Computer Vision: Algorithms and Applications

2nd Edition

Richard Szeliski

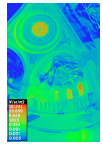
September 30, 2021

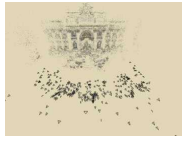
This book is dedicated to my parents,

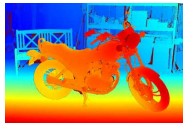
Zdzisław and Jadwiga,

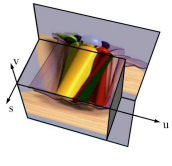
and my family,

Lyn, Anne, and Stephen.









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.

That initial course evolved into a more complete computer vision syllabus and project-oriented course structure that I used to co-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 were then 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 40 years' experience doing computer vision research in corporate research labs, mostly at Digital Equipment Corporation's Cambridge Research Lab, Microsoft Research, and Facebook. 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, e.g., by delving into more specialized topics, 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 four 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 assumptions 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).

- **Data-driven:** collect a representative set of test data (ideally, with labels or ground-truth answers) and use these data to either tune or learn your model parameters, or at least to validate and quantify its performance.

These four 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, Appendix C includes pointers to commonly used datasets and software libraries that contain implementations of a wide variety of computer vision algorithms, which can enable you to tackle more ambitious projects (with your instructor's consent).

Notes on the Second Edition

The last decade has seen a truly dramatic explosion in the performance and applicability of computer vision algorithms, much of it engendered by the application of machine learning algorithms to large amounts of visual training data (Su and Crandall 2021).

Deep neural networks now play an essential role in so many vision algorithms that the new edition of this book introduces them early on as a fundamental technique that gets used extensively in subsequent chapters.

The most notable changes in the second edition include:

- Machine learning, deep learning, and deep neural networks are introduced early on in Chapter 5, as they play just as fundamental a role in vision algorithms as more classical techniques, such as image processing, graphical/probabilistic models, and energy minimization, which are introduced in the preceding two chapters.

- The recognition chapter has been moved earlier in the book to Chapter 6, since end-to-end deep learning systems no longer require the development of building blocks such as feature detection, matching, and segmentation. Many of the students taking vision classes are primarily interested in visual recognition, so presenting this material earlier in the course makes it easier for students to base their final project on these topics. This chapter also includes sections on semantic segmentation, video understanding, and vision and language.

- The application of neural networks and deep learning to myriad computer vision algorithms and applications, including flow and stereo, 3D shape modeling, and newly

emerging fields such as neural rendering.

- New technologies such as SLAM (simultaneous localization and mapping) and VIO (visual inertial odometry) that now run reliably and are used in real-time applications such as augmented reality and autonomous navigation.

In addition to these larger changes, the book has been updated to reflect the latest state-of-the-art techniques such as internet-scale image search and phone-based computational photography. The new edition includes over 1500 new citations (papers) and has over 200 new figures.

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 PhD thesis advisors at Carnegie Mellon University, taught me the fundamentals of good research, writing, and presentation and mentored several generations of outstanding students and researchers. They fired up my interest in visual processing, 3D modeling, and statistical methods, while Larry Matthies introduced me to Kalman filtering and stereo matching. Geoff continues to inspire so many of us with this undiminished passion for trying to figure out "what makes the brain work". It's been a delight to see his pursuit of connectionist ideas bear so much fruit in this past decade.

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 Carlbom, Gudrun Klinker, Keith Waters, William Hsu, 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.

At Microsoft Research, I had the outstanding fortune to work with some of the world's best researchers in computer vision and computer graphics, including Michael Cohen, Matt

Uyttendaele, Sing Bing Kang, Harry Shum, Larry Zitnick, Sudipta Sinha, Drew Steedly, Simon Baker, Johannes Kopf, Neel Joshi, Krishnan Ramnath, Anandan, Phil Torr, Antonio Criminisi, Simon Winder, Matthew Brown, Michael Goesele, Richard Hartley, Hugues Hoppe, Stephen Gortler, Steve Shafer, Matthew Turk, Georg Petschnigg, Kentaro Toyama, Ramin Zabih, Shai Avidan, Patrice Simard, Chris Pal, Nebojsa Jojic, Patrick Baudisch, Dani Lischinski, Raanan Fattal, Eric Stollnitz, David Nistér, Blaise Aguera y Arcas, Andrew Fitzgibbon, Jamie Shotton, Wolf Kienzle, Piotr Dollar, and Ross Girshick. I was also lucky to have as interns such great students as Polina Golland, Simon Baker, Mei Han, Arno Schödl, Ron Dror, Ashley Eden, Jonathan Shade, 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, Dilip Krishnan, Adarsh Kowdle, Edward Hsiao, Yong Seok Heo, Fabian Langguth, Andrew Owens, and Tianfan Xue. Working with such outstanding students also gave me the opportunity to collaborate with some of their amazing advisors, including Bill Freeman, Irfan Essa, Marc Pollefeys, Michael Black, Marc Levoy, and Andrew Zisserman.

Since moving to Facebook, I've had the pleasure to continue my collaborations with Michael Cohen, Matt Uyttendaele, Johannes Kopf, Wolf Kienzle, and Krishnan Ramnath, and also new colleagues including Kevin Matzen, Bryce Evans, Suhib Alsisan, Changil Kim, David Geraghty, Jan Herling, Nils Plath, Jan-Michael Frahm, True Price, Richard Newcombe, Thomas Whelan, Michael Goesele, Steven Lovegrove, Julian Straub, Simon Green, Brian Cabral, Michael Toksvig, Albert Para Pozzo, Laura Sevilla-Lara, Georgia Gkioxari, Justin Johnson, Chris Sweeney, and Vassileios Balntas. I've also had the pleasure to collaborate with some outstanding summer interns, including Tianfan Xue, Scott Wehrwein, Peter Hedman, Joel Janai, Aleksander Hołyński, Xuan Luo, Rui Wang, Olivia Wiles, and Yulun Tian. I'd like to thank in particular Michael Cohen, my mentor, colleague, and friend for the last 25 years for his unwavering support of my sprint to complete this second edition.

While working at Microsoft and Facebook, 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 Fréderic Pighin, Yung-Yu Chuang, Doug Zongker, Colin Zheng, Aseem Agarwala, Dan Goldman, Noah Snavely, Ian Simon, Rahul Garg, Ryan Kaminsky, Juliet Fiss, Aleksander Hołyński, and Yifan Wang. 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 informal book editor, Vladimir Kolmogorov, 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, Rico Malvar, 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, Srikrishna Bhat, and Toru Tamaki, who wrote the Japanese translation and provided many useful errata.

For the second edition, I received significant help and advice from three key contributors. Daniel Scharstein helped me update the chapter on stereo, Matt Deitke contributed descriptions of the newest papers in deep learning, including the sections on transformers, variational autoencoders, and text-to-image synthesis, along with the exercises in Chapters 5 and 6 and some illustrations. Sing Bing Kang reviewed multiple drafts and provided useful suggestions. I'd also like to thank Andrew Glassner, whose book (Glassner 2018) and figures were a tremendous help, Justin Johnson, Sean Bell, Ishan Misra, David Fouhey, Michael Brown, Abdelrahman Abdelhamed, Frank Dellaert, Xinlei Chen, Ross Girshick, Andreas Geiger, Dmytro Mishkin, Aleksander Hołyński, Joel Janai, Christoph Feichtenhofer, Yuandong Tian, Alyosha Efros, Pascal Fua, Torsten Sattler, Laura Leal-Taixé, Aljosa Osep, Qunjie Zhou, Jiří Matas, Eddy Ilg, Yann LeCun, Larry Jackel, Vasileios Balntas, Daniel DeTone, Zachary Teed, Junhwa Hur, Jun-Yan Zhu, Filip Radenović, Michael Zollhöfer, Matthias Nießner, Andrew Owens, Hervé Jégou, Luowei Zhou, Ricardo Martin Brualla, Pratul Srinivasan, Matteo Poggi, Fabio Tosi, Ahmed Osman, Dave Howell, Holger Heidrich, Howard Yen, Anton Papst, Syamprasad K. Rajagopalan, Abhishek Nagar, Vladimir Kuznetsov, Raphaël Fouque, Marian Ciobanu, Darko Simonovic, and Guilherme Schlinker.

In preparing the second edition, I taught some of the new material in two courses that I helped co-teach in 2020 at Facebook and UW. I'd like to thank my co-instructors Jan-Michael Frahm, Michael Goesele, Georgia Gkioxari, Ross Girshick, Jakob Julian Engel, Daniel Scharstein, Fernando de la Torre, Steve Seitz, and Harpreet Sawhney, from whom I learned a lot about the latest techniques that are included in the new edition. I'd also like to thank the TAs, including David Geraghty, True Price, Kevin Matzen, Akash Bapat, Alek-

sander Hołyński, Keunhong Park, and Svetoslav Kolev, for the wonderful job they did in creating and grading the assignments. I'd like to give a special thanks to Justin Johnson, whose excellent class slides (Johnson 2020), based on earlier slides from Stanford (Li, Johnson, and Yeung 2019), taught me the fundamentals of deep learning and which I used extensively in my own class and in preparing the new chapter on deep learning.

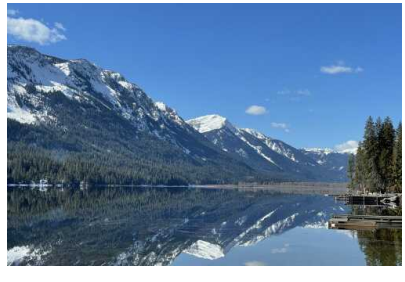
Shena Deuchers and Ian Kingston did a fantastic job copy-editing the first and second editions, respectively and suggesting many useful improvements, and Wayne Wheeler and Simon Rees at Springer were most helpful throughout the whole book publishing process. Keith Price's Annotated Computer Vision Bibliography was invaluable in tracking down references and 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.

The last year of writing this second edition took place during the worldwide COVID-19 pandemic. I would like to thank all of the first responders, medical and front-line workers, and everyone else who helped get us through these difficult and challenging times and to acknowledge the impact that this and other recent tragedies have had on all of us.

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 always inspired me; to my sister Basia for her lifelong friendship; and especially to Lyn, Anne, and Stephen, whose love and support in all matters (including my book projects) makes it all worthwhile.

Lake Wenatchee
May 2021



Contents

Chapter 1

Introduction

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.*

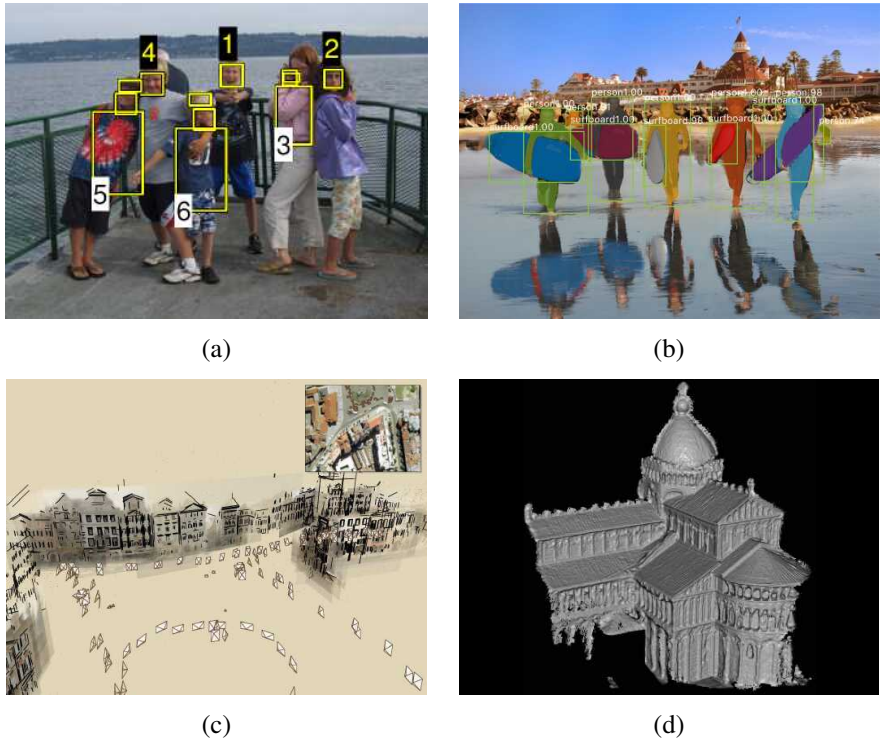


Figure 1.2 *Some examples of computer vision algorithms and applications. (a)* Face detection *algorithms, coupled with color-based clothing and hair detection algorithms, can locate and recognize the individuals in this image (Sivic, Zitnick, and Szeliski 2006) © 2006 Springer. (b)* Object instance segmentation *can delineate each person and object in a complex scene (He, Gkioxari* et al. *2017) © 2017 IEEE. (c)* 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. (d)* 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* et al. *2007) © 2007 IEEE.*

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 expressions (Figure 1.2a). Perceptual psychologists have spent decades trying to understand how the visual system works and, even though they can devise optical illusions[1] to tease apart some of its principles (Figure 1.3), a complete solution to this puzzle remains elusive (Marr 1982; Wandell 1995; Palmer 1999; Livingstone 2008; Frisby and Stone 2010).

Researchers in computer vision have been developing, in parallel, mathematical techniques for recovering the three-dimensional shape and appearance of objects in imagery. Here, the progress in the last two decades has been rapid. We now have reliable techniques for accurately computing a 3D model of an environment from thousands of partially overlapping photographs (Figure 1.2c). 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 1.2d). We can even, with moderate success, delineate most of the people and objects in a photograph (Figure 1.2a). However, despite all of these advances, the dream of having a computer explain an image at the same level of detail and causality as a two-year old remains elusive.

Why is vision so difficult? In part, it is because it 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*, or machine learning from large sets of examples, 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, in many domains, such as rendering a still scene composed of everyday objects or animating extinct creatures such

[1] Some fun pages with striking illusions include https://michaelbach.de/ot, https://www.illusionsindex.org, and http://www.ritsumei.ac.jp/~akitaoka/index-e.html.

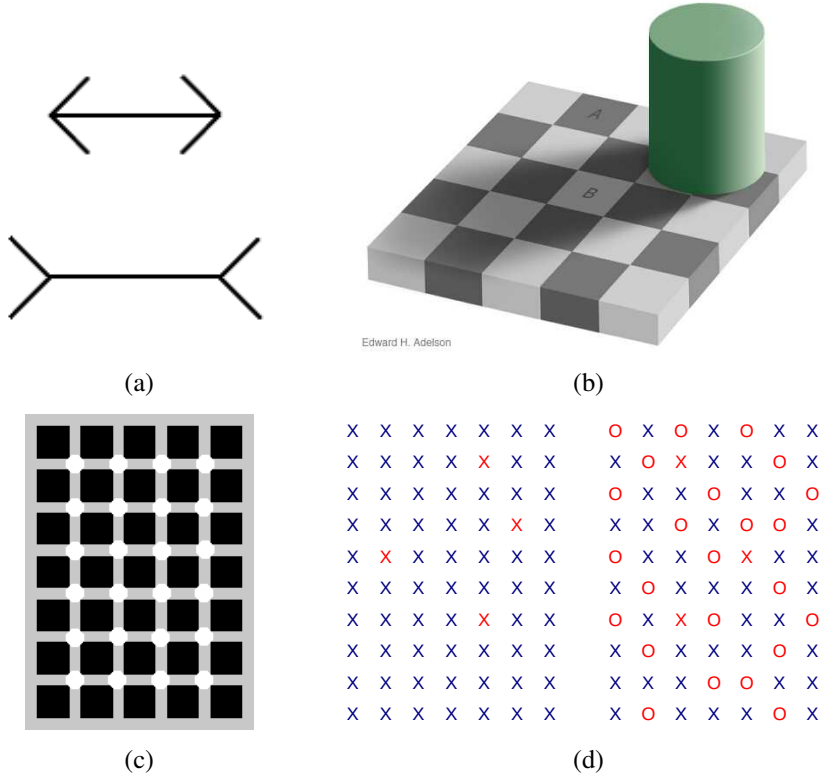


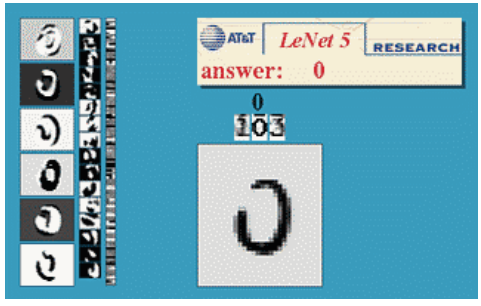
Figure 1.3 *Some common optical illusions and what they might tell us about the visual system: (a) The classic Müller-Lyer illusion, where the lengths of the two horizontal lines appear different, probably due to the imagined perspective effects. (b) The "white" square B in the shadow and the "black" 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://persci.mit.edu/gallery/checkershadow. (c) A variation of the Hermann grid illusion, courtesy of Hany Farid. 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.*

as dinosaurs, the illusion of reality is essentially there.

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. This 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).

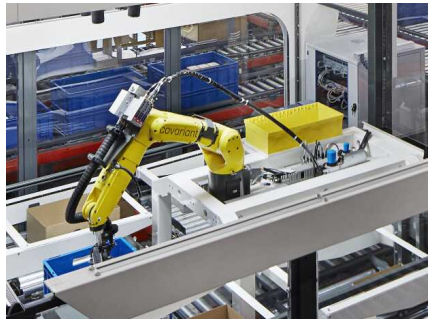
The good news is that computer vision *is* being used today in a wide variety of real-world applications, which include:

- **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 and fully automated stores (Wingfield 2019);
- **Warehouse logistics:** autonomous package delivery and pallet-carrying "drives" (Guizzo 2008; O'Brian 2019) and parts picking by robotic manipulators (Figure 1.4c; Ackerman 2020);
- **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;
- **Self-driving vehicles:** capable of driving point-to-point between cities (Figure 1.4e; Montemerlo, Becker *et al.* 2008; Urmson, Anhalt *et al.* 2008; Janai, Güney *et al.* 2020) as well as autonomous flight (Kaufmann, Gehrig *et al.* 2019);
- **3D model building (photogrammetry):** fully automated construction of 3D models from aerial and drone photographs (Figure 1.4f);
- **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



(a)

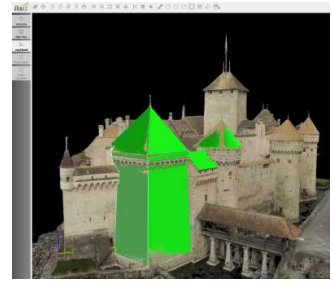
(b)



(c)

(d)

(e)



(f)

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) warehouse picking, https://covariant.ai; (d) medical imaging, http://www.clarontech.com; (e) self-driving cars, (Montemerlo, Becker* et al. *2008) © 2008 Wiley; (f) drone-based photogrammetry, https://www.pix4d.com/blog/mapping-chillon-castle-with-drone.*

precise *matting* to insert new elements between foreground and background elements (Chuang, Agarwala *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 and monitoring pools for drowning victims (e.g., https://swimeye.com);
- **Fingerprint recognition and biometrics:** for automatic access authentication as well as forensic applications.

David Lowe's website 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 addition to all of these industrial applications, there exist myriad *consumer-level* applications, such as things you can do with your own personal photographs and video. These include:

- **Stitching:** turning overlapping photos into a single seamlessly stitched panorama (Figure 1.5a), as described in Section 8.2;
- **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 13.6;
- **Video match move and stabilization:** inserting 2D pictures or 3D models into your videos by automatically tracking nearby reference points (see Section 11.4.4)[2] or using motion estimates to remove shake from your videos (see Section 9.2.1);
- **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 14.1.2 and 14.5.5);

[2]For a fun student project on this topic, see the "PhotoBook" project at http://www.cc.gatech.edu/dvfx/videos/dvfx2005.html.

- **Face detection:** for improved camera focusing as well as more relevant image searching (see Section 6.3.1);

- **Visual authentication:** automatically logging family members onto your home computer as they sit down in front of the webcam (see Section 6.2.4).

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 covered[3] 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. 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 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. If machine learning is being used, it is even more important to have representative unbiased **training data** in sufficient quantity to obtain good results on real-world inputs.

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

[3] 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.

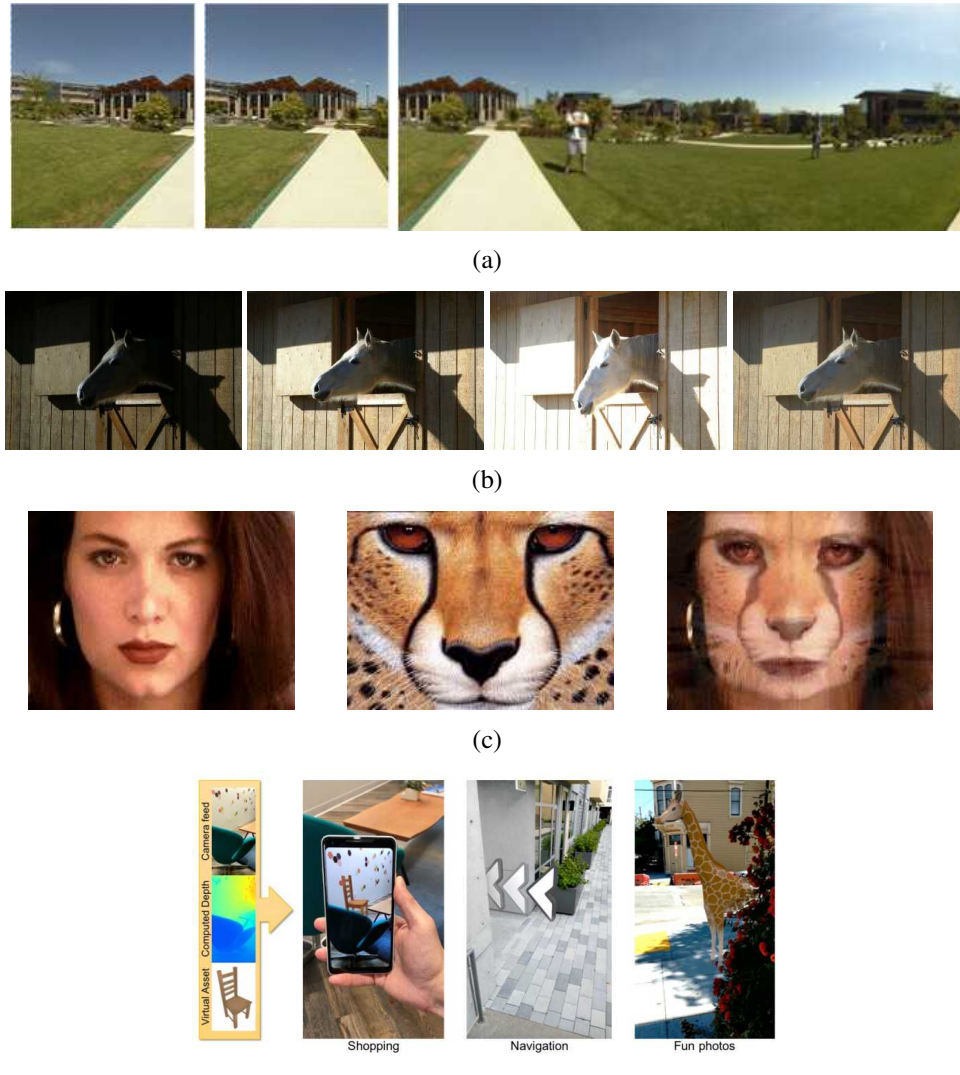


Figure 1.5 *Some consumer applications of computer vision: (a) image stitching: merging different views (Szeliski and Shum 1997) © 1997 ACM; (b) exposure bracketing: merging different exposures; (c) morphing: blending between two photographs (Gomes, Darsa* et al. *1999) © 1999 Morgan Kaufmann; (d) smartphone augmented reality showing real-time depth occlusion effects (Valentin, Kowdle* et al. *2018) © 2018 ACM.*

misestimating 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.[4] 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 interdependencies). 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.

1.2 A brief history

In this section, I provide a brief personal synopsis of the main developments in computer vision over the last fifty years (Figure 1.6) with a focus on advances I find personally interesting and that 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.

[4] In some cases, deep neural networks have also been shown to be an effective way to speed up algorithms that previously relied on iteration (Chen, Xu, and Koltun 2017).

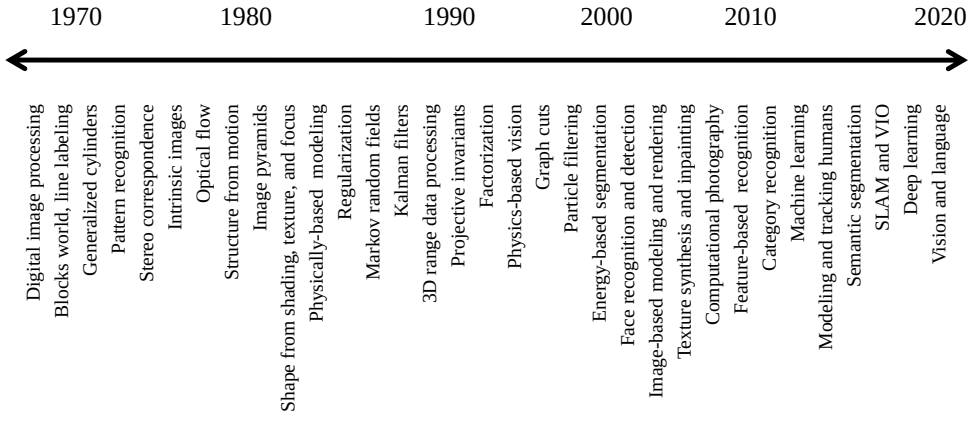


Figure 1.6 *A rough timeline of some of the most active topics of research in computer vision.*

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

[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 six decades, have a look at some of the videos on the Boston Dynamics https://www.bostondynamics.com, Skydio https://www.skydio.com, and Covariant https://covariant.ai websites.

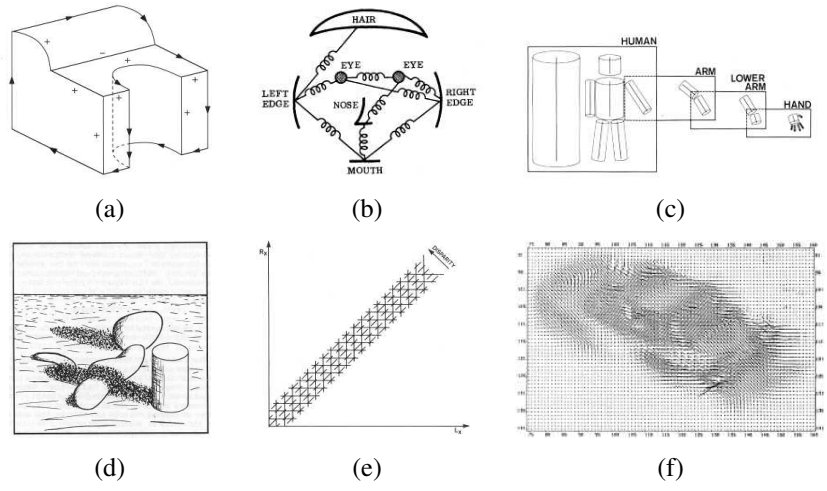


Figure 1.7 *Some early (1970s) examples of computer vision algorithms: (a) line labeling (Nalwa 1993) © 1993 Addison-Wesley, (b) pictorial structures (Fischler and Elschlager 1973) © 1973 IEEE, (c) articulated body model (Marr 1982) © 1982 David Marr, (d) intrinsic images (Barrow and Tenenbaum 1981)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

3D structure of an object or a "blocks world" from the topological structure of the 2D lines (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 relationships[7] (Hinton 1977; Marr 1982) (Figure 1.7c). Fischler and Elschlager (1973) called such *elastic* arrangements of parts *pictorial structures* (Figure 1.7b).

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 has seen

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

periodic revivals, e.g., in the work of Tappen, Freeman, and Adelson (2005) and Barron and Malik (2012).

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, 1979; Barnard and Fischler 1982; Ohta and Kanade 1985; Grimson 1985; Pollard, Mayhew, and Frisby 1985) and intensity-based optical flow algorithms (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 11) 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 prevalent use of graphics chips (GPUs) and many-core architectures for computer vision, this question is again 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

[8] More recent developments in visual perception theory are covered in (Wandell 1995; Palmer 1999; Livingstone 2008; Frisby and Stone 2010).

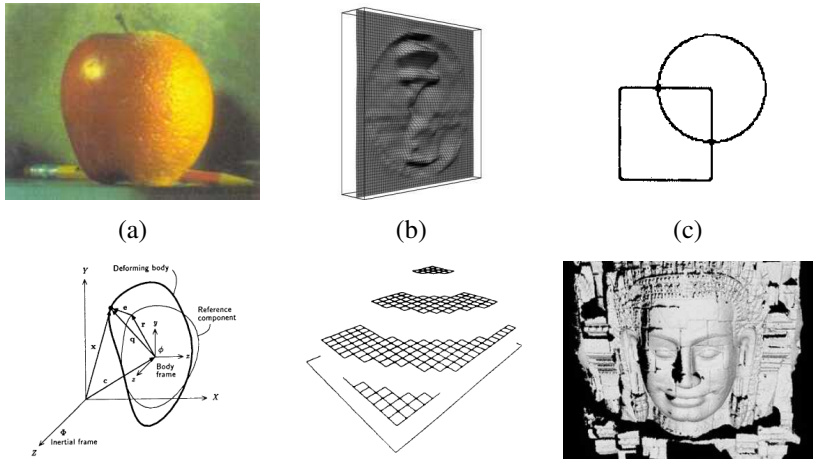


Figure 1.8 *Examples of computer vision algorithms from the 1980s: (a) pyramid blending (Burt and Adelson 1983b) © 1983 ACM, (b) shape from shading (Freeman and Adelson 1991) © 1991 IEEE, (c) edge detection (Freeman and Adelson 1991) © 1991 IEEE, (d) physically based models (Terzopoulos and Witkin 1988)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

and Adelson 1983b; 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 (Mallat 1989; Simoncelli and Adelson 1990a; Simoncelli, Freeman *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 13.1.1 and Horn 1975; Pentland 1984; Blake, Zisserman, and Knowles 1985; Horn and Brooks 1986, 1989), photometric stereo (see Section 13.1.1 and Woodham 1981), shape from texture (see Section 13.1.2 and Witkin 1981; Pentland 1984; Malik and Rosenholtz 1997), and shape from focus (see Section 13.1.3 and Nayar, Watanabe, and Noguchi 1995). Horn (1986) has a nice discussion of most of these techniques.

Research into better edge and contour detection (Figure 1.8c) (see Section 7.2) was also active during this period (Canny 1986; Nalwa and Binford 1986), including the introduction of dynamically evolving contour trackers (Section 7.3.1) 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).

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 and made more robust (well-posed) using regularization (Figure 1.8e) (see Section 4.2 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 4.3), 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 *et al.* 1988; Fischler, Firschein *et al.* 1989). The book by Fischler and Firschein (1987) contains a nice collection of articles focusing on all of these topics (stereo, flow, 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 11). 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 11.4.1) 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 11.4.2 and Taylor, Kriegman, and Anandan 1991;

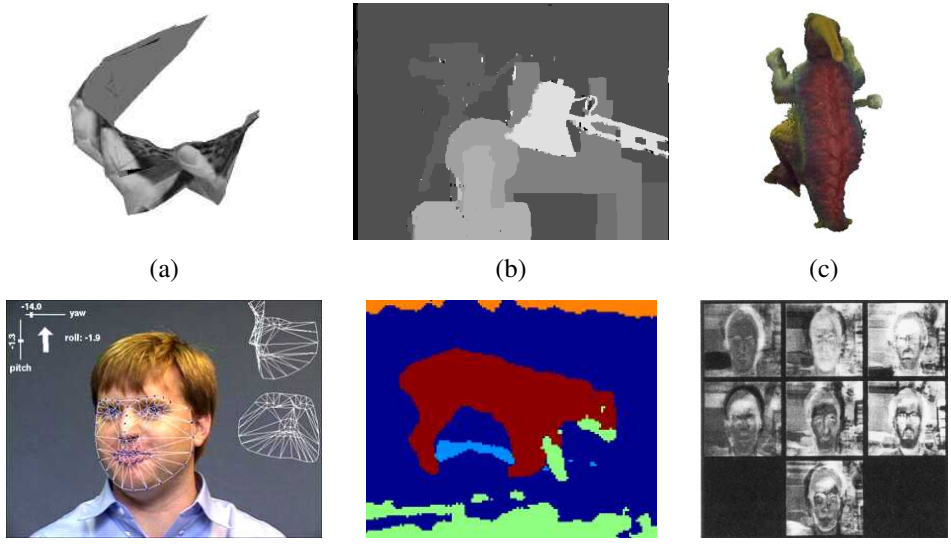


Figure 1.9 *Examples of computer vision algorithms from the 1990s: (a) factorization-based structure from motion (Tomasi and Kanade 1992) © 1992 Springer, (b) dense stereo matching (Boykov, Veksler, and Zabih 2001), (c) multi-view reconstruction (Seitz and Dyer 1999) © 1999 Springer, (d) face tracking (Matthews, Xiao, and Baker 2007), (e) image segmentation (Belongie, Fowlkes* et al. *2002) © 2002 Springer, (f) face recognition (Turk and Pentland 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 *et al.* 1999). Fully automated 3D modeling systems were built using such techniques (Beardsley, Torr, and Zisserman 1996; Schaffalitzky and Zisserman 2002; Snavely, Seitz, and Szeliski 2006; Agarwal, Furukawa *et al.* 2011; Frahm, Fite-Georgel *et al.* 2010).

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 9) continued to be improved (Nagel and Enkelmann 1986; Bolles, Baker, and Marimont 1987; Horn and Weldon Jr. 1988; Anandan 1989; Bergen,

Anandan *et al.* 1992; Black and Anandan 1996; Bruhn, Weickert, and Schnörr 2005; Papenberg, Bruhn *et al.* 2006), with (Nagel 1986; Barron, Fleet, and Beauchemin 1994; Baker, Scharstein *et al.* 2011) being good surveys. Similarly, a lot of progress was made on dense stereo correspondence algorithms (see Chapter 12, 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 12.7) were also an active topic of research (Seitz and Dyer 1999; Kutulakos and Seitz 2000) that continues to be active today (Seitz, Curless *et al.* 2006; Schöps, Schönberger *et al.* 2017; Knapitsch, Park *et al.* 2017). Techniques for producing 3D volumetric descriptions from binary silhouettes (see Section 12.7.3) 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 12.2.1 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 7.3), 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 Section 7.5) (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 5.2.3 and Turk and Pentland 1991) and linear dynamical systems for curve tracking (see Section 7.3.1 and Blake and Isard 1998).

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

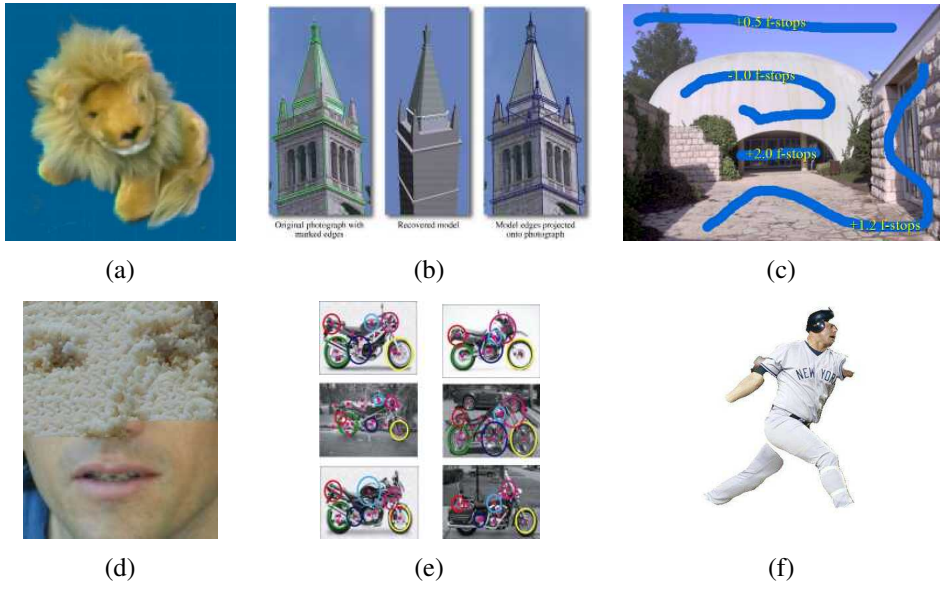


Figure 1.10 *Examples of computer vision algorithms from the 2000s: (a) image-based rendering (Gortler, Grzeszczuk* et al. *1996), (b) image-based modeling (Debevec, Taylor, and Malik 1996) © 1996 ACM, (c) interactive tone mapping (Lischinski, Farbman* et al. *2006) (d) texture synthesis (Efros and Freeman 2001), (e) feature-based recognition (Fergus, Perona, and Zisserman 2007), (f) region-based recognition (Mori, Ren* et al. *2004) © 2004 IEEE.*

cross-disciplinary area of *image-based modeling and rendering* (see Chapter 14). The idea of manipulating real-world imagery directly to create new animations first came to prominence with *image morphing* techniques (Figure1.5c) (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 (Figure1.5a) (see Section 8.2 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 14.3 and Gortler, Grzeszczuk *et al.* 1996; Levoy and Hanrahan 1996; Shade, Gortler *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 decade continued to deepen the interplay between the vision and graphics fields, but more importantly embraced data-driven and learning approaches as core compo-

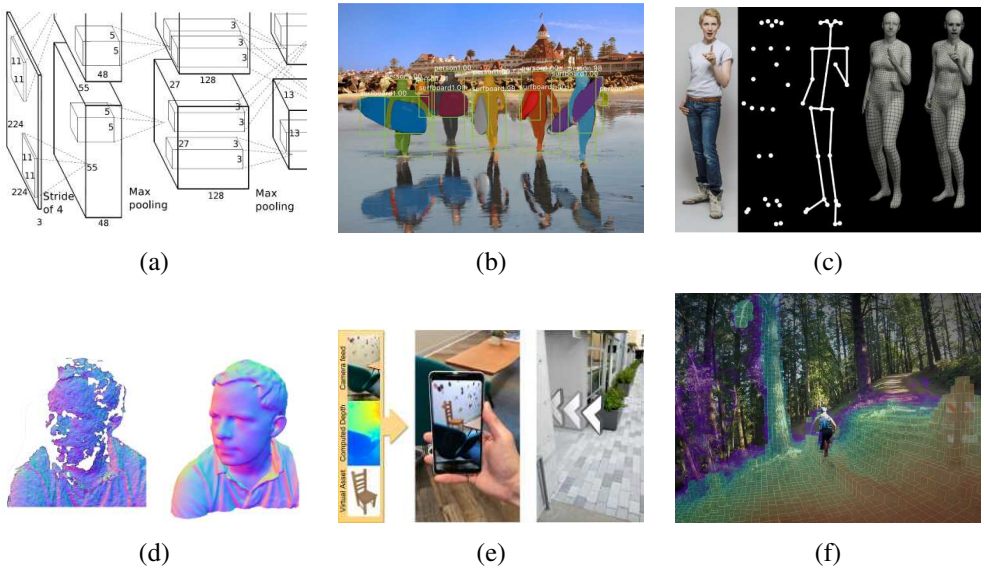
nents of vision. Many of the topics introduced under the rubric of image-based rendering, such as image stitching (see Section 8.2), light-field capture and rendering (see Section 14.3), and *high dynamic range* (HDR) image capture through exposure bracketing (Figure1.5b) (see Section 10.2 and Mann and Picard 1995; Debevec and Malik 1997), were re-christened 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 *et al.* 2002; Lischinski, Farbman *et al.* 2006). In addition to merging multiple exposures, techniques were developed to merge flash images with non-flash counterparts (Eisemann and Durand 2004; Petschnigg, Agrawala *et al.* 2004) and to interactively or automatically select different regions from overlapping images (Agarwala, Dontcheva *et al.* 2004).

Texture synthesis (Figure 1.10d) (see Section 10.5), quilting (Efros and Leung 1999; Efros and Freeman 2001; Kwatra, Schödl *et al.* 2003), and inpainting (Bertalmio, Sapiro *et al.* 2000; Bertalmio, Vese *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 decade was the emergence of feature-based techniques (combined with learning) for object recognition (see Section 6.1 and Ponce, Hebert *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 *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 *et al.* 2004).

Another significant trend from this decade was the development of more efficient algorithms for complex global optimization problems (see Chapter 4 and Appendix B.5 and Szeliski, Zabih *et al.* 2008; Blake, Kohli, and Rother 2011). 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 most notable trend from this decade, which has by now completely taken over visual recognition and most other aspects of computer vision, was the application of sophisticated



(a) (b) (c) (d) (e) (f)

Figure 1.11 *Examples of computer vision algorithms from the 2010s: (a) the SuperVision deep neural network © Krizhevsky, Sutskever, and Hinton (2012); (b) object instance segmentation (He, Gkioxari* et al. *2017) © 2017 IEEE; (c) whole body, expression, and gesture fitting from a single image (Pavlakos, Choutas* et al. *2019) © 2019 IEEE; (d) fusing multiple color depth images using the KinectFusion real-time system (Newcombe, Izadi* et al. *2011) © 2011 IEEE; (e) smartphone augmented reality with real-time depth occlusion effects (Valentin, Kowdle* et al. *2018) © 2018 ACM; (f) 3D map computed in real-time on a fully autonomous Skydio R1 drone (Cross 2019).*

machine learning techniques to computer vision problems (see Chapters 5 and 6). This trend coincided with the increased availability of immense quantities of partially labeled data on the internet, as well as significant increases in computational power, which makes it more feasible to learn object categories without the use of careful human supervision.

2010s. The trend towards using large labeled (and also self-supervised) datasets to develop machine learning algorithms became a tidal wave that totally revolutionized the development of image recognition algorithms as well as other applications, such as denoising and optical flow, which previously used Bayesian and global optimization techniques.

This trend was enabled by the development of high-quality large-scale annotated datasets such as ImageNet (Deng, Dong *et al.* 2009; Russakovsky, Deng *et al.* 2015), Microsoft COCO (Common Objects in Context) (Lin, Maire *et al.* 2014), and LVIS (Gupta, Dollár, and Gir-

shick 2019). These datasets provided not only reliable metrics for tracking the progress of recognition and semantic segmentation algorithms, but more importantly, sufficient labeled data to develop complete solutions based on machine learning.

Another major trend was the dramatic increase in computational power available from the development of general purpose (data-parallel) algorithms on graphical processing units (GPGPU). The breakthrough SuperVision ("AlexNet") deep neural network (Figure 1.11a; Krizhevsky, Sutskever, and Hinton 2012), which was the first neural network to win the yearly ImageNet large-scale visual recognition challenge, relied on GPU training, as well as a number of technical advances, for its dramatic performance. After the publication of this paper, progress in using deep convolutional architectures accelerated dramatically, to the point where they are now the only architecture considered for recognition and semantic segmentation tasks (Figure 1.11b), as well as the preferred architecture for many other vision tasks (Chapter 5; LeCun, Bengio, and Hinton 2015), including optical flow (Sun, Yang *et al.* 2018)), denoising, and monocular depth inference (Li, Dekel *et al.* 2019).

Large datasets and GPU architectures, coupled with the rapid dissemination of ideas through timely publications on arXiv as well as the development of languages for deep learning and the open sourcing of neural network models, all contributed to an explosive growth in this area, both in rapid advances and capabilities, and also in the sheer number of publications and researchers now working on these topics. They also enabled the extension of image recognition approaches to video understanding tasks such as action recognition (Feichtenhofer, Fan *et al.* 2019), as well as structured regression tasks such as real-time multi-person body pose estimation (Cao, Simon *et al.* 2017).

Specialized sensors and hardware for computer vision tasks also continued to advance. The Microsoft Kinect depth camera, released in 2010, quickly became an essential component of many 3D modeling (Figure 1.11d) and person tracking (Shotton, Fitzgibbon *et al.* 2011) systems. Over the decade, 3D body shape modeling and tracking systems continued to evolve, to the point where it is now possible to infer a person's 3D model with gestures and expression from a single image (Figure 1.11c).

And while depth sensors have not yet become ubiquitous (except for security applications on high-end phones), computational photography algorithms run on all of today's smartphones. Innovations introduced in the computer vision community, such as panoramic image stitching and bracketed high dynamic range image merging, are now standard features, and multi-image low-light denoising algorithms are also becoming commonplace (Liba, Murthy *et al.* 2019). Lightfield imaging algorithms, which allow the creation of soft depth-of-field effects, are now also becoming more available (Garg, Wadhwa *et al.* 2019). Finally, mobile augmented reality applications that perform real-time pose estimation and environment

augmentation using combinations of feature tracking and inertial measurements are commonplace, and are currently being extended to include pixel-accurate depth occlusion effects (Figure 1.11e).

On higher-end platforms such as autonomous vehicles and drones, powerful real-time SLAM (simultaneous localization and mapping) and VIO (visual inertial odometry) algorithms (Engel, Schöps, and Cremers 2014; Forster, Zhang *et al.* 2017; Engel, Koltun, and Cremers 2018) can build accurate 3D maps that enable, e.g., autonomous flight through challenging scenes such as forests (Figure 1.11f).

In summary, this past decade has seen incredible advances in the performance and reliability of computer vision algorithms, brought in part by the shift to machine learning and training on very large sets of real-world data. It has also seen the application of vision algorithms in myriad commercial and consumer scenarios as well as new challenges engendered by their widespread use (Su and Crandall 2021).

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 requiring 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 1.12 shows a rough layout of the contents of this book. Since computer vision involves going from images to both a semantic understanding as well as a 3D structural description of the scene, I have positioned the chapters horizontally in terms of where in this spectrum they land, in addition to vertically according to their dependence.[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,

[9] For an interesting comparison with what is known about the human visual system, e.g., the largely parallel *what* and *where* pathways (Goodale and Milner 1992), see some textbooks on human perception (Palmer 1999; Livingstone 2008; Frisby and Stone 2010).

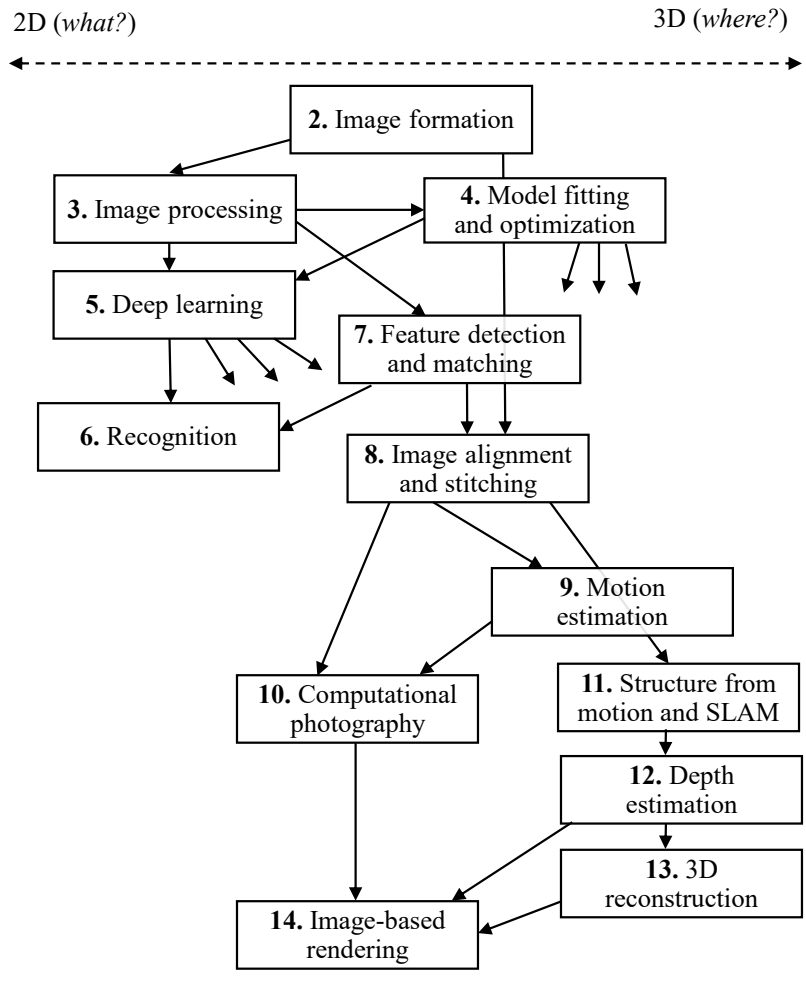


Figure 1.12 *A taxonomy of the topics covered in this book, showing the (rough) dependencies between different chapters, which are roughly positioned along the left–right axis depending on whether they are more closely related to images (left) or 3D geometry (right) representations. The "what-where" along the top axis is a reference to separate visual pathways in the visual system (Goodale and Milner 1992), but should not be taken too seriously. Foundational techniques such as optimization and deep learning are widely used in subsequent chapters.*

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.

If the students or curriculum do not have a strong preference for programming languages, Python, with the NumPy scientific and array arithmetic library plus the OpenCV vision library, are a good environment to develop algorithms and learn about vision. Not only will the students learn how to program using array/tensor notation and linear/matrix algebra (which is a good foundation for later use of PyTorch for deep learning), you can also prepare classroom assignments using Jupyter notebooks, giving you the option to combine descriptive tutorials, sample code, and code to be extended/modified in one convenient location.[10]

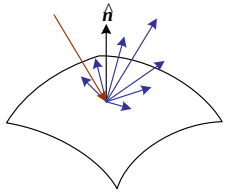
As this is a reference book, I try wherever possible to discuss which techniques and algorithms work well in practice, as well as provide up-to-date pointers to the latest research results in the areas that I cover. The exercises can be used to build up your own personal library of self-tested and validated 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 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), and geometric transformations such as image warping (Section 3.6). Chapter 3 also presents applications such as seamless image blending and image morphing.

Chapter 4 begins with a new section on data fitting and interpolation, which provides a

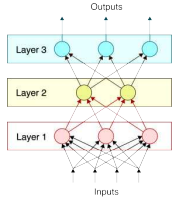
[10] You may also be able to run your notebooks and train your models using the Google Colab service at https://colab.research.google.com.



2. Image formation

3. Image processing

4. Optimization

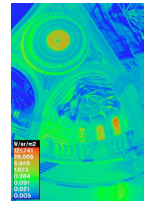


5. Deep learning

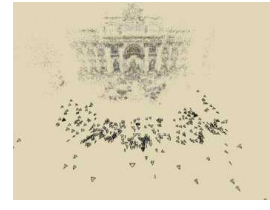
6. Recognition

7–8. Features & alignment

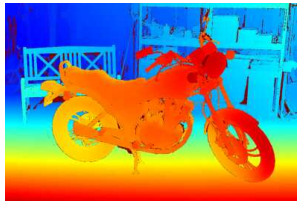
9. Motion estimation



10. Computational Photography

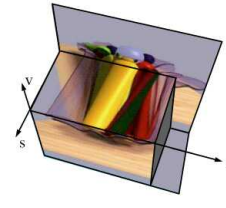


11. Structure from motion



12. Depth estimation

13. 3D reconstruction



14. Image-based Rendering

Figure 1.13 *A pictorial summary of the chapter contents. Sources: Burt and Adelson (1983b); Agarwala, Dontcheva* et al. *(2004); Glassner (2018); He, Gkioxari* et al. *(2017); Brown, Szeliski, and Winder (2005); Butler, Wulff* et al. *(2012); Debevec and Malik (1997); Snavely, Seitz, and Szeliski (2006); Scharstein, Hirschmüller* et al. *(2014); Curless and Levoy (1996); Gortler, Grzeszczuk* et al. *(1996)—see the figures in the respective chapters for copyright information.*

conceptual framework for global optimization techniques such as *regularization* and *Markov random fields* (MRFs), as well as *machine learning*, which we cover in the next chapter. Section 4.2 covers classic regularization techniques, i.e., piecewise-continuous smoothing splines (aka *variational techniques*) implemented using fast iterated linear system solvers, which are still often the method of choice in time-critical applications such as mobile augmented reality. The next section (4.3) presents the related topic of *MRFs*, which also serve as an introduction to Bayesian inference techniques, covered at a more abstract level in Appendix B. The chapter also discusses applications to interactive colorization and segmentation.

Chapter 5 is a completely new chapter covering machine learning, deep learning, and deep neural networks. It begins in Section 5.1 with a review of classic *supervised machine learning* approaches, which are designed to classify images (or regress values) based on intermediate-level features. Section 5.2 looks at *unsupervised learning*, which is useful for both understanding unlabeled training data and providing models of real-world distributions. Section 5.3 presents the basic elements of feedforward neural networks, including weights, layers, and activation functions, as well as methods for network training. Section 5.4 goes into more detail on convolutional networks and their applications to both recognition and image processing. The last section in the chapter discusses more complex networks, including 3D, spatio-temporal, recurrent, and generative networks.

Chapter 6 covers the topic of *recognition*. In the first edition of this book this chapter came last, since it built upon earlier methods such as segmentation and feature matching. With the advent of deep networks, many of these intermediate representations are no longer necessary, since the network can learn them as part of the training process. As so much of computer vision research is now devoted to various recognition topics, I decided to move this chapter up so that students can learn about it earlier in the course.

The chapter begins with the classic problem of *instance recognition*, i.e., finding instances of known 3D objects in cluttered scenes. Section 6.2 covers both traditional and deep network approaches to whole *image classification*, i.e., what used to be called *category recognition*. It also discusses the special case of facial recognition. Section 6.3 presents algorithms for *object detection* (drawing bounding boxes around recognized objects), with a brief review of older approaches to face and pedestrian detection. Section 6.4 covers various flavors of *semantic segmentation* (generating per-pixel labels), including *instance segmentation* (delineating separate objects), *pose estimation* (labeling pixels with body parts), and *panoptic segmentation* (labeling both things and stuff). In Section 6.5, we briefly look at some recent papers in *video understanding* and *action recognition*, while in Section 6.6 we mention some recent work in image captioning and visual question answering.

In Chapter 7, we cover feature detection and matching. A lot of current 3D reconstruction

and recognition techniques are built on extracting and matching *feature points* (Section 7.1), so this is a fundamental technique required by many subsequent chapters (Chapters 8 and 11) and even in instance recognition (Section 6.1). We also cover edge and straight line detection in Sections 7.2 and 7.4, contour tracking in Section 7.3, and low-level segmentation techniques in Section 7.5.

Feature detection and matching are used in Chapter 8 to perform *image alignment* (or *registration*) and *image stitching*. We introduce the basic techniques of feature-based alignment 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 both 2D applications such as image stitching (Section 8.2) and computational photography (Chapter 10), as well as 3D geometric alignment tasks such as pose estimation and structure from motion (Chapter 11).

The second part of Chapter 8 is devoted to *image stitching*, i.e., the construction of large panoramas and composites. While stitching is just one example of *computational photography* (see Chapter 10), there is enough depth here to warrant a separate section. We start by discussing various possible motion models (Section 8.2.1), including planar motion and pure camera rotation. We then discuss global alignment (Section 8.3), 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 8.4), 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. Sections 8.2–8.4 also present 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.

In Chapter 9, we generalize the concept of feature-based image alignment to cover dense intensity-based motion estimation, i.e., *optical flow*. We start with the simplest possible motion models, translational motion (Section 9.1), 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 compensate for camera rotation and zooming, as well as affine or planar perspective motion (Section 9.2). This is then generalized to spline-based motion models (Section 9.2.2) and finally to general per-pixel

optical flow (Section 9.3). We close the chapter in Section 9.4 with a discussion of layered and learned motion models as well as video object segmentation and tracking. Applications of motion estimation techniques include automated morphing, video denoising, and frame interpolation (slow motion).

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 and style transfer.

Starting in Chapter 11, we delve more deeply into techniques for reconstructing 3D models from images. We begin by introducing methods for *intrinsic* camera calibration in Section 11.1 and *3D pose estimation*, i.e., *extrinsic* calibration, in Section 11.2. These sections also describe the applications of single-view reconstruction of building models and 3D *location recognition*. We then cover the topic of *triangulation* (Section 11.2.4), which is the 3D reconstruction of points from matched features when the camera positions are known.

Chapter 11 then moves on to 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. We begin with two-frame structure from motion (Section 11.3), for which algebraic techniques exist, as well as robust sampling techniques such as RANSAC that can discount erroneous feature matches. We then cover techniques for multi-frame structure from motion, including factorization (Section 11.4.1), bundle adjustment (Section 11.4.2), and constrained motion and structure models (Section 11.4.8). We present applications in visual effects (*match move*) and sparse 3D model construction for large (e.g., internet) photo collections. The final part of this chapter (Section 11.5) has a new section on *simultaneous localization and mapping* (SLAM) as well as its applications to autonomous navigation and mobile augmented reality (AR).

In Chapter 12, we turn to the topic of stereo correspondence, which can be thought of as a special case of motion estimation where the camera positions are already known (Section 12.1). This additional knowledge enables stereo algorithms to search over a much smaller space of correspondences to produce dense depth estimates using various combinations of matching criteria, optimization algorithm, and/or deep networks (Sections 12.3–12.6). We also cover *multi-view* stereo algorithms that build a true 3D surface representation instead of just a single depth map (Section 12.7), as well as *monocular depth inference* algorithms

that hallucinate depth maps from just a single image (Section 12.8). Applications of stereo matching include head and gaze tracking, as well as depth-based background replacement (*Z-keying*).

Chapter 13 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 13.1). An alternative to all of these *passive* computer vision techniques is to use *active rangefinding* (Section 13.2), 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 13.3), or using alternative representations such as surface point sets (Section 13.4) or implicit functions (Section 13.5).

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 13.6 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 13.7 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 14, we discuss the large number of image-based rendering techniques that have been developed in the last three decades, including simpler techniques such as view interpolation (Section 14.1), layered depth images (Section 14.2), and sprites and layers (Section 14.2.1), as well as the more general framework of light fields and Lumigraphs (Section 14.3) and higher-order fields such as environment mattes (Section 14.4). Applications of these techniques include navigating 3D collections of photographs using *photo tourism*.

Next, we discuss video-based rendering, which is the temporal extension of image-based rendering. The topics we cover include video-based animation (Section 14.5.1), periodic video turned into *video textures* (Section 14.5.2), and 3D video constructed from multiple video streams (Section 14.5.4). Applications of these techniques include animating still images and creating home tours based on 360° video. We finish the chapter with an overview of the new emerging field of *neural rendering*.

To support the book's use as a textbook, the appendices and associated website 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 that can be used to complement this book, including images and datasets, pointers to software, and course slides.

Week	Chapter	Topics
1.	Chapters 1–2	Introduction and image formation
2.	Chapter 3	Image processing
3.	Chapters 4–5	Optimization and learning
4.	Chapter 5	Deep learning
5.	Chapter 6	Recognition
6.	Chapter 7	Feature detection and matching
7.	Chapter 8	Image alignment and stitching
8.	Chapter 9	Motion estimation
9.	Chapter 10	Computational photography
10.	Chapter 11	Structure from motion
11.	Chapter 12	Depth estimation
12.	Chapter 13	3D reconstruction
13.	Chapter 14	Image-based rendering

Table 1.1 *Sample syllabus for a one semester 13-week course. A 10-week quarter could go into lesser depth or omit some topics.*

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.[11] 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 as both an undergraduate and a graduate-level course in computer vision. The undergraduate course[12] tends to go lighter on the mathematics and takes more time reviewing basics, while the graduate-level course[13] dives more deeply into techniques and assumes the students already have a decent grounding in either vision or related mathematical techniques. Related courses have also been taught on the topics of 3D photography and computational photography. Appendix C.3 and the book's website list other courses that use this book to teach a similar curriculum.

[11] Some universities, such as Stanford (CS231A & 231N), Berkeley (CS194-26/294-26 & 280), and the University of Michigan (EECS 498/598 & 442), now split the material over two courses.

[12] http://www.cs.washington.edu/education/courses/455

[13] http://www.cs.washington.edu/education/courses/576

When Steve and I teach the course, we prefer to give the students several small programming assignments 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.

In the past, we have also asked 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 reserved the last week of the class for student presentations. Sometimes, a few of these projects have actually turned 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 feel for how vision techniques work and how they fail. 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.

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 $\mathbf{v}$ are lower case bold, matrices $\mathbf{M}$ 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, $\mathbf{Mv}$, 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 $\mathbf{R}$ for rotations, $\mathbf{K}$ for calibration matrices, and $\mathbf{I}$ 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 pre-

suppose 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), and Gonzalez and Woods (2017). For computer graphics, popular texts include Hughes, van Dam *et al.* (2013) and Marschner and Shirley (2015), 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, while Murphy (2012) provides a more recent take on the field and Hastie, Tibshirani, and Friedman (2009) a more classic treatment. A great introductory text to deep learning is Glassner (2018), while Goodfellow, Bengio, and Courville (2016) and Zhang, Lipton *et al.* (2021) provide more comprehensive treatments. 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 (Nalwa 1993; Trucco and Verri 1998; Hartley and Zisserman 2004; Forsyth and Ponce 2011; Prince 2012; Davies 2017).

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.[14] 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, graphics, and machine learning conferences, such as CVPR, ECCV, ICCV, SIGGRAPH, and NeurIPS, as well as keeping an eye out for the latest publications on arXiv, will provide a wealth of new ideas. The tutorials offered at these conferences, for which slides or notes are often available online, are also an invaluable resource.

[14] For a comprehensive bibliography and taxonomy of computer vision research, Keith Price's Annotated Computer Vision Bibliography https://www.visionbib.com/bibliography/contents.html is an invaluable resource.

Chapter 2

Image formation

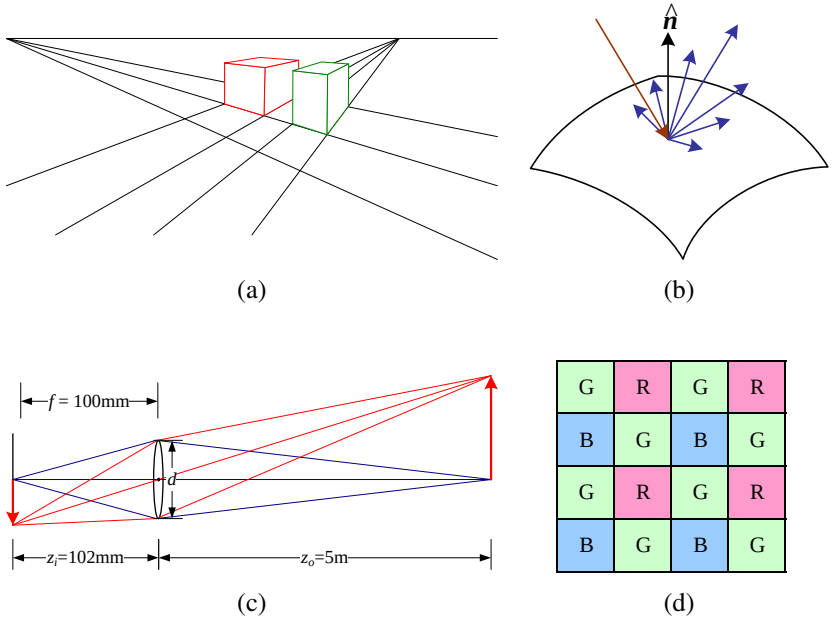


Figure 2.1 *A few components of the image formation process: (a) perspective projection; (b) light scattering when hitting a surface; (c) lens optics; (d) Bayer color filter array.*

Before we can 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 this 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 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 (Hughes, van Dam *et al.* 2013). The image formation (synthesis) process is traditionally taught as part of a computer graphics curriculum (Glassner 1995; Watt 1995; Hughes, van Dam *et al.* 2013; Marschner and Shirley 2015) 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; Ray 2002; Hecht 2015). Some good books on color theory are Healey and Shafer (1992), Wandell (1995), and Wyszecki and Stiles (2000), 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; Gonzalez and Woods 2017). The recent book by Ikeuchi, Matsushita *et al.* (2020) also covers 3D geometry, photometry, and sensor models, with an emphasis on *active illumination* systems.

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 2.1.4 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).

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 curves (Sections 7.3 and 12.2), surfaces (Section 13.3), and volumes (Section 13.5).

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,

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}. \tag{2.1}$$

(As stated in the introduction, we use the $(x_1, x_2, \ldots)$ 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.,

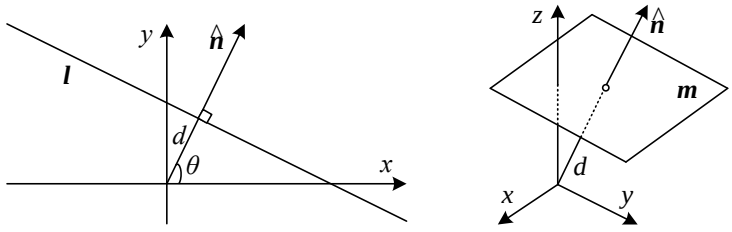
$$\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{w}) = \tilde{w}(x, y, 1) = \tilde{w}\bar{\mathbf{x}}, \tag{2.2}$$

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

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

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.)



(a) (b)

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*.*

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 7.4.2. The combination (θ, d) is also known as *polar coordinates*.

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

$$\tilde{\mathbf{x}} = \tilde{\mathbf{l}}_1 \times \tilde{\mathbf{l}}_2, \tag{2.4}$$

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

$$\tilde{\mathbf{l}} = \tilde{\mathbf{x}}_1 \times \tilde{\mathbf{x}}_2. \tag{2.5}$$

When trying to fit an intersection point to multiple lines or, conversely, a line to multiple points, least squares techniques (Section 8.1.1 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

$$\tilde{\mathbf{x}}^T \mathbf{Q} \tilde{\mathbf{x}} = 0. \tag{2.6}$$

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 $\mathbf{x} = (x, y, z) \in \mathcal{R}^3$ or homogeneous coordinates $\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{z}, \tilde{w}) \in \mathcal{P}^3$. As before,

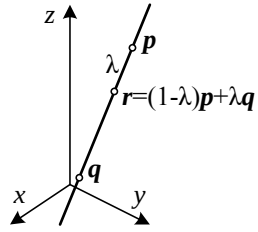


Figure 2.3 *3D line equation,* $\mathbf{r} = (1 - \lambda)\mathbf{p} + \lambda\mathbf{q}$.

it is sometimes useful to denote a 3D point using the augmented vector $\bar{\mathbf{x}} = (x, y, z, 1)$ with $\tilde{\mathbf{x}} = \tilde{w}\bar{\mathbf{x}}$.

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

$$\bar{\mathbf{x}} \cdot \tilde{\mathbf{m}} = ax + by + cz + d = 0. \tag{2.7}$$

We can also normalize the plane equation as $\mathbf{m} = (\hat{n}_x, \hat{n}_y, \hat{n}_z, d) = (\hat{\mathbf{n}}, d)$ with $\|\hat{\mathbf{n}}\| = 1$. In this case, $\hat{\mathbf{n}}$ 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* $\tilde{\mathbf{m}} = (0, 0, 0, 1)$, 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 $\hat{\mathbf{n}}$ as a function of two angles (θ, ϕ),

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

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, $(\mathbf{p}, \mathbf{q})$. Any other point on the line can be expressed as a linear combination of these two points

$$\mathbf{r} = (1 - \lambda)\mathbf{p} + \lambda\mathbf{q}, \tag{2.9}$$

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

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

$$\tilde{\mathbf{r}} = \mu\tilde{\mathbf{p}} + \lambda\tilde{\mathbf{q}}. \tag{2.10}$$

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

$$\mathbf{r} = \mathbf{p} + \lambda\hat{\mathbf{d}}. \tag{2.11}$$

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 14 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, Section 3.2; Faugeras and Luong 2001, Chapter 3). These coordinates are the six independent non-zero entries in the 4 $\times$ 4 skew symmetric matrix

$$\mathbf{L} = \tilde{\mathbf{p}}\tilde{\mathbf{q}}^T - \tilde{\mathbf{q}}\tilde{\mathbf{p}}^T, \tag{2.12}$$

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 $|\mathbf{L}| = 0$, 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 11.4.8) 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.

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

$$\bar{\mathbf{x}}^T\mathbf{Q}\bar{\mathbf{x}} = 0 \tag{2.13}$$

(Hartley and Zisserman 2004, Chapter 3). 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.

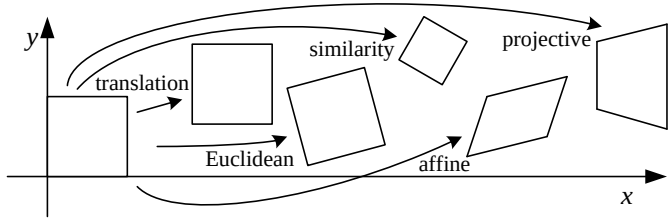


Figure 2.4 *Basic set of 2D planar transformations.*

2.1.1 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 are illustrated in Figure 2.4.

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

$$\mathbf{x}' = \begin{bmatrix} \mathbf{I} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}}, \tag{2.14}$$

where $\mathbf{I}$ is the (2×2) identity matrix or

$$\bar{\mathbf{x}}' = \begin{bmatrix} \mathbf{I} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \bar{\mathbf{x}}, \tag{2.15}$$

where $\mathbf{0}$ 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 $[\mathbf{0}^T \; 1]$ row) makes it possible to chain transformations using matrix multiplication as well as to compute inverse transforms. Note that in any equation where an augmented vector such as $\bar{\mathbf{x}}$ appears on both sides, it can always be replaced with a full homogeneous vector $\tilde{\mathbf{x}}$.

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 $\mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t}$ or

$$\mathbf{x}' = \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}}. \tag{2.16}$$

where

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{2.17}$$

is an orthonormal rotation matrix with $\mathbf{R}\mathbf{R}^T = \mathbf{I}$ and $|\mathbf{R}| = 1$.

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

$$\mathbf{x}' = \begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}} = \begin{bmatrix} a & -b & t_x \\ b & a & t_y \end{bmatrix} \bar{\mathbf{x}}, \tag{2.18}$$

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.,

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

Parallel lines remain parallel under affine transformations.

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

$$\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}\tilde{\mathbf{x}}, \tag{2.20}$$

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.,

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

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 subgroup of the more complex group below it. The mathematics of such *Lie groups* and their

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

Table 2.1 *Hierarchy of 2D coordinate transformations, listing the transformation name, its matrix form, the number of degrees of freedom, what geometric properties it preserves, and a mnemonic icon. 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.*

related algebras (tangent spaces at the origin) are discussed in a number of recent robotics tutorials (Dellaert and Kaess 2017; Blanco 2019; Solà, Deray, and Atchuthan 2019), where the 2D rotation and rigid transforms are called SO(2) and SE(2), which stand for the *special orthogonal* and *special Euclidean* groups.[1]

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 $\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}\tilde{\mathbf{x}}$, we obtain

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

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

[1] The term *special* refers to the desired condition of no reflection, i.e., $\det|\mathbf{R}| = 1$.

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,

$$x' = s_x x + t_x$$
$$y' = s_y y + t_y,$$

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 *et al.* 1992; Girod, Greiner, and Niemann 2000),

$$x' = a_0 + a_1 x + a_2 y + a_6 x^2 + a_7 xy$$
$$y' = a_3 + a_4 x + a_5 y + a_6 xy + a_7 y^2,$$

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),

$$x' = a_0 + a_1 x + a_2 y + a_6 xy$$
$$y' = a_3 + a_4 x + a_5 y + a_7 xy,$$

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 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 9.2.2).

2.1.2 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.

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix}\mathbf{I} & \mathbf{t}\end{bmatrix}_{3\times4}$	3	orientation	
rigid (Euclidean)	$\begin{bmatrix}\mathbf{R} & \mathbf{t}\end{bmatrix}_{3\times4}$	6	lengths	
similarity	$\begin{bmatrix}s\mathbf{R} & \mathbf{t}\end{bmatrix}_{3\times4}$	7	angles	
affine	$\begin{bmatrix}\mathbf{A}\end{bmatrix}_{3\times4}$	12	parallelism	
projective	$\begin{bmatrix}\tilde{\mathbf{H}}\end{bmatrix}_{4\times4}$	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* $[\mathbf{0}^T\ 1]$ *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.*

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

$$\mathbf{x}' = \begin{bmatrix}\mathbf{I} & \mathbf{t}\end{bmatrix}\bar{\mathbf{x}}, \tag{2.23}$$

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

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

$$\mathbf{x}' = \begin{bmatrix}\mathbf{R} & \mathbf{t}\end{bmatrix}\bar{\mathbf{x}}, \tag{2.24}$$

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

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

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

$$\mathbf{x}' = \begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}}. \tag{2.26}$$

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.,

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

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,

$$\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}\tilde{\mathbf{x}}, \tag{2.28}$$

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).

2.1.3 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, as several different 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.[2] 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.[3] For these reasons, we do not even

[2] However, in special situations, such as describing the motion of a pan-tilt head, these angles may be more intuitive.

[3] In robotics, this is sometimes referred to as *gimbal lock*.

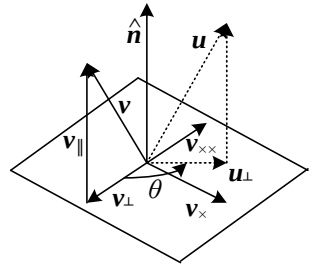


Figure 2.5 *Rotation around an axis* $\hat{\mathbf{n}}$ *by an angle* θ.

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

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

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

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

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

$$\mathbf{v}_{\times} = \hat{\mathbf{n}} \times \mathbf{v}_{\perp} = \hat{\mathbf{n}} \times \mathbf{v} = [\hat{\mathbf{n}}]_{\times}\mathbf{v}, \tag{2.31}$$

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

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

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

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

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

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

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

The product of the axis $\hat{\mathbf{n}}$ and angle θ, $\boldsymbol{\omega} = \theta\hat{\mathbf{n}} = (\omega_x, \omega_y, \omega_z)$, 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, Rodrigues' formula simplifies to

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

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 $\boldsymbol{\omega}$,

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

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 $k \to \infty$, we obtain

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

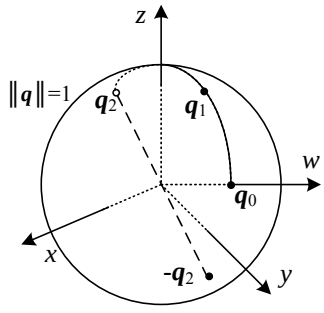


Figure 2.6 *Unit quaternions live on the unit sphere* $\|\mathbf{q}\| = 1$. *This figure shows a smooth trajectory through the three quaternions* $\mathbf{q}_0$, $\mathbf{q}_1$, *and* $\mathbf{q}_2$. *The* antipodal *point to* $\mathbf{q}_2$, *namely* $-\mathbf{q}_2$, *represents the same rotation as* $\mathbf{q}_2$.

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),

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

which yields the familiar Rodrigues' formula.

In robotics (and group theory), rotations are called SO(3), i.e., the *special orthogonal* group in 3D. The incremental rotations $\boldsymbol{\omega}$ are associated with a Lie algebra se(3) and are the preferred way to formulate rotation derivatives and to model uncertainties in rotation estimates (Blanco 2019; Solà, Deray, and Atchuthan 2019).

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, you 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).

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

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

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 $(1-\cos\theta) = 2\sin^2\frac{\theta}{2}$, Rodrigues' formula can be converted to

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

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

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

We thus obtain

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

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

$$\mathbf{q}_2 = \mathbf{q}_0\mathbf{q}_1 = (\mathbf{v}_0 \times \mathbf{v}_1 + w_0\mathbf{v}_1 + w_1\mathbf{v}_0,\ w_0w_1 - \mathbf{v}_0 \cdot \mathbf{v}_1), \tag{2.42}$$

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.

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

$$\mathbf{q}_2 = \mathbf{q}_0/\mathbf{q}_1 = \mathbf{q}_0\mathbf{q}_1^{-1} = (\mathbf{v}_0 \times \mathbf{v}_1 + w_0\mathbf{v}_1 - w_1\mathbf{v}_0,\ -w_0w_1 - \mathbf{v}_0 \cdot \mathbf{v}_1). \tag{2.43}$$

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.*

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,

$$\mathbf{q}_2 = \mathbf{q}_r^\alpha\mathbf{q}_0, \tag{2.44}$$

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

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

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 11.2.2.

2.1.4 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

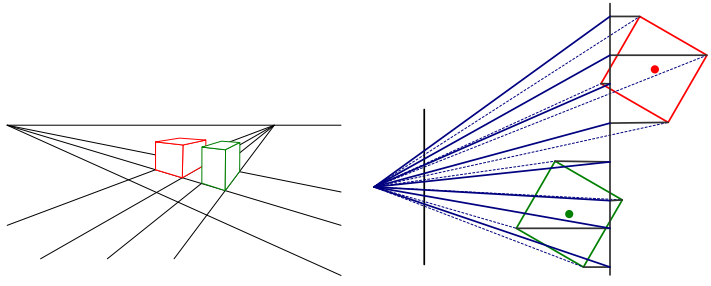
An orthographic projection simply drops the z component of the three-dimensional coordinate $\mathbf{p}$ to obtain the 2D point $\mathbf{x}$. (In this section, we use $\mathbf{p}$ to denote 3D points and $\mathbf{x}$ to denote 2D points.) This can be written as

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

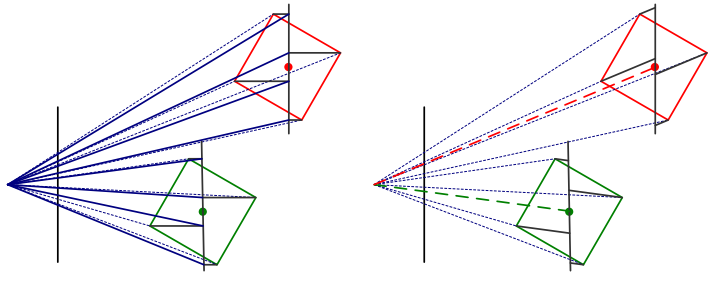
If we are using homogeneous (projective) coordinates, we can write

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

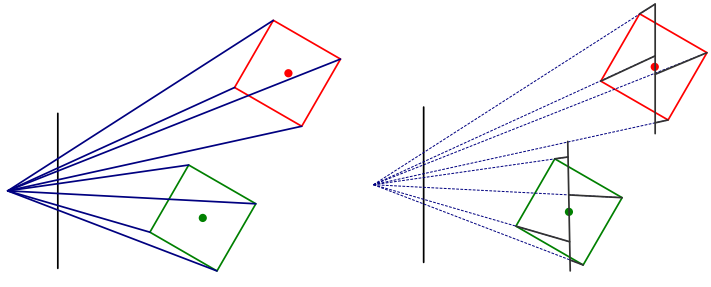
i.e., we drop the z component but keep the w component. Orthography is an approximate model for long focal length (telephoto) lenses and objects whose depth is *shallow* relative to their distance to the camera (Sawhney and Hanson 1991). It is exact only for *telecentric* lenses (Baker and Nayar 1999,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



(a) 3D view (b) orthography



(c) scaled orthography (d) para-perspective



(e) perspective (f) object-centered

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.*

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,

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

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 orientation) can be estimated using simple least squares (Section 11.2.1). Under orthography, structure and motion can simultaneously be estimated using *factorization* (singular value decomposition), as discussed in Section 11.4.1 (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

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

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

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

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

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

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* $(x, y, z) \in [-1, 1] \times [-1, 1] \times [0, 1]$, 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

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

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 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 $z_{\text{near}} = 1$, $z_{\text{far}} \to \infty$, 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 13.2) and stereo matching algorithms (Chapter 12) 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 2.1.4). We can do this if we represent perspective projection using a full-rank 4×4 matrix, as in (2.64).

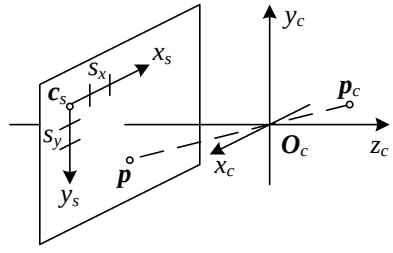


Figure 2.8 *Projection of a 3D camera-centered point $\mathbf{p}_c$ onto the sensor planes at location $\mathbf{p}$. $\mathbf{O}_c$ is the optical center (nodal point), $\mathbf{c}_s$ is the 3D origin of the sensor plane coordinate system, and s_x and s_y are the pixel spacings.*

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 $\mathbf{M}_s$, 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 $\mathbf{K}$, which is used to map 3D camera-centered points $\mathbf{p}_c$ to 2D pixel coordinates $\tilde{\mathbf{x}}_s$.

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 $\mathbf{O}_c$ with an origin $\mathbf{c}_s$ and a 3D rotation $\mathbf{R}_s$ (Figure 2.8).

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

$$\mathbf{p} = \begin{bmatrix} \mathbf{R}_s & \mathbf{c}_s \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix} = \mathbf{M}_s \bar{\mathbf{x}}_s. \tag{2.53}$$

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

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

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 (Section 11.1) 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,

$$\tilde{\mathbf{x}}_s = \mathbf{K} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \mathbf{p}_w = \mathbf{P}\mathbf{p}_w, \tag{2.55}$$

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

$$\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}] \tag{2.56}$$

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.

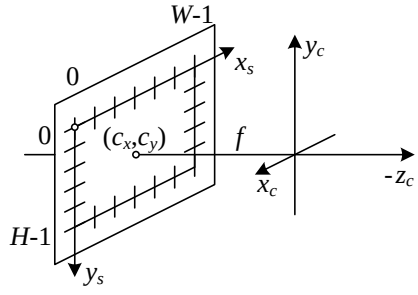


Figure 2.9 *Simplified camera intrinsics showing the focal length f and the image center (c_x, c_y). The image width and height are W and H.*

The choice of an upper-triangular form for $\mathbf{K}$ seems to be conventional. Given a full 3 $\times$ 4 camera matrix $\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}]$, 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

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

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 *image center* expressed in pixel coordinates. The image center is also often called the *principal point* in the computer vision literature (Hartley and Zisserman 2004), although in optics, the principal points are 3D points usually inside the lens where the principal planes intersect the principal (optical) axis (Hecht 2015). Another possibility is

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

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 $a = 1$

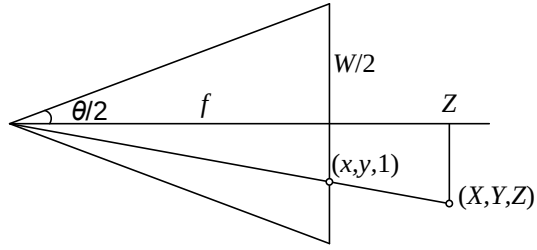


Figure 2.10 *Central projection, showing the relationship between the 3D and 2D coordinates,* $\mathbf{p}$ *and* $\mathbf{x}$*, as well as the relationship between the focal length* f*, image width* W*, and the horizontal field of view* θ_{H}.

and $s = 0$,

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

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 width and height, respectively, 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.

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 horizontal field of view θ_{H}, which obey the formula

$$\tan\frac{\theta_{\mathrm{H}}}{2} = \frac{W}{2f} \qquad \text{or} \qquad f = \frac{W}{2}\left[\tan\frac{\theta_{\mathrm{H}}}{2}\right]^{-1}. \tag{2.60}$$

For a traditional 35mm film camera, whose active exposure area is 24mm × 36mm, we have $W = 36$mm, and hence f is also expressed in millimeters.[4] For example, the "stock" lens that often comes with SLR (single lens reflex) cameras is 50mm, which is a good length, whereas 85mm is the standard for portrait photography. Since we work with digital images, however, 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*,

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

This has the advantage that the focal length f and image 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).[5] 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).) Setting $S = W = 2$ in (2.60), we obtain the simpler (unitless) relationship

$$f^{-1} = \tan \frac{\theta_{\mathrm{H}}}{2}. \tag{2.62}$$

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 18mm.

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*

$$\mathbf{P} = \mathbf{K} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix}. \tag{2.63}$$

[4] 35mm denotes the width of the film strip, of which 24mm is used for exposing each frame and the remaining 11mm for perforation and frame numbering.

[5] 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, as they can become non-integer if they are ever odd at a larger resolution in the pyramid.

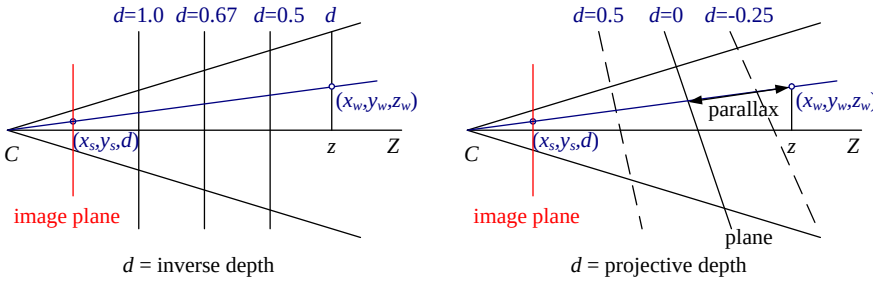


Figure 2.11 *Regular disparity (inverse depth) and projective depth (parallax from a reference plane).*

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,

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

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), $\mathbf{x}_s = (x_s, y_s, 1, d)$,

$$\mathbf{x}_s \sim \tilde{\mathbf{P}}\bar{\mathbf{p}}_w, \tag{2.65}$$

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

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

where $z = \mathbf{p}_2 \cdot \bar{\mathbf{p}}_w = \mathbf{r}_z \cdot (\mathbf{p}_w - \mathbf{c})$ 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 *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,

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

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 stereo reconstruction algorithms, since it allows us to sweep a series of planes (Section 12.1.2) 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

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

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

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

and then project it into another image yielding

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

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

$$\tilde{\mathbf{x}}_1 \sim \tilde{\mathbf{H}}_{10}\tilde{\mathbf{x}}_0, \tag{2.71}$$

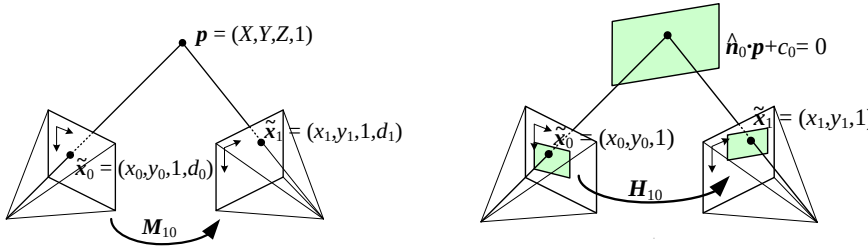


Figure 2.12 *A point is projected into two images: (a) relationship between the 3D point coordinate* $(X, Y, Z, 1)$ *and the 2D projected point* $(x, y, 1, d)$*; (b) planar homography induced by points all lying on a common plane* $\hat{\mathbf{n}}_0 \cdot \mathbf{p} + c_0 = 0$.

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).

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 8.2.3), i.e., when $\mathbf{t}_0 = \mathbf{t}_1$. In this case, we can write

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

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 8.2.3).

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 to a motion towards the user. (This effect was put to dramatic use in some scenes 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),

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

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

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

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

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

(Szeliski and Kang 1994; Pighin, Hecker *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 $\eta_z \to 0$ (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 11.4.1).

2.1.5 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 misregistration of corresponding features before pixel blending (Section 8.2).

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 image center (c_x, c_y), i.e.,

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

The radial distortion model says that coordinates in the observed images are displaced towards (*barrel* distortion) or away (*pincushion* distortion) from the image center by an amount proportional to their radial distance (Figure 2.13a–b).[6] The simplest radial distortion models use low-order polynomials, e.g.,

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

where $r_c^2 = x_c^2 + y_c^2$ and κ_1 and κ_2 are called the *radial distortion parameters*.[7] This model, which also includes a *tangential* component to account for lens decentering, was first proposed in the photogrammetry literature by Brown (1966), and so is sometimes called the *Brown* or *Brown–Conrady* model. However, the tangential components of the distortion are usually ignored because they can lead to less stable estimates (Zhang 2000).

After the radial distortion step, the final pixel coordinates can be computed using

$$\begin{aligned} x_s &= f\hat{x}_c + c_x \\ y_s &= f\hat{y}_c + c_y. \end{aligned} \tag{2.79}$$

A variety of techniques can be used to estimate the radial distortion parameters for a given lens, as discussed in Section 11.1.4.

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).

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*

[6] 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 scaling, i.e., non-square pixels.

[7] 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.

(a) (b) (c)

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

projectors of angles away from the optical axis (Xiong and Turkowski 1997),

$$r = f\theta, \tag{2.80}$$

which is the same as the *polar projection* described by Equations (8.55–8.57). Because of the mostly linear mapping between distance from the center (pixels) and viewing angle, such lenses are sometimes called *f-theta lenses*, which is likely where the popular RICOH THETA 360° camera got its name. 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 *et al.* 1992a; Grossberg and Nayar 2001; Sturm and Ramalingam 2004; Tardif, Sturm *et al.* 2009). Some of these techniques are described in more detail in Section 11.1.4, which discusses how to calibrate lens distortions.

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 $\mathbf{K}$ 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

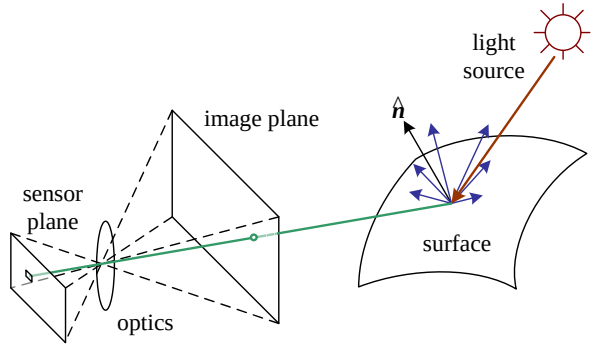


Figure 2.14 *A simplified model of photometric image formation. Light is emitted by one or more light sources and is then reflected from an object's surface. A portion of this light is directed towards the camera. This simplified model ignores multiple reflections, which often occur in real-world scenes.*

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 textbooks on computer graphics and image synthesis (Cohen and Wallace 1993; Sillion and Puech 1994; Watt 1995; Glassner 1995; Weyrich, Lawrence *et al.* 2009; Hughes, van Dam *et al.* 2013; Marschner and Shirley 2015).

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 fluorescence 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 $\hat{\mathbf{v}}$ to color values (or wavelengths, λ),

$$L(\hat{\mathbf{v}}; \lambda), \tag{2.81}$$

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 (sometimes accompanied by a darker sphere to capture highlights) 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.

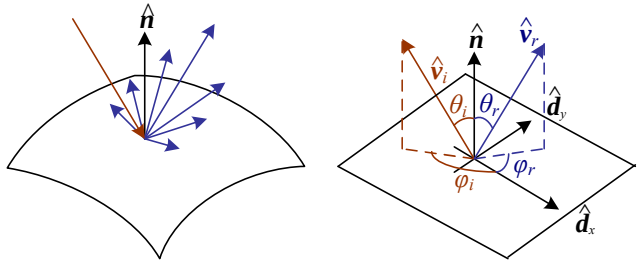


Figure 2.15 *(a) Light scatters when it hits a surface. (b) The bidirectional reflectance distribution function (BRDF) $f(\theta_i, \phi_i, \theta_r, \phi_r)$ is parameterized by the angles that the incident, $\hat{\mathbf{v}}_i$, and reflected, $\hat{\mathbf{v}}_r$, light ray directions make with the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

The Bidirectional Reflectance Distribution Function (BRDF)

The most general model of light scattering is the *bidirectional reflectance distribution function* (BRDF).[8] 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* direction $\hat{\mathbf{v}}_i$ is emitted in a *reflected* direction $\hat{\mathbf{v}}_r$ (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

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r; \lambda). \tag{2.82}$$

The BRDF is *reciprocal*, i.e., because of the physics of light transport, you can interchange the roles of $\hat{\mathbf{v}}_i$ and $\hat{\mathbf{v}}_r$ 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

$$f_r(\theta_i, \theta_r, |\phi_r - \phi_i|; \lambda) \quad \text{or} \quad f_r(\hat{\mathbf{v}}_i, \hat{\mathbf{v}}_r, \hat{\mathbf{n}}; \lambda), \tag{2.83}$$

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

[8] 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 13.7.1—(Dorsey, Rushmeier, and Sillion 2007; Weyrich, Lawrence *et al.* 2009).

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.)*

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

$$L_r(\hat{\mathbf{v}}_r; \lambda) = \int L_i(\hat{\mathbf{v}}_i; \lambda) f_r(\hat{\mathbf{v}}_i, \hat{\mathbf{v}}_r, \hat{\mathbf{n}}; \lambda) \cos^+ \theta_i \, d\hat{\mathbf{v}}_i, \tag{2.84}$$

where

$$\cos^+ \theta_i = \max(0, \cos \theta_i). \tag{2.85}$$

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

$$L_r(\hat{\mathbf{v}}_r; \lambda) = \sum_i L_i(\lambda) f_r(\hat{\mathbf{v}}_i, \hat{\mathbf{v}}_r, \hat{\mathbf{n}}; \lambda) \cos^+ \theta_i. \tag{2.86}$$

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; Lafortune, Foo *et al.* 1997), or through empirical observation (Ward 1992; Westin, Arvo, and Torrance 1992; Dana, van Ginneken *et al.* 1999; Marschner, Westin *et al.* 2000; Matusik, Pfister *et al.* 2003; Dorsey, Rushmeier, and Sillion 2007; Weyrich, Lawrence *et al.* 2009; Shi, Mo *et al.* 2019).[9] Typical BRDFs can often be split into their *diffuse* and *specular* components, as described below.

[9] See http://www1.cs.columbia.edu/CAVE/software/curet for a database of some empirically sampled BRDFs.

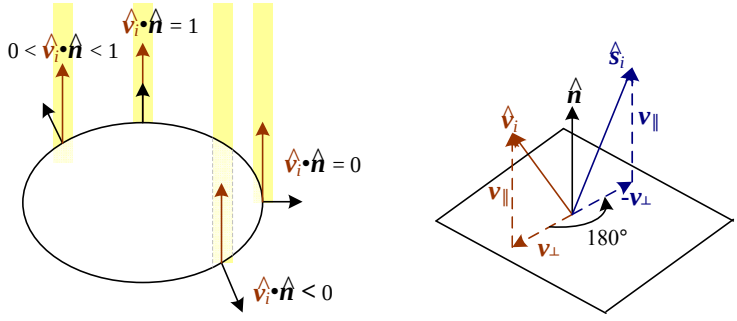


Figure 2.17 *(a) The diminution of returned light caused by* foreshortening *depends on* $\hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}$*, the cosine of the angle between the incident light direction* $\hat{\mathbf{v}}_i$ *and the surface normal* $\hat{\mathbf{n}}$*. (b) Mirror (specular) reflection: The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

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, as 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,

$$f_d(\hat{\mathbf{v}}_i, \hat{\mathbf{v}}_r, \hat{\mathbf{n}}; \lambda) = f_d(\lambda), \tag{2.87}$$

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

$$L_d(\hat{\mathbf{v}}_r; \lambda) = \sum_i L_i(\lambda) f_d(\lambda) \cos^+ \theta_i = \sum_i L_i(\lambda) f_d(\lambda) [\hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}]^+, \tag{2.88}$$

where

$$[\hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}]^+ = \max(0, \hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}). \tag{2.89}$$

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 $\hat{\mathbf{n}}$. Using the same notation as in Equations (2.29–2.30), we can compute the *specular reflection* direction $\hat{\mathbf{s}}_i$ as

$$\hat{\mathbf{s}}_i = \mathbf{v}_{\parallel} - \mathbf{v}_{\perp} = (2\hat{\mathbf{n}}\hat{\mathbf{n}}^T - \mathbf{I})\mathbf{v}_i. \tag{2.90}$$

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

$$f_s(\theta_s; \lambda) = k_s(\lambda) \cos^{k_e} \theta_s, \tag{2.91}$$

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

$$f_s(\theta_s; \lambda) = k_s(\lambda) \exp(-c_s^2 \theta_s^2). \tag{2.92}$$

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 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 $k_a(\lambda)$,

$$f_a(\lambda) = k_a(\lambda) L_a(\lambda). \tag{2.93}$$

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

$$L_r(\hat{\mathbf{v}}_r; \lambda) = k_a(\lambda) L_a(\lambda) + k_d(\lambda) \sum_i L_i(\lambda) [\hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}]^+ + k_s(\lambda) \sum_i L_i(\lambda) (\hat{\mathbf{v}}_r \cdot \hat{\mathbf{s}}_i)^{k_e}. \tag{2.94}$$

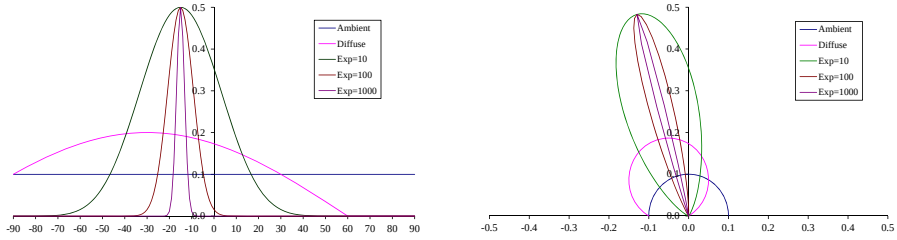


Figure 2.18 *Cross-section through a Phong shading model BRDF for a fixed incident illumination direction: (a) component values as a function of angle away from surface normal; (b) polar plot. The value of the Phong exponent k_e is indicated by the "Exp" labels and the light source is at an angle of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

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 is emphmetallic 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 $\hat{\mathbf{v}}_i$, while the specular component depends on the relative angle between the viewer $\mathbf{v}_r$ and the specular reflection direction $\hat{\mathbf{s}}_i$ (which itself depends on the incoming light direction $\hat{\mathbf{v}}_i$ and the surface normal $\hat{\mathbf{n}}$).

The Phong shading model has been superseded in terms of physical accuracy by newer 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). While, initially, computer graphics hardware implemented the Phong model, the advent of programmable pixel shaders has made 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,

$$L_r(\hat{\mathbf{v}}_r; \lambda) = L_i(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}}; \lambda) + L_b(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}}; \lambda) \tag{2.95}$$

$$= c_i(\lambda) m_i(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}}) + c_b(\lambda) m_b(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}}), \tag{2.96}$$

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 $m(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}})$, 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 rearranging 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 has inspired local two-color models for applications such as Bayer pattern demosaicing (Bennett, Uyttendaele *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; Marschner and Shirley 2015), 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 (2.94))

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 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 *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 13.7.1 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 analog or digital sensor. 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; Ray 2002; Hecht 2015).

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

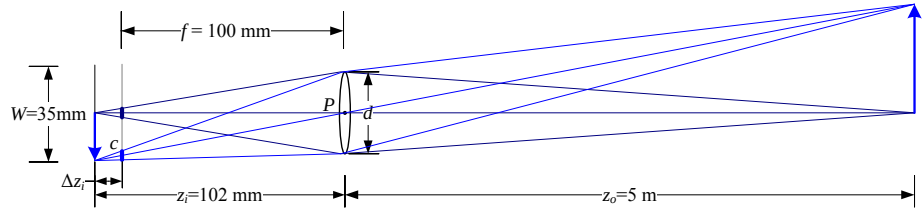


Figure 2.19 *A thin lens of focal length f focuses the light from a plane at a distance z_o in front of the lens onto a plane at a distance z_i behind the lens, where $\frac{1}{z_o} + \frac{1}{z_i} = \frac{1}{f}$. If the focal plane (vertical gray line next to c) is moved forward, the images are no longer 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).*

relationship between the distance to an object z_o and the distance behind the lens at which a focused image is formed z_i can be expressed as

$$\frac{1}{z_o} + \frac{1}{z_i} = \frac{1}{f} \ , \tag{2.97}$$

where f is called the *focal length* of the lens. If we let $z_o \to \infty$, 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 at 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 misfocus is measured by the *circle of confusion* c (shown as short thick blue line segments on the gray plane).[10] 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

[10] 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 *et al.* 2007; Joshi, Szeliski, and Kriegman 2008) or in the "glints" that are seen when shooting into the Sun.

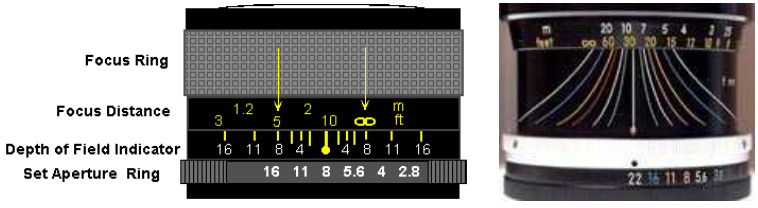


Figure 2.20 *Regular and zoom lens depth of field indicators.*

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

$$f/\# = N = \frac{f}{d} \ , \tag{2.98}$$

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., $f/1.4, f/2, f/2.8, \ldots, f/22$. (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.[11]

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 $1/250$ to $1/125$; 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; Ray 2002; Hecht 2015).

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

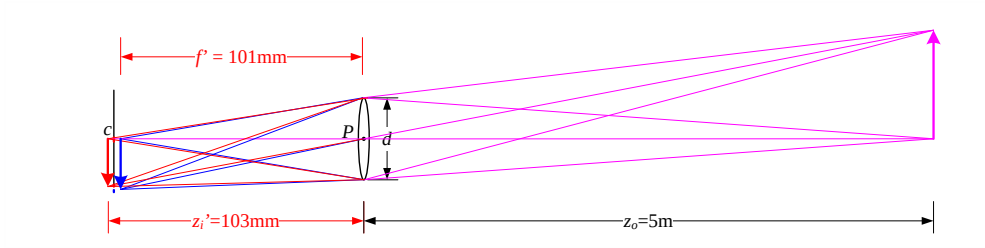


Figure 2.21 *In a lens subject to* chromatic aberration*, light at different wavelengths (e.g., the red and blue arrows) is focused with a different focal length f' and hence a different depth z_i', resulting in both a geometric (in-plane) displacement and a loss of focus.*

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 2.1.5) and, hence, calibrated using the techniques described in Section 11.1.4. 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.

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 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 8.2.3 and Littlefield (2006) and Houghton (2013)).

Not all lenses, however, can be modeled as having a single nodal point. In particular, very wide-angle lenses such as fisheye lenses (Section 2.1.5) 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

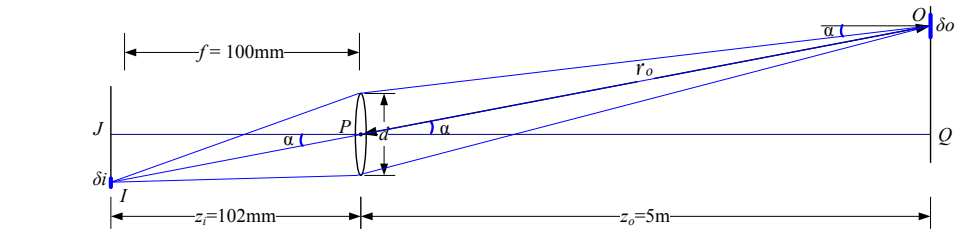


Figure 2.22 *The amount of light hitting a pixel of surface area δi depends on the square of the ratio of the aperture diameter d to the focal length f, as well as the fourth power of the off-axis angle α cosine,* $\cos^4 \alpha$.

explicitly construct a mapping function (look-up table) between pixel coordinates and 3D rays in space (Gremban, Thorpe, and Kanade 1988; Champleboux, Lavallée *et al.* 1992a; Grossberg and Nayar 2001; Sturm and Ramalingam 2004; Tardif, Sturm *et al.* 2009), as mentioned in Section 2.1.5.

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

$$\frac{\delta o \cos \alpha}{r_o^2} \pi \left(\frac{d}{2}\right)^2 \cos \alpha = \delta o \frac{\pi}{4} \frac{d^2}{z_o^2} \cos^4 \alpha. \tag{2.99}$$

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 $z_o : z_i$,

$$\frac{\delta o}{\delta i} = \frac{z_o^2}{z_i^2}. \tag{2.100}$$

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,

$$\delta o \frac{\pi}{4} \frac{d^2}{z_o^2} \cos^4 \alpha = \delta i \frac{\pi}{4} \frac{d^2}{z_i^2} \cos^4 \alpha \approx \delta i \frac{\pi}{4} \left(\frac{d}{f}\right)^2 \cos^4 \alpha, \tag{2.101}$$

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

$$E = L \frac{\pi}{4} \left(\frac{d}{f}\right)^2 \cos^4 \alpha, \tag{2.102}$$

(Horn 1986; Nalwa 1993; Ray 2002; Hecht 2015). 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 $N = f/d$ (2.98), 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.[12] 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), and Liu, Szeliski *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 digital cameras, while Kim, Lin *et al.* (2012), Hasinoff, Sharlet *et al.* (2016), and Karaimer and Brown

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

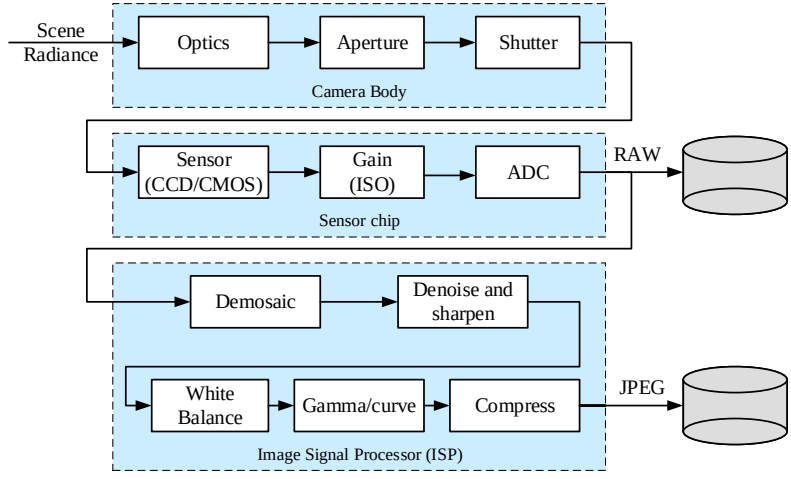


Figure 2.23 *Image sensing pipeline, showing the various sources of noise as well as typical digital post-processing steps.*

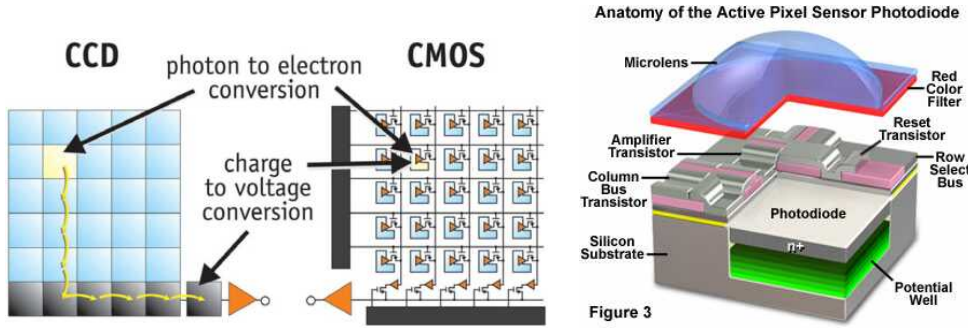
(2016) provide more recent models of modern in-camera processing pipelines. Most recently, Brooks, Mildenhall *et al.* (2019) have developed detailed models of in-camera image processing pipelines to invert (*unprocess*) noisy JPEG images into their RAW originals, so that they can be better denoised, while Tseng, Yu *et al.* (2019) develop a tunable model of camera processing pipelines that can be used for image quality optimization.

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" until they are deposited at the sense amplifiers, which amplify the signal and pass it to an analog-to-digital converter (ADC).[13] 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

[13] 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*.



(a) (b)

Figure 2.24 *Digital imaging sensors: (a) CCDs move photogenerated charge from pixel to pixel and convert it to voltage at the output node; CMOS imagers convert charge to voltage inside each pixel (Litwiller 2005) © 2005 Photonics Spectra; (b) cutaway diagram of a CMOS pixel sensor, from https://micro.magnet.fsu.edu/primer/digitalimaging/cmosimagesensors.html.*

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 websites.[14]

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 is 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. 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.

[14] http://www.clarkvision.com/imagedetail/digital.sensor.performance.summary

Sampling pitch. The sampling pitch is the physical spacing between adjacent sensor cells on the imaging chip (Figure 2.24). 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). While the fill factor was originally limited by the need to place additional electronics between the active sensing areas, modern *backside illumination* (or *back-illuminated*) sensors, coupled with efficient microlens designs, have largely removed this limitation (Fontaine 2015).[15] 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 sensors[16]), while digital SLR cameras try to come closer to the traditional size of a 35mm film frame.[17] 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 exponentially 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

[15] https://en.wikipedia.org/wiki/Back-illuminated_sensor

[16] These numbers refer to the "tube diameter" of the old vidicon tubes used in video cameras. 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.

[17] 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 × 1.6 = 80mm lens, as demonstrated in (2.60).

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*.

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 (Alter, Matsushita, and Tang 2006; Matsushita and Lin 2007a; Wilburn, Xu, and Matsushita 2008; Takamatsu, Matsushita, and Ikeuchi 2008).

As discussed in more detail in Section 10.1.1, Liu, Szeliski *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 *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.

Newer imaging sensors. The capabilities of imaging sensor and related technologies such as depth sensors continue to evolve rapidly. Conferences that track these developments include the IS&T Symposium on Electronic Imaging Science and Technology sponsored by the Society for Imaging Science and Technology and the Image Sensors World blog.

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, as shown in Figure 2.24. 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 2.25), 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*.[18] 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

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

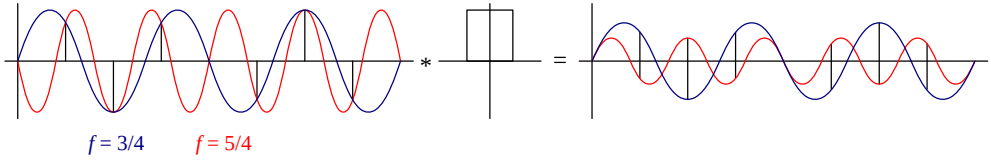


Figure 2.25 *Aliasing of a one-dimensional signal: The blue sine wave at $f = 3/4$ and the red sine wave at $f = 5/4$ have the same digital samples, when sampled at $f = 2$. Even after convolution with a 100% fill factor box filter, the two signals, while no longer of the same magnitude, are still aliased 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.)*

from its instantaneous samples must be at least twice the highest frequency,[19]

$$f_s \geq 2 f_{\max}. \tag{2.103}$$

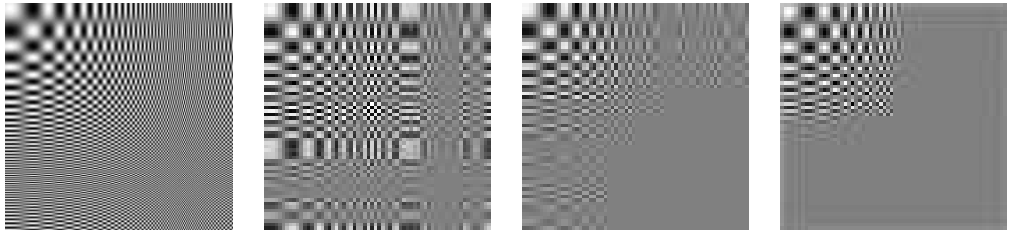
The maximum frequency in a signal is known as the *Nyquist frequency* and the inverse of the minimum sampling frequency $r_s = 1/f_s$ is known as the *Nyquist rate*.

However, you may ask, as 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 2.25, 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 2.26 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-quality 9-tap filter. Additional examples of downsampling (*decimation*) filters can be found in Section 3.5.2 and Figure 3.29.

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

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



(a) (b) (c) (d)

Figure 2.26 *Aliasing of a two-dimensional signal: (a) original full-resolution image; (b) downsampled 4 × with a 25% fill factor box filter; (c) downsampled 4 × with a 100% fill factor box filter; (d) downsampled 4 × 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.*

integration area of a chip sensor.[20]

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 2.27a shows the one-dimensional cross-section of a PSF for a lens whose blur function is assumed to be a disc with 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.[21] If we defocus the lens so that the blur function has a radius of $2s$ (Figure 2.27c), 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 pinhole in a black piece of cardboard lit from behind. However, this PSF (the actual image of the pinhole) 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

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

[21] 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).

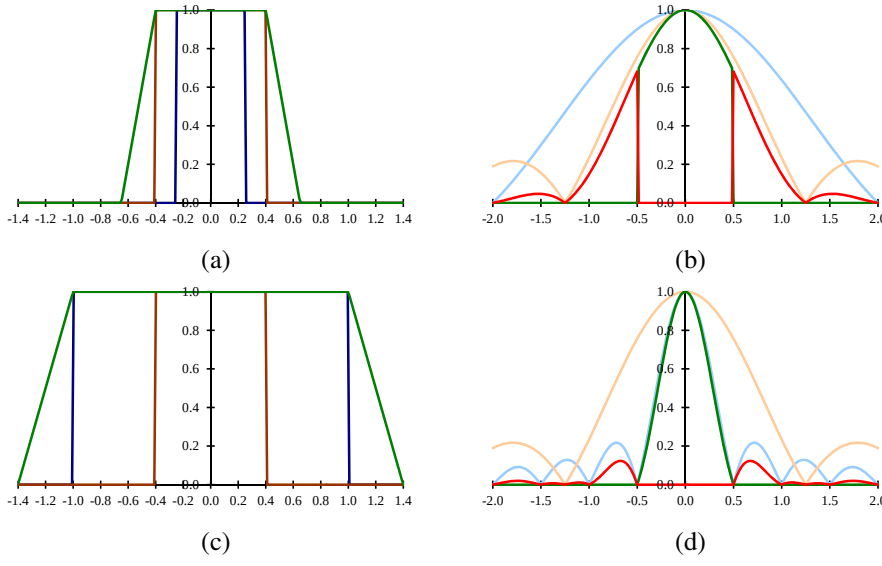


Figure 2.27 *Sample point spread functions (PSF): The diameter of the blur disc (blue) in (a) is equal to half the pixel spacing, while the diameter in (c) is twice the pixel spacing. The horizontal fill factor of the sensing chip is 80% and is shown in brown. The convolution of these two kernels gives 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.*

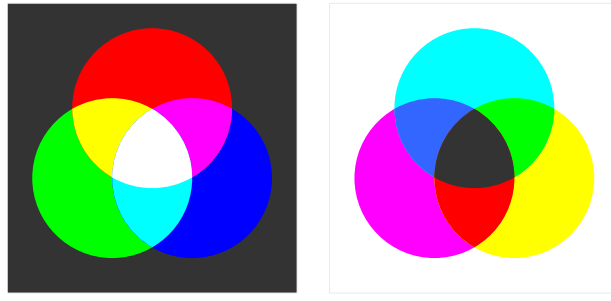
Kriegman 2008)) whose ideal appearance can be re-synthesized to sub-pixel precision.

In addition to occurring during image acquisition, aliasing can also be introduced in various 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.

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



(a) (b)

Figure 2.28 *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.*

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 2.28b), although black is also often used in four-color printing (CMYK).[22] 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.

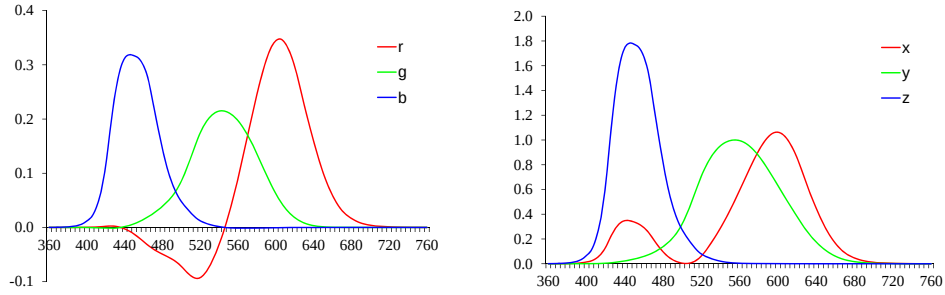
Later on, you may have learned about the *additive* primary colors (red, green, and blue) and how they can be added (with a slide projector or on a computer monitor) to produce cyan, magenta, yellow, white, and all the other colors we typically see on our TV sets and monitors (Figure 2.28a).

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 cells called cones, each of which responds selectively to a different portion of the color spectrum (Glassner 1995; Wandell 1995; Wyszecki and Stiles 2000; Livingstone 2008; Frisby and Stone 2010; Reinhard, Heidrich *et al.* 2010; Fairchild 2013).[23] Note that for machine

[22] It is possible to use additional inks such as orange, green, and violet to further extend the color gamut.

[23] See also Mark Fairchild's web page, http://markfairchild.org/WhyIsColor/books_links.html.



(a) (b)

Figure 2.29 *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.*

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 2.29 shows the results of performing these experiments with a *standard observer*, i.e., averaging perceptual results over a large number of subjects.[24] 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 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 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

[24] As Michael Brown notes in his tutorial on color (Brown 2019), the standard observer is actually an average taken over only 17 British subjects in the 1920s.

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

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \frac{1}{0.17697} \begin{bmatrix} 0.49 & 0.31 & 0.20 \\ 0.17697 & 0.81240 & 0.01063 \\ 0.00 & 0.01 & 0.99 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{2.104}$$

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 2.29a according to (2.104), we obtain the resulting $(\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda))$ curves shown in Figure 2.29b. 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*

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z}, \tag{2.105}$$

which sum 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 2.29b from $\lambda = 380$nm to $\lambda = 800$nm, we obtain the familiar *chromaticity diagram* shown in Figure 2.30a. 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 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*. The inset triangle spans the red, green, and blue single-wavelength primaries used in the original color matching experiments, while E denotes the white point.

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.

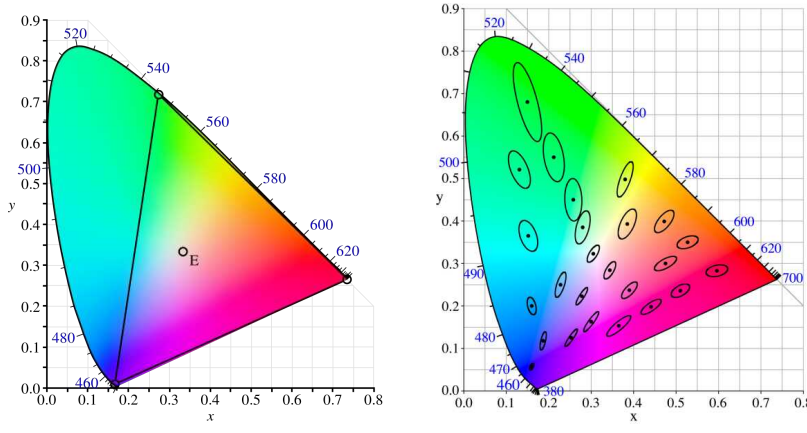


Figure 2.30 *CIE chromaticity diagram, showing the pure single-wavelength spectral colors along the perimeter and the white point at E, plotted along their corresponding* (x, y) *values. (a) the red, green, and blue primaries do not span the complete gamut, so that negative amounts of red need to be added to span the blue–green range; (b) the MacAdam ellipses show color regions of equal discriminability, and form the basis of the Lab perceptual color space.*

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, as shown in Figure 2.30b.[25]

The L* component of *lightness* is defined as

$$L^* = 116 f\left(\frac{Y}{Y_n}\right), \tag{2.106}$$

where Y_n is the luminance value for nominal white (Fairchild 2013) and

$$f(t) = \begin{cases} t^{1/3} & t > \delta^3 \\ t/(3\delta^2) + 2\delta/3 & \text{else,} \end{cases} \tag{2.107}$$

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.

[25] Another perceptually motivated color space called L*u*v* was developed and standardized simultaneously (Fairchild 2013).

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

$$a^* = 500\left[f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right)\right] \quad \text{and} \quad b^* = 200\left[f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right)\right], \tag{2.108}$$

where again, (X_n, Y_n, Z_n) is the measured white point. Figure 2.33i–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 tristimulus 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,

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \begin{bmatrix} R_{709} \\ G_{709} \\ B_{709} \end{bmatrix}. \tag{2.109}$$

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

$$\begin{aligned} R &= \int L(\lambda) S_R(\lambda) d\lambda, \\ G &= \int L(\lambda) S_G(\lambda) d\lambda, \\ B &= \int L(\lambda) S_B(\lambda) d\lambda, \end{aligned} \tag{2.110}$$

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 these 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 (2.109), 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.[26] 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.

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 (Figure 2.24).[27]

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 2.31). 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 prefilter RGB (and monochrome) images to enhance

[26] The latest OLED TV monitors are now introducing higher dynamic range (HDR) and wide color gamut (WCG), https://www.cnet.com/how-to/what-is-wide-color-gamut-wcg.

[27] A chip design by Foveon stacked the red, green, and blue sensors beneath each other, but it never gained widespread adoption. Descriptions of alternative color filter arrays that have been proposed over the years can be found at https://en.wikipedia.org/wiki/Color_filter_array.

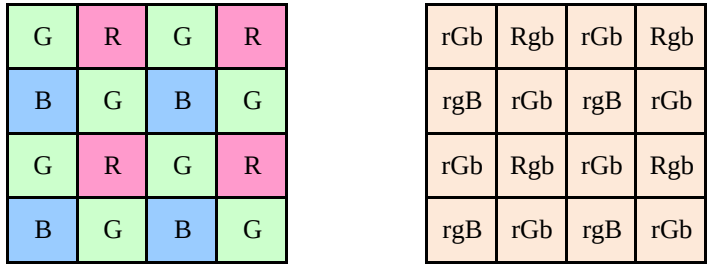


Figure 2.31 *Bayer RGB pattern: (a) color filter array layout; (b) interpolated pixel values, with unknown (guessed) values shown as lower case.*

the perception of crispness (Betrisey, Blinn *et al.* 2000; Platt 2000b).

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 D_{65} 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 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.[28] Exercise 2.8 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

$$B = V^{\gamma}, \tag{2.111}$$

[28] 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.

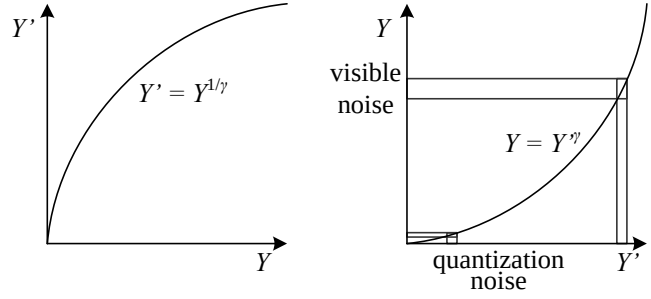


Figure 2.32 *Gamma compression: (a) The relationship between the input signal luminance Y and the transmitted signal Y' is given by $Y' = Y^{1/\gamma}$. (b) At the receiver, the signal Y' is exponentiated by the factor γ, $\hat{Y} = Y'^{\gamma}$. Noise introduced during transmission is squashed in the dark regions, which corresponds to the more noise-sensitive region of the visual system.*

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,

$$Y' = Y^{\frac{1}{\gamma}}, \tag{2.112}$$

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 2.32).[29] (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 remains useful.

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 (2.88)) 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

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

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 13.1.1) 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.

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 2.9 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

$$Y'_{601} = 0.299R' + 0.587G' + 0.114B', \tag{2.113}$$

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

$$Y'_{709} = 0.2125R' + 0.7154G' + 0.0721B'. \tag{2.114}$$

The UV components are derived from scaled versions of $(B'-Y')$ and $(R'-Y')$, namely,

$$U = 0.492111(B' - Y') \quad \text{and} \quad V = 0.877283(R' - Y'), \tag{2.115}$$

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 $[16 \ldots 235]$ range of values, while the Cb and Cr signals are scaled to fit within $[16 \ldots 240]$ (Gomes and Velho 1997; Fairchild 2013). For still images, the JPEG standard uses the full eight-bit range with no reserved values,

$$\begin{bmatrix} Y' \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}, \tag{2.116}$$

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 4.3), 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), Hughes, van Dam *et al.* (2013), and Brown (2019)). Such a decomposition is quite natural in graphics applications such as color picking (it approximates the Munsell chart for color description). Figure 2.33l–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 (2.105) or the even simpler *color ratios*,

$$r = \frac{R}{R+G+B}, \quad g = \frac{G}{R+G+B}, \quad b = \frac{B}{R+G+B} \tag{2.117}$$

(Figure 2.33e–h). After manipulating the luma (2.113), 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*, https://www.poynton.com/ColorFAQ.html, notes that the perceptually motivated L*a*b* system is qualitatively similar 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.

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 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 3.4.1). The DCT is a reasonable approximation to the Karhunen–Loève 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;

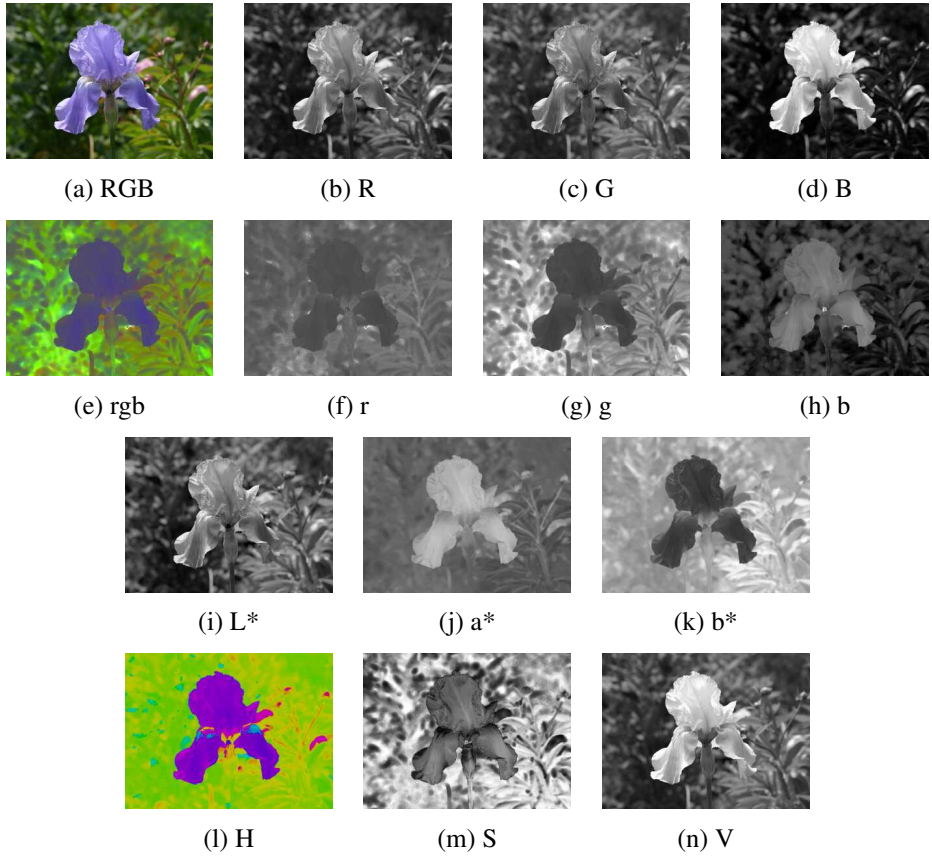


Figure 2.33 *Color space transformations: (a–d) RGB; (e–h) rgb. (i–k) L*a*b*; (l–n) HSV. Note that the rgb, L*a*b*, and HSV values are all re-scaled to fit the dynamic range of the printed page.*

Figure 2.34 *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.*

Le Gall 1991), although newer variants, including the new AV1 open standard,[30] use smaller 4×4 or even 2×2 blocks. Alternative transformations, such as wavelets (Taubman and Marcellin 2002) and lapped transforms (Malvar 1990, 1998, 2000) are used in compression standards such as JPEG 2000 and JPEG XR.

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; Marpe, Schwarz, and Wiegand 2003). (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 2.34).

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 blocks, sub-pixel motions, and the ability to reference blocks from older frames (Sullivan, Ohm *et al.* 2012). 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*,

$$MSE = \frac{1}{n} \sum_{\mathbf{x}} \left[I(\mathbf{x}) - \hat{I}(\mathbf{x}) \right]^2, \tag{2.118}$$

[30] https://aomedia.org

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

$$RMS = \sqrt{MSE}. \tag{2.119}$$

The PSNR is defined as

$$PSNR = 10 \log_{10} \frac{I_{\max}^2}{MSE} = 20 \log_{10} \frac{I_{\max}}{RMS}, \tag{2.120}$$

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. Note also that researchers are currently developing novel image and video compression algorithms based on deep neural networks, e.g., (Rippel and Bourdev 2017; Mentzer, Agustsson *et al.* 2019; Rippel, Nair *et al.* 2019) and https://www.compression.cc. It will be interesting to see what kinds of different artifacts these techniques produce.

2.4 Additional reading

As we mentioned at the beginning of this chapter, this book 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 (Faugeras and Luong 2001; Hartley and Zisserman 2004) and computer graphics (Watt 1995; OpenGL-ARB 1997; Hughes, van Dam *et al.* 2013; Marschner and Shirley 2015). 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 (Glassner 1995; Watt 1995; Hughes, van Dam *et al.* 2013; Marschner and Shirley 2015) 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; Ray 2002; Hecht 2015).

Some good books on color theory have been written by Healey and Shafer (1992), Wandell (1995), Wyszecki and Stiles (2000), and Fairchild (2013), with Livingstone (2008) providing a more fun and informal introduction to the topic of color perception. Mark Fairchild's page of color books and links[31] lists many other sources.

[31] http://markfairchild.org/WhyIsColor/books_links.html.

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; Gonzalez and Woods 2017).

Two courses that cover many of the above topics (image formation, lenses, color and sampling theory) in wonderful detail are Marc Levoy's Digital Photography course at Stanford (Levoy 2010) and Michael Brown's tutorial on the image processing pipeline at ICCV 2019 (Brown 2019). The recent book by Ikeuchi, Matsushita *et al.* (2020) also covers 3D geometry, photometry, and sensor models, but with an emphasis on *active illumination* systems.

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 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 $\tilde{\mathbf{x}}$ that minimizes the sum of squared distances to each line,

$$D = \sum_i (\tilde{\mathbf{x}} \cdot \tilde{\mathbf{l}}_i)^2, \tag{2.121}$$

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

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 Section 8.2). 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: 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 2.9: 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.

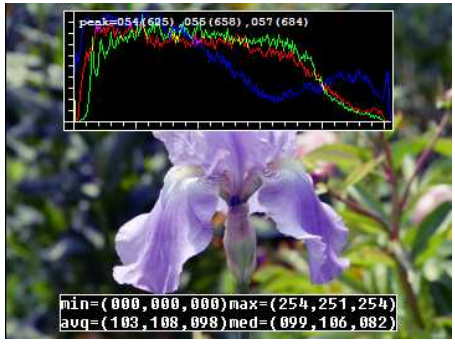
2. Evaluate the quality of the demosaicing algorithm by taking pictures of challenging scenes which contain strong color edges (such as those shown in in Section 10.3.1).

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 in (Chakrabarti, Scharstein, and Zickler 2009), Kim, Lin *et al.* (2012), Hasinoff, Sharlet *et al.* (2016), Karaimer and Brown (2016), and Brooks, Mildenhall *et al.* (2019) or use the data available in their database of color images.

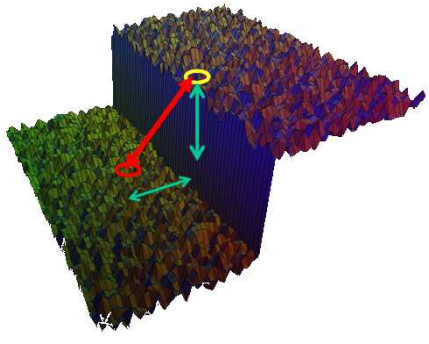
Chapter 3

Image processing

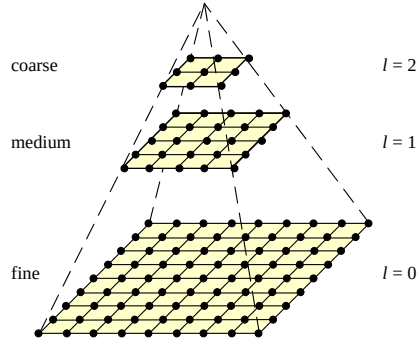


(a)

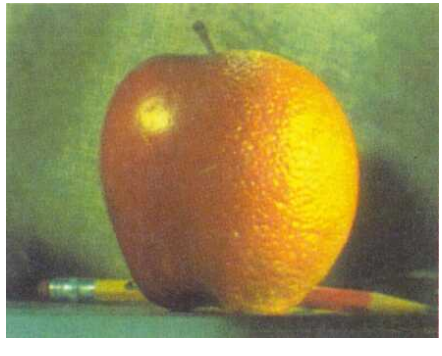
(b)



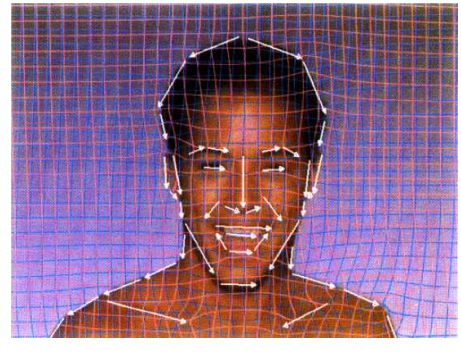
(c)



(d)



(e)



(f)

Figure 3.1 *Some common image processing operations: (a) partial histogram equalization; (b) orientation map computed from the second-order steerable filter (Freeman 1992) © 1992 IEEE; (c) bilateral filter (Durand and Dorsey 2002) © 2002 ACM; (d) image pyramid; (e) Laplacian pyramid blending (Burt and Adelson 1983b) © 1983 ACM; (f) line-based image warping (Beier and Neely 1992) © 1992 ACM.*

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 algorithms, 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, reducing image noise, increasing sharpness, or straightening the image by rotating it. Additional examples include image warping and image blending, which are often used for visual effects (Figures 3.1 and Section 3.6.3). 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 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, including Gomes and Velho (1997), Jähne (1997), Pratt (2007), Burger and Burge (2009), and Gonzalez and Woods (2017).

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).

While this chapter covers *classical* image processing techniques that consist mostly of linear and non-linear filtering operations, the next two chapters introduce energy-based and Bayesian graphical models, i.e., *Markov random fields* (Chapter 4), and then deep convolutional networks (Chapter 5), both of which are now widely used in image processing applications.

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

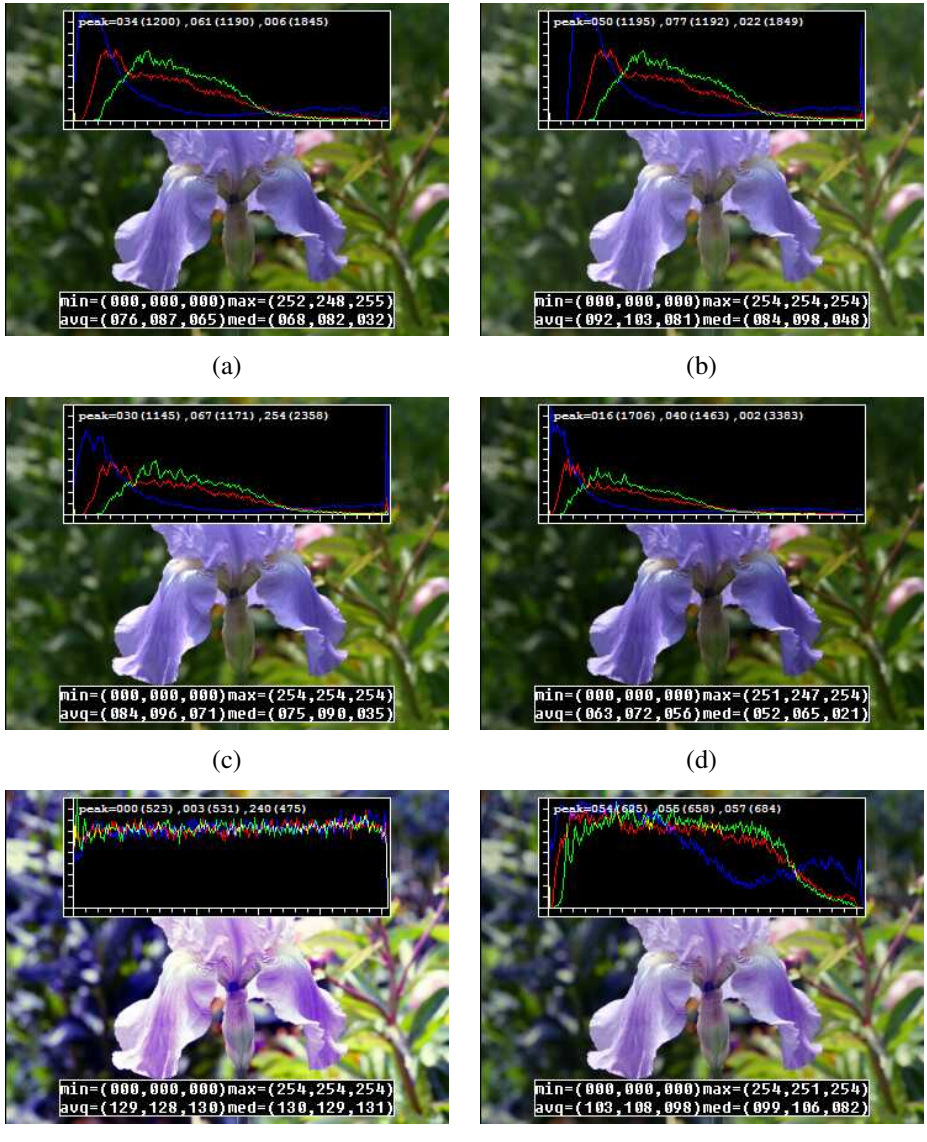
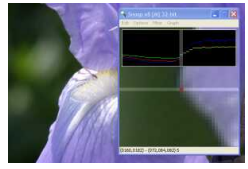


Figure 3.2 *Some local image processing operations: (a) original image along with its three color (per-channel) histograms; (b) brightness increased (additive offset, $b = 16$); (c) contrast increased (multiplicative gain, $a = 1.1$); (d) gamma (partially) linearized ($\gamma = 1.2$); (e) full histogram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

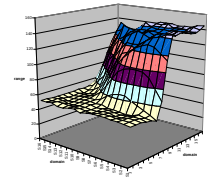
(a)



(b)

45	60	98	127	132	133	137	133
46	65	98	123	126	128	131	133
47	65	96	115	119	123	135	137
47	63	91	107	113	122	138	134
50	59	80	97	110	123	133	134
49	53	68	83	97	113	128	133
50	50	58	70	84	102	116	126
50	50	52	58	69	86	101	120

(c)



(d)

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.*

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).[1]

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.

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

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

where $\mathbf{x}$ is in the D-dimensional (usually $D = 2$ for images) *domain* of the input and output functions f and g, which 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

$$g(i, j) = h(f(i, j)). \tag{3.2}$$

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).

[1] In convolutional neural networks (Section 5.4), such operations are sometimes called 1×1 convolutions.

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

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

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).[2] The bias and gain parameters can also be spatially varying,

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

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, as it obeys the *superposition principle*,

$$h(f_0 + f_1) = h(f_0) + h(f_1). \tag{3.5}$$

(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.

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

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

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

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

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

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

(a) (b) (c) (d)

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

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 (2.105) or even simpler color ratios (2.117) 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. Figures 2.33f–h show 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 2.8 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 3.5 *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:

$$C = (1 - \alpha)B + \alpha F. \tag{3.8}$$

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 *et al.* 2001), the pure un-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

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.*

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 3.3).

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 9.4.2).

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 *et al.* (1999) review *difference matting*. Since then, there has been a lot of activity in computational photography relating to *natural image matting* (Ruzon and Tomasi 2000; Chuang, Curless *et al.* 2001; Wang and Cohen 2009; Xu, Price *et al.* 2017), which attempts to extract the mattes from a single natural image (Figure 3.4a) or from extended video sequences (Chuang, Agarwala *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

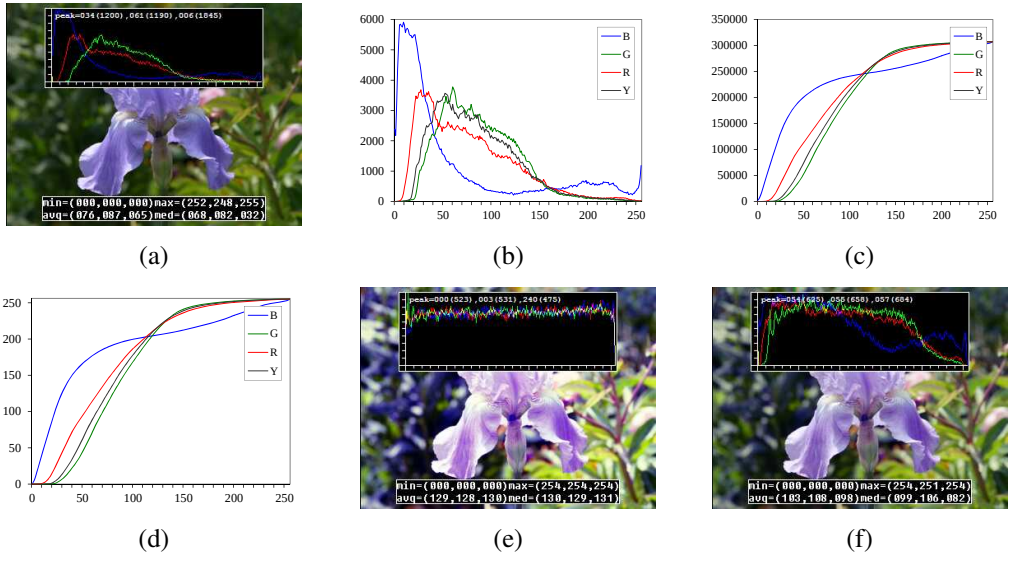


Figure 3.7 *Histogram analysis and equalization: (a) original image; (b) color channel and intensity (luminance) histograms; (c) cumulative distribution functions; (d) equalization (transfer) functions; (e) full histogram equalization; (f) partial histogram equalization.*

towards middle gray, and expand the *range* so that it more closely fills the displayable values (Kopf, Uyttendaele *et al.* 2007).

How can we visualize the set of lightness values in an image to test some of these heuristics? The answer is to plot the *histogram* of the individual color channels and luminance values, as shown in Figure 3.7b.[3] 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*

[3]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.

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 $c(I)$,

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

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 the pixel should take. When working with eight-bit pixel values, the I and c axes are rescaled from $[0, 255]$.

Figure 3.7e 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 3.7f, 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 3.7 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)

(a) (b) (c)

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

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-limited (gain-limited) version is known as CLAHE (Pizer, Amburn *et al.* 1987).[4] 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 $\{f_{00}, \ldots, f_{11}\}$, 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 3.8). 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.,

$$h_{k,l}(I(i, j)) \mathrel{+}= w(i, j, k, l), \quad (3.11)$$

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 7.1.3) and vocabulary trees (Section 7.1.4).

[4] The CLAHE algorithm is part of OpenCV.

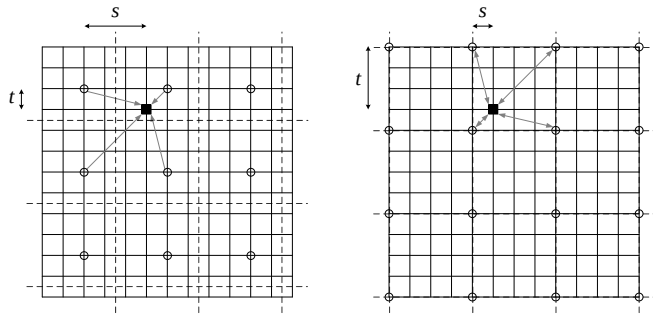


Figure 3.9 *Local histogram interpolation using relative (s, t) coordinates: (a) block-based histograms, with block centers shown as circles; (b) corner-based "spline" histograms. Pixels are located on grid intersections. The black square pixel's transfer function is interpolated from the four adjacent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

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 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, 3.6, and 3.7 have you implement some of these operations, to become familiar with basic image processing operators. More sophisticated techniques for tonal adjustment (Bae, Paris, and Durand 2006; Reinhard, Heidrich *et al.* 2010) 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 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 fixed weighted combinations of pixels in small neighborhoods. In Section 3.3,

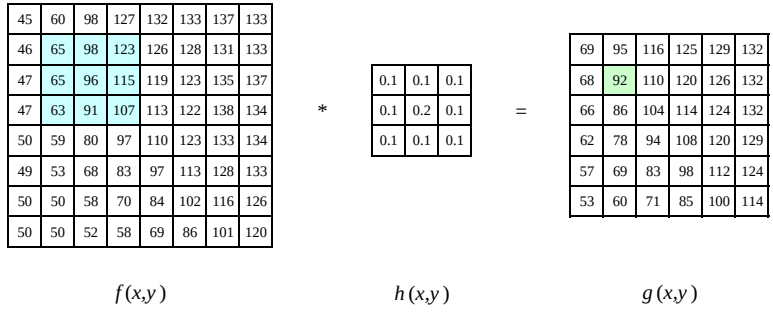


Figure 3.10 *Neighborhood filtering (convolution): The image on the left is convolved with the filter in the middle to yield the image on the right. The light blue pixels indicate the source neighborhood for the light green destination pixel.*

we look at non-linear operators such as morphological filters and distance transforms.

The most widely used type of neighborhood operator is a *linear filter*, where an output pixel's value is a weighted sum of pixel values within a small neighborhood $\mathcal{N}$ (Figure 3.10),

$$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l). \tag{3.12}$$

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

$$g = f \otimes h. \tag{3.13}$$

A common variant on this formula is

$$g(i,j) = \sum_{k,l} f(i-k, j-l)h(k,l) = \sum_{k,l} f(k,l)h(i-k, j-l), \tag{3.14}$$

where the sign of the offsets in f has been reversed, This is called the *convolution* operator,

$$g = f * h, \tag{3.15}$$

and h is then called the *impulse response function*.[5] 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.)

[5]The continuous version of convolution can be written as $g(\mathbf{x}) = \int f(\mathbf{x}-\mathbf{u})h(\mathbf{u})d\mathbf{u}$.

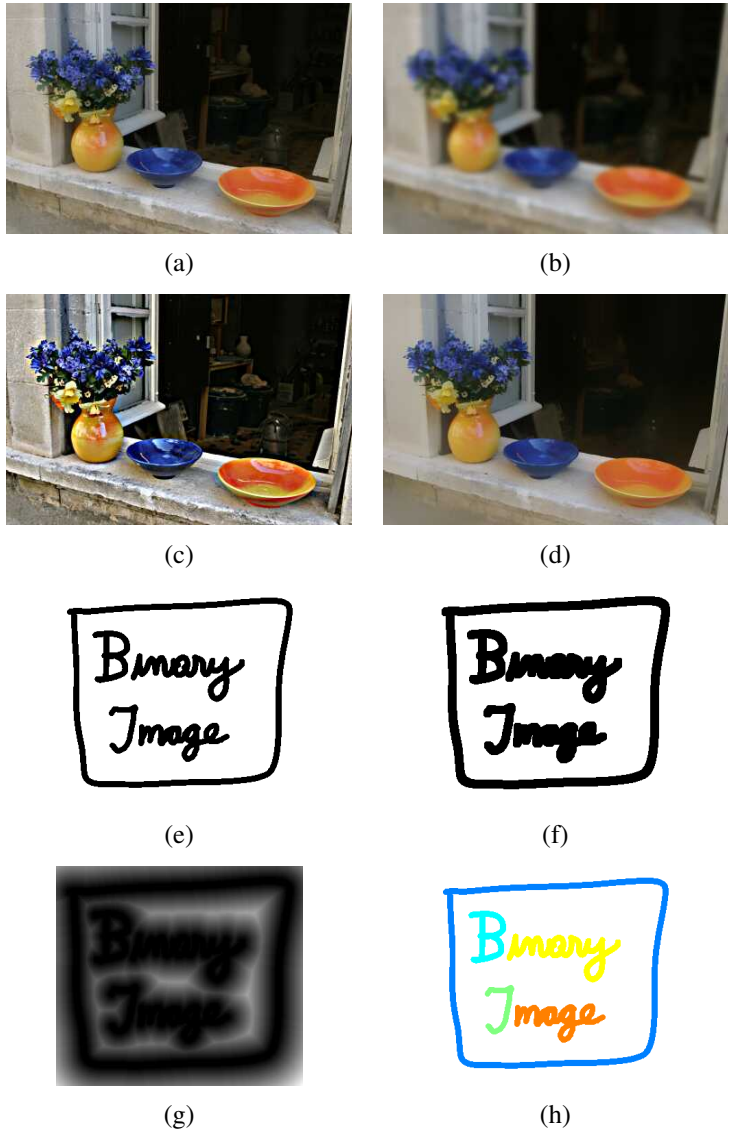


Figure 3.11 *Some neighborhood operations: (a) original image; (b) blurred; (c) sharpened; (d) smoothed with edge-preserving filter; (e) binary image; (f) dilated; (g) distance transform; (h) connected components. For the dilation and connected components, black (ink) pixels are assumed to be active, i.e., to have a value of 1 in Equations (3.44–3.48).*

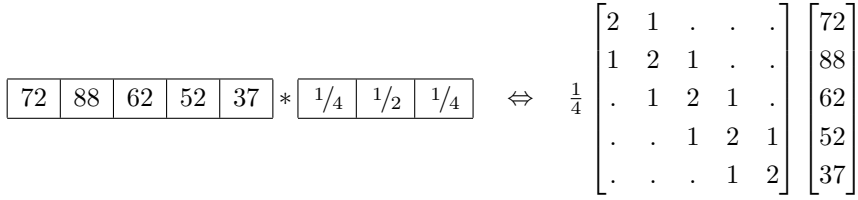


Figure 3.12 *One-dimensional signal convolution as a sparse matrix-vector multiplication,* $\mathbf{g} = \mathbf{H}\mathbf{f}$.

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),

$$h \circ (f_0 + f_1) = h \circ f_0 + h \circ f_1, \tag{3.16}$$

and the *shift invariance* principle,

$$g(i, j) = f(i + k, j + l) \quad \Leftrightarrow \quad (h \circ g)(i, j) = (h \circ f)(i + k, j + l), \tag{3.17}$$

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.,

$$g(i, j) = \sum_{k,l} f(i - k, j - l)h(k, l; i, j), \tag{3.18}$$

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 multiplication, if we first convert the two-dimensional images $f(i, j)$ and $g(i, j)$ into raster-ordered vectors $\mathbf{f}$ and $\mathbf{g}$,

$$\mathbf{g} = \mathbf{H}\mathbf{f}, \tag{3.19}$$

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.

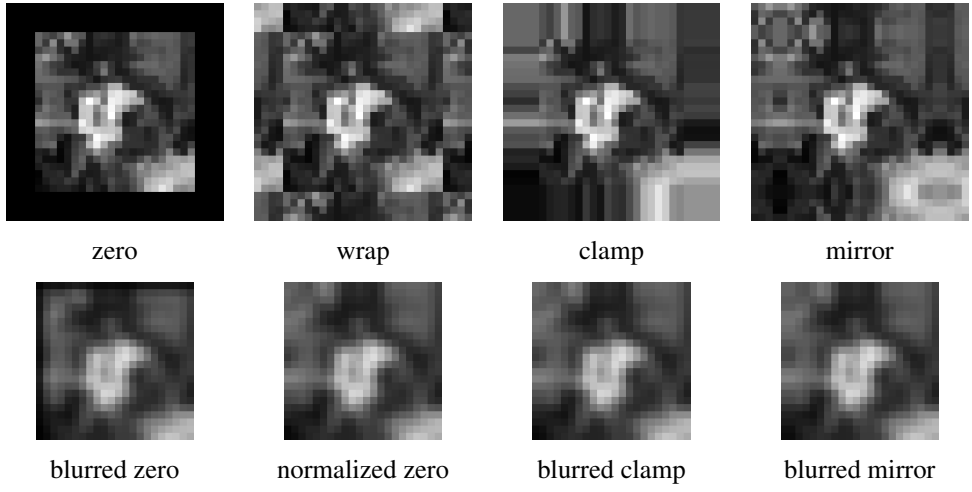


Figure 3.13 *Border padding (top row) and the results of blurring the padded image (bottom row). The normalized zero image is the result of dividing (normalizing) the blurred zero-padded RGBA image by its corresponding soft alpha value.*

Padding (border effects)

The astute reader will notice that the correlation shown in Figure 3.10 produces a result that is smaller than the original image, which may not be desirable in many applications.[6] This is because the neighborhoods of typical correlation and convolution operations extend beyond the image boundaries near the edges, and so the filtered images suffer from *boundary effects*

To deal with this, a number of different *padding* or extension modes have been developed for neighborhood operations (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;

[6] Note, however, that early convolutional networks such as LeNet (LeCun, Bottou *et al.* 1998) adopted this structure.

- *extend*: extend the signal by subtracting the mirrored version of the signal from the edge pixel value.

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 these modes are left to the reader (Exercise 3.9).

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,

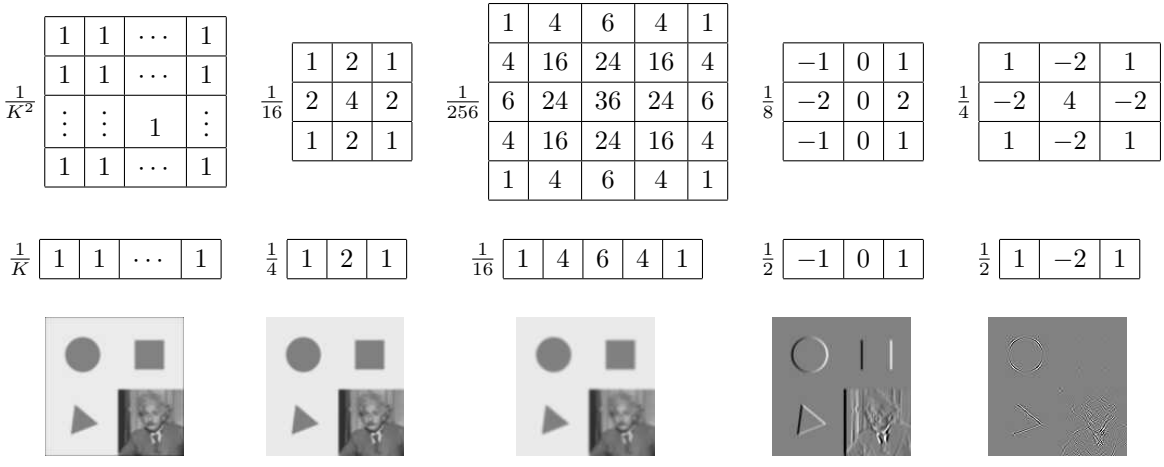
$$\mathbf{K} = \mathbf{v}\mathbf{h}^T \tag{3.20}$$

(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),

$$\mathbf{K} = \sum_i \sigma_i \mathbf{u}_i \mathbf{v}_i^T \tag{3.21}$$

(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



(a) box, $K = 5$ (b) bilinear (c) "Gaussian" (d) Sobel (e) corner

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× and 4×, respectively, and added to a gray offset before display.*

kernels (Perona 1995). For example, the Laplacian of Gaussian kernel (3.26 and 7.23) can be implemented as the sum of two separable filters (7.24) (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).

Convolving 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 3.11).

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 of smoothing to remove noise later (see Sections 3.3.1, 3.4, and as well as Chapters 4 and 5).

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,

$$g_{\text{sharp}} = f + \gamma(f - h_{\text{blur}} * f). \tag{3.22}$$

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

$$g_{\text{unsharp}} = f(1 - \gamma h_{\text{blur}} * f). \tag{3.23}$$

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 7.2) and interest point detection (Section 7.1) 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 vertical 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 7.1.

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,

$$G(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{3.24}$$

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,

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}, \tag{3.25}$$

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,

$$\nabla^2 G(x, y; \sigma) = \left(\frac{x^2 + y^2}{\sigma^4} - \frac{2}{\sigma^2} \right) G(x, y; \sigma), \tag{3.26}$$

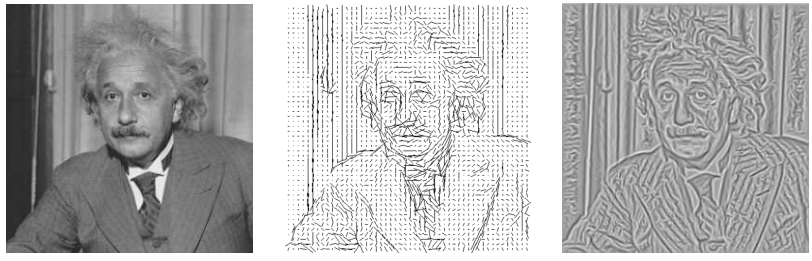
which has certain nice *scale-space properties* (Witkin 1983; Witkin, Terzopoulos, and Kass 1986). The five-point Laplacian is just a compact approximation to this more sophisticated filter.

Likewise, the Sobel operator is a simple approximation to a *directional* or *oriented* filter, which can obtained by smoothing with a Gaussian (or some other filter) and then taking a *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 $\hat{\mathbf{u}} = (\cos\theta, \sin\theta)$,

$$\hat{\mathbf{u}} \cdot \nabla (G * f) = \nabla_{\hat{\mathbf{u}}} (G * f) = (\nabla_{\hat{\mathbf{u}}} G) * f. \tag{3.27}$$

The smoothed directional derivative filter,

$$G_{\hat{\mathbf{u}}} = u G_x + v G_y = u \frac{\partial G}{\partial x} + v \frac{\partial G}{\partial y}, \tag{3.28}$$



(a) (b) (c)

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.*

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.

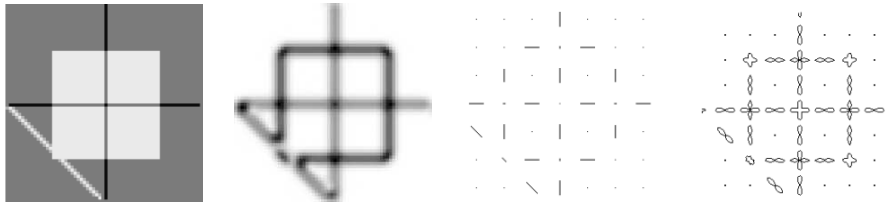
How about steering a directional second derivative filter $\nabla_{\hat{\mathbf{u}}} \cdot \nabla_{\hat{\mathbf{u}}} G$, which is the result of taking a (smoothed) directional derivative and then taking the directional derivative again? For example, G_{xx} is the second directional derivative in the 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,

$$G_{\hat{\mathbf{u}}\hat{\mathbf{u}}} = u^2 G_{xx} + 2uv G_{xy} + v^2 G_{yy}. \tag{3.29}$$

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 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



(a) (b) (c) (d)

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).*

higher filters, as described in (Freeman and Adelson 1991).

Steerable filters are often used to construct both feature descriptors (Section 7.1.3) and edge detectors (Section 7.2). 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 3.13 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,

$$s(i,j) = \sum_{k=0}^{i} \sum_{l=0}^{j} f(k,l). \tag{3.30}$$

This can be efficiently computed using a recursive (raster-scan) algorithm,

$$s(i,j) = s(i-1,j) + s(i,j-1) - s(i-1,j-1) + f(i,j). \tag{3.31}$$

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,

$$S(i_0 \ldots i_1, j_0 \ldots j_1) = s(i_1, j_1) - s(i_1, j_0 - 1) - s(i_0 - 1, j_1) + s(i_0 - 1, j_0 - 1). \tag{3.32}$$

3	2	7	2	3
1	5	1	3	4
5	1	**3**	5	1
4	3	2	1	6
2	4	1	4	8

(a) S = 24

3	5	12	14	17
4	*11*	**19**	24	31
9	**17**	28	38	46
13	24	37	48	62
15	30	44	59	81

(b) s = 28

3	5	12	*14*	17
4	11	19	24	31
9	17	28	38	46
13	24	37	**48**	62
15	30	44	59	81

(c) S = 24

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.*

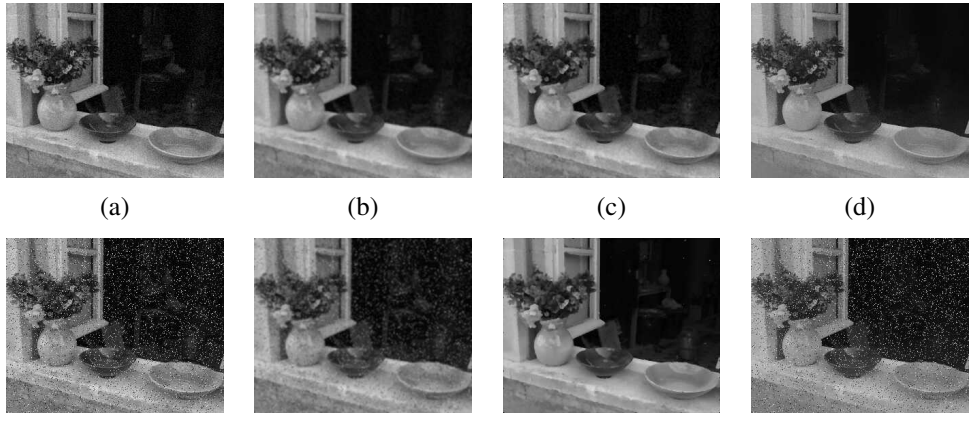
A potential disadvantage of summed area tables is that they require $\log M + \log N$ extra bits in the accumulation image compared to the original image, where M and N are the image width and height. Extensions of summed area tables can also be used to approximate other convolution kernels (Wolberg (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 *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 12.4). In practice, separable moving average filters are usually preferred (Kanade, Yoshida *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 quan-



(a) (b) (c) (d)

(e) (f) (g) (h)

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.*

tities that involve large area interactions, such as two-dimensional distance functions (Section 3.3.3) and connected components (Section 3.3.3).

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.

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).

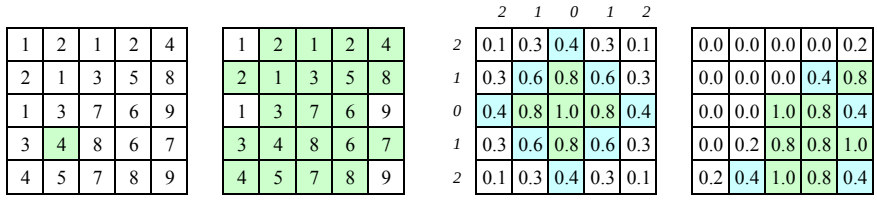


Figure 3.19 *Median and bilateral filtering: (a) median pixel (green); (b) selected α-trimmed mean pixels; (c) domain filter (numbers along edge are pixel distances); (d) range filter.*

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 developed (Tomasi and Manduchi 1998; Bovik 2000, Section 3.2), as well as a constant time algorithm that is independent of window size (Perreault and Hébert 2007). 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 3.18g).

One downside of the median filter, in addition to its moderate computational cost, is that because 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 *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

$$\sum_{k,l} w(k,l)|f(i+k,j+l) - g(i,j)|^p, \tag{3.33}$$

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 (Haralick and Shapiro 1992, Section 7.2.6; Bovik 2000, Section 3.2). 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 *et al.* 1986).

Non-linear smoothing has another, perhaps even more important property, especially as 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 well at smoothing away from 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.

3.3.2 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), although it had been proposed earlier by Aurich and Weule (1995) and Smith and Brady (1997). Paris, Kornprobst *et al.* (2008) provide a nice review of work in this area as well as myriad applications in computer vision, graphics, and computational photography.

In the bilateral filter, the output pixel value depends on a weighted combination of neigh-

boring pixel values

$$\mathbf{g}(i,j) = \frac{\sum_{k,l} \mathbf{f}(k,l) w(i,j,k,l)}{\sum_{k,l} w(i,j,k,l)}. \tag{3.34}$$

The weighting coefficient $w(i,j,k,l)$ depends on the product of a *domain kernel*, (Figure 3.19c),

$$d(i,j,k,l) = \exp\left(-\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2}\right), \tag{3.35}$$

and a data-dependent *range kernel* (Figure 3.19d),

$$r(i,j,k,l) = \exp\left(-\frac{\|\mathbf{f}(i,j) - \mathbf{f}(k,l)\|^2}{2\sigma_r^2}\right). \tag{3.36}$$

When multiplied together, these yield the data-dependent *bilateral weight function*

$$w(i,j,k,l) = \exp\left(-\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2} - \frac{\|\mathbf{f}(i,j) - \mathbf{f}(k,l)\|^2}{2\sigma_r^2}\right). \tag{3.37}$$

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 for color images, 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.[7]

Since bilateral filtering is quite slow compared to regular separable filtering, a number of acceleration techniques have been developed, as discussed in Durand and Dorsey (2002), Paris and Durand (2009), Chen, Paris, and Durand (2007), and Paris, Kornprobst *et al.* (2008). In particular, the *bilateral grid* (Chen, Paris, and Durand 2007), which subsamples the higher-dimensional color/position space on a uniform grid, continues to be widely used, including the application of the *bilateral solver* (Section 4.2.3 and Barron and Poole (2016)). An even faster implementation of bilateral filtering can be obtained using the *permutohedral lattice* approach developed by Adams, Baek, and Davis (2010).

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.

[7] 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).

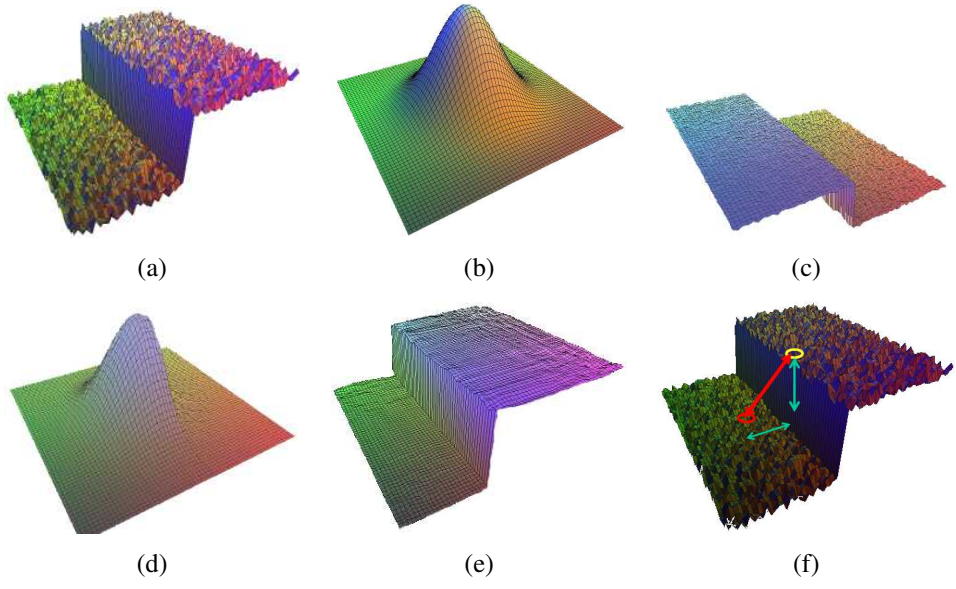


Figure 3.20 *Bilateral filtering (Durand and Dorsey 2002) © 2002 ACM: (a) noisy step edge input; (b) domain filter (Gaussian); (c) range filter (similarity to center pixel value); (d) bilateral filter; (e) filtered step edge output; (f) 3D distance between pixels.*

Consider, for example, using only the four nearest neighbors, i.e., restricting $|k-i|+|l-j| \leq 1$ in (3.34). Observe that

$$d(i,j,k,l) = \exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2}\right) \tag{3.38}$$

$$= \begin{cases} 1, & |k-i|+|l-j| = 0, \\ e^{-1/2\sigma_d^2}, & |k-i|+|l-j| = 1. \end{cases} \tag{3.39}$$

We can thus re-write (3.34) as

$$f^{(t+1)}(i,j) = \frac{f^{(t)}(i,j) + \eta \sum_{k,l} f^{(t)}(k,l) r(i,j,k,l)}{1 + \eta \sum_{k,l} r(i,j,k,l)} \tag{3.40}$$

$$= f^{(t)}(i,j) + \frac{\eta}{1+\eta R} \sum_{k,l} r(i,j,k,l)[f^{(t)}(k,l) - f^{(t)}(i,j)],$$

where $R = \sum_{(k,l)} r(i,j,k,l)$, (k,l) are the $\mathcal{N}_4$ (nearest four) 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).[8] Since its original introduction, anisotropic diffusion has been extended and applied to a wide range of problems (Nielsen, Florack, and Deriche 1997; Black, Sapiro *et al.* 1998; Weickert, ter Haar Romeny, and Viergever 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 $r'(x) \to 0$ as $x \to \infty$. Black, Sapiro *et al.* (1998) show how anisotropic diffusion is equivalent to minimizing a robust penalty function on the image gradients, which we discuss in Sections 4.2 and 4.3. 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 4.2 and 4.3 and by Scharr, Black, and Haussecker (2003), which introduce such a bias in a principled manner.

Guided image filtering

While so far we have discussed techniques for filtering an image to obtain an improved version, e.g., one with less noise or sharper edges, it is also possible to use a different *guide* image to adaptively filter a noisy input (Eisemann and Durand 2004; Petschnigg, Agrawala *et al.* 2004; He, Sun, and Tang 2013). An example of this is using a flash image, which has strong edges but poor color, to adaptively filter a low-light non-flash color image, which has large amounts of noise, as described in Section 10.2.2. In their papers, where they apply the range filter (3.36) to a different guide image $\mathbf{h}()$, Eisemann and Durand (2004) call their approach a *cross-bilateral filter*, while Petschnigg, Agrawala *et al.* (2004) call it *joint bilateral filtering*.

He, Sun, and Tang (2013) point out that these papers are just two examples of the more general concept of *guided image filtering*, where the guide image $\mathbf{h}()$ is used to compute the

[8] The $1/(1 + \eta R)$ factor is not present in anisotropic diffusion but becomes negligible as $\eta \to 0$.

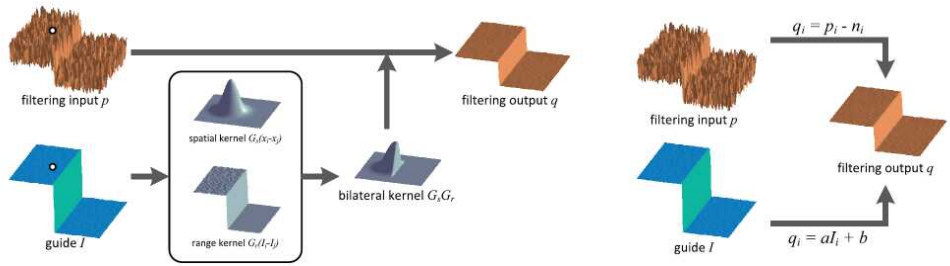


Figure 3.21 *Guided image filtering (He, Sun, and Tang 2013) © 2013 IEEE. Unlike joint bilateral filtering, shown on the left, which computes a per pixel weight mask from the guide image (shown as I in the figure, but $\mathbf{h}$ in the text), the guided image filter models the output value (shown as q_i in the figure, but denoted as $\mathbf{g}(i,j)$ in the text) as a local affine transformation of the guide pixels.*

locally adapted inter-pixel weights $w(i,j,k,l)$, i.e.,

$$\mathbf{g}(i,j) = \sum_{k,l} w(\mathbf{h}; i,j,k,l)\mathbf{f}(k,l). \tag{3.41}$$

In their paper, the authors suggest modeling the relationship between the guide and input images using a local affine transformation,

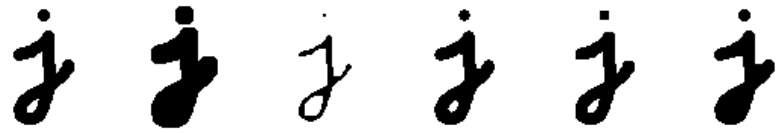
$$\mathbf{g}(i,j) = \mathbf{A}_{k,l}\mathbf{h}(i,j) + \mathbf{b}_{k,l}, \tag{3.42}$$

where the estimates for $\mathbf{A}_{k,l}$ and $\mathbf{b}_{k,l}$ are obtained from a regularized least squares fit over a square neighborhood centered around pixel (k,l), i.e., minimizing

$$\sum_{(i,j)\in\mathcal{N}_{k,l}} \|\mathbf{A}_{k,l}\mathbf{h}(i,j) + \mathbf{b}_{k,l} - \mathbf{f}(i,j)\|^2 + \lambda\|\mathbf{A}\|^2. \tag{3.43}$$

These kinds of regularized least squares problems are called *ridge regression* (Section 4.1). The concept behind this algorithm is illustrated in Figure 3.21.

Instead of just taking the predicted value of the filtered pixel $\mathbf{g}(i,j)$ from the window centered on that pixel, an average across all windows that cover the pixel is used. The resulting algorithm (He, Sun, and Tang 2013, Algorithm 1) consists of a series of local mean image and image moment filters, a per-pixel linear system solve (which reduces to a division if the guide image is scalar), and another set of filtering steps. The authors describe how this fast and simple process has been applied to a wide variety of computer vision problems, including image matting (Section 10.4.3), high dynamic range image tone mapping (Section 10.2.1), stereo matching (Hosni, Rhemann *et al.* 2013), and image denoising.



(a) (b) (c) (d) (e) (f)

Figure 3.22 *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, as it is not wide enough.*

3.3.3 Binary image processing

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,

$$\theta(f,t) = \begin{cases} 1 & \text{if } f \geq t, \\ 0 & \text{else,} \end{cases} \tag{3.44}$$

e.g., converting a scanned grayscale document into a binary image for further processing, such as *optical character recognition*.

Morphology

The most common binary image operations are called *morphological operations*, because 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 3.22 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

$$c = f \otimes s \tag{3.45}$$

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**: $\text{dilate}(f, s) = \theta(c, 1)$;
- **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 3.22, 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).

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 *et al.* 1995; Felzenszwalb and Huttenlocher 2012; Fabbri, Costa *et al.* 2008). It has many applications, including level sets (Section 7.3.2), fast *chamfer matching* (binary image alignment) (Huttenlocher, Klanderman, and Rucklidge 1993), feathering in image stitching and blending (Section 8.4.2), and nearest point alignment (Section 13.2.1).

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

$$d_1(k, l) = |k| + |l| \tag{3.46}$$

and the *Euclidean* distance

$$d_2(k, l) = \sqrt{k^2 + l^2}. \tag{3.47}$$

The distance transform is then defined as

$$D(i, j) = \min_{k,l:b(k,l)=0} d(i - k, j - l), \tag{3.48}$$

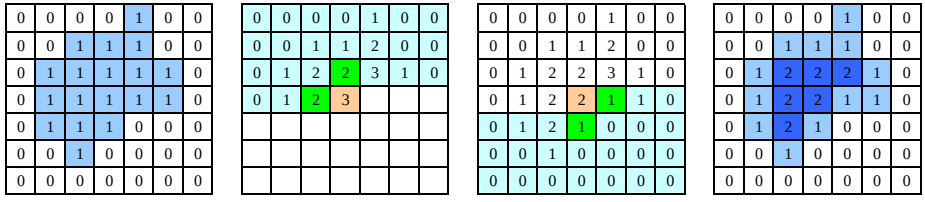


Figure 3.23 *City block distance transform: (a) original binary image; (b) top to bottom (forward) raster sweep: green values are used to compute the orange value; (c) bottom to top (backward) raster sweep: green values are merged with old orange value; (d) final distance transform.*

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 3.23. 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 3.23).

Efficiently computing the Euclidean distance transform is more complicated (Danielsson 1980; Borgefors 1986). 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.

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 (Lavallée and Szeliski 1995; Szeliski and Lavallée 1996; Frisken, Perry *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 Rucklidge 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 7.3.2), where they are called *characteristic functions*.

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. Pixels are said to be $\mathcal{N}_4$ adjacent if they are immediately horizontally or vertically adjacent, and $\mathcal{N}_8$ if they can also be diagonally adjacent. Both variants of connected components are widely 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. Over the years, a wide variety of efficient algorithms have been developed to find such components, including the ones described in Haralick and Shapiro (1992, Section 2.3) and He, Ren *et al.* (2017). Such algorithms are usually included in image processing libraries such as OpenCV.

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,

$$\mathbf{M} = \sum_{(x,y)\in\mathcal{R}} \begin{bmatrix} x-\overline{x} \\ y-\overline{y} \end{bmatrix} \begin{bmatrix} x-\overline{x} & y-\overline{y} \end{bmatrix}, \tag{3.49}$$

  from which the major and minor axis orientation and lengths can be computed using eigenvalue analysis.

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 (i.e., 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), and 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

$$s(x) = \sin(2\pi f x + \phi_i) = \sin(\omega x + \phi_i) \tag{3.50}$$

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).

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,

$$o(x) = h(x) * s(x) = A \sin(\omega x + \phi_o), \tag{3.51}$$

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.[9]

[9] If h is a general (non-linear) transform, additional *harmonic* frequencies are introduced. This was traditionally

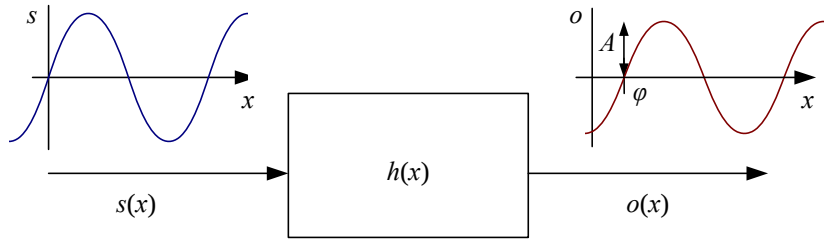


Figure 3.24 *The Fourier Transform as the response of a filter* $h(x)$ *to an input sinusoid* $s(x) = e^{j\omega x}$ *yielding an output sinusoid* $o(x) = h(x) * s(x) = Ae^{j(\omega x+\phi)}$.

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

$$s(x) = e^{j\omega x} = \cos\omega x + j\sin\omega x. \tag{3.52}$$

In that case, we can simply write,

$$o(x) = h(x) * s(x) = Ae^{j(\omega x+\phi)}. \tag{3.53}$$

The *Fourier transform* is simply a tabulation of the magnitude and phase response at each frequency,

$$H(\omega) = \mathcal{F}\{h(x)\} = Ae^{j\phi}, \tag{3.54}$$

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

$$h(x) \overset{\mathcal{F}}{\leftrightarrow} H(\omega). \tag{3.55}$$

Unfortunately, (3.54) 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,

$$H(\omega) = \int_{-\infty}^{\infty} h(x)e^{-j\omega x}dx, \tag{3.56}$$

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".

and in the discrete domain,

$$H(k) = \frac{1}{N}\sum_{x=0}^{N-1} h(x)e^{-j\frac{2\pi kx}{N}}, \tag{3.57}$$

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 (3.57) is known as the *Discrete Fourier Transform* (DFT). Note that while (3.57) 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.

The Fourier transform comes with a set of extremely useful properties relating original signals and their Fourier transforms, including superposition, shifting, reversal, convolution, correlation, multiplication, differentiation, domain scaling (stretching), and energy preservation (Parseval's Theorem). To make room for all of the new material in this second edition, I have removed all of these details, as well as a discussion of commonly used Fourier transform pairs. Interested readers should refer to (Szeliski 2010, Section 3.1, Tables 3.1–3.3) or standard textbooks on signal processing and Fourier transforms (Bracewell 1986; Glassner 1995; Oppenheim and Schafer 1996; Oppenheim, Schafer, and Buck 1999).

We can also compute the Fourier transforms for the small discrete kernels shown in Figure 3.14 (see Table 3.1). 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

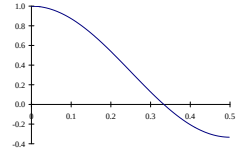
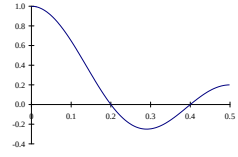
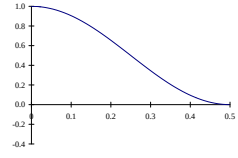
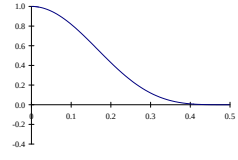
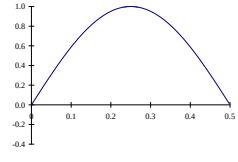
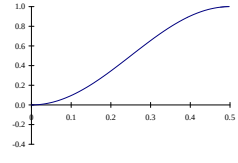
Name	Kernel	Transform	Plot
box-3	$\frac{1}{3}\boxed{1}\boxed{1}\boxed{1}$	$\frac{1}{3}(1+2\cos\omega)$	
box-5	$\frac{1}{5}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}$	$\frac{1}{5}(1+2\cos\omega+2\cos 2\omega)$	
linear	$\frac{1}{4}\boxed{1}\boxed{2}\boxed{1}$	$\frac{1}{2}(1+\cos\omega)$	
binomial	$\frac{1}{16}\boxed{1}\boxed{4}\boxed{6}\boxed{4}\boxed{1}$	$\frac{1}{4}(1+\cos\omega)^2$	
Sobel	$\frac{1}{2}\boxed{-1}\boxed{0}\boxed{1}$	$\sin\omega$	
corner	$\frac{1}{2}\boxed{-1}\boxed{2}\boxed{-1}$	$\frac{1}{2}(1-\cos\omega)$	

Table 3.1 *Fourier transforms of the separable kernels shown in Figure 3.14, obtained by evaluating* $\sum_k h(k)e^{-jk\omega}$.

columns. We look at additional examples of small kernel Fourier transforms in Section 3.5.2, where we study better kernels for prefiltering before decimation (size reduction).

3.4.1 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),

$$s(x, y) = \sin(\omega_x x + \omega_y y). \tag{3.58}$$

The corresponding two-dimensional Fourier transforms are then

$$H(\omega_x, \omega_y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(x, y) e^{-j(\omega_x x + \omega_y y)} dx\, dy, \tag{3.59}$$

and in the discrete domain,

$$H(k_x, k_y) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h(x, y) e^{-j2\pi(k_x x/M + k_y y/N)} \tag{3.60}$$

where M and N are the width and height of the image.

All of the Fourier transform properties from 1D 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.

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.,

$$\left\langle [S(\omega_x, \omega_y)]^2 \right\rangle = P_s(\omega_x, \omega_y), \tag{3.61}$$

where the angle brackets $\langle \cdot \rangle$ denote the expected (mean) value of a random variable.[10] 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 Gaussian of variance $P_s(\omega_x, \omega_y)$ and then take its inverse FFT.

[10] The notation $E[\cdot]$ is also commonly used.

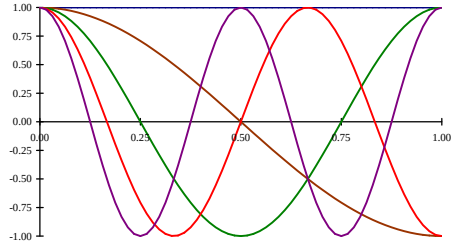


Figure 3.25 *Discrete cosine transform (DCT) basis functions: The first DC (i.e., constant) basis is the horizontal blue line, the second is the brown half-cycle waveform, etc. These bases are widely used in image and video compression standards such as JPEG.*

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*.

The first edition of this book contains a derivation of the Wiener filter (Szeliski 2010, Section 3.4.3), but I've decided to remove this from the current edition, since it is almost never used in practice any more, having been replaced with better-performing non-linear filters.

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,

$$F(k) = \sum_{i=0}^{N-1} \cos\left(\frac{\pi}{N}(i+\frac{1}{2})k\right) f(i), \tag{3.62}$$

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 3.25. 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–Loève 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 5.2.3. The KL-

transform decorrelates 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,

$$F(k,l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \cos\left(\frac{\pi}{N}(i+\frac{1}{2})k\right) \cos\left(\frac{\pi}{N}(j+\frac{1}{2})l\right) f(i,j). \tag{3.63}$$

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 alternatives such as wavelet transforms (Simoncelli and Adelson 1990b; Taubman and Marcellin 2002), discussed in Section 3.5.4, and overlapped variants of the DCT (Malvar 1990, 1998, 2000), are used in the JPEG2000 and JPEG XR standards. 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 4.3 for ideas on how to remove blocking artifacts from compressed JPEG images.

3.4.2 *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 operations were performed using linear filtering (see Sections 3.2 and Section 3.4.1). 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 4.2), especially those using non-quadratic (robust) norms such as the L_1 norm (which is called *total variation*), are also often used. Most recently, deep neural networks have taken over the denoising community (Section 10.3). 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 (2.120), 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 *et al.* 2004; Wang, Bovik, and Simoncelli 2005) or FLIP image difference evaluator (Andersson, Nilsson *et al.* 2020). More recently,

people have started measuring similarity using neural "perceptual" similarity metrics (Johnson, Alahi, and Fei-Fei 2016; Dosovitskiy and Brox 2016; Zhang, Isola *et al.* 2018; Tariq, Tursun *et al.* 2020; Czolbe, Krause *et al.* 2020), which, unlike L_2 (PSNR) or L_1 metrics, which encourage smooth or flat average results, prefer images with similar amounts of texture (Cho, Joshi *et al.* 2012). When the clean image is not available, it is also possible to assess the quality of an image using *no-reference image quality assessment* (Mittal, Moorthy, and Bovik 2012; Talebi and Milanfar 2018).

Exercises 3.12, 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 6.3.1). 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 9.1.1). 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).

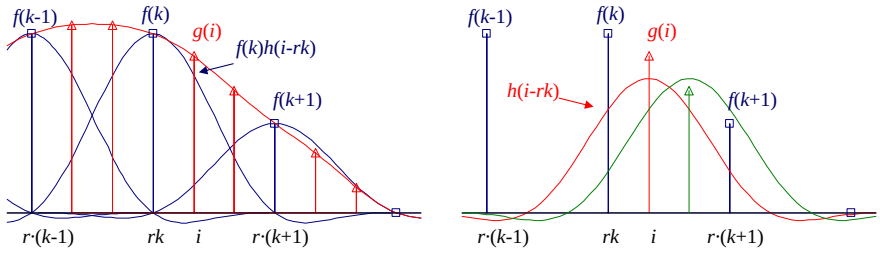


Figure 3.26 *Signal interpolation, $g(i) = \sum_k f(k)h(i - rk)$: (a) weighted summation of input values; (b) polyphase filter interpretation.*

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,

$$g(i, j) = \sum_{k,l} f(k, l)h(i - rk, j - rl). \tag{3.64}$$

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 3.26a 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 3.26b, 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.

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 Table 3.1 can be used after appropriate re-scaling.[11] The *linear* interpolator (corresponding to the tent kernel) produces interpolating piecewise linear curves, which result in unappealing *creases* when applied to images (Figure 3.27a). The cubic B-spline, whose discrete $1/2$-pixel sampling appears as the *binomial kernel* in Table 3.1, 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.

[11] The smoothing kernels in Table 3.1 have a unit area. To turn them into interpolating kernels, we simply scale

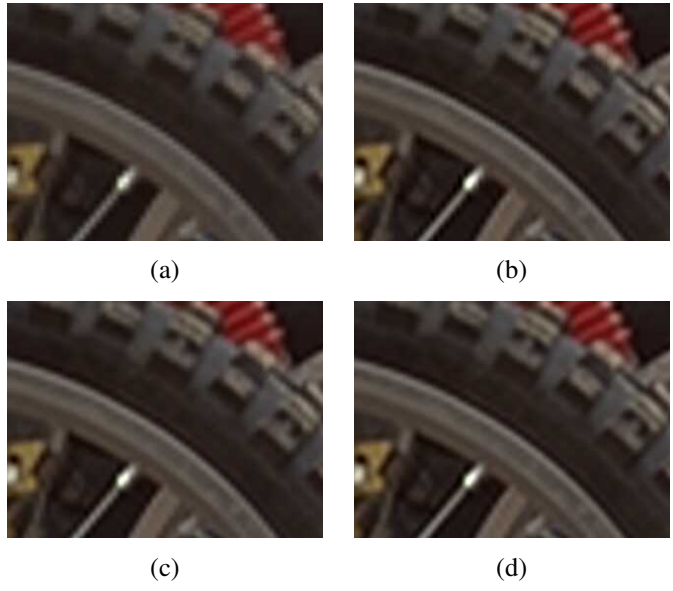


Figure 3.27 *Two-dimensional image interpolation: (a) bilinear; (b) bicubic ($a = -1$); (c) bicubic ($a = -0.5$); (d) windowed sinc (nine taps).*

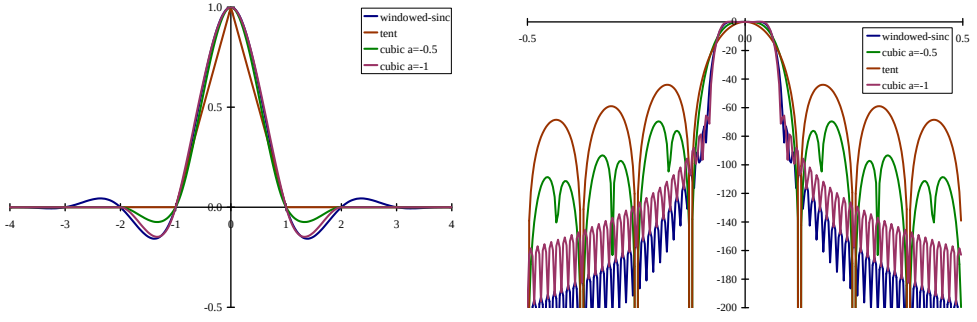
While most graphics cards use the bilinear kernel (optionally combined with a MIP-map—see Section 3.5.3), most photo editing packages use *bicubic* interpolation. The cubic interpolant is a C^1 (derivative-continuous) piecewise-cubic *spline* (the term "spline" is synonymous with "piecewise-polynomial")[12] whose equation is

$$h(x) = \begin{cases} 1 - (a+3)x^2 + (a+2)|x|^3 & \text{if} \quad |x| < 1 \\ a(|x|-1)(|x|-2)^2 & \text{if} \quad 1 \le |x| < 2 \\ 0 & \text{otherwise,} \end{cases} \tag{3.65}$$

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 3.28). 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 3.28 shows the $a = -1$

them up by the interpolation rate r.

[12] 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.



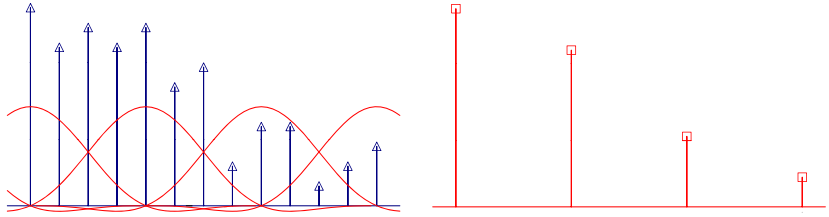
(a) (b)

Figure 3.28 *(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.*

and $a = -0.5$ cubic interpolating kernel along with their Fourier transforms; Figure 3.27b and c shows them being applied to two-dimensional interpolation.

Splines have long been used for function and data value interpolation because of the ability to precisely specify derivatives at control points and efficient *incremental* algorithms for their evaluation (Bartels, Beatty, and Barsky 1987; Farin 1992, 2002). 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), scattered data interpolation (Section 4.1), motion estimation (Section 9.2.2), and surface interpolation (Section 13.3). 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; Nehab and Hoppe 2014).

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 3.27d). For this reason, some photographers prefer to repeatedly interpolate images by a small fractional amount (this tends to decorrelate the original pixel grid with the final image). Additional possibilities include using the bilateral filter as an interpolator (Kopf, Cohen *et al.* 2007), using global optimization (Section 3.6) or



(a) (b)

Figure 3.29 *Signal decimation: (a) the original samples are (b) convolved with a low-pass filter before being downsampled.*

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.[13] To perform decimation, we first (conceptually) convolve the image with a low-pass filter (to avoid aliasing) and then keep every rth sample. In practice, we usually only evaluate the convolution at every rth sample,

$$g(i,j) = \sum_{k,l} f(k,l)h(ri-k, rj-l), \tag{3.66}$$

as shown in Figure 3.29. 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

$$g(i,j) = \frac{1}{r}\sum_{k,l} f(k,l)h(i-k/r, j-l/r) \tag{3.67}$$

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 3.1, 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

[13] 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.

desired. For high downsampling rates, the windowed sinc prefilter is a good choice (Figure 3.28). However, for small downsampling rates, e.g., $r = 2$, more careful filter design is required.

Table 3.2 shows a number of commonly used $r = 2$ downsampling filters, while Figure 3.30 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 (3.65); the $a = -1$ filter has a sharper fall-off than the $a = -0.5$ filter (Figure 3.30);
- a cosine-windowed sinc function;
- 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.

3.5.3 Multi-resolution representations

Now that we have described interpolation and decimation algorithms, we can build a complete image pyramid (Figure 3.31). As we mentioned before, pyramids can be used to accelerate

$\|n\|$	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 3.2 *Filter coefficients for 2 × decimation. These filters are of odd length, are symmetric, and are normalized to have unit DC gain (sum up to 1). See Figure 3.30 for their associated frequency responses.*

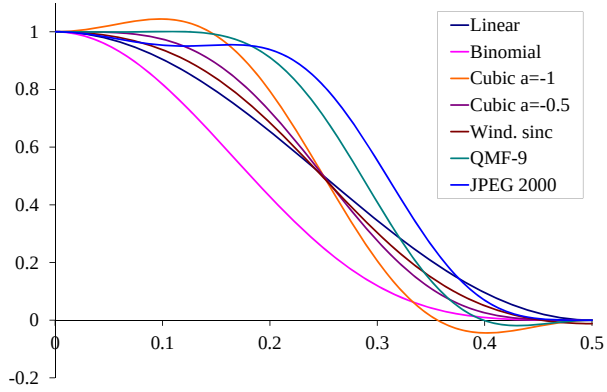


Figure 3.30 *Frequency response for some 2 × decimation filters. The cubic $a = -1$ filter has the sharpest fall-off but also a bit of ringing; the wavelet analysis filters (QMF-9 and JPEG 2000), while useful for compression, have more aliasing.*

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 discuss 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 (Figures 3.31 and 3.32). 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

$$\begin{array}{|c|c|c|c|c|}\hline c & b & a & b & c \\ \hline\end{array}, \tag{3.68}$$

with $b = 1/4$ and $c = 1/4 - a/2$. In practice, they and everyone else uses $a = 3/8$, which results in the familiar binomial kernel,

$$\frac{1}{16}\begin{array}{|c|c|c|c|c|}\hline 1 & 4 & 6 & 4 & 1 \\ \hline\end{array}, \tag{3.69}$$

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.[14]

[14] Then again, this is true for any smoothing kernel (Wells 1986).

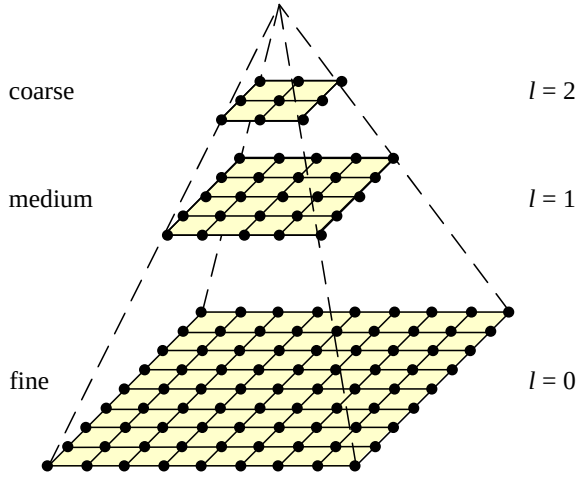


Figure 3.31 *A traditional image pyramid: each level has half the resolution (width and height), and hence a quarter of the pixels, of its parent level.*

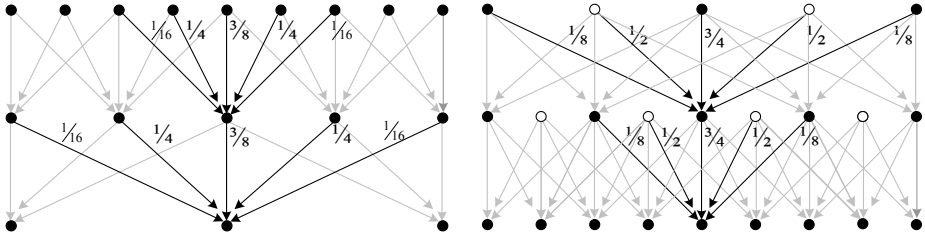


Figure 3.32 *The Gaussian pyramid shown as a signal processing diagram: The (a) analysis and (b) re-synthesis stages are shown as using similar computations. The white circles indicate zero values inserted by the $\uparrow 2$ upsampling operation. Notice how the reconstruction filter coefficients are 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.*

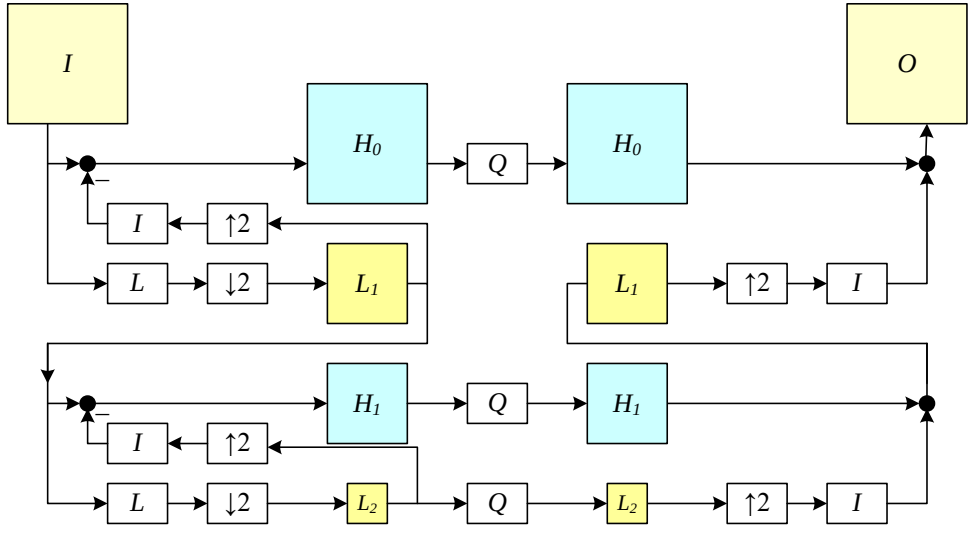


Figure 3.33 *The Laplacian pyramid. The yellow images form the* Gaussian *pyramid, which is obtained by successively low-pass filtering and downsampling the input image. The blue images, together with the smallest low-pass image, which is needed for reconstruction, form the* Laplacian *pyramid. Each band-pass (blue) image is computed by upsampling and interpolating the lower-resolution Gaussian pyramid image, resulting in a blurred version of that level's low-pass image, which is subtracted from the low-pass to yield the blue band-pass image. During reconstruction, the interpolated images and the (optionally filtered) high-pass images are added back together starting with the coarsest level. The Q box indicates quantization or some other pyramid processing, e.g., noise removal by* coring *(setting small wavelet values to 0).*

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 3.33). They then subtract this low-pass version from the original to yield the band-pass "Laplacian" 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 3.33) are sufficient to exactly reconstruct the original image. Figure 3.32 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 of the Laplacian pyramid, where the low-pass image is taken from the original blurred image rather than the reconstructed pyramid (piping

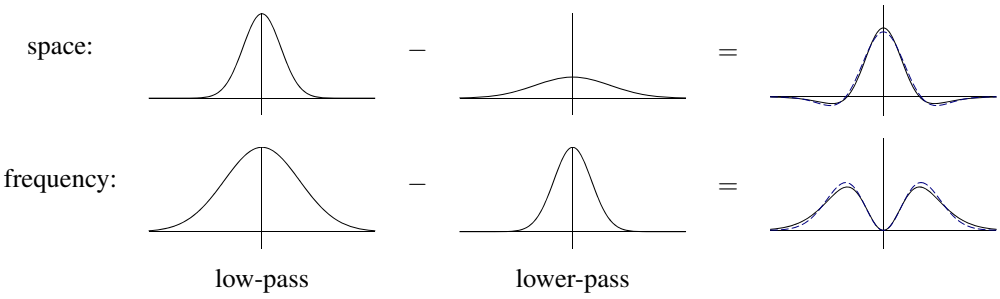


Figure 3.34 *The difference of two low-pass filters results in a band-pass filter. The dashed blue lines show the close fit to a half-octave Laplacian of Gaussian.*

the output of the L box directly to the subtraction in Figure 3.33). 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,

$$\mathrm{DoG}\{I;\sigma_1,\sigma_2\} = G_{\sigma_1} * I - G_{\sigma_2} * I = (G_{\sigma_1} - G_{\sigma_2}) * I. \tag{3.70}$$

A Laplacian of Gaussian (which we saw in (3.26)) is actually its second derivative,

$$\mathrm{LoG}\{I;\sigma\} = \nabla^2(G_\sigma * I) = (\nabla^2 G_\sigma) * I, \tag{3.71}$$

where

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{3.72}$$

is the Laplacian (operator) of a function. Figure 3.34 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).

One particularly useful application of the Laplacian pyramid is in the manipulation of local contrast as well as the tone mapping of high dynamic range images (Section 10.2.1). Paris, Hasinoff, and Kautz (2011) present a technique they call *local Laplacian filters*, which uses local range clipping in the construction of a modified Laplacian pyramid, as well as

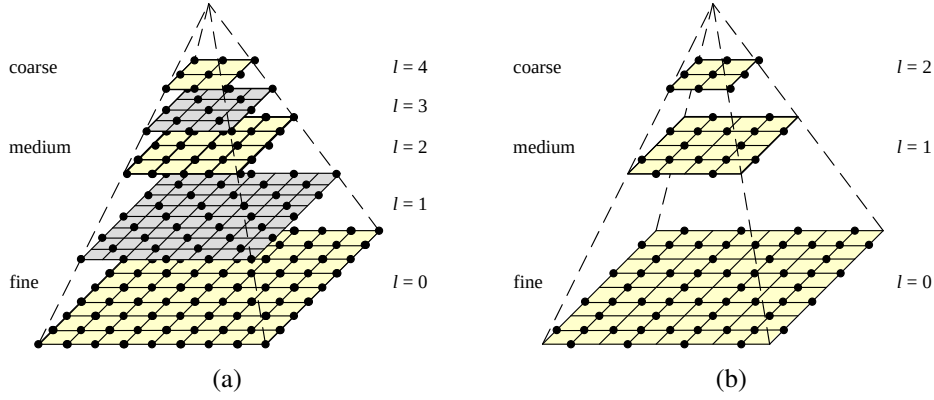


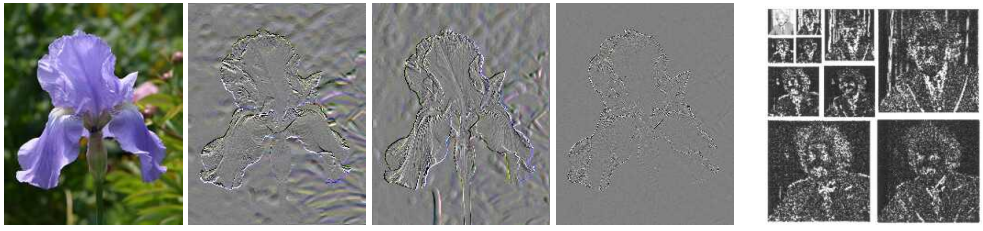
Figure 3.35 *Multiresolution pyramids: (a) pyramid with half-octave (*quincunx*) sampling (odd levels are colored gray for clarity). (b) wavelet pyramid—each wavelet level stores 3/4 of the original pixels (usually the horizontal, vertical, and mixed gradients), so that the total number of wavelet coefficients and original pixels is the same.*

different accentuation and attenuation curves for small and large details, to implement edge-preserving filtering and tone mapping. Aubry, Paris *et al.* (2014) discuss how to accelerate this processing for monotone (single channel) images and also show style transfer applications.

A less widely used variant is *half-octave pyramids*, shown in Figure 3.35a. 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 adjacent 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 7.1.1), 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 7.11).

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) 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.



(a) (b)

Figure 3.36 *A wavelet decomposition of an image: (a) single level decomposition with horizontal, vertical, and diagonal detail wavelets constructed using PyWavelet code (https://pywavelets.readthedocs.io); (b) coefficient magnitudes of a multi-level decomposition, with the high–high components in the lower right corner and the base in the upper left (Buccigrossi and Simoncelli 1999) © 1999 IEEE. Notice how the low–high and high–low components accentuate horizontal and vertical edges and gradients, while the high-high components store the less frequent mixed derivatives.*

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), and 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 *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; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013), as well as for multi-scale oriented filtering (Simoncelli, Freeman *et al.* 1992) and denoising (Portilla, Strela *et al.* 2003).

As 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 3.35b). However, some wavelet families *are*, in fact, overcomplete in order to provide better shiftability or steering in orientation (Simoncelli, Freeman *et al.* 1992). A better distinction, therefore, might be that wavelets are more orientation selective than regular band-pass pyramids.

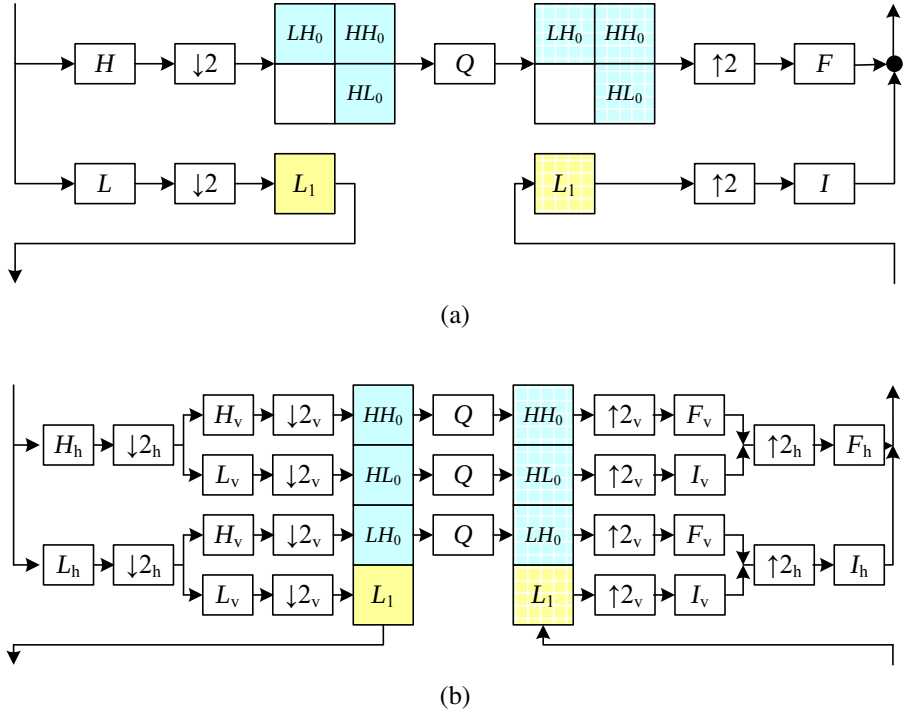


Figure 3.37 *Two-dimensional wavelet decomposition: (a) high-level diagram showing the low-pass and high-pass transforms as single boxes; (b) separable implementation, which involves first performing the wavelet transform horizontally and then vertically. The I and F boxes are the interpolation and filtering boxes required to re-synthesize the image from its wavelet components.*

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 (Figure 3.36).

How are the high-pass H and low-pass L filters shown in Figure 3.37b chosen and how

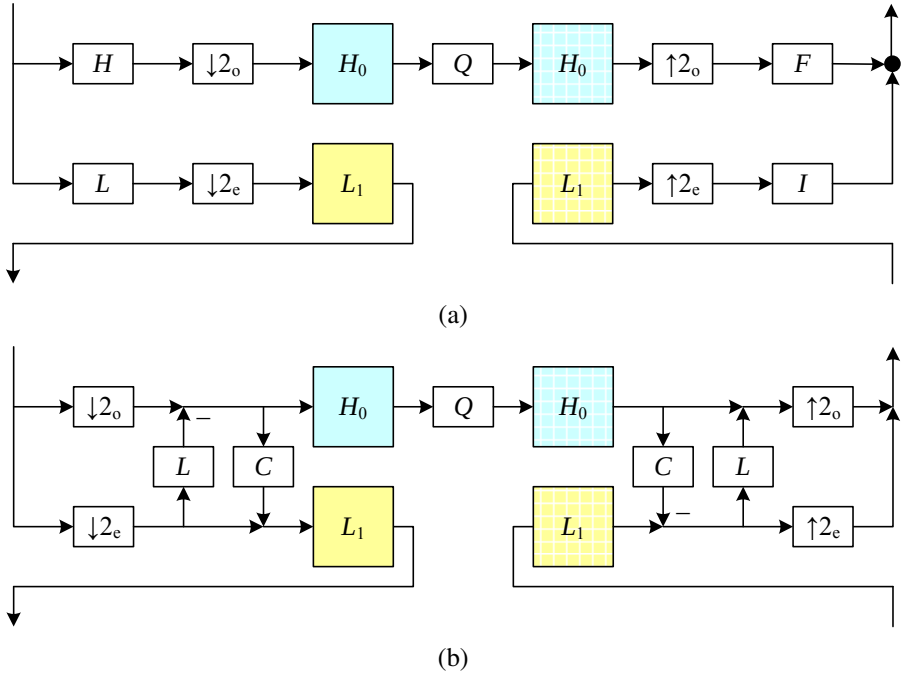
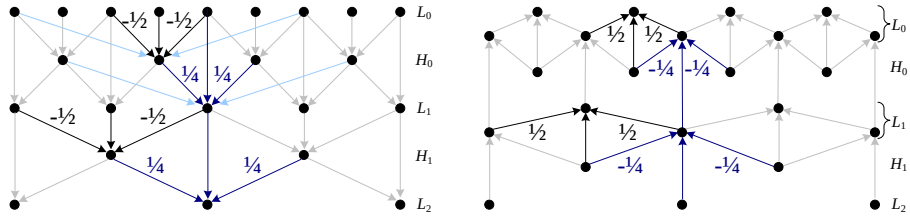


Figure 3.38 *One-dimensional wavelet transform: (a) usual high-pass + low-pass filters followed by odd ($\downarrow 2_o$) and even ($\downarrow 2_e$) downsampling; (b) lifted version, which first selects the odd and even subsequences and then applies a low-pass prediction stage L and a high-pass correction 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)



(a) (b)

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.*

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 odd 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 difference signal 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.

This process can 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,

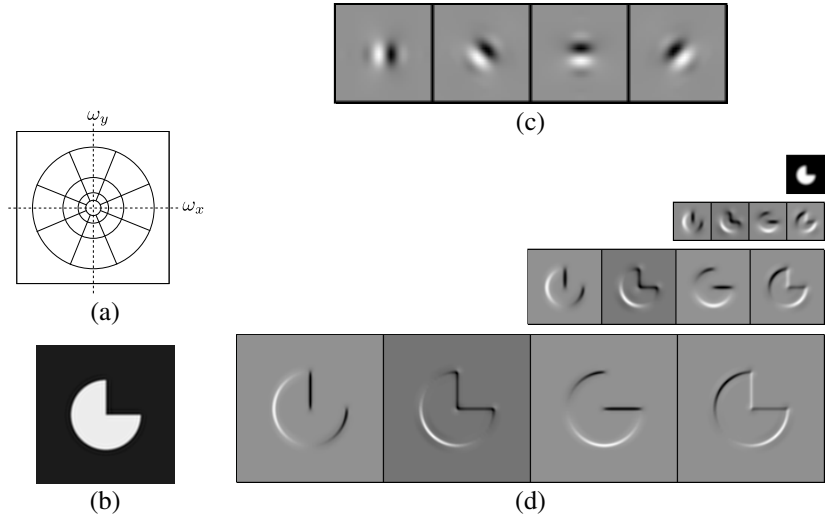


Figure 3.40 *Steerable shiftable multiscale transforms (Simoncelli, Freeman* et al. *1992)* © *1992 IEEE: (a) radial multi-scale frequency domain decomposition; (b) original image; (c) a set of four steerable filters; (d) the radial multi-scale wavelet decomposition.*

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 Chapter 4 (Szeliski 2006b; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013).

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.[15] Simoncelli, Freeman *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*. Their representation is not only overcomplete (which eliminates the aliasing problem) but is

[15] Such aliasing can often be seen as the signal content moving between bands as the original signal is slowly shifted.

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 3.40d illustrates the resulting pyramid sub-bands. Even through 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 *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, *splining* 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 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 3.41g) 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 3.41e–h 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 3.18). This is particularly useful in image stitching and compositing applications, where the exposures may vary between different images, as described in Section 8.4.4, where we also present more recent variants such as Poisson and gradient-domain blending (Pérez, Gangnet, and Blake 2003; Levin, Zomet *et al.* 2004).

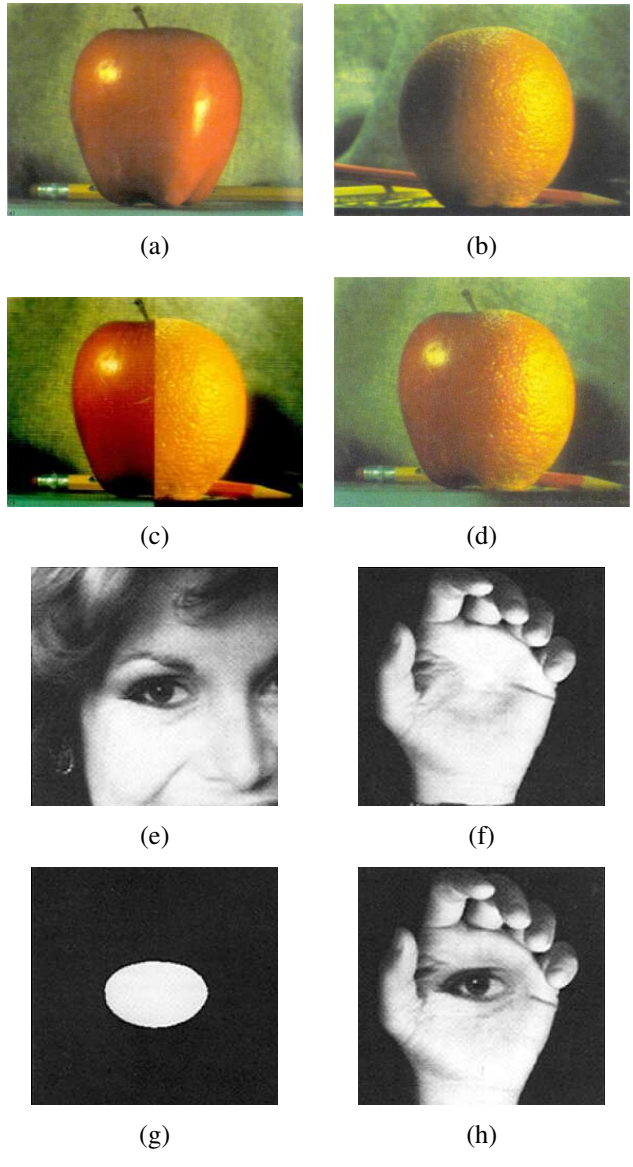


Figure 3.41 *Laplacian pyramid blending (Burt and Adelson 1983b) © 1983 ACM: (a) original image of apple, (b) original image of orange, (c) regular splice, (d) pyramid blend. A masked blend of two images: (e) first input image, (f) second input image, (g) region mask, (h) blended image.*

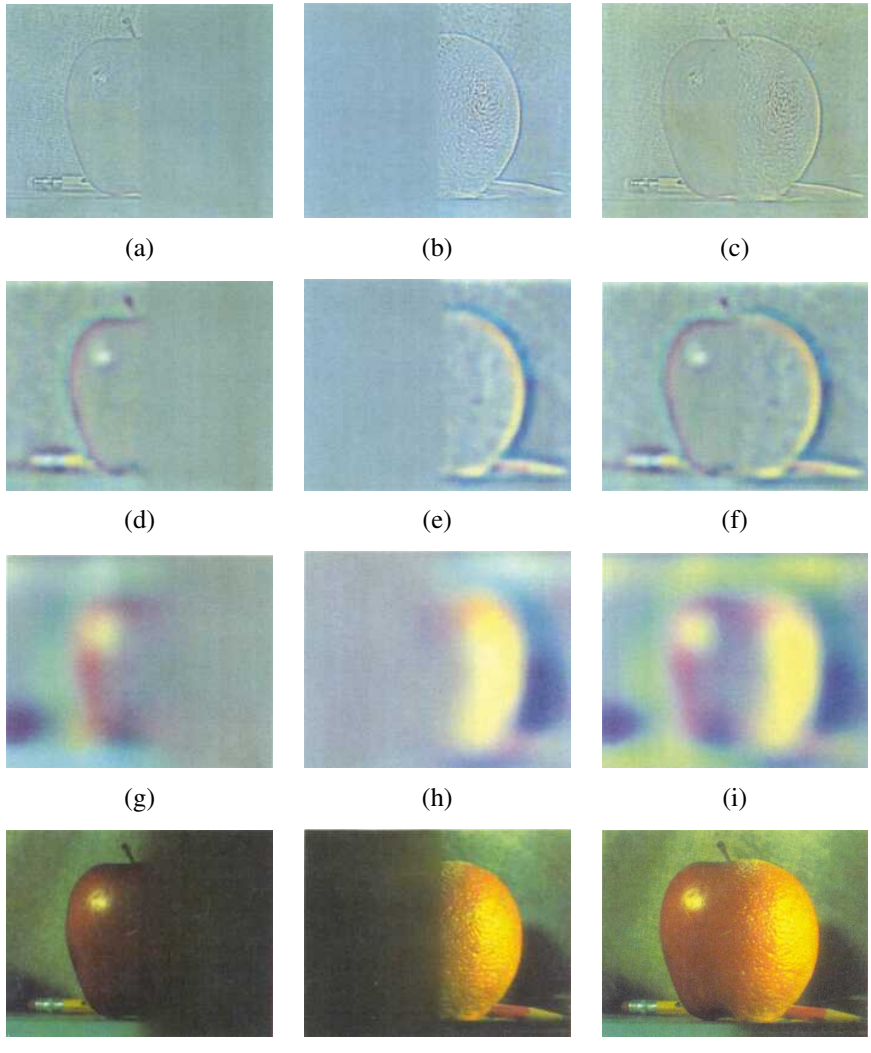


Figure 3.42 *Laplacian pyramid blending details (Burt and Adelson 1983b) © 1983 ACM. The first three rows show the high, medium, and low-frequency parts of the Laplacian pyramid (taken from levels 0, 2, and 4). The left and middle columns show the original apple and orange images weighted by the smooth interpolation functions, while the right column shows the averaged contributions.*

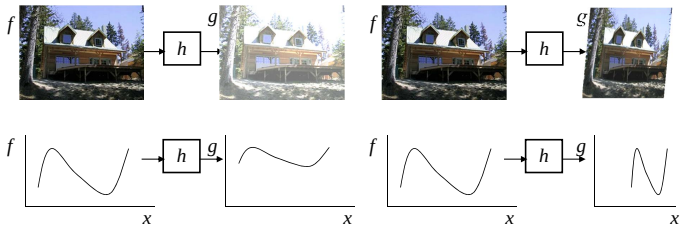


Figure 3.43 *Image warping involves modifying the* domain *of an image function rather than its* range.

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,

$$g(\mathbf{x}) = h(f(\mathbf{x})), \tag{3.73}$$

here we look at functions that transform the *domain*,

$$g(\mathbf{x}) = f(\mathbf{h}(\mathbf{x})), \tag{3.74}$$

as shown in Figure 3.43.

We begin by studying the global *parametric* 2D transformation first introduced in Section 2.1.1. (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 in Section 3.6.3 how image warps can be combined with cross-dissolves to create interesting *morphs* (in-between animations). 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 *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.

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 3.44 shows a few ex-

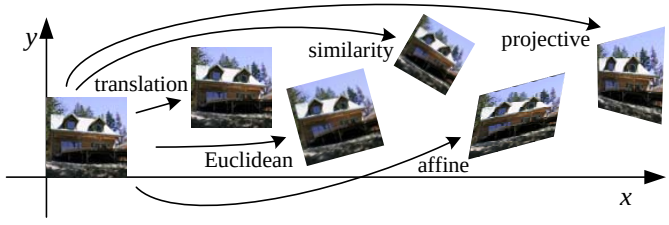


Figure 3.44 *Basic set of 2D geometric image transformations.*

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

Table 3.3 *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.*

amples 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 3.3 for ease of reference.

In general, given a transformation specified by a formula $\mathbf{x}' = \mathbf{h}(\mathbf{x})$ and a source image $f(\mathbf{x})$, how do we compute the values of the pixels in the new image $g(\mathbf{x})$, as given in (3.74)? 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 3.45a. Can you think of any problems with this approach?

In fact, this approach suffers from several limitations. The process of copying a pixel $f(\mathbf{x})$ to a location $\mathbf{x}'$ in g is not well defined when $\mathbf{x}'$ has a non-integer value. What do we

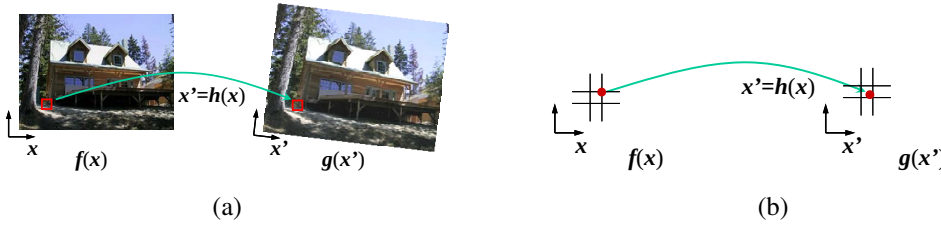


Figure 3.45 *Forward warping algorithm: (a) a pixel $f(\mathbf{x})$ is copied to its corresponding location $\mathbf{x}' = \mathbf{h}(\mathbf{x})$ in image $g(\mathbf{x}')$; (b) detail of the source and destination pixel locations.*

procedure *forwardWarp*$(f, \mathbf{h}, \textbf{out } g)$:

For every pixel $\mathbf{x}$ in $f(\mathbf{x})$

1. Compute the destination location $\mathbf{x}' = \mathbf{h}(\mathbf{x})$.
2. Copy the pixel $f(\mathbf{x})$ to $g(\mathbf{x}')$.

Algorithm 3.1 *Forward warping algorithm for transforming an image $f(\mathbf{x})$ into an image $g(\mathbf{x}')$ through the parametric transform $\mathbf{x}' = \mathbf{h}(\mathbf{x})$.*

do in such a case? What would you do?

You can round the value of $\mathbf{x}'$ 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).

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 $g(\mathbf{x}')$ is sampled from the original image $f(\mathbf{x})$ (Figure 3.46).

How does this differ from the forward warping algorithm? For one thing, since $\hat{\mathbf{h}}(\mathbf{x}')$ is (presumably) defined for all pixels in $g(\mathbf{x}')$, we no longer have holes. More importantly,

Figure 3.46 *Inverse warping algorithm: (a) a pixel $g(\mathbf{x}')$ is sampled from its corresponding location $\mathbf{x} = \hat{\mathbf{h}}(\mathbf{x}')$ in image $f(\mathbf{x})$; (b) detail of the source and destination pixel locations.*

procedure *inverseWarp*$(f, \mathbf{h}, \textbf{out } g)$:

For every pixel $\mathbf{x}'$ in $g(\mathbf{x}')$

1. Compute the source location $\mathbf{x} = \hat{\mathbf{h}}(\mathbf{x}')$
2. Resample $f(\mathbf{x})$ at location $\mathbf{x}$ and copy to $g(\mathbf{x}')$

Algorithm 3.2 *Inverse warping algorithm for creating an image $g(\mathbf{x}')$ from an image $f(\mathbf{x})$ using the parametric transform $\mathbf{x}' = \mathbf{h}(\mathbf{x})$.*

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 $\hat{\mathbf{h}}(\mathbf{x}')$ come from? Quite often, it can simply be computed as the inverse of $\mathbf{h}(\mathbf{x})$. In fact, all of the parametric transforms listed in Table 3.3 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 $f(\mathbf{x})$ given a mapping $\mathbf{x} = \hat{\mathbf{h}}(\mathbf{x}')$ from destination pixels $\mathbf{x}'$ to source pixels $\mathbf{x}$. For example, in optical flow (Section 9.3), we estimate the flow field as the location of the *source* pixel that 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 2.1.5), 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 9.1.3), 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,

$$g(x,y) = \sum_{k,l} f(k,l)h(x-k,y-l), \tag{3.75}$$

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 is

$$g(\mathbf{Ax}) \Leftrightarrow |\mathbf{A}|^{-1} G(\mathbf{A}^{-T}\mathbf{f}). \tag{3.76}$$

For all of the transforms in Table 3.3 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 3.47a), i.e., the local affine approximation to the stretching induced by the projection (Heckbert 1986; Wolberg 1990; Gomes, Darsa *et al.* 1999; Akenine-Möller and Haines 2002).

To prevent aliasing, we need to prefilter 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 prefilter images being used for *texture mapping* in computer graphics. A MIP-map[16] is a standard image pyramid (Figure 3.31), where each level is prefiltered with a high-quality filter rather than a poorer quality approximation, such as Burt and Adelson's (1983b) five-tap binomial. To resample

[16] The term "MIP" stands for *multi in parvo*, meaning "many in one".

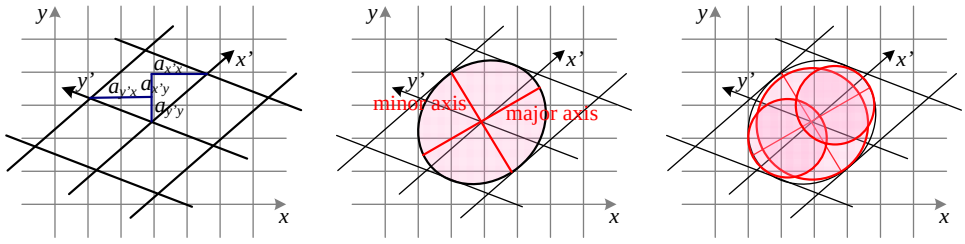


Figure 3.47 *Anisotropic texture filtering: (a) Jacobian of transform* $\mathbf{A}$ *and the induced horizontal and vertical resampling rates* $\{a_{x'x}, a_{x'y}, a_{y'x}, a_{y'y}\}$*; (b) elliptical footprint of an EWA smoothing kernel; (c) anisotropic filtering using multiple samples along the major axis. Image pixels lie at line intersections.*

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,

$$l = \log_2 r. \tag{3.77}$$

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 3.47a). For every destination pixel $\mathbf{x}'$, the ellipsoidal projection of a small pixel grid in $\mathbf{x}'$ onto $\mathbf{x}$ is computed (Figure 3.47b). This is then used to filter the source image $g(\mathbf{x})$ with a Gaussian whose inverse

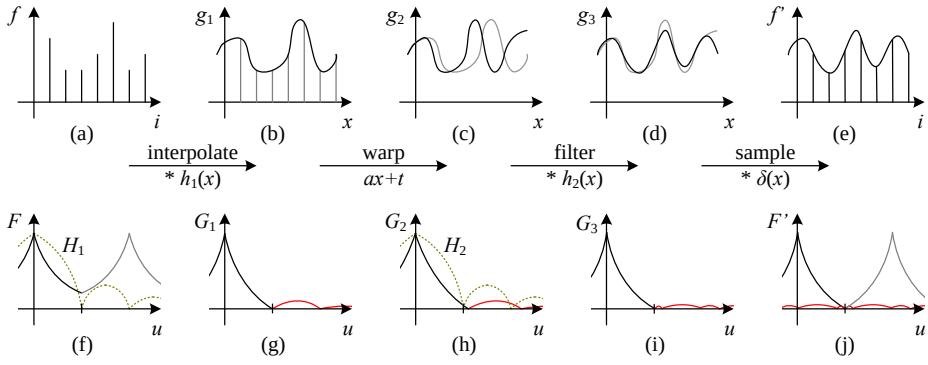


Figure 3.48 *One-dimensional signal resampling (Szeliski, Winder, and Uyttendaele 2010): (a) original sampled signal $f(i)$; (b) interpolated signal $g_1(x)$; (c) warped signal $g_2(x)$; (d) filtered signal $g_3(x)$; (e) sampled signal $f'(i)$. The corresponding spectra are shown below the signals, with the aliased portions shown in red.*

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, as described in (Szeliski, Winder, and Uyttendaele 2010).

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 3.47c).

Multi-pass transforms

The optimal approach to warping images without excessive blurring or aliasing is to adaptively prefilter 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 3.47a). Figure 3.48 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

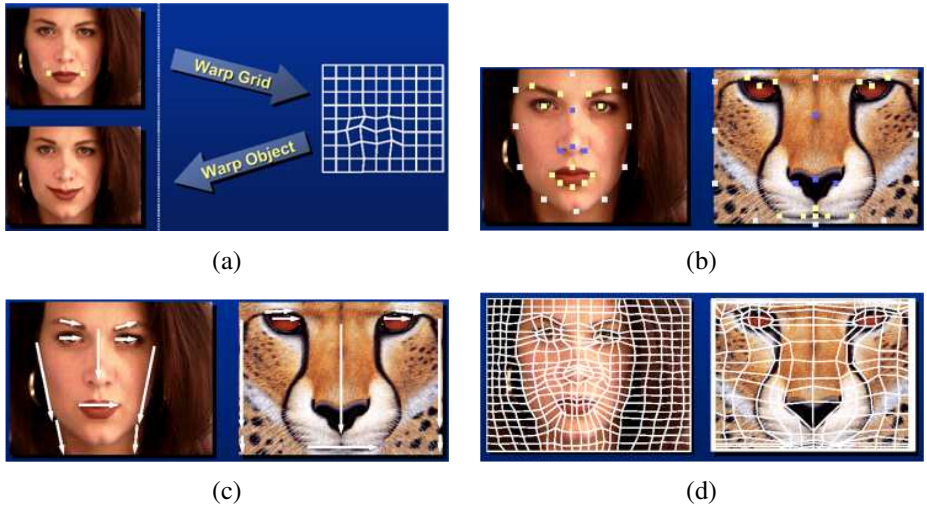


Figure 3.49 *Image warping alternatives (Gomes, Darsa* et al. *1999) © 1999 Morgan Kaufmann: (a) sparse control points ⟶ deformation grid; (b) denser set of control point correspondences; (c) oriented line correspondences; (d) uniform quadrilateral grid.*

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 *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).

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 3.49a). What is needed in this case is to curve the corners of the mouth upwards while

leaving the rest of the face intact.[17] To perform such a transformation, different amounts of motion are required in different parts of the image. Figure 3.49 shows some of the commonly used approaches.

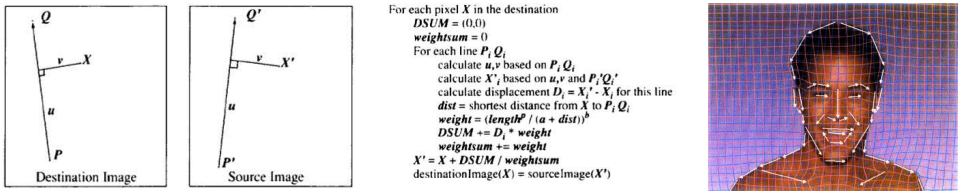
The first approach, shown in Figure 3.49a–b, is to specify a *sparse* set of corresponding points. The displacement of these points can then be interpolated to a dense *displacement field* (Chapter 9) using a variety of techniques, which are described in more detail in Section 4.1 on scattered data interpolation. One possibility is to *triangulate* the set of points in one image (de Berg, Cheong *et al.* 2006; Litwinowicz and Williams 1994; Buck, Finkelstein *et al.* 2000) and to use an *affine* motion model (Table 3.3), 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 3.46) 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 3.49a, using *variational* (energy minimizing) interpolants such as regularization (Litwinowicz and Williams 1994), see Section 4.2, or using locally weighted (*radial basis function*) combinations of displacements (Section 4.1.1). (See Section 4.1 for additional *scattered data interpolation* techniques.) If quadrilateral 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 2002; Lee, Wolberg *et al.* 1996).

In some cases, e.g., if a dense depth map has been estimated for an image (Shade, Gortler *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 3.45), 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 *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 3.49c and 3.50. 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

[17] See Section 6.2.4 on active appearance models for more sophisticated examples of changing facial expression and appearance.



(a) (b) (c)

Figure 3.50 *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.*

segment correspondence specifies a translation, rotation, and scaling, i.e., a *similarity transform* (Table 3.3), for pixels in its vicinity, as shown in Figure 3.50a. 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 3.50a if $u \in [0, 1]$, else the shorter of the two distances to P and Q).

One final possibility for specifying displacement fields is to use a mesh specifically *adapted* to the underlying image content, as shown in Figure 3.49d. Specifying such meshes by hand can involve a fair amount of work; Gomes, Darsa *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 *et al.* 1996; Gomes, Darsa *et al.* 1999).

Figure 3.51 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 3.49), 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 the time parameter that describes the sequence of interpolated frames. The weighting func-

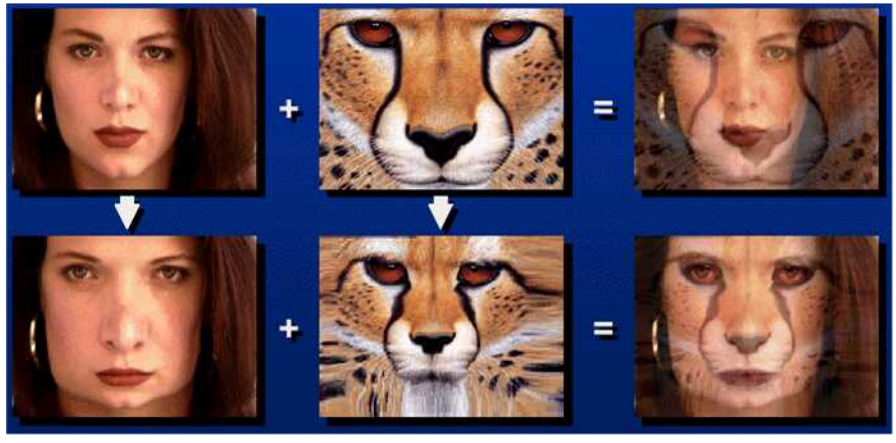


Figure 3.51 *Image morphing (Gomes, Darsa* et al. *1999) © 1999 Morgan Kaufmann. Top row: if the two images are just blended, visible ghosting results. Bottom row: both images are first warped to the same intermediate location (e.g., halfway towards the other image) and the resulting warped images are then blended resulting in a seamless morph.*

tions for the two warped images in the blend are $(1-t)$ and t and the movements of the pixels specified by the correspondences are also linearly interpolated. 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 *et al.* 1993). Exercise 3.25 has you implement a morphing algorithm and test it out under such challenging conditions.

3.7 Additional reading

If you are interested in exploring the topic of image processing in more depth, some popular textbooks have been written by Gomes and Velho (1997), Jähne (1997), Pratt (2007), Burger and Burge (2009), and Gonzalez and Woods (2017). The pre-eminent conference and journal in this field are the IEEE International 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 (2009) provide a good 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; Chen, Paris, and Durand 2007; Paris and Durand 2009; Paris, Kornprobst *et al.* 2008), related iterative algorithms (Saint-Marc, Chen, and Medioni 1991; Nielsen, Florack, and Deriche 1997; Black, Sapiro *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), and guided filtering (Eisemann and Durand 2004; Petschnigg, Agrawala *et al.* 2004; He, Sun, and Tang 2013).

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), and 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; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013), as well as for multi-scale oriented filtering (Simoncelli, Freeman *et al.* 1992) and denoising (Portilla, Strela *et al.* 2003).

While image pyramids (Section 3.5.3) are usually constructed using linear filtering operators, more recent work uses 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 *et al.* (2008), and Fattal (2009).

High-quality algorithms for image warping and resampling are covered both in the image processing literature (Wolberg 1990; Dodgson 1992; Gomes, Darsa *et al.* 1999; Szeliski, Winder, and Uyttendaele 2010) and in computer graphics (Williams 1983; Heckbert 1986; Barkans 1997; Weinhaus and Devarajan 1997; Akenine-Möller and Haines 2002), where they go under the name of *texture mapping*. Combinations 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 *et al.* 1999).

3.8 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. 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 2.8 for some related ideas.

Ex 3.2: Demosaicing. If you have access to the RAW image for the camera, perform the demosaicing yourself (Section 10.3.1). If not, just subsample an RGB image in a Bayer mosaic pattern. Instead of just bilinear interpolation, try one of the more advanced techniques described in Section 10.3.1. Compare your result to the one produced by the camera. 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.

Ex 3.3: 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 3.4: 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 3.5: Difference keying. Implement a difference keying algorithm (see Section 3.1.3) (Toyama, Krumm *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 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 3.3.3), compute their centroids, and track them from frame to frame. Use this to build a "people counter".

Ex 3.6: 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 3.7: 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 (2.117).

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 3.8: 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 3.7 (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 3.9: 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),

$$\tilde{f}(i,j) = f(k,l), \qquad \begin{aligned} k &= \max(0, \min(M-1, i)), \\ l &= \max(0, \min(N-1, j)), \end{aligned}$$

(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 3.10: 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 3.11: 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 1? 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 prefilter 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 3.12: 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 3.4.2.

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 3.13: 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 7.4).

(a) (b) (c)

Figure 3.52 *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.*

Ex 3.14: Bilateral and guided image filters. Implement or download code for bilateral and/or guided image filtering and use this to implement some image enhancement or processing application, such as those described in Section 3.3.2

Ex 3.15: Fourier transform. Prove the properties of the Fourier transform listed in Szeliski (2010, Table 3.1) and derive the formulas for the Fourier transforms pairs listed in Szeliski (2010, Table 3.2) and Table 3.1. 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: High-quality image resampling. Implement several of the low-pass filters presented in Section 3.5.2 and also the windowed sinc shown in Figure 3.28. 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 3.52a) and natural images with lots of high-frequency detail (Figure 3.52b–c).

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 the tradeoff between speed and quality.

Ex 3.17: 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:

- 2×2 block filtering;
- Burt and Adelson's binomial kernel $^1/_{16}(1, 4, 6, 4, 1)$ (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 3.18: 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 3.41).
4. Reconstruct the final image from the blended Laplacian pyramid.

Generalize your algorithm to input n images and a label image with values $1 \ldots n$ (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 3.19: Pyramid blending in PyTorch. Re-write your pyramid blending exercise in PyTorch.

1. PyTorch has support for all of the primitives you need, i.e., fixed size convolutions (make sure they filter each channel separately), downsampling, upsampling, and addition, subtraction, and multiplication (although the latter is rarely used).
2. The goal of this exercise is *not* to train the convolution weights, but just to become familiar with the DNN primitives available in PyTorch.
3. Compare your results to the ones using a standard Python or C++ computer vision library. They should be identical.
4. Discuss whether you like this API better or worse for these kinds of fixed pipeline imaging tasks.

Ex 3.20: Local Laplacian—challenging. Implement the local Laplacian contrast manipulation technique (Paris, Hasinoff, and Kautz 2011; Aubry, Paris *et al.* 2014) and use this to implement edge-preserving filtering and tone manipulation.

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 3.17). 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, $1/4$ 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 that forward warps the displacement field, fills in small holes, and then uses inverse warping (Shade, Gortler *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.

Figure 3.53 *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)?*

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 3.53, and enhance the strength (saturation) of the rainbow.

1. Draw an arc in the image delineating the extent of the rainbow.
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.

Chapter 4

Model fitting and optimization

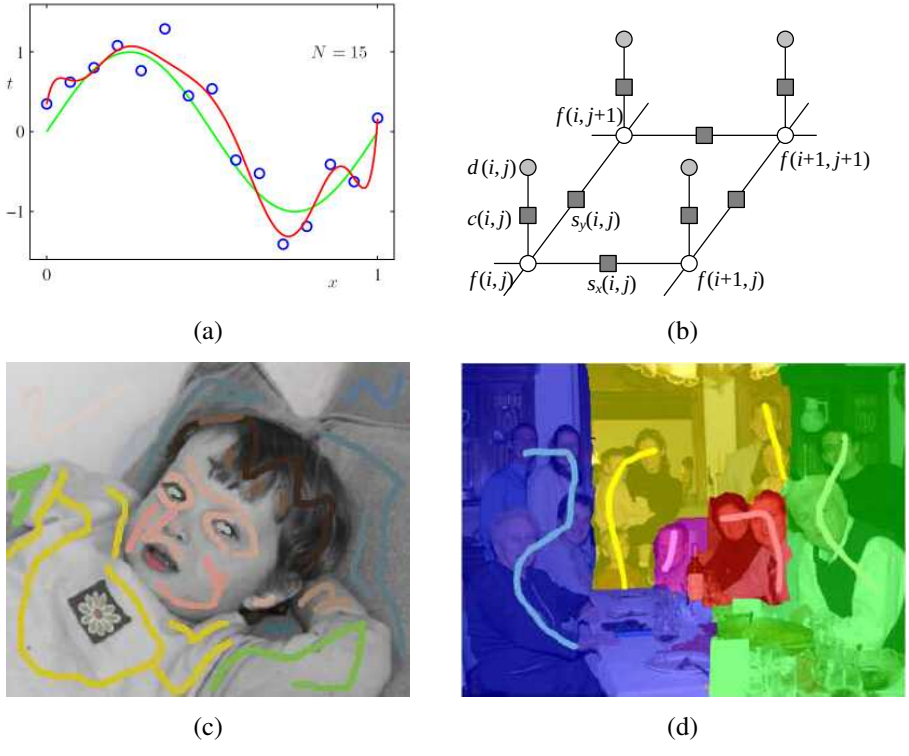


Figure 4.1 *Examples of data interpolation and global optimization: (a) scattered data interpolation (curve fitting) (Bishop 2006) © 2006 Springer; (b) graphical model interpretation of first-order regularization; (c) colorization using optimization (Levin, Lischinski, and Weiss 2004) © 2004 ACM; (d) multi-image photomontage formulated as an unordered label MRF (Agarwala, Dontcheva* et al. *2004) © 2004 ACM.*

In the previous chapter, we 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 situations, however, we are given *incomplete* data as input, such as depths at a sparse number of locations, or user scribbles suggesting how an image should be colorized or segmented (Figure 4.1c–d).

The problem of interpolating a complete image (or more generally a *function* or *field*) from incomplete or varying quality data is often called *scattered data interpolation*. We begin this chapter with a review of techniques in this area, since in addition to being widely used in computer vision, they also form the basis of most machine learning algorithms, which we will study in the next chapter.

Instead of doing an exhaustive survey, we present in Section 4.1 some easy-to-use techniques, such as triangulation, spline interpolation, and radial basis functions. While these techniques are widely used, they cannot easily be modified to provide *controlled continuity*, i.e., to produce the kinds of piecewise continuous reconstructions we expect when estimating depth maps, label maps, or even color images.

For this reason, we introduce in Section 4.2 *variational methods*, which formulate the interpolation problem as the recovery of a piecewise smooth function subject to exact or approximate data constraints. Because the smoothness is controlled using penalties formulated as norms of the function, this class of techniques are often called *regularization* or *energy-based* approaches. To find the minimum-energy solutions to these problems, we discretize them (typically on a pixel grid), resulting in a discrete energy, which can then be minimized using sparse linear systems or related iterative techniques.

In the last part of this chapter, Section 4.3, we show how such energy-based formulations are related to Bayesian inference techniques formulated as *Markov random fields*, which are a special case of general probabilistic *graphical models*. In these formulations, data constraints can be interpreted as noisy and/or incomplete measurements, and piecewise smoothness constraints as *prior assumptions* or *models* over the solution space. Such formulations are also often called *generative models*, since we can, in principle, generate random samples from the prior distribution to see if they conform with our expectations. Because the prior models can be more complex than simple smoothness constraints, and because the solution space can have multiple local minima, more sophisticated optimization techniques have been developed, which we discuss in this section.

4.1 Scattered data interpolation

The goal of *scattered data interpolation* is to produce a (usually continuous and smooth) function $\mathbf{f}(\mathbf{x})$ that passes *through* a set of data points $\mathbf{d}_k$ placed at locations $\mathbf{x}_k$ such that

$$\mathbf{f}(\mathbf{x}_k) = \mathbf{d}_k. \tag{4.1}$$

The related problem of *scattered data approximation* only requires the function to pass *near* the data points (Amidror 2002; Wendland 2004; Anjyo, Lewis, and Pighin 2014). This is usually formulated using a penalty function such as

$$E_{\mathrm{D}} = \sum_k \|\mathbf{f}(\mathbf{x}_k) - \mathbf{d}_k\|^2, \tag{4.2}$$

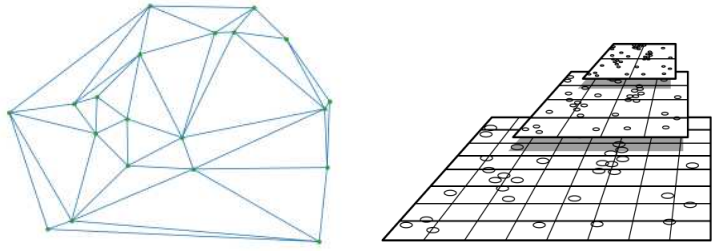
with the squared norm in the above formula sometimes replaced by a different norm or robust function (Section 4.1.3). In statistics and machine learning, the problem of predicting an output function given a finite number of samples is called *regression* (Section 5.1). The $\mathbf{x}$ vectors are called the *inputs* and the outputs $\mathbf{y}$ are called the *targets*. Figure 4.1a shows an example of one-dimensional scattered data interpolation, while Figures 4.2 and 4.8 show some two-dimensional examples.

At first glance, scattered data interpolation seems closely related to *image interpolation*, which we studied in Section 3.5.1. However, unlike images, which are regularly gridded, the data points in scattered data interpolation are irregularly placed throughout the domain, as shown in Figure 4.2. This requires some adjustments to the interpolation methods we use.

If the domain $\mathbf{x}$ is two-dimensional, as is the case with images, one simple approach is to *triangulate* the domain $\mathbf{x}$ using the data locations $\mathbf{x}_k$ as the triangle vertices. The resulting triangular network, shown in Figure 4.2a, is called a *triangular irregular network* (TIN), and was one of the early techniques used to produce elevation maps from scattered field measurements collected by surveys.

The triangulation in Figure 4.2a was produced using a *Delaunay triangulation*, which is the most widely used planar triangulation technique due to its attractive computational properties, such as the avoidance of long skinny triangles. Algorithms for efficiently computing such triangulation are readily available[1] and covered in textbooks on computational geometry (Preparata and Shamos 1985; de Berg, Cheong *et al.* 2008). The Delaunay triangulation can be extended to higher-dimensional domains using the property of circumscribing spheres, i.e., the requirement that all selected simplices (triangles, tetrahedra, etc.) have no other vertices inside their circumscribing spheres.

[1] For example, https://docs.scipy.org/doc/scipy/reference/tutorial/spatial.html



(a) (b)

Figure 4.2 *Some simple scattered data interpolation and approximation algorithms: (a) a Delaunay triangulation defined over a set of data point locations; (b) data structure and intermediate results for the* pull-push *algorithm (Gortler, Grzeszczuk* et al. *1996)* © *1996 ACM.*

Once the triangulation has been defined, it is straightforward to define a piecewise-linear interpolant over each triangle, resulting in an interpolant that is C_0 but not generally C_1 continuous. The formulas for the function inside each triangle are usually derived using *barycentric coordinates*, which attain their maximal values at the vertices and sum up to one (Farin 2002; Amidror 2002).

If a smoother surface is desired as the interpolant, we can replace the piecewise linear functions on each triangle with higher-order *splines*, much as we did for image interpolation (Section 3.5.1). However, since these splines are now defined over irregular triangulations, more sophisticated techniques must be used (Farin 2002; Amidror 2002). Other, more recent interpolators based on geometric modeling techniques in computer graphics include subdivision surfaces (Peters and Reif 2008).

An alternative to triangulating the data points is to use a regular n-dimensional grid, as shown in Figure 4.2b. Splines defined on such domains are often called *tensor product splines* and have been used to interpolate scattered data (Lee, Wolberg, and Shin 1997).

An even faster, but less accurate, approach is called the *pull-push* algorithm and was originally developed for interpolating missing 4D lightfield samples in a Lumigraph (Gortler, Grzeszczuk *et al.* 1996). The algorithm proceeds in three phases, as schematically illustrated in Figure 4.2b.

First, the irregular data samples are *splatted* onto (i.e., spread across) the nearest grid vertices, using the same approach we discussed in Section 3.6.1 on parametric image transformations. The splatting operations accumulate both values and weights at nearby vertices. In the second, *pull*, phase, values and weights are computed at a hierarchical set of lower resolution grids by combining the coefficient values from the higher resolution grids. In the lower

resolution grids, the gaps (regions where the weights are low) become smaller. In the third, *push*, phase, information from each lower resolution grid is combined with the next higher resolution grid, filling in the gaps while not unduly blurring the higher resolution information already computed. Details of these three stages can be found in (Gortler, Grzeszczuk *et al.* 1996).

The pull-push algorithm is very fast, since it is essentially linear in the number of input data points and fine-level grid samples.

4.1.1 Radial basis functions

While the mesh-based representations I have just described can provide good-quality interpolants, they are typically limited to low-dimensional domains, because the size of the mesh grows combinatorially with the dimensionality of the domain. In higher dimensions, it is common to use *mesh-free* approaches that define the desired interpolant as a weighted sum of basis functions, similar to the formulation used in image interpolation (3.64). In machine learning, such approaches are often called *kernel functions* or *kernel regression* (Bishop 2006, Chapter 6; Murphy 2012, Chapter 14; Schölkopf and Smola 2001).

In more detail, the interpolated function f is a weighted sum (or *superposition*) of basis functions centered at each input data point

$$\mathbf{f}(\mathbf{x}) = \sum_k \mathbf{w}_k \phi(\|\mathbf{x} - \mathbf{x}_k\|), \tag{4.3}$$

where the $\mathbf{x}_k$ are the locations of the scattered data points, the ϕs are the *radial basis functions* (or kernels), and $\mathbf{w}_k$ are the local *weights* associated with each kernel. The basis functions $\phi()$ are called *radial* because they are applied to the radial distance between a data sample $\mathbf{x}_k$ and an evaluation point $\mathbf{x}$. The choice of ϕ determines the smoothness properties of the interpolant, while the choice of weights w_k determines how closely the function approximates the input.

Some commonly used basis functions (Anjyo, Lewis, and Pighin 2014) include

$$\text{Gaussian} \qquad \phi(r) = \exp(-r^2/c^2) \tag{4.4}$$

$$\text{Hardy multiquadric} \qquad \phi(r) = \sqrt{(r^2 + c^2)} \tag{4.5}$$

$$\text{Inverse multiquadric} \qquad \phi(r) = 1/\sqrt{(r^2 + c^2)} \tag{4.6}$$

$$\text{Thin plate spline} \qquad \phi(r) = r^2 \log r. \tag{4.7}$$

In these equations, r is the radial distance and c is a scale parameter that controls the size (radial falloff) of the basis functions, and hence its smoothness (more compact bases lead to

"peakier" solutions). The thin plate spline equation holds for two dimensions (the general n-dimensional spline is called the *polyharmonic spline* and is given in (Anjyo, Lewis, and Pighin 2014)) and is the analytic solution to the second degree variational spline derived in (4.19).

If we want our function to exactly interpolate the data values, we solve the linear system of equations (4.1), i.e.,

$$\mathbf{f}(\mathbf{x}_k) = \sum_l \mathbf{w}_l \phi(\|\mathbf{x}_k - \mathbf{x}_l\|) = \mathbf{d}_k, \tag{4.8}$$

to obtain the desired set of weights $\mathbf{w}_k$. Note that for large amounts of basis function overlap (large values of c), these equations may be quite *ill-conditioned*, i.e., small changes in data values or locations can result in large changes in the interpolated function. Note also that the solution of such a system of equations is in general $O(m^3)$, where m is the number of data points (unless we use basis functions with finite extent to obtain a sparse set of equations).

A more prudent approach is to solve the *regularized data approximation problem*, which involves minimizing the data constraint energy (4.2) together with a weight penalty (*regularizer*) of the form

$$E_{\mathrm{W}} = \sum_k \|\mathbf{w}_k\|^p, \tag{4.9}$$

and to then minimize the regularized least squares problem

$$E(\{w_k\}) = E_{\mathrm{D}} + \lambda E_{\mathrm{W}} \tag{4.10}$$

$$= \sum_k \| \sum_l \mathbf{w}_l \phi(\|\mathbf{x}_k - \mathbf{x}_l\|) - \mathbf{d}_k\|^2 + \lambda \sum_k \|\mathbf{w}_k\|^p. \tag{4.11}$$

When $p = 2$ (quadratic weight penalty), the resulting energy is a pure least squares problem, and can be solved using the *normal equations* (Appendix A.2), where the λ value gets added along the diagonal to stabilize the system of equations.

In statistics and machine learning, the quadratic (regularized least squares) problem is called *ridge regression*. In neural networks, adding a quadratic penalty on the weights is called *weight decay*, because it encourages weights to decay towards zero (Section 5.3.3). When $p = 1$, the technique is called *lasso* (least absolute shrinkage and selection operator), since for sufficiently large values of λ, many of the weights $\mathbf{w}_k$ get driven to zero (Tibshirani 1996; Bishop 2006; Murphy 2012; Deisenroth, Faisal, and Ong 2020). This results in a *sparse* set of basis functions being used in the interpolant, which can greatly speed up the computation of new values of $\mathbf{f}(\mathbf{x})$. We will have more to say on sparse kernel techniques in the section on Support Vector Machines (Section 5.1.4).

An alternative to solving a set of equations to determine the weights $\mathbf{w}_k$ is to simply set them to the input data values $\mathbf{d}_k$. However, this fails to interpolate the data, and instead

produces higher values in higher density regions. This can be useful if we are trying to estimate a probability density function from a set of samples. In this case, the resulting density function, obtained after normalizing the sum of sample-weighted basis functions to have a unit integral, is called the *Parzen window* or *kernel* approach to probability density estimation (Duda, Hart, and Stork 2001, Section 4.3; Bishop 2006, Section 2.5.1). Such probability densities can be used, among other things, for (spatially) clustering color values together for image segmentation in what is known as the *mean shift* approach (Comaniciu and Meer 2002) (Section 7.5.2).

If, instead of just estimating a density, we wish to actually interpolate a set of data values $\mathbf{d}_k$, we can use a related technique known as *kernel regression* or the *Nadaraya-Watson* model, in which we divide the data-weighted summed basis functions by the sum of all the basis functions,

$$\mathbf{f}(\mathbf{x}) = \frac{\sum_k \mathbf{d}_k \phi(\|\mathbf{x} - \mathbf{x}_k\|)}{\sum_l \phi(\|\mathbf{x} - \mathbf{x}_l\|)}. \tag{4.12}$$

Note how this operation is similar, in concept, to the *splatting* method for forward rendering we discussed in Section 3.6.1, except that here, the bases can be much wider than the nearest-neighbor bilinear bases used in graphics (Takeda, Farsiu, and Milanfar 2007).

Kernel regression is equivalent to creating a new set of spatially varying normalized shifted basis functions

$$\phi'_k(\mathbf{x}) = \frac{\phi(\|\mathbf{x} - \mathbf{x}_k\|)}{\sum_l \phi(\|\mathbf{x} - \mathbf{x}_l\|)}, \tag{4.13}$$

which form a *partition of unity*, i.e., sum up to 1 at every location (Anjyo, Lewis, and Pighin 2014). While the resulting interpolant can now be written more succinctly as

$$\mathbf{f}(\mathbf{x}) = \sum_k \mathbf{d}_k \phi'_k(\|\mathbf{x} - \mathbf{x}_k\|), \tag{4.14}$$

in most cases, it is more expensive to precompute and store the K ϕ'_k functions than to evaluate (4.12).

While not that widely used in computer vision, kernel regression techniques have been applied by Takeda, Farsiu, and Milanfar (2007) to a number of low-level image processing operations, including state-of-the-art handheld multi-frame super-resolution (Wronski, Garcia-Dorado *et al.* 2019).

One last scattered data interpolation technique worth mentioning is *moving least squares*, where a weighted subset of nearby points is used to compute a local smooth surface. Such techniques are mostly widely used in 3D computer graphics, especially for point-based surface modeling, as discussed in Section 13.4 and (Alexa, Behr *et al.* 2003; Pauly, Keiser *et al.* 2003; Anjyo, Lewis, and Pighin 2014).

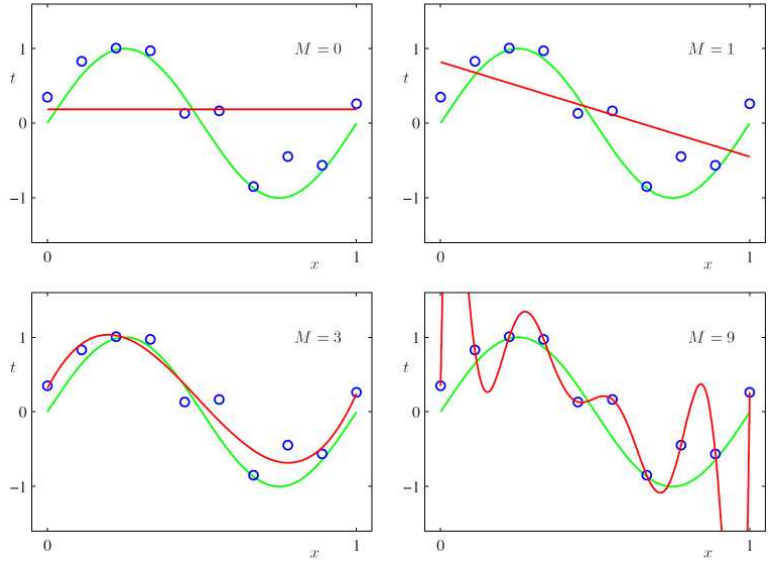


Figure 4.3 *Polynomial curve fitting to the blue circles, which are noisy samples from the green sine curve (Bishop 2006) © 2006 Springer. The four plots show the 0th order constant function, the first order linear fit, the $M = 3$ cubic polynomial, and the 9th degree polynomial. Notice how the first two curves exhibit* underfitting*, while the last curve exhibits* overfitting*, i.e., excessive wiggle.*

4.1.2 Overfitting and underfitting

When we introduced weight regularization in (4.9), we said that it was usually preferable to approximate the data but we did not explain why. In most data fitting problems, the samples $\mathbf{d}_k$ (and sometimes even their locations $\mathbf{x}_k$) are noisy, so that fitting them exactly makes no sense. In fact, doing so can introduce a lot of spurious wiggles, when the true solution is likely to be smoother.

To delve into this phenomenon, let us start with a simple polynomial fitting example taken from (Bishop 2006, Chapter 1.1). Figure 4.3 shows a number of polynomial curves of different orders M fit to the blue circles, which are noisy samples from the underlying green sine curve. Notice how the low-order ($M = 0$ and $M = 1$) polynomials severely *underfit* the underlying data, resulting in curves that are too flat, while the $M = 9$ polynomial, which exactly fits the data, exhibits far more wiggle than is likely.

How can we quantify this amount of underfitting and overfitting, and how can we get just the right amount? This topic is widely studied in machine learning and covered in a number of

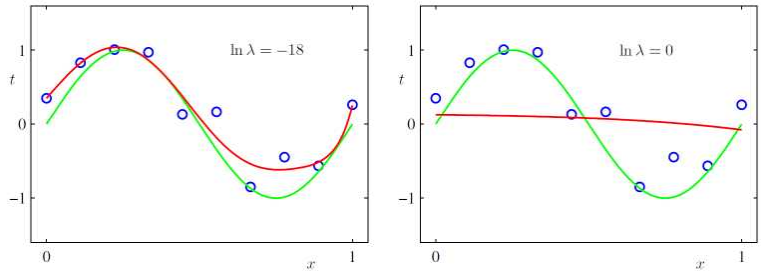


Figure 4.4 *Regularized $M = 9$ polynomial fitting for two different values of λ (Bishop 2006) © 2006 Springer. The left plot shows a reasonable amount of regularization, resulting in a plausible fit, while the larger value of λ on the right causes underfitting.*

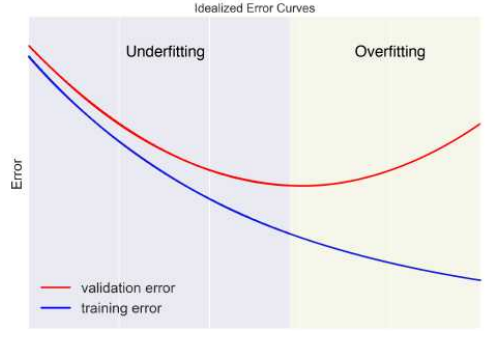


Figure 4.5 *Fitting (training) and validation errors as a function of the amount of regularization or smoothing © Glassner (2018). The less regularized solutions on the right, while exhibiting lower fitting error, perform less well on the validation data.*

texts, including Bishop (2006, Chapter 1.1), Glassner (2018, Chapter 9), Deisenroth, Faisal, and Ong (2020, Chapter 8), and Zhang, Lipton *et al.* (2021, Section 4.4.3).

One approach is to use regularized least squares, introduced in (4.11). Figure 4.4 shows an $M = 9$th degree polynomial fit obtained by minimizing (4.11) with the polynomial basis functions $\phi_k(x) = x^k$ for two different values of λ. The left plot shows a reasonable amount of regularization, resulting in a plausible fit, while the larger value of λ on the right causes underfitting. Note that the $M = 9$ interpolant shown in the lower right quadrant of Figure 4.3 corresponds to the unregularized $\lambda = 0$ case.

If we were to now measure the difference between the red (estimated) and green (noise-free) curves, we see that choosing a good intermediate value of λ will produce the best result.

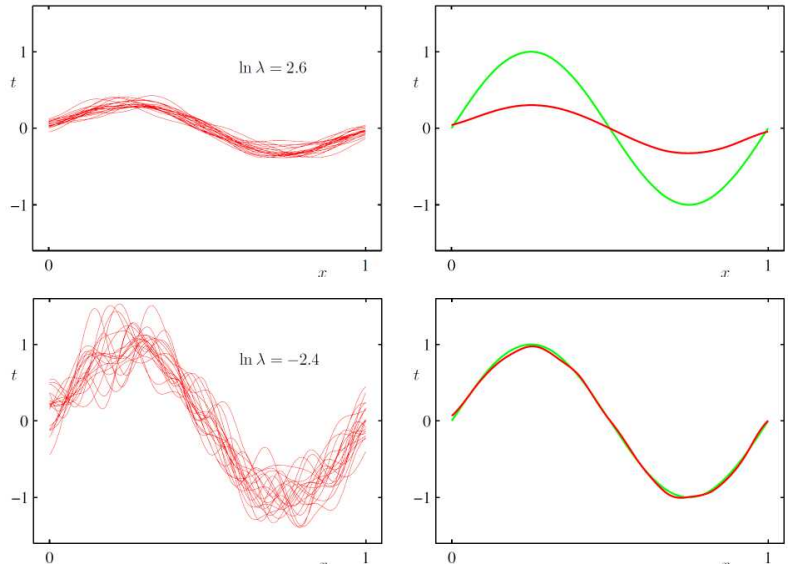


Figure 4.6 *The more heavily regularized solution* $\log \lambda = 2.6$ *exhibits higher bias (deviation from original curve) than the less heavily regularized version (*$\log \lambda = -2.4$*), which has much higher variance (Bishop 2006) © 2006 Springer. The red curves on the left are* $M = 24$ *Gaussian basis fits to 25 randomly sampled points on the green curve. The red curve on the right is their mean.*

In practice, however, we never have access to samples from the noise-free data.

Instead, if we are given a set of samples to interpolate, we can save some in a *validation* set in order to see if the function we compute is underfitting or overfitting. When we vary a parameter such as λ (or use some other measure to control smoothness), we typically obtain a curve such as the one shown in Figure 4.5. In this figure, the blue curve denotes the fitting error, which in this case is called the *training error*, since in machine learning, we usually split the given data into a (typically larger) training set and a (typically smaller) validation set.

To obtain an even better estimate of the ideal amount of regularization, we can repeat the process of splitting our sample data into training and validation sets several times. One well-known technique, called *cross-validation* (Craven and Wahba 1979; Wahba and Wendelberger 1980; Bishop 2006, Section 1.3; Murphy 2012, Section 1.4.8; Deisenroth, Faisal, and Ong 2020, Chapter 8; Zhang, Lipton *et al.* 2021, Section 4.4.2), splits the training data into K *folds* (equal sized pieces). You then put aside each fold, in turn, and train on the remaining

data. You can then estimate the best regularization parameter by averaging over all K training runs. While this generally works well ($K = 5$ is often used), it may be too expensive when training large neural networks because of the long training times involved.

Cross-validation is just one example of a class of *model selection* techniques that estimate *hyperparameters* in a training algorithm to achieve good performance. Additional methods include *information criteria* such as the Bayesian information criterion (BIC) (Torr 2002) and the Akaike information criterion (AIC) (Kanatani 1998), and Bayesian modeling approaches (Szeliski 1989; Bishop 2006; Murphy 2012).

One last topic worth mention with regard to data fitting, since it comes up often in discussions of statistical machine learning techniques, is the *bias-variance tradeoff* (Bishop 2006, Section 3.2). As you can see in Figure 4.6, using a large amount of regularization (top row) results in much lower variance between different random sample solutions, but much higher bias away from the true solution. Using insufficient regularization increases the variance dramatically, although an average over a large number of samples has low bias. The trick is to determine a reasonable compromise in terms of regularization so that any individual solution has a good expectation of being close to the *ground truth* (original clean continuous) data.

4.1.3 Robust data fitting

When we added a regularizer on the weights in (4.9), we noted that it did not have to be a quadratic penalty and could, instead, be a lower-order monomial that encouraged *sparsity* in the weights.

This same idea can be applied to data terms such as (4.2), where, instead of using a quadratic penalty, we can use a *robust loss function* $\rho()$,

$$E_{\mathrm{R}} = \sum_k \rho(\|\mathbf{r}_k\|), \quad \text{with } \mathbf{r}_k = \mathbf{f}(\mathbf{x}_k) - \mathbf{d}_k, \tag{4.15}$$

which gives lower weights to larger data fitting errors, which are more likely to be outlier measurements. (The fitting error term $\mathbf{r}_k$ is called the *residual error*.)

Some examples of loss functions from (Barron 2019) are shown in Figure 4.7 along with their derivatives. The regular quadratic ($\alpha = 2$) penalty gives full (linear) weight to each error, whereas the $\alpha = 1$ loss gives equal weight to all larger residuals, i.e., it behaves as an L_1 loss for large residuals, and L_2 for small ones. Even larger values of α discount large errors (outliers) even more, although they result in optimization problems that are *non-convex*, i.e., that can have multiple local minima. We will discuss techniques for finding good initial guesses for such problems later on in Section 8.1.4.

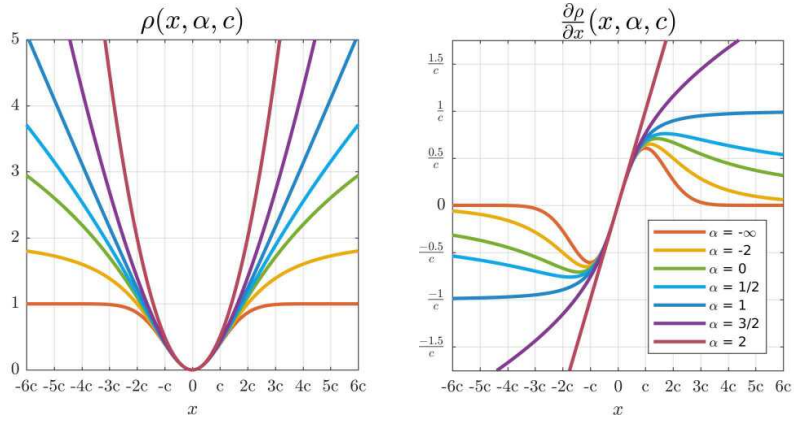
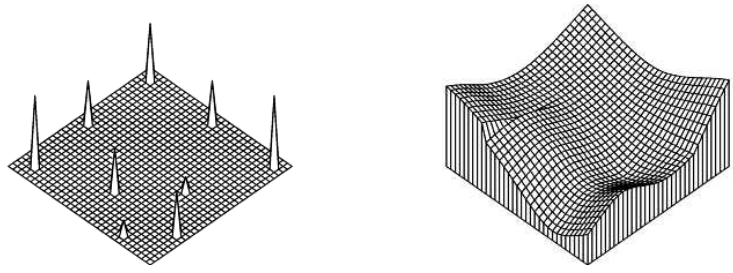


Figure 4.7 *A general and adaptive loss function (left) and its gradient (right) for different values of its shape parameter α (Barron 2019) © 2019 IEEE. Several values of α reproduce existing loss functions: L_2 loss ($\alpha = 2$), Charbonnier loss ($\alpha = 1$), Cauchy loss ($\alpha = 0$), Geman-McClure loss ($\alpha = -2$), and Welsch loss ($\alpha = -1$).*

In statistics, minimizing non-quadratic loss functions to deal with potential outlier measurements is known as *M-estimation* (Huber 1981; Hampel, Ronchetti *et al.* 1986; Black and Rangarajan 1996; Stewart 1999). Such estimation problems are often solved using *iteratively reweighted least squares*, which we discuss in more detail in Section 8.1.4 and Appendix B.3. The Appendix also discusses the relationship between robust statistics and non-Gaussian probabilistic models.

The generalized loss function introduced by Barron (2019) has two free parameters. The first one, α, controls how drastically outlier residuals are downweighted. The second (scale) parameter c controls the width of the quadratic well near the minimum, i.e., what range of residual values roughly corresponds to inliers. Traditionally, the choice of α, which corresponds to a variety of previously published loss functions, was determined heuristically, based on the expected shape of the outlier distribution and computational considerations (e.g., whether a convex loss was desired). The scale parameter c could be estimated using a robust measure of variance, as discussed in Appendix B.3.

In his paper, Barron (2019) discusses how both parameters can be determined at run time by maximizing the likelihood (or equivalently, minimizing the negative log-likelihood) of the given residuals, making such an algorithm self-tuning to a wide variety of noise levels and outlier distributions.



(a) (b)

Figure 4.8 *A simple surface interpolation problem: (a) nine data points of various heights 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.*

4.2 Variational methods and regularization

The theory of regularization we introduced in the previous section 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 4.8). 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 points $d(x_i, y_i)$ 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 the previous section, we attacked this problem using basis functions placed at the data points, or other heuristics such as the pull-push algorithm. While such techniques can provide reasonable solutions, they do not let us directly *quantify* and hence *optimize* the amount of smoothness in the solution, nor do they give us local control over where the solution should be discontinuous (Figure 4.8).

To do this, we use norms (measures) on function derivatives (described below) to formulate the problem and then find minimal energy solutions to these norms. Such techniques are often called *energy-based* or *optimization-based* approaches to computer vision. They are also often called *variational*, since we can use the *calculus of variations* to find the optimal solutions. Variational methods have been widely used in computer vision since the early

1980s to pose and solve a number of fundamental problems, including optical flow (Horn and Schunck 1981; Black and Anandan 1993; Brox, Bruhn *et al.* 2004; Werlberger, Pock, and Bischof 2010), segmentation (Kass, Witkin, and Terzopoulos 1988; Mumford and Shah 1989; Chan and Vese 2001), denoising (Rudin, Osher, and Fatemi 1992; Chan, Osher, and Shen 2001; Chan and Shen 2005), and multi-view stereo (Faugeras and Keriven 1998; Pons, Keriven, and Faugeras 2007; Kolev, Klodt *et al.* 2009). A more detailed list of relevant papers can be found in the Additional Reading section at the end of this chapter.

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,

$$\mathcal{E}_1 = \int f_x^2(x)\, dx \tag{4.16}$$

or perhaps integrate the squared second derivative,

$$\mathcal{E}_2 = \int f_{xx}^2(x)\, dx. \tag{4.17}$$

(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

$$\mathcal{E}_1 = \int f_x^2(x,y) + f_y^2(x,y)\, dx\, dy = \int \|\nabla f(x,y)\|^2\, dx\, dy \tag{4.18}$$

and

$$\mathcal{E}_2 = \int f_{xx}^2(x,y) + 2f_{xy}^2(x,y) + f_{yy}^2(x,y)\, dx\, dy, \tag{4.19}$$

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* (Terzopoulos 1986b).

The regularizers (smoothness functions) we have just described force the solution to be smooth and C_0 and/or C_1 continuous everywhere. In most computer vision applications, however, the fields we are trying to model or recover are only piecewise continuous, e.g.,

depth maps and optical flow fields jump at object discontinuities. Color images are even more discontinuous, since they also change appearance at albedo (surface color) and shading discontinuities.

To better model such functions, Terzopoulos (1986b) introduced *controlled-continuity splines*, where each derivative term is multiplied by a local weighting function,

$$\mathcal{E}_{\mathrm{CC}} = \int \rho(x,y)\{[1-\tau(x,y)][f_x^2(x,y)+f_y^2(x,y)] + \tau(x,y)[f_{xx}^2(x,y)+2f_{xy}^2(x,y)+f_{yy}^2(x,y)]\}\,dx\,dy. \tag{4.20}$$

Here, $\rho(x,y) \in [0,1]$ controls the *continuity* of the surface and $\tau(x,y) \in [0,1]$ controls the local *tension*, i.e., how flat the surface wants to be. Figure 4.8 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 9.3) and second-order smoothness associated with surfaces (Section 13.3.1).

In addition to the smoothness term, variational problems also require 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 $d_i = d(x_i, y_i)$,

$$\mathcal{E}_{\mathrm{D}} = \sum_i [f(x_i,y_i) - d_i]^2. \tag{4.21}$$

For a problem like noise removal, a continuous version of this measure can be used,

$$\mathcal{E}_{\mathrm{D}} = \int [f(x,y) - d(x,y)]^2\,dx\,dy. \tag{4.22}$$

To obtain a global energy that can be minimized, the two energy terms are usually added together,

$$\mathcal{E} = \mathcal{E}_{\mathrm{D}} + \lambda \mathcal{E}_{\mathrm{S}}, \tag{4.23}$$

where $\mathcal{E}_{\mathrm{S}}$ is the *smoothness penalty* ($\mathcal{E}_1$, $\mathcal{E}_2$ or some weighted blend such as $\mathcal{E}_{\mathrm{CC}}$) and λ is the *regularization parameter*, which controls the smoothness of the solution. As we saw in Section 4.1.2, good values for the regularization parameter can be estimated using techniques such as cross-validation.

4.2.1 Discrete energy minimization

In order to find the minimum of this continuous problem, the function $f(x,y)$ is usually first discretized on a regular grid.[2] The most principled way to perform this discretization is to use

[2]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 13.3.1.

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 (Terzopoulos 1983). The corresponding *discrete* smoothness energy functions become

$$\begin{aligned} E_1 = \sum_{i,j} & s_x(i,j)[f(i+1,j) - f(i,j) - g_x(i,j)]^2 \\ & + s_y(i,j)[f(i,j+1) - f(i,j) - g_y(i,j)]^2 \end{aligned} \tag{4.24}$$

and

$$\begin{aligned} E_2 = h^{-2} \sum_{i,j} & c_x(i,j)[f(i+1,j) - 2f(i,j) + f(i-1,j)]^2 \\ & + 2c_m(i,j)[f(i+1,j+1) - f(i+1,j) - f(i,j+1) + f(i,j)]^2 \\ & + c_y(i,j)[f(i,j+1) - 2f(i,j) + f(i,j-1)]^2, \end{aligned} \tag{4.25}$$

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 *et al.* 2006), 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 13.1.1), HDR tone mapping (Section 10.2.1) (Fattal, Lischinski, and Werman 2002), Poisson blending (Section 8.4.4) (Pérez, Gangnet, and Blake 2003), gradient-domain blending (Section 8.4.4) (Levin, Zomet *et al.* 2004), and Poisson surface reconstruction (Section 13.5.1) (Kazhdan, Bolitho, and Hoppe 2006; Kazhdan and Hoppe 2013). They are set to zero when just discretizing the conventional first-order smoothness functional (4.18). Note how separate smoothness and curvature terms can be imposed in the x, y, and mixed directions to produce local tears or creases (Terzopoulos 1988; Szeliski 1990a).

The two-dimensional discrete data energy is written as

$$E_{\mathrm{D}} = \sum_{i,j} c(i,j)[f(i,j) - d(i,j)]^2, \tag{4.26}$$

where the local confidence weights $c(i,j)$ 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 4.3).

The total energy of the discretized problem can now be written as a *quadratic form*

$$E = E_{\mathrm{D}} + \lambda E_{\mathrm{S}} = \mathbf{x}^T \mathbf{A}\mathbf{x} - 2\mathbf{x}^T \mathbf{b} + c, \tag{4.27}$$

where $\mathbf{x} = [f(0,0) \ldots f(m-1, n-1)]$ is called the *state vector*.[3]

The sparse symmetric positive-definite matrix $\mathbf{A}$ is called the *Hessian* since it encodes the second derivative of the energy function.[4] For the one-dimensional, first-order problem, $\mathbf{A}$ 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

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{4.28}$$

which can be done using a variety of sparse matrix techniques, such as multigrid (Briggs, Henson, and McCormick 2000) and hierarchical preconditioners (Szeliski 2006b; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013), as described in Appendix A.5 and illustrated in Figure 4.11. Using such techniques is essential to obtaining reasonable runtimes, since properly preconditioned sparse linear systems have convergence times that are *linear* in the number of pixels.

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 7.2), optical flow (Section 9.3), and shape from shading (Section 13.1) (Poggio, Torre, and Koch 1985; Horn and Brooks 1986; Terzopoulos 1986b; Bertero, Poggio, and Torre 1988; Brox, Bruhn *et al.* 2004). Poggio, Torre, and Koch (1985) also showed how the discrete energy defined by Equations (4.24–4.26) could be implemented in a resistive grid, as shown in Figure 4.9. 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 *et al.* 2006), Poisson and gradient-domain blending (Pérez, Gangnet, and Blake 2003; Levin, Zomet *et al.* 2004; Agarwala, Dontcheva *et al.*

[3] 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).

[4] In numerical analysis, $\mathbf{A}$ is called the *coefficient* matrix (Saad 2003); in finite element analysis (Bathe 2007), it is called the *stiffness* matrix.

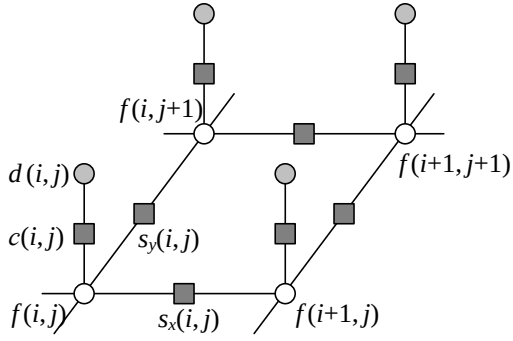


Figure 4.9 *Graphical model interpretation of first-order regularization. The white circles are the unknowns $f(i,j)$ while the dark circles are the input data $d(i,j)$. In the resistive grid interpretation, the d and f values encode input and output voltages and the black squares denote resistors whose* conductance *is set to $s_x(i,j)$, $s_y(i,j)$, and $c(i,j)$. 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 4.12).*

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, i.e., the squared derivatives in (4.16–4.19) and squared differences in (4.24–4.25), it can also be formulated using the non-quadratic *robust* penalty functions first introduced in Section 4.1.3 and discussed in more detail in Appendix B.3. For example, (4.24) can be generalized to

$$\begin{aligned} E_{1\mathrm{R}} = \sum_{i,j} & s_x(i,j)\rho(f(i+1,j) - f(i,j)) \\ & + s_y(i,j)\rho(f(i,j+1) - f(i,j)), \end{aligned} \tag{4.29}$$

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 of Blake and Zisserman (1987). Here, the norms on the data and derivatives are

clamped,

$$\rho(x) = \min(x^2, V). \tag{4.30}$$

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.)

4.2.2 Total variation

Today, many regularized problems are formulated using the L_1 ($p = 1$) norm, which is often called *total variation* (Rudin, Osher, and Fatemi 1992; Chan, Osher, and Shen 2001; Chambolle 2004; Chan and Shen 2005; Tschumperlé and Deriche 2005; Tschumperlé 2006; Cremers 2007; Kaftory, Schechner, and Zeevi 2007; Kolev, Klodt *et al.* 2009; Werlberger, Pock, and Bischof 2010). The advantage of this norm is that it tends to better preserve discontinuities, but still results in a convex problem that has a globally unique solution. Other norms, for which the *influence* (derivative) more quickly decays to zero, are presented by Black and Rangarajan (1996), Black, Sapiro *et al.* (1998), and Barron (2019) and discussed in Section 4.1.3 and 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 4.3 (4.43).

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, alternation between local non-linear subproblems and global quadratic regularization (Krishnan and Fergus 2009), or primal-dual algorithms (Chambolle and Pock 2011). Such techniques are discussed in Section 8.1.3 and Appendices A.3 and B.3.

4.2.3 Bilateral solver

In our discussion of variational methods, we have focused on energy minimization problems based on gradients and higher-order derivatives, which in the discrete setting involves

evaluating weighted errors between neighboring pixels. As we saw previously in our discussion of bilateral filtering in Section 3.3.2, we can often get better results by looking at a larger spatial neighborhood and combining pixels with similar colors or grayscale values. To extend this idea to a variational (energy minimization) setting, Barron and Poole (2016) propose replacing the usual first-order nearest-neighbor smoothness penalty (4.24) with a wider-neighborhood, bilaterally weighted version

$$E_{\mathrm{B}} = \sum_{i,j} \sum_{k,l} \hat{w}(i,j,k,l)[f(k,l) - f(i,j)]^2, \tag{4.31}$$

where

$$\hat{w}(i,j,k,l) = \frac{w(i,j,k,l)}{\sum_{m,n} w(i,j,m,n)}, \tag{4.32}$$

is the *bistochastized* (normalized) version of the *bilateral weight function* given in (3.37), which may depend on an input guide image, but not on the estimated values of f.[5]

To efficiently solve the resulting set of equations (which are much denser than nearest-neighbor versions), the authors use the same approach originally used to accelerate bilateral filtering, i.e., solving a related problem on a (spatially coarser) bilateral grid. The sequence of operations resembles those used for bilateral filtering, except that after splatting and before slicing, an iterative least squares solver is used instead of a multi-dimensional Gaussian blur. To further speed up the conjugate gradient solver, Barron and Poole (2016) use a multi-level preconditioner inspired by previous work on image-adapted preconditioners (Szeliski 2006b; Krishnan, Fattal, and Szeliski 2013).

Since its introduction, the bilateral solver has been used in a number of video processing and 3D reconstruction applications, including the stitching of binocular omnidirectional panoramic videos (Anderson, Gallup *et al.* 2016). The smartphone AR system developed by Valentin, Kowdle *et al.* (2018) extends the bilateral solver to have local *planar* models and uses a hardware-friendly real-time implementation (Mazumdar, Alaghi *et al.* 2017) to produce dense occlusion effects.

4.2.4 *Application*: Interactive colorization

A good use of edge-aware interpolation techniques is in *colorization*, i.e., manually adding colors to a "black and white" (grayscale) image. In most applications of colorization, the user draws some scribbles indicating the desired colors in certain regions (Figure 4.10a) and the system interpolates the specified chrominance (u, v) values to the whole image, which

[5] Note that in their paper, Barron and Poole (2016) use different σ_r values for the luminance and chrominance components of pixel color differences.

(a) (b) (c)

Figure 4.10 *Colorization using optimization (Levin, Lischinski, and Weiss 2004) © 2004 ACM: (a) grayscale image with some color scribbles overlaid; (b) resulting colorized image; (c) original color image from which the grayscale image and the chrominance values for the scribbles were derived. Original photograph by Rotem Weiss.*

are then re-combined with the input luminance channel to produce a final colorized image, as shown in Figure 4.10b. In the system developed by Levin, Lischinski, and Weiss (2004), the interpolation is performed using locally weighted regularization (4.24), where the local smoothness weights are inversely proportional to luminance gradients. This approach to locally weighted regularization has inspired later algorithms for high dynamic range tone mapping (Lischinski, Farbman *et al.* 2006)(Section 10.2.1, as well as other applications of the weighted least squares (WLS) formulation (Farbman, Fattal *et al.* 2008). These techniques have benefitted greatly from image-adapted regularization techniques, such as those developed in Szeliski (2006b), Krishnan and Szeliski (2011), Krishnan, Fattal, and Szeliski (2013), and Barron and Poole (2016), as shown in Figure 4.11. An alternative approach to performing the sparse chrominance interpolation based on geodesic (edge-aware) distance functions has been developed by Yatziv and Sapiro (2006). Neural networks can also be used to implement *deep priors* for image colorization (Zhang, Zhu *et al.* 2017).

4.3 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* or *generative* model, which separately models the noisy image formation (*measurement*) process, as well as assuming a statistical *prior* model over the solution space (Bishop 2006, Section 1.5.4). 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 (Kin-

Figure 4.11 *Speeding up the inhomogeneous least squares colorization solver using locally adapted hierarchical basis preconditioning (Szeliski 2006b) © 2006 ACM: (a) input gray image with color strokes overlaid; (b) solution after 20 iterations of conjugate gradient; (c) using one iteration of hierarchical basis function preconditioning; (d) using one iteration of locally adapted hierarchical basis functions.*

dermann and Snell 1980; Geman and Geman 1984; Marroquin, Mitter, and Poggio 1987; Li 1995; Szeliski, Zabih *et al.* 2008; Blake, Kohli, and Rother 2011).

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 later 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.

According to Bayes' rule (Appendix B.4), the *posterior* distribution $p(\mathbf{x}|\mathbf{y})$ over the unknowns $\mathbf{x}$ given the measurements $\mathbf{y}$ can be obtained by multiplying the measurement likelihood $p(\mathbf{y}|\mathbf{x})$ by the prior distribution $p(\mathbf{x})$ and normalizing,

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{y})}, \tag{4.33}$$

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 (4.33), we get

$$-\log p(\mathbf{x}|\mathbf{y}) = -\log p(\mathbf{y}|\mathbf{x}) - \log p(\mathbf{x}) + C, \tag{4.34}$$

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*,

$$E(\mathbf{x}, \mathbf{y}) = E_{\mathrm{D}}(\mathbf{x}, \mathbf{y}) + E_{\mathrm{P}}(\mathbf{x}). \tag{4.35}$$

(We drop the constant C because its value does not matter during energy minimization.) The first term $E_{\mathrm{D}}(\mathbf{x}, \mathbf{y})$ 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 $E_{\mathrm{P}}(\mathbf{x})$ 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, as 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 *et al.* (2009).

For the remainder of this section, we focus on Markov random fields, which are probabilistic models defined over two or three-dimensional pixel or voxel grids. Before we dive into this, however, we should mention that MRFs are just one special case of the more general family of *graphical models* (Bishop 2006, Chapter 8; Koller and Friedman 2009; Nowozin and Lampert 2011; Murphy 2012, Chapters 10, 17, 19), which have sparse interactions between variables that can be captured in a *factor graph* (Dellaert and Kaess 2017; Dellaert 2021), such as the one shown in Figure 4.12. Graphical models come in a wide variety of topologies, including chains (used for audio and speech processing), trees (often used for modeling kinematic chains in tracking people (e.g., Felzenszwalb and Huttenlocher 2005)), stars (simplified models for people; Dalal and Triggs 2005; Felzenszwalb, Girshick *et al.* 2010, and constellations (Fergus, Perona, and Zisserman 2007). Such models were widely used for part-based recognition, as discussed in Section 6.2.1. For graphs that are acyclic, efficient linear-time inference algorithms based on dynamic programming can be used.

For image processing applications, the unknowns $\mathbf{x}$ are the set of output pixels

$$\mathbf{x} = [f(0,0) \ldots f(m-1, n-1)], \tag{4.36}$$

and the data are (in the simplest case) the input pixels

$$\mathbf{y} = [d(0,0) \ldots d(m-1, n-1)] \tag{4.37}$$

as shown in Figure 4.12.

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,

$$E_{\mathrm{P}}(\mathbf{x}) = \sum_{\{(i,j),(k,l)\} \in \mathcal{N}(i,j)} V_{i,j,k,l}(f(i,j), f(k,l)), \tag{4.38}$$

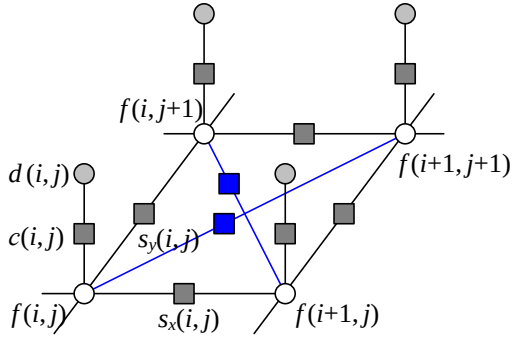


Figure 4.12 *Graphical model for an $\mathcal{N}_4$ neighborhood Markov random field. (The blue edges are added for an $\mathcal{N}_8$ neighborhood.) The white circles are the unknowns $f(i,j)$, while the dark circles are the input data $d(i,j)$. The $s_x(i,j)$ and $s_y(i,j)$ black boxes denote arbitrary* interaction potentials *between adjacent nodes in the random field, and the $c(i,j)$ denote the* data penalty *functions. The same graphical model can be used to depict a discrete version of a first-order regularization problem (Figure 4.9).*

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 4.12, which we previously used in Figure 4.9 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 $c(i,j)$ 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; Dellaert and Kaess 2017; Dellaert 2021), 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 (4.41–4.42), 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 *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,

$$E_{\mathrm{D}}(i,j) = w\delta(f(i,j), d(i,j)) \tag{4.39}$$

and the smoothness penalty to reflect the agreement between neighboring pixels

$$E_{\mathrm{P}}(i,j) = s\delta(f(i,j), f(i+1,j)) + s\delta(f(i,j), f(i,j+1)). \tag{4.40}$$

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 *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 2011). 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 *et al.* (2007) provide good algorithms for the cases when they do not, i.e., for energy functions that are not *regular* or *sub-modular*.

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 papers by Szeliski, Zabih *et al.* (2008) and Kappes, Andres *et al.* (2015), which have associated benchmarks and code at https://vision.middlebury.edu/MRF and http://hciweb2.iwr.uni-heidelberg.de/opengm.

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

$$E_{\mathrm{D}}(i,j) = c(i,j)\rho_d(f(i,j) - d(i,j)) \tag{4.41}$$

and

$$E_{\mathrm{P}}(i,j) = s_x(i,j)\rho_p(f(i,j) - f(i+1,j)) + s_y(i,j)\rho_p(f(i,j) - f(i,j+1)), \tag{4.42}$$

which are robust generalizations of the quadratic penalty terms (4.26) and (4.24), first introduced in (4.29). As before, the $c(i,j)$, $s_x(i,j)$, 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

$$\rho_p(d) = |d|^p, \quad p < 1, \tag{4.43}$$

which better encodes the distribution of gradients (mainly edges) in an image than either a quadratic or linear (total variation) penalty.[6] Levin and Weiss (2007) use such a penalty to

[6] 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*.

(a) (b) (c) (d)

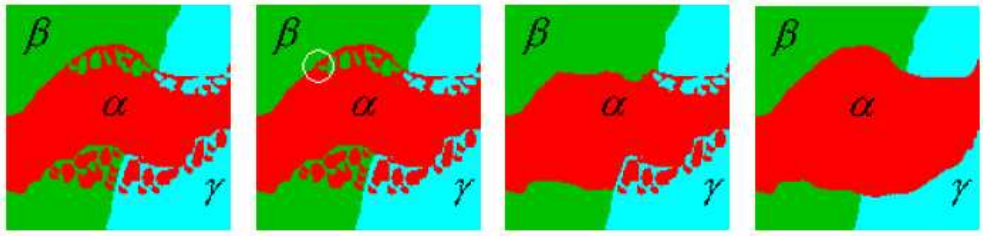
Figure 4.13 *Grayscale image denoising and inpainting: (a) original image; (b) image corrupted by noise and with missing data (black bar); (c) image restored using loopy belief propagation; (d) image restored using expansion move graph cuts. Images are from https://vision.middlebury.edu/MRF/results (Szeliski, Zabih* et al. *2008).*

separate a transmitted and reflected image (Figure 9.16) by encouraging gradients to lie in one or the other image, but not both. Levin, Fergus *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 (4.28). When the weighting functions are uniform, the GMRF becomes a special case of Wiener filtering (Section 3.4.1). 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 *et al.* 2006), natural image matting (Levin, Lischinski, and Weiss 2008), and image restoration (Tappen, Liu *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



(a) initial labeling (b) standard move (c) α-β-swap (d) α-expansion

Figure 4.14 *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.*

sub-problems to find a good solution (Figure 4.14). 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).

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 comparative survey papera on MRF optimization (Szeliski, Zabih *et al.* 2008; Kappes, Andres *et al.* 2015).

Figure 4.13 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 (4.42) for the smoothness term $E_{\mathrm{P}}(i,j)$ just shows the simplest case. In follow-on 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 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

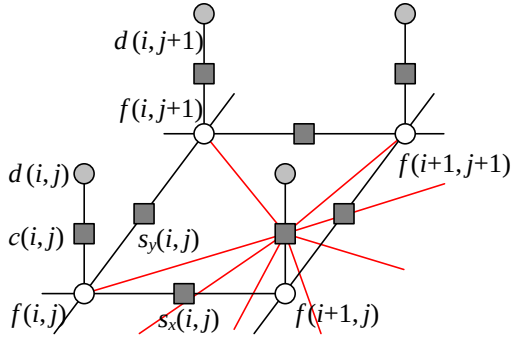


Figure 4.15 *Graphical model for a Markov random field with a more complex measurement model. The additional colored edges show how combinations of unknown values (say, in a sharp image) produce the measured values (a noisy blurred image). The resulting graphical model is still a classic MRF and is just 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.*

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 4.15. This is the commonly occurring case when trying to deblur an image. While this kind of a model is still a traditional generative Markov random field, i.e., we can in principle generate random samples from the prior distribution, 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 *et al.* 2009). Exercise 4.4 has you explore some of these issues.

Unordered labels

Another case with multi-valued labels where Markov random fields are often applied is that of *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 numerical difference between the labels assigned to forest, field, water, and pavement. In fact, the adjacencies of these various kinds of terrain

Figure 4.16 *An unordered label MRF (Agarwala, Dontcheva* 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.*

each have different likelihoods, so it makes more sense to use a prior of the form

$$E_{\mathrm{P}}(i,j) = s_x(i,j)V(l(i,j), l(i+1,j)) + s_y(i,j)V(l(i,j), l(i,j+1)), \tag{4.44}$$

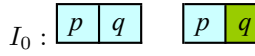
where $V(l_0, l_1)$ is a general *compatibility* or *potential* function. (Note that we have also replaced $f(i,j)$ with $l(i,j)$ to make it clearer that these are labels rather than function samples.) An alternative way to write this prior energy (Boykov, Veksler, and Zabih 2001; Szeliski, Zabih *et al.* 2008) is

$$E_{\mathrm{P}} = \sum_{(p,q)\in\mathcal{N}} V_{p,q}(l_p, l_q), \tag{4.45}$$

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 *et al.* 2004) (see Figure 4.16, which is explained in more detail in Section 8.4.2). 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 l_p next to a pixel q from image l_q. As with most MRFs, we assume that $V_{p,q}(l,l) = 0$. 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, $I_1(q) = G$.



In this case, $V_{p,q}(1,0) = 0$ (the colors agree), while $V_{p,q}(0,1) > 0$ (the colors disagree).

4.3.1 Conditional random fields

In a classic Bayesian model (4.33–4.35),

$$p(\mathbf{x}|\mathbf{y}) \propto p(\mathbf{y}|\mathbf{x})p(\mathbf{x}), \tag{4.46}$$

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.

Consider an interactive image segmentation system such as the one described in Boykov and Funka-Lea (2006). In this application, the user draws foreground and background 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 4.3.2), the MRF is modified so that the smoothness terms $s_x(x, y)$ and $s_y(x, y)$ in Figure 4.12 and (4.42) depend on the magnitude of the gradient between adjacent pixels.[7]

Since the smoothness term now depends on the data, Bayes' rule (4.46) 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

$$\begin{aligned} E(\mathbf{x}|\mathbf{y}) &= E_{\mathrm{D}}(\mathbf{x}, \mathbf{y}) + E_{\mathrm{S}}(\mathbf{x}, \mathbf{y}) \\ &= \sum_p V_p(x_p, \mathbf{y}) + \sum_{(p,q)\in\mathcal{N}} V_{p,q}(x_p, x_q, \mathbf{y}), \end{aligned} \tag{4.47}$$

using the notation introduced in (4.45). 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 4.17 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 (4.47).

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, and it is now widely used in image segmentation (Section 4.3.2) (Blake, Rother *et al.* 2004; Rother, Kolmogorov, and Blake 2004), denoising (Tappen, Liu *et al.* 2007), and object recognition (Section 6.4) (Winn and Shotton 2006; Shotton, Winn *et al.* 2009).

[7] 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).

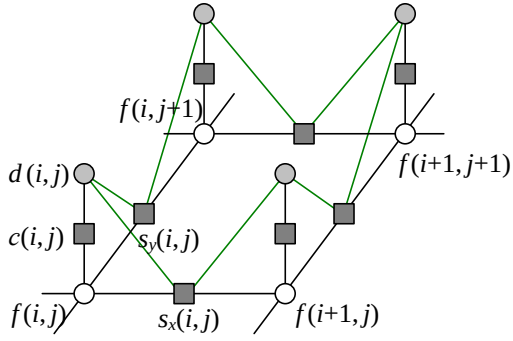


Figure 4.17 *Graphical model for a conditional random field (CRF). The additional green edges show how combinations of sensed data influence the smoothness in the underlying MRF prior model, i.e., $s_x(i,j)$ and $s_y(i,j)$ in (4.42) depend on adjacent $d(i,j)$ values. These additional links (factors) enable the smoothness to depend on the input data. However, they make sampling from this MRF more complex.*

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 12.5).

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 4.18, instead of using the classic unary MRF data term $V_p(x_p, y_p)$ shown in Figure 4.12.[8] Because such neighborhood functions can be thought of as *discriminant* 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 and binary image denoising.

Here again, one could argue that previous stereo correspondence algorithms also look at a neighborhood of input data, either explicitly, because they compute correlation measures (Criminisi, Cross *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

[8] 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*.

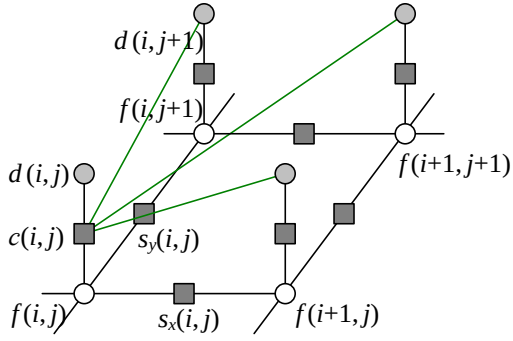


Figure 4.18 *Graphical model for a discriminative random field (DRF). The additional green edges show how combinations of sensed data, e.g., $d(i, j + 1)$, influence the data term for $f(i, j)$. The generative model is therefore more complex, i.e., we cannot just apply a simple function to the unknown variables and add noise.*

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 4.2.4) (Levin, Lischinski, and Weiss 2004), a commonly used way to transfer the continuity information from the input grayscale image to the unknown color image is to modify the local smoothness constraints. Similarly, for simultaneous segmentation and recognition (Winn and Shotton 2006; Shotton, Winn *et al.* 2009), it makes a lot of sense to permit strong color edges to increase the likelihood of semantic image label discontinuities.

In other cases, such as image denoising, the situation is more subtle. Using a non-quadratic (robust) smoothness term as in (4.42) plays a qualitatively similar role to setting the smoothness based on local gradient information in a Gaussian MRF (GMRF) (Tappen, Liu *et al.* 2007; Tanaka and Okutomi 2008). The advantage of Gaussian MRFs, when the smoothness can be correctly inferred, is that the resulting quadratic energy can be minimized

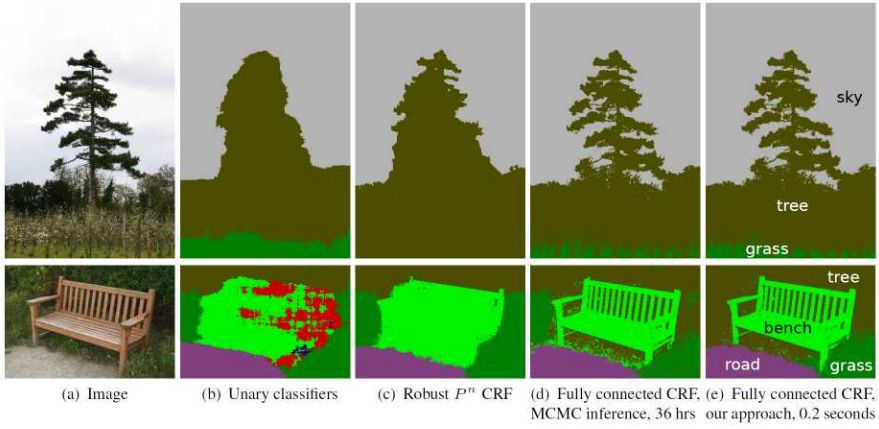


Figure 4.19 *Pixel-level classification with a fully connected CRF, from © Krähenbühl and Koltun (2011). The labels in each column describe the image or algorithm being run, which include a robust P^n CRF (Kohli, Ladický, and Torr 2009) and a very slow MCMC optimization algorithm.*

in a single step, i.e., by solving a sparse set of linear equations. 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 *et al.* (2007), and Blake, Rother *et al.* (2004), is that learning the model parameters is more principled and 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 publications by Kumar and Hebert (2006), Roth and Black (2007a), Tappen, Liu *et al.* (2007), Tappen (2007), and Li and Huttenlocher (2008).

Dense Conditional Random Fields (CRFs)

As with regular Markov random fields, conditional random fields (CRFs) are normally defined over small neighborhoods, e.g., the $\mathcal{N}_4$ neighborhood shown in Figure 4.17. However, images often contain longer-range interactions, e.g., pixels of similar colors may belong to

related classes (Figure 4.19). In order to model such longer-range interactions, Krähenbühl and Koltun (2011) introduced what they call a *fully connected CRF*, which many people now call a *dense CRF*.

As with traditional conditional random fields (4.47), their energy function consists of both unary terms and pairwise terms

$$E(\mathbf{x}|\mathbf{y}) = \sum_p V_p(x_p, \mathbf{y}) + \sum_{(p,q)} V_{p,q}(x_p, x_q, y_p, y_q), \tag{4.48}$$

where the (p, q) summation is now taken over *all* pairs of pixels, and not just adjacent ones.[9] The $\mathbf{y}$ denotes the input (guide) image over which the random field is conditioned. The pairwise interaction potentials have a restricted form

$$V_{p,q}(x_p, x_q, y_p, y_q) = \mu(x_p, x_q) \sum_{m=1}^{M} s_m w_m(p, q) \tag{4.49}$$

that is the product of a spatially invariant *label compatibility function* $\mu(x_p, x_q)$ and a sum of M Gaussian kernels of the same form (3.37) as is used in bilateral filtering and the bilateral solver. In their seminal paper, Krähenbühl and Koltun (2011) use two kernels, the first of which is an *appearance kernel* similar to (3.37) and the second is a spatial-only *smoothness kernel*.

Because of the special form of the long-range interaction potentials, which encapsulate all spatial and color similarity terms into a bilateral form, higher-dimensional filtering algorithms similar to those used in fast bilateral filters and solvers (Adams, Baek, and Davis 2010) can be used to efficiently compute a *mean field approximation* to the posterior conditional distribution (Krähenbühl and Koltun 2011). Figure 4.19 shows a comparison of their results (rightmost column) with previous approaches, including using simple unary terms, a robust CRF (Kohli, Ladický, and Torr 2009), and a very slow MCMC (Markov chain Monte Carlo) inference algorithm. As you can see, the fully connected CRF with a mean field solver produces dramatically better results in a very short time.

Since the publication of this paper, provably convergent and more efficient inference algorithms have been developed both by the original authors (Krähenbühl and Koltun 2013) and others (Vineet, Warrell, and Torr 2014; Desmaison, Bunel *et al.* 2016). Dense CRFs have seen widespread use in image segmentation problems and also as a "clean-up" stage for deep neural networks, as in the widely cited DeepLab paper by Chen, Papandreou *et al.* (2018).

[9] In practice, as with bilateral filtering and the bilateral solver, the spatial extent may be over a large but finite region.

4.3.2 *Application*: Interactive segmentation

The goal of image segmentation algorithms is 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* (Section 4.2) or as a binary Markov random field (Section 4.3).

Examples of the continuous approach include Mumford and Shah (1989), Chan and Vese (2001), Zhu and Yuille (1996), and Tabb and Ahuja (1997) along with the level set approaches discussed in Section 7.3.2. 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 earlier in this chapter, the energy corresponding to a segmentation problem can be written (c.f. Equations (4.24) and (4.35–4.42)) as

$$E(f) = \sum_{i,j} E_{\mathrm{R}}(i,j) + E_{\mathrm{P}}(i,j), \tag{4.50}$$

where the region term

$$E_{\mathrm{R}}(i,j) = C(I(i,j); R(f(i,j))) \tag{4.51}$$

is the negative log likelihood that pixel intensity (or color) $I(i,j)$ is consistent with the statistics of region $R(f(i,j))$ and the boundary term

$$E_{\mathrm{P}}(i,j) = s_x(i,j)\delta(f(i,j), f(i+1,j)) + s_y(i,j)\delta(f(i,j), f(i,j+1)) \tag{4.52}$$

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

$$C(I;\mu_k) = \|I - \mu_k\|^2. \tag{4.53}$$

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

(a) (b) (c)

Figure 4.20 *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.*

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. These pixels then become the *seeds* that tie nodes in the *S–T graph* to the source and sink labels S and T. 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 which they remain connected. While graph cuts is just one of several known techniques for MRF energy minimization, 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 4.20a)—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. Cui, Yang *et al.* (2008) use color and edge models derived from previous segmentations of similar objects to improve the local models used in GrabCut. Graph cut algorithms and other variants of Markov and conditional

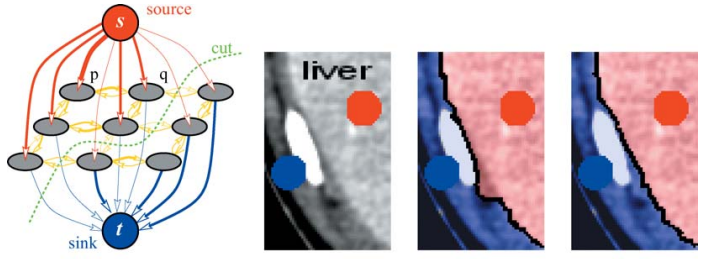


Figure 4.21 *Segmentation with a directed graph cut (Boykov and Funka-Lea 2006) © 2006 Springer: (a) directed graph; (b) image with seed points; (c) the undirected graph incorrectly continues the boundary along the bright object; (d) the directed graph correctly segments the light gray region from its darker surround.*

random fields have been applied to the *semantic segmentation* problem (Shotton, Winn *et al.* 2009; Krähenbühl and Koltun 2011), an example of which is shown in Figure 4.19 and which we study in more detail in Section 6.4.

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 4.21 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).

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 (4.50) 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 (4.24–4.27). The resulting quadratic energy function can then be solved using standard linear system solvers (4.27–4.28), although if speed is an issue, you should use multigrid 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). Couprie, Grady *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 *et al.* (2011).

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.

4.4 Additional reading

Scattered data interpolation and approximation techniques are fundamental to many different branches of applied mathematics. Some good introductory texts and articles include Amidror (2002), Wendland (2004), and Anjyo, Lewis, and Pighin (2014). These techniques are also related to geometric modeling techniques in computer graphics, which continues to be a very active research area. A nice introduction to basic spline techniques for curves and surfaces can be found in Farin (2002), while more recent approaches using subdivision surfaces are covered in Peters and Reif (2008).

Data interpolation and approximation also lie at the heart of *regression techniques*, which form the mathematical basis for most of the machine learning techniques we study in the next chapter. You can find good introductions to this topic (as well as underfitting, overfitting,

and model selection) in texts on classic machine learning (Bishop 2006; Hastie, Tibshirani, and Friedman 2009; Murphy 2012; Deisenroth, Faisal, and Ong 2020) and deep learning (Goodfellow, Bengio, and Courville 2016; Glassner 2018; Zhang, Lipton *et al.* 2021).

Robust data fitting is also central to most computer vision problems. While introduced in this chapter, it is also revisited in Appendix B.3. Classic textbooks and articles on robust fitting and statistics include Huber (1981), Hampel, Ronchetti *et al.* (1986), Black and Rangarajan (1996), Rousseeuw and Leroy (1987), and Stewart (1999). The recent paper by Barron (2019) unifies many of the commonly used robust potential functions and shows how they can be used in machine learning applications.

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 *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).

Variational formulations have been extensively used in low-level computer vision tasks, including optical flow (Horn and Schunck 1981; Nagel and Enkelmann 1986; Black and Anandan 1993; Alvarez, Weickert, and Sánchez 2000; Brox, Bruhn *et al.* 2004; Zach, Pock, and Bischof 2007a; Wedel, Cremers *et al.* 2009; Werlberger, Pock, and Bischof 2010), segmentation (Kass, Witkin, and Terzopoulos 1988; Mumford and Shah 1989; Caselles, Kimmel, and Sapiro 1997; Paragios and Deriche 2000; Chan and Vese 2001; Osher and Paragios 2003; Cremers 2007), denoising (Rudin, Osher, and Fatemi 1992), stereo (Pock, Schoenemann *et al.* 2008), multi-view stereo (Faugeras and Keriven 1998; Yezzi and Soatto 2003; Pons, Keriven, and Faugeras 2007; Labatut, Pons, and Keriven 2007; Kolev, Klodt *et al.* 2009), and scene flow (Wedel, Brox *et al.* 2011).

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 (2011). Other articles that contain nice literature reviews or experimental comparisons include Boykov and Funka-Lea (2006), Szeliski, Zabih *et al.* (2008), Kumar, Veksler, and Torr (2011), and Kappes, Andres *et al.* (2015). MRFs are just one version of the more general topic of graphical models, which is covered in several textbooks and survey, including Bishop (2006, Chapter 8), Koller and Friedman (2009), Nowozin and Lampert (2011), and Murphy (2012, Chapters 10, 17, 19)).

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 4.3 and 4.3.2, and Appendix B.5.

Continuous-energy-based (variational) approaches to interactive segmentation include Leclerc (1989), Mumford and Shah (1989), Chan and Vese (2001), Zhu and Yuille (1996), and 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 *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; Couprie, Grady *et al.* 2009). Boykov and Funka-Lea (2006) present a good survey of various energy-based techniques for binary object segmentation.

4.5 Exercises

Ex 4.1: Data fitting (scattered data interpolation). Generate some random samples from a smoothly varying function and then implement and evaluate one or more data interpolation techniques.

1. Generate a "random" 1-D or 2-D function by adding together a small number of sinusoids or Gaussians of random amplitudes and frequencies or scales.

2. Sample this function at a few dozen random locations.

3. Fit a function to these data points using one or more of the scattered data interpolation techniques described in Section 4.1.

4. Measure the fitting error between the estimated and original functions at some set of location, e.g., on a regular grid or at different random points.

5. Manually adjust any parameters your fitting algorithm may have to minimize the output sample fitting error, or use an automated technique such as cross-validation.

6. Repeat this exercise with a new set of random input sample and output sample locations. Does the optimal parameter change, and if so, by how much?

7. (Optional) Generate a piecewise-smooth test function by using different random parameters in different parts of of your image. How much more difficult does the data fitting problem become? Can you think of ways you might mitigate this?

Try to implement your algorithm in NumPy (or Matlab) using only array operations, in order to become more familiar with data-parallel programming and the linear algebra operators built into these systems. Use data visualization techniques such as those in Figures 4.3–4.6 to debug your algorithms and illustrate your results.

Ex 4.2: Graphical model optimization. Download and test out the software on the OpenGM2 library and benchmarks web site http://hciweb2.iwr.uni-heidelberg.de/opengm (Kappes, Andres *et al.* 2015). Try applying these algorithms to your own problems of interest (segmentation, de-noising, etc.). Which algorithms are more suitable for which problems? How does the quality compare to deep learning based approaches, which we study in the next chapter?

Ex 4.3: 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, or 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? See Ehrlich, Lim *et al.* (2020) for a recent paper on this topic.

2. Because the quantization occurs in the DCT transformed YCbCr space (2.116), 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 4.4: Inference in deblurring—challenging. Write down the graphical model corresponding to Figure 4.15 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?