adjustment" refers to the iterative minimization of re-projection error. Alternative terms for this in the vision community include *optimal motion estimation* (Weng, Ahuja, and Huang 1993) and *non-linear least squares* (Appendix A.3) (Taylor, Kriegman, and Anandan 1991; Szeliski and Kang 1994).

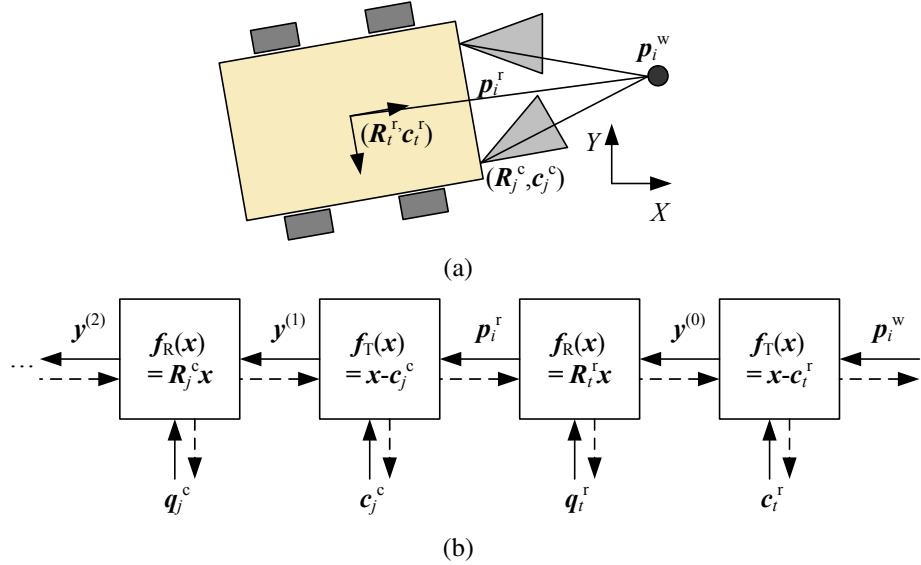


Figure 7.8 A camera rig and its associated transform chain. (a) As the mobile rig (robot) moves around in the world, its pose with respect to the world at time t is captured by $(\mathbf{R}_t^r, \mathbf{c}_t^r)$. Each camera’s pose with respect to the rig is captured by $(\mathbf{R}_j^c, \mathbf{c}_j^c)$. (b) A 3D point with world coordinates \mathbf{p}_i^w is first transformed into rig coordinates \mathbf{p}_i^r , and then through the rest of the camera-specific chain, as shown in Figure 7.7.

sured 2D locations $\hat{\mathbf{x}}_{ij}$ and $\tilde{\mathbf{x}}_{ij}$ after re-scaling by the measurement noise covariance Σ_{ij} . In more detail, this operation can be written as

$$\mathbf{r}_{ij} = \tilde{\mathbf{x}}_{ij} - \hat{\mathbf{x}}_{ij}, \quad (7.51)$$

$$s_{ij}^2 = \mathbf{r}_{ij}^T \Sigma_{ij}^{-1} \mathbf{r}_{ij}, \quad (7.52)$$

$$e_{ij} = \hat{\rho}(s_{ij}^2), \quad (7.53)$$

where $\hat{\rho}(r^2) = \rho(r)$. The corresponding Jacobians (partial derivatives) can be written as

$$\frac{\partial e_{ij}}{\partial s_{ij}^2} = \hat{\rho}'(s_{ij}^2), \quad (7.54)$$

$$\frac{\partial s_{ij}^2}{\partial \tilde{\mathbf{x}}_{ij}} = \Sigma_{ij}^{-1} \mathbf{r}_{ij}. \quad (7.55)$$

The advantage of the chained representation introduced above is that it not only makes the computations of the partial derivatives and Jacobians simpler but it can also be adapted to any camera configuration. Consider for example a pair of cameras mounted on a robot that is moving around in the world, as shown in Figure 7.8a. By replacing the rightmost two transformations in Figure 7.7 with the transformations shown in Figure 7.8b, we can simultaneously recover the position of the robot at each time and the calibration of each camera with respect to the rig, in addition to the 3D structure of the world.

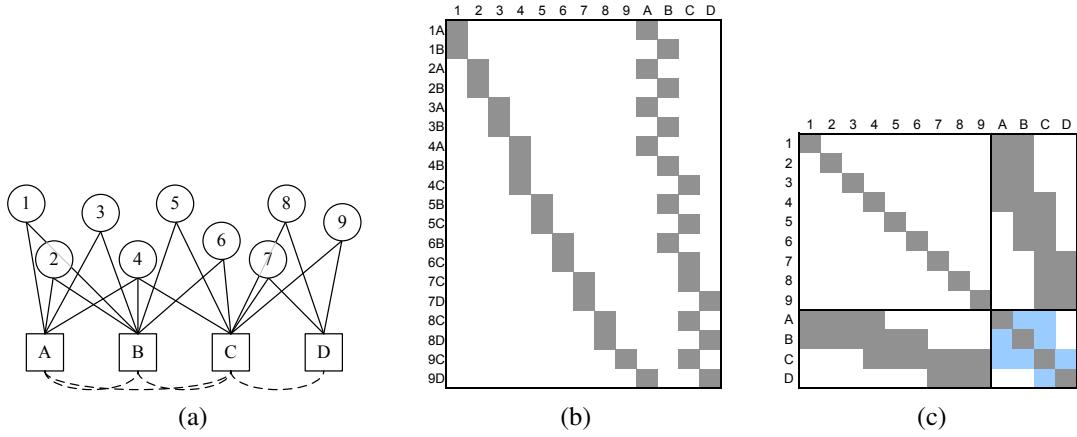


Figure 7.9 (a) Bipartite graph for a toy structure from motion problem and (b) its associated Jacobian J and (c) Hessian A . Numbers indicate 3D points and letters indicate cameras. The dashed arcs and light blue squares indicate the fill-in that occurs when the structure (point) variables are eliminated.

7.4.1 Exploiting sparsity

Large bundle adjustment problems, such as those involving reconstructing 3D scenes from thousands of Internet photographs (Snavely, Seitz, and Szeliski 2008b; Agarwal, Snavely, Simon *et al.* 2009; Agarwal, Furukawa, Snavely *et al.* 2010; Snavely, Simon, Goesele *et al.* 2010), can require solving non-linear least squares problems with millions of measurements (feature matches) and tens of thousands of unknown parameters (3D point positions and camera poses). Unless some care is taken, these kinds of problem can become intractable, since the (direct) solution of dense least squares problems is cubic in the number of unknowns.

Fortunately, structure from motion is a *bipartite* problem in structure and motion. Each feature point x_{ij} in a given image depends on one 3D point position p_i and one 3D camera pose (R_j, c_j). This is illustrated in Figure 7.9a, where each circle (1–9) indicates a 3D point, each square (A–D) indicates a camera, and lines (edges) indicate which points are visible in which cameras (2D features). If the values for all the points are known or fixed, the equations for all the cameras become independent, and vice versa.

If we order the structure variables before the motion variables in the Hessian matrix A (and hence also the right hand side vector b), we obtain a structure for the Hessian shown in Figure 7.9c.¹⁴ When such a system is solved using sparse Cholesky factorization (see Appendix A.4) (Björck 1996; Golub and Van Loan 1996), the *fill-in* occurs in the smaller motion Hessian A_{cc} (Szeliski and Kang 1994; Triggs, McLauchlan, Hartley *et al.* 1999; Hartley and Zisserman 2004; Lourakis and Argyros 2009; Engels, Stewénius, and Nistér 2006). Some recent papers by (Byröd and øAström 2009), Jeong, Nistér, Steedly *et al.* (2010) and (Agarwal, Snavely, Seitz *et al.* 2010) explore the use of iterative (conjugate gradient) techniques for the solution of bundle adjustment problems.

¹⁴ This ordering is preferable when there are fewer cameras than 3D points, which is the usual case. The exception is when we are tracking a small number of points through many video frames, in which case this ordering should be reversed.

In more detail, the *reduced* motion Hessian is computed using the *Schur complement*,

$$\mathbf{A}'_{cc} = \mathbf{A}_{cc} - \mathbf{A}_{pc}^T \mathbf{A}_{pp}^{-1} \mathbf{A}_{pc}, \quad (7.56)$$

where \mathbf{A}_{pp} is the point (structure) Hessian (the top left block of Figure 7.9c), \mathbf{A}_{pc} is the point-camera Hessian (the top right block), and \mathbf{A}_{cc} and \mathbf{A}'_{cc} are the motion Hessians before and after the point variable elimination (the bottom right block of Figure 7.9c). Notice that \mathbf{A}'_{cc} has a non-zero entry between two cameras if they see any 3D point in common. This is indicated with dashed arcs in Figure 7.9a and light blue squares in Figure 7.9c.

Whenever there are global parameters present in the reconstruction algorithm, such as camera intrinsics that are common to all of the cameras, or camera rig calibration parameters such as those shown in Figure 7.8, they should be ordered last (placed along the right and bottom edges of \mathbf{A}) in order to reduce fill-in.

Engels, Stewénius, and Nistér (2006) provide a nice recipe for sparse bundle adjustment, including all the steps needed to initialize the iterations, as well as typical computation times for a system that uses a fixed number of backward-looking frames in a real-time setting. They also recommend using homogeneous coordinates for the structure parameters p_i , which is a good idea, since it avoids numerical instabilities for points near infinity.

Bundle adjustment is now the standard method of choice for most structure-from-motion problems and is commonly applied to problems with hundreds of weakly calibrated images and tens of thousands of points, e.g., in systems such as Photosynth. (Much larger problems are commonly solved in photogrammetry and aerial imagery, but these are usually carefully calibrated and make use of surveyed ground control points.) However, as the problems become larger, it becomes impractical to re-solve full bundle adjustment problems at each iteration.

One approach to dealing with this problem is to use an incremental algorithm, where new cameras are added over time. (This makes particular sense if the data is being acquired from a video camera or moving vehicle (Nistér, Naroditsky, and Bergen 2006; Pollefeys, Nistér, Frahm *et al.* 2008).) A Kalman filter can be used to incrementally update estimates as new information is acquired. Unfortunately, such sequential updating is only statistically optimal for linear least squares problems.

For non-linear problems such as structure from motion, an extended Kalman filter, which linearizes measurement and update equations around the current estimate, needs to be used (Gelb 1974; Viéville and Faugeras 1990). To overcome this limitation, several passes can be made through the data (Azarbayejani and Pentland 1995). Because points disappear from view (and old cameras become irrelevant), a *variable state dimension filter* (VSDF) can be used to adjust the set of state variables over time, for example, by keeping only cameras and point tracks seen in the last k frames (McLauchlan 2000). A more flexible approach to using a fixed number of frames is to propagate corrections backwards through points and cameras until the changes on parameters are below a threshold (Steedly and Essa 2001). Variants of these techniques, including methods that use a fixed window for bundle adjustment (Engels, Stewénius, and Nistér 2006) or select keyframes for doing full bundle adjustment (Klein and Murray 2008) are now commonly used in real-time tracking and augmented-reality applications, as discussed in Section 7.4.2.

When maximum accuracy is required, it is still preferable to perform a full bundle adjustment over all the frames. In order to control the resulting computational complexity, one

approach is to lock together subsets of frames into locally rigid configurations and to optimize the relative positions of these cluster (Steedly, Essa, and Dellaert 2003). A different approach is to select a smaller number of frames to form a *skeletal set* that still spans the whole dataset and produces reconstructions of comparable accuracy (Snavely, Seitz, and Szeliski 2008b). We describe this latter technique in more detail in Section 7.4.4, where we discuss applications of structure from motion to large image sets.

While bundle adjustment and other robust non-linear least squares techniques are the methods of choice for most structure-from-motion problems, they suffer from initialization problems, i.e., they can get stuck in local energy minima if not started sufficiently close to the global optimum. Many systems try to mitigate this by being conservative in what reconstruction they perform early on and which cameras and points they add to the solution (Section 7.4.4). An alternative, however, is to re-formulate the problem using a norm that supports the computation of global optima.

Kahl and Hartley (2008) describe techniques for using L_∞ norms in geometric reconstruction problems. The advantage of such norms is that globally optimal solutions can be efficiently computed using second-order cone programming (SOCP). The disadvantage is that L_∞ norms are particularly sensitive to outliers and so must be combined with good outlier rejection techniques before they can be used.

7.4.2 Application: Match move and augmented reality

One of the neatest applications of structure from motion is to estimate the 3D motion of a video or film camera, along with the geometry of a 3D scene, in order to superimpose 3D graphics or computer-generated images (CGI) on the scene. In the visual effects industry, this is known as the *match move* problem (Roble 1999), since the motion of the synthetic 3D camera used to render the graphics must be *matched* to that of the real-world camera. For very small motions, or motions involving pure camera rotations, one or two tracked points can suffice to compute the necessary visual motion. For planar surfaces moving in 3D, four points are needed to compute the homography, which can then be used to insert planar overlays, e.g., to replace the contents of advertising billboards during sporting events.

The general version of this problem requires the estimation of the full 3D camera pose along with the focal length (zoom) of the lens and potentially its radial distortion parameters (Roble 1999). When the 3D structure of the scene is known ahead of time, pose estimation techniques such as *view correlation* (Bogart 1991) or *through-the-lens camera control* (Gleicher and Witkin 1992) can be used, as described in Section 6.2.3.

For more complex scenes, it is usually preferable to recover the 3D structure simultaneously with the camera motion using structure-from-motion techniques. The trick with using such techniques is that in order to prevent any visible jitter between the synthetic graphics and the actual scene, features must be tracked to very high accuracy and ample feature tracks must be available in the vicinity of the insertion location. Some of today's best known match move software packages, such as the *boujou* package from 2d3,¹⁵ which won an Emmy award in 2002, originated in structure-from-motion research in the computer vision community (Fitzgibbon and Zisserman 1998).

¹⁵ <http://www.2d3.com/>.

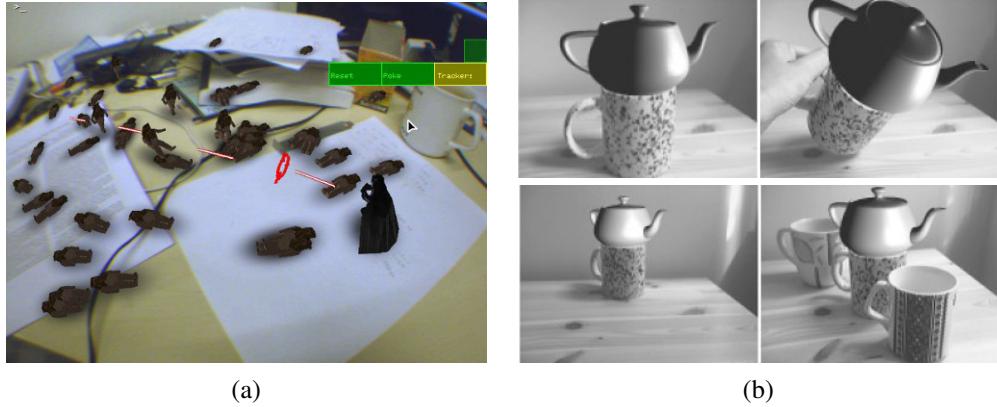


Figure 7.10 3D augmented reality: (a) Darth Vader and a horde of Ewoks battle it out on a table-top recovered using real-time, keyframe-based structure from motion (Klein and Murray 2007) © 2007 IEEE; (b) a virtual teapot is fixed to the top of a real-world coffee cup, whose pose is re-recognized at each time frame (Gordon and Lowe 2006) © 2007 Springer.

Closely related to the match move problem is robotics navigation, where a robot must estimate its location relative to its environment, while simultaneously avoiding any dangerous obstacles. This problem is often known as *simultaneous localization and mapping* (SLAM) (Thrun, Burgard, and Fox 2005) or *visual odometry* (Levin and Szeliski 2004; Nistér, Naroditsky, and Bergen 2006; Maimone, Cheng, and Matthies 2007). Early versions of such algorithms used range-sensing techniques, such as ultrasound, laser range finders, or stereo matching, to estimate local 3D geometry, which could then be fused into a 3D model. Newer techniques can perform the same task based purely on visual feature tracking, sometimes not even requiring a stereo camera rig (Davison, Reid, Molton *et al.* 2007).

Another closely related application is *augmented reality*, where 3D objects are inserted into a video feed in real time, often to annotate or help users understand a scene (Azuma, Baillot, Behringer *et al.* 2001). While traditional systems require prior knowledge about the scene or object being visually tracked (Rosten and Drummond 2005), newer systems can simultaneously build up a model of the 3D environment and then track it, so that graphics can be superimposed.

Klein and Murray (2007) describe a *parallel tracking and mapping* (PTAM) system, which simultaneously applies full bundle adjustment to keyframes selected from a video stream, while performing robust real-time pose estimation on intermediate frames. Figure 7.10a shows an example of their system in use. Once an initial 3D scene has been reconstructed, a dominant plane is estimated (in this case, the table-top) and 3D animated characters are virtually inserted. Klein and Murray (2008) extend their previous system to handle even faster camera motion by adding edge features, which can still be detected even when interest points become too blurred. They also use a direct (intensity-based) rotation estimation algorithm for even faster motions.

Instead of modeling the whole scene as one rigid reference frame, Gordon and Lowe (2006) first build a 3D model of an individual object using feature matching and structure from motion. Once the system has been initialized, for every new frame, they find the object and its pose using a 3D instance recognition algorithm, and then superimpose a graphical

object onto that model, as shown in Figure 7.10b.

While reliably tracking such objects and environments is now a well-solved problem, determining which pixels should be occluded by foreground scene elements still remains an open problem (Chuang, Agarwala, Curless *et al.* 2002; Wang and Cohen 2007a).

7.4.3 Uncertainty and ambiguities

Because structure from motion involves the estimation of so many highly coupled parameters, often with no known “ground truth” components, the estimates produced by structure from motion algorithms can often exhibit large amounts of uncertainty (Szeliski and Kang 1997). An example of this is the classic *bas-relief ambiguity*, which makes it hard to simultaneously estimate the 3D depth of a scene and the amount of camera motion (Oliensis 2005).¹⁶

As mentioned before, a unique coordinate frame and scale for a reconstructed scene cannot be recovered from monocular visual measurements alone. (When a stereo rig is used, the scale can be recovered if we know the distance (baseline) between the cameras.) This seven-degree-of-freedom *gauge ambiguity* makes it tricky to compute the covariance matrix associated with a 3D reconstruction (Triggs, McLauchlan, Hartley *et al.* 1999; Kanatani and Morris 2001). A simple way to compute a covariance matrix that ignores the gauge freedom (indeterminacy) is to throw away the seven smallest eigenvalues of the information matrix (inverse covariance), whose values are equivalent to the problem Hessian \mathbf{A} up to noise scaling (see Section 6.1.4 and Appendix B.6). After we do this, the resulting matrix can be inverted to obtain an estimate of the parameter covariance.

Szeliski and Kang (1997) use this approach to visualize the largest directions of variation in typical structure from motion problems. Not surprisingly, they find that (ignoring the gauge freedoms), the greatest uncertainties for problems such as observing an object from a small number of nearby viewpoints are in the depths of the 3D structure relative to the extent of the camera motion.¹⁷

It is also possible to estimate *local* or *marginal* uncertainties for individual parameters, which corresponds simply to taking block sub-matrices from the full covariance matrix. Under certain conditions, such as when the camera poses are relatively certain compared to 3D point locations, such uncertainty estimates can be meaningful. However, in many cases, individual uncertainty measures can mask the extent to which reconstruction errors are correlated, which is why looking at the first few modes of greatest joint variation can be helpful.

The other way in which gauge ambiguities affect structure from motion and, in particular, bundle adjustment is that they make the system Hessian matrix \mathbf{A} rank-deficient and hence impossible to invert. A number of techniques have been proposed to mitigate this problem (Triggs, McLauchlan, Hartley *et al.* 1999; Bartoli 2003). In practice, however, it appears that simply adding a small amount of the Hessian diagonal $\lambda \text{diag}(\mathbf{A})$ to the Hessian \mathbf{A} itself, as is done in the Levenberg–Marquardt non-linear least squares algorithm (Appendix A.3), usually works well.

¹⁶ Bas-relief refers to a kind of sculpture in which objects, often on ornamental friezes, are sculpted with less depth than they actually occupy. When lit from above by sunlight, they appear to have true 3D depth because of the ambiguity between relative depth and the angle of the illuminant (Section 12.1.1).

¹⁷ A good way to minimize the amount of such ambiguities is to use wide field of view cameras (Antone and Teller 2002; Levin and Szeliski 2006).



Figure 7.11 Incremental structure from motion (Snavely, Seitz, and Szeliski 2006) © 2006 ACM: Starting with an initial two-frame reconstruction of Trevi Fountain, batches of images are added using pose estimation, and their positions (along with the 3D model) are refined using bundle adjustment.

7.4.4 Application: Reconstruction from Internet photos

The most widely used application of structure from motion is in the reconstruction of 3D objects and scenes from video sequences and collections of images (Pollefeys and Van Gool 2002). The last decade has seen an explosion of techniques for performing this task automatically without the need for any manual correspondence or pre-surveyed ground control points. A lot of these techniques assume that the scene is taken with the same camera and hence the images all have the same intrinsics (Fitzgibbon and Zisserman 1998; Koch, Pollefeys, and Van Gool 2000; Schaffalitzky and Zisserman 2002; Tuytelaars and Van Gool 2004; Pollefeys, Nistér, Frahm *et al.* 2008; Moons, Van Gool, and Vergauwen 2010). Many of these techniques take the results of the sparse feature matching and structure from motion computation and then compute dense 3D surface models using multi-view stereo techniques (Section 11.6) (Koch, Pollefeys, and Van Gool 2000; Pollefeys and Van Gool 2002; Pollefeys, Nistér, Frahm *et al.* 2008; Moons, Van Gool, and Vergauwen 2010).

The latest innovation in this space has been the application of structure from motion and multi-view stereo techniques to thousands of images taken from the Internet, where very little is known about the cameras taking the photographs (Snavely, Seitz, and Szeliski 2008a). Before the structure from motion computation can begin, it is first necessary to establish sparse correspondences between different pairs of images and to then link such correspondences into *feature tracks*, which associate individual 2D image features with global 3D points. Because the $O(N^2)$ comparison of all pairs of images can be very slow, a number of techniques have been developed in the recognition community to make this process faster (Section 14.3.2) (Nistér and Stewénius 2006; Philbin, Chum, Sivic *et al.* 2008; Li, Wu, Zach *et al.* 2008; Chum, Philbin, and Zisserman 2008; Chum and Matas 2010).

To begin the reconstruction process, it is important to select a good pair of images, where there are both a large number of consistent matches (to lower the likelihood of incorrect correspondences) and a significant amount of out-of-plane parallax¹⁸ to ensure that a stable reconstruction can be obtained (Snavely, Seitz, and Szeliski 2006). The EXIF tags associated with the photographs can be used to get good initial estimates for camera focal lengths, although this is not always strictly necessary, since these parameters are re-adjusted

¹⁸ A simple way to compute this is to robustly fit a homography to the correspondences and measure reprojection errors.

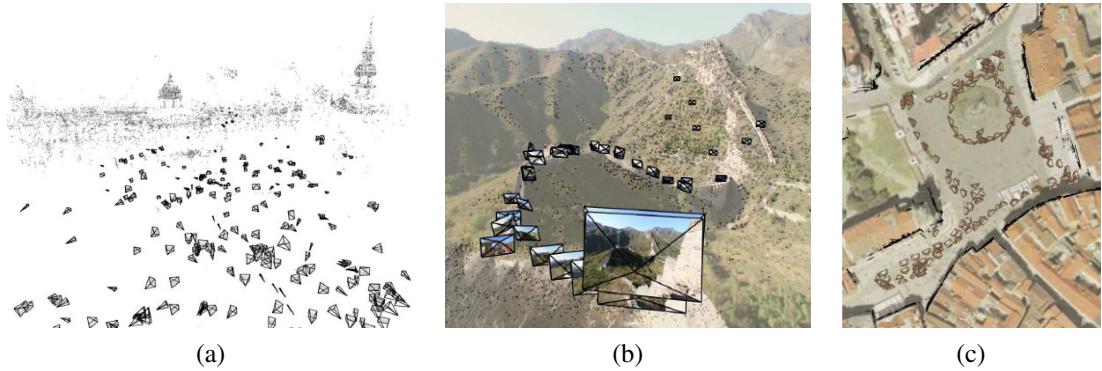


Figure 7.12 3D reconstructions produced by the incremental structure from motion algorithm developed by Snavely, Seitz, and Szeliski (2006) © 2006 ACM: (a) cameras and point cloud from Trafalgar Square; (b) cameras and points overlaid on an image from the Great Wall of China; (c) overhead view of a reconstruction of the Old Town Square in Prague registered to an aerial photograph.

as part of the bundle adjustment process.

Once an initial pair has been reconstructed, the pose of cameras that see a sufficient number of the resulting 3D points can be estimated (Section 6.2) and the complete set of cameras and feature correspondences can be used to perform another round of bundle adjustment. Figure 7.11 shows the progression of the incremental bundle adjustment algorithm, where sets of cameras are added after each successive round of bundle adjustment, while Figure 7.12 shows some additional results. An alternative to this kind of *seed and grow* approach is to first reconstruct triplets of images and then hierarchically merge triplets into larger collections, as described by Fitzgibbon and Zisserman (1998).

Unfortunately, as the incremental structure from motion algorithm continues to add more cameras and points, it can become extremely slow. The direct solution of a dense system of $O(N)$ equations for the camera pose updates can take $O(N^3)$ time; while structure from motion problems are rarely dense, scenes such as city squares have a high percentage of cameras that see points in common. Re-running the bundle adjustment algorithm after every few camera additions results in a quartic scaling of the run time with the number of images in the dataset. One approach to solving this problem is to select a smaller number of images for the original scene reconstruction and to fold in the remaining images at the very end.

Snavely, Seitz, and Szeliski (2008b) develop an algorithm for computing such a *skeletal set* of images, which is guaranteed to produce a reconstruction whose error is within a bounded factor of the optimal reconstruction accuracy. Their algorithm first evaluates all pairwise uncertainties (position covariances) between overlapping images and then chains them together to estimate a lower bound for the relative uncertainty of any distant pair. The skeletal set is constructed so that the maximal uncertainty between any pair grows by no more than a constant factor. Figure 7.13 shows an example of the skeletal set computed for 784 images of the Pantheon in Rome. As you can see, even though the skeletal set contains just a fraction of the original images, the shapes of the skeletal set and full bundle adjusted reconstructions are virtually indistinguishable.

The ability to automatically reconstruct 3D models from large, unstructured image collections has opened a wide variety of additional applications, including the ability to automati-

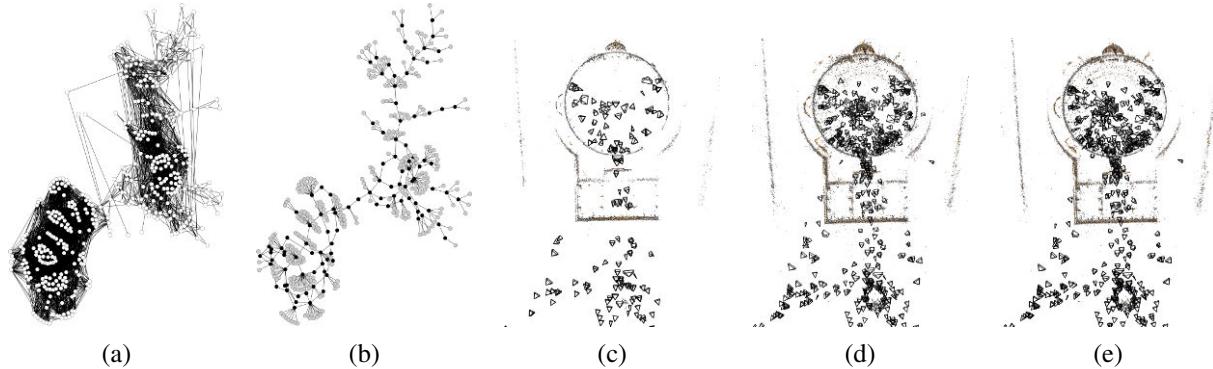


Figure 7.13 Large scale structure from motion using skeletal sets (Snavely, Seitz, and Szeliski 2008b) © 2008 IEEE: (a) original match graph for 784 images; (b) skeletal set containing 101 images; (c) top-down view of scene (Pantheon) reconstructed from the skeletal set; (d) reconstruction after adding in the remaining images using pose estimation; (e) final bundle adjusted reconstruction, which is almost identical.

ically find and label locations and regions of interest (Simon, Snavely, and Seitz 2007; Simon and Seitz 2008; Gammeter, Bossard, Quack *et al.* 2009) and to cluster large image collections so that they can be automatically labeled (Li, Wu, Zach *et al.* 2008; Quack, Leibe, and Van Gool 2008). Some of these application are discussed in more detail in Section 13.1.2.

7.5 Constrained structure and motion

The most general algorithms for structure from motion make no prior assumptions about the objects or scenes that they are reconstructing. In many cases, however, the scene contains higher-level geometric primitives, such as lines and planes. These can provide information complementary to interest points and also serve as useful building blocks for 3D modeling and visualization. Furthermore, these primitives are often arranged in particular relationships, i.e., many lines and planes are either parallel or orthogonal to each other. This is particularly true of architectural scenes and models, which we study in more detail in Section 12.6.1.

Sometimes, instead of exploiting regularity in the scene structure, it is possible to take advantage of a constrained motion model. For example, if the object of interest is rotating on a turntable (Szeliski 1991b), i.e., around a fixed but unknown axis, specialized techniques can be used to recover this motion (Fitzgibbon, Cross, and Zisserman 1998). In other situations, the camera itself may be moving in a fixed arc around some center of rotation (Shum and He 1999). Specialized capture setups, such as mobile stereo camera rigs or moving vehicles equipped with multiple fixed cameras, can also take advantage of the knowledge that individual cameras are (mostly) fixed with respect to the capture rig, as shown in Figure 7.8.¹⁹

¹⁹ Because of mechanical compliance and jitter, it may be prudent to allow for a small amount of individual camera rotation around a nominal position.

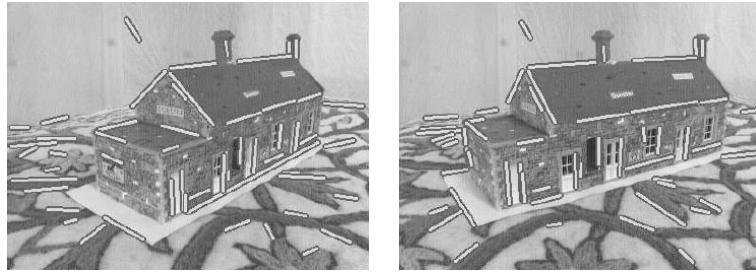


Figure 7.14 Two images of a toy house along with their matched 3D line segments (Schmid and Zisserman 1997) © 1997 Springer.

7.5.1 Line-based techniques

It is well known that pairwise epipolar geometry cannot be recovered from line matches alone, even if the cameras are calibrated. To see this, think of projecting the set of lines in each image into a set of 3D planes in space. You can move the two cameras around into any configuration you like and still obtain a valid reconstruction for 3D lines.

When lines are visible in three or more views, the trifocal tensor can be used to transfer lines from one pair of images to another (Hartley and Zisserman 2004). The trifocal tensor can also be computed on the basis of line matches alone.

Schmid and Zisserman (1997) describe a widely used technique for matching 2D lines based on the average of 15×15 pixel correlation scores evaluated at all pixels along their common line segment intersection.²⁰ In their system, the epipolar geometry is assumed to be known, e.g., computed from point matches. For wide baselines, all possible homographies corresponding to planes passing through the 3D line are used to warp pixels and the maximum correlation score is used. For triplets of images, the trifocal tensor is used to verify that the lines are in geometric correspondence before evaluating the correlations between line segments. Figure 7.14 shows the results of using their system.

Bartoli and Sturm (2003) describe a complete system for extending three view relations (trifocal tensors) computed from manual line correspondences to a full bundle adjustment of all the line and camera parameters. The key to their approach is to use the Plücker coordinates (2.12) to parameterize lines and to directly minimize reprojection errors. It is also possible to represent 3D line segments by their endpoints and to measure either the reprojection error perpendicular to the detected 2D line segments in each image or the 2D errors using an elongated uncertainty ellipse aligned with the line segment direction (Szeliski and Kang 1994).

Instead of reconstructing 3D lines, Bay, Ferrari, and Van Gool (2005) use RANSAC to group lines into likely coplanar subsets. Four lines are chosen at random to compute a homography, which is then verified for these and other plausible line segment matches by evaluating color histogram-based correlation scores. The 2D intersection points of lines belonging to the same plane are then used as virtual measurements to estimate the epipolar geometry, which is more accurate than using the homographies directly.

²⁰ Because lines often occur at depth or orientation discontinuities, it may be preferable to compute correlation scores (or to match color histograms (Bay, Ferrari, and Van Gool 2005)) separately on each side of the line.

An alternative to grouping lines into coplanar subsets is to group lines by parallelism. Whenever three or more 2D lines share a common vanishing point, there is a good likelihood that they are parallel in 3D. By finding multiple vanishing points in an image (Section 4.3.3) and establishing correspondences between such vanishing points in different images, the relative rotations between the various images (and often the camera intrinsics) can be directly estimated (Section 6.3.2).

Shum, Han, and Szeliski (1998) describe a 3D modeling system which first constructs calibrated panoramas from multiple images (Section 7.4) and then has the user draw vertical and horizontal lines in the image to demarcate the boundaries of planar regions. The lines are initially used to establish an absolute rotation for each panorama and are later used (along with the inferred vertices and planes) to infer a 3D structure, which can be recovered up to scale from one or more images (Figure 12.15).

A fully automated approach to line-based structure from motion is presented by Werner and Zisserman (2002). In their system, they first find lines and group them by common vanishing points in each image (Section 4.3.3). The vanishing points are then used to calibrate the camera, i.e., to perform a “metric upgrade” (Section 6.3.2). Lines corresponding to common vanishing points are then matched using both appearance (Schmid and Zisserman 1997) and trifocal tensors. The resulting set of 3D lines, color coded by common vanishing directions (3D orientations) is shown in Figure 12.16a. These lines are then used to infer planes and a block-structured model for the scene, as described in more detail in Section 12.6.1.

7.5.2 Plane-based techniques

In scenes that are rich in planar structures, e.g., in architecture and certain kinds of manufactured objects such as furniture, it is possible to directly estimate homographies between different planes, using either feature-based or intensity-based methods. In principle, this information can be used to simultaneously infer the camera poses and the plane equations, i.e., to compute plane-based structure from motion.

Luong and Faugeras (1996) show how a fundamental matrix can be directly computed from two or more homographies using algebraic manipulations and least squares. Unfortunately, this approach often performs poorly, since the algebraic errors do not correspond to meaningful reprojection errors (Szeliski and Torr 1998).

A better approach is to *hallucinate* virtual point correspondences within the areas from which each homography was computed and to feed them into a standard structure from motion algorithm (Szeliski and Torr 1998). An even better approach is to use full bundle adjustment with explicit plane equations, as well as additional constraints to force reconstructed co-planar features to lie exactly on their corresponding planes. (A principled way to do this is to establish a coordinate frame for each plane, e.g., at one of the feature points, and to use 2D in-plane parameterizations for the other points.) The system developed by Shum, Han, and Szeliski (1998) shows an example of such an approach, where the directions of lines and normals for planes in the scene are pre-specified by the user.

7.6 Additional reading

The topic of structure from motion is extensively covered in books and review articles on multi-view geometry (Faugeras and Luong 2001; Hartley and Zisserman 2004; Moons, Van Gool, and Vergauwen 2010). For two-frame reconstruction, Hartley (1997a) wrote a highly cited paper on the “eight-point algorithm” for computing an essential or fundamental matrix with reasonable point normalization. When the cameras are calibrated, the five-point algorithm of Nistér (2004) can be used in conjunction with RANSAC to obtain initial reconstructions from the minimum number of points. When the cameras are uncalibrated, various self-calibration techniques can be found in work by Hartley and Zisserman (2004); Moons, Van Gool, and Vergauwen (2010)—I only briefly mention one of the simplest techniques, the Kruppa equations (7.35).

In applications where points are being tracked from frame to frame, factorization techniques, based on either orthographic camera models (Tomasi and Kanade 1992; Poelman and Kanade 1997; Costeira and Kanade 1995; Morita and Kanade 1997; Morris and Kanade 1998; Anandan and Irani 2002) or projective extensions (Christy and Horaud 1996; Sturm and Triggs 1996; Triggs 1996; Oliensis and Hartley 2007), can be used.

Triggs, McLauchlan, Hartley *et al.* (1999) provide a good tutorial and survey on bundle adjustment, while Lourakis and Argyros (2009) and Engels, Stewénius, and Nistér (2006) provide tips on implementation and effective practices. Bundle adjustment is also covered in textbooks and surveys on multi-view geometry (Faugeras and Luong 2001; Hartley and Zisserman 2004; Moons, Van Gool, and Vergauwen 2010). Techniques for handling larger problems are described by Snavely, Seitz, and Szeliski (2008b); Agarwal, Snavely, Simon *et al.* (2009); Jeong, Nistér, Steedly *et al.* (2010); Agarwal, Snavely, Seitz *et al.* (2010). While bundle adjustment is often called as an inner loop inside incremental reconstruction algorithms (Snavely, Seitz, and Szeliski 2006), hierarchical (Fitzgibbon and Zisserman 1998; Farenzena, Fusello, and Gherardi 2009) and global (Rother and Carlsson 2002; Martinec and Pajdla 2007) approaches for initialization are also possible and perhaps even preferable.

As structure from motion starts being applied to dynamic scenes, the topic of non-rigid structure from motion (Torresani, Hertzmann, and Bregler 2008), which we do not cover in this book, will become more important.

7.7 Exercises

Ex 7.1: Triangulation Use the calibration pattern you built and tested in Exercise 6.7 to test your triangulation accuracy. As an alternative, generate synthetic 3D points and cameras and add noise to the 2D point measurements.

1. Assume that you know the camera pose, i.e., the camera matrices. Use the 3D distance to rays (7.4) or linearized versions of Equations (7.5–7.6) to compute an initial set of 3D locations. Compare these to your known ground truth locations.
2. Use iterative non-linear minimization to improve your initial estimates and report on the improvement in accuracy.
3. (Optional) Use the technique described by Hartley and Sturm (1997) to perform two-frame triangulation.

4. See if any of the failure modes reported by Hartley and Sturm (1997) or Hartley (1998) occur in practice.

Ex 7.2: Essential and fundamental matrix Implement the two-frame E and F matrix estimation techniques presented in Section 7.2, with suitable re-scaling for better noise immunity.

1. Use the data from Exercise 7.1 to validate your algorithms and to report on their accuracy.
2. (Optional) Implement one of the improved F or E estimation algorithms, e.g., using renormalization (Zhang 1998b; Torr and Fitzgibbon 2004; Hartley and Zisserman 2004), RANSAC (Torr and Murray 1997), least media squares (LMS), or the five-point algorithm developed by Nistér (2004).

Ex 7.3: View morphing and interpolation Implement automatic view morphing, i.e., compute two-frame structure from motion and then use these results to generate a smooth animation from one image to the next (Section 7.2.3).

1. Decide how to represent your 3D scene, e.g., compute a Delaunay triangulation of the matched point and decide what to do with the triangles near the border. (Hint: try fitting a plane to the scene, e.g., behind most of the points.)
2. Compute your in-between camera positions and orientations.
3. Warp each triangle to its new location, preferably using the correct perspective projection (Szeliski and Shum 1997).
4. (Optional) If you have a denser 3D model (e.g., from stereo), decide what to do at the “cracks”.
5. (Optional) For a non-rigid scene, e.g., two pictures of a face with different expressions, not all of your matched points will obey the epipolar geometry. Decide how to handle them to achieve the best effect.

Ex 7.4: Factorization Implement the factorization algorithm described in Section 7.3 using point tracks you computed in Exercise 4.5.

1. (Optional) Implement uncertainty rescaling (Anandan and Irani 2002) and comment on whether this improves your results.
2. (Optional) Implement one of the perspective improvements to factorization discussed in Section 7.3.1 (Christy and Horaud 1996; Sturm and Triggs 1996; Triggs 1996). Does this produce significantly lower reprojection errors? Can you upgrade this reconstruction to a metric one?

Ex 7.5: Bundle adjuster Implement a full bundle adjuster. This may sound daunting, but it really is not.

1. Devise the internal data structures and external file representations to hold your camera parameters (position, orientation, and focal length), 3D point locations (Euclidean or homogeneous), and 2D point tracks (frame and point identifier as well as 2D locations).

2. Use some other technique, such as factorization, to initialize the 3D point and camera locations from your 2D tracks (e.g., a subset of points that appears in all frames).
3. Implement the code corresponding to the forward transformations in Figure 7.7, i.e., for each 2D point measurement, take the corresponding 3D point, map it through the camera transformations (including perspective projection and focal length scaling), and compare it to the 2D point measurement to get a residual error.
4. Take the residual error and compute its derivatives with respect to all the unknown motion and structure parameters, using backward chaining, as shown, e.g., in Figure 7.7 and Equation (6.47). This gives you the sparse Jacobian \mathbf{J} used in Equations (6.13–6.17) and Equation (6.43).
5. Use a sparse least squares or linear system solver, e.g., MATLAB, SparseSuite, or SPARSKIT (see Appendix A.4 and A.5), to solve the corresponding linearized system, adding a small amount of diagonal preconditioning, as in Levenberg–Marquardt.
6. Update your parameters, make sure your rotation matrices are still orthonormal (e.g., by re-computing them from your quaternions), and continue iterating while monitoring your residual error.
7. (Optional) Use the “Schur complement trick” (7.56) to reduce the size of the system being solved (Triggs, McLauchlan, Hartley *et al.* 1999; Hartley and Zisserman 2004; Lourakis and Argyros 2009; Engels, Stewénius, and Nistér 2006).
8. (Optional) Implement your own iterative sparse solver, e.g., conjugate gradient, and compare its performance to a direct method.
9. (Optional) Make your bundle adjuster robust to outliers, or try adding some of the other improvements discussed in (Engels, Stewénius, and Nistér 2006). Can you think of any other ways to make your algorithm even faster or more robust?

Ex 7.6: Match move and augmented reality Use the results of the previous exercise to superimpose a rendered 3D model on top of video. See Section 7.4.2 for more details and ideas. Check for how “locked down” the objects are.

Ex 7.7: Line-based reconstruction Augment the previously developed bundle adjuster to include lines, possibly with known 3D orientations.

Optionally, use co-planar sets of points and lines to hypothesize planes and to enforce co-planarity (Schaffalitzky and Zisserman 2002; Robertson and Cipolla 2002)

Ex 7.8: Flexible bundle adjuster Design a bundle adjuster that allows for arbitrary chains of transformations and prior knowledge about the unknowns, as suggested in Figures 7.7–7.8.

Ex 7.9: Unordered image matching Compute the camera pose and 3D structure of a scene from an arbitrary collection of photographs (Brown and Lowe 2003; Snavely, Seitz, and Szeliski 2006).

Chapter 8

Dense motion estimation

8.1	Translational alignment	337
8.1.1	Hierarchical motion estimation	341
8.1.2	Fourier-based alignment	341
8.1.3	Incremental refinement	345
8.2	Parametric motion	350
8.2.1	<i>Application:</i> Video stabilization	354
8.2.2	Learned motion models	354
8.3	Spline-based motion	355
8.3.1	<i>Application:</i> Medical image registration	358
8.4	Optical flow	360
8.4.1	Multi-frame motion estimation	363
8.4.2	<i>Application:</i> Video denoising	364
8.4.3	<i>Application:</i> De-interlacing	364
8.5	Layered motion	365
8.5.1	<i>Application:</i> Frame interpolation	368
8.5.2	Transparent layers and reflections	368
8.6	Additional reading	370
8.7	Exercises	371

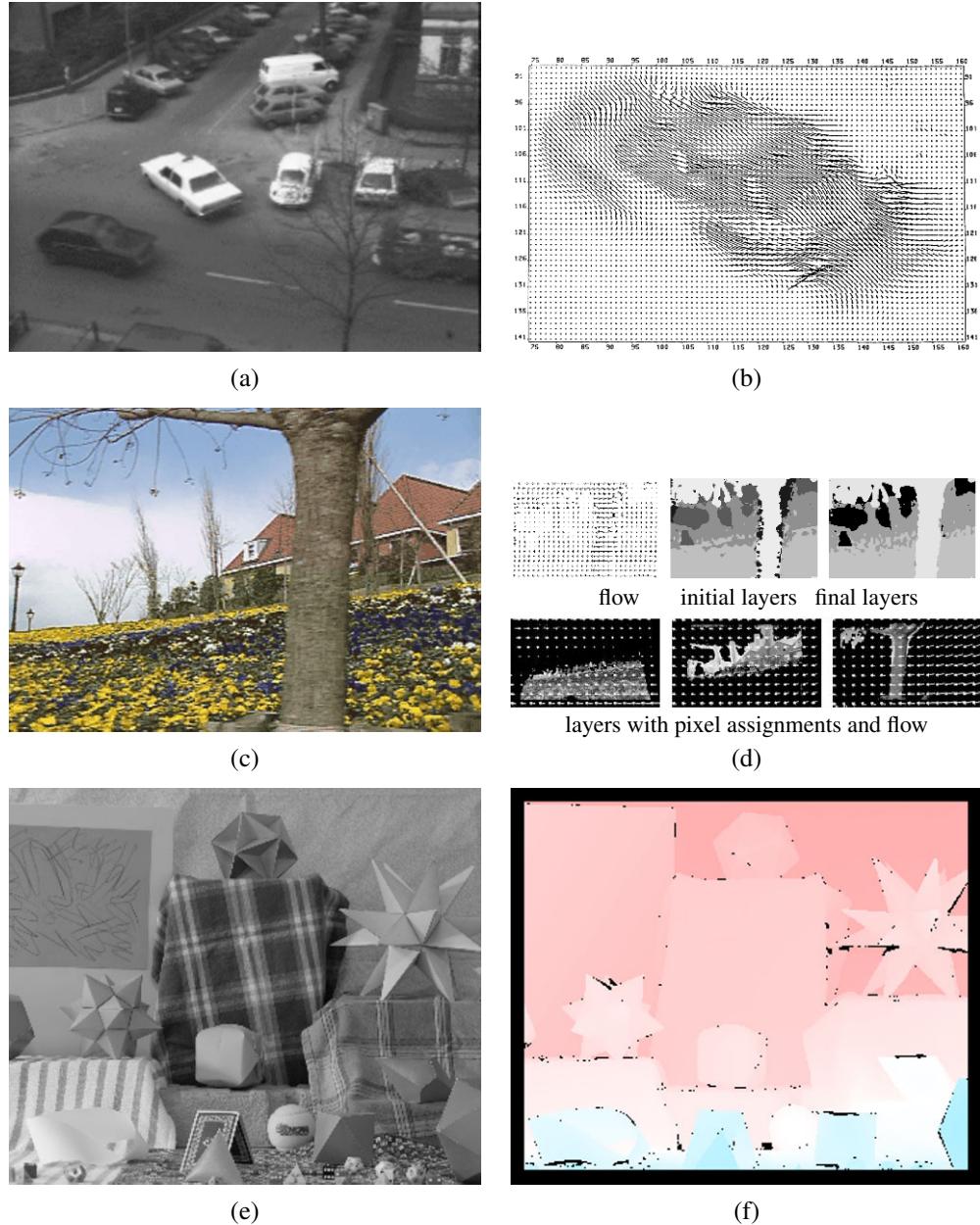


Figure 8.1 Motion estimation: (a–b) regularization-based optical flow (Nagel and Enkelmann 1986) © 1986 IEEE; (c–d) layered motion estimation (Wang and Adelson 1994) © 1994 IEEE; (e–f) sample image and ground truth flow from evaluation database (Baker, Black, Lewis *et al.* 2007) © 2007 IEEE.

Algorithms for aligning images and estimating motion in video sequences are among the most widely used in computer vision. For example, frame-rate image alignment is widely used in camcorders and digital cameras to implement their image stabilization (IS) feature.

An early example of a widely used image registration algorithm is the patch-based translational alignment (optical flow) technique developed by Lucas and Kanade (1981). Variants of this algorithm are used in almost all motion-compensated video compression schemes such as MPEG and H.263 (Le Gall 1991). Similar parametric motion estimation algorithms have found a wide variety of applications, including video summarization (Teodosio and Bender 1993; Irani and Anandan 1998), video stabilization (Hansen, Anandan, Dana *et al.* 1994; Srinivasan, Chellappa, Veeraraghavan *et al.* 2005; Matsushita, Ofek, Ge *et al.* 2006), and video compression (Irani, Hsu, and Anandan 1995; Lee, ge Chen, lung Bruce Lin *et al.* 1997). More sophisticated image registration algorithms have also been developed for medical imaging and remote sensing. Image registration techniques are surveyed by Brown (1992), Zitov'a and Flusser (2003), Goshtasby (2005), and Szeliski (2006a).

To estimate the motion between two or more images, a suitable *error metric* must first be chosen to compare the images (Section 8.1). Once this has been established, a suitable *search* technique must be devised. The simplest technique is to exhaustively try all possible alignments, i.e., to do a *full search*. In practice, this may be too slow, so *hierarchical coarse-to-fine* techniques (Section 8.1.1) based on image pyramids are normally used. Alternatively, Fourier transforms (Section 8.1.2) can be used to speed up the computation.

To get sub-pixel precision in the alignment, *incremental* methods (Section 8.1.3) based on a Taylor series expansion of the image function are often used. These can also be applied to *parametric motion models* (Section 8.2), which model global image transformations such as rotation or shearing. Motion estimation can be made more reliable by *learning* the typical dynamics or motion statistics of the scenes or objects being tracked, e.g., the natural gait of walking people (Section 8.2.2). For more complex motions, piecewise parametric *spline motion models* (Section 8.3) can be used. In the presence of multiple independent (and perhaps non-rigid) motions, general-purpose *optical flow* (or *optic flow*) techniques need to be used (Section 8.4). For even more complex motions that include a lot of occlusions, *layered motion models* (Section 8.5), which decompose the scene into coherently moving layers, can work well.

In this chapter, we describe each of these techniques in more detail. Additional details can be found in review and comparative evaluation papers on motion estimation (Barron, Fleet, and Beauchemin 1994; Mitiche and Boutheny 1996; Stiller and Konrad 1999; Szeliski 2006a; Baker, Black, Lewis *et al.* 2007).

8.1 Translational alignment

The simplest way to establish an alignment between two images or image patches is to shift one image relative to the other. Given a *template* image $I_0(\mathbf{x})$ sampled at discrete pixel locations $\{\mathbf{x}_i = (x_i, y_i)\}$, we wish to find where it is located in image $I_1(\mathbf{x})$. A least squares solution to this problem is to find the minimum of the *sum of squared differences* (SSD) function

$$E_{\text{SSD}}(\mathbf{u}) = \sum_i [I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)]^2 = \sum_i e_i^2, \quad (8.1)$$

where $\mathbf{u} = (u, v)$ is the *displacement* and $e_i = I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)$ is called the *residual error* (or the *displaced frame difference* in the video coding literature).¹ (We ignore for the moment the possibility that parts of I_0 may lie outside the boundaries of I_1 or be otherwise not visible.) The assumption that corresponding pixel values remain the same in the two images is often called the *brightness constancy constraint*.²

In general, the displacement \mathbf{u} can be fractional, so a suitable interpolation function must be applied to image $I_1(\mathbf{x})$. In practice, a bilinear interpolant is often used but bicubic interpolation can yield slightly better results (Szeliski and Scharstein 2004). Color images can be processed by summing differences across all three color channels, although it is also possible to first transform the images into a different color space or to only use the luminance (which is often done in video encoders).

Robust error metrics. We can make the above error metric more robust to outliers by replacing the squared error terms with a robust function $\rho(e_i)$ (Huber 1981; Hampel, Ronchetti, Rousseeuw *et al.* 1986; Black and Anandan 1996; Stewart 1999) to obtain

$$E_{\text{SRD}}(\mathbf{u}) = \sum_i \rho(I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)) = \sum_i \rho(e_i). \quad (8.2)$$

The robust norm $\rho(e)$ is a function that grows less quickly than the quadratic penalty associated with least squares. One such function, sometimes used in motion estimation for video coding because of its speed, is the *sum of absolute differences* (SAD) metric³ or L_1 norm, i.e.,

$$E_{\text{SAD}}(\mathbf{u}) = \sum_i |I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)| = \sum_i |e_i|. \quad (8.3)$$

However, since this function is not differentiable at the origin, it is not well suited to gradient-descent approaches such as the ones presented in Section 8.1.3.

Instead, a smoothly varying function that is quadratic for small values but grows more slowly away from the origin is often used. Black and Rangarajan (1996) discuss a variety of such functions, including the *Geman–McClure* function,

$$\rho_{\text{GM}}(x) = \frac{x^2}{1 + x^2/a^2}, \quad (8.4)$$

where a is a constant that can be thought of as an *outlier threshold*. An appropriate value for the threshold can itself be derived using robust statistics (Huber 1981; Hampel, Ronchetti, Rousseeuw *et al.* 1986; Rousseeuw and Leroy 1987), e.g., by computing the *median absolute deviation*, $MAD = \text{med}_i |e_i|$, and multiplying it by 1.4 to obtain a robust estimate of the standard deviation of the inlier noise process (Stewart 1999).

¹ The usual justification for using least squares is that it is the optimal estimate with respect to Gaussian noise. See the discussion below on robust error metrics as well as Appendix B.3.

² Brightness constancy (Horn 1974) is the tendency for objects to maintain their perceived brightness under varying illumination conditions.

³ In video compression, e.g., the H.264 standard (<http://www.itu.int/rec/T-REC-H.264>), the sum of absolute transformed differences (SATD), which measures the differences in a frequency transform space, e.g., using a Hadamard transform, is often used since it more accurately predicts quality (Richardson 2003).

Spatially varying weights. The error metrics above ignore the fact that for a given alignment, some of the pixels being compared may lie outside the original image boundaries. Furthermore, we may want to partially or completely downweight the contributions of certain pixels. For example, we may want to selectively “erase” some parts of an image from consideration when stitching a mosaic where unwanted foreground objects have been cut out. For applications such as background stabilization, we may want to downweight the middle part of the image, which often contains independently moving objects being tracked by the camera.

All of these tasks can be accomplished by associating a spatially varying per-pixel weight value with each of the two images being matched. The error metric then becomes the weighted (or *windowed*) SSD function,

$$E_{\text{WSSD}}(\mathbf{u}) = \sum_i w_0(\mathbf{x}_i)w_1(\mathbf{x}_i + \mathbf{u})[I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)]^2, \quad (8.5)$$

where the weighting functions w_0 and w_1 are zero outside the image boundaries.

If a large range of potential motions is allowed, the above metric can have a bias towards smaller overlap solutions. To counteract this bias, the windowed SSD score can be divided by the overlap area

$$A = \sum_i w_0(\mathbf{x}_i)w_1(\mathbf{x}_i + \mathbf{u}) \quad (8.6)$$

to compute a *per-pixel* (or mean) squared pixel error E_{WSSD}/A . The square root of this quantity is the *root mean square* intensity error

$$RMS = \sqrt{E_{\text{WSSD}}/A} \quad (8.7)$$

often reported in comparative studies.

Bias and gain (exposure differences). Often, the two images being aligned were not taken with the same exposure. A simple model of linear (affine) intensity variation between the two images is the *bias and gain* model,

$$I_1(\mathbf{x} + \mathbf{u}) = (1 + \alpha)I_0(\mathbf{x}) + \beta, \quad (8.8)$$

where β is the *bias* and α is the *gain* (Lucas and Kanade 1981; Gennert 1988; Fuh and Maragos 1991; Baker, Gross, and Matthews 2003; Evangelidis and Psarakis 2008). The least squares formulation then becomes

$$E_{\text{BG}}(\mathbf{u}) = \sum_i [I_1(\mathbf{x}_i + \mathbf{u}) - (1 + \alpha)I_0(\mathbf{x}_i) - \beta]^2 = \sum_i [\alpha I_0(\mathbf{x}_i) + \beta - e_i]^2. \quad (8.9)$$

Rather than taking a simple squared difference between corresponding patches, it becomes necessary to perform a *linear regression* (Appendix A.2), which is somewhat more costly. Note that for color images, it may be necessary to estimate a different bias and gain for each color channel to compensate for the automatic *color correction* performed by some digital cameras (Section 2.3.2). Bias and gain compensation is also used in video codecs, where it is known as *weighted prediction* (Richardson 2003).

A more general (spatially varying, non-parametric) model of intensity variation, which is computed as part of the registration process, is used in (Negahdaripour 1998; Jia and Tang

2003; Seitz and Baker 2009). This can be useful for dealing with local variations such as the *vignetting* caused by wide-angle lenses, wide apertures, or lens housings. It is also possible to pre-process the images before comparing their values, e.g., using band-pass filtered images (Anandan 1989; Bergen, Anandan, Hanna *et al.* 1992), gradients (Scharstein 1994; Papenberg, Bruhn, Brox *et al.* 2006), or using other local transformations such as histograms or rank transforms (Cox, Roy, and Hingorani 1995; Zabih and Woodfill 1994), or to maximize *mutual information* (Viola and Wells III 1997; Kim, Kolmogorov, and Zabih 2003). Hirschmüller and Scharstein (2009) compare a number of these approaches and report on their relative performance in scenes with exposure differences.

Correlation. An alternative to taking intensity differences is to perform *correlation*, i.e., to maximize the *product* (or *cross-correlation*) of the two aligned images,

$$E_{\text{CC}}(\mathbf{u}) = \sum_i I_0(\mathbf{x}_i) I_1(\mathbf{x}_i + \mathbf{u}). \quad (8.10)$$

At first glance, this may appear to make bias and gain modeling unnecessary, since the images will prefer to line up regardless of their relative scales and offsets. However, this is actually not true. If a very bright patch exists in $I_1(\mathbf{x})$, the maximum product may actually lie in that area.

For this reason, *normalized cross-correlation* is more commonly used,

$$E_{\text{NCC}}(\mathbf{u}) = \frac{\sum_i [I_0(\mathbf{x}_i) - \bar{I}_0] [I_1(\mathbf{x}_i + \mathbf{u}) - \bar{I}_1]}{\sqrt{\sum_i [I_0(\mathbf{x}_i) - \bar{I}_0]^2} \sqrt{\sum_i [I_1(\mathbf{x}_i + \mathbf{u}) - \bar{I}_1]^2}}, \quad (8.11)$$

where

$$\bar{I}_0 = \frac{1}{N} \sum_i I_0(\mathbf{x}_i) \quad \text{and} \quad (8.12)$$

$$\bar{I}_1 = \frac{1}{N} \sum_i I_1(\mathbf{x}_i + \mathbf{u}) \quad (8.13)$$

are the *mean images* of the corresponding patches and N is the number of pixels in the patch. The normalized cross-correlation score is always guaranteed to be in the range $[-1, 1]$, which makes it easier to handle in some higher-level applications, such as deciding which patches truly match. Normalized correlation works well when matching images taken with different exposures, e.g., when creating high dynamic range images (Section 10.2). Note, however, that the NCC score is undefined if either of the two patches has zero variance (and, in fact, its performance degrades for noisy low-contrast regions).

A variant on NCC, which is related to the bias–gain regression implicit in the matching score (8.9), is the *normalized SSD* score

$$E_{\text{NSSD}}(\mathbf{u}) = \frac{1}{2} \frac{\sum_i [(I_0(\mathbf{x}_i) - \bar{I}_0) - (I_1(\mathbf{x}_i + \mathbf{u}) - \bar{I}_1)]^2}{\sqrt{\sum_i [I_0(\mathbf{x}_i) - \bar{I}_0]^2 + [I_1(\mathbf{x}_i + \mathbf{u}) - \bar{I}_1]^2}} \quad (8.14)$$

recently proposed by Criminisi, Shotton, Blake *et al.* (2007). In their experiments, they find that it produces comparable results to NCC, but is more efficient when applied to a large number of overlapping patches using a moving average technique (Section 3.2.2).

8.1.1 Hierarchical motion estimation

Now that we have a well-defined alignment cost function to optimize, how can we find its minimum? The simplest solution is to do a *full search* over some range of shifts, using either integer or sub-pixel steps. This is often the approach used for *block matching* in *motion compensated video compression*, where a range of possible motions (say, ± 16 pixels) is explored.⁴

To accelerate this search process, *hierarchical motion estimation* is often used: an image pyramid (Section 3.5) is constructed and a search over a smaller number of discrete pixels (corresponding to the same range of motion) is first performed at coarser levels (Quam 1984; Anandan 1989; Bergen, Anandan, Hanna *et al.* 1992). The motion estimate from one level of the pyramid is then used to initialize a smaller *local* search at the next finer level. Alternatively, several seeds (good solutions) from the coarse level can be used to initialize the fine-level search. While this is not guaranteed to produce the same result as a full search, it usually works almost as well and is much faster.

More formally, let

$$I_k^{(l)}(\mathbf{x}_j) \leftarrow \tilde{I}_k^{(l-1)}(2\mathbf{x}_j) \quad (8.15)$$

be the *decimated* image at level l obtained by subsampling (*downsampling*) a smoothed version of the image at level $l-1$. See Section 3.5 for how to perform the required downsampling (pyramid construction) without introducing too much aliasing.

At the coarsest level, we search for the best displacement $\mathbf{u}^{(l)}$ that minimizes the difference between images $I_0^{(l)}$ and $I_1^{(l)}$. This is usually done using a full search over some range of displacements $\mathbf{u}^{(l)} \in 2^{-l}[-S, S]^2$, where S is the desired *search range* at the finest (original) resolution level, optionally followed by the incremental refinement step described in Section 8.1.3.

Once a suitable motion vector has been estimated, it is used to *predict* a likely displacement

$$\hat{\mathbf{u}}^{(l-1)} \leftarrow 2\mathbf{u}^{(l)} \quad (8.16)$$

for the next finer level.⁵ The search over displacements is then repeated at the finer level over a much narrower range of displacements, say $\hat{\mathbf{u}}^{(l-1)} \pm 1$, again optionally combined with an incremental refinement step (Anandan 1989). Alternatively, one of the images can be *warped* (resampled) by the current motion estimate, in which case only small incremental motions need to be computed at the finer level. A nice description of the whole process, extended to parametric motion estimation (Section 8.2), is provided by Bergen, Anandan, Hanna *et al.* (1992).

8.1.2 Fourier-based alignment

When the search range corresponds to a significant fraction of the larger image (as is the case in image stitching, see Chapter 9), the hierarchical approach may not work that well, since

⁴ In stereo matching (Section 11.1.2), an explicit search over all possible disparities (i.e., a *plane sweep*) is almost always performed, since the number of search hypotheses is much smaller due to the 1D nature of the potential displacements.

⁵ This doubling of displacements is only necessary if displacements are defined in integer *pixel* coordinates, which is the usual case in the literature (Bergen, Anandan, Hanna *et al.* 1992). If *normalized device coordinates* (Section 2.1.5) are used instead, the displacements (and search ranges) need not change from level to level, although the step sizes will need to be adjusted, to keep search steps of roughly one pixel.

it is often not possible to coarsen the representation too much before significant features are blurred away. In this case, a Fourier-based approach may be preferable.

Fourier-based alignment relies on the fact that the Fourier transform of a shifted signal has the same magnitude as the original signal but a linearly varying phase (Section 3.4), i.e.,

$$\mathcal{F}\{I_1(\mathbf{x} + \mathbf{u})\} = \mathcal{F}\{I_1(\mathbf{x})\} e^{-ju \cdot \boldsymbol{\omega}} = \mathcal{I}_1(\boldsymbol{\omega}) e^{-ju \cdot \boldsymbol{\omega}}, \quad (8.17)$$

where $\boldsymbol{\omega}$ is the vector-valued angular frequency of the Fourier transform and we use calligraphic notation $\mathcal{I}_1(\boldsymbol{\omega}) = \mathcal{F}\{I_1(\mathbf{x})\}$ to denote the Fourier transform of a signal (Section 3.4).

Another useful property of Fourier transforms is that convolution in the spatial domain corresponds to multiplication in the Fourier domain (Section 3.4).⁶ Thus, the Fourier transform of the cross-correlation function E_{CC} can be written as

$$\mathcal{F}\{E_{CC}(\mathbf{u})\} = \mathcal{F}\left\{\sum_i I_0(\mathbf{x}_i) I_1(\mathbf{x}_i + \mathbf{u})\right\} = \mathcal{F}\{I_0(\mathbf{u}) \bar{*} I_1(\mathbf{u})\} = \mathcal{I}_0(\boldsymbol{\omega}) \mathcal{I}_1^*(\boldsymbol{\omega}), \quad (8.18)$$

where

$$f(\mathbf{u}) \bar{*} g(\mathbf{u}) = \sum_i f(\mathbf{x}_i) g(\mathbf{x}_i + \mathbf{u}) \quad (8.19)$$

is the *correlation* function, i.e., the convolution of one signal with the reverse of the other, and $\mathcal{I}_1^*(\boldsymbol{\omega})$ is the *complex conjugate* of $\mathcal{I}_1(\boldsymbol{\omega})$. This is because convolution is defined as the summation of one signal with the reverse of the other (Section 3.4).

Thus, to efficiently evaluate E_{CC} over the range of all possible values of \mathbf{u} , we take the Fourier transforms of both images $I_0(\mathbf{x})$ and $I_1(\mathbf{x})$, multiply both transforms together (after conjugating the second one), and take the inverse transform of the result. The Fast Fourier Transform algorithm can compute the transform of an $N \times M$ image in $O(NM \log NM)$ operations (Bracewell 1986). This can be significantly faster than the $O(N^2 M^2)$ operations required to do a full search when the full range of image overlaps is considered.

While Fourier-based convolution is often used to accelerate the computation of image correlations, it can also be used to accelerate the sum of squared differences function (and its variants). Consider the SSD formula given in (8.1). Its Fourier transform can be written as

$$\begin{aligned} \mathcal{F}\{E_{SSD}(\mathbf{u})\} &= \mathcal{F}\left\{\sum_i [I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)]^2\right\} \\ &= \delta(\boldsymbol{\omega}) \sum_i [I_0^2(\mathbf{x}_i) + I_1^2(\mathbf{x}_i)] - 2\mathcal{I}_0(\boldsymbol{\omega}) \mathcal{I}_1^*(\boldsymbol{\omega}). \end{aligned} \quad (8.20)$$

Thus, the SSD function can be computed by taking twice the correlation function and subtracting it from the sum of the energies in the two images.

Windowed correlation. Unfortunately, the Fourier convolution theorem only applies when the summation over \mathbf{x}_i is performed over *all* the pixels in both images, using a circular shift of the image when accessing pixels outside the original boundaries. While this is

⁶ In fact, the Fourier shift property (8.17) derives from the convolution theorem by observing that shifting is equivalent to convolution with a displaced delta function $\delta(\mathbf{x} - \mathbf{u})$.

acceptable for small shifts and comparably sized images, it makes no sense when the images overlap by a small amount or one image is a small subset of the other.

In that case, the cross-correlation function should be replaced with a *windowed* (weighted) cross-correlation function,

$$E_{\text{WCC}}(\mathbf{u}) = \sum_i w_0(\mathbf{x}_i) I_0(\mathbf{x}_i) w_1(\mathbf{x}_i + \mathbf{u}) I_1(\mathbf{x}_i + \mathbf{u}), \quad (8.21)$$

$$= [w_0(\mathbf{x}) I_0(\mathbf{x})] \bar{*} [w_1(\mathbf{x}) I_1(\mathbf{x})] \quad (8.22)$$

where the weighting functions w_0 and w_1 are zero outside the valid ranges of the images and both images are padded so that circular shifts return 0 values outside the original image boundaries.

An even more interesting case is the computation of the *weighted* SSD function introduced in Equation (8.5),

$$E_{\text{WSSD}}(\mathbf{u}) = \sum_i w_0(\mathbf{x}_i) w_1(\mathbf{x}_i + \mathbf{u}) [I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)]^2. \quad (8.23)$$

Expanding this as a sum of correlations and deriving the appropriate set of Fourier transforms is left for Exercise 8.1.

The same kind of derivation can also be applied to the bias–gain corrected sum of squared difference function E_{BG} (8.9). Again, Fourier transforms can be used to efficiently compute all the correlations needed to perform the linear regression in the bias and gain parameters in order to estimate the exposure-compensated difference for each potential shift (Exercise 8.1).

Phase correlation. A variant of regular correlation (8.18) that is sometimes used for motion estimation is *phase correlation* (Kuglin and Hines 1975; Brown 1992). Here, the spectrum of the two signals being matched is *whitened* by dividing each per-frequency product in (8.18) by the magnitudes of the Fourier transforms,

$$\mathcal{F}\{E_{\text{PC}}(\mathbf{u})\} = \frac{\mathcal{I}_0(\omega)\mathcal{I}_1^*(\omega)}{\|\mathcal{I}_0(\omega)\| \|\mathcal{I}_1(\omega)\|} \quad (8.24)$$

before taking the final inverse Fourier transform. In the case of noiseless signals with perfect (cyclic) shift, we have $I_1(\mathbf{x} + \mathbf{u}) = I_0(\mathbf{x})$ and hence, from Equation (8.17), we obtain

$$\begin{aligned} \mathcal{F}\{I_1(\mathbf{x} + \mathbf{u})\} &= \mathcal{I}_1(\omega)e^{-2\pi j u \cdot \omega} = \mathcal{I}_0(\omega) \text{ and} \\ \mathcal{F}\{E_{\text{PC}}(\mathbf{u})\} &= e^{-2\pi j u \cdot \omega}. \end{aligned} \quad (8.25)$$

The output of phase correlation (under ideal conditions) is therefore a single spike (impulse) located at the correct value of \mathbf{u} , which (in principle) makes it easier to find the correct estimate.

Phase correlation has a reputation in some quarters of outperforming regular correlation, but this behavior depends on the characteristics of the signals and noise. If the original images are contaminated by noise in a narrow frequency band (e.g., low-frequency noise or peaked frequency “hum”), the whitening process effectively de-emphasizes the noise in these regions. However, if the original signals have very low signal-to-noise ratio at some frequencies (say, two blurry or low-textured images with lots of high-frequency noise), the whitening process can actually decrease performance (see Exercise 8.1).

Recently, gradient cross-correlation has emerged as a promising alternative to phase correlation (Argyriou and Vlachos 2003), although further systematic studies are probably warranted. Phase correlation has also been studied by Fleet and Jepson (1990) as a method for estimating general optical flow and stereo disparity.

Rotations and scale. While Fourier-based alignment is mostly used to estimate translational shifts between images, it can, under certain limited conditions, also be used to estimate in-plane rotations and scales. Consider two images that are related *purely* by rotation, i.e.,

$$I_1(\hat{\mathbf{R}}\mathbf{x}) = I_0(\mathbf{x}). \quad (8.26)$$

If we re-sample the images into *polar coordinates*,

$$\tilde{I}_0(r, \theta) = I_0(r \cos \theta, r \sin \theta) \text{ and } \tilde{I}_1(r, \theta) = I_1(r \cos \theta, r \sin \theta), \quad (8.27)$$

we obtain

$$\tilde{I}_1(r, \theta + \hat{\theta}) = \tilde{I}_0(r, \theta). \quad (8.28)$$

The desired rotation can then be estimated using a Fast Fourier Transform (FFT) shift-based technique.

If the two images are also related by a scale,

$$I_1(e^{\hat{s}} \hat{\mathbf{R}}\mathbf{x}) = I_0(\mathbf{x}), \quad (8.29)$$

we can re-sample into *log-polar coordinates*,

$$\tilde{I}_0(s, \theta) = I_0(e^s \cos \theta, e^s \sin \theta) \text{ and } \tilde{I}_1(s, \theta) = I_1(e^s \cos \theta, e^s \sin \theta), \quad (8.30)$$

to obtain

$$\tilde{I}_1(s + \hat{s}, \theta + \hat{\theta}) = I_0(s, \theta). \quad (8.31)$$

In this case, care must be taken to choose a suitable range of s values that reasonably samples the original image.

For images that are also translated by a small amount,

$$I_1(e^{\hat{s}} \hat{\mathbf{R}}\mathbf{x} + \mathbf{t}) = I_0(\mathbf{x}), \quad (8.32)$$

De Castro and Morandi (1987) propose an ingenious solution that uses several steps to estimate the unknown parameters. First, both images are converted to the Fourier domain and only the magnitudes of the transformed images are retained. In principle, the Fourier magnitude images are insensitive to translations in the image plane (although the usual caveats about border effects apply). Next, the two magnitude images are aligned in rotation and scale using the polar or log-polar representations. Once rotation and scale are estimated, one of the images can be de-rotated and scaled and a regular translational algorithm can be applied to estimate the translational shift.

Unfortunately, this trick only applies when the images have large overlap (small translational motion). For more general motion of patches or images, the parametric motion estimator described in Section 8.2 or the feature-based approaches described in Section 6.1 need to be used.

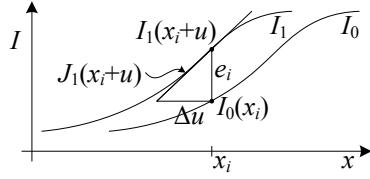


Figure 8.2 Taylor series approximation of a function and the incremental computation of the optical flow correction amount. $\mathbf{J}_1(\mathbf{x}_i + \mathbf{u})$ is the image gradient at $(\mathbf{x}_i + \mathbf{u})$ and e_i is the current intensity difference.

8.1.3 Incremental refinement

The techniques described up till now can estimate alignment to the nearest pixel (or potentially fractional pixel if smaller search steps are used). In general, image stabilization and stitching applications require much higher accuracies to obtain acceptable results.

To obtain better *sub-pixel* estimates, we can use one of several techniques described by Tian and Huhns (1986). One possibility is to evaluate several discrete (integer or fractional) values of (u, v) around the best value found so far and to *interpolate* the matching score to find an analytic minimum.

A more commonly used approach, first proposed by Lucas and Kanade (1981), is to perform *gradient descent* on the SSD energy function (8.1), using a Taylor series expansion of the image function (Figure 8.2),

$$E_{\text{LK-SSD}}(\mathbf{u} + \Delta\mathbf{u}) = \sum_i [I_1(\mathbf{x}_i + \mathbf{u} + \Delta\mathbf{u}) - I_0(\mathbf{x}_i)]^2 \quad (8.33)$$

$$\approx \sum_i [I_1(\mathbf{x}_i + \mathbf{u}) + \mathbf{J}_1(\mathbf{x}_i + \mathbf{u})\Delta\mathbf{u} - I_0(\mathbf{x}_i)]^2 \quad (8.34)$$

$$= \sum_i [\mathbf{J}_1(\mathbf{x}_i + \mathbf{u})\Delta\mathbf{u} + e_i]^2, \quad (8.35)$$

where

$$\mathbf{J}_1(\mathbf{x}_i + \mathbf{u}) = \nabla I_1(\mathbf{x}_i + \mathbf{u}) = \left(\frac{\partial I_1}{\partial x}, \frac{\partial I_1}{\partial y} \right)(\mathbf{x}_i + \mathbf{u}) \quad (8.36)$$

is the *image gradient* or *Jacobian* at $(\mathbf{x}_i + \mathbf{u})$ and

$$e_i = I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i), \quad (8.37)$$

first introduced in (8.1), is the current intensity error.⁷ The gradient at a particular sub-pixel location $(\mathbf{x}_i + \mathbf{u})$ can be computed using a variety of techniques, the simplest of which is to simply take the horizontal and vertical differences between pixels \mathbf{x} and $\mathbf{x} + (1, 0)$ or $\mathbf{x} + (0, 1)$. More sophisticated derivatives can sometimes lead to noticeable performance improvements.

The linearized form of the incremental update to the SSD error (8.35) is often called the *optical flow constraint* or *brightness constancy constraint* equation

$$I_x u + I_y v + I_t = 0, \quad (8.38)$$

⁷ We follow the convention, commonly used in robotics and by Baker and Matthews (2004), that derivatives with respect to (column) vectors result in row vectors, so that fewer transposes are needed in the formulas.

where the subscripts in I_x and I_y denote spatial derivatives, and I_t is called the *temporal derivative*, which makes sense if we are computing instantaneous velocity in a video sequence. When squared and summed or integrated over a region, it can be used to compute optic flow (Horn and Schunck 1981).

The above least squares problem (8.35) can be minimized by solving the associated *normal equations* (Appendix A.2),

$$\mathbf{A}\Delta\mathbf{u} = \mathbf{b} \quad (8.39)$$

where

$$\mathbf{A} = \sum_i \mathbf{J}_1^T(\mathbf{x}_i + \mathbf{u}) \mathbf{J}_1(\mathbf{x}_i + \mathbf{u}) \quad (8.40)$$

and

$$\mathbf{b} = - \sum_i e_i \mathbf{J}_1^T(\mathbf{x}_i + \mathbf{u}) \quad (8.41)$$

are called the (Gauss–Newton approximation of the) *Hessian* and *gradient-weighted residual vector*, respectively.⁸ These matrices are also often written as

$$\mathbf{A} = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \text{ and } \mathbf{b} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}. \quad (8.42)$$

The gradients required for $\mathbf{J}_1(\mathbf{x}_i + \mathbf{u})$ can be evaluated at the same time as the image warps required to estimate $I_1(\mathbf{x}_i + \mathbf{u})$ (Section 3.6.1 (3.89)) and, in fact, are often computed as a side-product of image interpolation. If efficiency is a concern, these gradients can be replaced by the gradients in the *template* image,

$$\mathbf{J}_1(\mathbf{x}_i + \mathbf{u}) \approx \mathbf{J}_0(\mathbf{x}_i), \quad (8.43)$$

since near the correct alignment, the template and displaced target images should look similar. This has the advantage of allowing the pre-computation of the Hessian and Jacobian images, which can result in significant computational savings (Hager and Belhumeur 1998; Baker and Matthews 2004). A further reduction in computation can be obtained by writing the warped image $I_1(\mathbf{x}_i + \mathbf{u})$ used to compute e_i in (8.37) as a convolution of a sub-pixel interpolation filter with the discrete samples in I_1 (Peleg and Rav-Acha 2006). Precomputing the inner product between the gradient field and shifted version of I_1 allows the iterative re-computation of e_i to be performed in constant time (independent of the number of pixels).

The effectiveness of the above incremental update rule relies on the quality of the Taylor series approximation. When far away from the true displacement (say, 1–2 pixels), several iterations may be needed. It is possible, however, to estimate a value for \mathbf{J}_1 using a least squares fit to a series of larger displacements in order to increase the range of convergence (Jurie and Dhome 2002) or to “learn” a special-purpose recognizer for a given patch (Avidan 2001; Williams, Blake, and Cipolla 2003; Lepetit, Pilet, and Fua 2006; Hinterstoisser, Benhimane, Navab *et al.* 2008; Özuyusal, Calonder, Lepetit *et al.* 2010) as discussed in Section 4.1.4.

A commonly used stopping criterion for incremental updating is to monitor the magnitude of the displacement correction $\|\mathbf{u}\|$ and to stop when it drops below a certain threshold (say,

⁸ The true Hessian is the full second derivative of the error function E , which may not be positive definite—see Section 6.1.3 and Appendix A.3.

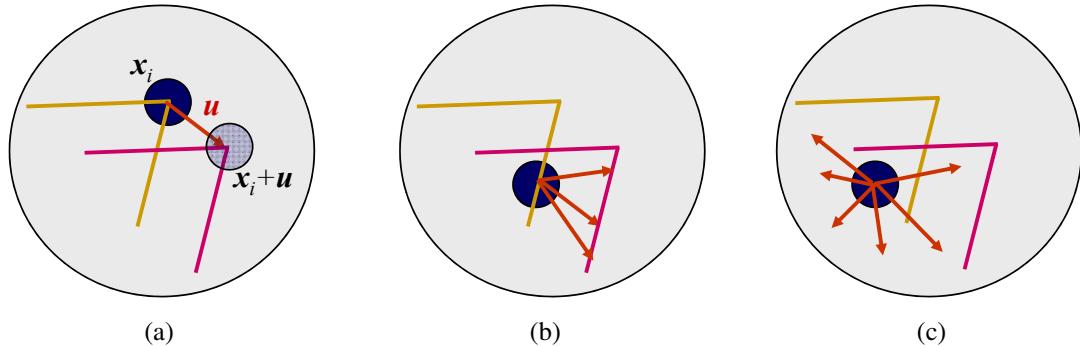


Figure 8.3 Aperture problems for different image regions, denoted by the orange and red L-shaped structures, overlaid in the same image to make it easier to diagram the flow. (a) A window $w(x_i)$ centered at x_i (black circle) can uniquely be matched to its corresponding structure at $x_i + u$ in the second (red) image. (b) A window centered on the edge exhibits the classic aperture problem, since it can be matched to a 1D family of possible locations. (c) In a completely textureless region, the matches become totally unconstrained.

$1/10$ of a pixel). For larger motions, it is usual to combine the incremental update rule with a hierarchical coarse-to-fine search strategy, as described in Section 8.1.1.

Conditioning and aperture problems. Sometimes, the inversion of the linear system (8.39) can be poorly conditioned because of lack of two-dimensional texture in the patch being aligned. A commonly occurring example of this is the *aperture problem*, first identified in some of the early papers on optical flow (Horn and Schunck 1981) and then studied more extensively by Anandan (1989). Consider an image patch that consists of a slanted edge moving to the right (Figure 8.3). Only the *normal* component of the velocity (displacement) can be reliably recovered in this case. This manifests itself in (8.39) as a *rank-deficient* matrix \mathbf{A} , i.e., one whose smaller eigenvalue is very close to zero.⁹

When Equation (8.39) is solved, the component of the displacement along the edge is very poorly conditioned and can result in wild guesses under small noise perturbations. One way to mitigate this problem is to add a *prior* (soft constraint) on the expected range of motions (Simoncelli, Adelson, and Heeger 1991; Baker, Gross, and Matthews 2004; Govindu 2006). This can be accomplished by adding a small value to the diagonal of \mathbf{A} , which essentially biases the solution towards smaller Δu values that still (mostly) minimize the squared error.

However, the pure Gaussian model assumed when using a simple (fixed) quadratic prior, as in (Simoncelli, Adelson, and Heeger 1991), does not always hold in practice, e.g., because of aliasing along strong edges (Triggs 2004). For this reason, it may be prudent to add some small fraction (say, 5%) of the larger eigenvalue to the smaller one before doing the matrix inversion.

Uncertainty modeling. The reliability of a particular patch-based motion estimate can be captured more formally with an *uncertainty model*. The simplest such model is a *covariance matrix*, which captures the expected variance in the motion estimate in all possible directions.

⁹The matrix \mathbf{A} is by construction always guaranteed to be symmetric positive semi-definite, i.e., it has real non-negative eigenvalues.

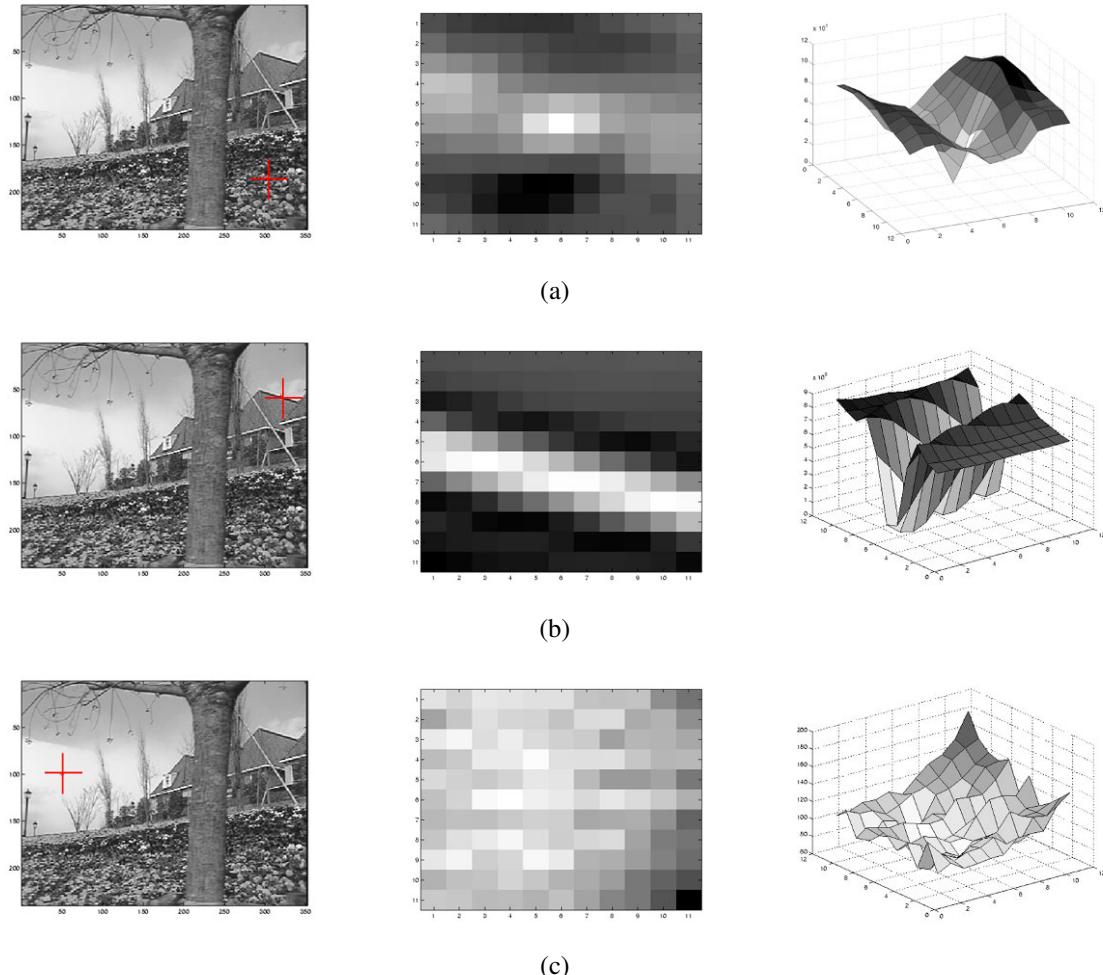


Figure 8.4 SSD surfaces corresponding to three locations (red crosses) in an image: (a) highly textured area, strong minimum, low uncertainty; (b) strong edge, aperture problem, high uncertainty in one direction; (c) weak texture, no clear minimum, large uncertainty.

As discussed in Section 6.1.4 and Appendix B.6, under small amounts of additive Gaussian noise, it can be shown that the covariance matrix $\Sigma_{\mathbf{u}}$ is proportional to the inverse of the Hessian \mathbf{A} ,

$$\Sigma_{\mathbf{u}} = \sigma_n^2 \mathbf{A}^{-1}, \quad (8.44)$$

where σ_n^2 is the variance of the additive Gaussian noise (Anandan 1989; Matthies, Kanade, and Szeliski 1989; Szeliski 1989).

For larger amounts of noise, the linearization performed by the Lucas–Kanade algorithm in (8.35) is only approximate, so the above quantity becomes a *Cramer–Rao lower bound* on the true covariance. Thus, the minimum and maximum eigenvalues of the Hessian \mathbf{A} can now be interpreted as the (scaled) inverse variances in the least-certain and most-certain directions of motion. (A more detailed analysis using a more realistic model of image noise is given by Steele and Jaynes (2005).) Figure 8.4 shows the local SSD surfaces for three different pixel locations in an image. As you can see, the surface has a clear minimum in the highly textured region and suffers from the aperture problem near the strong edge.

Bias and gain, weighting, and robust error metrics. The Lucas–Kanade update rule can also be applied to the bias–gain equation (8.9) to obtain

$$E_{\text{LK–BG}}(\mathbf{u} + \Delta\mathbf{u}) = \sum_i [\mathbf{J}_1(\mathbf{x}_i + \mathbf{u})\Delta\mathbf{u} + e_i - \alpha I_0(\mathbf{x}_i) - \beta]^2 \quad (8.45)$$

(Lucas and Kanade 1981; Gennert 1988; Fuh and Maragos 1991; Baker, Gross, and Matthews 2003). The resulting 4×4 system of equations can be solved to simultaneously estimate the translational displacement update $\Delta\mathbf{u}$ and the bias and gain parameters β and α .

A similar formulation can be derived for images (templates) that have a *linear appearance variation*,

$$I_1(\mathbf{x} + \mathbf{u}) \approx I_0(\mathbf{x}) + \sum_j \lambda_j B_j(\mathbf{x}), \quad (8.46)$$

where the $B_j(\mathbf{x})$ are the *basis images* and the λ_j are the unknown coefficients (Hager and Belhumeur 1998; Baker, Gross, Ishikawa *et al.* 2003; Baker, Gross, and Matthews 2003). Potential linear appearance variations include illumination changes (Hager and Belhumeur 1998) and small non-rigid deformations (Black and Jepson 1998).

A weighted (windowed) version of the Lucas–Kanade algorithm is also possible:

$$E_{\text{LK–WSSD}}(\mathbf{u} + \Delta\mathbf{u}) = \sum_i w_0(\mathbf{x}_i) w_1(\mathbf{x}_i + \mathbf{u}) [\mathbf{J}_1(\mathbf{x}_i + \mathbf{u})\Delta\mathbf{u} + e_i]^2. \quad (8.47)$$

Note that here, in deriving the Lucas–Kanade update from the original weighted SSD function (8.5), we have neglected taking the derivative of the $w_1(\mathbf{x}_i + \mathbf{u})$ weighting function with respect to \mathbf{u} , which is usually acceptable in practice, especially if the weighting function is a binary mask with relatively few transitions.

Baker, Gross, Ishikawa *et al.* (2003) only use the $w_0(\mathbf{x})$ term, which is reasonable if the two images have the same extent and no (independent) cutouts in the overlap region. They also discuss the idea of making the weighting proportional to $\nabla I(\mathbf{x})$, which helps for very noisy images, where the gradient itself is noisy. Similar observations, formulated in terms of *total least squares* (Van Huffel and Vandewalle 1991; Van Huffel and Lemmerling 2002),

have been made by other researchers studying optical flow (Weber and Malik 1995; Bab-Hadiashar and Suter 1998b; Mühlich and Mester 1998). Lastly, Baker, Gross, Ishikawa *et al.* (2003) show how evaluating Equation (8.47) at just the *most reliable* (highest gradient) pixels does not significantly reduce performance for large enough images, even if only 5–10% of the pixels are used. (This idea was originally proposed by Dellaert and Collins (1999), who used a more sophisticated selection criterion.)

The Lucas–Kanade incremental refinement step can also be applied to the robust error metric introduced in Section 8.1,

$$E_{\text{LK-SRD}}(\mathbf{u} + \Delta\mathbf{u}) = \sum_i \rho(\mathbf{J}_1(\mathbf{x}_i + \mathbf{u})\Delta\mathbf{u} + e_i), \quad (8.48)$$

which can be solved using the *iteratively reweighted least squares* technique described in Section 6.1.4.

8.2 Parametric motion

Many image alignment tasks, for example image stitching with handheld cameras, require the use of more sophisticated motion models, as described in Section 2.1.2. Since these models, e.g., affine deformations, typically have more parameters than pure translation, a full search over the possible range of values is impractical. Instead, the incremental Lucas–Kanade algorithm can be generalized to parametric motion models and used in conjunction with a hierarchical search algorithm (Lucas and Kanade 1981; Rehg and Witkin 1991; Fuh and Maragos 1991; Bergen, Anandan, Hanna *et al.* 1992; Shashua and Toelg 1997; Shashua and Wexler 2001; Baker and Matthews 2004).

For parametric motion, instead of using a single constant translation vector \mathbf{u} , we use a spatially varying *motion field* or *correspondence map*, $\mathbf{x}'(\mathbf{x}; \mathbf{p})$, parameterized by a low-dimensional vector \mathbf{p} , where \mathbf{x}' can be any of the motion models presented in Section 2.1.2. The parametric incremental motion update rule now becomes

$$E_{\text{LK-PM}}(\mathbf{p} + \Delta\mathbf{p}) = \sum_i [I_1(\mathbf{x}'(\mathbf{x}_i; \mathbf{p} + \Delta\mathbf{p})) - I_0(\mathbf{x}_i)]^2 \quad (8.49)$$

$$\approx \sum_i [I_1(\mathbf{x}'_i) + \mathbf{J}_1(\mathbf{x}'_i)\Delta\mathbf{p} - I_0(\mathbf{x}_i)]^2 \quad (8.50)$$

$$= \sum_i [\mathbf{J}_1(\mathbf{x}'_i)\Delta\mathbf{p} + e_i]^2, \quad (8.51)$$

where the Jacobian is now

$$\mathbf{J}_1(\mathbf{x}'_i) = \frac{\partial I_1}{\partial \mathbf{p}} = \nabla I_1(\mathbf{x}'_i) \frac{\partial \mathbf{x}'}{\partial \mathbf{p}}(\mathbf{x}_i), \quad (8.52)$$

i.e., the product of the image gradient ∇I_1 with the Jacobian of the correspondence field, $\mathbf{J}_{\mathbf{x}'} = \partial \mathbf{x}' / \partial \mathbf{p}$.

The motion Jacobians $\mathbf{J}_{\mathbf{x}'}$ for the 2D planar transformations introduced in Section 2.1.2 and Table 2.1 are given in Table 6.1. Note how we have re-parameterized the motion matrices so that they are always the identity at the origin $\mathbf{p} = 0$. This becomes useful later, when we talk about the compositional and inverse compositional algorithms. (It also makes it easier to impose priors on the motions.)

For parametric motion, the (Gauss–Newton) *Hessian* and *gradient-weighted residual vector* become

$$\mathbf{A} = \sum_i \mathbf{J}_{\mathbf{x}'}^T(\mathbf{x}_i) [\nabla I_1^T(\mathbf{x}'_i) \nabla I_1(\mathbf{x}'_i)] \mathbf{J}_{\mathbf{x}'}(\mathbf{x}_i) \quad (8.53)$$

and

$$\mathbf{b} = - \sum_i \mathbf{J}_{\mathbf{x}'}^T(\mathbf{x}_i) [e_i \nabla I_1^T(\mathbf{x}'_i)]. \quad (8.54)$$

Note how the expressions inside the square brackets are the same ones evaluated for the simpler translational motion case (8.40–8.41).

Patch-based approximation. The computation of the Hessian and residual vectors for parametric motion can be significantly more expensive than for the translational case. For parametric motion with n parameters and N pixels, the accumulation of \mathbf{A} and \mathbf{b} takes $O(n^2N)$ operations (Baker and Matthews 2004). One way to reduce this by a significant amount is to divide the image up into smaller sub-blocks (patches) P_j and to only accumulate the simpler 2×2 quantities inside the square brackets at the pixel level (Shum and Szeliski 2000),

$$\mathbf{A}_j = \sum_{i \in P_j} \nabla I_1^T(\mathbf{x}'_i) \nabla I_1(\mathbf{x}'_i) \quad (8.55)$$

$$\mathbf{b}_j = \sum_{i \in P_j} e_i \nabla I_1^T(\mathbf{x}'_i). \quad (8.56)$$

The full Hessian and residual can then be approximated as

$$\mathbf{A} \approx \sum_j \mathbf{J}_{\mathbf{x}'}^T(\hat{\mathbf{x}}_j) [\sum_{i \in P_j} \nabla I_1^T(\mathbf{x}'_i) \nabla I_1(\mathbf{x}'_i)] \mathbf{J}_{\mathbf{x}'}(\hat{\mathbf{x}}_j) = \sum_j \mathbf{J}_{\mathbf{x}'}^T(\hat{\mathbf{x}}_j) \mathbf{A}_j \mathbf{J}_{\mathbf{x}'}(\hat{\mathbf{x}}_j) \quad (8.57)$$

and

$$\mathbf{b} \approx - \sum_j \mathbf{J}_{\mathbf{x}'}^T(\hat{\mathbf{x}}_j) [\sum_{i \in P_j} e_i \nabla I_1^T(\mathbf{x}'_i)] = - \sum_j \mathbf{J}_{\mathbf{x}'}^T(\hat{\mathbf{x}}_j) \mathbf{b}_j, \quad (8.58)$$

where $\hat{\mathbf{x}}_j$ is the *center* of each patch P_j (Shum and Szeliski 2000). This is equivalent to replacing the true motion Jacobian with a piecewise-constant approximation. In practice, this works quite well. The relationship of this approximation to feature-based registration is discussed in Section 9.2.4.

Compositional approach. For a complex parametric motion such as a homography, the computation of the motion Jacobian becomes complicated and may involve a per-pixel division. Szeliski and Shum (1997) observed that this can be simplified by first warping the target image I_1 according to the current motion estimate $\mathbf{x}'(\mathbf{x}; \mathbf{p})$,

$$\tilde{I}_1(\mathbf{x}) = I_1(\mathbf{x}'(\mathbf{x}; \mathbf{p})), \quad (8.59)$$

and then comparing this *warped* image against the template $I_0(\mathbf{x})$,

$$E_{\text{LK-ss}}(\Delta \mathbf{p}) = \sum_i [\tilde{I}_1(\tilde{\mathbf{x}}(\mathbf{x}_i; \Delta \mathbf{p})) - I_0(\mathbf{x}_i)]^2 \quad (8.60)$$

$$\approx \sum_i [\tilde{J}_1(\mathbf{x}_i) \Delta \mathbf{p} + e_i]^2 \quad (8.61)$$

$$= \sum_i [\nabla \tilde{I}_1(\mathbf{x}_i) \mathbf{J}_{\tilde{\mathbf{x}}}(\mathbf{x}_i) \Delta \mathbf{p} + e_i]^2. \quad (8.62)$$

Note that since the two images are assumed to be fairly similar, only an *incremental* parametric motion is required, i.e., the incremental motion can be evaluated around $\mathbf{p} = 0$, which can lead to considerable simplifications. For example, the Jacobian of the planar projective transform (6.19) now becomes

$$\mathbf{J}_{\tilde{\mathbf{x}}} = \left. \frac{\partial \tilde{\mathbf{x}}}{\partial \mathbf{p}} \right|_{\mathbf{p}=0} = \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -x^2 & -xy \\ 0 & 0 & 0 & x & y & 1 & -xy & -y^2 \end{bmatrix}. \quad (8.63)$$

Once the incremental motion $\tilde{\mathbf{x}}$ has been computed, it can be *prepended* to the previously estimated motion, which is easy to do for motions represented with transformation matrices, such as those given in Tables 2.1 and 6.1. Baker and Matthews (2004) call this the *forward compositional* algorithm, since the target image is being re-warped and the final motion estimates are being composed.

If the appearance of the warped and template images is similar enough, we can replace the gradient of $\tilde{I}_1(\mathbf{x})$ with the gradient of $I_0(\mathbf{x})$, as suggested previously (8.43). This has potentially a big advantage in that it allows the pre-computation (and inversion) of the Hessian matrix \mathbf{A} given in Equation (8.53). The residual vector \mathbf{b} (8.54) can also be partially precomputed, i.e., the *steepest descent images* $\nabla I_0(\mathbf{x}) \mathbf{J}_{\tilde{\mathbf{x}}}(\mathbf{x})$ can be precomputed and stored for later multiplication with the $e(\mathbf{x}) = \tilde{I}_1(\mathbf{x}) - I_0(\mathbf{x})$ error images (Baker and Matthews 2004). This idea was first suggested by Hager and Belhumeur (1998) in what Baker and Matthews (2004) call a *inverse additive* scheme.

Baker and Matthews (2004) introduce one more variant they call the *inverse compositional* algorithm. Rather than (conceptually) re-warping the warped target image $\tilde{I}_1(\mathbf{x})$, they instead warp the template image $I_0(\mathbf{x})$ and minimize

$$E_{\text{LK-BM}}(\Delta \mathbf{p}) = \sum_i [\tilde{I}_1(\mathbf{x}_i) - I_0(\tilde{\mathbf{x}}(\mathbf{x}_i; \Delta \mathbf{p}))]^2 \quad (8.64)$$

$$\approx \sum_i [\nabla I_0(\mathbf{x}_i) \mathbf{J}_{\tilde{\mathbf{x}}}(\mathbf{x}_i) \Delta \mathbf{p} - e_i]^2. \quad (8.65)$$

This is identical to the forward warped algorithm (8.62) with the gradients $\nabla \tilde{I}_1(\mathbf{x})$ replaced by the gradients $\nabla I_0(\mathbf{x})$, except for the sign of e_i . The resulting update $\Delta \mathbf{p}$ is the *negative* of the one computed by the modified Equation (8.62) and hence the *inverse* of the incremental transformation must be prepended to the current transform. Because the inverse compositional algorithm has the potential of pre-computing the inverse Hessian and the steepest descent images, this makes it the preferred approach of those surveyed by Baker and Matthews (2004). Figure 8.5 (Baker, Gross, Ishikawa *et al.* 2003) beautifully shows all of the steps required to implement the inverse compositional algorithm.

Baker and Matthews (2004) also discuss the advantage of using Gauss–Newton iteration (i.e., the first-order expansion of the least squares, as above) compared to other approaches such as steepest descent and Levenberg–Marquardt. Subsequent parts of the series (Baker, Gross, Ishikawa *et al.* 2003; Baker, Gross, and Matthews 2003, 2004) discuss more advanced topics such as per-pixel weighting, pixel selection for efficiency, a more in-depth discussion of robust metrics and algorithms, linear appearance variations, and priors on parameters. They make for invaluable reading for anyone interested in implementing a highly tuned implementation of incremental image registration. Evangelidis and Psarakis (2008) provide some detailed experimental evaluations of these and other related approaches.

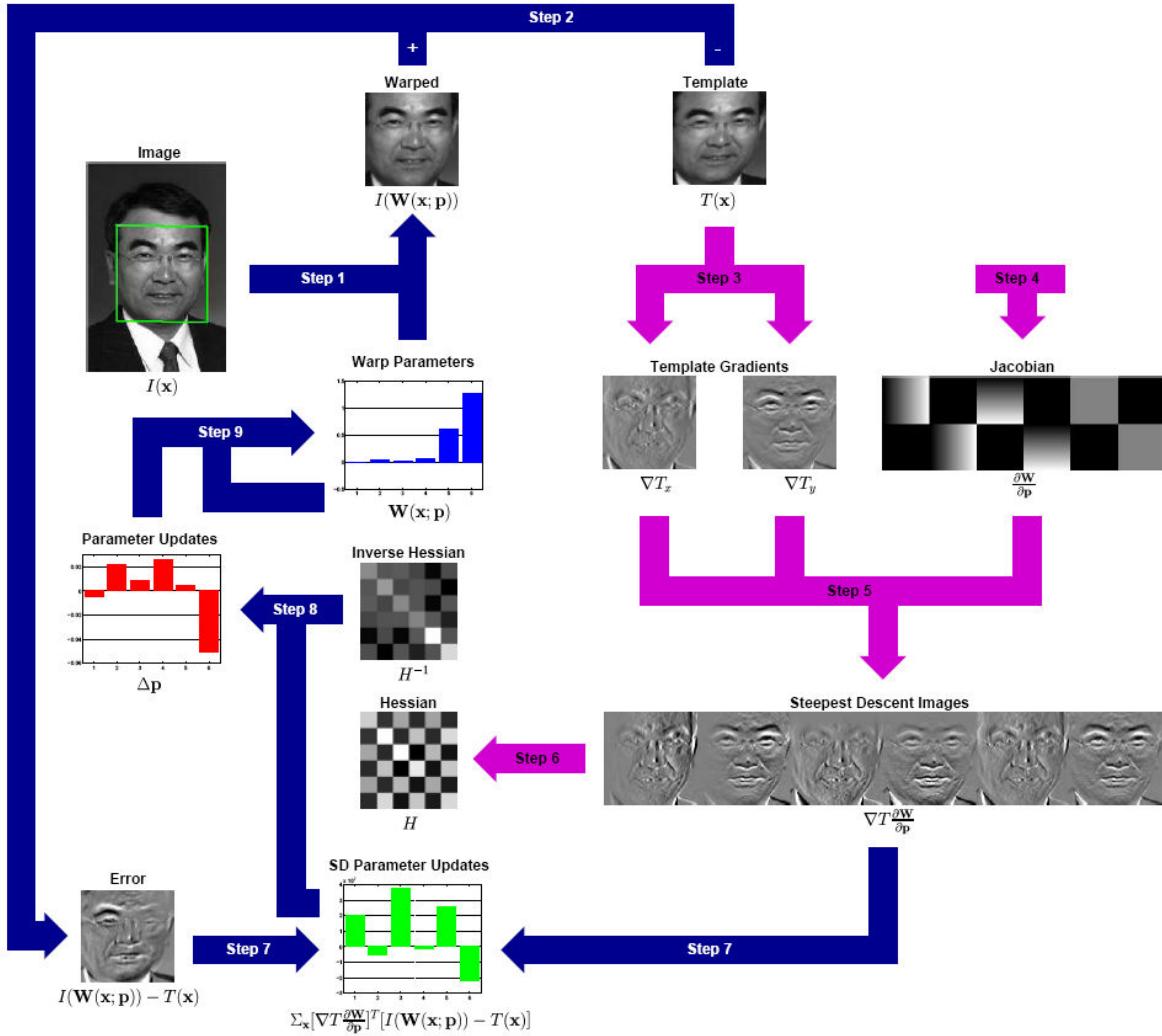


Figure 8.5 A schematic overview of the inverse compositional algorithm (copied, with permission, from (Baker, Gross, Ishikawa *et al.* 2003)). Steps 3–6 (light-colored arrows) are performed once as a pre-computation. The main algorithm simply consists of iterating: image warping (Step 1), image differencing (Step 2), image dot products (Step 7), multiplication with the inverse of the Hessian (Step 8), and the update to the warp (Step 9). All of these steps can be performed efficiently.

8.2.1 Application: Video stabilization

Video stabilization is one of the most widely used applications of parametric motion estimation (Hansen, Anandan, Dana *et al.* 1994; Irani, Rousso, and Peleg 1997; Morimoto and Chellappa 1997; Srinivasan, Chellappa, Veeraraghavan *et al.* 2005). Algorithms for stabilization run inside both hardware devices, such as camcorders and still cameras, and software packages for improving the visual quality of shaky videos.

In their paper on full-frame video stabilization, Matsushita, Ofek, Ge *et al.* (2006) give a nice overview of the three major stages of stabilization, namely motion estimation, motion smoothing, and image warping. Motion estimation algorithms often use a similarity transform to handle camera translations, rotations, and zooming. The tricky part is getting these algorithms to lock onto the background motion, which is a result of the camera movement, without getting distracted by independent moving foreground objects. Motion smoothing algorithms recover the low-frequency (slowly varying) part of the motion and then estimate the high-frequency shake component that needs to be removed. Finally, image warping algorithms apply the high-frequency correction to render the original frames as if the camera had undergone only the smooth motion.

The resulting stabilization algorithms can greatly improve the appearance of shaky videos but they often still contain visual artifacts. For example, image warping can result in missing borders around the image, which must be cropped, filled using information from other frames, or hallucinated using inpainting techniques (Section 10.5.1). Furthermore, video frames captured during fast motion are often blurry. Their appearance can be improved either using deblurring techniques (Section 10.3) or stealing sharper pixels from other frames with less motion or better focus (Matsushita, Ofek, Ge *et al.* 2006). Exercise 8.3 has you implement and test some of these ideas.

In situations where the camera is translating a lot in 3D, e.g., when the videographer is walking, an even better approach is to compute a full structure from motion reconstruction of the camera motion and 3D scene. A smooth 3D camera path can then be computed and the original video re-rendered using view interpolation with the interpolated 3D point cloud serving as the proxy geometry while preserving salient features (Liu, Gleicher, Jin *et al.* 2009). If you have access to a camera array instead of a single video camera, you can do even better using a light field rendering approach (Section 13.3) (Smith, Zhang, Jin *et al.* 2009).

8.2.2 Learned motion models

An alternative to parameterizing the motion field with a geometric deformation such as an affine transform is to learn a set of basis functions tailored to a particular application (Black, Yacoob, Jepson *et al.* 1997). First, a set of dense motion fields (Section 8.4) is computed from a set of training videos. Next, singular value decomposition (SVD) is applied to the stack of motion fields $\mathbf{u}_t(\mathbf{x})$ to compute the first few singular vectors $\mathbf{v}_k(\mathbf{x})$. Finally, for a new test sequence, a novel flow field is computed using a coarse-to-fine algorithm that estimates the unknown coefficient a_k in the parameterized flow field

$$\mathbf{u}(\mathbf{x}) = \sum_k a_k \mathbf{v}_k(\mathbf{x}). \quad (8.66)$$

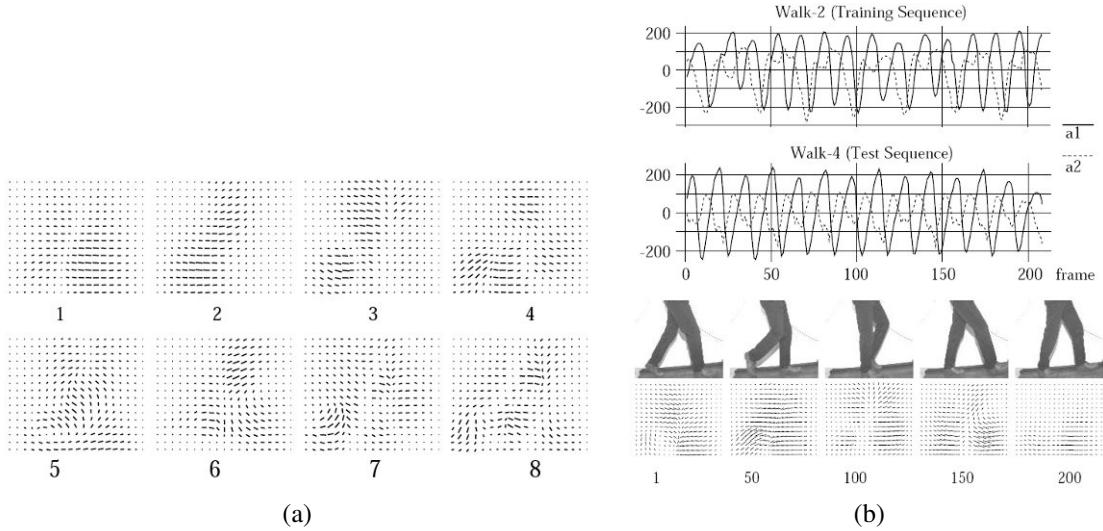


Figure 8.6 Learned parameterized motion fields for a walking sequence (Black, Yacoob, Jepson *et al.* 1997) © 1997 IEEE: (a) learned basis flow fields; (b) plots of motion coefficients over time and corresponding estimated motion fields.

Figure 8.6a shows a set of basis fields learned by observing videos of walking motions. Figure 8.6b shows the temporal evolution of the basis coefficients as well as a few of the recovered parametric motion fields. Note that similar ideas can also be applied to feature tracks (Torresani, Hertzmann, and Bregler 2008), which is a topic we discuss in more detail in Sections 4.1.4 and 12.6.4.

8.3 Spline-based motion

While parametric motion models are useful in a wide variety of applications (such as video stabilization and mapping onto planar surfaces), most image motion is too complicated to be captured by such low-dimensional models.

Traditionally, optical flow algorithms (Section 8.4) compute an independent motion estimate for each pixel, i.e., the number of flow vectors computed is equal to the number of input pixels. The general optical flow analog to Equation (8.1) can thus be written as

$$E_{SSD-OF}(\{\mathbf{u}_i\}) = \sum_i [I_1(\mathbf{x}_i + \mathbf{u}_i) - I_0(\mathbf{x}_i)]^2. \quad (8.67)$$

Notice how in the above equation, the number of variables $\{\mathbf{u}_i\}$ is twice the number of measurements, so the problem is underconstrained.

The two classic approaches to this problem, which we study in Section 8.4, are to perform the summation over overlapping regions (the *patch-based* or *window-based* approach) or to add smoothness terms on the $\{\mathbf{u}_i\}$ field using *regularization* or *Markov random fields* (Section 3.7). In this section, we describe an alternative approach that lies somewhere between general optical flow (independent flow at each pixel) and parametric flow (a small number of global parameters). The approach is to represent the motion field as a two-dimensional *spline*

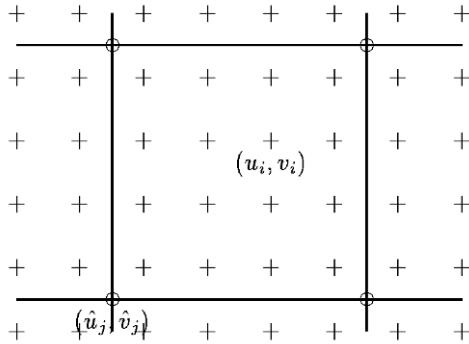


Figure 8.7 Spline motion field: the displacement vectors $\mathbf{u}_i = (u_i, v_i)$ are shown as pluses (+) and are controlled by the smaller number of control vertices $\hat{\mathbf{u}}_j = (\hat{u}_j, \hat{v}_j)$, which are shown as circles (○).

controlled by a smaller number of *control vertices* $\{\hat{\mathbf{u}}_j\}$ (Figure 8.7),

$$\mathbf{u}_i = \sum_j \hat{\mathbf{u}}_j B_j(\mathbf{x}_i) = \sum_j \hat{\mathbf{u}}_j w_{i,j}, \quad (8.68)$$

where the $B_j(\mathbf{x}_i)$ are called the *basis functions* and are only non-zero over a small *finite support* interval (Szeliski and Coughlan 1997). We call the $w_{ij} = B_j(\mathbf{x}_i)$ *weights* to emphasize that the $\{\mathbf{u}_i\}$ are known linear combinations of the $\{\hat{\mathbf{u}}_j\}$. Some commonly used spline basis functions are shown in Figure 8.8.

Substituting the formula for the individual per-pixel flow vectors \mathbf{u}_i (8.68) into the SSD error metric (8.67) yields a parametric motion formula similar to Equation (8.50). The biggest difference is that the Jacobian $\mathbf{J}_1(\mathbf{x}'_i)$ (8.52) now consists of the sparse entries in the weight matrix $\mathbf{W} = [w_{ij}]$.

In situations where we know something more about the motion field, e.g., when the motion is due to a camera moving in a static scene, we can use more specialized motion models. For example, the *plane plus parallax* model (Section 2.1.5) can be naturally combined with a spline-based motion representation, where the in-plane motion is represented by a homography (6.19) and the out-of-plane parallax d is represented by a scalar variable at each spline control point (Szeliski and Kang 1995; Szeliski and Coughlan 1997).

In many cases, the small number of spline vertices results in a motion estimation problem that is well conditioned. However, if large textureless regions (or elongated edges subject to the aperture problem) persist across several spline patches, it may be necessary to add a *regularization* term to make the problem well posed (Section 3.7.1). The simplest way to do this is to directly add squared difference penalties between adjacent vertices in the spline control mesh $\{\hat{\mathbf{u}}_j\}$, as in (3.100). If a multi-resolution (coarse-to-fine) strategy is being used, it is important to re-scale these smoothness terms while going from level to level.

The linear system corresponding to the spline-based motion estimator is sparse and regular. Because it is usually of moderate size, it can often be solved using direct techniques such as Cholesky decomposition (Appendix A.4). Alternatively, if the problem becomes too large and subject to excessive fill-in, iterative techniques such as hierarchically preconditioned conjugate gradient (Szeliski 1990b, 2006b) can be used instead (Appendix A.5).

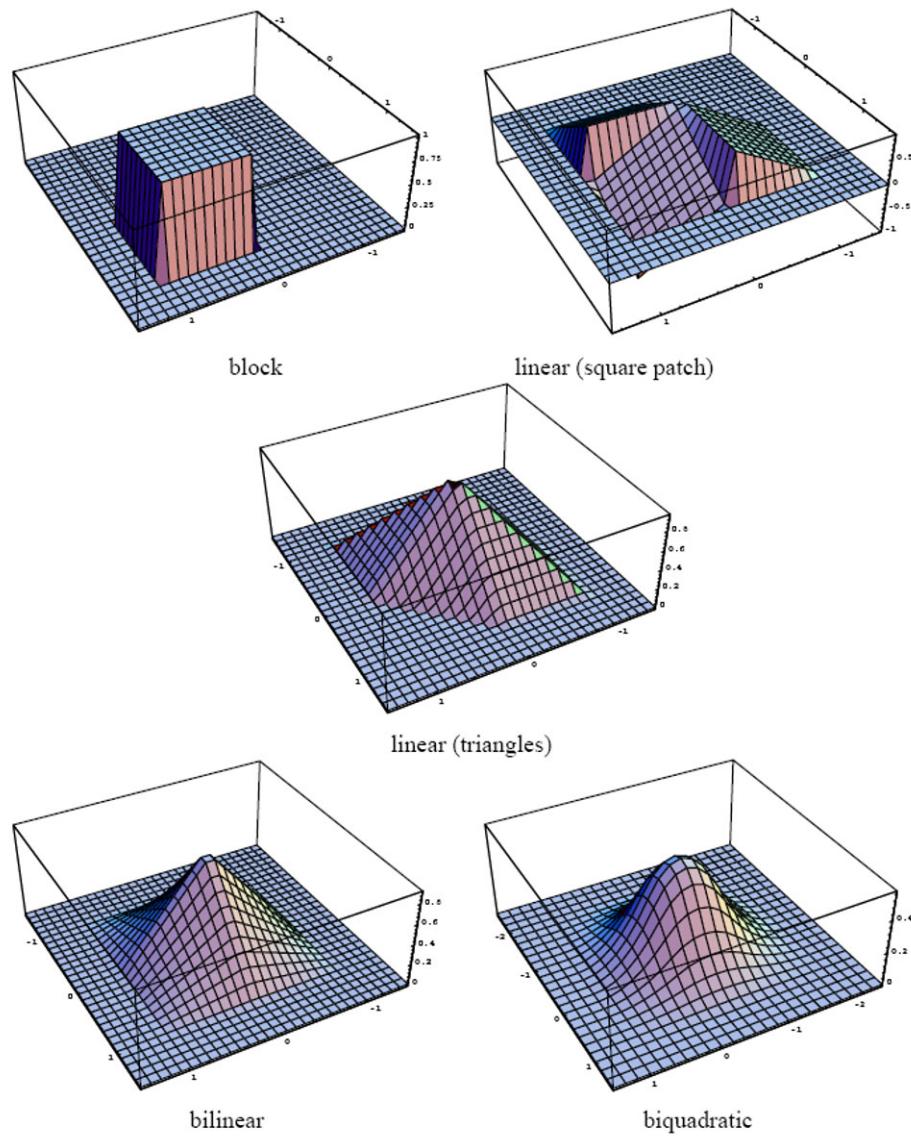


Figure 8.8 Sample spline basis functions (Szeliski and Coughlan 1997) © 1997 Springer. The block (constant) interpolator/basis corresponds to block-based motion estimation (Le Gall 1991). See Section 3.5.1 for more details on spline functions.

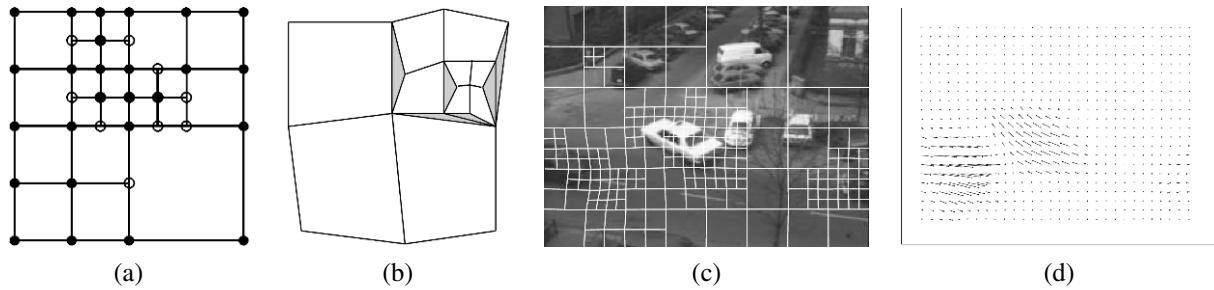


Figure 8.9 Quadtree spline-based motion estimation (Szeliski and Shum 1996) © 1996 IEEE: (a) quadtree spline representation, (b) which can lead to *cracks*, unless the white nodes are constrained to depend on their parents; (c) deformed quadtree spline mesh overlaid on grayscale image; (d) flow field visualized as a needle diagram.

Because of its robustness, spline-based motion estimation has been used for a number of applications, including visual effects (Roble 1999) and medical image registration (Section 8.3.1) (Szeliski and Lavallée 1996; Kybic and Unser 2003).

One disadvantage of the basic technique, however, is that the model does a poor job near motion discontinuities, unless an excessive number of nodes is used. To remedy this situation, Szeliski and Shum (1996) propose using a *quadtree* representation embedded in the spline control grid (Figure 8.9a). Large cells are used to present regions of smooth motion, while smaller cells are added in regions of motion discontinuities (Figure 8.9c).

To estimate the motion, a coarse-to-fine strategy is used. Starting with a regular spline imposed over a lower-resolution image, an initial motion estimate is obtained. Spline patches where the motion is inconsistent, i.e., the squared residual (8.67) is above a threshold, are subdivided into smaller patches. In order to avoid *cracks* in the resulting motion field (Figure 8.9b), the values of certain nodes in the refined mesh, i.e., those adjacent to larger cells, need to be *restricted* so that they depend on their parent values. This is most easily accomplished using a hierarchical basis representation for the quadtree spline (Szeliski 1990b) and selectively setting some of the hierarchical basis functions to 0, as described in (Szeliski and Shum 1996).

8.3.1 Application: Medical image registration

Because they excel at representing smooth *elastic* deformation fields, spline-based motion models have found widespread use in medical image registration (Bajcsy and Kovacic 1989; Szeliski and Lavallée 1996; Christensen, Joshi, and Miller 1997).¹⁰ Registration techniques can be used both to track an individual patient's development or progress over time (a *longitudinal* study) or to match different patient images together to find commonalities and detect variations or pathologies (*cross-sectional* studies). When different imaging *modalities* are being registered, e.g., computed tomography (CT) scans and magnetic resonance images (MRI), *mutual information* measures of similarity are often necessary (Viola and Wells III 1997; Maes, Collignon, Vandermeulen *et al.* 1997).

¹⁰ In computer graphics, such elastic volumetric deformation are known as *free-form deformations* (Sederberg and Parry 1986; Coquillart 1990; Celniker and Gossard 1991).

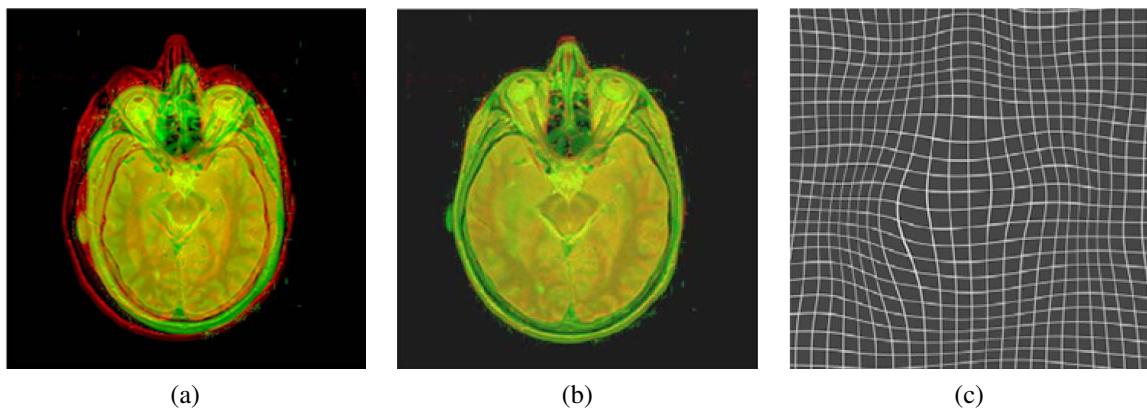


Figure 8.10 Elastic brain registration (Kybic and Unser 2003) © 2003 IEEE: (a) original brain atlas and patient MRI images overlaid in red–green; (b) after elastic registration with eight user-specified landmarks (not shown); (c) a cubic B-spline deformation field, shown as a deformed grid.

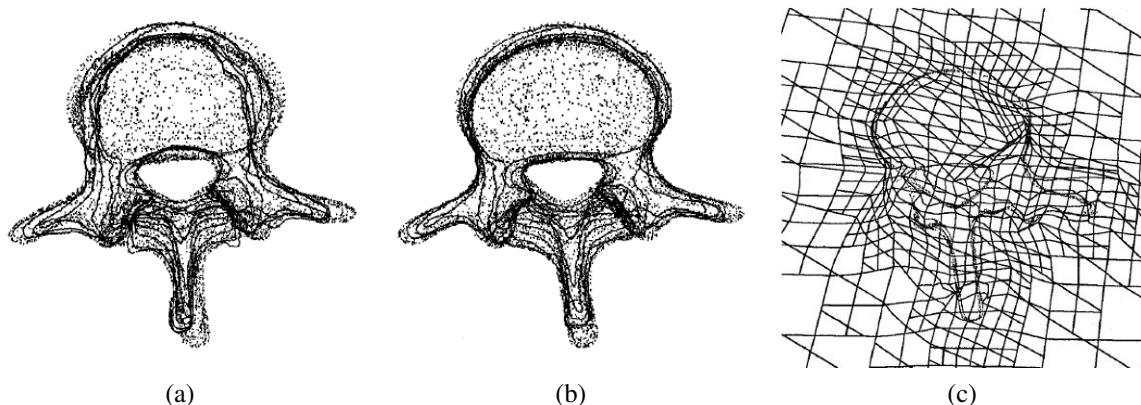


Figure 8.11 Octree spline-based image registration of two vertebral surface models (Szeliski and Lavallée 1996) © 1996 Springer: (a) after initial rigid alignment; (b) after elastic alignment; (c) a cross-section through the adapted octree spline deformation field.

Kybic and Unser (2003) provide a nice literature review and describe a complete working system based on representing both the images and the deformation fields as multi-resolution splines. Figure 8.10 shows an example of the Kybic and Unser system being used to register a patient’s brain MRI with a labeled brain atlas image. The system can be run in a fully automatic mode but more accurate results can be obtained by locating a few key *landmarks*. More recent papers on deformable medical image registration, including performance evaluations, include (Klein, Staring, and Pluim 2007; Glocker, Komodakis, Tziritas *et al.* 2008).

As with other applications, regular volumetric splines can be enhanced using selective refinement. In the case of 3D volumetric image or surface registration, these are known as *octree splines* (Szeliski and Lavallée 1996) and have been used to register medical surface models such as vertebrae and faces from different patients (Figure 8.11).

8.4 Optical flow

The most general (and challenging) version of motion estimation is to compute an independent estimate of motion at *each* pixel, which is generally known as *optical* (or *optic*) *flow*. As we mentioned in the previous section, this generally involves minimizing the brightness or color difference between corresponding pixels summed over the image,

$$E_{\text{SSD-OF}}(\{\mathbf{u}_i\}) = \sum_i [I_1(\mathbf{x}_i + \mathbf{u}_i) - I_0(\mathbf{x}_i)]^2. \quad (8.69)$$

Since the number of variables $\{\mathbf{u}_i\}$ is twice the number of measurements, the problem is underconstrained. The two classic approaches to this problem are to perform the summation *locally* over overlapping regions (the *patch-based* or *window-based* approach) or to add smoothness terms on the $\{\mathbf{u}_i\}$ field using regularization or Markov random fields (Section 3.7) and to search for a global minimum.

The patch-based approach usually involves using a Taylor series expansion of the displaced image function (8.35) in order to obtain sub-pixel estimates (Lucas and Kanade 1981). Anandan (1989) shows how a series of local discrete search steps can be interleaved with Lucas–Kanade incremental refinement steps in a coarse-to-fine pyramid scheme, which allows the estimation of large motions, as described in Section 8.1.1. He also analyzes how the *uncertainty* in local motion estimates is related to the eigenvalues of the local Hessian matrix \mathbf{A}_i (8.44), as shown in Figures 8.3–8.4.

Bergen, Anandan, Hanna *et al.* (1992) develop a unified framework for describing both parametric (Section 8.2) and patch-based optic flow algorithms and provide a nice introduction to this topic. After each iteration of optic flow estimation in a coarse-to-fine pyramid, they re-warp one of the images so that only incremental flow estimates are computed (Section 8.1.1). When overlapping patches are used, an efficient implementation is to first compute the outer products of the gradients and intensity errors (8.40–8.41) at every pixel and then perform the overlapping window sums using a moving average filter.¹¹

Instead of solving for each motion (or motion update) independently, Horn and Schunck (1981) develop a regularization-based framework where (8.69) is simultaneously minimized over all flow vectors $\{\mathbf{u}_i\}$. In order to constrain the problem, smoothness constraints, i.e., squared penalties on flow derivatives, are added to the basic per-pixel error metric. Because the technique was originally developed for small motions in a variational (continuous function) framework, the linearized *brightness constancy constraint* corresponding to (8.35), i.e., (8.38), is more commonly written as an analytic integral

$$E_{\text{HS}} = \int (I_x u + I_y v + I_t)^2 dx dy, \quad (8.70)$$

where $(I_x, I_y) = \nabla I_1 = \mathbf{J}_1$ and $I_t = e_i$ is the *temporal derivative*, i.e., the brightness change between images. The Horn and Schunck model can also be viewed as the limiting case of spline-based motion estimation as the splines become 1x1 pixel patches.

It is also possible to combine ideas from local and global flow estimation into a single framework by using a locally aggregated (as opposed to single-pixel) Hessian as the brightness constancy term (Bruhn, Weickert, and Schnörr 2005). Consider the discrete analog

¹¹Other smoothing or aggregation filters can also be used at this stage (Bruhn, Weickert, and Schnörr 2005).

(8.35) to the analytic global energy (8.70),

$$E_{\text{HSD}} = \sum_i \mathbf{u}_i^T [\mathbf{J}_i \mathbf{J}_i^T] \mathbf{u}_i + 2e_i \mathbf{J}_i^T \mathbf{u}_i + e_i^2. \quad (8.71)$$

If we replace the per-pixel (rank 1) Hessians $\mathbf{A}_i = [\mathbf{J}_i \mathbf{J}_i^T]$ and residuals $\mathbf{b}_i = \mathbf{J}_i e_i$ with area-aggregated versions (8.40–8.41), we obtain a global minimization algorithm where region-based brightness constraints are used.

Another extension to the basic optic flow model is to use a combination of global (parametric) and local motion models. For example, if we know that the motion is due to a camera moving in a static scene (rigid motion), we can re-formulate the problem as the estimation of a per-pixel depth along with the parameters of the global camera motion (Adiv 1989; Hanna 1991; Bergen, Anandan, Hanna *et al.* 1992; Szeliski and Coughlan 1997; Nir, Bruckstein, and Kimmel 2008; Wedel, Cremers, Pock *et al.* 2009). Such techniques are closely related to stereo matching (Chapter 11). Alternatively, we can estimate either per-image or per-segment affine motion models combined with per-pixel *residual* corrections (Black and Jepson 1996; Ju, Black, and Jepson 1996; Chang, Tekalp, and Sezan 1997; Mémin and Pérez 2002). We revisit this topic in Section 8.5.

Of course, image brightness may not always be an appropriate metric for measuring appearance consistency, e.g., when the lighting in an image is varying. As discussed in Section 8.1, matching gradients, filtered images, or other metrics such as image Hessians (second derivative measures) may be more appropriate. It is also possible to locally compute the *phase* of steerable filters in the image, which is insensitive to both bias and gain transformations (Fleet and Jepson 1990). Papenberg, Bruhn, Brox *et al.* (2006) review and explore such constraints and also provide a detailed analysis and justification for iteratively re-warping images during incremental flow computation.

Because the brightness constancy constraint is evaluated at each pixel independently, rather than being summed over patches where the constant flow assumption may be violated, global optimization approaches tend to perform better near motion discontinuities. This is especially true if robust metrics are used in the smoothness constraint (Black and Anandan 1996; Bab-Hadiashar and Suter 1998a).¹² One popular choice for robust metrics in the L_1 norm, also known as *total variation* (TV), which results in a convex energy whose global minimum can be found (Bruhn, Weickert, and Schnörr 2005; Papenberg, Bruhn, Brox *et al.* 2006). Anisotropic smoothness priors, which apply a different smoothness in the directions parallel and perpendicular to the image gradient, are another popular choice (Nagel and Enkelmann 1986; Sun, Roth, Lewis *et al.* 2008; Werlberger, Trobin, Pock *et al.* 2009). It is also possible to learn a set of better smoothness constraints (derivative filters and robust functions) from a set of paired flow and intensity images (Sun, Roth, Lewis *et al.* 2008). Additional details on some of these techniques are given by Baker, Black, Lewis *et al.* (2007) and Baker, Scharstein, Lewis *et al.* (2009).

Because of the large, two-dimensional search space in estimating flow, most algorithms use variations of gradient descent and coarse-to-fine continuation methods to minimize the global energy function. This contrasts starkly with stereo matching (which is an “easier” one-dimensional disparity estimation problem), where combinatorial optimization techniques have been the method of choice for the last decade.

¹² Robust brightness metrics (Section 8.1, (8.2)) can also help improve the performance of window-based approaches (Black and Anandan 1996).

Optical flow evaluation results		Statistics:		Average	SD	R0.5	R1.0	R2.0	A50	A75	A95
	Error type:	endpoint	angle	interpolation	normalized interpolation						
Average endpoint error		Army (Hidden texture) GT im0 im1	Meuron (Hidden texture) GT im0 im1	Schefflera (Hidden texture) GT im0 im1	Wooden (Hidden texture) GT im0 im1	Grove (Synthetic) GT im0 im1	Urban (Synthetic) GT im0 im1	Yosemite (Synthetic) GT im0 im1	Teddy (Stereo) GT im0 im1		
avg. rank		all disc untext	all disc untext	all disc untext	all disc untext	all disc untext	all disc untext	all disc untext	all disc untext	all disc untext	all disc untext
Adaptive [20]	4.4	0.09 0.26 0.061	0.23 0.78 0.16	0.58 1.25 0.213	0.18 0.93 0.101	0.883 1.25 0.73	0.503 1.28 0.313	0.14 0.16 0.12	0.22 0.22	0.65 1.37	0.3 0.794
Complementary OF [21]	5.7	0.11 0.28 0.10 9	0.18 0.63 0.12 1	0.13 0.75 0.18	0.19 0.97 0.12 3	0.97 1.31 0.10 1	1.78 20 1.73 0.87 14	0.11 0.12 0.22 10	0.11 0.12 0.22 10	0.68 1.48	0.95 8
Aniso. Huber-L1 [22]	5.8	0.10 0.28 0.08 3	0.31 0.88 0.28 12	0.19 0.11 0.13 29	0.20 0.94 0.13 5	0.842 1.20 0.70 2	0.391 1.23 0.28 1	0.17 0.15 0.19 27	0.27 0.26 0.27 16	0.64 1.36	0.79 4
DPOF [18]	6.1	0.13 0.35 0.12 0.09 4	0.25 0.79 0.15 0.09	0.19 0.41 0.21 3	0.19 0.62 0.15 1 1	0.74 1.09 0.49 1	0.66 1.80 10 0.63 8	0.19 0.17 0.14 35	0.20 0.20 0.14 35	0.50 1.	0.81 0.55 1
TV-L1-improved [17]	7.2	0.09 0.26 0.07 2	0.20 0.73 0.16 2	0.53 1.18 0.22 6	0.21 1.27 0.11 11 2	0.904 1.31 0.76 2	1.51 14 0.93 11 0.84 11	0.18 0.17 0.14 31	0.20 0.20 0.14 31	0.73 1.62	0.87 7
CBF [12]	7.8	0.10 0.28 0.09 4	0.34 0.80 0.37 13	0.43 0.95 0.26 8	0.21 1.14 0.13 5	0.90 1.27 0.82 7	0.412 1.23 0.30 2	0.23 0.22 0.19 20	0.39 0.39	0.76 1.56	1.02 9
Brox et al. [5]	8.4	0.11 0.35 0.28 0.11 2	0.27 0.93 0.10 22 9	0.39 0.94 0.24 7	0.24 0.25 0.12 15 3	1.10 13 0.92 14 3 17	0.89 1.78 0.55 7	0.10 0.12 0.13 4	0.11 0.11 0.13 4	0.91 1.18	1.32 11.13 2
Rannacher [23]	8.5	0.11 0.36 0.31 0.09 4	0.26 0.84 0.21 17 6	0.57 1.27 0.15 26 8	0.24 0.32 0.14 13 5	0.91 1.33 0.78 2 3	1.49 13 0.95 13 0.78 9	0.15 0.14 0.17 2 6	0.23 0.23 0.16 13	0.66 1.58	0.86 6
F-TV-L1 [15]	8.8	0.14 0.35 0.12 0.14 16	0.34 0.92 0.18 26 11	0.59 14 0.19 10 26 8	0.27 1.36 0.15 16 12	0.90 1.30 0.76 8 6	0.54 1.62 0.36 4 3	0.13 0.16 0.15 9 0	0.20 0.20 0.15 9 0	0.68 1.56	0.66 2
Second-order prior [8]	9.0	0.11 0.35 0.31 0.09 24	0.26 0.93 0.10 20 7	0.57 1.25 0.14 26 8	0.20 0.46 0.12 13 2	0.94 1.34 0.83 9 8	0.61 1.93 11 0.47 6	0.20 0.18 0.16 12 0.34 19	0.27 0.27 0.16 12 0.34 19	0.77 1.10	1.40 1.07 1.07 19
Fusion [6]	9.4	0.11 0.34 0.10 10 9	0.19 0.69 0.16 2	0.29 0.86 0.23 6	0.20 1.19 0.10 14 9	1.07 11 1.42 12 22 13	1.35 10 1.49 5 0.86 13	0.20 0.18 0.20 21 0.26 13	0.17 0.17 0.20 16 1.39 16		
Dynamic MRF [7]	11.1	0.12 0.11 0.34 0.10 11 1	0.22 0.89 0.16 2	0.46 1.13 0.20 2	0.24 0.29 1.13 0.14 9	1.11 14 1.52 1.17 13 12	1.54 1.25 2.37 0.93 0.13 16	0.16 0.12 0.2 0.31 17	0.27 0.23 0.20 16 1.67 17		
SegOF [10]	11.7	0.15 0.14 0.36 0.14 10 9	0.57 15 1.16 15 0.59 19	0.68 1.24 12 0.64 14	0.32 0.86 0.26 15	1.18 17 1.50 16 1.47 16	1.63 18 2.09 14 0.96 16	0.08 0.13 0.14 0.12 2	0.70 0.7 1.50 5	0.69 3	
Learning Flow [11]	13.3	0.11 0.15 0.32 0.09 24	0.29 0.99 0.13 23 10	0.55 12 0.14 22 0.29 12	0.36 0.16 0.15 27 15	1.25 19 1.64 21 0.41 16	1.57 15 2.32 19 0.85 12	0.14 0.18 0.18 0.24 12	0.19 0.20 0.18 12 1.27 13		
Filter Flow [19]	14.3	0.17 0.16 0.39 0.16 13 14	0.43 14 0.19 0.14 38 14	0.75 16 1.34 0.16 0.78 19	0.70 19 1.54 0.68 16 19	1.13 16 1.38 11 0.51 19	0.57 1.32 0.44 45 22	0.22 0.23 0.23 0.26 13	0.96 1.66 1.11 1.21 11		
GraphCuts [14]	14.5	0.16 0.15 0.38 0.15 0.14 16	0.59 18 0.16 36 0.46 15	0.50 1.07 0.64 14 6	0.26 12 1.14 0.18 17 3	0.96 8 1.35 10 0.84 20	2.25 23 1.79 0.22 21 22	0.22 0.20 0.17 14 0.43 22	0.17 0.22 0.15 0.15 17 1.58 19		
Black & Anandan [4]	15.0	0.18 0.17 0.42 0.17 0.19 16	0.58 17 0.13 17 0.50 16	0.95 19 1.58 0.70 16	0.49 17 1.59 0.45 17	1.08 12 1.42 13 22 13	1.43 11 2.28 17 0.83 10	0.15 0.12 0.17 14 0.17 6	0.11 0.16 0.19 0.17 11 1.16 19	0.98 1.43 1.30 14	
SPSA-learn [13]	15.7	0.17 0.17 0.45 0.18 0.17 17	0.57 15 0.13 18 0.51 17	0.84 17 1.50 0.70 17 2	0.52 18 1.64 0.19 14 9	1.12 15 1.42 13 1.39 15	1.75 19 2.14 15 0.60 20	0.13 0.16 0.13 0.14 19 7	0.13 0.16 0.12 0.2 0.31 17	0.32 0.20 0.18 17 1.73 18	
GroupFlow [9]	15.9	0.21 0.19 0.51 0.19 0.21 19	0.79 21 0.16 0.21 0.72 21	0.88 16 1.64 0.19 0.74 18	0.30 14 0.17 0.07 0.26 15	1.29 22 1.81 0.82 0.77 21	1.94 21 0.23 0.18 10 13 22	0.11 0.14 0.17 0.17 0.19 7	0.16 0.13 0.16 0.13 0.15 19	0.27 0.23 0.18 10 13 1.35 19	
2D-CLG [1]	17.4	0.28 0.21 0.62 0.22 0.21 19	0.67 20 1.21 0.16 0.70 20	1.12 21 1.80 0.19 0.99 22	1.07 22 0.26 01 1.12 22	1.23 18 1.52 17 0.62 22	1.54 12 2.15 16 0.96 18	0.10 0.12 0.11 0.11 0.16 4	0.13 0.18 0.20 0.22 0.26 13	1.38 20 0.26 19 1.83 20	
Horn & Schunck [3]	18.6	0.22 0.20 0.55 0.20 0.22 19	0.61 19 0.15 0.20 0.52 18	0.10 20 1.73 0.20 0.80 19	0.78 20 0.20 0.20 0.77 20	1.26 20 1.58 0.15 0.55 20	1.43 11 2.59 22 0.10 0.08	0.16 0.14 0.18 0.18 0.15 3	0.15 0.12 0.15 0.15 0.15 3	1.51 21 0.25 0.21 1.88 21	
TI-DOFE [24]	19.6	0.38 0.23 0.64 0.23 0.47 20	1.16 22 1.72 22 0.26 22	1.39 23 0.26 04 1.17 23	1.29 22 2.21 23 1.41 23	1.27 21 1.61 0.20 1.57 21	1.28 9 2.57 21 0.10 0.19	0.13 0.16 0.15 0.16 0.16 4	0.18 0.22 0.21 0.22 0.23 2	1.87 22 0.27 0.22 2.53 22	
FOLKI [16]	22.6	0.29 0.22 0.73 0.24 0.33 22	0.52 23 0.19 0.24 0.18 20	0.22 23 0.20 0.03 0.95 21	0.99 21 0.22 0.02 1.08 21	1.53 23 0.18 2.05 0.27 23	2.14 23 0.23 0.24 1.60 23	0.26 0.23 0.21 0.22 0.22 68 83	0.26 0.27 0.23 0.27 0.23 43 22	2.67 23 0.27 0.23 43 22	
Pyramid LK [2]	23.7	0.39 0.24 0.61 0.21 0.61 24	1.67 24 1.78 23 0.20 24	1.50 24 1.97 22 1.38 24	1.57 24 2.39 24 1.78 24	2.94 24 3.72 24 0.98 24	3.33 24 2.74 23 0.24 24	0.30 0.24 0.24 0.24 0.24 73 24	0.38 0.20 0.24 0.24 0.24 73 24	3.80 24 0.20 0.24 0.24 4.88 24	

Move the mouse over the numbers in the table to see the corresponding images. Click to compare with the ground truth.



Figure 8.12 Evaluation of the results of 24 optical flow algorithms, October 2009, <http://vision.middlebury.edu/flow/>, (Baker, Scharstein, Lewis *et al.* 2009). By moving the mouse pointer over an underlined performance score, the user can interactively view the corresponding flow and error maps. Clicking on a score toggles between the computed and ground truth flows. Next to each score, the corresponding rank in the current column is indicated by a smaller blue number. The minimum (best) score in each column is shown in boldface. The table is sorted by the average rank (computed over all 24 columns, three region masks for each of the eight sequences). The average rank serves as an *approximate* measure of performance *under the selected metric/statistic*.

Fortunately, combinatorial optimization methods based on Markov random fields are beginning to appear and tend to be among the better-performing methods on the recently released optical flow database (Baker, Black, Lewis *et al.* 2007).¹³

Examples of such techniques include the one developed by Glockner, Paragios, Komodakis *et al.* (2008), who use a coarse-to-fine strategy with per-pixel 2D uncertainty estimates, which are then used to guide the refinement and search at the next finer level. Instead of using gradient descent to refine the flow estimates, a combinatorial search over discrete displacement labels (which is able to find better energy minima) is performed using their Fast-PD algorithm (Komodakis, Tziritas, and Paragios 2008).

Lempitsky, Roth, and Rother. (2008) use fusion moves (Lempitsky, Rother, and Blake 2007) over proposals generated from basic flow algorithms (Horn and Schunck 1981; Lucas and Kanade 1981) to find good solutions. The basic idea behind fusion moves is to replace portions of the current best estimate with hypotheses generated by more basic techniques (or their shifted versions) and to alternate them with local gradient descent for better energy minimization.

The field of accurate motion estimation continues to evolve at a rapid pace, with significant advances in performance occurring every year. The optical flow evaluation Web site (<http://vision.middlebury.edu/flow/>) is a good source of pointers to high-performing recently developed algorithms (Figure 8.12).

8.4.1 Multi-frame motion estimation

So far, we have looked at motion estimation as a two-frame problem, where the goal is to compute a motion field that aligns pixels from one image with those in another. In practice, motion estimation is usually applied to video, where a whole sequence of frames is available to perform this task.

One classic approach to multi-frame motion is to *filter* the spatio-temporal volume using oriented or steerable filters (Heeger 1988), in a manner analogous to oriented edge detection (Section 3.2.3). Figure 8.13 shows two frames from the commonly used *flower garden* sequence, as well as a horizontal slice through the spatio-temporal volume, i.e., the 3D volume created by stacking all of the video frames together. Because the pixel motion is mostly horizontal, the slopes of individual (textured) pixel tracks, which correspond to their horizontal velocities, can clearly be seen. Spatio-temporal filtering uses a 3D volume around each pixel to determine the best orientation in space–time, which corresponds directly to a pixel’s velocity.

Unfortunately, in order to obtain reasonably accurate velocity estimates everywhere in an image, spatio-temporal filters have moderately large extents, which severely degrades the quality of their estimates near motion discontinuities. (This same problem is endemic in 2D window-based motion estimators.) An alternative to full spatio-temporal filtering is to estimate more local spatio-temporal derivatives and use them inside a global optimization framework to fill in textureless regions (Bruhn, Weickert, and Schnörr 2005; Govindu 2006).

Another alternative is to simultaneously estimate multiple motion estimates, while also optionally reasoning about occlusion relationships (Szeliski 1999). Figure 8.13c shows schematically one potential approach to this problem. The horizontal arrows show the locations of

¹³ <http://vision.middlebury.edu/flow/>.

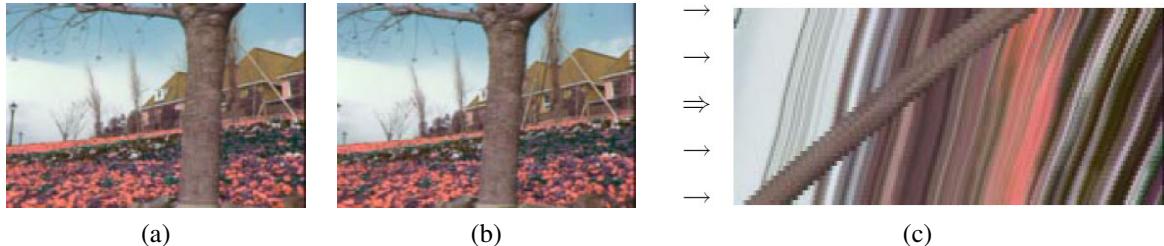


Figure 8.13 Slice through a spatio-temporal volume (Szeliski 1999) © 1999 IEEE: (a–b) two frames from the *flower garden* sequence; (c) a horizontal slice through the complete spatio-temporal volume, with the arrows indicating locations of potential key frames where flow is estimated. Note that the colors for the flower garden sequence are incorrect; the correct colors (yellow flowers) are shown in Figure 8.15.

keyframes s where motion is estimated, while other slices indicate video frames t whose colors are matched with those predicted by interpolating between the keyframes. Motion estimation can be cast as a global energy minimization problem that simultaneously minimizes brightness compatibility and flow compatibility terms between keyframes and other frames, in addition to using robust smoothness terms.

The multi-view framework is potentially even more appropriate for rigid scene motion (multi-view stereo) (Section 11.6), where the unknowns at each pixel are disparities and occlusion relationships can be determined directly from pixel depths (Szeliski 1999; Kolmogorov and Zabih 2002). However, it may also be applicable to general motion, with the addition of models for object accelerations and occlusion relationships.

8.4.2 Application: Video denoising

Video denoising is the process of removing noise and other artifacts such as scratches from film and video (Kokaram 2004). Unlike single image denoising, where the only information available is in the current picture, video denoisers can average or borrow information from adjacent frames. However, in order to do this without introducing blur or jitter (irregular motion), they need accurate per-pixel motion estimates.

Exercise 8.7 lists some of the steps required, which include the ability to determine if the current motion estimate is accurate enough to permit averaging with other frames. Gai and Kang (2009) describe their recently developed restoration process, which involves a series of additional steps to deal with the special characteristics of vintage film.

8.4.3 Application: De-interlacing

Another commonly used application of per-pixel motion estimation is video de-interlacing, which is the process of converting a video taken with alternating fields of even and odd lines to a non-interlaced signal that contains both fields in each frame (de Haan and Bellers 1998). Two simple de-interlacing techniques are *bob*, which copies the line above or below the missing line from the same field, and *weave*, which copies the corresponding line from the field before or after. The names come from the visual artifacts generated by these two simple techniques: *bob* introduces an up-and-down bobbing motion along strong horizontal

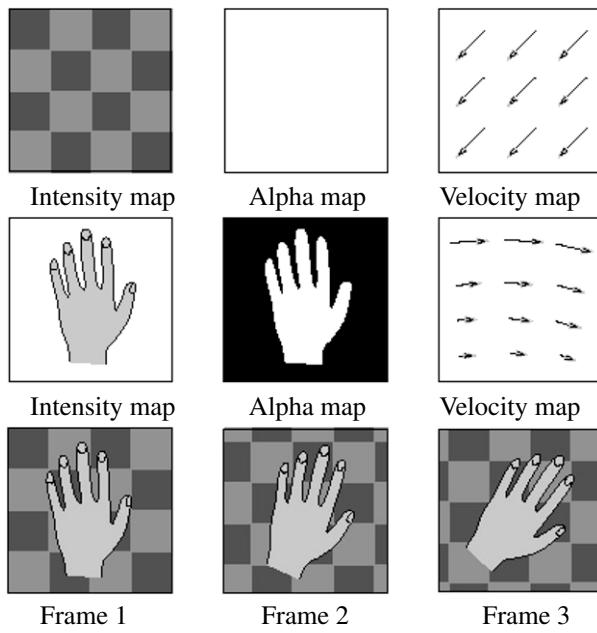


Figure 8.14 Layered motion estimation framework (Wang and Adelson 1994) © 1994 IEEE: The top two rows describe the two layers, each of which consists of an intensity (color) image, an alpha mask (black=transparent), and a parametric motion field. The layers are composited with different amounts of motion to recreate the video sequence.

lines; weave can lead to a “zippering” effect along horizontally translating edges. Replacing these copy operators with averages can help but does not completely remove these artifacts.

A wide variety of improved techniques have been developed for this process, which is often embedded in specialized DSP chips found inside video digitization boards in computers (since broadcast video is often interlaced, while computer monitors are not). A large class of these techniques estimates local per-pixel motions and interpolates the missing data from the information available in spatially and temporally adjacent fields. Dai, Baker, and Kang (2009) review this literature and propose their own algorithm, which selects among seven different interpolation functions at each pixel using an MRF framework.

8.5 Layered motion

In many situations, visual motion is caused by the movement of a small number of objects at different depths in the scene. In such situations, the pixel motions can be described more succinctly (and estimated more reliably) if pixels are grouped into appropriate objects or *layers* (Wang and Adelson 1994).

Figure 8.14 shows this approach schematically. The motion in this sequence is caused by the translational motion of the checkered background and the rotation of the foreground hand. The complete motion sequence can be reconstructed from the appearance of the foreground and background elements, which can be represented as alpha-matted images (*sprites* or *video objects*) and the parametric motion corresponding to each layer. Displacing and compositing

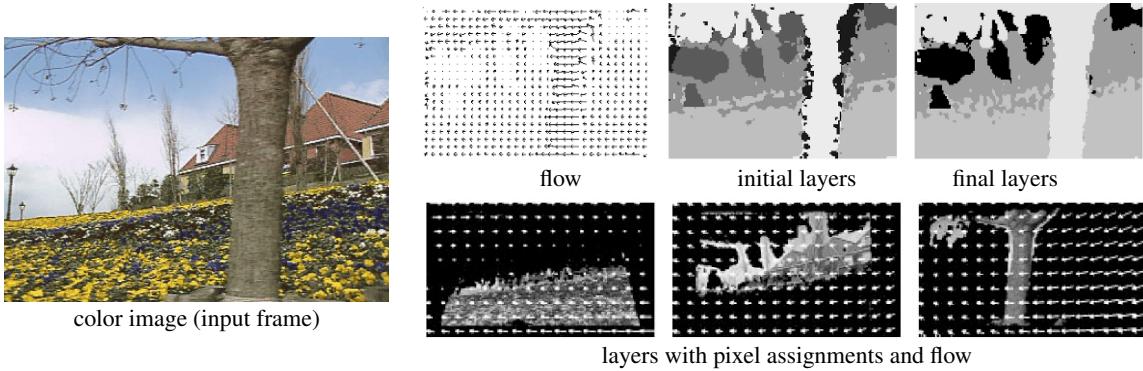


Figure 8.15 Layered motion estimation results (Wang and Adelson 1994) © 1994 IEEE.

these layers in back to front order (Section 3.1.3) recreates the original video sequence.

Layered motion representations not only lead to compact representations (Wang and Adelson 1994; Lee, ge Chen, lung Bruce Lin *et al.* 1997), but they also exploit the information available in multiple video frames, as well as accurately modeling the appearance of pixels near motion discontinuities. This makes them particularly suited as a representation for image-based rendering (Section 13.2.1) (Shade, Gortler, He *et al.* 1998; Zitnick, Kang, Uyttendaele *et al.* 2004) as well as object-level video editing.

To compute a layered representation of a video sequence, Wang and Adelson (1994) first estimate affine motion models over a collection of non-overlapping patches and then cluster these estimates using k-means. They then alternate between assigning pixels to layers and recomputing motion estimates for each layer using the assigned pixels, using a technique first proposed by Darrell and Pentland (1991). Once the parametric motions and pixel-wise layer assignments have been computed for each frame independently, layers are constructed by warping and merging the various layer pieces from all of the frames together. Median filtering is used to produce sharp composite layers that are robust to small intensity variations, as well as to infer occlusion relationships between the layers. Figure 8.15 shows the results of this process on the *flower garden* sequence. You can see both the initial and final layer assignments for one of the frames, as well as the composite flow and the alpha-matted layers with their corresponding flow vectors overlaid.

In follow-on work, Weiss and Adelson (1996) use a formal probabilistic mixture model to infer both the optimal number of layers and the per-pixel layer assignments. Weiss (1997) further generalizes this approach by replacing the per-layer affine motion models with smooth regularized per-pixel motion estimates, which allows the system to better handle curved and undulating layers, such as those seen in most real-world sequences.

The above approaches, however, still make a distinction between estimating the motions and layer assignments and then later estimating the layer colors. In the system described by Baker, Szeliski, and Anandan (1998), the generative model illustrated in Figure 8.14 is generalized to account for real-world rigid motion scenes. The motion of each frame is described using a 3D camera model and the motion of each layer is described using a 3D plane equation plus per-pixel residual depth offsets (the *plane plus parallax* representation (Section 2.1.5)). The initial layer estimation proceeds in a manner similar to that of Wang and Adelson (1994),

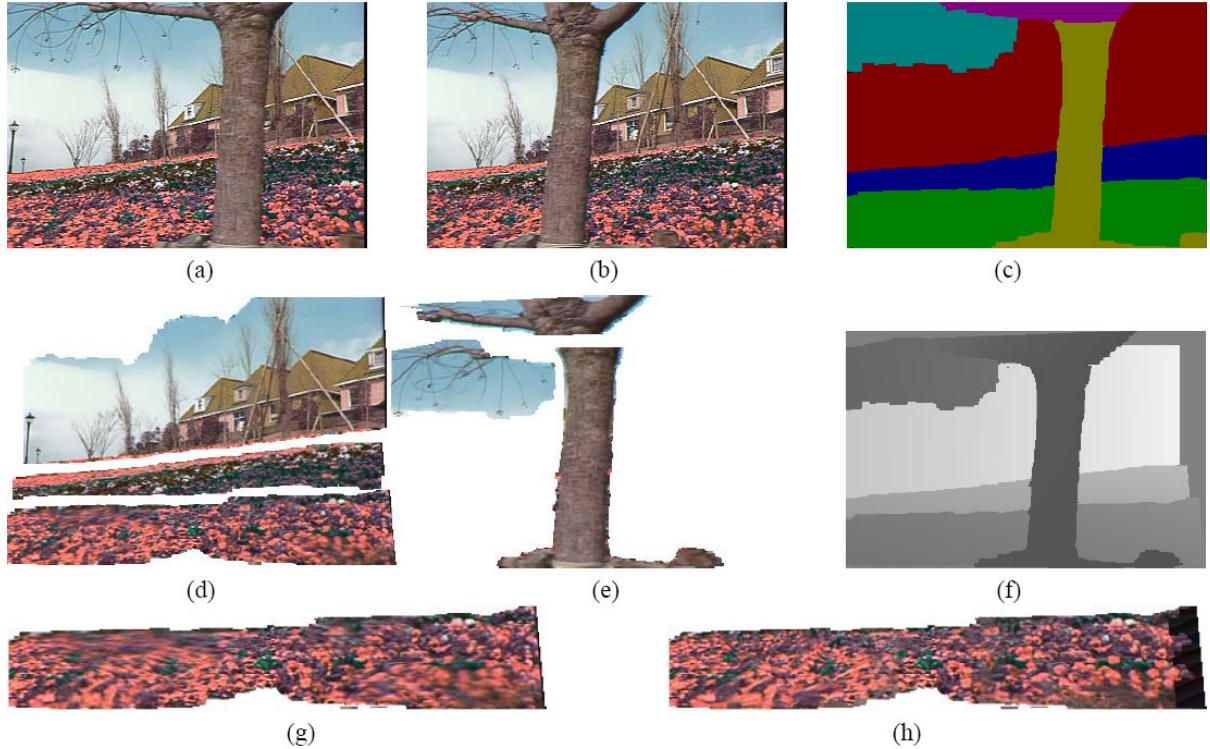


Figure 8.16 Layered stereo reconstruction (Baker, Szeliski, and Anandan 1998) © 1998 IEEE: (a) first and (b) last input images; (c) initial segmentation into six layers; (d) and (e) the six layer sprites; (f) depth map for planar sprites (darker denotes closer); front layer (g) before and (h) after residual depth estimation. Note that the colors for the flower garden sequence are incorrect; the correct colors (yellow flowers) are shown in Figure 8.15.

o

except that rigid planar motions (homographies) are used instead of affine motion models. The final model refinement, however, jointly re-optimizes the layer pixel color and opacity values L_l and the 3D depth, plane, and motion parameters z_l , \mathbf{n}_l , and \mathbf{P}_t by minimizing the discrepancy between the re-synthesized and observed motion sequences (Baker, Szeliski, and Anandan 1998).

Figure 8.16 shows the final results obtained with this algorithm. As you can see, the motion boundaries and layer assignments are much crisper than those in Figure 8.15. Because of the per-pixel depth offsets, the individual layer color values are also sharper than those obtained with affine or planar motion models. While the original system of Baker, Szeliski, and Anandan (1998) required a rough initial assignment of pixels to layers, Torr, Szeliski, and Anandan (2001) describe automated Bayesian techniques for initializing this system and determining the optimal number of layers.

Layered motion estimation continues to be an active area of research. Representative papers in this area include (Sawhney and Ayer 1996; Jojic and Frey 2001; Xiao and Shah 2005; Kumar, Torr, and Zisserman 2008; Thayananthan, Iwasaki, and Cipolla 2008; Schoenemann and Cremers 2008).

Of course, layers are not the only way to introduce segmentation into motion estimation.

A large number of algorithms have been developed that alternate between estimating optic flow vectors and segmenting them into coherent regions (Black and Jepson 1996; Ju, Black, and Jepson 1996; Chang, Tekalp, and Sezan 1997; Mémin and Pérez 2002; Cremers and Soatto 2005). Some of the more recent techniques rely on first segmenting the input color images and then estimating per-segment motions that produce a coherent motion field while also modeling occlusions (Zitnick, Kang, Uyttendaele *et al.* 2004; Zitnick, Jojic, and Kang 2005; Stein, Hoiem, and Hebert 2007; Thayanathan, Iwasaki, and Cipolla 2008).

8.5.1 Application: Frame interpolation

Frame interpolation is another widely used application of motion estimation, often implemented in the same circuitry as de-interlacing hardware required to match an incoming video to a monitor's actual refresh rate. As with de-interlacing, information from novel in-between frames needs to be interpolated from preceding and subsequent frames. The best results can be obtained if an accurate motion estimate can be computed at each unknown pixel's location. However, in addition to computing the motion, occlusion information is critical to prevent colors from being contaminated by moving foreground objects that might obscure a particular pixel in a preceding or subsequent frame.

In a little more detail, consider Figure 8.13c and assume that the arrows denote keyframes between which we wish to interpolate additional images. The orientations of the streaks in this figure encode the velocities of individual pixels. If the same motion estimate \mathbf{u}_0 is obtained at location \mathbf{x}_0 in image I_0 as is obtained at location $\mathbf{x}_0 + \mathbf{u}_0$ in image I_1 , the flow vectors are said to be *consistent*. This motion estimate can be transferred to location $\mathbf{x}_0 + t\mathbf{u}_0$ in the image I_t being generated, where $t \in (0, 1)$ is the time of interpolation. The final color value at pixel $\mathbf{x}_0 + t\mathbf{u}_0$ can be computed as a linear blend,

$$I_t(\mathbf{x}_0 + t\mathbf{u}_0) = (1 - t)I_0(\mathbf{x}_0) + tI_1(\mathbf{x}_0 + \mathbf{u}_0). \quad (8.72)$$

If, however, the motion vectors are different at corresponding locations, some method must be used to determine which is correct and which image contains colors that are occluded. The actual reasoning is even more subtle than this. One example of such an interpolation algorithm, based on earlier work in depth map interpolation (Shade, Gortler, He *et al.* 1998; Zitnick, Kang, Uyttendaele *et al.* 2004) which is the one used in the flow evaluation paper of Baker, Black, Lewis *et al.* (2007); Baker, Scharstein, Lewis *et al.* (2009). An even higher-quality frame interpolation algorithm, which uses gradient-based reconstruction, is presented by Mahajan, Huang, Matusik *et al.* (2009).

8.5.2 Transparent layers and reflections

A special case of layered motion that occurs quite often is transparent motion, which is usually caused by reflections seen in windows and picture frames (Figures 8.17 and 8.18).

Some of the early work in this area handles transparent motion by either just estimating the component motions (Shizawa and Mase 1991; Bergen, Burt, Hingorani *et al.* 1992; Darrell and Simoncelli 1993; Irani, Rousso, and Peleg 1994) or by assigning individual pixels to competing motion layers (Darrell and Pentland 1995; Black and Anandan 1996; Ju, Black, and Jepson 1996), which is appropriate for scenes partially seen through a fine occluder (e.g., foliage). However, to accurately separate truly transparent layers, a better model for



Figure 8.17 Light reflecting off the transparent glass of a picture frame: (a) first image from the input sequence; (b) dominant motion layer *min-composite*; (c) secondary motion residual layer *max-composite*; (d–e) final estimated picture and reflection layers. The original images are from Black and Anandan (1996), while the separated layers are from Szeliski, Avidan, and Anandan (2000) © 2000 IEEE.

motion due to reflections is required. Because of the way that light is both reflected from and transmitted through a glass surface, the correct model for reflections is an *additive* one, where each moving layer contributes some intensity to the final image (Szeliski, Avidan, and Anandan 2000).

If the motions of the individual layers are known, the recovery of the individual layers is a simple constrained least squares problem, with the individual layer images are constrained to be positive. However, this problem can suffer from extended low-frequency ambiguities, especially if either of the layers lacks dark (black) pixels or the motion is uni-directional. In their paper, Szeliski, Avidan, and Anandan (2000) show that the simultaneous estimation of the motions and layer values can be obtained by alternating between robustly computing the motion layers and then making conservative (upper- or lower-bound) estimates of the layer intensities. The final motion and layer estimates can then be polished using gradient descent on a joint constrained least squares formulation similar to (Baker, Szeliski, and Anandan 1998), where the *over* compositing operator is replaced with addition.

Figures 8.17 and 8.18 show the results of applying these techniques to two different picture frames with reflections. Notice how, in the second sequence, the amount of reflected light is quite low compared to the transmitted light (the picture of the girl) and yet the algorithm is still able to recover both layers.

Unfortunately, the simple parametric motion models used in (Szeliski, Avidan, and Anandan 2000) are only valid for planar reflectors and scenes with shallow depth. The extension of these techniques to curved reflectors and scenes with significant depth has also been studied (Swaminathan, Kang, Szeliski *et al.* 2002; Criminisi, Kang, Swaminathan *et al.* 2005), as has the extension to scenes with more complex 3D depth (Tsin, Kang, and Szeliski 2006).

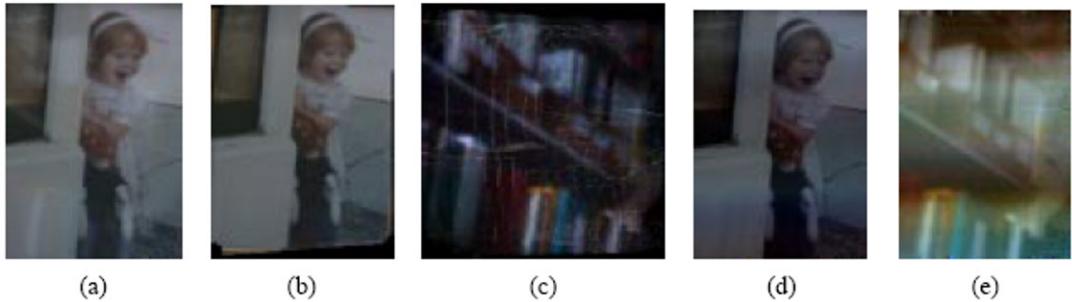


Figure 8.18 Transparent motion separation (Szeliski, Avidan, and Anandan 2000) © 2000 IEEE: (a) first image from input sequence; (b) dominant motion layer *min-composite*; (c) secondary motion residual layer *max-composite*; (d–e) final estimated picture and reflection layers. Note that the reflected layers in (c) and (e) are doubled in intensity to better show their structure.

8.6 Additional reading

Some of the earliest algorithms for motion estimation were developed for motion-compensated video coding (Netravali and Robbins 1979) and such techniques continue to be used in modern coding standards such as MPEG, H.263, and H.264 (Le Gall 1991; Richardson 2003).¹⁴ In computer vision, this field was originally called *image sequence analysis* (Huang 1981). Some of the early seminal papers include the variational approaches developed by Horn and Schunck (1981) and Nagel and Enkelmann (1986), and the patch-based translational alignment technique developed by Lucas and Kanade (1981). Hierarchical (coarse-to-fine) versions of such algorithms were developed by Quam (1984), Anandan (1989), and Bergen, Anandan, Hanna *et al.* (1992), although they have also long been used in motion estimation for video coding.

Translational motion models were generalized to affine motion by Rehg and Witkin (1991), Fuh and Maragos (1991), and Bergen, Anandan, Hanna *et al.* (1992) and to quadric reference surfaces by Shashua and Toelg (1997) and Shashua and Wexler (2001)—see Baker and Matthews (2004) for a nice review. Such parametric motion estimation algorithms have found widespread application in video summarization (Teodosio and Bender 1993; Irani and Anandan 1998), video stabilization (Hansen, Anandan, Dana *et al.* 1994; Srinivasan, Chellappa, Veeraraghavan *et al.* 2005; Matsushita, Ofek, Ge *et al.* 2006), and video compression (Irani, Hsu, and Anandan 1995; Lee, ge Chen, lung Bruce Lin *et al.* 1997). Surveys of parametric image registration include those by Brown (1992), Zitov'aa and Flusser (2003), Goshtasby (2005), and Szeliski (2006a).

Good general surveys and comparisons of optic flow algorithms include those by Aggarwal and Nandhakumar (1988), Barron, Fleet, and Beauchemin (1994), Otte and Nagel (1994), Mitiche and Bouthemy (1996), Stiller and Konrad (1999), McCane, Novins, Cranitch *et al.* (2001), Szeliski (2006a), and Baker, Black, Lewis *et al.* (2007). The topic of matching primitives, i.e., pre-transforming images using filtering or other techniques before matching, is treated in a number of papers (Anandan 1989; Bergen, Anandan, Hanna *et al.* 1992; Scharstein 1994; Zabih and Woodfill 1994; Cox, Roy, and Hingorani 1995; Viola and

¹⁴ <http://www.itu.int/rec/T-REC-H.264>.

Wells III 1997; Negahdaripour 1998; Kim, Kolmogorov, and Zabih 2003; Jia and Tang 2003; Papenberg, Bruhn, Brox *et al.* 2006; Seitz and Baker 2009). Hirschmüller and Scharstein (2009) compare a number of these approaches and report on their relative performance in scenes with exposure differences.

The publication of a new benchmark for evaluating optical flow algorithms (Baker, Black, Lewis *et al.* 2007) has led to rapid advances in the quality of estimation algorithms, to the point where new datasets may soon become necessary. According to their updated technical report (Baker, Scharstein, Lewis *et al.* 2009), most of the best performing algorithms use robust data and smoothness norms (often L_1 TV) and continuous variational optimization techniques, although some techniques use discrete optimization or segmentations (Papenberg, Bruhn, Brox *et al.* 2006; Trobin, Pock, Cremers *et al.* 2008; Xu, Chen, and Jia 2008; Lempitsky, Roth, and Rother 2008; Werlberger, Trobin, Pock *et al.* 2009; Lei and Yang 2009; Wedel, Cremers, Pock *et al.* 2009).

8.7 Exercises

Ex 8.1: Correlation Implement and compare the performance of the following correlation algorithms:

- sum of squared differences (8.1)
- sum of robust differences (8.2)
- sum of absolute differences (8.3)
- bias–gain compensated squared differences (8.9)
- normalized cross-correlation (8.11)
- windowed versions of the above (8.22–8.23)
- Fourier-based implementations of the above measures (8.18–8.20)
- phase correlation (8.24)
- gradient cross-correlation (Argyriou and Vlachos 2003).

Compare a few of your algorithms on different motion sequences with different amounts of noise, exposure variation, occlusion, and frequency variations (e.g., high-frequency textures, such as sand or cloth, and low-frequency images, such as clouds or motion-blurred video). Some datasets with illumination variation and ground truth correspondences (horizontal motion) can be found at <http://vision.middlebury.edu/stereo/data/> (the 2005 and 2006 datasets).

Some additional ideas, variants, and questions:

1. When do you think that phase correlation will outperform regular correlation or SSD? Can you show this experimentally or justify it analytically?
2. For the Fourier-based masked or windowed correlation and sum of squared differences, the results should be the same as the direct implementations. Note that you will have to expand (8.5) into a sum of pairwise correlations, just as in (8.22). (This is part of the exercise.)

3. For the bias–gain corrected variant of squared differences (8.9), you will also have to expand the terms to end up with a 3×3 (least squares) system of equations. If implementing the Fast Fourier Transform version, you will need to figure out how all of these entries can be evaluated in the Fourier domain.
4. (Optional) Implement some of the additional techniques studied by Hirschmüller and Scharstein (2009) and see if your results agree with theirs.

Ex 8.2: Affine registration Implement a coarse-to-fine direct method for affine and projective image alignment.

1. Does it help to use lower-order (simpler) models at coarser levels of the pyramid (Bergen, Anandan, Hanna *et al.* 1992)?
2. (Optional) Implement patch-based acceleration (Shum and Szeliski 2000; Baker and Matthews 2004).
3. See the Baker and Matthews (2004) survey for more comparisons and ideas.

Ex 8.3: Stabilization Write a program to *stabilize* an input video sequence. You should implement the following steps, as described in Section 8.2.1:

1. Compute the translation (and, optionally, rotation) between successive frames with robust outlier rejection.
2. Perform temporal high-pass filtering on the motion parameters to remove the low-frequency component (smooth the motion).
3. Compensate for the high-frequency motion, zooming in slightly (a user-specified amount) to avoid missing edge pixels.
4. (Optional) Do not zoom in, but instead borrow pixels from previous or subsequent frames to fill in.
5. (Optional) Compensate for images that are blurry because of fast motion by “stealing” higher frequencies from adjacent frames.

Ex 8.4: Optical flow Compute optical flow (spline-based or per-pixel) between two images, using one or more of the techniques described in this chapter.

1. Test your algorithms on the motion sequences available at <http://vision.middlebury.edu/flow/> or <http://people.csail.mit.edu/celiu/motionAnnotation/> and compare your results (visually) to those available on these Web sites. If you think your algorithm is competitive with the best, consider submitting it for formal evaluation.
2. Visualize the quality of your results by generating in-between images using frame interpolation (Exercise 8.5).
3. What can you say about the relative efficiency (speed) of your approach?

Ex 8.5: Automated morphing / frame interpolation Write a program to automatically morph between pairs of images. Implement the following steps, as sketched out in Section 8.5.1 and by Baker, Scharstein, Lewis *et al.* (2009):

1. Compute the flow both ways (previous exercise). Consider using a multi-frame ($n > 2$) technique to better deal with occluded regions.
2. For each intermediate (morphed) image, compute a set of flow vectors and which images should be used in the final composition.
3. Blend (cross-dissolve) the images and view with a sequence viewer.

Try this out on images of your friends and colleagues and see what kinds of morphs you get. Alternatively, take a video sequence and do a high-quality slow-motion effect. Compare your algorithm with simple cross-fading.

Ex 8.6: Motion-based user interaction Write a program to compute a low-resolution motion field in order to interactively control a simple application (Cutler and Turk 1998). For example:

1. Downsample each image using a pyramid and compute the optical flow (spline-based or pixel-based) from the previous frame.
2. Segment each training video sequence into different “actions” (e.g., hand moving inwards, moving up, no motion) and “learn” the velocity fields associated with each one. (You can simply find the mean and variance for each motion field or use something more sophisticated, such as a support vector machine (SVM).)
3. Write a recognizer that finds successive actions of approximately the right duration and hook it up to an interactive application (e.g., a sound generator or a computer game).
4. Ask your friends to test it out.

Ex 8.7: Video denoising Implement the algorithm sketched in Application 8.4.2. Your algorithm should contain the following steps:

1. Compute accurate per-pixel flow.
2. Determine which pixels in the reference image have good matches with other frames.
3. Either average all of the matched pixels or choose the sharpest image, if trying to compensate for blur. Don’t forget to use regular single-frame denoising techniques as part of your solution, (see Section 3.4.4, Section 3.7.3, and Exercise 3.11).
4. Devise a fall-back strategy for areas where you don’t think the flow estimates are accurate enough.

Ex 8.8: Motion segmentation Write a program to segment an image into separately moving regions or to reliably find motion boundaries.

Use the human-assisted motion segmentation database at <http://people.csail.mit.edu/celiu/motionAnnotation/> as some of your test data.

Ex 8.9: Layered motion estimation Decompose into separate layers (Section 8.5) a video sequence of a scene taken with a moving camera:

1. Find the set of dominant (affine or planar perspective) motions, either by computing them in blocks or finding a robust estimate and then iteratively re-fitting outliers.
2. Determine which pixels go with each motion.
3. Construct the layers by blending pixels from different frames.
4. (Optional) Add per-pixel residual flows or depths.
5. (Optional) Refine your estimates using an iterative global optimization technique.
6. (Optional) Write an interactive renderer to generate in-between frames or view the scene from different viewpoints (Shade, Gortler, He *et al.* 1998).
7. (Optional) Construct an *unwrap mosaic* from a more complex scene and use this to do some video editing (Rav-Acha, Kohli, Fitzgibbon *et al.* 2008).

Ex 8.10: Transparent motion and reflection estimation Take a video sequence looking through a window (or picture frame) and see if you can remove the reflection in order to better see what is inside.

The steps are described in Section 8.5.2 and by Szeliski, Avidan, and Anandan (2000). Alternative approaches can be found in work by Shizawa and Mase (1991), Bergen, Burt, Hingorani *et al.* (1992), Darrell and Simoncelli (1993), Darrell and Pentland (1995), Irani, Rousso, and Peleg (1994), Black and Anandan (1996), and Ju, Black, and Jepson (1996).

Chapter 9

Image stitching

9.1	Motion models	378
9.1.1	Planar perspective motion	379
9.1.2	<i>Application:</i> Whiteboard and document scanning	379
9.1.3	Rotational panoramas	380
9.1.4	Gap closing	382
9.1.5	<i>Application:</i> Video summarization and compression	383
9.1.6	Cylindrical and spherical coordinates	385
9.2	Global alignment	387
9.2.1	Bundle adjustment	388
9.2.2	Parallax removal	391
9.2.3	Recognizing panoramas	392
9.2.4	Direct vs. feature-based alignment	393
9.3	Compositing	396
9.3.1	Choosing a compositing surface	396
9.3.2	Pixel selection and weighting (de-ghosting)	398
9.3.3	<i>Application:</i> Photomontage	403
9.3.4	Blending	403
9.4	Additional reading	406
9.5	Exercises	407



Figure 9.1 Image stitching: (a) portion of a cylindrical panorama and (b) a spherical panorama constructed from 54 photographs (Szeliski and Shum 1997) © 1997 ACM; (c) a multi-image panorama automatically assembled from an unordered photo collection; a multi-image stitch (d) without and (e) with moving object removal (Uyttendaele, Eden, and Szeliski 2001) © 2001 IEEE.

Algorithms for aligning images and stitching them into seamless photo-mosaics are among the oldest and most widely used in computer vision (Milgram 1975; Peleg 1981). Image stitching algorithms create the high-resolution photo-mosaics used to produce today's digital maps and satellite photos. They also come bundled with most digital cameras and can be used to create beautiful ultra wide-angle panoramas.

Image stitching originated in the photogrammetry community, where more manually intensive methods based on surveyed *ground control points* or manually registered *tie points* have long been used to register aerial photos into large-scale photo-mosaics (Slama 1980). One of the key advances in this community was the development of *bundle adjustment* algorithms (Section 7.4), which could simultaneously solve for the locations of all of the camera positions, thus yielding globally consistent solutions (Triggs, McLauchlan, Hartley *et al.* 1999). Another recurring problem in creating photo-mosaics is the elimination of visible seams, for which a variety of techniques have been developed over the years (Milgram 1975, 1977; Peleg 1981; Davis 1998; Agarwala, Dontcheva, Agrawala *et al.* 2004)

In film photography, special cameras were developed in the 1990s to take ultra-wide-angle panoramas, often by exposing the film through a vertical slit as the camera rotated on its axis (Meehan 1990). In the mid-1990s, image alignment techniques started being applied to the construction of wide-angle seamless panoramas from regular hand-held cameras (Mann and Picard 1994; Chen 1995; Szeliski 1996). More recent work in this area has addressed the need to compute globally consistent alignments (Szeliski and Shum 1997; Sawhney and Kumar 1999; Shum and Szeliski 2000), to remove "ghosts" due to parallax and object movement (Davis 1998; Shum and Szeliski 2000; Uyttendaele, Eden, and Szeliski 2001; Agarwala, Dontcheva, Agrawala *et al.* 2004), and to deal with varying exposures (Mann and Picard 1994; Uyttendaele, Eden, and Szeliski 2001; Levin, Zomet, Peleg *et al.* 2004; Agarwala, Dontcheva, Agrawala *et al.* 2004; Eden, Uyttendaele, and Szeliski 2006; Kopf, Uyttendaele, Deussen *et al.* 2007).¹ These techniques have spawned a large number of commercial stitching products (Chen 1995; Sawhney, Kumar, Gendel *et al.* 1998), of which reviews and comparisons can be found on the Web.²

While most of the earlier techniques worked by directly minimizing pixel-to-pixel dissimilarities, more recent algorithms usually extract a sparse set of features and match them to each other, as described in Chapter 4. Such feature-based approaches to image stitching have the advantage of being more robust against scene movement and are potentially faster, if implemented the right way. Their biggest advantage, however, is the ability to "recognize panoramas", i.e., to automatically discover the adjacency (overlap) relationships among an unordered set of images, which makes them ideally suited for fully automated stitching of panoramas taken by casual users (Brown and Lowe 2007).

What, then, are the essential problems in image stitching? As with image alignment, we must first determine the appropriate mathematical model relating pixel coordinates in one image to pixel coordinates in another; Section 9.1 reviews the basic models we have studied and presents some new motion models related specifically to panoramic image stitching. Next, we must somehow estimate the correct alignments relating various pairs (or collections) of images. Chapter 4 discussed how distinctive features can be found in each image and then

¹ A collection of some of these papers was compiled by Benosman and Kang (2001) and they are surveyed by Szeliski (2006a).

² The Photosynth Web site, <http://photosynth.net>, allows people to create and upload panoramas for free.

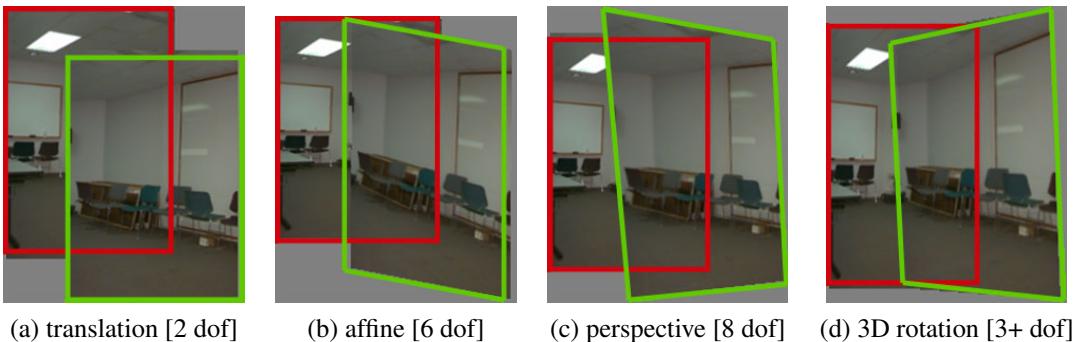


Figure 9.2 Two-dimensional motion models and how they can be used for image stitching.

efficiently matched to rapidly establish correspondences between pairs of images. Chapter 8 discussed how direct pixel-to-pixel comparisons combined with gradient descent (and other optimization techniques) can also be used to estimate these parameters. When multiple images exist in a panorama, bundle adjustment (Section 7.4) can be used to compute a globally consistent set of alignments and to efficiently discover which images overlap one another. In Section 9.2, we look at how each of these previously developed techniques can be modified to take advantage of the imaging setups commonly used to create panoramas.

Once we have aligned the images, we must choose a final compositing surface for warping the aligned images (Section 9.3.1). We also need algorithms to seamlessly cut and blend overlapping images, even in the presence of parallax, lens distortion, scene motion, and exposure differences (Section 9.3.2–9.3.4).

9.1 Motion models

Before we can register and align images, we need to establish the mathematical relationships that map pixel coordinates from one image to another. A variety of such *parametric motion models* are possible, from simple 2D transforms, to planar perspective models, 3D camera rotations, lens distortions, and mapping to non-planar (e.g., cylindrical) surfaces.

We already covered several of these models in Sections 2.1 and 6.1. In particular, we saw in Section 2.1.5 how the parametric motion describing the deformation of a planar surface as viewed from different positions can be described with an eight-parameter homography (2.71) (Mann and Picard 1994; Szeliski 1996). We also saw how a camera undergoing a pure rotation induces a different kind of homography (2.72).

In this section, we review both of these models and show how they can be applied to different stitching situations. We also introduce spherical and cylindrical compositing surfaces and show how, under favorable circumstances, they can be used to perform alignment using pure translations (Section 9.1.6). Deciding which alignment model is most appropriate for a given situation or set of data is a *model selection* problem (Hastie, Tibshirani, and Friedman 2001; Torr 2002; Bishop 2006; Robert 2007), an important topic we do not cover in this book.

9.1.1 Planar perspective motion

The simplest possible motion model to use when aligning images is to simply translate and rotate them in 2D (Figure 9.2a). This is exactly the same kind of motion that you would use if you had overlapping photographic prints. It is also the kind of technique favored by David Hockney to create the collages that he calls *joiners* (Zelnik-Manor and Perona 2007; Nomura, Zhang, and Nayar 2007). Creating such collages, which show visible seams and inconsistencies that add to the artistic effect, is popular on Web sites such as Flickr, where they more commonly go under the name *panography* (Section 6.1.2). Translation and rotation are also usually adequate motion models to compensate for small camera motions in applications such as photo and video stabilization and merging (Exercise 6.1 and Section 8.2.1).

In Section 6.1.3, we saw how the mapping between two cameras viewing a common plane can be described using a 3×3 homography (2.71). Consider the matrix \mathbf{M}_{10} that arises when mapping a pixel in one image to a 3D point and then back onto a second image,

$$\tilde{\mathbf{x}}_1 \sim \tilde{\mathbf{P}}_1 \tilde{\mathbf{P}}_0^{-1} \tilde{\mathbf{x}}_0 = \mathbf{M}_{10} \tilde{\mathbf{x}}_0. \quad (9.1)$$

When the last row of the \mathbf{P}_0 matrix is replaced with a plane equation $\hat{\mathbf{n}}_0 \cdot \mathbf{p} + c_0$ and points are assumed to lie on this plane, i.e., their disparity is $d_0 = 0$, we can ignore the last column of \mathbf{M}_{10} and also its last row, since we do not care about the final z-buffer depth. The resulting homography matrix $\tilde{\mathbf{H}}_{10}$ (the upper left 3×3 sub-matrix of \mathbf{M}_{10}) describes the mapping between pixels in the two images,

$$\tilde{\mathbf{x}}_1 \sim \tilde{\mathbf{H}}_{10} \tilde{\mathbf{x}}_0. \quad (9.2)$$

This observation formed the basis of some of the earliest automated image stitching algorithms (Mann and Picard 1994; Szeliski 1994, 1996). Because reliable feature matching techniques had not yet been developed, these algorithms used direct pixel value matching, i.e., direct parametric motion estimation, as described in Section 8.2 and Equations (6.19–6.20).

More recent stitching algorithms first extract features and then match them up, often using robust techniques such as RANSAC (Section 6.1.4) to compute a good set of inliers. The final computation of the homography (9.2), i.e., the solution of the least squares fitting problem given pairs of corresponding features,

$$x_1 = \frac{(1 + h_{00})x_0 + h_{01}y_0 + h_{02}}{h_{20}x_0 + h_{21}y_0 + 1} \quad \text{and} \quad y_1 = \frac{h_{10}x_0 + (1 + h_{11})y_0 + h_{12}}{h_{20}x_0 + h_{21}y_0 + 1}, \quad (9.3)$$

uses iterative least squares, as described in Section 6.1.3 and Equations (6.21–6.23).

9.1.2 Application: Whiteboard and document scanning

The simplest image-stitching application is to stitch together a number of image scans taken on a flatbed scanner. Say you have a large map, or a piece of child's artwork, that is too large to fit on your scanner. Simply take multiple scans of the document, making sure to overlap the scans by a large enough amount to ensure that there are enough common features. Next, take successive pairs of images that you know overlap, extract features, match them up, and estimate the 2D rigid transform (2.16),

$$\mathbf{x}_{k+1} = \mathbf{R}_k \mathbf{x}_k + \mathbf{t}_k, \quad (9.4)$$

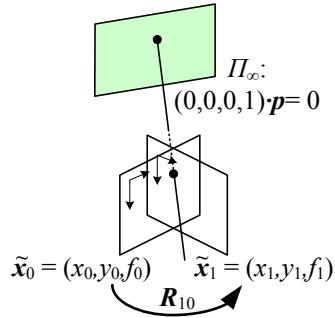


Figure 9.3 Pure 3D camera rotation. The form of the homography (mapping) is particularly simple and depends only on the 3D rotation matrix and focal lengths.

that best matches the features, using two-point RANSAC, if necessary, to find a good set of inliers. Then, on a final compositing surface (aligned with the first scan, for example), resample your images (Section 3.6.1) and average them together. Can you see any potential problems with this scheme?

One complication is that a 2D rigid transformation is non-linear in the rotation angle θ , so you will have to either use non-linear least squares or constrain R to be orthonormal, as described in Section 6.1.3.

A bigger problem lies in the pairwise alignment process. As you align more and more pairs, the solution may drift so that it is no longer globally consistent. In this case, a global optimization procedure, as described in Section 9.2, may be required. Such global optimization often requires a large system of non-linear equations to be solved, although in some cases, such as linearized homographies (Section 9.1.3) or similarity transforms (Section 6.1.2), regular least squares may be an option.

A slightly more complex scenario is when you take multiple overlapping handheld pictures of a whiteboard or other large planar object (He and Zhang 2005; Zhang and He 2007). Here, the natural motion model to use is a homography, although a more complex model that estimates the 3D rigid motion relative to the plane (plus the focal length, if unknown), could in principle be used.

9.1.3 Rotational panoramas

The most typical case for panoramic image stitching is when the camera undergoes a pure rotation. Think of standing at the rim of the Grand Canyon. Relative to the distant geometry in the scene, as you snap away, the camera is undergoing a pure rotation, which is equivalent to assuming that all points are very far from the camera, i.e., on the *plane at infinity* (Figure 9.3). Setting $t_0 = t_1 = 0$, we get the simplified 3×3 homography

$$\tilde{H}_{10} = K_1 R_1 R_0^{-1} K_0^{-1} = K_1 R_{10} K_0^{-1}, \quad (9.5)$$

where $K_k = \text{diag}(f_k, f_k, 1)$ is the simplified camera intrinsic matrix (2.59), assuming that $c_x = c_y = 0$, i.e., we are indexing the pixels starting from the optical center (Szeliski 1996).

This can also be re-written as

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \sim \begin{bmatrix} f_1 & & \\ & f_1 & \\ & & 1 \end{bmatrix} \mathbf{R}_{10} \begin{bmatrix} f_0^{-1} & & \\ & f_0^{-1} & \\ & & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \quad (9.6)$$

or

$$\begin{bmatrix} x_1 \\ y_1 \\ f_1 \end{bmatrix} \sim \mathbf{R}_{10} \begin{bmatrix} x_0 \\ y_0 \\ f_0 \end{bmatrix}, \quad (9.7)$$

which reveals the simplicity of the mapping equations and makes all of the motion parameters explicit. Thus, instead of the general eight-parameter homography relating a pair of images, we get the three-, four-, or five-parameter *3D rotation* motion models corresponding to the cases where the focal length f is known, fixed, or variable (Szeliski and Shum 1997).³ Estimating the 3D rotation matrix (and, optionally, focal length) associated with each image is intrinsically more stable than estimating a homography with a full eight degrees of freedom, which makes this the method of choice for large-scale image stitching algorithms (Szeliski and Shum 1997; Shum and Szeliski 2000; Brown and Lowe 2007).

Given this representation, how do we update the rotation matrices to best align two overlapping images? Given a current estimate for the homography $\tilde{\mathbf{H}}_{10}$ in (9.5), the best way to update \mathbf{R}_{10} is to prepend an *incremental* rotation matrix $\mathbf{R}(\omega)$ to the current estimate \mathbf{R}_{10} (Szeliski and Shum 1997; Shum and Szeliski 2000),

$$\tilde{\mathbf{H}}(\omega) = \mathbf{K}_1 \mathbf{R}(\omega) \mathbf{R}_{10} \mathbf{K}_0^{-1} = [\mathbf{K}_1 \mathbf{R}(\omega) \mathbf{K}_1^{-1}] [\mathbf{K}_1 \mathbf{R}_{10} \mathbf{K}_0^{-1}] = \mathbf{D} \tilde{\mathbf{H}}_{10}. \quad (9.8)$$

Note that here we have written the update rule in the *compositional* form, where the incremental update \mathbf{D} is *prepended* to the current homography $\tilde{\mathbf{H}}_{10}$. Using the small-angle approximation to $\mathbf{R}(\omega)$ given in (2.35), we can write the incremental update matrix as

$$\mathbf{D} = \mathbf{K}_1 \mathbf{R}(\omega) \mathbf{K}_1^{-1} \approx \mathbf{K}_1 (\mathbf{I} + [\omega]_\times) \mathbf{K}_1^{-1} = \begin{bmatrix} 1 & -\omega_z & f_1 \omega_y \\ \omega_z & 1 & -f_1 \omega_x \\ -\omega_y/f_1 & \omega_x/f_1 & 1 \end{bmatrix}. \quad (9.9)$$

Notice how there is now a nice one-to-one correspondence between the entries in the \mathbf{D} matrix and the h_{00}, \dots, h_{21} parameters used in Table 6.1 and Equation (6.19), i.e.,

$$(h_{00}, h_{01}, h_{02}, h_{00}, h_{11}, h_{12}, h_{20}, h_{21}) = (0, -\omega_z, f_1 \omega_y, \omega_z, 0, -f_1 \omega_x, -\omega_y/f_1, \omega_x/f_1). \quad (9.10)$$

We can therefore apply the chain rule to Equations (6.24 and 9.10) to obtain

$$\begin{bmatrix} \hat{x}' - x \\ \hat{y}' - y \end{bmatrix} = \begin{bmatrix} -xy/f_1 & f_1 + x^2/f_1 & -y \\ -(f_1 + y^2/f_1) & xy/f_1 & x \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}, \quad (9.11)$$

which give us the linearized update equations needed to estimate $\omega = (\omega_x, \omega_y, \omega_z)$.⁴ Notice that this update rule depends on the focal length f_1 of the *target* view and is independent

³ An initial estimate of the focal lengths can be obtained using the intrinsic calibration techniques described in Section 6.3.4 or from EXIF tags.

⁴ This is the same as the rotational component of instantaneous rigid flow (Bergen, Anandan, Hanna *et al.* 1992) and the update equations given by Szeliski and Shum (1997) and Shum and Szeliski (2000).

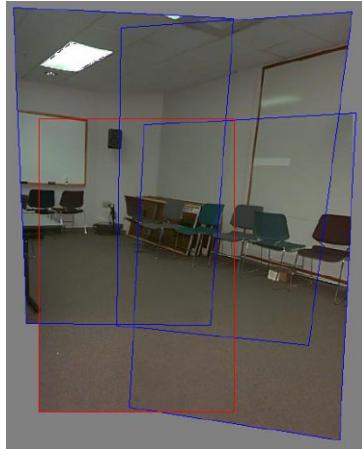


Figure 9.4 Four images taken with a hand-held camera registered using a 3D rotation motion model (Szeliski and Shum 1997) © 1997 ACM. Notice how the homographies, rather than being arbitrary, have a well-defined keystone shape whose width increases away from the origin.

of the focal length f_0 of the *template* view. This is because the compositional algorithm essentially makes small perturbations to the target. Once the incremental rotation vector ω has been computed, the R_1 rotation matrix can be updated using $R_1 \leftarrow R(\omega)R_1$.

The formulas for updating the focal length estimates are a little more involved and are given in (Shum and Szeliski 2000). We will not repeat them here, since an alternative update rule, based on minimizing the difference between back-projected 3D rays, is given in Section 9.2.1. Figure 9.4 shows the alignment of four images under the 3D rotation motion model.

9.1.4 Gap closing

The techniques presented in this section can be used to estimate a series of rotation matrices and focal lengths, which can be chained together to create large panoramas. Unfortunately, because of accumulated errors, this approach will rarely produce a closed 360° panorama. Instead, there will invariably be either a gap or an overlap (Figure 9.5).

We can solve this problem by matching the first image in the sequence with the last one. The difference between the two rotation matrix estimates associated with the repeated first indicates the amount of misregistration. This error can be distributed evenly across the whole sequence by taking the quotient of the two quaternions associated with these rotations and dividing this “error quaternion” by the number of images in the sequence (assuming relatively constant inter-frame rotations). We can also update the estimated focal length based on the amount of misregistration. To do this, we first convert the error quaternion into a *gap angle*, θ_g and then update the focal length using the equation $f' = f(1 - \theta_g/360^\circ)$.

Figure 9.5a shows the end of registered image sequence and the first image. There is a big gap between the last image and the first which are in fact the same image. The gap is 32° because the wrong estimate of focal length ($f = 510$) was used. Figure 9.5b shows the registration after closing the gap with the correct focal length ($f = 468$). Notice that both

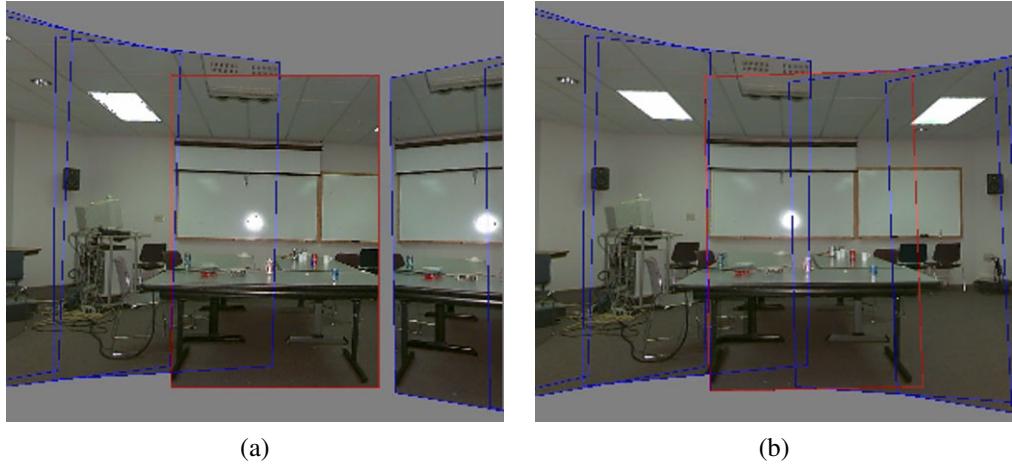


Figure 9.5 Gap closing (Szeliski and Shum 1997) © 1997 ACM: (a) A gap is visible when the focal length is wrong ($f = 510$). (b) No gap is visible for the correct focal length ($f = 468$).

mosaics show very little visual misregistration (except at the gap), yet Figure 9.5a has been computed using a focal length that has 9% error. Related approaches have been developed by Hartley (1994b), McMillan and Bishop (1995), Stein (1995), and Kang and Weiss (1997) to solve the focal length estimation problem using pure panning motion and cylindrical images.

Unfortunately, this particular gap-closing heuristic only works for the kind of “one-dimensional” panorama where the camera is continuously turning in the same direction. In Section 9.2, we describe a different approach to removing gaps and overlaps that works for arbitrary camera motions.

9.1.5 Application: Video summarization and compression

An interesting application of image stitching is the ability to summarize and compress videos taken with a panning camera. This application was first suggested by Teodosio and Bender (1993), who called their mosaic-based summaries *salient stills*. These ideas were then extended by Irani, Hsu, and Anandan (1995), Kumar, Anandan, Irani *et al.* (1995), and Irani and Anandan (1998) to additional applications, such as video compression and video indexing. While these early approaches used affine motion models and were therefore restricted to long focal lengths, the techniques were generalized by Lee, ge Chen, lung Bruce Lin *et al.* (1997) to full eight-parameter homographies and incorporated into the MPEG-4 video compression standard, where the stitched background layers were called *video sprites* (Figure 9.6).

While video stitching is in many ways a straightforward generalization of multiple-image stitching (Steedly, Pal, and Szeliski 2005; Baudisch, Tan, Steedly *et al.* 2006), the potential presence of large amounts of independent motion, camera zoom, and the desire to visualize dynamic events impose additional challenges. For example, moving foreground objects can often be removed using *median filtering*. Alternatively, foreground objects can be extracted into a separate layer (Sawhney and Ayer 1996) and later composited back into the stitched panoramas, sometimes as multiple instances to give the impressions of a “Chronophotograph” (Massey and Bender 1996) and sometimes as video overlays (Irani and Anandan 1998).

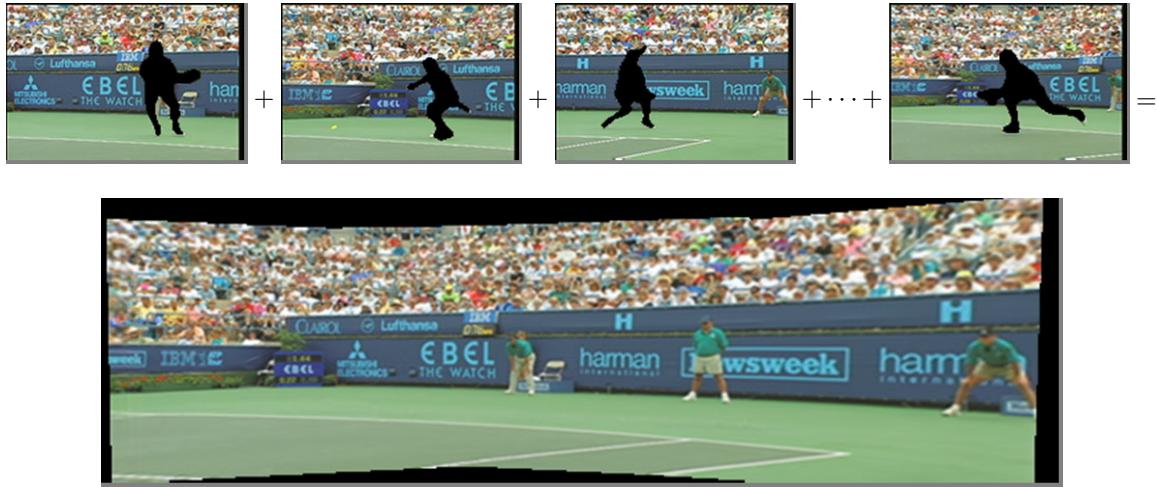


Figure 9.6 Video stitching the background scene to create a single *sprite* image that can be transmitted and used to re-create the background in each frame (Lee, ge Chen, lung Bruce Lin *et al.* 1997) © 1997 IEEE.

Videos can also be used to create animated *panoramic video textures* (Section 13.5.2), in which different portions of a panoramic scene are animated with independently moving video loops (Agarwala, Zheng, Pal *et al.* 2005; Rav-Acha, Pritch, Lischinski *et al.* 2005), or to shine “video flashlights” onto a composite mosaic of a scene (Sawhney, Arpa, Kumar *et al.* 2002).

Video can also provide an interesting source of content for creating panoramas taken from moving cameras. While this invalidates the usual assumption of a single point of view (optical center), interesting results can still be obtained. For example, the VideoBrush system of Sawhney, Kumar, Gendel *et al.* (1998) uses thin strips taken from the center of the image to create a panorama taken from a horizontally moving camera. This idea can be generalized to other camera motions and compositing surfaces using the concept of mosaics on adaptive manifold (Peleg, Rousso, Rav-Acha *et al.* 2000), and also used to generate panoramic stereograms (Peleg, Ben-Ezra, and Pritch 2001). Related ideas have been used to create panoramic matte paintings for multi-plane cel animation (Wood, Finkelstein, Hughes *et al.* 1997), for creating stitched images of scenes with parallax (Kumar, Anandan, Irani *et al.* 1995), and as 3D representations of more complex scenes using *multiple-center-of-projection images* (Rademacher and Bishop 1998) and *multi-perspective panoramas* (Román, Garg, and Levoy 2004; Román and Lensch 2006; Agarwala, Agrawala, Cohen *et al.* 2006).

Another interesting variant on video-based panoramas are *concentric mosaics* (Section 13.3.3) (Shum and He 1999). Here, rather than trying to produce a single panoramic image, the complete original video is kept and used to re-synthesize views (from different camera origins) using ray remapping (light field rendering), thus endowing the panorama with a sense of 3D depth. The same data set can also be used to explicitly reconstruct the depth using multi-baseline stereo (Peleg, Ben-Ezra, and Pritch 2001; Li, Shum, Tang *et al.* 2004; Zheng, Kang, Cohen *et al.* 2007).

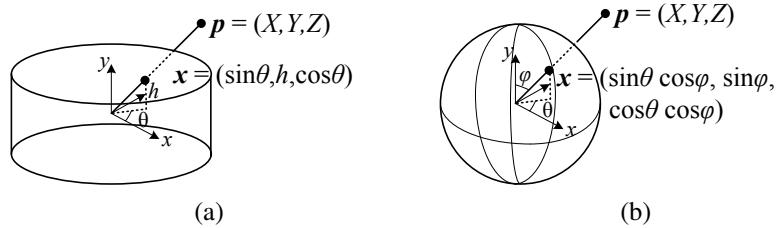


Figure 9.7 Projection from 3D to (a) cylindrical and (b) spherical coordinates.

9.1.6 Cylindrical and spherical coordinates

An alternative to using homographies or 3D motions to align images is to first warp the images into *cylindrical* coordinates and then use a pure translational model to align them (Chen 1995; Szeliski 1996). Unfortunately, this only works if the images are all taken with a level camera or with a known tilt angle.

Assume for now that the camera is in its canonical position, i.e., its rotation matrix is the identity, $\mathbf{R} = \mathbf{I}$, so that the optical axis is aligned with the z axis and the y axis is aligned vertically. The 3D ray corresponding to an (x, y) pixel is therefore (x, y, f) .

We wish to project this image onto a *cylindrical surface* of unit radius (Szeliski 1996). Points on this surface are parameterized by an angle θ and a height h , with the 3D cylindrical coordinates corresponding to (θ, h) given by

$$(\sin \theta, h, \cos \theta) \propto (x, y, f), \quad (9.12)$$

as shown in Figure 9.7a. From this correspondence, we can compute the formula for the warped or mapped coordinates (Szeliski and Shum 1997),

$$x' = s\theta = s \tan^{-1} \frac{x}{f}, \quad (9.13)$$

$$y' = sh = s \frac{y}{\sqrt{x^2 + f^2}}, \quad (9.14)$$

where s is an arbitrary scaling factor (sometimes called the *radius* of the cylinder) that can be set to $s = f$ to minimize the distortion (scaling) near the center of the image.⁵ The inverse of this mapping equation is given by

$$x = f \tan \theta = f \tan \frac{x'}{s}, \quad (9.15)$$

$$y = h\sqrt{x^2 + f^2} = \frac{y'}{s}f\sqrt{1 + \tan^2 x'/s} = f\frac{y'}{s}\sec\frac{x'}{s}. \quad (9.16)$$

Images can also be projected onto a *spherical surface* (Szeliski and Shum 1997), which is useful if the final panorama includes a full sphere or hemisphere of views, instead of just a cylindrical strip. In this case, the sphere is parameterized by two angles (θ, ϕ) , with 3D spherical coordinates given by

$$(\sin \theta \cos \phi, \sin \phi, \cos \theta \cos \phi) \propto (x, y, f), \quad (9.17)$$

⁵ The scale can also be set to a larger or smaller value for the final compositing surface, depending on the desired output panorama resolution—see Section 9.3.



Figure 9.8 A cylindrical panorama (Szeliski and Shum 1997) © 1997 ACM: (a) two cylindrically warped images related by a horizontal translation; (b) part of a cylindrical panorama compositing from a sequence of images.

as shown in Figure 9.7b.⁶ The correspondence between coordinates is now given by (Szeliski and Shum 1997):

$$x' = s\theta = s \tan^{-1} \frac{x}{f}, \quad (9.18)$$

$$y' = s\phi = s \tan^{-1} \frac{y}{\sqrt{x^2 + f^2}}, \quad (9.19)$$

while the inverse is given by

$$x = f \tan \theta = f \tan \frac{x'}{s}, \quad (9.20)$$

$$y = \sqrt{x^2 + f^2} \tan \phi = \tan \frac{y'}{s} f \sqrt{1 + \tan^2 x'/s} = f \tan \frac{y'}{s} \sec \frac{x'}{s}. \quad (9.21)$$

Note that it may be simpler to generate a scaled (x, y, z) direction from Equation (9.17) followed by a perspective division by z and a scaling by f .

Cylindrical image stitching algorithms are most commonly used when the camera is known to be level and only rotating around its vertical axis (Chen 1995). Under these conditions, images at different rotations are related by a pure horizontal translation.⁷ This makes it attractive as an initial class project in an introductory computer vision course, since the full complexity of the perspective alignment algorithm (Sections 6.1, 8.2, and 9.1.3) can be avoided. Figure 9.8 shows how two cylindrically warped images from a leveled rotational panorama are related by a pure translation (Szeliski and Shum 1997).

Professional panoramic photographers often use pan-tilt heads that make it easy to control the tilt and to stop at specific *detents* in the rotation angle. Motorized rotation heads are also sometimes used for the acquisition of larger panoramas (Kopf, Uyttendaele, Deussen *et al.* 2007).⁸ Not only do they ensure a uniform coverage of the visual field with a desired amount of image overlap but they also make it possible to stitch the images using cylindrical or spherical coordinates and pure translations. In this case, pixel coordinates (x, y, f) must first

⁶ Note that these are not the usual spherical coordinates, first presented in Equation (2.8). Here, the y axis points at the north pole instead of the z axis, since we are used to viewing images taken horizontally, i.e., with the y axis pointing in the direction of the gravity vector.

⁷ Small vertical tilts can sometimes be compensated for with vertical translations.

⁸ See also <http://gigapan.org>.



Figure 9.9 A spherical panorama constructed from 54 photographs (Szeliski and Shum 1997) © 1997 ACM.

be rotated using the known tilt and panning angles before being projected into cylindrical or spherical coordinates (Chen 1995). Having a roughly known panning angle also makes it easier to compute the alignment, since the rough relative positioning of all the input images is known ahead of time, enabling a reduced search range for alignment. Figure 9.9 shows a full 3D rotational panorama unwrapped onto the surface of a sphere (Szeliski and Shum 1997).

One final coordinate mapping worth mentioning is the *polar* mapping, where the north pole lies along the optical axis rather than the vertical axis,

$$(\cos \theta \sin \phi, \sin \theta \sin \phi, \cos \phi) = s(x, y, z). \quad (9.22)$$

In this case, the mapping equations become

$$x' = s\phi \cos \theta = s \frac{x}{r} \tan^{-1} \frac{r}{z}, \quad (9.23)$$

$$y' = s\phi \sin \theta = s \frac{y}{r} \tan^{-1} \frac{r}{z}, \quad (9.24)$$

where $r = \sqrt{x^2 + y^2}$ is the *radial distance* in the (x, y) plane and $s\phi$ plays a similar role in the (x', y') plane. This mapping provides an attractive visualization surface for certain kinds of wide-angle panoramas and is also a good model for the distortion induced by *fisheye lenses*, as discussed in Section 2.1.6. Note how for small values of (x, y) , the mapping equations reduce to $x' \approx sx/z$, which suggests that s plays a role similar to the focal length f .

9.2 Global alignment

So far, we have discussed how to register pairs of images using a variety of motion models. In most applications, we are given more than a single pair of images to register. The goal is then to find a globally consistent set of alignment parameters that minimize the mis-registration between all pairs of images (Szeliski and Shum 1997; Shum and Szeliski 2000; Sawhney and Kumar 1999; Coorg and Teller 2000).

In this section, we extend the pairwise matching criteria (6.2, 8.1, and 8.50) to a global energy function that involves all of the per-image pose parameters (Section 9.2.1). Once we have computed the global alignment, we often need to perform *local adjustments*, such as *parallax removal*, to reduce double images and blurring due to local mis-registrations (Section 9.2.2). Finally, if we are given an unordered set of images to register, we need to discover which images go together to form one or more panoramas. This process of *panorama recognition* is described in Section 9.2.3.

9.2.1 Bundle adjustment

One way to register a large number of images is to add new images to the panorama one at a time, aligning the most recent image with the previous ones already in the collection (Szeliski and Shum 1997) and discovering, if necessary, which images it overlaps (Sawhney and Kumar 1999). In the case of 360° panoramas, accumulated error may lead to the presence of a gap (or excessive overlap) between the two ends of the panorama, which can be fixed by stretching the alignment of all the images using a process called *gap closing* (Szeliski and Shum 1997). However, a better alternative is to simultaneously align all the images using a least-squares framework to correctly distribute any mis-registration errors.

The process of simultaneously adjusting pose parameters for a large collection of overlapping images is called *bundle adjustment* in the photogrammetry community (Triggs, McLauchlan, Hartley *et al.* 1999). In computer vision, it was first applied to the general structure from motion problem (Szeliski and Kang 1994) and then later specialized for panoramic image stitching (Shum and Szeliski 2000; Sawhney and Kumar 1999; Coorg and Teller 2000).

In this section, we formulate the problem of global alignment using a feature-based approach, since this results in a simpler system. An equivalent direct approach can be obtained either by dividing images into patches and creating a virtual feature correspondence for each one (as discussed in Section 9.2.4 and by Shum and Szeliski (2000)) or by replacing the per-feature error metrics with per-pixel metrics.

Consider the feature-based alignment problem given in Equation (6.2), i.e.,

$$E_{\text{pairwise-LS}} = \sum_i \|\mathbf{r}_i\|^2 = \|\tilde{\mathbf{x}}'_i(\mathbf{x}_i; \mathbf{p}) - \hat{\mathbf{x}}'_i\|^2. \quad (9.25)$$

For multi-image alignment, instead of having a single collection of pairwise feature correspondences, $\{(\mathbf{x}_i, \hat{\mathbf{x}}'_i)\}$, we have a collection of n features, with the location of the i th feature point in the j th image denoted by \mathbf{x}_{ij} and its scalar confidence (i.e., inverse variance) denoted by c_{ij} .⁹ Each image also has some associated pose parameters.

In this section, we assume that this pose consists of a rotation matrix \mathbf{R}_j and a focal length f_j , although formulations in terms of homographies are also possible (Szeliski and Shum 1997; Sawhney and Kumar 1999). The equation mapping a 3D point \mathbf{x}_i into a point \mathbf{x}_{ij} in frame j can be re-written from Equations (2.68) and (9.5) as

$$\tilde{\mathbf{x}}_{ij} \sim \mathbf{K}_j \mathbf{R}_j \mathbf{x}_i \quad \text{and} \quad \mathbf{x}_i \sim \mathbf{R}_j^{-1} \mathbf{K}_j^{-1} \tilde{\mathbf{x}}_{ij}, \quad (9.26)$$

⁹ Features that are not seen in image j have $c_{ij} = 0$. We can also use 2×2 inverse covariance matrices Σ_{ij}^{-1} in place of c_{ij} , as shown in Equation (6.11).

where $\mathbf{K}_j = \text{diag}(f_j, f_j, 1)$ is the simplified form of the calibration matrix. The motion mapping a point \mathbf{x}_{ij} from frame j into a point \mathbf{x}_{ik} in frame k is similarly given by

$$\tilde{\mathbf{x}}_{ik} \sim \tilde{\mathbf{H}}_{kj} \tilde{\mathbf{x}}_{ij} = \mathbf{K}_k \mathbf{R}_k \mathbf{R}_j^{-1} \mathbf{K}_j^{-1} \tilde{\mathbf{x}}_{ij}. \quad (9.27)$$

Given an initial set of $\{(\mathbf{R}_j, f_j)\}$ estimates obtained from chaining pairwise alignments, how do we refine these estimates?

One approach is to directly extend the pairwise energy $E_{\text{pairwise-LS}}$ (9.25) to a multiview formulation,

$$E_{\text{all-pairs-2D}} = \sum_i \sum_{jk} c_{ij} c_{ik} \|\tilde{\mathbf{x}}_{ik}(\hat{\mathbf{x}}_{ij}; \mathbf{R}_j, f_j, \mathbf{R}_k, f_k) - \hat{\mathbf{x}}_{ik}\|^2, \quad (9.28)$$

where the $\tilde{\mathbf{x}}_{ik}$ function is the *predicted* location of feature i in frame k given by (9.27), $\hat{\mathbf{x}}_{ij}$ is the *observed* location, and the “2D” in the subscript indicates that an image-plane error is being minimized (Shum and Szeliski 2000). Note that since $\tilde{\mathbf{x}}_{ik}$ depends on the $\hat{\mathbf{x}}_{ij}$ observed value, we actually have an *errors-in-variable* problem, which in principle requires more sophisticated techniques than least squares to solve (Van Huffel and Lemmerling 2002; Matei and Meer 2006). However, in practice, if we have enough features, we can directly minimize the above quantity using regular non-linear least squares and obtain an accurate multi-frame alignment.

While this approach works well in practice, it suffers from two potential disadvantages. First, since a summation is taken over all pairs with corresponding features, features that are observed many times are overweighted in the final solution. (In effect, a feature observed m times gets counted $\binom{m}{2}$ times instead of m times.) Second, the derivatives of $\tilde{\mathbf{x}}_{ik}$ with respect to the $\{(\mathbf{R}_j, f_j)\}$ are a little cumbersome, although using the incremental correction to \mathbf{R}_j introduced in Section 9.1.3 makes this more tractable.

An alternative way to formulate the optimization is to use true bundle adjustment, i.e., to solve not only for the pose parameters $\{(\mathbf{R}_j, f_j)\}$ but also for the 3D point positions $\{\mathbf{x}_i\}$,

$$E_{\text{BA-2D}} = \sum_i \sum_j c_{ij} \|\tilde{\mathbf{x}}_{ij}(\mathbf{x}_i; \mathbf{R}_j, f_j) - \hat{\mathbf{x}}_{ij}\|^2, \quad (9.29)$$

where $\tilde{\mathbf{x}}_{ij}(\mathbf{x}_i; \mathbf{R}_j, f_j)$ is given by (9.26). The disadvantage of full bundle adjustment is that there are more variables to solve for, so each iteration and also the overall convergence may be slower. (Imagine how the 3D points need to “shift” each time some rotation matrices are updated.) However, the computational complexity of each linearized Gauss–Newton step can be reduced using sparse matrix techniques (Section 7.4.1) (Szeliski and Kang 1994; Triggs, McLauchlan, Hartley *et al.* 1999; Hartley and Zisserman 2004).

An alternative formulation is to minimize the error in 3D projected ray directions (Shum and Szeliski 2000), i.e.,

$$E_{\text{BA-3D}} = \sum_i \sum_j c_{ij} \|\tilde{\mathbf{x}}_i(\hat{\mathbf{x}}_{ij}; \mathbf{R}_j, f_j) - \mathbf{x}_i\|^2, \quad (9.30)$$

where $\tilde{\mathbf{x}}_i(\mathbf{x}_i; \mathbf{R}_j, f_j)$ is given by the second half of (9.26). This has no particular advantage over (9.29). In fact, since errors are being minimized in 3D ray space, there is a bias towards estimating longer focal lengths, since the angles between rays become smaller as f increases.

However, if we eliminate the 3D rays \mathbf{x}_i , we can derive a pairwise energy formulated in 3D ray space (Shum and Szeliski 2000),

$$E_{\text{all-pairs-3D}} = \sum_i \sum_{jk} c_{ij} c_{ik} \|\tilde{\mathbf{x}}_i(\hat{\mathbf{x}}_{ij}; \mathbf{R}_j, f_j) - \tilde{\mathbf{x}}_i(\hat{\mathbf{x}}_{ik}; \mathbf{R}_k, f_k)\|^2. \quad (9.31)$$

This results in the simplest set of update equations (Shum and Szeliski 2000), since the f_k can be folded into the creation of the homogeneous coordinate vector as in Equation (9.7). Thus, even though this formula over-weights features that occur more frequently, it is the method used by Shum and Szeliski (2000) and Brown, Szeliski, and Winder (2005). In order to reduce the bias towards longer focal lengths, we multiply each residual (3D error) by $\sqrt{f_j f_k}$, which is similar to projecting the 3D rays into a “virtual camera” of intermediate focal length.

Up vector selection. As mentioned above, there exists a global ambiguity in the pose of the 3D cameras computed by the above methods. While this may not appear to matter, people prefer that the final stitched image is “upright” rather than twisted or tilted. More concretely, people are used to seeing photographs displayed so that the vertical (gravity) axis points straight up in the image. Consider how you usually shoot photographs: while you may pan and tilt the camera any which way, you usually keep the horizontal edge of your camera (its x -axis) parallel to the ground plane (perpendicular to the world gravity direction).

Mathematically, this constraint on the rotation matrices can be expressed as follows. Recall from Equation (9.26) that the 3D to 2D projection is given by

$$\tilde{\mathbf{x}}_{ik} \sim \mathbf{K}_k \mathbf{R}_k \mathbf{x}_i. \quad (9.32)$$

We wish to post-multiply each rotation matrix \mathbf{R}_k by a global rotation \mathbf{R}_g such that the projection of the global y -axis, $\hat{\mathbf{j}} = (0, 1, 0)$ is perpendicular to the image x -axis, $\hat{\mathbf{i}} = (1, 0, 0)$.¹⁰

This constraint can be written as

$$\hat{\mathbf{i}}^T \mathbf{R}_k \mathbf{R}_g \hat{\mathbf{j}} = 0 \quad (9.33)$$

(note that the scaling by the calibration matrix is irrelevant here). This is equivalent to requiring that the first row of \mathbf{R}_k , $\mathbf{r}_{k0} = \hat{\mathbf{i}}^T \mathbf{R}_k$ be perpendicular to the second column of \mathbf{R}_g , $\mathbf{r}_{g1} = \mathbf{R}_g \hat{\mathbf{j}}$. This set of constraints (one per input image) can be written as a least squares problem,

$$\mathbf{r}_{g1} = \arg \min_{\mathbf{r}} \sum_k (\mathbf{r}^T \mathbf{r}_{k0})^2 = \arg \min_{\mathbf{r}} \mathbf{r}^T \left[\sum_k \mathbf{r}_{k0} \mathbf{r}_{k0}^T \right] \mathbf{r}. \quad (9.34)$$

Thus, \mathbf{r}_{g1} is the smallest eigenvector of the *scatter* or *moment* matrix spanned by the individual camera rotation x -vectors, which should generally be of the form $(c, 0, s)$ when the cameras are upright.

To fully specify the \mathbf{R}_g global rotation, we need to specify one additional constraint. This is related to the *view selection* problem discussed in Section 9.3.1. One simple heuristic is to prefer the average z -axis of the individual rotation matrices, $\bar{\mathbf{k}} = \sum_k \hat{\mathbf{k}}^T \mathbf{R}_k$ to be close to the world z -axis, $\mathbf{r}_{g2} = \mathbf{R}_g \hat{\mathbf{k}}$. We can therefore compute the full rotation matrix \mathbf{R}_g in three steps:

¹⁰ Note that here we use the convention common in computer graphics that the vertical world axis corresponds to y . This is a natural choice if we wish the rotation matrix associated with a “regular” image taken horizontally to be the identity, rather than a 90° rotation around the x -axis.

1. $\mathbf{r}_{g1} = \min \text{ eigenvector } (\sum_k \mathbf{r}_{k0}\mathbf{r}_{k0}^T);$
2. $\mathbf{r}_{g0} = \mathcal{N}((\sum_k \mathbf{r}_{k2}) \times \mathbf{r}_{g1});$
3. $\mathbf{r}_{g2} = \mathbf{r}_{g0} \times \mathbf{r}_{g1},$

where $\mathcal{N}(\mathbf{v}) = \mathbf{v}/\|\mathbf{v}\|$ normalizes a vector \mathbf{v} .

9.2.2 Parallax removal

Once we have optimized the global orientations and focal lengths of our cameras, we may find that the images are still not perfectly aligned, i.e., the resulting stitched image looks blurry or ghosted in some places. This can be caused by a variety of factors, including unmodeled radial distortion, 3D parallax (failure to rotate the camera around its optical center), small scene motions such as waving tree branches, and large-scale scene motions such as people moving in and out of pictures.

Each of these problems can be treated with a different approach. Radial distortion can be estimated (potentially ahead of time) using one of the techniques discussed in Section 2.1.6. For example, the *plumb-line method* (Brown 1971; Kang 2001; El-Melegy and Farag 2003) adjusts radial distortion parameters until slightly curved lines become straight, while mosaic-based approaches adjust them until mis-registration is reduced in image overlap areas (Stein 1997; Sawhney and Kumar 1999).

3D parallax can be handled by doing a full 3D bundle adjustment, i.e., by replacing the projection equation (9.26) used in Equation (9.29) with Equation (2.68), which models camera translations. The 3D positions of the matched feature points and cameras can then be simultaneously recovered, although this can be significantly more expensive than parallax-free image registration. Once the 3D structure has been recovered, the scene could (in theory) be projected to a single (central) viewpoint that contains no parallax. However, in order to do this, dense *stereo* correspondence needs to be performed (Section 11.3) (Li, Shum, Tang *et al.* 2004; Zheng, Kang, Cohen *et al.* 2007), which may not be possible if the images contain only partial overlap. In that case, it may be necessary to correct for parallax only in the overlap areas, which can be accomplished using a *multi-perspective plane sweep* (MPPS) algorithm (Kang, Szeliski, and Uyttendaele 2004; Uyttendaele, Criminisi, Kang *et al.* 2004).

When the motion in the scene is very large, i.e., when objects appear and disappear completely, a sensible solution is to simply *select* pixels from only one image at a time as the source for the final composite (Milgram 1977; Davis 1998; Agarwala, Dontcheva, Agrawala *et al.* 2004), as discussed in Section 9.3.2. However, when the motion is reasonably small (on the order of a few pixels), general 2D motion estimation (optical flow) can be used to perform an appropriate correction before blending using a process called *local alignment* (Shum and Szeliski 2000; Kang, Uyttendaele, Winder *et al.* 2003). This same process can also be used to compensate for radial distortion and 3D parallax, although it uses a weaker motion model than explicitly modeling the source of error and may, therefore, fail more often or introduce unwanted distortions.

The local alignment technique introduced by Shum and Szeliski (2000) starts with the global bundle adjustment (9.31) used to optimize the camera poses. Once these have been estimated, the *desired* location of a 3D point \mathbf{x}_i can be estimated as the *average* of the back-

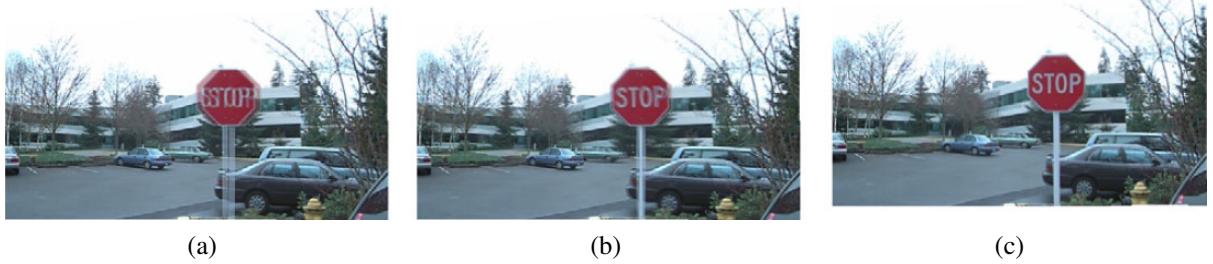


Figure 9.10 Deghosting a mosaic with motion parallax (Shum and Szeliski 2000) © 2000 IEEE: (a) composite with parallax; (b) after a single deghosting step (patch size 32); (c) after multiple steps (sizes 32, 16 and 8).

projected 3D locations,

$$\bar{\mathbf{x}}_i \sim \sum_j c_{ij} \tilde{\mathbf{x}}_i(\hat{\mathbf{x}}_{ij}; \mathbf{R}_j, f_j) / \sum_j c_{ij}, \quad (9.35)$$

which can be projected into each image j to obtain a *target location* $\bar{\mathbf{x}}_{ij}$. The difference between the target locations $\bar{\mathbf{x}}_{ij}$ and the original features \mathbf{x}_{ij} provide a set of local motion estimates

$$\mathbf{u}_{ij} = \bar{\mathbf{x}}_{ij} - \mathbf{x}_{ij}, \quad (9.36)$$

which can be interpolated to form a dense correction field $\mathbf{u}_j(\mathbf{x}_j)$. In their system, Shum and Szeliski (2000) use an *inverse warping* algorithm where the sparse $-\mathbf{u}_{ij}$ values are placed at the new target locations $\bar{\mathbf{x}}_{ij}$, interpolated using bilinear kernel functions (Nielson 1993) and then added to the original pixel coordinates when computing the warped (corrected) image. In order to get a reasonably dense set of features to interpolate, Shum and Szeliski (2000) place a feature point at the center of each patch (the patch size controls the smoothness in the local alignment stage), rather than relying of features extracted using an interest operator (Figure 9.10).

An alternative approach to motion-based de-ghosting was proposed by Kang, Uyttendaele, Winder *et al.* (2003), who estimate dense optical flow between each input image and a central *reference* image. The accuracy of the flow vector is checked using a photo-consistency measure before a given warped pixel is considered valid and is used to compute a high dynamic range radiance estimate, which is the goal of their overall algorithm. The requirement for a reference image makes their approach less applicable to general image mosaicing, although an extension to this case could certainly be envisaged.

9.2.3 Recognizing panoramas

The final piece needed to perform fully automated image stitching is a technique to recognize which images actually go together, which Brown and Lowe (2007) call *recognizing panoramas*. If the user takes images in sequence so that each image overlaps its predecessor and also specifies the first and last images to be stitched, bundle adjustment combined with the process of *topology inference* can be used to automatically assemble a panorama (Sawhney and Kumar 1999). However, users often jump around when taking panoramas, e.g., they may start a new row on top of a previous one, jump back to take a repeat shot, or create

360° panoramas where end-to-end overlaps need to be discovered. Furthermore, the ability to discover multiple panoramas taken by a user over an extended period of time can be a big convenience.

To recognize panoramas, Brown and Lowe (2007) first find all pairwise image overlaps using a feature-based method and then find connected components in the overlap graph to “recognize” individual panoramas (Figure 9.11). The feature-based matching stage first extracts scale invariant feature transform (SIFT) feature locations and feature descriptors (Lowe 2004) from all the input images and places them in an indexing structure, as described in Section 4.1.3. For each image pair under consideration, the nearest matching neighbor is found for each feature in the first image, using the indexing structure to rapidly find candidates and then comparing feature descriptors to find the best match. RANSAC is used to find a set of *inlier* matches; pairs of matches are used to hypothesize similarity motion models that are then used to count the number of inliers. (A more recent RANSAC algorithm tailored specifically for rotational panoramas is described by Brown, Hartley, and Nistér (2007).)

In practice, the most difficult part of getting a fully automated stitching algorithm to work is deciding which pairs of images actually correspond to the same parts of the scene. Repeated structures such as windows (Figure 9.12) can lead to false matches when using a feature-based approach. One way to mitigate this problem is to perform a direct pixel-based comparison between the registered images to determine if they actually are different views of the same scene. Unfortunately, this heuristic may fail if there are moving objects in the scene (Figure 9.13). While there is no magic bullet for this problem, short of full scene understanding, further improvements can likely be made by applying domain-specific heuristics, such as priors on typical camera motions as well as machine learning techniques applied to the problem of match validation.

9.2.4 Direct vs. feature-based alignment

Given that there exist these two approaches to aligning images, which is preferable?

Early feature-based methods would get confused in regions that were either too textured or not textured enough. The features would often be distributed unevenly over the images, thereby failing to match image pairs that should have been aligned. Furthermore, establishing correspondences relied on simple cross-correlation between patches surrounding the feature points, which did not work well when the images were rotated or had foreshortening due to homographies.

Today, feature detection and matching schemes are remarkably robust and can even be used for known object recognition from widely separated views (Lowe 2004). Features not only respond to regions of high “cornerness” (Förstner 1986; Harris and Stephens 1988) but also to “blob-like” regions (Lowe 2004), and uniform areas (Matas, Chum, Urban *et al.* 2004; Tuytelaars and Van Gool 2004). Furthermore, because they operate in scale-space and use a dominant orientation (or orientation invariant descriptors), they can match images that differ in scale, orientation, and even foreshortening. Our own experience in working with feature-based approaches is that if the features are well distributed over the image and the descriptors reasonably designed for repeatability, enough correspondences to permit image stitching can usually be found (Brown, Szeliski, and Winder 2005).

The biggest disadvantage of direct pixel-based alignment techniques is that they have a limited range of convergence. Even though they can be used in a hierarchical (coarse-to-

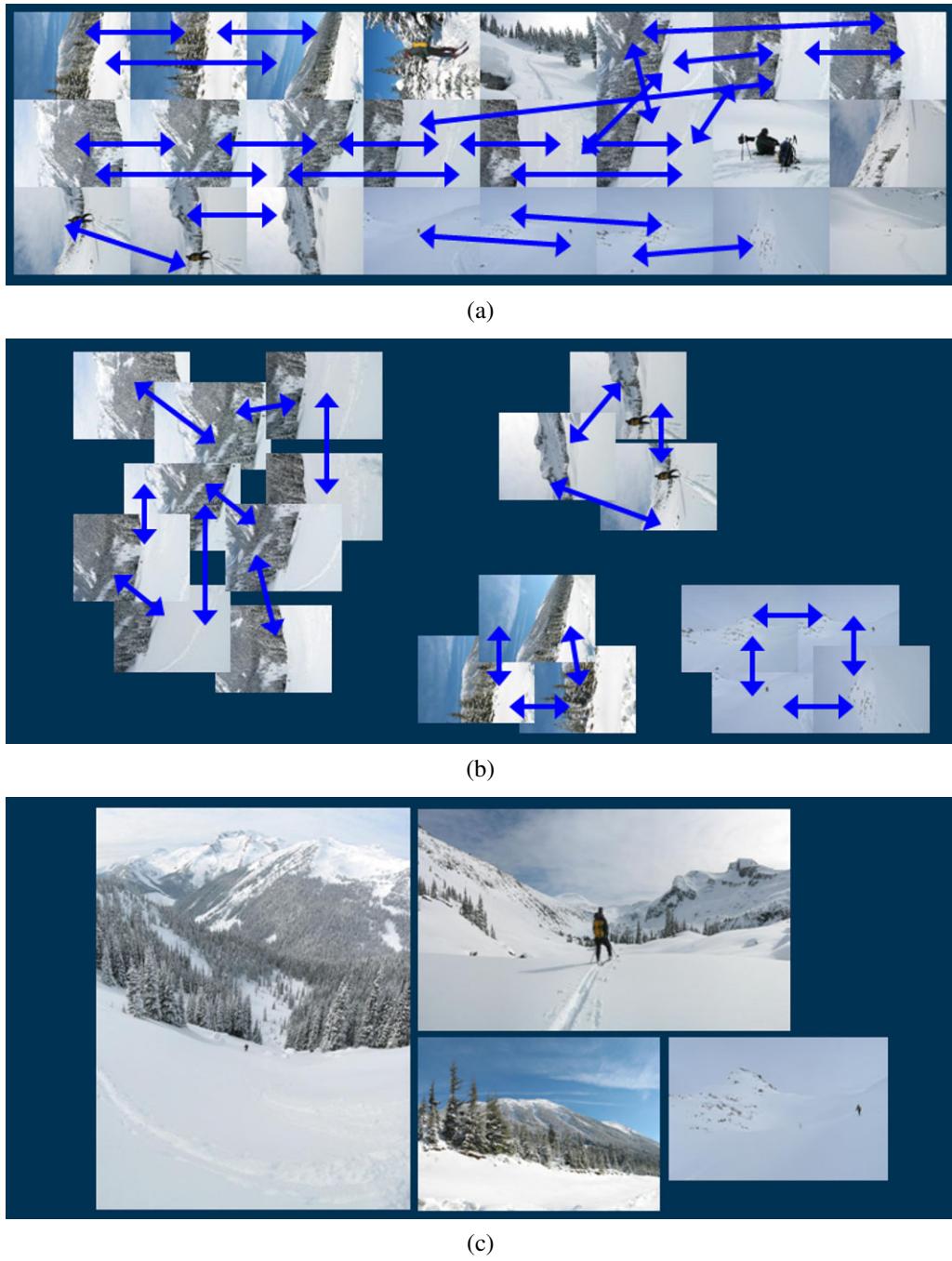


Figure 9.11 Recognizing panoramas (Brown, Szeliski, and Winder 2005), figures courtesy of Matthew Brown: (a) input images with pairwise matches; (b) images grouped into connected components (panoramas); (c) individual panoramas registered and blended into stitched composites.

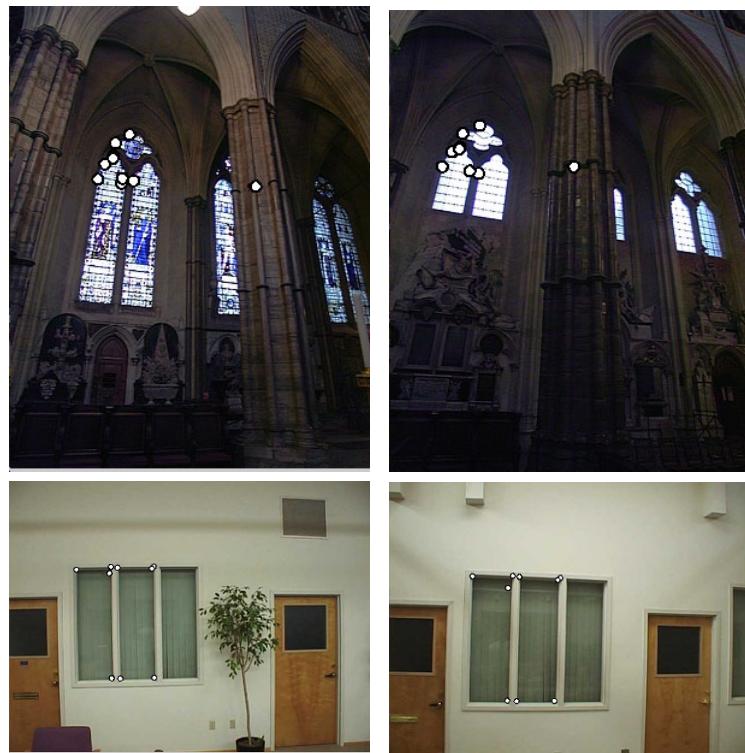


Figure 9.12 Matching errors (Brown, Szeliski, and Winder 2004): accidental matching of several features can lead to matches between pairs of images that do not actually overlap.

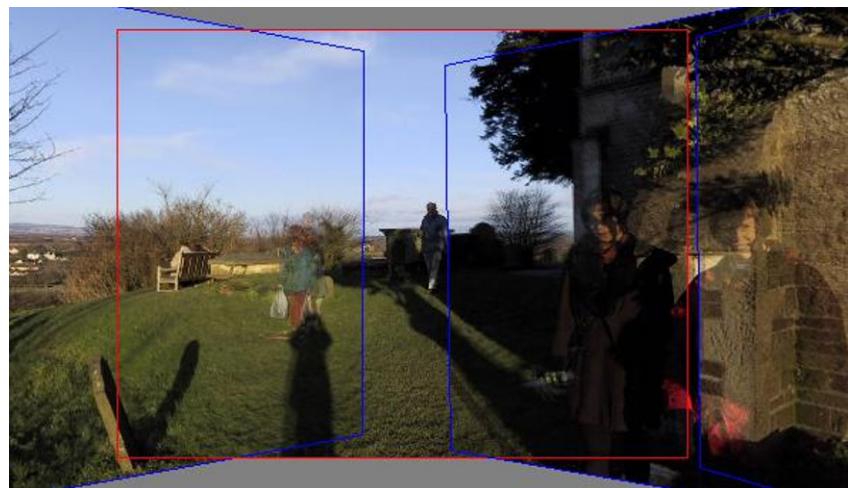


Figure 9.13 Validation of image matches by direct pixel error comparison can fail when the scene contains moving objects (Uyttendaele, Eden, and Szeliski 2001) © 2001 IEEE.

fine) estimation framework, in practice it is hard to use more than two or three levels of a pyramid before important details start to be blurred away.¹¹ For matching sequential frames in a video, direct approaches can usually be made to work. However, for matching partially overlapping images in photo-based panoramas or for image collections where the contrast or content varies too much, they fail too often to be useful and feature-based approaches are therefore preferred.

9.3 Compositing

Once we have registered all of the input images with respect to each other, we need to decide how to produce the final stitched mosaic image. This involves selecting a final compositing surface (flat, cylindrical, spherical, etc.) and view (reference image). It also involves selecting which pixels contribute to the final composite and how to optimally blend these pixels to minimize visible seams, blur, and ghosting.

In this section, we review techniques that address these problems, namely compositing surface parameterization, pixel and seam selection, blending, and exposure compensation. My emphasis is on fully automated approaches to the problem. Since the creation of high-quality panoramas and composites is as much an artistic endeavor as a computational one, various interactive tools have been developed to assist this process (Agarwala, Dontcheva, Agrawala *et al.* 2004; Li, Sun, Tang *et al.* 2004; Rother, Kolmogorov, and Blake 2004). Some of these are covered in more detail in Section 10.4.

9.3.1 Choosing a compositing surface

The first choice to be made is how to represent the final image. If only a few images are stitched together, a natural approach is to select one of the images as the *reference* and to then warp all of the other images into its reference coordinate system. The resulting composite is sometimes called a *flat* panorama, since the projection onto the final surface is still a perspective projection, and hence straight lines remain straight (which is often a desirable attribute).¹²

For larger fields of view, however, we cannot maintain a flat representation without excessively stretching pixels near the border of the image. (In practice, flat panoramas start to look severely distorted once the field of view exceeds 90° or so.) The usual choice for compositing larger panoramas is to use a cylindrical (Chen 1995; Szeliski 1996) or spherical (Szeliski and Shum 1997) projection, as described in Section 9.1.6. In fact, any surface used for *environment mapping* in computer graphics can be used, including a *cube map*, which represents the full viewing sphere with the six square faces of a cube (Greene 1986; Szeliski and Shum 1997). Cartographers have also developed a number of alternative methods for representing the globe (Bugayevskiy and Snyder 1995).

The choice of parameterization is somewhat application dependent and involves a trade-off between keeping the local appearance undistorted (e.g., keeping straight lines straight)

¹¹ Fourier-based correlation (Szeliski 1996; Szeliski and Shum 1997) can extend this range but requires cylindrical images or motion prediction to be useful.

¹² Recently, some techniques have been developed to straighten curved lines in cylindrical and spherical panoramas (Carroll, Agrawala, and Agarwala 2009; Kopf, Lischinski, Deussen *et al.* 2009).