

---

# **Springer Tracts in Advanced Robotics**

## **Volume 73**

---

Editors: Bruno Siciliano · Oussama Khatib · Frans Groen

---

Peter Corke

# **Robotics, Vision and Control**

## **Fundamental Algorithms in MATLAB®**

With 393 Images

Additional material is provided at [www.petercorke.com/RVC](http://www.petercorke.com/RVC)



**Springer**

---

**Professor Bruno Siciliano**, Dipartimento di Informatica e Sistemistica, Università di Napoli Federico II,  
Via Claudio 21, 80125 Napoli, Italy, E-mail: siciliano@unina.it

**Professor Oussama Khatib**, Artificial Intelligence Laboratory, Department of Computer Science,  
Stanford University, Stanford, CA 94305-9010, USA, E-mail: khatib@cs.stanford.edu

**Professor Frans Groen**, Department of Computer Science, Universiteit van Amsterdam, Kruislaan 403,  
1098 SJ Amsterdam, The Netherlands, E-mail: groen@science.uva.nl

## Author

### Peter Corke

Faculty of Built Environment and Engineering  
School of Engineering Systems  
Queensland University of Technology (QUT)  
Brisbane QLD 4000  
Australia  
e-mail: rvc@petercorke.com

ISBN 978-3-642-20143-1

e-ISBN 978-3-642-20144-8

DOI 10.1007/978-3-642-20144-8

Springer Tracts in Advanced Robotics

ISSN 1610-7438

Library of Congress Control Number: 2011934624

© Springer-Verlag Berlin Heidelberg 2011

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitations, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

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

Production: Armin Stasch and Scientific Publishing Services Pvt. Ltd. Chennai, India  
Typesetting and layout: Büro Stasch · Bayreuth (stasch@stasch.com)

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

---

## **Editorial Advisory Board**

Oliver Brock, TU Berlin, Germany  
Herman Bruyninckx, KU Leuven, Belgium  
Raja Chatila, LAAS, France  
Henrik Christensen, Georgia Tech, USA  
Peter Corke, Queensland Univ. Technology, Australia  
Paolo Dario, Scuola S. Anna Pisa, Italy  
Rüdiger Dillmann, Univ. Karlsruhe, Germany  
Ken Goldberg, UC Berkeley, USA  
John Hollerbach, Univ. Utah, USA  
Makoto Kaneko, Osaka Univ., Japan  
Lydia Kavraki, Rice Univ., USA  
Vijay Kumar, Univ. Pennsylvania, USA  
Sukhan Lee, Sungkyunkwan Univ., Korea  
Frank Park, Seoul National Univ., Korea  
Tim Salcudean, Univ. British Columbia, Canada  
Roland Siegwart, ETH Zurich, Switzerland  
Gaurav Sukhatme, Univ. Southern California, USA  
Sebastian Thrun, Stanford Univ., USA  
Yangsheng Xu, Chinese Univ. Hong Kong, PRC  
Shin'ichi Yuta, Tsukuba Univ., Japan

STAR (Springer Tracts in Advanced Robotics) has been promoted under the auspices of EURON (European Robotics Research Network)



*To my family Phillipa, Lucy and Madeline for their indulgence and support;  
my parents Margaret and David for kindling my curiosity;  
and to Lou Paul who planted the seed that became this book.*

---

# Foreword

Once upon a time, a very thick document of a dissertation from a faraway land came to me for evaluation. *Visual robot control* was the thesis theme and *Peter Corke* was its author. Here, I am reminded of an excerpt of my comments, which reads, *this is a masterful document, a quality of thesis one would like all of one's students to strive for, knowing very few could attain – very well considered and executed.*

The connection between robotics and vision has been, for over two decades, the central thread of Peter Corke's productive investigations and successful developments and implementations. This rare experience is bearing fruit in his new book on *Robotics, Vision, and Control*. In its melding of theory and application, this new book has considerably benefited from the author's unique mix of academic and real-world application influences through his many years of work in robotic mining, flying, underwater, and field robotics.

There have been numerous textbooks in robotics and vision, but few have reached the level of integration, analysis, dissection, and practical illustrations evidenced in this book. The discussion is thorough, the narrative is remarkably informative and accessible, and the overall impression is of a significant contribution for researchers and future investigators in our field. Most every element that could be considered as relevant to the task seems to have been analyzed and incorporated, and the effective use of Toolbox software echoes this thoroughness.

The reader is taken on a realistic walkthrough the fundamentals of mobile robots, navigation, localization, manipulator-arm kinematics, dynamics, and joint-level control, as well as camera modeling, image processing, feature extraction, and multi-view geometry. These areas are finally brought together through extensive discussion of visual servo system. In the process, the author provides insights into how complex problems can be decomposed and solved using powerful numerical tools and effective software.

The *Springer Tracts in Advanced Robotics (STAR)* is devoted to bringing to the research community the latest advances in the robotics field on the basis of their significance and quality. Through a wide and timely dissemination of critical research developments in robotics, our objective with this series is to promote more exchanges and collaborations among the researchers in the community and contribute to further advancements in this rapidly growing field.

Peter Corke brings a great addition to our STAR series with an authoritative book, reaching across fields, thoughtfully conceived and brilliantly accomplished.

Oussama Khatib  
Stanford, California  
July 2011

# Preface

*Tell me and I will forget.  
Show me and I will remember.  
Involve me and I will understand.*  
Chinese proverb

The practice of robotics and machine vision involves the application of computational algorithms to data. The data comes from sensors measuring the velocity of a wheel, the angle of a robot arm's joint or the intensities of millions of pixels that comprise an image of the world that the robot is observing. For many robotic applications the amount of data that needs to be processed, in real-time, is massive. For vision it can be of the order of tens to hundreds of megabytes per second.

Progress in robots and machine vision has been, and continues to be, driven by more effective ways to process data. This is achieved through new and more efficient algorithms, and the dramatic increase in computational power that follows Moore's law. When I started in robotics and vision, in the mid 1980s, the IBM PC had been recently released – it had a 4.77 MHz 16-bit microprocessor and 16 kbytes (expandable to 256 k) of memory. Over the intervening 25 years computing power has doubled 16 times which is an increase by a factor of 65 000. In the late 1980s systems capable of real-time image processing were large 19 inch racks of equipment such as shown in Fig. 0.1. Today there is far more computing in just a small corner of a modern microprocessor chip.

Over the fairly recent history of robotics and machine vision a very large body of algorithms has been developed – a significant, tangible, and collective achievement of the research community. However its sheer size and complexity presents a barrier to somebody entering the field. Given the many algorithms from which to choose the obvious question is:

*What is the right algorithm for this particular problem?*

One strategy would be to try a few different algorithms and see which works best for the problem at hand but this raises the next question:

*How can I evaluate algorithm X on my own data without spending days coding and debugging it from the original research papers?*



**Fig. 0.1.**

Once upon a time a lot of equipment was needed to do vision-based robot control. The author with a large rack full of image processing and robot control equipment (1992)

Two developments come to our aid. The first is the availability of general purpose mathematical software which it makes it easy to prototype algorithms. There are commercial packages such as MATLAB®, Mathematica and MathCad, and open source projects include SciLab, Octave, and PyLab. All these tools deal naturally and effortlessly with vectors and matrices, can create complex and beautiful graphics, and can be used interactively or as a programming environment. The second is the open-source movement. Many algorithms developed by researchers are available in open-source form. They might be coded in one of the general purpose mathematical languages just mentioned, or written in a mainstream language like C, C++ or Java.

Respectively the trademarks of  
The Mathworks Inc., Wolfram  
Research, and PTC.

For more than fifteen years I have been part of the open-source community and maintained two open-source MATLAB® Toolboxes: one for robotics and one for machine vision. They date back to my own PhD work and have evolved since then, growing features and tracking changes to the MATLAB® language (which have been significant over that period). The Robotics Toolbox has also been translated into a number of different languages such as Python, SciLab and LabView.

The Toolboxes have some important virtues. Firstly, they have been around for a long time and used by many people for many different problems so the code is entitled to some level of trust. The Toolbox provides a “gold standard” with which to compare new algorithms or even the same algorithms coded in new languages or executing in new environments.

Secondly, they allow the user to work with real problems, not trivial examples. For real robots, those with more than two links, or real images with millions of pixels the computation is beyond unaided human ability. Thirdly, they allow us to gain insight which is otherwise lost in the complexity. We can rapidly and easily experiment, play *what if* games, and depict the results graphically using MATLAB®’s powerful display tools such as 2D and 3D graphs and images.

Fourthly, the Toolbox code makes many common algorithms tangible and accessible. You can read the code, you can apply it to your own problems, and you can extend it or rewrite it. At the very least it gives you a headstart.

The Toolboxes were always accompanied by short tutorials as well as reference material. Over the years many people have urged me to turn this into a book and finally it has happened! The purpose of this book is to expand on the tutorial material provided with the Toolboxes, add many more examples, and to weave it into a narrative that covers robotics and computer vision separately and together. I want to show how complex problems can be decomposed and solved using just a few simple lines of code.

By inclination I am a *hands on* person. I like to program and I like to analyze data, so it has always seemed natural to me to build tools to solve problems in robotics and vision. The topics covered in this book are based on my own interests but also guided by real problems that I observed over many years as a practitioner of both robotics and computer vision. I hope that by the end of this book you will share my enthusiasm for these topics.

I was particularly motivated to present a solid introduction to machine vision for roboticists. The treatment of vision in robotics textbooks tends to concentrate on simple binary vision techniques. In the book we will cover a broad range of topics including color vision, advanced segmentation techniques such as maximally stable extremal regions and graphcuts, image warping, stereo vision, motion estimation and image retrieval. We also cover non-perspective imaging using fisheye lenses and catadioptric optics. These topics are growing in importance for robotics but are not commonly covered. Vision is a powerful sensor, and roboticists should have a solid grounding in modern fundamentals. The last part of the book shows how vision can be used as the primary sensor for robot control.

This book is unlike other text books, and deliberately so. Firstly, there are already a number of excellent text books that cover robotics and computer vision separately and in depth, but few that cover both in an integrated fashion. Achieving this integration is a principal goal of this book.

Secondly, software is a first-class citizen in this book. Software is a tangible instantiation of the algorithms described – it can be read and it can be pulled apart, modified and put back together again. There are a number of classic books that use software in this illustrative fashion for problem solving. In this respect I've been influenced by books such as *LaTeX: A document preparation system* (Lamport 1994), *Numerical Recipes in C* (Press et al. 2007), *The Little Lisper* (Friedman et al. 1987) and *Structure and Interpretation of Classical Mechanics* (Sussman et al. 2001). The many examples in this book illustrate how the Toolbox software can be used and generally provide *instant gratification* in just a couple of lines of MATLAB® code.

Thirdly, building the book around MATLAB® and the Toolboxes means that we are able to tackle more realistic and more complex problems than other books.

The emphasis on software and examples does not mean that rigour and theory are unimportant, they are very important, but this book provides a complementary approach. It is best read in conjunction with standard texts which provide rigour and theoretical nourishment. The end of each chapter has a section on further reading and provides pointers to relevant textbooks and key papers.

Writing this book provided a good opportunity to look critically at the Toolboxes and to revise and extend the code. In particular I've made much greater use of the ever-evolving object-oriented features of MATLAB® to simplify the user interface and to reduce the number of separate files within the Toolboxes.

The rewrite also made me look more widely at complementary open-source code. There is a lot of great code out there, particularly on the computer vision side, so rather than reinvent some wheels I've tried to integrate the best code I could find for particular algorithms. The complication is that every author has their own naming conventions and preferences about data organization, from simple matters like the use of row or column vectors to more complex issues involving structures – arrays of structures or structures of arrays. My solution has been, as much as possible, to not modify any of these packages but to encapsulate them with light weight wrappers, particularly as classes.

I am grateful to the following for code that has been either incorporated into the Toolboxes or which has been wrapped into the Toolboxes. Robotics Toolbox contributions include: mobile robot localization and mapping by Paul Newman at Oxford and a quadcopter simulator by Paul Pounds at Yale. Machine Vision Toolbox contributions include: RANSAC code by Peter Kovesi; pose estimation by Francesco Moreno-Noguer, Vincent Lepetit, Pascal Fua at the CVLab-EPFL; color space conversions by Pascal Getreuer; numerical routines for geometric vision by various members of the Visual Geometry Group at Oxford (from the web site of the Hartley and Zisserman book; Hartley and Zisserman 2003); the *k*-means and MSER algorithms by Andrea Vedaldi and Brian Fulkerson; the graph-based image segmentation software by Pedro Felzenszwalb; and the SURF feature detector by Dirk-Jan Kroon at U. Twente. The Camera Calibration Toolbox by Jean-Yves Bouguet is used unmodified.

Along the way I got interested in the mathematicians, physicists and engineers whose work, hundreds of years later, is critical to the science of robotic and vision today. Some of their names have become adjectives like Coriolis, Gaussian, Laplacian or Cartesian; nouns like Jacobian, or units like Newton and Coulomb. They are interesting characters from a distant era when science was a hobby and their day jobs were as doctors, alchemists, gamblers, astrologers, philosophers or mercenaries. In order to know whose shoulders we are standing on I have included small vignettes about the lives of these people – a smattering of history as a backstory.

In my own career I have had the good fortune to work with many wonderful people who have inspired and guided me. Long ago at the University of Melbourne John Anderson fired my interest in control and Graham Holmes encouraged me to “think before I code” – excellent advice that I sometimes heed. Early on I spent a life-direction-changing ten months working with Richard (Lou) Paul in the GRASP laboratory at the University of Pennsylvania in the period 1988–1989. The genesis of the Toolboxes was my

PhD research and my advisors Malcolm Good (University of Melbourne) and Paul Dunn (CSIRO) asked me good questions and guided my research. Laszlo Nemes provided sage advice about life and the ways of organizations and encouraged me to publish more and to open-source my software. Much of my career was spent at CSIRO where I had the privilege and opportunity to work on a diverse range of real robotics projects and to work with a truly talented set of colleagues and friends. Mid book I joined Queensland University of Technology which has generously made time available to me to complete the project. My former students Jasmine Banks, Kane Usher, Paul Pounds and Peter Hansen taught me a lot of about stereo, non-holonomy, quad-copters and wide-angle vision respectively.

I would like to thank Paul Newman for generously hosting me several times at Oxford where significant sections of the book were written, and Daniela Rus for hosting me at MIT for a burst of intense writing that was the first complete book draft. Daniela, Paul and Cédric Pradalier made constructive suggestions and comments on early drafts of the material. I would also like to thank the MathWorks, the publishers of MATLAB® for the support they offered me through their author program. Springer have been enormously supportive of the whole project and a pleasure to work with. I would specially like to thank Thomas Ditzinger, my editor, and Armin Stasch for the layout and typesetting which has transformed my manuscript into a book.

I have tried my hardest to eliminate errors but inevitably some will remain. Please email me bug reports as well as suggestions for improvements and extensions.

Finally, it can't be easy living with a writer – there are books and websites devoted to this topic. My deepest thanks are to Phillipa for supporting and encouraging me in the endeavour and living with “the book” for so long and in so many different places.

*Peter Corke*  
Brisbane, Queensland  
June 2011

---

# Contents

<b>1</b>	<b>Introduction</b>	1
1.1	About the Book	6
1.1.1	The MATLAB Software	7
1.1.2	Audience and Prerequisites	8
1.1.3	Notation and Conventions	9
1.1.4	How to Use the Book	9
1.1.5	Teaching with the Book	10
1.1.6	Outline	10
<b>Part I Foundations</b>		13
<b>2</b>	<b>Representing Position and Orientation</b>	15
2.1	Representing Pose in 2-Dimensions	19
2.2	Representing Pose in 3-Dimensions	24
2.2.1	Representing Orientation in 3-Dimensions	25
2.2.2	Combining Translation and Orientation	37
2.3	Wrapping Up	39
	Further Reading	40
	Exercises	41
<b>3</b>	<b>Time and Motion</b>	43
3.1	Trajectories	43
3.1.1	Smooth One-Dimensional Trajectories	43
3.1.2	Multi-Dimensional Case	46
3.1.3	Multi-Segment Trajectories	46
3.1.4	Interpolation of Orientation in 3D	48
3.1.5	Cartesian Motion	49
3.2	Time Varying Coordinate Frames	51
3.2.1	Rotating Coordinate Frame	51
3.2.2	Incremental Motion	52
3.2.3	Inertial Navigation Systems	53
3.3	Wrapping Up	56
	Further Reading	56
	Exercises	56
<b>Part II Mobile Robots</b>		59
<b>4</b>	<b>Mobile Robot Vehicles</b>	65
4.1	Mobility	65
4.2	Car-like Mobile Robots	67
4.2.1	Moving to a Point	71
4.2.2	Following a Line	72
4.2.3	Following a Path	74
4.2.4	Moving to a Pose	75

4.3	Flying Robots .....	78
4.4	Wrapping Up .....	84
	Further Reading .....	84
	Exercises .....	85
<b>5</b>	<b>Navigation .....</b>	<b>87</b>
5.1	Reactive Navigation .....	88
5.1.1	Braitenberg Vehicles .....	88
5.1.2	Simple Automata .....	90
5.2	Map-Based Planning .....	91
5.2.1	Distance Transform .....	93
5.2.2	D* .....	95
5.2.3	Voronoi Roadmap Method .....	97
5.2.4	Probabilistic Roadmap Method .....	99
5.2.5	RRT .....	102
5.3	Wrapping Up .....	104
	Further Reading .....	105
	Exercises .....	106
<b>6</b>	<b>Localization .....</b>	<b>107</b>
6.1	Dead Reckoning .....	111
6.1.1	Modeling the Vehicle .....	111
6.1.2	Estimating Pose .....	113
6.2	Using a Map .....	116
6.3	Creating a Map .....	120
6.4	Localization and Mapping .....	123
6.5	Monte-Carlo Localization .....	125
6.6	Wrapping Up .....	128
	Further Reading .....	129
	Notes on Toolbox Implementation .....	130
	Exercises .....	130
<b>Part III Arm-Type Robots .....</b>		<b>133</b>
<b>7</b>	<b>Robot Arm Kinematics .....</b>	<b>137</b>
7.1	Describing a Robot Arm .....	137
7.2	Forward Kinematics .....	140
7.2.1	A 2-Link Robot .....	141
7.2.2	A 6-Axis Robot .....	143
7.3	Inverse Kinematics .....	146
7.3.1	Closed-Form Solution .....	146
7.3.2	Numerical Solution .....	149
7.3.3	Under-Actuated Manipulator .....	149
7.3.4	Redundant Manipulator .....	150
7.4	Trajectories .....	152
7.4.1	Joint-Space Motion .....	153
7.4.2	Cartesian Motion .....	155
7.4.3	Motion through a Singularity .....	156
7.4.4	Configuration Change .....	157
7.5	Advanced Topics .....	158
7.5.1	Joint Angle Offsets .....	158
7.5.2	Determining Denavit-Hartenberg Parameters .....	159
7.5.3	Modified Denavit-Hartenberg Notation .....	160
7.6	Application: Drawing .....	162

---

7.7	Application: a Simple Walking Robot .....	163
7.7.1	Kinematics .....	163
7.7.2	Motion of One Leg .....	165
7.7.3	Motion of Four Legs .....	166
7.8	Wrapping Up .....	167
	Further Reading .....	168
	The plot Method .....	168
	Exercises .....	170
<b>8</b>	<b>Velocity Relationships</b> .....	171
8.1	Manipulator Jacobian .....	171
8.1.1	Transforming Velocities between Coordinate Frames .....	174
8.1.2	Jacobian in the End-Effector Coordinate Frame .....	175
8.1.3	Analytical Jacobian .....	176
8.1.4	Jacobian Condition and Manipulability .....	177
8.2	Resolved-Rate Motion Control .....	180
8.2.1	Jacobian Singularity .....	182
8.2.2	Jacobian for under-Actuated Robot .....	183
8.2.3	Jacobian for over-Actuated Robot .....	184
8.3	Force Relationships .....	186
8.3.1	Transforming Wrenches between Frames .....	186
8.3.2	Transforming Wrenches to Joint Space .....	186
8.4	Inverse Kinematics: a General Numerical Approach .....	187
8.5	Wrapping Up .....	188
	Further Reading .....	189
	Exercises .....	189
<b>9</b>	<b>Dynamics and Control</b> .....	191
9.1	Equations of Motion .....	191
9.1.1	Gravity Term .....	193
9.1.2	Inertia Matrix .....	195
9.1.3	Coriolis Matrix .....	196
9.1.4	Effect of Payload .....	197
9.1.5	Base Force .....	198
9.1.6	Dynamic Manipulability .....	198
9.2	Drive Train .....	200
9.2.1	Friction .....	201
9.3	Forward Dynamics .....	202
9.4	Manipulator Joint Control .....	204
9.4.1	Actuators .....	204
9.4.2	Independent Joint Control .....	204
9.4.3	Rigid-Body Dynamics Compensation .....	211
9.4.4	Flexible Transmission .....	213
9.5	Wrapping Up .....	215
	Further Reading .....	216
	Exercises .....	217
	<b>Part IV Computer Vision</b> .....	219
<b>10</b>	<b>Light and Color</b> .....	223
10.1	Spectral Representation of Light .....	223
10.1.1	Absorption .....	225
10.1.2	Reflection .....	226

10.2	Color .....	227
10.2.1	Reproducing Colors .....	230
10.2.2	Chromaticity Space .....	233
10.2.3	Color Names .....	236
10.2.4	Other Color Spaces .....	236
10.2.5	Transforming between Different Primaries .....	238
10.2.6	What Is White? .....	240
10.3	Advanced Topics .....	240
10.3.1	Color Constancy .....	241
10.3.2	White Balancing .....	241
10.3.3	Color Change Due to Absorption .....	242
10.3.4	Gamma .....	243
10.3.5	Application: Color Image .....	245
10.4	Wrapping Up .....	247
	Further Reading .....	248
	Data Sources .....	249
	Exercises .....	249
<b>11</b>	<b>Image Formation .....</b>	<b>251</b>
11.1	Perspective Transform .....	251
11.1.1	Lens Distortion .....	261
11.2	Camera Calibration .....	262
11.2.1	Homogeneous Transformation Approach .....	262
11.2.2	Decomposing the Camera Calibration Matrix .....	264
11.2.3	Pose Estimation .....	266
11.2.4	Camera Calibration Toolbox .....	266
11.3	Non-Perspective Imaging Models .....	269
11.3.1	Fisheye Lens Camera .....	270
11.3.2	Catadioptric Camera .....	272
11.3.3	Spherical Camera .....	274
11.4	Unified Imaging .....	275
11.4.1	Mapping Wide-Angle Images to the Sphere .....	276
11.4.2	Synthetic Perspective Images .....	278
11.5	Wrapping Up .....	280
	Further Reading .....	280
	Camera Classes .....	282
	Exercises .....	283
<b>12</b>	<b>Image Processing .....</b>	<b>285</b>
12.1	Obtaining an Image .....	285
12.1.1	Images from Files .....	285
12.1.2	Images from an Attached Camera .....	289
12.1.3	Images from a Movie File .....	289
12.1.4	Images from the Web .....	290
12.1.5	Images from Code .....	291
12.2	Monadic Operations .....	293
12.3	Diadic Operations .....	296
12.4	Spatial Operations .....	299
12.4.1	Convolution .....	300
12.4.2	Template Matching .....	311
12.4.3	Non-Linear Operations .....	316
12.5	Mathematical Morphology .....	317
12.5.1	Noise Removal .....	321

---

12.5.2	Boundary Detection .....	322
12.5.3	Hit and Miss Transform .....	322
12.6	Shape Changing .....	324
12.6.1	Cropping .....	324
12.6.2	Image Resizing .....	324
12.6.3	Image Pyramids .....	326
12.6.4	Image Warping .....	327
12.7	Wrapping Up .....	330
	Further Reading .....	330
	Sources of Image Data .....	332
	MATLAB® Software Tools .....	332
	General Software Tools .....	332
	Exercises .....	333
<b>13</b>	<b>Image Feature Extraction</b> .....	<b>335</b>
13.1	Region Features .....	337
13.1.1	Classification .....	337
13.1.2	Representation .....	346
13.1.3	Description .....	350
13.1.4	Recap .....	360
13.2	Line Features .....	361
13.3	Point Features .....	365
13.3.1	Classical Corner Detectors .....	366
13.3.2	Scale-Space Corner Detectors .....	371
13.4	Wrapping Up .....	376
	Further Reading .....	376
	Exercises .....	378
<b>14</b>	<b>Using Multiple Images</b> .....	<b>381</b>
14.1	Feature Correspondence .....	382
14.2	Geometry of Multiple Views .....	386
14.2.1	The Fundamental Matrix .....	388
14.2.2	The Essential Matrix .....	390
14.2.3	Estimating the Fundamental Matrix .....	391
14.2.4	Planar Homography .....	396
14.3	Stereo Vision .....	401
14.3.1	Sparse Stereo .....	401
14.3.2	Dense Stereo Matching .....	405
14.3.3	Peak Refinement .....	412
14.3.4	Cleaning up and Reconstruction .....	413
14.3.5	3D Texture Mapped Display .....	415
14.3.6	Anaglyphs .....	416
14.3.7	Image Rectification .....	417
14.3.8	Plane Fitting .....	419
14.3.9	Matching Sets of 3D Points .....	420
14.4	Structure and Motion .....	422
14.5	Application: Perspective Correction .....	428
14.6	Application: Mosaicing .....	431
14.7	Application: Image Matching and Retrieval .....	433
14.8	Application: Image Sequence Processing .....	439
14.9	Wrapping Up .....	442
	Further Reading .....	442
	Resources .....	445
	Exercises .....	446

<b>Part V</b>	<b>Robotics, Vision and Control</b>	451
<b>15</b>	<b>Vision-Based Control</b>	455
15.1	Position-Based Visual Servoing	456
15.2	Image-Based Visual Servoing	459
15.2.1	Camera and Image Motion	460
15.2.2	Controlling Feature Motion	464
15.2.3	Depth	469
15.2.4	Performance Issues	471
15.3	Using Other Image Features	473
15.3.1	Line Features	473
15.3.2	Circle Features	474
15.4	Wrapping Up	476
	Further Reading	476
	Exercises	478
<b>16</b>	<b>Advanced Visual Servoing</b>	481
16.1	XY/Z-Partitioned IBVS	481
16.2	IBVS Using Polar Coordinates	484
16.3	IBVS for a Spherical Camera	486
16.4	Application: Arm-Type Robot	488
16.5	Application: Mobile Robot	489
16.5.1	Holonomic Mobile Robot	489
16.5.2	Non-Holonomic Mobile Robot	491
16.6	Application: Aerial Robot	492
16.7	Wrapping Up	494
	Further Reading	494
	Exercises	495
	<b>Appendices</b>	497
<b>A</b>	<b>Installing the Toolboxes</b>	499
<b>B</b>	<b>Simulink®</b>	501
<b>C</b>	<b>MATLAB® Objects</b>	505
<b>D</b>	<b>Linear Algebra Refresher</b>	511
<b>E</b>	<b>Ellipses</b>	517
<b>F</b>	<b>Gaussian Random Variables</b>	523
<b>G</b>	<b>Jacobians</b>	527
<b>H</b>	<b>Kalman Filter</b>	529
<b>I</b>	<b>Homogeneous Coordinates</b>	533
<b>J</b>	<b>Graphs</b>	535
<b>K</b>	<b>Peak Finding</b>	539
	<b>Bibliography</b>	543
	<b>Index</b>	553
	Index of People	553
	Index of Functions, Classes and Methods	554
	General Index	558

# Nomenclature

The notation used in robotics and computer vision varies considerably from book to book. The symbols used in this book, and their units where appropriate, are listed below. Some symbols have multiple meanings and their context must be used to disambiguate them.

The elements of a vector  $\mathbf{x}[i]$  or a matrix  $\mathbf{x}[i,j]$  are indicated by square brackets. The elements of a time series  $\mathbf{x}\langle k \rangle$  are indicated by angle brackets.

Symbol	Description	Unit
$\hat{\mathbf{x}}$	an estimate of $\mathbf{x}$	
$\bar{\mathbf{x}}$	mean of $\mathbf{x}$ or relative value	
$\mathbf{x}^*$	desired value of $\mathbf{x}$	
$\mathbf{v}$	a vector	
$\hat{\mathbf{v}}$	a unit-vector parallel to $\mathbf{v}$	
$ \mathbf{v} $	scalar norm or length of the vector $\mathbf{v}$	
$ \mathring{\mathbf{q}} $	scalar norm of the quaternion $\mathring{\mathbf{q}}$	
$\tilde{\mathbf{v}}$	homogeneous representation of vector $\mathbf{v}$	
$v_x$	a component of a vector	
$\mathbf{v}_1 \cdot \mathbf{v}_2$	dot, or inner, product, also $\mathbf{v}_1^T \mathbf{v}_2$	
$\mathbf{v}_1 \times \mathbf{v}_2$	cross, or vector, product	
$A$	a matrix	
$A^{-1}$	inverse of $A$	
$A^+$	pseudo-inverse of $A$	
$A^T$	transpose of $A$	
$A^{-T}$	transpose of inverse $A$	
$A_{i,j}$	the element $(i,j)$ of $A$	
$A[i,j]$	the element $(i,j)$ of $A$	
$F(x)$	a function of $x$	
$F_x(x)$	the derivative $\partial F / \partial x$	
$B$	viscous friction coefficient	$\text{N m s rad}^{-1}$
$\mathcal{C}$	configuration space of a robot	
$C$	camera matrix, $C \in \mathbb{R}^{3 \times 4}$	
$C(\mathbf{q}, \dot{\mathbf{q}})$	manipulator centripetal and Coriolis term	$\text{kg m}^2 \text{ s}^{-1}$
$\mathbb{C}$	the set of complex numbers	
$\mathcal{D}(\cdot)$	manipulator dynamics function: $Q, q, \dot{q} \mapsto \ddot{q}$	
$\Delta(\xi)$	maps incremental pose change to differential motion: $SE(3) \mapsto \mathbb{R}^6$	
$\Delta^{-1}(\delta)$	maps differential motion to incremental pose change: $\mathbb{R}^6 \mapsto SE(3)$	
$E$	illuminance (lux)	lx
$f$	focal length	m
$\mathbf{f}$	force	N
$\mathbf{f}$	vector of image features	

Symbol	Description	Unit
$F(\dot{q})$	friction torque	N m
$\phi$	luminous flux (lumens)	lm
$g$	wrench, a vector of forces and moments ( $f_x, f_y, f_z, m_x, m_y, m_z$ )	N, Nm
$G(q)$	manipulator gravity loading term	N m
$\gamma$	robot steering angle	rad
$\Gamma$	3-angle representation of rotation, $\Gamma \in \mathbb{R}^3$	rad
$\Gamma$	body torque $\Gamma \in \mathbb{R}^3$	N m
$I_{n \times n}$	$n \times n$ identity matrix	
$J$	inertia	$\text{kg m}^2$
$J$	inertia tensor, $3 \times 3$ matrix	$\text{kg m}^2$
$J$	Jacobian matrix	
${}^A J_B$	Jacobian transforming velocities in frame $A$ to frame $B$	
$k, K$	constant	
$K$	camera calibration matrix	
$K_i$	amplifier gain (transconductance)	$\text{A V}^{-1}$
$K_m$	motor torque constant	$\text{N m A}^{-1}$
$\mathcal{K}(\cdot)$	forward kinematics	
$\mathcal{K}^{-1}(\cdot)$	inverse kinematics	
$L$	luminance (nit)	nt
$\lambda$	wavelength	m
$\lambda$	an eigenvalue	
$m_i$	mass of link $i$	kg
$M(q)$	manipulator inertia matrix	$\text{kg m}^2$
$p$	an image plane point	
$P$	a world point	
$\mathbb{P}^2$	the projective space of all 2-D points, a 3-tuple	
$\mathbb{P}^3$	the projective space of all 3-D points, a 4-tuple	
$\mathcal{P}(\cdot)$	projection function: $\mathbb{R}^3 \mapsto \mathbb{R}^2$	
$\mathring{q}$	quaternion	
$\mathring{q}(v)$	pure quaternion of vector $v$	
$q$	configuration, generalized coordinates	m, rad
$Q$	generalized force	N, Nm
$\rho_w, \rho_h$	pixel width and height	m
$R$	an orthonormal rotation matrix, $R \in SO(2)$ or $SO(3)$	
$\mathbb{R}$	set of real numbers	
$\mathbb{R}^2$	the space of all 2-D points	
$\mathbb{R}^3$	the space of all 3-D points	
$s$	Laplace transform operator	
$\mathbb{S}$	set of all angles in the circle $[0, 2\pi)$	
$SE(n)$	special Euclidean group (all poses) in $n$ dimensions	
$SO(n)$	special orthogonal group, the set of all orientations in $n$ dimensions	
$S(v)$	skew symmetric matrix of $v$	
$s_i$	COM of link $i$ with respect to the link $i$ coordinate frame	m
$S_i$	first moment of link $i$ . $S_i = m_i s_i$	$\text{kg m}$
$\sigma$	standard deviation	
$\sigma$	robot joint type, $\sigma = 0$ for revolute and $\sigma = 1$ for prismatic	
$t$	time	s

Symbol	Description	Unit
$T$	sample interval	s
$T$	temperature	K
$T$	optical transmission	
$T$	homogeneous transformation, $T \in SE(2)$ or $SE(3)$	
${}^A T_B$	homogeneous transform representing frame $\{B\}$ with respect to frame $\{A\}$ . If $A$ is not given then assumed relative to world coordinate frame 0. Note that ${}^A T_B = ({}^B T_A)^{-1}$	
$\theta$	angle	rad
$\boldsymbol{\theta}$	vector of angles, generally robot joint angles	rad
$\theta_r, \theta_p, \theta_y$	roll pitch yaw angles	rad
$\tau$	torque	N m
$\tau_C$	Coulomb friction torque	N m
$u, v$	camera image plane coordinates	pixels
$\bar{u}, \bar{v}$	normalized image plane coordinates, relative to the principal point	m
$u_0, v_0$	coordinates of the principal point	pixels
$v$	velocity	$m s^{-1}$
$\boldsymbol{v}$	velocity vector	$m s^{-1}$
$\nu$	innovation	
$\boldsymbol{\nu}$	velocity screw, $\boldsymbol{\nu} \in \mathbb{R}^6$ , $\boldsymbol{\nu} = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$	
$\omega$	rotational rate	$rad s^{-1}$
$\boldsymbol{\omega}$	angular velocity vector	$rad s^{-1}$
$X, Y, Z$	Cartesian coordinates	
$\xi$	abstract representation of 3-dimensional Cartesian pose (pronounced ksi)	
${}^A \xi_B$	abstract representation of 3-dimensional relative pose, frame $\{B\}$ with respect to frame $\{A\}$	
$\boldsymbol{\nu}$	Cartesian velocity screw ( $v_x, v_y, v_z, \omega_x, \omega_y, \omega_z$ )	
$\bar{x}, \bar{y}$	normalized image-plane coordinates	
$\mathbf{0}_{m \times n}$	an $m \times n$ matrix of zeros	
$\mathbf{1}_{m \times n}$	an $m \times n$ matrix of ones	
$\mathbb{Z}$	the set of all integers	
$\mathbb{Z}^+$	the set of all integers greater than zero	
$\sim$	equivalence of representations	
$\simeq$	homogeneous coordinate equivalence	
$\oplus$	pose composition operator	
$\equiv$	colormetric equivalence	
$\ominus$	inverse of a pose (unary operator)	
$\cdot$	transformation of a point by a relative pose, e.g. $\xi \cdot p$	
$\ominus$	smallest angular difference on a circle	rad
$\otimes$	convolution	
$\oplus$	morphological dilation	
$\ominus$	morphological erosion	
$\circ$	morphological opening	
$\bullet$	morphological closing	
$\{F\}$	coordinate frame $F$	
$[a, b]$	interval $a$ to $b$ inclusive	
$(a, b)$	interval $a$ to $b$ , not including $a$ or $b$	
$[a, b)$	interval $a$ to $b$ , not including $b$	
$(a, b]$	interval $a$ to $b$ exclusive, not including $a$	

**MATLAB® Toolbox Conventions**

- A Cartesian coordinate, a point, is expressed as a column vector.
- A set of points is expressed as a matrix with columns representing the coordinates of individual points.
- A robot configuration, a set of joint angles, is expressed as a row vector.
- Time series data is expressed as a matrix with rows representing time steps.

# Introduction



The term robot means different things to different people. Science fiction books and movies have strongly influenced what many people expect a robot to be or what it can do. Sadly the practice of robotics is far behind this popular conception. One thing is certain though – robotics will be an important technology in this century. Products such as vacuum cleaning robots are the vanguard of a wave of smart machines that will appear in our homes and workplaces.

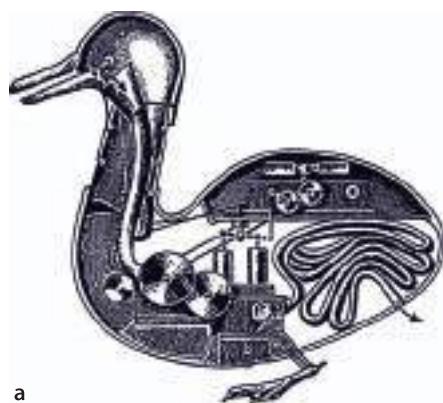
In the eighteenth century the people of Europe were fascinated by automata such as Vaucanson's duck shown in Fig. 1.1a. These machines, complex by the standards of the day, demonstrated what then seemed *life-like* behaviour. The duck used a cam mechanism to sequence its movements and Vaucanson went on to explore mechanization of silk weaving. Jacquard extended these ideas and developed a loom, shown in Fig. 1.1b, that was essentially a programmable weaving machine. The pattern to be woven was encoded as a series of holes on punched cards. This machine has many hallmarks of a modern robot: it performed a physical task and was reprogrammable.

The term robot was coined in a 1921 Czech science fiction play “Rossum’s Universal Robots” by Karel Čapek. The robots were artificial people or androids and the word, in Czech, is derived from the word for slave. In the play, as in so many robot stories that follow, the robots rebel and it ends badly for humanity. Isaac Asimov’s robot series, comprising many books and short stories written between 1950 and 1985, explored issues of human and robot interaction and morality. The robots in these stories are equipped with “positronic brains” in which the “Three laws of robotics” are encoded. These stories have influenced subsequent books and movies which in turn have shaped the public perception of what robots are. The mid twentieth century also saw the advent of the field of *cybernetics* – an uncommon term today but then an exciting science at the frontiers of understanding life and creating intelligent machines.

The first patent for what we would now consider a robot was filed in 1954 by George C. Devol and issued in 1961. The device comprised a mechanical arm with a gripper that was mounted on tracks and the sequence of motions was encoded as magnetic patterns stored on a rotating drum. The first robotics company, Unimation, was founded by Devol and Joseph Engelberger in 1956 and their first industrial robot shown

Fig. 1.1.

Early programmable machines. **a** Vaucanson's duck (1739) was an automaton that could flap its wings, eat grain and defecate. It was driven by a clockwork mechanism and executed a single program; **b** The Jacquard loom (1801) was a reprogrammable machine and the program was held on punched cards (photograph by George P. Landow from [www.victorianweb.org](http://www.victorianweb.org))



a



b

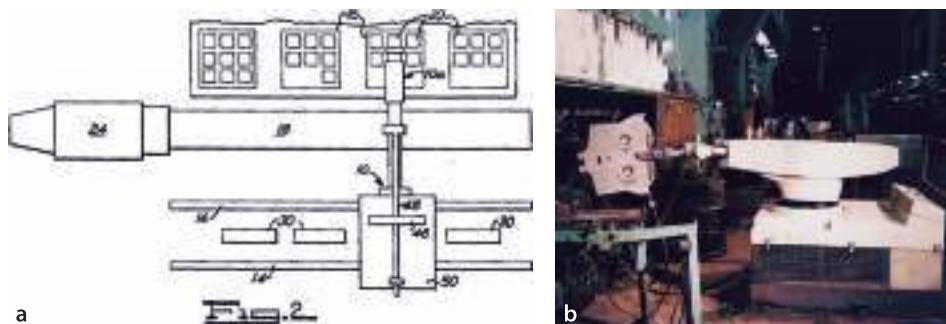


Fig. 1.2.

Universal automation. **a** A plan view of the machine from Devol's patent  
**b** the first Unimate robot working at a General Motors factory (photo courtesy of George C. Devol)

in Fig. 1.2 was installed in 1961. The original vision of Devol and Engelberger for robotic automation has become a reality and many millions of arm-type robots such as shown in Fig. 1.3 have been built and put to work at tasks such as welding, painting, machine loading and unloading, electronic assembly, packaging and palletising. The use of robots has led to increased productivity and improved product quality. Rather than take jobs it has helped to keep manufacturing industries viable in high-labour cost countries. Today many products we buy have been assembled or handled by a robot.

These first generation robots are now a subclass of robotics known as manufacturing robots. Other subclasses include service robots which supply services such as cleaning, personal assistance or medical rehabilitation; field robots which work outdoors such as those shown in Fig. 1.4; and humanoid robots such as shown in Fig. 1.6b that have the physical form of a human being.

A manufacturing robot is typically an arm-type manipulator on a fixed base that performs repetitive tasks within a local work cell. Parts are presented to the robot in an orderly fashion which maximizes the advantage of the robot's high speed and precision. High-speed robots are hazardous and safety is achieved by excluding people from robotic work places.

Field and service robots present important challenges. The first challenge is that the robot must operate and move in a complex, cluttered and changing environment. A delivery robot in a hospital must operate despite crowds of people and a time-varying configuration of parked carts and trolleys. A Mars rover must navigate rocks and small craters despite not having an accurate local map in advance of its travel. Robotic cars, such as demonstrated in the DARPA Grand Challenges (Buehler et al. 2007), must follow roads, obey traffic signals and the rules of the road.

The second challenge for these types of robots is that they must operate safely in the presence of people. The hospital delivery robot operates amongst people, the robotic car contains people and a robotic surgical device operates *inside* people.

**Rossum's Universal Robots (RUR).** In the introductory scene Helena Glory is visiting Harry Domin the director general of Rossum's Universal Robots and his robotic secretary Sulla.

*Domin* Sulla, let Miss Glory have a look at you.

*Helena* (stands and offers her hand) Pleased to meet you. It must be very hard for you out here, cut off from the rest of the world [the factory is on an island]

*Sulla* I do not know the rest of the world Miss Glory. Please sit down.

*Helena* (sits) Where are you from?

*Sulla* From here, the factory

*Helena* Oh, you were born here.

*Sulla* Yes I was made here.

*Helena* (startled) What?

*Domin* (laughing) Sulla isn't a person, Miss Glory, she's a robot.

*Helena* Oh, please forgive me ...

The full play can be found at <http://ebooks.adelaide.edu.au/c/cepek/karel/rur>. (Image on the right: Library of Congress item 96524672)



Fig. 1.3. A modern six-axis robot from ABB that would be used for factory automation. This type of robot is a technological descendant of the Unimate shown in Fig. 1.2





**George Devol, Jr. (1912–)** is a prolific American inventor. He was born in Louisville, Kentucky, and in 1932 founded United Cinephone Corp. which manufactured phonograph arms and amplifiers, registration controls for printing presses and packaging machines. In 1954, he applied for US patent 2,988,237 for Programmed Article Transfer which introduced the concept of Universal Automation or “Unimation”. Specifically it described a track-mounted polar-coordinate arm mechanism with a gripper and a programmable controller – the precursor of all modern robots.

In 2011 he was inducted into the National Inventors Hall of Fame. (Photo on the left: courtesy of George C. Devol)



**Joseph F. Engelberger (1925–)** is an American engineer and entrepreneur who is often referred to as the “Father of Robotics”. He received his B.S. and M.S. degrees in physics from Columbia University, in 1946 and 1949, respectively. Engelberger has been a tireless promoter of robotics. In 1966, he appeared on *The Tonight Show Starring Johnny Carson* with a Unimate robot which poured a beer, putted a golf ball, and directed the band. He promoted robotics heavily in Japan, which led to strong investment and development of robotic technology in that country, and gave testimony to Congress on the value of using automation in space. He has written two books *Robotics in Practice* (1980) and *Robotics in Service* (1989), and the former was translated into six languages.

Engelberger served as chief executive of Unimation until 1982, and in 1984 founded Transitions Research Corporation which became HelpMate Robotics Inc. and was later sold. He remains active in the promotion and development of robots for use in elder care. He was elected to the National Academy of Engineering and received the Beckman Award and the Japan Prize. Each year the Robotics Industries Association presents an award in his honour to “persons who have contributed outstandingly to the furtherance of the science and practice of robotics.”



**Fig. 1.4.** Non land-based mobile robots. **a** SeaBed type Autonomous Underwater Vehicle (AUV) operated by the Australian Centre for Field Robotics (photo by Roger T. Hanlon), **b** Global Hawk unmanned aerial vehicle (UAV) (photo: courtesy of NASA)

So what is a robot? There are many definitions and not all of them are particularly helpful. A definition that will serve us well in this book is

*a goal oriented machine that can sense, plan and act.*

A robot *senses* its environment and uses that information, together with a goal, to *plan* some *action*. The action might be to move the tool of an arm-robot to grasp an object or it might be to drive a mobile robot to some place.

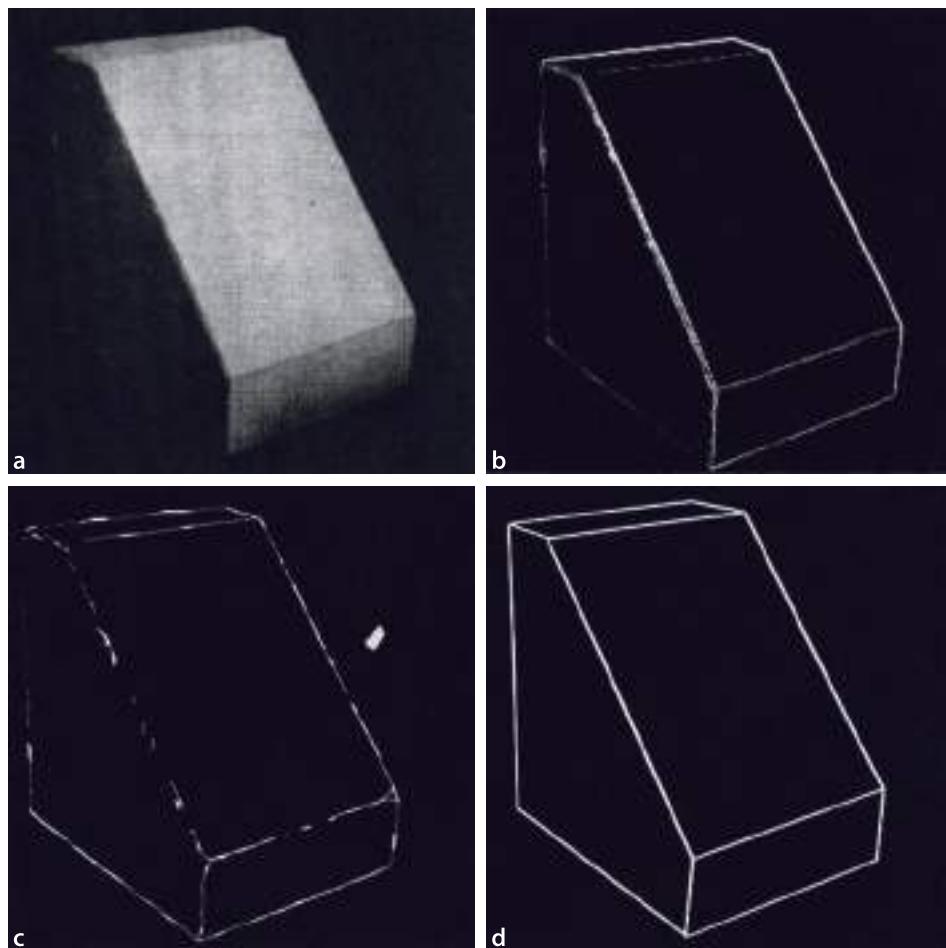
Sensing is critical to robots. Proprioceptive sensors measure the state of the robot itself: the angle of the joints on a robot arm, the number of wheel revolutions on a mobile robot or the current drawn by an electric motor. Exteroceptive sensors measure the state of the world with respect to the robot. The sensor might be a simple contact switch on a vacuum cleaner robot to detect collision. It might be a GPS receiver that measures distances to an orbiting satellite constellation, or a compass that measures the direction of the Earth’s magnetic field relative to the robot’s heading. It might also be an active sensor

**Cybernetics, artificial intelligence and robotics.** Cybernetics flourished as a research field from the 1930s until the 1960s and was fueled by a heady mix of new ideas and results from neurology, feedback and control and information theory. Research in neurology had shown that the brain was an electrical network of neurons. Harold Black, Henrik Bode and Harry Nyquist at Bell Labs were researching negative feedback and the stability of electrical networks, Claude Shannon's information theory described digital signals, and Alan Turing was exploring the fundamentals of computation. Walter Pitts and Warren McCulloch proposed an artificial neuron in 1943 and showed how it might perform simple logical functions. In 1951 Marvin Minsky built SNARC which was perhaps the first neural-network-based learning machine as his graduate project. William Grey Walter's robotic tortoises showed life-life behaviour. Maybe an electronic brain could be built!

An important early book was Norbert Wiener's *Cybernetics or Control and Communication in the Animal and the Machine*

(Wiener 1965). A characteristic of a cybernetic system is the use of feedback which is common in engineering and biological systems. The ideas were later applied to evolutionary biology, psychology and economics.

In 1956 a watershed conference was hosted by John McCarthy at Dartmouth College and attended by Minsky, Shannon, Herbert Simon, Allen Newell and others. This meeting defined the term artificial intelligence (AI) as we know it today with an emphasis on digital computers and symbolic manipulation and led to new research in robotics, vision, natural language, semantics and reasoning. McCarthy and Minsky formed the AI group at MIT, and McCarthy left in 1962 to form the Stanford AI Laboratory. Minsky focused on artificially simple "blocks world". Simon, and his student Newell, were influential in AI research at Carnegie-Mellon University from which the Robotics Institute was spawned in 1979. These AI groups were to be very influential in the development of robotics and computer vision in the USA. Societies and publications focusing on cybernetics are still active today.



**Fig. 1.5.**  
Early results in computer vision for estimating the shape and pose of objects, from the PhD work of L. G. Roberts at MIT Lincoln Lab in 1963 (Roberts 1963). **a** Original picture; **b** gradient image; **c** connected feature points; **d** reconstructed line drawing

that emits acoustic, optical or radio pulses in order to measure the distance to points in the world based on the time taken for a reflection to return to the sensor.

A camera is a passive sensor that captures patterns of energy reflected from the scene. Our own experience is that eyes are a very effective sensor for recognition, navigation, obstacle

avoidance and manipulation so vision has long been of interest to robotics researchers. Figure 1.5 shows early work in reconstructing a 3-dimensional wireframe model from an image and gives some idea of the difficulties involved. An important limitation of a single camera is that the 3-dimensional structure must be inferred from the 2-dimensional image. An alternative approach is stereo vision, using two or more cameras, to compute the 3-dimensional structure of the world. The Mars rover shown in Fig. 1.6a has a stereo camera on its mast.

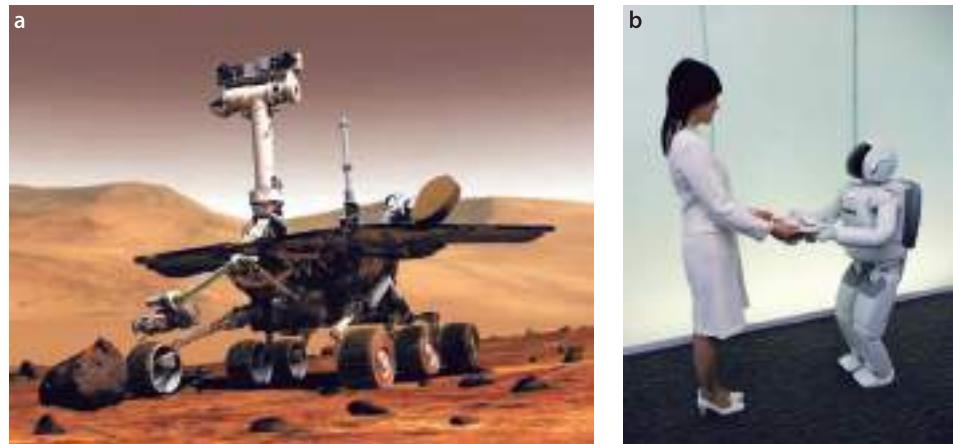
In this book we focus on the use of cameras as sensors for robots. Machine vision, discussed in Part IV, is the use of computers to process images from one or more cameras and to extract numerical features. For example determining the coordinate of a round red object in the scene, or how far a robot has moved based on how the world appears to move relative to the robot.

If the robot's environment is unchanging it can make do with an accurate map and have little need to sense the state of the world, apart from determining where it is. Imagine driving a car with the front window covered over and just looking at the GPS navigation system. If you had the roads to yourself you could probably drive from A to B quite successfully albeit slowly. However if there were other cars, pedestrians, traffic signals or roadworks then you would be in some difficulty. To deal with this you need to look outwards – to sense the world and plan your actions accordingly. For humans this is easy, done without conscious thought, but it is not easy to program a machine to do the same.

Tele-robots are robot-like machines that are remotely controlled by a human operator. Perhaps the earliest was a radio controlled boat demonstrated by Nikola Tesla in 1898 and which he called a teleautomaton. According to the definition above these are not robots but they were an important precursor to robots and are still important today (Goldberg and Siegwart 2001; Goldberg 2001) for many tasks where people cannot work but which are too complex for a machine to perform by itself. For example the underwater robots that surveyed the wreck of the Titanic were technically remotely operated vehicles (ROVs). The Mars rovers Spirit and Opportunity



The Manhattan Project in World War 2 (WW II) developed the first nuclear weapons and this required handling of radioactive material. Remotely controlled arms were developed by Ray Goertz at Argonne National Laboratory to exploit the manual dexterity of human operators while keeping them away from the hazards of the material they were handling. The operators viewed the task they were doing through thick lead-glass windows or via a television link. Tele-robotics is still important today for many tasks where people cannot work but which are too complex for a machine to perform by itself, for instance the underwater robots that surveyed the wreck of the Titanic. (Photo on the left: Courtesy Argonne National Laboratory)



**Fig. 1.6.**

Two very different types of mobile robots. **a** Mars rover. Note the two cameras on the mast which provide stereo vision from which the robot can compute the 3-dimensional structure of its environment (image courtesy of NASA/JPL/Cornell University); **b** Honda's Asimo humanoid robot (photo courtesy Honda Motor Co. Japan)

**Unimation Inc.(1956–1982).** Devol sought financing to develop his unimation technology and eventually met with Joseph Engelberger who was then an engineer with Manning, Maxwell and Moore. In 1956 they jointly established Unimation, the first robotics company, in Danbury Connecticut. The company was acquired by Consolidated Diesel Corp. (Condec) and became Unimate Inc. a division of Condec. Their first robot went to work in 1961 at a General Motors die-casting plant in New Jersey. In 1968 they licenced technology to Kawasaki Heavy Industries which produced the first Japanese industrial robot. Engelberger served as chief executive until it was acquired by Westinghouse in 1982. People and technologies from this company have gone on to be very influential on the whole field of robotics.

autonomously navigate the surface of Mars but human operators provide the high-level goals. That is, the operators tell the robot where to go and the robot itself determines the details of the route. Local decision making on Mars is essential given that the communications delay is several minutes. Some robots are hybrids and the control task is shared or traded with a human operator. In traded control, the control function is passed back and forth between the human operator and the computer. For example an aircraft pilot can pass control to an autopilot and take back control back. In shared control, the control function is performed by the human operator and the computer. For example an autonomous passenger car might have the computer keeping the car in the lane and avoiding collisions, while the human operator just controls the speed.

## 1.1 About the Book

This book is about robotics and computer vision – separately, and together as robotic vision. These are big topics and the combined coverage is necessarily broad. The intent is not to be shallow but rather to give the reader a flavour of what robotics and vision is about and what it can do – consider it a grand tasting menu.

The goals of the book are:

- to provide a broad and solid base of knowledge through theory and examples;
- to tackle more complex problems than other textbooks by virtue of the powerful numerical tools and software that underpins it;
- to provide instant gratification by solving complex problems with relatively little code;
- to complement the many excellent texts in robotics and computer vision;
- to encourage intuition through hands on numerical experimentation; and
- to limit the number of equations presented to where (in my judgement) they add value or clarity.

The approach used is to present background, theory and examples in an integrated fashion. Code and examples are first-class citizens in this book and are not relegated to the end of the chapter or an associated web site. The examples are woven into the discussion like this

```
>> Ts = c traj(T1, T2, t);
>> p = transl(Ts);
>> plot(t, p);
```

where the MATLAB® code illuminates the topic being discussed and generally results in a figure or a crisp numerical result that is then discussed. The examples illustrate how to use the associated MATLAB® Toolboxes and that knowledge can then be applied to other problems. Most of the figures in this book have been generated by the code examples provided – in fact the book is just one very large MATLAB® script.

### 1.1.1 The MATLAB® Software

*To do good work, one must first have good tools.*  
Chinese proverb

The computational foundation of this book is MATLAB®, a product of The Mathworks Inc. MATLAB® is an interactive mathematical software environment that makes linear algebra, data analysis and high-quality graphics a breeze. MATLAB® is a popular package and one that is very likely to be familiar to students and researchers. It also supports a programming language which allows the creation of complex algorithms.

A strength of MATLAB® is its support for Toolboxes which are collections of functions targeted at particular topics. Toolboxes are available from The MathWorks, third party companies and individuals. Some Toolboxes are products and others are open-source. This book is based on two open-source Toolboxes written by the author: the Robotics Toolbox for MATLAB® and the Machine VisionToolbox for MATLAB®. These Toolboxes, with MATLAB® turn a modern personal computer into a powerful and convenient environment for investigating complex problems in robotics, machine vision and vision-based control. The Toolboxes are free to use and distributed under the GNU Lesser General Public License (GNU LGPL).

The *Robotics Toolbox* (RTB) provides a diverse range of functions for simulating mobile and arm-type robots. The original toolbox, dating back to the early 1990s, was concerned only with arm-type robots and supported a very general method of representing the structure of serial-link manipulators using matrices and later MATLAB® objects. Arbitrary serial-link manipulators could be created and the Toolbox provides functions for forward and inverse kinematics and dynamics. The Toolbox includes functions for manipulating and converting between datatypes such as vectors, homogeneous transformations, 3-angle representations and unit-quaternions which are necessary to represent 3-dimensional position and orientation.

The Toolbox released with this book adds significant new functionality for simulating mobile robots. The RTB now includes models of car-like vehicles and quadcopters and controllers for these vehicles. It also provides standard algorithms for robot path planning, localization and map making.

The *Machine Vision Toolbox* (MVTB) provides a rich collection of functions for camera modeling, image processing, image feature extraction, multi-view geometry and vision-based control. This Toolbox is younger than the Robotics Toolbox but it is

The MATLAB® software we use today has a long history. It starts with the LINPACK and EISPACK projects run by the Argonne National Laboratory in the 1970s to produce high quality, tested and portable mathematical software. LINPACK is a collection of routines for linear algebra and EISPACK is a library of numerical algorithms for computing eigenvalues and eigenvectors of matrices. These packages were written in Fortran which was then, and even today is, the language of choice for large-scale numerical problems.

Cleve Moler, then at the University of New Mexico, contributed to both projects and wrote the first version of MATLAB® in the late 1970s. It allowed interactive use of LINPACK and EISPACK for problem solving without having to write and compile Fortran code. MATLAB® quickly spread to other universities and found a strong audience within the applied mathematics and engineering community. In 1984 Cleve Moler and Jack Little founded The MathWorks Inc. which exploited the newly released IBM PC – the first widely available desktop computer.

Cleve Moler received his bachelor's degree from Caltech in 1961, and a Ph.D. from Stanford University. He was a professor of mathematics and computer science at universities including University of Michigan, Stanford University, and the University of New Mexico. He has served as president of the Society for Industrial and Applied Mathematics (SIAM) and was elected to the National Academy of Engineering in 1997.

See also <http://www.mathworks.com/company/aboutus/founders/clevemoler.html> which includes a video of Cleve Moler and also [http://history.siam.org/pdfs2/Moler\\_final.pdf](http://history.siam.org/pdfs2/Moler_final.pdf).

not a clone of the MATLAB® Image Processing Toolbox (IPT). Although there is some common functionality the Machine Vision Toolbox predates IPT by many years. The MVTB contains many functions for image acquisition and display; filtering; blob, point and line feature extraction; mathematical morphology; image warping; stereo vision; homography and fundamental matrix estimation; robust estimation; visual Jacobians; geometric camera models; camera calibration and color space operations. For modest image sizes on a modern computer the processing rate can be sufficiently “real-time” to allow for closed-loop control.

The Toolboxes are provided in source code form. The bulk of the code is written in the MATLAB® M-language but some functions are written in C for increased computational efficiency.► In general the Toolbox code is written in a straightforward manner to facilitate understanding, perhaps at the expense of computational efficiency. If you’re starting out in robotics or vision then the Toolboxes are a significant initial base of code on which to base your project.

This book provides examples of the usage of many Toolbox functions in the context of solving specific problems but it is not a reference manual. Comprehensive documentation of all Toolbox functions is available through MATLAB’s builtin help mechanism. This approach allows the code to evolve and develop new features over time while maintaining backward compatibility and not obsoleting the book.

Appendix A provides details of how to obtain the Toolboxes and pointers to online resources including discussion groups.

These are implemented as MEX files, which are written in C in a very specific way that allows them to be invoked from MATLAB® just like a function written in M-language.

### 1.1.2 Audience and Prerequisites

The book is intended primarily for third or fourth year undergraduate students and graduate students in their first year. For undergraduates the book will serve as a companion text for a robotics or machine vision course or to support a major project in robotics or vision. Students should study Part I and the appendices for foundational concepts, and then the relevant part of the book: mobile robotics, arm robots, computer vision or vision-based control. The Toolboxes provide a solid set of tools for problem solving, and the exercises at the end of each chapter provide additional problems beyond the worked examples in the book.

For students commencing graduate study in robotics, and who have previously studied engineering or computer science, the book will help fill in the gaps between what you learned as an undergraduate and what will be required to underpin your deeper study of robotics and computer vision. The book’s working code base can help bootstrap your research, enabling you to get started quickly and working productively on your own problems and ideas. Since the source code is available you can reshape it to suit your need, and when the time comes (as it usually does) to code your algorithms in some other language then the Toolboxes can be used to cross-check your implementation.

For those who are no longer students, the researcher or industry practitioner, the book will serve as a useful companion for your own reference to a wide range of topics in robotics and computer vision, as well as a Handbook and guide for the Toolboxes.

The book assumes undergraduate-level knowledge of linear algebra (matrices, vectors, eigenvalues), basic set theory, basic graph theory, probability, dynamics (forces, torques, inertia), the Laplace transform and transfer functions, linear control (proportional control, proportional-derivative control, proportional-integral control) and block diagram notation. Computer science students are less likely to have encountered the Laplace transform and classical control but this appears only in Sect. 9.4, and Hellerstein et al. (2004) may be a useful introduction to the topics. The book also assumes the reader is familiar with programming in MATLAB® and also familiar with object-oriented programming techniques (perhaps C++, Java or Python). Familiarity with Simulink®, MATLAB’s graphical block-diagram modeling tool will be helpful but not essential. The appendices provide concise refreshers on many of these topics.

---

### 1.1.3 Notation and Conventions

The mathematical notation used in the book is summarized in the nomenclature section on page xvii. Since the coverage of the book is broad there are just not enough good symbols to go around, so it is unavoidable that some symbols have different meanings in different parts of the book.

There is a lot of MATLAB® code in the book and this is indicated in blue fixed-width font such as

```
>> a = 2 + 2  
a =  
4
```

The MATLAB® command prompt is `>>` and what follows is the command issued to MATLAB® by the user. Subsequent lines without the prompt are MATLAB's response. All functions, classes and methods mentioned in the text or in code segments are cross-referenced and have their own indexes at the end of the book. All the MATLAB® code segments are available from the book's web page as described in Appendix A. This book is not a manual and although it illustrates the use of many functions within the Toolbox the definitive source for information about all Toolbox functions is the online documentation. Every MATLAB® and Toolbox function used in the code examples is included in the index of functions on page 554 allowing you to find different ways that particular functions have been used.

Colored boxes are used to indicate different types of material. Orange informational boxes highlight material that is particularly important while orange and red warning boxes high-light points that are often traps for those starting out. Blue boxes provide technical, historical or biographical information that augment the main text but they are not critical to its understanding.

They are placed as marginal notes near the corresponding marker.

As an author there is a tension between completeness, clarity and conciseness. For this reason a lot of detail has been pushed into notes and blue boxes and on a first reading these can be skipped. However if you are trying to understand a particular algorithm and apply it to your own problem then understanding the details and nuances can be important and the notes are for you.

Each chapter ends with a *Wrapping up* section that summarizes the important lessons from the chapter, discusses some suggested further reading, and provides some exercises. References are cited sparingly in the text of each chapter. The *Further reading* subsection discusses prior work and references that provide more rigour or complete description of the algorithms. *Exercises* extend the concepts discussed within the chapter and are generally related to specific code examples discussed in the chapter. The exercises vary in difficulty from straightforward extension of the code examples to more challenging problems.

---

### 1.1.4 How to Use the Book

The best way to learn is by doing. Although the book shows the MATLAB® commands and the response there is something special about doing it for yourself. Consider the book as an invitation to tinker. By running the commands yourself you can look at the results in ways that you prefer, plot the results in a different way, or try the algorithm on different data or with different parameters. You can also look at the online documentation for the Toolbox functions, discover additional features and options, and experiment with those, or read the code to see how it really works and then modify it.

Most of the commands are quite short so typing them in to MATLAB® is not too onerous. However the book's web site, see Appendix A, includes all the MATLAB® commands shown in the book (more than 1 600 lines) and these can be cut and pasted into MATLAB® or downloaded and used to create your own scripts.

### 1.1.5 Teaching with the Book

The book can be used in support of courses in robotics, mechatronics and computer vision. All courses should include the introduction to coordinate frames and their composition which is discussed in Chap. 2. For a mobile robotics or image processing course it is sufficient to teach only the 2-dimensional case. For robotics or multi-view geometry the 2- and 3-dimensional cases should be taught. Every figure in this book is available as a PDF format file from the book's web site and is free for use with attribution. All the code in this book can be downloaded from the web site and used as the basis for demonstrations in lectures or tutorials.

The exercises at the end of each chapter can be used as the basis of assignments, or as examples to be worked in class or in tutorials. Most of the questions are rather open ended in order to encourage exploration and discovery of the effects of parameters and the limits of performance of algorithms. This exploration should be supported by discussion and debate about performance measures and what *best* means. True understanding of algorithms involves an appreciation of the effects of parameters, how algorithms fail and under what circumstances.

The teaching approach could also be inverted, by diving headfirst into a particular problem and then teaching the appropriate prerequisite material. Suitable problems could be chosen from the Application sections of Chap. 7, 14 or 16, or from any of the exercises. Particularly challenging exercises are so marked.

For graduate students the papers and textbooks mentioned in the *Further Reading* could form the basis of a student's reading list. They could also serve as candidate papers for a reading group or journal club.

---

### 1.1.6 Outline

I promised a book with instant gratification but before we can get started in robotics there are some fundamental concepts that we absolutely need to understand, and understand well. Part I introduces the concepts of pose and coordinate frames – how we represent the position and orientation of a robot and objects that the robot needs to work with. We discuss how motion between two poses can be *decomposed* into elementary translations and rotations, and how elementary motions can be *composed* into more complex motions. Chapter 2 discusses how pose can be represented in a computer, and Chap. 3 discusses how we can generate a sequence of poses that smoothly follow some path in space and time and the relationship between velocity and the derivative of pose.

With these formalities out of the way we move on to the first main event – robots. There are two important classes of robot: mobile robots and manipulator arms and these are covered in Parts II and III respectively.

Part II begins, in Chap. 4, with a discussion of robot mobility which covers concepts such as under-actuation and non-holonomy and then introduces motion models for a car-like vehicle and a quad-rotor flying vehicle. Various control laws are discussed for the car-like vehicle such as moving to a point, following a path and moving to a specific pose. Chapter 5 is concerned with navigation, that is, how a robot finds a path between points A and B in the world. Two important cases, with and without a map, are discussed. Most navigation techniques require knowledge of the robot's position and Chap. 6 discusses various approaches to this problem based on dead-reckoning, or landmark observation and a map. We also show how a robot can make a map, and even determine its location while simultaneously mapping an unknown region.

Part III is concerned with arm-type robots, or more precisely serial-link manipulators. Manipulator arms are used for tasks such as assembly, welding, material handling and even surgery. Chapter 7 introduces the topic of kinematics which relates the angles of the robot's joints to the 3-dimensional pose of the robot's tool. Techniques to gener-

ate smooth paths for the tool are discussed and two examples show how an arm-robot can draw a letter on a surface and how multiple arms (acting as legs) can be used to create a model for a simple walking robot. Chapter 8 discusses the relationships between the rates of change of joint angles and tool pose. It introduces the Jacobian matrix and concepts such as singularities, manipulability, null-space motion, and resolved-rate motion control. It also discusses under- and over-actuated robots and the general numerical solution to inverse kinematics. Chapter 9 introduces the dynamic equations of motion for a serial-link manipulator and the relationship between joint forces and joint motion. The design of joint control systems is discussed and covers important topics such as variation in inertia and payload, flexible transmissions and independent joint versus non-linear control strategies.

Computer vision is a large field concerned with processing images in order to enhance them for human benefit, interpret the contents of the scene or create a 3D model corresponding to the scene. Part IV is concerned with machine vision, a subset of computer vision, and defined here as the extraction of numerical features from images to provide input for control of a robot. The discussion starts in Chap. 10 with the fundamentals of light, illumination and color. Chapter 11 describes the geometric model of perspective image creation using lenses and discusses topics such as camera calibration and pose estimation. We also introduce non-perspective imaging using wide-angle lenses and mirror systems and the relationship to perspective images. Chapter 12 discusses *image processing* which is a domain of 2-dimensional signal processing that transforms one image into another image. The discussion starts with acquiring real-world images and then covers various arithmetic and logical operations that can be performed on images. We then introduce spatial operators such as convolution, segmentation, morphological filtering and finally image shape and size changing. These operations underpin the discussion in Chap. 13 which describe how numerical features are extracted from images. The features describe homogeneous regions (blobs), lines or distinct points in the scene and are the basis for vision-based robot control. Chapter 14 discusses how features found in different views of the scene can provide information about its underlying three-dimensional geometry and the spatial relationship between the camera views.

Part V discusses how visual features extracted from the camera's view can be used to control arm-type and mobile robots – an approach known as vision-based control or visual servoing. This part pulls together concepts introduced in the earlier parts of the book. Chapter 15 introduces the classical approaches to visual servoing known as position-based and image-based visual servoing and discusses their respective limitations. Chapter 16 discusses more recent approaches that address these limitations and also covers the use of non-perspective cameras, under-actuated robots and mobile robots.

This is a big book but any one of the parts can be read standalone, with more or less frequent visits to the required earlier material. Chapter 2 is the only mandatory material. Parts II, III or IV could be read standalone for an introduction to mobile robots, arm robots or machine vision respectively. An alternative approach, following the instant gratification theme, is to jump straight into any chapter and start exploring – visiting the earlier material as required.

# Representing Position and Orientation

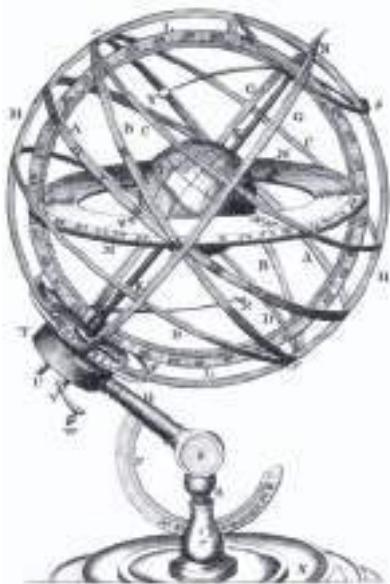


Fig. 2.1.

a The point P is described by a coordinate vector with respect to an absolute coordinate frame. b The points are described with respect to the object's coordinate frame  $\{B\}$  which in turn is described by a relative pose  $\xi_B$ . Axes are denoted by thick lines with an open arrow, vectors by thin lines with a swept arrow head and a pose by a thick line with a solid head

A fundamental requirement in robotics and computer vision is to represent the position and orientation of objects in an environment. Such objects include robots, cameras, workpieces, obstacles and paths.

A point in space is a familiar concept from mathematics and can be described by a coordinate vector, also known as a bound vector, as shown in Fig. 2.1a. The vector represents the displacement of the point with respect to some reference coordinate frame. A coordinate frame, or Cartesian coordinate system, is a set of orthogonal axes which intersect at a point known as the origin.

More frequently we need to consider a set of points that comprise some object. We assume that the object is *rigid* and that its constituent points maintain a constant relative position with respect to the object's coordinate frame as shown in Fig. 2.1b. Instead of describing the individual points we describe the position and orientation of the object by the position and orientation of its coordinate frame. A coordinate frame is labelled,  $\{B\}$  in this case, and its axis labels  $x_B$  and  $y_B$  adopt the frame's label as their subscript.

The position and orientation of a coordinate frame is known as its pose and is shown graphically as a set of coordinate axes. The relative pose of a frame with respect to a reference coordinate frame is denoted by the symbol  $\xi$

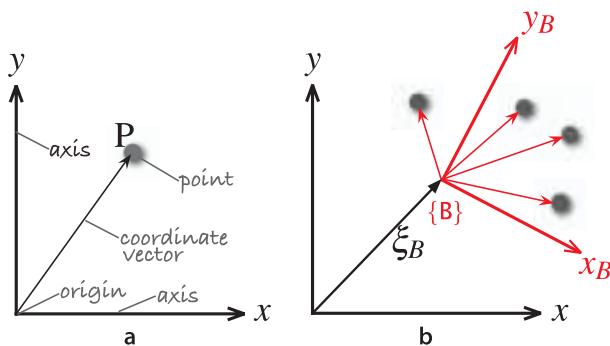
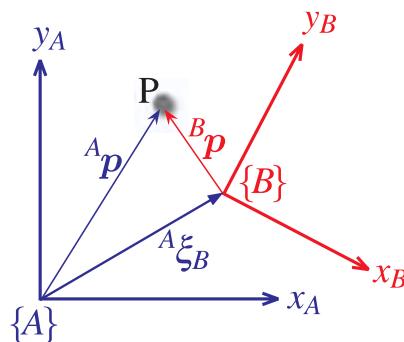


Fig. 2.2.

The point P can be described by coordinate vectors relative to either frame  $\{A\}$  or  $\{B\}$ . The pose of  $\{B\}$  relative to  $\{A\}$  is  ${}^A\xi_B$



– pronounced ksi. Figure 2.2 shows two frames  $\{A\}$  and  $\{B\}$  and the relative pose  ${}^A\xi_B$  which describes  $\{B\}$  with respect to  $\{A\}$ . The leading superscript denotes the reference coordinate frame and the subscript denotes the frame being described. We could also think of  ${}^A\xi_B$  as describing some motion – imagine picking up  $\{A\}$  and applying a displacement and a rotation so that it is transformed to  $\{B\}$ . If the initial superscript is missing we assume that the change in pose is relative to the world coordinate frame denoted  $O$ .

The point  $P$  in Fig. 2.2 can be described with respect to *either* coordinate frame. Formally we express this as

$${}^A p = {}^A\xi_B \cdot {}^B p \quad (2.1)$$

where the right-hand side expresses the motion from  $\{A\}$  to  $\{B\}$  and then to  $P$ . The operator  $\cdot$  transforms the vector, resulting in a new vector that describes the same point but with respect to a different coordinate frame.

An important characteristic of relative poses is that they can be *composed* or *compounded*. Consider the case shown in Fig. 2.3. If one frame can be described in terms of another by a relative pose then they can be applied sequentially

$${}^A\xi_C = {}^A\xi_B \oplus {}^B\xi_C$$

which says, in words, that the pose of  $\{C\}$  relative to  $\{A\}$  can be obtained by compounding the relative poses from  $\{A\}$  to  $\{B\}$  and  $\{B\}$  to  $\{C\}$ . We use the operator  $\oplus$  to indicate composition of relative poses.

For this case the point  $P$  can be described

$${}^A p = ({}^A\xi_B \oplus {}^B\xi_C) \cdot {}^C p$$

Later in this chapter we will convert these abstract notions of  $\xi$ ,  $\cdot$  and  $\oplus$  into standard mathematical objects and operators that we can implement in MATLAB®.

In the examples so far we have shown 2-dimensional coordinate frames. This is appropriate for a large class of robotics problems, particularly for mobile robots which operate in a planar world. For other problems we require 3-dimensional coordinate frames to describe objects in our 3-dimensional world such as the pose of a flying or underwater robot or the end of a tool carried by a robot arm.

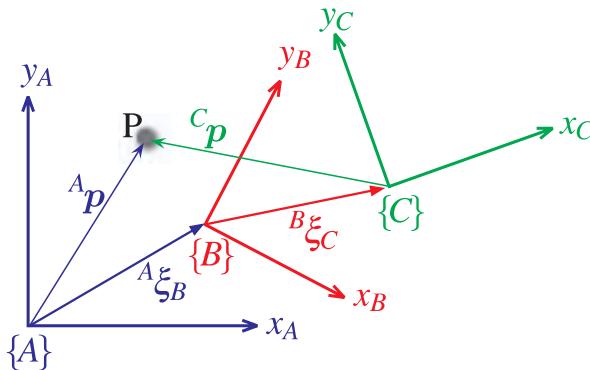


Fig. 2.3.

The point  $P$  can be described by coordinate vectors relative to either frame  $\{A\}$ ,  $\{B\}$  or  $\{C\}$ . The frames are described by relative poses

In relative pose composition we can check that we have our reference frames correct by ensuring that the subscript and superscript on each side of the  $\oplus$  operator are matched. We can then *cancel out* the intermediate subscripts and superscripts

$${}^x\xi_z = {}^{x\xi_x} \oplus {}^{x\xi_z}$$

leaving just the end most subscript and superscript which are shown circled.

Figure 2.4 shows a more complex 3-dimensional example in a graphical form where we have attached 3D coordinate frames to the various entities and indicated some relative poses. The fixed camera observes the object from its fixed viewpoint and estimates the object's pose relative to itself. The other camera is not fixed, it is attached to the robot at some constant relative pose and estimates the object's pose relative to itself.

An alternative representation of the spatial relationships is a directed graph (see Appendix J) which is shown in Fig. 2.5. Each node in the graph represents a pose and each edge represents a relative pose. An arrow from  $X$  to  $Y$  is denoted  ${}^X\xi_Y$  and describes the pose of  $Y$  relative to  $X$ . Recalling that we can compose relative poses using the  $\oplus$  operator we can write some spatial relationships

$$\xi_F \oplus {}^F\xi_B = \xi_R \oplus {}^R\xi_C \oplus {}^C\xi_B$$

$$\xi_F \oplus {}^F\xi_R = {}^0\xi_R$$

and each equation represents a loop in the graph. Each side of the equation represents a path through the network, a sequence of edges (arrows) that are written in head to tail order. Both sides of the equation start and end at the same node.

A very useful property of poses is the ability to perform algebra. ▶ The second loop equation says, in words, that the pose of the robot is the same as composing two relative poses: from the world frame to the fixed camera and from the fixed camera to the

In mathematical objects terms poses constitute a group – a set of objects that supports an associative binary operator (composition) whose result belongs to the group, an inverse operation and an identity element. In this case the group is the special Euclidean group in either 2 or 3 dimensions which are commonly referred to as  $SE(2)$  or  $SE(3)$  respectively.

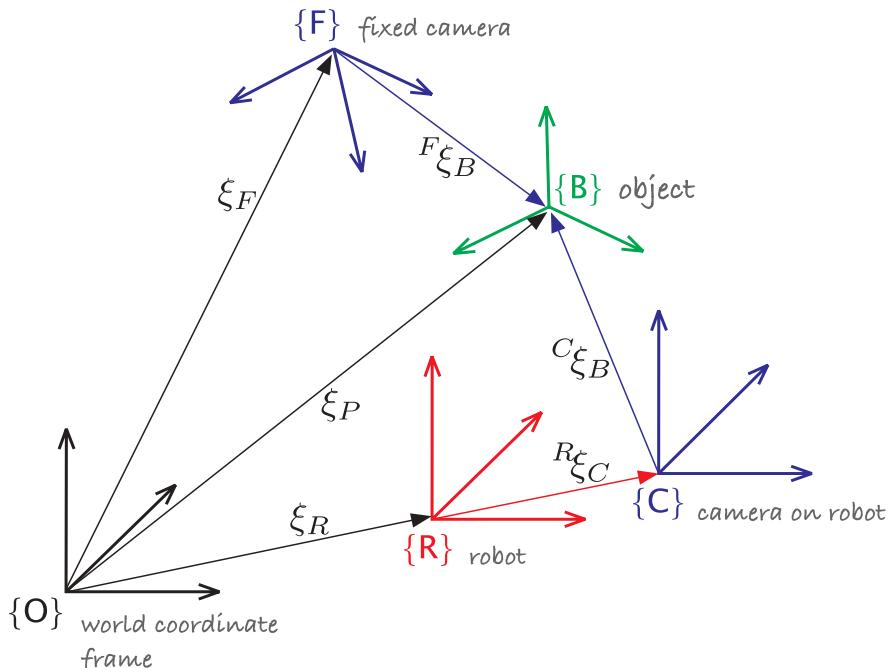
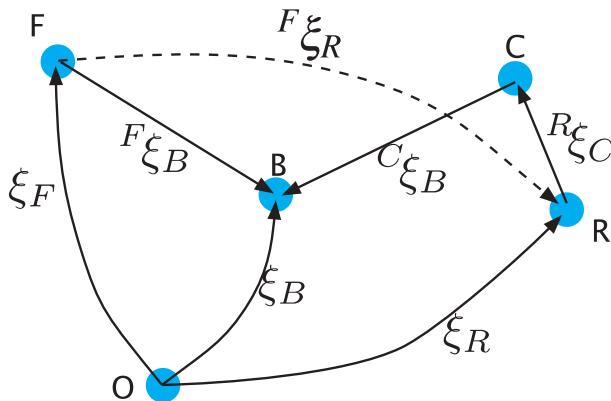


Fig. 2.4.  
Multiple 3-dimensional coordinate frames and relative poses



**René Descartes** (1596–1650) was a French philosopher, mathematician and part-time mercenary. He is famous for the philosophical statement “*Cogito, ergo sum*” or “*I am thinking, therefore I exist*” or “*I think, therefore I am*”. He was a sickly child and developed a life-long habit of lying in bed and thinking until late morning. A possibly apocryphal story is that during one such morning he was watching a fly walk across the ceiling and realized that he could describe its position in terms of its distance from the two edges of the ceiling. This coordinate system, the *Cartesian* system, forms the basis of modern (analytic) geometry and influenced the development of modern calculus. In Sweden at the invitation of Queen Christine he was obliged to rise at 5 A.M., breaking his lifetime habit – he caught pneumonia and died. His remains were later moved to Paris, and then moved several times, and there is now some debate about where his remains are. After his death, the Roman Catholic Church placed his works on the Index of Prohibited Books.



**Fig. 2.5.**  
Spatial example of Fig. 2.4  
expressed as a directed graph

robot. We can subtract  $\xi_F$  from both sides of the equation by adding the inverse of  $\xi_F$  which we denote as  $\ominus \xi_F$  and this gives

$$\begin{aligned}\ominus \xi_F \oplus \xi_F \oplus {}^F\xi_R &= \ominus \xi_F \oplus \xi_R \\ {}^F\xi_R &= \ominus \xi_F \oplus \xi_R\end{aligned}$$

which is the pose of the robot relative to the fixed camera.

There are just a few algebraic rules:

$$\begin{aligned}\xi \oplus 0 &= \xi, \quad \xi \ominus 0 = \xi \\ \xi \ominus \xi &= 0, \quad \ominus \xi \oplus \xi = 0\end{aligned}$$

where 0 represents a zero relative pose. A pose has an inverse

$$\ominus {}^X\xi_Y = {}^Y\xi_X$$

which is represented graphically by an arrow from  $Y$  to  $X$ . Relative poses can also be composed or compounded

$${}^X\xi_Y \oplus {}^Y\xi_Z = {}^X\xi_Z$$

It is important to note that the algebraic rules for poses are different to normal algebra and that composition is *not* commutative

$$\xi_1 \oplus \xi_2 \neq \xi_2 \oplus \xi_1$$

with the exception being the case where  $\xi_1 \oplus \xi_2 = 0$ . A relative pose can transform a point expressed as a vector relative to one frame to a vector relative to another

$${}^X p = {}^X\xi_Y \cdot {}^Y p$$

So what is  $\xi$ ? It can be any mathematical object that supports the algebra described above and is suited to the problem at hand. It will depend on whether we are considering a 2- or 3-dimensional problem. Some of the objects that we will discuss in the rest of this chapter include vectors as well as more exotic mathematical objects such as homogeneous transformations, orthonormal rotation matrices and quaternions. Fortunately all these mathematical objects are well suited to the mathematical programming environment of MATLAB®.

To recap:

1. A point is described by a coordinate vector that represents its displacement from a reference coordinate system;
2. A set of points that represent a rigid object can be described by a single coordinate frame, and its constituent points are described by displacements from that coordinate frame;
3. The position and orientation of an object's coordinate frame is referred to as its pose;
4. A relative pose describes the pose of one coordinate frame with respect to another and is denoted by an algebraic variable  $\xi$ ;
5. A coordinate vector describing a point can be represented with respect to a different coordinate frame by applying the relative pose to the vector using the  $\cdot$  operator;
6. We can perform algebraic manipulation of expressions written in terms of relative poses.

The remainder of this chapter discusses concrete representations of  $\xi$  for various common cases that we will encounter in robotics and computer vision.

## 2.1 Representing Pose in 2-Dimensions

A 2-dimensional world, or plane, is familiar to us from high-school Euclidean geometry. We use a Cartesian coordinate system or coordinate frame with orthogonal axes denoted  $x$  and  $y$  and typically drawn with the  $x$ -axis horizontal and the  $y$ -axis vertical. The point of intersection is called the origin. Unit-vectors parallel to the axes are denoted  $\hat{x}$  and  $\hat{y}$ . A point is represented by its  $x$ - and  $y$ -coordinates  $(x, y)$  or as a bound vector

$$\mathbf{p} = x\hat{x} + y\hat{y} \quad (2.2)$$

Figure 2.6 shows a coordinate frame  $\{B\}$  that we wish to describe with respect to the reference frame  $\{A\}$ . We can see clearly that the origin of  $\{B\}$  has been displaced by the vector  $\mathbf{t} = (x, y)$  and then rotated counter-clockwise by an angle  $\theta$ . A concrete representation of pose is therefore the 3-vector  ${}^A\xi_B \sim (x, y, \theta)$ , and we use the symbol  $\sim$  to denote that the two representations are equivalent. Unfortunately this *representation* is not convenient for compounding since

$$(x_1, y_1, \theta_1) \oplus (x_2, y_2, \theta_2)$$

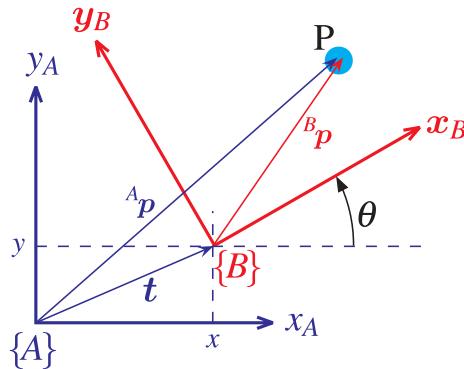
is a complex trigonometric function of both poses. Instead we will use a different way of representing rotation.

The approach is to consider an arbitrary point  $P$  with respect to each of the coordinate frames and to determine the relationship between  ${}^A\mathbf{p}$  and  ${}^B\mathbf{p}$ . Referring again to Fig. 2.6 we will consider the problem in two parts: rotation and then translation.



**Euclid of Alexandria** (ca. 325 BCE–265 BCE) was an Egyptian mathematician who is considered the “father of geometry”. His book *Elements* deduces the properties of geometrical objects and integers from a small set of axioms.

*Elements* is probably the most successful book in the history of mathematics. It describes plane geometry and is the basis for most people’s first introduction to geometry and formal proof, and is the basis of what we now call Euclidean geometry. Euclidean distance is simply the distance between two points on a plane. Euclid also wrote *Optics* which describes geometric vision and perspective.



**Fig. 2.6.**  
Two 2D coordinate frames  $\{A\}$  and  $\{B\}$  and a world point  $P$ .  $\{B\}$  is rotated and translated with respect to  $\{A\}$

To consider just rotation we create a new frame  $\{V\}$  whose axes are parallel to those of  $\{A\}$  but whose origin is the same as  $\{B\}$ , see Fig. 2.7. According to Eq. 2.2 we can express the point  $P$  with respect to  $\{V\}$  in terms of the unit-vectors that define the axes of the frame

$$\begin{aligned} {}^T p &= {}^T x \hat{x}_T + {}^T y \hat{y}_T \\ &= (\hat{x}_T \quad \hat{y}_T) \begin{pmatrix} {}^T x \\ {}^T y \end{pmatrix} \end{aligned} \quad (2.3)$$

which we have written as the product of a row and a column vector.

The coordinate frame  $\{B\}$  is completely described by its two orthogonal axes which we represent by two unit vectors

$$\begin{aligned} \hat{x}_B &= \cos \theta \hat{x}_V + \sin \theta \hat{y}_V \\ \hat{y}_B &= -\sin \theta \hat{x}_V + \cos \theta \hat{y}_V \end{aligned}$$

which can be written in matrix form as

$$(\hat{x}_B \quad \hat{y}_B) = (\hat{x}_V \quad \hat{y}_V) \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \quad (2.4)$$

Using Eq. 2.2 we can represent the point  $P$  with respect to  $\{B\}$  as

$$\begin{aligned} {}^B p &= {}^B x \hat{x}_B + {}^B y \hat{y}_B \\ &= (\hat{x}_B \quad \hat{y}_B) \begin{pmatrix} {}^B x \\ {}^B y \end{pmatrix} \end{aligned}$$

and substituting Eq. 2.4 we write

$${}^B p = (\hat{x}_V \quad \hat{y}_V) \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} {}^B x \\ {}^B y \end{pmatrix} \quad (2.5)$$

Now by equating the coefficients of the right-hand sides of Eq. 2.3 and Eq. 2.5 we write

$$\begin{pmatrix} {}^V x \\ {}^V y \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} {}^B x \\ {}^B y \end{pmatrix}$$

which describes how points are transformed from frame  $\{B\}$  to frame  $\{V\}$  when the frame is rotated. This type of matrix is known as a rotation matrix and denoted  ${}^V R_B$

$$\begin{pmatrix} {}^V x \\ {}^V y \end{pmatrix} = {}^V R_B \begin{pmatrix} {}^B x \\ {}^B y \end{pmatrix} \quad (2.6)$$

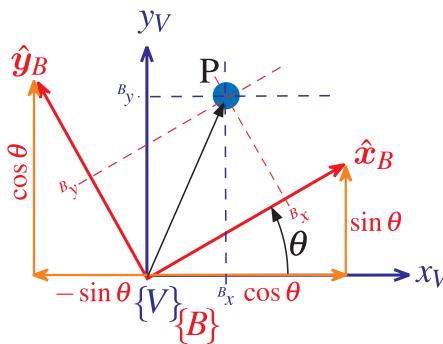


Fig. 2.7.

Rotated coordinate frames in 2D. The point P can be considered with respect to the red or blue coordinate frame

See Appendix D which provides a refresher on vectors, matrices and linear algebra.

The rotation matrix  ${}^V R_B$  has some special properties. Firstly it is *orthonormal* (also called *orthogonal*) since each of its columns is a unit vector and the columns are orthogonal. In fact the columns are simply the unit vectors that define  $\{B\}$  with respect to  $\{V\}$  and are by definition both unit-length and orthogonal.

Secondly, its *determinant* is +1, which means that  $R$  belongs to the special orthogonal group of dimension 2 or  $R \in SO(2) \subset \mathbb{R}^{2 \times 2}$ . The unit determinant means that the length of a vector is unchanged after transformation, that is,  $|{}^B p| = |{}^V p|, \forall \theta$ .

Orthonormal matrices have the very convenient property that  $R^{-1} = R^T$ , that is, the inverse is the same as the transpose. We can therefore rearrange Eq. 2.6 as

$$\begin{pmatrix} {}^B x \\ {}^B y \end{pmatrix} = ({}^V R_B)^{-1} \begin{pmatrix} {}^V x \\ {}^V y \end{pmatrix} = ({}^V R_B)^T \begin{pmatrix} {}^V x \\ {}^V y \end{pmatrix} = {}^B R_V \begin{pmatrix} {}^V x \\ {}^V y \end{pmatrix}$$

Note that inverting the matrix is the same as swapping the superscript and subscript, which leads to the identity  $R(-\theta) = R(\theta)^T$ .

It is interesting to observe that instead of representing an angle, which is a scalar, we have used a  $2 \times 2$  matrix that comprises four elements, however these elements are not independent. Each column has a unit magnitude which provides two constraints. The columns are orthogonal which provides another constraint. Four elements and three constraints are effectively one independent value. The rotation matrix is an example of a non-minimum representation and the disadvantages such as the increased memory it requires are outweighed, as we shall see, by its advantages such as composability.

The second part of representing pose is to account for the translation between the origins of the frames shown in Fig. 2.6. Since the axes  $\{V\}$  and  $\{A\}$  are parallel this is simply vectorial addition

$$\begin{pmatrix} {}^A x \\ {}^A y \end{pmatrix} = \begin{pmatrix} {}^V x \\ {}^V y \end{pmatrix} + \begin{pmatrix} x \\ y \end{pmatrix} \quad (2.7)$$

$$= \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} {}^B x \\ {}^B y \end{pmatrix} + \begin{pmatrix} x \\ y \end{pmatrix} \quad (2.8)$$

$$= \begin{pmatrix} \cos \theta & -\sin \theta & x \\ \sin \theta & \cos \theta & y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} {}^B x \\ {}^B y \\ 1 \end{pmatrix} \quad (2.9)$$

or more compactly as

$$\begin{pmatrix} {}^A x \\ {}^A y \\ 1 \end{pmatrix} = \begin{pmatrix} {}^A R_B & \mathbf{t} \\ \mathbf{0}_{1 \times 2} & 1 \end{pmatrix} \begin{pmatrix} {}^B x \\ {}^B y \\ 1 \end{pmatrix} \quad (2.10)$$

A vector  $(x, y)$  is written in homogeneous form as  $\tilde{p} \in \mathbb{P}^2$ ,  $\tilde{p} = (x_1, x_2, x_3)$  where  $x = x_1/x_3, y = x_2/x_3$  and  $x_3 \neq 0$ . The dimension has been increased by one and a point on a plane is now represented by a 3-vector. To convert a point to homogeneous form we typically append a one  $\tilde{p} = (x, y, 1)$ . The tilde indicates the vector is homogeneous.

Homogeneous vectors have the important property that  $\tilde{p}$  is equivalent to  $\lambda\tilde{p}, \forall \lambda \neq 0$  which we write as  $\tilde{p} \simeq \lambda\tilde{p}$ . That is  $\tilde{p}$  represents the same point in the plane irrespective of the overall scaling factor. Homogeneous representation is important for computer vision that we discuss in Part IV. Additional details are provided in Appendix I.

where  $t = (x, y)$  is the translation of the frame and the orientation is  ${}^A R_B$ . Note that  ${}^A R_B = {}^T R_B$  since the axes of  $\{A\}$  and  $\{V\}$  are parallel. The coordinate vectors for point P are now expressed in homogenous form and we write

$$\begin{aligned} {}^A \tilde{p} &= \begin{pmatrix} {}^T R_B & t \\ \mathbf{0}_{1 \times 2} & 1 \end{pmatrix} {}^B \tilde{p} \\ &= {}^A T_B {}^B \tilde{p} \end{aligned}$$

and  ${}^A T_B$  is referred to as a homogeneous transformation. The matrix has a very specific structure and belongs to the special Euclidean group of dimension 2 or  $T \in SE(2) \subset \mathbb{R}^{3 \times 3}$ .

By comparison with Eq. 2.1 it is clear that  ${}^A T_B$  represents relative pose

$$\xi(x, y, \theta) \sim \begin{pmatrix} \cos \theta & \sin \theta & x \\ -\sin \theta & \cos \theta & y \\ 0 & 0 & 1 \end{pmatrix}$$

A concrete representation of relative pose  $\xi$  is  $\xi \sim T \in SE(2)$  and  $T_1 \oplus T_2 \mapsto T_1 T_2$  which is standard matrix multiplication.

$$T_1 T_2 = \begin{pmatrix} R_1 & t_1 \\ \mathbf{0}_{1 \times 2} & 1 \end{pmatrix} \begin{pmatrix} R_2 & t_2 \\ \mathbf{0}_{1 \times 2} & 1 \end{pmatrix} = \begin{pmatrix} R_1 R_2 & t_1 + R_1 t_2 \\ \mathbf{0}_{1 \times 2} & 1 \end{pmatrix}$$

One of the algebraic rules from page 18 is  $\xi \oplus 0 = \xi$ . For matrices we know that  $TI = T$ , where  $I$  is the identity matrix, so for pose  $0 \mapsto I$  the identity matrix. Another rule was that  $\xi \ominus \xi = 0$ . We know for matrices that  $TT^{-1} = I$  which implies that  $\ominus T \mapsto T^{-1}$

$$T^{-1} = \begin{pmatrix} R & t \\ \mathbf{0}_{1 \times 2} & 1 \end{pmatrix}^{-1} = \begin{pmatrix} R^T & -R^T t \\ \mathbf{0}_{1 \times 2} & 1 \end{pmatrix}$$

For a point  $\tilde{p} \in \mathbb{P}^2$  then  $T \cdot \tilde{p} \mapsto T\tilde{p}$  which is a standard matrix-vector product.

To make this more tangible we will show some numerical examples using MATLAB® and the Toolbox. We create a homogeneous transformation using the function `se2`

```
>> T1 = se2(1, 2, 30*pi/180)
T1 =
    0.8660   -0.5000    1.0000
    0.5000    0.8660    2.0000
        0         0    1.0000
```

which represents a translation of  $(1, 2)$  and a rotation of  $30^\circ$ . We can plot this, relative to the world coordinate frame, by

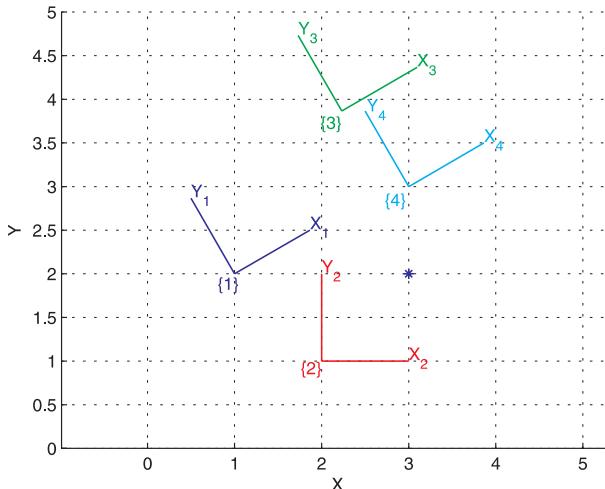


Fig. 2.8.

Coordinate frames drawn using the Toolbox function `trplot2`

```
>> axis([0 5 0 5]);
>> trplot2(T1, 'frame', '1', 'color', 'b')
```

The options specify that the label for the frame is {1} and it is colored blue and this is shown in Fig. 2.8. We create another relative pose which is a displacement of (2, 1) and zero rotation

```
>> T2 = se2(2, 1, 0)
T2 =
    1     0     2
    0     1     1
    0     0     1
```

which we plot in red

```
>> hold on
>> trplot2(T2, 'frame', '2', 'color', 'r');
```

Now we can compose the two relative poses

```
>> T3 = T1*T2
T3 =
    0.8660   -0.5000   2.2321
    0.5000    0.8660   3.8660
    0           0       1.0000
```

and plot it, in green, as

```
>> trplot2(T3, 'frame', '3', 'color', 'g');
```

We see that the displacement of (2, 1) has been applied with respect to frame {1}. It is important to note that our final displacement is not (3, 3) because the displacement is with respect to the rotated coordinate frame. The non-commutativity of composition is clearly demonstrated by

```
>> T4 = T2*T1;
>> trplot2(T4, 'frame', '4', 'color', 'c');
```

and we see that frame {4} is different to frame {3}.

Now we define a point (3, 2) relative to the world frame

```
>> P = [3 ; 2];
```

which is a column vector and add it to the plot

```
>> plot_point(P, '*');
```

To determine the coordinate of the point with respect to  $\{1\}$  we use Eq. 2.1 and write down

$${}^0 p = {}^0 \xi_1 \cdot {}^1 p$$

and then rearrange as

$$\begin{aligned} {}^1 p &= {}^1 \xi_0 \cdot {}^0 p \\ &= ({}^0 \xi_1)^{-1} \cdot {}^0 p \end{aligned}$$

Substituting numerical values

```
>> P1 = inv(T1) * [P; 1]
P1 =
    1.7321
   -1.0000
    1.0000
```

where we first converted the Euclidean point to *homogeneous form* by appending a one. The result is also in homogeneous form and has a negative  $y$ -coordinate in frame  $\{1\}$ . Using the Toolbox we could also have expressed this as

```
>> h2e( inv(T1) * e2h(P) )
ans =
    1.7321
   -1.0000
```

where the result is in Euclidean coordinates. The helper function `e2h` converts Euclidean coordinates to homogeneous and `h2e` performs the inverse conversion. More compactly this can be written as

```
>> homtrans( inv(T1), P)
ans =
    1.7321
   -1.0000
```

The same point with respect to  $\{2\}$  is

```
>> P2 = homtrans( inv(T2), P)
P2 =
    1
    1
```

## 2.2 Representing Pose in 3-Dimensions

The 3-dimensional case is an extension of the 2-dimensional case discussed in the previous section. We add an extra coordinate axis, typically denoted by  $z$ , that is orthogonal to both the  $x$ - and  $y$ -axes. The direction of the  $z$ -axis obeys the *right-hand rule* and forms a *right-handed coordinate frame*. Unit vectors parallel to the axes are denoted  $\hat{x}, \hat{y}$  and  $\hat{z}$  such that ▶

$$\hat{z} = \hat{x} \times \hat{y}, \quad \hat{x} = \hat{y} \times \hat{z}; \quad \hat{y} = \hat{z} \times \hat{x} \quad (2.11)$$

In all these identities, the symbols from left to right (across the equals sign) are a cyclic permutation of the sequence  $xyz$ .

A point  $P$  is represented by its  $x$ -,  $y$ - and  $z$ -coordinates  $(x, y, z)$  or as a bound vector

$$P = x\hat{x} + y\hat{y} + z\hat{z}$$

Figure 2.9 shows a coordinate frame  $\{B\}$  that we wish to describe with respect to the reference frame  $\{A\}$ . We can see clearly that the origin of  $\{B\}$  has been displaced by the vector  $t = (x, y, z)$  and then rotated in some complex fashion. Just as for the 2-dimensional case the way we represent orientation is very important.

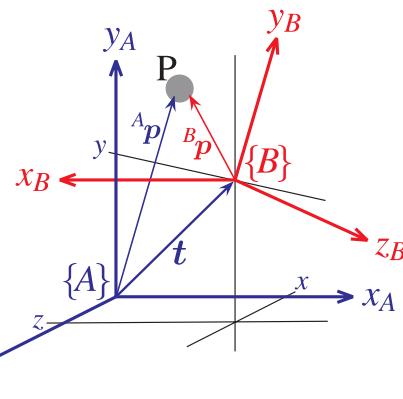


Fig. 2.9.

Two 3D coordinate frames  $\{A\}$  and  $\{B\}$ .  $\{B\}$  is rotated and translated with respect to  $\{A\}$

Our approach is to again consider an arbitrary point  $P$  with respect to each of the coordinate frames and to determine the relationship between  ${}^A p$  and  ${}^B p$ . We will again consider the problem in two parts: rotation and then translation. Rotation is surprisingly complex for the 3-dimensional case and we devote all of the next section to it.

## 2.2.1 Representing Orientation in 3-Dimensions

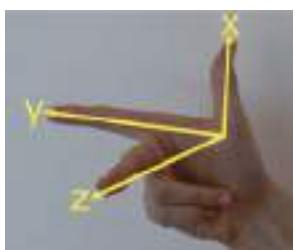
*Any two independent orthonormal coordinate frames can be related by a sequence of rotations (not more than three) about coordinate axes, where no two successive rotations may be about the same axis.*  
Euler's rotation theorem (Kuipers 1999).

Figure 2.9 showed a pair of right-handed coordinate frames with very different orientations, and we would like some way to describe the orientation of one with respect to the other. We can imagine picking up frame  $\{A\}$  in our hand and rotating it until it looked just like frame  $\{B\}$ . Euler's rotation theorem states that any rotation can be considered as a sequence of rotations about different coordinate axes.

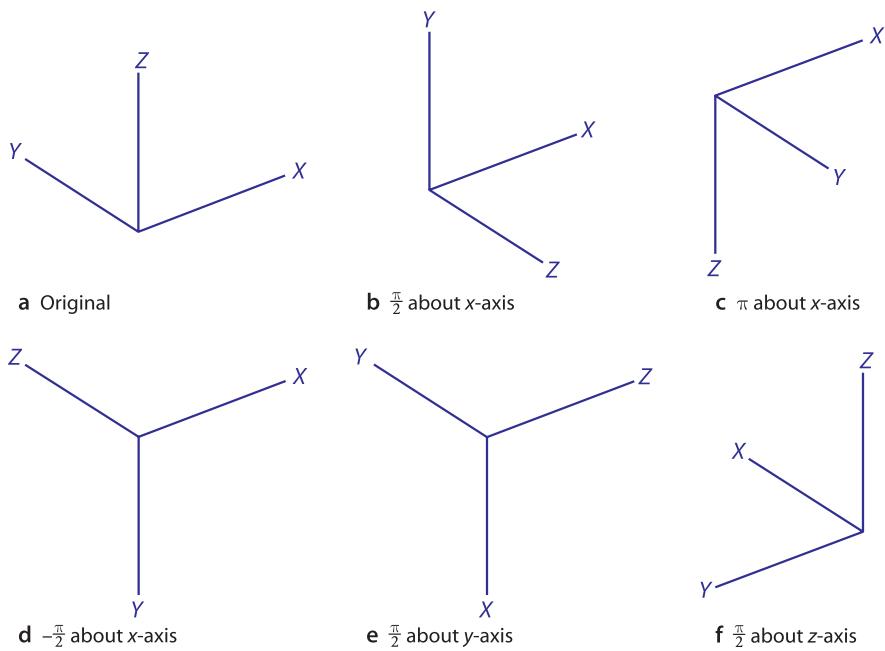
We start by considering rotation about a single coordinate axis. Figure 2.10 shows a right-handed coordinate frame, and that same frame after it has been rotated by various angles about different coordinate axes.

The issue of rotation has some subtleties which are illustrated in Fig. 2.11. This shows a sequence of two rotations applied in different orders. We see that the final orientation depends on the order in which the rotations are applied. This is a deep and confounding characteristic of the 3-dimensional world which has intrigued mathematicians for a long time. The implication for the pose algebra we have used in this chapter is that the  $\oplus$  operator is not commutative – the order in which rotations are applied is very important.

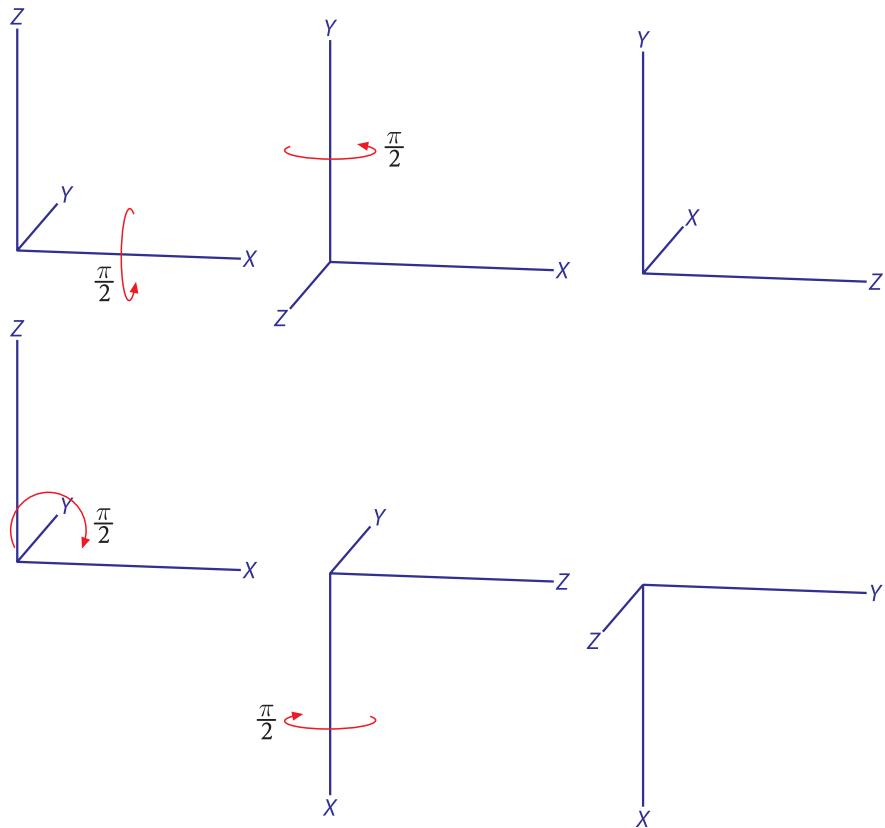
Mathematicians have developed many ways to represent rotation and we will discuss several of them in the remainder of this section: orthonormal rotation matrices, Euler and Cardan angles, rotation axis and angle, and unit quaternions. All can be represented as vectors or matrices, the natural datatypes of MATLAB® or as a Toolbox defined class. The Toolbox provides many function to convert between these representations and this is shown diagrammatically in Fig. 2.15.



**Right-hand rule.** A right-handed coordinate frame is defined by the first three fingers of your right hand which indicate the relative directions of the  $x$ -,  $y$ - and  $z$ -axes respectively.

**Fig. 2.10.**

Rotation of a 3D coordinate frame. a The original coordinate frame, b-f frame a after various rotations as indicated

**Fig. 2.11.**

Example showing the non-commutativity of rotation. In the top row the coordinate frame is rotated by  $\frac{\pi}{2}$  about the x-axis and then  $\frac{\pi}{2}$  about the y-axis. In the bottom row the order of rotations has been reversed. The results are clearly different

**Rotation about a vector.** Wrap your right hand around the vector with your thumb (your x-finger) in the direction of the arrow. The curl of your fingers indicates the direction of increasing angle.



### 2.2.1.1 Orthonormal Rotation Matrix

Just as for the 2-dimensional case we can represent the orientation of a coordinate frame by its unit vectors expressed in terms of the reference coordinate frame. Each unit vector has three elements and they form the columns of a  $3 \times 3$  *orthonormal matrix*  ${}^A\mathbf{R}_B$

$$\begin{pmatrix} {}^A\mathbf{x} \\ {}^A\mathbf{y} \\ {}^A\mathbf{z} \end{pmatrix} = {}^A\mathbf{R}_B \begin{pmatrix} {}^B\mathbf{x} \\ {}^B\mathbf{y} \\ {}^B\mathbf{z} \end{pmatrix} \quad (2.12)$$

which rotates a vector defined with respect to frame  $\{B\}$  to a vector with respect to  $\{A\}$ . The matrix  $\mathbf{R}$  belongs to the special orthogonal group of dimension 3 or  $\mathbf{R} \in SO(3) \subset \mathbb{R}^{3 \times 3}$ . It has the properties of an orthonormal matrix that were mentioned on page 16 such as  $\mathbf{R}^T = \mathbf{R}^{-1}$  and  $\det(\mathbf{R}) = 1$ .

The orthonormal rotation matrices for rotation of  $\theta$  about the  $x$ -,  $y$ - and  $z$ -axes are

$$\begin{aligned} R_x(\theta) &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \\ R_y(\theta) &= \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \\ R_z(\theta) &= \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

The Toolbox provides functions to compute these elementary rotation matrices, for example  $R_x(\theta)$  is

```
>> R = rotx(pi/2)
R =
    1.0000      0      0
    0     0.0000   -1.0000
    0     1.0000     0.0000
```

Such a rotation is also shown graphically in Fig. 2.10b. The functions `roty` and `rotz` compute  $R_y(\theta)$  and  $R_z(\theta)$  respectively.

The corresponding coordinate frame can be displayed graphically

```
>> trplot(R)
```

which is shown in Fig. 2.12a. We can visualize a rotation more powerfully using the Toolbox function `tranimate` which animates a rotation

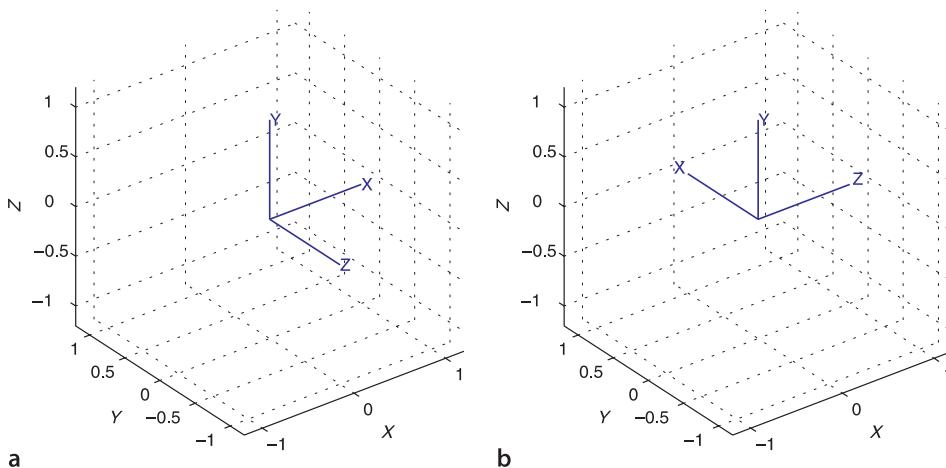
```
>> tranimate(R)
```

showing the world frame rotating into the specified coordinate frame.

**Reading the columns of an orthonormal *rotation matrix*** from left to right tells us the directions of the new frame's axes in terms of the current coordinate frame. For example if

```
R =
    1.0000      0      0
    0     0.0000   -1.0000
    0     1.0000     0.0000
```

the new frame has its  $x$ -axis in the old  $x$ -direction  $(1, 0, 0)$ , its  $y$ -axis in the old  $z$ -direction  $(0, 0, 1)$ , and the new  $z$ -axis in the old negative  $y$ -direction  $(0, -1, 0)$ . In this case the  $x$ -axis was unchanged since this is the axis around which the rotation occurred.



**Fig. 2.12.**  
Coordinate frames displayed using `trplot`. **a** Reference frame rotated by  $\frac{\pi}{2}$  about the  $x$ -axis, **b** frame **a** rotated by  $\frac{\pi}{2}$  about the  $y$ -axis

To illustrate compounding of rotations we will rotate the frame of Fig. 2.12a again, this time around its  $y$ -axis

```
>> rotx(pi/2) * roty(pi/2)
ans =
    0.0000      0      1.0000
    1.0000      0.0000     -0.0000
   -0.0000      1.0000      0.0000
>> trplot(R)
```

to give the frame shown in Fig. 2.12b. In this frame the  $x$ -axis now points in the direction of the world  $y$ -axis.

The non-commutativity of rotation can be shown by reversing the order of the rotations above

```
>> roty(pi/2)*rotx(pi/2)
ans =
    0.0000      1.0000      0.0000
        0      0.0000     -1.0000
   -1.0000      0.0000      0.0000
```

which has a very different value.

We recall that Euler's rotation theorem states that *any* rotation can be represented by *not more than three* rotations about coordinate axes. This means that in general an arbitrary rotation between frames can be decomposed into a sequence of three rotation angles and associated rotation axes – this is discussed in the next section.

The orthonormal matrix has nine elements but they are not independent. The columns have unit magnitude which provides three constraints. The columns are orthogonal to each other which provides another three constraints. ▶ Nine elements and six constraints is effectively three independent values.

If the column vectors are  $c_i$ ,  $i \in 1 \dots 3$   
then  $c_1 \cdot c_2 = c_2 \cdot c_3 = c_3 \cdot c_1 = 0$ .

### 2.2.1.2 Three-Angle Representations

Euler's rotation theorem requires successive rotation about three axes such that no two successive rotations are about the same axis. There are two classes of rotation sequence: Eulerian and Cardanian, named after Euler and Cardano respectively.

The Eulerian type involves repetition, but not successive, of rotations about one particular axis: XYX, XZX, YXY, YZY, ZXZ, or ZYZ. The Cardanian type is characterized by rotations about all three axes: XYZ, XZY, YZX, YXZ, ZXY, or ZYX. In common usage all these sequences are called Euler angles and there are a total of twelve to choose from.



**Leonhard Euler (1707–1783)** was a Swiss mathematician and physicist who dominated eighteenth century mathematics. He was a student of Johann Bernoulli and applied new mathematical techniques such as calculus to many problems in mechanics and optics. He also developed the functional notation,  $y = F(x)$ , that we use today. In robotics we use his rotation theorem and his equations of motion in rotational dynamics.

He was prolific and his collected works fill 75 volumes. Almost half of this was produced during the last seventeen years of his life when he was completely blind.

It is common practice to refer to all 3-angle representations as Euler angles but this is underspecified since there are twelve different types to choose from. The particular angle sequence is often a convention within a particular technological field.

The ZYZ sequence

$$R = R_z(\phi)R_y(\theta)R_z(\psi) \quad (2.13)$$

is commonly used in aeronautics and mechanical dynamics, and is used in the Toolbox. The Euler angles are the 3-vector  $\Gamma = (\phi, \theta, \psi)$ .

For example, to compute the equivalent rotation matrix for  $\Gamma = (0.1, 0.2, 0.3)$  we write

```
>> R = rotz(0.1) * roty(0.2) * rotz(0.3);
```

or more conveniently

```
>> R = eul2r(0.1, 0.2, 0.3)
R =
    0.9021   -0.3836    0.1977
    0.3875    0.9216    0.0198
   -0.1898    0.0587    0.9801
```

The inverse problem is finding the Euler angles that correspond to a given rotation matrix

```
>> gamma = tr2eul(R)
gamma =
    0.1000    0.2000    0.3000
```

However if  $\theta$  is negative

```
>> R = eul2r(0.1, -0.2, 0.3)
R =
    0.9021   -0.3836   -0.1977
    0.3875    0.9216   -0.0198
   -0.1898   -0.0587    0.9801
```

the inverse function

```
>> tr2eul(R)
ans =
   -3.0416    0.2000   -2.8416
```

returns a positive value for  $\theta$  and quite different values for  $\phi$  and  $\psi$ . However the corresponding rotation matrix

```
>> eul2r(ans)
ans =
    0.9021   -0.3836   -0.1977
    0.3875    0.9216   -0.0198
   -0.1898   -0.0587    0.9801
```

is the same – the two different sets of Euler angles correspond to the one rotation matrix. The mapping from rotation matrix to Euler angles is not unique and *always* returns a positive angle for  $\theta$ .

**Gerolamo Cardano** (1501–1576) was an Italian Renaissance mathematician, physician, astrologer, and gambler. He was born in Pavia, Italy, the illegitimate child of a mathematically gifted lawyer. He studied medicine at the University of Padua and later was the first to describe typhoid fever. He partly supported himself through gambling and his book about games of chance *Liber de ludo aleae* contains the first systematic treatment of probability as well as effective cheating methods. His family life was problematic: his eldest son was executed for poisoning his wife, and his daughter was a prostitute who died from syphilis about which he wrote a treatise. He computed and published the horoscope of Jesus, was accused of heresy, and spent time in prison until he abjured and gave up his professorship. He died on the day he had predicted astrologically.

He published the solutions to the cubic and quartic equations in his book *Ars magna* in 1545, and also invented the combination lock, the gimbal consisting of three concentric rings allowing a compass or gyroscope to rotate freely (see Fig. 2.13), and the Cardan shaft with universal joints which is used in vehicles today.



For the case where  $\theta = 0$

```
>> R = euler(0.1, 0, 0.3)
R =
0.9211 -0.3894 0
0.3894 0.9211 0
0 0 1.0000
```

the inverse function returns

```
>> tr2eul(R)
ans =
0 0 0.4000
```

which is quite different. For this case the rotation matrix from Eq. 2.13 is

$$R = R_z(\phi)R_z(\psi) = R_z(\phi + \psi)$$

since  $R_y = I$  and is a function only of the sum  $\phi + \psi$ . For the inverse operation we can therefore only determine this sum, and by convention  $\phi = 0$ . The case  $\theta = 0$  is a singularity and will be discussed in more detail in the next section.

Another widely used convention is the roll-pitch-yaw angle sequence angle

$$R = R_x(\theta_r)R_y(\theta_p)R_z(\theta_y) \quad (2.14)$$

which are intuitive when describing the attitude of vehicles such as ships, aircraft and cars. Roll, pitch and yaw (also called bank, attitude and heading) refer to rotations about the  $x$ -,  $y$ -,  $z$ -axes, respectively. This XYZ angle sequence, technically Cardan angles, are also known as Tait-Bryan angles<sup>▶</sup> or nautical angles. For aerospace and ground vehicles the  $x$ -axis is commonly defined in the forward direction,  $z$ -axis downward and the  $y$ -axis to the right-hand side.<sup>▶</sup> For example

```
>> R = rpy2r(0.1, 0.2, 0.3)
R =
0.9752 -0.0370 0.2184
0.0978 0.9564 -0.2751
-0.1987 0.2896 0.9363
```

and the inverse is

```
>> gamma = tr2rpy(R)
gamma =
0.1000 0.2000 0.3000
```

The roll-pitch-yaw sequence allows all angles to have arbitrary sign and it has a singularity when  $\theta_p = \pm\frac{\pi}{2}$  which is fortunately outside the range of feasible attitudes for most vehicles.

Named after Peter Tait a Scottish physicist and quaternion supporter, and George Bryan an early Welsh aerodynamicist.

Note that most robotics texts (Paul 1981; Siciliano et al. 2008; Spong et al. 2006) swap the  $x$ - and  $z$ -axes by defining the heading direction as the  $z$ - rather than  $x$ -direction, that is roll is about the  $z$ -axis not the  $x$ -axis. This was the convention in the Robotics Toolbox prior to Release 8. The default in the Toolbox is now XYZ order but the ZYX order can be specified with the 'zyx' option.

### 2.2.1.3 Singularities and Gimbal Lock

*"The LM Body coordinate system is right-handed, with the +X axis pointing up through the thrust axis, the +Y axis pointing right when facing forward which is along the +Zaxis. The rotational transformation matrix is constructed by a 2-3-1 Euler sequence, that is: Pitch about Y, then Roll about Z and, finally, Yaw about X. Positive rotations are pitch up, roll right, yaw left." (Hoag 1963).*

Operationally this was a significant limiting factor with this particular gyroscope (Hoag 1963) and could have been alleviated by adding a fourth gimbal, as was used on other spacecraft. It was omitted on the Lunar Module for reasons of weight and space.

A fundamental problem with the three-angle representations just described is singularity. This occurs when the rotational axis of the middle term in the sequence becomes parallel to the rotation axis of the first or third term. This is the same problem as gimbal lock, a term made famous in the movie Apollo 13.

A mechanical gyroscope used for navigation such as shown in Fig. 2.13 has as its innermost assembly three orthogonal gyroscopes which hold the *stable member* stationary with respect to the universe. It is mechanically connected to the spacecraft via a gimbal mechanism which allows the spacecraft to move around the stable platform without exerting any torque on it. The attitude of the spacecraft is determined by measuring the angles of the gimbal axes with respect to the stable platform – giving a direct indication of roll-pitch-yaw angles which in this design are a Cardanian YZX sequence. ▶

Consider the situation when the rotation angle of the middle gimbal (rotation about the spacecraft's z-axis) is  $90^\circ$  – the axes of the inner and outer gimbals are aligned and they share the *same* rotation axis. Instead of the original three rotational axes, since two are parallel, there are now only two effective rotational axes – we say that one degree of freedom has been lost. ▶

In mathematical, rather than mechanical, terms this problem can be seen using the definition of the Lunar module's coordinate system where the rotation of the spacecraft's body-fixed frame  $\{B\}$  with respect to the stable platform frame  $\{S\}$  is

$${}^S R_B = R_y(\theta_p) R_z(\theta_r) R_x(\theta_y)$$

Rotations obey the cyclic rotation rules

$$Rx\left(\frac{\pi}{2}\right) Ry(\theta) Rx\left(\frac{\pi}{2}\right)^T \equiv Rz(\theta)$$

$$Ry\left(\frac{\pi}{2}\right) Rz(\theta) Ry\left(\frac{\pi}{2}\right)^T \equiv Rx(\theta)$$

$$Rz\left(\frac{\pi}{2}\right) Rx(\theta) Rz\left(\frac{\pi}{2}\right)^T \equiv Ry(\theta)$$

and anti-cyclic rotation rules

$$Ry\left(\frac{\pi}{2}\right)^T Rx(\theta) Ry\left(\frac{\pi}{2}\right) \equiv Rz(\theta)$$

$$Rz\left(\frac{\pi}{2}\right)^T Ry(\theta) Rz\left(\frac{\pi}{2}\right) \equiv Rx(\theta).$$

For the case when  $\theta_r = \frac{\pi}{2}$  we can apply the identity ▶

$$R_y(\theta) R_z\left(\frac{\pi}{2}\right) \equiv R_z\left(\frac{\pi}{2}\right) R_x(\theta)$$

leading to

$${}^S R_B = R_z\left(\frac{\pi}{2}\right) R_x(\theta_p) R_x(\theta_y) = R_z\left(\frac{\pi}{2}\right) R_x(\theta_p + \theta_y)$$

which cannot represent rotation about the  $y$ -axis. This is not a good thing because spacecraft rotation about the  $y$ -axis will rotate the stable element and thus ruin its precise alignment with the stars.

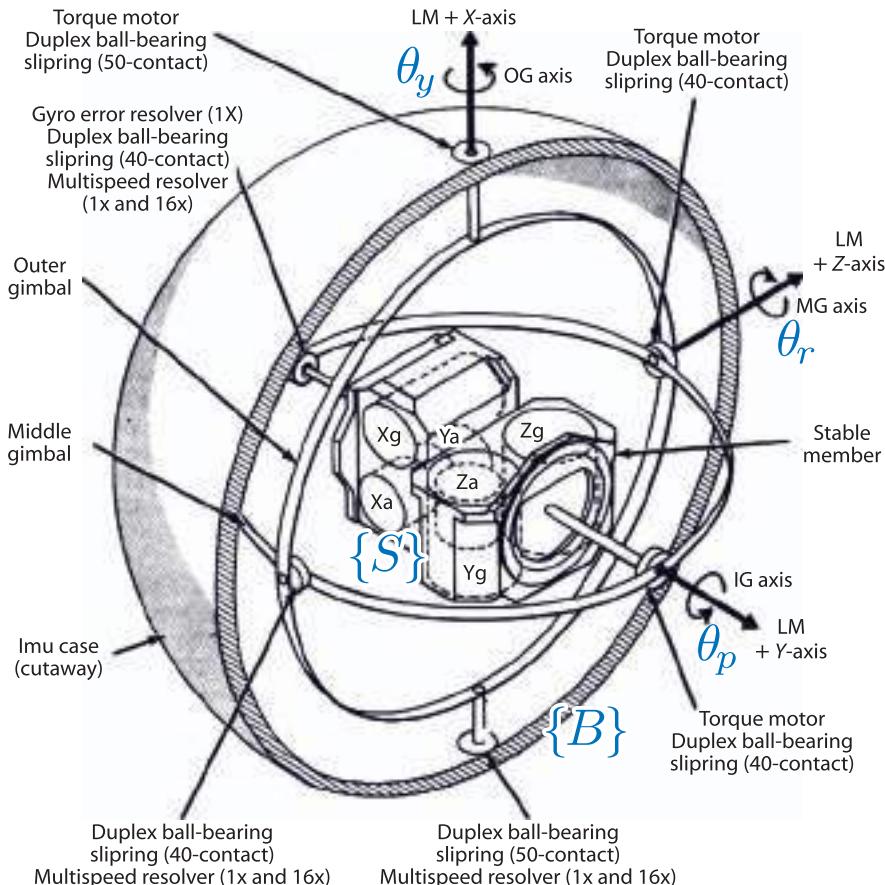
The loss of a degree of freedom means that mathematically we cannot invert the transformation, we can only establish a linear relationship between two of the angles. In such a case the best we can do is determine the sum of the pitch and yaw angles. We observed a similar phenomena with the Euler angle singularity earlier.



Mission clock: 02 08 12 47

- Flight: "Go, Guidance."
- Guido: "He's getting close to gimbal lock there."
- Flight: "Roger. CapCom, recommend he bring up C3, C4, B3, B4, C1 and C2 thrusters, and advise he's getting close to gimbal lock."
- CapCom: "Roger."

Apollo 13, mission control communications loop (1970) (Lovell and Kluger 1994, p 131; NASA 1970).



**Fig. 2.13.**  
Schematic of Apollo Lunar Module (LM) inertial measurement unit (IMU). The vehicle's coordinate system has the  $y$ -axis pointing up through the thrust axis, the  $z$ -axis forward, and the  $x$ -axis pointing right. Starting at the stable platform  $\{S\}$  and working outwards toward the spacecraft's body frame  $\{B\}$  the rotation angle sequence is  $YZX$ . The components labelled  $X_g$ ,  $Y_g$  and  $Z_g$  are the  $x$ -,  $y$ - and  $z$ -axis gyroscopes and those labelled  $X_a$ ,  $Y_a$  and  $Z_a$  are the  $x$ -,  $y$ - and  $z$ -axis accelerometers (Apollo Operations Handbook, LMA790-3-LM)

All three-angle representations of attitude, whether Eulerian or Cardanian, suffer this problem of *gimbal lock* when two consecutive axes become aligned. For  $YZX$ -Euler angles this occurs when  $\theta = k\pi$ ,  $k \in \mathbb{Z}$  and for roll-pitch-yaw angles when pitch  $\theta_p = \pm(2k+1)\frac{\pi}{2}$ . The best that can be hoped for is that the singularity occurs for an attitude which does not occur during normal operation of the vehicle – it requires judicious choice of angle sequence and coordinate system.

Singularities are an unfortunate consequence of using a minimal representation. To eliminate this problem we need to adopt different representations of orientation. Many in the Apollo LM team would have preferred a four gimbal system and the clue to success, as we shall see shortly in Sect. 2.2.1.6, is to introduce a fourth parameter.

#### 2.2.1.4 Two Vector Representation

For arm-type robots it is useful to consider a coordinate frame  $\{E\}$  attached to the end-effector as shown in Fig. 2.14. By convention the axis of the tool is associated with the  $z$ -axis and is called the approach vector and denoted  $\hat{a} = (a_x, a_y, a_z)$ . For some applications it is more convenient to specify the approach vector than to specify Euler or roll-pitch-yaw angles.

However specifying the direction of the  $z$ -axis is insufficient to describe the coordinate frame – we also need to specify the direction of the  $x$ - and  $y$ -axes. An orthogonal vector that provides orientation, perhaps between the two fingers of the robot's gripper is called the *orientation vector*,  $\hat{o} = (o_x, o_y, o_z)$ . These two unit vectors are sufficient to completely define the rotation matrix

$$R = \begin{pmatrix} n_x & o_x & a_x \\ n_y & o_y & a_y \\ n_z & o_z & a_z \end{pmatrix} \quad (2.15)$$

So long as they are not parallel.

since the remaining column can be computed using Eq. 2.11 as  $\hat{n} = \hat{o} \times \hat{a}$ .

Even if the two vectors  $\hat{a}$  and  $\hat{o}$  are not orthogonal they still define a plane $\blacktriangleleft$  and the computed  $\hat{n}$  is normal to that plane. In this case we need to compute a new value for  $\hat{o}' = \hat{a} \times \hat{n}$  which lies in the plane but is orthogonal to each of  $\hat{a}$  and  $\hat{n}$ .

Using the Toolbox this is implemented by

```
>> a = [1 0 0]';
>> o = [0 1 0]';
>> oa2r(o, a)
ans =
    0      0      1
    0      1      0
   -1      0      0
```

Any two non-parallel vectors are sufficient to define a coordinate frame. For a camera we might use the optical axis, by convention the  $z$ -axis, and the left side of the camera which is by convention the  $x$ -axis. For a mobile robot we might use the gravitational acceleration vector (measured with accelerometers) which is by convention the  $z$ -axis and the heading direction (measured with an electronic compass) which is by convention the  $x$ -axis.

### 2.2.1.5 Rotation about an Arbitrary Vector

Two coordinate frames of arbitrary orientation are related by a *single* rotation about some axis in space. For the example rotation used earlier

```
>> R = rpy2r(0.1, 0.2, 0.3);
```

we can determine such an angle and vector by

```
>> [theta, v] = tr2angvec(R)
theta =
    0.3816
v =
    0.3379    0.4807    0.8092
```

where `theta` is the amount of rotation and `v` is the vector around which the rotation occurs.

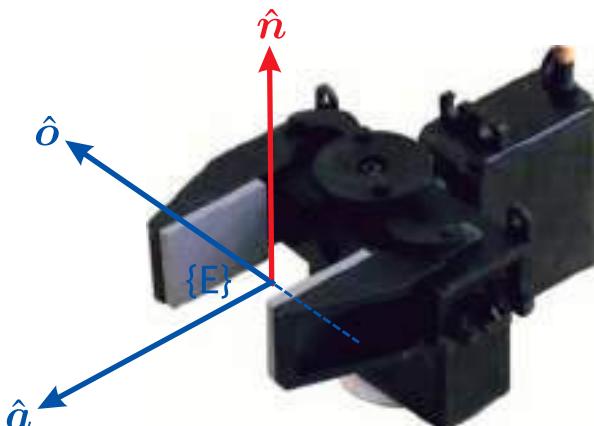


Fig. 2.14.

Robot end-effector coordinate system defines the pose in terms of an *approach* vector  $\hat{a}$  and an *orientation* vector  $\hat{o}$ , from which  $\hat{n}$  can be computed (courtesy of lynxmotion.com)

This information is encoded in the eigenvalues and eigenvectors of  $R$ . Using the builtin MATLAB® function `eig`

```
>> [v,lambda] = eig(R)
v =
    0.6655      0.6655      0.3379
   -0.1220 - 0.6079i  -0.1220 + 0.6079i  0.4807
   -0.2054 + 0.3612i  -0.2054 - 0.3612i  0.8092
lambda =
    0.9281 + 0.3724i      0            0
            0      0.9281 - 0.3724i      0
            0            0            1.0000
```

the eigenvalues are returned on the diagonal of the matrix `lambda` and the corresponding eigenvectors are the corresponding columns of `v`.

An orthonormal rotation matrix will always have one real eigenvalue at  $\lambda = 1$  and a complex pair  $\lambda = \cos \theta \pm i \sin \theta$  where  $\theta$  is the rotation angle. From the definition of eigenvalues and eigenvectors we recall that

$$Rv = \lambda v$$

where  $v$  is the eigenvector corresponding to  $\lambda$ . For the case  $\lambda = 1$  then

$$Rv = v$$

which implies that the corresponding eigenvector  $v$  is *unchanged* by the rotation. There is only one such vector and that is the one *about which* the rotation occurs. In the example the third eigenvalue is equal to one, so the rotation axis is the third column of `v`.

Converting from angle and vector to a rotation matrix is achieved using Rodrigues' rotation formula ►

$S(\cdot)$  is the skew-symmetric matrix defined in Eq. 3.5 and also Appendix D.

$$R = I_{3 \times 3} + \sin \theta S(v) + (1 - \cos \theta)(vv^T - I_{3 \times 3})$$

Our familiar example of a rotation of  $\frac{\pi}{2}$  about the  $x$ -axis can be found by

```
>> R = angvec2r(pi/2, [1 0 0])
R =
    1.0000      0            0
            0      0.0000    -1.0000
            0      1.0000      0.0000
```

It is interesting to note that this representation of an arbitrary rotation is parameterized by four numbers: three for the rotation axis, and one for the angle of rotation. However the direction can be represented by a unit vector with only two parameters since the third element can be computed by

$$v_3 = \sqrt{1 - v_1^2 - v_2^2}$$

giving a total of three parameters. Alternatively we can multiply the unit vector by the angle to give another 3-parameter representation  $v\theta$ . While these forms are minimal

Olinde Rodrigues (1795–1850) was a French-Jewish banker and mathematician who wrote extensively on politics, social reform and banking. He received his doctorate in mathematics in 1816 from the University of Paris, for work on his first well known formula which is related to Legendre polynomials. His eponymous rotation formula was published in 1840 and is perhaps the first time the representation of a rotation as a scalar and a vector was articulated. His formula is sometimes, and inappropriately, referred to as the Euler-Rodrigues formula. He is buried in the Pere-Lachaise cemetery in Paris.



and efficient in terms of data storage they are analytically problematic. Many variants have been proposed including  $v \sin(\theta/2)$  and  $v \tan(\theta)$  but all are ill-defined for  $\theta=0$ .

### 2.2.1.6 Unit Quaternions

*Quaternions came from Hamilton after his really good work had been done; and, though beautifully ingenious, have been an unmixed evil to those who have touched them in any way, including Clark Maxwell.*  
Lord Kelvin, 1892

Quaternions have been controversial since they were discovered by W. R. Hamilton over 150 years ago but they have great utility for roboticists. The quaternion is an extension of the complex number – a hyper-complex number – and is written as a scalar plus a vector

$$\begin{aligned}\dot{q} &= s + v \\ &= s + v_1 i + v_2 j + v_3 k\end{aligned}\tag{2.16}$$

The multiplication rules are the same as for cross products of the orthogonal vectors  $\hat{x}$ ,  $\hat{y}$  and  $\hat{z}$ .

where  $s \in \mathbb{R}$ ,  $v \in \mathbb{R}^3$  and the orthogonal complex numbers  $i, j$  and  $k$  are defined such that

$$i^2 = j^2 = k^2 = ijk = -1\tag{2.17}$$

We will denote a quaternion as

$$\dot{q} = s < v_1, v_2, v_3 >$$

One early objection to quaternions was that multiplication was not commutative but as we have seen above this is exactly the case for rotations. Despite the initial controversy quaternions are elegant, powerful and computationally straightforward and widely used for robotics, computer vision, computer graphics and aerospace inertial navigation applications.

In the Toolbox quaternions are implemented by the `Quaternion` class. The constructor converts the passed argument to a quaternion, for example

```
>> q = Quaternion( rpy2tr(0.1, 0.2, 0.3) )
q =
0.98186 < 0.064071, 0.091158, 0.15344 >
```



**Sir William Rowan Hamilton (1805–1865)** was an Irish mathematician, physicist, and astronomer. He was a child prodigy with a gift for languages and by age thirteen knew classical and modern European languages as well as Persian, Arabic, Hindustani, Sanskrit, and Malay. Hamilton taught himself mathematics at age 17, and discovered an error in Laplace's Celestial Mechanics. He spent his life at Trinity College, Dublin, and was appointed Professor of Astronomy and Royal Astronomer of Ireland while still an undergraduate. In addition to quaternions he contributed to the development of optics, dynamics, and algebra. He also wrote poetry and corresponded with Wordsworth who advised him to devote his energy to mathematics.

According to legend the key quaternion equation, Eq. 2.17, occurred to Hamilton in 1843 while walking along the Royal Canal in Dublin with his wife, and this is commemorated by a plaque on Broome bridge:

*Here as he walked by on the 16<sup>th</sup> of October 1843 Sir William Rowan Hamilton in a flash of genius discovered the fundamental formula for quaternion multiplication  $i^2 = j^2 = k^2 = ijk = -1$  & cut it on a stone of this bridge.*

His original carving is no longer visible, but the bridge is a pilgrimage site for mathematicians and physicists.

To represent rotations we use unit-quaternions. These are quaternions of unit magnitude, that is, those for which  $|\dot{q}| = 1$  or  $s^2 + v_1^2 + v_2^2 + v_3^2 = 1$ . For our example

```
>> q.norm
ans =
1.0000
```

The unit-quaternion has the special property that it can be considered as a rotation of  $\theta$  about the unit vector  $\hat{n}$  which are related to the quaternion components by

$$s = \cos \frac{\theta}{2}, \quad \mathbf{v} = \left( \sin \frac{\theta}{2} \right) \hat{\mathbf{n}} \quad (2.18)$$

and is similar to the angle-axis representation of Sect. 2.2.1.5.

For the case of quaternions our generalized pose is  $\xi$  is  $\xi \sim \dot{q} \in \mathbb{Q}$  and

$$\dot{q}_1 \oplus \dot{q}_2 \mapsto s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, \quad < s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2 >$$

which is known as the quaternion or Hamilton product,<sup>►</sup> and

$$\ominus \dot{q} \mapsto \dot{q}^{-1} = s, < -\mathbf{v} >$$

which is the quaternion conjugate. The zero pose  $0 \mapsto 1 < 0, 0, 0 >$  which is the identity quaternion. A vector  $\mathbf{v} \in \mathbb{R}^3$  is rotated  $\dot{q} \cdot \mathbf{v} \mapsto \dot{q} \dot{q}(\mathbf{v}) \dot{q}^{-1}$  where  $\dot{q}(\mathbf{v}) = 0, < \mathbf{v} >$  is known as a pure quaternion.

If we write the quaternion as a 4-vector  $(s, v_1, v_2, v_3)$  then multiplication can be expressed as a matrix-vector product where

$$\dot{q} \oplus \dot{q}' = \begin{pmatrix} s & v_1 & v_2 & v_3 \\ -v_1 & s & -v_3 & v_2 \\ -v_2 & v_3 & s & -v_1 \\ -v_3 & -v_2 & v_1 & s \end{pmatrix} \begin{pmatrix} s' \\ v'_1 \\ v'_2 \\ v'_3 \end{pmatrix}$$

The [Quaternion](#) class overloads a number of standard methods and functions. Quaternion multiplication<sup>►</sup> is invoked through the overloaded multiplication operator

```
>> q = q * q;
```

and inversion, the quaternion conjugate, is

```
>> q.inv()
ans =
0.98186 < -0.064071, -0.091158, -0.15344 >
```

Multiplying a quaternion by its inverse

```
>> q*q.inv()
ans =
1 < 0, 0, 0 >
```

or

```
>> q/q
ans =
1 < 0, 0, 0 >
```

results in the identity quaternion which represents a null rotation.

The quaternion can be converted to an orthonormal rotation matrix by

```
>> q.R
ans =
0.9363   -0.2896    0.1987
0.3130    0.9447   -0.0978
-0.1593    0.1538    0.9752
```

and we can also plot the orientation represented by a quaternion

```
>> q.plot()
```

which produces a result similar in style to that shown in Fig. 2.12.

Compounding two orthonormal rotation matrices requires 27 multiplications and 18 additions. The quaternion form requires 16 multiplications and 12 additions. This saving can be particularly important for embedded systems.

A 3-vector passed to the constructor yields a pure quaternion

```
>> Quaternion([1 2 3])
ans =
0 < 1, 2, 3 >
```

which has a zero scalar component. A vector is rotated by a quaternion using the overloaded multiplication operator

```
>> q*[1 0 0]'
ans =
0.9363
0.3130
-0.1593
```

The Toolbox implementation is quite complete and the `Quaternion` has many methods and properties which are described fully in the online documentation.

## 2.2.2 Combining Translation and Orientation

We return now to representing relative pose in three dimensions, the position and orientation change, between two coordinate frames as shown in Fig. 2.9. We have discussed several different representations of orientation, and we need to combine this with translation, to create a tangible representation of relative pose. The two most practical representations are: the quaternion vector pair and the  $4 \times 4$  homogeneous transformation matrix.

For the vector-quaternion case  $\xi \sim (t, \dot{q})$  where  $t \in \mathbb{R}^3$  is the Cartesian position of the frame's origin with respect to the reference coordinate frame, and  $\dot{q} \in \mathbb{Q}$  is the frame's orientation with respect to the reference frame.

Composition is defined by

$$\xi_1 \oplus \xi_2 = (t_1 + \dot{q}_1 \cdot t_2, \dot{q}_1 \oplus \dot{q}_2)$$

and negation is

$$\ominus \xi = (-\dot{q}^{-1} \cdot t, \dot{q}^{-1})$$

and a point coordinate vector is transformed to a coordinate frame by

$${}^X p = {}^X \xi_Y \cdot {}^Y p = \dot{q} \cdot {}^Y p + t$$

Alternatively we can use a homogeneous transformation matrix to describe rotation and translation. The derivation is similar to the 2D case of Eq. 2.10 but extended to account for the  $z$ -dimension

$$\begin{pmatrix} {}^A x \\ {}^A y \\ {}^A z \\ 1 \end{pmatrix} = \begin{pmatrix} {}^A R_B & t \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix} \begin{pmatrix} {}^B x \\ {}^B y \\ {}^B z \\ 1 \end{pmatrix}$$

The Cartesian translation vector between the origin of the coordinates frames is  $t$  and the change in orientation is represented by a  $3 \times 3$  orthonormal submatrix  $R$ . The vectors are expressed in homogenous form and we write

$$\begin{aligned} {}^A\tilde{\mathbf{p}} &= \begin{pmatrix} {}^A\mathbf{R}_B & \mathbf{t} \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix} {}^B\tilde{\mathbf{p}} \\ &= {}^A\mathbf{T}_B {}^B\tilde{\mathbf{p}} \end{aligned} \quad (2.19)$$

and  ${}^A\mathbf{T}_B$  is a  $4 \times 4$  homogeneous transformation. The matrix has a very specific structure and belongs to the special Euclidean group of dimension 3 or  $\mathbf{T} \in SE(3) \subset \mathbb{R}^{4 \times 4}$ .

A concrete representation of relative pose  $\xi$  is  $\xi \sim \mathbf{T} \in SE(3)$  and  $\mathbf{T}_1 \oplus \mathbf{T}_2 \mapsto \mathbf{T}_1 \mathbf{T}_2$  which is standard matrix multiplication.

$$\mathbf{T}_1 \mathbf{T}_2 = \begin{pmatrix} \mathbf{R}_1 & \mathbf{t}_1 \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}_2 & \mathbf{t}_2 \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}_1 \mathbf{R}_2 & \mathbf{t}_1 + \mathbf{R}_1 \mathbf{t}_2 \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix} \quad (2.20)$$

One of the rules of pose algebra from page 18 is  $\xi \oplus 0 = \xi$ . For matrices we know that  $TI = T$ , where  $I$  is the identity matrix, so for pose  $0 \mapsto I$  the identity matrix. Another rule of pose algebra was that  $\xi \ominus \xi = 0$ . We know for matrices that  $TT^{-1} = I$  which implies that  $\ominus T \mapsto T^{-1}$

$$\mathbf{T}^{-1} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix}^{-1} = \begin{pmatrix} \mathbf{R}^T & -\mathbf{R}^T \mathbf{t} \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix} \quad (2.21)$$

The  $4 \times 4$  homogeneous transformation is very commonly used in robotics and computer vision, is supported by the Toolbox and will be used throughout this book as a concrete representation of 3-dimensional pose.

The Toolbox has many functions to create homogeneous transformations. For example we can demonstrate composition of transforms by

```
>> T = transl(1, 0, 0) * trotx(pi/2) * transl(0, 1, 0)
T =
1.0000      0      0    1.0000
0    0.0000   -1.0000    0.0000
0    1.0000    0.0000    1.0000
0      0      0    1.0000
```

The function `transl` creates a relative pose with a finite translation but no rotation, and `trotx` returns a  $4 \times 4$  homogeneous transform matrix corresponding to a rotation of  $\frac{\pi}{2}$  about the  $x$ -axis: the rotation part is the same as `rotx(pi/2)` and the translational component is zero.► We can think of this expression as representing a walk along the  $x$ -axis for 1 unit, then a rotation by  $90^\circ$  about the  $x$ -axis and then a walk of 1 unit along the new  $y$ -axis which was the previous  $z$ -axis. The result, as shown in the last column of the resulting matrix is a translation of 1 unit along the original  $x$ -axis and 1 unit along the original  $z$ -axis. The orientation of the final pose shows the effect of the rotation about the  $x$ -axis. We can plot the corresponding coordinate frame by

```
>> trplot(T)
```

The rotation matrix component of `T` is

```
>> t2r(T)
ans =
1.0000      0      0
0    0.0000   -1.0000
0    1.0000    0.0000
```

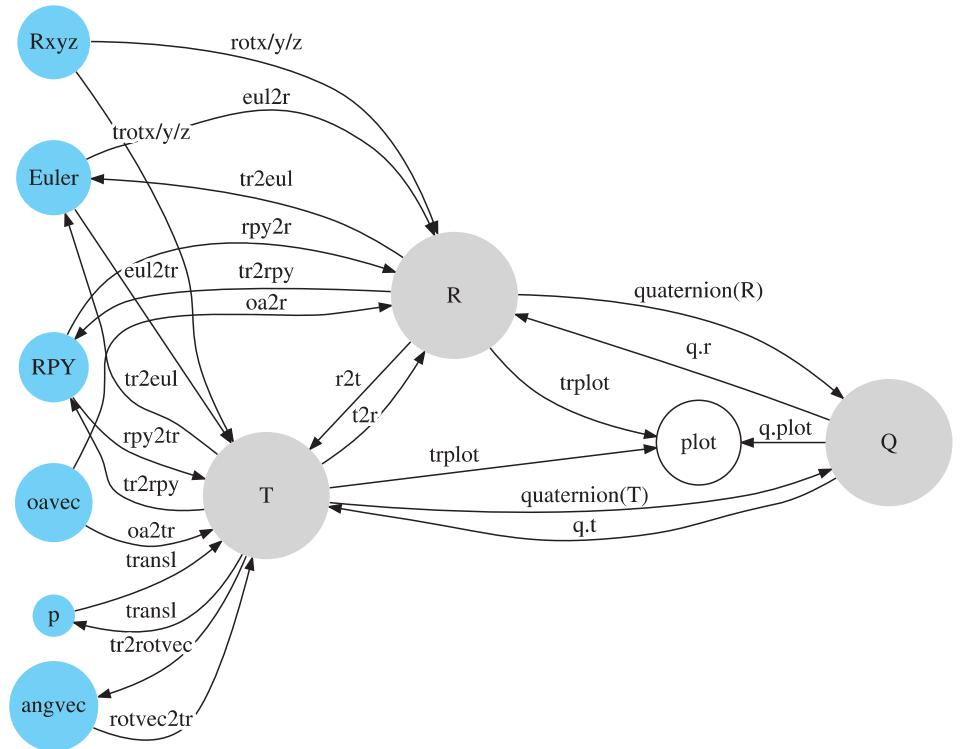
and the translation component is a vector

```
>> transl(T)'
ans =
1.0000    0.0000    1.0000
```

Many Toolbox functions have variants that return orthonormal rotation matrices or homogeneous transformations, for example, `rotx` and `trotx`, `rpy2r` and `rpy2tr` etc. Some Toolbox functions accept an orthonormal rotation matrix or a homogeneous transformation and ignore the translational component, for example, `tr2rpy` or `Quaternion`.

**Table 2.1.**  
Summary of the various concrete representations of pose  $\xi$  introduced in this chapter

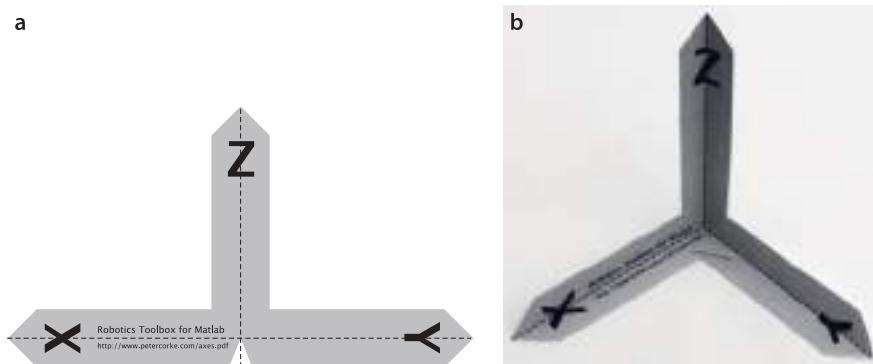
Representation	$\oplus$	$\ominus$	transl.	rotn.	dim	MATLAB
$(x, y, \theta) \in \mathbb{R}^2 \times \mathbb{S}$			✓	✓	2D	
$T \in SE(2)$	$T_1 T_2$	$T^{-1}$	✓	✗	2D	<code>se2(x, y)</code>
$R \in SO(2)$	$R_1 R_2$	$R^T$	✗	✓	2D	<code>se2(0, 0, th)</code>
$T \in SE(2)$	$T_1 T_2$	$T^{-1}$	✓	✓	2D	<code>se2(x, y, th)</code>
$(x, y, z, \Gamma) \in \mathbb{R}^3 \times \mathbb{S}^3$			✓	✓	3D	
$R \in SO(3)$	$R_1 R_2$	$R^T$	✗	✓	3D	<code>rotx, roty, ...</code>
$\Gamma \in \mathbb{S}^3$	✗			✓	3D	<code>tr2eul, eul2tr</code>
$\Gamma \in \mathbb{S}^3$	✗			✓	3D	<code>tr2rpy1, rpy2tr</code>
$T \in SE(3)$	$T_1 T_2$	$T^{-1}$	✓	✓	3D	<code>transl(x,y,z)</code>
$\dot{q} \in \mathbb{Q}$	$\dot{q}_1 \dot{q}_2$	$\dot{q}^{-1}$	✗	✓	3D	<code>quaternion</code>



**Fig. 2.15.**  
Conversion between rotational representations

## 2.3 Wrapping Up

In this chapter we learned how to represent points and poses in 2- and 3-dimensional worlds. Points are represented by coordinate vectors relative to a coordinate frame. A set of points that belong to a rigid object can be described by a coordinate frame, and its constituent points are described by displacements from the object's coordinate frame. The position and orientation of any coordinate frame can be described relative to another coordinate frame by its relative pose  $\xi$ . Relative poses can be applied sequentially (composed or compounded), and we have shown how relative poses can be manipulated algebraically. An important algebraic rule is that composition is non-commutative – the order in which relative poses are applied is important.

**Fig. 2.16.**

Build your own coordinate frame.  
**a** Get the PDF file from <http://www.petercorke.com/axes.pdf>;  
**b** cut it out, fold along the dotted lines and add a staple. Voila!

We have explored orthonormal rotation matrices for the 2- and 3-dimensional case to represent orientation and its extension, the homogeneous transformation matrix, to represent orientation and translation. Rotation in 3-dimensions has subtlety and complexity and we have looked at other representations such as Euler angles, roll-pitch-yaw angles and quaternions. Some of these mathematical objects are supported natively by MATLAB® while others are supported by functions or classes within the Toolbox.

There are two important lessons from this chapter. The first is that there are *many* mathematical objects that can be used to represent pose and these are summarized in Table 2.1. There is no right or wrong – each has strengths and weaknesses and we typically choose the representation to suit the problem at hand. Sometimes we wish for a vectorial representation in which case  $(x, y, \theta)$  or  $(x, y, z, \Gamma)$  might be appropriate, but this representation cannot be easily compounded. Sometime we may only need to describe 3D rotation in which case  $\Gamma$  or  $\dot{q}$  is appropriate. The Toolbox supports conversions between many different representations as shown in Fig. 2.15. In general though, we will use homogeneous transformations throughout the rest of this book.

The second lesson is that coordinate frames are your friend. The essential first step in many vision and robotics problems is to assign coordinate frames to all objects of interest, indicate the relative poses as a directed graph, and write down equations for the loops. Figure 2.16 shows you how to build a coordinate frame out of paper that you can pick up and rotate.

We now have solid foundations for moving forward. The notation has been defined and illustrated, and we have started our hands-on work with MATLAB®. The next chapter discusses coordinate frames that change with time, and after that we are ready to move on and discuss robots.

### Further Reading

The treatment in this chapter is a hybrid mathematical and graphical approach that covers the 2D and 3D cases by means of abstract representations and operators which are later made tangible. The standard robotics textbooks such as Spong et al. (2006), Craig (2004), Siciliano et al. (2008) and Paul (1981) all introduce homogeneous transformation matrices for the 3-dimensional case but differ in their approach. These books also provide good discussion of the other representations such as angle-vector and 3-angle representations. Spong et al. (2006, Sec 2.5.1) have a good discussion of singularities. Siegwart et al. (2011) explicitly cover the 2D case in the context of mobile robot navigation.

Hamilton and his supporters, including Peter Tait, were vigorous in defending Hamilton's precedence in inventing quaternions, and for muddying the water with respect to vectors which were then beginning to be understood and used. Rodrigues developed the key idea in 1840 and Gauss discovered it in 1819 but, as usual, did not

publish it. Quaternions had a tempestuous beginning. The paper by Altmann (1989) is an interesting description on this tussle of ideas, and quaternions have even been woven into fiction (Pynchon 2006).

Quaternions are discussed briefly in Siciliano et al. (2008). The book by Kuipers (1999) is a very readable and comprehensive introduction to quaternions. Quaternion interpolation is widely used in computer graphics and animation and the classic paper by Shoemake (1985) is very readable introduction to this topic. The first publications about quaternions for robotics is probably Taylor (1979) and with subsequent work by Funda (1990).

---

### Exercises

1. Explore the effect of negative roll, pitch or yaw angles. Does transforming from RPY angles to a rotation matrix then back to RPY angles give a different result to the starting value as it does for Euler angles?
2. Explore the many options associated with `trplot`.
3. Use `tranimate` to show translation and rotational motion about various axes.
4. Animate a tumbling cube
  - a) Write a function to plot the edges of a cube centred at the origin.
  - b) Modify the function to accept an argument which is a homogeneous transformation which is applied to the cube vertices before plotting.
  - c) Animate rotation about the  $x$ -axis.
  - d) Animate rotation about all axes.
5. Using Eq. 2.21 show that  $TT^{-1} = I$ .
6. Generate the sequence of plots shown in Fig. 2.11.
7. Where is Descarte's skull?
8. Create a vector-quaternion class that supports composition, negation and point transformation.

## 3

## Time and Motion

*The only reason for time is  
so that everything doesn't happen at once*  
Albert Einstein

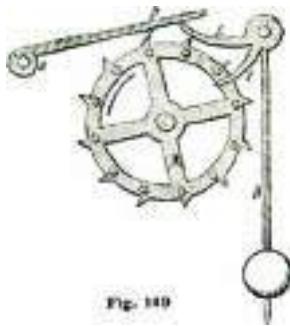


Fig. 3.0

In the previous chapter we learnt how to describe the pose of objects in 2- or 3-dimensional space. This chapter extends those concepts to objects whose pose is varying as a function of time.

For robots we wish to create a time varying pose that the robot can follow, for example the pose of a robot's end-effector should follow a path to the object that it is to grasp. Section 3.1 discusses how to generate a temporal sequence of poses, a trajectory, that smoothly changes from an initial pose to a final pose.

Section 3.2 discusses the concept of rate of change of pose, its temporal derivative, and how that relates to concepts from mechanics such as velocity and angular velocity. This allows us to solve the inverse problem – given measurements from velocity and angular velocity sensors how do we update the estimate of pose for a moving object. This is the principle underlying inertial navigation.

### 3.1 Trajectories

A path is a spatial construct – a locus in space that leads from an initial pose to a final pose. A trajectory is a path with specified timing. For example there is a path from A to B, but there is a trajectory from A to B in 10 s or at  $2 \text{ m s}^{-1}$ .

An important characteristic of a trajectory is that it is *smooth* – position and orientation vary smoothly with time. We start by discussing how to generate smooth trajectories in one dimension. We then extend that to the multi-dimensional case and then to piecewise-linear trajectories that visit a number of intermediate points without stopping.

#### 3.1.1 Smooth One-Dimensional Trajectories

We start our discussion with a scalar function of time. Important characteristics of this function are that its initial and final value are specified and that it is *smooth*. Smoothness in this context means that its first few temporal derivatives are continuous. Typically velocity and acceleration are required to be continuous and sometimes also the derivative of acceleration or jerk.

An obvious candidate for such a function is a polynomial function of time. Polynomials are simple to compute and can easily provide the required smoothness and boundary conditions. A quintic (fifth-order) polynomial is often used

$$S(t) = At^5 + Bt^4 + Ct^3 + Dt^2 + Et + F \quad (3.1)$$

where time  $t \in [0, T]$ . The first- and second-derivatives are also smooth polynomials

$$\dot{S}(t) = 5At^4 + 4Bt^3 + 3Ct^2 + 2Dt + E \quad (3.2)$$

$$\ddot{S}(t) = 20At^3 + 12Bt^2 + 6Ct + 2D \quad (3.3)$$

The trajectory has defined boundary conditions for position, velocity and acceleration► and commonly the velocity and acceleration boundary conditions are all zero.

Writing Eq. 3.1 to Eq. 3.3 for the boundary conditions  $t = 0$  and  $t = T$  gives six equations which we can write in matrix form as

$$\begin{pmatrix} s_0 \\ s_T \\ \dot{s}_0 \\ \dot{s}_T \\ \ddot{s}_0 \\ \ddot{s}_T \end{pmatrix} = \left( \begin{array}{cccccc|c} 0 & 0 & 0 & 0 & 0 & 1 & A \\ T^5 & T^4 & T^3 & T^2 & T & 1 & B \\ 0 & 0 & 0 & 0 & 1 & 0 & C \\ 5T^4 & 4T^3 & 3T^2 & 2T & 1 & 0 & D \\ 0 & 0 & 0 & 2 & 0 & 0 & E \\ 20T^3 & 12T^2 & 6T & 2 & 0 & 0 & F \end{array} \right)$$

Since the matrix is square► we can solve for the coefficient vector ( $A, B, C, D, E, F$ ) using standard linear algebra methods such as the MATLAB® \-operator. For a quintic polynomial acceleration will be a smooth cubic polynomial, and jerk will be a parabola.

The Toolbox function `tpoly` generates a quintic polynomial trajectory as described by Eq. 3.1. For example

```
>> s = tpoly(0, 1, 50);
```

returns a  $50 \times 1$  column vector with values varying smoothly from 0 to 1 in 50 time steps which we can plot

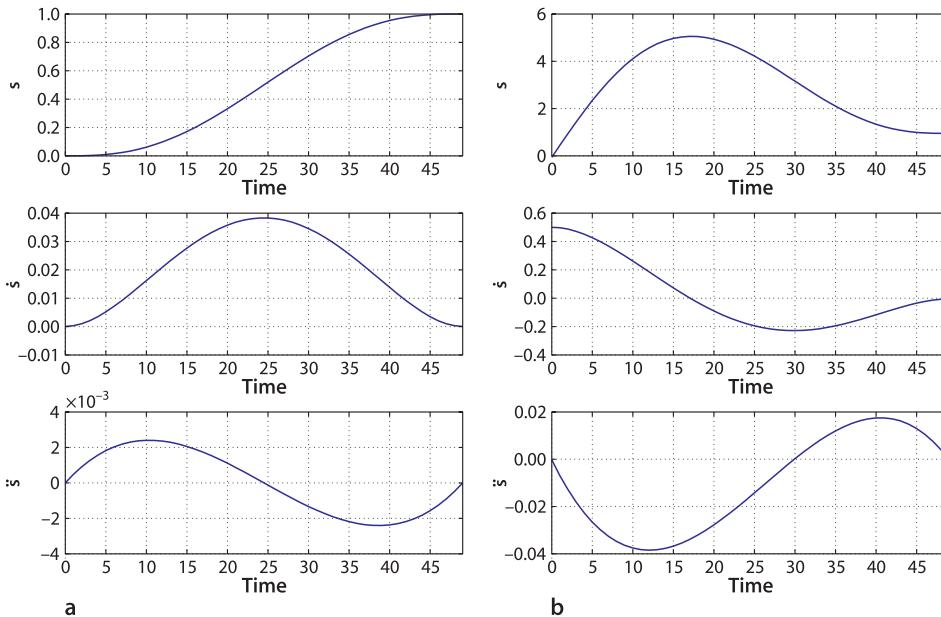
```
>> plot(s)
```

The corresponding velocity and acceleration can be returned via optional output arguments

```
>> [s, sd, sdd] = tpoly(0, 1, 50);
```

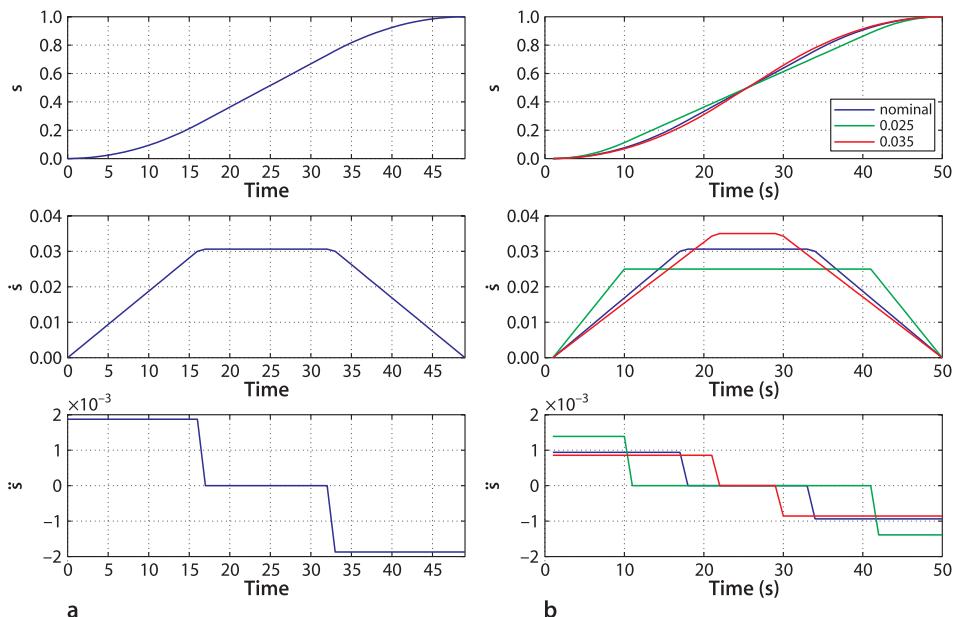
as `sd` and `sdd` respectively. These are shown in Fig. 3.1a and we observe that the initial and final velocity and acceleration are all zero – the default value. The initial and final velocities can be set to non-zero values

```
>> s = tpoly(0, 1, 50, 0.5, 0);
```



Time	$s$	$\dot{s}$	$\ddot{s}$
$t = 0$	$s_0$	$\dot{s}_0$	$\ddot{s}_0$
$t = T$	$s_T$	$\dot{s}_T$	$\ddot{s}_T$

**Fig. 3.1.**  
Quintic polynomial trajectory.  
From top to bottom is position,  
velocity and acceleration versus  
time. **a** With zero-velocity bound-  
ary conditions, **b** initial velocity  
of 0.5 and a final velocity of 0



**Fig. 3.2.**  
Linear segment with parabolic  
blend (LSPB) trajectory: **a** default  
velocity for linear segment;  
**b** specified linear segment velo-  
city values

in this case, an initial velocity of 0.5 and a final velocity of 0. The results shown in Fig. 3.1b illustrate a problem with polynomials. The non-zero initial velocity causes the polynomial to overshoot the terminal value – it peaks at 5 on a trajectory from 0 to 1.

Another problem with polynomials, a very practical one, can be seen in the middle graph of Fig. 3.1a. The velocity peaks at  $t = 25$  which means that for most of the time the velocity is far less than the maximum. The mean velocity

```
>> mean(sd) / max(sd)
ans =
0.5231
```

is only 52% of the peak. A real robot joint has a well defined maximum velocity and for minimum-time motion we want to be operating at that maximum for as much of the time as possible. We would like the velocity curve to be *flatter* on top.

A well known alternative is a hybrid trajectory which has a constant velocity segment with polynomial segments for acceleration and deceleration. Revisiting our first example the hybrid trajectory is

```
>> s = lspb(0, 1, 50);
```

where the arguments have the same meaning as for `tpoly` and the trajectory is shown in Fig. 3.2a. The trajectory comprises a linear segment (constant velocity) with parabolic blends, hence the name `lspb`. The term blend is commonly used to refer to a trajectory segment that smoothly joins linear segments. As with `tpoly` we can also return the velocity and acceleration

```
>> [s,sd,sdd] = lspb(0, 1, 50);
```

This type of trajectory is also referred to as trapezoidal due to the shape of the velocity curve versus time, and is commonly used in industrial motor drives. ▶

The function `lspb` has *chosen* the velocity of the linear segment to be

```
>> max(sd)
ans =
0.0306
```

but this can be overridden by specifying it as a fourth input argument

```
>> s = lspb(0, 1, 50, 0.025);
>> s = lspb(0, 1, 50, 0.035);
```

The trapezoidal trajectory is smooth in velocity, but not in acceleration.

The trajectories for these different cases are overlaid in Fig. 3.2b. We see that as the velocity of the linear segment increases its duration decreases and ultimately its duration would be zero. In fact the velocity cannot be chosen arbitrarily, too high or too low a value for the maximum velocity will result in an infeasible trajectory and the function returns an error.

The system is over-constrained, having five constraints (total time, initial and final position and velocity) but six degrees of freedom (blend time, three parabolic coefficients and two linear coefficients).

### 3.1.2 Multi-Dimensional Case

Most useful robots have more than one axis of motion or degree of freedom. We represent this in vector form as  $\mathbf{x} \in \mathbb{R}^M$  where  $M$  is the number of degrees of freedom. A wheeled mobile robot is characterised by its position  $(x, y)$  or pose  $(x, y, \theta)$ . The tool of an arm robot has position  $(x, y, z)$ , orientation  $(\theta_x, \theta_y, \theta_z)$  or pose  $(x, y, z, \theta_x, \theta_y, \theta_z)$ . We therefore require smooth multi-dimensional motion from an initial vector to a final vector.

It is quite straightforward to extend the smooth scalar trajectory to the vector case and in the Toolbox this is achieved using the function `mtraj`. For example to move from  $(0, 2)$  to  $(1, -1)$  in 50 steps

```
>> x = mtraj(@tpoly, [0 2], [1 -1], 50);
>> x = mtraj(@lspb, [0 2], [1 -1], 50);
```

which results in a  $50 \times 2$  matrix `x` with one row per time step and one column per axis. The first argument is a function that generates a *scalar* trajectory, either `tpoly` or `lspb`. The trajectory for the `lspb` case

```
>> plot(x)
```

is shown in Fig. 3.3.

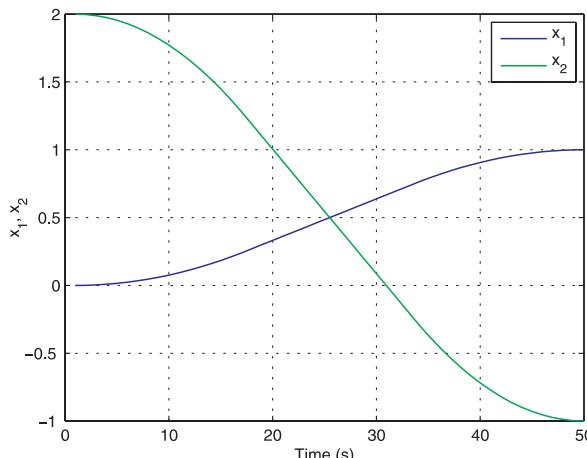
For a 3-dimensional problem we might consider converting a pose  $T$  to a 6-vector by

```
>> x = [transl(T); tr2rpy(T)']
```

though as we shall see later interpolation of 3-angle representations has some limitations.

### 3.1.3 Multi-Segment Trajectories

In robotics applications there is often a need to move smoothly along a path through one or more intermediate or *via* points without stopping. This might be to avoid obstacles in the workplace, or to perform a task that involves following a piecewise continuous trajectory such as applying a bead of sealant in a manufacturing application.



**Fig. 3.3.**  
Multi-dimensional motion.  
 $x_1$  varies from  $0 \rightarrow 1$  and  
 $x_2$  varies from  $2 \rightarrow -1$

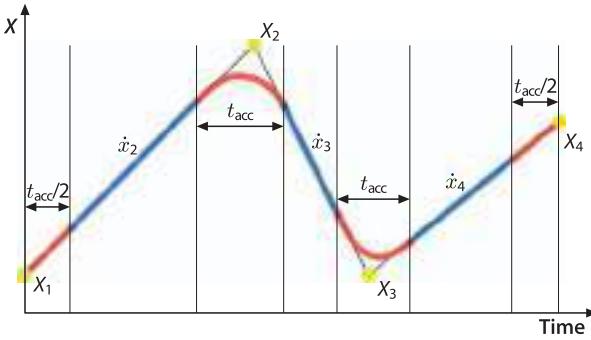


Fig. 3.4.

Notation for multi-segment trajectory showing four points and three motion segments. Blue indicates constant velocity motion, red indicates regions of acceleration

To formalize the problem consider that the trajectory is defined by  $N$  points  $\mathbf{x}_k$ ,  $k \in [1, N]$  and there are  $N - 1$  motion segments. As in the previous section  $\mathbf{x}_k \in \mathbb{R}^M$  is a vector representation of pose.

The robot starts from  $\mathbf{x}_1$  at rest and finishes at  $\mathbf{x}_N$  at rest, but moves through (or close to) the intermediate points without stopping. The problem is over constrained and in order to attain continuous velocity we surrender the ability to reach each intermediate point. This is easiest to understand for the one dimensional case shown in Fig. 3.4. The motion comprises linear motion segments with polynomial blends, like `lpsb`, but here we choose quintic polynomials because they are able to match boundary conditions on position, velocity and acceleration at their start and end points.

The first segment of the trajectory accelerates from the initial pose  $\mathbf{x}_1$  and zero velocity, and joins the line heading toward the second pose  $\mathbf{x}_2$ . The blend time is set to be a constant  $t_{acc}$  and  $t_{acc}/2$  before reaching  $\mathbf{x}_2$  the trajectory executes a polynomial blend, of duration  $t_{acc}$ , onto the line from  $\mathbf{x}_2$  to  $\mathbf{x}_3$ , and the process repeats. The constant velocity  $\dot{\mathbf{x}}_k$  can be specified for each segment. The acceleration during the blend is

$$\ddot{\mathbf{x}} = \frac{\dot{\mathbf{x}}_{k+1} - \dot{\mathbf{x}}_k}{t_{acc}}$$

If the maximum acceleration capability of the axis is known then the minimum blend time can be computed.

For the multi-axis case it is likely that some axes will need to move further than others on a particular motion segment and this becomes complex if the joints have different velocity limits. The first step is to determine which axis will be the slowest to complete the motion segment, based on the distance that each axis needs to travel for the segment and its maximum achievable velocity. From this the duration of the segment can be computed and then the required velocity of each axis. This ensures that all axes reach the next target  $\mathbf{x}_k$  at the same time.

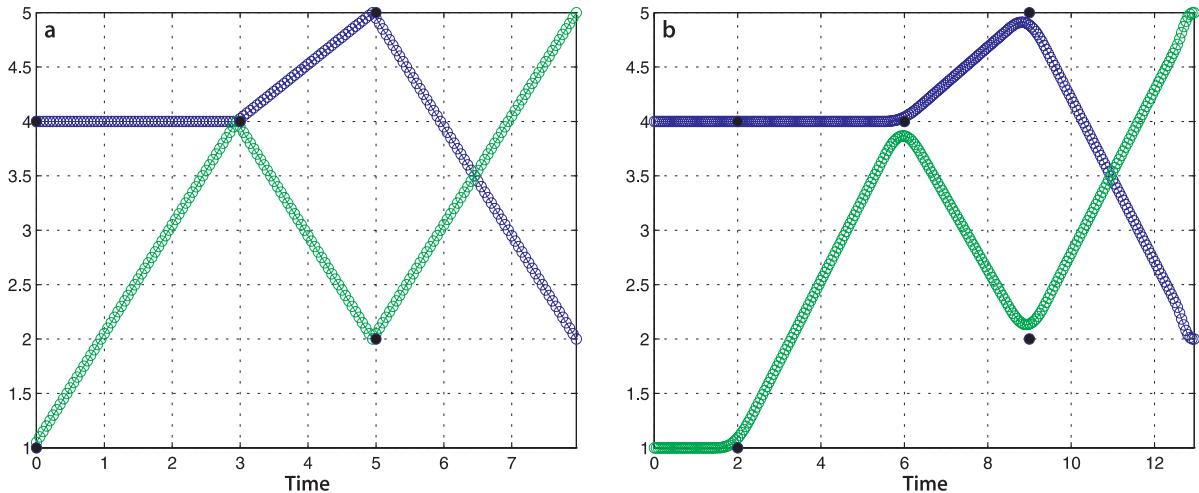
The Toolbox function `mstraj` generates a multi-segment multi-axis trajectory based on a matrix of via points. For example 2-axis motion with four points can be generated by

```
>> via = [ 4,1; 4,4; 5,2; 2,5 ];
>> q = mstraj(via, [2,1], [], [4,1], 0.05, 0);
```

Only one of the maximum axis speed or time per segment can be specified, the other is set to MATLAB's empty matrix `[]`.

Acceleration time if given is rounded up to a multiple of the time increment.

The first argument is the matrix of via points, one row per point. The remaining arguments are respectively: a vector of maximum speeds per axis, a vector of durations for each segment, the initial axis coordinates, the sample interval, and the acceleration time. The function `mstraj` returns a matrix with one row per time step and the columns correspond to the axes. If no output argument is provided `mstraj` will plot the trajectory as shown in Fig. 3.5a. The parameters in this example indicate that the first axis has a higher maximum speed than the second. However for the last segment both axes move at the same speed since the segment time is dominated by the slowest axis.



If we increase the acceleration time

```
>> q = mstraj(via, [2 1], [], [4 1], 0.05, 1);
```

the trajectory becomes more rounded, Fig. 3.5b, as the polynomial blending functions do their work, and the trajectory takes more time to complete. The function also accepts optional initial and final velocity arguments and  $t_{\text{acc}}$  can be a vector giving acceleration time for each of the  $N$  blends.

Keep in mind that this function simply interpolates pose represented as a vector. In this example the vector was assumed to be Cartesian coordinates, but this function could also be applied to Euler or roll-pitch-yaw angles but this is not an ideal way to interpolate rotation. This leads us nicely to the next section where we discuss interpolation of orientation.

**Fig. 3.5.** Multisegment trajectories:  
**a** no acceleration time  $t_{\text{acc}} = 0$ ;  
**b** acceleration time of  $t_{\text{acc}} = 1$  s.  
The discrete-time points are indicated with circular markers, and the via points are indicated by solid black markers

### 3.1.4 Interpolation of Orientation in 3D

In robotics we often need to interpolate orientation, for example, we require the end-effector of a robot to smoothly change from orientation  $\xi_0$  to  $\xi_1$ . Using the notation from Chap. 2 we require some function  $\xi(s) = \sigma(\xi_0, \xi_1, s)$  where  $s \in [0, 1]$  which has the boundary conditions  $\sigma(\xi_0, \xi_1, 0) = \xi_0$  and  $\sigma(\xi_0, \xi_1, 1) = \xi_1$  and where  $\sigma(\xi_0, \xi_1, s)$  varies *smoothly* for intermediate values of  $s$ . How we implement this depends very much on our concrete representation of  $\xi$ .

If pose is represented by an orthonormal rotation matrix,  $\xi \sim R \in SO(3)$ , we might consider a simple linear interpolation  $\sigma(R_0, R_1, s) = (1 - s)R_0 + sR_1$  but this would not, in general, be a valid orthonormal matrix which has strict column norm and inter-column orthogonality constraints.

A workable and commonly used alternative is to consider a 3-angle representation such as Euler or roll-pitch-yaw angles,  $\xi \sim \Gamma \in \mathbb{S}^3$  and use linear interpolation

$$\sigma(\Gamma_0, \Gamma_1, s) = (1 - s)\Gamma_0 + \Gamma_1$$

For example we define two orientations

```
>> R0 = rotz(-1) * roty(-1);
>> R1 = rotz(1) * roty(1);
```

and find the equivalent roll-pitch-yaw angles

```
>> rpy0 = tr2rpy(R0); rpy1 = tr2rpy(R1);
```

and create a trajectory between them over 50 time steps

```
>> rpy = mtraj(@tpoly, rpy0, rpy1, 50);
```

`rpy` is a  $50 \times 3$  matrix and the result of `rpy2tr` is a  $4 \times 4 \times 50$  matrix which is explained in Sect. 3.1.5.

which is mostly easily visualized as an animation◀

```
>> tranimate( rpy2tr(rpy) );
```

For large orientation changes we see that the axis around which the coordinate frame rotates changes along the trajectory. The motion, while smooth, sometimes looks uncoordinated. There will also be problems if either  $\xi_0$  or  $\xi_1$  is close to a singularity in the particular 3-angle system being used.

Interpolation of unit-quaternions is only a little more complex than for 3-angle vectors and produces a change in orientation that is a rotation around a fixed axis in space. Using the Toolbox we first find the two equivalent quaternions

```
>> q0 = Quaternion(R0);
>> q1 = Quaternion(R1);
```

and then interpolate them

```
>> q = interp(q0, q1, [0:49]'/49);
>> about(q)
q [Quaternion] : 1x50 (1656 bytes)◀
```

The size of the object in bytes, shown in parentheses, will vary between MATLAB® versions and computer types.

which results in a vector of 50 `Quaternion` objects which we can animate by

```
>> tranimate(q)
```

Quaternion interpolation is achieved using spherical linear interpolation (*slerp*) in which the unit quaternions follow a great circle path on a 4-dimensional hypersphere. The result in 3-dimensions is rotation about a fixed axis in space.

### 3.1.5 Cartesian Motion

Another common requirement is a smooth path between two poses in  $SE(3)$  which involves change in position as well as in orientation. In robotics this is often referred to as Cartesian motion.

We represent the initial and final poses as homogeneous transformations

```
>> T0 = transl(0.4, 0.2, 0) * trotx(pi);
>> T1 = transl(-0.4, -0.2, 0.3) * troty(pi/2)*trotz(-pi/2);
```

The Toolbox function `trinterp` provides interpolation for normalized distance along the path  $s \in [0, 1]$ , for example the mid pose is

```
>> trinterp(T0, T1, 0.5)
ans =
-0.0000    1.0000         0         0
      0   -0.0000   -1.0000         0
-1.0000         0   -0.0000    0.1500
      0         0         0    1.0000
```

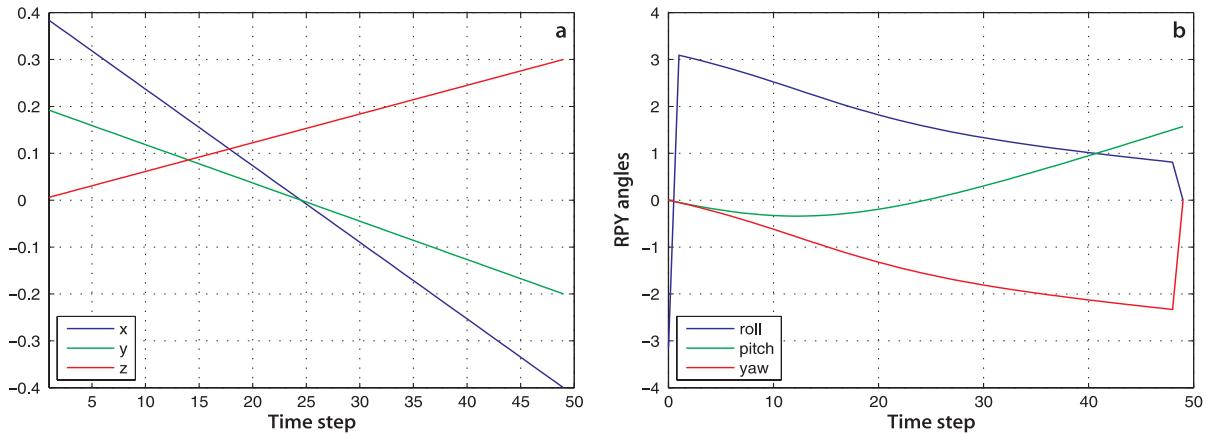
where the translational component is linearly interpolated and the rotation is spherically interpolated using the quaternion interpolation method `interp`.

A trajectory between the two poses in 50 steps is created by

```
>> Ts = trinterp(T0, T1, [0:49]/49);
```

where the arguments are the initial and final pose and a path length varying linearly from zero to one. The resulting trajectory `Ts` is a matrix with three dimensions

```
>> about(Ts)
Ts [double] : 4x4x50 (6400 bytes)
```



representing the homogeneous transformation (first 2 indices) for each time step (third index). The homogeneous transformation for the first point on the path is

```
>> Ts(:,:,1)
ans =
    1.0000      0      0    0.4000
        0   -1.0000      0    0.2000
        0      0   -1.0000      0
        0      0      0    1.0000
```

and once again the easiest way to visualize this is by animation

```
>> tranimate(Ts)
```

which shows the coordinate frame moving and rotating from pose `T1` to pose `T2`.

The translational part of this trajectory is obtained by

```
>> P = transl(Ts);
```

which returns the Cartesian position for the trajectory in matrix form

```
>> about(P)
P [double] : 50x3 (1200 bytes)
```

which has one row per time step, and each row is the corresponding position vector. This is plotted

```
>> plot(P);
```

in Fig. 3.6 along with the orientation in roll-pitch-yaw format

```
>> rpy = tr2rpy(Ts);
>> plot(rpy);
```

We see that the position coordinates vary smoothly and linearly with time and that orientation varies smoothly with time.►

However the translational motion has a velocity and acceleration *discontinuity* at the first and last points. The problem is that while the trajectory is smooth in space the distance  $s$  along the trajectory is not smooth in time. Speed along the path jumps from zero to some finite value and then drops to zero at the end – there is no initial acceleration or final deceleration. The scalar functions `tpoly` and `lspb` discussed earlier can be used to generate  $s$  so that motion *along* the path is smooth. We can pass a vector of normalized distances along the path as the third argument to `trinterp`

```
>> Ts = trinterp(T0, T1, lspb(0,1, 50));
```

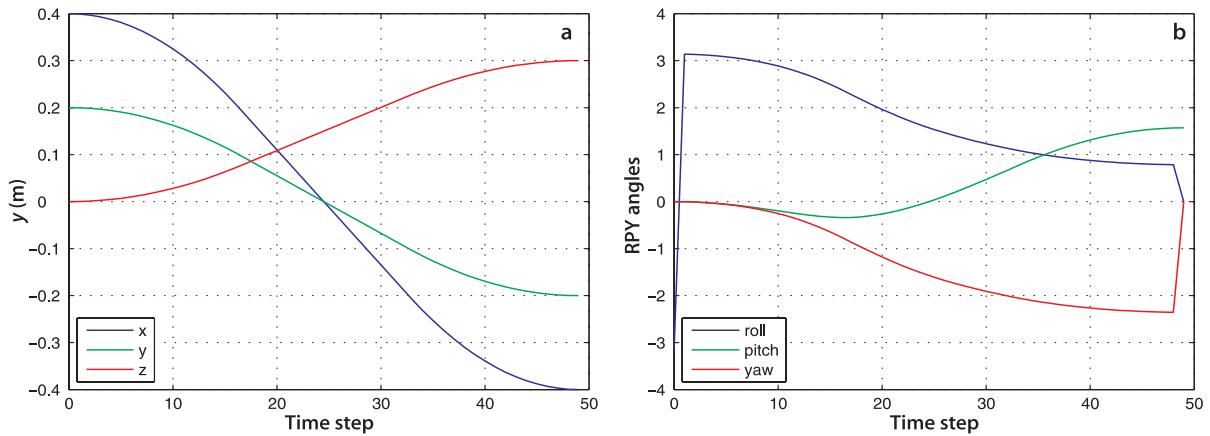
The trajectory is unchanged but the coordinate frame now accelerates to a constant speed along the path and then decelerates and this is reflected in smoother curves for

**Fig. 3.6.** Cartesian motion. **a** Cartesian position versus time, **b** roll-pitch-yaw angles versus time

The roll-pitch-yaw angles do not vary linearly with time because they represent a non-linear transformation of the linearly varying quaternion.

The discontinuity in roll angle after the first point is due to angle wrapping around the circle, moving from  $-\pi$  to  $+\pi$ .

The discontinuity between the last two points is because the final orientation is a singularity for roll-pitch-yaw angles.



**Fig. 3.7.** Cartesian motion with LSPB path distance profile. **a** Cartesian position versus time, **b** roll-pitch-yaw angles versus time

the translational components of the trajectory shown in Fig. 3.7b. The Toolbox provides a convenient shorthand `c traj` for the above

```
>> Ts = c traj(T0, T1, 50);
```

where the arguments are the initial and final pose and the number of time steps.

## 3.2 Time Varying Coordinate Frames

The previous section discussed the generation of coordinate frame motion which has a translational and rotational velocity component. The translational velocity is the rate of change of the position of the origin of the coordinate frame. Rotational velocity is a little more complex.

### 3.2.1 Rotating Coordinate Frame

A body rotating in 3-dimensional space has an angular velocity which is a vector quantity  $\omega = (\omega_x, \omega_y, \omega_z)$ . The direction of this vector defines the instantaneous axis of rotation, that is, the axis about which the coordinate frame is rotating at a particular instant of time. In general this axis changes with time. The magnitude of the vector is the rate of rotation about the axis – in this respect it is similar to the angle-axis representation for rotation introduced in Sect. 2.2.1.5. From mechanics there is a well known expression for the derivative of a time-varying rotation matrix

$$\dot{R}(t) = S(\omega)R(t) \quad (3.4)$$

where  $R(t) \in SO(2)$  or  $SO(3)$  and  $S(\cdot)$  is a skew-symmetric matrix that, for the 3-dimensional case, has the form

$$S(\omega) = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix} \quad (3.5)$$

and its properties are described in Appendix D. Using the Toolbox we can write

```
>> S = skew([1 2 3])
S =
    0     -3      2
    3      0     -1
   -2      1      0
```

The Toolbox function `vex` performs the inverse function<sup>►</sup> of converting a skew-symmetric matrix to a vector

```
>> vex(S)'
ans =
    1      2      3
```

We might ask what does  $\dot{R}$  mean? Consider the approximation to the derivative

$$\dot{R} \approx \frac{R(t+\delta_t) - R(t)}{\delta_t} \quad (3.6)$$

which we rearrange as

$$R(t+\delta_t) \approx \delta_t \dot{R} + R(t)$$

and substituting Eq. 3.4 we obtain

$$R(t+\delta_t) \approx \delta_t S(\omega) R(t) + R(t) \approx (\delta_t S(\omega) + I_{3 \times 3}) R(t) \quad (3.7)$$

which describes how the orthonormal rotation matrix changes as a function of angular velocity.

Each element appears twice in the skew-symmetric matrix, with different sign. In some algorithms we compute an approximation of the skew-symmetric matrix and these elements may have slightly different magnitudes so `vex` takes the average of both elements.

### 3.2.2 Incremental Motion

Consider a coordinate frame that undergoes a small rotation from  $R_0$  to  $R_1$ . We can write Eq. 3.7 as

$$R_1 = (\delta_t S(\omega) + I_{3 \times 3}) R_0$$

and rearrange it as

$$\delta_t S(\omega) = R_1 R_0^T - I_{3 \times 3}$$

and then apply the `vex` operator, the inverse of  $S(\cdot)$ , to both sides

$$\delta_\Theta = \text{vex}(R_1 R_0^T - I_{3 \times 3}) \quad (3.8)$$

where  $\delta_\Theta = \delta_t \omega$  is a 3-vector with units of angle that represents an infinitesimal rotation about the world  $x$ -,  $y$ - and  $z$ -axes.<sup>►</sup>

We have strongly, and properly, cautioned about the non-commutativity of rotations but for infinitesimal angular changes multiplication *is commutative*. We can demonstrate this numerically by

```
>> Rdelta = rotx(0.001)*roty(0.002)*rotz(0.003)
ans =
    1.0000   -0.0030    0.0020
    0.0030    1.0000   -0.0010
   -0.0020    0.0010    1.0000
```

which is, to four significant figures, the same as

```
>> roty(0.002) * rotx(0.001)*rotz(0.003)
ans =
    1.0000   -0.0030    0.0020
    0.0030    1.0000   -0.0010
   -0.0020    0.0010    1.0000
```

Using Eq. 3.8 we can recover the infinitesimal rotation angle  $\delta_\Theta$

```
>> vex( Rdelta - eye(3,3) )
ans =
    0.0010    0.0020    0.0030
```

It is in the world coordinate frame because the term  $(\delta_t S(\omega) + I_{3 \times 3})$  pre-multiplies  $R_0$ .

Given two poses  $\xi_0$  and  $\xi_1$  that differ infinitesimally we can represent the difference between them as a 6-vector

$$\delta = \Delta(\xi_0, \xi_1) = (\delta_d, \delta_\Theta) \quad (3.9)$$

comprising the incremental displacement and the incremental rotation. The quantity  $\delta \in \mathbb{R}^6$  is effectively the spatial velocity which we discuss further in Chap. 8 multiplied by  $\delta_r$ . If the poses are represented in homogeneous transformation form then the difference is

$$\delta = \Delta(T_0, T_1) = \begin{pmatrix} t_1 - t_0 \\ \text{vex}(R_1 R_0^T - I_{3 \times 3}) \end{pmatrix} \quad (3.10)$$

where  $T_0 = (R_0, t_0)$  and  $T_1 = (R_1, t_1)$ . In the Toolbox this is the function `tr2delta`.

The inverse operation is

$$\xi = \Delta^{-1}(\delta) \quad (3.11)$$

and for homogenous transformation representation is

$$T = \begin{pmatrix} S(\delta_\Theta) & \delta_d \\ \mathbf{0}_{3 \times 1} & 0 \end{pmatrix} + I_{3 \times 3} \quad (3.12)$$

and in the Toolbox this is the function `delta2tr`.

For example

```
>> T0 = transl(1,2,3)*trotx(1)*trot(y(1))*trotz(1);
>> T1 = T0*transl(0.01,0.02,0.03)*trotx(0.001)*trot(y(0.002))*trotz(0.003)
T1 =
    0.2889   -0.4547    0.8425    1.0191
    0.8372   -0.3069   -0.4527    1.9887
    0.4644    0.8361    0.2920    3.0301
        0         0         0     1.0000
```

the function  $\Delta(\cdot)$  is computed by the Toolbox function `tr2delta`

```
>> d = tr2delta(T0, T1);
>> d'
ans =
    0.0191   -0.0113    0.0301    0.0019   -0.0011    0.0030
```

which comprises the incremental translation and rotation expressed in the world coordinate frame. Given this displacement and the initial pose, the final pose is

```
>> delta2tr(d) * T0
    0.2889   -0.4547    0.8425    1.0096
    0.8372   -0.3069   -0.4527    1.9859
    0.4644    0.8361    0.2920    3.0351
        0         0         0     1.0000
```

which is quite close to the true value given above and the error is due to the fact that the displacement is not infinitesimal. The displacement 6-vector is used in the next section and several times in Chap. 8.

### 3.2.3 Inertial Navigation Systems

An inertial navigation system is a “black box” that estimates its velocity, orientation and position with respect to the inertial reference frame (the universe). Importantly it has no external inputs such as radio signals from satellites and this makes it well suited to applications such as submarine, spacecraft and missile guidance. An inertial navigation system works by measuring accelerations and angular velocities and integrating them over time.

Early inertial navigation systems, such as shown in Fig. 2.13, used mechanical gimbals to keep the accelerometers at a constant attitude with respect to the stars using a gyro-stabilized platform. The acceleration measured on this platform was integrated to obtain the velocity of the platform, and integrated again to obtain its position. In order to achieve accurate position estimates over periods of hours or days the gimbals and gyroscopes had to be of extremely high quality so that the stable platform did not drift, and the acceleration sensors needed to be extremely accurate.

In a modern strapdown inertial measurement system the acceleration and angular velocity sensors are rigidly attached to the vehicle. The three orthogonally mounted gyroscopes measure the components of the angular velocity  $\omega$  and use Eq. 3.7 to continuously update the estimated orientation  ${}^0R_B$  of the vehicle's body-fixed frame  $\{B\}$  with respect to the stars  $\{0\}$ .

A discrete-time version of Eq. 3.7 such as

$$R(k+1) = \delta_t S(\omega) R(k) + R(k) \quad (3.13)$$

is used to numerically integrate changes in pose in order to estimate the orientation of the vehicle. The measured acceleration  ${}^B\mathbf{a}$  of the vehicle's body frame is rotated into the inertial frame

$${}^0\mathbf{a} = {}^0R_B {}^B\mathbf{a}$$

and can then be integrated twice to update the estimate of the vehicle's position in the inertial frame. Practical systems work at high sample rate, typically hundreds of Hertz, and would employ higher-order numerical integration techniques rather than the simple rectangular integration of Eq. 3.13.

In Eq. 3.13 we added the matrix  $\delta_t S(\omega) R(t)$  to an orthonormal rotation matrix and this is not quite proper – the result *will not* be an orthonormal matrix. However if the added term is small▶ the result will be close to orthonormal and we can *straighten it up*. This process is called normalization and enforces the constraints on the elements of an orthonormal matrix. It involves the following steps where  $c_i$  is the  $i^{\text{th}}$  column of  $R$ . We first assume that column 3 is correct

Which is why inertial navigation systems operate at a high sample rate and  $\delta_t$  is small.

An Inertial Navigation System (INS) comprises a computer and motion sensors and continuously calculates the position, orientation, and velocity of a moving object via dead reckoning. It has no need for external references which is important for vehicles, such as submarines or spacecraft, that are unable to communicate with radio navigation aids or which must be immune to radio jamming.

A gyroscope or gyro is a sensor that measures angular velocity about a particular axis. Early rotational sensors actually employed a spinning disk, like the childhood toy, but today the term gyroscope is generically applied to all angular velocity sensors. To measure the angular velocity a triaxial assembly of gyroscopes is used – three gyroscopes with their measurement axes in orthogonal directions. Modern gyroscopes are based on a variety of physical principles such as tiny vibrating beams or relativistic effects in ring-laser and fibre-optic gyros. An accelerometer is a sensor that measures acceleration along a particular axis. Accelerometers work by measuring the forces on a small proof mass within the sensor. The vehicle's acceleration vector is measured using a triaxial assembly of accelerometers. The sensors are collectively referred to as an inertial measurement unit (IMU).

Much important development was undertaken by the MIT Instrumentation Laboratory under the leadership of Charles Stark Draper. In 1953 the Space Inertial Reference Equipment (SPIRE) system, 1 200 kg of equipment, guided a B-29 bomber on a 12 hour trip from Massachusetts to Los Angeles without the aid of a pilot and with Draper aboard. In 1954 the first self-contained submarine navigation system (SINS) was introduced to service. The Instrumentation Lab also developed the Apollo Guidance Computer, a one-cubic-foot computer that guided the Apollo Lunar Module to the surface of the Moon in 1969.

Today high-performance inertial navigation systems based on fibre-optic gyroscopes are widely available and weigh around one 1 kg and low-cost systems based on MEMS technology can weigh less than 100 g and cost tens of dollars.



**Charles Stark (Doc) Draper** (1901–1987) was an American scientist and engineer, often referred to as “the father of inertial navigation.” Born in Windsor, Missouri, he studied at the University of Missouri then Stanford where he earned a B.A. in psychology in 1922, then at MIT an S.B. in electrochemical engineering and an S.M. and Sc.D. in physics in 1928 and 1938 respectively. He started teaching while at MIT and became a full professor in aeronautical engineering in 1939. He was the founder and director of the MIT Instrumentation Laboratory which made important contributions to the theory and practice of inertial navigation to meet the needs of the cold war and the space program.

Draper was named one of Time magazine’s Men of the Year in 1961 and inducted to the National Inventors Hall of Fame in 1981. The Instrumentation lab was renamed Charles Stark Draper Laboratory (CSDL) in his honour. (Photo on the left: courtesy of The Charles Stark Draper Laboratory Inc.)

$$\mathbf{c}'_3 = \mathbf{c}_3$$

then the first column is made orthogonal to the last two

$$\mathbf{c}'_1 = \mathbf{c}_2 \times \mathbf{c}'_3$$

However the last two columns may not have been orthogonal so

$$\mathbf{c}'_2 = \mathbf{c}'_1 \times \mathbf{c}'_3$$

Finally the columns are normalized to unit magnitude

$$\mathbf{c}''_i = \frac{\mathbf{c}'_i}{\|\mathbf{c}'_i\|}, \quad i = 1 \dots 3$$

In the Toolbox normalization is implemented by

```
>> T = trnorm(T);
```

and is quite similar to the problem of representing pose using two unit vectors discussed in Sect. 2.2.1.4. In an orientation estimation system using Eq. 3.13 the attitude  $R$  should be normalized after each integration step.◀

Alternatively we could use unit-quaternions and things, as is generally the case, are a little simpler. The derivative of a quaternion, the quaternion equivalent of Eq. 3.4 is defined as

$$\dot{\hat{q}} = \frac{1}{2}\dot{q}(\omega)\hat{q} \tag{3.14}$$

which is implemented by the `dot` method

```
>> qd = q.dot(omega);
```

Integration of quaternion rates is achieved by

$$\dot{q}\langle k+1 \rangle = \dot{q}\langle k \rangle + \delta_t \dot{\hat{q}}$$

As with Eq. 3.7 the addition is not quite proper and the result will no longer be a unit quaternion. Normalization is achieved by ensuring that the quaternion norm is unity, a straightforward division of all elements by the quaternion norm

$$\dot{q}\langle k+1 \rangle' = \frac{\dot{q}\langle k+1 \rangle}{\|\dot{q}\langle k+1 \rangle\|}$$

or in the Toolbox

```
>> q = q.unit();
```

Quaternions are more commonly used in the rotation update equations for strapdown inertial navigation systems than orthonormal rotation matrices. The results will of course be the same but the computational cost for the quaternion version is significantly less.

### 3.3 Wrapping Up

In this chapter we have considered pose that varies as a function of time from two perspectives. The first perspective was to create a sequence of poses, a trajectory, that a robot can follow. An important characteristic of a trajectory is that it is *smooth* – the position and orientation varies smoothly with time. We start by discussing how to generate smooth trajectories in one dimension and then extended that to the multi-dimensional case and then to piecewise-linear trajectories that visit a number of intermediate points.

The second perspective was to examine the temporal derivative of an orthonormal rotation matrix and how that relates to concepts from mechanics such as velocity and angular velocity. This allows us to solve the inverse problem, given measurements from sensors we are able to update the estimate of pose for a moving object – the principle underlying inertial navigation. We introduced the infinitesimal motion  $\delta$  which is related to spatial velocity and which we will encounter again in Chap. 8.

Interpolation of rotation and integration of angular velocity was treated using both orthonormal rotation matrices and quaternions. The results are equivalent but the quaternion formulation is more elegant and computationally more efficient.

---

### Further Reading

The earliest work on manipulator Cartesian trajectory generation was by Paul (1972, 1979) and Taylor (1979). The muti-segment trajectory is discussed by Paul (1979, 1981) and the concept of segment transitions or blends is discussed by Lloyd and Hayward (1991). These early papers, and others, are included in the compilation on Robot Motion (Brady et al. 1982). Polynomial and LSPB trajectories are described in detail by Spong et al. (2006) and multi-segment trajectories are covered at length in Siciliano et al. (2008) and Craig (2004).

The relationship between orthonormal rotation matrices, the skew-symmetric matrix and angular velocity is well described in Spong et al. (2006).

The principles of inertial navigation are covered in the book by Groves (2008) which also covers GPS and other radio-based localization systems which are the subject of Part II. The book *Digital Apollo* (Mindell 2008) is a very readable story of the development of the inertial navigation system for the Apollo Moon landings. The paper by Corke et al. (2007) describes the principles of inertial sensors and the functionally equivalent sensors located in the inner ear of mammals that play a key role in maintaining balance.

---

### Exercises

1. For a `tpoly` trajectory from 0 to 1 in 50 steps explore the effects of different initial and final velocities, both positive and negative. Under what circumstances does the quintic polynomial overshoot and why?
2. For a `lspb` trajectory from 0 to 1 in 50 steps explore the effects of specifying the velocity for the constant velocity segment. What are the minimum and maximum bounds possible?
3. For a trajectory from 0 to 1 and given a maximum possible velocity of 0.025 compare how many time steps are required for each of the `tpoly` and `lspb` trajectories?

4. Use `tranimate` to compare rotational interpolation using quaternions, Euler angles and roll-pitch-yaw angles. Hint: use the quaternion `interp` method, and `mtraj` with `tr2eul` and `eul2tr`
  - a) Use `mtraj` to interpolate Euler angles between two orientations and display the rotating frame using `tranimate`.
  - b) Repeat for the case choose where the final orientation is at a singularity. What happens?
5. Repeat for the case where the interpolation passes through a singularity. What happens?
6. Develop a method to quantitatively compare the performance of the different orientation interpolation methods. Hint: plot the locus followed by  $\hat{z}$  on a unit sphere.
7. For the `mstraj` example (page 47)
  - a) Repeat with different initial and final velocity.
  - b) Investigate the effect of increasing the acceleration time. Plot total time as a function of acceleration time.
8. Modify `mstraj` so that acceleration limits are taken into account when determining the segment time.
9. Implement an inertial measurement system. First create an angular velocity signal as a function of time, for example

```
>> t = [0:0.01:10];
>> w = [0.1*sin(t) 0.2*sin(0.6*t) 0.3*sin(0.4*t)];
```

- a) Estimate the rotation matrix at each time step, then animate it.
- b) Repeat using quaternions.
- c) Add a small amount of Gaussian noise  $w = w + \text{randn}(\text{size}(w)) \times 0.001$  and repeat. What changes?
- d) Investigate performance for increasing amounts of noise.



This chapter discusses how a robot platform moves, that is, how its pose changes with time as a function of its control inputs. There are many different types of robot platform as shown on pages 61–63 but in this chapter we will consider only two which are important exemplars. The first is a wheeled vehicle like a car which operates in a 2-dimensional world. It can be propelled forwards or backwards and its heading direction controlled by changing the angle of its steered wheels. The second platform is a quadcopter, a flying vehicle, which is an example of a robot that moves in 3-dimensional space. Quadcopters are becoming increasing popular as a robot platform since they can be quite easily modelled and controlled.

However before we start to discuss these two robot platforms it will be helpful to consider some general, but important, concepts regarding mobility.

### 4.1 Mobility

We have already touched on the diversity of mobile robots and their modes of locomotion. In this section we will discuss mobility which is concerned with how a vehicle moves in space.

We first consider the simple example of a train. The train moves along rails and its position is described by its distance along the rail from some datum. The *configuration* of the train can be completely described by a scalar parameter  $q$  which is called its generalized coordinate. The set of all possible configurations is the configuration space, or C-space, denoted by  $\mathcal{C}$  and  $q \in \mathcal{C}$ . In this case  $\mathcal{C} \subset \mathbb{R}$ . We also say that the train has one degree of freedom since  $q$  is a scalar. The train also has one actuator (motor) that propels it forwards or backwards along the rail. With one motor and one degree of freedom the train is fully actuated and can achieve any desired configuration, that is, any position along the rail.

Another important concept is task space which is the set of all possible poses  $\xi$  of the vehicle and  $\xi \in \mathcal{T}$ . The task space depends on the application or task. If our task was motion along the rail then  $\mathcal{T} \subset \mathbb{R}$ . If we cared only about the position of the train in a plane then  $\mathcal{T} \subset \mathbb{R}^2$ . If we considered a 3-dimensional world then  $\mathcal{T} \subset SE(3)$ , and its height changes as it moves up and down hills and its orientation changes as it moves around curves. Clearly for these last two cases the dimensions of the task space exceed the dimensions of the configuration space and the train cannot attain an arbitrary pose since it is constrained to move along fixed rails. In these cases we say that the train moves along a manifold in the task space and there is a mapping from  $q \mapsto \xi$ .

Interestingly many vehicles share certain characteristics with trains – they are good at moving forward but not so good at moving sideways. Cars, hovercrafts, ships and aircraft all exhibit this characteristic and require complex manoeuvring in order to move sideways. Nevertheless this is a very sensible design approach since it caters to the motion we most commonly require of the vehicle. The less common motions such as parking a car, docking a ship or landing an aircraft are more complex, but not impossible, and humans can learn this skill. The benefit of this type of design comes from simplification and in particular reducing the number of actuators required.

Next consider a hovercraft which has two propellers whose axes are parallel but not collinear. The sum of their thrusts provide a forward force and the difference in thrusts generates a yawing torque for steering. The hovercraft moves over a planar surface and its configuration is entirely described by three generalized coordinates  $q = (x, y, \theta) \in \mathcal{C}$  and in this case  $\mathcal{C} \subset \mathbb{R}^2 \times \mathbb{S}$ . The configuration space has 3 dimensions and the vehicle therefore has three degrees of freedom.

The hovercraft has only two actuators, one fewer than it has degrees of freedom, and it is therefore an under-actuated system. This imposes limitations on the way in which it can move. At any point in time we can control the forward (parallel to the thrust vectors) acceleration and the rotational acceleration of the the hovercraft but there is zero sideways (or lateral) acceleration since it does not generate any lateral thrust. Nevertheless with some clever manoeuvring, like with a car, the hovercraft can follow a path that will take it to a place to one side of where it started. The advantage of under-actuation is the reduced number of actuators, in this case two instead of three. The penalty is that the vehicle cannot move directly to an any point in its configuration space, it must follow some path. If we added a third propeller to the hovercraft with its axis normal to the first two then it would be possible to command an arbitrary forward, sideways and rotational acceleration. The task space of the hovercraft is  $\mathcal{T} \subset SE(2)$  which is equivalent, in this case, to the configuration space.

A helicopter has four actuators. The main rotor generates a thrust vector whose magnitude is controlled by the collective pitch, and the thrust vector's direction is controlled by the lateral and longitudinal cyclic pitch. The fourth actuator, the tail rotor, provides a yawing moment. The helicopter's configuration can be described by six generalized coordinates  $q = (x, y, z, \theta_r, \theta_p, \theta_y) \in \mathcal{C}$  which is its position and orientation in 3-dimensional space, with orientation expressed in roll-pitch-yaw angles. The configuration space  $\mathcal{C} \subset \mathbb{R}^3 \times \mathbb{S}^3$  has six dimensions and therefore the vehicle has six degrees of freedom. The helicopter is under-actuated and it has no means to rotationally accelerate in the pitch and roll directions but cleverly these unactuated degrees of freedom are not required for helicopter operation – the helicopter naturally maintains stable equilibrium values for roll and pitch angle. Gravity acts like an additional actuator and provides a constant downward force. This allows the helicopter to accelerate sideways using the horizontal component of its thrust vector, while the vertical component of thrust is counteracted by gravity – without gravity a helicopter could not fly sideways. The task space of the helicopter is  $\mathcal{T} \subset SE(3)$ .

A fixed-wing aircraft moves forward very efficiently and also has four actuators (forward thrust, ailerons, elevator and rudder). The aircraft's thrust provides acceleration in the forward direction and the control surfaces exert various moments on the aircraft: rudder (yaw torque), ailerons (roll torque), elevator (pitch torque). The aircraft's configuration space is the same as the helicopter and has six dimensions. The aircraft is under-actuated and it has no way to accelerate in the lateral direction. The task space of the aircraft is  $\mathcal{T} \subset SE(3)$ .

The DEPTHX underwater robot shown on page 62 also has a configuration space  $\mathcal{C} \subset \mathbb{R}^3 \times \mathbb{S}^3$  of six dimensions, but by contrast is fully actuated. Its six actuators can exert an arbitrary force and torque on the vehicle, allowing it to accelerate in any direction or about any axis. Its task space is  $\mathcal{T} \subset SE(3)$ .

Finally we come to the wheel – one of humanity's greatest achievements. The wheel was invented around 3000 BCE and the two wheeled cart was invented around 2000 BCE. Today four wheeled vehicles are ubiquitous and the total automobile population of the planet is approaching one billion. The effectiveness of cars, and our familiarity with them, makes them a natural choice for robot platforms that move across the ground.

The configuration of a car moving over a plane is described by its generalized coordinates  $q = (x, y, \theta) \in \mathcal{C}$  and  $\mathcal{C} \subset \mathbb{R}^2 \times \mathbb{S}$  which has 3 dimensions. A car has only two actuators, one to move forwards or backwards and one to change the heading direction. The car is therefore under-actuated. As we have already remarked an under-actuated vehicle cannot move directly to an any point in its configuration space, it

Some low-cost hobby aircraft have no rudder and rely only on ailerons to bank and turn the aircraft. Even cheaper hobby aircraft have no elevator and rely on engine speed to control height.

<http://hypertextbook.com/facts/2001/MarinaStasenko.shtml>

Unlike the aircraft and underwater robot the motion of a car is generally considered in terms of velocities rather than forces and torques.



**Fig. 4.1.** Omni-directional (or Swedish) wheel. Note the circumferential rollers which make motion in the direction of the wheel's axis almost frictionless. (Courtesy Vex Robotics)

We can also consider this in control theoretic terms. Brockett's theorem (Brockett 1983) states that such systems are controllable but they cannot be stabilized to a desired state using a differentiable, or even continuous, pure state feedback controller. A time varying or non-linear control strategy is required.

must follow generally nonlinear some path. We know from our everyday experience with cars that it is not possible to drive sideways, but with some practice we can learn to follow a path that results in the vehicle being to one side of its initial position – this is parallel parking. Neither can a car rotate on spot, but we can follow a path that results in the vehicle being at the same position but rotated by  $180^\circ$  – a three-point turn. The challenges this introduces for control and path planning will be discussed in the rest of this part of the book. Despite this limitation the car is the simplest and most effective means of moving in a planar world that we have yet found.

The standard wheel is highly directional and prefers to roll in the direction normal to the axis of rotation. We might often wish for an ability to roll sideways but the standard wheel provides significant benefit when cornering – lateral friction between the wheels and the road counteracts, for free, the centripetal acceleration which would otherwise require an extra actuator to provide that force. More radical types of wheels have been developed that can roll sideways. An omni-directional wheel or Swedish wheel is shown in Fig. 4.1. It is similar to a normal wheel but has a number of passive rollers around its circumference and their rotational axes lie in the plane of the wheel. It is driven like an ordinary wheel but has very low friction in the lateral direction. A spherical wheel is similar to the roller ball in an old-fashioned computer mouse but driven by two actuators so that it can achieve a velocity in any direction.

In robotics a car is often described as a non-holonomic vehicle. The term non-holonomic comes from mathematics and means that the motion of the car is subject to one or more non-holonomic constraints. A holonomic constraint is an equation that can be written in terms of the configuration variables  $x$ ,  $y$  and  $\theta$ . A non-holonomic constraint can only be written in terms of the *derivatives* of the configuration variables and *cannot be integrated* to a constraint in terms of configuration variables. Such systems are therefore also known as non-integrable systems. A key characteristic of these systems, as we have already discussed, is that they cannot move directly from one configuration to another – they must perform a manoeuvre or sequence of motions.

A skid-steered vehicle, such as a tank, can turn on the spot but to move sideways it would have to stop, turn, proceed, stop then turn – this is a manoeuvre or time-varying control strategy which is the hallmark of a non-holonomic system. The tank has two actuators, one for each track, and just like a car is under-actuated.

Mobility parameters for the vehicles that we have discussed are tabulated in Table 4.1. The second column is the number of degrees of freedom of the vehicle or the dimension of its configuration space. The third column is the number of actuators and the fourth column indicates whether or not the vehicle is fully actuated.

## 4.2 Car-like Mobile Robots

Wheeled cars are a very effective class of vehicle and the archetype for most ground robots such as those shown on page 62. In this section we will create a model for a car-like vehicle and develop controllers that can drive the car to a point, along a line, follow an arbitrary path, and finally, drive to a specific pose.

**Table 4.1.**  
Summary of parameters for three different types of vehicle. The  $+g$  notation indicates that the gravity field can be considered as an extra actuator

Vehicle	Degrees of freedom	Number of actuators	Fully actuated?
Train	1	1	✓
Hovercraft	3	2	✗
Helicopter	6	$4+g$	✗
Fixed wing aircraft	6	$4+g$	✗
6-thruster AUV	6	6	✓
Car	3	2	✗

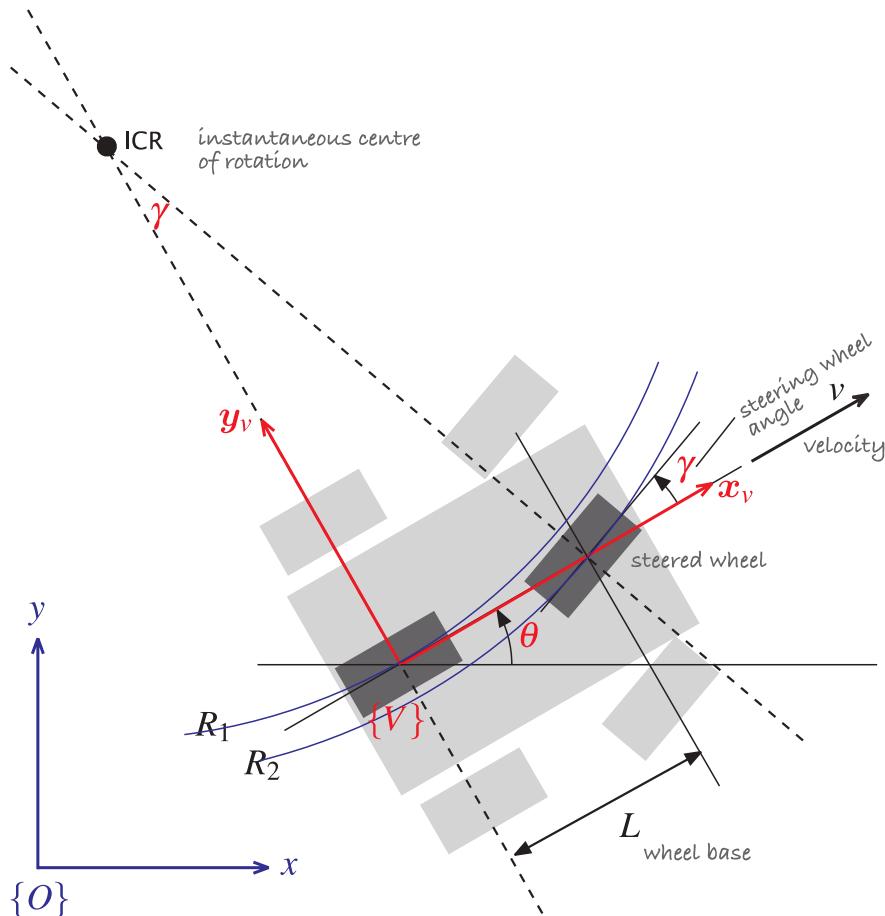


Fig. 4.2.

Bicycle model of a car. The car is shown in light grey, and the bicycle approximation is dark grey. The vehicle's coordinate frame is shown in red, and the world coordinate frame in blue. The steering wheel angle is  $\gamma$  and the velocity of the back wheel, in the  $x$ -direction, is  $v$ . The two wheel axes are extended as dashed lines and intersect at the Instantaneous Centre of Rotation (ICR) and the distance from the ICR to the back and front wheels is  $R_1$  and  $R_2$  respectively

A commonly used model for a four-wheeled car-like vehicle is the bicycle model shown in Fig. 15.1. The bicycle has a rear wheel fixed to the body and the plane of the front wheel rotates about the vertical axis to steer the vehicle.

The pose of the vehicle is represented by the coordinate frame  $\{V\}$  shown in Fig. 4.2, with its  $x$ -axis in the vehicle's forward direction and its origin at the centre of the rear axle. The configuration of the vehicle is represented by the generalized coordinates  $q = (x, y, \theta) \in \mathcal{C}$  where  $\mathcal{C} \subset SE(2)$ . The vehicle's velocity is by definition  $v$  in the vehicle's  $x$ -direction, and zero in the  $y$ -direction since the wheels cannot slip sideways. In the vehicle frame  $\{V\}$  this is

$$v_{\dot{x}} = v, \quad v_{\dot{y}} = 0$$

The dashed lines show the direction along which the wheels cannot move, the lines of no motion, and these intersect at a point known as the Instantaneous Centre of Rotation (ICR). The reference point of the vehicle thus follows a circular path and its angular velocity is

$$\dot{\theta} = \frac{v}{R_1} \tag{4.1}$$

and by simple geometry the turning radius is  $R_1 = L / \tan \gamma$  where  $L$  is the length of the vehicle or *wheel base*. As we would expect the turning circle increases with vehicle length. The steering angle  $\gamma$  is limited mechanically and its maximum value dictates the minimum value of  $R_1$ .

Often incorrectly called the Ackerman model.

Other well known models include the Reeds-Shepp model which has only three speeds: forward, backward and stopped, and the Dubins car which has only two speeds: forward and stopped.

Paths that arcs with smoothly varying radius.

**Vehicle coordinate system.** The coordinate system that we will use, and a common one for vehicles of all sorts is that the  $x$ -axis is forward (longitudinal motion), the  $y$ -axis is to the left side (lateral motion) which implies that the  $z$ -axis is upward. For aerospace and underwater applications the  $z$ -axis is often downward and the  $x$ -axis is forward.

For a fixed steering wheel angle the car moves along a circular arc. For this reason curves on roads are circular arcs or clothoids<sup>4</sup> which makes life easier for the driver since constant or smoothly varying steering wheel angle allow the car to follow the road. Note that  $R_2 > R_1$  which means the front wheel must follow a longer path and therefore rotate more quickly than the back wheel. When a four-wheeled vehicle goes around a corner the two steered wheels follow circular paths of different radius and therefore the angles of the steered wheels  $\gamma_L$  and  $\gamma_R$  should be very slightly different. This is achieved by the commonly used Ackerman steering mechanism which results in lower wear and tear on the tyres. The driven wheels must rotate at different speeds on corners which is why a differential gearbox is required between the motor and the driven wheels.

The velocity of the robot in the world frame is  $(v \cos \theta, v \sin \theta)$  and combined with Eq. 4.1 we write the equations of motion as

$$\begin{aligned}\dot{x} &= v \cos \theta \\ \dot{y} &= v \sin \theta \\ \dot{\theta} &= \frac{v}{L} \tan \gamma\end{aligned}\tag{4.2}$$

This model is referred to as a kinematic model since it describes the velocities of the vehicle but not the forces or torques that cause the velocity. The rate of change of heading  $\dot{\theta}$  is referred to as turn rate, heading rate or yaw rate and can be measured by a gyroscope. It can also be deduced from the angular velocity of the wheels on the left- and right-hand sides of the vehicle which follow arcs of different radius and therefore rotate at different speeds.

In the world coordinate frame we can write an expression for velocity in the vehicle's  $y$ -direction

$$\dot{y} \cos \theta - \dot{x} \sin \theta \equiv 0$$

which is the non-holonomic constraint. This equation cannot be integrated to form a relationship between  $x$ ,  $y$  and  $\theta$ .

Equation 4.2 captures some other important characteristics of a wheeled vehicle. When  $v = 0$  then  $\dot{\theta} = 0$ , that is, it is not possible to change the vehicle's orientation when it is not moving. As we know from driving we must be moving in order to turn. If the steering angle is  $\frac{\pi}{2}$  then the front wheel is orthogonal to the back wheel, the vehicle cannot move forward and the model enters an undefined region.



**Rudolph Ackermann (1764–1834)** was a German inventor born at Schneeberg, in Saxony. For financial reasons he was unable to attend university and became a saddler like his father. For a time he worked as a saddler and coach-builder and in 1795 established a print-shop and drawing-school in London. He published a popular magazine "The Repository of Arts, Literature, Commerce, Manufactures, Fashion and Politics" that included an eclectic mix of articles on water pumps, gas-lighting, and lithographic presses, along with fashion plates and furniture designs. He manufactured paper for landscape and miniature painters, patented a method for waterproofing cloth and paper and built a factory in Chelsea to produce it. He is buried in Kensal Green Cemetery, London.

In 1818 Ackermann took out British patent 4212 on behalf of the German inventor George Lankensperger for a steering mechanism which ensures that the steered wheels move on circles with a common centre. The same scheme was proposed and tested by Erasmus Darwin (grandfather of Charles) in the 1760s. Subsequent refinement by the Frenchman Charles Jeantaud led to the mechanism used in cars to this day which is known as Ackermann steering.

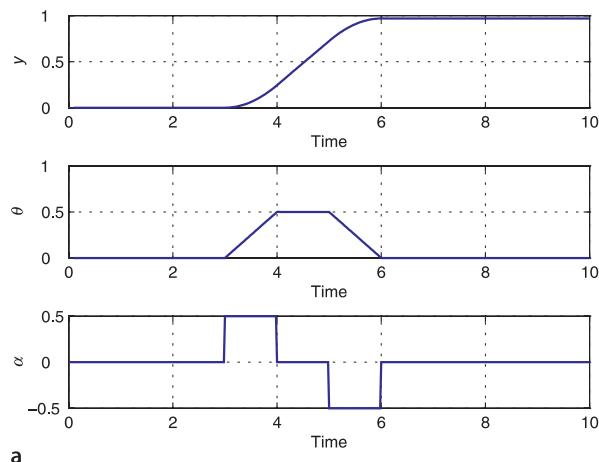
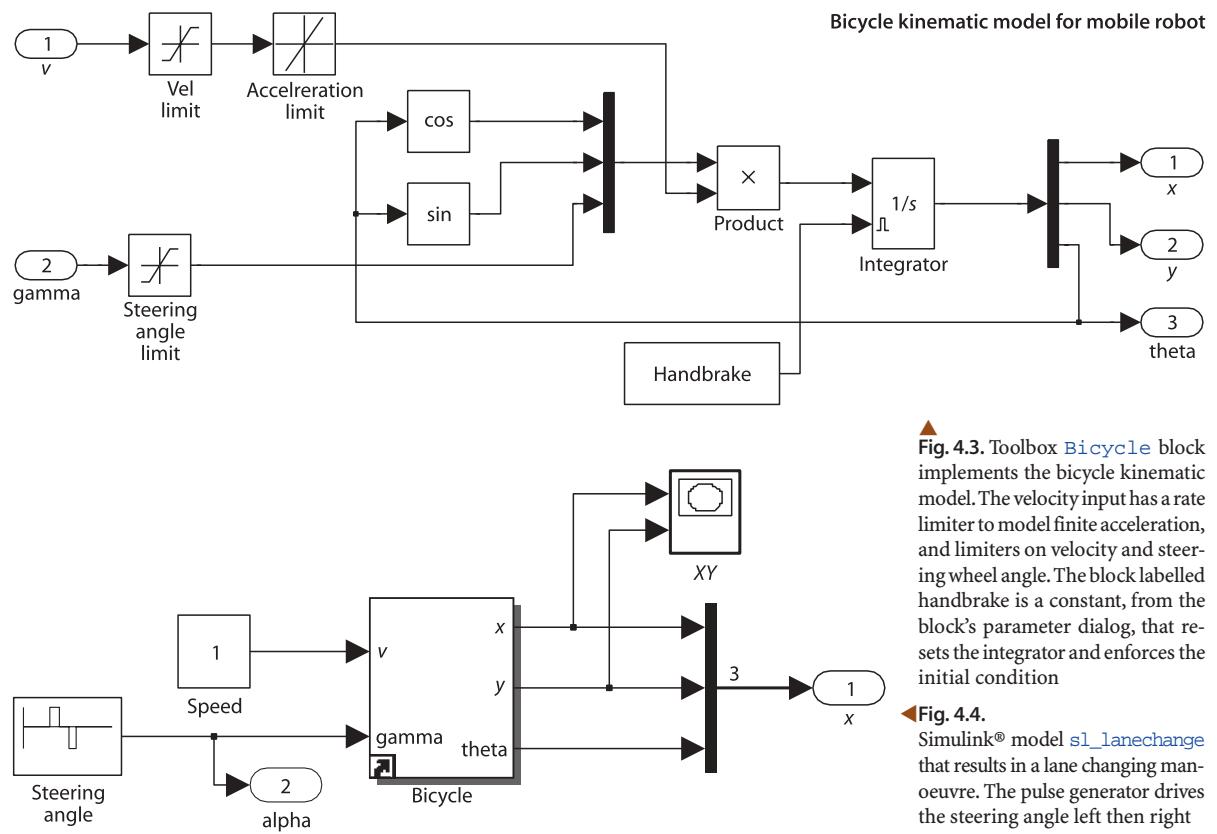
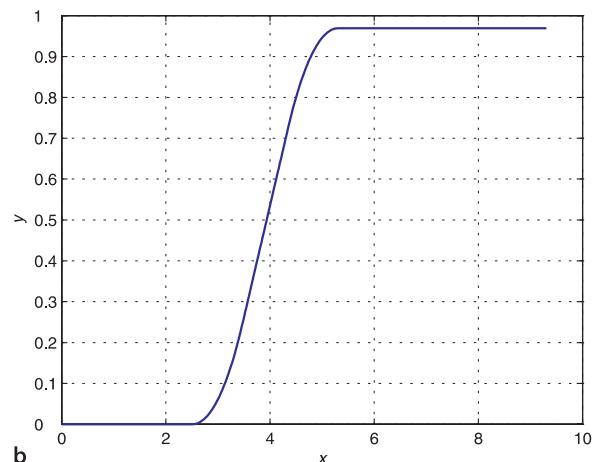


Figure 4.3 shows a Simulink® implementation of the bicycle model. It implements Eq. 4.2 and also includes a maximum velocity limit, a velocity rate limiter to model finite acceleration and a limiter on the steering angle to model the finite range of the steered wheel. The Simulink® system

>> `s1_lanechange`

shown in Fig. 4.4 uses the `Bicycle` block in a system with a constant velocity demand. The steering input is a positive then negative pulse on the steering angle and the configuration is plotted against time in Fig. 4.5a. The result, in the  $xy$ -plane, is shown in Fig. 4.5b and shows a simple *lane-changing* trajectory.



**Fig. 4.5.** Simple lane changing maneuver. **a** Vehicle response as a function of time, **b** motion in the  $xy$ -plane, the vehicle moves in the positive  $x$ -direction

## 4.2.1 Moving to a Point

Consider the problem of moving toward a goal point  $(x^*, y^*)$  in the plane. We will control the robot's velocity to be proportional to its distance from the goal

$$v^* = K_v \sqrt{(x^* - x)^2 + (y^* - y)^2}$$

and to steer toward the goal which is at the vehicle-relative angle

$$\theta^* = \tan^{-1} \frac{y^* - y}{x^* - x}$$

using a proportional controller

$$\gamma = K_h(\theta^* \ominus \theta), \quad K_h > 0$$

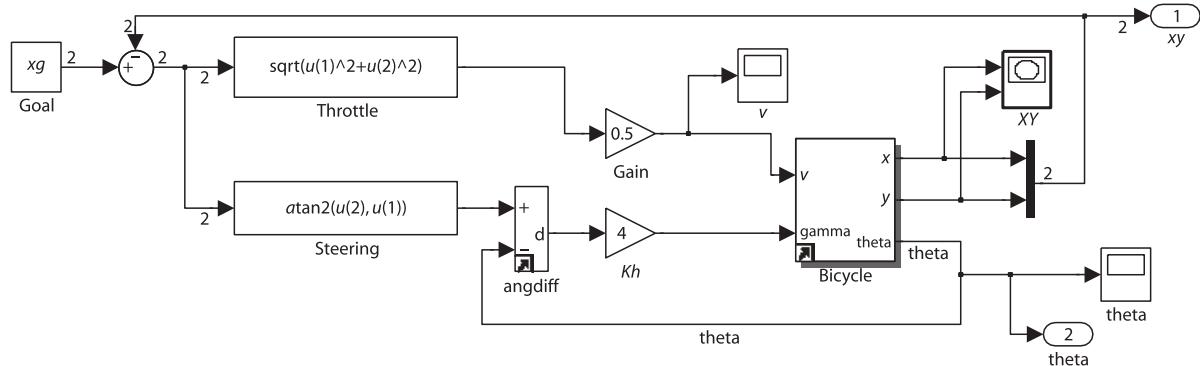
The Toolbox function `angdiff` computes the difference between two angles and returns a difference in the interval  $[-\pi, \pi]$ . Also available in the Toolbox Simulink® blockset `robblocks`.

which turns the steering wheel toward the target. Note the use of the  $\ominus$  operator since  $\theta^*, \theta \in \mathbb{S}$  we require the angular difference<sup>4</sup> to also lie within  $\mathbb{S}$ . A Simulink® model

`>> sl_drivepoint`

is shown in Fig. 4.6. We specify a goal coordinate

`>> xg = [5 5];`



**Fig. 4.6.** `sl_drivepoint`, the Simulink® model that drives the vehicle to a point

To run the Simulink® model called `model` we first load it

`>> model`

and a new window is popped up that displays the model in block-diagram form. The simulation can be started by typing `control-T` or by using `Simulation+Start` option from the toolbar on the model's window. The model can also be run directly from the MATLAB® command line

`>> sim('model')`

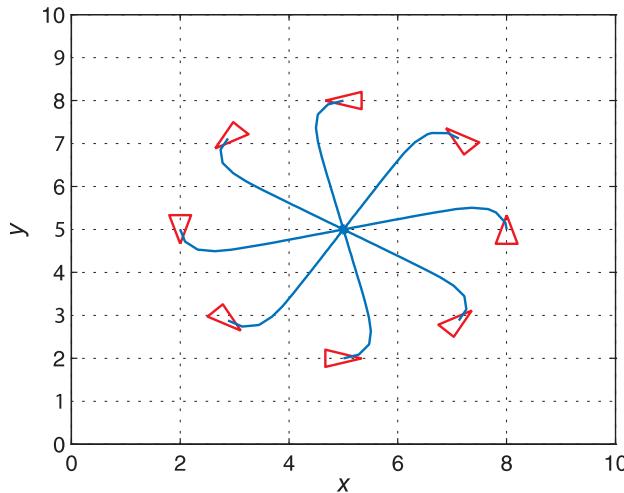
Many Toolbox models create additional figures to display robot animations or graphs.

Some models write data to the MATLAB® workspace for subsequent analysis. Some models simply have unconnected output ports. All models in this chapter have the simulation data export option set to `Format=Array`, so the signals are concatenated, in port number order, to form a row vector and these are stacked to form a matrix `yout` with one row per timestep. The corresponding time values form a vector `tout`. These variables can be returned from the simulation

`>> r = sim('model')`

in the object `r`. Displaying `r` lists the variables that it contains and their value is obtained using the `find` method, for example

`>> t = r.find('tout');`



**Fig. 4.7.**  
Simulation results for  
`sl_drivepoint` for different  
initial poses. The goal is (5, 5)

and an initial pose

```
>> x0 = [8 5 pi/2];
```

and then simulate the motion

```
>> r = sim('sl_drivepoint');
```

The variable `r` is an object that contains the simulation results from which we extract the configuration as a function of time

```
>> q = r.find('yout');
```

The vehicle's path in the plane is

```
>> plot(q(:,1), q(:,2));
```

which is shown in Fig. 4.7 for a number of starting poses. In each case the vehicle has moved forward and turned onto a path toward the goal point. The final part of each path is a straight line and the final orientation therefore depends on the starting point.

## 4.2.2 Following a Line

Another useful task for a mobile robot is to follow a line on the plane defined by  $ax + by + c = 0$ . This requires two controllers to adjust steering. One controller steers the robot to minimize the robot's normal distance from the line which according to Eq. I.1 is

$$d = \frac{(a, b, c) \cdot (x, y, 1)}{\sqrt{a^2 + b^2}}$$

The proportional controller

$$\alpha_d = -K_d d, K_d > 0$$

turns the robot toward the line. The second controller adjusts the heading angle, or orientation, of the vehicle to be parallel to the line

$$\theta^* = \tan^{-1} \frac{-a}{b}$$

using the proportional controller

$$\alpha_h = K_h(\theta^* \ominus \theta), K_h > 0$$

The combined control law

$$\gamma = -K_d d + K_h(\theta^* \ominus \theta)$$

turns the steering wheel so as to drive the robot toward the line and move along it.

The Simulink® model

`>> sl_driveline`

is shown in Fig. 4.8. We specify the target line as a 3-vector ( $a, b, c$ )

`>> L = [1 -2 4];`

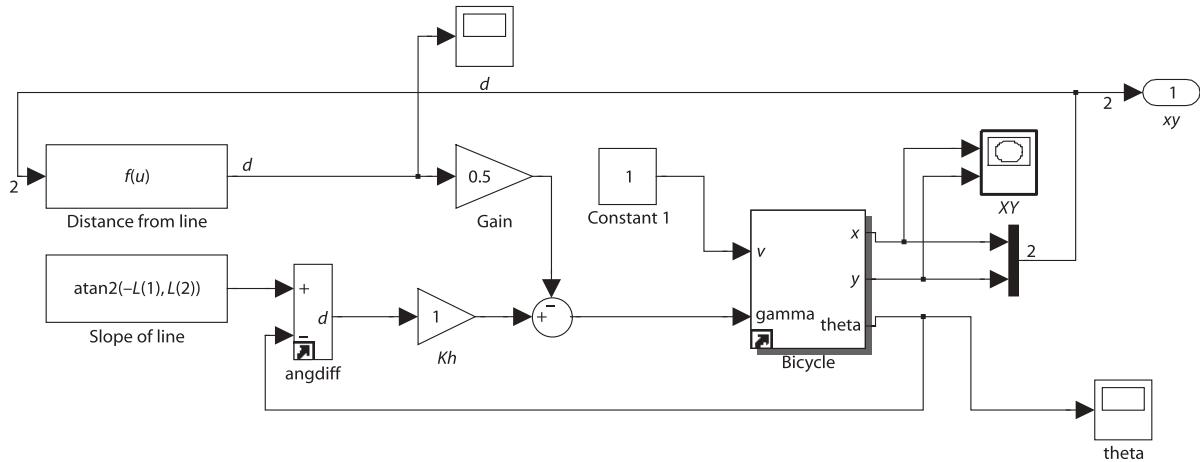
and an initial pose

`>> x0 = [8 5 pi/2];`

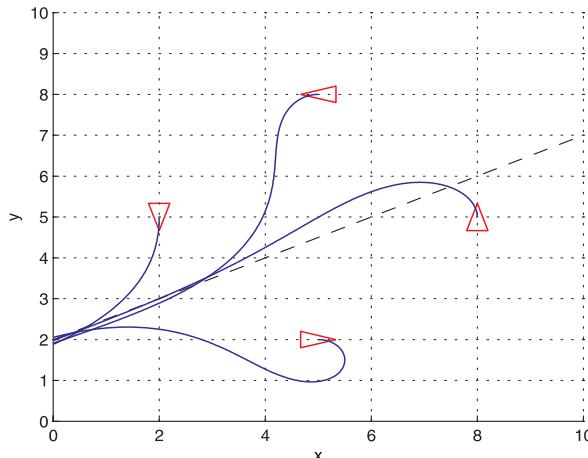
and then simulate the motion

`>> r = sim('sl_driveline');`

The vehicle's path for a number of different starting poses is shown in Fig. 4.9.



**Fig. 4.8.** The Simulink® model `sl_driveline` drives the vehicle along a line. The line parameters ( $a, b, c$ ) are set in the workspace variable `L`



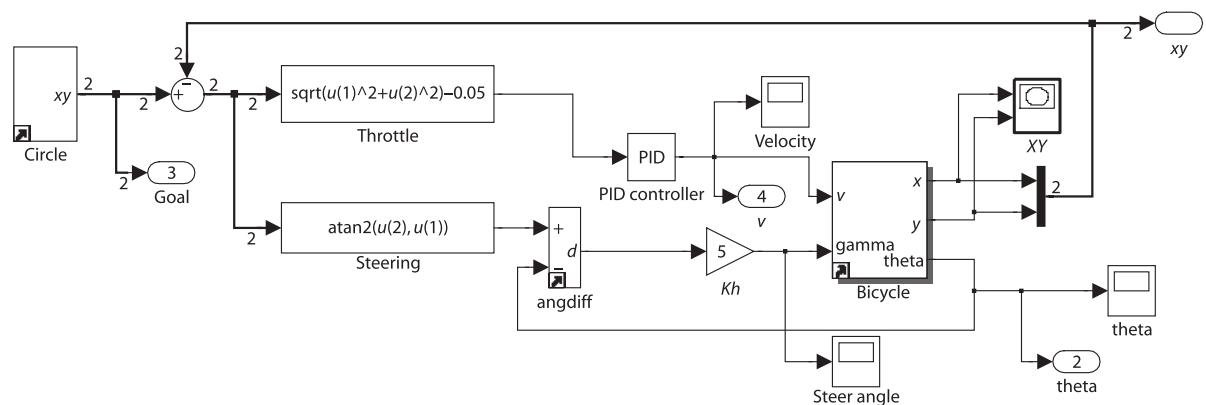
**Fig. 4.9.**

Simulation results from different initial poses for the line  $(1, -2, 4)$

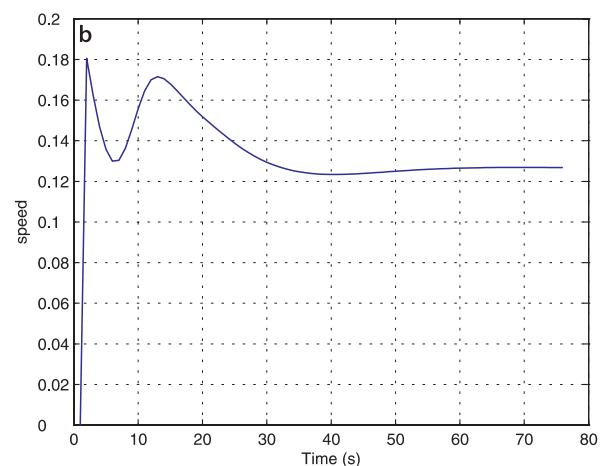
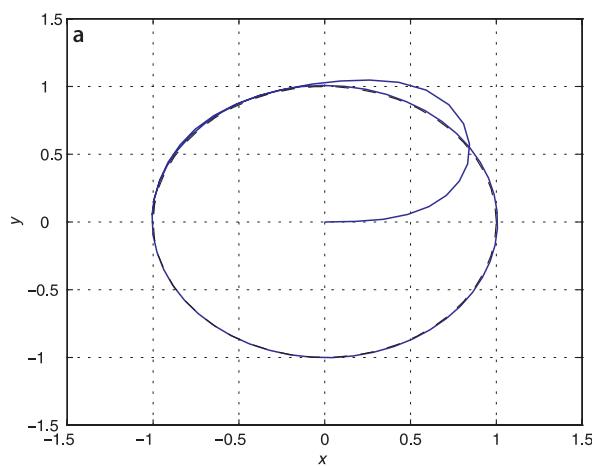
### 4.2.3 Following a Path

Instead of a straight line we might wish to follow a path that is defined more generally as some locus on the  $xy$ -plane. The path might come from a sequence of coordinates generated by a motion planner, such as discussed in Sect. 5.2, or in real-time based on the robot's sensors.

A simple and effective algorithm for path following is pure pursuit in which the goal ( $x^*(t), y^*(t)$ ) moves along the path, in its simplest form at constant speed, and the vehicle always heads toward the goal – think carrot and donkey.



**Fig. 4.10.** The Simulink® model [sl\\_pursuit](#) drives the vehicle to follow an arbitrary moving target using pure pursuit. In this example the vehicle follows a point moving around a unit circle with a frequency of 0.1 Hz



**Fig. 4.11.** Simulation results from pure pursuit. **a** Path of the robot in the  $xy$ -plane. The black dashed line is the circle to be tracked and the blue line in the path followed by the robot. **b** The speed of the vehicle versus time

This problem is very similar to the control problem we tackled in Sect. 4.2.1, moving to a point, except this time the point is moving. The robot maintains a distance  $d^*$  behind the pursuit point and we formulate an error

$$e = \sqrt{(x^* - x)^2 + (y^* - y)^2} - d^*$$

that we regulate to zero by controlling the robot's velocity using a proportional-integral (PI) controller

$$v^* = K_v e + K_i \int e dt$$

The integral term is required to provide a finite velocity demand  $v^*$  when the following error is zero. The second controller steers the robot toward the target which is at the relative angle

$$\theta^* = \tan^{-1} \frac{y^* - y}{x^* - x}$$

and a simple proportional controller

$$\alpha = K_h(\theta^* \ominus \theta), \quad K_h > 0$$

turns the steering wheel so as to drive the robot toward the target.

The Simulink® model

`>> sl_pursuit`

shown in Fig. 4.10 includes a target that moves around a unit circle. It can be simulated

`>> r = sim('sl_pursuit')`

and the results are shown in Fig. 4.11a. The robot starts at the origin but catches up to, and follows, the moving goal. Figure 4.11b shows how the speed demand picks up smoothly and converges to a steady state value at the desired following distance.

#### 4.2.4 Moving to a Pose

The final control problem we discuss is driving to a specific pose  $(x^*, y^*, \theta^*)$ . The controller of Fig. 4.6 could drive the robot to a goal position but the final orientation depended on the starting position.

In order to control the final orientation we first rewrite Eq. 4.2 in matrix form

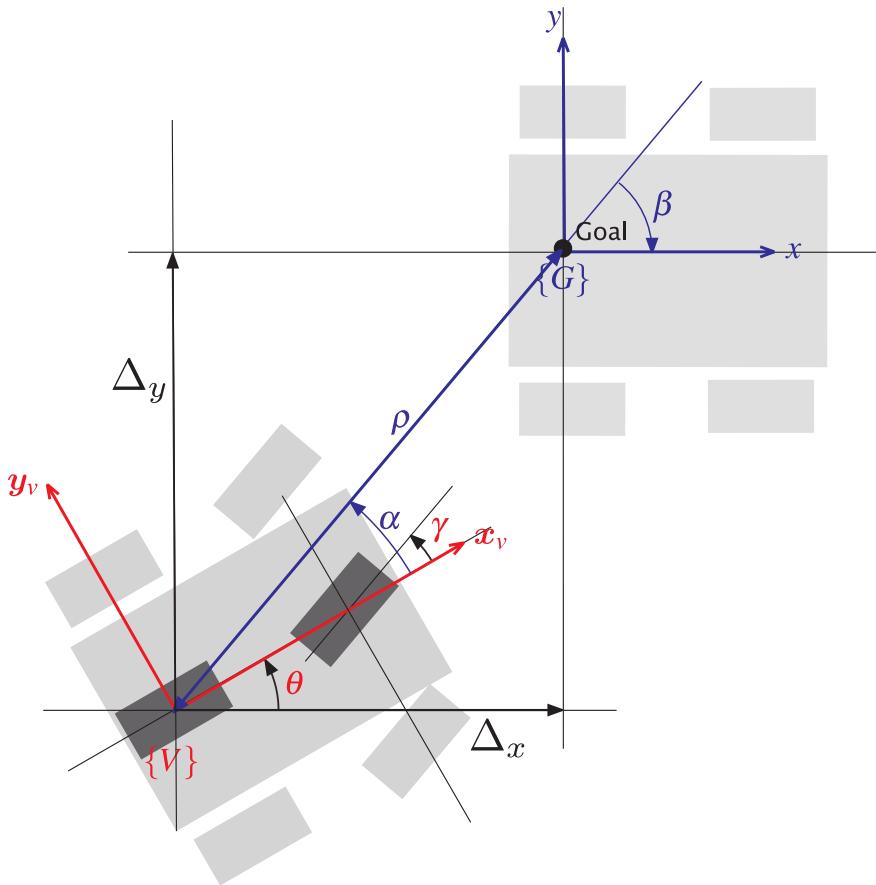
$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} v \\ \gamma \end{pmatrix}$$

and then transform the equations into polar coordinate form using the notation shown in Fig. 4.12. We apply a change of variables

$$\rho = \sqrt{\Delta_x^2 + \Delta_y^2}$$

$$\alpha = \tan^{-1} \frac{\Delta_y}{\Delta_x} - \theta$$

$$\beta = -\theta - \alpha$$



**Fig. 4.12.**  
Polar coordinate notation for the bicycle model vehicle moving toward a goal pose:  $\rho$  is the distance to the goal,  $\beta$  is the angle of the goal vector with respect to the world frame, and  $\alpha$  is the angle of the goal vector with respect to the vehicle frame

which results in

$$\begin{pmatrix} \dot{\rho} \\ \dot{\alpha} \\ \dot{\beta} \end{pmatrix} = \begin{pmatrix} -\cos\alpha & 0 \\ \frac{\sin\alpha}{\rho} & -1 \\ -\frac{\sin\alpha}{\rho} & 0 \end{pmatrix} \begin{pmatrix} v \\ \gamma \end{pmatrix}, \text{ if } \alpha \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right]$$

and assumes the goal  $\{G\}$  is in front of the vehicle. The linear control law

$$\begin{aligned} v &= k_\rho \rho \\ \gamma &= k_\alpha \alpha + k_\beta \beta \end{aligned}$$

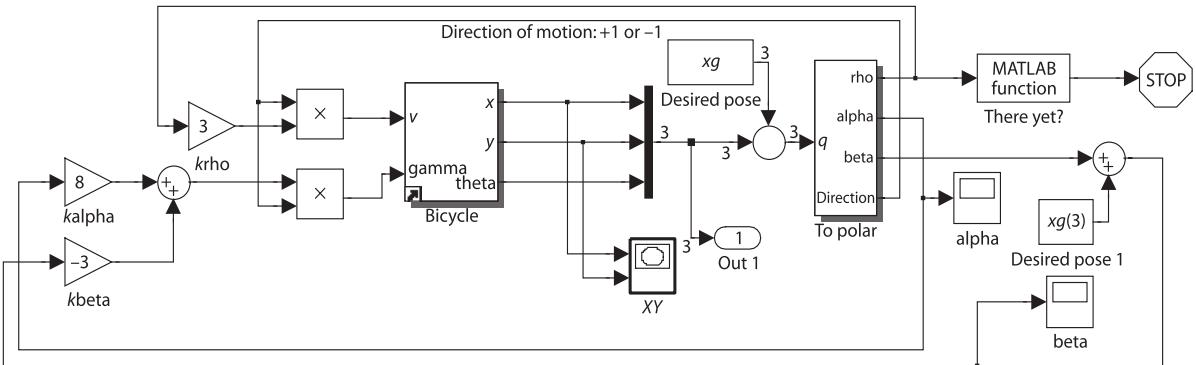
drives the robot to a unique equilibrium► at  $(\rho, \alpha, \beta) = (0, 0, 0)$ . The intuition behind this controller is that the terms  $k_\rho \rho$  and  $k_\alpha \alpha$  drive the robot along a line toward  $\{G\}$  while the term  $k_\beta \beta$  rotates the line so that  $\beta \rightarrow 0$ . The closed-loop system

$$\begin{pmatrix} \dot{\rho} \\ \dot{\alpha} \\ \dot{\beta} \end{pmatrix} = \begin{pmatrix} -k_\rho \cos\alpha \\ k_\rho \sin\alpha - k_\alpha \alpha - k_\beta \beta \\ -k_\rho \sin\alpha \end{pmatrix}$$

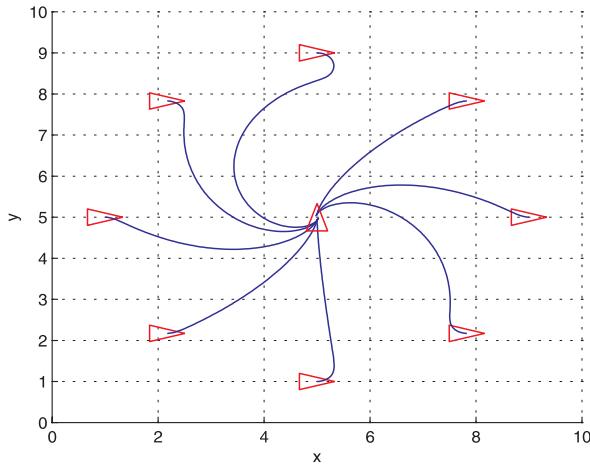
The control law introduces a discontinuity at  $\rho = 0$  which satisfies Brockett's theorem.

is stable so long as

$$k_\rho > 0, k_\beta < 0, k_\alpha - k_\rho > 0$$



**Fig. 4.13.** The Simulink® model `sl_drivepose` drives the vehicle to a pose. The initial and final poses are set by the workspace variable `x0` and `xf` respectively



**Fig. 4.14.** Simulation results from different initial poses to the final pose  $(5, 5, \frac{\pi}{2})$ . Note that in some cases the robot has *backed into* the final pose

The distance and bearing to the goal  $(\rho, \alpha)$  could be measured by a camera or laser range finder, and the angle  $\beta$  derived from  $\alpha$  and vehicle orientation  $\theta$  as measured by a compass.

For the case where the goal is behind the robot, that is  $\alpha \notin (-\frac{\pi}{2}, \frac{\pi}{2}]$ , we reverse the vehicle by negating  $v$  and  $\gamma$  in the control law. The velocity  $v$  always has a constant sign which depends on the initial value of  $\alpha$ .

So far we have described a *regulator* that drives the vehicle to the pose  $(0, 0, 0)$ . To move the robot to an arbitrary pose  $(x^*, y^*, \theta^*)$  we perform a change of coordinates

$$x' = x - x^*, \quad y' = y - y^*, \quad \theta' = \theta, \quad \beta = \beta' + \theta^*$$

The pose controller is implemented by the Simulink® model

```
>> sl_drivepose
```

shown in Fig. 4.13. We specify a goal pose

```
>> xg = [5 5 pi/2];
```

and an initial pose

```
>> x0 = [8 5 pi/2];
```

and then simulate the motion

```
>> r = sim('sl_drivepose');
```

As before, the simulation results are stored in `r` and can be plotted

```
>> y = r.find('yout');
>> plot(y(:,1), y(:,2));
```

to show the vehicle's path in the plane. The vehicle's path for a number of starting poses is shown in Fig. 4.14. The vehicle moves forwards or backward and takes a smooth path to the goal pose.

The controller is based on the linear bicycle model but the Simulink® model **Bicycle** has hard non-linearities including steering angle limits and velocity rate limiting.

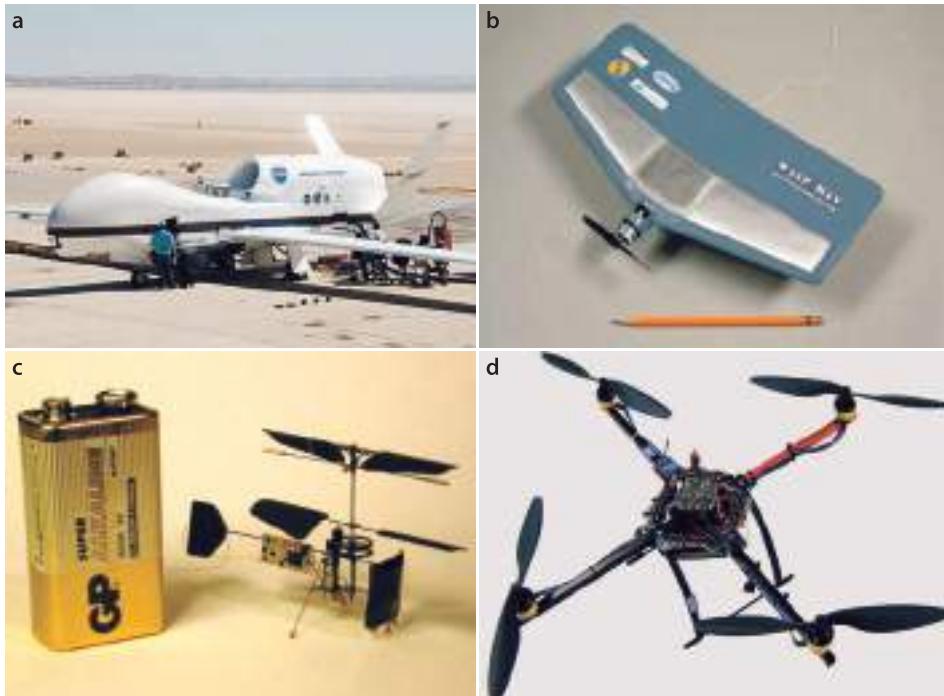
### 4.3 Flying Robots

*In order to fly, all one must do is simply miss the ground.*  
Douglas Adams

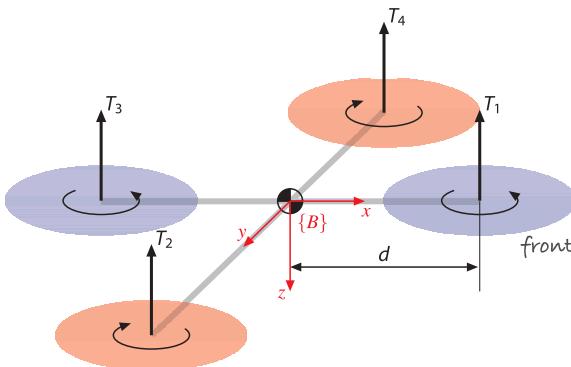
Flying robots or unmanned aerial vehicles (UAV) are becoming increasingly common and span a huge range of size and shape as shown in Fig. 4.15. Applications include military operations, surveillance, meteorological investigations and robotics research. Fixed wing UAVs are similar in principle to passenger aircraft with wings to provide lift, a propellor or jet to provide forward thrust and control surface for manoeuvring. Rotorcraft UAVs have a variety of configurations that include conventional helicopter design with a main and tail rotor, a *coax* with counter-rotating co-axial rotors and quadcopters. Rotorcraft UAVs are used for inspection and research and have the advantage of being able to take off vertically.

Flying robots differ from ground robots in some important ways. Firstly they have 6 degrees of freedom and their configuration  $q \in SE(3)$ . Secondly they are actuated by forces, that is their motion model is expressed in terms of forces and torques rather than velocities as was the case for the bicycle model – we use a dynamic rather than a kinematic model. Underwater robots have many similarities to flying robots and can be considered as vehicles that *fly through water* and there are underwater equivalents to fixed wing aircraft and rotorcraft. The principle differences underwater are an upward buoyancy force, drag forces that are much more significant than in air, and added mass.

In this section we will create a model for a quadcopter flying vehicle such as shown in Fig. 4.15d. Quadcopters are now widely available, both as commercial products and



**Fig. 4.15.**  
Flying robots. **a** Global Hawk unmanned aerial vehicle (UAV) (photo: courtesy of NASA), **b** a micro air vehicle (MAV) (photo: courtesy of AeroVironment, Inc.), **c** a 1 gram co-axial helicopter with 70 mm rotor diameter (photo courtesy of Petter Muren and Proxflyer AS), **d** a quad-copter, also known as an X4, which has four rotors and a block of sensing and control electronics in the middle (photo: courtesy of Inkyu Sa)



**Fig. 4.16.**  
Quad-rotor notation showing the four rotors, their thrust vectors and directions of rotation. The body-fixed frame  $\{B\}$  is attached to the vehicle and has its origin at the vehicle's centre of mass. Rotors 1 and 3 rotate counter-clockwise (viewed from above) while rotors 2 and 4 rotate clockwise

as open-source projects. Compared to fixed wing aircraft they are highly manoeuvrable and can be flown safely indoors which makes them well suited for laboratory or hobbyist use. Compared to conventional helicopters, with the large main rotor and tail rotor, the quadcopter is easier to fly, does not have the complex swash plate mechanism and is easier to model and control.

The notation for the quadcopter model is shown in Fig. 4.16. The body-fixed coordinate frame  $\{B\}$  has its  $z$ -axis downward following the aerospace convention. The quadcopter has four rotors, labelled 1 to 4, mounted at the end of each cross arm. The rotors are driven by electric motors powered by electronic speed controllers. Some low-cost quadcopters use small motors and reduction gearing to achieve sufficient torque. The rotor speed is  $\omega_i$  and the thrust is an upward vector

$$T_i = b\omega_i^2, \quad i = 1, 2, 3, 4 \quad (4.3)$$

in the vehicle's negative  $z$ -direction, where  $b > 0$  is the lift constant that depends on the air density, the cube of the rotor blade radius, the number of blades, and the chord length of the blade.

The translational dynamics of the vehicle in world coordinates is given by Newton's second law

$$m\dot{v} = \begin{pmatrix} 0 \\ 0 \\ mg \end{pmatrix} - {}^0R_B \begin{pmatrix} 0 \\ 0 \\ T \end{pmatrix} \quad (4.4)$$

where  $v$  is the velocity of the vehicle in the world frame,  $g$  is gravitational acceleration,  $m$  is the total mass of the vehicle and  $T = \sum T_i$  is the total upward thrust. The first term is the force of gravity which acts downward in the world frame and the second term is the total thrust in the vehicle frame rotated into the world coordinate frame.

Pairwise differences in rotor thrusts cause the vehicle to rotate. The torque about the vehicle's  $x$ -axis, the *rolling* torque, is

$$\tau_x = dT_4 - dT_2$$

where  $d$  is the distance from the motor to the centre of mass. We can write this in terms of rotor speeds by substituting Eq. 4.3

$$\tau_x = db(\omega_4^2 - \omega_2^2) \quad (4.5)$$

and similarly for the  $y$ -axis, the *pitching* torque is

$$\tau_y = db(\omega_1^2 - \omega_3^2) \quad (4.6)$$

The torque applied to each propellor by the motor is opposed by aerodynamic drag

$$Q_i = k\omega_i^2$$

where  $k$  depends on the same factors as  $b$ . This torque exerts a reaction torque on the airframe which acts to rotate the airframe about the propeller shaft in the opposite direction to its rotation. The total reaction torque about the  $z$ -axis is

$$\begin{aligned}\tau_z &= Q_1 - Q_2 + Q_3 - Q_4 \\ &= k(\omega_1^2 + \omega_3^2 - \omega_2^2 - \omega_4^2)\end{aligned}\quad (4.7)$$

where the different signs are due to the different rotation directions of the rotors. A yaw torque can be created simply by appropriate coordinated control of all four rotor speeds.

The rotational acceleration of the airframe is given by Euler's equation of motion

$$J\dot{\omega} = -\omega \times J\omega + \Gamma \quad (4.8)$$

where  $J$  is the  $3 \times 3$  inertia matrix of the vehicle,  $\omega$  is the angular velocity vector and  $\Gamma = (\tau_x, \tau_y, \tau_z)^T$  is the torque applied to the airframe according to Eq. 4.5 to 4.7.

The motion of the quadcopter is obtained by integrating the forward dynamics equations Eq. 4.4 and Eq. 4.8 where the forces and moments on the airframe

$$\begin{pmatrix} T \\ \Gamma \end{pmatrix} = \begin{pmatrix} -b & -b & -b & -b \\ 0 & -db & 0 & db \\ db & 0 & -db & 0 \\ k & -k & k & -k \end{pmatrix} \begin{pmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{pmatrix} = A \begin{pmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{pmatrix} \quad (4.9)$$

are functions of the rotor speeds. The matrix  $A$  is of full rank if  $b, k, d > 0$  and can be inverted

$$\begin{pmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{pmatrix} = A^{-1} \begin{pmatrix} T \\ \tau_x \\ \tau_y \\ \tau_z \end{pmatrix} \quad (4.10)$$

to give the rotor speeds required to apply a specified thrust  $T$  and moment  $\Gamma$  to the airframe.

To control the vehicle we will employ a nested control structure which we illustrate for pitch and  $x$ -translational motion. The innermost loop uses a proportional and derivative controller► to compute the required pitching torque on the airframe

$$\tau_y = K_p(\theta_p^* - \theta_p) + K_d(\dot{\theta}_p^* - \dot{\theta}_p)$$

based on the error between desired and actual pitch angle.► The gains  $K_{\tau_p}$  and  $K_{\tau_d}$  are determined by classical control design approaches based on an approximate dynamic model and then tuned to achieve good performance. The actual vehicle pitch angle  $\theta_p$  would be estimated by an inertial navigation system. The required rotor speeds are then determined using Eq. 4.10.

Consider a coordinate frame  $\{V\}$  attached to the vehicle and with the same origin as  $\{B\}$  but with its  $x$ - and  $y$ -axes parallel to the ground. To move the vehicle in the  $V_x$ -direction we pitch the nose down which generates a force

$$f = R_y(\theta_p) \begin{pmatrix} 0 \\ 0 \\ T \end{pmatrix} = \begin{pmatrix} T \sin \theta_p \\ 0 \\ T \cos \theta_p \end{pmatrix}$$

The rotational dynamics has a second-order transfer function of  $\Theta_y(s)/\tau_y(s) = 1/(J_s^2 + B_s)$  where  $B$  is aerodynamic damping which is generally quite small. To regulate a second-order system requires a proportional-derivative controller.

The term  $\dot{\theta}_p^*$  is commonly ignored.

which has a component

$$f_x = T \sin \theta_p \approx T \theta_p$$

that accelerates the vehicle in the  ${}^Vx$ -direction. We can control the velocity in this direction with a proportional control law

$$f_x^* = m K_f \left( {}^Vv_x^* - {}^Vv_x \right)$$

Combine these two equations we obtain the pitch angle

$$\theta_p^* = \frac{m}{T} K_f \left( {}^Vv_x^* - {}^Vv_x \right) \quad (4.11)$$

required to achieve the desired forward velocity. The actual vehicle velocity  ${}^Vv_x$  would be estimated by an inertial navigation system or GPS receiver. For a vehicle in vertical equilibrium the total thrust equals the weight force so  $m / T \approx 1 / g$ .

If the position of the vehicle in the  $xy$ -plane of the world frame is  $\mathbf{p} \in \mathbb{R}^2$  then the desired velocity is given by the proportional control law

$$v^* = K_p (\mathbf{p}^* - \mathbf{p}) \quad (4.12)$$

based on the error between the desired and actual position. The desired velocity in frame  $\{V\}$  is

$${}^Vv = {}^V\mathbf{R}_0(\theta_y)v = {}^0\mathbf{R}_V^T(\theta_y)v$$

which is a function of the yaw angle  $\theta_y$

$$\begin{pmatrix} {}^Vv_x \\ {}^Vv_y \end{pmatrix} = {}^0\mathbf{R}_V^T(\theta_y) \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

To reach a desired position we can compute the appropriate velocity and from that the appropriate pitch angle which generates a force to move the vehicle. This indirection is a consequence of the vehicle being underactuated – we have just four rotor speeds to adjust but the vehicle's configuration space is 6-dimensional. To move forward the quadcopter airframe must first pitch down so that the thrust vector has a horizontal component to accelerate it. As it approaches its goal the airframe must be rotated in the opposite direction, pitching up, so that the backward component of thrust decelerates the forward motion. Finally the airframe rotates to the horizontal with the thrust vector vertical. The cost of under-actuation is once again a manoeuvre. The pitch angle cannot be arbitrarily set, it is a means to achieve translation control.

The total thrust must be increased so that the vertical thrust component still balances gravity.

The rotational inertia of a body that moves in  $SE(3)$  is represented by the  $3 \times 3$  symmetric matrix

$$\mathbf{J} = \begin{pmatrix} J_{xx} & J_{xy} & J_{xz} \\ J_{xy} & J_{yy} & J_{yz} \\ J_{xz} & J_{yz} & J_{zz} \end{pmatrix}$$

The diagonal elements are the moments of inertia, and the off-diagonal elements are products of inertia. Only six of these nine elements are unique: three moments and three products of inertia. The products of inertia are zero if the object's mass distribution is symmetrical with respect to the coordinate frame.

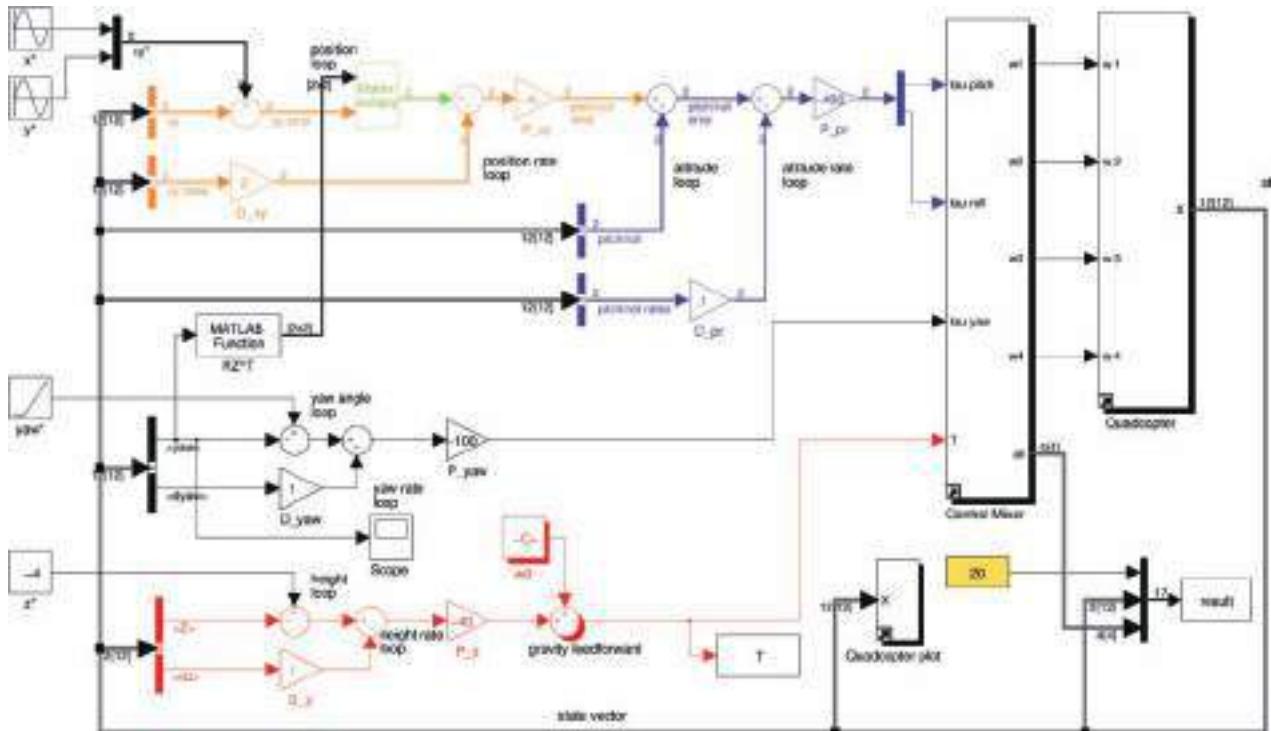


Figure 4.17 shows a Simulink® model of the quadcopter in a closed-loop control structure. The inputs to the quadcopter block are the speeds of the four rotors and from Eq. 4.9 the torques and forces on the quadcopter are computed and it integrates Eq. 4.4, Eq. 4.8 and Eq. 4.9 to give the position, velocity, orientation and orientation rate. The output of this block is the state vector  $\mathbf{x} = (x, y, z, \theta_r, \dot{\theta}_r, \theta_p, \dot{\theta}_p, \dot{x}, \dot{y}, \dot{z}, \dot{\theta}_r, \dot{\theta}_p, \dot{\theta}_y)$ . As is common in aerospace applications we represent orientation and orientation rate in terms of roll-pitch-yaw angles. The control part of the block diagram involves multiple nested control loops that compute the required thrust and torques so that the vehicle moves to the setpoint.

The vehicle's position control loops, as just discussed, are shown in the top left of the diagram. The innermost loop, shown in blue, controls the attitude of the vehicle and its inputs are the actual and desired roll and pitch angles, as well as the roll and pitch angular rates to provide damping. The outer loop, shown in orange, controls the  $xy$ -position of the flyer by requesting changes in roll and pitch angle so as to provide a component of thrust in the direction of desired  $xy$ -plane motion. In the diagram Eq. 4.11 and Eq. 4.12 have been combined into the form

$$\theta_p^* = K_1 \left( {}^V p_x^* - {}^V p_x - K_2 {}^V v_x \right)$$

The  $xy$ -position error is computed in the world frame and rotated by  ${}^0 R_V(\theta_y)$  into frame  $\{V\}$ . Note that according to the coordinate conventions shown in Fig. 4.16  ${}^V x$ -direction motion requires a negative rotation about the  $y$ -axis (pitch angle) and  ${}^V y$ -direction motion requires a positive rotation about the  $x$ -axis (roll angle) so the gains have different signs for the roll and pitch loops.

Yaw is controlled by a proportional-derivative controller

$$\tau_z = K_p (\theta_y^* - \theta_y) + K_d (\dot{\theta}_y^* - \dot{\theta}_y)$$

shown in black and  $\dot{\theta}_y^*$  is ignored since it is typically small.

**Fig. 4.17.** The Simulink® model [s1\\_quadcopter](#) which is a closed-loop simulation of the quadcopter. The flyer takes off and flies in a circle at constant altitude. The dynamics block implements Eq. 4.9, and the mixer block implements its inverse while also enforcing limits on rotor speed. A Simulink® bus is used for the 12-element state vector  $\mathbf{X}$  output by the [Quadcopter](#) block

Altitude is controlled by a proportional-derivative controller

$$T = K_p(z^* - z) + K_d(\dot{z}^* - \dot{z}) + \omega_0$$

shown in red which determines the average rotor speed. The additive term

$$\omega_0 = \sqrt{\frac{mg}{4b}} \quad (4.13)$$

is the rotor speed necessary to generate a thrust equal to the weight of the vehicle. This is an example of feedforward control – used here to counter the effect of gravity which otherwise is a constant disturbance to the altitude control loop. The alternatives to feedforward control would be to have very high gain for the altitude loop which often leads to actuator saturation and instability, or a proportional-integral (PI) controller which might require a long time for the integral term to increase to a useful value and then lead to overshoot. We will revisit gravity compensation in Chap. 9 applied to arm-type robots.

The parameters of a specific quadcopter can be loaded

```
>> mdl_quadcopter
```

which creates a structure called `quad` in the workspace, and its elements are the various dynamic properties of the quadcopter. The simulation can be run using the Simulink® menu or from the MATLAB® command line

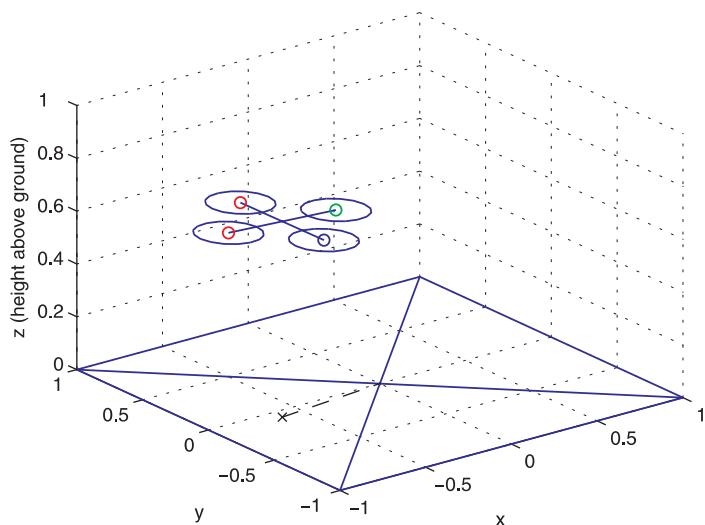
```
>> sim('sl_quadcopter');
```

and it displays an animation in a separate window. The vehicles lifts off and flies in a circle while spinning slowly about its own  $z$ -axis. A snapshot is shown in Fig. 4.18. The simulation writes the results from each timestep into a matrix in the workspace

```
>> about(result)
result [double] : 419x17 (56984 bytes)
```

which has one row per timestep, and each row contains the time followed by the state vector (elements 2–13) and the commanded rotor speeds  $\omega_i$  (elements 14–17). To plot  $x$  and  $y$  versus time is

```
>> plot(result(:,1), result(:,2:3));
```



**Fig. 4.18.**

One frame from the quadcopter simulation. The marker on the ground plane is a projection of the vehicle's centroid

## 4.4 Wrapping Up

In this chapter we have discussed general concepts about mobility, configuration space and task space. We created detailed models of two quite different robot platforms. We first discussed the car-like vehicle which is an exemplar of many ground robots. We developed a kinematic model which we used to develop a number of different controllers in order that the platform could perform useful tasks such as driving to a point, following a path or driving to a pose. We then discussed a simple flying vehicle, the quadcopter, and developed a dynamic model. We then implemented a number of nested control loops that allowed the quadcopter to fly a circuit. The nested control approach is described in more detail in Sect. 9.4.2.

The next chapters will discuss how to plan paths for robots through complex environments that contain obstacles and then how to determine the location of a robot.

### Further Reading

Comprehensive modelling of mobile ground robots is provided in the book by Siegwart et al. (2011). In addition to the bicycle model covered here, it presents in depth discussion of a variety of wheel configurations with different combinations of driven wheels, steered wheels and passive casters. These topics are covered more succinctly in the Handbook of Robotics (Siciliano and Khatib 2008, § 17). Siegwart et al. also provide strong coverage of perception, localization and navigation which we will discuss in the coming chapters.

Ackermann's magazine can be found online at <http://smithandgosling.wordpress.com/2009/12/02/ackermann-s-repository-of-arts> and the carriage steering mechanism is published in the March and April issues of 1818. King-Hele (2002) provides a comprehensive discussion about the prior work of Erasmus Darwin invention of the steering geometry and Darwin's earlier invention.

Mobile ground robots are now a mature technology for transporting parts around manufacturing plants. The research frontier is now for vehicles that operate autonomously in outdoor environments (Siciliano and Khatib 2008, part F). Research into the automation of passenger cars has been ongoing since the 1980s but as yet there is no commercial offering – perhaps society is not yet ready for this technology or perhaps the legal complexities that might arise in the case of accidents is overwhelming.

The Navlab project at Carnegie-Mellon University started in 1984 and a series of autonomous vehicles, Navlabs, have been built and a large body of research has resulted. All vehicles make strong use of computer vision for navigation. In 1995 the supervised autonomous Navlab 5 made a 3 000-mile journey, dubbed “No Hands Across America” (Pomerleau and Jochem 1995, 1996). The vehicle steered itself 98% of the time largely by visual sensing of the white lines at the edge of the road.

In Europe, Ernst Dickmanns and his team at Bundeswehr Universität München demonstrated autonomous control of vehicles. In 1988 the VaMoRs system, a 5 tonne Mercedes-Benz van, could drive itself at speeds over  $90 \text{ km h}^{-1}$  (Dickmanns and Graefe 1988b; Dickmanns and Zapp 1987; Dickmanns 2007). The European Prometheus Project ran from 1987–1995 and in 1994 the robot vehicles VaMP and VITA-2 drove more than 1 000 km on a Paris multi-lane highway in standard heavy traffic at speeds up to  $130 \text{ km h}^{-1}$ . They demonstrated autonomous driving in free lanes, convoy driving, automatic tracking of other vehicles, and lane changes with autonomous passing of other cars. In 1995 an autonomous S-Class Mercedes-Benz made a 1 600 km trip from Munich to Copenhagen and back. On the German Autobahn speeds exceeded  $175 \text{ km h}^{-1}$  and the vehicle executed traffic manoeuvres such as overtaking. The mean time between human interventions was 9 km and it drove up to 158 km without any human intervention. The UK part of the project demonstrated autonomous driving of an XJ6 Jaguar with vision (Matthews et al. 1995) and radar-based sensing for lane

keeping and collision avoidance. More recently, in the USA a series of Grand Challenges were run for autonomous cars. The 2005 desert and 2007 urban challenges are comprehensively described in compilations of papers from the various teams in Buehler et al. (2007, 2010).

Flying robots and underwater are not yet discussed in standard robotics texts. The Handbook of Robotics (Siciliano and Khatib 2008) provides summaries of the state of the art of aerial and underwater robotics in chapters 41 and 43 respectively. The theory of helicopters with an emphasis on robotics is provided by Mettler (2003) but the definitive reference for helicopter dynamics is the very large book by Prouty (2002). The recent book by Antonelli (2006), now in second edition, provides comprehensive coverage of modelling and control of underwater robots.

Some of the earliest papers on quadcopter modelling and control are by Pounds, Mahony and colleagues (Hamel et al. 2002; Pounds et al. 2004, 2006). The thesis by Pounds (2007) presents comprehensive aerodynamic modelling of a quadcopter with a particular focus on blade flapping, a phenomena well known in conventional helicopters but mostly ignored for quadcopters. Quad-rotors have been built at a number of laboratories and some are available commercially for hobbyists or for researchers. The MikroKopter open-source project has designs for the airframe and propulsion system as well as control software and can be found at <http://www.mikrokopter.de/ucwiki/en>. The basic principle of the quadcopter is easily extended by adding more rotors and vehicles with 6, 8 and 16 rotors have been developed and this provides increasing payload capability and even redundancy to rotor failures.

---

### Exercises

1. For a 4-wheel vehicle with  $L = 2$  m and width between wheel centres of 1.5 m
  - a) compute the difference in wheel steer angle for Ackerman steering around curves of radius 10, 50 and 100 m.
  - b) If the vehicle is moving at  $80 \text{ km h}^{-1}$  compute the difference in wheel rotation rates for curves of radius 10, 50 and 100 m.
2. Write an expression for turn rate in terms of the angular rotation rate of the two wheels. Investigate the effect of errors in wheel radius and vehicle width.
3. Implement the  $\ominus$  operator used in Sect. 4.2.1 and check against the code for [angdiff](#).
4. For Sect. 4.2.1 plot  $x$ ,  $y$  and  $\theta$  against time.
5. Pure pursuit example (page 74)
  - a) Investigate what happens when the integral gain is zero. Now reduce the frequency of circular motion to  $0.01 \text{ rev s}^{-1}$  and see what happens.
  - b) With integral set to zero, add a constant to the output of the controller. What should the value of the constant be?
  - c) Modify the pure pursuit example so the robot follows a slalom course.
  - d) Modify the pure pursuit example to follow a target moving back and forth along a line.
6. Moving to a pose (page 75)
  - a) Repeat the example with a different initial orientation.
  - b) Implement a parallel parking manoeuvre. Is the resulting path practical?
7. Use the MATLAB® GUI interface to make a simple steering wheel and velocity control, use this to create a very simple driving simulator. Alternatively interface a gaming steering wheel and peddle to MATLAB®.
8. Quadcopter (page 78)
  - a) Experiment with different control gains. What happens if you reduce the damping gains to zero?
  - b) Remove the gravity feedforward and experiment with high altitude-gain or a PI controller.
  - c) Derive Eq. 4.13.

- d) When the vehicle has non-zero roll and pitch angles, the magnitude of the vertical thrust is reduced and the vehicle will slowly descend. Add compensation to the vertical thrust to correct this.
- e) Simulate the quadcopter flying inverted, that is, its  $z$ -axis is pointing upwards.
- f) Program a ballistic motion. Have the quadcopter take off at 45 deg to horizontal then remove all thrust.
- g) Program a smooth landing.
- h) Program a barrel roll manoeuvre. Have the quadcopter fly horizontally in its  $x$ -direction and then increase the roll angle from 0 to  $2\pi$ .
- i) Program a flip manoeuvre. Have the quadcopter fly horizontally in its  $x$ -direction and then increase the pitch angle from 0 to  $2\pi$ .
- j) Add another four rotors.
- k) Use the function `mstraj` to create a trajectory through ten via points  $(X_p, Y_p, Z_p, \theta_y)$  and modify the controller of Fig. 4.17 for smooth pursuit of this trajectory.
- l) Use the MATLAB® GUI interface to make a simple joystick control, and use this to create a very simple flying simulator. Alternatively interface a gaming joystick to MATLAB®.

# 5 Navigation

*the process of directing a vehicle so as to reach the intended destination*  
IEEE Standard 172-1983



Robot navigation is the problem of guiding a robot towards a goal. The human approach to navigation is to make maps and erect signposts, and at first glance it seems obvious that robots should operate the same way. However many robotic tasks can be achieved without any map at all, using an approach referred to as *reactive navigation*. For example heading towards a light, following a white line on the ground, moving through a maze by following a wall, or vacuuming a room by following a random path. The robot is reacting directly to its environment: the intensity of the light, the relative position of the white line or contact with a wall. Grey Walter's tortoise Elsie from page 61 demonstrated "life-like" behaviours – she *reacted* to her environment and could seek out a light source. Today more than 5 million Roomba vacuum cleaners are cleaning floors without using any map of the rooms they work in. The robots work by making random moves and sensing only that they have made contact with an obstacle.

The more familiar human-style *map-based navigation* is used by more sophisticated robots. This approach supports more complex tasks but is itself more complex. It imposes a number of requirements, not the least of which is a map of the environment. It also requires that the robot's position is always known. In the next chapter we will discuss how robots can determine their position and create maps. The remainder of this chapter discusses the reactive and map-based approaches to robot navigation with a focus on wheeled robots operating in a planar environment.



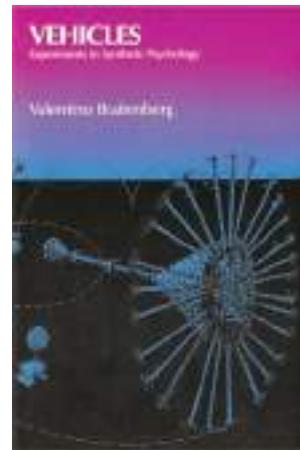
Fig. 5.1.

Time lapse photograph of a Roomba robot cleaning a room  
(photo by Chris Bartlett)

Valentino Braitenberg (1926–) is an Italian-Austrian neuro-scientist and cyberneticist, and former director at the Max Planck Institute for Biological Cybernetics in Tübingen, Germany. His 1986 book “*Vehicles: Experiments in Synthetic Psychology*” (image on right is of the cover this book, published by The MIT Press, ©MIT 1984) describes reactive goal-achieving vehicles, and such systems are now commonly known as Braitenberg Vehicles.

A Braitenberg vehicle is an automaton or robot which combines sensors, actuators and their direct interconnection to produce goal-oriented behaviors. Grey Walter’s tortoise predates the use of this term but is nevertheless an example of such a vehicle.

These vehicles are described as conceptually as analog circuits, but more recently small robots based on a digital realization of the same principles have been developed.



## 5.1 Reactive Navigation

Surprisingly complex tasks can be performed by a robot even if it has no map and no real *idea* about where it is. As already mentioned robotic vacuum cleaners use only random motion and information from contact sensors to perform a complex task as shown in Fig. 5.1. Insects such as ants and bees gather food and return it to the nest based on input from their senses, they have far too few neurons to create any kind of mental map of the world and plan paths through it. Even single-celled organisms such as flagellate protozoa exhibited goal seeking behaviours. In this case we need to revise our earlier definition of a robot to

*a goal oriented machine that can sense, plan and act.*

The manifestation of complex behaviours by simple organisms was of interest to early researchers in cybernetics. Grey Walter’s robotic tortoise demonstrated that it could move toward a light source, a behaviour known as phototaxis.<sup>►</sup> This was an important result in the then emerging scientific field of cybernetics.

More generally a *taxis* is the response of an organism to a stimulus gradient.

### 5.1.1 Braitenberg Vehicles

A very simple class of goal achieving robots are known as Braitenberg vehicles and are characterised by direct connection between sensors and motors. They have no explicit internal representation of the environment in which they operate and nor do they make explicit plans.<sup>►</sup>

Consider the problem of a robot moving in two dimensions that is seeking the maxima of a scalar field – the field could be light intensity or the concentration of some chemical.<sup>►</sup> The Simulink® model

`>> sl_braitenberg`

shown in Fig. 5.2 achieves this using a steering signal derived directly from the sensors.<sup>►</sup>

This is a fine philosophical point, the plan could be considered to be implicit in the details of the connections between the motors and sensors.

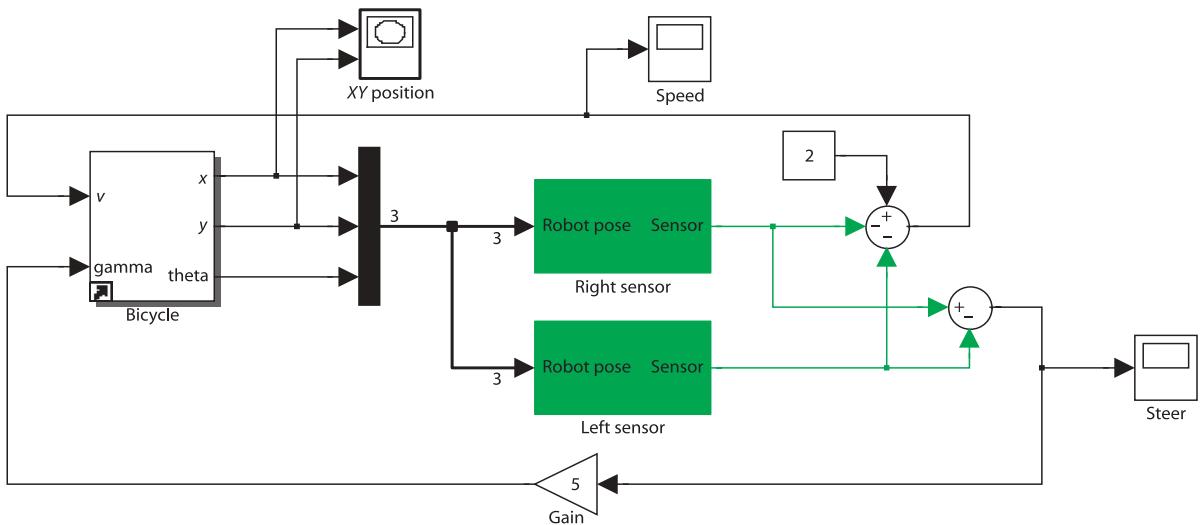
This is similar to the problem of moving to a point discussed in Sect. 4.2.1.

This is similar to Braitenberg’s Vehicle 4a.

William Grey Walter (1910–1977) was a neurophysiologist and pioneering cyberneticist born in Kansas City, Missouri and studied at King’s College, Cambridge. Unable to obtain a research fellowship at Cambridge he worked on neurophysiological research in hospitals in London and from 1939 at the Burden Neurological Institute in Bristol. He developed electroencephalographic brain topography which used multiple electrodes on the scalp and a triangulation algorithm to determine the amplitude and location of brain activity. (Image: courtesy of the Reuben Hoggett Collection)

Walter was influential in the then new field of cybernetics. He built robots to study how complex reflex behavior could arise from neural interconnections. His tortoise Elsie (of the species *Machina Speculatrix*) is shown, without its shell, on page 61. Built in 1948 Elsie was a three-wheeled robot capable of phototaxis that could also find its way to a recharging station. A second generation tortoise (from 1951) is in the collection of the Smithsonian Institution. He published popular articles in “*Scientific American*” (1950 and 1951) and a book “*The Living Brain*” (1953). He was badly injured in a car accident in 1970 from which he never fully recovered. (Image courtesy Reuben Hoggett collection)





**Fig. 5.2.** The Simulink® model `sl_braitenberg` drives the vehicle toward the maxima of a provided scalar function. The vehicle plus controller is an example of a Braitenberg vehicle

We can make the measurements simultaneously using two spatially separated sensors or from one sensor over time as the robot moves.

To ascend the gradient we need to estimate the gradient direction at the current location and this requires at least two measurements of the field.<sup>►</sup> In this example we use two sensors, bilateral sensing, with one on each side of the robot's body. The sensors are modelled by the green sensor blocks shown in Fig. 5.2 and are parameterized by the position of the sensor with respect to the robot's body, and the sensing function. In this example the sensors are at  $\pm 2$  units in the vehicle's lateral or  $y$ -direction.

The field to be sensed is a simple inverse square field defined by

```
1 function sensor = sensorfield(x, y)
2     xc = 60; yc = 90;
3     sensor = 200 ./((x-xc).^2 + (y-yc).^2 + 200);
```

which returns the sensor value  $s(x, y) \in [0, 1]$  which is a function of the sensor's position in the plane. This particular function has a peak value at the point (60, 90).

The vehicle speed is

$$v = 2 - s_R - s_L$$

where  $s_R$  and  $s_L$  are the right and left sensor readings respectively. At the goal, where  $s_R = s_L = 1$  the velocity becomes zero.

Steering angle is based on the difference between the sensor readings

$$\gamma = k(s_R - s_L)$$

Similar strategies are used by moths whose two antennae are exquisitely sensitive odor detectors that are used to steer a male moth toward a pheromone emitting female.

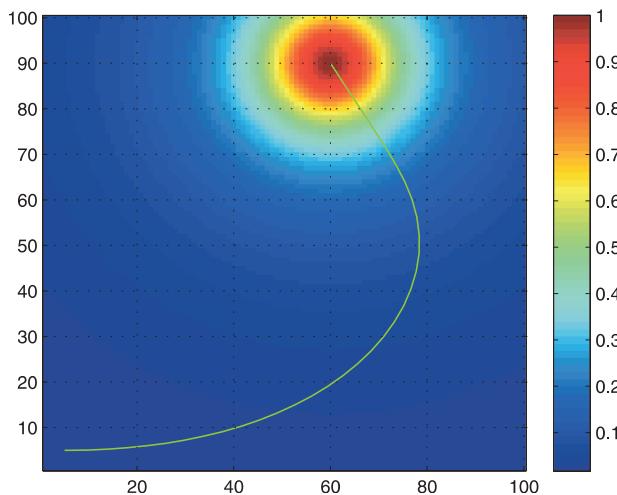
so when the field is equal in the left- and right-hand sensors the robot moves straight ahead.<sup>►</sup>

We start the simulation from the Simulink® menu or the command line

```
>> sim('sl_braitenberg');
```

and the path of the robot is shown in Fig. 5.3. The starting pose can be changed through the parameters of the `Bicycle` block. We see that the robot turns toward the goal and slows down as it approaches, asymptotically achieving the goal position.

This particular sensor-action control law results in a specific robotic behaviour. We could add additional logic to the robot to detect that it had arrived near the goal and then switch to a stopping behaviour. An obstacle would block this robot since its only behaviour is to steer toward the goal, but an additional behaviour could be added to handle this case and drive around an obstacle. We could add another behaviour to search randomly for the source if none was visible. Grey Walter's tortoise had four behaviours and switching was based on light level and a touch sensor.



**Fig. 5.3.**  
Path of the Braitenberg vehicle moving toward (and past) the maximum of a 2D scalar field whose magnitude is shown color coded

Multiple behaviours and the ability to switch between them leads to an approach known as behaviour-based robotics. The subsumption architecture was proposed as a means to formalize the interaction between different behaviours. Complex, some might say *intelligent looking*, behaviours can be manifested by such systems. However as more behaviours are added the complexity of the system grows rapidly and interactions between behaviours become more complex to express and debug. Ultimately the penalty of not using a map becomes too great.

### 5.1.2 Simple Automata

Another class of reactive robots are known as *bugs* – simple automata that perform goal seeking in the presence of non-driveable areas or obstacles. There are a large number of *bug* algorithms and they share the ability to sense when they are in proximity to an obstacle. In this respect they are similar to the Braitenberg class vehicle, but the *bug* includes a state machine and other logic in between the sensor and the motors. The automata have memory which our earlier Braitenberg vehicle lacked.► In this section we will investigate a specific *bug* algorithm known as *bug2*.

We start by loading an obstacle field to challenge the robot

```
>> load map1
```

which defines a  $100 \times 100$  matrix variable `map` in the workspace. The elements are zero or one representing free space or obstacle respectively and this is shown in Fig. 5.4. Tools to generate such maps are discussed on page 92. This matrix is an example of an occupancy grid which will be discussed further in the next section.

Braatenberg's book describes a series of increasingly complex vehicles, some of which incorporate memory. However the term *Braatenberg vehicle* has become associated with the simplest vehicles he described.

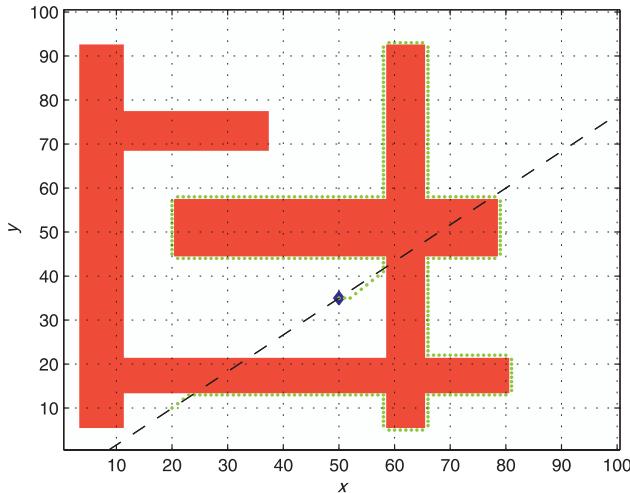
At this point we state some assumptions. Firstly, the robot operates in a grid world and occupies one grid cell. Secondly, the robot does not have any non-holonomic constraints and can move to any neighbouring grid cell. Thirdly, it is able to determine its position on the plane which is a non-trivial problem that will be discussed in detail in Chap. 6. Finally, the robot can only sense its immediate locale and the goal. The robot does not use the map – the map is used by the simulator to provide sensory inputs to the robot.

We create an instance of the `bug2` class

```
>> bug = Bug2(map);
```

and the goal is

```
>> bug.goal = [50; 35];
```

**Fig. 5.4.**

The path taken by the *bug2* algorithm is marked by green dots. The goal is a blue diamond, the black dashed line is the M-line, the direct path from the start to the goal. Obstacles are indicated by red pixels

The simulation is run using the `path` method

```
>> bug.path([20; 10]);
```

where the argument is the initial position of the robot. The method displays an animation of the robot moving toward the goal and the path is shown as a series of green dots in Fig. 5.4.

The strategy of the *bug2* algorithm is quite simple. It moves along a straight line towards its goal. If it encounters an obstacle it moves around the obstacle (always counter-clockwise) until it encounters a point that lies along its original line that is closer to the goal than where it first encountered the obstacle. ▶

If an output argument is specified

```
>> p = bug.path([20; 10]);
```

it returns the path as a matrix `p`

```
>> about(p)
p [double] : 332x2 (5312 bytes)
```

which has one row per point, and comprises 332 points for this example. Invoking the function without the starting position

```
>> p = bug.path();
```

will prompt for a starting point to be selected by clicking on the plot.

The *bug2* algorithms has taken a path that is clearly not optimal. It has wasted time by continuing to follow the perimeter of the obstacle until it rejoined its original line. It would also have been quicker in this case to go clockwise around the obstacle. Many variants of the *bug* algorithm have been developed, but while they improve the performance for one type of environment they can degrade performance in others. Fundamentally the robot is limited by not using a map. It cannot see the big picture and therefore takes paths that are locally, rather than globally, optimal.

## 5.2 Map-Based Planning

Beyond the trivial straight line case.

The key to achieving the *best* path ▶ between points A and B, as we know from everyday life, is to use a map. Typically best means the shortest distance but it may also include some penalty term or cost related to traversability which is how easy the terrain is to drive over – it might be quicker to travel further but over better roads. A more sophisticated planner might also consider the kinematics and dynamics of the

**Making a map.** An occupancy grid is a matrix that corresponds to a region of 2-dimensional space. Elements containing zeros are free space where the robot can move, and those with ones are obstacles where the robot cannot move. We can use many approaches to create a map. For example we could create a matrix filled with zeros (representing all free space)

```
>> map = zeros(100, 100);
```

and use MATLAB® operations such as

```
>> map(40:50,20:80) = 1;
```

to create an obstacle but this is quite cumbersome. Instead we can use the Toolbox map editor `makemap` to create more complex maps using a simple interactive editor

```
>> map = makemap(100)
makeworld:
    left button, click and drag to create a rectangle
    p - draw polygon
    c - draw circle
    e - erase map
    u - undo last action
    q - leave editing mode
```

which allows you to add rectangles, circles and polygons to an occupancy grid, in this example the grid is  $100 \times 100$ .

Note that the occupancy grid is a matrix whose coordinates are conventionally expressed as (row, column) and the row is the vertical dimension of a matrix. We use the Cartesian convention of a horizontal  $x$ -coordinate first, followed by the  $y$ -coordinate therefore the matrix is always indexed as `y, x` in the code.

vehicle and avoid paths that involve turns that are tighter than the vehicle can execute. Recalling our earlier definition of a robot as a

*goal oriented machine that can sense, plan and act,*

this section concentrates on planning.

There are many ways to represent a map and the position of the vehicle within the map. One approach is to represent the vehicle position as  $(x, y) \in \mathbb{R}^2$  and the driveable regions or obstacles as polygons, each comprising lists of vertices or edges. This is potentially a very compact format but determining potential collisions between the robot and obstacles may involve testing against long lists of edges.

A simpler and very computer-friendly representation is the occupancy grid. As its name implies the world is treated as a grid of cells and each cell is marked as occupied or unoccupied. We use zero to indicate an unoccupied cell or free space where the robot can drive. A value of one indicates an occupied or non-driveable cell. The size of the cell depends on the application. The memory required to hold the occupancy grid increases with the spatial area represented and inversely with the cell size. However for modern computers this representation is very feasible. For example a cell size  $1 \times 1$  m requires just 125 kbyte km<sup>-2</sup>.

In the remainder of this section we use code examples to illustrate several different planners and all are based on the occupancy grid representation. To create uniformity the planners are all implemented as classes derived from the `Navigation` superclass which is briefly described on page 93. The `bug2` class we used previously was also an instance of this class so the remaining examples follow a familiar pattern.

Once again we state some assumptions. Firstly, the robot operates in a grid world and occupies one grid cell. Secondly, the robot does not have any non-holonomic constraints and can move to any neighbouring grid cell. Thirdly, it is able to determine its position on the plane. Fourthly, the robot is able to use the map to compute the path it will take.

Considering a single bit to represent each cell. The occupancy grid could be compressed or could be kept on a disk with only the local region in memory.

**Navigation superclass.** The examples in this chapter are all based on classes derived from the `Navigation` class which is designed for 2D grid-based navigation. Each example consists of essentially the following pattern. Firstly we create an instance of an object derived from the `Navigation` class by calling the class constructor.

```
>> nav = MyNavController(world)
```

which is passed the occupancy grid. Then a plan to reach the goal is computed

```
>> nav.plan(goal)
```

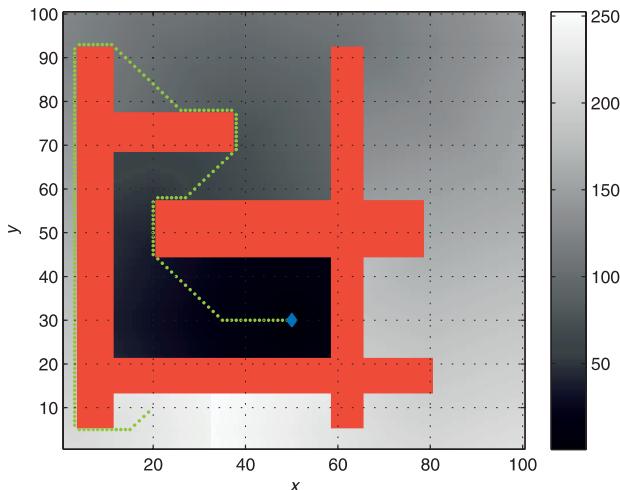
The plan can be visualized by

```
>> nav.visualize()
```

and a path from an initial position to the goal is computed by

```
>> p = nav.path(start)
>> p = nav.path()
```

where `p` is the path, a sequence of points from `start` to `goal`, one row per point, and each row comprises the `x`- and `y`-coordinate. If `start` is not specified, as in the second example, the user is prompted to interactively click the start point. If no output argument is provided an animation of the robot's motion is displayed.



**Fig. 5.5.**

The distance transform path. Obstacles are indicated by red cells. The background grey intensity represents the cell's distance from the goal in units of cell size as indicated by the scale on the right-hand side

In all examples we use the following parameters

```
>> goal = [50; 30];
>> start = [20; 10];
>> load map
```

for the goal position, start position and world map respectively. The world map is loaded into the workspace variable `map`. These parameters can be varied, and the occupancy grid changed using the tools described on page 92.

## 5.2.1 Distance Transform

The distance between two points  $(x_1, y_1)$  and  $(x_2, y_2)$  where  $\Delta_x = x_2 - x_1$  and  $\Delta_y = y_2 - y_1$  can be Euclidean  $\sqrt{\Delta_x^2 + \Delta_y^2}$  or CityBlock (also known as Manhattan) distance  $|\Delta_x| + |\Delta_y|$ .

Consider a matrix of zeros with just a single non-zero element representing the goal. The distance transform of this matrix is another matrix, of the same size, but the value of each element is its distance from the original non-zero pixel. For robot path planning we use the default Euclidean distance. The distance transform is actually an image processing technique and will be discussed further in Chap. 12.

To use the distance transform for robot navigation we create a `DXform` object, which is derived from the `Navigation` class

```
>> dx = DXform(map);
```

and create a plan to reach the specified goal

```
>> dx.plan(goal)
```

which can be visualized

```
>> dx.visualize()
```

as shown in Fig. 5.5. We see the obstacle regions in red and the background grey level at any point indicates the distance from that point to the goal, taking into account travel *around* the obstacle.

The hard work has been done and finding a path from *any* point to the goal is now very simple. Wherever the robot starts, it moves to the neighbouring cell that has the smallest distance to the goal. The process is repeated until the robot reaches a cell with a distance value of zero which is the goal. For example to find a path from to the goal from position `start` is

```
>> dx.path(start);
```

which displays an animation of the robot moving toward the goal. The path is indicated by a series of green dots as shown in Fig. 5.5.

If the `path` method is called with an output argument the animation is skipped and the path

```
>> p = dx.path(start);
```

is returned as a matrix, one row per point, which we can visualize

```
>> dx.visualize(p)
```

The path comprises

```
>> numrows(p)
ans =
205
```

points which is shorter than the path found by *bug2*. Unlike *bug2* this planner has found the shorter clockwise path around the obstacle.

This navigation algorithm has exploited its global view of the world and has, through exhaustive computation, found the shortest possible path. In contrast, *bug2* without the global view has just bumped its way through the world. The penalty for achieving the optimal path is computational cost. The distance transform is iterative. Each iteration has a cost of  $O(N^2)$  and the number of iterations is at least  $O(N)$ , where  $N$  is the dimension of the map.

We can visualize the iterations of the distance transform by

```
>> dx.plan(goal, show 0.1);
```

which shows the distance values propagating as a wavefront outward from the goal. The wavefront moves upward, splits to the left and right, moves downward and the two fronts collide at the bottom of the map along the line  $x = 32$ . The last argument specifies a pause of 0.1 s between frames. Although the plan is expensive to create, once it has been created it can be used to plan a path from *any* initial point to the goal.

We have converted a fairly complex planning problem into one that can now be handled by a Braitenberg-class robot that makes local decisions based on the distance to the goal. Effectively the robot is rolling *downhill* on the distance function which we can plot as a 3D surface

```
>> dx.visualize3d(p)
```

shown in Fig. 5.6 with the robot's path overlaid.

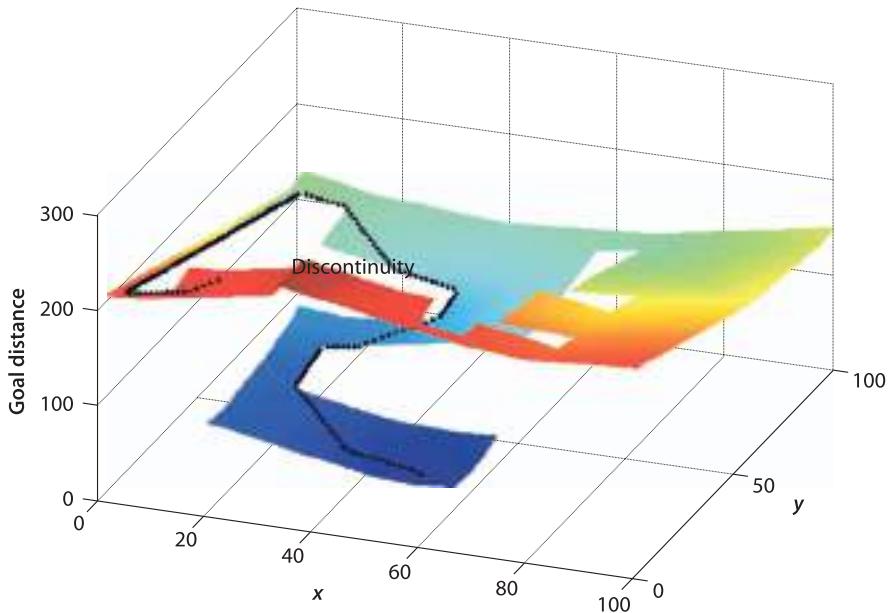


Fig. 5.6.

The distance transform as a 3D function, where height is distance from the goal. Navigation is simply a downhill run. Note the discontinuity in the distance transform where the split wavefronts met

For large occupancy grids this approach to planning will become impractical. The roadmap methods that we discuss later in this chapter provide an effective means to find paths in large maps at greatly reduced computational cost.

### 5.2.2 D\*

D\* is an extension of the A\* algorithm for finding minimum cost paths through a graph, see Appendix J.

A popular algorithm for robot path planning is called D\*, and it has a number of features that are useful for real-world applications. ▶ D\* generalizes the occupancy grid to a cost map which represents the cost  $c \in \mathbb{R}$ ,  $c > 0$  of traversing each cell in the horizontal or vertical direction. The cost of traversing the cell diagonally is  $c\sqrt{2}$ . For cells corresponding to obstacles  $c = \infty$  (`Inf` in MATLAB®).

D\* finds the path which minimizes the total cost of travel. If we are interested in the shortest time to reach the goal then cost is the time to drive across the cell and is inversely related to traversability. If we are interested in minimizing damage to the vehicle or maximizing passenger comfort then cost might be related to the roughness of the terrain within the cell. The costs assigned to cells will depend on the characteristics of the vehicle: a large 4-wheel drive vehicle may have a finite cost to cross a rough area whereas for a small car that cost might be infinite.

The key feature of D\* is that it supports incremental replanning. This is important if, while we are moving, we discover that the world is different to our map. If we discover that a route has a higher than expected cost or is completely blocked we can incrementally replan to find a better path. The incremental replanning has a lower computational cost than completely replanning as would be required using the distance transform method just discussed.

To implement the D\* planner using the Toolbox we use a similar pattern and first create a D\* navigation object

```
>> ds = Dstar(map);
```

The D\* planner converts the passed occupancy grid `map` into a cost map which we can retrieve

```
>> c = ds.costmap();
```

where the elements of `c` will be 1 or  $\infty$  representing free and occupied cells respectively.

A plan for moving to the goal is generated by

```
>> ds.plan(goal);
```

which creates a very dense directed graph (see Appendix J). Every cell is a graph vertex and has a cost, a distance to the goal, and a link to the neighbouring cell that is closest to the goal. Each cell also has a state  $t \in \{\text{NEW}, \text{OPEN}, \text{CLOSED}\}$ . Initially every cell is in the NEW state, the cost of the goal cell is zero and its state is OPEN. We can consider the set of all cells in the OPEN state as a wavefront propagating outward from the goal.<sup>►</sup> The cost of reaching cells that are neighbours of an OPEN cell is computed and these cells in turn are set to OPEN and the original cell is removed from the open list and becomes CLOSED. In MATLAB® this initial planning phase is quite slow<sup>►</sup> and takes tens of seconds and

```
>> ds.niter
ans =
    10558
```

iterations of the planning loop.

The path from an arbitrary starting point to the goal

```
>> ds.path(start);
```

is shown in Fig. 5.7. The robot has again taken the short path to the left of the obstacles and is almost the same as that generated by the distance transform.

The real power of  $D^*$  comes from being able to efficiently change the cost map during the mission. This is actually quite a common requirement in robotics since real sensors have a finite range and a robot discovers more of world as it proceeds. We inform  $D^*$  about changes using the `modify_cost` method, for example

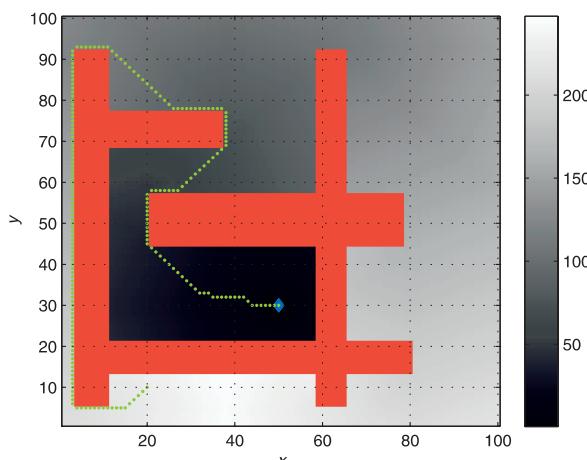
```
>> for y=78:85
>>     for x=12:45
>>         ds.modify_cost([x,y], 2);
>>     end
>> end
```

where we have raised the cost to 2 for a small rectangular region to simulate a patch of terrain with lower traversability. This region is indicated by the white dashed rectangle in Fig. 5.8. The other driveable cells have a cost of 1. The plan is updated by invoking the planning algorithm again

```
>> ds.plan();
```

but this time the number of iterations is only

```
>> ds.niter
ans =
    3178
```



The distance transform also evolves as a wavefront outward from the goal. However  $D^*$  represents the frontier efficiently as a list of cells whereas the distance transform computes the frontier on a per-pixel basis at every iteration – the frontier is implicitly where a cell with infinite cost (the initial value of all cells) is adjacent to a cell with finite cost.

$D^*$  is more efficient than the distance transform but it executes more slowly because it is implemented entirely in MATLAB® code whereas the distance transform is a MEX-file written in C.

**Fig. 5.7.**  
The  $D^*$  planner path. Obstacles are indicated by red cells and all driveable cells have a cost of 1. The background grey intensity represents the cell's distance from the goal in units of cell size as indicated by the scale on the right-hand side

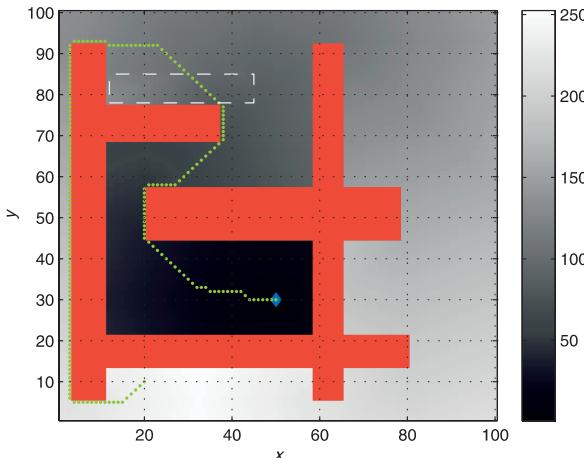
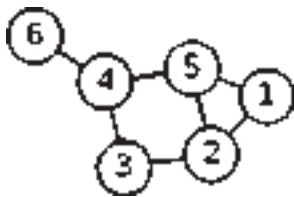


Fig. 5.8.

Path from  $D^*$  planner with modified map. The higher-cost region is indicated by the white dashed rectangle and has changed the path compared to Fig. 5.7



A graph is an abstract representation of a set of objects connected by links typically denoted  $G(V, E)$  and depicted diagrammatically as shown to the left. The objects,  $V$ , are called vertices or nodes, and the links,  $E$ , that connect some pairs of vertices are called edges or arcs. Edges can be directed (arrows) or undirected as in this case. Edges can have an associated weight or cost associated with moving from one of its vertices to the other. A sequence of edges from one vertex to another is a path. Graphs can be used to represent transport or communications networks and even social relationships, and the branch of mathematics is graph theory. Minimum cost path between two nodes in the graph can be computed using well known algorithms such as Dijkstra's method or  $A^*$ .

The navigation classes use a simple MATLAB® graph class called `PGraph`, see Appendix J.

which is 30% of that required to create the original plan. The new path for the robot

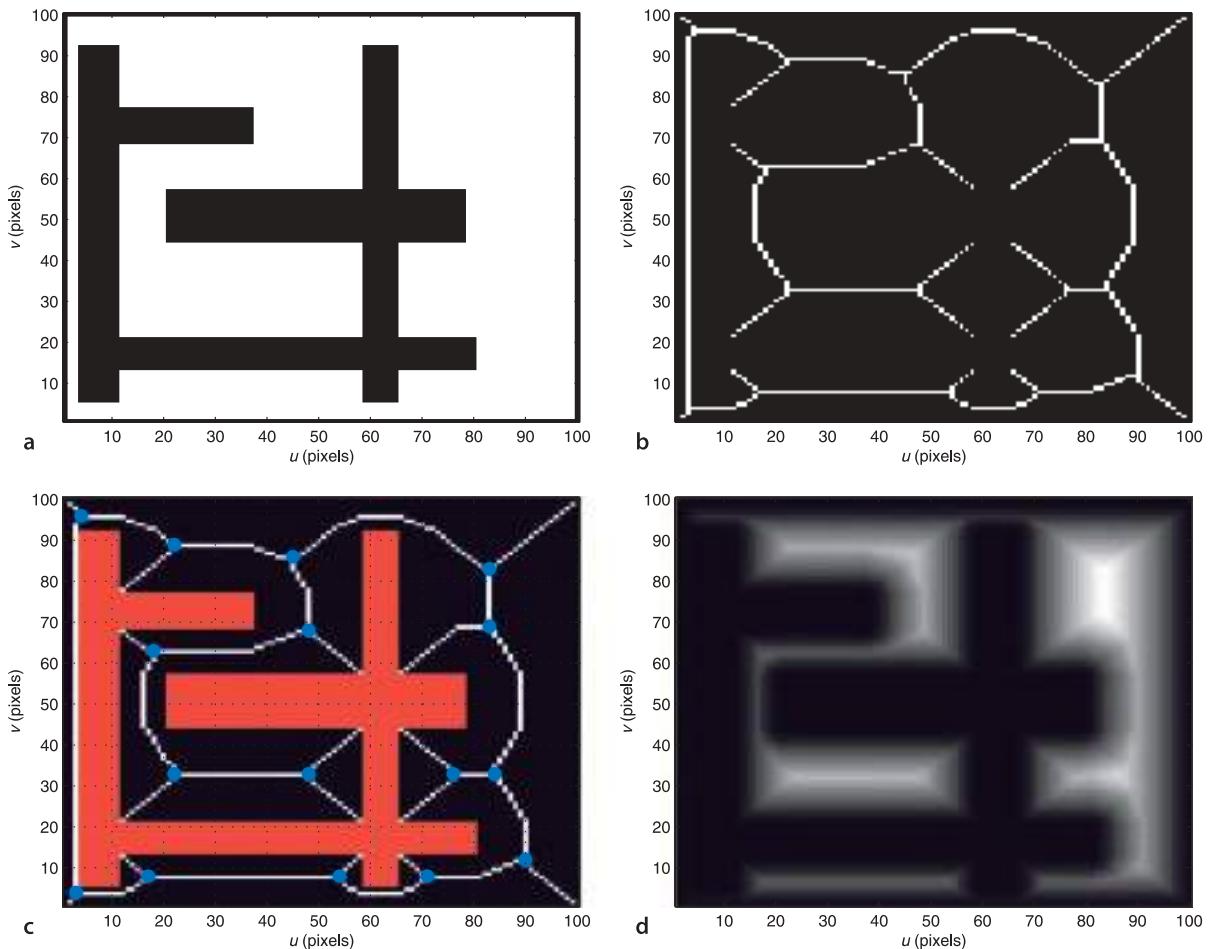
```
>> ds.path(start);
```

is shown in Fig. 5.8. The cost change is relatively small but we notice that the increased cost of driving within this region is indicated by a subtle brightening of those cells – in a cost sense these cells are now further from the goal. Compared to Fig. 5.7 the robot's path has moved to the right in order to minimize the distance it travels through the high-cost region.  $D^*$  allows updates to the map to be made at any time while the robot is moving. After replanning the robot simply moves to the adjacent cell with the lowest cost which ensures continuity of motion even if the plan has changed.

### 5.2.3 Voronoi Roadmap Method

In planning terminology the creation of a plan is referred to as the *planning phase*. The *query phase* uses the result of the planning phase to find a path from A to B. The two previous planning algorithms, distance transform and  $D^*$ , require a significant amount of computation for the planning phase, but the query phase is very cheap. However the plan depends on the goal. If the goal changes the expensive planning phase must be re-executed. Even though  $D^*$  allows the path to be recomputed as the costmap changes it does not support a changing goal.

The disparity in planning and query costs has led to the development of roadmap methods where the query can include both the start and goal positions. The planning phase provides analysis that supports changing starting points and changing goals. A good analogy is making a journey by train. We first find a local path to the nearest train station, travel through the train network, get off at the station closest to our goal, and then take a local path to the goal. The train network is invariant and planning a path through the train network is straightforward. Planning paths to and from the entry and exit stations respectively is also straightforward since they are, ideally, short



paths. The robot navigation problem then becomes one of building a network of obstacle free paths through the environment which serve the function of the train network. In the literature such a network is referred to as a roadmap. The roadmap need only be computed once and can then be used like the train network to get us from any start location to any goal location.

We will illustrate the principles by creating a roadmap from the occupancy grid's free space using some image processing techniques. The essential steps in creating the roadmap are shown in Fig. 5.9. The first step is to find the free space in the map which is simply the complement of the occupied space

```
>> free = 1 - map;
```

and is a matrix with non-zero elements where the robot is free to move. The boundary is also an obstacle so we mark the outermost cells as being not free

```
>> free(1,:) = 0; free(100,:) = 0;
>> free(:,1) = 0; free(:,100) = 0;
```

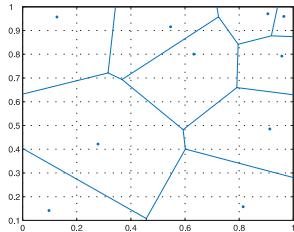
and this map is shown in Fig. 5.9a where free space is depicted as white.

The topological skeleton of the free space is computed by a morphological image processing algorithm known as thinning applied to the free space of Fig. 5.9a

```
>> skeleton = ithin(free);
```

and the result is shown in Fig. 5.9b. We see that the obstacles have grown and the free space, the white cells, have become a thin network of connected white cells which are

**Fig. 5.9.** Steps in the creation of a Voronoi roadmap. **a** Free space is indicated by white cells, **b** the skeleton of the free space is a network of adjacent cells no more than one cell thick, **c** the skeleton with the obstacles overlaid in red and roadmap junction points indicated in blue, **d** the distance transform of the obstacles, pixel values correspond to distance to the nearest obstacle



The Voronoi tessellation of a set of planar points, known as sites, is a set of Voronoi cells as shown to the left. Each cell corresponds to a site and consists of all points that are closer to its site than to any other site. The edges of the cells are the points that are equidistant to the two nearest sites. A generalized Voronoi diagram comprises cells defined by measuring distances to objects rather than points. In MATLAB® we can generate a Voronoi diagram by

```
>> sites = rand(10,2)
>> voronoi(sites(:,1), sites(:,2))
```

Georgy Voronoi (1868–1908) was a Russian mathematician, born in what is now Ukraine. He studied at Saint Petersburg University and was a student of Andrey Markov. One of his students Boris Delaunay defined the eponymous triangulation which has dual properties with the Voronoi diagram.

The junctions in the roadmap are indicated by blue circles. The junctions, or triple points, are identified using the morphological image processing function `triplepoint`.

equidistant from the boundaries of the original obstacles. Image processing functions, and morphological operations in particular, will be explained more fully in Chap. 12.

Figure 5.9c shows the free space network overlaid on the original map. We have created a network of paths that span the space and which can be used for obstacle-free travel around the map.◀ These paths are the edges of a generalized Voronoi diagram. We could obtain a similar result by computing the distance transform of the obstacles, Fig. 5.9a, and this is shown in Fig. 5.9d. The value of each pixel is the distance to the nearest obstacle and the ridge lines correspond to the skeleton of Fig. 5.9b. Thinning or skeletonization, like the distance transform, is a computationally expensive iterative algorithm but it illustrates well the principles of finding paths through free space. In the next section we will examine a cheaper alternative.

#### 5.2.4 Probabilistic Roadmap Method

The high computational cost of the distance transform and skeletonization methods makes them infeasible for large maps and has led to the development of probabilistic methods. These methods sparsely sample the world map and the most well known of these methods is the probabilistic roadmap or PRM method.

To use the Toolbox PRM planner for our problem we first create a `PRM` object

```
>> prm = PRM(map)
```

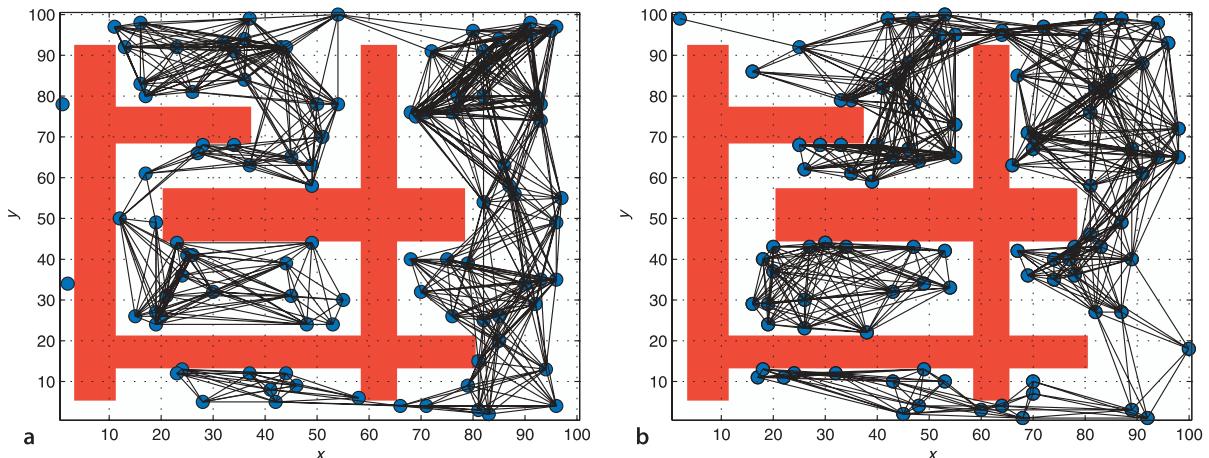
and then create the plan

```
>> prm.plan()◀
```

Note that in this case we do not pass `goal` as an argument to the planner since the plan is independent of the goal. Creating the path is a two phase process: planning, and query. The planning phase finds  $N$  random points, 100 by default, that lie in free space. Each point is connected to its nearest neighbours by a straight line path that does not cross any obstacles, so as to create a network, or graph, with a minimal number of disjoint components and no cycles. The advantage of PRM is that relatively few points need to be tested to ascertain that the points and the paths between them are obstacle free. The resulting network is stored within the `PRM` object and a summary can be displayed

```
>> prm
prm =
PRM: 100x100
graph size: 100
dist thresh: 30.000000
100 vertices
712 edges
3 components
```

To replicate the following result be sure to initialize the random number generator first using `randinit`. See p. 101.



which indicates the number of edges and connected components in the graph. The graph can be visualized

```
>> prm.visualize()
```

as shown in Fig. 5.10a. The dots represent the randomly selected points and the lines are obstacle-free paths between the points. Only paths less than 30 cells long are selected which is the distance threshold parameter of the `PRM` class. Each edge of the graph has an associated cost which is the distance between its two nodes. The color of the node indicates which component it belongs to and each component is assigned a unique color. We see two nodes on the left-hand side that are disconnected from the bulk of the roadmap.

The query phase is to find a path from the start point to the goal. This is simply a matter of moving to the closest node in the roadmap, following the roadmap, and getting off at the node closest to the goal and

```
>> prm.path(start, goal)
```

shows an animation of the robot moving through the graph and the path followed is shown in Fig. 5.11. Note that this time we provide the start and the goal position to the query phase. The next node on the roadmap is indicated in yellow and a line of green dots shows the robot's path. Travel along the roadmap involves moving toward the neighbouring node which has the lowest cost, that is, closest to the goal. We repeat the process until we arrive at the node in the graph closest to the goal, and from there we move directly to the goal.

An advantage of this planner is that once the roadmap is created by the planning phase we can change the goal and starting points very cheaply, only the query phase needs to be repeated.

However the path is not optimal and the distance travelled

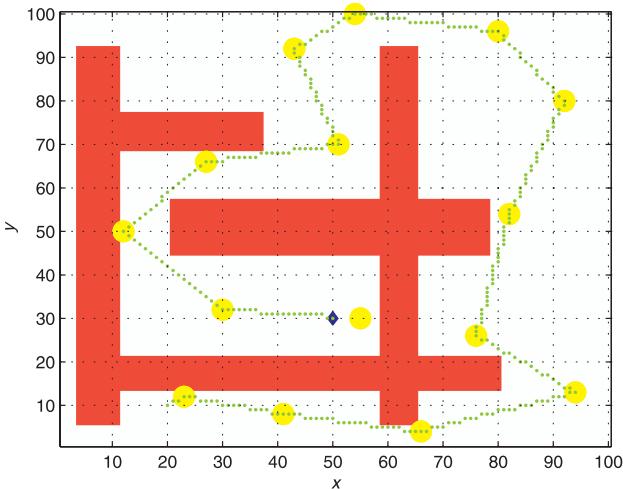
```
>> p = prm.path(start, goal);
>> numcols(p)
ans =
    299
```

is greater than the optimal value found by the distance transform but less than that found by *bug2*.

There are some important tradeoffs in achieving this computational efficiency. Firstly, the underlying random sampling of the free space means that a different graph is created every time the planner is run, resulting in different paths and path lengths. For example rerunning the planner

```
>> prm.plan();
```

**Fig. 5.10.** Probabilistic roadmap (PRM) planner and the random graphs produced in the planning phase. **a** Almost fully connected graph, apart from two nodes on the left-hand edge, **b** graph with a large disconnected component



**Fig. 5.11.**

Probabilistic roadmap (PRM) planner showing the path taken by the robot, shown as green dots. The nodes of the roadmap that are visited are highlighted in yellow

**Random numbers.** The MATLAB® random number generator (used for `rand` and `randn`) generates a very long sequence of numbers that are an excellent approximation to a random sequence. The generator maintains an internal state which is effectively the position within the sequence. After startup MATLAB® always generates the following random number sequence

```
>> rand
ans =
    0.8147
>> rand
ans =
    0.9058
>> rand
ans =
    0.1270
```

Many algorithms discussed in this book make use of random numbers and this means that the results can never be repeated. Before all such examples in this book is an invisible call to `randinit` which resets the random number generator to a known state

```
>> randinit
>> rand
ans =
    0.8147
>> rand
ans =
    0.9058
```

and we see that the random sequence has been restarted.

**A real robot is not a point.** We have assumed that the robot is a point, occupying a single cell in the occupancy grid. Some of the resulting paths which hug the sides of obstacles are impractical for a robot larger than a single cell. Rather than change the planning algorithms which are powerful and work very well we transform the obstacles. The Minkowski sum is used to *inflate* the obstacles to accomodate the worst-case pose of the robot in close proximity. The obstacles are replaced by virtual obstacles which are union of the obstacle and the robot in all its possible poses and just touching the boundary. The robot's various poses can be conservatively modelled as a circle that contains the robot.

If we consider the occupancy grid as an image then obstacle inflation can be achieved using the image processing operation known as dilation (discussed further in Chap. 12.). To inflate the obstacles with a circle of radius 3 cells the Toolbox command would be

```
>> map_new = imorph(map, kcircle(3), 'max');
```

produces the graph shown in Fig. 5.10b which has nodes at different locations and has a different number of edges.

Secondly, the planner can fail by creating a network consisting of disjoint components. The graph in Fig. 5.10a shows some disjoint components on the left-hand side, while the graph in Fig. 5.10b has a large disconnected component. If the start and goal positions are not connected by the roadmap, that is, they are close to different components the `path` method will report an error. The only solution is to rerun the planner.

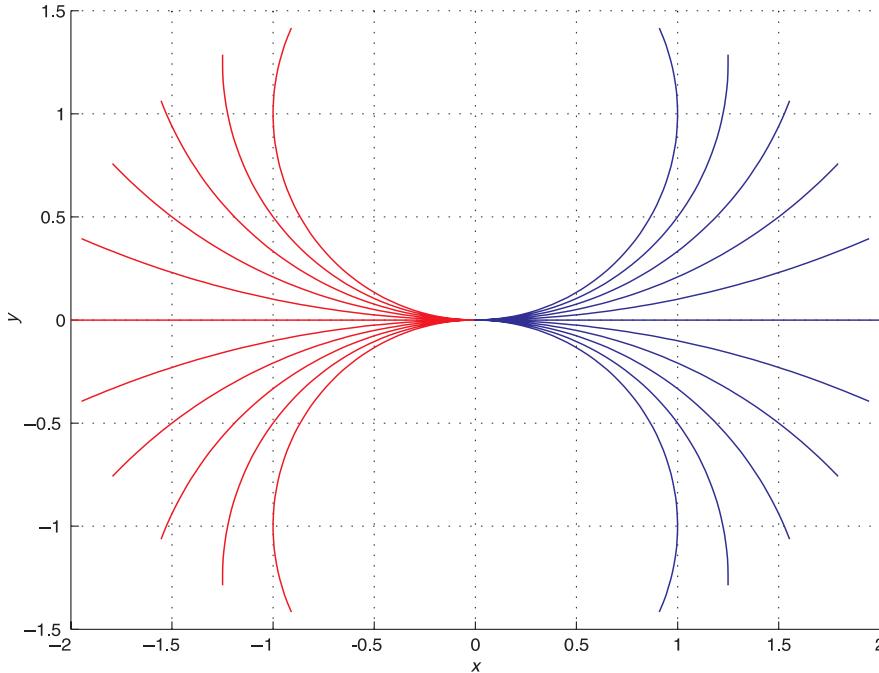
Thirdly, long narrow gaps between obstacles are unlikely to be exploited since the probability of randomly choosing points that lie along such gaps is very low. In this example the planner has taken the longer counter-clockwise path around the obstacle unlike the optimal distance transform planner which was able to exploit the narrow vertical path on the left-hand side.

### 5.2.5 RRT

The next, and final, planner that we introduce is able to take into account the motion model of the vehicle, relaxing the assumption that the robot is capable of omni-directional motion.

Figure 5.12 shows a family of paths that the bicycle model of Eq. 4.2 would follow for discrete values of velocity, forward and backward, and steering wheel angle over a fixed time interval. This demonstrates clearly the subset of all possible configurations that a non-holonomic vehicle can reach from a given initial configuration. In this discrete example, from the initial pose we have computed 22 poses that the vehicle could achieve. From each of these we could compute another 22 poses that the vehicle could reach after two periods, and so on. After just a few periods we would have a very large number of possible poses.

For any desired goal pose we could find the closest precomputed pose, and working backward toward the starting pose we could determine the sequence of steering angles and velocities needed to move from initial to the goal pose. This has some similarities to the roadmap methods discussed previously, but the limiting factor is the combinatoric explosion in the number of possible poses.



**Fig. 5.12.**  
A set of possible paths that the bicycle model robot could follow from an initial configuration of  $(0, 0, 0)$ . For  $v = \pm 1$ ,  $\alpha \in [-1, 1]$  over a 2 s period. Red lines correspond to  $v < 0$

The distance measure must account for a difference in position and orientation and requires appropriate weighting of these quantities. From a consideration of units this is not quite proper since we are adding metres and radians.

Uniformly randomly distributed between the steering angle limits.

The particular planner that we discuss is the Rapidly-exploring Random Tree or RRT. Like PRM it is a probabilistic algorithm and the main steps are as follows. A graph of robot configurations is maintained and each node is a configuration  $\xi \in SE(2)$  which is represented by a 3-vector  $\xi \sim (x, y, \theta)$ . The first node in the graph is some initial configuration of the robot. A random configuration  $\xi_{\text{rand}}$  is chosen, and the node with the closest configuration  $\xi_{\text{near}}$  is found – this point is near in terms of a cost function that includes distance and orientation. ▶ A control is computed that moves the robot from  $\xi_{\text{near}}$  toward  $\xi_{\text{rand}}$  over a fixed period of time. The point that it reaches is  $\xi_{\text{new}}$  and this is added to the graph.

We create an RRT roadmap for an obstacle free environment by following our familiar programming pattern. We create an `RRT` object

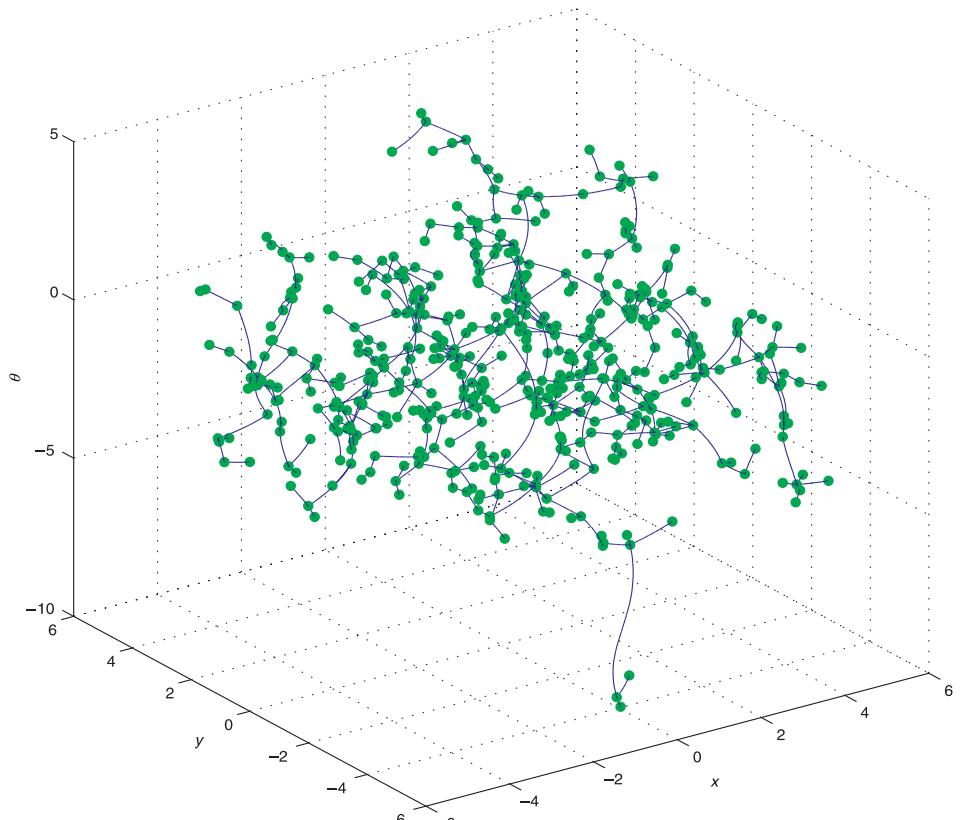
```
>> rrt = RRT()
```

which includes a default bicycle kinematic model, velocity and steering angle limits. We create a plan and visualize the results

```
>> rrt.plan();
>> rrt.visualize();
```

which are shown in Fig. 5.13. We see how the paths have a good coverage of the configuration space, not just in the  $x$ - and  $y$ -directions but also in orientation, which is why the algorithm is known as *rapidly exploring*.

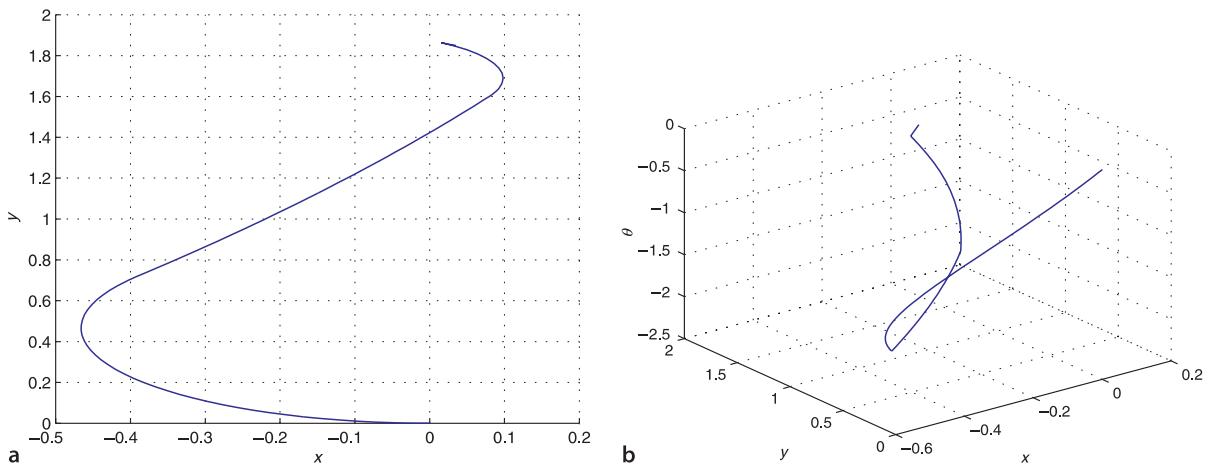
An important part of the RRT algorithm is computing the control input that moves the robot from an existing point in the graph to  $\xi_{\text{rand}}$ . From Sect. 4.2 we understand the difficulty of driving a non-holonomic vehicle to a specified pose. Rather than the complex non-linear controller of Sect. 4.2.4 we will use something simpler that fits with the randomized sampling strategy used in this class of planner. The controller randomly chooses whether to drive forwards or backwards and the steering angle, ▶



**Fig. 5.13.**

An RRT computed for the bicycle model with a velocity of  $\pm 1 \text{ m s}^{-1}$ , steering angle limits of  $\pm 1.2 \text{ rad}$ , integration period of 1 s, and initial configuration of  $(0, 0, 0)$ .

Each node is indicated by a green circle in the 3-dimensional space of vehicle poses  $(x, y, \theta)$



simulates motion of the bicycle model for a fixed period of time, and computes the closest distance to  $\xi_{\text{rand}}$ . This is repeated multiple times and the control input with the best performance is chosen. The point on its path that was closest to  $\xi_{\text{rand}}$  is chosen as  $\xi_{\text{near}}$  and becomes a new node in the graph.

Handling obstacles with the RRT is quite straightforward. The point  $\xi_{\text{rand}}$  is discarded if it lies within an obstacle, and the point  $\xi_{\text{near}}$  will not be added to the graph if the path from  $\xi_{\text{near}}$  toward  $\xi_{\text{rand}}$  intersects an obstacle. The result is a set of paths, a roadmap, that is collision free and driveable with this non-holonomic vehicle.

We will illustrate this with the challenging problem of moving a non-holonomic vehicle sideways. Specifically we want to find a path to move the robot 2 m in the lateral direction with its final heading angle the same as its initial heading angle

```
>> p = rrt.path([0 0 0], [0 2 0]);
```

The result is a continuous path

```
>> about(p)
p [double] : 3x126 (3024 bytes)
```

which we can plot

```
>> plot2(p')
>> iprint('rrt_path2')
```

and the result is shown in Fig. 5.14. This is a smooth path► that is feasible for the non-holonomic vehicle. The robot has an initial heading angle of 0 which means it is facing in the positive  $x$ -direction, so here it has driven backwards to the desired pose. Note also that the motion does not quite finish at the desired pose but at the node in the tree closest to the desired pose. This could be remedied by computing a denser tree, this one had 500 nodes, some adjustment of the steering command on the last segment of the motion, or using a local motion planner to move from the end of this path to the goal pose.

**Fig. 5.14.** The path computed by RRT that translates the non-holonomic vehicle sideways. **a** Path in the  $xy$ -plane; **b** path in  $(x, y, \theta)$  space

Although the steering angle is not continuous.

### 5.3 Wrapping Up

Robot navigation is the problem of guiding a robot towards a goal and we have covered a spectrum of approaches. The simplest was the purely reactive Braitenberg-type vehicle. Then we added limited memory to create state machine based automata such as *bug2* which can deal with obstacles, however the paths that it finds are far from optimal.

A number of different map-based planning algorithms were then introduced. The distance transform is a computationally intense approach that finds an optimal path to the goal.  $D^*$  also finds an optimal path, but accounts for traversability of individual

cells rather than considering them as either free space or obstacle.  $D^*$  also supports computationally cheap incremental replanning for small changes in the map. PRM reduces the computational burden by probabilistic sampling but at the expense of less optimal paths. In particular it may not discover narrow routes between areas of free space. Another sampling method is RRT which uses a kinematic model of the vehicle to create paths which are feasible to drive, and can readily account for the orientation of the vehicle as well as its position. All the map-based approaches require a map and knowledge of the robot's location, and these are both topics that we will cover in the next chapter.

---

### Further Reading

The defining book in cybernetics was written by Wiener in 1948 and updated in 1965 (Wiener 1965). Grey Walter published a number of popular articles (1950, 1951) and a book (1953) based on his theories and experiments with robotic tortoises.

The definitive reference for Braitenberg vehicles is Braitenberg's own book (1986) which is a whimsical, almost poetic, set of thought experiments. Vehicles of increasing complexity (fourteen vehicle families in all) are developed, some including nonlinearities, memory and logic to which he attributes anthropomorphic characteristics such as love, fear, aggression and egotism. The second part of the book outlines the factual basis of these machines in the neural structure of animals. The *bug1* and *bug2* algorithms were described by Lumelsky and Stepanov (1986). More recently eleven variations of Bug algorithm were implemented and compared for a number of different environments (Ng and Bräunl 2007).

Early behaviour-based robots included the Johns Hopkins Beast, built in the 1960s, and Genghis (Brooks 1989) built in 1989. Behaviour-based robotics are covered in the book by Arkin (1999) and the Robotics Handbook (Siciliano and Khatib 2008, § 38). Matarić's Robotics Primer (Matarić 2007) and associated comprehensive web-based resources is also an excellent introduction to reactive control, behaviour based control and robot navigation. A rich collection of archival material about early cybernetic machines, including Gray-Walter's tortoise and the Johns Hopkins Beast can be found at the Cybernetic Zoo <http://cyberneticzoo.com>.

The distance transform is well described by Borgefors (1986) and its early application to robotic navigation was explored by Jarvis and Byrne (1988). The  $D^*$  algorithm is an extension of the classic  $A^*$  algorithm for graphs (Nilsson 1971). It was proposed by Stentz (1994) and later extensions include Field  $D^*$  (Ferguson and Stentz 2006) and  $D^*$  lite (Koenig and Likhachev 2002).  $D^*$  was used by many vehicles in the DARPA challenges (Buehler et al. 2007, 2010).

The ideas behind PRM started to emerge in the mid 1990s and it was first described by Kavraki et al. (1996). Geraerts and Overmars (2004) compare the efficacy of a number of subsequent variations that have been proposed to the basic PRM algorithm. Approaches to planning that incorporate the vehicles dynamics include state-space sampling (Howard et al. 2008), and the RRT which is described in LaValle (1998, 2006) as well as <http://msl.cs.uiuc.edu/rrt>.

Two recent books provide almost encyclopedic coverage of planning for robots. The book on robot motion by Choset et al. (2005) covers geometric and probabilistic approaches to planning as well as the application to robots with dynamics and non-holonomic constraints. The book on robot planning by LaValle (2006) covers motion planning, planning under uncertainty, sensor-based planning, reinforcement learning, nonlinear systems, trajectory planning and nonholonomic planning. The powerful planning techniques discussed in these books can be applied beyond robotics to very high order systems such as complex mechanisms or even the shape of molecules. More succinct coverage is provided by Siegwart et al. (2011), the Robotics Handbook (Siciliano and Khatib 2008, § 35), and also in Spong et al. (2006) and Siciliano et al. (2008).

---

## Exercises

1. Braatenberg vehicles (page 88)
  - a) Experiment with different starting configurations and control gains.
  - b) Modify the signs on the steering signal to make the vehicle light-phobic.
  - c) Modify the `sensorfield` function so that the peak moves with time.
  - d) The vehicle approaches the maxima asymptotically. Add a stopping rule so that the vehicle stops when the when either sensor detects a value greater than 0.95.
  - e) Create a scalar field with two peaks. Can you create a starting pose where the robot gets confused?
2. Bug algorithms (page 90)
  - a) Using the function `makemap` create a new map to challenge *bug2*. Try different starting points. Is it possible to trap *bug2*?
  - b) Create an obstacle map that contains a maze. Can *bug2* solve the maze?
  - c) Implement other bug algorithms such as *bug1* and *tangent bug*. Do they perform better or worse?
3. At 1 m cell size how much memory is required to represent the surface of the Earth? How much memory is required to represent just the land area of Earth? What cell size is needed in order for a map of your country to fit in 1 Gbyte of memory?
4. Distance transform (page 93). A real robot has finite dimensions and a common technique for use with the point-robot planning methods is to grow the obstacles by half the radius of the robot. Use the Toolbox function `imorph` (see page 317) to dilate the obstacles by 4 grid cells.
5. For the D\* planner (page 95) increase the cost of the rough terrain and observe what happens. Add a region of very low-cost terrain (less than one) near the robot's path and observe what happens.
6. PRM planner (page 99)
  - a) Run the PRM planner 100 times and gather statistics on the resulting path length.
  - b) Vary the value of the distance threshold parameter and observe the effect.
  - c) Implement a non-grid based version of PRM. The robot is represented by an arbitrary polygon as are the obstacles. You will need functions to determine if a polygon intersects or is contained by another polygon (see the Toolbox `Polygon` class). Test the algorithm on the piano movers problem.
7. RRT planner (page 102)
  - a) Find a path to implement a 3-point turn.
  - b) Define an obstacle field and repeat the planning.
  - c) Experiment with RRT parameters such as the number of points, the vehicle steering angle limits, and the path integration time.
  - d) The current RRT chooses the steering angle as a uniform distribution between the steering angle limits. People tend to drive more gently, what happens if you choose a Gaussian distribution for the steering angle?
  - e) Additional information in the node of each graph holds the control input that was computed to reach the node. Plot the steering angle and velocity sequence required to move from start to goal pose.
  - f) Add a local planner to move from initial pose to the closest vertex, and from the final vertex to the goal pose.
  - g) Determine a path through the graph that minimizes the number of reversals of direction.
  - h) Add a more sophisticated collision detector where the vehicle is a finite sized rectangle and the world has polygonal obstacles. You will need functions to determine if a polygon intersects or is contained by another polygon (see the Toolbox `Polygon` class).

# 6

# Localization

*in order to get somewhere we need to know where we are*



In our discussion of map-based navigation we assumed that the robot had a means of knowing its position. In this chapter we discuss some of the common techniques used to estimate the location of a robot in the world – a process known as localization.

Today GPS makes outdoor localization so easy that we often take this capability for granted. Unfortunately GPS is a far from perfect sensor since it relies on very weak radio signals received from distant orbiting satellites. This means that GPS cannot work where there is no *line of sight* radio reception, for instance indoors, underwater, underground, in urban canyons or in deep mining pits. GPS signals are also extremely weak and can be easily jammed and this is not acceptable for some applications.

GPS has only been in use since 1995 yet human-kind has been navigating the planet and localizing for many thousands of years. In this chapter we will introduce the *classical* navigation principles such as dead reckoning and the use of landmarks on which modern robotic navigation is founded.

Dead reckoning is the estimation of location based on estimated speed, direction and time of travel with respect to a previous estimate. Figure 6.1 shows how a ship's position is updated on a chart. Given the average compass heading over the previous hour and a distance travelled the position at 3 P.M. can be found using elementary geometry from the position at 2 P.M. However the measurements on which the update is based are subject to both systematic and

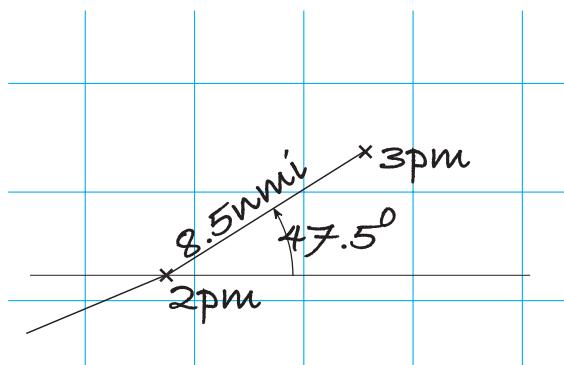


Fig. 6.1.

Location estimation by dead reckoning. The ship's position at 3 P.M. is based on its position at 2 P.M., the estimated distance travelled since, and the average compass heading

**Measuring speed at sea.** A ship's log is an instrument that provides an estimate of the distance travelled. The oldest method of determining the speed of a ship at sea was the Dutchman's log – a floating object was thrown into the water at the ship's bow and the time for it to pass the stern was measured using an hourglass. Later came the chip log, a flat quarter-circle of wood with a lead weight on the circular side causing it to float upright and resist towing. It was tossed overboard and a line with knots at 50 foot intervals was payed out. A special hourglass, called a log glass, ran for 30 s, and each knot on the line over that interval corresponds to approximately  $1 \text{ nmi h}^{-1}$  or 1 knot. A nautical mile (nmi) is now defined as 1.852 km.

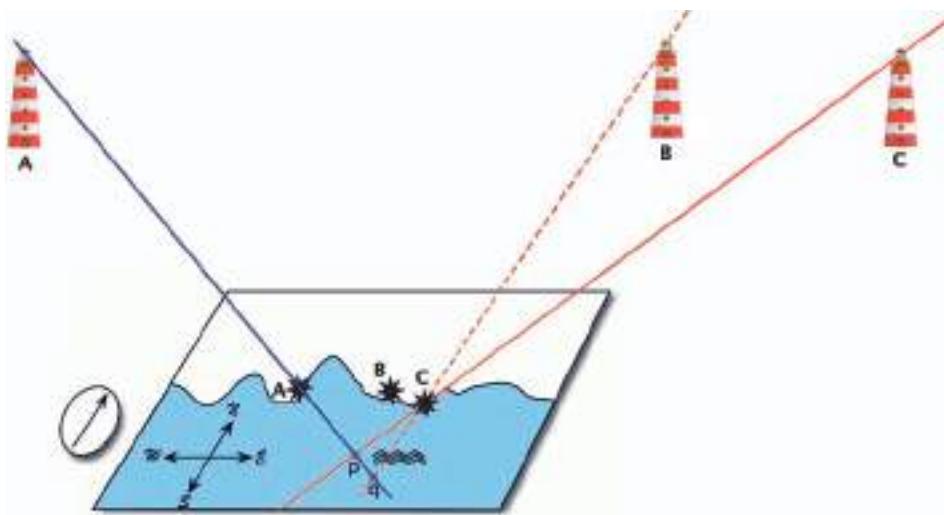


Fig. 6.2.

Location estimation using a map. Lines of sight from two lighthouses, A and C, and their corresponding locations on the map provide an estimate  $p$  of our location. However if we mistake lighthouse C for B then we obtain an incorrect estimate  $q$

random error. Modern instruments are quite precise but 500 years ago clocks, compasses and speed measurement were primitive. The recursive nature of the process, each estimate is based on the previous one, means that errors will accumulate over time and for sea voyages of many-months this approach was quite inadequate.

The Phoenicians were navigating at sea more than 4 000 years ago and they did not even have a compass – that was developed 2 000 years later in China. The Phoenicians navigated with crude dead reckoning but wherever possible they used *additional information* to correct their position estimate – sightings of islands and headlands, primitive maps and observations of the Sun and the Pole Star.

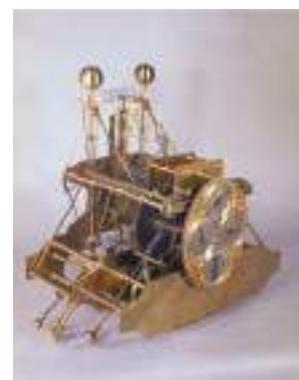
A landmark is a visible feature in the environment whose location is known with respect to some coordinate frame. Figure 6.2 shows schematically a map and a number of lighthouse landmarks. We first of all use a compass to align the north axis of our map with the direction of the north pole. The direction of a single landmark constrains our position to lie along a line on the map. Sighting a second landmark places our position on another constraint line, and our position must be at their intersection – a process known as resectioning.<sup>►</sup> For example lighthouse A constrains us to lie along the blue line. Lighthouse B constrains us to lie along the red line and the intersection is our true position  $p$ .

**Resectioning** is the estimation of position by measuring the bearing angles to known landmarks. Triangulation is the estimation of position by measuring the bearing angles to the unknown point from each of the landmarks.

**Celestial navigation.** The position of celestial bodies in the sky is a predictable function of the time and the observer's latitude and longitude. This information can be tabulated and is known as ephemeris (meaning daily) and such data has been published annually in Britain since 1767 as the “*The Nautical Almanac*” by HM Nautical Almanac Office. The elevation of a celestial body with respect to the horizon can be measured using a sextant, a handheld optical instrument.

Time and longitude are coupled, the star field one hour later is the same as the star field  $15^\circ$  to the east. However the northern Pole Star, *Polaris* or the *North Star*, is very close to the celestial pole and its elevation angle is independent of longitude and time, allowing latitude to be determined very conveniently from a single sextant measurement.

Solving the longitude problem was the greatest scientific challenge to European governments in the eighteenth century since it was a significant impediment to global navigation and maritime supremacy – the British Longitude Act of 1714 created a prize of £20 000. This spurred the development of nautical chronometers, clocks that could maintain high accuracy onboard ships. More than fifty years later a suitable chronometer was developed by John Harrison and was used by Captain James Cook on his second voyage of 1772–1775. After a three year journey the error in estimated longitude was just 13 km. With accurate knowledge of time, the elevation angle of stars could be used to estimate latitude and longitude. This technological advance enabled global exploration and trade.



Harrison's H1 chronometer (1735), © National Maritime Museum, Greenwich, London

**Radio-based localization.** One of the earliest systems was LORAN, based on the British World War II GEE system. LORAN transmitters around the world emit synchronized radio pulses and a receiver measures the difference in arrival time between pulses from a pair of radio transmitters. Knowing the identity of two transmitters and the time difference (TD) constrains the receiver to lie along a hyperbolic curve shown on navigation charts as *TD lines*. Using a second pair of transmitters (which may include one of the first pair) gives another hyperbolic constraint curve, and the receiver must lie at the intersection of the two curves.

The Global Positioning System (GPS) was proposed in 1973 but did not become fully operational until 1995. It currently comprises around 30 active satellites orbiting the earth in six planes at a distance of 20 200 km. A GPS receiver works by measuring the time of travel of radio signals from four or more satellites whose orbital position is encoded in the GPS signal. With four known points in space and four measured time delays it is possible to compute the  $(x, y, z)$  position of the receiver and the time. If the GPS signals are received after reflecting off some surface the distance trav-

elled is longer and this will introduce an error in the position estimate. This effect is known as multi-pathing and is common in large-scale industrial facilities.

Variations in the propagation speed of radio waves through the atmosphere is the main cause of error in the position estimate. However these errors vary slowly with time and are approximately constant over large areas. This allows the error to be measured at a reference station and transmitted to compatible nearby receivers which can offset the error – this is known as Differential GPS (DGPS). Many countries have coastal radio networks that broadcast this correction information, and for aircraft it is broadcast by another satellite network called the Wide Area Augmentation System (WAAS). RTK GPS achieves much higher precision in time measurement by using phase information from the carrier signal. The original GPS system deliberately added error, euphemistically termed selective availability, to reduce its utility to military opponents but this *feature* was disabled in May 2000. Other satellite navigation systems include the Russian GLONASS and the European Galileo.

It is just as bad is to see C but think it is B on the map.

However this process is critically reliant on correctly associating the observed landmark with the feature on the map. If we mistake one lighthouse for another, for example we see B but think it is C on the map, then the red dashed line leads to a significant error in estimated position – we would believe we were at q instead of p. This belief would lead us to overestimate our distance from the coastline. If we decided to sail toward the coast we would run aground on rocks and be surprised since they were not where we expected them to be. This is unfortunately a very common error and countless ships have foundered because of this fundamental data association error. This is why lighthouses flash! In the eighteenth century technological advances enabled lighthouses to emit unique flashing patterns so that the identity of the particular lighthouse could be reliably determined and associated with a point on a navigation chart.

Of course for the earliest mariners there were no maps, or lighthouses or even compasses. They had to create maps as they navigated by incrementally adding new non-manmade features to their maps just beyond the boundaries of what was already known. It is perhaps not surprising that so many early explorers came to grief and that maps were tightly kept state secrets.

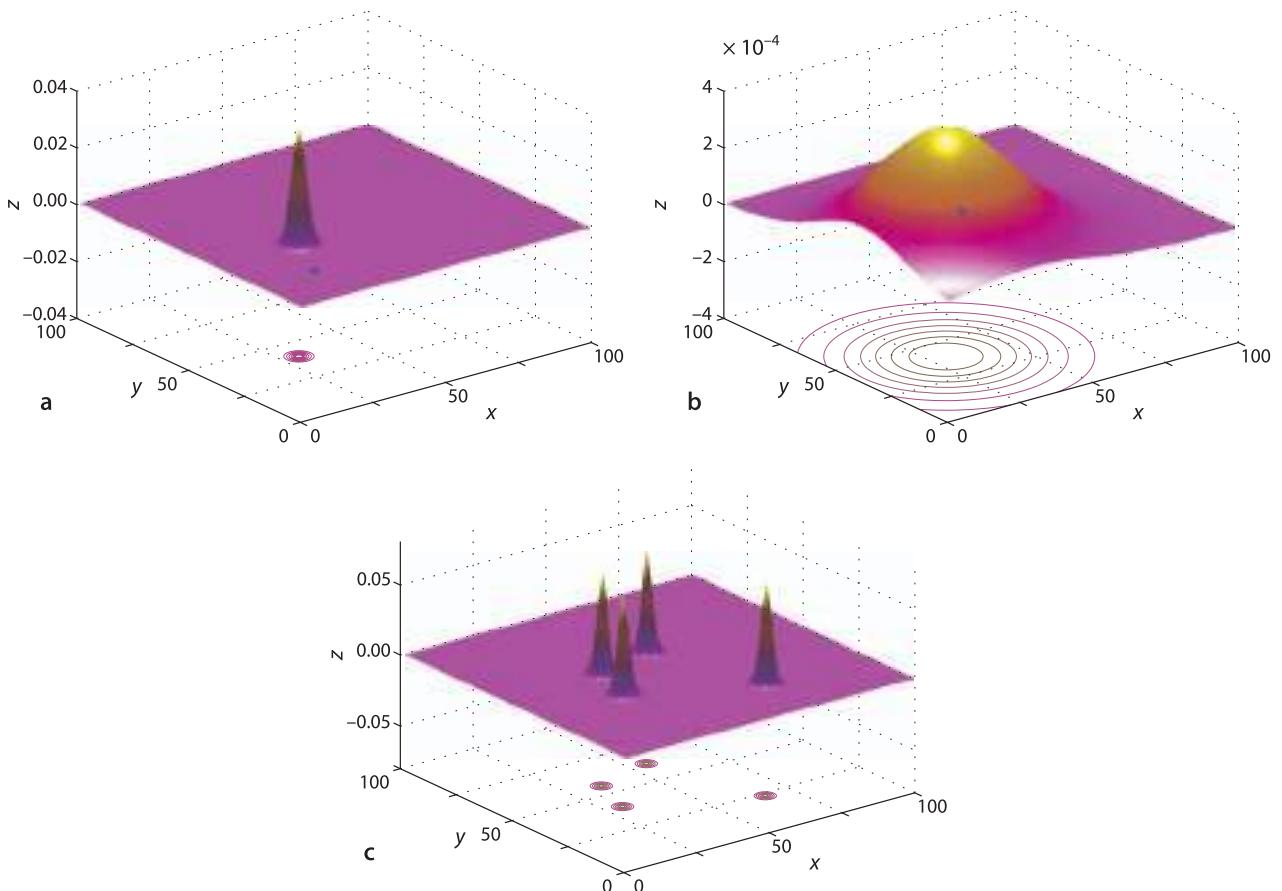
Robots operating today in environments without GPS face *exactly* the same problems as ancient navigators and, perhaps surprisingly, borrow heavily from navigational strategies that are centuries old. A robot's estimate of distance travelled will be imperfect and it may have no map, or perhaps an imperfect or incomplete map. Additional information from observation of features is critical to minimizing a robot's localization error but the possibility of data association error remains.

For our navigation problem we define  $x$  as the true, but unknown, position of the robot and  $\hat{x}$  as our best estimate of that position. We also wish to know the *uncertainty* of the estimate which we can consider in statistical terms as the standard deviation associated with the position estimate  $\hat{x}$ .

It is useful to describe the robot's position in terms of a probability density function (PDF) over all possible positions of the robot. Some example PDFs are shown in Fig. 6.3 where the magnitude of the function is the relative likelihood of the vehicle being at that position. Commonly a Gaussian function is used which can be described succinctly in terms of its mean and standard deviation. The robot is most likely to be at the location of the peak (the mean) and increasingly less likely to be at positions further away from the peak. Figure 6.3a shows a peak with a small standard deviation which indicates that the vehicle's position is very well known. There is an almost zero

Magellan's 1519 expedition started with 237 men and 5 ships but most, including Magellan, were lost along the way. Only 18 men and 1 ship returned.

A wheeled robot can estimate distance travelled by measuring wheel rotation, but an aerial or underwater robot cannot do this. Wheel rotation is imperfect due to variation and uncertainty in wheel radius, slippage and the effects of turning. Computer vision can be used to create a visual odometry system based on observations of the world moving past the robot.



probability that the vehicle is at the point indicated by the  $*$ -marker. In contrast the peak in Fig. 6.3b has a large standard deviation which means that we are less certain about the location of the vehicle. There is a reasonable probability that the vehicle is at the point indicated by the  $*$ -marker. Using a PDF also allows for multiple hypotheses about the robot's position. For example the PDF of Fig. 6.3c describes a robot that is quite certain that it is at one of four places. This is more useful than it seems at face value. Consider an indoor robot that has observed a vending machine and there are four such machines marked on the map. In the absence of any other information the robot must be equally likely to be in the vicinity of *any* of the four vending machines. We will revisit this approach in Sect. 6.5.

Determining the PDF based on knowledge of how the vehicle moves and its observations of the world is a problem in estimation which we can usefully define as:

*the process of inferring the value of some quantity of interest,  $x$ , by processing data that is in some way dependent on  $x$ .*

For example a ship's navigator or a surveyor estimates location by measuring the bearing angles to known landmarks or celestial objects, and a GPS receiver estimates latitude and longitude by observing the time delay from moving satellites whose location is known.

For our robot localization problem the true and estimated state are vector quantities so uncertainty will be represented as a covariance matrix, see Appendix F. The diagonal elements represent uncertainty of corresponding states, and the off-diagonal elements represent correlations between states.

**Fig. 6.3.** Notions of vehicle position and uncertainty in the  $xy$ -plane, where the vertical axis is the relative likelihood of the vehicle being at that position. Contour lines are displayed on the lower plane. **a** The vehicle has low position uncertainty,  $\sigma = 1$ ; **b** the vehicle has much higher position uncertainty,  $\sigma = 20$ ; **c** the vehicle has multiple hypotheses for its position, each  $\sigma = 1$

## 6.1 Dead Reckoning

Dead reckoning is the estimation of a robot's location based on its estimated speed, direction and time of travel with respect to a previous estimate.

### 6.1.1 Modeling the Vehicle

The first step in estimating the robot's position is to write a function,  $f(\cdot)$ , that describes how the vehicle's configuration changes from one time step to the next. A vehicle model such as Eq. 4.2 describes the evolution of the robot's configuration as a function of its control inputs, however for real robots we rarely have access to these control inputs. Most robotic platforms have proprietary motion control systems that accept motion commands from the user (speed and direction) and report odometry information.

An odometer is a sensor that measures distance travelled, typically by measuring the angular rotation of the wheels. The direction of travel can be measured using an electronic compass, or the change in heading can be measured using a gyroscope or differential odometry.<sup>◀</sup> These sensors are imperfect due to systematic errors such as an incorrect wheel radius or gyroscope bias, and random errors such as slip between wheels and the ground, or the effect of turning.<sup>◀</sup> We consider odometry to comprise both distance and heading information.

Instead of using Eq. 4.2 directly we will write a discrete-time model for the evolution of configuration based on odometry where  $\delta(k) = (\delta_d, \delta_\theta)$  is the distance travelled and change in heading over the preceding interval, and  $k$  is the time step. The initial pose is represented in SE(2) as

$$\xi(k) \sim \begin{pmatrix} \cos\theta(k) & -\sin\theta(k) & x(k) \\ \sin\theta(k) & \cos\theta(k) & y(k) \\ 0 & 0 & 1 \end{pmatrix}$$

We assume that motion over the time interval is *small* so the order of applying the displacements is not significant. We choose to move forward in the vehicle  $x$ -direction by  $\delta_d$  and then rotate by  $\delta_\theta$  giving the new configuration

$$\begin{aligned} \xi(k+1) &\sim \begin{pmatrix} \cos\theta(k) & -\sin\theta(k) & x(k) \\ \sin\theta(k) & \cos\theta(k) & y(k) \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & \delta_d \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\delta_\theta & -\sin\delta_\theta & 0 \\ \sin\delta_\theta & \cos\delta_\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ &\sim \begin{pmatrix} \cos(\theta(k) + \delta_\theta) & -\sin(\theta(k) + \delta_\theta) & x(k) + \delta_d \cos\theta(k) \\ \sin(\theta(k) + \delta_\theta) & \cos(\theta(k) + \delta_\theta) & y(k) + \delta_d \sin\theta(k) \\ 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

or as a 3-vector

$$\xi(k+1) \sim \begin{pmatrix} x(k) + \delta_d \cos(\theta(k) + \delta_\theta) \\ y(k) + \delta_d \sin(\theta(k) + \delta_\theta) \\ \theta(k) + \delta_\theta \end{pmatrix} \quad (6.1)$$

which gives the new configuration in terms of the previous configuration and the odometry.

However this assumes that odometry is perfect, which is not realistic. To model the error in odometry we add continuous random variables  $v_d$  and  $v_\theta$  to  $\delta_d$  and  $\delta_\theta$  respectively. The robot's configuration at the next time step, including the odometry error, is

**Measuring the difference in angular velocity of a left- and right-hand side wheel.**

**When turning, the outside wheel travels faster than the inside wheel, and this can be accounted for by measuring the speed of both wheels.**

$$\xi_{(k+1)} = \begin{pmatrix} x(k) + (\delta_d(k) + v_d) \cos(\theta(k) + \delta_\theta + v_\theta) \\ y(k) + (\delta_d(k) + v_d) \sin(\theta(k) + \delta_\theta + v_\theta) \\ \theta(k) + \delta_\theta + v_\theta \end{pmatrix} \quad (6.2)$$

which is the required function  $f(\cdot)$

$$\mathbf{x}(k+1) = f(\mathbf{x}(k), \delta(k), \mathbf{v}(k)) \quad (6.3)$$

where  $k$  is the time step,  $\delta(k)$  is the odometry measurement and  $\mathbf{v}(k)$  the random measurement noise over the preceding interval.►

In the absence of any information to the contrary we model the odometry noise as  $\mathbf{v} = (v_d, v_\theta) \sim N(0, V)$ , a zero-mean Gaussian processes with variance

$$V = \begin{pmatrix} \sigma_d^2 & 0 \\ 0 & \sigma_\theta^2 \end{pmatrix}$$

This matrix, the covariance matrix, is diagonal which means that the errors in distance and heading are *independent*.► Choosing a value for  $V$  is not always easy but we can conduct experiments or make some reasonable engineering assumptions. In the examples which follow we choose  $\sigma_d = 2$  cm and  $\sigma_\theta = 0.5^\circ$  per sample interval which leads to a covariance matrix of

```
>> V = diag([0.005, 0.5*pi/180].^2);
```

The Toolbox `Vehicle` class simulates the bicycle model of Eq. 4.2 and the odometric configuration update Eq. 6.2. To use it we create a `Vehicle` object

```
>> veh = Vehicle(V)
Vehicle object
L=1, maxspeed=5, alphalim=0.5, T=0.100000, V=(0.0004,0.00121847), nhist=0
x=0, y=0, theta=0
```

which shows the default parameters such as the vehicle's length, speed, steering limit and the sample interval which defaults to 0.1 s. The object provides a method to simulate one time step

```
>> odo = veh.step(1, 0.3)
odo =
0.1002    0.0322
>> odo = veh.step(1, 0.3)
odo =
0.0991    0.0311
```

where we have specified a speed of  $1 \text{ m s}^{-1}$  and a steering angle of  $0.3 \text{ rad}$ . The function updates the robot's true configuration and returns a noise corrupted odometer reading. With a sample interval of 0.1 s the robot reports that is moving approximately 0.1 m each interval and changing its heading by approximately  $0.03 \text{ rad}$ . The robot's true (but hidden) configuration can be seen by displaying the object

```
>> veh
veh =
Vehicle object
L=1, maxspeed=5, alphalim=0.5, T=0.100000, V=(0.0004,0.00121847), nhist=2
x=0.199955, y=0.00299955, theta=0.06
```

We want to run the simulation over a long time period but we also want to keep the vehicle within a defined spatial region. The `RandomPath` class is a *driver* that steers the robot to randomly selected waypoints within a specified region. We create an instance of the driver object and connect it to the robot

In this case the odometry noise is *inside* the process model and is referred to as process noise.

In reality this is unlikely to be the case since odometry distance errors tend to be worse when change of heading is high.

```
>> veh.add_driver( RandomPath(10) )
```

where the argument to the `RandomPath` constructor specifies a working region that spans  $\pm 10$  m in the  $x$ - and  $y$ -directions. We can display an animation of the robot with its driver by

```
>> veh.run()
```

The number of history records is indicated by `nhist=` in the displayed value of the object.

This repeatedly calls `step` and maintains a history of the true state of the vehicle over the course of the simulation within the `Vehicle` object.◀ The `RandomPath` and `Vehicle` classes have many parameters and methods which are described in the online documentation.

### 6.1.2 Estimating Pose

The problem we face, just like the ship's navigator, is how to best estimate our new pose given the previous pose and noisy odometry. The mathematical tool that we will use is the Kalman filter which is described more completely in Appendix H. The filter provides the optimal estimate for a system in which the noise is zero-mean and Gaussian. In this application the state of the Kalman filter is the estimated configuration of the robot. The filter is a recursive algorithm that updates, at each time step, the optimal estimate of the unknown true configuration and the uncertainty associated with that estimate. That is, it provides the best estimate of where we are and how certain we are about that.

A truncated Taylor series.

The Kalman filter is formulated for linear systems but our model of the vehicle's motion Eq. 6.3 is non-linear. We create a local linear approximation or linearization◀ of the function  $\hat{x}\langle k \rangle$  by

$$\hat{x}\langle k+1 \rangle = \hat{x}\langle k \rangle + F_x(\mathbf{x}\langle k \rangle - \hat{x}\langle k \rangle) + F_v v\langle k \rangle$$

with respect to the current state estimate  $\hat{x}\langle k \rangle$ . The terms  $F_x$  and  $F_v$  are Jacobians which are vector versions of derivatives which are sometimes written as  $\partial f / \partial x$  and  $\partial f / \partial v$  respectively. This approach to estimation of a non-linear system is known as the extended Kalman filter or EKF. Jacobians are reviewed in Appendix G.

The Jacobians are obtained by differentiating Eq. 6.2 and evaluating them for  $v = 0$  giving

$$F_x = \frac{\partial f}{\partial x} \Big|_{v=0} = \begin{pmatrix} 1 & 0 & -\delta_d\langle k \rangle - \sin(\theta\langle k \rangle + \delta_\theta) \\ 0 & 1 & \delta_d\langle k \rangle \cos(\theta\langle k \rangle + \delta_\theta) \\ 0 & 0 & 1 \end{pmatrix} \quad (6.4)$$



**Rudolf Kálmán** (1930–) is a mathematical system theorist born in Budapest. He obtained his bachelors and masters degrees in electrical engineering from MIT, and PhD in 1957 from Columbia University. He worked as a Research Mathematician at the Research Institute for Advanced Study, in Baltimore, from 1958–1964 where he developed his ideas on estimation. These were met with some skepticism amongst his peers and he chose a mechanical (rather than electrical) engineering journal for his paper *A new approach to linear filtering and prediction problems* because “When you fear stepping on hallowed ground with entrenched interests, it is best to go sideways”. He has received many awards including the IEEE Medal of Honor, the Kyoto Prize and the Charles Stark Draper Prize.

Stanley F. Schmidt is a research scientist who worked at NASA Ames Research Center and was an early advocate of the Kalman filter. He developed the first implementation as well as the non-linear version now known as the extended Kalman filter. This lead to its incorporation in the Apollo navigation computer for trajectory estimation. (Extract from Kálmán’s famous paper (1960) on the left reprinted with permission of ASME)

$$F_v = \left. \frac{\partial f}{\partial v} \right|_{v=0} = \begin{pmatrix} \cos(\theta(k) + \delta_\theta) & -\delta_d(k) \sin(\theta(k) + \delta_\theta) \\ \sin(\theta(k) + \delta_\theta) & \delta_d(k) \cos(\theta(k) + \delta_\theta) \\ 0 & 1 \end{pmatrix} \quad (6.5)$$

The `Vehicle` object provides methods `Fx` and `Fv` to compute these Jacobians, for example

```
>> veh.Fx( [0,0,0], [0.5, 0.1] )
ans =
1.0000      0    -0.0499
0    1.0000    0.4975
0      0    1.0000
```

where the first argument is the state about which the Jacobian is computed and the second is the odometry.

Now we can write the EKF prediction equations▶

$$\hat{x}(k+1|k) = f(\hat{x}(k), \delta(k), 0) \quad (6.6)$$

$$\hat{P}(k+1|k) = F_x(k) \hat{P}(k|k) F_x(k)^T + F_v(k) \hat{V} F_v(k)^T \quad (6.7)$$

that describe how the state and covariance evolve with time. The term  $\hat{x}(k+1|k)$  is read as the estimate of  $x = (\hat{x}, \hat{y}, \hat{\theta})$  at the time  $k + 1$  based on information up to time  $k$ .  $\hat{P} \in \mathbb{R}^{3 \times 3}$  is a covariance matrix representing uncertainty in the estimated vehicle configuration. The second term in Eq. 6.7 is positive definite which means that  $\hat{P}$ , the position uncertainty, can never decrease.  $\hat{V}$  is our estimate of the covariance of the odometry noise which in reality we do not know.

To simulate the vehicle and the EKF using the Toolbox we define the initial covariance to be quite small since, we assume, we have a good idea of where we are to begin with

```
>> P0 = diag([0.005, 0.005, 0.001].^2);
```

and we pass this to the constructor for an `EKF` object

```
>> ekf = EKF(veh, V, P0);
```

Runing the filter for 1 000 time steps

```
>> ekf.run(1000);
```

drives the robot as before, along a random path. At each time step the filter updates the state estimate using various methods provided by the `Vehicle` object.

**Error ellipses.** If the position of the robot (ignoring orientation) is considered as a PDF such as shown in Fig. 6.3 then a horizontal cross-section will be an ellipse. The 2-dimensional Gaussian probability density function is

$$p(x, y) = \frac{1}{(2\pi)\det(P)^{1/2}} \exp\left\{-\frac{1}{2}(x - \mu_x)^T P^{-1} (x - \mu_x)\right\}$$

where  $\mu_x \in \mathbb{R}^2$  is the mean of  $x$  and  $P \in \mathbb{R}^{2 \times 2}$  is the covariance matrix. The  $1\sigma$  boundary is defined by the points  $x$  such that

$$(x - \mu_x)^T P^{-1} (x - \mu_x) = 1$$

It is useful to plot such an ellipse, as shown in Fig. 6.4, to represent the positional uncertainty. A large ellipse corresponds to a wider PDF peak and less certainty about position.

A handy scalar measure of total uncertainty is the area of the ellipse  $\pi r_1 r_2$  where the radii  $r_i = \sqrt{\lambda_i}$  and  $\lambda_i$  are the eigenvalues of  $P$ . Since  $\det(P) = \prod \lambda_i$  the ellipse area – the scalar uncertainty – is proportional to  $\sqrt{\det(P)}$ . See also Appendices E and F.

The Kalman filter, Appendix H, has two steps: prediction based on the model and update based on sensor data. In this dead-reckoning case we use only the prediction equation.

Stored within the `Vehicle` object.

We can plot the true path taken by the vehicle ▶

```
>> veh.plot_xy()
```

Stored within the `EKF` object.

and the filter's estimate of the path ▶

```
>> hold on
>> ekf.plot_xy('r')
```

These are shown in Fig. 6.4 and we see some divergence between the true and estimated robot path.

The covariance at the 700<sup>th</sup> time step is

```
>> P700 = ekf.history(700).P
P700 =
0.4674    0.0120   -0.0295
0.0120    0.7394    0.0501
-0.0295    0.0501   0.0267
```

The diagonal elements are the estimated variance associated with the states, that is  $\sigma_x^2$ ,  $\sigma_y^2$  and  $\sigma_\theta^2$  respectively. The standard deviation of the PDF associate with the  $x$ -coordinate is

```
>> sqrt(P700(1,1))
ans =
0.6837
```

There is a 95% chance that the robot's  $x$ -coordinate is within the  $\pm 2\sigma$  bound or  $\pm 1.37$  m in this case. We can consider uncertainty for  $y$  and  $\theta$  similarly.

The off-diagonal terms are correlation coefficients and indicate that the uncertainties between the corresponding variables are related. For example the value  $P_{2,3} = P_{3,2} = 0.0501$  indicates that the uncertainties in  $x$  and  $\theta$  are related as we would expect – changes in heading angle will affect the  $x$ -position. Conversely new information about  $\theta$  can be used to correct  $\theta$  as well as  $x$ . The uncertainty in position is described by the top-left  $2 \times 2$  covariance submatrix of  $\hat{P}$ . This can be interpreted as an ellipse defining a confidence bound on position. We can overlay such ellipses on the plot by

```
>> ekf.plot_ellipse([], 'g')
```

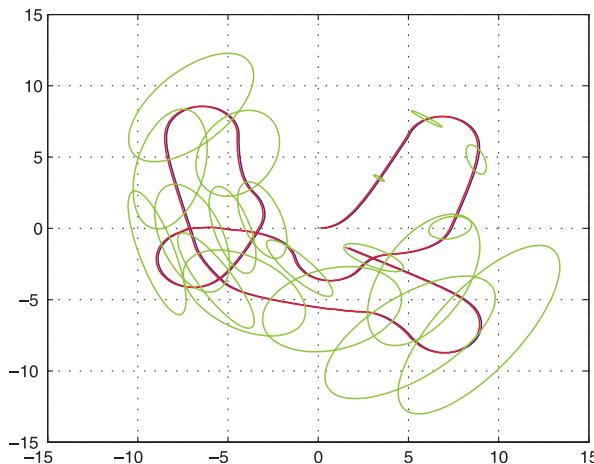
as shown in Fig. 6.4. These correspond to the  $1\sigma$  confidence bound. The vehicle started at the origin and as it progresses we see that the ellipses become larger as the estimated uncertainty increases. The ellipses only show  $x$ - and  $y$ -position but uncertainty in  $\theta$  also grows.

The total uncertainty, ▶ position and heading, is given by  $\sqrt{\det(\hat{P})}$  and is plotted as a function of time

```
>> ekf.plot_P();
```

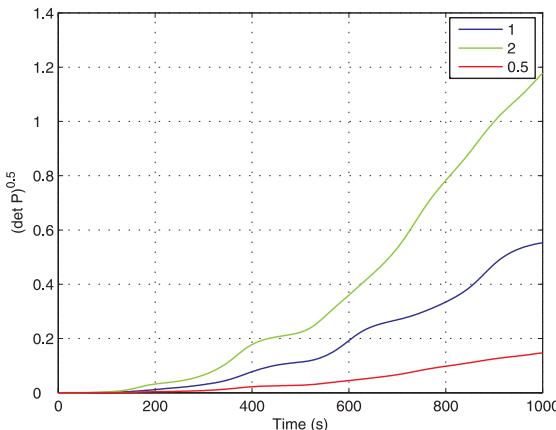
as shown in Fig. 6.5. We observe that it never decreases.

The elements of  $P$  have different units: m<sup>2</sup> and rad. The uncertainty is therefore a mixture of spatial and angular uncertainty with an implicit weighting. Typically  $x, y \gg \pi$  so positional uncertainty dominates.



**Fig. 6.4.**

Deadreckoning using the EKF.  
The true path of the robot,  
blue, and the path estimated  
from odometry in red. The robot  
starts at the origin, uncertainty  
ellipses are indicated in green

**Fig. 6.5.**

Overall uncertainty is given by  $\sqrt{\det(\bar{P})}$  which shows monotonically increasing uncertainty (blue). The effect of changing the magnitude of  $V$  is to change the rate of uncertainty growth. Curves are shown for  $V = \alpha \bar{V}$  where  $\alpha = 1/2, 1, 2$

Note that we have used the odometry covariance matrix  $V$  twice. The first usage, in the `Vehicle` constructor, is the covariance  $V$  of the Gaussian noise that is *actually added* to the true odometry to simulate odometry error in Eq. 6.3. In a real application this noise process would be *hidden inside* the robot and we would not know its parameters. The second usage, in the `EKF` constructor, is  $\hat{V}$  which is our best *estimate* of the odometry covariance and is used in the filter's state covariance update equation Eq. 6.7. The relative values of  $V$  and  $\hat{V}$  control the rate of uncertainty growth as shown in Fig. 6.5. If  $\hat{V} > V$  then  $P$  will be larger than it should be and the filter is pessimistic. If  $\hat{V} < V$  then  $P$  will be smaller than it should be and the filter will be *overconfident* of its estimate. That is, the actual uncertainty is greater than the estimated uncertainty. In practice some experimentation is required to determine the appropriate value for the estimated covariance.

## 6.2 Using a Map

We have seen how uncertainty in position grows without bound using dead-reckoning alone. The solution, as the Phoenicians worked out 4 000 years ago, is to bring in new information from observations of known features in the world. In the examples that follow we will use a map that contains  $N$  fixed but randomly located landmarks whose position is known.

The Toolbox supports a `Map` object

```
>> map = Map(20, 10)
```

that in this case contains  $N = 20$  features uniformly randomly spread over a region spanning  $\pm 10$  m in the  $x$ - and  $y$ -directions and this can be displayed by

```
>> map.plot()
```

The robot is equipped with a sensor that provides observations of the features *with respect to the robot* as described by

$$z = h(x_v, x_f, w) \quad (6.8)$$

where  $x_v$  the vehicle state,  $x_f$  is the known location of the observed feature in the world frame and  $w$  is a random variable that models errors in the sensor.

To make this tangible we will consider a common type of sensor that measures the range and bearing angle to a landmark in the environment, for instance a radar or a scanning-laser rangefinder such as shown in Fig. 6.6. The sensor is mounted onboard the robot so the observation of the  $i^{\text{th}}$  feature is  $x_{fi} = (x_i, y_i)$  is



**Fig. 6.6.** A scanning laser range finder. The sensor has a rotating assembly that emits pulses of infra-red laser light and measures the time taken for the reflection to return. This sensor has a maximum range of 30 m and an angular range of 270 deg. Angular resolution is 0.25 deg and the sensor makes 40 scans per second (Courtesy of Hokuyo Automatic Co. Ltd.)

$$\mathbf{z} = \begin{pmatrix} \sqrt{(y_i - y_v)/(x_i - x_v)} \\ \tan^{-1}(y_i - y_v)/(x_i - x_v) - \theta_v \end{pmatrix} + \begin{pmatrix} w_r \\ w_\beta \end{pmatrix} \quad (6.9)$$

where  $\mathbf{z} = (r, \beta)^T$  and  $r$  is the range,  $\beta$  the bearing angle, and the measurement noise is

It also indicates that covariance is independent of range but in reality covariance may increase with range since the strength of the return signal, laser or radar, drops rapidly ( $1/d^4$ ) with distance ( $d$ ) to the target.

$$\begin{pmatrix} w_r \\ w_\beta \end{pmatrix} \sim N(0, \mathbf{W}), \quad \mathbf{W} = \begin{pmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\beta^2 \end{pmatrix}$$

The diagonal covariance matrix indicates that range and bearing errors are independent. ▶

For this example we set the sensor uncertainty to be  $\sigma_r = 0.1$  m and  $\sigma_\beta = 1^\circ$  giving a sensor covariance matrix

```
>> W = diag([0.1, 1*pi/180].^2);
```

A subclass of [Sensor](#).

In the Toolbox we model this type of sensor with a [RangeBearingSensor](#) object ▶

```
>> sensor = RangeBearingSensor(veh, map, W)
```

which is connected to the vehicle and the map, and the sensor covariance matrix  $\mathbf{W}$  is specified. The [reading](#) method provides the range and bearing ▶ to a randomly selected map feature along with the identity of the map feature it has sensed

```
>> [z,i] = sensor.reading()
z =
    31.9681
    1.6189
i =
    18
```

The identity is an integer  $i \in [1, 20]$  since the map was created with 20 features. We have avoided the data association problem by assuming that we know the identity of the sensed feature. The position of feature 18 can be looked up in the map

```
>> map.feature(18)
-8.0574
 6.4692
```

Using Eq. 6.9 the robot can estimate the range and bearing angle to the feature based on its own estimated position and the known position of the feature from the map. Any difference between the observation and the estimated observation indicates an error in the robot's position estimate – it isn't where it thought it was. This *difference*

$$\mathbf{v}_{\langle k+1 \rangle} = \mathbf{z}_{\langle k+1 \rangle} - \mathbf{h}(\hat{\mathbf{x}}(k+1|k), x_f, 0)$$

See Appendix H.

is key to the operation of the Kalman filter. ▶ It is called the innovation since it represents *new* information. The Kalman filter uses the innovation to correct the state estimate and update the uncertainty estimate  $P_{\langle k \rangle}$ .

As we did previously on page 113 we linearize the observation Eq. 6.8 and write

$$\mathbf{z}_{\langle k \rangle} = \hat{\mathbf{h}} + \mathbf{H}_x(\mathbf{x}_{\langle k \rangle} - \hat{\mathbf{x}}_{\langle k \rangle}) + \mathbf{H}_w \mathbf{w}_{\langle k \rangle} \quad (6.10)$$

where the Jacobians are obtained by differentiating Eq. 6.9 yielding

$$\mathbf{H}_{x_i} = \frac{\partial \mathbf{h}}{\partial \mathbf{x}_v} \Big|_{w=0} = \begin{pmatrix} -\frac{x_i - x_v \langle k \rangle}{r} & -\frac{y_i - y_v \langle k \rangle}{r} & 0 \\ \frac{x_i - x_v \langle k \rangle}{r^2} & -\frac{y_i - y_v \langle k \rangle}{r^2} & -1 \end{pmatrix} \quad (6.11)$$

$$\mathbf{H}_w = \frac{\partial \mathbf{h}}{\partial \mathbf{w}} \Big|_{w=0} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad (6.12)$$

The `RangeBearingSensor` object above includes methods `h` to implement Eq. 6.9 and `H_x` and `H_w` to compute these Jacobians respectively.

Now we use the innovation to *update* the predicted state computed earlier using Eq. 6.6 and Eq. 6.7

$$\hat{x}_{k+1|k+1} = \hat{x}_{k+1|k} + K_{k+1}\nu_{k+1} \quad (6.13)$$

$$\hat{P}_{k+1|k+1} = \hat{P}_{k+1|k} F_x(k)^T - K_{k+1} H_x(k+1) \hat{P}_{k+1|k} \quad (6.14)$$

which are the Kalman filter update equations. These take the *predicted* values for the next time step denoted  $k+1|k$  and apply information from time step  $k+1$  to compute values denoted  $k+1|k+1$ . The innovation has been added to the estimated state after multiplying by the Kalman gain matrix  $K$  which is defined as

$$S_{k+1} = H_x(k+1) \hat{P}_{k+1|k} H_x(k+1)^T + H_w(k+1) \hat{W}_{k+1} H_w(k+1)^T \quad (6.15)$$

$$K_{k+1} = \hat{P}_{k+1|k} H_x(k+1)^T S_{k+1}^{-1} \quad (6.16)$$

where  $\hat{W}$  is the estimated covariance of the sensor noise. Note that the second term in Eq. 6.14 is *subtracted* from the covariance and this provides a means for covariance to decrease which was not possible for the dead-reckoning case of Eq. 6.7.

We now have all the piece to build an estimator that uses odometry and observations of map features. The Toolbox implementation is

```
>> map = Map(20);
>> veh = Vehicle(V);
>> veh.add_driver( RandomPath(map.dim) );
>> sensor = RangeBearingSensor(veh, map, W);
>> ekf = EKF(veh, V, P0, sensor, W, map);
```

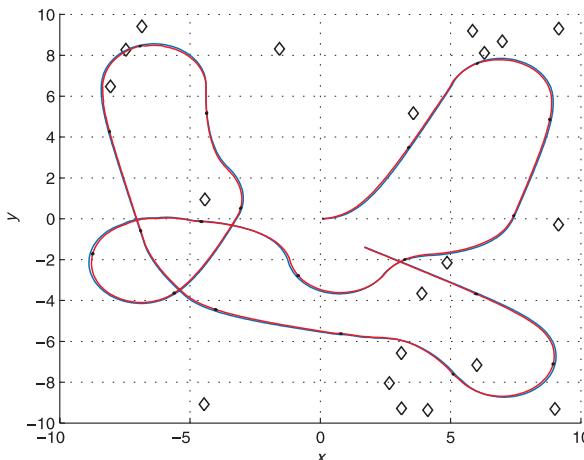
The `Map` constructor has a default map dimension of  $\pm 10$  m which is accessed by its `dim` property.

Running the simulation for 1 000 time steps

```
>> ekf.run(1000);
```

shows an animation of the robot moving and observations being made to the landmarks. We plot the saved results

```
>> map.plot();
>> veh.plot_xy();
>> ekf.plot_xy('r');
>> ekf.plot_ellipse([], 'k')
```



**Fig. 6.7.**

EKF localization showing the true path of the robot (blue) and the path estimated from odometry and landmarks (red). The robot starts at the origin

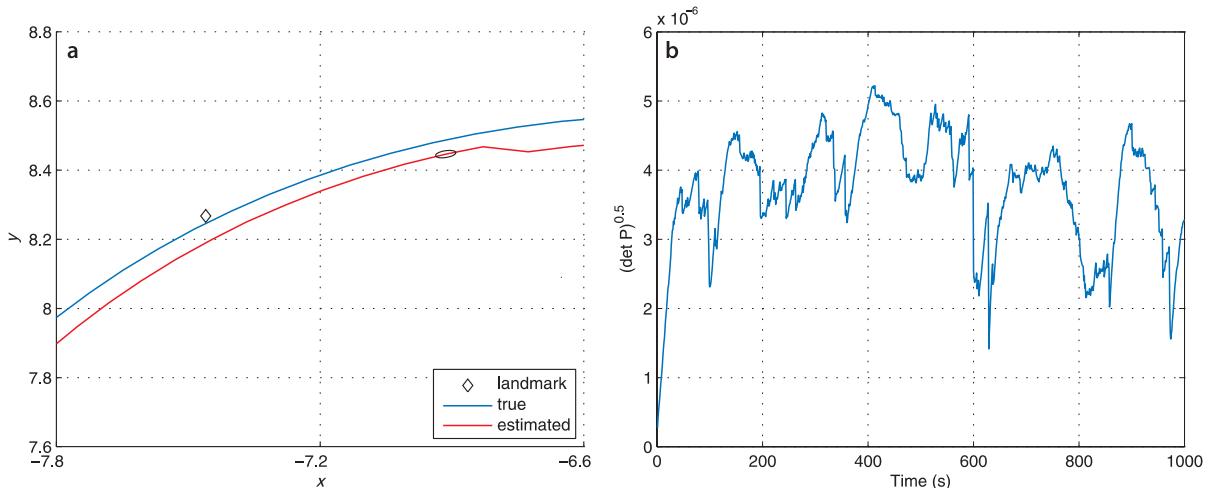


**Reverend Thomas Bayes (1702–1761)** was a non-conformist Presbyterian minister. He studied logic and theology at the University of Edinburgh and lived and worked in Tunbridge-Wells in Kent. There, through his association with the 2<sup>nd</sup> Earl Stanhope he became interested in mathematics and was elected to the Royal Society in 1742. After his death his friend Richard Price edited and published his work in 1763 as *An Essay towards solving a Problem in the Doctrine of Chances* which contains a statement of a special case of Bayes' theorem. Bayes is buried in Bunhill Fields Cemetery in London.

Bayes' theorem shows the relation between a conditional probability and its inverse: the probability of a hypothesis given observed evidence and the probability of that evidence given the hypothesis. Consider the hypothesis that the robot is at location X and it makes a sensor observation S of a known landmark. The *posterior* probability that the robot is at X given the observation S is

$$P(X|S) = \frac{P(S|X)P(X)}{P(S)}$$

where  $P(X)$  is the *prior* probability that the robot is at X (not accounting for any sensory information),  $P(S|X)$  is the likelihood of the sensor observation S given that the robot is at X, and  $P(S)$  is the prior probability of the observation S. The Kalman filter, and the Monte-Carlo estimator we discuss later in this chapter, are essentially two different approaches to solving this inverse problem.



**Fig. 6.8.** a Closeup of the robot's true and estimated path; b the covariance magnitude as a function of time. Overall uncertainty is given by  $\det(P)$  and shows that uncertainty is not increasing with time

which are shown in Fig. 6.7. We are hard pressed to see the error ellipses since they are now so small.

Figure 6.8a shows a zoomed view of the robot's actual and estimated path. We can see a small error ellipse and we can also see a *jag* in the estimated path. The vehicle state evolves smoothly with time according to the bicycle model of Eq. 4.2 but new information from a sensor reading updates the state and can sometimes cause noticeable changes, jumping the state estimate either forwards, backwards or sideways. Figure 6.8b shows that the uncertainty is no longer growing monotonically, new information is reducing the uncertainty through Eq. 6.14.

As discussed earlier for  $V$  we also use  $W$  twice. The first usage, in the constructor for the `RangeBearingSensor` object, is the covariance  $W$  of the Gaussian noise that is *actually added* to the computed range and bearing to simulate sensor error as in Eq. 6.9. The second usage,  $\hat{W}$  is our best estimate of the sensor covariance which is used by the Kalman filter Eq. 6.15.

This EKF framework allows data from many and varied sensors to update the state which is why the estimation problem is also referred to as sensor fusion. For example heading angle from a compass, yaw rate from a gyroscope, target bearing angle from a camera, position from GPS could all be used to update the state. For each sensor we

need only to provide the observation function  $h(\cdot)$ , the Jacobians  $H_x$  and  $H_w$  and some estimate of the sensor output covariance  $W$ . The function  $h(\cdot)$  can be non-linear and even non-invertible – the EKF will do the rest.

In this example, and in the next two sections, we assume that the sensor provides information about the position of the target with respect to the robot and also the *identity* of the target. In practice most landmarks are anonymous,► that is, we do not know their identity and hence do not know their true location in the world. We therefore need to solve the correspondence or target association problem – given the map and the observation, determine which feature is the *most likely* to have been seen. Errors in this step can lead rapidly to failure of the estimator – the system sees target  $i$  but thinks it is target  $j$ . The state vector is updated incorrectly making it more likely to incorrectly associate targets and a downward spiral ensues. The covariance may not necessarily increase greatly and this is a dangerous thing – a confident but wrong robot. In indoor robotic problems the sensor may detect spurious targets such as people moving in the environment and the people will also obscure real landmarks.

An alternative is a multi-hypothesis estimator, such as the particle filter that we will discuss in Sect. 6.5, which can model the possibility of observing landmark A *or* landmark B, and future observations will reinforce one hypothesis and weaken the others. The extended Kalman filter uses a Gaussian probability model, with just one peak, which limits it to holding only a single hypothesis about location.

Bar codes could be used to provide distinct target identity in some applications such as indoor mobile robots.

### 6.3 Creating a Map

So far we have taken the existence of the map for granted, an understandable mindset given that maps today are common and available for free via the internet. Nevertheless somebody, or something, has to create maps. Our next example considers the problem of a robot moving in an environment with landmarks and creating a map of their locations.

As before we have a range and bearing sensor mounted on the robot which measures, imperfectly, the position of features with respect to the robot. There are a total of  $N$  features in the environment and as for the previous example we assume that the sensor can determine the identity of each observed feature. However for this case we assume that the robot knows its own location perfectly – it has ideal localization. This is unrealistic but this scenario is an important stepping stone to the next section.►

Since the vehicle pose is known perfectly we do not need to estimate it, but we do need to estimate the coordinates of the landmarks. For this problem the state vector comprises the estimated coordinates of the  $M$  landmarks that have been observed so far

$$\hat{x} = (x_1, y_1, x_2, y_2, \dots, x_M, y_M)^T$$

and has  $2M$  elements. The corresponding estimated covariance  $\hat{P}$  will be a  $2M \times 2M$  matrix. The state vector has a variable length since we do not know in advance how many landmarks exist in the environment. Initially  $M = 0$  and is incremented every time a previously unseen feature is observed.

The prediction equation is straightforward in this case since the features do not move

$$\hat{x}\langle k+1|k \rangle = \hat{x}\langle k|k \rangle \quad (6.17)$$

$$\hat{P}\langle k+1|k \rangle = \hat{P}\langle k|k \rangle \quad (6.18)$$

A close and realistic approximation would be a high-end RTK GPS system operating in an environment with no buildings or hills to obscure satellites.

We introduce the function  $g(\cdot)$  which is the inverse of  $h(\cdot)$  and gives the coordinates of the observed feature based on the known vehicle pose and the sensor observation

$$g(x_v, z) = \begin{pmatrix} x_v + r_z \cos(\theta_v + \theta_z) \\ y_v + r_z \sin(\theta_v + \theta_z) \end{pmatrix}$$

Since  $\hat{x}$  has a variable length we need to extend the state vector and the covariance matrix whenever we encounter a landmark we have not previously seen. The state vector is extended by the function  $y(\cdot)$

$$\mathbf{x}\langle k|k \rangle^* = \mathbf{y}(\mathbf{x}\langle k|k \rangle, \mathbf{z}\langle k \rangle, \mathbf{x}_v\langle k|k \rangle) \quad (6.19)$$

$$= \begin{pmatrix} \mathbf{x}\langle k|k \rangle \\ \mathbf{g}(\mathbf{x}_v\langle k|k \rangle, \mathbf{z}\langle k \rangle) \end{pmatrix} \quad (6.20)$$

which appends the estimate of the new feature's coordinates to the coordinates already in the map. The order of feature coordinates within  $\hat{x}$  therefore depends on the order in which they are observed.

The covariance matrix also needs to be extended when a new landmark is observed and this is achieved by

$$\hat{\mathbf{P}}\langle k|k \rangle^* = \mathbf{Y}_z \begin{pmatrix} \hat{\mathbf{P}}\langle k|k \rangle & 0 \\ 0 & \hat{W} \end{pmatrix} \mathbf{Y}_z^T$$

where  $\mathbf{Y}_z$  is another Jacobian

$$\mathbf{Y}_z = \frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} \mathbf{I}_{n \times n} & & \mathbf{0}_{n \times 2} \\ \mathbf{G}_x & \mathbf{0}_{2 \times n-3} & \mathbf{G}_z \end{pmatrix} \quad (6.21)$$

$$\mathbf{G}_x = \frac{\partial \mathbf{g}}{\partial \mathbf{x}_v} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad (6.22)$$

$$\mathbf{G}_z = \frac{\partial \mathbf{g}}{\partial \mathbf{z}} = \begin{pmatrix} \cos(\theta_v + \theta_z) & -r_z \sin(\theta_v + \theta_z) \\ \sin(\theta_v + \theta_z) & r_z \cos(\theta_v + \theta_z) \end{pmatrix} \quad (6.23)$$

where  $n$  is the dimension of  $\hat{\mathbf{P}}$  prior to it being extended.

An additional Jacobian for  $\mathbf{h}(\cdot)$  is

$$\mathbf{H}_{x_i} = \frac{\partial \mathbf{h}}{\partial \mathbf{x}_i} = \begin{pmatrix} \frac{x_i - x_v}{r} & \frac{y_i - y_v}{r} \\ -\frac{x_i - x_v}{r^2} & \frac{y_i - y_v}{r^2} \end{pmatrix} \quad (6.24)$$

which relates change in map features to change in observation and is implemented by the `Sensor` class method `H_xf`.

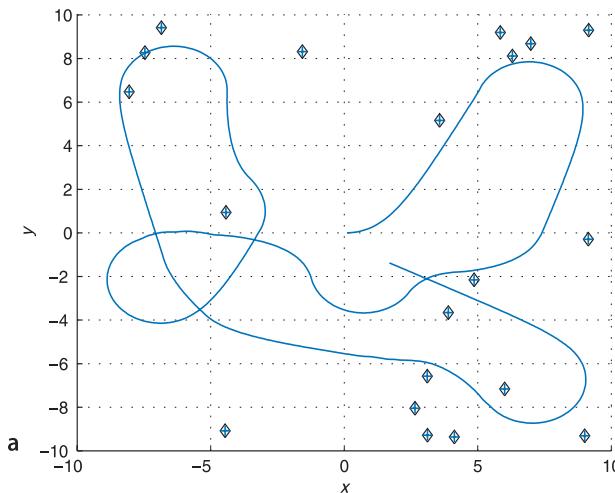
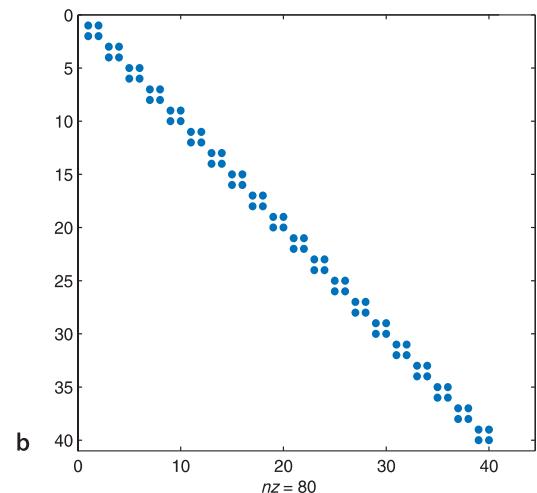
The Jacobian  $\mathbf{H}_x$  used in Eq. 6.14 describes how the feature observation changes with respect to the state vector. However in this case, the observation depends only on a single observed feature so this Jacobian is mostly zeros

$$\mathbf{H}_x = (0 \cdots \mathbf{H}_{x_i} \cdots 0) \quad (6.25)$$

where  $\mathbf{H}_{x_i}$  is at the location corresponding to the state  $x_i$ .

The Toolbox implementation is

```
>> map = Map(20);
>> veh = Vehicle([]); % error free vehicle
>> veh.add_driver(RandomPath(map.dim));
>> W = diag([0.1, 1*pi/180].^2);
>> sensor = RangeBearingSensor(veh, map, W);
>> ekf = EKF(veh, [], [], sensor, W, []);
```

**a****b**

**Fig. 6.9.** EKF mapping results.  
**a** The estimated landmarks are indicated by + -markers with  $5\sigma$  confidence ellipses (green), the true location (black  $\diamond$ -marker) and the robot's path (blue); **b** the non-zero elements of the final covariance matrix

the empty matrices passed to `EKF` indicate respectively that there is no estimated odometry covariance for the vehicle (the estimate is perfect), no initial vehicle state covariance, and the map is unknown. We run the simulation for 1 000 time steps

```
>> ekf.run(1000);
```

and see an animation of the robot moving and the covariance ellipses associated with the map features evolving over time. The estimated landmark positions

```
>> map.plot();
>> ekf.plot_map(5, 'g');
>> veh.plot_xy('b');
```

are shown in Fig. 6.9a as  $5\sigma$  confidence ellipses (in order to be visible) along with the true landmark positions and the path taken by the robot. The covariance matrix has a block diagonal structure which is displayed graphically

```
>> spy( ekf.P_est );
```

in Fig. 6.9b. The blue dots represent non-zero elements<sup>►</sup> and each  $2 \times 2$  block represents the covariance of the position of a map feature. The correlations, the off-diagonal elements are zero, which implies that the feature estimates are uncorrelated or independent. This is to be expected since observing feature  $i$  provides no new information about feature  $j \neq i$ .

Internally the `EKF` object maintains a table to relate the feature identity, returned by the `RangeBearingSensor`, to the position of that feature's coordinates in the state vector. For example the landmark with identity 10

```
>> ekf.features(:,10)
ans =
    19
    51
```

was seen a total of 51 times during the simulation and comprises elements 19 and 20 of  $\hat{x}$

```
>> ekf.x_est(19:20)'
ans =
    5.8441    9.1898
```

which is its estimated location. Its estimated covariance is

```
>> ekf.P_est(19:20,19:20)
ans =
    1.0e-03 *
    0.2363   -0.0854
   -0.0854    0.2807
```

This is known as the sparsity structure of the matrix, and a large class of numerical algorithms exist that work efficiently on matrices that are predominantly zero. MATLAB's `spy` function shows the sparsity structure of matrices graphically. See [help sparse](#) for more details.

## 6.4 Localization and Mapping

Finally we tackle the problem of determining our position and creating a map at the same time. This is an old problem in marine navigation and cartography – incrementally extending maps while also using the map for navigation. In robotics this problem is known as simultaneous localization and mapping (SLAM) or concurrent mapping and localization (CML). This is sometimes referred to as a “chicken and egg” problem but based on what we have learnt in the previous sections this problem is now quite straightforward to solve.

The state vector comprises the vehicle configuration *and* the coordinates of the  $M$  landmarks that have been observed so far

$$\hat{x} = (x_v, y_v, \theta_v, x_1, y_1, x_2, y_2, \dots, x_M, y_M)$$

and has  $2M + 3$  elements. The covariance is a  $(2M + 3) \times (2M + 3)$  matrix and has the structure

$$\hat{P} = \begin{pmatrix} \hat{P}_{vv} & \hat{P}_{vm} \\ \hat{P}_{vm}^T & \hat{P}_{mm} \end{pmatrix}$$

where  $\hat{P}_{vv}$  is the covariance of the vehicle state,  $\hat{P}_{mm}$  the covariance of the map features, and  $\hat{P}_{vm}$  is the correlation between vehicle and map states.

The predicted vehicle state and covariance are given by Eq. 6.6 and Eq. 6.7 and the sensor update is given by Eq. 6.13 to 6.16. When a new feature is observed the state vector is updated using the Jacobian  $Y_z$  given by Eq. 6.21 but in this case  $G_x$  is non-zero

$$G_x = \frac{\partial g}{\partial x_v} = \begin{pmatrix} 1 & 0 & -r_z \sin(\theta_v + \theta_z) \\ 0 & 1 & r_z \cos(\theta_v + \theta_z) \end{pmatrix}$$

since the estimate of the new feature depends on the state vector which now contains the vehicle’s pose.

The Jacobian  $H_x$  describes how the feature observation changes with respect to the state vector. The observation will depend on the position of the vehicle and on the position of the observed feature and is

$$H_x = (H_{x_v} \cdots 0 \cdots H_{x_i} \cdots 0) \quad (6.26)$$

where  $H_{x_i}$  is at the location corresponding to the state  $x_i$ . This is similar to Eq. 6.25 but with the non-zero block  $H_{xv}$  at the left to account for the effect of vehicle position.

The Kalman gain matrix  $K$  multiplies innovation from the landmark observation, a 2-vector, so as to update *every* element of the state vector – the pose of the vehicle *and* the position of *every* map feature.

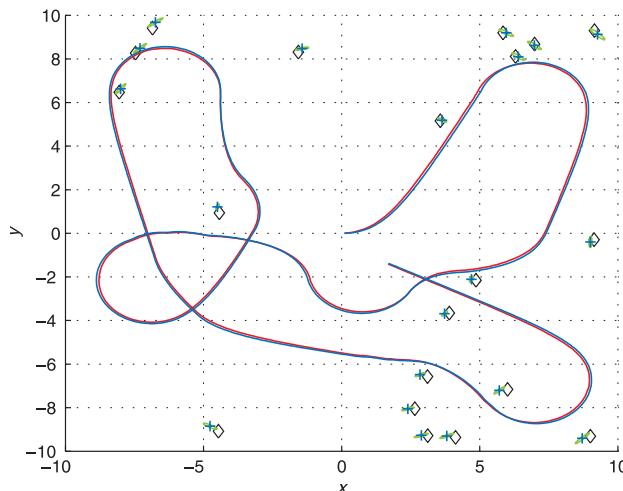
The Toolbox implementation is by now quite familiar

```
>> P0 = diag([.01, .01, 0.005].^2);
>> map = Map(20);
>> veh = Vehicle(W);
>> veh.add_driver(RandomPath(map.dim));
>> sensor = RangeBearingSensor(veh, map, W);
>> ekf = EKF(veh, V, P0, sensor, W, []);
```

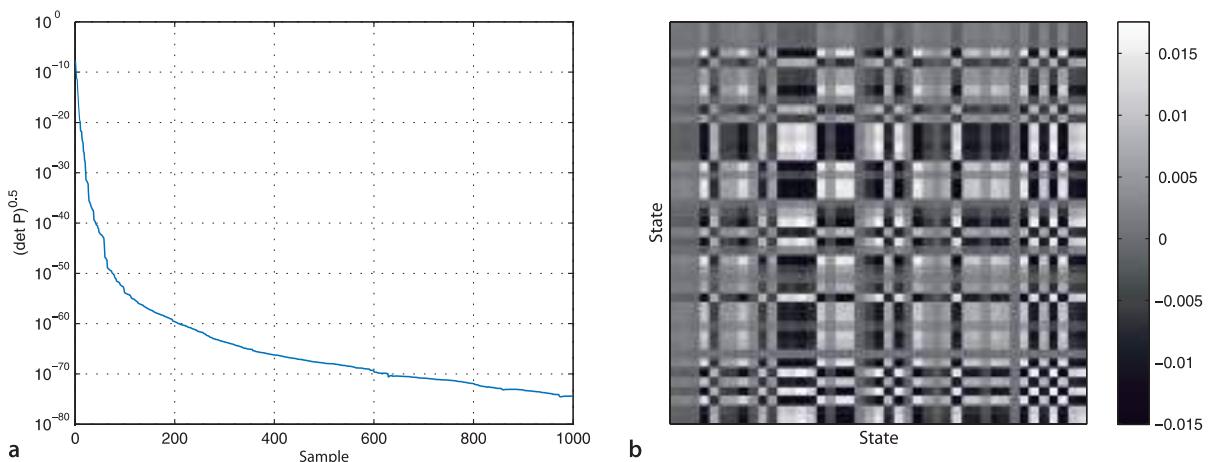
and the empty matrix passed to `EKF` indicates that the map is unknown. `P0` is the initial  $3 \times 3$  covariance for the vehicle state.

We run the simulation for 1 000 time steps

```
>> ekf.run(1000);
```

**Fig. 6.10.**

Simultaneous localization and mapping showing the true (blue) and estimated (red) robot path superimposed on the true map (black  $\diamond$ -marker). The estimated map features are indicated by  $+$ -markers and the  $5\sigma$  confidence ellipses (green)



**Fig. 6.11.** Simultaneous localization and mapping. **a** Covariance versus time; **b** the final covariance matrix (values have been scaled in the interval 0 to 255)

and as usual an animation is shown of the vehicle moving. We also see the covariance ellipses associated with the map features evolving over time. We can plot the results

```
>> map.plot();
>> ekf.plot_map(5, 'g');
>> ekf.plot_xy('r');
>> veh.plot_xy('b');
```

which are shown in Fig. 6.10.

Figure 6.11a shows that uncertainty is decreasing over time. Figure 6.11b shows the final covariance matrix as an image and we see a complex structure. Unlike the mapping case  $\hat{P}_{mm}$  is not block diagonal, and the finite off-diagonal terms represent correlation *between* the features in the map. The feature uncertainties never increase, the prediction model is that they don't change, but they also never drop below the initial uncertainty of the vehicle. The block  $\hat{P}_{vm}$  is the correlation between errors in the vehicle and the map features. A feature's location estimate is a function of the vehicle's location and errors in the vehicle location appear as errors in the feature location – and vice versa.

The correlations cause information about the observation of any feature to affect the estimate of every other feature in the map and the vehicle pose. It is as if all the states were connected by springs and the movement of any one affects all the others.

## 6.5 Monte-Carlo Localization

The estimation examples so far have assumed that the error in sensors such as odometry and landmark range and bearing have a Gaussian probability density function. In practice we might find that a sensor has a one sided distribution (like a Poisson distribution) or a multimodal distribution with several peaks. The functions we used in the Kalman filter such as Eq. 6.3 and Eq. 6.8 are strongly non-linear which means that sensor noise with a Gaussian distribution will not result in a Gaussian error distribution on the value of the function – this is discussed further in Appendix H. The probability density function associated with a robot's configuration may have multiple peaks to reflect several hypotheses that equally well explain the data from the sensors as shown for example in Fig. 6.3c.

The Monte-Carlo estimator that we discuss in this section makes no assumptions about the distribution of errors. It can also handle multiple hypotheses for the state of the system. The basic idea is disarmingly simple. We maintain many *different* versions of the vehicle's state vector. When a new measurement is available we score how well each version of the state explains the data. We keep the best fitting states and randomly perturb them to form a new generation of states. Collectively these many possible states and their scores approximate a probability density function for the state we are trying to estimate. There is never any assumption about Gaussian distributions nor any need to linearize the system. While computationally expensive it is quite feasible to use this technique with standard desktop computers. If we plot these state vectors as points we have a cloud of particles hence this type of estimator is often referred to as a particle filter.

We will apply Monte-Carlo estimation to the problem of localization using odometry and a map. Estimating only three states ( $x, y, \theta$ ) is computationally tractable to solve with straightforward MATLAB® code. The estimator is initialized by creating  $N$  particles  $\mathbf{x}_{v,i}$ ,  $i \in [1, N]$  distributed randomly over the configuration space of the vehicle. All particles have the same initial weight or likelihood  $w_i = 1 / N$ . The steps in the main iteration of the algorithm are:

1. Apply the state update to each particle

$$\mathbf{x}_{v,i}^{(k+1)} = f(\mathbf{x}_{v,i}^{(k)}, \delta^{(k)}) + \mathbf{q}^{(k)}$$

moving each particle according to the measured odometry. We also add a random vector  $\mathbf{q}^{(k)}$  which represents uncertainty in the position of the vehicle. Often  $\mathbf{q}$  is drawn from a Gaussian random variable with covariance  $\mathbf{Q}$  but any physically meaningful distribution can be used.

**Monte Carlo methods** are a class of computational algorithms that rely on repeated random sampling to compute their results. An early example of this idea is Buffon's needle problem posed in the eighteenth century by Georges-Louis Leclerc, Comte de Buffon: *Suppose we have a floor made of parallel strips of wood of equal width  $t$ , and a needle of length  $l$  is dropped onto the floor. What is the probability that the needle will lie across a line between the strips?* If  $n$  needles are dropped and  $h$  cross the lines, the probability can be shown to be  $h/n = 2l/2\pi t$  and in 1901 an Italian mathematician Mario Lazzarini performed the experiment, tossing a needle 3 408 times, and obtained the estimate  $\pi \approx 355 / 113$  (3.1415929).

Monte Carlo methods are often used when simulating systems with a large number of coupled degrees of freedom with significant uncertainty in inputs. Monte Carlo methods tend to be used when it is infeasible or impossible to compute an exact result with a deterministic algorithm. Their reliance on repeated computation and random or pseudo-random numbers make them well suited to calculation by a computer. The method was developed at Los Alamos as part of the Manhattan project during WW II by the mathematicians John Von Neuman, Stanislaw Ulam and Nicholas Metropolis. The name Monte Carlo alludes to games of chance and was the code name for the secret project.

2. We make an observation  $z$  of feature  $i$  which has, according to the map, coordinate  $x_f$ . For each particle we compute the innovation

$$\nu_i = h(x_{v,i}, x_f) - z$$

which is the error between the predicted and actual landmark observation. A likelihood function provides a scalar measure of how well the particular particle explains this observation. In this example we choose a Gaussian likelihood function

$$w = \exp(-0.5\nu^T L \nu) + w_0$$

where  $w$  is referred to as the *importance* or *weight* of the particle,  $L$  is a covariance matrix, and  $w_0 > 0$  ensures that there is a finite but small probability associated with any point in the state space. We use a Gaussian function only for convenience, the function does not need to be smooth or invertible but only to adequately describe the likelihood of an observation.

3. Select the particles that best explain the observation, a process known as resampling. A common scheme is to randomly select particles according to their weight. Given  $N$  particles  $x_i$  with corresponding weights  $w_i$  we first normalize the weights  $w'_i = w_i / \sum_{i=1}^N w_i$  and construct a cumulative histogram  $c_j = \sum_{i=1}^j w'_i$ . We then draw a uniform random number  $r \in [0, 1]$  and find  $\arg \min |c_j - r|$  where particle  $j$  is selected for the next generation. The process is repeated  $N$  times.

Particles with a large weight will correspond to a larger fraction of the vertical span of the cumulative histogram and therefore be more likely to be chosen. The result will have the same number of particles, some will have been copied▶ multiple times, others not at all.

Step 1 of the next iteration will spread out these copies through the addition of  $q(k)$ .

The Toolbox implementation is broadly similar to the previous examples. We create a map

```
>> map = Map(20);
```

and a robot with noisy odometry and an initial condition

```
>> W = diag([0.1, 1*pi/180].^2);
>> veh = Vehicle(W);
>> veh.add_driver(RandomPath(10));
```

and then a sensor with noisy readings

```
>> V = diag([0.005, 0.5*pi/180].^2);
>> sensor = RangeBearingSensor(veh, map, V);
```

For the particle filter we need to define two covariance matrices. The first is the covariance of the random noise added to the particle states at each iteration to represent uncertainty in configuration. We choose the covariance values to be comparable with those of  $W$

```
>> Q = diag([0.1, 0.1, 1*pi/180]).^2;
```

and the covariance of the likelihood function applied to innovation

```
>> L = diag([0.1 0.1]);
```

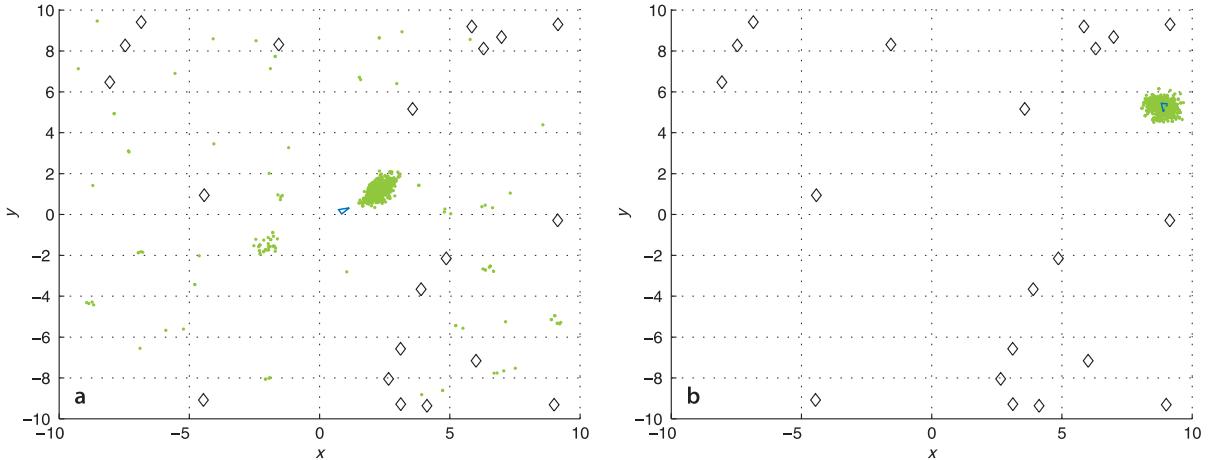
Finally we construct a `ParticleFilter` estimator

```
>> pf = ParticleFilter(veh, sensor, Q, L, 1000);
```

which is configured with 1000 particles. The particles are initially uniformly distributed over the 3-dimensional configuration space.

We run the simulation for 1000 time steps

```
>> pf.run(1000);
```



**Fig. 6.12.** Particle filter results showing the evolution of the evolution of the particle cloud (green dots) over time. The vehicle is shown as a blue triangle

and watch the animation, two snapshots of which are shown in Fig. 6.12. We see the particles move about as their states are updated by odometry and random perturbation. The initially randomly distributed particles begin to aggregate around those regions of the configuration space that best *explain* the sensor observations that are made. In Darwinian fashion these particles become more highly weighted and survive the resampling step while the lower weight particles are extinguished. The plot is 3-dimensional, so you can rotate the graph while the animation is running to see the heading angle state which is the height ( $z$ -coordinate) of the particles.

The particles approximate the probability density function of the robot's configuration. The most likely configuration is the expected value or mean of all the particles. A measure of uncertainty of the estimate is the spread of the particle cloud or its standard deviation. The `ParticleFilter` object keeps the history of the mean and standard deviation of the particle state at each time step. As usual we plot the results of the simulation

```
>> map.plot();
>> veh.plot_xy('b');
```

and overlay the mean of the particle cloud

```
>> pf.plot_xy('r');
```

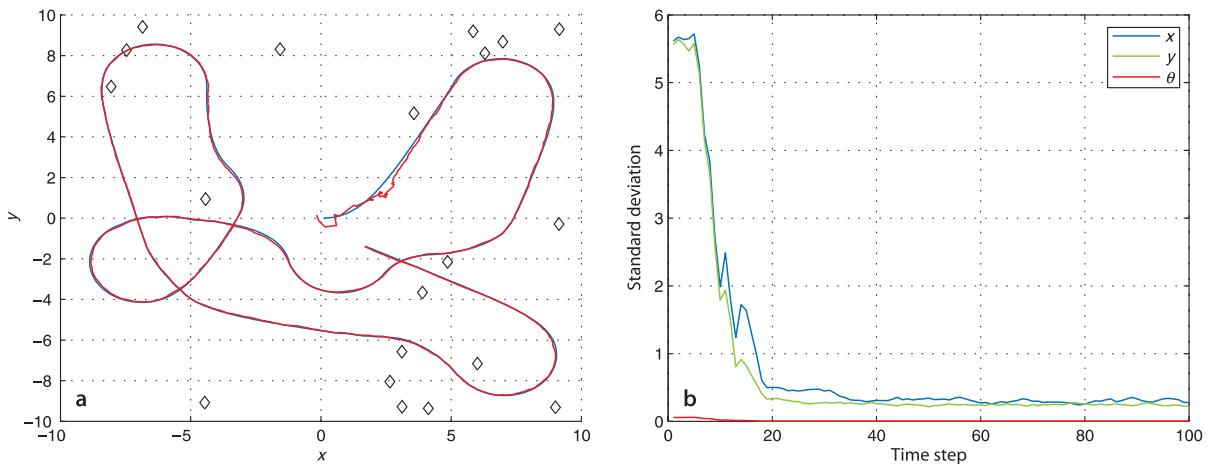
which is shown in Fig. 6.13. The initial part of the estimated path has quite high standard deviation since the particles have not converged on the true configuration. We can plot the standard deviation against time

```
>> plot(pf.std(1:100,:))
```

and this is shown in Fig. 6.13b. We can see the sudden drop between timesteps 10–20 as the particles that are distant from the true solution are eliminated. As mentioned at the outset the particles are a sampled approximation to the PDF and we can display this as

```
>> pf.plot_pdf()
```

The problem we have just solved is known in robotics as the kidnapped robot problem where a robot is placed in the world with no idea of its initial location. To represent this large uncertainty we uniformly distribute the particles over the 3-dimensional configuration space and their sparsity can cause the particle filter to take a long time to converge unless a very large number of particles is used. It is debatable whether this is a realistic problem. Typically we have some approximate initial pose of the robot and the particles would be initialized to that part of the configuration space. For example if we know the robot is in a corridor then the particles would be placed in those areas of the map that are corridors, or if we know the robot is pointing north then set all particles to have that orientation.



Setting the parameters of the particle filter requires a little experience and the best way to learn is to experiment. For the kidnapped robot problem we set  $Q$  and the number of particles high so that the particles explore the configuration space but once the filter has converged lower values could be used. There are many variations on the particle filter in the shape of the likelihood function and the resampling strategy.

**Fig. 6.13.** Particle filter results.  
a True (blue) and estimated (red) robot path; b standard deviation of the particles versus time

## 6.6 Wrapping Up

In this chapter we learnt about two ways of estimating a robot's position: by dead reckoning, and by observing features whose true position is known from a map. Dead reckoning is based on the integration of odometry information, the distance travelled and the change in heading angle. Over time errors accumulate leading to increased uncertainty about the pose of the robot.

We modelled the error in odometry by adding noise to the sensor outputs. The noise values are drawn from some distribution that describes the errors of that particular sensor. For our simulations we used zero-mean Gaussian noise with a specified covariance, but only because we had no other information about the specific sensor. The most realistic noise model available should be used. We then introduced the Kalman filter which provides an optimal estimate of the true configuration of the robot based on noisy measurements. The Kalman filter is however only optimal for the case of zero-mean Gaussian noise and a linear model. The model that describes how the robot's configuration evolves with time is non-linear and we approximated it with a linear model which requires some Jacobians to be computed, an approach known as extended Kalman filtering.

The Kalman filter also estimates uncertainty associated with the configuration estimate and we see that the magnitude can never decrease and typically grows without bound. Only additional sources of information can reduce this growth and we looked at how observations of landmarks, whose location are known, relative to the robot can be used. Once again we use the Kalman filter but in this case we use both the prediction and the update phases of the filter. We see that in this case the uncertainty can be decreased by a landmark observation and that over the longer term the uncertainty does not grow.

We then applied the Kalman filter to the problem of estimating the positions of the landmarks given that we knew the precise position of the vehicle. In this case the state vector of the filter was the coordinates of the landmarks themselves.

Finally we brought all this together and estimated the vehicle's position, the position of the landmarks and their uncertainties. The state vector in this case contained the configuration of the robot and the coordinates of the landmarks.

An important problem when using landmarks is data association, being able to determine which landmark has been known or observed by the sensor so that its position can be looked up in a map or in a table of known or estimated landmark positions. If the wrong landmark is looked up then an error will be introduced in the robot's position.

Finally we learnt about Monte-Carlo estimation and introduced the particle filter. This technique is computationally intensive but makes no assumptions about the distribution of errors from the sensor or the linearity of the vehicle model, and supports multiple hypotheses.

---

### Further Reading

The book by Borenstein et al. (1996) has an excellent discussion of robotic sensors in general and odometry in particular. Although out of print it is available online. The book by Everett (1995) covers odometry, range and bearing sensors, as well as radio, ultrasonic and optical localization systems. Unfortunately the discussion of range and bearing sensors is now quite dated since this technology has evolved rapidly over the last decade. The handbook (Siciliano and Khatib 2008, § 20, § 22) provides a brief but more modern treatment of these sensors.

The books of Borenstein et al. (1996) and Everett (1995) were published *before* GPS became operational. The principles of GPS and other radio-based localization systems are covered in some detail in the book by Groves (2008). The Robotics Handbook (Siciliano and Khatib 2008, § 20) also briefly describes GPS, and a number of links to GPS technical data are provided from this book's web site.

There are many published and online resources for Kalman filtering. Kálmán's original paper was published in 1960. The book by Zarchan and Musoff (2005) is a very clear and readable introduction to Kalman filtering. I have always found the classic 1970 book by Jazwinski (1970) to be very readable and it has recently been republished. Bar-Shalom et al. (2001) provide comprehensive coverage of estimation theory and also the use of GPS. An excellent reference for EKF and Monte-Carlo localization is the book by Thrun et al. (2005). Data association is an important topic and is covered in detail in, the now very old, book by Bar-Shalom and Fortmann (1988). The Robotics Handbook (Siciliano and Khatib 2008, § 4) covers Kalman filter and data association. Welch and Bishop's online resources at <http://www.cs.unc.edu/~welch/kalman> have pointers to papers, courses, software and links to other relevant web sites.

A significant limitation of the EKF is its first-order linearization, particularly for processes with strong non-linearity. Alternatives include the iterated EKF described by Jazwinski (1970) or the Unscented Kalman Filter (UKF) (Julier and Uhlmann 2004) which uses discrete sample points to approximate the PDF. Some of these topics are covered in the Handbook (Siciliano and Khatib 2008, § 25) as multi-sensor fusion. The book by Siegwart et al. (2011) also has a good treatment of robot localization.

There is a very large literature on SLAM and this chapter has only touched the surface with a very classical EKF-based approach which does not scale well for large numbers of features. FastSLAM (Montemerlo et al. 2002) is a state-of-the-art algorithm for large-scale applications. There are a lot of online resources related to SLAM. Many of the SLAM summer schools have websites that host excellent online resources such as lecture notes and practicals.

A collection of open-source SLAM implementations such as gmapping and iSam is available from OpenSLAM at <http://www.openslam.org>. MATLAB® implementations include the CAS Robot Navigation Toolbox for planar SLAM at <http://www.cas.kth.se/toolbox> and a 6DOF SLAM system at <http://homepages.laas.fr/jsola/JuanSola/eng/toolbox.html>.

The book *Longitude* (Sobel 1996) is a very readable account of the longitude problem and John Harrison's quest to build a marine chronometer.

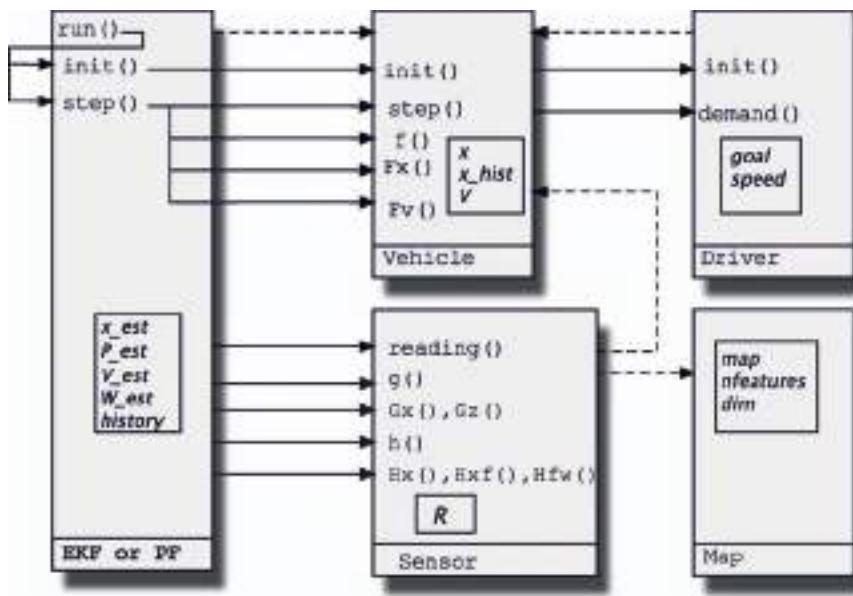


Fig. 6.14.

Toolbox class relationship for localization and mapping. Each class is shown as a rectangle, method calls are shown as arrows from caller to callee, properties are boxed, and dashed lines represent object references

### Notes on Toolbox Implementation

This chapter has introduced a number of Toolbox classes to solve mapping and localization problems. The principle was to decompose the problem into clear functional subsystems and implement these as a set of cooperating classes, and this allows quite complex problems to be expressed in very few lines of code.

The relationships between the objects and their methods and properties are shown in Fig. 6.14. As always more documentation is available through the online help system or comments in the code.

### Exercises

1. What is the value of the Longitude Prize in todays currency?
2. Implement a driver object (page 113) that drives the robot around inside a circle with specified centre and radius.
3. Derive an equation for heading odometry in terms of the rotational rate of the left and right wheels.
4. Dead-reckoning (page 111)
  - a) Experiment with different values of  $P_0$ ,  $V$  and  $\hat{V}$ .
  - b) Fig. 6.4 compares the actual and estimated position. Plot the actual and estimated heading angle.
  - c) Compare the variance associated with heading to the variance associated with position. How do these change with increasing levels of range and bearing angle variance in the sensor?
5. Using a map (page 116)
  - a) Vary the characteristics of the sensor (covariance, sample rate, range limits and bearing angle limits) and investigate the effect on performance
  - b) Vary  $W$  and  $\hat{W}$  and investigate what happens to estimation error and final covariance.
  - c) Modify the `RangeBearingSensor` to create a bearing-only sensor, that is, as a sensor that returns angle but not range. The implementation includes all the Jacobians. Investigate performance.

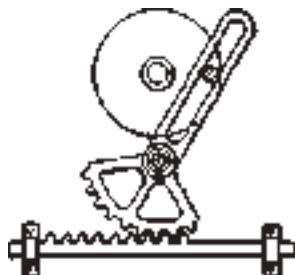
- d) Modify the sensor model to return occasional errors (specify the error rate) such as incorrect range or beacon identity. What happens?
  - e) Modify the EKF to perform data association instead of using the landmark identity returned by the sensor.
  - f) Figure 6.7 compares the actual and estimated position. Plot the actual and estimated heading angle.
  - g) Compare the variance associated with heading to the variance associated with position. How do these change with increasing levels of range and bearing angle variance in the sensor?
6. Making a map (page 120)
- a) Vary the characteristics of the sensor (covariance, sample rate, range limits and bearing angle limits) and investigate the effect on performance.
  - b) Use the bearing-only sensor from above and investigate performance relative to using a range and bearing sensor.
  - c) Modify the EKF to perform data association instead of using identity returned by the sensor.
7. Simultaneous localization and mapping (page 123)
- a) Vary the characteristics of the sensor (covariance, sample rate, range limits and bearing angle limits) and investigate the effect on performance.
  - b) Use the bearing-only sensor from above and investigate performance relative to using a range and bearing sensor.
  - c) Modify the EKF to perform data association instead of using the landmark identity returned by the sensor.
  - d) Fig. 6.10 compares the actual and estimated position. Plot the actual and estimated heading angle.
  - e) Compare the variance associated with heading to the variance associated with position. How do these change with increasing levels of range and bearing angle variance in the sensor?
8. Particle filter (page 125)
- a) Run the filter numerous times. Does it always converge?
  - b) Vary the parameters  $Q$ ,  $L$ ,  $w_0$  and  $N$  and understand their effect on convergence speed and final standard deviation.
  - c) Investigate variations to the kidnapped robot problem. Place the initial particles around the initial pose. Place the particles uniformly over the  $xy$ -plane but set their orientation to its actual value.
  - d) Use a different type of likelihood function, perhaps inverse distance, and compare performance.

# Robot Arm Kinematics

*Take to Kinematics. It will repay you.*

*It is more fecund than geometry; it adds a fourth dimension to space.*

Chebyshev



Kinematics is the branch of mechanics that studies the motion of a body or a system of bodies without consideration given to its mass or the forces acting on it. A serial-link manipulator comprises a chain of mechanical links and joints. Each joint can move its outward neighbouring link with respect to its inward neighbour. One end of the chain, the base, is generally fixed and the other end is free to move in space and holds the tool or end-effector.

Figure 7.1 shows two classical robots that are the precursor to all arm-type robots today. Each robot has six joints and clearly the pose of the end-effector will be a complex function of the state of each joint. Section 7.1 describes a notation for describing the link and joint structure of a robot and Sect. 7.2 discusses how to compute the pose of the end-effector. Section 7.3 discusses the inverse problem, how to compute the position of each joint given the end-effector pose. Section 7.4 describes methods for generating smooth paths for the end-effector. The remainder of the chapter covers advanced topics and two complex applications: writing on a plane surface and a four-legged walking robot.

## 7.1 Describing a Robot Arm

A serial-link manipulator comprises a set of bodies, called links, in a chain and connected by joints. Each joint has one degree of freedom, either translational (a sliding or prismatic joint) or rotational (a revolute joint). Motion of the joint changes the relative angle or position of its neighbouring links. The joints of most common robot are revolute but the Stanford arm shown in Fig. 7.1b has one prismatic joint.

The joint structure of a robot can be described by a string such as "RRRRRR" for the Puma and "RRP RRR" for the Stanford arm, where the  $j^{\text{th}}$  character represents the type of joint  $j$ , either Revolute or Prismatic. A systematic way of describing the geometry of a serial chain of links and joints was proposed by Denavit and Hartenberg in 1955 and is known today as Denavit-Hartenberg notation.

For a manipulator with  $N$  joints numbered from 1 to  $N$ , there are  $N + 1$  links, numbered from 0 to  $N$ . Link 0 is the base of the manipulator and link  $N$  carries the end-

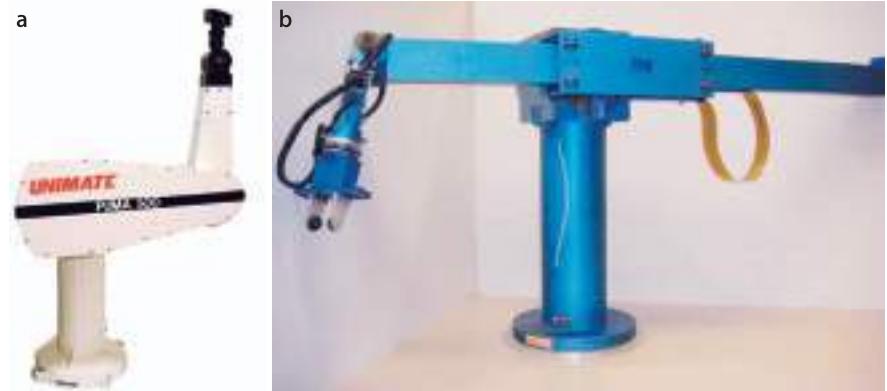


Fig. 7.1.

- a The Puma 560 robot was the first modern industrial robot (courtesy Oussama Khatib).
- b The Stanford arm was an early research arm and is unusual in that it has a prismatic joint (Stanford University AI Lab 1972; courtesy Oussama Khatib). Both arms were designed by robotics pioneer Victor Scheinman and both robots can be seen in the Smithsonian Museum of American History, Washington DC

Jacques Denavit and Richard Hartenberg introduced many of the key concepts of kinematics for serial-link manipulators in a 1955 paper (Denavit and Hartenberg 1955) and their later classic text *Kinematic Synthesis of Linkages* (Hartenberg and Denavit 1964).

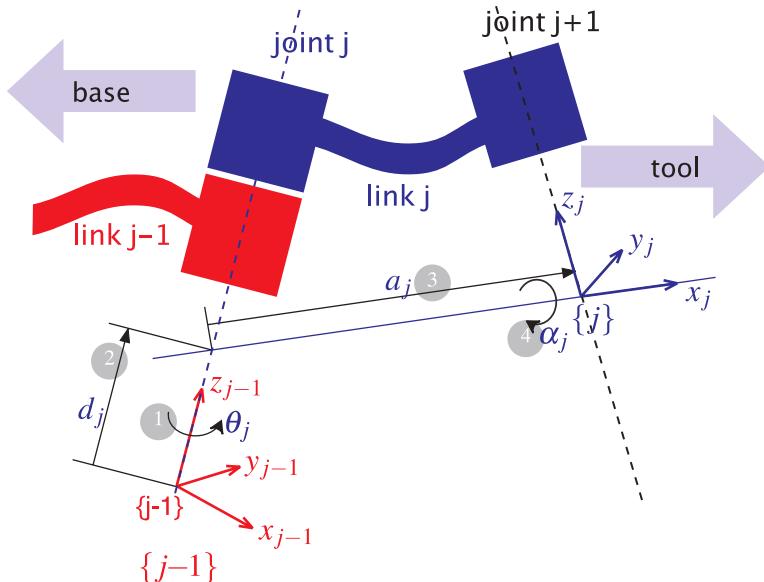


Fig. 7.2.

Definition of standard Denavit and Hartenberg link parameters. The colors red and blue denote all things associated with links  $j - 1$  and  $j$  respectively. The numbers in circles represent the order in which the elementary transforms are applied

effector or tool. Joint  $j$  connects link  $j - 1$  to link  $j$  and therefore joint  $j$  moves link  $j$ . A link is considered a rigid body that defines the spatial relationship between two neighbouring joint axes. A link can be specified by two parameters, its length  $a_j$  and its twist  $\alpha_j$ . Joints are also described by two parameters. The link offset  $d_j$  is the distance from one link coordinate frame to the next along the axis of the joint. The joint angle  $\theta_j$  is the rotation of one link with respect to the next about the joint axis.

Figure 7.2 illustrates this notation. The coordinate frame  $\{j\}$  is attached to the far (distal) end of link  $j$ . The axis of joint  $j$  is aligned with the  $z$ -axis. These link and joint parameters are known as Denavit-Hartenberg parameters and are summarized in Table 7.1.

Following this convention the first joint, joint 1, connects link 0 to link 1. Link 0 is the base of the robot. Commonly for the first link  $d_1 = \alpha_1 = 0$  but we could set  $d_1 > 0$  to represent the height of the shoulder joint above the base. In a manufacturing system the base is usually fixed to the environment but it could be mounted on a mobile base such as a space shuttle, an underwater robot or a truck.

The final joint, joint  $N$  connects link  $N - 1$  to link  $N$ . Link  $N$  is the tool of the robot and the parameters  $d_N$  and  $a_N$  specify the length of the tool and its  $x$ -axis offset respectively. The tool is generally considered to be pointed along the  $z$ -axis as shown in Fig. 2.14.

The transformation from link coordinate frame  $\{j - 1\}$  to frame  $\{j\}$  is defined in terms of elementary rotations<sup>►</sup> and translations as

$${}^{j-1}A_j(\theta_j, d_j, a_j, \alpha_j) = T_{Rz}(\theta_j)T_z(d_j)T_x(a_j)T_{Rx}(\alpha_j) \quad (7.1)$$

which can be expanded as

$${}^{j-1}A_j = \begin{pmatrix} \cos\theta_j & -\sin\theta_j \cos\alpha_j & \sin\theta_j \sin\alpha_j & a_j \cos\theta_j \\ \sin\theta_j & \cos\theta_j \cos\alpha_j & -\cos\theta_j \sin\alpha_j & a_j \sin\theta_j \\ 0 & \sin\alpha_j & \cos\alpha_j & d_j \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (7.2)$$

The  $3 \times 3$  orthonormal matrix is augmented with a zero translational component to form a  $4 \times 4$  homogenous transformation.

**Table 7.1.**  
Denavit-Hartenberg parameters:  
their physical meaning, symbol  
and formal definition

Joint angle	$\theta_j$	the angle between the $x_{j-1}$ and $x_j$ axes about the $z_{j-1}$ axis	revolute joint variable
Link offset	$d_j$	the distance from the origin of frame $j-1$ to the $x_j$ axis along the $z_{j-1}$ axis	prismatic joint variable
Link length	$a_j$	the distance between the $z_{j-1}$ and $z_j$ axes along the $x_j$ axis; for intersecting axes is parallel to $\hat{z}_{j-1} \times \hat{z}_j$	constant
Link twist	$\alpha_j$	the angle from the $z_{j-1}$ axis to the $z_j$ axis about the $x_j$ axis	constant
Joint type	$\sigma_j$	$\sigma = 0$ for a revolute joint, $\sigma = 1$ for a prismatic joint	constant

Jacques Denavit (1930–) was born in Paris where he studied for his Bachelor degree before pursuing his masters and doctoral degrees in mechanical engineering at Northwestern University, Illinois. In 1958 he joined the Department of Mechanical Engineering and Astronautical Science at Northwestern where the collaboration with Hartenberg was formed. In addition to his interest in dynamics and kinematics Denavit was also interested in plasma physics and kinetics. After the publication of the book he moved to Lawrence Livermore National Lab, Livermore, California, where he undertook research on computer analysis of plasma physics problems. He retired in 1982.

The parameters  $\alpha_j$  and  $a_j$  are always constant. For a revolute joint  $\theta_j$  is the joint variable and  $d_j$  is constant, while for a prismatic joint  $d_j$  is variable,  $\theta_j$  is constant and  $\alpha_j = 0$ . In many of the formulations that follow we use generalized joint coordinates

$$q_j = \begin{cases} \theta_j & \sigma_j = 0, \text{ for a revolute joint} \\ d_j & \sigma_j = 1, \text{ for a prismatic joint} \end{cases}$$

This is the same concept as was introduced for mobile robots on page 65.

For an  $N$ -axis robot the generalized joint coordinates  $\mathbf{q} \in \mathcal{C}$  where  $\mathcal{C} \subset \mathbb{R}^N$  is called the joint space or configuration space. ▶ For the common case of an all-revolute robot  $\mathcal{C} \subset \mathbb{S}^N$  the joint coordinates are referred to as joint angles. The joint coordinates are also referred to as the *pose of the manipulator* which is different to the *pose of the end-effector* which is a Cartesian pose  $\xi \in SE(3)$ . The term *configuration* is shorthand for *kinematic configuration* which will be discussed in Sect. 7.3.1.

Within the Toolbox we represent a robot link with a [Link](#) object which is created by

```
>> L = Link([0, 0.1, 0.2, pi/2, 0])
L =
    theta=q, d=0.1, a=0.2, alpha=1.571 (R,stdDH)
```

where the elements of the input vector are given in the order  $\theta_k, d_j, a_j, \alpha_j$ . The optional fifth element  $\sigma_j$  indicates whether the joint is revolute ( $\sigma_i = 0$ ) or prismatic ( $\sigma_i = 1$ ). If not specified a revolute joint is assumed.

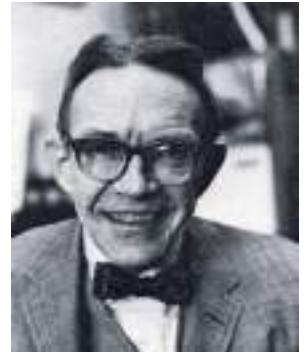
The displayed values of the [Link](#) object shows its kinematic parameters as well as other status such the fact that it is a revolute joint (the tag [R](#)) and that the standard Denavit-Hartenberg parameter convention is used (the tag [stdDH](#)). ▶

A slightly different notation, modified Denavit-Hartenberg notation is discussed in Sect. 7.5.3.

Although a value was given for  $\theta$  it is not displayed – that value simply served as a placeholder in the list of kinematic parameters.  $\theta$  is substituted by the joint variable  $q$  and its value in the [Link](#) object is ignored. The value will be managed by the [Link](#) object.

A [Link](#) object has many parameters and methods which are described in the online documentation, but the most common ones are illustrated by the following examples. The link transform Eq. 7.2 for  $q = 0.5$  rad is

Richard Hartenberg (1907–1997) was born in Chicago and studied for his degrees at the University of Wisconsin, Madison. He served in the merchant marine and studied aeronautics for two years at the University of Goettingen with space-flight pioneer Theodor von Karman. He was Professor of mechanical engineering at Northwestern University where he taught for 56 years. His research in kinematics led to a revival of interest in this field in the 1960s, and his efforts helped put kinematics on a scientific basis for use in computer applications in the analysis and design of complex mechanisms. He also wrote extensively on the history of mechanical engineering.



```
>> L.A(0.5)
ans =
0.8776   -0.0000    0.4794    0.1755
0.4794    0.0000   -0.8776    0.0959
  0       1.0000    0.0000    0.1000
  0           0        0     1.0000
```

is a  $4 \times 4$  homogeneous transformation. Various link parameters can be read or altered, for example

```
>> L.RP
ans =
R
```

indicates that the link is prismatic and

```
>> L.a
ans =
0.2000
```

returns the kinematic parameter  $a$ . Finally a link can contain an offset

```
>> L.offset = 0.5;
>> L.A(0)
ans =
0.8776   -0.0000    0.4794    0.1755
0.4794    0.0000   -0.8776    0.0959
  0       1.0000    0.0000    0.1000
  0           0        0     1.0000
```

which is added to the joint variable before computing the link transform and will be discussed in more detail in Sect. 7.5.1.

## 7.2 Forward Kinematics

The forward kinematics is often expressed in functional form

$$\xi_E = \mathcal{K}(q) \quad (7.3)$$

with the end-effector pose as a function of joint coordinates. Using homogeneous transformations this is simply the product of the individual link transformation matrices given by Eq. 7.2 which for an  $N$ -axis manipulator is

$$\xi_E \sim {}^0T_E = {}^0A_1 {}^1A_2 \cdots {}^{N-1}A_N \quad (7.4)$$

The forward kinematic solution can be computed for *any* serial-link manipulator irrespective of the number of joints or the types of joints. Determining the Denavit-Hartenberg parameters for each of the robot's links is described in more detail in Sect. 7.5.2.

The pose of the end-effector  $\xi_E \sim T_E \in SE(3)$  has six degrees of freedom – three in translation and three in rotation. Therefore robot manipulators commonly have six joints or degrees of freedom to allow them to achieve an arbitrary end-effector pose. The overall manipulator transform is frequently written as  $T_6$  for a 6-axis robot.

### 7.2.1 A 2-Link Robot

The first robot that we will discuss is the two-link planar manipulator shown in Fig. 7.3. It has the following Denavit-Hartenberg parameters which we use to create a vector of `Link` objects

Link	$\theta_i$	$d_i$	$a_i$	$\alpha_i$	$\sigma_i$
1	$q_1$	0	1	0	0
2	$q_2$	0	1	0	0

```
>> L(1) = Link([0 0 1 0]);
>> L(2) = Link([0 0 1 0]);
>> L
L =
theta=q1, d=0, a=1, alpha=0 (R,stdDH)
theta=q2, d=0, a=1, alpha=0 (R,stdDH)
```

which are passed to the constructor `SerialLink`

```
>> two_link = SerialLink(L, 'name', 'two link');
```

which returns a `SerialLink` object that we can display

```
>> two_link
two_link =
two link (2 axis, RR, stdDH)
+-----+-----+-----+-----+
| j | theta | d | a | alpha |
+-----+-----+-----+-----+
| 1 | q1 | 0 | 1 | 0 |
| 2 | q2 | 0 | 1 | 0 |
+-----+-----+-----+-----+
grav =      0   base = 1 0 0 0   tool =  1 0 0 0
            0       0 1 0 0       0 1 0 0
            9.81     0 0 1 0       0 0 1 0
                  0 0 0 1       0 0 0 1
```

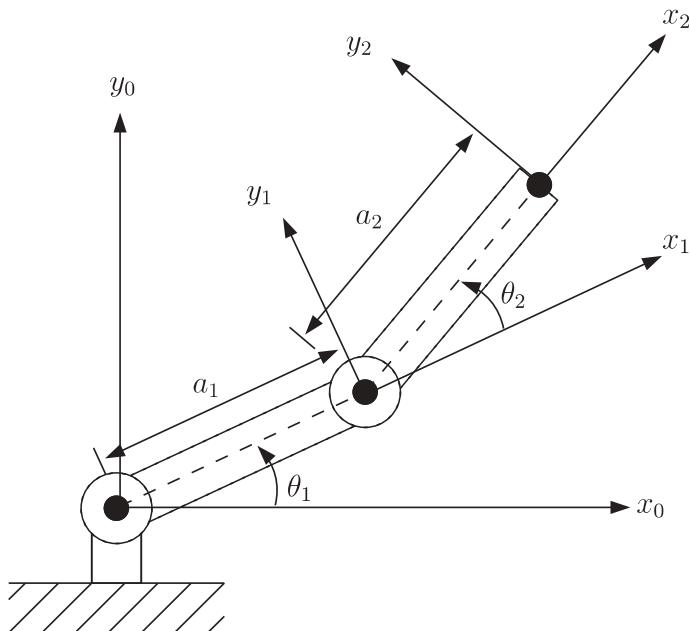


Fig. 7.3.  
Two-link robot as per  
Spong et al. (2006, fig. 3.6, p. 84)

This provides a concise description of the robot. We see that it has 2 revolute joints as indicated by the structure string '`'RR'`', it is defined in terms of standard Denavit-Hartenberg parameters, that gravity is acting in the default  $z$ -direction.► The kinematic parameters of the link objects are also listed and the joint variables are shown as variables `q1` and `q2`. We have also assigned a name to the robot which will be shown whenever the robot is displayed graphically. The script

```
>> mdl_twolink
```

performs the above steps and defines this robot in the MATLAB® workspace, creating a `SerialLink` object named `twolink`.

The `SerialLink` object is key to the operation of the Robotics Toolbox. It has a great many methods and properties which are illustrated in the rest of this part, and described in detail in the online documentation. Some simple examples are

```
>> twolink.n
ans =
2
```

which returns the number of joints, and

```
>> links = twolink.links
L =
theta=q1, d=0, a=1, alpha=0 (R,stdDH)
theta=q2, d=0, a=1, alpha=0 (R,stdDH)
```

which returns a vector of `Link` objects comprising the robot.► We can also make a copy of a `SerialLink` object

```
>> clone = SerialLink(twolink, 'name', 'bob')
clone =
bob (2 axis, RR, stdDH)
+-----+-----+-----+-----+
| j | theta | d | a | alpha |
+-----+-----+-----+-----+
| 1 | q1 | 0 | 1 | 0 |
| 2 | q2 | 0 | 1 | 0 |
+-----+-----+-----+-----+
grav = 0 base = 1 0 0 0 tool = 1 0 0 0
      0 0 1 0 0 1 0 0
      9.81 0 0 1 0 0 1 0
                  0 0 0 1 0 0 0 1
```

which has the name '`bob`'.►

Now we can put the robot arm to work. The forward kinematics are computed using the `fkine` method. For the case where  $q_1 = q_2 = 0$

```
>> twolink.fkine([0 0])
ans =
1 0 0 2
0 1 0 0
0 0 1 0
0 0 0 1
```

the method returns the homogenous transform that represents the pose of the second link coordinate frame of the robot,  $T_2$ . For a different configuration the tool pose is

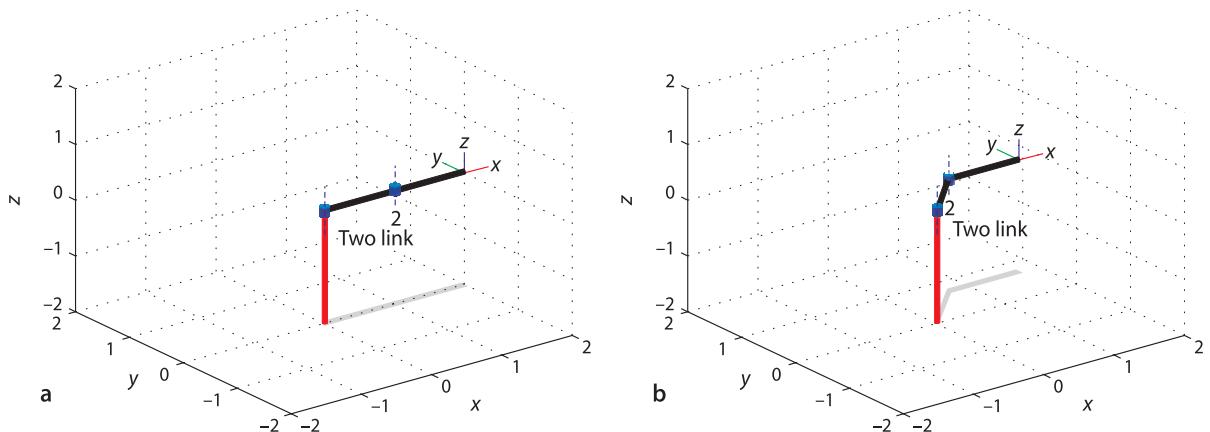
```
>> twolink.fkine([pi/4 -pi/4])
ans =
1.0000 0 0 1.7071
0 1.0000 0 0.7071
0 0 1.0000 0
0 0 0 1.0000
```

Normal to the plane in which the robot moves.

`Link` objects are derived from the MATLAB® `handle` class and can be set *in place*. Thus we can write code like `twolink.links(1).a = 2` to change the kinematic parameter  $a_2$ .

A unique name is required when multiple robots are displayed graphically, see Sect. 7.8 for more details.

By convention, joint coordinates with the Toolbox are row vectors.



**Fig. 7.4.** The two-link robot in two different poses, **a** the pose  $(0, 0)$ ; **b** the pose  $(\frac{\pi}{4}, -\frac{\pi}{4})$ . Note the graphical details. Revolute joints are indicated by cylinders and the joint axes are shown as line segments. The final-link coordinate frame and a shadow on the ground are also shown

The robot can be visualized graphically

```
>> twolink.plot([0 0])
>> twolink.plot([pi/4 -pi/4])
```

as stick figures shown in Fig. 7.4. The graphical representation includes the robot's name, the final-link coordinate frame,  $T_2$  in this case, the joints and their axes, and a shadow on the ground plane. Additional features of the `plot` method such as multiple views and multiple robots are described in Sect. 7.8 with additional details in the online documentation.

The simple two-link robot introduced above is limited in the poses that it can achieve since it operates entirely within the  $xy$ -plane, its task space is  $\mathcal{T} \subset \mathbb{R}^2$ .

## 7.2.2 A 6-Axis Robot

Truly useful robots have a task space  $\mathcal{T} \subset SE(3)$  enabling arbitrary position and attitude of the end-effector – the task space has six spatial degrees of freedom: three translational and three rotational. This requires a robot with a configuration space  $\mathcal{C} \subset \mathbb{R}^6$  which can be achieved by a robot with six joints. In this section we will use the Puma 560 as an example of the class of all-revolute six-axis robot manipulators. We define an instance of a Puma 560 robot using the script

```
>> mdl_puma560
```

which creates a `SerialLink` object, `p560`, in the workspace

```
>> p560
p560 =
Puma 560 (6 axis, RRRRRR, stdDH)
  Animation; viscous friction; params of 8/95;
+-----+-----+-----+-----+-----+
| j |     theta |         d |         a |       alpha |
+-----+-----+-----+-----+-----+
| 1 |      q1 |         0 |         0 |    1.571 |
| 2 |      q2 |         0 |    0.4318 |        0 |
| 3 |      q3 |      0.15 |    0.0203 |   -1.571 |
| 4 |      q4 |    0.4318 |         0 |    1.571 |
| 5 |      q5 |         0 |         0 |   -1.571 |
| 6 |      q6 |         0 |         0 |        0 |
+-----+-----+-----+-----+-----+
grav =    0  base = 1  0  0  0  tool =  1  0  0  0
          0           0  1  0  0           0  1  0  0
          9.81         0  0  1  0           0  0  1  0
          0  0  0  1           0  0  0  1
```

The Toolbox has scripts to define a number of common industrial robots including the Motoman HP6, Fanuc 10L, ABB S4 2.8 and the Stanford arm.

The Puma 560 robot (Programmable Universal Manipulator for Assembly) shown in Fig. 7.1 was released in 1978 and became enormously popular. It featured an anthropomorphic design, electric motors and a spherical wrist which makes it the first *modern* industrial robot – the archetype of all that followed.

The Puma 560 catalyzed robotics research in the 1980s and it was a very common laboratory robot. Today it is obsolete and rare but in homage to its important role in robotics research we use it here. For our purposes the advantages of this robot are that it has been well studied and its parameters are very well known – it has been described as the “white rat” of robotics research.

Most modern 6-axis industrial robots are very similar in structure and can be accommodated simply by changing the Denavit-Hartenberg parameters. The Toolbox has kinematic models for a number of common industrial robots including the Motoman HP6, Fanuc 10L, and ABB S4 2.8.

Note that  $a_j$  and  $d_j$  are in SI units which means that the translational part of the forward kinematics will also have SI units.

The script `mdl_puma560` also creates a number of joint coordinate vectors in the workspace which represent the robot in some canonic configurations:

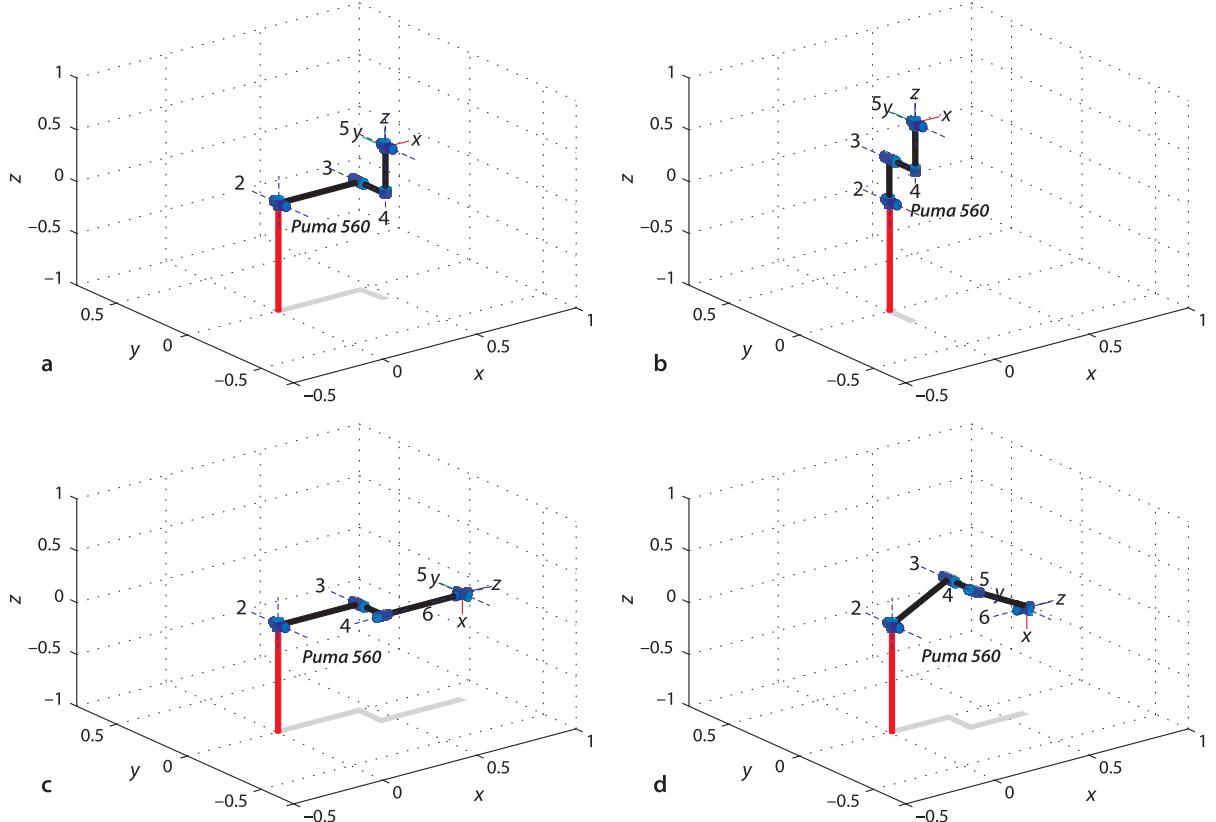
<code>qz</code>	$(0, 0, 0, 0, 0, 0)$	<i>zero angle</i>
<code>qr</code>	$(0, \frac{\pi}{2}, -\frac{\pi}{2}, 0, 0, 0)$	<i>ready</i> , the arm is straight and vertical
<code>qs</code>	$(0, 0, -\frac{\pi}{2}, 0, 0, 0)$	<i>stretch</i> , the arm is straight and horizontal
<code>qn</code>	$(0, \frac{\pi}{4}, -\pi, 0, \frac{\pi}{4}, 0)$	<i>nominal</i> , the arm is in a dexterous working pose▶

Well away from singularities, which will be discussed in Sect. 7.4.3.

and these are shown graphically in Fig. 7.5. These plots are generated using the `plot` method, for example

```
>> p560.plot(qz)
```

**Fig. 7.5.** The Puma robot in 4 different poses. **a** Zero angle; **b** ready pose; **c** stretch; **d** nominal



**Anthropomorphic** means having human-like characteristics. The Puma 560 robot was designed to have approximately the dimensions and reach of a human worker. It also had a spherical joint at the wrist just as humans have.

Roboticians also tend to use anthropomorphic terms when describing robots. We use words like waist, shoulder, elbow and wrist when describing serial link manipulators. For the Puma these terms correspond respectively to joint 1, 2, 3 and 4–6.

Forward kinematics can be computed as before

```
>> p560.fkine(qz)
ans =
    1.0000      0      0    0.4521
        0    1.0000      0   -0.1500
        0      0    1.0000    0.4318
        0      0      0    1.0000
```

By the `mdl_puma560` script.

which returns the homogeneous transformation corresponding to the end-effector pose  $T_6$ . The origin of this frame, the link 6 coordinate frame {6}, is defined<sup>◀</sup> as the point of intersection of the axes of the last 3 joints – physically this point is inside the robot's wrist mechanism. We can define a tool transform, from the  $T_6$  frame to the actual tool tip by

```
>> p560.tool = transl(0, 0, 0.2);
```

in this case a 200 mm extension in the  $T_6$  z-direction.<sup>◀</sup> The pose of the tool tip, often referred to as the tool centre point or TCP, is now

```
>> p560.fkine(qz)
ans =
    1.0000      0      0    0.4521
        0    1.0000      0   -0.1500
        0      0    1.0000    0.6318
        0      0      0    1.0000
```

The kinematic definition we have used considers that the base of the robot is the intersection point of the waist and shoulder axes which is a point inside the structure of the robot. The Puma 560 robot includes a 30 inch tall pedestal. We can shift the origin of the robot from the point inside the robot to the base of the pedestal using a base transform

```
>> p560.base = transl(0, 0, 30*0.0254);
```

where for consistency we have converted the pedestal height to SI units. Now, with both base and tool transform, the forward kinematics are

```
>> p560.fkine(qz)
ans =
    1.0000      0      0    0.4521
        0    1.0000      0   -0.1500
        0      0    1.0000    1.3938
        0      0      0    1.0000
```

and we can see that the z-coordinate of the tool is now greater than before.

We can also do more interesting things, for example

```
>> p560.base = transl(0,0,3) * trotx(pi);
>> p560.fkine(qz)
ans =
    1.0000      0      0    0.4521
        0   -1.0000   -0.0000    0.1500
        0    0.0000   -1.0000    2.3682
        0      0      0    1.0000
```

which positions the robot's origin 3 m above the world origin with its coordinate frame rotated by 180° about the x-axis. This robot is now hanging from the ceiling!

The Toolbox supports joint angle time series or trajectories such as

```
>> q
q =
    0      0      0      0      0      0
    0  0.0365 -0.0365    0      0      0
    0  0.2273 -0.2273    0      0      0
    0  0.5779 -0.5779    0      0      0
    0  0.9929 -0.9929    0      0      0
    0  1.3435 -1.3435    0      0      0
    0  1.5343 -1.5343    0      0      0
    0  1.5708 -1.5708    0      0      0
```

where each row represents the joint coordinates at a different timestep and the columns represent the joints.► In this case the method `fkine`

Generated by the `jtraj` function,  
which is discussed in Sect. 7.4.1.

```
>> T = p560.fkine(q);
```

returns a 3-dimensional matrix

```
>> about(T)
T [double] : 4x4x8 (1024 bytes)
```

where the first two dimensions are a  $4 \times 4$  homogeneous transformation and the third dimension is the timestep. The homogeneous transform corresponding to the joint coordinates in the fourth row of `q` is

```
>> T(:,:,4)
ans =
    1.0000    0.0000      0    0.3820
    0.0000    1.0000   -0.0000   -0.1500
        0   -0.0000    1.0000    1.6297
        0        0        0    1.0000
```

Creating a trajectory is the topic of Sect. 7.4.

### 7.3 Inverse Kinematics

We have shown how to determine the pose of the end-effector given the joint coordinates and optional tool and base transforms. A problem of real practical interest is the inverse problem: given the desired pose of the end-effector  $\xi_E$  what are the required joint coordinates? For example, if we know the Cartesian pose of an object, what joint coordinates does the robot need in order to reach it? This is the inverse kinematics problem which is written in functional form as

$$\mathbf{q} = \mathcal{K}^{-1}(\xi) \quad (7.5)$$

In general this function is not unique and for some classes of manipulator no closed-form solution exists, necessitating a numerical solution.

#### 7.3.1 Closed-Form Solution

A necessary condition for a closed-form solution of a 6-axis robot is that the three wrist axes intersect at a single point. This means that motion of the wrist joints only changes the end-effector orientation, not its translation. Such a mechanism is known as a spherical wrist and almost all industrial robots have such wrists.

We will explore inverse kinematics using the Puma robot model

```
>> mdl_puma560
```

At the *nominal* joint coordinates

```
>> qn
qn =
    0    0.7854    3.1416    0    0.7854    0
```

the end-effector pose is

```
>> T = p560.fkine(qn)
T =
    -0.0000    0.0000    1.0000    0.5963
    -0.0000    1.0000   -0.0000   -0.1501
    -1.0000   -0.0000   -0.0000   -0.0144
        0         0         0       1.0000
```

The method `ikine6s` checks the Denavit-Hartenberg parameters to determine if the robot meets these criteria.

Since the Puma 560 is a 6-axis robot arm with a spherical wrist we use the method `ikine6s` to compute the inverse kinematics using a closed-form solution. The required joint coordinates to achieve the pose  $T$  are

```
>> qi = p560.ikine6s(T)
qi =
    2.6486   -3.9270    0.0940    2.5326    0.9743    0.3734
```

Surprisingly, these are quite different to the joint coordinates we started with. However if we investigate a little further

```
>> p560.fkine(qi)
ans =
    -0.0000    0.0000    1.0000    0.5963
     0.0000    1.0000   -0.0000   -0.1500
    -1.0000    0.0000   -0.0000   -0.0144
        0         0         0       1.0000
```

we see that these two different sets of joint coordinates result in the *same* end-effector pose and these are shown in Fig. 7.6. The shoulder of the Puma robot is horizontally offset from the waist, so in one solution the arm is to the left of the waist, in the other it is to the right. These are referred to as the left- and right-handed kinematic configurations respectively. In general there are eight sets of joint coordinates that give the same end-effector pose – as mentioned earlier the inverse solution is not unique.

We can force the right-handed solution

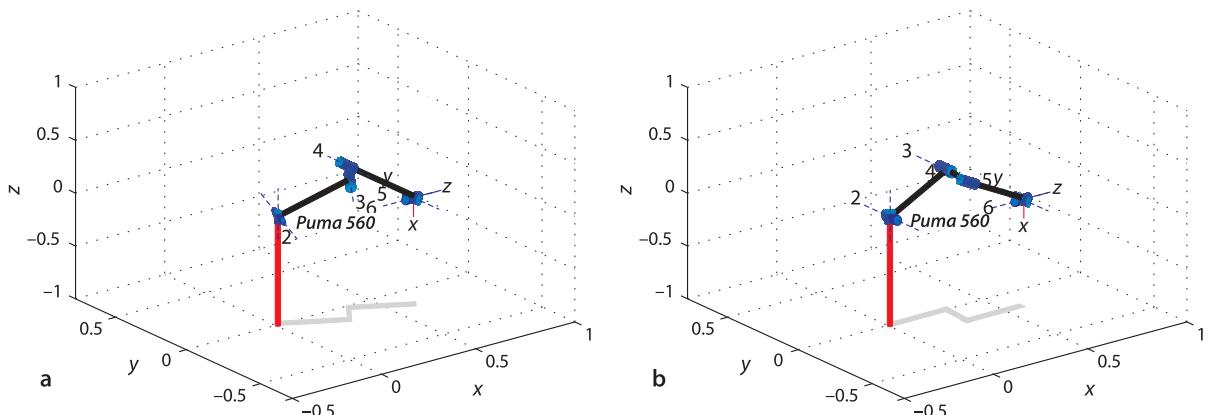
```
>> qi = p560.ikine6s(T, 'ru')
qi =
    -0.0000    0.7854    3.1416    0.0000    0.7854   -0.0000
```

which gives the original set of joint angles by specifying a *right handed* configuration with the elbow up.

In addition to the left- and right-handed solutions, there are solutions with the elbow either up or down, and with the wrist flipped or not flipped. For the Puma 560 robot the wrist joint,  $\theta_4$ , has a large rotational range and can adopt one of two angles

More precisely the elbow is above or below the shoulder.

**Fig. 7.6.** Two solutions to the inverse kinematic problem, left-handed and right-handed solutions. The shadow shows clearly the two different configurations



that differ by  $\pi$  rad. Since most robot grippers have just two fingers, see Fig. 2.14, this makes no difference in its ability to grasp an object.

The various kinematic configurations are shown in Fig. 7.7. The kinematic configuration returned by `ikine6s` is controlled by one or more of the character flags:

left or right handed	'l', 'r'
elbow up or down	'u', 'd'
wrist flipped or not flipped	'f', 'n'

Due to mechanical limits on joint angles and possible collisions between links not all eight solutions are physically achievable. It is also possible that no solution can be achieved, for example

```
>> p560.ikine6s( transl(3, 0, 0) )
Warning: point not reachable
ans =
NaN   NaN   NaN   NaN   NaN   NaN
```

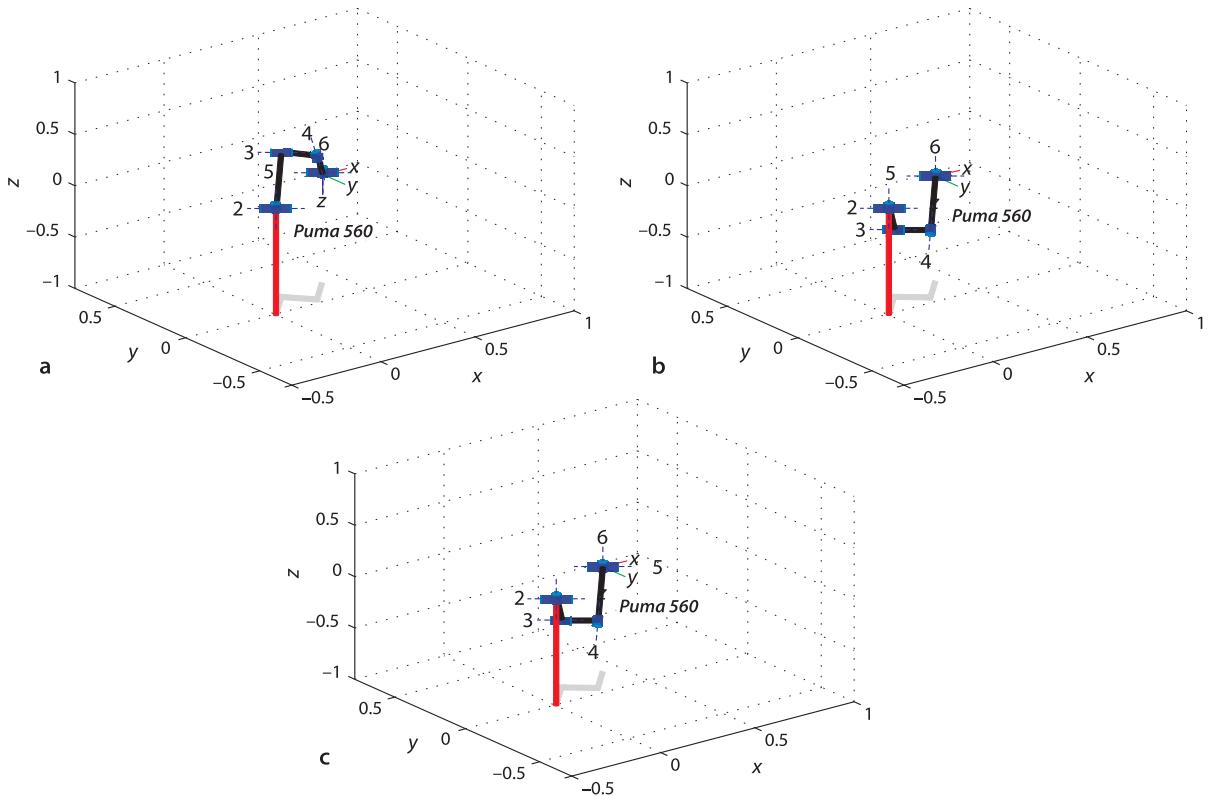
fails because the arm is simply not long enough to reach this pose. A pose may also be unachievable due to singularity where the alignment of axes reduces the effective degrees of freedom (the gimbal lock problem again). The Puma 560 has a singularity when  $q_5$  is equal to zero and the axes of joints 4 and 6 become aligned. In this case the best that `ikine6s` can do is to constrain  $q_4 + q_6$  but their individual values are arbitrary. For example consider the configuration

```
>> q = [0 pi/4 pi 0.1 0 0.2];
```

for which  $q_4 + q_6 = 0.3$ . The inverse kinematic solution is

```
>> p560.ikine6s(p560.fkine(q), 'ru')
ans =
-0.0000    0.7854    3.1416    2.9956    0.0000   -2.6956
```

Fig. 7.7. Different configurations of the Puma 560 robot. **a** Right-up-noflip; **b** right-down-noflip; **c** right-down-flip



has quite different values for  $q_4$  and  $q_6$  but their sum

```
>> q(4)+q(6)
ans =
0.3000
```

remains the same.

### 7.3.2 Numerical Solution

For the case of robots which do not have six joints and a spherical wrist we need to use an iterative numerical solution. Continuing with the example of the previous section we use the method `ikine` to compute the general inverse kinematic solution

```
>> T = p560.fkine(qn)
ans =
-0.0000    0.0000    1.0000    0.5963
-0.0000    1.0000   -0.0000   -0.1501
-1.0000   -0.0000   -0.0000   -0.0144
     0         0         0     1.0000
>> qi = p560.ikine(T)
qi =
  0.0000   -0.8335    0.0940   -0.0000   -0.8312    0.0000
```

which is different to the original value

```
>> qn
qn =
  0    0.7854    3.1416        0    0.7854        0
```

but does result in the correct tool pose

```
>> p560.fkine(qi)
ans =
-0.0000    0.0000    1.0000    0.5963
-0.0000    1.0000   -0.0000   -0.1501
-1.0000   -0.0000   -0.0000   -0.0144
     0         0         0     1.0000
```

Plotting the pose

```
>> p560.plot(qi)
```

shows clearly that `ikine` has found the elbow-down configuration.

A limitation of this general numeric approach is that it does not provide explicit control over the arm's kinematic configuration as did the analytic approach – the only control is implicit via the initial value of joint coordinates (which defaults to zero). If we specify the initial joint coordinates

```
>> qi = p560.ikine(T, [0 0 3 0 0 0])
qi =
  0.0000    0.7854    3.1416    0.0000    0.7854   -0.0000
```

the solution converges on the elbow-up configuration.◀

As would be expected the general numerical approach of `ikine` is considerably slower than the analytic approach of `ikine6s`. However it has the great advantage of being able to work with manipulators at singularities and manipulators with less than six or more than six joints. Details of the principle behind `ikine` is provided in Sect. 8.4.

### 7.3.3 Under-Actuated Manipulator

An under-actuated manipulator is one that has fewer than six joints, and is therefore limited in the end-effector poses that it can attain. SCARA robots such as shown on page 135 are a common example, they have typically have an  $x$ - $y$ - $z$ - $\theta$  task space,  $\mathcal{T} \subset \mathbb{R}^3 \times \mathbb{S}$  and a configuration space  $\mathcal{C} \subset \mathbb{S}^3 \times \mathbb{R}$ .

When solving for a trajectory as on p. 146 the inverse kinematic solution for one point is used to initialize the solution for the next point on the path.

We will revisit the two-link manipulator example from Sect. 7.2.1, first defining the robot

```
>> mdl_twolink
```

and then defining the desired end-effector pose

```
>> T = transl(0.4, 0.5, 0.6);
```

However this *pose* is over-constrained for the two-link robot – the tool cannot meet the orientation constraint on the direction of  $\hat{x}_2$  and  $\hat{y}_2$  nor a  $z$ -axis translational value other than zero. Therefore we require the `ikine` method to consider only error in the  $x$ - and  $y$ -axis translational directions, and ignore all other Cartesian degrees of freedom. We achieve this by specifying a mask vector as the fourth argument

```
>> q = twolink.ikine(T, [0 0], [1 1 0 0 0 0])
q =
-0.3488    2.4898
```

The elements of the mask vector correspond respectively to the three translations and three orientations:  $t_x, t_y, t_z, r_x, r_y, r_z$  in the end-effector coordinate frame. In this example we specified that only errors in  $x$ - and  $y$ -translation are to be considered (the non-zero elements). The resulting joint angles correspond to an endpoint pose

```
>> twolink.fkine(q)
ans =
ans =
-0.5398    -0.8418         0    0.4000
  0.8418    -0.5398         0    0.5000
    0         0    1.0000         0
    0         0         0    1.0000
```

which has the desired  $x$ - and  $y$ -translation, but the orientation and  $z$ -translation are incorrect, as *we allowed them to be*.

### 7.3.4 Redundant Manipulator

A redundant manipulator is a robot with more than six joints. As mentioned previously, six joints is theoretically sufficient to achieve any desired pose in a Cartesian taskspace  $\mathcal{T} \subset SE(3)$ . However practical issues such as joint limits and singularities mean that not all poses within the robot's reachable space can be achieved. Adding additional joints is one way to overcome this problem.

To illustrate this we will create a redundant manipulator. We place our familiar Puma robot

```
>> mdl_puma560
```

on a platform that moves in the  $xy$ -plane, mimicking a robot mounted on a vehicle. This robot has the same task space  $\mathcal{T} \subset SE(3)$  as the Puma robot, but a configuration space  $\mathcal{C} \subset \mathbb{R}^2 \times \mathbb{S}^6$ . The dimension of the configuration space exceeds the dimensions of the task space.

The Denavit-Hartenberg parameters for the base are

Link	$\theta_i$	$d_i$	$a_i$	$\alpha_i$	$\sigma_i$
1	0	$q_1$	0	$-\frac{\pi}{2}$	1
2	$-\frac{\pi}{2}$	$q_2$	0	$\frac{\pi}{2}$	1

and using a shorthand syntax we create the `SerialLink` object directly from the Denavit-Hartenberg parameters

```
>> platform = SerialLink([0 0 0 -pi/2 1; -pi/2 0 0 pi/2 1], ...
    'base', trot(y(pi/2), 'name', 'platform'));
```

The Denavit-Hartenberg notation requires that prismatic joints cause translation in the local  $z$ -direction and the base-transform rotates that  $z$ -axis into the world  $x$ -axis direction. This is a common complexity with Denavit-Hartenberg notation and an alternative will be introduced in Sect. 7.5.2.

The joint coordinates of this robot are its  $x$ - and  $y$ -position and we test that our Denavit-Hartenberg parameters are correct

```
>> platform.fkine([1, 2])
ans =
    1.0000    0.0000    0.0000    1.0000
   -0.0000    1.0000    0.0000    2.0000
     0    -0.0000    1.0000    0.0000
     0        0        0    1.0000
```

We see that the rotation submatrix is the identity matrix, meaning that the coordinate frame of the platform is parallel with the world coordinate frame, and that the  $x$ - and  $y$ -displacement is as requested.

We mount the Puma robot on the platform by *connecting* the two robots in series. In the Toolbox we can express this in two different ways, by multiplication

```
>> p8 = platform * p560;
```

or by concatenation

```
p8 = SerialLink( [platform, p560]);
```

which also allows other attributes of the created `SerialLink` object to be specified.

However there is a small complication. We would like the Puma arm to be sitting on its tall pedestal on top of the platform. Previously we added the pedestal as a base transform, but base transforms can only exist at the start of a kinematic chain, and we require the transform between joints 2 and 3 of the 8-axis robot. Instead we implement the pedestal by setting  $d_1$  of the Puma to the pedestal height

```
>> p560.links(1).d = 30 * 0.0254;
```

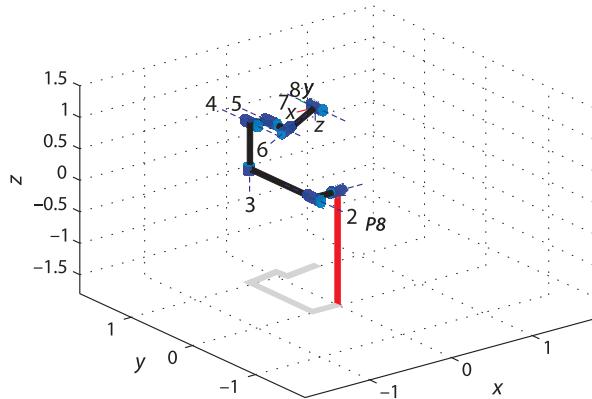
and now we can compound the two robots

```
>> p8 = SerialLink( [platform, p560], 'name', 'P8')
p8 =

P8 (8 axis, PPRRRRR, stdDH)
+-----+-----+-----+-----+
| j | theta | d | a | alpha |
+-----+-----+-----+-----+
| 1 |      0 | q1 | 0 | -1.571 |
| 2 | -1.571 | q2 | 0 | 1.571 |
| 3 |      q3 | 0.762 | 0 | 1.571 |
| 4 |      q4 | 0 | 0.4318 | 0 |
| 5 |      q5 | 0.15 | 0.0203 | -1.571 |
| 6 |      q6 | 0.4318 | 0 | 1.571 |
| 7 |      q7 | 0 | 0 | -1.571 |
| 8 |      q8 | 0 | 0 | 0 |
+-----+-----+-----+-----+
grav = 0 base = 0 0 0 0 tool = 1 0 0 0
       0      0 1 0 0      0 1 0 0
       9.81    1 0 0 0      0 0 1 0
                   0 0 0 1      0 0 0 1
```

resulting in a new 8-axis robot which we have named the P8. We note that the new robot has inherited the base transform of the platform, and that the pedestal displacement appears now as  $d_3$ .

We can find the inverse kinematics for this new redundant manipulator using the general numerical solution. For example to move the end effector to (0.5, 1.0, 0.7) with the tool pointing downward in the  $xz$ -plane the Cartesian pose is



**Fig. 7.8.**  
Redundant manipulator P8

```
>> T = transl(0.5, 1.0, 0.7) * rpy2tr(0, 3*pi/4, 0);
```

The required joint coordinates are

```
>> qi = p8.ikine(T)
qi =
-0.3032    1.0168    0.1669   -0.4908   -0.6995   -0.1276
-1.1758    0.1679
```

The first two elements are displacements of the prismatic joints in the base, and the last six are the robot arm's joint angles. We see that the solution has made good use of the second joint, the platform's  $y$ -axis translation to move the base of the arm close to the desired point. We can visualize the configuration

```
>> p8.plot(qi)
```

which is shown in Fig. 7.8. We can also show that the plain old Puma cannot reach the point we specified

```
>> p560.ikine6s(T)
Warning: point not reachable
```

This robot has 8 degrees of freedom. Intuitively it is more dexterous than its 6-axis counterpart because there is more than one joint that can contribute to every Cartesian degree of freedom. This is more formally captured in the notion of manipulability which we will discuss in Sect. 8.1.4. For now we will consider it as a scalar measure that reflects the ease with which the manipulator's tool can move in different Cartesian directions. The manipulability of the 6-axis robot

```
>> p560.maniply(qn)
ans =
0.0786
```

is much less than that for the 8-axis robot

```
>> p8.maniply([0 0 qn])
ans =
1.1151
```

which indicates the increased dexterity of the 8-axis robot.

## 7.4 Trajectories

One of the most common requirements in robotics is to move the end-effector smoothly from pose A to pose B. Building on what we learnt in Sect. 3.1 we will discuss two approaches to generating such trajectories: straight lines in joint space and straight lines in Cartesian space. These are known respectively as joint-space and Cartesian motion.

### 7.4.1 Joint-Space Motion

We choose the tool z-axis downward on these poses as it would be if the robot was reaching down to work on some horizontal surface. For the Puma 560 robot it be physically impossible for the tool z-axis to point upward in such a situation.

Consider the end-effector moving between two Cartesian poses

```
>> T1 = transl(0.4, 0.2, 0) * trotx(pi);
>> T2 = transl(0.4, -0.2, 0) * trotx(pi/2);
```

which lie in the  $xy$ -plane with the end-effector oriented downward. The initial and final joint coordinate vectors associated with these poses are

```
>> q1 = p560.ikine6s(T1);
>> q2 = p560.ikine6s(T2);
```

and we require the motion to occur over a time period of 2 seconds in 50 ms time steps

```
>> t = [0:0.05:2]';
```

A joint-space trajectory is formed by smoothly interpolating between two configurations  $q_1$  and  $q_2$ . The scalar interpolation functions `tpoly` or `lspb` from Sect. 3.1 can be used in conjunction with the multi-axis *driver* function `mtraj`

```
>> q = mtraj(@tpoly, q1, q2, t);
```

or

```
>> q = mtraj(@lspb, q1, q2, t);
```

which each result in a  $50 \times 6$  matrix  $q$  with one row per time step and one column per joint. From here on we will use the `jtraj` convenience function

```
>> q = jtraj(q1, q2, t);
```

which is equivalent to `mtraj` with `tpoly` interpolation but optimized for the multi-axis case and also allowing initial and final velocity to be set using additional arguments. For `mtraj` and `jtraj` the final argument can be a time vector, as here, or an integer specifying the number of time steps.

We obtain the joint velocity and acceleration vectors, as a function of time, through optional output arguments

```
>> [q,qd,qdd] = jtraj(q1, q2, t);
```

An even more concise expression for the above steps is provided by the `jtraj` method of the `SerialLink` class

```
>> q = p560.jtraj(T1, T2, t)
```

The trajectory is best viewed as an animation

```
>> p560.plot(q)
```

but we can also plot the joint angle, for instance  $q_2$ , versus time

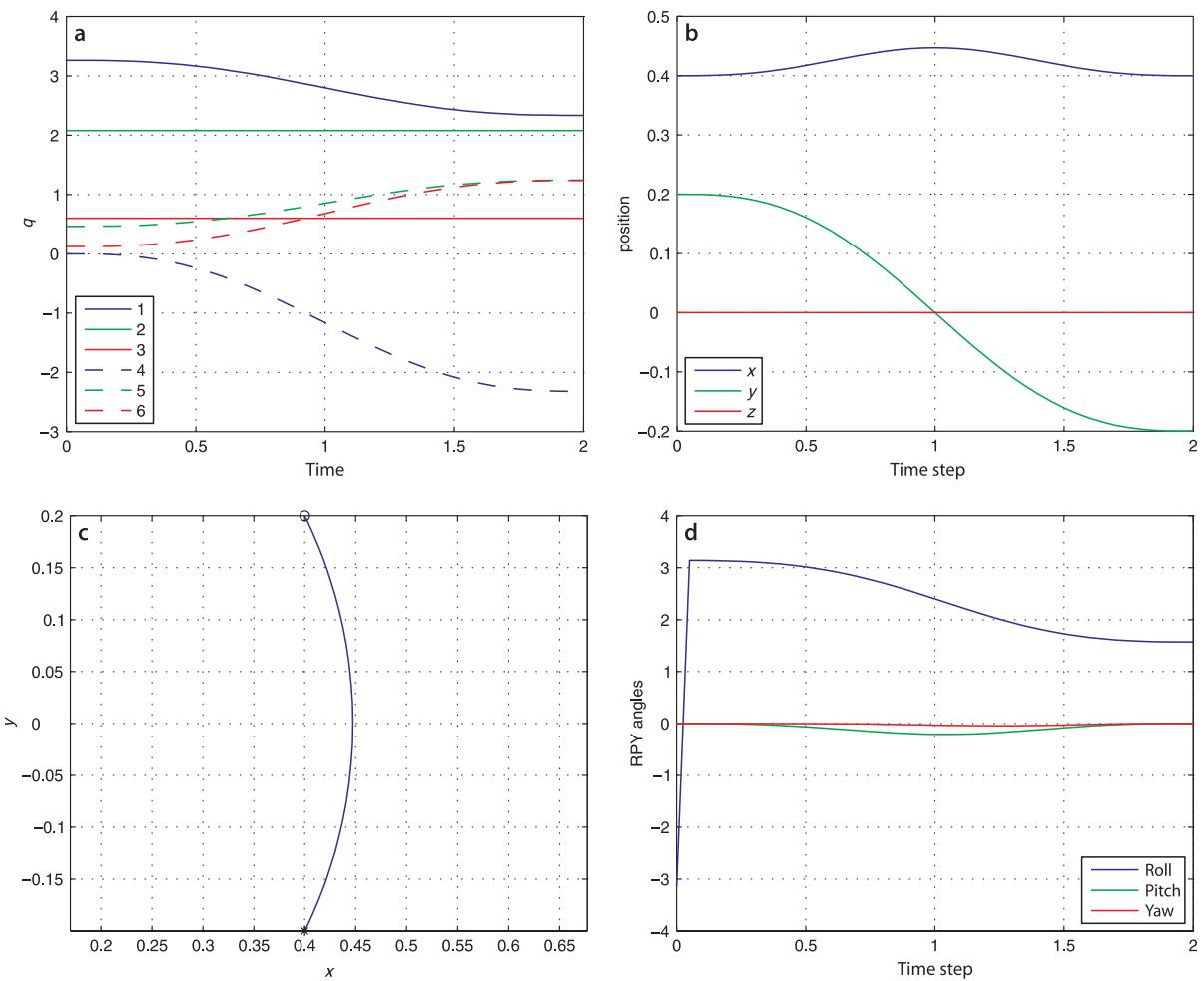
```
>> plot(t, q(:,2))
```

or all the angles versus time

```
>> qplot(t, q);
```

as shown in Fig. 7.9a. The joint coordinate trajectory is smooth but we do not know how the robot's end-effector will move in Cartesian space. However we can easily determine this by applying forward kinematics to the joint coordinate trajectory

```
>> T = p560.fkine(q);
```



which results in a 3-dimensional Cartesian trajectory. The translational part of this trajectory is

```
>> p = transl(T);
```

which is in matrix form

```
>> about(p)
p [double] : 41x3 (984 bytes)
```

and has one row per time step, and each row is the end-effector position vector. This is plotted against time in Fig. 7.9b. The path of the end-effector in the  $xy$ -plane

```
>> plot(p(:,1), p(:,2))
```

is shown in Fig. 7.9c and it is clear that the path is not a straight line. This is to be expected since we only specified the Cartesian coordinates of the end-points. As the robot rotates about its waist joint during the motion the end-effector will naturally follow a circular arc. In practice this could lead to collisions between the robot and nearby objects even if they do not lie on the path between poses A and B. The orientation of the end-effector, in roll-pitch-yaw angles form, can be plotted against time

```
>> plot(t, tr2rpy(T))
```

as shown in Fig. 7.9d. Note that the roll angle  $\alpha$  varies from  $\pi$  to  $\frac{\pi}{2}$  as we specified. However while the pitch angle has met its boundary conditions it has varied along

**Fig. 7.9.** Joint space motion. **a** Joint coordinates versus time; **b** Cartesian position versus time; **c** Cartesian position locus in the  $xy$ -plane **d** roll-pitch-yaw angles versus time

Rotation about  $x$ -axis from Sect. 2.2.1.2.

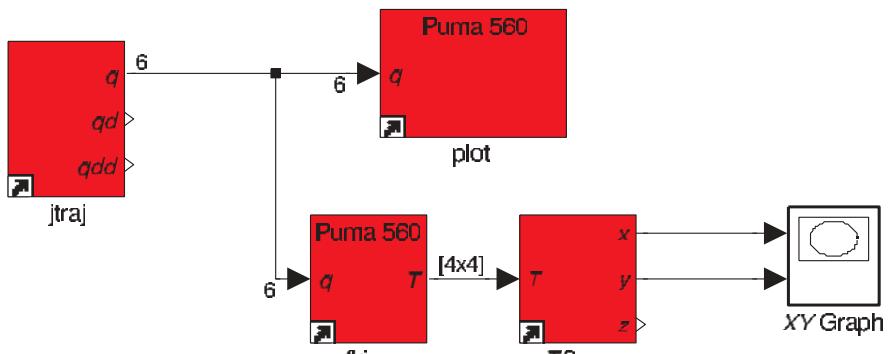


Fig. 7.10.  
Simulink® model `sl_jspace`  
for joint-space motion

the path. Also note that the initial roll angle is shown as  $-\pi$  but at the next time step it is  $\pi$ . This is an artifact of finite precision arithmetic and both angles are equivalent on the circle.

We can also implement this example in Simulink®

```
>> sl_jspace
```

The Simulink® integration time needs to be set equal to the motion segment time, through the **Simulation** menu or from the MATLAB® command line

```
>> sim('sl_jspace', 10);
```

and the block diagram model is shown in Fig. 7.10. The parameters of the `jtraj` block are the initial and final values for the joint coordinates and the time duration of the motion segment. The smoothly varying joint angles are wired to a `plot` block which will animate a robot in a separate window, and to an `fkine` block to compute the forward kinematics. Both the `plot` and `fkine` blocks have a parameter which is a `SerialLink` object, in this case `p560`. The Cartesian position of the end-effector pose is extracted using the `T2xyz` block which is analogous to the Toolbox function `transl`. The `XY Graph` block plots `y` against `x`.

## 7.4.2 Cartesian Motion

For many applications we require straight-line motion in Cartesian space which is known as Cartesian motion. This is implemented using the Toolbox function `ctraj` which was introduced in Chap. 3. Its usage is very similar to `jtraj`

```
>> Ts = ctraj(T1, T2, length(t));
```

where the arguments are the initial and final pose and the *number of time steps* and it returns the trajectory as a 3-dimensional matrix.

As for the previous joint-space example we will extract and plot the translation

```
>> plot(t, transl(Ts));
```

and orientation components

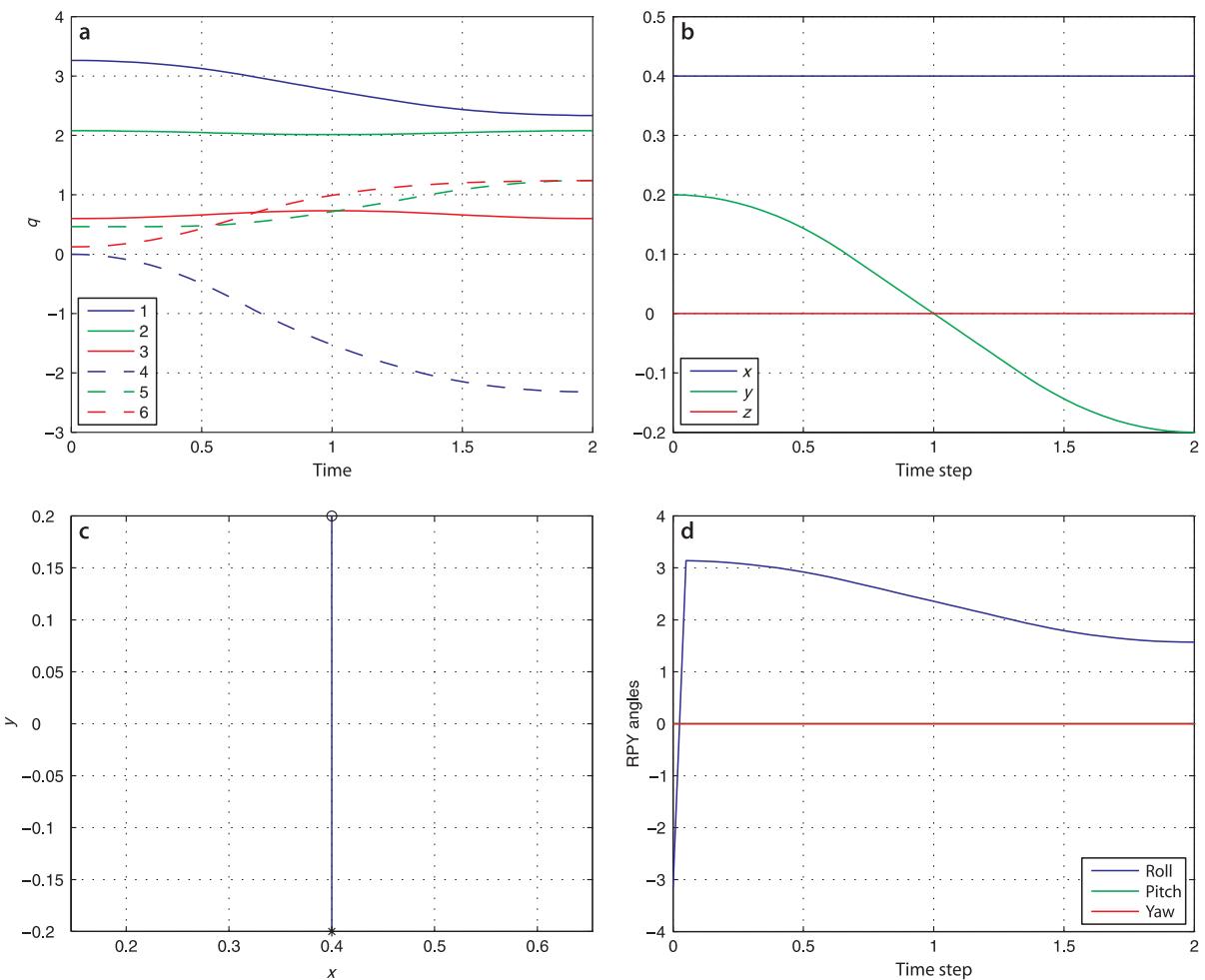
```
>> plot(t, tr2rpy(Ts));
```

of this motion which is shown in Fig. 7.11 along with the path of the end-effector in the *xy*-plane. Compared to Fig. 7.9b and c we note some important differences. Firstly the position and orientation, Fig. 7.11b and d vary linearly with time. For orientation the pitch angle is constant at zero and does not vary along the path. Secondly the end-effector follows a straight line in the *xy*-plane as shown in Fig. 7.9c.

The corresponding joint-space trajectory is obtained by applying the inverse kinematics

```
>> qc = p560.ikine6s(Ts);
```

and is shown in Fig. 7.11a. While broadly similar to Fig. 7.9a the minor differences are what result in the straight line Cartesian motion.



### 7.4.3 Motion through a Singularity

We have already briefly touched on the topic of singularities (page 148) and we will revisit it again in the next chapter. In the next example we deliberately choose a trajectory that moves through a robot singularity. We change the Cartesian endpoints of the previous example to

```
>> T1 = transl(0.5, 0.3, 0.44) * trotz(pi/2);
>> T2 = transl(0.5, -0.3, 0.44) * trotz(pi/2);
```

which results in motion in the  $y$ -direction with the end-effector  $z$ -axis pointing in the  $x$ -direction. The Cartesian path is

```
>> Ts = ctraj(T1, T2, length(t));
```

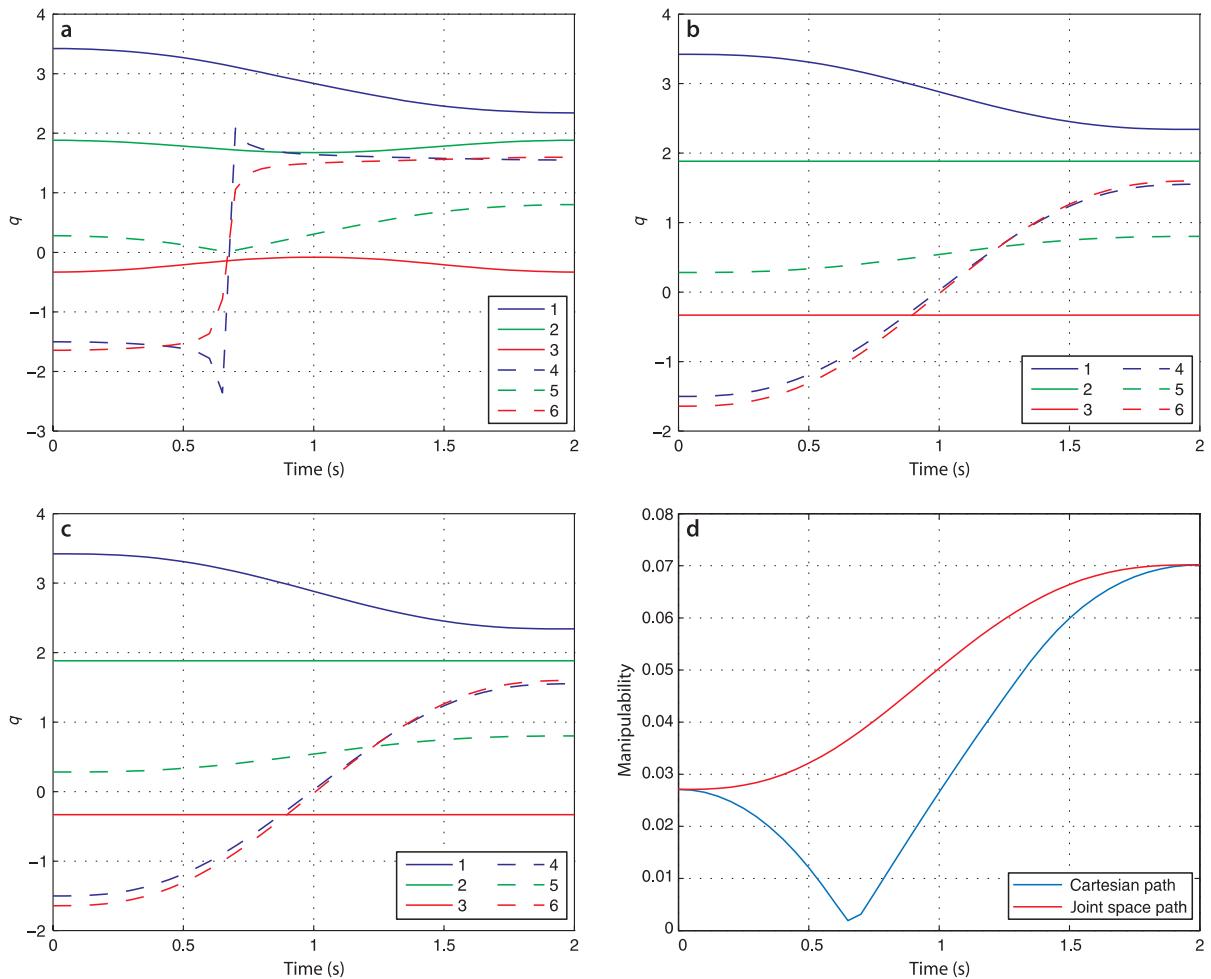
which we convert to joint coordinates

```
>> qc = p560.ikine6s(Ts)
```

and is shown in Fig. 7.12a. At time  $t \approx 0.7$  s we observe that the rate of change of the wrist joint angles  $q_4$  and  $q_6$  has become very high.► The cause is that  $q_5$  has become almost zero which means that the  $q_4$  and  $q_6$  rotational axes are almost aligned – another gimbal lock situation or singularity.

**Fig. 7.11.** Cartesian motion. **a** Joint coordinates versus time; **b** Cartesian position versus time; **c** Cartesian position locus in the  $xy$ -plane; **d** roll-pitch-yaw angles versus time

$q_6$  has increased rapidly, while  $q_4$  has decreased rapidly and wrapped around from  $-\pi$  to  $\pi$ .



**Fig. 7.12.** Cartesian motion through a wrist singularity. **a** Joint coordinates computed by inverse kinematics (`ikine6s`); **b** joint coordinates computed by generalized inverse kinematics (`ikine`); **c** joint coordinates for joint-space motion; **d** manipulability

The joint alignment means that the robot has lost one degree of freedom and is now effectively a 5-axis robot. Kinematically we can only solve for the sum  $q_4 + q_6$  and there are an infinite number of solutions for  $q_4$  and  $q_6$  that would have the same sum. The joint-space motion between the two poses, Fig. 7.12b, is immune to this problem since it does not require the inverse kinematics. However it will not maintain the pose of the tool in the  $x$ -direction for the whole path – only at the two end points. From Fig. 7.12c we observe that the generalized inverse kinematics method `ikine` handles the singularity with ease.

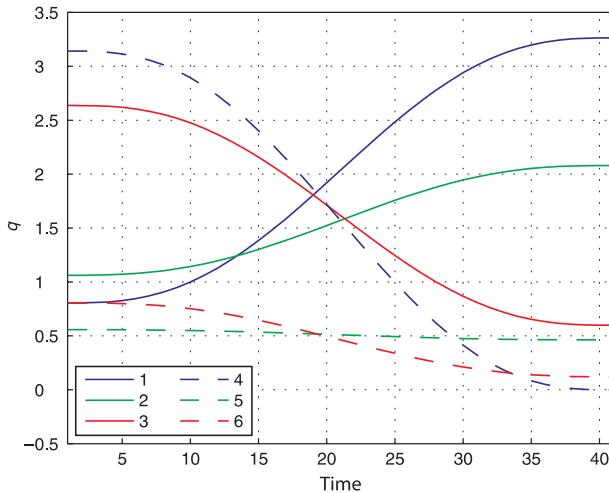
The manipulability measure for this path

```
>> m = p560.maniplty(qc);
```

is plotted in Fig. 7.12d and shows that manipulability was almost zero around the time of the rapid wrist joint motion. Manipulability and the generalized inverse kinematics function are based on the manipulator's Jacobian matrix which is the topic of the next chapter.

#### 7.4.4 Configuration Change

Earlier (page 147) we discussed the kinematic configuration of the manipulator arm and how it can work in a left- or right-handed manner and with the elbow up or down. Consider the problem of a robot that is working for a while left-handed at one work



**Fig. 7.13.**  
Joint space motions for configuration change from right-handed to left-handed

station, then working right-handed at another. Movement from one configuration to another ultimately results in no change in the end-effector pose since both configuration have the same kinematic solution – therefore we *cannot* create a trajectory in Cartesian space. Instead we must use joint-space motion.

For example to move the robot arm from the right- to left-handed configuration we first define some end-effector pose

```
>> T = transl(0.4, 0.2, 0) * trotx(pi);
```

and then determine the joint coordinates for the right- and left-handed elbow-up configurations

```
>> qr = p560.ikine6s(T, 'ru');
>> ql = p560.ikine6s(T, 'lu');
```

and then create a path between these two joint coordinates

```
>> q = jtraj(qr, ql, t);
```

Although the initial and final end-effector pose is the same, the robot makes some quite significant joint space motion as shown in Fig. 7.13 – in the real world you need to be careful the robot doesn't hit something. Once again, the best way to visualize this is in animation

```
>> p560.plot(q)
```

## 7.5 Advanced Topics

### 7.5.1 Joint Angle Offsets

The pose of the robot with zero joint angles is frequently some rather unusual (or even mechanically unachievable) pose. For the Puma robot the zero-angle pose is a non-obvious *L-shape* with the upper arm horizontal and the lower arm vertically upward as shown in Fig. 7.5a. This is a consequence of constraints imposed by the Denavit-Hartenberg formalism. A robot control designer may choose the zero-angle configuration to be something more obvious such as that shown in Fig. 7.5b or c.

The joint coordinate offset provides a mechanism to set an arbitrary configuration for the zero joint coordinate case. The offset vector,  $q_0$ , is added to the user specified joint angles before any kinematic or dynamic function is invoked, ▶ for example

It is actually implemented within the Link object.

$$\xi = \mathcal{K}(\mathbf{q} + \mathbf{q}_0) \quad (7.6)$$

Similarly it is subtracted after an operation such as inverse kinematics

$$\mathbf{q} = \mathcal{K}^{-1}(\xi) - \mathbf{q}_0 \quad (7.7)$$

The offset is set by assigning the `offset` property of the `Link` object, for example

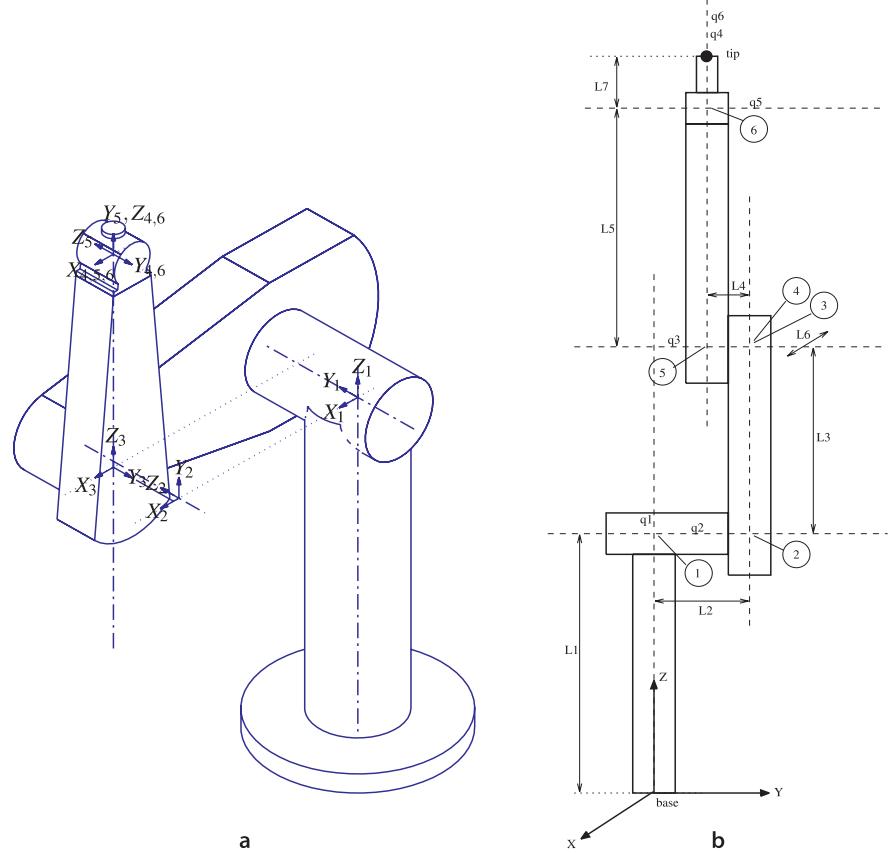
```
>> L = Link([0 0 1 0]);
>> L.offset = pi/4
```

or

```
>> p560.links(2).offset = pi/2;
```

### 7.5.2 Determining Denavit-Hartenberg Parameters

The classical method of determining Denavit-Hartenberg parameters is to systematically assign a coordinate frame to each link. The link frames for the Puma robot using the standard Denavit-Hartenberg formalism are shown in Fig. 7.14a. However there are strong constraints on placing each frame since joint rotation must be about the  $z$ -axis and the link displacement must be in the  $x$ -direction. This in turn imposes constraints on the placement of the coordinate frames for the base and the end-effector, and ultimately dictates the zero-angle pose just discussed. Determining the Denavit-Hartenberg parameters and link coordinate frames for a completely new mechanism is therefore more difficult than it should be – even for an experienced roboticist.



**Fig. 7.14.**

Puma 560 robot coordinate frames. **a** Standard Denavit-Hartenberg link coordinate frames for Puma in the zeroangle pose (Corke 1996b); **b** alternative approach showing the sequence of elementary transforms from base to tip. Rotations are about the axes shown as dashed lines (Corke 2007)

An alternative approach, supported by the Toolbox, is to simply describe the manipulator as a series of elementary translations and rotations from the base to the tip of the end-effector. Some of the the elementary operations are constants such as translations that represent link lengths or offsets, and some are functions of the generalized joint coordinates  $q_i$ . Unlike the conventional approach we impose no constraints on the axes about which these rotations or translations can occur.

An example for the Puma robot is shown in Fig. 7.14b. We first define a convenient coordinate frame at the base and then write down the sequence of translations and rotations in a string

```
>> s = 'Tz(L1) Rz(q1) Ry(q2) Ty(L2) Tz(L3) Ry(q3) Tx(L6) Ty(L4)
      Tz(L5) Rz(q4) Ry(q5) Rz(q6)'
```

Note that we have described the second joint as `Ry(q2)`, a rotation about the  $y$ -axis, which is not possible using the Denavit-Hartenberg formalism.

This string is input to a symbolic algebra function▶

```
>> dh = DHFactor(s);
```

which returns a `DHFactor` object▶ that holds the kinematic structure of the robot that has been factorized into Denavit-Hartenberg parameters. We can display this in a human-readable form

```
>> dh.display
DH(q1, L1, 0, -90).DH(q2+90, 0, -L3, 0).DH(q3-90, L2+L4, L6, 90).
DH(q4, L5, 0, -90).DH(q5, 0, 0, 90).DH(q6, 0, 0, 0)
```

which shows the Denavit-Hartenberg parameters for each joint in the order  $\theta, d, a$  and  $\alpha$ . Joint angle offsets (the constants added to or subtracted from joint angle variables such as `q2` and `q3`) are generated automatically, as are base and tool transformations. The object can generate a string that is a complete Toolbox command to create the robot named “puma”

```
>> cmd = dh.command('puma')
cmd =
SerialLink([-pi/2, 0, 0, L1; 0, -L3, 0, 0; pi/2, L6, 0, L2+L4;
             -pi/2, 0, 0, L5; pi/2, 0, 0, 0; 0, 0, 0, 0; ], ...
             'name', 'puma', ...
             'base', eye(4,4), 'tool', eye(4,4), ...
             'offset', [0 pi/2 -pi/2 0 0 0])
```

which can be executed

```
>> robot = eval(cmd)
```

to create a workspace variable called `robot` that is a `SerialLink` object.▶

Written in Java and part of the Robotics Toolbox, the MATLAB® Symbolic Toolbox is not required.

Actually a Java object.

The length parameters `L1` to `L6` must be defined in the workspace first.

### 7.5.3 Modified Denavit-Hartenberg Notation

The Denavit-Hartenberg parameters introduced in this chapter is commonly used and described in many robotics textbooks. Craig (1986) first introduced the modified Denavit-Hartenberg parameters where the link coordinate frames shown in Fig. 7.15 are attached to the near (proximal), rather than the far end of each link. This modified notation in in some ways clearer and tidier and is also now commonly used. However this has increased the scope for confusion, particularly for those who are new to robot kinematics. The root of the problem is that the algorithms for kinematics, Jacobians and dynamics depend on the kinematic conventions used. According to Craig's convention the link transform matrix is

$${}^{j-1}A_j = R_x(\alpha_{j-1})T_x(a_{j-1})R_z(\theta_j)T_z(d_j) \quad (7.8)$$

denoted by Craig as  ${}^{j-1}A$ . This has the same terms as Eq. 7.1 but in a different order – remember rotations are not commutative – and this is the nub of the problem.  $a_j$  is

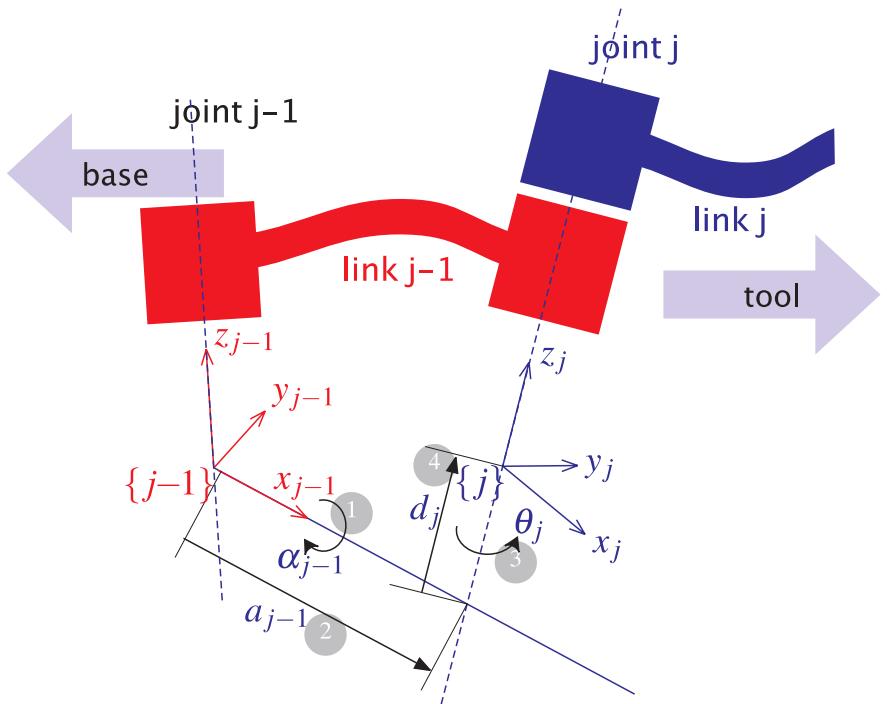


Fig. 7.15.

Definition of modified Denavit and Hartenberg link parameters. The colors red and blue denote all things associated with links  $j - 1$  and  $j$  respectively. The numbers in circles represent the order in which the elementary transforms are applied

always the length of link  $j$ , but it is the displacement between the origins of frame  $\{j\}$  and frame  $\{j + 1\}$  in one convention, and frame  $\{j - 1\}$  and frame  $\{j\}$  in the other.

If you intend to build a Toolbox robot model from a table of kinematic parameters provided in a paper it is really important to know which convention the author of the table used. Too often this important fact is not mentioned. An important clue lies in the column headings. If they all have the same subscript, i.e.  $\theta_j$ ,  $d_j$ ,  $a_j$  and  $\alpha_j$  then this is standard Denavit-Hartenberg notation. If half the subscripts are different, i.e.  $\theta_j$ ,  $d_j$ ,  $a_{j-1}$  and  $\alpha_{j-1}$  then you are dealing with modified Denavit-Hartenberg notation. In short, you must know which kinematic convention your Denavit-Hartenberg parameters conform to.

You can also help the cause when publishing by stating clearly which kinematic convention is used for your parameters.

The Toolbox can handle either form, it only needs to be specified, and this is achieved via an optional argument when creating a link object

```
>> L1 = link([0 1 0 0 0], 'modified')
L1 =
    q1          0          1          0 (modDH)
```

rather than

```
>> L1 = link([0 1 0 0 0])
L1 =
    q1          0          1          0 (stdDH)
```

Everything else from here on, creating the robot object, kinematic and dynamic functions works as previously described.

The two forms can be interchanged by considering the link transform as a string of elementary rotations and translations as in Eq. 7.1 or Eq. 7.8. Consider the transformation chain for standard Denavit-Hartenberg notation

$$\frac{T_{Rz}(\theta_1)T_z(d_1)T_x(a_1)T_{Rx}(\alpha_1)}{DH_1} \cdot \frac{T_{Rz}(\theta_2)T_z(d_2)T_x(a_2)T_{Rx}(\alpha_2)}{DH_2} \cdots$$

which we can regroup as

$$\underbrace{T_{Rz}(\theta_1)T_z(d_1)}_{\text{base}} \cdot \underbrace{T_x(a_1)T_{Rx}(\alpha_1)T_{Rz}(\theta_2)T_z(d_2)}_{\text{MDH}_1} \cdot \underbrace{T_x(a_2)T_{Rx}(\alpha_2)\dots}_{\text{MDH}_2}$$

where the terms marked as  $\text{MDH}_j$  have the form of Eq. 7.8 taking into account that translation along, and rotation about the same axis *is* commutative, that is,  $T_{Rk}(\theta)T_k(d) = T_k(d)T_{Rk}(\theta)$  for  $k \in \{x, y, z\}$ .

## 7.6 Application: Drawing

Our goal is to create a trajectory that will allow a robot to draw the letter ‘E’. Firstly we define a number of via points that define the strokes of the letter

```
>> path = [ 1 0 1; 1 0 0; 0 0 0; 0 2 0; 1 2 0;
           1 2 1; 0 1 1; 0 1 0; 1 1 0; 1 1 1];
```

which is defined in the  $xy$ -plane in arbitrary units. The pen is down when  $z = 0$  and up when  $z > 0$ . The path segments can be plotted

```
>> plot3(path(:,1), path(:,2), path(:,3), 'color', 'k', 'LineWidth', 2)
```

as shown in Fig. 7.16. We convert this to a continuous path

```
>> p = mstraj(path, [0.5 0.5 0.3], [], [2 2 2], 0.02, 0.2);
```

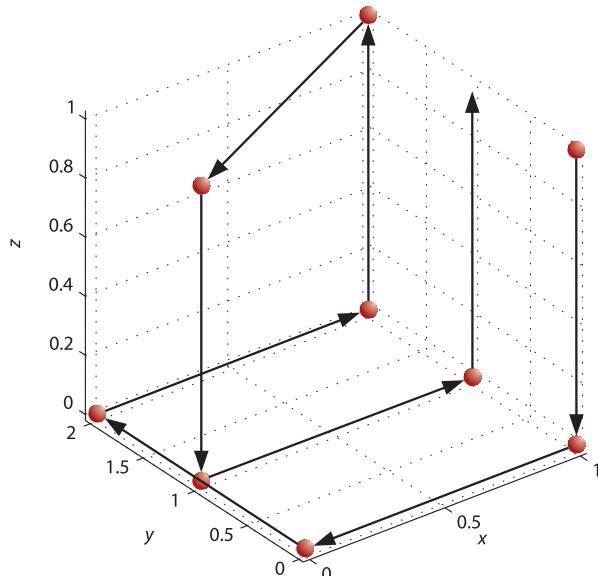
where the second argument is the maximum speed in the  $x$ -,  $y$ - and  $z$ -directions, the fourth argument is the initial coordinate followed by the sample interval and the acceleration time. The number of steps in the interpolated path is

```
>> about(p)
p [double] : 1563x3 (37512 bytes)
```

and will take

```
>> numrows(p) * 0.02
ans =
    31.2600
```

seconds to execute at the 20 ms sample interval.



**Fig. 7.16.**  
The letter ‘E’ drawn with a 10-point path. Markers show the via points and solid lines the motion segments

`p` is a sequence of  $x$ - $y$ - $z$ -coordinates which we must convert into a sequence of Cartesian poses. The path we have defined draws a letter that is two units tall and one unit wide so the coordinates will be scaled by a factor of 0.1 (making the letter 200 mm tall by 100 mm wide)

```
>> Tp = transl(0.1 * p);
```

which is a sequence of homogeneous transformations describing the pose at every point along the path. The origin of the letter will be placed at (0.4, 0, 0) in the workplace

```
>> Tp = homtrans(transl(0.4, 0, 0), Tp);
```

which premultiplies each pose in `Tp` by the first argument.

Finally we need to consider orientation. Each of the coordinate frames defined in `Tp` assumes its  $z$ -axis is vertically upward. However the Puma robot is working on a horizontal surface with its elbow up and writing just like a person at a desk. The orientation of its tool, its  $z$ -axis, must therefore be downward. One way to fix this mismatch is by setting the robot's tool transform to make the tool axis point upwards

```
>> p560.tool = trotx(pi);
```

Now we can apply inverse kinematics

```
>> q = p560.ikine6s(Tp);
```

to determine the joint coordinates and then animate it.

```
>> p560.plot(q)
```

The Puma is drawing the letter 'E', and lifting its pen in between strokes! The approach is quite general and we could easily change the size of the letter, draw it on an arbitrary plane or use a robot with different kinematics.

## 7.7 Application: a Simple Walking Robot

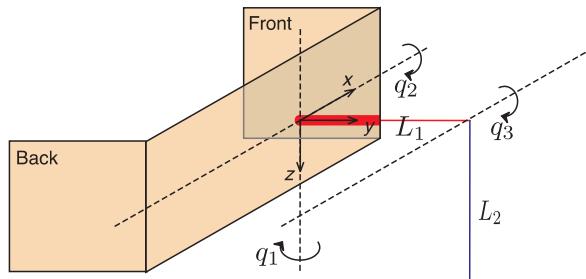
*Four legs good, two legs bad!*  
Snowball the pig, Animal Farm by George Orwell

Our goal is to create a four-legged walking robot. We start by creating a 3-axis robot *arm* that we use as a leg, plan a trajectory for the leg that is suitable for walking, and then instantiate four instances of the leg to create the walking robot.

### 7.7.1 Kinematics

Kinematically a robot leg is much like a robot arm, and for this application a three joint serial-link manipulator is sufficient. Determining the Denavit-Hartenberg parameters, even for a simple robot like this, is an involved procedure and the zero-angle offsets need to be determined in a separate step. Therefore we will use the procedure introduced in Sect. 7.5.2.

As always we start by defining our coordinate frame. This is shown in Fig. 7.17 along with the robot leg in its zero-angle pose. We have chosen the aerospace coordinate convention which has the  $x$ -axis forward and the  $z$ -axis downward, constraining the  $y$ -axis to point to the right-hand side. The first joint will be hip motion, forward and backward, which is rotation about the  $z$ -axis or  $R_z(q_1)$ . The second joint is hip motion up and down, which is rotation about the  $x$ -axis,  $R_x(q_2)$ . These form a spherical hip joint since the axes of rotation intersect. The knee is translated by  $L_1$  in the  $y$ -direction or  $T_y(L_1)$ . The third joint is knee motion, toward and away from the body,

**Fig. 7.17.**

The coordinate frame and axis rotations for the simple leg. The leg is shown in its zero angle pose

which is  $R_x(q_3)$ . The foot is translated by  $L_2$  in the  $z$ -direction or  $T_z(L_2)$ . The transform sequence of this robot, from hip to toe, is therefore  $R_z(q_1)R_x(q_2)T_y(L_1)R_x(q_3)T_z(L_2)$ .

Using the technique of Sect. 7.5.2 we write this sequence as the string

```
>> s = 'Rz(q1).Rx(q2).Ty(L1).Rx(q3).Tz(L2)';
```

Note that length constants must start with L. The string is automatically manipulated into Denavit-Hartenberg factors

```
>> dh = DHFactor(s)
DH(q1+90, 0, 0, +90).DH(q2, L1, 0, 0).
DH(q3-90, L2, 0, 0).Rz(+90).Rx(-90).Rz(-90)
```

The last three terms in this factorized sequence is a tool transform

```
>> dh.tool
ans =
trotz(pi/2)*trotx(-pi/2)*trotz(-pi/2)
```

that changes the orientation of the frame at the foot. However for this problem the foot is simply a point that contacts the ground so we are not concerned about its orientation. The method `dh.command` generates a string that is the Toolbox command to create a `SerialLink` object

```
>> dh.command('leg')
ans =
SerialLink([0, 0, 0, pi/2; 0, 0, L1, 0; 0, 0, -L1, 0; ],
            'name', 'leg', 'base', eye(4,4),
            'tool', trotz(pi/2)*trotx(-pi/2)*trotz(-pi/2),
            'offset', [pi/2 0 -pi/2 ])
```

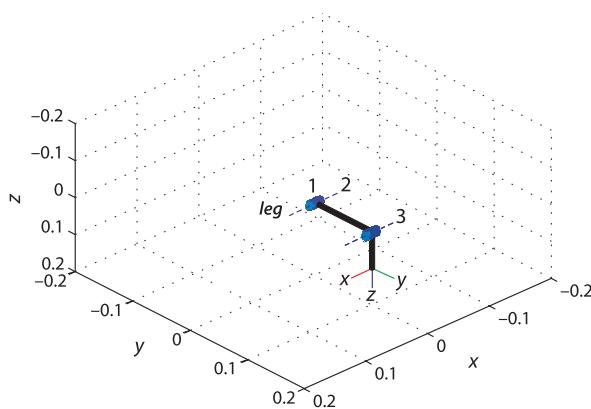
which is input to the MATLAB® `eval` command

```
>> L1 = 0.1; L2 = 0.1;
>> leg = eval( dh.command('leg') )
>> leg
leg =
leg (3 axis, RRR, stdDH)
+-----+
| j | theta | d | a | alpha |
+-----+
| 1 | q1 | 0 | 0 | 1.571 |
| 2 | q2 | 0 | 0.1 | 0 |
| 3 | q3 | 0 | -0.1 | 0 |
+-----+
grav = 0 base = 1 0 0 0 tool = 0 0 -1 0
      0 0 1 0 0 1 0 0
      9.81 0 0 1 0 1 0 0
                  0 0 0 1
```

after first setting the the length of each leg segment to 100 mm in the MATLAB® workspace.

As usual we perform a quick sanity check of our robot. For zero joint angles the foot is at

```
>> transl( leg.fkine([0,0,0]) )
ans =
          0    0.1000    0.1000
```



**Fig. 7.18.**  
Robot leg in its zero angle pose

as we designed it. We can visualize the zero-angle pose

```
>> leg.plot([0,0,0], 'nobase', 'noshadow')
>> set(gca, 'zdir', 'reverse'); view(137,48);
```

which is shown in Fig. 7.18. Note that we tell MATLAB® that our  $z$ -axis points downward. Now we should test that the other joints result in the expected motion. Increasing  $q_1$

```
>> transl( leg.fkine([0.2,0,0]) )
ans =
-0.0199    0.0980    0.1000
```

results in motion in the  $xy$ -plane, and increasing  $q_2$

```
>> transl( leg.fkine([0,0.2,0]) )
ans =
-0.0000    0.0781    0.1179
```

results in motion in the  $yz$ -plane, as does increasing  $q_3$

```
>> transl( leg.fkine([0,0,0.2]) )
ans =
-0.0000    0.0801    0.0980
```

We have now created and verified a simple robot leg.

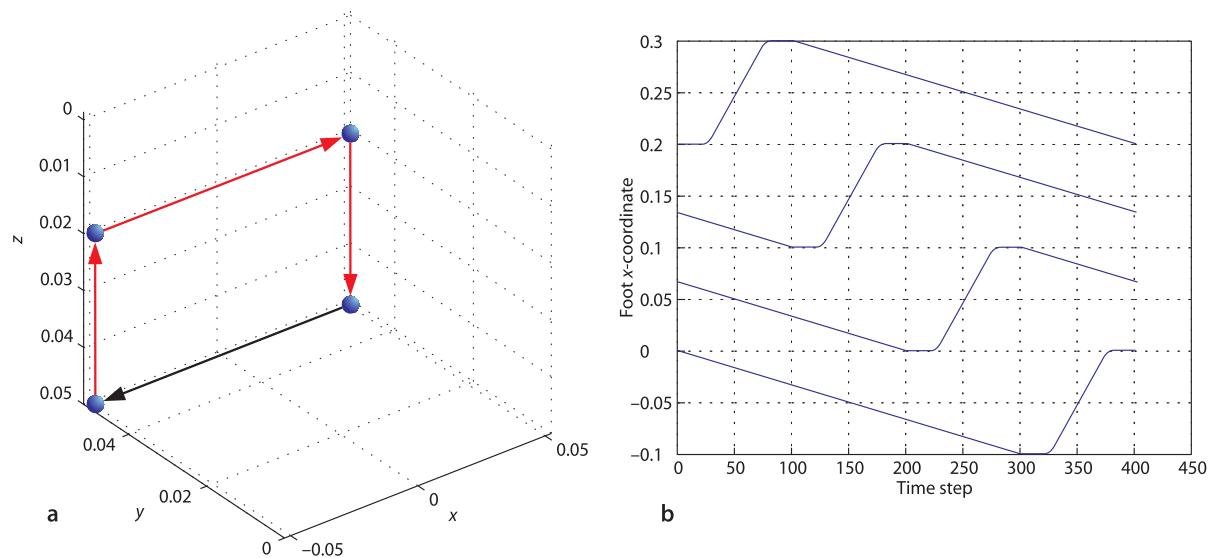
## 7.7.2 Motion of One Leg

The next step is to define the path that the end-effector of the leg, its foot, will follow. The first consideration is that the end-effector of all feet move backwards at the same speed in the ground plane – propelling the robot's body forward without its feet slipping. Each leg has a limited range of movement so it cannot move backward for very long. At some point we must reset the leg – lift the foot, move it forward and place it on the ground again. The second consideration comes from static stability – the robot must have at least three feet on the ground at all times so each leg must take its turn to reset. This requires that any leg is in contact with the ground for  $\frac{3}{4}$  of the cycle and is resetting for  $\frac{1}{4}$  of the cycle. A consequence of this is that the leg has to move much faster during reset since it has a longer path and less time to do it in.

The required trajectory is defined by the via points

```
>> xf = 50; xb = -xf; y = 50; zu = 20; zd = 50;
>> path = [xf y zd; xb y zd; xb y zu; xf y zu; xf y zd] * 1e-3;
```

where  $xf$  and  $xb$  are the forward and backward limits of leg motion in the  $x$ -direction (in units of mm),  $y$  is the distance of the foot from the body in the  $y$ -direction, and  $zu$  and  $zd$  are respectively the height of the foot in the  $z$ -direction for foot up and foot down. In this case the foot moves from 50 mm forward of the hip to 50 mm behind.



When the foot is down it is 50 mm below the hip and it is raised to 20 mm below the hip during reset. The points in `path` comprise a complete cycle correspond to the start of the stance phase, the end of stance, top of the leg lift, top of the leg return and the start of stance. This is shown in Fig. 7.19a.

Next we sample the multi-segment path at 100 Hz

```
>> p = mstraj(path, [], [0, 3, 0.25, 0.5, 0.25], path(1,:), 0.01, 0);
```

In this case we have specified a vector of desired segment times rather than maximum joint velocities. The final three arguments are the initial position, the sample interval and the acceleration time. This trajectory has a total time of 4 s and therefore comprises 400 points.

We apply inverse kinematics to determine the joint angle trajectories required for the foot to follow the path. This robot is underactuated so we use the generalized inverse kinematics `ikine` and set the mask so as to solve only for end-effector translation

```
>> qcycles = leg.ikine(transl(p), [], [1 1 1 0 0 0]);
```

We can view the motion of the leg in animation

```
>> leg.plot(qcycles, 'loop')
```

to verify that it does what we expect: slow motion along the ground, then a rapid lift, forward motion and foot placement. The '`loop`' option displays the trajectory in an endless loop and you need to type control-C to stop it.

### 7.7.3 Motion of Four Legs

Our robot has width and length

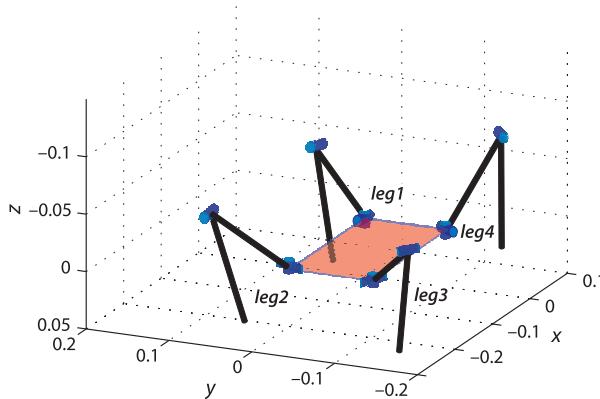
```
>> W = 0.1; L = 0.2;
```

We create multiple instances of the leg by cloning the `leg` object we created earlier, and providing different base transforms so as to attach the legs to different points on the body

```
>> legs(1) = SerialLink(leg, 'name', 'leg1');
>> legs(2) = SerialLink(leg, 'name', 'leg2', 'base', transl(-L, 0, 0));
>> legs(3) = SerialLink(leg, 'name', 'leg3', 'base', transl(-L, -W, 0)*trotz(pi));
>> legs(4) = SerialLink(leg, 'name', 'leg4', 'base', transl(0, -W, 0)*trotz(pi));
```

**Fig. 7.19.** a Trajectory taken by a single leg. Recall from Fig. 7.17 that the z-axis is downward. The red segments are the leg reset. b The x-direction motion of each leg (offset vertically) to show the gait. The leg reset is the period of high x-direction velocity

This way we can ensure that the reset takes exactly one quarter of the cycle.



**Fig. 7.20.**  
The walking robot

The result is a vector of `SerialLink` objects. Note that legs 3 and 4, on the left-hand side of the body have been rotated about the  $z$ -axis so that they point away from the body.

As mentioned earlier each leg must take its turn to reset. Since the trajectory is a cycle, we achieve this by having each leg run the trajectory with a phase shift equal to one quarter of the total cycle time. Since the total cycle has 400 points, each leg's trajectory is offset by 100, and we use modulo arithmetic to index into the cyclic gait for each leg. The result is the gait pattern shown in Fig. 7.19b.

The core of the walking program is

```
k = 1;
while 1
    q = qleg(p,:);
    legs(1).plot( gait(qcycle, k, 0, 0) );
    legs(2).plot( gait(qcycle, k, 100, 0) );
    legs(3).plot( gait(qcycle, k, 200, 1) );
    legs(4).plot( gait(qcycle, k, 300, 1) );
    drawnow
    k = k+1;
end
```

where the function

```
gait(q, k, ph, flip)
```

returns the  $k+ph^{\text{th}}$  element of `q` with modulo arithmetic that considers `q` as a cycle. The argument `flip` reverses the sign of the joint 1 motion for the legs on the left-hand side of the robot. A snapshot from the simulation is shown in Fig. 7.20. The entire implementation, with some additional refinement, is in the file `examples/walking.m` and detailed explanation is provided by the comments.

## 7.8 Wrapping Up

In this chapter we have learnt how to describe a serial-link manipulator in terms of a set of Denavit-Hartenberg parameters for each link. We can compute the relative pose of each link as a function of its joint variable and compose these into the pose of the robot's end-effector relative to its base. For robots with six joints and a spherical wrist we can compute the inverse kinematics which is the set of joint angles required to achieve a particular end-effector pose. This inverse is not unique and the robot may have several joint configurations that result in the same end-effector pose.

For robots which do not have six joints and a spherical wrist we can use an iterative numerical approach to solving the inverse kinematics. We showed how this could be applied to an under-actuated 2-link robot and a redundant 8-link robot. We also touched briefly on the topic of singularities which are due to the alignment of joints.

We also learnt about creating paths to move the end-effector smoothly between poses. Joint-space paths are simple to compute but do not result in straight line paths in Cartesian space which may be problematic for some applications. Straight line paths in Cartesian space can be generated but singularities in the workspace may lead to very high joint rates.

---

### Further Reading

Serial-link manipulator kinematics are covered in all the standard robotics textbooks such as by Paul (1981), Spong et al. (2006), Siciliano et al. (2008). Craig's text (2004) is also an excellent introduction to robot kinematics and uses the modified Denavit-Hartenberg notation, and the examples in the third edition are based on an older version of the Robotics Toolbox. Most of these books derive the inverse kinematics for a two-link arm and a six-link arm with a spherical wrist. The first full description of the latter was by Paul and Zhang (1986).

Closed-form inverse kinematic solution can be derived symbolically by writing down a number of kinematic relationships and solving for the joint angles, as described in Paul (1981). Software packages to automatically generate the forward and inverse kinematics for a given robot have been developed and these include Robotica (Nethery and Spong 1994) and SYMORO+ (Khalil and Creusot 1997).

The original work by Denavit and Hartenberg was their 1955 paper (Denavit and Hartenberg 1955) and their textbook (Hartenberg and Denavit 1964). The book has an introduction to the field of kinematics and its history but is currently out of print, although a version is available online. Siciliano et al. (2008) provide a very clear description of the process of assigning Denavit-Hartenberg parameters to an arbitrary robot. The alternative approach based on symbolic factorization is described in detail by Corke (2007). This has some similarities to the product of exponential (POE) form proposed by Park (1994). The definitive values for the parameters of the Puma 560 robot are described in the paper by Corke and Armstrong-Hélouvy (1995).

Robotic walking is a huge field in its own right and the example given here is very simplistic. Machines have been demonstrated with complex gaits such as running and galloping that rely on dynamic rather than static balance. A good introduction to legged robots is given in the Robotics Handbook (Siciliano and Khatib 2008, § 16).

Parallel-link manipulators were mentioned briefly on page 136 and have advantages such as increased actuation force and stiffness (since the actuators form a truss-like structure). For this class of mechanism the inverse kinematics is usually closed-form and forward kinematics often requiring numerical solution. Useful starting points for this class of robots are a brief section in Siciliano et al. (2008), the handbook (Siciliano and Khatib 2008, § 12) and in Merlet (2006).

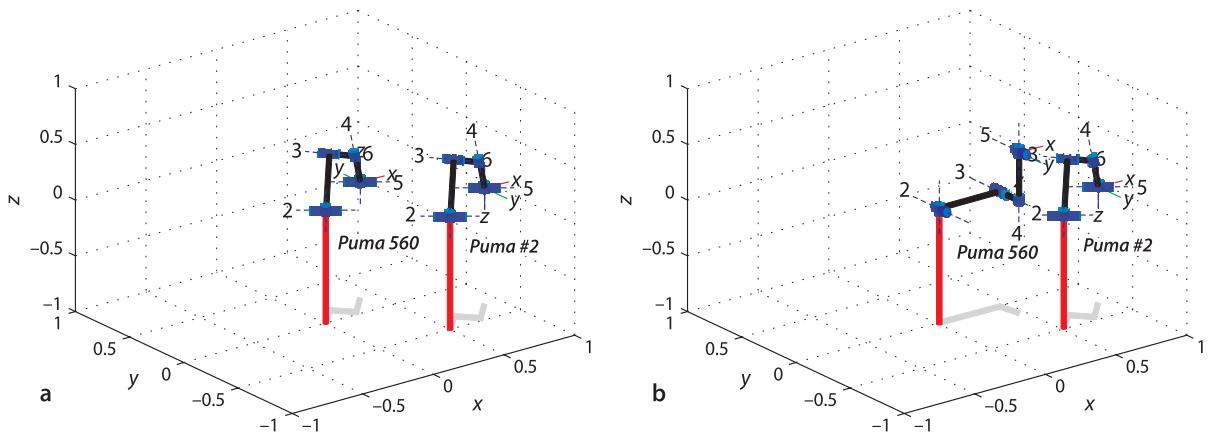
---

### The `plot` Method

The `plot` method was introduced in Sect. 7.2.1 to display the pose of a robot. Many aspects of the created figure such as scaling, shadows, joint axes and labels, tool coordinate frame labels and so on can be customized. The `plot` method also supports multiple views of the same robot, and figures can contain multiple robots. This section provides examples of some of these features and, as always, the full description is available in the online documentation.

The `hold` method works in an analogous way to normal data plotting and allows multiple robots to be drawn into the one set of axes. First create a clone of our standard Puma robot, with its base at a different location

```
>> p560_2 = SerialLink(p560, 'name', 'puma #2', ...
    'base', transl(0.5,-0.5,0));
```



**Fig. 7.21.** `plot` displaying multiple robots per axis. **a** Both robots have the same joint coordinates; **b** the robots have different joint coordinates

Draw the first robot

```
>> p560.plot(qr)
```

then add the second robot

```
>> hold on
>> p560_2.plot(qr)
```

which gives the result shown in Fig. 7.21a. The graphical robots can be separately animated, for instance

```
>> p560.plot(qz)
```

changes the pose of only the first robot as shown in Fig. 7.21b.

We can also create multiple views of the same robot or robots. Following on from the example above, we create a new figure

```
>> figure
>> p560.plot(qz)
```

and plot the *same* robot. You could alter the viewpoint in this new figure using the MATLAB® figure toolbar options. Now

```
>> p560.plot(qr)
```

causes the robot to change in *both* figures. The key to this is the robot's name which is used to identify the robot irrespective of which figure it appears in. The name of each robot must be unique for this feature to work.

The algorithm adopted by `plot` with respect to the usage of existing figures is:

1. If a robot with the specified name exists in the current figure, then redraw it with the specified joint angles. Also update the same named robot in all other figures.
2. If no such named robot exists, then if hold is on the robot will be added to the current plot else a new graphical robot of the specified name will be created in the current window.
3. If there are no figures, then a new figure will be created.

The robot can be *driven* manually using a graphical teach-pendant interface

```
>> p560.teach()
```

which displays a GUI with one slider per joint. As the sliders are moved the specified graphical robot joint moves, in all the figures in which it has been drawn.

**Exercises**

1. Experiment with the `teach` method.
2. Derive the forward kinematics for the 2-link robot from first principles. What is end-effector position ( $x, y$ ) given  $q_1$  and  $q_2$ ?
3. Derive the inverse kinematics for the 2-link robot from first principles. What are the joint angles ( $q_1, q_2$ ) end-effector given ( $x, y$ )?
4. Compare the solutions generated by `ikine6s` and `ikine` for the Puma 560 robot at different poses. Is there any difference in accuracy? How much slower is `ikine`?
5. For the Puma 560 at configuration `qn` demonstrate a configuration change from elbow up to elbow down.
6. Drawing an 'E' (page 162)
  - a) Change the size of the letter.
  - b) Construct the path for the letter 'C'.
  - c) Write the letter on a vertical plane.
  - d) Write the letter on an inclined plane.
  - e) Change the robot from a Puma 560 to the Fanuc 10L.
  - f) Write the letter on a sphere.
  - g) This writing task does not require 6DOF since the rotation of the pen about its axis is not important. Remove the final link from the Puma 560 robot model and repeat the exercise.
7. Walking robot (page 163)
  - a) Shorten the reset trajectory by reducing the leg lift during reset.
  - b) Increase the stride of the legs.
  - c) Figure out how to steer the robot by changing the stride length on one side of the body.
  - d) Change the gait so the robot moves sideways like a crab.
  - e) Add another pair of legs. Change the gait to reset two legs or three legs at a time.
  - f) Currently in the simulation the legs move but the body does not move forward. Modify the simulation so the body moves.

# 8

# Velocity Relationships



In this chapter we consider the relationship between the rate of change of joint coordinates, the joint velocity, and the velocity of the end-effector. The 3-dimensional end-effector pose  $\xi \in SE(3)$  has a velocity which is represented by a 6-vector known as a spatial velocity. The joint velocity and the end-effector velocity are related by the manipulator Jacobian matrix which is a function of manipulator pose.

Section 8.1 uses a numerical approach to introduce the manipulator Jacobian. Next we introduce additional Jacobians to transform velocity between coordinate frames and angular velocity between different angular representations. The numerical properties of the Jacobian matrix are shown to provide insight into the dexterity of the manipulator – the directions in which it can move easily and those in which it cannot – and understanding about singular configurations. In Sect. 8.2 the inverse Jacobian is used to generate Cartesian paths without requiring inverse kinematics, and this can be applied to over- and under-actuated robots. Section 8.3 demonstrates how the Jacobian transpose is used to transform forces from the end-effector to the joints and between coordinate frames. Finally, in Sect. 8.4 the numeric inverse kinematic solution, used in the previous chapter, is fully described.

## 8.1 Manipulator Jacobian

We start by investigating how small changes in joint coordinates affect the pose of the end-effector. Using the homogeneous transformation representation of pose we can approximate its derivative with respect to joint coordinates by a first-order difference

$$\frac{dT}{dq} \approx \frac{T(q + \delta_q) - T(q)}{\delta_q}$$

and recalling the definition of  $T$  from Eq. 2.19 we can write

$$\frac{dT}{dq} \approx \frac{1}{\delta_q} \begin{pmatrix} R(q + \delta_q) - R(q) & \left| \begin{array}{c} \delta_x \\ \delta_y \\ \delta_z \end{array} \right| \\ \hline 0 & 0 \end{pmatrix} \quad (8.1)$$

where  $(\delta_x, \delta_y, \delta_z)$  is the translational displacement of the end-effector.

For the purpose of example we again use the Puma 560 robot

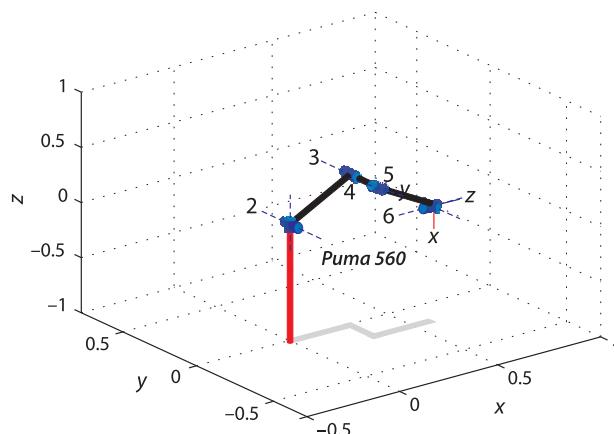
`>> mdl_puma560`

and at the nominal configuration `qn` shown in Fig. 8.1 the end-effector pose is

`>> T0 = p560.fkine(qn);`

We perturb joint one by a small but finite amount `dq`

`>> dq = 1e-6;`



**Fig. 8.1.**  
Puma robot in its nominal pose  $q_n$ . The end-effector  $z$ -axis points in the world  $x$ -direction, and the  $x$ -axis points downward

and compute the new end-effector pose

```
>> Tp = p560.fkine(qn + [dq 0 0 0 0 0]);
```

and using Eq. 8.1 the derivative is

```
>> dTdq1 = (Tp - T0) / dq
dTdq1 =
0    -1.0000   -0.0000   0.1500
-0.0000   -0.0000   1.0000   0.5963
0        0        0        0
0        0        0        0
```

This is clearly no longer a homogeneous transformation – the upper left  $3 \times 3$  matrix is not an orthonormal matrix and the lower-right element is not a 1. What does this result mean?

Equating the elements of column four of  $dTdq1$  and the matrix in Eq. 8.1 we can write

$$\begin{pmatrix} \delta_x \\ \delta_y \\ \delta_z \end{pmatrix} = \begin{pmatrix} 0.1500 \\ 0.5963 \\ 0 \end{pmatrix} \delta_{q_1}$$

which is the displacement of the end-effector position as a function of a displacement in  $q_1$ . From Fig. 8.1 this makes sense – a small rotation of the waist joint ( $q_1$ ) will move the end-effector in the horizontal  $xy$ -plane but not vertically.

We can repeat the process for the next joint

```
>> Tp = p560.fkine(qn + [0 dq 0 0 0 0]);
>> dTdq2 = (Tp - T0) / dq
dTdq =
1.0000   -0.0000   -0.0000   0.0144
0.0000     0     0.0000     0
0.0000   0.0000   1.0000   0.5963
0        0        0        0
```

and write

$$\begin{pmatrix} \delta_x \\ \delta_y \\ \delta_z \end{pmatrix} = \begin{pmatrix} 0.0144 \\ 0 \\ 0.5963 \end{pmatrix} \delta_{q_2}$$

A small motion of the shoulder joint causes end-effector motion in the vertical  $xz$ -plane, as expected, but not the  $y$ -direction. Dividing both sides by an infinitesimal time step  $\delta_t$  we find a relationship

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} = \begin{pmatrix} 0.0144 \\ 0 \\ 0.5963 \end{pmatrix} \dot{q}_1$$

between joint angle velocity and end-effector velocity.

Now we consider the top-left  $3 \times 3$  submatrix of the matrix in Eq. 8.1 and multiply it by  $\delta_q / \delta_t$  to achieve a first-order approximation to the derivative of  $R$

$$\dot{R} \approx \left( \frac{R(q + \delta_q) - R(q)}{\delta_q} \right) \frac{\delta_q}{\delta_t}$$

Recalling an earlier definition of the derivative of an orthonormal rotation matrix Eq. 3.4 we write

$$S(\omega)R \approx \frac{R(q + \delta_q) - R(q)}{\delta_q} \dot{q}_1$$

$$S(\omega) \approx \left( \frac{R(q + \delta_q) - R(q)}{\delta_q} R^T \right) \dot{q}_1$$

from which we find a relationship between end-effector angular velocity and joint velocity

$$\omega \approx \text{vex} \left( \frac{R(q + \delta_q) - R(q)}{\delta_q} R^T \right) \dot{q}_1$$

Continuing with the joint 1 derivative computed above

```
>> dRdq1 = dTdq1(1:3,1:3);
>> R = T0(1:3, 1:3);
>> S = dRdq1 * R'
S =
-0.0000   -1.0000    0.0000
 1.0000   -0.0000    0.0000
  0         0         0
```

which is a skew symmetric matrix from which we extract the angular velocity vector

```
>> vex(S)
ans =
  0     0     1
```

and finally we write

$$\begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \dot{q}_1$$

From Fig. 8.1 this makes sense. The end-effector's angular velocity is the same as the angular velocity of joint one which rotates about the world  $z$ -axis.

Repeating the process for small motion of the second joint we obtain

```
>> dRdq2 = dTdq2(1:3,1:3);
>> S = dRdq2 * inv(R)
S =
-0.0000   -0.0000   -1.0000
 0.0000   -0.0000   -0.0000
 1.0000   -0.0000   -0.0000
>> vex(S)
  0.0000   -1.0000    0.0000
```

A Jacobian is the matrix equivalent of the derivative – the derivative of a vector-valued function of a vector with respect to a vector. If  $\mathbf{y} = F(\mathbf{x})$  and  $\mathbf{x} \in \mathbb{R}^n$  and  $\mathbf{y} \in \mathbb{R}^m$  then the Jacobian is the  $m \times n$  matrix

$$J = \frac{\partial F}{\partial \mathbf{x}} \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

The Jacobian is named after Carl Jacobi, and more details are given in Appendix G.

from which we write

$$\begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} \dot{q}_2$$

Once again, with reference to Fig. 8.1, this makes sense. Rotation of joint two, whose rotational axis is in the negative  $y$ -direction, results in an angular velocity in the negative  $y$ -direction.

We have established, numerically, the relationship between the velocity of individual joints and the translational and angular velocity of the robot's end-effector. Earlier Eq. 7.3 we wrote the forward kinematics in functional form as

$$\xi = \mathcal{K}(\mathbf{q})$$

and taking the derivative we write

$$\nu = J(\mathbf{q})\dot{\mathbf{q}} \quad (8.2)$$

which is the instantaneous forward kinematics where  $\nu = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z) \in \mathbb{R}^6$  is a spatial velocity and comprises translational and rotational velocity components. The matrix  $J(\mathbf{q}) \in \mathbb{R}^{6 \times N}$  is the manipulator Jacobian or the geometric Jacobian.

The Jacobian matrix can be computed directly by the `jacob0` method of the `SerialLink` object

```
>> J = p560.jacob0(qn)
J =
  0.1501    0.0144    0.3197      0      0      0
  0.5963    0.0000    0.0000      0      0      0
   0    0.5963    0.2910      0      0      0
   0   -0.0000   -0.0000    0.7071   -0.0000   -0.0000
   0   -1.0000   -1.0000   -0.0000   -1.0000   -0.0000
  1.0000    0.0000    0.0000   -0.7071    0.0000   -1.0000
```

The function `jacob0` does not use finite differences. It has a direct form based on the Denavit-Hartenberg parameters of the robot arm (Paul and Shimano 1978).

The rows correspond to Cartesian degrees of freedom and the columns correspond to joints – they are the end-effector spatial velocities corresponding to unit velocity of the corresponding joints. The results we computed earlier, using derivative approximation, can be seen in the first two columns. The  $3 \times 3$  block of zeros in the top right indicates that motion of the wrist joints has no effect on the end-effector translational motion – this is an artifact of the spherical wrist and a zero length tool.

### 8.1.1 Transforming Velocities between Coordinate Frames

Consider two frames  $\{A\}$  and  $\{B\}$  related by



**Carl Gustav Jacob Jacobi (1804–1851)** was a Prussian mathematician. He obtained a Doctor of Philosophy degree from Berlin University in 1825. In 1827 he was appointed professor of mathematics at Königsberg University and held this position until 1842 when he suffered a breakdown from overwork.

Jacobi wrote a classic treatise on elliptic functions in 1829 and also described the derivative of  $m$  functions of  $n$  variables which bears his name. He was elected a foreign member of the Royal Swedish Academy of Sciences in 1836. He is buried in the Friedhof I der Dreifaltigkeits-Kirchengemeinde in Berlin.

$${}^A T_B = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}$$

then a spatial velocity with respect to frame  $\{A\}$  can be expressed relative to frame  $\{B\}$  by

$${}^B \nu = {}^B J_A {}^A \nu \quad (8.3)$$

where the Jacobian

$${}^B J_A = J_\nu({}^A T_B) = \begin{pmatrix} R^T & (S(t)R)^T \\ \mathbf{0}_{3 \times 3} & R^T \end{pmatrix} \quad (8.4)$$

As discussed on page 16 the middle indices in Eq. 8.3, the ‘‘A’s, can be considered to *cancel out*.

is a  $6 \times 6$  matrix and a function of the relative pose between the frames. The matrix has an interesting structure. The lower-left block of zeros means that rotational velocity in frame  $\{B\}$  depends only on the rotational velocity in frame  $\{A\}$ . If the frames are displaced by a translation the top-right  $3 \times 3$  block is non-zero, and this means that the translational velocity in frame  $\{B\}$  has a component due to both translational and rotational velocity in frame  $\{A\}$ . The rotational velocity in  $\{A\}$  is simply rotated to frame  $\{B\}$ .

For example if the second frame is related to the first by the transform

```
>> T = transl(1, 0, 0)*trrot(y(pi/2));
```

then the Jacobian is given by

```
>> J = tr2jac(T)
J =
    0.0000      0     -1.0000      0      1.0000      0
        0     1.0000      0      0      0     1.0000
    1.0000      0     0.0000      0     -0.0000      0
        0      0      0     0.0000      0     -1.0000
        0      0      0      0     1.0000      0
        0      0      0     1.0000      0     0.0000
```

A unit velocity in the  $x$ -direction of frame  $\{A\}$  is transformed to

```
>> vB = J*[1 0 0 0 0 0]';
>> vB'
ans =
    0.0000      0     1.0000      0      0      0
```

in frame  $\{B\}$ .

### 8.1.2 Jacobian in the End-Effector Coordinate Frame

The Jacobian computed by the method `jacob0` maps joint velocity to the end-effector spatial velocity expressed in the *world coordinate frame* – hence the zero suffix for

the method `jacob0`. To obtain the spatial velocity in the end-effector coordinate frame we use the method `jacobn` instead

```
>> p560.jacobn(qn)
ans =
-0.0000 -0.5963 -0.2910 0 0 0
 0.5963 0.0000 0.0000 0 0 0
 0.1500 0.0144 0.3197 0 0 0
-1.0000 0 0 0.7071 0 0
-0.0000 -1.0000 -1.0000 -0.0000 -1.0000 0
-0.0000 0.0000 0.0000 0.7071 0.0000 1.0000
```

The code for the two Jacobian methods reveals that `jacob0` discussed earlier is actually based on `jacobn` with a velocity transformation from the end-effector frame to the world frame based on the inverse of the  $T_6$  matrix. Starting with Eq. 8.3 we write

$$\begin{aligned} {}^0\nu &= {}^0J_N {}^N\nu \\ &= J_v({}^N T_0) {}^N J(q) \dot{q} \\ &= {}^0J(q) \dot{q} \end{aligned}$$

### 8.1.3 Analytical Jacobian

In Eq. 8.2 the spatial velocity was expressed in terms of translational and angular velocity vectors. It can be more intuitive to consider the rotational velocity in terms of rates of change of roll-pitch-yaw angles or Euler angles. Consider the case of roll-pitch-yaw angles  $\Gamma = (\theta_r, \theta_p, \theta_y)$  for which the rotation matrix is

$$\begin{aligned} R &= R_x(\theta_r)R_y(\theta_p)R_z(\theta_y) \\ &= \begin{pmatrix} c\theta_p c\theta_y & -c\theta_p s\theta_y & s\theta_p \\ c\theta_r s\theta_y + c\theta_y s\theta_p s\theta_r & -s\theta_p s\theta_r s\theta_y + c\theta_r c\theta_y & -c\theta_p s\theta_r \\ s\theta_r s\theta_y - c\theta_r c\theta_y s\theta_p & c\theta_r s\theta_p s\theta_y + c\theta_y s\theta_r & c\theta_p c\theta_r \end{pmatrix} \end{aligned}$$

where we use the shorthand  $c\theta$  and  $s\theta$  to mean  $\cos \theta$  and  $\sin \theta$  respectively. With some tedium we can write the derivative  $\dot{R}$  and recalling Eq. 3.4

$$\dot{R} = S(\omega)R$$

we can solve for  $\omega$  in terms of roll-pitch-yaw angles and rates to obtain

$$\begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} = \begin{pmatrix} s\theta_p \dot{\theta}_y + \dot{\theta}_r \\ -c\theta_p s\theta_r \dot{\theta}_y + c\theta_r \dot{\theta}_p \\ c\theta_p c\theta_r \dot{\theta}_y + s\theta_r \dot{\theta}_p \end{pmatrix}$$

which can be factored as

$$\omega = \begin{pmatrix} 1 & 0 & s\theta_p \\ 0 & c\theta_r & -c\theta_p s\theta_r \\ 0 & s\theta_r & c\theta_p c\theta_r \end{pmatrix} \begin{pmatrix} \dot{\theta}_r \\ \dot{\theta}_p \\ \dot{\theta}_y \end{pmatrix}$$

and written concisely as

$$\omega = B(\Gamma) \dot{\Gamma}$$

This matrix  $B$  is itself a Jacobian that maps roll-pitch-yaw angle rates to angular velocity. It can be computed by the Toolbox function

```
>> rpy2jac(0.1, 0.2, 0.3)
ans =
    1.0000      0     0.1987
        0     0.9950   -0.0978
        0     0.0998    0.9752
```

The analytical Jacobian is

$$J_a(q) = \begin{pmatrix} I_{3 \times 3} & \mathbf{0}_{3 \times 3} \\ \mathbf{0}_{3 \times 3} & B^{-1}(I) \end{pmatrix} J(q)$$

provided that  $B$  is not singular.  $B$  is singular when  $\cos \phi = 0$  or pitch angle  $\phi = \pm \frac{\pi}{2}$  and is referred to as a representational singularity. A similar approach can be taken for Euler angles using the corresponding function `eul2jac`.

The analytical Jacobian can be computed by passing an extra argument to the Jacobian function `jacob0`, for example

```
>> p560.jacob0(qn, 'eul');
```

to specify the Euler angle analytical form.

#### 8.1.4 Jacobian Condition and Manipulability

We have discussed how the Jacobian matrix maps joint rates to end-effector Cartesian velocity but the inverse problem has strong practical use – what joint velocities are needed to achieve a required end-effector Cartesian velocity? We can invert Eq. 8.2 and write

$$\dot{q} = J(q)^{-1} \nu \quad (8.5)$$

provided that  $J$  is square and non singular. This is the basis of a motion control algorithm known as resolved-rate motion control which will be discussed in the next section.

For an  $N$ -link robot the Jacobian is a  $6 \times N$  matrix so a square Jacobian requires a robot with 6 joints. A robot configuration  $q$  at which  $\det(J(q)) = 0$  is described as singular or degenerate. Singularities occur when one or more axes become aligned resulting in the loss of degrees of freedom – the gimbal lock problem again.

For example at the Puma's *ready* pose two of the wrist joints (joints 4 and 6) are aligned resulting in the loss of one degree of freedom. The Jacobian is this case is

```
>> J = p560.jacob0(qr)
J =
    0.1500   -0.8636   -0.4318      0      0      0
    0.0203    0.0000    0.0000      0      0      0
        0    0.0203    0.0203      0      0      0
        0        0        0      0      0      0
        0   -1.0000   -1.0000      0   -1.0000      0
    1.0000    0.0000    0.0000    1.0000    0.0000    1.0000
```

which is rank deficient. The rank is only

```
>> rank(J)
ans =
    5
```

compared to a maximum of six for the  $6 \times 6$  Jacobian. Looking at the Jacobian it is clear that columns 4 and 6 are identical meaning that motion of these joints will result

in the same Cartesian velocity.► The function `jsingu` performs this analysis automatically, for example

```
>> jsingu(J)
1 linearly dependent joints:
q6 depends on: q4
```

indicating velocity of  $q_6$  can be expressed completely in terms of the velocity of  $q_4$ .

However if the robot is close to, but not actually at, a singularity we encounter problems where some Cartesian end-effector velocities require very high joint rates – at the singularity those rates will go to infinity. We can illustrate this by choosing a pose slightly away from `qr` which we just showed was singular. We set  $q_5$  to a small but non-zero value of 5 deg

```
>> q = qr
>> q(5) = 5 * pi/180
q =
    0      1.5708   -1.5708         0      0.0873         0
```

and the Jacobian is now

```
>> J=p560.jacob0(q);
```

To achieve relatively slow end-effector motion of  $0.1 \text{ m s}^{-1}$  in the  $z$ -direction requires

```
>> qd = inv(J)*[0 0 0.1 0 0 0]';
>> qd'
ans =   -0.0000   -4.9261    9.8522    0.0000   -4.9261         0
```

very high-speed motion of the shoulder and elbow – the elbow would have to move at  $9.85 \text{ rad s}^{-1}$  or nearly  $600 \text{ deg s}^{-1}$ . The reason is that although the robot is no longer at a singularity, the determinant of the Jacobian is still very small

```
>> det(J)
ans =
-1.5509e-05
```

Alternatively we can say that its condition number is very high

```
>> cond(J)
ans =
235.2498
```

and the Jacobian is *poorly conditioned*.

However for some motions, such as rotation in this case, the poor condition of the Jacobian is not problematic. If we wished to rotate the tool about the  $y$ -axis then

```
>> qd = inv(J)*[0 0 0 0.2 0]';
>> qd'
ans =    0.0000   -0.0000         0      0.0000   -0.2000         0
```

the required joint rates are very modest.

This leads to the concept of manipulability. Consider the set of joint velocities with a unit norm

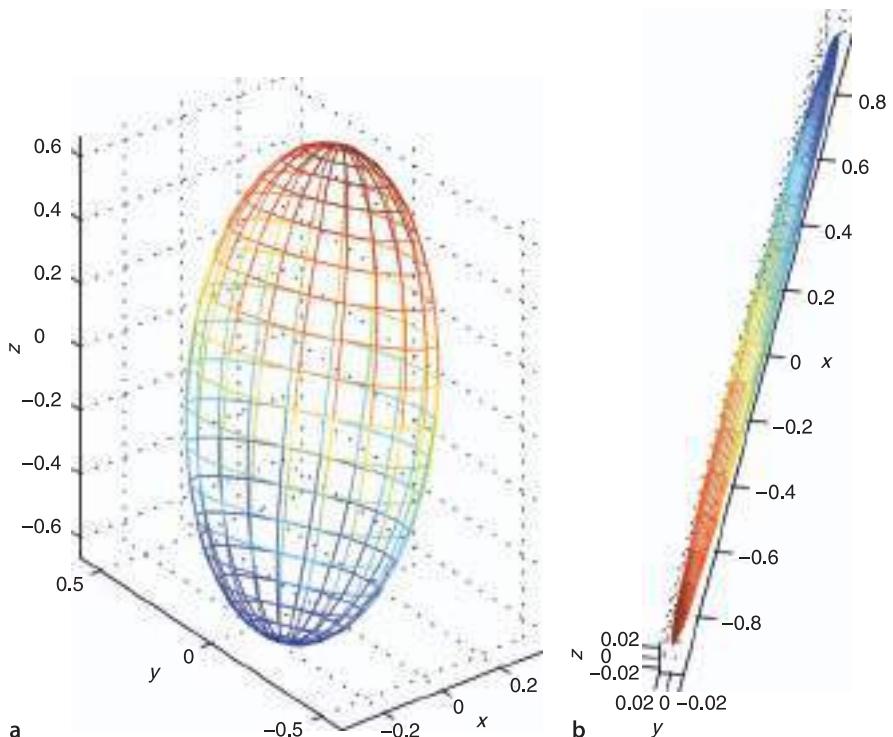
$$\dot{q}^T \dot{q} = 1$$

which lie on the surface of a hypersphere in the  $N$ -dimensional joint velocity space. Substituting Eq. 8.5 we can write

$$\nu^T (J(q)J(q)^T)^{-1} \nu = 1$$

which is the equation of points on the surface of a 6-dimensional ellipsoid in the end-effector velocity space. If this ellipsoid is close to spherical, that is, its radii are of the same order of magnitude then all is well – the end-effector can achieve arbitrary Cartesian velocity. However if one or more radii are very small this indicates that the end-effector cannot achieve velocity in the directions corresponding to those small radii.

For the Puma 560 robot arm joints 4 and 6 are the only ones that can become aligned and lead to singularity. The offset distances,  $d_j$  and  $a_j$ , between links prevents other axes becoming aligned.

**Fig. 8.2.**

End-effector velocity ellipsoids.  
a Translational velocity ellipsoid for the nominal pose; b rotational velocity ellipsoid for a near singular pose, the ellipsoid is an elliptical plate

Since we can only plot three dimensions.

To illustrate this we return the robot to the *nominal* configuration and compute the ellipsoid corresponding to *translational* velocity in the world frame

```
>> J = p560.jacob0(qn);
>> J = J(1:3, :);
```

and plot the corresponding velocity ellipsoid

```
>> plot_ellipse(J'*J')
```

which is shown in Fig. 8.2a. We see that the end-effector can achieve higher velocity in the  $y$ - and  $z$ -directions than in the  $x$ -direction. Ellipses and ellipsoids are discussed in more detail in Appendix E.

The *rotational* velocity ellipsoid for the near singular case

```
>> J = p560.jacob0(qr);
>> J = J(4:6, :);
>> plot_ellipse(J'*J')
```

This is much easier to see if you change the viewpoint interactively.

The radii are the square roots of the eigenvalues of the  $J(q)J(q)^T$  as discussed in Appendix E.

is shown in Fig. 8.2b and is an elliptical plate with almost zero thickness. This indicates an inability to rotate about the direction corresponding to the small radius, which in this case is rotation about the  $x$ -axis. This is the degree of freedom that was lost – both joints 4 and 6 provide rotation about the world  $z$ -axis, joint 5 provides rotation about the world  $y$ -axis, but none allow rotation about the world  $x$ -axis. The shape of the ellipsoid describes how well-conditioned the manipulator is for making certain motions. Manipulability is a succinct scalar measure that describes how spherical the ellipsoid is, for instance the ratio of the smallest to the largest radius. The Toolbox method `maniply` computes Yoshikawa's manipulability measure

$$m = \sqrt{\det(JJ^T)}$$

which is proportional to the volume of the ellipsoid. For example

```
>> p560.maniply(qr, 'yoshikawa')
ans =
0
```

indicates a total lack of manipulability in this pose – the robot is at a singularity. At the nominal pose the manipulability is higher▶

```
>> p560.maniplty(qn, 'yoshikawa')
ans =
0.0786
```

but still not particularly high. In practice we find that the seemingly large workspace of a robot is greatly reduced by joint limits, self collision, singularities and regions of reduced manipulability. This measure is based only on the kinematics of the mechanism and does not take into account mass and inertia – it is easier to move a small wrist joint than the larger waist joint. Other manipulability measures, based on acceleration and inertia, take this into account and are discussed in Sect. 9.1.6.

The manipulability measure combines translational and rotational velocity information which have different units. The options '**T**' and '**R**' can be used to compute manipulability on just the translational or rotational velocity respectively.

## 8.2 Resolved-Rate Motion Control

Resolved-rate motion control exploits Eq. 8.5

$$\dot{q} = J(q)^{-1} \nu$$

to map or *resolve* desired Cartesian velocity to joint velocity without requiring inverse kinematics as we used earlier. For now we will assume that the Jacobian is square ( $6 \times 6$ ) and non-singular but we will relax these constraints later.

The motion control scheme is typically implemented in discrete-time form as

$$\begin{aligned}\dot{q}^*(k) &= J(q(k))^{-1} \nu^* \\ q^*(k+1) &= q(k) + \delta t \dot{q}^*(k)\end{aligned}\quad (8.6)$$

which is essentially an integrator that gives the desired joint angles for the next time step,  $q^*(k+1)$  in terms of the current joint angles and the desired end-effector velocity  $\nu^*$ .

The algorithm is implemented by the Simulink® model

```
>> sl_rrmc
```

shown in Fig. 8.3. The Cartesian velocity is a constant  $0.1 \text{ m s}^{-1}$  in the  $y$ -direction. The **Jacobian** block has as its input the current manipulator joint angles and outputs a  $6 \times 6$  Jacobian matrix. This is inverted and multiplied by the desired velocity to form the desired joint rates. The robot is modelled by an integrator, that is, it acts as a velocity servo.▶

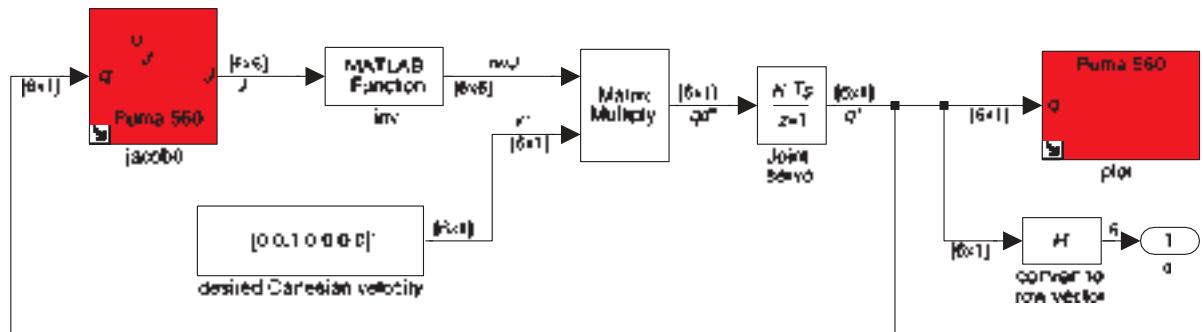
To run the simulation

```
>> r = sim('sl_rrmc');
```

and we see an animation of the manipulator end-effector moving at constant velocity in Cartesian space. Simulation results are returned in the simulation object **r** from which we extract time and joint coordinates

In this model we assume that the robot is perfect, that is, the actual joint angles are equal to the desired joint angles  $q^*$ . The issue of tracking error is discussed in Sect. 9.4.

**Fig. 8.3.** The Simulink® model **sl\_rrmc** for resolved-rate motion control for constant end-effector velocity



```
>> t = r.find('tout');
>> q = r.find('yout');
```

We apply forward kinematics to determine the end-effector position

```
>> T = p560.fkine(q);
>> xyz = transl(T);
```

The function `mplot` is a Toolbox utility that plots columns of a matrix in separate subgraphs.

which we then plot<sup>4</sup> as a function of time

```
>> mplot(t, xyz(:,1:3))
```

which is shown in Fig. 8.4a. The Cartesian motion is  $0.1 \text{ m s}^{-1}$  in the  $y$ -direction as demanded but we observe some small and unwanted motion in the  $x$ - and  $z$ -directions. The motion of the first three joints

```
>> mplot(t, q(:,1:3))
```

is shown in Fig. 8.4b and is not linear with time – reflecting the changing kinematic configuration of the arm.

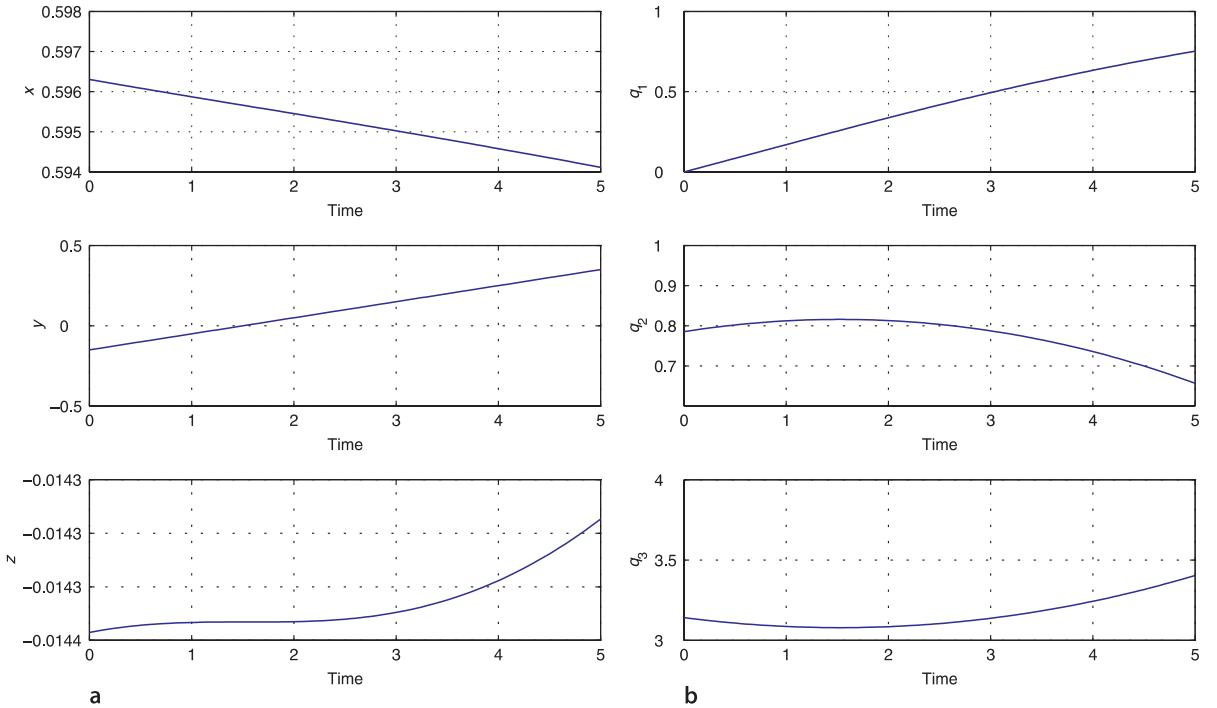
The approach just described, based purely on integration, suffers from an accumulation of error which we observed as the unwanted  $x$ - and  $z$ -direction motion in Fig. 8.4a. We can eliminate this by changing the algorithm to a *closed-loop* form based on the difference between the desired and actual pose

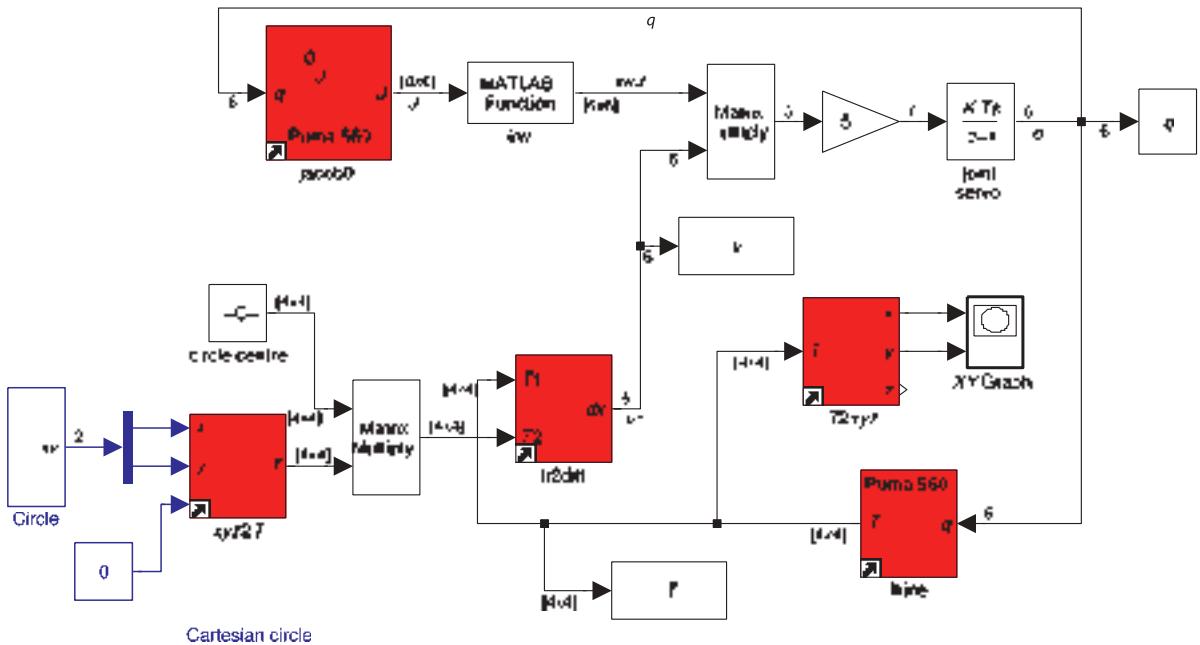
$$\dot{q}^*(k) = J(\mathbf{q}(k))^{-1} (\xi^*(k) \ominus \mathcal{K}(\mathbf{q}(k))) \quad (8.7)$$

$$\mathbf{q}^*(k+1) = \mathbf{q}(k) + K_p \delta t \dot{q}^*(k)$$

**Fig. 8.4.** Resolved-rate motion control, Cartesian and joint coordinates versus time. **a** Cartesian end-effector position; **b** joint coordinates

where  $K_p$  is a proportional gain. The input is now the desired pose  $\xi^*(k)$  as a function of time rather than  $\nu^*$ . The current pose is determined by forward kinematics based on the measured joint coordinates. The difference between two poses is a 6-vector computed using the function  $\Delta(\cdot)$  given by Eq. 3.10 and implemented by the Toolbox function `tr2delta`.





A Simulink® example to demonstrate this for a circular path is

```
>> sl_rrmc2
```

shown in Fig. 8.5. The tool of a Puma 560 robot traces out a circle of radius 50 mm. The  $x$ -,  $y$ - and  $z$ -coordinates as a function of time are computed by the blue colored blocks and converted to a homogenous transformation. The difference between the Cartesian demand and the current Cartesian pose is computed by the `tr2delta` block which produces a Cartesian differential, or scaled spatial velocity, described by a 6-vector. The Jacobian block has as its input the current manipulator joint angles and outputs the Jacobian matrix. The result, after application of a proportional gain, is the joint-space motion required to correct the Cartesian error.

So far we have assumed that the Jacobian is square. For the non-square cases it is helpful to consider the velocity relationship

$$\nu = J(q)\dot{q}$$

in the diagrammatic form shown in Fig. 8.6. The Jacobian is a  $6 \times N$  matrix, the joint velocity is an  $N$ -vector, and  $\nu$  is a 6-vector.

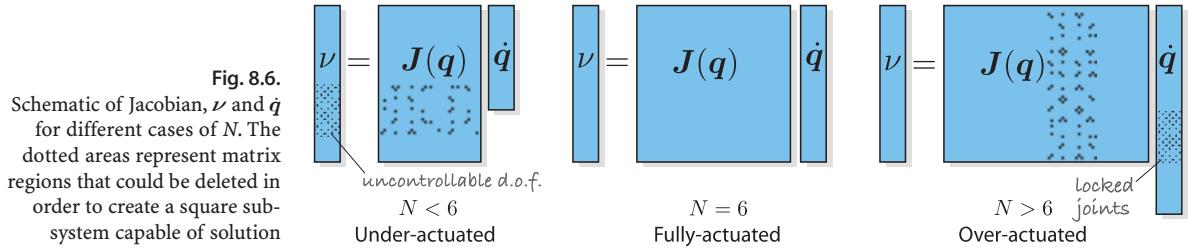
The case of  $N < 6$  is referred to as under-actuated robot, and  $N > 6$  is over-actuated or redundant. The under-actuated case cannot be solved because the system of equations is under-constrained but the system can be *squared up* by deleting some rows of  $\nu$  and  $J$  – accepting that some Cartesian degrees of freedom are not controllable given the low number of joints. For the over-actuated case the system of equations is over-constrained and we could find a least squares solution. Alternatively we can *square up* the Jacobian to make it invertible by deleting some columns – effectively *locking* the corresponding axes.

Fig. 8.5. The Simulink® model `sl_rrmc2` for closed-loop resolved-rate motion control with circular end-effector motion

### 8.2.1 Jacobian Singularity

For the case of a square Jacobian where  $\det(J(q)) = 0$  we cannot solve Eq. 8.5 directly. One simple strategy to deal with singularity is to replace the inverse with the damped inverse Jacobian

$$\dot{q} = (J(q) + pI)^{-1}\nu$$



where  $p$  is a small constant added to the diagonal which places a *floor* under the determinant. However this will introduce some error in  $\dot{q}$ , which integrated over time could lead to a significant discrepancy in tool position. The closed-loop resolved-rate motion scheme of Eq. 8.7 would minimize any such error.

The pseudo-inverse of the Jacobian  $J^+$  has the property that

$$J^+ J = I$$

just as the inverse does, and is defined as

$$J^+ = (J^T J)^{-1} J^T$$

This is the left generalized- or pseudo-inverse, see Appendix D for more details.

$$\dot{q} = J(q)^+ \nu$$

provides a least squares solution for which  $|J\dot{q} - \nu|$  is smallest.

Yet another approach is to delete from the Jacobian all those columns that are linearly dependent on other columns. This is effectively locking the joints corresponding to the deleted columns and we now have an underactuated system which we treat as per the next section.

## 8.2.2 Jacobian for under-Actuated Robot

An under-actuated robot has  $N < 6$ , and a Jacobian that is taller than it is wide. For example the two-link manipulator from Sect. 7.3.3 at a nominal pose

```
>> mdl_twolink
>> qn = [1 1];
```

has the Jacobian

```
>> J = jacob0(twolink, qn)
J =
-1.7508   -0.9093
 0.1242   -0.4161
   0         0
   0         0
   0         0
  1.0000    1.0000
```

We cannot solve the inverse problem Eq. 8.5 using the pseudo-inverse since it will attempt to satisfy motion constraints that the manipulator cannot meet. For example the desired motion of  $0.1 \text{ m s}^{-1}$  in the  $x$ -direction gives the required joint velocity

```
>> qd = pinv(J) * [0.1 0 0 0 0 0]'
qd =
-0.0698
 0.0431
```

which results in end-effector velocity

```
>> xd = J*qd;
ans =
0.0829 -0.0266 0 0 0 -0.0266
```

This has the desired motion in the  $x$ -direction but undesired motion in  $y$ -axis translation and  $z$ -axis rotation. The end-effector rotation cannot be independently controlled (since it is a function of  $q_1$  and  $q_2$ ) yet this solution has taken it into account in the least squares solution.

We have to confront the reality that we have *only* two degrees of freedom which we will use to control just  $v_x$  and  $v_y$ . We rewrite Eq. 8.2 in partitioned form as

$$\begin{pmatrix} v_x \\ v_y \\ \hline v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} = \begin{pmatrix} J_{xy} & J_0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \end{pmatrix}$$

and taking the top partition, the first two rows, we write

$$\begin{pmatrix} v_x \\ v_y \end{pmatrix} = J_{xy} \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \end{pmatrix}$$

where  $J_{xy}$  is a  $2 \times 2$  matrix. We invert this

$$\begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \end{pmatrix} = J_{xy}^{-1} \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

which we can solve if  $\det(J_{xy}) \neq 0$ .

```
>> Jxy = J(1:2,:);
>> qd = inv(Jxy)* [0.1 0]';
qd =
-0.0495
-0.0148
```

which results in end-effector velocity

```
>> xd = J*qd;
>> xd'
ans =
0.1000 0.0000 0 0 0 -0.0642
```

We have achieved the desired  $x$ -direction motion with no unwanted motion apart from the  $z$ -axis rotation which is unavoidable – we have used the two degrees of freedom to control  $x$ - and  $y$ -translation, not  $z$ -rotation.

### 8.2.3 Jacobian for over-Actuated Robot

An over-actuated or redundant robot has  $N > 6$ , and a Jacobian that is wider than it is tall. In this case we rewrite Eq. 8.5 to use the left pseudo-inverse

$$\dot{q} = J(q)^+ \nu \quad (8.8)$$

which, of the infinite number of solutions possible, will yield the one for which  $|\dot{q}|$  is smallest – the minimum-norm solution.

For example, consider the 8-axis P8 robot from Sect. 7.3.4 at a nominal pose

```
>> qn8 = [0 0 qn];
```

and its Jacobian

```
>> J = jacob0(p8, qn8);
>> about(J)
J [double] : 6x8 (384 bytes)
```

is a  $6 \times 8$  matrix. Now consider that we want the end-effector to move at  $0.2 \text{ m s}^{-1}$  in the  $x$ -,  $y$ - and  $z$ -directions. Using Eq. 8.8 we compute the required joint rates

```
>> xd = [0.2 0.2 0.2 0 0 0]';
>> q = pinv(J) * xd;
>> q'
ans =
    0.1801    0.1800    0.0336    0.3197    0.0322    0.0475   -0.3519   -0.0336
```

We see that all eight joints have non-zero velocity and contribute to the desired end-effector motion. If the robot follows a repetitive path the joint angles may *drift* over time, that is they may not follow a repetitive path, potentially moving toward joint limits. We can use null-space control to provide additional constraints.

The Jacobian has eight columns and a rank of six

```
>> rank(J)
ans =
    6
```

and a null-space whose basis has two columns

```
>> N = null(J)
N =
    0.2543    -0.0320
    0.1086    0.2635
   -0.1821   -0.4419
    0.3543   -0.1534
   -0.7260    0.3144
   -0.2576   -0.6250
    0.3718   -0.1610
    0.1821    0.4419
```

These columns are orthogonal vectors that span the null-space, that is, any joint velocity that is a linear combination of these two column vectors will result in *no* end-effector motion. We can demonstrate this by

```
>> norm( J * (N(:,1) + N(:,2)) )
ans =
    5.7168e-16
```

This is remarkably useful because it allows Eq. 8.8 to be written as

$$\dot{q} = \underbrace{J(q)^+ \nu}_{\text{end-effector motion}} + \underbrace{NN^+ \dot{q}_{ns}}_{\text{null-space motion}} \quad (8.9)$$

where the  $N \times N$  matrix  $NN^+$  projects the desired joint motion into the null-space so that it will not affect the end-effector Cartesian motion, allowing the two motions to be superimposed.

Null-space motion can be used for highly-redundant robots to avoid collisions between the links and obstacles (including other links), or to keep joint coordinates away from their mechanical limit stops. Consider that in addition to the desired Cartesian velocity  $xd$  we wish to simultaneously move joint 5 (the Puma's elbow joint) angle closer to zero (it is currently  $\pi$ ), so we set a desired joint velocity

```
>> qd_ns = [0 0 0 0 -0.1 0 0 0]';
```

and project it into the null-space

```
>> qp = N * pinv(N) * qd_ns;
>> qp'
0.0195 -0.0004 0.0007 0.0305 -0.0626 0.0009 0.0321 -0.0007
```

The projection has introduced a scaling, the joint 5 velocity is not the  $-0.1$  we desired but we can apply a scale factor to correct this

```
>> qp = qp / qp(5) * qd_ns(5)
qp =
0.0312 -0.0006 0.0011 0.0487 -0.1000 0.0014 0.0513 -0.0011
```

The other joint velocities provide the required compensating motion in order that the end-effector pose is not disturbed as shown by

```
>> norm(J * qp)
ans =
3.1107e-17
```

## 8.3 Force Relationships

Forces and wrenches are properly the subject of the next chapter, about dynamics, but it is helpful now to introduce another very useful property of the Jacobian.

In the earlier discussion of motion we introduced the concept of a spatial velocity  $\nu = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$ . For forces there is a spatial equivalent called a wrench  $g = (f_x, f_y, f_z, m_x, m_y, m_z) \in \mathbb{R}^6$  which is a vector of forces and moments.

### 8.3.1 Transforming Wrenches between Frames

Just as the Jacobian transforms a spatial velocity from one coordinate frame to another using Eq. 8.3 it can be used to transform wrenches between coordinate frames

$${}^B g = ({}^A J_B)^T {}^A g \quad (8.10)$$

where  ${}^A J_B$  is given by Eq. 8.4 and is a function of the relative pose  ${}^A T_B$  from frame  $\{A\}$  to frame  $\{B\}$ . Note that the force transform differs from the velocity transform in using the transpose rather than the inverse of the Jacobian, and is therefore never singular.

For example consider two frames, displaced by 2 m in the  $x$ -direction. We create the Jacobian

```
>> J = tr2jac(transl(2, 0, 0));
```

Then a force of 3 N in the  $y$ -direction of the first frame is transformed to

```
>> F = J'*[0 3 0 0 0 0];
>> F'
ans =
0 3 0 0 0 6
```

a force of 3 N force in the  $y$ -direction *plus* a moment of 6 N m about the  $z$ -axis in the second frame due to a *lever arm* effect.

### 8.3.2 Transforming Wrenches to Joint Space

The manipulator Jacobian transforms joint velocity to an end-effector spatial velocity according to Eq. 8.2 and the Jacobian transpose transforms a wrench applied at the end-effector to torques and forces experienced at the joints

$$Q = {}^0 J(q)^T {}^0 g \quad (8.11)$$

Derived through the principle of virtual work, see for instance Spong et al. (2006, sect. 4.10).

where  $\mathbf{g}$  is a wrench in the world coordinate frame and  $\mathbf{Q}$  is the generalized joint force vector. The elements of  $\mathbf{Q}$  are joint torque or force for revolute or prismatic joints respectively.

If the wrench is defined in the end-effector coordinate frame then we use instead

$$\mathbf{Q} = {}^N\mathbf{J}(\mathbf{q})^T {}^N\mathbf{g} \quad (8.12)$$

Interestingly this mapping from external quantities (the wrench) to joint quantities (the generalized forces) can never be singular as it can be for velocity. We exploit this property in the next section to solve the inverse kinematic problem numerically.

For the Puma 560 robot in its nominal pose, see Fig. 8.1, a force of 20 N in the world  $y$ -direction results in joint torques of

```
>> tau = p560.jacob0(qn)' * [0 20 0 0 0 0]';
>> tau'
ans =
11.9261    0.0000    0.0000         0         0         0
```

The force pushes the arm *sideways* and only the waist joint will rotate in response – experiencing a torque of 11.93 Nm due to a lever arm effect. A force of 20 N applied in the world  $x$ -direction results in joint torques of

```
>> tau = p560.jacob0(qn)' * [20 0 0 0 0 0]';
>> tau'
ans =
3.0010    0.2871    6.3937         0         0         0
```

which is pulling the end-effector away from the base which results in torques being applied to the first three joints.

## 8.4 Inverse Kinematics: a General Numerical Approach

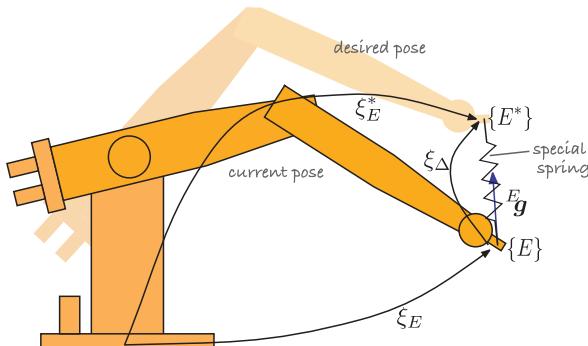
In Sect. 7.3 we solved the inverse kinematic problem using an explicit solution that required the robot to have 6 joints and a spherical wrist. For the case of robots which do not meet this specification, for example those with more or less than 6 joints, we need to consider a numerical solution. Here we will develop an approach based on the forward kinematics and the Jacobian transpose which we can compute for any manipulator configuration – these functions have no singularities.

The principle is shown in Fig. 8.7. The virtual robot is drawn solidly in its current pose and faintly in the desired pose. From the overlaid pose graph we write

$$\xi_E^* = \xi_E \oplus \xi_\Delta$$

which we can rearrange as

$$\xi_\Delta = \ominus \xi_E \oplus \xi_E^*$$



**Fig. 8.7.**

Schematic of the numerical inverse kinematic approach, showing the current  $\xi_E$  and the desired  $\xi_E^*$  manipulator pose

We postulate a *special* spring between the end-effector of the two poses which is pulling (and twisting) the robot's end-effector toward the desired pose<sup>►</sup> with a wrench proportional to the *difference* in pose

$${}^E\mathbf{g} \propto \Delta(\xi_E, \xi_E^*)$$

where the function  $\Delta(\cdot)$  given by Eq. 3.10 approximates the difference between two poses as a 6-vector comprising translational and rotational displacement.<sup>►</sup> The wrench is also a 6-vector and comprises forces and moments. We write

$${}^E\mathbf{g} = \gamma \Delta(\xi_E, \xi_E^*) \quad (8.13)$$

where  $\gamma$  is a constant and the current pose is computed using forward kinematics

$$\xi_E(k) = \mathcal{K}(\mathbf{q}(k)) \quad (8.14)$$

where  $\mathbf{q}(k)$  is the current estimate of the inverse kinematic solution.

The end-effector wrench Eq. 8.13 is *resolved* to joint forces

$$\mathbf{Q}(k) = {}^N\mathbf{J}(\mathbf{q}(k))^T {}^E\mathbf{g}(k) \quad (8.15)$$

using the Jacobian transpose Eq. 8.12. We assume that the virtual robot has no joint motors only viscous dampers so the joint velocity due to the applied forces will be proportional

$$\dot{\mathbf{q}}(k) = \mathbf{Q}(k)/B$$

where  $B$  is the joint damping coefficients (we assume all dampers are the same). Now we can write a discrete-time update for the joint coordinates

$$\mathbf{q}(k+1) = \alpha \dot{\mathbf{q}}(k) + \mathbf{q}(k) \quad (8.16)$$

where  $\alpha$  is some well chosen gain. We iterate Eq. 8.13 to Eq. 8.16 until the magnitude of the wrench  ${}^E\mathbf{g}$  is sufficiently small. Using the Jacobian transpose we do not face the problem of having to invert a Jacobian which is potentially non-square or singular.

In Section 7.3.3 we used a mask vector when computing the inverse kinematics of a robot with  $N < 6$ . The mask vector  $\mathbf{m}$  can be included in Eq. 8.15 which becomes

$$\mathbf{Q}(k) = {}^N\mathbf{J}(\mathbf{q}(k))^T \text{diag}(\mathbf{m}) {}^E\mathbf{g}(k) \quad (8.17)$$

It has to be a *special* spring in order to change the orientation as well as the position of the end-effector.

This assumes that the difference in orientation between the two poses is "small" so the approximation becomes better as the solution converges.

## 8.5 Wrapping Up

In this chapter we have learnt about Jacobians and their application to robotics. The manipulator Jacobian captures the relationship between the rate of change of joint coordinates and the spatial velocity of the end-effector. The numerical properties of the Jacobian tell us about manipulability, that is how well the manipulator is able to move in different directions. The extreme case, singularity, is indicated by linear dependence between columns of the Jacobian. We showed how the inverse Jacobian can be used to resolve desired Cartesian velocity into joint velocity as an alternative means of generating Cartesian paths for under- and over-actuated robots. For over-actuated robots we showed how null-space motions can be used to move the robot's joints without affecting the end-effector pose.

We created other Jacobians as well. A Jacobian can be used to map spatial velocities between coordinate frames. The analytic Jacobian maps angular velocity to roll-pitch-yaw or Euler angle rates.

The Jacobian transpose is used to map wrenches applied at the end-effector to joint torques, and also to map wrenches between coordinate frames. We showed, finally, how to use the Jacobian transpose and forward kinematics to compute inverse kinematics numerically for arbitrary robots and singular poses.

---

### Further Reading

The manipulator Jacobian is covered by almost all standard robotics texts such as Spong et al. (2006), Craig (2004), Siciliano et al. (2008), Paul (1981), and the handbook (Siciliano and Khatib 2008, § 1). An excellent discussion of manipulability and velocity ellipsoids is provided by Siciliano et al. (2008), and the most common manipulability measure is that proposed by Yoshikawa (1984). Computing the manipulator Jacobian based on Denavit-Hartenberg was first described by Paul and Shimano (1978).

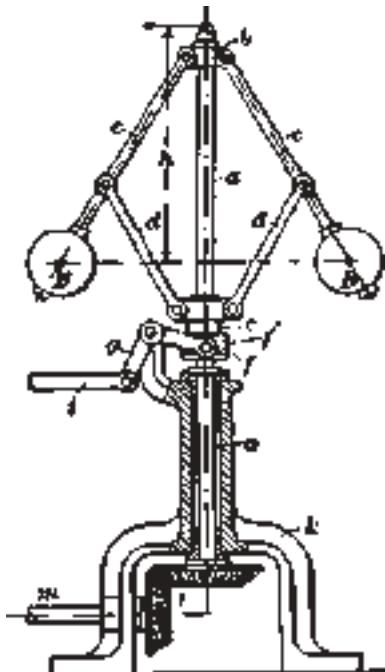
The resolved-rate motion control scheme was proposed by Whitney (1969). Extensions such as pseudo-inverse Jacobian-based control are reviewed by Klein and Huang (1983) and damped least-square methods are reviewed by Deo and Walker (1995). The approach to numeric inverse kinematics used in the Toolbox is based on results for control through singularities (Chiaverini et al. 1991).

---

### Exercises

1. For the Puma 560 robot can you devise a configuration in which three joint axes are parallel?
2. Derive the analytical Jacobian for Euler angles.
3. Manipulability (page 177)
  - a) Plot the velocity ellipse ( $x$ - and  $y$ -velocity) for the two-link manipulator at a grid of end-effector positions in its workspace. Each ellipsoid should be centred on the end-effector position.
  - b) For the Puma 560 manipulator find a configuration where manipulability is greater than at `qn`.
  - c) Overlay the translational or rotational velocity ellipsoid on the manipulator as displayed by the `plot` method, and create an animation that shows how it changes shape as the robot moves along a trajectory.
4. Resolved-rate motion control (page 180)
  - a) Experiment with different Cartesian translational and rotational velocity demands, and combinations.
  - b) Extend the Simulink® system of Fig. 8.4 to also record the determinant of the Jacobian matrix to the workspace.
  - c) In Fig. 8.4 the robot's motion is simulated for 5 s. Extend the simulation time to 10 s and explain what happens.
  - d) Set the initial pose and direction of motion to mimic that of Sect. 7.4.3. What happens when the robot reaches the singularity?
  - e) Replace the Jacobian inverse block in Fig. 8.3 with the MATLAB® function `pinv`.
  - f) Replace the Jacobian inverse block in Fig. 8.3 with a damped least squares function, and investigate the effect of different values of the damping factor.
  - g) Replace the Jacobian inverse block in Fig. 8.3 with a block based on the MATLAB® function `lscov`.
  - h) Modify the simulation of Fig. 8.5 to use the 8-axis robot from Sect. 7.3.4. Observe the joint coordinate trajectory, is it repetitive like the end-effector trajectory?

- i) Modify the above to include null-space motion to keep the robot joints away from their limits.
5. For the over-actuated P8 robot (page 150)
  - a) Develop a null-space controller that keeps the last six joints in the middle of their working range by using the first two joints to position the base of the Puma. Modify this so as to maximize the manipulability of the P8 robot. Consider now that the Puma robot is mounted on a non-holonomic robot, create a controller that generates appropriate steering and velocity inputs to the mobile robot (challenging).
  - b) For an arbitrary pose and end-point spatial velocity we will move six joints and lock two joints. Write an algorithm to determine which two joints should be locked.



In this chapter we consider the dynamics and control of a serial-link manipulator. Each link is supported by a reaction force and torque from the preceding link, and is subject to its own weight as well as the reaction forces and torques from the links that it supports.

Section 9.1 introduces the equations of motion, a set of coupled dynamic equations, that describe the joint torques necessary to achieve a particular manipulator state. The equations contains terms for inertia, gravity and gyroscopic coupling. The equations of motion provide insight into important issues such as how the motion of one joint exerts a disturbance force on other joints, how inertia and gravity load varies with configuration, and the effect of payload mass. Section 9.2 introduces real-world drive train issues such as gearing and friction. Section 9.3 introduces the forward dynamics which describe how the manipulator moves, that is, how its configuration evolves with time in response to forces and torques applied at the joints by the actuators, and by external forces such as gravity. Section 9.4 introduces control systems that compute the joint forces so that the robot end-effector follows a desired trajectory despite varying dynamic characteristics or joint flexibility.

### 9.1 Equations of Motion

Consider the motor which actuates the  $j^{\text{th}}$  revolute joint of a serial-link manipulator. From Fig. 7.2 we recall that joint  $j$  connects link  $j - 1$  to link  $j$ . The motor exerts a torque that causes the outward link,  $j$ , to rotationally accelerate but it also exerts a reaction torque on the inward link  $j - 1$ . Gravity acting on the outward links  $j$  to  $N$  exert a weight force, and rotating links also exert gyroscopic forces on each other. The inertia that the motor *experiences* is a function of the configuration of the outward links.

The situation at the individual link is quite complex but for the *series* of links the result can be written elegantly and concisely as a set of coupled differential equations in matrix form

$$\mathbf{Q} = \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{F}(\dot{\mathbf{q}}) + \mathbf{G}(\mathbf{q}) + \mathbf{J}(\mathbf{q})^T \mathbf{g} \quad (9.1)$$

where  $\mathbf{q}$ ,  $\dot{\mathbf{q}}$  and  $\ddot{\mathbf{q}}$  are respectively the vector of generalized joint coordinates, velocities and accelerations,  $\mathbf{M}$  is the joint-space inertia matrix,  $\mathbf{C}$  is the Coriolis and centripetal coupling matrix,  $\mathbf{F}$  is the friction force,  $\mathbf{G}$  is the gravity loading, and  $\mathbf{Q}$  is the vector of generalized actuator forces associated with the generalized coordinates  $\mathbf{q}$ . The last term gives the joint forces due to a wrench  $\mathbf{g}$  applied at the end effector and  $\mathbf{J}$  is the manipulator Jacobian. This equation describes the manipulator rigid-body dynamics and is known as the inverse dynamics – given the pose, velocity and acceleration it computes the required joint forces or torques.

These equations can be derived using any classical dynamics method such as Newton's second law and Euler's equation of motion or a Lagrangian energy-based

approach. A very efficient way for computing Eq. 9.1 is the recursive Newton-Euler algorithm which starts at the base and working outward adds the velocity and acceleration of each joint in order to determine the velocity and acceleration of each link. Then working from the tool back to the base, it computes the forces and moments acting on each link and thus the joint torques.► The recursive Newton-Euler algorithm has  $O(N)$  complexity and can be written in functional form as

$$Q = \mathcal{D}(q, \dot{q}, \ddot{q}) \quad (9.2)$$

In the Toolbox it is implemented by the `rne` method of the `SerialLink` object. Consider the Puma 560 robot

```
>> mdl_puma560
```

at the nominal pose, and with zero joint velocity and acceleration. The generalized joint forces, or joint torques in this case, are

```
>> Q = p560.rne(qn, qz, qz)
Q =
-0.0000    31.6399     6.0351     0.0000     0.0283      0
```

Since the robot is not moving (we specified  $\dot{q} = \ddot{q} = 0$ ) these torques must be those required to *hold the robot up* against gravity. We can confirm this by computing the torques in the absence of gravity

```
>> Q = p560.rne(qn, qz, qz, [0 0 0]')
ans =
  0      0      0      0      0      0
```

where the last argument overrides the object's default gravity vector.

Like most Toolbox methods `rne` can operate on a trajectory

```
>> q = jtraj(qz, qr, 10)
>> Q = p560.rne(q, 0*q, 0*q)
```

which has returned

```
>> about(Q)
Q [double] : 10x6 (480 bytes)
```

a  $10 \times 6$  matrix with each row representing the generalized force for the corresponding row of `q`. The joint torques corresponding to the fifth time step is

```
>> Q(5,:)
ans =
  0.0000    29.8883     0.2489      0      0      0
```

Consider now a case where the robot is moving. It is *instantaneously* at the nominal pose but joint 1 is moving at  $1 \text{ rad s}^{-1}$  and the acceleration of all joints is zero. Then in the absence of gravity, the joint torques

```
>> p560.rne(qn, [1 0 0 0 0], qz, [0 0 0])
-24.8240     0.6280    -0.3607   -0.0003   -0.0000      0
```

**Dynamics in 3D.** The dynamics of an object moving in 3 dimensions is described by two important equations. The first equation, Newton's second law, describes the translational motion in 3D

$$\mathbf{f} = m\mathbf{\dot{v}}$$

where  $m$  is the mass,  $\mathbf{f}$  the applied force and  $\mathbf{v}$  the velocity. The second equation, Euler's equation of motion, describes the rotational motion

$$\boldsymbol{\tau} = \mathbf{J}\boldsymbol{\dot{\omega}} + \boldsymbol{\omega} \times \mathbf{J}\boldsymbol{\omega}$$

where  $\boldsymbol{\tau}$  is the torque,  $\boldsymbol{\omega}$  is the angular velocity, and  $\mathbf{J}$  is the rotational inertia matrix (see page 81).

The recursive form of the inverse dynamics does not explicitly calculate the matrices  $M$ ,  $C$  and  $G$  of Eq. 9.1. However we can use the recursive Newton-Euler algorithm to calculate these matrices and the Toolbox functions `inertia` and `coriolis` use Walker and Orin's (1982) '*Method 1*'. Whilst the recursive forms are computationally efficient for the inverse dynamics, to compute the coefficients of the individual dynamic terms ( $M$ ,  $C$  and  $G$ ) in Eq. 9.1 is quite costly –  $O(N^3)$  for an  $N$ -axis manipulator.

**Gyroscopic motion.** A spinning disc has an angular momentum  $\mathbf{h} = J\omega$ . If a torque  $\tau$  is applied to the gyroscope it rotates about an axis perpendicular to both  $\tau$  and  $\mathbf{h}$  with an angular velocity  $w_p$  known as precession. These quantities are related by

$$\tau = \omega_p \times \mathbf{h}$$

If you've ever tried to rotate the axis of a spinning bicycle wheel you will have observed this effect – “torquing” it about one axis causes it to turn in your hands in an orthogonal direction.

A strapdown gyroscopic sensor contains a high-speed flywheel which has a large  $\mathbf{h}$ . When the gyroscope is rotated a gyroscopic force is generated proportional to  $w_p$  which is measured by a force sensor.

are non zero. The torque on joint 1 is due to friction and opposes the direction of motion. More interesting is that torques have been exerted on joints 2, 3 and 4. These are gyroscopic effects (centripetal and Coriolis forces) and are referred to as velocity coupling torques since the rotational velocity of one joint has induced a torque on several other joints.

The elements of the matrices  $M$ ,  $C$ ,  $F$  and  $G$  are complex functions of the link's kinematic parameters ( $\theta_j$ ,  $d_j$ ,  $a_j$ ,  $c_j$ ) and inertial parameters. Each link has ten independent inertial parameters: the link mass  $m_j$ ; the centre of mass (COM)  $r_j$  with respect to the link coordinate frame; and six second moments which represent the inertia of the link about the COM but with respect to axes aligned with the link frame  $\{j\}$ , see page 81. We can view the dynamic parameters of a robot's link by

```
>> p560.links(1).dyn
l =
theta=q, d=0, a=0, alpha=1.571 (R,stdDH)
m    = 0.000000
r    = 0.000000 0.000000 0.000000
J    = | 0.000000 0.000000 0.000000 |
      | 0.000000 0.350000 0.000000 |
      | 0.000000 0.000000 0.000000 |
Jm   = 0.000200
Bm   = 0.001480
Tc   = 0.395000(+) -0.435000(-)
G    = -62.611100
```

which in order are: the kinematic parameters, link mass, COM position, inertia matrix, motor inertia, motor friction, Coulomb friction and gear ratio.

The remainder of this section examines the various matrix components of Eq. 9.1.

### 9.1.1 Gravity Term

$$Q = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + F(\dot{q}) + G(q) + J(q)^T f$$

We start our detailed discussion with the gravity term because it is generally the dominant term in Eq. 9.1 and is present even when the robot is stationary or moving slowly. Some robots use counterbalance weights or even springs to reduce the *gravity* torque that needs to be provided by the motors – this allows the motors to be smaller and thus lower in cost.

In the previous section we used the `rne` method to compute the gravity load by setting the joint velocity and acceleration to zero. A more convenient approach is to use the `gravload` method

```
>> gravload = p560.gravload(qn)
gravload =
-0.0000    31.6399     6.0351     0.0000     0.0283          0
```

Counterbalancing will however increase the inertia associated with a joint since it adds additional mass at the end of a lever arm, and increase the overall mass of the robot.

Sir Isaac Newton (1642–1727) was an English mathematician and alchemist. He was Lucasian professor of mathematics at Cambridge, Master of the Royal Mint, and the thirteenth president of the Royal Society. His achievements include the three laws of motion, the mathematics of gravitational attraction, the motion of celestial objects and the theory of light and color (see page 224), and building the first reflecting telescope.

Many of these results were published in 1687 in his great 3-volume work “The Philosophiae Naturalis Principia Mathematica” (Mathematical principles of natural philosophy). In 1704 he published “Opticks” which was a study of the nature of light and colour and the phenomena of diffraction. The SI unit of force is named in his honour. He is buried in Westminster Abbey, London.



The `SerialLink` object contains a default gravitational acceleration vector which is initialized to the nominal value for Earth<sup>4</sup>

```
>> p560.gravity
ans =
      0          0       9.8100
```

We could change gravity to the lunar value

```
>> p560.gravity = p560.gravity/6;
```

resulting in reduced joint torques

```
>> p560.gravload(qn)
ans =
      0.0000    5.2733   1.0059    0.0000    0.0047        0
```

or we could turn our lunar robot upside down

```
>> p560.base = trotx(pi);
>> p560.gravload(qn)
ans =
      0.0000   -5.2733   -1.0059   -0.0000   -0.0047        0
```

and see that the torques have changed sign. Before proceeding we bring our robot back to Earth and right-side up

```
>> mdl_puma560
```

The torque exerted on a joint due to gravity acting on the robot depends very strongly on the robot’s pose. Intuitively the torque on the shoulder joint is much greater when the arm is stretched out horizontally

```
>> Q = p560.gravload(qs)
Q =
      -0.0000    46.0069    8.7722    0.0000    0.0283        0
```

than when the arm is pointing straight up

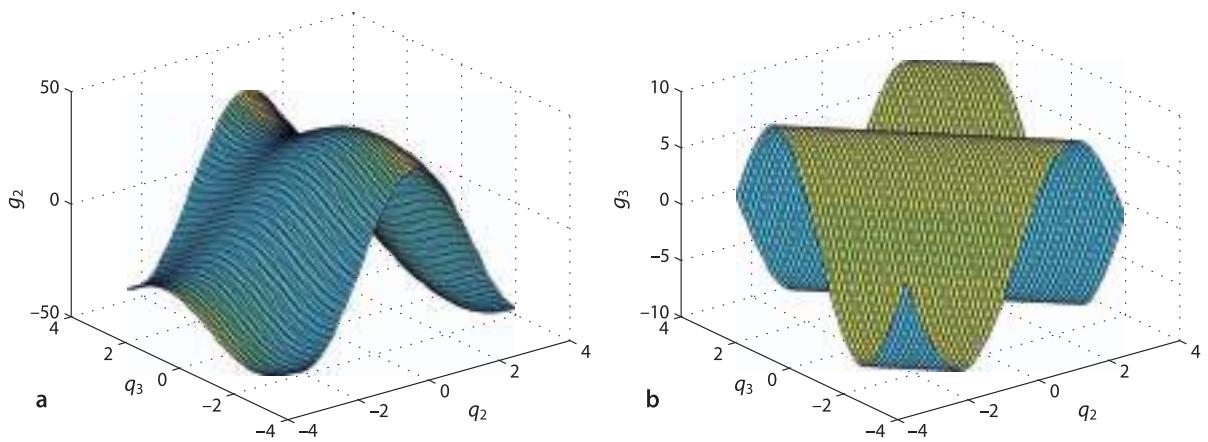
```
>> Q = p560.gravload(qr)
Q =
      0   -0.7752    0.2489        0        0        0
```

The gravity torque on the elbow is also very high in the first pose since it has to support the lower arm and the wrist. We can investigate how the gravity load on joints 2 and 3 varies with joint configuration by

```
1 [Q2,Q3] = meshgrid(-pi:0.1:pi, -pi:0.1:pi);
2 for i=1:numcols(Q2),
3     for j=1:numcols(Q3);
4         g = p560.gravload([0 Q2(i,j) Q3(i,j) 0 0 0]);
5         g2(i,j) = g(2);
6         g3(i,j) = g(3);
7     end
8 end
9 surf(Q2, Q3, g2); surf(Q2, Q3, g3);
```

and the results are shown in Fig. 9.1. The gravity torque on joint 2 varies between  $\pm 40$  N m and for joint 3 varies between  $\pm 10$  N m. This type of analysis is very important in robot design to determine the required torque capacity for the motors.

The ‘`gravity`’ option for the `SerialLink` constructor can change this.



**Fig. 9.1.** Gravity load variation with manipulator pose. **a** Shoulder gravity load,  $g_2(q_2, q_3)$ ; **b** elbow gravity load  $g_3(q_2, q_3)$

### 9.1.2 Inertia Matrix

$$Q = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + F(\dot{q}) + G(q) + J(q)^T f$$

This inertia matrix includes the motor inertias, which are added to the diagonal elements. Motor and rigid-body inertia are discussed further in Sect. 9.4.2.

The inertia matrix<sup>►</sup> is a function of the manipulator pose

```
>> M = p560.inertia(qn)
M =
    3.6594   -0.4044    0.1006   -0.0025    0.0000   -0.0000
   -0.4044    4.4137    0.3509    0.0000    0.0024    0.0000
    0.1006    0.3509    0.9378    0.0000    0.0015    0.0000
   -0.0025    0.0000    0.0000    0.1925    0.0000    0.0000
    0.0000    0.0024    0.0015    0.0000    0.1713    0.0000
   -0.0000    0.0000    0.0000    0.0000    0.0000    0.1941
```

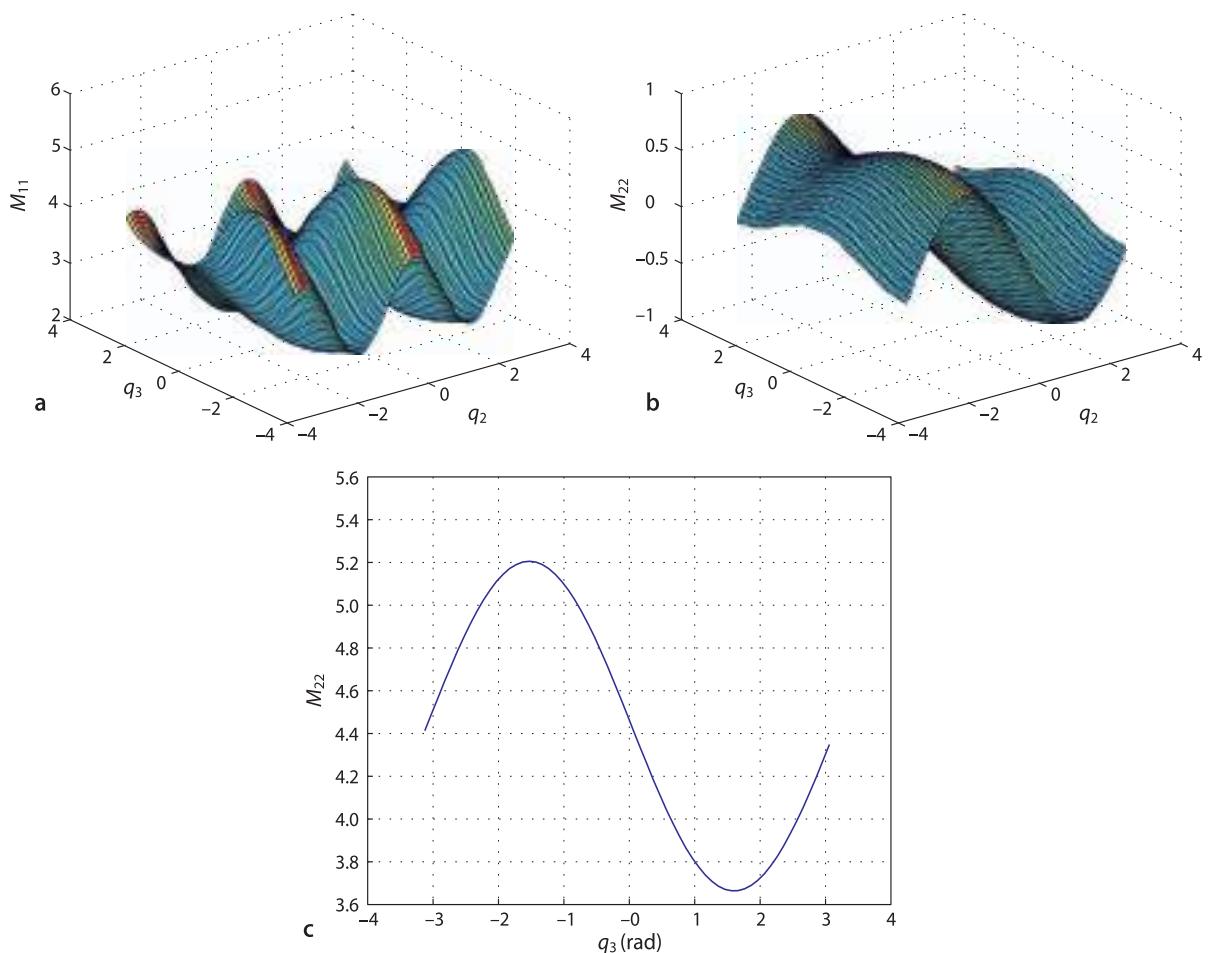
which we observe is symmetric. The diagonal elements  $M_{jj}$  describe the inertia seen by joint  $j$ , that is,  $Q_j = M_{jj}\ddot{q}_j$ . Note that the first two diagonal elements, corresponding to the robot's waist and shoulder joints, are large since motion of these joints involves rotation of the heavy upper- and lower-arm links. The off-diagonal terms  $M_{ij} = M_{ji}$ ,  $i \neq j$  represent coupling of acceleration from joint  $j$  to the generalized force on joint  $i$ .

We can investigate some of the elements of the inertia matrix and how they vary with robot configuration using the simple (but slow) commands

```
1 [Q2,Q3] = meshgrid(-pi:0.1:pi, -pi:0.1:pi);
2 for i=1:numcols(Q2),
3     for j=1:numcols(Q3),
4         M = p560.inertia([0 Q2(i,j) Q3(i,j) 0 0 0]);
5         M11(i,j) = M(1,1);
6         M12(i,j) = M(1,2);
7     end
8 end
9 surf(Q2, Q3, M11); surf(Q2, Q3, M12);
```



**Joseph-Louis Lagrange** (1736–1813) was an Italian-born French mathematician and astronomer. He made significant contributions to the fields of analysis, number theory, classical and celestial mechanics. In 1766 he succeeded Euler as the director of mathematics at the Prussian Academy of Sciences in Berlin, where he stayed for over twenty years, producing a large body of work and winning several prizes of the French Academy of Sciences. His treatise on analytical mechanics “*Mécanique Analytique*” first published in 1788, offered the most comprehensive treatment of classical mechanics since Newton and formed a basis for the development of mathematical physics in the nineteenth century. In 1787 he became a member of the French Academy, became the first professor of analysis at the École Polytechnique, a member of the Legion of Honour and a Count of the Empire in 1808. He is buried in the Panthéon in Paris.



The results are shown in Fig. 9.2 and we see significant variation in the value of  $M_{11}$  which changes by a factor of

```
>> max(M11(:)) / min(M11(:))
ans =
2.1558
```

This is important for robot design since, for a fixed maximum motor torque, inertia sets the upper bound on acceleration which in turn effects path following accuracy.

The off-diagonal term  $M_{12}$  represents coupling between the angular acceleration of joint 2 and the torque on joint 1. That is, if joint 2 accelerates then a torque will be exerted on joint 1 and vice versa.

**Fig. 9.2.** Variation of inertia matrix elements as a function of manipulator pose. **a** Joint 1 inertia as a function of joint 2 and 3 angles  $M_{11}(q_2, q_3)$ ; **b** product of inertia  $M_{12}(q_2, q_3)$ ; **c** joint 2 inertia as a function of joint 3 angle  $M_{22}(q_3)$ . Inertia has the units of  $\text{kg m}^2$

### 9.1.3 Coriolis Matrix

$$Q = M(\mathbf{q})\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + F(\dot{\mathbf{q}}) + G(\mathbf{q}) + J(\mathbf{q})^T \mathbf{f}$$

The Coriolis matrix  $C$  is a function of joint coordinates and joint velocity. The centripetal torques are proportional to  $\dot{q}_i^2$ , while the Coriolis torques are proportional to  $\dot{q}_i \dot{q}_j$ . For example, at the nominal pose with all joints moving at  $0.5 \text{ rad s}^{-1}$

```
>> qd = 0.5*[1 1 1 1 1 1];
```

the Coriolis matrix is



**Gaspard-Gustave de Coriolis** (1792–1843) was a French mathematician, mechanical engineer and scientist. Born in Paris, in 1816 he became a tutor at the École Polytechnique where he carried out experiments on friction and hydraulics and later became a professor at the École des Ponts and Chaussées (School of Bridges and Roads). He extended ideas about kinetic energy and work to rotating systems and in 1835 wrote the famous paper *Sur les équations du mouvement relatif des systèmes de corps* (On the equations of relative motion of a system of bodies) which dealt with the transfer of energy in rotating systems such as waterwheels. In the late 19<sup>th</sup> century his ideas were picked up by the meteorological community to incorporate effects due to the Earth's rotation. He is buried in Paris's Montparnasse Cemetery.

```
>> C = p560.coriolis(qn, qd)
C =
 0.0000   -0.9115    0.2173    0.0013   -0.0026    0.0001
 0.3140   -0.0000    0.5786   -0.0011   -0.0001   -0.0000
 -0.1804   -0.1929   -0.0000   -0.0005   -0.0023   -0.0000
 -0.0002    0.0006   -0.0000   -0.0000    0.0003   -0.0000
 -0.0000    0.0000    0.0014   -0.0002   -0.0000   -0.0000
 0     0.0000    0.0000    0.0000    0.0000      0
```

The off-diagonal terms  $C_{ij}$  represent coupling of joint  $j$  velocity to the generalized force acting on joint  $i$ .  $C_{1,2} = -0.9115$  is very significant and represents coupling from joint 2 velocity to torque on joint 1 – rotation of the shoulder exerts a torque on the waist. Since the elements of this matrix represents a coupling from velocity to joint force they have the same dimensions as viscous friction or damping, however the sign can be positive or negative. The joint torques in this example are

```
>> C*qd'
ans =
-1.3422
0.5101
-0.5583
0.0009
-0.0001
0.0001
```

#### 9.1.4 Effect of Payload

Any real robot has a specified maximum payload which is dictated by two dynamic effects. The first is that a mass at the end of the robot will increase the inertia *seen* by the joints which reduces acceleration and dynamic performance. The second is that mass generates a weight force which the joints needs to support. In the worst case the increased gravity torque component might exceed the rating of one or more motors. However even if the rating is not exceeded there is less torque available for acceleration which again reduces dynamic performance.

As an example we will add a 2.5 kg point mass to the Puma 560 which is its rated maximum payload. The centre of mass of the payload cannot be at the centre of the wrist coordinate frame, that is inside the wrist, so we will offset it 100 mm in the  $z$ -direction of the wrist frame. We achieve this by modifying the inertial parameters of the robot's last link<sup>◀</sup>

```
>> p560.payload(2.5, [0, 0, 0.1]);
```

The inertia at the nominal pose is now

```
>> M_loaded = p560.inertia(qn);
```

and the *ratio* with respect to the unloaded case, computed earlier, is

This assumes that the last link itself has no mass which is a reasonable approximation.

```
>> M_loaded ./ M;
ans =
 1.3363    0.9872    2.1490   49.3960   80.1821    1.0000
 0.9872    1.2667    2.9191   5.9299   74.0092    1.0000
 2.1490    2.9191    1.6601  -2.1092   66.4071    1.0000
 49.3960    5.9299   -2.1092   1.0647   18.0253    1.0000
 83.4369   74.0092   66.4071   18.0253   1.1454    1.0000
 1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
```

We see that the diagonal elements have increased significantly, for instance the elbow joint inertia has increased by 66% which reduces the maximum acceleration by nearly two thirds. Reduced acceleration impairs the robot's ability to accurately follow a high speed path. The inertia of joint 6 is unaffected since this added mass lies on the axis of this joint's rotation. The off-diagonal terms have increased significantly, particularly in rows and columns four and five. This indicates that motion of joints 4 and 5, the wrist joints, which are swinging the offset mass give rise to large reaction forces that are *felt* by all the other robot joints.

The gravity load has also increased by some significant factors

```
>> p560.gravload(qn) ./ gravload
ans =
 2.6282    1.4763    2.5489   29.5765  138.8889      NaN
```

particularly at the elbow and wrist.►

## 9.1.5 Base Force

The first and last elements should be ignored and are the results of numerical error in what should be 0 / 0, for the case of gravity load with and without payload  $g_1 = g_6 = 0$ .

A moving robot exerts a wrench on its base, a vertical force to hold it up and other forces and torques as the arm moves around. This wrench is returned as an optional output argument of the `rne` method

```
>> [Q,g] = p560.rne(qn, qz, qz);
```

In this case `g` is the wrench

```
>> g'
ans =
 0    -0.0000   230.0445  -48.4024  -31.6399   -0.0000
```

that needs to be applied to the base to keep it in equilibrium. The vertical force of 230 N is the total weight of the robot which has a mass of

```
>> sum([p560.links.m])
ans =
 23.4500
```

There is also a moment about the  $x$ - and  $y$ -axes since the centre of mass of the robot is not over origin of the base coordinate frame.

The base forces are important in situations where the robot does not have a rigid base such as on a satellite in space, on a boat, an underwater vehicle or even on a vehicle with soft suspension.

## 9.1.6 Dynamic Manipulability

In Sect. 8.1.4 we discussed a kinematic measure of manipulability, that is, how well configured the robot is to achieve velocity in particular directions. An extension of that measure is to consider how well the manipulator is able to accelerate in different Cartesian directions. Following a similar approach, we consider the set of generalized joint forces with unit norm

$$Q^T Q = 1$$

From Eq. 9.1 and ignoring gravity and assuming  $\ddot{q} = 0$  we write

$$Q = M\ddot{q}$$

Differentiating Eq. 8.2 and still assuming  $\dot{q} = 0$  we write

$$\dot{\nu} = J(q)\ddot{q}$$

Combining these we write

$$\dot{\nu}^T (JM^{-1}M^{-T}J^T)^{-1}\dot{\nu} = 1$$

or more compactly

$$\dot{\nu}^T M_x^{-1} \dot{\nu} = 1$$

which is the equation of a hyper-ellipsoid in Cartesian acceleration space. For example, at the nominal pose

```
>> J = p560.jacob0(qn);
>> M = p560.inertia(qn);
>> Mx = (J * inv(M) * inv(M)' * J');
```

If we consider just the translational acceleration, that is the top left  $3 \times 3$  submatrix of  $M_x$

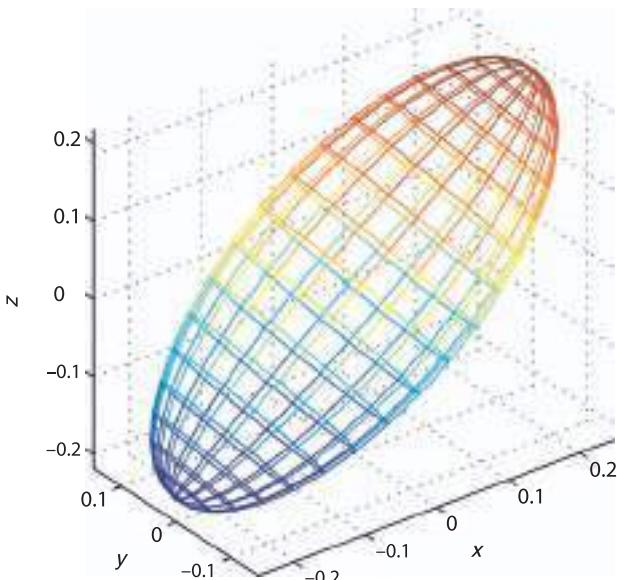
```
>> Mx = Mx(1:3, 1:3);
```

this is a 3-dimensional ellipsoid

```
>> plot_ellipse( Mx )
```

which is plotted in Fig. 9.3. The major axis of this ellipsoid is the direction in which the manipulator has maximum acceleration at this configuration. The radii of the ellipse are the square root of the eigenvalues

```
>> sqrt(eig(Mx))
ans =
0.4412
0.1039
0.1677
```



**Fig. 9.3.**

Spatial acceleration ellipsoid for Puma 560 robot in nominal pose

and the direction of maximum acceleration is given by the first eigenvector. The ratio of the minimum to maximum radius

```
>> min(ans)/max(ans)
ans =
0.2355
```

is a measure of the non-uniformity of end-effector acceleration.► It would be unity for isotropic acceleration capability. In this case acceleration capability is good in the  $x$ - and  $z$ -directions, but poor in the  $y$ -direction.

The manipulability measure proposed by Asada is similar but considers the ratios of the eigenvalues of

$$\ddot{x}^T J^{-T} M J^{-1} \ddot{x} = 1$$

and returns a uniformity measure  $m \in [0, 1]$  where 1 indicates uniformity of acceleration in all directions. For this example

```
>> p560.maniplty(qn, 'asada')
ans =
0.2094
```

The 6-dimensional ellipsoid has dimensions with different units:  $\text{m s}^{-2}$  and  $\text{rad s}^{-2}$ . This makes comparison of all 6 radii problematic.

## 9.2 Drive Train

The vast majority of robots today are driven by electric motors. Typically brushless servo motors are used for large industrial robots while small laboratory or hobby robots use brushed DC motors or stepper motors. Robots with very large payloads, hundreds of kilograms, would generally use electro-hydraulic actuators.

Electric motors are compact and efficient but do not produce very high torque. However they can rotate at very high speed so it is common to use a reduction gearbox to trade off speed for increased torque. The disadvantage of a gearbox is increased cost, weight, friction and mechanical noise. Many very high-performance robots, such as used in high-speed electronic assembly, use expensive high-torque motors with a direct drive or a very low gear ratio achieved using cables or thin metal bands rather than gears.

Figure 9.4 shows in schematic form the drive train of a typical robot joint. For a  $G:1$  reduction drive the torque at the link is  $G$  times the torque at the motor. For rotary joints the quantities measured at the link, reference frame  $l$ , are related to the motor referenced quantities, reference frame  $m$ , as shown in the table in Fig. 9.4. For example if you turned the motor shaft by hand you would *feel* the inertia of the load through the gearbox but it would be reduced by  $G^2$  as would the frictional force. However if you turned the load side shaft by hand you would *feel* the inertia of the motor through the gearbox but it would be increased by  $G^2$ , as would the frictional force.

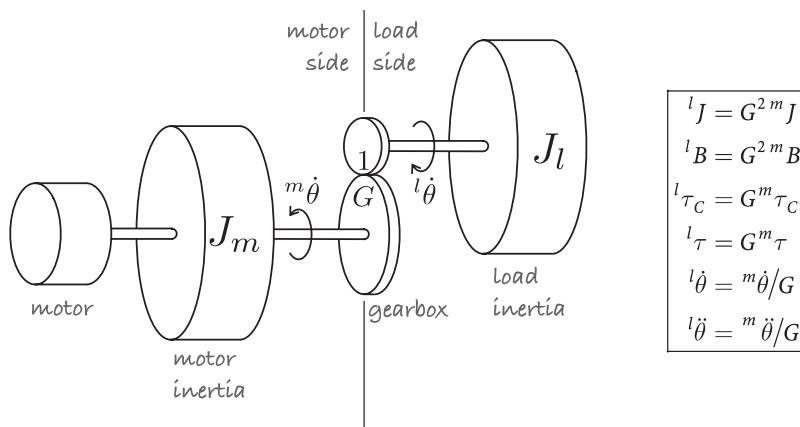


Fig. 9.4.

Schematic of a typical robot drivetrain showing motor and load inertia, gearbox and disturbance torque. The frame of reference, motor or load, is indicated by the leading superscript. The table at the right shows the relationship between load and motor referenced quantities for gear ratio  $G$

There are two components of inertia *seen* by the motor. The first is due to the rotating part of the motor itself, its rotor. It is denoted  $J_m$  and is an intrinsic characteristic of the motor and the value is provided in the motor manufacturer's data sheet. The second component is the load inertia  $J_l$  which is the inertia of the driven link and all the other links that are attached to it. For joint  $j$  this is element  $M_{jj}$  of the manipulator inertia matrix discussed previously and is a function of the robot's configuration.

The total inertia *seen* by the motor for joint  $j$  is therefore

$${}^m J_j = J_{m_j} + \frac{1}{G_j^2} M_{jj} \quad (9.3)$$

The gearing reduces the significance of the second term which is configuration dependent. Reducing the variability in total inertia is beneficial when it comes to designing a control system for the joint.

A negative aspect of gearing is an increase in viscous friction and non-linear effects such as backlash and Coulomb friction which will be discussed in the next section. Flexible couplings and long drive shafts between the motor and the joint can act as torsional springs and introduce complex dynamics into the system and this is discussed in Sect. 9.4.4.

## 9.2.1 Friction

$$\dot{Q} = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + F(\dot{q}) + G(q) + J(q)^T f$$

For the Puma robot joint friction varied from 10 to 47% of the maximum motor torque for the first three joints (Corke 1996b).

For most electric drive robots friction is the next most dominant joint force after gravity. For any rotating machinery, motor or gearbox, the friction torque versus speed characteristic has a form similar to that shown in Fig. 9.5. At zero speed we observe an effect known as stiction which is the applied torque must exceed before rotation can occur – a process known as *breaking stiction*.

Once the machine is moving the stiction force rapidly decreases and viscous friction dominates. Viscous friction, shown by the dashed line in Fig. 9.5, is commonly modeled by

$$Q_f = B\dot{q} + Q_C \quad (9.4)$$

where the slope  $B$  is the viscous friction coefficient and the offset is Coulomb friction. The latter is frequently modeled by the non-linear function

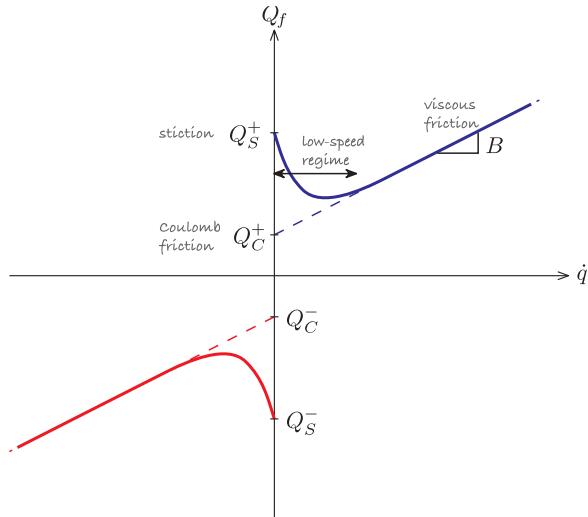


Fig. 9.5.

Typical friction versus speed characteristic. The dashed lines depict a simple piecewise-linear friction model characterized by slope (viscous friction) and intercept (Coulomb friction)

Charles-Augustin de Coulomb (1736–1806) was a French physicist. He was born in Angoulême to a wealthy family and studied mathematics at the Collège des Quatre-Nations under Pierre Charles Monnier, and later at the military school in Mézières. He spent eight years in Martinique involved in the construction of Fort Bourbon and there he contracted tropical fever.

Later he worked at the shipyards in Rochefort which he used as laboratories for his experiments in static and dynamic friction of sliding surfaces. His paper *Théorie des machines simples* won the Grand Prix from the Académie des Sciences in 1781. His later research was on electromagnetism and electrostatics and he is best known for the formula on electrostatic forces, named in his honor, as is the SI unit of charge. After the revolution he was involved in determining the new system of weights and measures.



$$Q_C = \begin{cases} 0 & \dot{q} = 0 \\ Q_C^+ & \dot{q} > 0 \\ Q_C^- & \dot{q} < 0 \end{cases} \quad (9.5)$$

In general the friction value depends on the direction of rotation but this asymmetry is more pronounced for Coulomb than for viscous friction.

There are several components of the friction *seen* by the motor. The first component is due to the motor itself: its bearings and, for a brushed motor, the brushes rubbing on the commutator. The viscous friction coefficient for a motor is constant and often provided in the manufacturer's data sheet. Information about Coulomb friction is not generally provided. Other components of friction are due to the gearbox and the bearings that support the link.

The Toolbox models friction within the [Link](#) object. The friction values are lumped and motor referenced, that is, they apply to the motor side of the gearbox. Viscous friction is a scalar that applies for positive and negative velocity. ▶ Coulomb friction is a 2-vector comprising  $(Q_C^+, Q_C^-)$ . For example, the dynamic parameters of the Puma robot's second link are

```
>> p560.links(2).dyn
l =
theta=q, d=0, a=0.4318, alpha=0 (R,stdDH)
m    = 17.400000
r    = -0.363800 0.006000 0.227500
J    = | 0.130000 0.000000 0.000000 |
      | 0.000000 0.524000 0.000000 |
      | 0.000000 0.000000 0.539000 |
Jm   = 0.000200
Bm   = 0.000817
Tc   = 0.126000(+) -0.071000(-)
G    = 107.815000
```

In practice some mechanisms have a velocity dependent friction characteristic.

The last three lines show the viscous friction coefficient, Coulomb friction coefficients and the gear ratio. The online documentation for the [Link](#) class describes how to set these parameters.

### 9.3 Forward Dynamics

To determine the motion of the manipulator in response to the forces and torques applied to its joints we require the forward dynamics or integral dynamics. Rearranging the equations of motion Eq. 9.1 we obtain the joint acceleration

$$\ddot{q} = M^{-1}(q)(Q - C(q, \dot{q})\dot{q} - F(\dot{q}) - G(q)) \quad (9.6)$$

This is computed by the [accel](#) method of the [SerialLink](#) class

```
>> qdd = p560.accel(q, qd, Q)
```

Puma560 collapsing under gravity

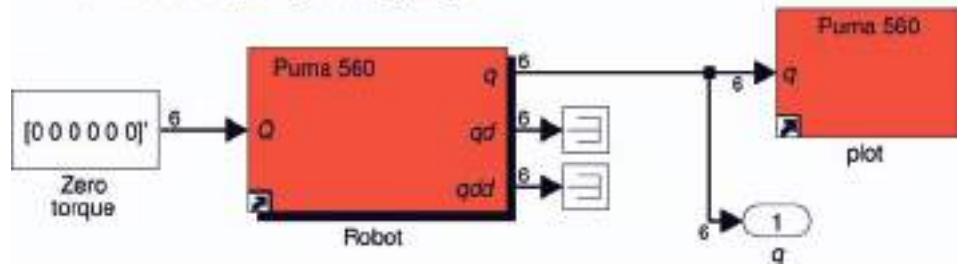


Fig. 9.6.  
Simulink® model `sl_ztorque`  
for the Puma 560 manipulator  
with zero joint torques

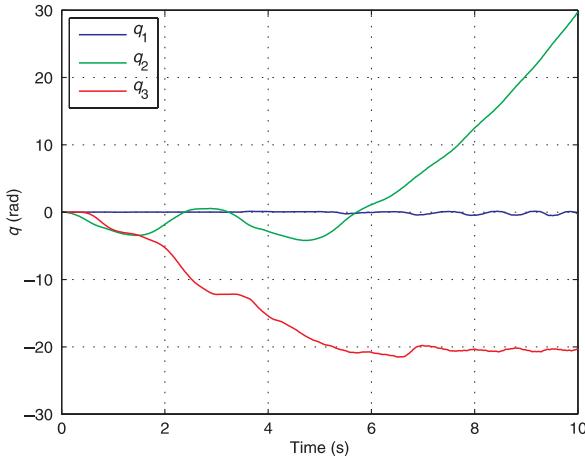


Fig. 9.7.  
Joint angle trajectory for  
Puma 560 robot collapsing  
under gravity and starting at `qz`

given the joint coordinates, joint velocity and applied joint torques. This functionality is also encapsulated in the Simulink® block `Robot` and an example of its use is

```
>> sl_ztorque
```

which is shown in Fig. 9.6. The torque applied to the robot is zero and the initial joint angles is set as a parameter of the `Robot` block, in this case to the *zero-angle pose*. The simulation is run

```
>> r = sim('sl_ztorque');
```

and the joint angles as a function of time are returned in the object `r`

```
>> t = r.find('tout');
>> q = r.find('yout');
```

We can show the robot's motion in animation

```
>> p560.plot(q)
```

These motions are not mechanically possible on the real robot.

and see it collapsing under gravity since there are no torques to counter gravity and hold in upright. The shoulder falls and swings back and forth as does the elbow, while the waist joint rotates because of Coriolis coupling. The motion will slowly decay as the energy is dissipated by viscous friction.

Alternatively we can plot the joint angles as a function of time

```
>> plot(t, q(:,1:3))
```

and this is shown in Fig. 9.7. The method `fdyn` can be used as a non-graphical alternative to Simulink® and is described in the online documentation.

This example is rather unrealistic and in reality the joint torques would be computed as some function of the actual and desired robot joint angles. This is the topic of the next section.

Coulomb friction is a strong non-linearity and can cause difficulty when using numerical integration routines to solve the forward dynamics. This is usually manifested by very long integration times. Fixed-step solvers tend to be more tolerant, and these can be selected through the Simulink® Simulation+Configuration Parameters menu item.

The default Puma 560 model, defined using `mdl_puma560`, has non-zero viscous and Coulomb friction parameters for each joint. Sometimes it is useful to zero all the friction parameters for a robot and this can be achieved by

```
>> p560_nf = p560.nofriction();
```

which returns a copy of the robot object that is similar in all respects except that the Coulomb friction is zero. Alternatively we can set Coulomb and viscous friction coefficients to zero

```
>> p560_nf = p560.nofriction('all');
```

## 9.4 Manipulator Joint Control

In order for the robot end-effector to follow a desired Cartesian trajectory each of its joints must follow a specific joint-space trajectory. In this section we discuss the two main approaches to robot joint control: independent control and model-based control.

### 9.4.1 Actuators

Most laboratory and industrial robots are electrically actuated. Electrical motors can be either current or voltage controlled and we consider here the current control case.► We assume a motor driver or amplifier which provides motor current

$$i = K_a u$$

that is linearly related to the applied control voltage  $u$  and where  $K_a$  is the transconductance of the amplifier with units of  $\text{A V}^{-1}$ . The torque generated by the motor is proportional to current

$$\tau = K_m i$$

where  $K_m$  is the motor torque constant with units of  $\text{N m A}^{-1}$ . The dynamics of the motor are described by

$$J_m \dot{\omega} + B\omega + \tau_c(\omega) = K_m K_a u \quad (9.7)$$

where  $J_m$  is the total inertia seen by the motor from Eq. 3.9,  $B$  is the viscous friction coefficient and  $\tau_c$  is the Coulomb friction torque.

Current control is implemented by an electronic constant current source, or a variable voltage source with feedback of actual motor current. In the latter case the electrical dynamics of the motor due to its resistance and inductance must be taken into account. A variable voltage source is most commonly implemented by a pulse-width modulated (PWM) switching circuit.

### 9.4.2 Independent Joint Control

A common approach to robot joint control is to consider each joint as an independent control system that attempts to accurately follow the joint angle trajectory. However as we shall see, this is complicated by various *disturbance* torques such as gravity, velocity and acceleration coupling and friction that act on the joint.

**Table 9.1.**  
Motor and drive parameters for Puma 560 shoulder joint (Corke 1996b)

Parameter	Symbol	Value	Unit
Motor torque constant	$K_m$	0.228	N m A <sup>-1</sup>
Motor inertia	$J_m$	$200 \times 10^{-6}$	kg m <sup>2</sup>
Drive viscous friction	$B_m$	$817 \times 10^{-6}$	N m s rad <sup>-1</sup>
Drive Coulomb friction	$\tau_C^+$ $\tau_C^-$	0.126 -0.709	N m N m
Gear ratio	$G$	107.815	
Maximum torque	$\tau_{\max}$	0.900	N m
Maximum speed	$\dot{q}_{\max}$	165	rad s <sup>-1</sup>

A very common control structure is the nested control loop. The outer loop is responsible for maintaining position and determines the velocity of the joint that will minimize position error. The inner loop is responsible for maintaining the velocity of the joint as demanded by the outer loop.

**Velocity loop.** We will study the inner velocity loop first and we will use as an example the shoulder joint of the Puma 560 robot since its parameters are well known, see Table 9.1.

Ignoring Coulomb friction we write the Laplace transform of Eq. 9.7 as

$$sJ\Omega(s) + B\Omega(s) = K_m K_a U(s)$$

where  $\Omega(s)$  and  $U(s)$  are the Laplace transform of the time domain signals  $\omega(t)$  and  $u(t)$  respectively. Rearranging as a linear transfer function we write

$$\frac{\Omega(s)}{U(s)} = \frac{K_m K_a}{Js + B}$$

The effective inertia from Eq. 9.3 is

$$J = J_m + \frac{1}{G^2} M_{22} \quad (9.8)$$

Figure 9.2 shows that  $M_{22}$  varies significantly with manipulator pose so for now we will take the mean value which is  $2 \text{ kg m}^2$  which yields a total inertia of

$$J = 200 \times 10^{-6} + \frac{2}{(107.815)^2} = 200 \times 10^{-6} + 172 \times 10^{-6} = 372 \times 10^{-6} \text{ kg m}^2$$

Note that the referred link inertia is comparable to the inertia of the motor itself.

The Simulink® model is shown in Fig. 9.8. A delay of 1 ms is included to model the computational time of the velocity loop control algorithm and a saturator models the finite maximum torque that the motor can deliver. We use a proportional controller based on the error between demanded and actual velocity to compute the demand to the motor driver

$$u^* = K_v (\dot{q}^* + \dot{q}) \quad (9.9)$$

The motor velocity is typically computed by taking the difference in motor position at each sample time, and the position is measured by a shaft encoder.

To test this velocity controller we create a test harness

`>> vloop_test`

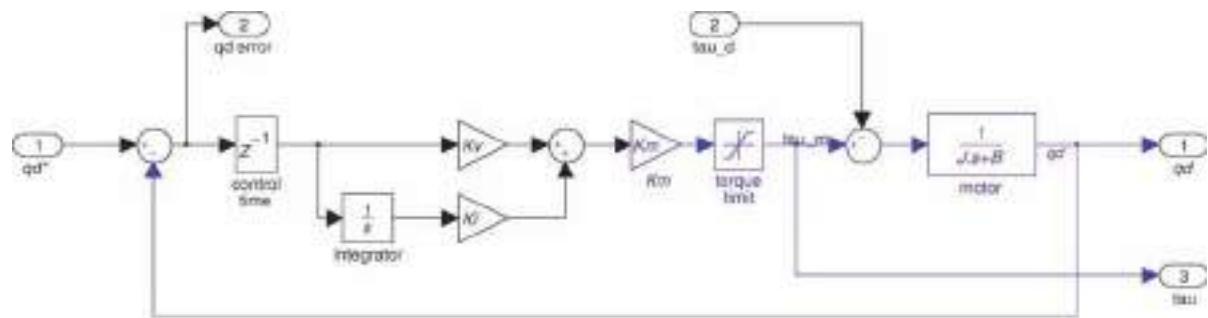


Fig. 9.8. Velocity control loop, Simulink® model [vloop](#)

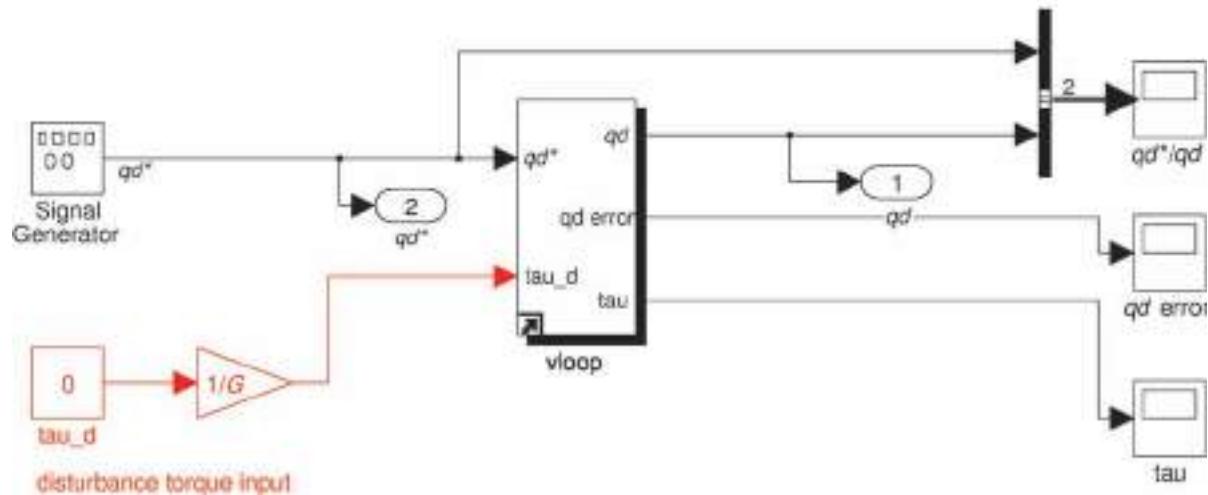


Fig. 9.9. Test harness for the velocity control loop, Simulink® model [vloop\\_test](#). The input  $\tau_u_d$  is used to simulate a disturbance torque acting on the joint

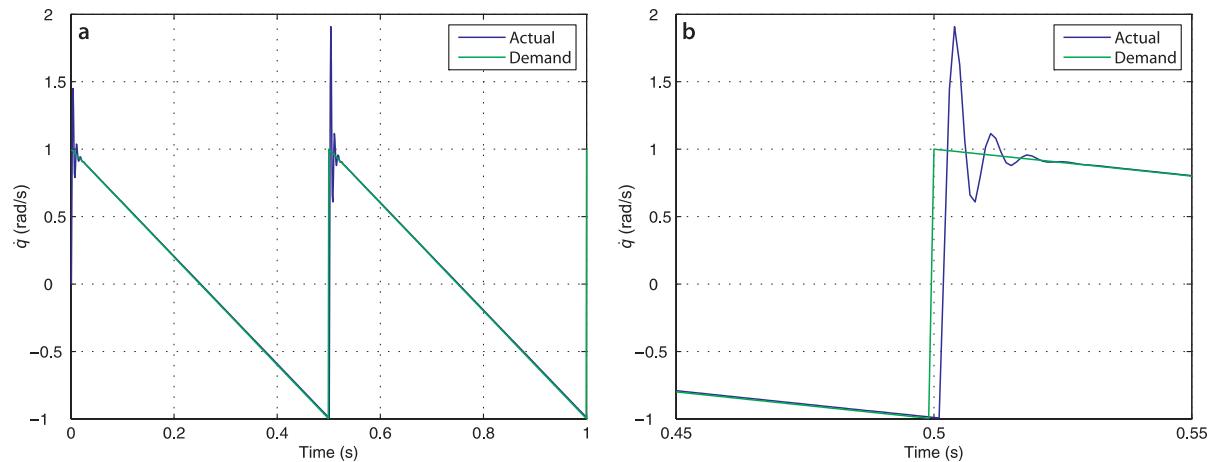


Fig. 9.10. Velocity loop with a sawtooth demand. a Response; b closeup of response

with a sawtooth-shaped velocity demand which is shown in Fig. 9.9. Running the simulator

```
>> sim('vloop_test');
```

**While a step response is a common and useful measure of control performance, in practice a velocity loop would never receive a step demand.**

and with a little experimentation we find that a gain of  $K_v = 1$  gives satisfactory performance as shown in Fig. 9.10. There is some overshoot at the step<sup>4</sup> but less gain leads to increased velocity error and more gain leads to oscillation – as always in control engineering it is a matter of tradeoffs.

So far we have ignored one very important dynamic effect on robot arms – gravity. Figure 9.1b shows that the gravity torque on this joint varies from approximately –40 to 40 N m. We now add a disturbance torque equal to just half that amount, 20 N m at the load. We edit the test harness and rerun the simulation. The results shown in Fig. 9.11 indicate that the control performance has been badly degraded – the tracking error has increased to more than 1 rad s<sup>–1</sup>.

There are three common strategies to counter this error. The simplest is to increase the gain. This will reduce the tracking error but push the system into instability.

The second strategy, commonly used in industrial motor drives, is to add integral action. We change Eq. 9.9 to a proportional-integral controller

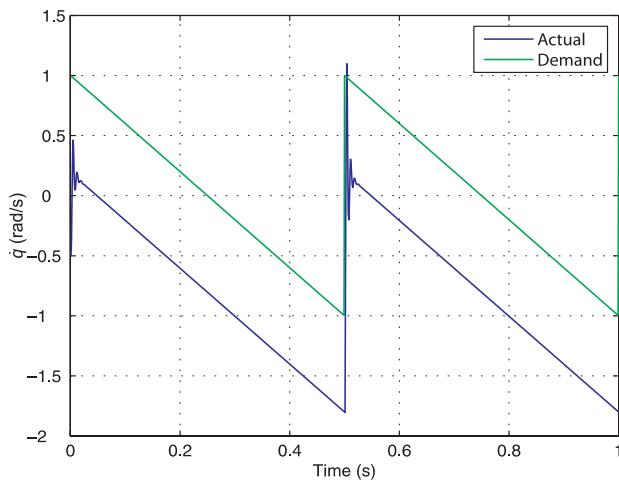
$$u^* = \left( K_v + \frac{K_i}{s} \right) (\dot{q}^* - \dot{q}), \quad K_i > 0$$

In the Simulink® model of Fig. 9.8 this is achieved by setting `Ki` to a non-zero value. With some experimentation we find the gains  $K_v = 1$  and  $K_i = 10$  work well and the performance is shown in Fig. 9.12. The integrator state evolves over time to cancel out the disturbance term and we can see the error decaying to zero. In practice the disturbance varies over time and the integrator's ability to track it depends on the value of the integral gain  $K_i$ . In reality other disturbances affect the joint, for instance Coulomb friction and torques due to velocity and acceleration coupling. The controller needs to be well tuned so that these have minimal effect on the tracking performance.

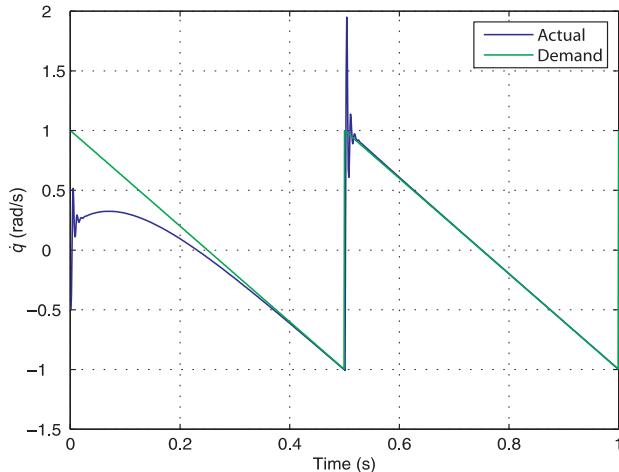
From a classical control system perspective the original velocity loop contains no integrator block which means it is classified as a Type 0 system. A characteristic of such systems is they exhibit a finite error for a constant input or constant disturbance input, just as we observed in Fig. 9.11. Adding the integral controller changed the system to Type 1 which has zero error for a constant input or constant disturbance. As always in engineering there are some tradeoffs. The integral term can lead to increased overshoot so increasing  $K_i$  usually requires some compensating reduction of  $K_v$ . If the joint actuator is pushed to its performance limit, for instance the torque limit is reached, then the tracking error will grow with time since the motor acceleration will be lower than required. The integral of this increasing error will grow leading to a condition known as integral windup. When the joint finally reaches its destination the large accumulated integral keeps driving the motor forward until the integral decays – leading to large overshoot. Various strategies are employed to combat this, such as limiting the maximum value of the integrator, or only allowing integral action when the motor is close to its setpoint.

**Motor limits.** Electric motors are limited in both torque and speed. The maximum torque is defined by the maximum current the drive electronics can provide. A motor also has a maximum rated current beyond which the motor can be damaged by overheating or demagnetization of its permanent magnets which irreversibly reduces its torque constant. As speed increases so does friction and the maximum speed is  $\omega_{\max} = \tau_{\max}/B$ .

The product of motor torque and speed is the mechanical output power and also has an upper bound. Motors can tolerate some overloading, peak power and peak torque, for short periods of time but the sustained rating is significantly lower than the peak.



**Fig. 9.11.**  
Velocity loop response to a saw-tooth demand with a gravity disturbance of 20 N m



**Fig. 9.12.**  
Velocity loop response to a saw-tooth demand with a gravity disturbance of 20 N m and proportional-integral control

Strategies one and two are collectively referred to as disturbance rejection and are concerned with reducing the effect of an unknown disturbance. However if we think about the problem in its robotics context the gravity disturbance is not unknown. In Sect. 9.1.1 we showed how to compute the torque due to gravity that acts on each joint. If we know this torque, and the motor torque constant, we can *add* it to the output of the PI controller. ▶ The third strategy to reduce the effect of disturbance is therefore to predict it and cancel it out – a strategy known as torque feedforward control. The block diagram of Fig. 9.8 is augmented with a feedforward term which is shown by the red wiring in Fig. 9.13.

The final consideration in control design is how inertia variation affects the closed-loop response. Using Eq. 9.8 and the data from Fig. 9.2c we find that the minimum and maximum joint inertia are  $320 \times 10^{-6}$  and  $450 \times 10^{-6} \text{ kg m}^2$  respectively. Figure 9.14 shows the velocity tracking error using the control gains chosen above for the case of minimum and maximum inertia. We can see that the tracking error decays more slowly for larger inertia, and is showing signs of instability for the minimum inertia case. In practice the gain would be chosen to optimize the closed-loop performance at both extremes.

Motor speed control is important for all types of robots, not just arms. For example it is used to control the speed of the wheels for car-like vehicles and the rotors of a quad-rotor as discussed in Chap. 4.

Even if the gravity load is known imprecisely feedforward can reduce the magnitude of the disturbance.

**Back EMF.** A spinning motor acts like a generator and produces a voltage  $V_b$  called the back EMF which opposes the current flowing into the motor. Back EMF is proportional to motor speed  $V_b = K_m \omega$  where  $K_m$  is the motor torque constant again whose units can also be interpreted as V s rad<sup>-1</sup>. When this voltage equals the maximum possible voltage the drive electronics can provide then no more current can flow into the motor and torque falls to zero. This provides a practical upper bound on motor speed.

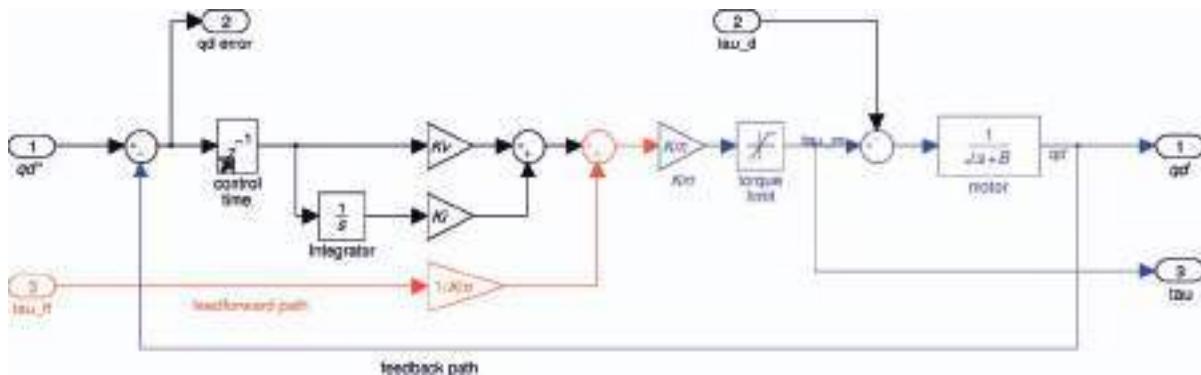


Fig. 9.13. Velocity control loop with feedforward (shown in red), from Simulink® model vloop\_test2

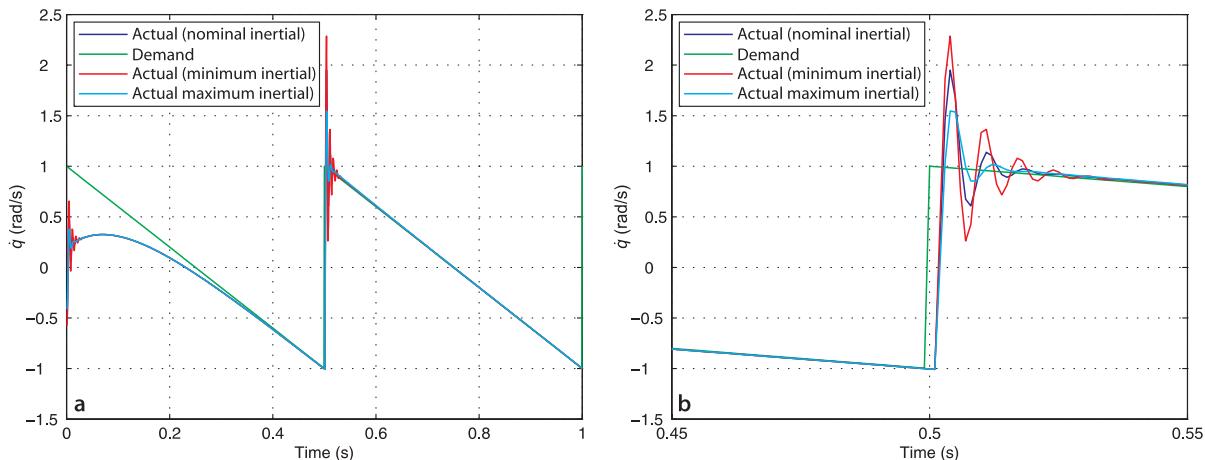


Fig. 9.14. Velocity loop with a saw-tooth demand with integral action and 20 N m torque disturbance, but varying inertia  $M_{22}$ . **a** Response; **b** closeup of response

**Position loop.** The outer loop is responsible for maintaining position and its Simulink® model is shown in Fig. 9.15. The error in position provides the velocity demand for the inner loop.

To test the position control loop we create another test harness

```
>> ploop_test
```

The position demand comes from an LSPB trajectory generator that moves from 0 to 0.5 rad in 1 s with a sample rate of 1 000 Hz. The test harness shown in Fig. 9.16 can also inject a disturbance torque into the velocity loop.

We use a proportional controller based on the error between actual and demanded position to compute the desired speed of the motor

$$\dot{q}^* = K_p(q^* - q) \quad (9.10)$$

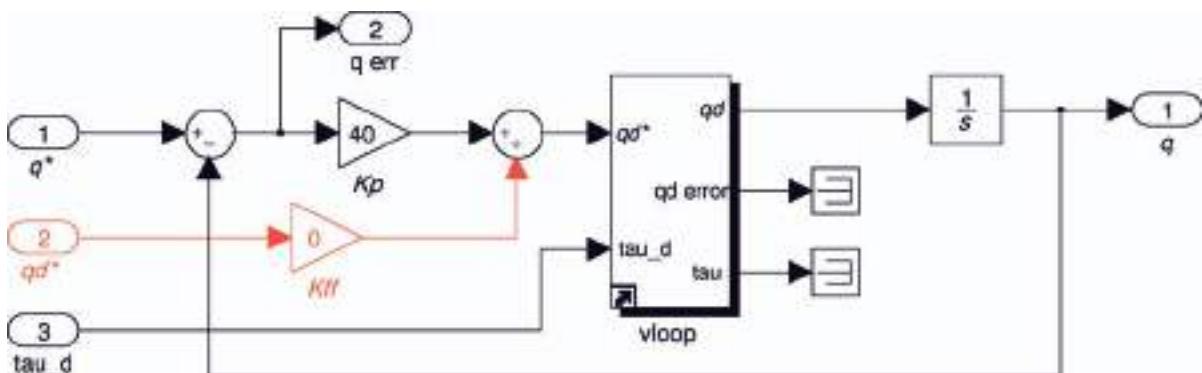


Fig. 9.15. Position control loop, Simulink® model [ploop](#)

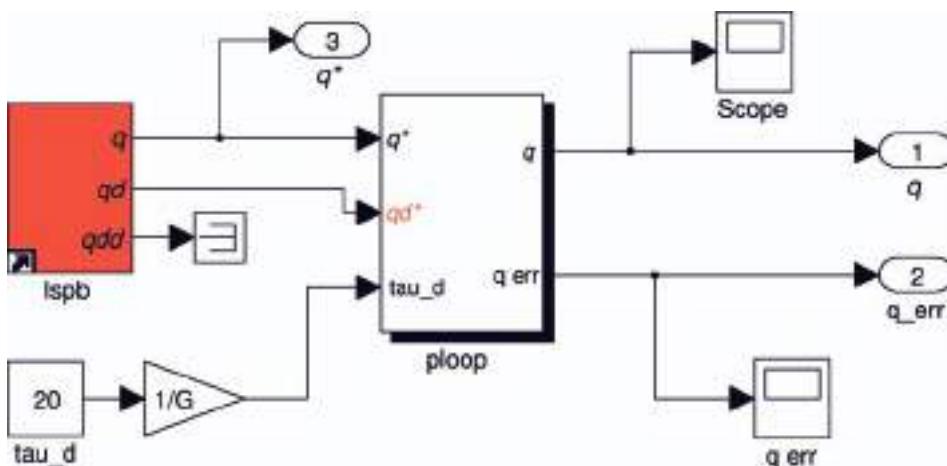


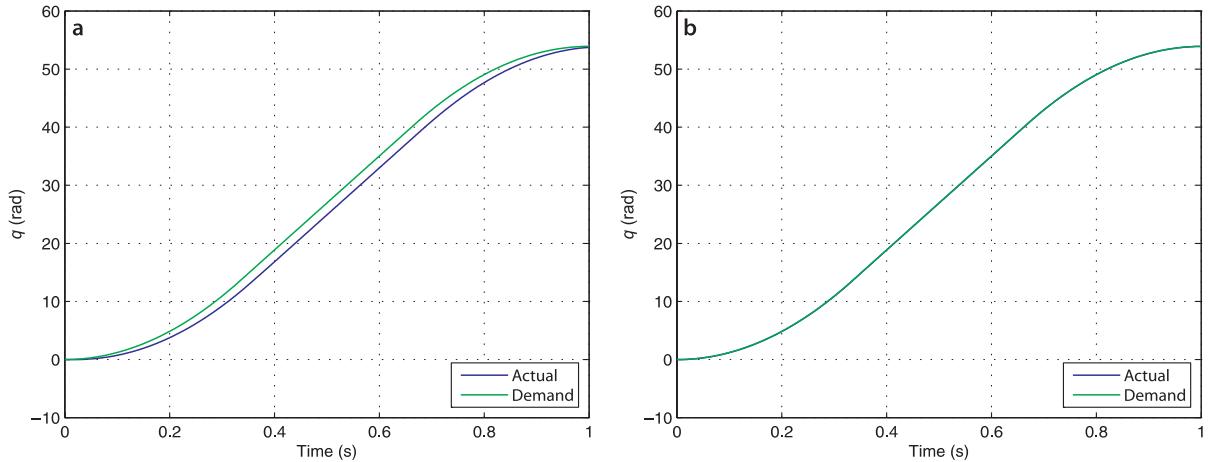
Fig. 9.16.  
Test harness for position control loop, Simulink® model [ploop\\_test](#)

The joint controller is tuned by adjusting the three gains:  $K_p$ ,  $K_v$ ,  $K_i$  in order to achieve good tracking performance at both low and high speed. For  $K_p = 40$  the tracking and error responses are shown in Fig. 9.17a. The error between the demand and actual curves is due to the integral of the error in the velocity loop which has units of angle.

The position loop is based on feedback of error which is of course a classical approach. An often overlooked characteristic of proportional control is that zero error means zero demand to the controlled system. In this case zero position error means zero demanded velocity to the inner loop – to achieve non-zero joint velocity demand from Eq. 9.10 requires non-zero error which is of course not desirable for accurate tracking. Usefully the LSPB trajectory function computes velocity as a function of time as well as position. If we know the velocity we can add it to the output of the proportional control loop, the input to the velocity loop – a strategy known as velocity feedforward control. The time response with feedforward is shown in Fig. 9.17b and we see that tracking error is greatly reduced.

Let us recap what we have learnt about independent joint control. A common structure is the nested control loop. The inner loop uses a proportional or proportional-integral control law to generate a torque so that the actual velocity closely follows the velocity demand. The outer loop uses a proportional control law to generate the velocity demand so that the actual position closely follows the position demand. Disturbance torques due to gravity and other dynamic coupling effects impact the performance of the velocity loop as do variation in the parameters of the plant being controlled, and this in turn lead to errors in position tracking. Gearing reduces the magnitude of disturbance torques by  $1/G$  and the variation in inertia and friction by  $1/G^2$  but at the expense of cost, weight, increased friction and mechanical noise.

The velocity loop performance can be improved by adding an integral control term, or by feedforward of the disturbance torque which is largely predictable. The position loop performance can also be improved by feedforward of the desired joint velocity. In



**Fig. 9.17.** Position loop following an LSPB trajectory. **a** Proportional control only **b** proportional control plus velocity demand feedforward

practice control systems use both feedforward and feedback control. Feedforward is used to inject signals that we can compute, in this case the joint velocity, and in the earlier case the gravity torque. Feedback control compensates for all remaining sources of error including variation in inertia due to manipulator pose and payload, changes in friction with time and temperature, and all the disturbance torques due to velocity and acceleration coupling. In general the use of feedforward allows the feedback gain to be reduced since a large part of the demand signal now comes from the feedforward.

### 9.4.3 Rigid-Body Dynamics Compensation

The previous section showed the limitations of independent joint control and introduced the concept of feedforward to compensate for the gravity disturbance torque. Inertia variation and other dynamic coupling forces were not explicitly dealt with and were left for the feedback controller to handle. However inertia and coupling torques can be computed according to Eq. 9.1 given knowledge of joint angles, joint velocities and accelerations, and the inertial parameters of the links. We can incorporate these torques into the control law using one of two *model-based* approaches: feedforward control, and computed torque control. The structural differences are contrasted in Fig. 9.18.

#### 9.4.3.1 Feedforward Control

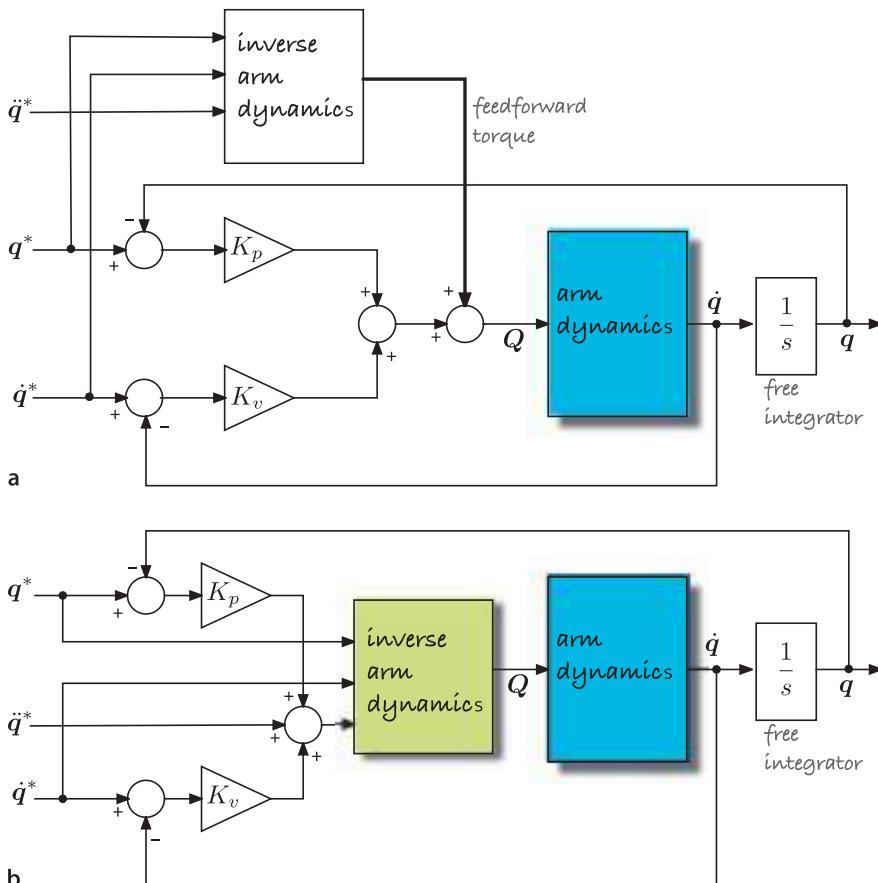
The torque feedforward controller shown in Fig. 9.18a is given by

$$\begin{aligned} Q^* &= \underbrace{M(q^*)\ddot{q}^* + C(q^*, \dot{q}^*)\dot{q}^* + F(\dot{q}^*) + G(q^*)}_{\text{feedforward}} + \underbrace{\{K_v(q^* - \dot{q}) + K_p(q^* - q)\}}_{\text{feedback}} \\ &= \mathcal{D}(q^*, \dot{q}^*, \ddot{q}^*) + \{K_v(\dot{q}^* - \dot{q}) + K_p(q^* - q)\} \end{aligned}$$

where  $K_p$  and  $K_v$  are the position and velocity gain (or damping) matrices respectively, and  $\mathcal{D}(\cdot)$  is the inverse dynamics function. The gain matrices are typically diagonal. The feedforward term provides the joint forces required for the desired manipulator state  $(q^*, \dot{q}^*, \ddot{q}^*)$  and the feedback term compensates for any errors due to uncertainty in the inertial parameters, unmodeled forces or external disturbances.

We can also consider that the feedforward term linearizes the non-linear dynamics about the operating point  $(q^*, \dot{q}^*, \ddot{q}^*)$ . If the linearization is ideal then the dynamics of the error  $e = q^* - q$  are given by

$$M(q^*)\ddot{e} + K_v\dot{e} + K_p e = 0 \quad (9.11)$$



**Fig. 9.18.**  
Manipulator control structures.  
**a** Feedforward control, **b** computed torque control

For well chosen  $K_p$  and  $K_v$ , the error will decay to zero but the joint errors are coupled and their dynamics are dependent on the manipulator configuration.

To test this controller using Simulink® we first create a [SerialLink](#) object

```
>> mdl_puma560
```

and then load the torque feedforward controller model

```
>> sl_fforward
```

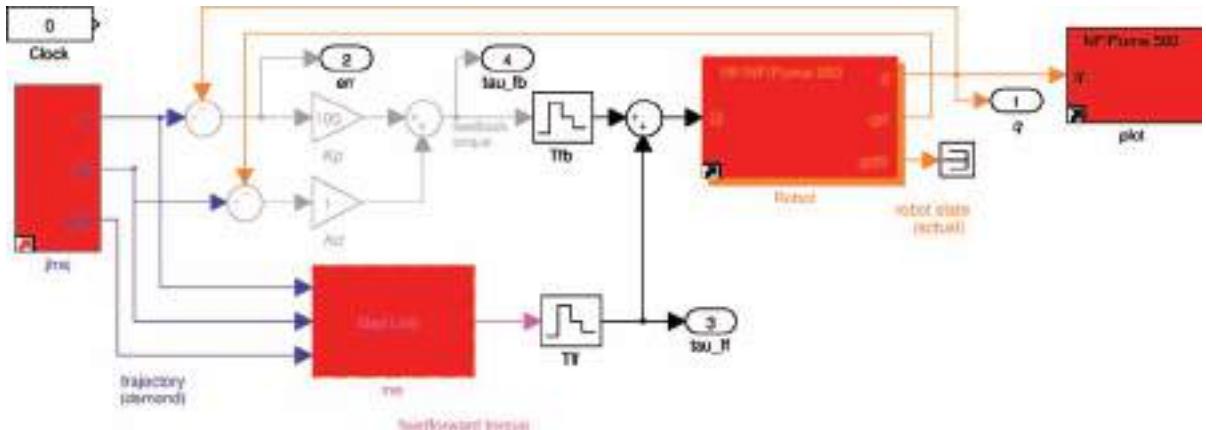
which is shown in Fig. 9.19. The feedforward torque is computed using the [RNE](#) block and added to the feedback torque computed from position and velocity error. The desired joint angles and velocity are generated using a [jtraj](#) block.

Since the robot configuration changes relatively slowly the feedforward torque can be evaluated at a lower rate,  $T_{ff}$  than the error feedback loops,  $T_{fb}$ . This is demonstrated in Fig. 9.19 by the zero-order hold block sampling at the relatively low sample rate of 20 Hz.

#### 9.4.3.2 Computed Torque Control

The computed torque controller is shown in Fig. 9.18b. It belongs to a class of controllers known as inverse dynamic control. The principle is that the non-linear system is cascaded with its inverse so that the overall system has a constant unity gain. In practice the inverse is not perfect so a feedback loop is required to deal with errors.

Due to the non-diagonal matrix  $M$ .



**Fig. 9.19.** The Simulink® model `sl_ffforward` for Puma 560 with torque feedforward control

The computed torque control is given by

$$\begin{aligned} Q &= M(q) \{ K_v(\ddot{q}^* + \dot{q}^* - \dot{q}) + K_p(q^* - q) \} + C(q, \dot{q})\dot{q} + F(\dot{q}) + G(q) \\ &= \mathcal{D}(q, \dot{q}, (\ddot{q}^* + K_v(\dot{q}^* - \dot{q}) + K_p(q^* - q))) \end{aligned}$$

where  $K_p$  and  $K_v$  are the position and velocity gain (or damping) matrices respectively, and  $\mathcal{D}(\cdot)$  is the inverse dynamics function.

In this case the inverse dynamics must be evaluated at each servo interval, although the coefficient matrices  $M$ ,  $C$ , and  $G$  could be evaluated at a lower rate since the robot configuration changes relatively slowly. Assuming ideal modelling and parameterization the error dynamics of the system are

$$\ddot{e} + K_v \dot{e} + K_p e = 0 \quad (9.12)$$

where  $e = q^* - q$ . Unlike the torque feedforward controller the joint errors are uncoupled and their dynamics are therefore independent of manipulator configuration. In the case of model error there will be some coupling between axes, and the right-hand side of Eq. 9.12 will be a non-zero forcing function.

Using Simulink® we first create a `SerialLink` object, remove Coulomb friction and then load the computed torque controller

```
>> mdl_puma560
>> p560 = p560.nofriction();
>> sl_ctorque
```

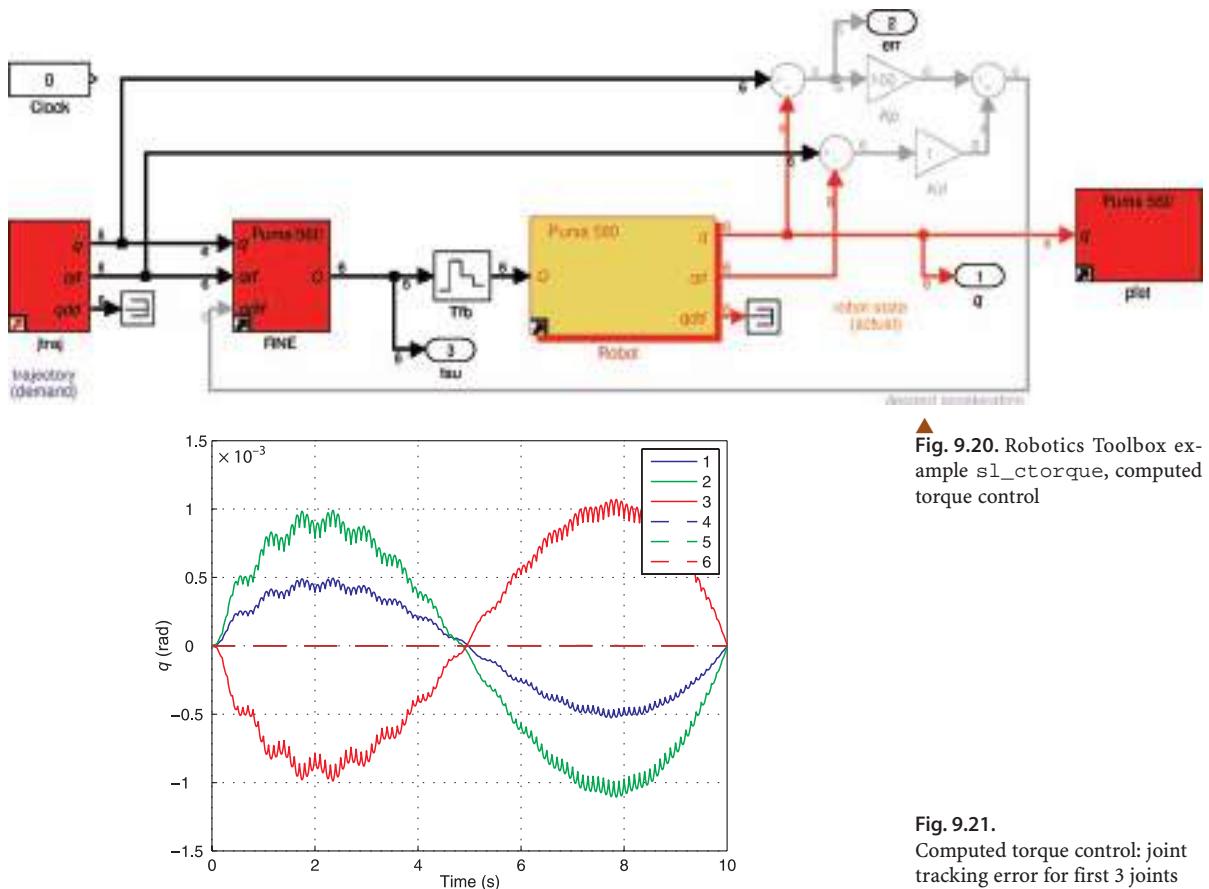
which is shown in Fig. 9.20. The torque is computed using the Toolbox `RNE` block and added to the feedback torque computed from position and velocity error. The desired joint angles and velocity are generated using a `jtraj` block whose parameters are the initial and final joint angles. We run the simulation

```
>> r = sim('sl_ctorque');
>> t = r.find('tout');
>> q = r.find('yout');
```

The tracking error is shown in Fig. 9.21.

#### 9.4.4 Flexible Transmission

In many high-performance robots the flexibility of the transmission between motor and link can be a significant dynamic effect. The flexibility might be caused by torsional flexure of an elastic coupling between the motor and the shaft, the drive shaft itself,



**Fig. 9.20.** Robotics Toolbox example `sl_ctorque`, computed torque control

**Fig. 9.21.**  
Computed torque control: joint tracking error for first 3 joints

**Series-elastic actuator (SEA).** So far we have considered the flexibility between motor and load as a nuisance, but in many situations it has a real benefit. A class of actuators known as series-elastic actuators deliberately introduce a soft spring between the motor and the load. This is useful for robots that interact closely with people since it makes the robot less dangerous in case of collision. For robots that must exert a force as part of their task the extension of the elastic element provides an estimate of the applied force which can be used for control.

or in the case of a cable driven robot, the longitudinal stiffness of the cable. In dynamic terms the result is to add a poorly damped spring between the motor and the load.

We will demonstrate this for the 2-link robot introduced in Sect. 7.2.1. The Simulink® model

```
>> mdl_twolink
>> sl_flex
```

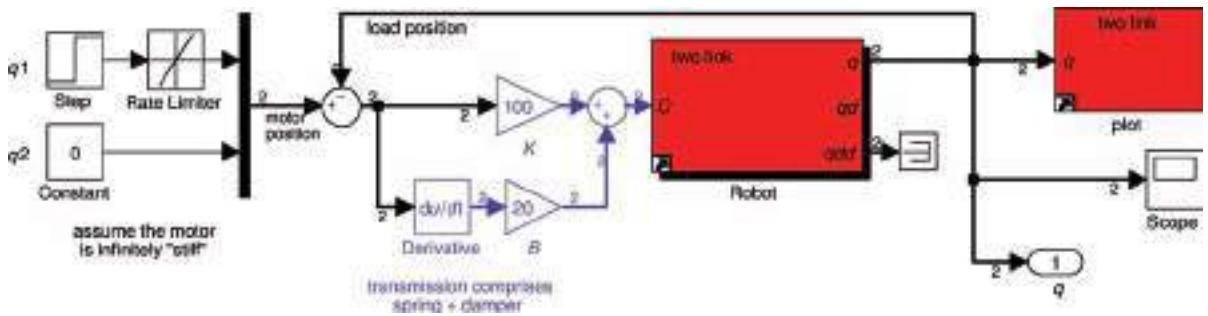
is shown in Fig. 9.22. The compliant drive is modeled by the motor angle  $q_m$  and the link angle  $q_l$ . The torque acting on the link is due to the spring constant and damping

$$\tau_l = K(q_m - q_l) + B(\dot{q}_m - \dot{q}_l)$$

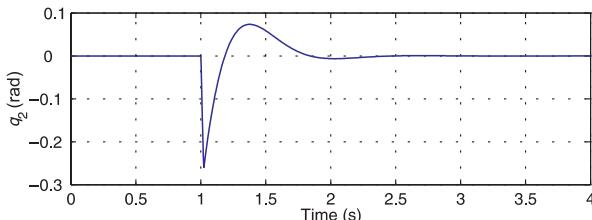
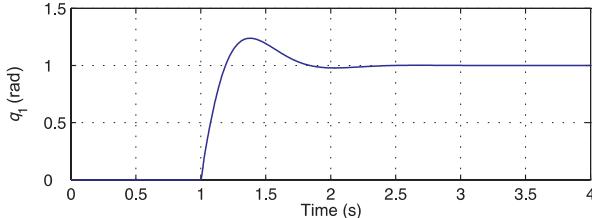
For simplicity we assume that the motor and its control loop is infinitely stiff, that is, the reaction torque from the spring does not influence the angle of the motor.

We run the simulation

```
>> r = sim('sl_flex')
```



**Fig. 9.22.** Robotics Toolbox example `s1_flex`, simple flexible 2-link manipulator



**Fig. 9.23.**  
Flexible link control: joint angle response to step input

and the results are shown in Fig. 9.23. The first joint receives a step position demand change at time 1 s and we see some significant overshoot in the response of both joints. Note that joint 2 has an initial negative error due to the inertial coupling from the acceleration of joint 1.

An effective controller for this system would need to measure the motor and the link angle, or the motor angle and the length of the spring for each joint. A more complex phenomena in high-performance robots is link flexibility, but this cannot be modelled using the rigid-link assumptions of the Toolbox.

## 9.5 Wrapping Up

In this Chapter we learnt how to model the forces and torques acting on the individual links of a serial-link manipulator. The equations of motion or inverse dynamics compute the joint forces required to achieve particular link angle, velocity and acceleration. The equations have terms corresponding to inertia, gravity, velocity coupling, friction and externally applied forces. We looked at the significance of these terms and how they vary with manipulator configuration. The equations of motion provide insight into important issues such as how the velocity or acceleration of one joint exerts a disturbance force on other joints which is important for control design. We then discussed the forward dynamics which describe how the configuration evolves with time in response to forces and torques applied at the joints by the actuators and by external forces such as gravity.

We then discussed approaches to control, starting with the simplest case of independent joint control, and explored the effect of disturbance torques and variation in inertia. We showed how feedforward of disturbances such as gravity could provide significant improvement in performance. We extended the feedforward notion to full model-based control using torque feedforward and computed torque controllers. Finally we discussed the effect of compliance between the robot motor and the link.

## Further Reading

The dynamics of serial-link manipulators is well covered by all the standard robotics textbooks such as Paul (1981), Spong et al. (2006), Siciliano et al. (2008) and the Robotics Handbook (Siciliano and Khatib 2008, § 2). The efficient recursive Newton-Euler method we use today is the culmination of much research in the early 1980s and described in Hollerbach (1982). The equations of motion can be derived via a number of techniques, including Lagrangian (energy based), Newton-Euler, d'Alembert (Fu et al. 1987; Lee et al. 1983) or Kane's (Kane and Levinson 1983) method. However the computational cost of Lagrangian methods (Uicker 1965; Kahn 1969) is enormous,  $O(N^4)$ , which made it infeasible for real-time use on computers of that era and many simplifications and approximation had to be made. Orin et al. (1979) proposed an alternative approach based on the Newton-Euler (NE) equations of rigid-body motion applied to each link. Armstrong (1979) then showed how recursion could be applied resulting in  $O(N)$  complexity. Luh et al. (1980) provided a recursive formulation of the Newton-Euler equations with linear and angular velocities referred to link coordinate frames which resulted in a 1 000-fold improvement in execution time making it practical to implement in real-time. Hollerbach (1980) showed how recursion could be applied to the Lagrangian form, and reduced the computation to within a factor of 3 of the recursive NE form, and Silver (1982) showed the equivalence of the recursive Lagrangian and Newton-Euler forms, and that the difference in efficiency was due to the representation of angular velocity.

The forward dynamics, Sect. 9.3, is computationally more expensive. An  $O(N^3)$  method was proposed by Walker and Orin (1982). Featherstone's (1987) articulated-body method has  $O(N)$  complexity but for  $N < 9$  is more expensive than Walker's method.

Critical to any consideration of robot dynamics is knowledge of the inertial parameters, ten per link, as well as the motor's parameters. Corke and Armstrong-Hélouvy (1994, 1995) published a meta-study of Puma parameters and provide a consensus estimate of inertial and motor parameters for the Puma 560 robot. Some of this data is obtained by painstaking disassembly of the robot and determining the mass and dimensions of the components. Inertia of components can be estimated from mass and dimensions by assuming mass distribution, or it can be measured using a bifilar pendulum.

Alternatively the parameters can be estimated by measuring the joint torques or the base reaction force and moment as the robot moves. A number of early works in this area include Mayeda et al. (1990), Izaguirre and Paul (1985), Khalil and Dombre (2002) and a more recent summary is Siciliano and Khatib (2008, § 14). Key to successful identification is that the robot moves in a way that is sufficiently exciting (Gautier and Khalil 1992; Armstrong 1989). Friction is an important dynamic characteristic and is well described in Armstrong's (1988) thesis. The survey by Armstrong-Hélouvy et al. (1994) is a very readable and thorough treatment of friction modelling and control. A technique for measuring the electrical parameters of motors, such as torque constant and armature resistance, without having to remove the motor from the robot is described by Corke (1996a).

The discussion on control has been quite brief and has strongly emphasized the advantages of feedforward control. Robot joint control techniques are well covered by Spong et al. (2006), Craig (2004) and Siciliano et al. (2008) and summarized in Siciliano and Khatib (2008, § 6). Siciliano et al. have a good discussion of actuators and sensors as does the, now quite old, book by Klafter et al. (1989). The control of flexible joint robots is discussed in Spong et al. (2006). Adaptive control can be used to accomodate the time varying inertial parameters and there is a large literature on this topic but some good early references include the book by Craig (1987) and key papers include Craig et al. (1987), Spong (1989), Middleton and Goodwin (1988) and Ortega and Spong (1989).

Dynamic manipulability is discussed in Spong et al. (2006) and Siciliano et al. (2008). The Asada measure used in the Toolbox is described in Asada (1983).

Newton's Principia was written in Latin but an English translation is available on line at <http://www.archive.org/details/newtonspmathema00newtrich>. His writing on other subjects, including transcripts of his notebooks, can be found online at <http://www.newtonproject.sussex.ac.uk>.

---

### Exercises

1. Run the code on page 194 to compute gravity loading on joints 2 and 3 as a function of configuration. Add a payload and repeat.
2. Run the code on page 195 to show how the inertia of joints 1 and 2 vary with payload?
3. Generate the curve of Fig. 9.2c. Add a payload and compare the results.
4. By what factor does this inertia vary over the joint angle range?
5. Why is the manipulator inertia matrix symmetric?
6. The robot exerts a wrench on the base as it moves (page 198). Consider that the robot is sitting on a frictionless horizontal table (say on a large air puck). Create a simulation model that includes the robot arm dynamics and the sliding dynamics on the table. Show that moving the arm causes the robot to translate and spin. Can you devise an arm motion that moves the robot base from one position to another and stops?
7. Overlay the manipulability ellipsoid on the display of the robot. Compare this with the velocity ellipsoid from Sect. 8.1.4.
8. Independent joint control (page 204)
  - a) Investigate different values of `Kv` and `Ki` as well as demand signal shape and amplitude.
  - b) Perform a root-locus analysis of `vloop` to determine the maximum permissible gain for the proportional case. Repeat this for the PI case.
  - c) Consider that the motor is controlled by a voltage source instead of a current source, and that the motor's impedance is  $1\text{ mH}$  and  $1.6\text{ }\Omega$ . Modify `vloop` accordingly. Extend the model to include the effect of back EMF.
  - d) Increase the required speed of motion so that the motor torque becomes saturated. With integral action you will observe a phenomena known as integral windup – examine what happens to the state of the integrator during the motion. Various strategies are employed to combat this, such as limiting the maximum value of the integrator, or only allowing integral action when the motor is close to its setpoint. Experiment with some of these.
  - e) Create a Simulink® model of the Puma robot with each joint controlled by `vloop` and `ploop`. Parameters for the different motors in the Puma are described in Corke and Armstrong-Hélouvy (1995).
9. The motor torque constant has units of  $\text{N m A}^{-1}$  and is equal to the back EMF constant which has units of  $\text{V s rad}^{-1}$ . Show that these units are equivalent.
10. Model-based control (page 211)
  - a) Compute and display the joint tracking error for the torque feedforward and computed torque cases. Experiment with different motions, control parameters and samplerate  $T_{fb}$ .
  - b) Reduce the rate at which the feedforward torque is computed and observe its effect on tracking error.
  - c) In practice the dynamic model of the robot is not exactly known, we can only invert our best estimate of the rigid-body dynamics. In simulation we can model this by using the `perturb` method, see the online documentation, which returns a robot object with inertial parameters varied by plus and minus the specified percentage. Modify the Simulink® models so that the `RNE` block is using a

robot model with parameters perturbed by 10%. This means that the inverse dynamics are computed for a slightly different dynamic model to the robot under control and shows the effect of model error on control performance. Investigate the effects on error for both the torque feedforward and computed torque cases.

11. Flexible drive robot (page 213)

- a) Experiment with different values of joint stiffness and control parameters.
- b) Modify the simulation to eliminate the assumption that the motor is infinitely stiff. That is, replace the motor position with a nested control loop that is subject to a disturbance torque from the elastic element. Tune the controller for good performance.
- c) Introduce a rigid object into the environment and modify the simulation so that the robot arm moves to touch the object and applies no more than 5 N force to it.

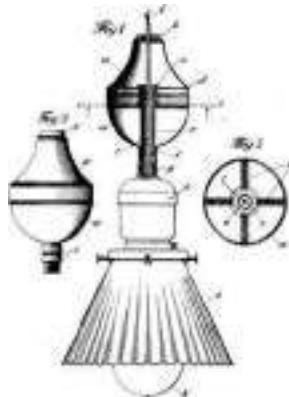
# 10

# Light and Color

*I cannot pretend to feel impartial about colours.*

*I rejoice with the brilliant ones  
and am genuinely sorry for the poor browns.*

Winston Churchill



In ancient times it was believed that the eye radiated a cone of visual flux which mixed with visible objects in the world to create a sensation in the observer, like the sense of touch, the extromission theory. Today we consider that light from an illuminant falls on the scene, some of which is reflected into the eye of the observer to create a perception about that scene. The light that reaches the eye, or the camera, is a function of the illumination impinging on the scene and the material property known as reflectivity.

This chapter is about light itself and our perception of light in terms of brightness and color. Section 10.1 describes light in terms of electro-magnetic radiation and mixtures of light as continuous spectra. Section 10.2 provides a brief introduction to colorimetry, the science of color perception, human trichromatic color perception and how colors can be represented in various color spaces. Section 10.3 covers a number of advanced topics such as color constancy, gamma correction, and an example concerned with distinguishing different colored objects in an image.

## 10.1 Spectral Representation of Light

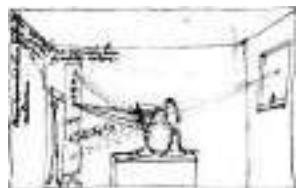
Around 1670 Sir Isaac Newton discovered that white light was a mixture of different colors. We now know that each of these colors is a single frequency or wavelength of electro-magnetic radiation. We perceive the wavelengths between 400 and 700 nm as colors as shown in Fig. 10.1.

In general the light that we observe is a mixture of many wavelengths and can be represented as a function  $E(\lambda)$  that describes intensity as a function of wavelength  $\lambda$ . Monochromatic light from a laser that emits light at a single wavelength in which case  $E$  is an impulse.

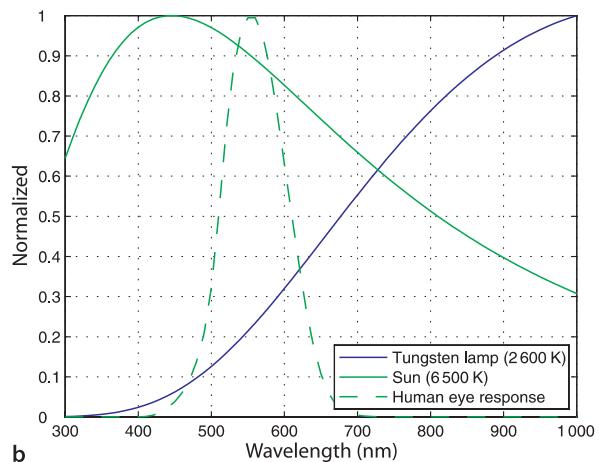
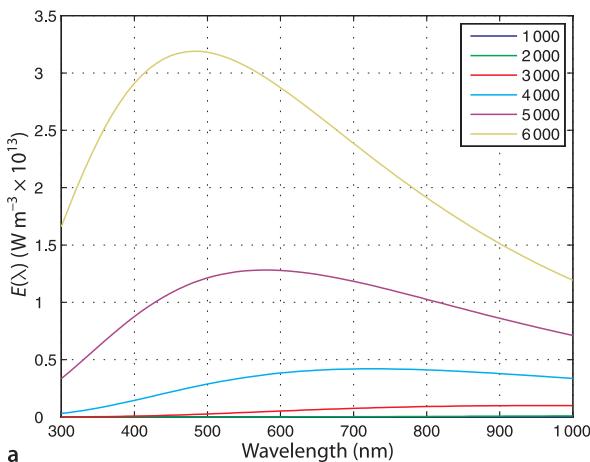
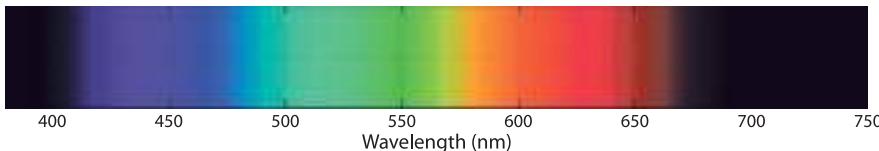
The most common source of light is incandescence which is the emission of light from a hot body such as the Sun or the filament of a light bulb. In physics this is modeled as a blackbody radiator or Planckian source. The emitted power is a function of wavelength  $\lambda$  and given by Planck's radiation formula

$$E(\lambda) = \frac{2\pi hc^2}{\lambda^5(e^{hc/k\lambda T} - 1)} \text{ W m}^{-2} \text{ m}^{-1} \quad (10.1)$$

$$\begin{aligned} c &= 2.998 \times 10^8 \text{ m s}^{-1}, \\ h &= 6.626 \times 10^{-34} \text{ Js}, \\ k &= 1.381 \times 10^{-23} \text{ J K}^{-1}. \end{aligned}$$



**Spectrum of light.** During the plague years of 1665–1666 Isaac Newton developed his theory of light and color. He demonstrated that a prism could decompose white light into a spectrum of colours, and that a lens and a second prism could recompose the multicoloured spectrum into white light. Importantly he showed that the color of the light did not change when it was reflected from different objects, from which he concluded that color is an intrinsic property of light not the object. (Newton's sketch to the left)



We can plot the emission spectra for a blackbody at different temperatures. First we define a range of wavelengths

```
>> lambda = [300:10:1000]*1e-9;
```

in this case from 300 to 1000 nm, and then compute the blackbody spectra

```
>> for T=1000:1000:6000
>> plot( lambda*1e9, blackbody(lambda, T)); hold all
>> end
```

as shown in Fig. 10.2a. We can see that as temperature increases the maximum amount of power increases and the wavelength at which the peak occurs decreases. The total amount of power radiated is the area under the blackbody curve and is given by the Stefan-Boltzman law

$$\frac{2\pi^5 k^4}{15c^2 h^3} T^4 \text{ W m}^{-2}$$

and the wavelength corresponding to the peak of the blackbody curve is given by Wien's displacement law

$$\lambda_{\max} = \frac{2.8978 \times 10^{-3}}{T} \text{ m}$$

The wavelength of the peak decreases with increasing temperature and in familiar terms this is what we observe when we heat an object. It starts to glow faintly red at around 800 K and moves through orange and yellow toward white as temperature increases.►

The filament of tungsten lamp has a temperature of 2600 K and glows *white hot*. The Sun has a surface temperature of 6500 K. The spectra of these sources

```
>> lamp = blackbody(lambda, 2600);
>> sun = blackbody(lambda, 6500);
>> plot(lambda*1e9, [lamp/max(lamp) sun/max(sun)])
```

**Fig. 10.1.**

The spectrum of visible colors as a function of wavelength in nanometres. The visible range depends on viewing conditions and the individual but is generally accepted as being the range 400–700 nm. Wavelengths greater than 700 nm are termed infra-red and those below 400 nm are ultra-violet

**Fig. 10.2.** Blackbody spectra.  
**a** Blackbody emission spectra for temperatures from 1000–6000 K.  
**b** Blackbody emissions for the Sun (6500 K), a tungsten lamp (2600 K) and the response of the human eye – all normalized to unity for readability

Incipient red heat	770–820 K,
dark red heat	920–1020 K,
bright red heat	1120–1220 K,
yellowish red heat	1320–1420 K,
incipient white heat	1520–1620 K,
white heat	1720–1820 K.



Sir Humphry Davy demonstrated the first electrical incandescent lamp using a platinum filament in 1802. Sir Joseph Swan demonstrated his first light bulbs in 1850 using carbonized paper filaments. However it was not until advances in vacuum pumps in 1865 that such lamps could achieve a useful lifetime. Swan patented a carbonized cotton filament in 1878 and a carbonized cellulose filament in 1881. His lamps came into use after 1880 and the Savoy Theatre in London was completely lit by electricity in 1881. In the USA Thomas Edison did not start research into incandescent lamps until 1878 but he patented a long-lasting carbonized bamboo filament the next year and was able to mass produce them. The Swan and Edison companies merged in 1883.

The light bulb subsequently became the dominant source of light on the planet but is now being phased out due to its poor energy efficiency. (Photo by Douglas Brackett, Inv., Edisonian.com)

are compared in Fig. 10.2b. The tungsten lamp curve is much lower in magnitude, but has been scaled up for readability. The peak of the Sun's emission is around 450 nm and it emits a significant amount of power in the visible part of the spectrum. The peak for the tungsten lamp is at a much longer wavelength and perversely very little of its power falls within the human visible region. The bulk of the power is infra-red which we perceive as heat not light.

### 10.1.1 Absorption

The Sun's spectrum at ground level on the Earth has been measured and tabulated

```
>> sun_ground = loadspectrum(lambda, 'solar.dat');
>> plot(lambda*1e9, sun_ground)
```

and is shown in Fig. 10.3a. It differs markedly from that of a blackbody since some wavelengths have been absorbed more than others by the atmosphere. Our eye's peak sensitivity has evolved to be closely aligned to the peak of the spectrum of atmospherically filtered sunlight.

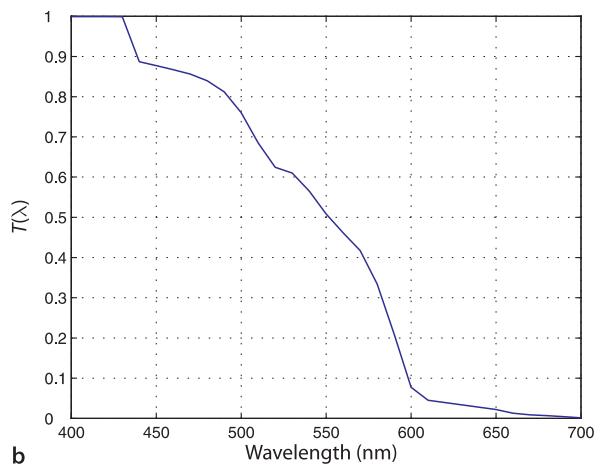
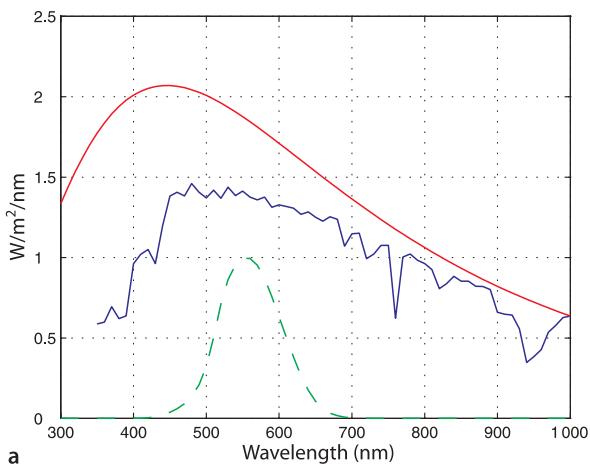
Transmission  $T$  is the inverse of absorption and is the fraction of light passed as a function of wavelength. It is described by Beer's law

$$T = 10^{-Ad} \quad (10.2)$$

where  $A$  is the absorption coefficient in units of  $\text{m}^{-1}$  and  $d$  is the path length. The absorption spectrum  $A(\lambda)$  for water is loaded from tabulated data

```
>> [A, lambda] = loadspectrum([400:10:700]*1e-9, 'water.dat');
```

**Fig. 10.3.** a Modified solar spectrum at ground level (blue). The dips in the solar spectrum correspond to various water absorption bands.  $\text{CO}_2$  absorbs radiation in the infra-red region, and ozone  $\text{O}_3$  absorbs strongly in the ultra-violet region. The Sun's blackbody spectrum (normalized) is shown in red and the response of the human eye is shown dashed. b Transmission through 5 m of water. The longer wavelengths, reds, have been strongly attenuated



and the transmission through 5 m of water is

```
>> d = 5;
>> T = 10.^(-A*d);
>> plot(lambda*1e9, T);
```

which is plotted in Fig. 10.3b. Differential absorption of wavelengths is a significant concern when imaging underwater and we revisit this topic in Sect. 10.3.1.

### 10.1.2 Reflection

The light reflected from a surface, its luminance, has a spectrum given by

$$L(\lambda) = E(\lambda)R(\lambda) \text{ W m}^{-2} \quad (10.3)$$

where  $E$  is the incident illumination and  $R \in [0, 1]$  is the reflectivity or reflectance of the surface and is a function of wavelength. White paper for example has a reflectance of around 70%. The reflectance spectra of many materials have been measured and tabulated.► Consider for example the reflectivity of a red house brick

```
>> [R, lambda] = loadspectrum([100:10:10000]*1e-9, 'redbrick.dat');
>> plot(lambda*1e6, R);
```

which is plotted in Fig. 10.4. We see that it reflects red colors more than blue.

The illuminance of the Sun in the visible region

```
>> lambda = [400:10:700]*1e-9; % visible spectrum
```

is

```
>> E = loadspectrum(lambda, 'solar.dat');
```

at ground level. The reflectivity of the brick is

```
>> R = loadspectrum(lambda, 'redbrick.dat');
```

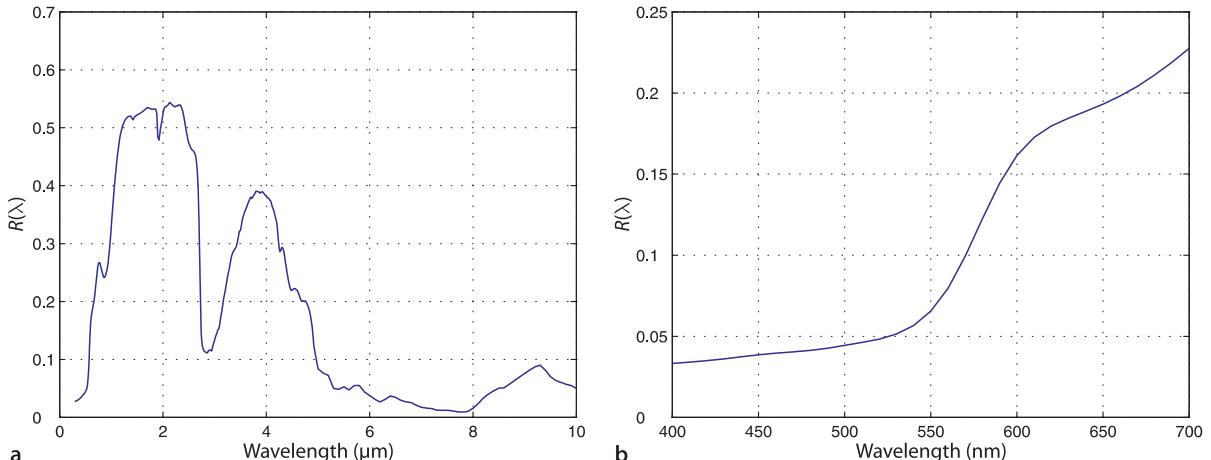
and the light reflected from the brick is

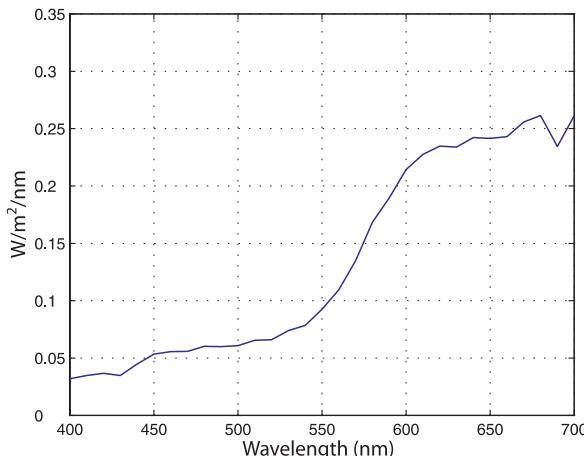
```
>> L = E .* R;
>> plot(lambda*1e9, L);
```

which is shown in Fig. 10.5. It is this spectrum that is interpreted by our eyes as the color red.

From <http://speclib.jpl.nasa.gov/>  
weathered red brick (041UUUBRK).

**Fig. 10.4.** Reflectance of a weathered red house brick (data from ASTER, Baldridge et al. 2009). **a** Full range measured from 300 nm visible to 10 000 nm (infrared); **b** closeup of visible region



**Fig. 10.5.**

Luminance of the weathered red house brick under illumination from the Sun at ground level, based on data from Fig. 10.3a and 10.4b

## 10.2 Color

*Color is the general name for all sensations arising from the activity of the retina of the eye and its attached nervous mechanisms, this activity being, in nearly every case in the normal individual, a specific response to radiant energy of certain wavelengths and intensities.*

T. L. Troland,  
Report of Optical Society of America  
Committee on Colorimetry 1920–1921

We have described the spectra of light in terms of power as a function of wavelength, but our own perception of light is in terms of subjective quantities such as brightness and color. Light that is visible to humans lies in the range of wavelengths from 400 nm (violet) to 700 nm (red) with the colors blue, green, yellow and orange in between, as shown in Fig. 10.1.

Our eyes contain two types of light sensitive cells as shown in Fig. 10.6. Cone cells respond to particular colors and provide us with our normal daytime vision. Rod cells are much more sensitive than cone cells but respond to intensity only and are used at night.◀

The brightness we associate with a particular wavelengths is known as luminosity and is measured in units of lumens per watt. For our daylight cone-cell vision the luminosity as a function of wavelength has been experimentally determined, tabulated and forms the basis of the 1931 CIE standard that represents the average human observer.◀ The luminosity function is provided by the Toolbox

```
>> human = luminos(lambda);
>> plot(lambda*1e9, human)
```

and is shown in Fig. 10.7a. Consider two lights emitting the same power (in watts) but one has a wavelength of 550 nm (green) and the other has a wavelength of 450 nm (blue). The perceived brightness of these two lights is quite different, in fact the blue light appears only

```
>> luminos(450e-9) / luminos(550e-9)
ans =
0.0382
```

or 3.8% as bright as the green one.

Therefore at night you have no color vision.

This is the photopic response for a light-adapted eye using the cone photoreceptor cells. The dark adapted, or scotopic response, using the eye's monochromatic rod photoreceptor cells is different, and peaks at around 510 nm.

**Radiometric and photometric quantities.** Two quite different sets of units are used when discussing light: radiometric and photometric. Radiometric units are used in Sect. 10.1 and are based on quantities such as power and are expressed in familiar SI units such as watts.

Photometric units are analogs of radiometric units but take into account the *visual sensation* in the observer. Luminous power or luminous flux is the *perceived* power of a light source and is measured in *lumens* (abbreviated to lm) rather than *watts*. A 1 W light source at 555 nm, the peak response, by definition emits a luminous flux of 683 lm. By contrast a 1 W light source at 800 nm emits a luminous flux of 0 lm – it causes no visual sensation at all.

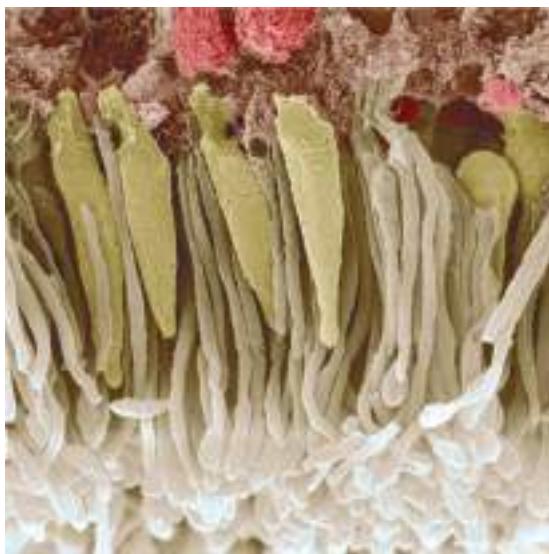
A 1 W incandescent lightbulb however produces a perceived visual sensation of less than 15 lm or a luminous efficiency of 15 lm W<sup>-1</sup>. Fluorescent lamps achieve efficiencies up to 100 lm W<sup>-1</sup> and white LEDs up to 150 lm W<sup>-1</sup>.

The eyes of different species have different spectral responses. Many insects are able to see well into the ultra-violet region of the spectrum. The silicon sensors used in digital cameras have strong sensitivity in the red and infra-red part of the spectrum ▶ which we can also plot

```
>> camera = ccdresponse(lambda);
>> hold on
>> plot(lambda*1e9, camera*max(human), '--')
```

and is shown superimposed in Fig. 10.7.

The LED on an infra-red remote control can be seen as a bright light in most digital cameras – try this with your mobile phone camera and TV remote. Some security cameras provide infra-red scene illumination for covert night time monitoring. Note that some cameras are fitted with infra-red filters to prevent the sensor becoming saturated by ambient infra-red radiation.



◀Fig. 10.6.

A coloured scanning electron micrograph of rod cells (white) and cone cells (yellow) in the human eye. The cell diameters are in the range 0.5–4 µm. The cells contain different types of light-sensitive protein called photopsin. The cell bodies (red) of the receptor cells are located in a layer above the rods and cones

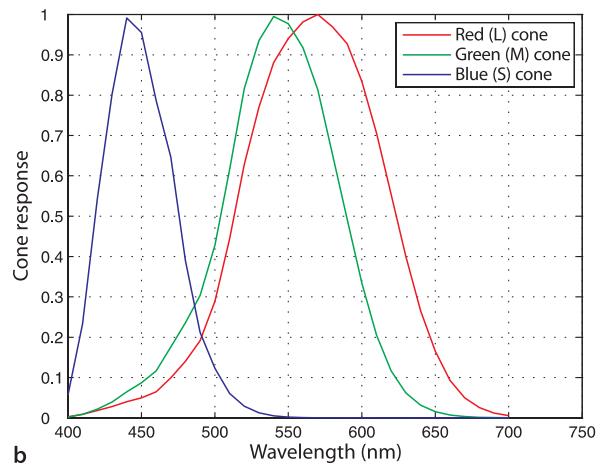
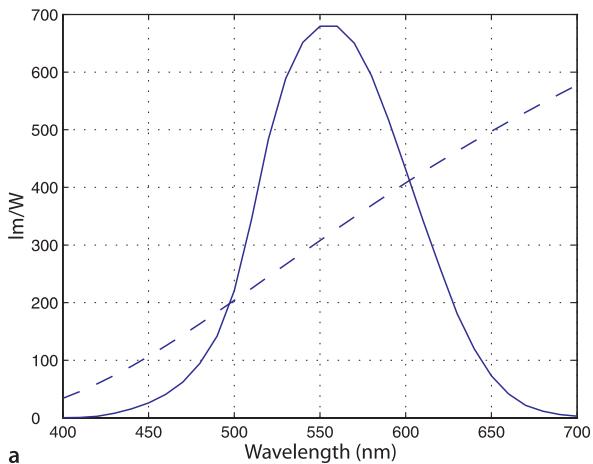


Fig. 10.7. a Luminosity curve for standard observer human observer. The peak response is 683 lm W<sup>-1</sup> at 555 nm (green). The response of a silicon CCD camera is shown dashed for comparison. b Spectral response of human cones (normalized)

Solid angle is measured in steradians, a full sphere is  $4\pi$  sr.

**Lightmeters, illuminance and luminance.** A photographic lightmeter measures luminous flux which has units of  $\text{lm m}^{-2}$  or lux (lx). The luminous intensity of a point light source is the luminous flux per unit solid angle<sup>►</sup> measured in  $\text{lm sr}^{-1}$  or candelas (cd). For a point source of luminous intensity  $I$  the illuminance  $E$  falling normally onto a surface is

$$E = \frac{I}{d^2} \text{ lx}$$

where  $d$  is the distance between source and the surface. Outdoor illuminance on a bright sunny day is approximately 10000 lx whereas office lighting levels are typically around 1000 lx.

The luminance or *brightness* of a surface is

$$L_s = E_i \cos\theta \text{ nt}$$

which has units of  $\text{cd m}^{-2}$  or nit (nt), and where  $E_i$  is the incident illuminance at an angle  $\theta$  to the surface normal.

In normal daylight conditions our cone photoreceptors are active and these are color sensitive. Humans are trichromats and have three types of cones that respond to different parts of the spectrum. They are referred to as long (L), medium (M) and short (S) according to the wavelength of their peak response, or more commonly as red, green and blue. The spectral responses of the cones can be loaded

```
>> cones = loadspectrum(lambda, 'cones.dat');
>> plot(lambda*1e9, cones)
```

where `cones` has three columns for each of the L, M and S cone responses and each row corresponds to the wavelength in `lambda`. The spectral response of the cones  $L(\lambda)$ ,  $M(\lambda)$  and  $S(\lambda)$  are shown in Fig. 10.7b.<sup>►</sup>

Other species have different numbers of cones. Birds, fish and amphibians are tetrachromats, that is, they have four types of cones. Most other mammals, for instance dogs, are dichromats and have only two types of cones. There is speculation that some human females are tetrachromats.<sup>►</sup>

The retina of the human eye has a central or foveal region which is only 0.6 mm in diameter and contains most of the 6 million cone cells: 65% sense red, 33% sense green and only 2% sense blue. We unconsciously scan our high-resolution fovea over the world to build a large-scale mental image of our surrounds. In addition there are 120 million rod cells, which are also motion sensitive, distributed over the retina.

The sensor in a digital camera is analogous to the retina, but instead of rod and cone cells there is a regular array of light sensitive photosites on a silicon chip. Each photosite is of the order  $1-10 \mu\text{m}$  square and outputs a signal proportional to the intensity of the light falling over its area.<sup>►</sup> For a color camera the photosites are covered by color filters which pass either red, green or blue light to the photosites. The spectral response of the filters is the functional equivalent of the cones' response  $M(\lambda)$  in Fig. 10.7b. A very common arrangement of color filters is the Bayer pattern shown in Fig. 10.8. It uses a regular  $2 \times 2$  photosite pattern comprising two green filters, one red and one blue.<sup>►</sup>

The luminance of an object  $L(\lambda)$  given by Eq. 10.3 results in a particular response from each of the three cones

$$\begin{aligned} \rho &= \int_{\lambda} L(\lambda) M_r(\lambda) d\lambda \\ \gamma &= \int_{\lambda} L(\lambda) M_g(\lambda) d\lambda \\ \beta &= \int_{\lambda} L(\lambda) M_b(\lambda) d\lambda \end{aligned} \tag{10.4}$$

where  $M_r(\lambda)$ ,  $M_g(\lambda)$  and  $M_b(\lambda)$  are the spectral response of the red, green and blue cones respectively as shown in Fig. 10.7b. The response is a 3-vector  $(\rho, \gamma, \beta)$  which is known as a tristimulus.

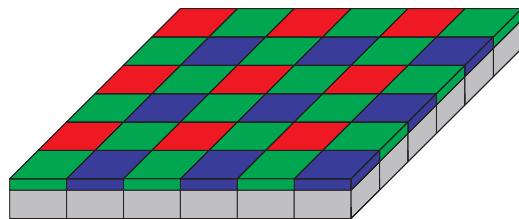
The spectral characteristics are due to the different photopsins in the cone cell. George Wald won the 1967 Nobel Prize in Medicine for his experiments in the 1950s that showed the absorbance of retinal photopsins.

They have an extra variant version of the long-wave (L) cone type which would lead to greater ability in color discrimination.

More correctly the output is proportional to the total number of photons captured by the photosite since the last time it was read. See page 260.

Each pixel therefore cannot provide independent measurements of red, green and blue but it can be estimated. For example, the amount of red at a blue sensitive pixel is obtained by interpolation from its red filtered neighbours. More expensive "3 CCD" cameras can make independent measurements at each pixel since the light is split by a set of prisms, filtered and presented to one CCD array for each primary color. Digital camera raw image files contain the actual outputs of the Bayer-filtered photosites.

**Color blindness**, or color deficiency, is the inability to perceive differences between some of the colors that others can distinguish. Protanopia, deutanopia, tritanopia refer to the absence of the L, M and S cones respectively. More common conditions are protanomaly, deutanomaly and tritanomaly where the cone pigments are mutated and the peak response frequency changed. It is most commonly a genetic condition since the red and green photopsins are coded in the X chromosome. The most common form (occurring in 6% of males including the author) is deutanomaly where the M-cone's response is shifted toward the red end of the spectrum resulting in reduced sensitivity to greens and poor discrimination of hues in the red, orange, yellow and green region of the spectrum.



	red	green	blue
$\lambda$ (nm)	700.0	546.1	435.8

For our red brick example the tristimulus can be computed by approximating the integrals of Eq. 10.4 as a summation

```
>> sum( (L*ones(1,3)) .* cones )
ans =
    16.3578    10.0702    2.8219
```

The dominant response is from the L cone, which is unsurprising since we know that the brick is red.

An arbitrary continuous spectrum is an infinite-dimensional vector and cannot be uniquely represented by just 3 parameters. A consequence of this is that many *different* spectral power distributions will produce the *same* visual stimulus and these are referred to as metamers. More important is the corollary – an arbitrary visual stimulus can be generated by a mixture of just three monochromatic stimuli. These are the three primary colors we learnt about as children.► There is no unique set of primaries – any three will do so long as none of them can be matched by a combination of the others. The wavelength of the CIE 1976 standard primaries are given in Table 10.1.

## 10.2.1 Reproducing Colors

A computer or television display is able to produce a variable amount of each of three primaries at every pixel. The primaries for a cathode ray tube (CRT) are created by exciting phosphors on the back of the screen. For a liquid crystal display (LCD) the colors are obtained by filtering white light emitted by the backlight. The important problem is to determine how much of each primary is required to match a given tristimulus.

We start by considering a monochromatic stimulus of wavelength  $\lambda_s$  which is defined as

$$L(\lambda) = \begin{cases} L_\lambda & \text{if } \lambda = \lambda_s \\ 0 & \text{otherwise} \end{cases}$$

**Fig. 10.8.**

Bayer filtering. The grey blocks represents the array of light-sensitive silicon photosites over which is an array of red, green and blue filters. Invented by Bryce E. Bayer of Eastman Kodak, U.S. Patent 3,971,065.

**Table 10.1.**

The CIE 1976 primaries (Commission Internationale de L'Éclairage 1987) are spectral colors corresponding to the emission lines in a mercury vapor lamp

Primary colors are not a fundamental property of light – they are a fundamental property of the observer. There are three primary colors only because we, as trichromats, have three types of cones. Birds would have four primary colors and dogs would have two.

The response of the cones to this stimulus is given by Eq. 10.4 but because  $L(\cdot)$  is an impulse we can drop the integral to obtain the tristimulus

$$\begin{aligned}\rho &= L_\lambda M_r(\lambda_s) \\ \gamma &= L_\lambda M_g(\lambda_s) \\ \beta &= L_\lambda M_b(\lambda_s)\end{aligned}\tag{10.5}$$

The units are chosen such that equal quantities of the primaries are required to match the equal-energy white stimulus.

Consider next three primary light sources denoted R, G and B with wavelengths  $\lambda_r$ ,  $\lambda_g$  and  $\lambda_b$  and intensities  $R$ ,  $G$  and  $B$  respectively. The tristimulus from these light sources is

$$\begin{aligned}\rho &= RM_r(\lambda_r) + GM_r(\lambda_g) + BM_r(\lambda_b) \\ \gamma &= RM_g(\lambda_r) + GM_g(\lambda_g) + BM_g(\lambda_b) \\ \beta &= RM_b(\lambda_r) + GM_b(\lambda_g) + BM_b(\lambda_b)\end{aligned}\tag{10.6}$$

For the perceived color of these three light sources to match that of the monochromatic stimulus the two tristimuli must be equal. We equate Eq. 10.5 and Eq. 10.6 and write compactly in matrix form as

$$L_\lambda \begin{pmatrix} M_r(\lambda_s) \\ M_g(\lambda_s) \\ M_b(\lambda_s) \end{pmatrix} = \begin{pmatrix} M_r(\lambda_r) & M_r(\lambda_g) & M_r(\lambda_b) \\ M_g(\lambda_r) & M_g(\lambda_g) & M_g(\lambda_b) \\ M_b(\lambda_r) & M_b(\lambda_g) & M_b(\lambda_b) \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

which we can invert to give the required amounts of primary colors

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = L_\lambda \begin{pmatrix} M_r(\lambda_r) & M_r(\lambda_g) & M_r(\lambda_b) \\ M_g(\lambda_r) & M_g(\lambda_g) & M_g(\lambda_b) \\ M_b(\lambda_r) & M_b(\lambda_g) & M_b(\lambda_b) \end{pmatrix}^{-1} \begin{pmatrix} M_r(\lambda_s) \\ M_g(\lambda_s) \\ M_b(\lambda_s) \end{pmatrix}$$

The required tristimulus values are simply a linear transformation of the cone's response to the monochromatic excitation. The transformation matrix is constant, but depends upon the spectral response of the cones to the chosen primaries ( $\lambda_r$ ,  $\lambda_g$ ,  $\lambda_b$ ). This is the basis of trichromatic matching.

**Color matching experiments** are performed using a light source comprising three adjustable lamps that correspond to the primary colors and whose intensity can be individually adjusted. The lights are mixed and diffused and compared to some test color. In color matching notation the primaries, the lamps, are denoted by R, G and B, and their intensities are R, G and B respectively. The three lamp intensities are adjusted by a human subject until they appear to match the test color. This is denoted

$$C \equiv RR + GG + BB$$

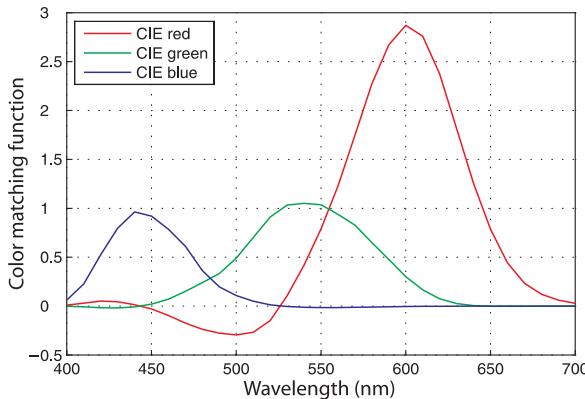
which is read as the visual stimulus C (the test color) is matched by, or looks the same as, a mixture of the three primaries with brightness R, G and B. The notation RR can be considered as the lamp R at intensity R.

Experiments show that color matching obeys the algebraic rules of additivity and linearity which is known as Grassmann's laws. For example two light stimuli  $C_1$  and  $C_2$

$$\begin{aligned}C_1 &\equiv R_1R + G_1G + B_1B \\ C_2 &\equiv R_2R + G_2G + B_2B\end{aligned}$$

when mixed will match

$$C_1 + C_2 \equiv (R_1 + R_2)R + (G_1 + G_2)G + (B_1 + B_2)B$$



**Fig. 10.9.**  
The 1931 color matching functions for the standard observer, based on the CIE standard primaries

We can write this in an even more compact form

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} \bar{r}(\lambda_s) \\ \bar{g}(\lambda_s) \\ \bar{b}(\lambda_s) \end{pmatrix} \quad (10.7)$$

where  $\bar{r}(\lambda)$ ,  $\bar{g}(\lambda)$ ,  $\bar{b}(\lambda)$  are known as color matching functions. These functions have been tabulated for the standard CIE primaries listed in Table 10.1 and are returned by the function `cmfrgb`

```
>> lambda = [400:10:700]*1e-9;
>> cmf = cmfrgb(lambda);
>> plot(lambda*1e9, cmf);
```

and shown graphically in Fig. 10.9. Each curve shows how much of the corresponding primary is required to match the monochromatic light of wavelength  $\lambda$ .

For example to create the sensation of light at 600 nm (orange) we would need

```
>> orange = cmfrgb(600e-9)
orange =
    2.8717    0.3007   -0.0043
```

Surprisingly this requires a very small *negative* amount of the blue primary. To create 500 nm green we would need

```
>> green = cmfrgb(500e-9)
green =
   -0.2950    0.4906    0.1075
```

and this requires a significant *negative* amount of the red primary. This is problematic since a light source cannot have a negative luminance.

We reconcile this by adding some white light ( $R = G = B = w$ , see Sect. 10.2.6) so that the tristimulus values are all positive. For instance

```
>> w = -green(1);
>> white = [w w w];
>> feasible_green = green + white
feasible_green =
    0    0.7856    0.4025
```

If we looked at this color side-by-side with the desired 500 nm green we would say that the generated color had the correct hue but was not as *saturated*.

Saturation refers to the purity of the color. Spectral colors are *fully saturated* but become less saturated (more pastel) as increasing amounts of white is added. In this case we have mixed a stimulus of

```
>> white
white =
    0.2950    0.2950    0.2950
```

which is a light grey.

This leads to a very important point about color reproduction – it is *not* possible to reproduce every possible color using just three primaries. This makes intuitive sense since a color is properly represented as an infinite-dimensional spectral function and a 3-vector can only approximate it. To understand this more fully we need to consider chromaticity spaces.

The Toolbox function `cmfrgb` can also compute the CIE tristimulus for an arbitrary spectral response. The luminance spectrum of the redbrick illuminated by sunlight at ground level was computed earlier and its tristimulus is

```
>> RGB_brick = cmfrgb(lambda, L)
RGB_brick =
    0.6137    0.1416    0.0374
```

These are the respective amounts of the three CIE primaries that are perceived as having the same color as the brick.

### 10.2.2 Chromaticity Space

The tristimulus values describe color as well as brightness. Relative tristimulus values are obtained by normalizing the tristimulus values

$$r = \frac{R}{R + G + B}, g = \frac{G}{R + G + B}, b = \frac{B}{R + G + B} \quad (10.8)$$

which results in chromaticity coordinates  $r$ ,  $g$  and  $b$  that are invariant to overall brightness. By definition  $r + g + b = 1$  so one coordinate is redundant and typically only  $r$  and  $g$  are considered. Since the effect of intensity has been eliminated the 2-dimensional quantity  $(r, g)$  represents *color*.

We can plot the locus of spectral colors, the colors of the rainbow, on the chromaticity diagram using a variant of the color-matching functions

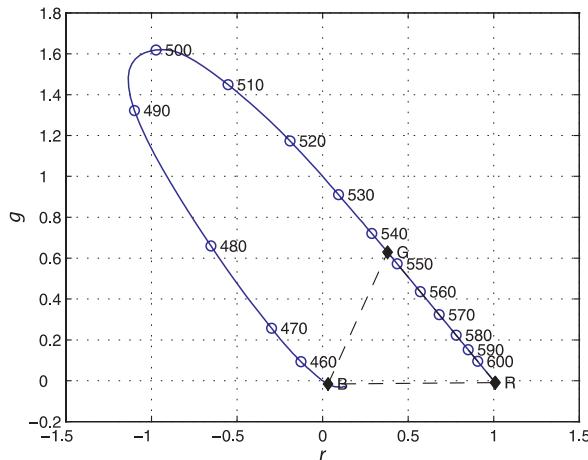
```
>> [r,g] = lambda2rg( [400:700]*1e-9 );
>> plot(r, g)
>> rg_adddticks
```

which results in the horseshoe-shaped curve shown in Fig. 10.10. The Toolbox function `lambda2rg` computes the color matching function Eq. 10.7 for the specified wavelength and then converts the tristimulus value to chromaticity coordinates using Eq. 10.8.

The CIE primaries listed in Table 10.1 can be plotted as well

```
>> primaries = cmfrgb( [700, 546.1, 435.8]*1e-9 );
>> plot(primaries(:,1), primaries(:,2), 'd')
```

and are shown as diamonds in Fig. 10.10.



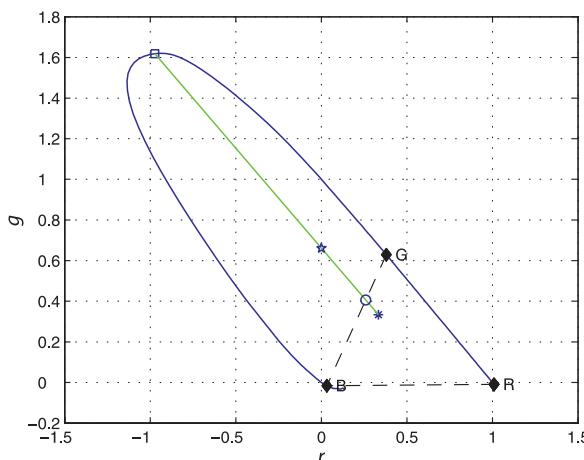
**Fig. 10.10.**

The spectral locus on the  $r$ - $g$  chromaticity plane. The CIE standard primary colors are marked by diamonds. Spectral wavelengths (in nm) are marked. The straight line joining the extremities is the purple boundary and is the locus of saturated purples

**Colorimetric standards.** Colorimetry is a complex topic and standards are very important. Two organizations, CIE and ITU, play a leading role in this area.

The Commission Internationale de l'Eclairage (CIE) or International Commission on Illumination was founded in 1913 and is an independent non-profit organisation that is devoted to worldwide cooperation and the exchange of information on all matters relating to the science and art of light and lighting, colour and vision, and image technology. The CIE's eighth session was held at Cambridge, UK, in 1931 and established international agreement on colorimetric specifications and formalized the XYZ color space. The CIE is recognized by ISO as an international standardization body. See <http://www.cie.co.at> for more information and CIE datasets.

The International Telecommunication Union (ITU) is an agency of the United Nations and was established to standardize and regulate international radio and telecommunications. It was founded as the International Telegraph Union in Paris on 17 May 1865. The International Radio Consultative Committee or CCIR (Comité Consultatif International des Radiocommunications) became, in 1992, the Radiocommunication Bureau of ITU or ITU-R. It publishes standards and recommendations relevant to colorimetry in its broadcasting service (television) or BT series. See <http://www.itu.int> for more detail.



**Fig. 10.11.**  
Chromaticity diagram showing 500 nm green (square), equal-energy white (asterisk), a feasible green (star) and a displayable green (circle). The locus of different saturated greens is shown as a green line

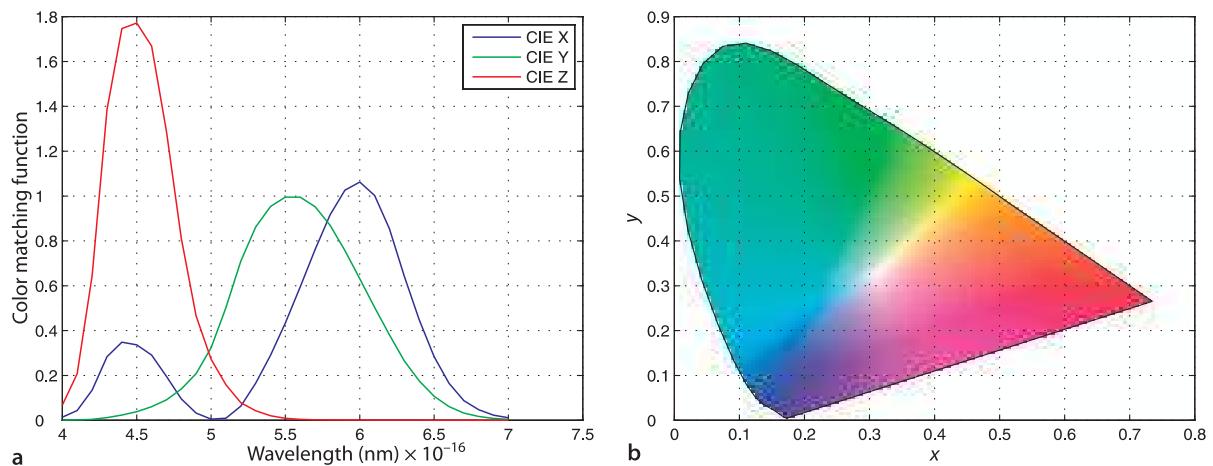
The centre of gravity law states that a mixture of two colors lies along a line between those two colors on the chromaticity plane. A mixture of  $N$  colors lies within a region bounded by those colors. Considered with respect to Fig. 10.10 this has significant implications. Firstly, since all color stimuli are combinations of spectral stimuli all real color stimuli must lie on or inside the spectral locus. Secondly, any colors we create from mixing the primaries can only lie *within* the triangle bounded by the primaries – the color gamut. It is clear from Fig. 10.10 that the CIE primaries define only a small subset of all possible colors – shown as a dashed triangle. Very many real colors *cannot* be created using these primaries, in particular the colors of the rainbow which lie on the spectral locus from 460–545 nm. In fact no matter where the primaries are located, not all possible colors can be produced.► In geometric terms there are no three points within the gamut that form a triangle that includes the entire gamut. Thirdly, we observe that much of the locus requires a negative amount of the red primary and cannot be represented.

The problem on page 232 with displaying 500 nm green is explained by it lying outside the gamut of the CIE primaries and this is shown in Fig. 10.11. We plot the chromaticity of the spectral green color

```
>> green_cc = lambda2rg(500e-9);
green_cc =
-0.9733    1.6187
>> plot2(green_cc, 's')
```

as a square marker. White is by definition  $R = G = B = 1$  and its chromaticity

We could increase the gamut by choosing different primaries, perhaps using a different green primary would make the gamut larger, but there is the practical constraint of finding a light source (LED or phosphor) that can efficiently produce that color.



**Fig. 10.12.** a The color matching functions for the standard observer, based on the imaginary primaries X, Y (intensity) and Z are tabulated by the CIE (Commission Internationale de l’Éclairage 1987). b Colors on the  $xy$ -chromaticity plane

The units are chosen such that equal quantities of the primaries are required to match the equal-energy white stimulus.

```
>> white_cc = tristim2cc([1 1 1])
white_cc =
    0.3333    0.3333
>> plot2(white_cc, '*')
```

is plotted as an asterisk. According to the centre of gravity law the mixture of our desired green and white must lie along the indicated line. The chromaticity of the least saturated displayable green lies at the intersection of this line and the gamut boundary and is indicated by a circle.

Earlier we said that there are no three points within the gamut that form a triangle that includes the entire gamut. The CIE therefore proposed, in 1931, a system of *imaginary non-physical primaries* known as X, Y and Z that totally enclose the spectral locus of Fig. 10.10. X and Z have zero luminance – the luminance is contributed entirely by Y. All real colors can thus be matched by positive amounts of these three primaries. ▶ The corresponding tristimulus values are denoted ( $X, Y, Z$ ).

The XYZ color matching functions defined by the CIE

```
>> cmf = cmfxyz(lambda);
>> plot(lambda*1e-9, cmf);
```

are shown graphically in Fig. 10.12a. This shows the amount of each CIE XYZ primary required to match a spectral color and we note that these curves are never negative. The corresponding chromaticity coordinates are

$$x = \frac{X}{X + Y + Z}, y = \frac{Y}{X + Y + Z}, z = \frac{Z}{X + Y + Z} \quad (10.9)$$

and once again  $x + y + z = 1$  so only two parameters are required – by convention  $y$  is plotted against  $x$  in a chromaticity diagram. The spectral locus can be plotted in a similar way as before

```
>> [x,y] = lambda2xy(lambda);
>> plot(x, y);
```

A more sophisticated plot, showing the colors within the spectral locus, can be created

```
>> xycolorspace
```

Note that the colors depicted are only approximation of the actual color at that point due to the gamut limitation of the printed colors. No display device has a gamut large enough to present an accurate representation of the chromaticity at every point.

and is shown ▶ in Fig. 10.12b. These coordinates are a *standard* way to represent color for graphics, printing and other purposes. For example the chromaticity coordinates of peak green (550 nm) is

```
>> lambda2xy(550e-9)
ans =
    0.3016    0.6924
```

and the chromaticity coordinates of a standard tungsten illuminant at 2 600 K is

```
>> lamp = blackbody(lambda, 2600);
>> lambda2xy(lambda, lamp)
ans =
    0.4679      0.4126
```

### 10.2.3 Color Names

Chromaticity coordinates provide a quantitative way to describe and compare colors, however humans refer to colors by name. Many computer operating systems contain a database or file<sup>▶</sup> that maps human understood names of colors to their corresponding ( $R, G, B$ ) tristimulus values. A typical database contains more than 800 uniquely named colors which says something about the importance of color to humans. The Toolbox provides a copy of such a file and an interface function `colorname`. For example we can query a color name that includes a particular substring

```
>> colorname('burnt')
ans =
    'burntsienna'      'burntumber'
```

The RGB tristimulus value of burntsienna is

```
>> colorname('burntsienna')
ans =
    0.5412      0.2118      0.0588
```

with the values normalized to the interval [0, 1]. We could also request  $xy$ -chromaticity coordinates

```
>> bs = colorname('burntsienna', 'xy')
bs =
    0.5258      0.3840
```

With reference to Fig. 10.12 we see that this point is in the red-brown part of the colorspace and not too far from the color of chocolate

```
>> colorname('chocolate', 'xy')
ans =
    0.5092      0.4026
```

We can also solve the inverse problem. For example consider a tristimulus value close to, but not exactly the same as, burnt Sienna

```
>> colorname([0.54 0.20 0.06])
ans =
    burntsienna
```

and the name of the closest color, in Euclidean terms, is returned. We can repeat this with the color specified in  $xy$ -chromaticity coordinates

```
>> colorname(bs, 'xy')
ans =
    'burntsienna'
```

The file is named `/etc/rgb.txt` on most Unix-based systems.

### 10.2.4 Other Color Spaces

A color space is a 3-dimensional space that contains all possible tristimulus values – all colors and all levels of brightness. If we think of this in terms of coordinate frames as discussed in Sect. 2.2 then there are an infinite number of choices of Cartesian frame with which to define colors. We have already discussed two different Cartesian color spaces: RGB and XYZ. However we could also use polar, spherical or hybrid coordinate systems.

The chromaticity spaces  $r\text{-}g$  or  $x\text{-}y$  do not account for brightness – we normalized it out in Eq. 10.8 and Eq. 10.9. Brightness is more precisely called luminance and is typically denoted by  $Y$ . The definition from ITU Recommendation 709

$$Y^{709} = 0.2126R + 0.7152G + 0.0722B \quad (10.10)$$

is a weighted sum of the RGB-tristimulus values and reflects the eye's high sensitivity to green and low sensitivity to blue. Chromaticity plus luminance leads to 3-dimensional color spaces such as  $r-g-Y$  or  $x-y-Y$ .

Humans seem to more naturally consider chromaticity in terms of two characteristics: hue and saturation. Hue is the dominant color, the closest spectral color, and saturation refers to the purity, or absence of mixed white. Stimuli on the spectral locus are completely saturated while those closer to its centroid are less saturated. The color spaces that we have discussed lack easy interpretation in terms of hue and saturation so alternative color spaces have been proposed. The two most commonly known are HSV and CIE  $L^*C^*h$ . In color-space notation H is hue, S is saturation which is also known as C or chroma. The intensity dimension is named either V for value or L for lightness but they are computed quite differently. The concepts of hue and saturation is illustrated in geometric terms in Fig. 10.13.

The function `colorspace` can be used to perform conversions between different color spaces. For example the hue, saturation and intensity for each of pure red, green and blue RGB tristimulus value is

```
>> colorspace('RGB->HSV', [1, 0, 0])
ans =
    0      1      1
>> colorspace('RGB->HSV', [0, 1, 0])
ans =
   120      1      1
>> colorspace('RGB->HSV', [0, 0, 1])
ans =
   240      1      1
```

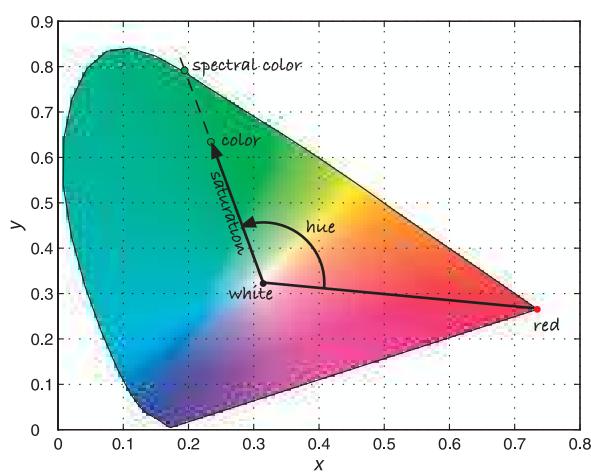
In each case the saturation is 1, the colors are pure, and the intensity is 1. As shown in Fig. 10.13 hue is represented as an angle in the range  $[0, 360]^\circ$  with red at  $0^\circ$  increasing through the spectral colors associated with decreasing wavelength (orange, yellow, green, blue, violet). If we reduce the amount of the green primary

```
>> colorspace('RGB->HSV', [0, 0.5, 0])
ans =
  120.0000  1.0000  0.5000
```

we see that intensity drops but hue and saturation are unchanged. For a medium grey

```
>> colorspace('RGB->HSV', [0.4, 0.4, 0.4])
ans =
  240.0000  0  0.4000
```

the saturation is zero, it is only a mixture of white, and the hue has no meaning since there is no color. If we add the green to the grey

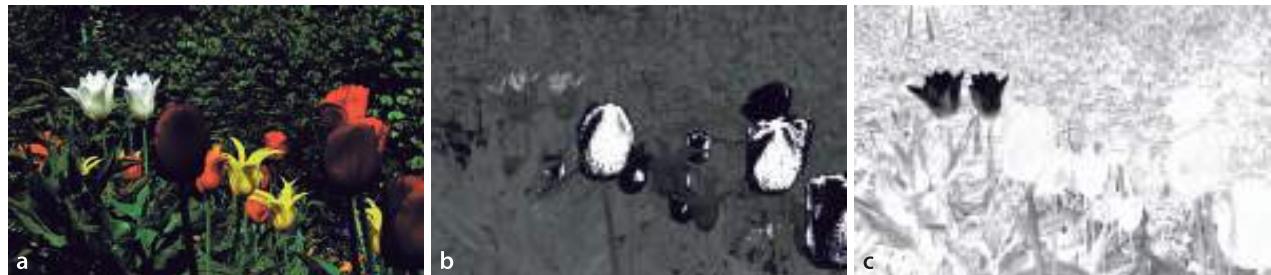


**Fig. 10.13.**

Hue and saturation. A line is extended from the white point through the chromaticity in question to the spectral locus. The angle of this line is hue, and saturation is the length of the vector normalized with respect to distance to the locus

$L^*$  is a non-linear function of relative luminance and approximates the non-linear response of the human eye. Value is given by  $V = \frac{1}{2}(\min R, G, B + \max R, G, B)$ .

For very dark colors numerical problems lead to imprecise hue and saturation coordinates.



```
>> colorspace('RGB->HSV', [0, 0.5, 0] + [0.4, 0.4, 0.4])
ans =
    120.0000    0.5556    0.9000
```

we have the green hue and a medium saturation value.

The `colorspace` function can convert between thirteen different color spaces including YUV, YCbCr, L<sup>\*</sup>a<sup>\*</sup>b<sup>\*</sup> and L<sup>\*</sup>u<sup>\*</sup>v<sup>\*</sup>. A limitation of many color spaces is that the *perceived* color difference between two closely spaced points depends on the position of those points in the space. This has led to the development of perceptually uniform color spaces such as the CIE L<sup>\*</sup>u<sup>\*</sup>v<sup>\*</sup> (CIELUV) and L<sup>\*</sup>a<sup>\*</sup>b<sup>\*</sup> spaces.

The `colorspace` function can also be applied to a color image as shown in Fig. 10.14. In the hue image dark represents red and bright white represents violet. The red flowers appear as both a very small hue angle (dark) and a very large angle close to 360°. The yellow flowers and the green background can be seen as distinct hue values. The saturation image shows that the red and yellow flowers are highly saturated, while the green leaves and stems are less saturated. The white flowers have very low saturation, since by definition the color white contains a lot of white. This example is explained in more detail, and extended, in Sect. 10.3.5.

### 10.2.5 Transforming between Different Primaries

The CIE standards were defined in 1931 which was well before the introduction of color television in the 1950s. The CIE primaries are based on the emission lines of a mercury lamp which are highly repeatable and suitable for laboratory use. Early television receivers used CRT monitors where the primary colors were generated by phosphors that emit light when bombarded by electrons. The phosphors used, and their colors has varied over the years in pursuit of brighter displays. An international agreement, ITU recommendation 709, defines the primaries for high definition television (HDTV) and these are listed in Table 10.2.

This raises the problem of converting tristimulus values from one sets of primaries to another. Consider for example that we wish to display an image, where the tristimulus values are with respect to CIE primaries, on a screen that uses ITU Rec. 709 primaries. Using the notation we introduced earlier we define two sets of primaries: P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub> with tristimulus values (S<sub>1</sub>, S<sub>2</sub>, S<sub>3</sub>), and P'<sub>1</sub>, P'<sub>2</sub>, P'<sub>3</sub> with tristimulus values (S'<sub>1</sub>, S'<sub>2</sub>, S'<sub>3</sub>). We can always express one set of primaries as a linear combination► of the other

$$\begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{pmatrix} \begin{pmatrix} \mathbf{P}'_1 \\ \mathbf{P}'_2 \\ \mathbf{P}'_3 \end{pmatrix} \quad (10.11)$$

and since the two tristimuli match then

$$(S'_1 \quad S'_2 \quad S'_3) \begin{pmatrix} \mathbf{P}'_1 \\ \mathbf{P}'_2 \\ \mathbf{P}'_3 \end{pmatrix} \equiv (S_1 \quad S_2 \quad S_3) \begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \end{pmatrix} \quad (10.12)$$

**Fig. 10.14.** Flower scene. **a** Original color image; **b** hue image; **c** saturation image. Note that the white flowers have low saturation (they appear dark)

The coefficients can be negative so the new primaries do not have to lie within the gamut of the old primaries.

**Table 10.2.**  
 $xyz$ -chromaticity of standard primaries and whites. The CIE primaries of Table 10.1 and the more recent ITU recommendation 709 primaries defined for HDTV.  $D_{65}$  is the white of a blackbody radiator at 6500 K, and  $E$  is equal-energy white

	$R_{\text{CIE}}$	$G_{\text{CIE}}$	$B_{\text{CIE}}$	$R_{709}$	$G_{709}$	$B_{709}$	$D_{65}$	$E$
$x$	0.7347	0.2738	0.1666	0.640	0.300	0.150	0.3127	0.3333
$y$	0.2653	0.7174	0.0089	0.330	0.600	0.060	0.3290	0.3333
$z$	0.0000	0.0088	0.8245	0.030	0.100	0.790	0.3582	0.3333

Substituting Eq. 10.11, equating tristimulus values and then transposing we obtain

$$\begin{pmatrix} S'_1 \\ S'_2 \\ S'_3 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{pmatrix}^T \begin{pmatrix} S_1 \\ S_2 \\ S_3 \end{pmatrix} = C \begin{pmatrix} S_1 \\ S_2 \\ S_3 \end{pmatrix} \quad (10.13)$$

which is simply a linear transformation of tristimulus values.

Consider the concrete problem of transforming from CIE primaries to XYZ tristimulus values. We know from Table 10.2 the CIE primaries in terms of XYZ primaries

```
>> C = [ 0.7347, 0.2653, 0; 0.2738, 0.7174, 0.0089; 0.1666,
          0.0089, 0.8245]'
```

$$C = \begin{matrix} 0.7347 & 0.2738 & 0.1666 \\ 0.2653 & 0.7174 & 0.0089 \\ 0 & 0.0088 & 0.8245 \end{matrix}$$

which is exactly the first three columns of Table 10.2. The transform is therefore

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = C \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Recall from page 235 that luminance is contributed entirely by the Y primary. It is common to apply the constraint that unity  $R, G, B$  values result in unity luminance  $Y$  and a white with a specified chromaticity. We will choose  $D_{65}$  white whose chromaticity is given in Table 10.2 and which we will denote  $(x_w, y_w, z_w)$ . We can now write

$$\frac{1}{y_w} \begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = C \begin{pmatrix} J_R & 0 & 0 \\ 0 & J_G & 0 \\ 0 & 0 & J_B \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

where the left-hand side has  $Y=1$  and we have introduced a diagonal matrix  $J$  which scales the luminance of the primaries. We can solve for the elements of  $J$

$$\begin{pmatrix} J_R \\ J_G \\ J_B \end{pmatrix} = C^{-1} \begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} \frac{1}{y_w}$$

Substituting real values we obtain

```
>> J = inv(C) * [0.3127 0.3290 0.3582]' * (1/0.3290)
J =
    0.5609
    1.1703
    1.3080
>> C * diag(J)
ans =
    0.4121    0.3204    0.2179
    0.1488    0.8395    0.0116
        0    0.0103    1.0785
```

The middle row of this matrix leads to the luminance relationship

$$Y = 0.1488R + 0.8395G + 0.0116B$$

which is similar to Eq. 10.10. The small variation is due to the different primaries used – CIE in this case versus Rec. 709 for Eq. 10.10.

The *RGB* tristimulus value of the redbrick was computed earlier and we can determine its *XYZ* tristimulus

```
>> XYZ_brick = C * diag(J) * RGB_brick';
ans =
0.0351
0.0224
0.0039
```

which we convert to chromaticity coordinates by Eq. 10.9

```
>> tristim2cc(XYZ_brick')
ans =
0.5729    0.3645
```

Referring to Fig. 10.12 we see that this *xy*-chromaticity lies in the red region and is named

```
>> colorname([0.5729    0.3645], 'xy')
ans =
'englishred'
```

as might be expected for a “weathered red brick”.

### 10.2.6 What Is White?

In the previous section we touched on the subject of white. White is both the absence of color and also the sum of all colors. One definition of white is *standard daylight* which is taken as the mid-day Sun in Western/Northern Europe which has been tabulated by the CIE as illuminant  $D_{65}$ . It can be closely approximated by a blackbody radiator at 6 500 K

```
>> d65 = blackbody(lambda, 6500);
>> lambda2xy(lambda, d65)
ans =
0.3136    0.3241
```

which we see is close to the  $D_{65}$  chromaticity given in Table 10.2.

Another definition is based on white light being an equal mixture of all spectral colors. This is represented by a uniform spectrum

```
>> ee = ones(size(lambda));
```

which is also known as the equal-energy stimulus and has chromaticity

```
>> lambda2xy(lambda, ee)
ans =
0.3333    0.3338
```

which is close to the defined value of  $(\frac{1}{3}, \frac{1}{3})$ .

### 10.3 Advanced Topics

In this section we will cover some advanced topics. The first is the effect of illumination on the apparent color of an object which is a very real problem for a robot using color cues in an environment with natural lighting. This leads to a discussion of the problem of white balancing. The next topic is an introduction to gamma encoding which is a very common non-linear relationship between tristimulus values and actual luminance. Finally we look at the distribution of colors in a real image and segment the image into regions of similar colors – this is a preview of techniques that we will cover in Chap. 12 and 13.

### 10.3.1 Color Constancy

We adapt our perception of color so that the integral, or average, over the entire scene is grey. This works well over a color temperature range 5 000–6 500 K.

Studies show that human perception of what is white is adaptive and has a remarkable ability to *tune out* the effect of scene illumination so that white objects always appear to be white. ▶ For example at night under a yellowish tungsten lamp the pages of a book still appear white to us, but a photograph of that scene viewed later under different lighting conditions will look yellow.

All of this poses real problems for a robot that is using color to understand the scene because the observed chromaticity varies with lighting. Outdoors the color of the morning or evening Sun (3 500 K) is different to that of the noon Sun (6 500 K), an overcast day is different to a clear day, and reflections from buildings or trees all conspire to change the illumination spectrum, and hence the luminance and color of the object. To illustrate this problem we revisit the red brick

```
>> lambda = [400:10:700]*1e-9;
>> R = loadspectrum(lambda, 'redbrick.dat');
```

under two different illumination conditions, the Sun at ground level

```
>> sun = loadspectrum(lambda, 'solar.dat');
```

and a tungsten lamp

```
>> lamp = blackbody(lambda, 2600);
```

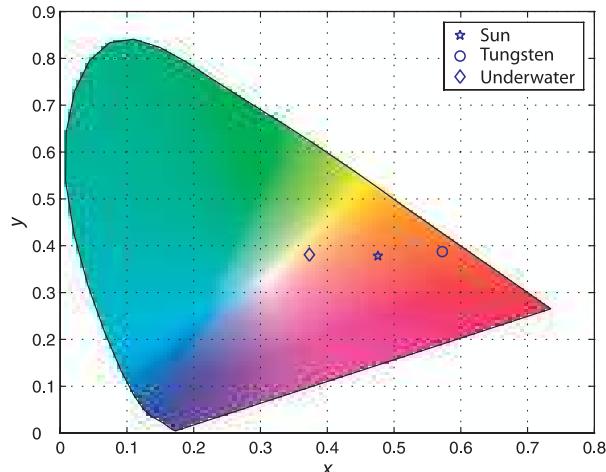
and compute the  $xy$ -chromaticity for each case

```
>> xy_sun = lambda2xy(lambda, sun .* R)
xy_sun =
    0.4760    0.3784
>> xy_lamp = lambda2xy(lambda, lamp .* R)
xy_lamp =
    0.5724    0.3877
```

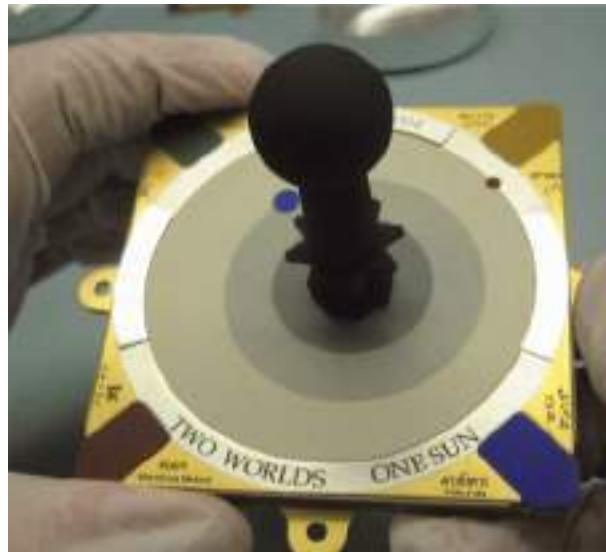
and we can see that the chromaticity, or apparent color, has changed significantly. These values are plotted on the chromaticity diagram in Fig. 10.15.

### 10.3.2 White Balancing

Photographers are well aware of the importance of illumination and refer to the color temperature of a light source – the equivalent black body temperature from Eq. 10.1. Compared to daylight an incandescent lamp appears more yellow, and a photographer



**Fig. 10.15.**  
Chromaticity of a red-brick  
under different illumination  
conditions



**Fig. 10.16.**  
The calibration target used for the Mars Rover's PanCam. Regions of known reflectance and chromaticity (red, yellow, green, blue and shades of grey) are used to set the white balance of the camera. The central stalk has a very low reflectance and also serves as a sundial. In the best traditions of sundials it bears a motto (photo courtesy NASA/JPL/Cornell/Jim Bell)

would use a blue filter (on the camera) to attenuate the red part of the spectrum to compensate. We can achieve a similar function by choosing the matrix  $J$

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} = \begin{pmatrix} J_R & 0 & 0 \\ 0 & J_G & 0 \\ 0 & 0 & J_B \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

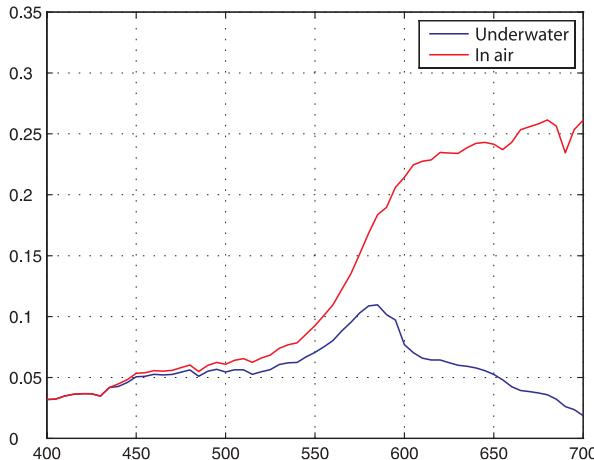
to adjust the gains of the color channels. For example, boosting  $J_B$  would compensate for the lack of blue under tungsten illumination. This is the process of white balancing – ensuring the appropriate chromaticity of objects that we know are white (or grey).

Some cameras allow the user to set the color temperature of the illumination through a menu, typically with options for tungsten, fluorescent, daylight and flash which select different preset values of  $J$ . In manual white balancing the camera is pointed at a grey or white object and a button is pressed. The camera adjusts its channel gains  $J$  so that equal tristimulus values are produced  $R' = G' = B'$  which as we recall results in the desired white chromaticity. For colors other than white these corrections introduces some color error but this nevertheless has a satisfactory appearance to the eye. Automatic white balancing is commonly used and involves heuristics to estimate the color temperature of the light source but it can be fooled by scenes with a predominance of a particular color.

The most practical solution is to use the tristimulus values of three objects with known chromaticity in the scene. This allows the matrix  $C$  in Eq. 10.13 to be estimated directly, mapping the tristimulus values from the sensor to XYZ coordinates which are an absolute lighting-independent representation of surface reflectance. From this the chromaticity of the illumination can also be estimated. This approach is used for the panoramic camera on the Mars Rover where the calibration target shown in Fig. 10.16 can be imaged periodically to update the white balance under changing Martian illumination.

### 10.3.3 Color Change Due to Absorption

A final and extreme example of problems with color occurs underwater. For example consider a robot trying to find a docking station identified by colored targets. As discussed earlier in Sect. 10.1.1 water acts as a filter that absorbs more red light than blue light. For an object underwater this filtering affects both the illumination falling on

**Fig. 10.17.**

Spectrum of the red brick when viewed underwater. The spectrum without the water absorption is shown in red

the object and the reflected light, the luminance, on its way to the camera. Consider again the red brick

```
>> [R,lambda] = loadspectrum([400:5:700]*1e-9, 'redbrick.dat');
```

which is now 1 m underwater and with a camera a further 1 m from the brick. The illumination on the water's surface is that of sunlight at ground level

```
>> sun = loadspectrum(lambda, 'solar.dat');
```

The absorption spectrum of water is

```
>> A = loadspectrum(lambda, 'water.dat');
```

and the total path length through the water is

```
>> d = 2
```

The transmission  $T$  is given by Beer's law Eq. 10.2.

```
>> T = 10.^(-d*A);
```

and the resulting luminance of the brick is

```
>> L = sun .* R .* T;
```

which is shown in Fig. 10.17. We see that the longer wavelengths, the reds, have been strongly attenuated. The apparent color of the brick is

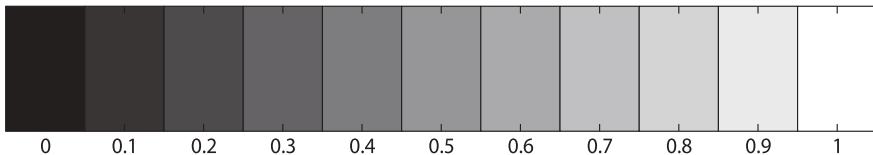
```
>> xy_water = lambda2xy(lambda, L)
xy_water =
    0.3722    0.3813    0.2465
```

which is also plotted in the chromaticity diagram of Fig. 10.15. The brick appears much more blue than it did before. The reality underwater is more complex than this due to the scattering of light by tiny suspended particles. These result in additional color filtering of light on its way to the camera. They also reflect ambient light into the camera that has not been reflected from the target.

#### 10.3.4 Gamma

In an old fashioned CRT monitor the luminance produced at the face of the display is non-linearly related to the control voltage  $V$  according to

$$L = V^\gamma \quad (10.14)$$



**Fig. 10.18.**  
The linear intensity wedge

where  $\gamma \approx 2.2$ . To correct for this non-linearity cameras generally apply the inverse non-linearity  $V = L^{1/\gamma}$  to their output signal which results in a system that is linear from end to end.►

Both operations are commonly referred to as gamma correction though more properly the camera-end operation is gamma encoding and the display-end operation is gamma decoding.► LCD displays have a stronger non-linearity than CRTs but correction tables are applied within the display to make it follow the standard  $\gamma = 2.2$  behavior.►

To show the effect of display gamma we create a simple test pattern

```
>> wedge = [0:0.1:1];
>> idisp(wedge)
```

that is shown in Fig. 10.18 and is like a photographer's *greyscale step wedge*. If we display this on our computer screen it will appear differently to the one printed in the book. We will most likely observe a large change in brightness between the second and third block – the effect of the gamma decoding non-linearity Eq. 10.14 in the display of your computer.

If we apply gamma encoding

```
>> idisp( wedge .^ (1/2.2) )
```

we observe that the intensity changes appear to be more linear► and closer to the one printed in the book.

The chromaticity coordinates of Eq. 10.8 and Eq. 10.9 are computed as ratios of tristimulus values which are linearly related to luminance in the scene. The non-linearity applied to the camera output must be corrected, gamma decoded, before any colometric operations. The Toolbox function `igamma` performs this operation. Gamma decoding can also be performed when an image is loaded using the '`gamma`' option to the function `iread`.

Today most digital cameras► encode images in sRGB format which uses the ITU Rec. 709 primaries and a gamma encoding function of

$$E' = \begin{cases} 12.92L, & L \leq 0.0031308 \\ 1.055L^{1/2.4} - 0.055, & L > 0.0031308 \end{cases}$$

which comprise a linear function for small values and a power law for larger values. The overall gamma is approximately 2.2.

The important property of colorspaces such as *HSV* or *xyY* is that the chromaticity coordinates are invariant to changes in intensity. Many digital video cameras provide output in *YUV* or  $YC_B C_R$  format which has a luminance component *Y* and two other components which are often mistaken for chromaticity coordinates – they are not. They are in fact color difference signals such that  $U, C_B \propto B' - Y'$  and  $V, C_R \propto R' - Y'$  where  $R', B'$  are gamma encoded tristimulus values, and  $Y'$  is gamma encoded intensity. The gamma nonlinearity means that *UV* or  $C_B C_R$  will not be a constant as overall lighting level changes.

The tristimulus values from the camera must be first converted to linear tristimulus values, by applying the appropriate gamma decoding, and then computing chromaticity. There is no shortcut.

Many cameras have an option to choose gamma as either 1 or 0.45 (=1 / 2.2).

Gamma encoding and decoding are often referred to as gamma compression and gamma decompression respectively, since the encoding operation compresses the range of the signal, while decoding decompresses it.

Macintosh computers are an exception and prior to MacOS 10.6 used  $\gamma = 1.8$  which tends to make colors appear brighter and more vivid.

For a Macintosh prior to MacOS 10.6 use 1.8 instead of 2.2.

The JPEG file header (JFIF file format) has a tag `Color Space` which is set to either `sRGB` or `Uncalibrated` if the gamma or color model is not known. See page 289.

### 10.3.5 Application: Color Image

In this section we bring together many of the concepts and tools introduced in this chapter. We will also preview a number of functions that will be properly introduced in the next chapter. We consider a garden scene

```
>> flowers = iread('flowers4.png', 'double', 'gamma', 'sRGB');
```

shown in Fig. 10.19a comprising three different colored flowers and background greenery. Importantly we have applied gamma decoding so that the tristimulus values are proportional to the luminance of the original scene. The image `flowers` has 3 dimensions as shown in Fig. 12.2. The first two dimensions are the vertical and horizontal pixel coordinate, and the third is the color plane that selects the red, green or blue pixels.

We can convert the image to hue, saturation and value

```
>> hsv = colorspace('RGB->HSV', flowers);
```

and the result is another 3-dimensional matrix but this time the color planes represent hue, saturation and value. We can display hue

```
>> idisp( hsv(:,:,1) )
```

and saturation

```
>> idisp( hsv(:,:,2) )
```

as images which are shown in Fig. 10.14b and c respectively.

If we plot the chromaticity of each pixel as points in the chromaticity plane we would observe clusters of points corresponding to different parts of the scenes such as red flowers, yellow flowers, green leaves and shadows. We convert the color RGB image to an XYZ image

```
>> XYZ = colorspace('RGB->XYZ', flowers);
```

and then to *xy*-chromaticity coordinates

```
>> [x,y] = tristim2cc(XYZ);
```

where `x` and `y` are each images the same size as `flowers`. Next we compute a 2-dimensional histogram of  $(x, y)$  values with 100 bins in each dimension

```
>> xbins = [0 0.01 100]; ybins = [0 0.01 100];
>> [h,vx,vy] = hist2d(x, y, xbins, ybins);
```

where `h` is the number of points in each bin and `vx` and `vy` are the  $x$ - and  $y$ -coordinates of the corresponding bins. We display the histogram as a contour map overlaid on the *xy*-chromaticity diagram

```
>> xycolorspace
>> hold on
>> contour(vx, vy, h)
```

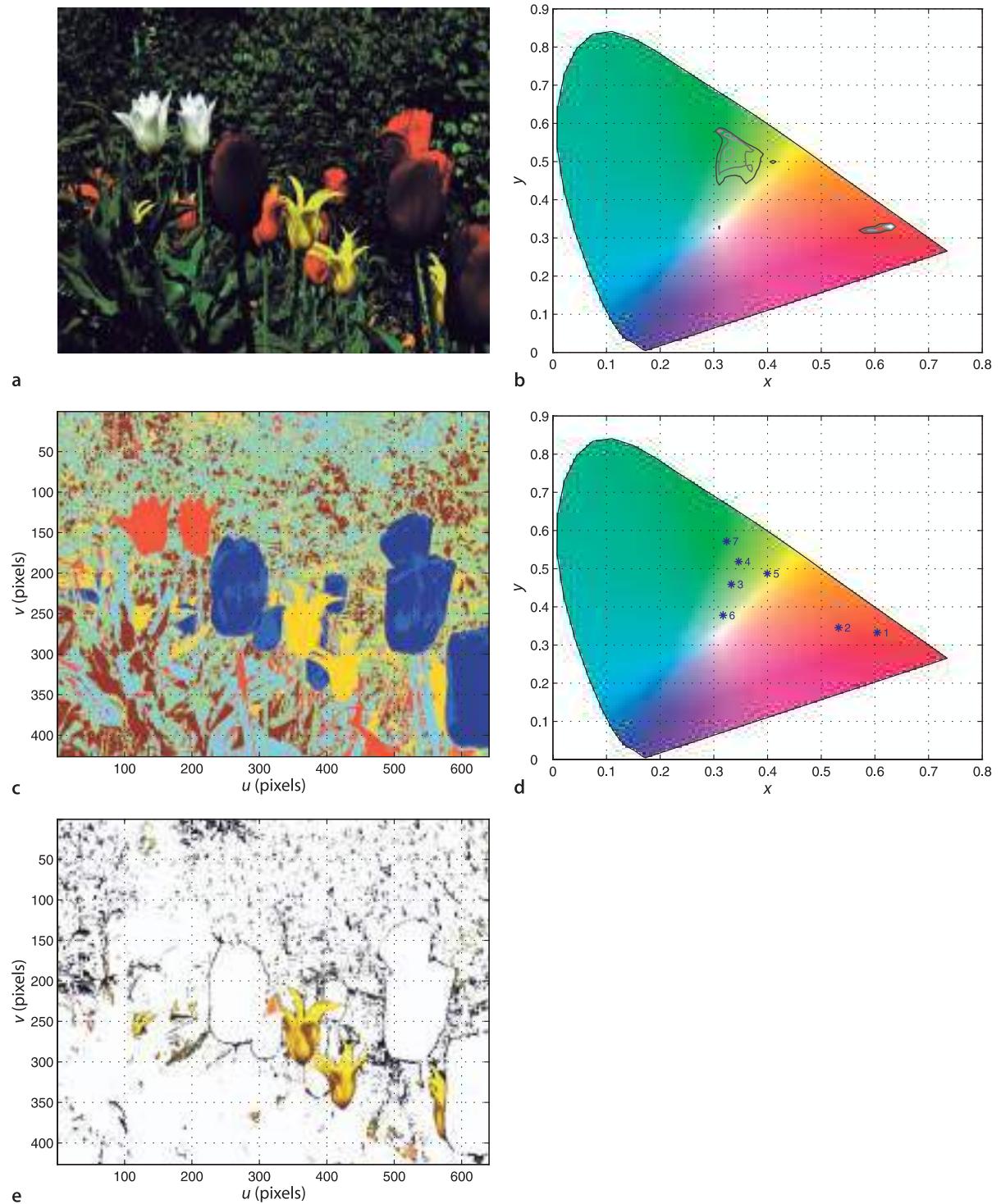
as shown in Fig. 10.19b. We see that there are 5 or 6 peaks: a broad peak in the red area, a narrow white peak and a number peaks in the green area.

Next we will perform unsupervised clustering using the k-means algorithm

```
>> [cls, cxy] = colorseg(flowers, 7);
```

where the second argument specifies the number of pixel chromaticity classes or clusters which we have set to seven. The k-means algorithm is iterative, it adjusts its estimate of the centre of each cluster and the assignment of pixels to clusters until equilibrium is reached. The initial cluster centres are chosen randomly which means that the function will give a different result each time it is run.

A limitation of k-means is that the number of clusters must be known in advance, typically guided by domain knowledge.



**Fig. 10.19.** Color image segmentation. **a** Original color image; **b**  $xy$ -chromaticity plane with overlaid frequency contours; **c** label image in false color; **d** cluster centroids on chromaticity diagram; **e** class 5 pixels (yellow) only

The result is another image `cls` where each pixel value indicates the cluster to which the corresponding pixel in `flowers` has been assigned – its color classification, or class, which is an integer in the interval 1 to 7. We can display this

```
>> idisp(cls, 'colormap', 'jet', 'nogui')
```

as shown in Fig. 10.19c. In this case class 5 corresponds to yellow flowers and we can display just those pixels

```
>> idisplabel(flowers, cls, 5)
```

as shown in Fig. 10.19e. In addition to the yellow flowers there are many very small groups of dark pixels that correspond to parts of the foliage – these have the same chromaticity as the flowers but a low luminance.

The `colorseg` function also returns the centre of the clusters and these are

```
cxy =
0.6082    0.3327
0.5378    0.3436
0.3328    0.4599
0.3466    0.5195
0.4017    0.4874
0.3176    0.3788
0.3238    0.5744
```

in *xy*-space. We plot these on the *xy*-chromaticity diagram

```
>> xycolorspace
>> plot_point(cxy, '*', 'sequence', 'textsize', 10, 'textcolor', 'b')
```

as shown in Fig. 10.19d and we can see that class 5 does indeed lie in the yellow area, and also that classes 1 and 2 are red, class 6 is white, while the rest are different greens.

Finally we convert these cluster centres from *xy*-coordinates to human meaningful names

```
>> colorname(cxy, 'xy')
ans =
Columns 1 through 5
'cadmiumreddeep'    'brown3'      'olive'       'terreverte'   'yellow4'
Columns 6 through 7
'darkseagreen4'     'yellowgreen'
```

The color names for class 1 “cadmiumreddeep” and class 4 “yellow4” corresponding to the red and yellow flowers respectively are quite apt. The color name for the white flowers “darkseagreen4” is surprising, implying dark-green rather than white, but the *xy*-chromaticity of this color is quite close to the white point. Dark-green has low luminance but the color name matching is based on chromaticity not luminance.

## 10.4 Wrapping Up

We have learnt that the light we observe is a mixture of frequencies, a continuous spectrum, which is modified by reflectance and absorption. The spectrum elicits a response from the eye which we interpret as color – for humans the response is a tristimulus, a 3-vector that represents the outputs of the three different types of cones in our eye. A digital color camera is functionally equivalent. The tristimulus can be considered as a 1-dimensional brightness coordinate and a 2-dimensional chromaticity coordinate which allows colors to be plotted on a plane. The spectral colors form a locus on this plane and all real colors lie within this locus. The three primary colors form a triangle on this plane which is the gamut of those primaries. Any color within the triangle can be matched by an appropriate mixture of the primaries. No set of primaries can define a gamut that contains all colors. An alternative set of

imaginary primaries, the CIE XYZ system, does contain all real colors and is the standard way to describe colors. Tristimulus values can be transformed using linear transformations to account for different sets of primaries. Non-linear transformations can be used to describe tristimulus values in terms of human-centric qualities such as hue and saturation.

We also discussed definition of white, the problem of white balancing, the non-linear response of display devices and how this effects the common representation of images and video. Finally we used chromaticity information to classify pixels in a colorful real-world image.

### Further Reading

At face value color is a simple concept that we learn in kindergarten but it is actually a complex topic. In this chapter we have only begun to scrape the surface of photometry and colorimetry. Photometry is the part of the science of radiometry concerned with measurement of visible light. It is challenging for engineers and computer scientists since it makes use of uncommon units such as lumen, steradian, nit, candela and lux. One source of complexity is that words like intensity and brightness are synonyms in everyday speech but have very specific meanings in photometry. Colorimetry is the science of color perception and is also a large and complex area since human perception of color depends on the individual observer, ambient illumination and even the field of view. Colorimetry is however critically important in the design of cameras, computer displays, video equipment and printers.

The computer vision textbooks by Gonzalez and Woods (2008) and Forsyth and Ponce (2002) each have a discussion on color and color spaces. The latter also has a discussion on the effects of shading and inter-reflections. Comprehensive online information about computer vision is available through CVonline at <http://homepages.inf.ed.ac.uk/rbf/CVonline>, and the material in this chapter is covered under the sections *Image Physics* and *Sensors and their Properties*.

Readable and comprehensive books on color science include Hunt (1987) and from a television or engineering perspective Benson (1986). A more conversational approach is given by Hunter and Harold (1987), which also covers other aspects of appearance such as gloss and lustre. The CIE standard (Commission Internationale de l'Éclairage 1987) is definitive but hard reading. The work of the CIE is ongoing and its standards are periodically updated at [www.cie.co.at](http://www.cie.co.at). The color matching functions were first tabulated in 1931 and revised in 1964.

Charles Poynton has for a long time maintained excellent online tutorials about color spaces and gamma at <http://www.poynton.com>. His book (Poynton 2003) is an excellent and readable introduction to these topics while also discussing digital video systems in great depth. Gamma is also described online at <http://www.w3.org/Graphics/Color/sRGB>.

**Infra-red cameras.** Consumer cameras are functionally equivalent to the human eye and are sensitive to the visible spectrum. Cameras are also available that are sensitive to infra-red and a number of infra-red bands are defined by CIE: IR-A (700–1 400 nm), IR-B (1 400–3 000 nm), and IR-C (3 000 nm–1 000 µm). In common usage IR-A and IR-B are known as near infra-red (NIR) and short-wavelength infra-red (SWIR) respectively, and the IR-C subbands are medium-wavelength (MWIR, 3 000–8 000 nm) and long-wavelength (LWIR, 8 000–15 000 nm). LWIR cameras are also called thermal or thermographic cameras. **Ultraviolet cameras** typically work in the near ultra-violet region (NUV, 200–380 nm) and are used in industrial applications such as detecting corona discharge from high-voltage electrical systems.

**Hyperspectral cameras** have more than three classes of photoreceptor, they sample the incoming spectrum at many points typically from infra-red to ultra-violet and with tens or even hundreds of spectral bands. Hyperspectral cameras are used for applications including aerial survey classification of land-use and identification of the mineral composition of rocks.

**Table 10.3.**

Various spectra provided with the Toolbox. Relative luminosity values lie in the interval [0,1], and relative spectral power distribution (SPD) are normalized to a value of 1.0 at 550 nm

Filename	Units	Description
cones.dat	Rel.luminosity	Spectral response of human cones
photopic.dat	Rel.luminosity	CIE 1924 photopic response
scotopic.dat	Rel.luminosity	CIE 1951 scoptic response
redbrick.dat	Reflectivity	Reflectivity spectrum of a weathered red brick
solar.dat	$\text{W m}^{-2} \text{nm}^{-1}$	Solar spectrum at ground level
water.dat	$1 \text{m}^{-1}$	Light absorption spectrum of water
D65.dat	Rel.SPD	CIE standard $D_{65}$ illuminant

Other MATLAB® tools include the ColorLab toolbox at [http://cs.joensuu.fi/colorlab\\_toolbox/](http://cs.joensuu.fi/colorlab_toolbox/) and the `colorspace` function at <http://www.math.ucla.edu/~getreuer/colorspace.html>.

### Data Sources

The Toolbox contains a number of data files describing various spectra which are summarized in Table 10.3. Each file has as its first column the wavelength in metres. The files have different wavelength ranges and intervals but the helper function `loadspectrum` interpolates the data to the user specified range and sample interval.

Several internet sites contain spectral data in tabular format and this is linked from the book's web site. This includes reflectivity data for many materials provided by NASA's online ASTER spectral library and the Spectral Database from the University of Eastern Finland Color Research Laboratory. Data on cone response and CIE color matching functions is available from the Colour & Vision Research Laboratory at University College London. CIE data is also available online.

### Exercises

1. You are a blackbody radiator! Plot your own blackbody emission spectrum. What is your peak emission frequency? What part of the EM spectrum is this? What sort of sensor would you use to detect this?
2. Consider a sensor that measures the amount of radiated power  $P_1$  and  $P_2$  at wavelengths  $\lambda_1$  and  $\lambda_2$  respectively. Write an equation to give the temperature  $T$  of the blackbody in terms of these quantities.
3. Using the Stefan-Boltzman law compute the power emitted per square metre of the Sun's surface. Compute the total power output of the Sun.
4. Use numerical integration to compute the power emitted in the visible band 400–700 nm per square metre of the Sun's surface.
5. Why is the peak luminosity defined as  $683 \text{ lm W}^{-1}$ ?
6. Given typical outdoor illuminance as per page 229 determine the luminous intensity of the Sun.
7. Sunlight at ground level. Of the incoming radiant power determine, in percentage terms, the fraction of infra-red, visible and ultra-violet light.
8. Use numerical integration to compute the power emitted in the visible band 400–700 nm per square metre for a tungsten lamp at 2 600 K. What fraction is this of the total power emitted?
9. Plot and compare the human photopic and scotopic spectral response.
10. Can you create a metamer for the red brick?
11. Prove the center of gravity law.



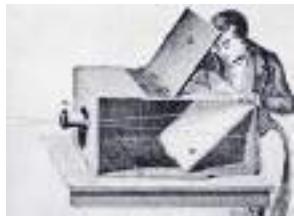
**Fig. 10.20.**  
The Gretag Macbeth Color Checker is an array of 24 printed color squares, which includes different greys and colors as well as spectral simulations of skin, sky, foliage etc. Spectral data for the squares is available online via <http://www.cis.rit.edu/research/mcls/online/cie.php>

12. On the  $xy$ -chromaticity plane plot the locus of a blackbody radiator with temperatures in the range 1 000–10 000 K.
13. Plot the XYZ primaries on the  $rg$ -plane.
14. The Gretag Macbeth Color Checker shown in Fig. 10.20 is an array of 24 printed color squares. Spectral data for the Color Checker is available at <http://www.rmimaging.com/information/colorchecker.html>. Compute and plot the  $xy$ -chromaticity for each square.
15. For Fig. 10.11 determine the chromaticity of the feasible green. Determine the brightest possible tristimulus value assuming that the value of any primary lies in the range [0, 1].
16. Modify the function `xycolorspaces` to generate an  $rg$ -chromaticity plane.
17. Determine the tristimulus values for the red brick using the Rec. 709 primaries.
18. Take a picture of a white object using incandescent illumination. Determine the average RGB tristimulus value and compute the  $xy$ -chromaticity. How far off white is it? Determine the color balance matrix  $J$  to correct the chromaticity. What is the chromaticity of the illumination?
19. What is the name of the color of the red brick when viewed underwater (page 242).
20. Image a target like Fig. 10.16 that has three colored patches of known chromaticity. From their observed chromaticity determine the transform from observed tristimulus values to Rec. 709 primaries. What is the chromaticity of the illumination?
21. Consider an underwater application where a target  $d$  metres below the surface is observed through  $m$  metres of water, and the water surface is illuminated by sunlight. From the observed chromaticity can you determine the true chromaticity of the target? How sensitive is this estimate to incorrect estimates of  $m$  and  $d$ ? If you knew the true chromaticity of the target could you determine its distance?
22. Is it possible that two different colors look the same under a particular lighting condition? Create an example of colors and lighting that would cause this?
23. Use one of your own pictures and repeat the exercise of Sect. 10.3.5. Can you distinguish different objects in the picture?
24. Show analytically or numerically that scaling a tristimulus value has no effect on the chromaticity. What happens if the chromaticity is computed on gamma encoded tristimulus values?

# 11

# Image Formation

*A little perspective,  
like a little humor,  
goes a long way.*  
Allen Klein



In this chapter we discuss how images are formed and captured, the first step in robot and human perception of the world. From images we can deduce the size, shape and position of objects in the world as well as other characteristics such as color and texture.

It has long been known that a simple pin-hole is able to create a perfect inverted image on the wall of a darkened room. Some marine molluscs, for example the Nautilus, have pin-hole camera eyes. All vertebrates have a lens that forms an inverted image on the retina where the light-sensitive cells rod and cone cells, shown previously in Fig. 10.6, are arranged. A digital camera is similar in principle – a glass or plastic lens forms an image on the surface of a semiconductor chip with an array of light sensitive devices to convert light to a digital image.

The process of image formation, in an eye or in a camera, involves a *projection* of the 3-dimensional world onto a 2-dimensional surface. The depth information is lost and we can no longer tell from the image whether it is of a large object in the distance or a smaller closer object. This transformation from 3 to 2 dimensions is known as perspective projection and is discussed in Sect. 11.1. Section 11.2 introduces the topic of camera calibration, the estimation of the parameters of the perspective transformation. In Sect. 11.2.3 we discuss the inverse problem, how to reconstruct 3-dimensional world points given a 2-dimensional image. Section 11.3 introduces alternative types of cameras capable of wide-angle or panoramic imaging.

## 11.1 Perspective Transform

The pin-hole camera produces a very dim image since its radiant power is the scene luminance in units of  $\text{W m}^{-2}$  multiplied by the size of the pin hole. The key to brighter images is to collect light over a larger area using a lens or a curved mirror. A convex lenses can form an image just like a pinhole but the larger diameter of the lens allows more light to pass which leads to much brighter images.

The elementary aspects of image formation with a thin lens are shown in Fig. 11.1. The positive  $z$ -axis is the camera's optical axis. The  $z$ -coordinate of the object and its image are related by the lens law

$$\frac{1}{z_o} + \frac{1}{z_i} = \frac{1}{f} \quad (11.1)$$

where  $z_o$  is the distance to the object,  $z_i$  the distance to the image, and  $f$  is the focal length of the lens. For  $z_o > f$  an inverted image is formed on the image plane at  $z < -f$ . In a camera the image plane is fixed at the surface of the sensor chip so the focus ring of the camera moves the lens along the optical axis so that it is a distance  $z_i$  from the image plane – for an object at infinity  $z_i = f$ . A pin-hole camera does not need focusing – the need to focus is the tradeoff for the increased light-gathering ability of a lens. Our own eye has a single convex lens made from transparent crystallin proteins, while a high-quality camera lens is a compound lenses made of multiple glass or plastic lenses.

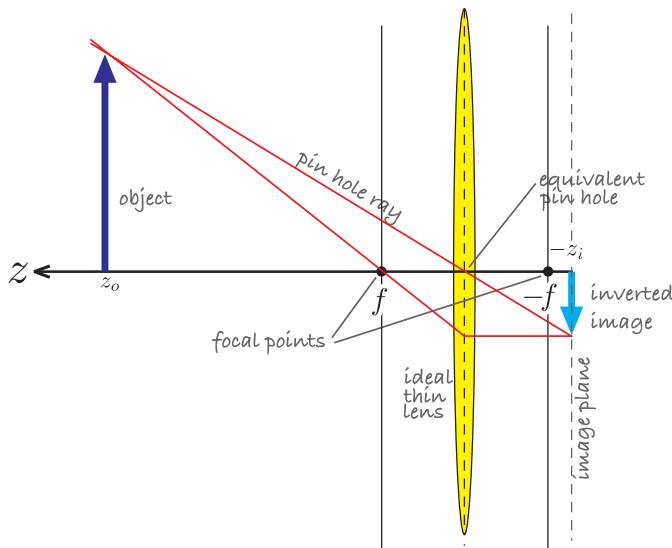
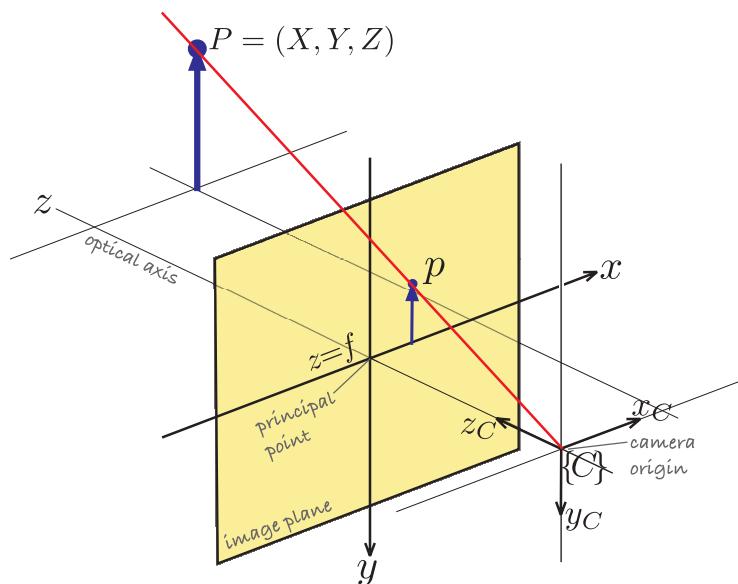
**Fig. 11.1.**

Image formation geometry for a thin convex lens shown in 2-dimensional cross section. A lens has two focal points at a distance of  $f$  on each side of the lens. By convention the camera's optical axis is the  $z$ -axis

**Fig. 11.2.**

The central-projection model. The image plane is  $f$  in front of the camera's origin and on which a non-inverted image is formed. The camera's coordinate frame is right-handed with the  $z$ -axis defining the centre of the field of view

In computer vision it is common to use the central perspective imaging model shown in Fig. 11.2. The rays converge on the origin of the camera frame  $\{C\}$  and a non-inverted image is *projected* onto the image plane located at  $z = f$ . Using similar triangles we can show that a point at the world coordinates  $P = (X, Y, Z)$  is projected to the image plane  $p = (x, y)$  by

$$x = f \frac{X}{Z}, \quad y = f \frac{Y}{Z} \quad (11.2)$$

In the 5<sup>th</sup> century BCE, the philosopher Mo Jing in ancient China mentioned the effect of an inverted image forming through a pinhole. The camera obscura is a darkened room where a dim inverted image of the world is cast on the wall by light entering through a small hole. Making the hole larger increases the brightness of the image but makes it less focussed.

Camera obscuras were popular tourist attractions in Victorian times, particularly in Britain, and many are still operating today. (Image on the right from the Drawing with Optical Instruments collection at <http://vision.mpiwg-berlin.mpg.de/elib>)

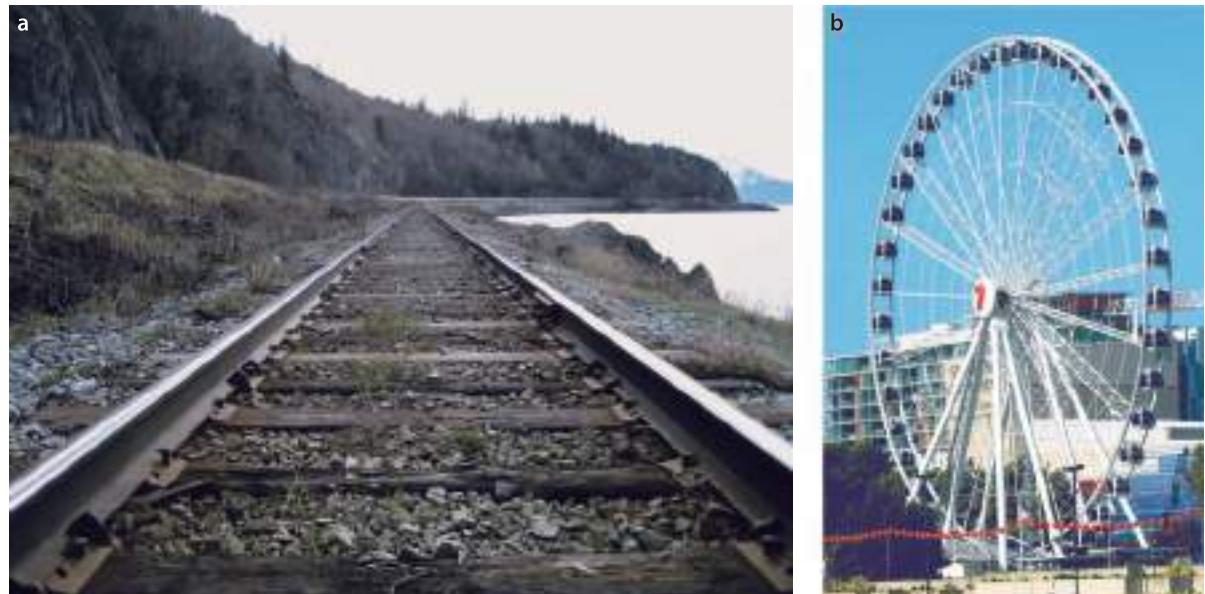


which is a projective transformation, or more specifically a perspective projection, from the world to the image plane and has the following characteristics:

1. It performs a mapping from 3-dimensional space to the 2-dimensional image plane:  $\mathbb{R}^3 \mapsto \mathbb{R}^2$ .
2. Straight lines in the world are projected to straight lines on the image plane.
3. Parallel lines in the world are projected to lines that intersect at a vanishing point as shown in Fig. 11.3a. In drawing, this effect is known as foreshortening. The exception are lines in the plane parallel to the image plane which do not converge.
4. Conics<sup>4</sup> in the world are projected to conics on the image plane. For example, a circle is projected as a circle or an ellipse as shown in Fig. 11.3b.
5. The mapping is not one-to-one and a unique inverse does not exist. That is, given  $(x, y)$  we cannot uniquely determine  $(X, Y, Z)$ . All that can be said is that the world point lies somewhere along the projecting ray OP shown in Fig. 11.2. This is an important topic that we will return to in Chap. 14.
6. The transformation is not conformal – it does not preserve shape since internal angles are not preserved. Translation, rotation and scaling are examples of conformal transformations. A general affine transformation comprises translation, rotation and different scaling for each axis and is not conformal.

We can write the image plane point in homogeneous form  $\tilde{p} = (x', y', z')$  where

**Fig. 11.3.** The effect of perspective transformation. **a** Parallel lines converge, **b** circles become ellipses



**Lens aperture.** The *f-number* of a lens, typically marked on the rim, is a dimensionless quantity  $F = f/d$  where  $d$  the diameter of the lens (often denoted  $\phi$  on the lens rim). The *f-number* is inversely related to the light gathering ability of the lens. To reduce the amount of light falling on the image plane the effective diameter is reduced by a mechanical aperture, or iris, which increases the *f-number*. Illuminance on the image plane is inversely proportional to  $F^2$  since it depends on light gathering area. To reduce illuminance by a factor of 2, the *f-number* must be increased by a factor of  $\sqrt{2}$  or “one stop”. The *f-number* graduations on the aperture ring of a lens increase by  $\sqrt{2}$  at each stop. An *f-number* is conventionally written in the form  $f/1.4$  for  $F = 1.4$ .

**Depth of field.** A pin-hole camera has no focus control and creates a focussed image of objects irrespective of their distance. A lens does not have this property – the focus ring changes the distance between the lens and the image plane and must be adjusted so that the object of interest is clearly focussed. Photographers refer to depth of field which is the range of object distances for which acceptably focussed images are formed. Depth of field is high for small aperture settings where the lens is more like a pin-hole, but this means less light and noisier images or longer exposure time and motion blur. This is the photographer's dilemma!

or in compact matrix form as

$$\tilde{p} = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad (11.3)$$

where the non-homogeneous image plane coordinates are

$$x = \frac{x'}{z'}, \quad y = \frac{y'}{z'}$$

These are often referred to as the retinal image plane coordinates. For the case where  $f=1$  the coordinates are referred to as the normalized or canonical image plane coordinates.

If we write the world coordinate in homogeneous form as well  ${}^C\tilde{P} = (X, Y, Z, 1)^T$  then the perspective projection can be written in *linear* form as

$$\tilde{p} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} {}^C\tilde{P} \quad (11.4)$$

or

$$\tilde{p} = C {}^C\tilde{P} \quad (11.5)$$

where  $C$  is a  $3 \times 4$  matrix known as the camera matrix. Note that we have written  ${}^C\tilde{P}$  to highlight the fact that this is the coordinate of the point with respect to the camera frame  $\{C\}$ . The tilde indicate homogeneous quantities and Appendix I provides a refresher on homogeneous coordinates. The third column of  $C$  is a vector parallel to the camera's optical axis in the world frame. The camera matrix can be factored

$$\tilde{p} = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} {}^C\tilde{P}$$

where the second matrix is the projection matrix.

The Toolbox allows us to create a model of a central-perspective camera. For example

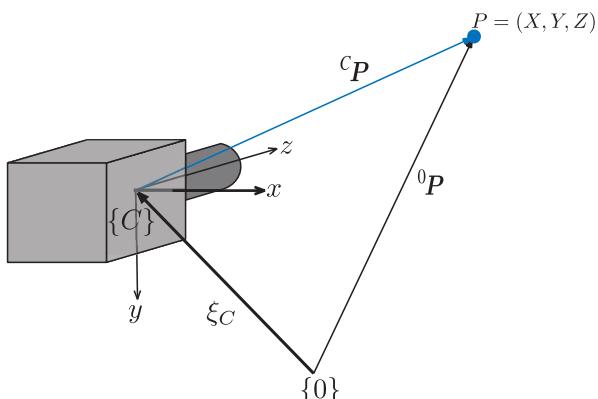
```
>> cam = CentralCamera('focal', 0.015);
```

returns an instance of a `CentralCamera` object with a 15 mm lens. By default the camera is at the origin of the world frame with its optical axis pointing in the world  $z$ -direction as shown in Fig. 11.2. We define a world point

```
>> P = [0.3, 0.4, 3.0]';
```

in units of metres and the corresponding image-plane coordinates are

```
>> cam.project(P)
ans =
    0.0015
    0.0020
```



**Fig. 11.4.**  
Camera coordinate frames

The point on the image plane is at (1.5, 2.0) mm with respect to the principal point. This is a very small displacement but it is commensurate with the size of a typical image sensor.

In general the camera will have an arbitrary pose  $\xi_C$  with respect to the world coordinate frame as shown in Fig. 11.4. The position of the point with respect to the camera is

$$^c\mathbf{P} = (\ominus \xi_C) \cdot {}^0\mathbf{P}$$

or in homogeneous coordinates

$$^c\mathbf{P} = T_C^{-1} {}^0\mathbf{P} \quad (11.6)$$

We can demonstrate this by moving our camera 0.5 m to the *left*

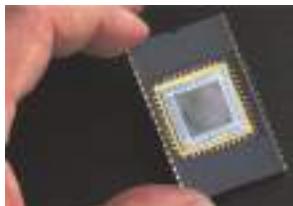
```
>> cam.project(P, 'Tcam', transl(-0.5, 0, 0))
ans =
    0.0040
    0.0020
```

where the third argument is the pose of the camera  $\xi_C$  as a homogeneous transformation. We see that the  $x$ -coordinate has increased from 1.5 mm to 4.0 mm, that is, the image point has moved to the *right*.

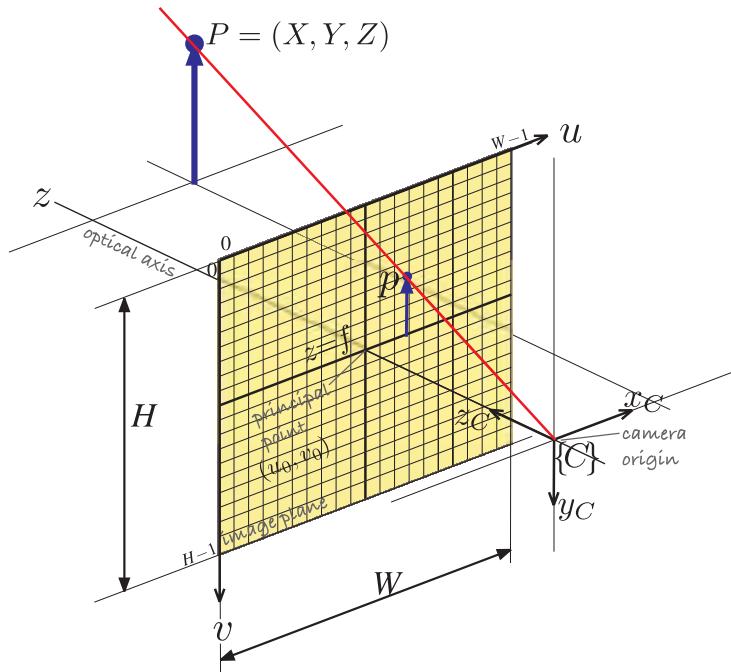
In a digital camera the image plane is a  $W \times H$  grid of light sensitive elements called photosites that correspond directly to the picture elements (or pixels) of the image as shown in Fig. 11.5. The pixel coordinates are a 2-vector  $(u, v)$  of non-negative integers and by convention the origin is at the top-left hand corner of the image plane. In MATLAB® the top-left pixel is (1, 1). The pixels are uniform in size and centred on a regular grid so the pixel coordinate is related to the image plane coordinate by

$$u = \frac{x}{\rho_w} + u_0, \quad v = \frac{y}{\rho_h} + v_0$$

where  $\rho_w$  and  $\rho_h$  are the width and height of each pixel respectively, and  $(u_0, v_0)$  is the principal point – the coordinate of the point where the optical axis intersects the image plane. We can write Eq. 11.4 for pixel coordinates by prepending a camera parameter matrix  $K$



**Image sensor.** The light sensitive cells in a camera chip, the photosites, are commonly square with a side length in the range 1–10 µm. Professional cameras have large photosites for increased light sensitivity whereas cellphone cameras have small sensors and therefore small less-sensitive photosites. The ratio of the number of horizontal to vertical pixels is the aspect ratio and is commonly 4:3 or 16:9 (see page 290). The dimension of the sensor is measured diagonally across the array and is commonly expressed in inches, e.g.  $\frac{1}{3}$ ,  $\frac{1}{4}$  or  $\frac{1}{2}$  inch.



**Fig. 11.5.**  
Central projection model  
showing image plane and  
discrete pixels

$$\tilde{p} = \underbrace{\begin{pmatrix} 1/\rho_w & 0 & u_0 \\ 0 & 1/\rho_h & v_0 \\ 0 & 0 & 1 \end{pmatrix}}_K \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}^c \tilde{P} \quad (11.7)$$

where  $\tilde{p} = (u', v', w')$  is the homogeneous coordinate of the world point  $P$  in pixel coordinates. The non-homogeneous image-plane pixel coordinates are

$$u = \frac{u'}{w'}, v = \frac{v'}{w'} \quad (11.8)$$

For example if the pixels are 10  $\mu\text{m}$  square and the pixel array is 1 280  $\times$  1 024 pixels with its principal point at image plane coordinate (640, 512) then

```
>> cam = CentralCamera('focal', 0.015, 'pixel', 10e-6, ...
    'resolution', [1280 1024], 'centre', [640 512], 'name', 'mycamera')
name: mycamera [central-perspective]
focal length: 0.015
pixel size: (1e-05, 1e-05)
principal pt: (640, 512)
number pixels: 1280 x 1024
Tcam:
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

which displays the parameters of the imaging model. The corresponding non-homogeneous image plane coordinates of the previously defined world point are

```
>> cam.project(P)
ans =
790
712
```

Combining Eq. 11.6 and Eq. 11.7 we can write the camera projection in general form as

The matrix  $K$  is often written with a finite value at  $K[1,2]$  to represent skew. This accounts for the fact that the  $u$ - and  $v$ -axes are not orthogonal, which with precise semiconductor fabrication processes is quite unlikely.

$$\begin{aligned}
\tilde{\mathbf{p}} &= \underbrace{\begin{pmatrix} f/\rho_w & 0 & u_0 \\ 0 & f/\rho_h & v_0 \\ 0 & 0 & 1 \end{pmatrix}}_{\text{intrinsic}} \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}}_{\text{extrinsic}} (\mathbf{^0T}_C)^{-1} \tilde{\mathbf{P}} \\
&= \mathbf{K} \mathbf{P}_0 \mathbf{^0T}_C^{-1} \tilde{\mathbf{P}} \\
&= \mathbf{C} \tilde{\mathbf{P}}
\end{aligned} \tag{11.9}$$

where all the terms are rolled up into the camera matrix  $\mathbf{C}$ . This is a  $3 \times 4$  homogeneous transformation which performs scaling, translation and perspective projection. It is often also referred to as the projection matrix or the camera calibration matrix.

We have already mentioned the fundamental ambiguity with perspective projection, that we cannot distinguish between a large distant object and a smaller closer object. We can rewrite Eq. 11.9 as

$$\tilde{\mathbf{p}} = (\mathbf{CH}^{-1})(\mathbf{H}\tilde{\mathbf{P}}) = \mathbf{C}'\tilde{\mathbf{P}}'$$

where  $\mathbf{H}$  is an arbitrary non-singular  $3 \times 3$  matrix. This implies that an infinite number of camera  $\mathbf{C}'$  and world point  $\tilde{\mathbf{P}}'$  combinations will result in the same image plane projection  $\tilde{\mathbf{p}}$ .

This illustrates the essential difficulty in determining 3-dimensional world coordinates from 2-dimensional projected coordinates. It can only be solved if we have information about the camera or the 3-dimensional object.

The projection can also be written in functional form as

$$\mathbf{p} = \mathcal{P}(\mathbf{P}, \mathbf{K}, \xi_C) \tag{11.10}$$

where  $\mathbf{P}$  is the point in the world frame,  $\mathbf{K}$  is the camera parameter matrix and  $\xi_C$  is the pose of the camera.

The intrinsic parameters are innate characteristics of the camera and sensor and comprise  $f, \rho_w, \rho_h, u_0$  and  $v_0$ . The extrinsic parameters describe the camera's pose and comprise a minimum of six parameters to describe translation and orientation in  $SE(3)$ . There are therefore a total of 11 parameters. The camera matrix has 12 elements so one degree of freedom, the overall scale factor, is unconstrained. In practice these camera parameters are not known and must be estimated using a camera calibration procedure which we will discuss in Sect. 11.2.

The camera matrix is implicitly created when the Toolbox camera object is constructed and for this example is

```
>> cam.C
ans =
1.0e+03 *
1.5000         0    0.6400         0
      0    1.5000    0.5120         0
      0         0    0.0010         0
```

and the camera parameter matrix  $\mathbf{K}$  is

```
>> cam.K
ans =
1.0e+05 *
1.0000         0    0.0064
      0    1.0000    0.0051
      0         0    0.0000
```

The field of view of a lens is an open rectangular pyramid that subtends angles  $\theta_h$  and  $\theta_v$  in the horizontal and vertical planes respectively. The dimension of a sensor chip  $d$  is measured diagonally between its corners and is typically expressed in inches. Common dimensions are  $\frac{1}{4}$ ,  $\frac{1}{3}$  and  $\frac{1}{2}$  inch. A *normal lens* has  $f \approx d$  and a wide-angle lens generally has  $f > d/3$  giving a maximum angular field of view of around  $110^\circ$ .

For wide-angle lenses it is more common to describe the field of view as a solid angle which is measured in units of steradians (or sr). This is the area of the field of view projected onto the surface of a unit sphere. A hemispherical field of view is  $2\pi\text{sr}$  and a full spherical view is  $4\pi\text{sr}$ . If we approximate the camera's field of view by a cone with apex angle  $\theta$  the corresponding solid angle is  $2\pi(1 - \cos \theta/2)\text{sr}$ .

A camera with a field of view greater than a full hemisphere is termed omni-directional or panoramic.

The field of view of a camera is a function of its focal length  $f$ . A wide-angle lens has a small focal length, a telephoto lens has a large focal length, and a zoom lens has an adjustable focal length. The field of view can be determined from the geometry of Fig. 11.5. In the horizontal direction the half-angle of view is

$$\frac{\theta_h}{2} = \tan^{-1} \frac{W/2\rho_w}{f}$$

where  $N$  is the number of pixels in the horizontal direction. We can then write

$$\theta_h = 2\tan^{-1} \frac{W\rho_w}{2f}, \theta_v = 2\tan^{-1} \frac{H\rho_h}{2f} \quad (11.11)$$

We note that the field of view is also a function of the dimensions of the camera chip which is  $W\rho_w \times H\rho_h$ . The field of view is computed by the `fov` method of the camera object

```
>> cam.fov() * 180/pi
ans =
    46.2127    37.6930
```

in degrees in the horizontal and vertical directions respectively.

The `CentralCamera` class is a subclass of the `Camera` class and inherits the ability to project multiple points or lines. Using the Toolbox we create a  $3 \times 3$  grid of points in the  $xy$ -plane with overall side length 0.2 m and centred at (0, 0, 1)

```
>> P = mkgrid(3, 0.2, 'T', transl(0, 0, 1));
```

which returns a  $3 \times 9$  matrix with one column per grid point where each column comprises the coordinates in  $X, Y, Z$  order. The first four columns are

```
>> P(:,1:4)
ans =
    -0.1000    -0.1000    -0.1000         0
    -0.1000         0     0.1000   -0.1000
    1.0000    1.0000    1.0000    1.0000
```

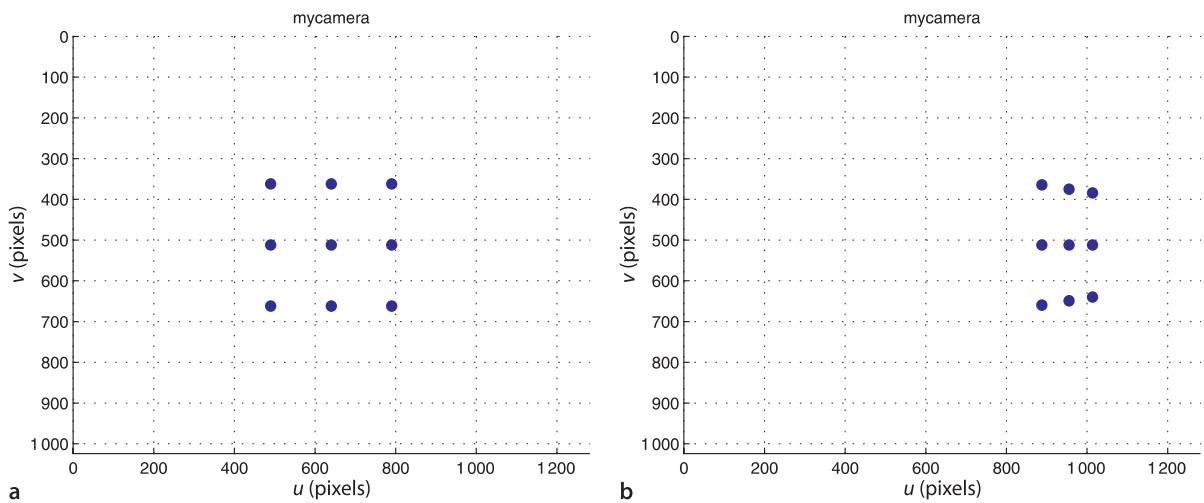
By default `mkgrid` generates a grid in the  $xy$ -plane that is centred at the origin. The optional last argument is a homogeneous transformation that is applied to the default points and allows the plane to be arbitrarily positioned and oriented.

The image plane coordinates of the vertices are

```
>> cam.project(P)
ans =
    490    490    490    640    640    640    790    790    790
    362    512    662    362    512    662    362    512    662
```

which can also be plotted

```
>> cam.plot(P)
```



**Fig. 11.6.** Two views of a planar square target, where the corner points are marked with ‘\*’. **a** A frontal view, **b** oblique view

giving the virtual camera view shown in Fig. 11.6a. The camera pose

```
>> Tcam = transl(-1,0,0.5)*trotz(0.9);
```

results in an oblique view of the plane

```
>> cam.plot(P, 'Tcam', Tcam)
```

shown in Fig. 11.6b. We can clearly see the effect of perspective projection which has distorted the shape of the square – the top and bottom edges, which are parallel lines, have been projected to lines that converge at a vanishing point.

The vanishing point for a line can be determined from the projection of its ideal line. The top and bottom lines of the grid are parallel to the world vector  $(1, 0, 0)$  and the ideal line is  $(1, 0, 0, 0)$ . This homogeneous line exists at infinity due to the final zero element. The vanishing point is therefore

```
>> cam.project([1 0 0 0]', 'Tcam', Tcam)
ans =
1.0e+03 *
1.8303
0.5120
```

which is  $(1803, 512)$  and just to the right of the visible image plane.

The `plot` method can optionally return the image-plane coordinates

```
>> p = cam.plot(P, 'Tcam', Tcam)
```

just like the `project` method. For the oblique viewing case the image plane coordinates

```
>> p(:,1:4)
ans =
887.7638 887.7638 887.7638 955.2451
364.3330 512.0000 659.6670 374.9050
```

have a fractional component which means that the point is not projected to the centre of the pixel. However a pixel responds to light equally over its surface area so the discrete pixel coordinate can be obtained by rounding.

A 3-dimensional object, a cube, can be defined and projected in a similar fashion. The vertices of a cube with side length 0.2 m and centred at  $(0, 0, 1)$  can be defined by

```
>> cube = mkcube(0.2, 'T', transl([0, 0, 1]));

```

which returns a  $3 \times 8$  matrix with one column per vertex. The image plane points can be plotted as before by

```
>> cam.plot(cube);
```

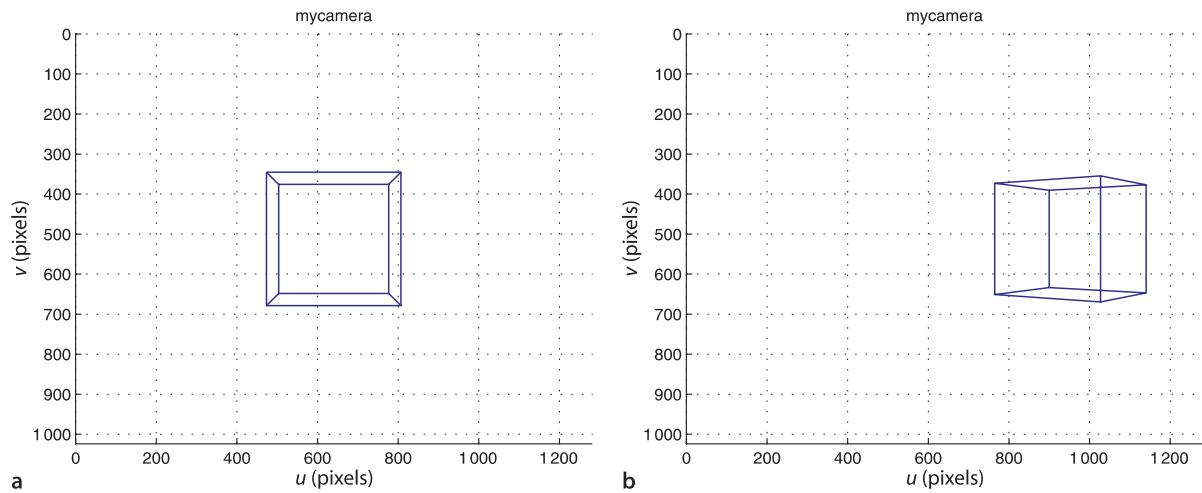
This is not strictly true for CMOS sensors where transistors reduce the light sensitive area by the fill factor – the fraction of each photosite’s area that is light sensitive.

**Photons to pixel values.** At each photosite photons are converted to electrons, and the fraction of incoming photons converted to electrons is the quantum efficiency of the sensor. Not all of a photosite is light sensitive due to the presence of transistors and other devices – the fraction of the photosite's area that is sensitive is called its fill factor and for CMOS sensors can be less than 50%. The electrons are accumulated in a charge well during the exposure interval. In a CMOS sensor, at the end of the exposure interval the charge is converted to a voltage and amplified, and then a switching network connects each pixel to an on-chip analog to digital converter. The amount of charge accumulated in each well is proportional to the product of the photon arrival rate (scene luminance) and the exposure interval. The first is a function of the lens *f*-number and overall scene brightness. The second has an upper bound of the frame interval, the time between consecutive frames in a video sequence. Digital cameras often adjust the exposure interval instead of relying on an expensive, and slow, mechanical aperture. The charge well has a maximum capacity and in some sensors surplus electrons can overflow into adjacent photosites.

At low light levels the camera uses an analog amplifier to boost the weak signal from the photosites, but this also amplifies noise and leads to a grainy appearance in the image. This noise results from a flow of electrons known as dark current which also accumulates into the charge well. The dark current is caused by thermal processes and sensitive sensors used for astronomy minimize this by cryogenically cooling the sensor.

Another source of noise is pixel non-uniformity due to adjacent pixels having a different gain or offset – uniform illumination therefore leads to pixels with different values which appears as additive noise. Image sensors typically have some pixels at the edge that are covered by metalization, and these provide a black reference which is subtracted from other pixel values.

One final consideration with a CMOS camera is that the pixels are read sequentially in raster order (left to right, top to bottom) so the bottom-right pixel is sampled much later than the top-left pixel. To combat motion blur a short exposure time is needed, which leads to darker and noisier images.



Alternatively we can create an *edge* representation of the cube by

```
>> [X,Y,Z] = mkcube(0.2, 'T', transl([0, 0, 1.0]), 'edge');
```

and display it

```
>> cam.mesh(X, Y, Z)
```

as shown in Fig. 11.7 along with an oblique view

```
>> cam.T = transl(-1,0,0.5)*trotz(0.8);
>> cam.mesh(X, Y, Z, 'Tcam', Tcam);
```

The edges are in the same 3-dimensional mesh format<sup>►</sup> as generated by MATLAB® builtin functions such as `sphere`, `ellipsoid` and `cylinder`.

Successive calls to `plot` will redraw the points or line segments and provides a simple method of animation. The short piece of code

**Fig. 11.7.** Line segment representation of a cube. **a** Frontal view, **b** oblique view

The elements of the mesh  $(i, j)$  have coordinates  $(X[i, j], Y[i, j], Z[i, j])$ .

```

1 theta = [0:500]/100*2*pi;
2 [X,Y,Z] = mkcube(0.2, [], 'edges');
3 for th=theta
4     T_cube = transl(0, 0, 1.5)*trotx(th)*trot(y(th*1.2))*trot(z(th*1.3))
5     cam.mesh( X, Y, Z, T_cube );
6 end

```

shows a cube tumbling in space. The cube is defined with its centre at the origin and its vertices are transformed at each time step.

### 11.1.1 Lens Distortion

No lenses are perfect and the low-cost lenses used in many webcams are far from perfect. Lens imperfections result in a variety of distortions including chromatic aberration (color fringing), spherical aberration or astigmatism (variation in focus across the scene), and geometric distortions where points on the image plane are displaced from where they should be according to Eq. 11.3.

Geometric distortion is generally the most problematic effect that we encounter for robotic applications, and comprises two components: radial and tangential. Radial distortion causes image points to be translated along radial lines from the principal point. The radial error is well approximated by a polynomial

$$\delta r = k_1 r^3 + k_2 r^5 + k_3 r^7 + \dots \quad (11.12)$$

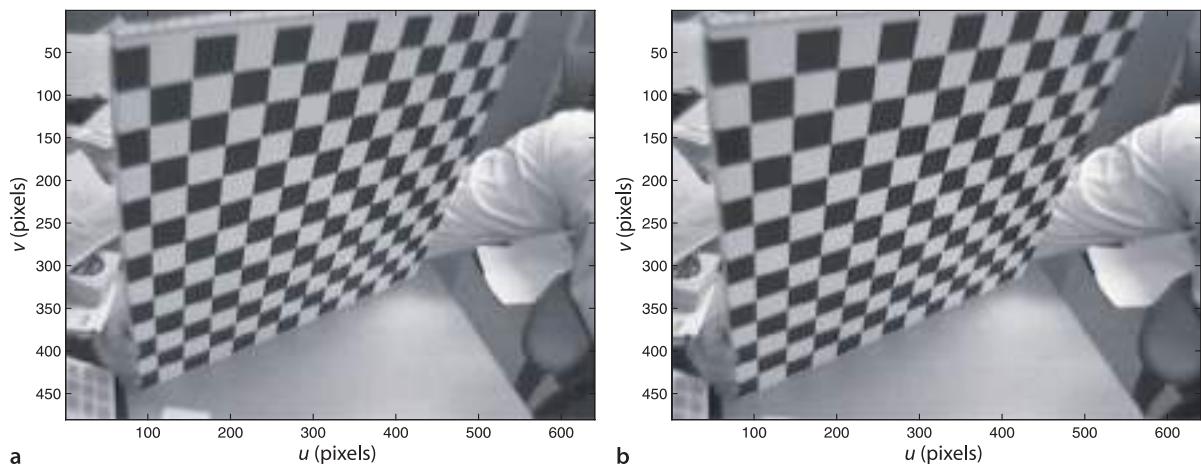
where  $r$  is the distance of the image point from the principal point. Barrel distortion occurs when magnification decreases with distance from the principal point which causes straight lines near the edge of the image to curve outward. Pincushion distortion occurs when magnification increases with distance from the principal point and causes straight lines near the edge of the image to curve inward. Tangential distortion, or decentering distortion, occurs at right angles to the radii but is generally less significant than radial distortion. Examples of a distorted and undistorted image are shown in Fig. 11.8.

**Fig. 11.8.** Lens distortion. **a** Distorted image, the curvature of the top row of the squares is quite pronounced, **b** undistorted image. This image is calibration image #19 from Bouguet's Camera Calibration Toolbox (Bouguet 2010)

The coordinate of the point  $(u, v)$  after distortion is given by

$$u^d = u + \delta_u, v^d = v + \delta_v \quad (11.13)$$

where the displacement is



$$\begin{pmatrix} \delta_u \\ \delta_v \end{pmatrix} = \underbrace{\begin{pmatrix} u(k_1r^2 + k_2r^4 + k_3r^6 + \dots) \\ v(k_1r^2 + k_2r^4 + k_3r^6 + \dots) \end{pmatrix}}_{\text{radial}} + \underbrace{\begin{pmatrix} 2p_1uv + p_2(r^2 + 2u^2) \\ p_1(r^2 + 2v^2) + 2p_1uv \end{pmatrix}}_{\text{tangential}} \quad (11.14)$$

This displacement vector can be plotted for different values of  $(u, v)$  as shown in Fig. 11.12b. The vectors indicate the displacement required to *correct* the distortion at different points in the image, in fact  $(-\delta_u, -\delta_v)$ , and shows dominant radial distortion.

Typically three coefficients are sufficient to describe the radial distortion and the distortion model is parameterized by  $(k_1, k_2, k_3, p_1, p_2)$  which are considered as additional intrinsic parameters. Distortion can be modeled by the `CentralCamera` class using the '`distortion`' option, for example

```
>> cam = CentralCamera('focal', 0.015, 'pixel', 10e-6, ...
    'resolution', [1280 1024], 'centre', [512 512], ...
    'distortion', [k1 k2 k3 p1 p2] )
```

## 11.2 Camera Calibration

The camera projection model Eq. 11.9 has a number of parameters that in practice are unknown. In general the principal point is *not* at the centre of the photosite array. The focal length of a lens is only accurate<sup>▶</sup> to 4% of what it purports to be, and is only correct if the lens is focussed at infinity. It is also common experience that the intrinsic parameters change if a lens is detached and reattached, or adjusted for focus or aperture.<sup>▶</sup> The only intrinsic parameters that it may be possible to obtain are the photosite dimensions  $\rho_w$  and  $\rho_h$  from the sensor manufacturer's data sheet. The extrinsic parameters, the camera's pose, raises the question of where exactly is the centre point of the camera.

Camera calibration is the process of determining the camera's intrinsic parameters and the extrinsic parameters with respect to the world coordinate system. Calibration techniques rely on sets of world points whose relative coordinates are known and whose corresponding image-plane coordinates are also known. State-of-the-art techniques such as Bouguet's Calibration Toolbox for MATLAB® (Bouguet 2010) simply require a number of images of a planar chessboard target such as shown in Fig. 11.11. From this, as discussed in Sect. 11.2.4, the intrinsic parameters (including distortion parameters) can be estimated as well as the relative pose of the chessboard in each image. Classical calibration techniques require a single view of a 3-dimensional calibration targets but are unable to estimate the distortion model. These methods are however easy to understand and they start of our discussion in the next section.

According to ANSI Standard PH3.13-1958 "Focal Length Marking of Lenses".

Changing focus rotates the lens about the optical axis and if the lens is not perfectly symmetric this will move the distortions with respect to the image plane. Changing the aperture alters the parts of the lens that light rays pass through and hence the distortion that they incur.

### 11.2.1 Homogeneous Transformation Approach

The homogeneous transform method allows direct estimation of the camera matrix  $C$  in Eq. 11.9. The elements of this matrix are functions of the intrinsic and extrinsic parameters. Setting  $\tilde{p} = (u, v, 1)$ , expanding equation Eq. 11.9 and substituting into Eq. 11.8 we can write

$$\begin{aligned} C_{11}X + C_{12}Y + C_{13}Z + C_{14} - C_{31}uX - C_{32}uY - C_{33}uZ - C_{34}u &= 0 \\ C_{21}X + C_{22}Y + C_{23}Z + C_{24} - C_{31}vX - C_{32}vY - C_{33}vZ - C_{34}v &= 0 \end{aligned} \quad (11.15)$$

where  $(u, v)$  are the pixel coordinates corresponding to the world point  $(X, Y, Z)$  and  $C_{ij} = C[i, j]$  are elements of the unknown camera matrix.

Calibration requires a 3-dimensional target such as shown in Fig. 11.9. The position of the centre of each marker  $(X_i, Y_i, Z_i)$ ,  $i \in [1, N]$  with respect to the target

**Where is the camera's centre?** A compound lens has many cardinal points including focal points, nodal points, principal points and planes, entry and exit pupils. The entrance pupil is a point on the optical axis of a compound lens system that is its centre of perspective or its *no-parallax point*. We could consider it to be the *virtual pinhole*. Rotating the camera and lens about this point will not change the relative geometry of targets at different distances in the perspective image.

Rotating about the entrance pupil is important in panoramic photography to avoid parallax errors in the final, stitched panorama. A number of web pages are devoted to discussion of techniques for determining the position of this point. Some sites even tabulate the position of the entrance pupil for popular lenses. Much of this online literature refers to this point incorrectly as the *nodal point* even though the techniques given do identify the *entrance pupil*.

Depending on the lens design, the entrance pupil may be behind, within or in front of the lens system.

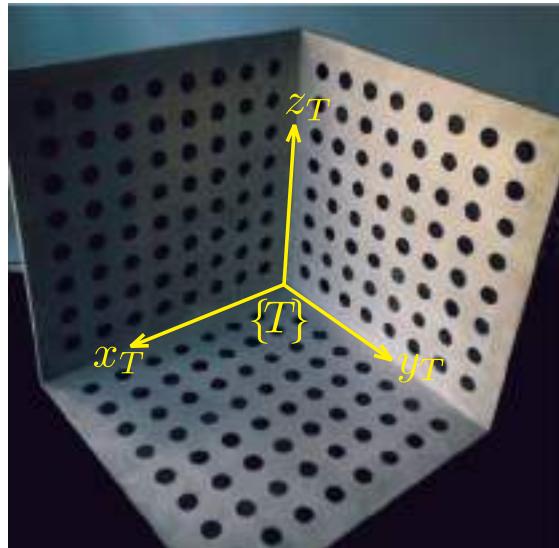


Fig. 11.9.

A 3D calibration target showing its coordinate frame  $\{T\}$ . The centroids of the circles are taken as the calibration points. Note that the calibration circles are situated on three planes (photo courtesy of Fabien Spindler)

frame  $\{T\}$  must be known, but  $\{T\}$  itself is not known. An image is captured and the corresponding image-plane coordinates  $(u_i, v_i)$  are determined. For each of the  $N$  markers we stack the two equations of Eq. 11.15 to form the matrix equation

$$\begin{pmatrix} X_1 & Y_1 & Z_1 & 1 & 0 & 0 & 0 & -u_1 X_1 & -u_1 Y_1 & -u_1 Z_1 \\ 0 & 0 & 0 & 0 & X_1 & Y_1 & Z_1 & 1 & -v_1 X_1 & -v_1 Y_1 & -v_1 Z_1 \\ & & & & \vdots & & & & & & \\ X_N & Y_N & Z_N & 1 & 0 & 0 & 0 & -u_N X_N & -u_N Y_N & -u_N Z_N \\ 0 & 0 & 0 & 0 & X_N & Y_N & Z_N & 1 & -v_N X_N & -v_N Y_N & -v_N Z_N \end{pmatrix} \begin{pmatrix} C_{11} \\ C_{12} \\ \vdots \\ C_{33} \end{pmatrix} = \begin{pmatrix} u_1 \\ v_1 \\ \vdots \\ u_N \\ v_N \end{pmatrix} \quad (11.16)$$

which can be solved for the camera matrix elements  $C_{11} \dots C_{33}$ . The solution can only be determined to within a scale factor and by convention  $C_{34}$  is set equal to 1. Equation 11.16 has 11 unknowns and for solution requires that  $N \geq 6$ . Often more than 6 points will be used leading to an over-determined set of equations which is solved using least squares.

If the points are coplanar then the left-hand matrix of Eq. 11.16 becomes rank deficient. This is why the calibration target must be 3-dimensional, typically an array of dots or squares on two or three planes as shown in Fig. 11.9.

We will illustrate this with an example. The calibration target is a cube, the markers are its vertices and its coordinate frame  $\{T\}$  is parallel to the cube faces with its origin at the centre of the cube. The coordinates of the markers with respect to  $\{T\}$  are

```
>> P = mkcube(0.2);
```

Now the calibration target is at some “unknown pose”  ${}^C\xi_T$  with respect to the camera which we choose to be

```
>> T_unknown = transl(0.1, 0.2, 1.5) * rpy2tr(0.1, 0.2, 0.3);
```

Next we create a perspective camera whose parameters we will attempt to estimate

```
>> cam = CentralCamera('focal', 0.015, ...
    'pixel', 10e-6, 'resolution', [1280 1024], 'centre', [512 512], ...
    'noise', 0.05);
```

We have also specified that zero-mean Gaussian noise with  $\sigma=0.05$  is added to the  $(u, v)$  coordinates to model camera noise and errors in the computer vision algorithms. The image plane coordinates of the calibration target at its “unknown” pose are

```
>> p = cam.project(P, 'Tobj', T_unknown);
```

Now using just the object model  $P$  and the observed image features  $p$  we estimate the camera matrix

```
>> C = camcald(P, p)
maxm residual 0.067393 pixels.
C =
 883.1620 -240.0720  531.4419  612.0432
 259.3786  994.1921  234.8182  712.0180
 -0.1043    0.0985    0.6494    1.0000
```

The maximum residual in this case is less than 0.1 pixel, that is, the worst error between the projection of a world point using the camera matrix  $C$  and the actual image plane location is very small.

Linear techniques such as this cannot estimate lens distortion parameters. The distortion will introduce errors into the camera matrix elements but for many situations this might be acceptably low. Distortion parameters are often estimated using a non-linear optimization over all parameters, typically 16 or more, with the linear solution used as the initial parameter estimate.

### 11.2.2 Decomposing the Camera Calibration Matrix

The elements of the camera matrix are functions of the intrinsic and extrinsic parameters. However given a camera matrix most of the parameter values can be recovered. Continuing the example from above we *decompose* the estimated camera matrix

```
>> est = invcamcal(C)
est =
name: invcamcal [central-perspective]
focal length: 1504
pixel size: (1, 0.9985)
principal pt: (518.6, 505)
Tcam:
 0.93695   -0.29037    0.19446    0.08339
 0.31233    0.94539   -0.093208   -0.39143
 -0.15677    0.14807    0.97647   -1.4671
    0         0         0         1
```

which returns a `CentralCamera` object with its parameters set to values that result in the same camera matrix. We note immediately that the focal length is very large compared to the true focal length of our lens which was 0.015 m, and that the pixel sizes are very large. From Eq. 11.9 we see that focal length and pixel dimensions always appear together as factors  $f/\rho_w$  and  $f/\rho_h$ . The function `invcamcal` has set  $\rho_w=1$  but the ratios of the estimated parameters

```
>> est.f/est.rho(1)
ans =
1.5044e+03
```

These quantities have units of pixels since  $\rho$  has units of m pixel $^{-1}$ . It is quite common in the literature to consider  $\rho=1$  and the focal length is given in pixels. If the pixels are not square then different focal lengths  $f_u$  and  $f_v$  must be used for the horizontal and vertical directions respectively.

are very close to the ratio for the true parameters of the camera

```
>> cam.f/cam.rho(2)
ans =
1.500e+03
```

The small error in the estimated parameter values is due to the noisy image-plane coordinate values we used for calibration.

The pose of the estimated camera is with respect to the calibration target  $\{T\}$  and is therefore  ${}^T\xi_C$ . The true position of the target with respect to the camera is  ${}^C\xi_T$ . If our estimation is accurate then  ${}^C\xi_T \oplus {}^T\xi_C$  will be 0. We earlier we set the variable `T_unknown` equal to  ${}^C\xi_T$  and for our example we find that

```
>> T_unknown*est.T
ans =
0.7557   -0.5163    0.4031   -0.0000
0.6037    0.7877   -0.1227   -0.0001
-0.2541    0.3361    0.9069   -0.0041
0          0         0        1.0000
```

which is the relative pose between the true and estimated camera pose. The camera pose is estimated to better than 5 mm in position.

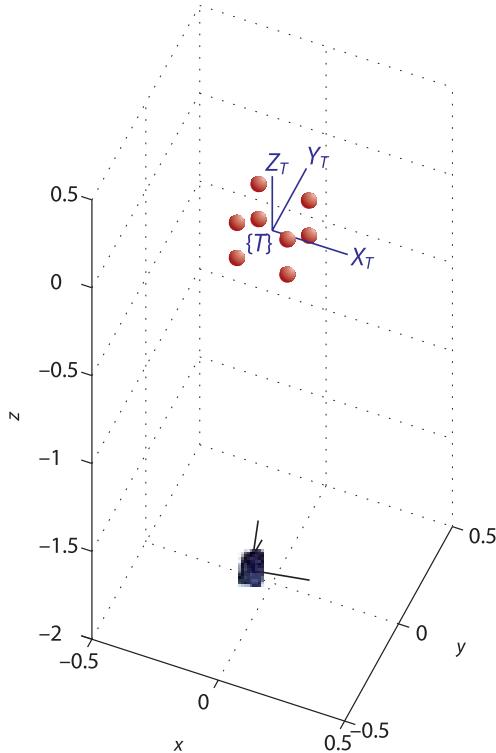
We can plot the calibration markers as small red spheres

```
>> plot_sphere(P, 0.03, 'r')
>> plot_frame(eye(4,4), 'frame', 'T', 'color', 'b', 'length', 0.3)
```

as well as  $\{T\}$  which we have set at the world origin. The estimated pose of the camera can be superimposed

```
>> est.plot_camera()
```

and the result is shown in Fig. 11.10. The problem of determining the position of a camera with respect to a calibration object is an important problem in photogrammetry known as the camera location determination problem.



**Fig. 11.10.**  
Calibration target points and estimated camera pose with respect to the target frame  $\{T\}$  which is assumed to be at the origin. The camera is depicted as a rectangular pyramid and the side represent the bounds of the field of view

### 11.2.3 Pose Estimation

The pose estimation problem is to determine the pose  ${}^C\xi_T$  of a target's coordinate frame  $\{T\}$  with respect to the camera. The geometry of the target is known, that is, we know the position of a number of points  $(X_i, Y_i, Z_i)$ ,  $i \in [1, N]$  on the target with respect to  $\{T\}$ . The camera's intrinsic parameters are also known. An image is captured and the *corresponding* image-plane coordinates  $(u_i, v_i)$  are determined using computer vision algorithms.

Estimating the pose using  $(u_i, v_i)$ ,  $(X_i, Y_i, Z_i)$  and camera intrinsic parameters is known as the Perspective-n-Point problem or PnP for short. It is a simpler problem than camera calibration and decomposition because there are fewer parameters to estimate. To illustrate pose estimation we will create a calibrated camera with known parameters

```
>> cam = CentralCamera('focal', 0.015, 'pixel', 10e-6, ...
    'resolution', [1280 1024], 'centre', [640 512]);
```

The object whose pose we wish to determine is a cube with side lengths of 0.2 m and the coordinates of the markers with respect to  $\{T\}$  are

```
>> P = mkcube(0.2);
```

which we can consider a simple geometric model of the object. The object is at some arbitrary but unknown pose  ${}^C\xi_T$  pose with respect to the camera

```
>> T_unknown = transl(0,0,2)*trotx(0.1)*trotz(0.2)
T_unknown =
  0.9801      0     0.1987      0
  0.0198    0.9950   -0.0978      0
 -0.1977    0.0998    0.9752    2.0000
    0         0         0    1.0000
```

The image plane coordinates of the object's points at its unknown pose are

```
>> p = cam.project(P, 'Tobj', T_unknown);
```

Now using just the object model **P**, the observed image features **p** and the calibrated camera **cam** we estimate the relative pose  ${}^C\xi_T$  of the object

```
>> T_est = cam.estpose(P, p)
T_est =
  0.9801    0.0000    0.1987   -0.0000
  0.0198    0.9950   -0.0978   -0.0000
 -0.1977    0.0998    0.9752    2.0000
    0         0         0    1.0000
```

which is the same (to four decimal places) as the unknown pose **T\_unknown** of the object.

In reality the image features coordinates are imperfectly estimated by the vision system and we would model this by adding zero-mean Gaussian noise to the image feature coordinates as we did in the camera calibration example.

### 11.2.4 Camera Calibration Toolbox

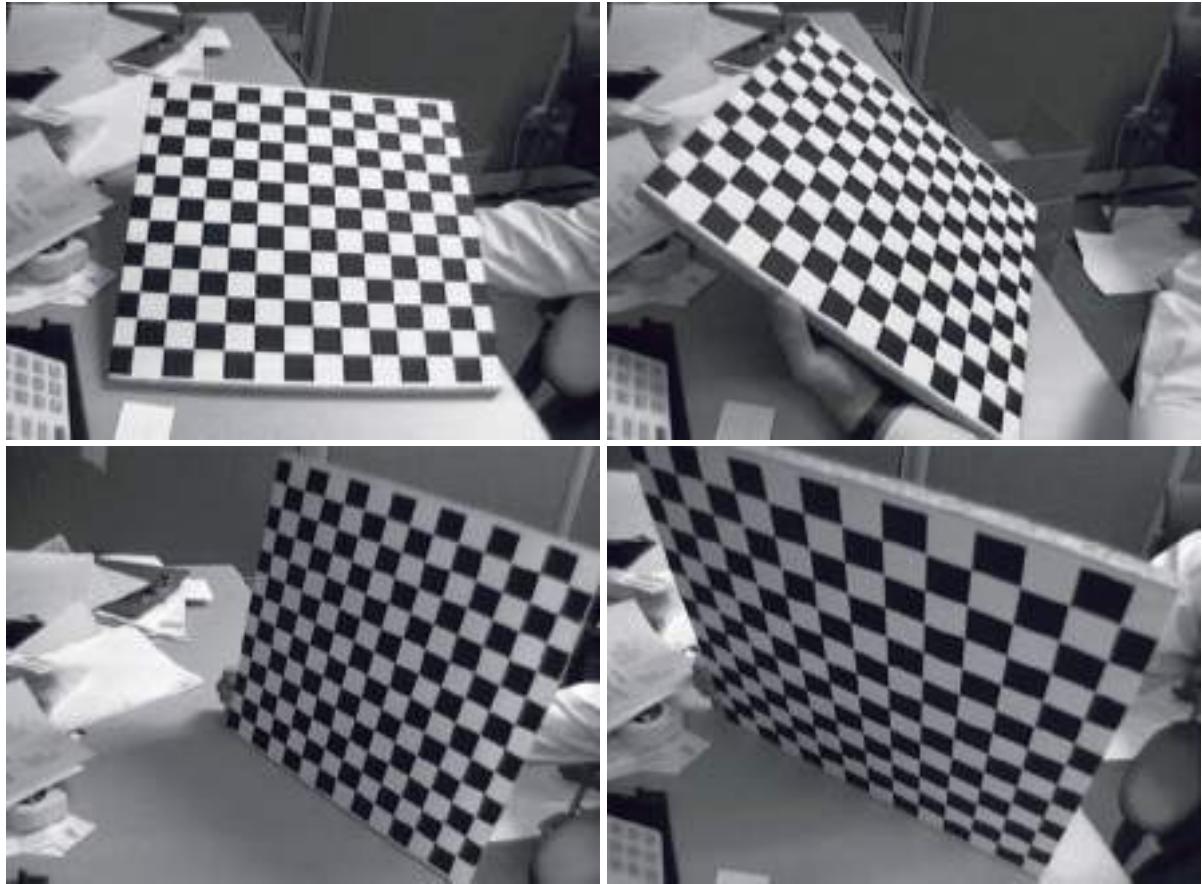
A popular and practical tool for calibrating cameras using a planar chessboard target is the Camera Calibration Toolbox. A number of images, typically twenty, are taken of the target at different distances and orientations as shown in Fig. 11.11.

The calibration tool is launched by

```
>> calib_gui
```

and a graphical user interface (GUI) is displayed.► The first step is to load the images using the **(Image Names)** button. The second step is the **(Extract Grid Corners)** button which

The GUI is optional, and the Toolbox functions can be called from inside your own programs. The function **calib\_gui\_normal** shows the mapping from GUI button names to Calibration Toolbox function names. Note that most of the functions are actually scripts and program **state** variables are kept in the workspace.



**Fig. 11.11.** Example frames from Bouguet’s Calibration Toolbox showing the calibration target in many different orientations. These are images 2, 5, 9, 18 from the Calibration Toolbox example

prompts you to pick the corners of the calibration target in each of the images. This is a little tedious but needs to be done carefully. The final step, the **Calibration** button, uses the calibration target information to estimate the camera parameter values

```
Focal Length:          fc = [ 661.66988   662.82841 ] ? [ 1.17902   1.26556 ]
Principal point:      cc = [ 306.09593   240.79019 ] ? [ 2.38421   2.17462 ]
Skew:                 alpha_c = [ 0.00000 ] ? [ 0.00000 ]
                      => angle of pixel axes = 90.00000 ? 0.00000 degrees
Distortion:           kc = [ -0.26424   0.22644   0.00020   0.00023   0.00000 ]
                      ? [ 0.00933   0.03826   0.00052   0.00053   0.00000 ]
Pixel error:          err = [ 0.45328   0.38910 ]
```

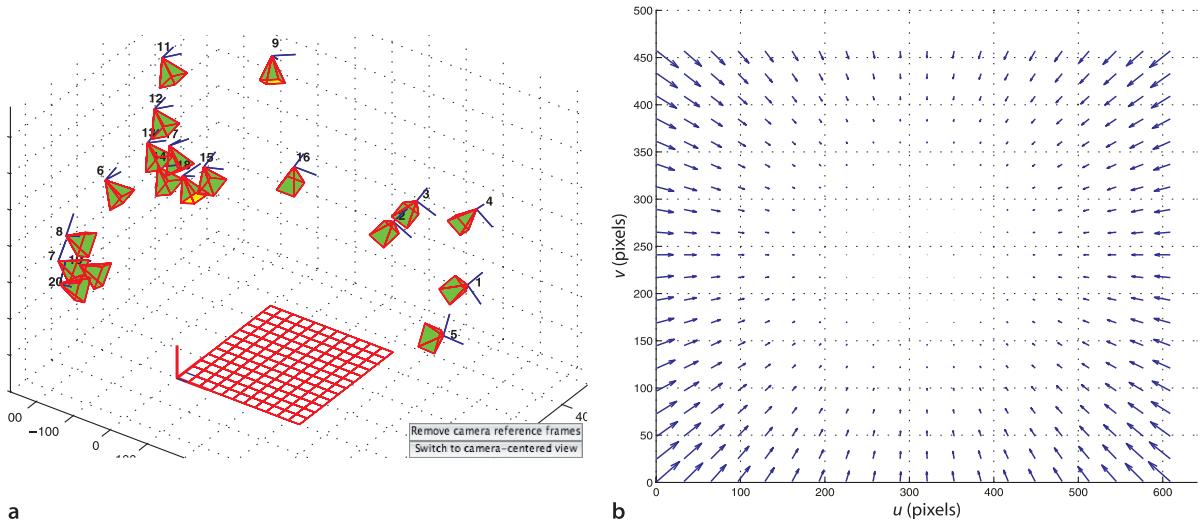
For each parameter the uncertainty ( $3\sigma$  bounds) is estimated and displayed after the question mark.

The camera pose relative to the target is estimated for each calibration image and can be displayed using the **Show Extrinsic** button. This target-centric view is shown in Fig. 11.12a indicates the estimated relative pose of the camera for each input image.

The distortion vector **kc** contains the parameters in the order  $(k_1, k_2, p_1, p_2, k_3)$  – note that  $k_3$  is out of sequence. The distortion map can be displayed by

```
>> visualize_distortions
```

and is shown in Fig. 11.12b. This indicates the displacement from true to distorted image-plane coordinates which in this case is predominately in the radial direction. This is consistent with  $k_1$  and  $k_2$  being orders of magnitude greater than  $p_1$  and  $p_2$  which is typical for most lenses. The **Undistort Image** button can be used to undistort a set of images and a distorted and undistorted image are compared in Fig. 11.8b. The details of this transformation using image warping will be discussed in Sect. 12.6.4.



**Fig. 11.12.** Calibration results from the example in Bouguet's Calibration Toolbox. **a** The estimated camera pose relative to the target for each calibration image, **b** the distortion map with vectors showing how points in the image will move due to distortion



**Fig. 11.13.**  
Images formation by reflection from a curved surface (*Hand with Reflecting Sphere*, M. C. Escher, 1935). Note that straight lines have become curves and the light hangs at an angle

**Fig. 11.14.**

Fisheye lens image. Note that straight lines in the world are no longer projected as straight lines. Note also that the field of view is mapped to a circular region on the image plane

### 11.3 Non-Perspective Imaging Models

We have discussed perspective imaging in quite some detail since it is the model of our own eyes and almost all cameras that we encounter. However perspective imaging constrains us to a fundamentally limited field of view. The perspective projection Eq. 11.3 is singular for points with  $Z = 0$  which limits the field of view to at most one hemisphere – real lenses achieve far less. As the focal length decreases radial distortion is increasingly difficult to eliminate and eventually a limit is reached beyond which lenses cannot practically be built. The only way forward is to drop the constraint of perspective imaging. In Sect. 11.3.1 we describe the geometry of image formation with wide-angle lens systems.

An alternative to refractive optics is to use a reflective surface to form the image such as shown in Fig. 11.13. Newtonian telescopes, are based on reflection from concave mirrors rather than refraction by lenses. Mirrors are free of color fringing and are easier to scale to larger sizes than a lens. Nature has also evolved reflective optics – the spookfish and some scallops (see page 221) have eyes based on reflectors formed from guanine crystals.

**Refraction and reflection.** The oldest example of a lens is from Assyria around 2000 BCE. It was a *burning-glass* used to start fires and these were well known by the time of the ancient Greeks. The word lens comes from the Latin name of the lentil because a convex lens is lentil-shaped. The Arab physicist and mathematician Ibn Sahl (circa 940–1000) used what we now know as Snell's law to calculate the shape of lenses. *Reading stones* appeared in the 11<sup>th</sup> century and spectacles were invented in Italy in the 1280s.

The first telescope was developed by spectacle makers in the Netherlands in 1608 and Galileo improved that design and went on to discover four moons (the Galilean satellites) of Jupiter. In 1668 Sir Isaac Newton built the first reflecting telescope which eliminated chromatic aberration that was a problem in refracting telescopes. Early mirrors and telescope reflectors were made from *speculum* metal, an alloy of copper, tin and arsenic. Speculum is Latin for mirror and leads to terms like specular reflection for mirror-like reflection, or glinting, from objects.

The terms dioptrics and catoptrics are derived from ancient Greek words for refraction and reflection respectively. A catadioptric camera comprises both reflecting and refractive elements – mirrors and lenses.

### 11.3.1 Fisheye Lens Camera

We start by considering a fisheye lens using the notation shown in Fig. 11.15 where the camera is positioned at the origin  $O$  and its optical axis is the  $z$ -axis. The world point  $P$  is represented in spherical coordinates  $(R, \theta, \phi)$ , where  $\theta$  is the angle outward from the optical axis and  $\phi$  is the angle of rotation around the optical axis. We can write

$$R = \sqrt{X^2 + Y^2 + Z^2}, \theta = \cos^{-1} \frac{R}{Z}, \phi = \tan^{-1} \frac{Y}{X}$$

On the image plane of the camera we represent the projection  $p$  in polar coordinates  $(r, \phi)$  with respect to the principal point, where  $r = r(\theta)$ . The Cartesian image plane coordinates are

$$u = r \cos \phi, v = r \sin \phi$$

The image-plane displacement from the principal point  $r(\theta)$  is a function of the angle  $\theta$  and depends on the type of fisheye lens. Some common projection models are listed in Table 11.1.

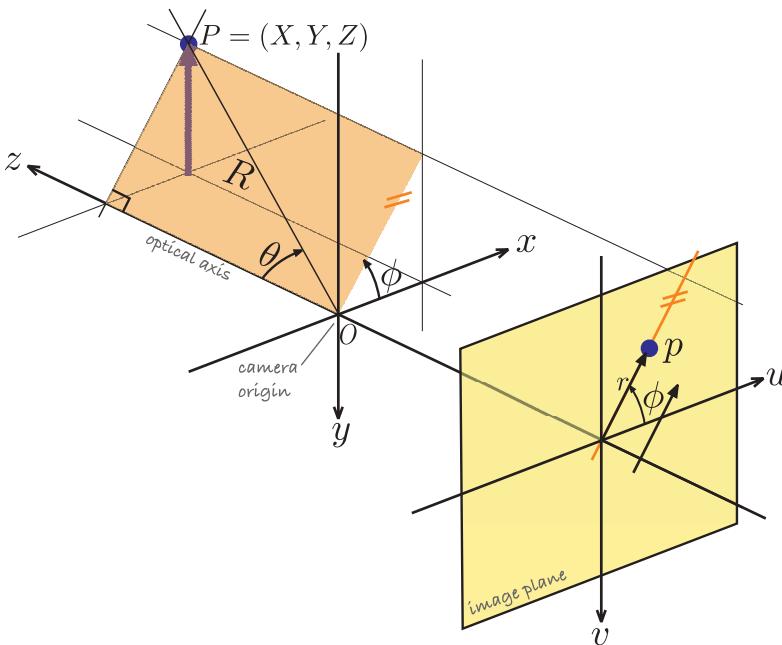
Using the Toolbox we can create a fisheye camera

```
>> cam = FishEyeCamera('name', 'fisheye', ...
    'projection', 'equiangular', ...
    'pixel', 10e-6, ...
    'resolution', [1280 1024])
```

which returns an instance of a `FishEyeCamera` object which is a subclass of the Toolbox's `Camera` object and polymorphic with the `CentralCamera` class discussed earlier. If  $k$  is not specified, as in this example, then it is computed such that a hemispheric field of view is projected into the maximal circle on the image plane. As is the case for perspective cameras the parameters such as principal point and pixel dimensions are generally not known and must be estimated using a calibration procedure.

We create an edge-based model of a cube with side length 0.2 m

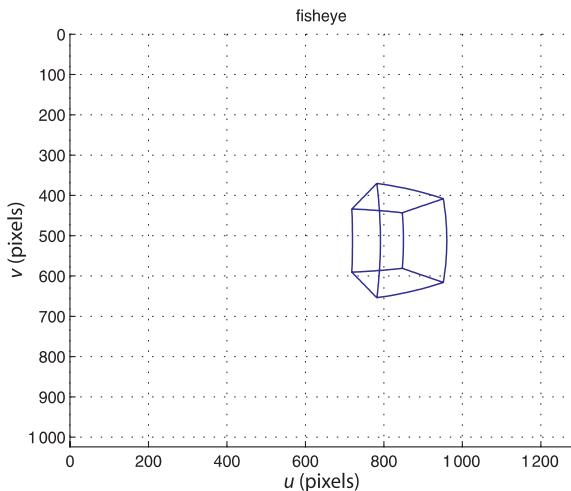
```
>> [x,y,z] = mkcube(0.2, 'centre', [0.2, 0, 0.3], 'edge');
```



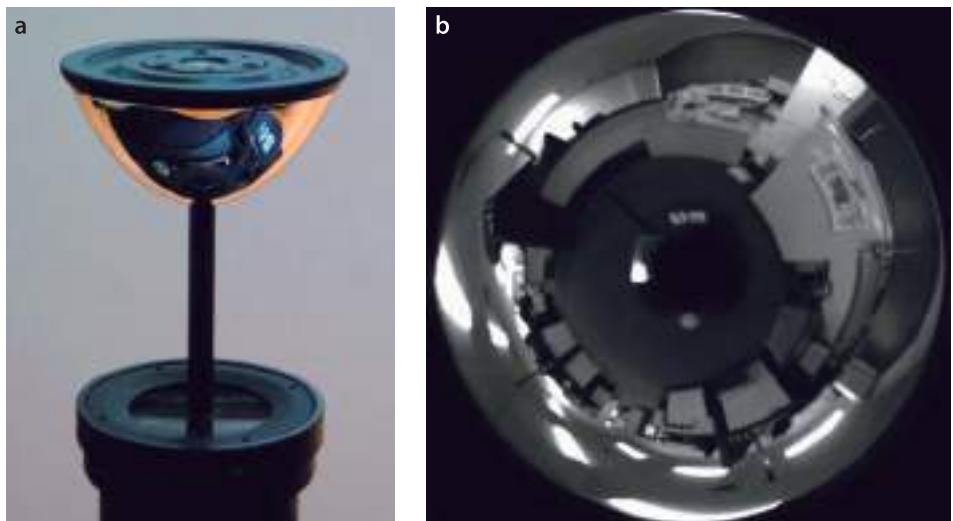
**Fig. 11.15.**  
Image formation for a fisheye lens camera. The world point  $P$  is represented in spherical coordinates  $(R, \theta, \phi)$  with respect to the camera's origin

**Table 11.1.**  
Fisheye lens projection models

Mapping	Equation
Equiangular	$r = k\theta$
Stereographic	$r = k \tan(\theta/2)$
Equisolid	$r = k \sin(\theta/2)$
Polynomial	$r = k_1\theta + k_2\theta^2 + \dots$



**Fig. 11.16.**  
A cube projected using the `FishEyeCamera` class. The straight edges of the cube are curves on the image plane



**Fig. 11.17.**  
Catadioptric imaging. **a** A catadioptric imaging system comprising a conventional perspective camera is looking upward at the mirror; **b** Catadioptric image. Note the dark spot in the centre which is the support that holds the mirror above the lens

and project it to the fisheye camera's image plane

```
>> cam.mesh(X, Y, Z)
```

and the result is shown in Fig. 11.16.

Wide angle lenses are available with  $180^\circ$  and even  $190^\circ$  field of view, however they have some practical drawbacks. Firstly, the spatial resolution is lower since the cameras pixels are spread over a wider field of view. We also note from Fig. 11.14 that the field of view is a circular region which means that nearly 25% of the rectangular image plane is effectively wasted. Secondly, outdoors images are more likely to include bright sky so the camera will automatically reduce its exposure which can result in some non-sky parts of the scene being very underexposed.

### 11.3.2 Catadioptric Camera

A catadioptric imaging system comprises both reflective and refractive elements, a mirror and a lens, as shown in Fig. 11.17a. An example catadioptric image is shown in Fig. 11.17b.

The geometry is fairly complex and is shown in Fig. 11.18. A ray is constructed from the point  $P$  to the focus of the mirror at  $O$  which is the origin of the camera system. The ray has an elevation angle of

$$\theta = \tan^{-1} \frac{Z}{X^2 + Y^2} + \frac{\pi}{2}$$

upward from the optical axis and intersects the mirror at the point  $M$ . The reflected ray makes an angle  $\psi$  with respect to the optical axis which is a function of the incoming ray angle, that is  $\psi(\theta)$ . The relationship between  $\theta$  and  $\psi$  is determined by the tangent to the mirror at the point  $M$  and is a function of the shape of the mirror. Many different mirror shapes are used for catadioptric imaging including spherical, parabolic, elliptical and hyperbolic. In general the function  $\psi(\theta)$  is non-linear but an interesting class of mirror is the equiangular mirror for which

$$\theta = \alpha\psi$$

The reflected ray enters the camera at angle  $\psi$  from the optical axis, and from the lens geometry we can write

$$r = f \tan \psi$$

which is the distance from the principal point. The polar coordinate of the image-plane point is  $p = (r, \phi)$  and the corresponding Cartesian coordinate is

$$u = r \cos \phi, v = r \sin \phi$$

where  $\phi$  is the azimuth angle

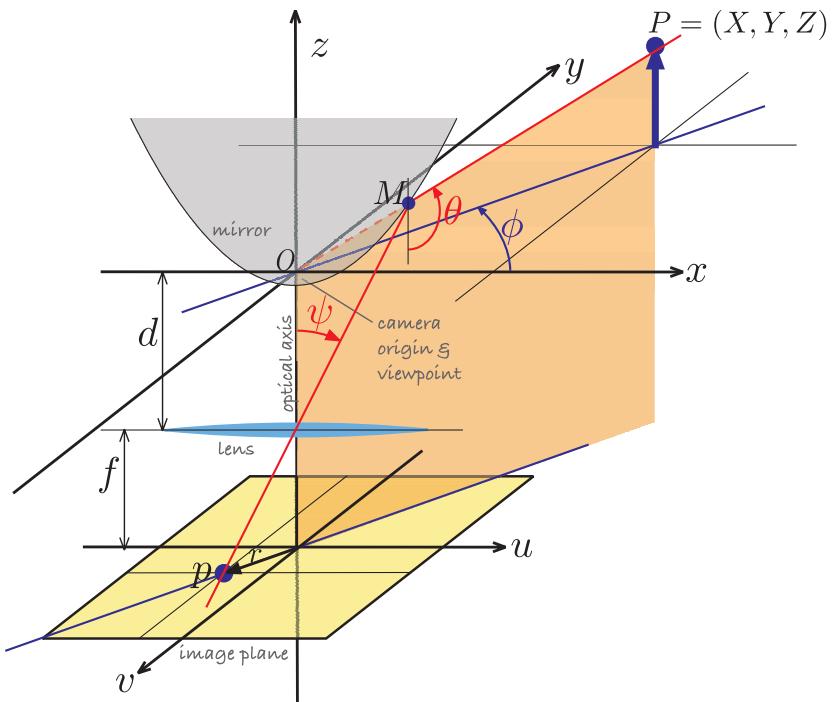
$$\phi = \tan^{-1} \frac{Y}{X}$$

In Fig. 11.18 we have assumed that all rays pass through a single focus point or viewpoint –  $O$  in this case. This is referred to as central imaging and the resulting image can be correctly transformed to a perspective image. The equiangular mirror does not meet this constraint and is therefore a non-central imaging system – the viewpoint varies with the angle of the incoming ray and lies along a short locus within the mirror known as the caustic. Conical, spherical and equiangular mirrors are all non-central. In practice the variation in the viewpoint is very small compared to the world scale and many such mirrors are well approximated by the central model.

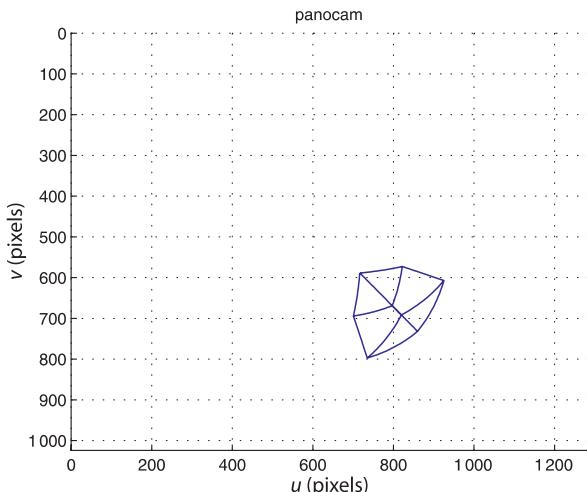
The Toolbox provides a model for catadioptric cameras. For example we can create an equiangular catadioptric camera

```
>> cam = CatadioptricCamera('name', 'panocam', ...
    'projection', 'equiangular', ...
    'maxangle', pi/4, ...
    'pixel', 10e-6, ...
    'resolution', [1280 1024])
```

which returns an instance of a `CatadioptricCamera` object which is a subclass of the Toolbox's `Camera` object and polymorphic with the `CentralCamera` class discussed earlier. The option `maxangle` specifies the maximum elevation angle  $\theta$  from which the parameters  $\alpha$  and  $f$  are determined such that the maximum elevation angle corresponds to a circle that maximally fits the image plane. The parameters can be individually specified using the options '`alpha`' and '`focal`'. Other supported

**Fig. 11.18.**

Catadioptric image formation. A ray from point  $P$  at elevation angle  $\theta$  and azimuth  $\phi$  toward  $O$  is reflected from the mirror surface at  $M$  and is projected by the lens on to the image plane at  $p$

**Fig. 11.19.**

A cube projected with an equiangular catadioptric camera

projection models include parabolic and spherical and each camera type has different options as described in the online documentation.

We create an edge-based cube model

```
>> [X,Y,Z] = mkcube(1, 'centre', [1, 1, 0.8], 'edge');
```

which we project onto the image plane

```
>> cam.mesh(X, Y, Z)
```

and the result is shown in Fig. 11.19.

Catadioptric cameras have the advantage that they can view 360° in azimuth but they also have some practical drawbacks. With fisheye lenses they share the problems of reduced spatial resolution, wasted image plane pixels and exposure control. Additionally there is a blind spot for the support that holds the mirror which is commonly a central stalk or a number of side supports.

### 11.3.3 Spherical Camera

The fisheye lens and catadioptric systems just discussed guide the light rays from a large field of view onto an image plane. Ultimately the 2-dimensional image plane is a limiting factor and it is advantageous to consider instead an image *sphere* as shown in Fig. 11.20.

The world point  $P$  is projected by a ray to the origin of the sphere. The projection is the point  $p$  where the ray intersects the surface of the sphere. If we write  $\mathbf{p} = (x, y, z)$  then

$$x = \frac{X}{R}, y = \frac{Y}{R}, \text{ and } z = \frac{Z}{R} \quad (11.17)$$

where  $R = \sqrt{X^2 + Y^2 + Z^2}$  is the radial distance to the world point. The surface of the sphere is defined by  $x^2 + y^2 + z^2 = 1$  so one of the three Cartesian coordinates is redundant. A minimal two-parameter representation for a point on the surface of a sphere  $(\phi, \theta)$  comprises the angle of colatitude

$$\theta = \sin^{-1} r, \quad \theta \in [0, \pi] \quad (11.18)$$

where  $r = \sqrt{x^2 + y^2}$ , and the azimuth angle (or longitude)

$$\phi = \tan^{-1} \frac{y}{x}, \quad \phi \in [-\pi, \pi] \quad (11.19)$$

Conversely the Cartesian coordinates for the point  $(\phi, \theta)$  are given by

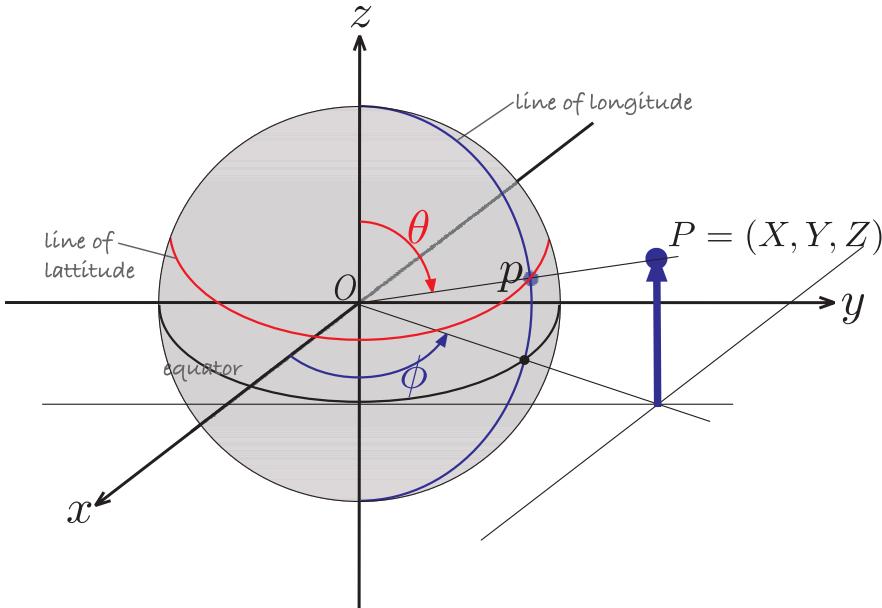
$$x = r \cos \phi, \quad y = r \sin \phi, \quad z = \cos \theta \quad (11.20)$$

where  $r = \sin \theta$ .

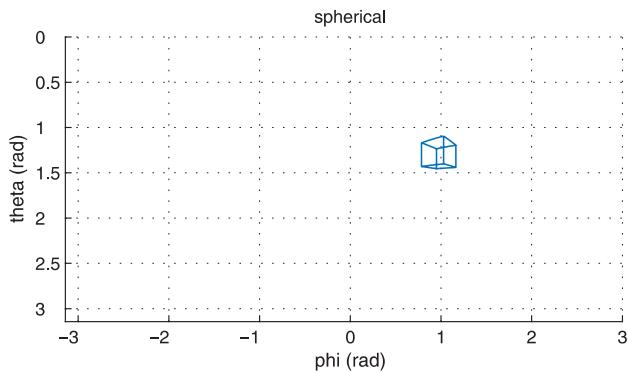
Using the Toolbox we can create a spherical camera

```
>> cam = SphericalCamera('name', 'spherical')
```

which returns an instance of a `SphericalCamera` object which is a subclass of the Toolbox's `Camera` object and polymorphic with the `CentralCamera` class discussed earlier.



**Fig. 11.20.**  
Spherical image formation. The world point  $P$  is mapped to  $p$  on the surface of the unit sphere and represented by the angles of colatitude  $\theta$  and longitude  $\phi$



**Fig. 11.21.**  
Cube projected by a spherical camera. The spherical image plane is represented in Cartesian coordinates: the horizontal and vertical axes are colatitude and longitude respectively

As previously we can create an edge-based cube model

```
>> [X,Y,Z] = mkcube(1, 'centre', [2, 3, 1], 'edge');
```

and project it onto the sphere

```
>> cam.mesh(X, Y, Z)
```

and this is shown in Fig. 11.21. To aid visualization the spherical image plane has been unwrapped into a rectangle – lines of longitude and latitude are displayed as vertical and horizontal lines respectively. The top and bottom edges correspond to the north and south poles respectively.

It is not yet possible to buy a spherical camera although prototypes have been demonstrated in several laboratories. The spherical camera is more useful as a conceptual construct to simplify the discussion of wide-angle imaging. As we show in the next section we can transform images from perspective, fisheye or catadioptric camera onto the sphere where we can treat them in a uniform manner.

## 11.4 Unified Imaging

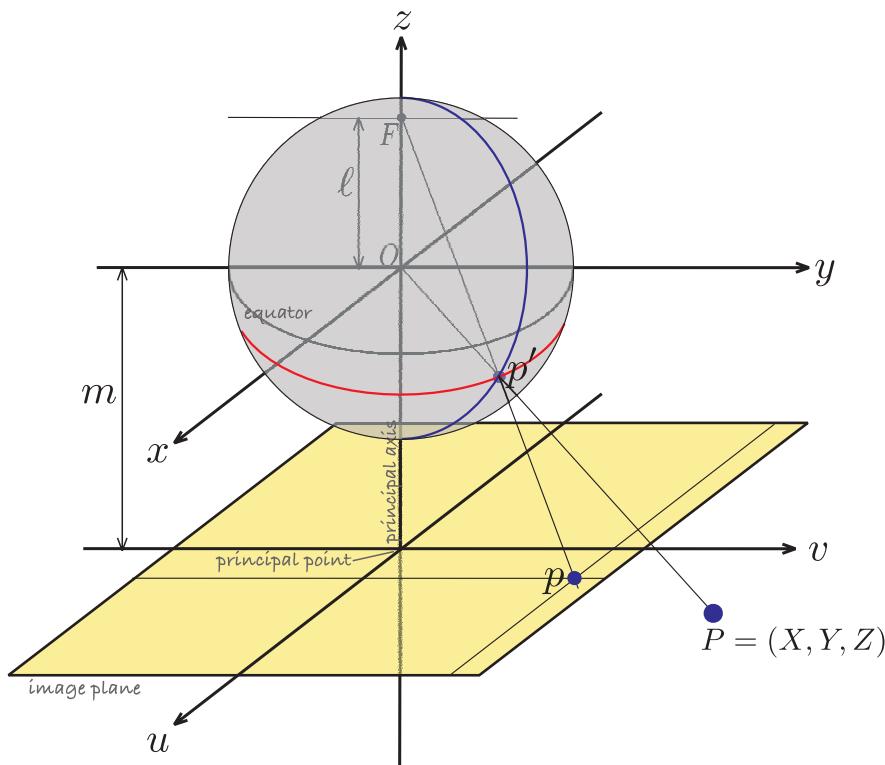
We have introduced a number of different imaging models in this chapter. Now we will discuss how to transform an image captured with one type of camera to the image that would have been captured with a different type of camera. For example, given a fisheye lens projection we will generate the corresponding projection for a spherical camera or a perspective camera. The unified imaging model provides a powerful framework to consider very different types of cameras such as standard perspective, catadioptric and many types of fisheye lens.

The unified imaging model is a two-step process and the notation is shown in Fig. 11.22. The first step is spherical projection of the world point  $P$  to the surface of the unit sphere  $p'$  as discussed in the previous section and described by Eq. 11.17 to Eq. 11.18. The view point  $O$  is the centre of the sphere which is a distance  $m$  from the image plane along its normal  $z$ -axis. The single view point implies a *central* camera.

In the second step the point  $p' = (\theta, \phi)$  is re-projected to the image plane  $p$  using the view point  $F$  which is at a distance  $\ell$  along the  $z$ -axis above  $O$ . The polar coordinates of the image plane point are  $p = (r, \phi)$  where

$$r = \frac{(\ell + m)\sin\theta}{\ell + \cos\theta} \quad (11.21)$$

The unified imaging model has only two parameters  $m$  and  $\ell$  and these are a function of the type of camera as listed in Table 11.2. For a perspective camera the two view points  $O$  and  $F$  are coincident and the geometry becomes the same as the central perspective model shown in Fig. 11.2.



**Fig. 11.22.**  
Unified imaging model of Geyer and Daniilidis (2000)

Imaging	$\ell$	$m$
Perspective	0	$f$
Stereographic	1	$f$
Fisheye	$>1$	$f$
Catadioptric (elliptical, $0 < \varepsilon < 1$ )	$\frac{2\varepsilon}{1+\varepsilon^2}$	$\frac{2\varepsilon(2p-1)}{1+\varepsilon^2}$
Catadioptric (parabolic, $\varepsilon = 1$ )	1	$2p - 1$
Catadioptric (hyperbolic, $\varepsilon > 1$ )	$\frac{2\varepsilon}{1+\varepsilon^2}$	$\frac{2\varepsilon(2p-1)}{1+\varepsilon^2}$

For catadioptric cameras with mirrors that are conics the focus  $F$  lies between the centre of the sphere and the north pole, that is,  $0 < \ell < 1$ . This projection model is somewhat simpler than the catadioptric camera geometry shown in Fig. 11.18. The imaging parameters are written in terms of the conic parameters eccentricity  $\varepsilon$  and latus rectum  $\triangleright 4p$ .

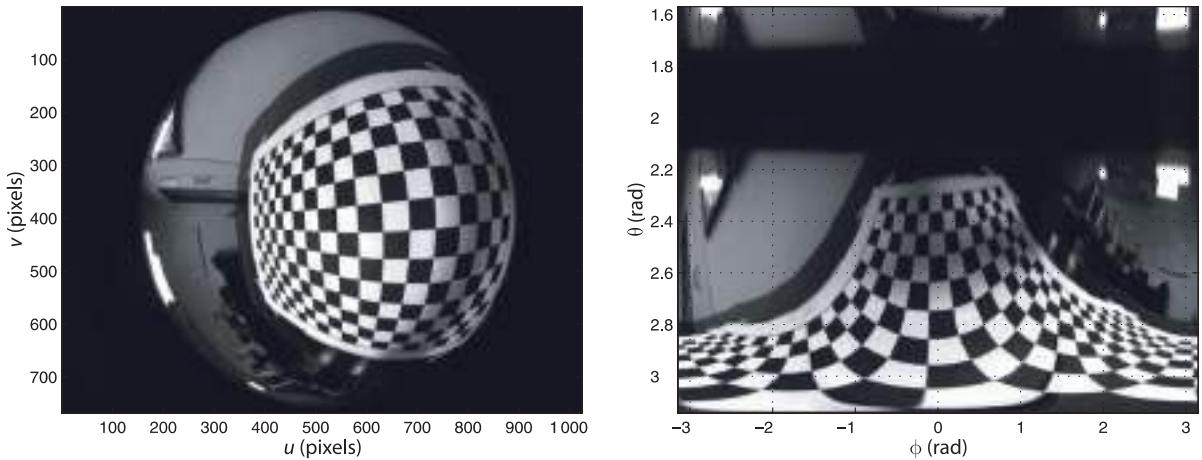
The projection with  $F$  at the north pole is known as stereographic projection and is used in many fields to project the surface of a sphere onto a plane. Many fisheye lenses are extremely well approximated by  $F$  above the north pole.

The length of a chord parallel to the directrix and passing through the focus.

#### 11.4.1 Mapping Wide-Angle Images to the Sphere

We can use the unified imaging model in reverse. Consider an image captured by a wide field of view camera such as the fisheye image shown in Fig. 11.23a. If we know the location of  $F$  then we can project each point from the image onto the sphere to create a spherical image, even though we do not have a spherical camera.

In order to achieve this inverse mapping we need to know some parameters of the camera that took the image in the first place. A common feature of images captured with



**Fig. 11.23.** Fisheye image of a planar calibration target. **a** Fisheye image (image courtesy of Peter Hansen); **b** Image warped to  $(\phi, \theta)$  coordinates

a fisheye lens or catadioptric camera is that the outer bound of the image is a circle. This circle can be found and its centre estimated quite precisely – this is the principal point. A variation of the camera calibration procedure of Sect. 11.2.4 is applied which uses corresponding world and image plane points from the planar calibration target shown in Fig. 11.23a. The calibration parameters for this particular camera have been estimated to be: principal point (528.1214, 384.0784),  $\ell = 2.7899$  and  $m = 996.4617$ .

Consider a point in the fisheye image  $p = (r, \phi)$  defined in polar coordinates relative to the principal point. The corresponding point on the sphere is  $p' = (\theta, \phi)$  where  $\theta$  is found by inverting the second projection Eq. 11.21 which gives

$$\theta = \cos^{-1} \left( \frac{(\ell + m) \sqrt{r^2(1 - \ell^2) + (\ell + m)^2} - \ell r^2}{r^2 + (\ell + m)^2} \right) \quad (11.22)$$

We will illustrate this by mapping the image of Fig. 11.23

```
>> fisheye = imread('fisheye_target.png', 'double', 'grey');
```

to the sphere using the known camera calibration parameters

```
>> u0 = 528.1214; v0 = 384.0784;
>> l=2.7899;
>> m=996.4617;
```

We will use image warping to achieve this mapping. Warping is discussed in detail in Sect. 12.6.4 but we will preview the approach here. First we define the domain of the input image

```
>> [Ui,Vi] = imeshgrid(fisheye);
```

The output domain covers the lower hemisphere

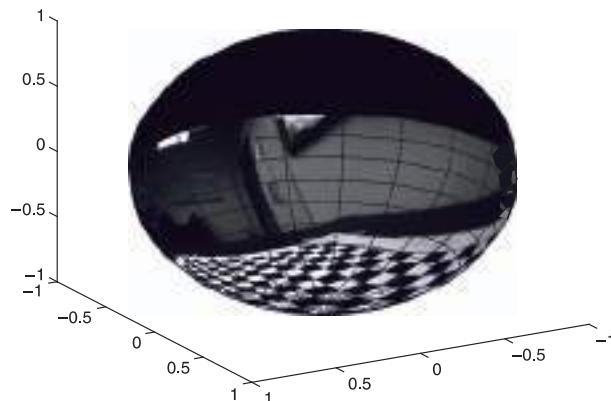
```
>> n = 500;
>> theta_range = (0:n)/n*pi/2 + pi/2;
>> phi_range = (-n:2:n)/n*pi;
>> [Phi,Theta] = meshgrid(phi_range, theta_range);
```

where  $n$  is the number of grid cells on the surface of the sphere in the colatitude and longitude directions.

For warping we require the inverse function, the input image coordinates for each point on the sphere. This is the second step of the unified imaging model Eq. 11.21 which we implement as

```
>> r = (l+m)*sin(Theta) ./ (l+cos(Theta));
```

from which the corresponding Cartesian coordinates in the input image are computed



```
>> U = r.*cos(Phi) + u0;
>> V = r.*sin(Phi) + v0;
```

We apply the warp

```
>> spherical = interp2(Ui, Vi, fisheye, U, V);
>> idisp(spherical)
```

which is shown in Fig. 11.23b with respect to colatitude and longitude coordinates. The top line of this image corresponds to perimeter of the input image, and the bottom line corresponds to the principal point.

The image is extremely distorted but the coordinate system is very convenient to texture map onto a sphere

```
>> sphere_paint(spherical, 'south')
```

and this is shown in Fig. 11.24. Using the MATLAB® figure toolbar we can rotate the sphere and look at the image from different view points. The option '**south**' indicates that the image covers only the lower or southern hemisphere, the undefined part of the spherical image is displayed as grey.

Any wide-angle image that can be expressed in terms of central imaging parameters can be similarly projected onto a sphere.

#### 11.4.2 Synthetic Perspective Images

We will finish this chapter with an example of mapping a spherical image to a perspective image. This is the second step of the unified imaging model where  $F$  is at the centre of the sphere in which case Fig. 11.22 becomes similar to Fig. 11.2. The perspective camera's optical axis is the negative  $z$ -axis of the sphere.

For this example we will use the spherical image created in the previous section. We wish to create a perspective image of  $1000 \times 1000$  pixels and with a field-of-view of  $45^\circ$ . The field of view can be written in terms of the image width  $W$  and the unified imaging parameter  $m$  as

$$\theta_{\text{FOV}} = 2 \tan^{-1} \frac{W}{2m}$$

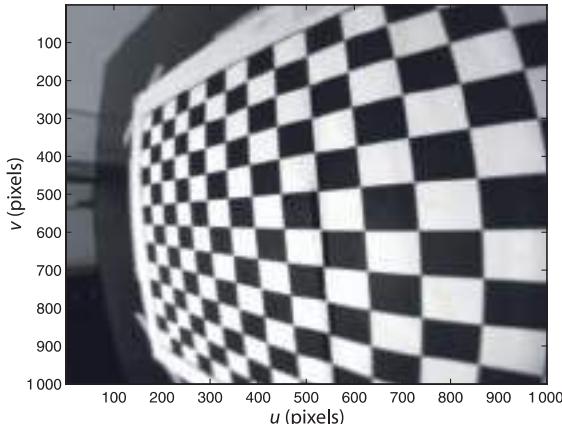
For a  $45^\circ$  field-of-view we require

```
>> W = 1000;
>> m = W / 2 / tan(45/2*pi/180)
m =
1.2071e+03
```

For perspective projection we require

```
>> l = 0;
```

**Fig. 11.24.**  
Fisheye image mapped to the unit sphere. We can clearly see the planar grid lying on a table, the ceiling light and other linear features in the room

**Fig. 11.25.**

Perspective projection of spherical image Fig. 11.24 with a field of view of 45°. Note that the lines on the chessboard are now straight

and we also require the principal point to be in the centre of the image

```
>> u0 = w/2; v0 = w/2;
```

The domain of the output image will be

```
>> [Uo,Vo] = meshgrid(0:W-1, 0:W-1);
```

The polar coordinate  $(r, \phi)$  of each point in the output image is

```
>> r = sqrt((Uo-u0).^2 + (Vo-v0).^2);
>> phi = atan2((Vo-v0), (Uo-u0));
```

and the corresponding spherical coordinates  $(\phi, \theta)$  are

```
>> Phi_o = phi;
>> Theta_o = pi - acos( ((l+m)*sqrt(r.^2*(l-l.^2) + (l+m).^2) -
l.^2) ./ (r.^2 + (l+m).^2) );
```

We now warp from spherical coordinates to the perspective image plane

```
>> perspective = interp2(Phi, Theta, spherical, Phi_o, Theta_o);
```

and the result

```
>> idisp(perspective)
```

is shown in Fig. 11.25. This is the view from a perspective camera at the centre of the sphere looking down through the south pole. We see that the lines on the chessboard calibration target are now straight as we would expect from a perspective image. Importantly we can only create this perspective image for a *small* field of view.

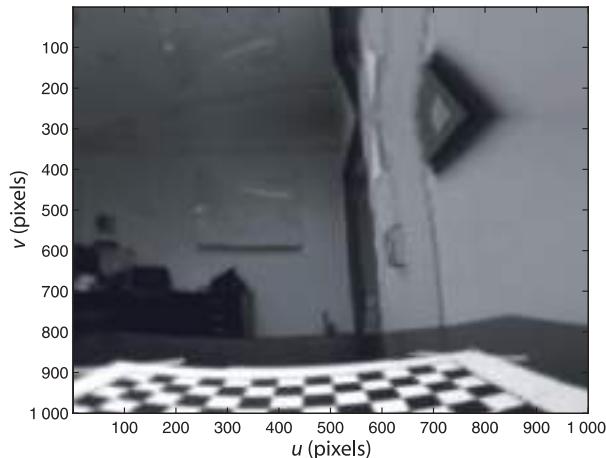
Of course we are not limited to just *looking* along the negative z-axis of the sphere. In Fig. 11.24 we can see some other features of the room such as door, a whiteboard and some ceiling lights. We can point our virtual perspective camera in their direction by rotating the spherical image

```
>> spherical = sphere_rotate(spherical, trotz(1.2)*trotz(-1.5));
```

so that the negative z-axis now points toward the distant wall. Repeating the warp process we obtain the result shown in Fig. 11.26 in which we can see the ceiling light and a whiteboard.►

The original wide-angle image contains a lot of detail though it can be hard to see because of the distortion. After mapping the image to the sphere we can create a virtual perspective camera view (where straight lines in the world are straight) along any line of sight in the hemisphere. This is only possible if the original image was taken with a central camera that has a single viewpoint. In theory we cannot create a perspective image from a non-central wide-angle image but in practice if the caustic is small the parallax errors introduced into the perspective image will be negligible.

Note that the top part of the image is a reflection about a line that that corresponds to the edge of the original field of view.



**Fig. 11.26.**  
Perspective projection of spherical image Fig. 11.24 with a field of view of 45°. This view is looking toward the far wall and ceiling

## 11.5 Wrapping Up

We have discussed the first steps in the computer vision process – the formation of an image of the world and its conversion to an array of pixels which comprise a digital image. The images with which we are familiar are perspective projections of the world in which 3 dimensions are compressed to 2 dimensions. This leads to ambiguity about object size – a large object in the distance looks the same as a small object that is close. Straight lines and conics are unchanged by this projection but shape distortion occurs – parallel lines can appear to converge and circles can appear as ellipses. We have modeled the perspective projection process and described it in terms of eleven parameters – intrinsic and extrinsic. Geometric lens distortion adds additional lens parameters. Camera calibration is the process of estimating these parameters and two approaches have been introduced. We also discussed pose estimation where the pose of an object of known geometry can be estimated from a perspective projection obtained using a calibrated camera.

Perspective images are limited in their field of view and we discussed several wide-angle imaging systems including the fisheye lens camera and the catadioptric camera. We also discussed the ideal wide-angle camera, the spherical camera, which is currently still a theoretical construct. However it can be used as an intermediate representation in the unified imaging model which provides one model for almost all camera geometries. We used the unified imaging model to convert a fisheye camera image to a spherical image and then to a perspective image along a specified view axis.

In this chapter we treated imaging as a problem of pure geometry with a small number of world points or line segments. In the next chapter we will discuss the acquisition and processing of images sourced from files, cameras and the web.

## Further Reading

Most computer vision textbooks such as Szeliski (2011), Forsyth and Ponce (2002) and Gonzalez and Woods (2008) provide coverage of the topics in this chapter. Understanding the geometry of the world from images is the science of photogrammetry. The techniques were pioneered by the French scientist Aimé Laussedat in the 1850s for balloon-based aerial map making. The Manual of Photogrammetry (Slama 1980) provides comprehensive and definitive coverage of the field including history, theory and applications of aircraft and satellite imagery. The revised classic textbook by DeWitt and Wolf (2000) is a thorough and readable introduction to photogrammetry. Photogrammetry is normally concerned with making maps from images acquired at great

distance but the sub-field of close-range or terrestrial photogrammetry is concerned with camera to object distances less than 100 m which is directly relevant to robotics. Many topics in geometric computer vision are also considered by the photogrammetric community, but different language is used. For example camera calibration is known as camera resectioning, and pose estimation is known as space resectioning.

The homogenous transformation calibration (Sutherland 1974) approach of Sect. 11.2.1 is also known as the direct linear transform (DLT) in the photogrammetric literature. The Toolbox implementation `camcalib` requires that the centroids of the calibration markers have already been determined which is a non-trivial problem (Corke 1996b, § 4.2). It also cannot estimate lens distortion. Wolf (1974) describes extensions to the linear camera calibration with models that include up to 18 parameters and suitable non-linear optimization estimation techniques. A more concise description of non-linear calibration is provided by Forsyth and Ponce (2002). Hartley and Zisserman (2003) describe how the linear calibration model can be obtained using features such as lines within the scene.

There are a number of good camera calibration toolboxes available on the web. The MATLAB® Toolbox, discussed in Sect. 11.2.4, is by Jean-Yves Bouguet and available from [http://www.vision.caltech.edu/bouguetj/calib\\_doc](http://www.vision.caltech.edu/bouguetj/calib_doc). It has extensive online documentation and includes example calibration images which were used in Sect. 11.2.4. The MATLAB® Toolbox by Janne Heikkilä is available at <http://www.ee.oulu.fi/~jth/calibr/> and works for planar or 3D targets with circular dot features and estimates lens distortion.

Pose estimation is a classic and hard problem in computer vision and for which there exists a very large literature. The approaches can be broadly divided into analytic and iterative solutions. Assuming that lens distortion has been corrected the analytic solutions for three and four non-collinear points are given by Fischler and Bolles (1981), DeMenthon and Davis (1992) and Horaud et al. (1989). Typically multiple solutions exist but for four coplanar points there is a unique solution. Six or more points always yield unique solutions, as well as the intrinsic camera calibration parameters. Iterative solutions were described by Rosenfeld (1959) and Lowe (1991). A more recent discussion based around the concept of bundle adjustment is provided by Triggs et al. (2000). The pose estimation in the Toolbox is a wrapper around an efficient non-iterative perspective n-point pose estimator described by Lepetit et al. (2009) and available at <http://cvlab.epfl.ch/software/EPnP>. Pose estimation requires a geometric model of the object and such computer vision approaches are known as *model-based vision*. An interesting historical perspective on model-based vision is the 1987 video by the late Joe Mundy which is available at <http://www.archive.org/details/JosephMu1987>.

There is recent and growing interest in wide-angle viewing systems and today good quality lightweight fisheye lenses and catadioptric camera systems are available. Nayar (1997) provides an excellent motivation for, and introduction to, wide-angle imaging. A very useful online resource is the catadioptric sensor design page at <http://www.math.drexel.edu/~ahicks/design> and a page of links to research groups, companies and workshops at <http://www.cis.upenn.edu/~kostas/omni.html>. Equiangular mirror systems were described by Chahl and Srinivasan (1997) and Ollis et al. (1999). Nature's solution, the reflector-based scallop eye is described in Colicchia et al. (2009). A number of workshops on Omnidirectional Vision have been held, starting in 2000, and their proceedings are a useful introduction to the field. The book of Daniilidis and Klette (2006) is a collection of papers on non-perspective imaging and Benosman and Kang (2001) is another, earlier, published collection of papers. Some information is available through CVonline at <http://homepages.inf.ed.ac.uk/rbf/CVonline> in the section *Sensors and their Properties*.

A MATLAB® Toolbox for calibrating wide-angle cameras by Davide Scaramuzza is available at [http://asl.epfl.ch/~scaramuz/research/Davide\\_Scaramuzza\\_files/Research/OcamCalib\\_Tutorial.htm](http://asl.epfl.ch/~scaramuz/research/Davide_Scaramuzza_files/Research/OcamCalib_Tutorial.htm). It is inspired by, and similar in usage, to Bouguet's Toolbox for perspective cameras. Another MATLAB® Toolbox, by Juho Kannala, handles wide angle central cameras and is available at <http://www.ee.oulu.fi/~jkannala/calibration>.

The unified imaging model was introduced by Geyer and Daniilidis (2000) in the context of catadioptric cameras. Later it was shown (Ying and Hu 2004) that many fisheye cameras can also be described by this model. The fisheye calibration of Sect. 11.5.1 was described by Hansen et al. (2010) and estimates  $\ell$  and  $m$  rather than a polynomial function  $r(\theta)$  as does Scaramuzza's Toolbox.

### Camera Classes

The Toolbox camera classes `CentralCamera`, `FishEyeCamera` and `SphericalCamera` are all derived from the abstract superclass `Camera`. Common methods of all classes are shown in Table 11.3.

The camera view has similar behaviour to a MATLAB® figure. By default `plot` and `mesh` will redraw the camera's view. If no camera view exists one will be created. The methods `clf` and `hold` are analogous to the MATLAB® commands `clf` and `hold`.

The constructor of all camera classes accepts a number of option arguments which are listed in Table 11.4. Specific camera subclasses have unique options which are described in the online documentation. With no arguments the default `CentralCamera` parameters are for a  $1\,024 \times 1\,024$  image, 8 mm focal length lens and  $10\,\mu\text{m}$  square pixels. If the principal point is not set explicitly it is assumed to be in the middle of the image plane.

Method	Description
<code>p = cam.project(P)</code>	Project the world point <code>P</code> to the image plane <code>p</code>
<code>cam.plot(P)</code>	Plot the world points defined by the columns of <code>P</code>
<code>cam.mesh(X, Y, Z)</code>	Plot the mesh defined by <code>X</code> , <code>Y</code> and <code>Z</code>
<code>cam.showpose(T)</code>	Show the camera at specified pose
<code>cam.clf</code>	Clear the current camera view
<code>cam.hold</code>	Hold the current camera view, future calls to <code>plot</code> add to the camera view
<code>cam.rpy(r, p, y)</code>	Set the camera transform to specified roll-pitch-yaw angles
<code>cam.name</code>	Property: name of camera
<code>cam.Tcam</code>	Property: default camera transform (read and write)

**Table 11.3.**  
Common methods for all  
Toolbox camera classes

Option	Description
<code>'name', name</code>	The name of the camera which is displayed in the window's title bar
<code>'resolution', npix</code>	The dimensions of the image. <code>npix</code> is a scalar for a square image or a 2-vector
<code>'centre', pp</code>	The coordinate of the principal point
<code>'pixel', rho</code>	The dimensions of the pixel. <code>rho</code> is a scalar for a square pixel or a 2-vector
<code>'noise', sigma</code>	The standard-deviation of Gaussian noise added to the image-plane coordinates
<code>'pose', T</code>	The default pose of the camera. Default is as shown in Fig. 11.2
<code>'image', im</code>	Set camera image-plane dimensions according to the image dimensions and display the image

**Table 11.4.**  
Common options for camera  
class constructors

---

### Exercises

1. Create a central camera and a cube target and visualize it for different camera and cube poses.
2. Write a script to fly the camera in an orbit around the cube, always facing toward the centre of the cube.
3. Write a script to fly the camera through the cube.
4. Create a central camera with lens distortion and which is viewing a  $10 \times 10$  planar grid of points. Vary the distortion parameters and see the effect this has on the shape of the projected grid. Create pincushion and barrel distortion.
5. Repeat the homogeneous camera calibration exercise of Sect. 11.2.1 and investigate the effect of the number of calibration points, noise and camera distortion on the calibration residual and estimated target pose.
6. Determine the solid angle for a rectangular pyramidal field of view that subtends angles  $\theta_h$  and  $\theta_v$ .
7. Do example 1 from Bouguet's Camera Calibration Toolbox.
8. Calibrate the camera on your computer.
9. For the camera calibration matrix decomposition example (Sect. 11.2.2) determine the roll-pitch-yaw orientation error between the true and estimated camera pose.
10. Pose estimation (Sect. 11.2.3)
  - a) Repeat the pose estimation exercise for different object poses (closer, further away).
  - b) Repeat for different levels of camera noise.
  - c) What happens as the number of points is reduced?
  - d) Does increasing the number of points counter the effects of increased noise?
  - e) Change the intrinsic parameters of the camera `cam` before invoking the `estpose` method. What is the effect of changing the focal length and the principal point by say 5%.
11. Repeat exercises 2 and 3 for the fisheye camera and the spherical camera.
12. With reference to Fig. 11.18 derive the function  $\psi(\theta)$  for a parabolic mirror.
13. With reference to Fig. 11.18 derive the equation of the equi-angular mirror  $z(x)$  in the  $xz$ -plane.

# 12

# Image Processing



Image processing is a computational process that transforms one or more input images into an output image. Image processing is frequently used to enhance an image for *human* viewing or interpretation, for example to improve contrast. Alternatively, and of more interest to robotics, it is the foundation for the process of feature extraction which will be discussed in much more detail in the next chapter.

An image is a rectangular array of picture elements (pixels) so we will use a MATLAB® matrix to represent an image in the workspace. This allows us to use MATLAB's powerful and efficient armoury of matrix operators and functions.

We start in Sect. 12.1 by describing how to load images into MATLAB® from sources such as files (images and movies), cameras and the internet. We then discuss various classes of image processing algorithms. These algorithms operate pixel-wise on a single image or a pair of images, or on local groups of pixels within an image and we refer to these as monadic, diadic, and spatial operations respectively. Monadic and diadic operations are covered in Sect. 12.2 and 12.3. Spatial operators are described in Sect. 12.4 and include operations such as smoothing, edge detection, and template matching. A closely related technique is shape-specific filtering or mathematical morphology and this is described in Sect. 12.5. Finally in Sect. 12.6 we discuss shape changing operations such as cropping, shrinking, expanding, as well as more complex operations such as rotation and generalized image warping.

Robots will always gather imperfect images of the world due to noise, shadows, reflections and uneven illumination. In this chapter we discuss some fundamental tools and “tricks of the trade” that can be applied to real-world images.

## 12.1 Obtaining an Image

Today digital images are ubiquitous. We obtain them quickly and easily using our digital cameras, our phones and our laptops, and have personal collections of thousands or tens of thousands of images. Beyond our personal collections are massive online collection of digital images such as Google Images, Picasa or Flickr which also have metadata describing the image content and increasingly its geographic location. Images of Earth from space, the Moon and Mars are also available. We also have access to live image streams from other people's cameras – there are tens of thousands of webcams around the world capturing images and broadcasting them on the internet.

### 12.1.1 Images from Files

We start with images stored in files since it is very likely that you already have lots of images stored on your computer. In this chapter we will work with some images provided with the Toolbox, but you can easily substitute your own images. We import an image into the MATLAB® workspace using the Toolbox function `imread`

```
>> street = imread('street.png');
which returns a matrix
```

```
>> about(street)
street [uint8] : 851x1280 (1089280 bytes)
```

which has 851 rows and 1280 columns. We normally describe the dimensions of an image in terms of its width  $\times$  height, so this would be a  $1280 \times 851$  pixel image.

In Chap. 11 we wrote the coordinates of a pixel as  $(u, v)$  which are the horizontal and vertical coordinates respectively. In MATLAB® this is the matrix element  $(v, u)$  – note the reversal of coordinates. Note also that the top-left pixel is  $(1, 1)$  in MATLAB® not  $(0, 0)$ .

This image was read from a file called `street.png` which is in portable network graphics (PNG) format – a lossless compression format► widely used on the internet. The function `imread` searches for the image in the current folder, and then in each folder along your MATLAB® path.► This particular image has no color, it is a greyscale or monochromatic image.

The `about` command used above indicates that the matrix `street` belongs to the class `uint8` – the elements of the matrix are unsigned 8-bit integers in the interval  $[0, 255]$ . The elements are referred to as pixel values or grey values and are proportional to the luminance of that point in the original scene. For this 8-bit image the pixel values vary from 0 (darkest) to 255 (brightest). For example the pixel at image coordinate  $(300, 200)$  is

```
>> street(200,300)
ans =
42
```

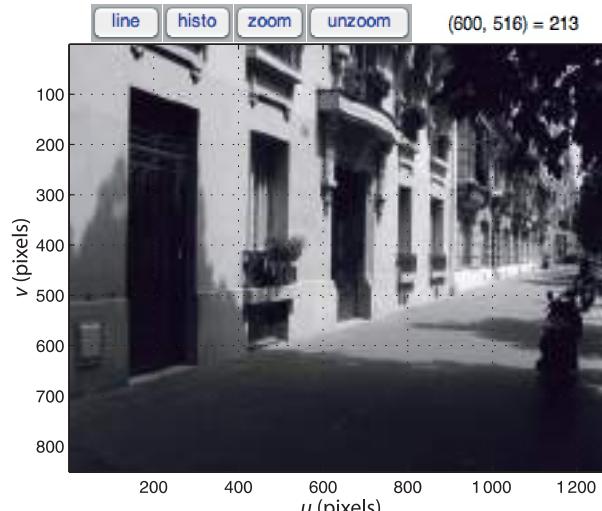
which is quite dark – the pixel corresponds to a point in the closest doorway.

For some image processing operations that we will consider later it is useful to consider the pixel values as floating point numbers. In this case each pixel is an 8-byte MATLAB® double precision number in the range  $[0, 1]$ . We can convert a `uint8` class image to a floating point image by

```
>> street_d = idouble(street);
>> about(street_d)
street_d [double] : 851x1280 (8714240 bytes)
```

Alternatively we can specify this as an option when we load the image

```
>> street_d = imread('street.png', 'double');
```



Lossless means that the compressed image, when uncompressed, will be exactly the same as the original image.

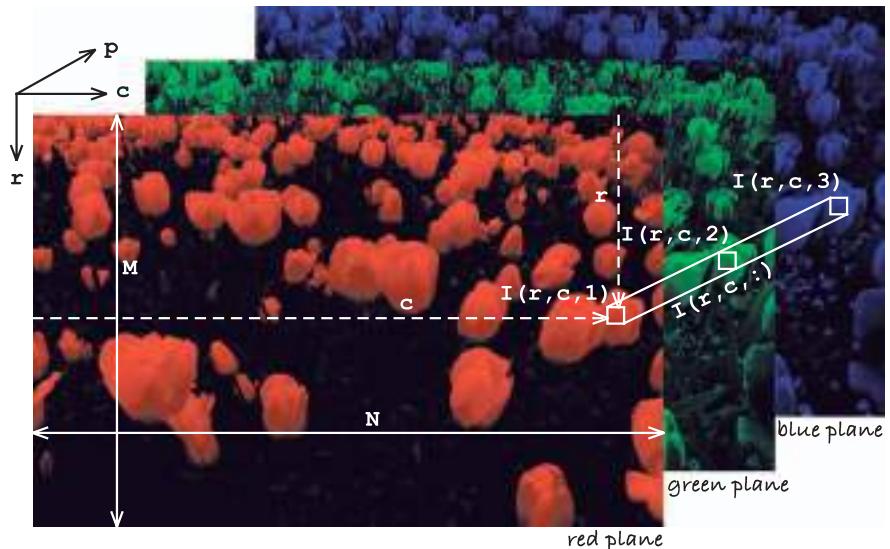
The example images are kept within the `images` folder of the Machine Vision Toolbox distribution which is automatically searched by the `imread` function.

**Fig. 12.1.**  
The `idisp` image browsing window. The top right shows the coordinate and value of the last pixel clicked on the image. The buttons at the top left allow the pixel values along a line to be plotted, a histogram to be displayed, or the image to be zoomed

The **dynamic range** of a sensor is the ratio of its largest value to its smallest value. For images it is useful to express the  $\log_2$  of this ratio which makes it equivalent to the photographic concepts of stops or exposure value. Each photosite contains a charge well in which photon-generated electrons are captured during the exposure period (see page 260). The charge well has a finite capacity before the photosite saturates and this defines the maximum value. The minimum number of electrons is not zero but a finite number of thermally generated electrons.

An 8-bit image has a dynamic range of around 8 stops, a high-end 10-bit camera has a range of 10 stops, and negative film is perhaps in the range 10–12 stops but is quite non-linear.

At a particular state of adaptation, the human eye has a range of 10 stops, but the total adaptation range is an impressive 20 stops. This is achieved by using the iris and slower (tens of minutes) chemical adaptation of the sensitivity of rod cells. Dark adaptation to low luminance is slow, from bright Sun to a dark room takes many minutes. Adaptation from dark to bright is faster but sometimes painful.



**Fig. 12.2.**

Color image shown as a 3-dimensional structure with dimensions: row, column, and color plane

A tool that we will see and use a lot to display an image is `idisp`

```
>> idisp(street)
```

which displays the matrix as an image and allows interactive inspection of pixel values as shown in Fig. 12.1. Clicking on a pixel will display the pixel coordinate and its grey value in the top right of the window. The image can be zoomed (and unzoomed) and we can display a histogram or the intensity profile along a line between any two selected points. It has many options and these are described in the online documentation.

We can just as easily load a color image

```
>> flowers = iread('flowers8.png');
>> about(flowers)
flowers [uint8] : 426x640x3 (817920 bytes)
```

which is a 3-dimensional matrix of uint8 values as shown in Fig. 12.2. The color of each pixel is represented as an RGB tristimulus which is a 3-vector. For example the pixel at (318, 276)

```
>> pix = flowers(276,318,:)
ans(:,:,1) =
    57
ans(:,:,2) =
    91
ans(:,:,3) =
   198
```

has a tristimulus value (57, 91, 198). This pixel corresponds to one of the small blue flowers and has a large blue component. We can display the image and examine it interactively using `idisp` and clicking on a pixel will display its tristimulus value.

The variable `pix` has been displayed by MATLAB® in an unusual and non-compact manner. This is because the pixel value is

```
>> about(pix)
pix [uint8] : 1x1x3 (3 bytes)
```

a  $1 \times 1 \times 3$  matrix. The first two dimensions are called singleton dimensions and we can *squeeze* them out

```
>> squeeze(pix)'
ans =
    57    91   198
```

which results in a more familiar 3-vector.

The third dimension of the image shown in Fig. 12.2 is known as the color plane index. For example

```
>> idisp( flowers(:,:,1) )
```

would display the red color plane as a greyscale image that shows the red stimulus at each pixel. The index 2 or 3 would select the green or blue plane respectively.

The tristimulus values are of type `uint8` in the range [0, 255] but the image can be converted to double precision values in the range [0, 1] using the '`double`' option to `iread` or by applying the function `idouble` to the integer color image, just as for a greyscale image. The option '`grey`' ensures that a greyscale image is returned irrespective of whether or not the file contains a color image.►

The `iread` function can also accept a wildcard filename allowing it to load a sequence of files. For example

```
>> seq = iread('seq/*.png');
>> about(seq)
seq [uint8] : 512x512x9 (2359296 bytes)
```

loads nine images in PNG format from the folder `seq`. The result is a 3-dimensional matrix and the last index represents the image number within the sequence. That is `seq(:,:,k)` is the  $k^{\text{th}}$  image in the sequence and is a  $512 \times 512$  greyscale image. In terms of Fig. 12.2 the images in the sequence extend in the `p` direction. If the images were color then the result would be a 4-dimensional matrix where the last index represents the image number within the sequence, and the third index represents the color plane.

If `iread` is called with no arguments a file browsing window pops up allowing navigation through the file system to find the image. The function also accepts a URL allowing it to load an image, but not a sequence, from the web. The function can read most common image file formats including JPEG, TIFF, GIF, PNG, PGM, PPM, PNM. Other options for `iread` are described in the online documentation.

Using ITU Rec. 709 by default. See also the function `imono`.

A very large number of **image file formats** have been developed and are comprehensively catalogued at [http://en.wikipedia.org/wiki/Image\\_file\\_formats](http://en.wikipedia.org/wiki/Image_file_formats). The most popular is JPEG which is used for digital cameras and webcams. TIFF is common in many computer systems and often used for scanners. PNG and GIF are widely used on the web. The internal format of these files are complex but a large amount of good quality open-source software exists in a variety of languages to read and write such files. MATLAB® is able to read many of these image file formats.

A much simpler set of formats, widely used on Unix systems, are PBM, PGM and PPM (generically PNM) which represent images without compression, and optionally as readable ASCII text. A host of open-source tools such as ImageMagick provide format conversions and image manipulation under Unix, MacOS X and Windows.

### 12.1.2 Images from an Attached Camera

Most laptop computers today have a builtin camera for video conferencing. For computers without a builtin camera an external camera can be easily attached via a USB or FireWire connection. The means of accessing a camera is operating system specific and the Toolbox provides a simple interface to a camera for MacOS, Linux and Windows. A list of all attached cameras and their capability can be obtained by

```
>> VideoCamera('?)
```

We open a particular camera

```
>> cam = VideoCamera('name')
```

which returns an instance of a `VideoCamera` object that is a subclass of the `ImageSource` class. If name is not provided the first camera found is used. The constructor accepts a number of additional arguments such as '`grey`' which ensures that the returned image is greyscale irrespective of the camera type, and '`framerate`' which sets the number of frames captured per second.

The dimensions of the image returned by the camera are given by the `size` method

```
>> cam.size()
```

and an image is obtained using the `grab` method

```
>> im = cam.grab();
```

which waits until the next frame becomes available.◀

Since the frames are generated at a rate of  $R$  per second as specified by the '`framerate`' option, then the worst case wait is uniformly distributed in the interval  $[0, 1/R]$ .

### 12.1.3 Images from a Movie File

In Sect. 12.1.1 we loaded an image sequence into memory where each image came from a separate image file. More commonly image sequences are stored in a movie file format such as MPEG4 or AVI and it may not be practical or possible to keep the whole sequence in memory.

The Toolbox supports reading frames from a movie file stored in any of the popular formats such as AVI, MPEG and MPEG4. For example we can open a movie file

```
>> cam = Movie('traffic_sequence.mpg');
720 x 576 @ 2.999970e+01 fps
350 frames
```

JPEG employs *lossy* compression to reduce the size of the file. This means that the decompressed image isn't quite the same as the original image. It exploits limitations of the human eye and discards information that won't be noticed such as very small color changes (which are perceived less accurately than small changes in brightness) and fine texture. It is very important to remember that JPEG is intended for compressing images that will be *viewed by humans*. The loss of color detail and fine texture may be problematic for computer algorithms that analyze images.

JPEG was designed to work well for natural scenes but it does not do so well on lettering and line drawings with high spatial-frequency content. The degree of lossiness can be varied by adjusting the so-called quality factor which allows a tradeoff between image quality and file size. JPEG can be used for greyscale or color images.

What is commonly referred to as a JPEG file, often with an extension of `.jpg` or `.jpeg`, is more correctly a JPEG JFIF file. JFIF is the format of the file that holds a JPEG-compressed image as well as metadata. EXIF file format (Exchangeable Image File Format) is a standard for camera related metadata such as camera settings, time, location and so on. This metadata can be retrieved as a second output argument to `iread` as a cell array, or by using a command-line utility such as `exiftool` (<http://www.sno.phy.queensu.ca/~phil/exiftool>). See the Independent JPEG group web site <http://www.ijg.org> for more details.

which returns a `Movie` object that is an instance of a subclass of the `ImageSource` class and therefore polymorphic with the `VideoCamera` class just described. This movie has 350 frames and was captured at 30 frames per second.

The size of each frame within the movie is

```
>> cam.size()
ans =
    720    576
```

and the next frame is read from the movie file by

```
>> im = cam.grab();
>> about(im)
im [uint8] : 576x720x3 (1244160 bytes)
```

which is a  $720 \times 576$  color image. With these few primitives we can write a very simple movie player

```
1 while 1
2     im = cam.grab;
3     if isempty(im), break; end
4     image(im)
5 end
```

where the test at line 3 is to detect the end of file, in which case `grab` returns an empty matrix.

The methods `nframes` and `framerate` provide the total number of frames and the number of frames per second. The methods `skiptotime` and `skiptoframe` provide an ability to move to desired frames within the movie.

#### 12.1.4 Images from the Web

The term web camera has come to mean *any* USB or Firewire connected local camera but here we use it to refer to an *internet* connected camera that runs a web server that can deliver images on request. There are tens of thousands of these web cameras around the world that are pointed at scenes from the mundane to the spectacular. Given the URL of a webcam from Axis Communications<sup>▶</sup> we can acquire an image from a camera anywhere in the world and place it in a matrix in our MATLAB® workspace.

For example we can connect to a camera at Dartmouth College in New Hampshire

```
>> cam = AxisWebCamera('http://wc2.dartmouth.edu');
```

which returns an `AxisWebCamera` object which is an instance of a subclass of the `ImageSource` class and therefore polymorphic with the `VideoCamera` and `Movie` classes previously described.

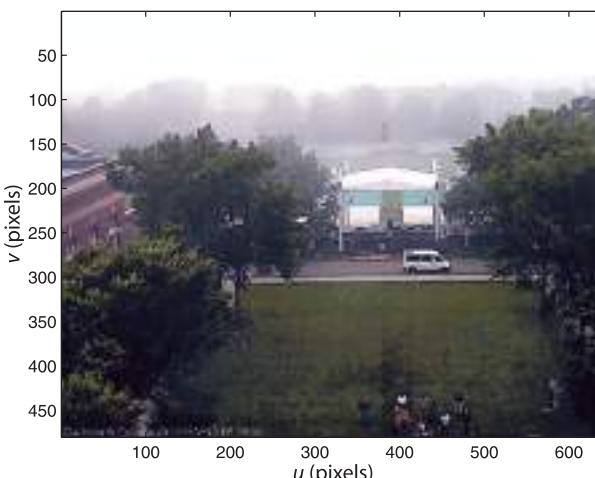
The image size in this case is

```
>> cam.size()
ans =
    480    640
```

Webcams support a variety of options that can be embedded in the URL and there is no standard for these. This function works only with recent webcams from Axis Communications.

**Aspect ratio** is the ratio of an image's width to its height. It varies widely across different imaging and display technologies. For 35 mm film it is 3:2 (1.5) which matches a  $4 \times 6"$  (1.5) print. Other print sizes have different aspect ratios:  $5 \times 7"$  (1.4), and  $8 \times 10"$  (1.25) which require cropping the vertical edges of the image in order to fit.

TV and early computer monitors used 4:3 (1.33), for example the ubiquitous  $640 \times 480$  format. HDTV has settled on 16:9 (1.78). Modern digital SLR cameras typically use 1.81 which is close to the ratio for HDTV. In movie theatres very-wide images are preferred with aspect ratios of 1.85 or even 2.39. CinemaScope was developed by 20<sup>th</sup> Century Fox from the work of Henri Chrétien in the 1920s. An anamorphic lens on the camera compresses a wide image into a standard aspect ratio in the camera, and the process is reversed at the projector.

**Fig. 12.3.**

An image from the Dartmouth University webcam which looks out over the main college green

and the next image is obtained by

```
>> im = cam.grab();
```

which returns a color image such as the one shown in Fig. 12.3. Webcams are configured by their owner to take pictures periodically, anything from once per second to once per minute. Repeated access will return the same image until the camera takes its next picture.

### 12.1.5 Images from Code

When debugging an algorithm it can be very helpful to start with a perfect and simple image before moving on to more challenging real-world images. The Toolbox function `testpattern` generates simple images with a variety of patterns including lines, grids of dots or squares, intensity ramps and intensity sinusoids. For example

```
>> im = testpattern('rampx', 256, 2);
>> im = testpattern('siny', 256, 2);
>> im = testpattern('squares', 256, 50, 25);
>> im = testpattern('dots', 256, 256, 100);
```

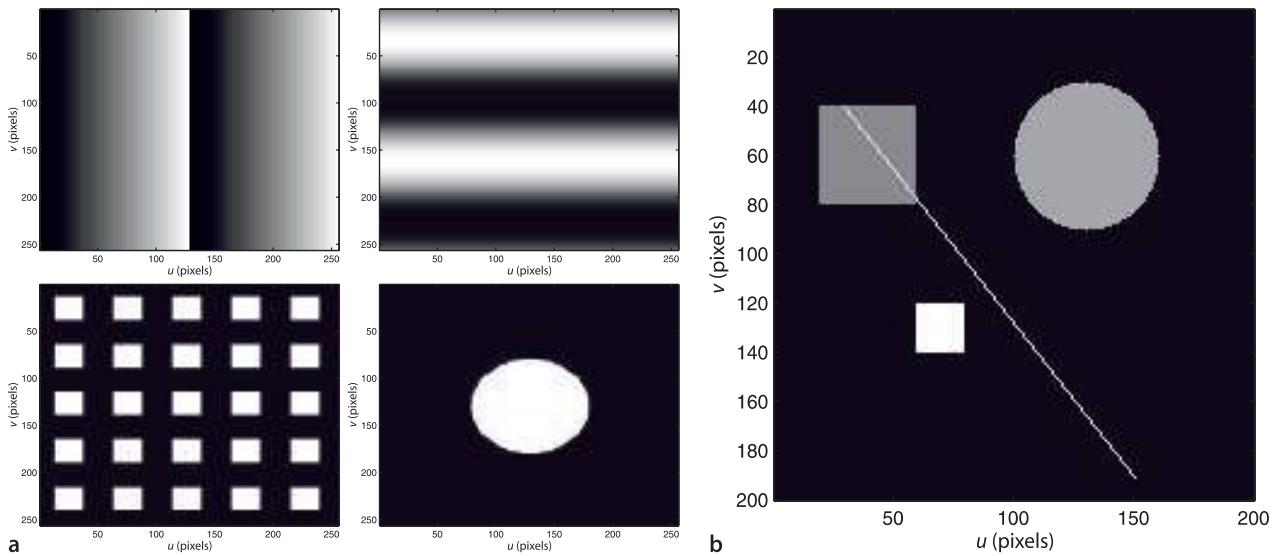
are shown in Fig. 12.4a. The second argument is the size of the created image, in this case they are all  $256 \times 256$  pixels, and the remaining arguments are specific to the type of pattern requested. See the online documentation for details.

We can also construct an image from simple graphical primitives. First we create a blank *canvas* containing all black pixels (pixel value of zero)

```
>> canvas = zeros(200, 200);
```

**Video file formats.** Just as for image files there are a large number of different file formats for videos. The most common formats are MPEG and AVI. It is important to distinguish between the format of the file (the *container*), technically AVI is a file format, and the type of compression (the *codec*) used on the images within the file.

MPEG and AVI format files can be converted to a sequence of frames as individual files using tools such as FFmpeg and `convert` from the ImageMagick suite. The individual frames can then be loaded individually into MATLAB® for processing using `iread`. The Toolbox `Movie` class provides a more convenient way to read frames from common movie formats without having to first convert the movies to a set of individual frames.



and then we create two squares

```
>> sq1 = 0.5 * ones(40, 40);
>> sq2 = 0.9 * ones(20, 20);
```

The first has pixel values of 0.5 (medium grey) and is  $40 \times 40$ . The second is smaller (just  $20 \times 20$ ) but brighter with pixel values of 0.9. Now we can paste these onto the canvas

```
>> canvas = ipaste(canvas, sq1, [20, 40]);
>> canvas = ipaste(canvas, sq2, [60, 120]);
```

where the last argument specifies the canvas coordinate  $(u, v)$  where the pattern will be pasted – the top-left corner of the pattern on the canvas. We can also create a circle

```
>> circle = 0.6 * kcircle(30);
```

of radius 30 pixels with a grey value of 0.6. The Toolbox function `kcircle` returns a square matrix

```
>> size(circle)
ans =
    61    61
```

of zeros with a centred maximal disk of values set to one. We can also paste that on to the canvas

```
>> canvas = ipaste(canvas, circle, [100, 30]);
```

Finally, we draw a line segment onto our canvas

```
>> canvas = iline(canvas, [30, 40], [150, 190], 0.8);
```

which extends from  $(30, 40)$  to  $(150, 190)$  and its pixels are all set to 0.8. The result

```
>> idisp(canvas)
```

is shown in Fig. 12.4b. We can clearly see that the shapes have different brightness, and we note that the line and the circle show the effects of quantization which results in a *steppy* or jagged shape.

Note that all these functions take coordinates expressed in  $(u, v)$  notation not MATLAB® row column notation. The top-left pixel is  $(1, 1)$  not  $(0, 0)$ .

**Fig. 12.4.** Images from code. **a** Some Toolbox generated test patterns; **b** Simple image created from graphical primitives

## 12.2 Monadic Operations

Monadic image-processing operations are shown schematically in Fig. 12.5. The result is an image of the same size  $W \times H$  as the input image, and each output pixel is a function of the corresponding input pixel

$$O[u, v] = f(I[u, v]), \quad \forall(u, v) \in I$$

Since an image is represented by a matrix any MATLAB® element-wise matrix function or operator can be applied, for example scalar multiplication or addition, or functions such `abs` or `sqrt`.

The datatype of each pixel can be changed, for example from `uint8` (integer pixels in the range [0, 255]) to double precision values in the range [0, 1]

```
>> imd = idouble(im);
```

and vice versa

```
>> im = iint(imd);
```

A color image has 3-dimensions which we can also consider as a 2-dimensional image where each pixel value is a 3-vector. A monadic operation can convert a color image to a greyscale image where each output pixel value is a scalar representing the luminance of the corresponding input pixel

```
>> grey = imono(flowers);
```

The inverse operation is

```
>> color = icolor(grey);
```

which returns a 3-dimensional color image where each color plane is equal to `grey` – when displayed it is still appears as a monochrome image. We can create a color image where the red plane is equal to the input image by

```
>> color = icolor(grey, [1 0 0]);
```

which is a red tinted version of the original image.

Many monadic operations are concerned with altering the distribution of grey levels within the image. The distribution can be determined by computing the histogram of the image ▶ which indicates the number of times each pixel value occurs. For example the histogram of the street scene is computed and displayed by

```
>> ihist( street )
```

and the result is shown in Fig. 12.6a. We see that the grey values (horizontal axis) span the complete range from 0 to 255 but the distribution is far from uniform. By inspec-

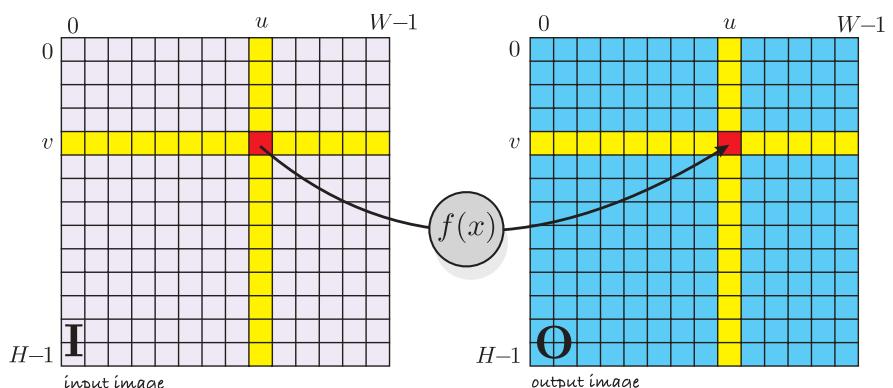
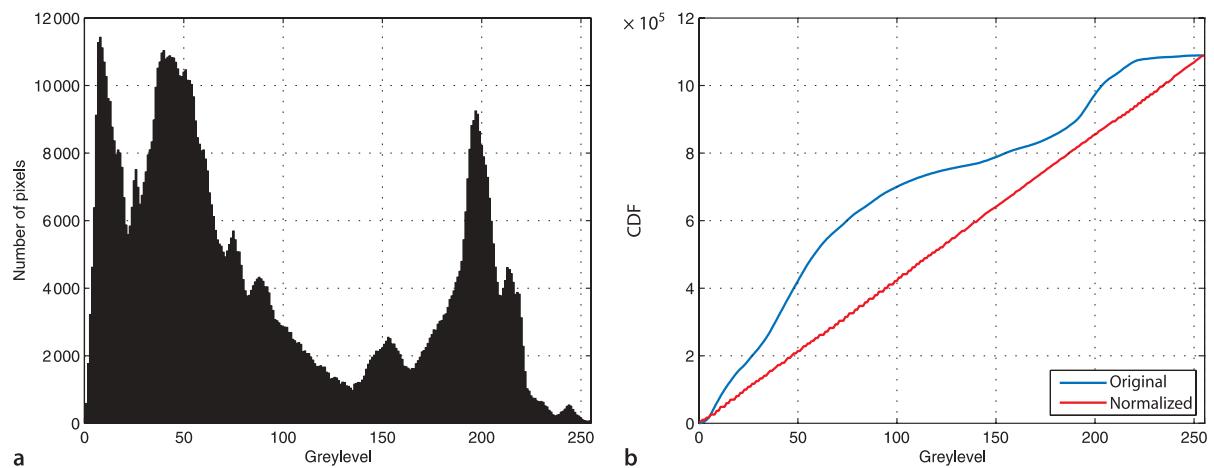


Fig. 12.5.

Monadic image processing operations. Each output pixel is a function of the corresponding input pixel (shown in red)



tion we see that there are three significant peaks, but if we look more closely there are perhaps nine peaks, and if we zoomed right in we would see lots of very minor peaks. The concept of a peak depends on the scale at which we consider the data. We can obtain the histogram as a pair of vectors

```
>> [n,v] = ihist(street);
```

where the elements of `n` are the number of times pixels occur with the value of the corresponding element of `v`. The Toolbox function `peak` will automatically find the peaks

```
>> peak(n, v)'
ans =
    8     40     43     51    197     17     60     26     75    213
    88     92    218    176    153    108    111    147    113    119
   121    126    130    138    230    244
```

and in this case has found 26 peaks most of which are quite minor. Peaks that are *significant* are not only greater than their immediate neighbours they are greater than all other values *nearby* – the problem now is to specify what we mean by nearby. For example the peaks that are greater than all other values within  $\pm 25$  pixel values in the horizontal direction are

```
>> peak(n, v, 'scale', 25)'
ans =
    8     40    197
```

which are the three significant peaks that we observe by eye. The critical part of finding the peaks is choosing the appropriate scale. Peak finding is a topic that we will encounter again later and is also discussed in Appendix K.

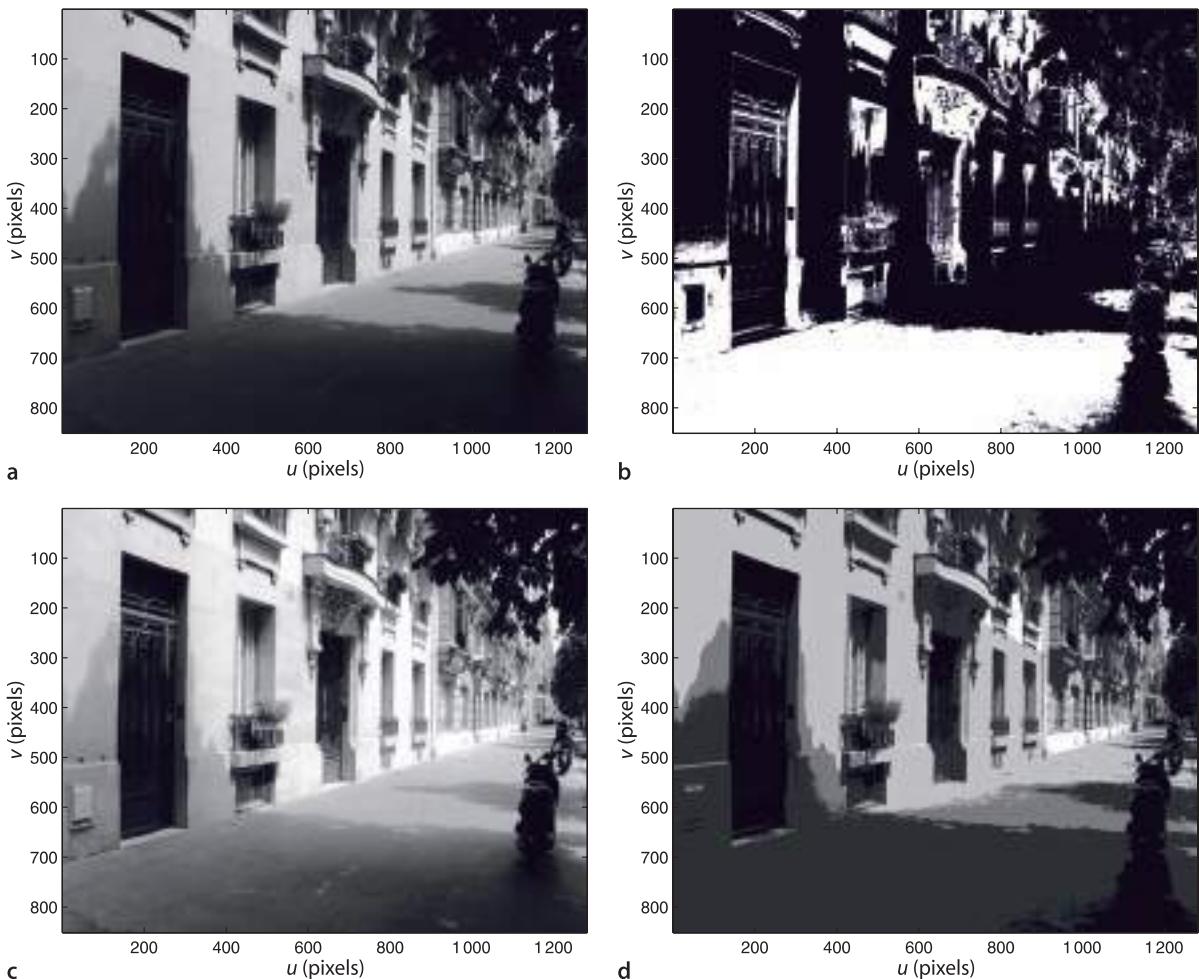
The peaks in the histogram correspond to particular *populations* of pixels in the image. The peak around the greylevel of 40 for example corresponds to a large population of dark pixels which are mostly due to shadows. We can identify the pixels in this peak by a logical monadic operation

```
>> shadows = (street >= 30) & (street <= 80);
>> idisp(shadows)
```

and the resulting image is shown in Fig. 12.7b. All pixels that lie in the interval  $[30, 80]$  are shown as white. This image is of type *logical* but MATLAB® automatically converts logical true and false values to one and zero respectively when used in arithmetic operations. This is a rather simple, and adhoc, example of thresholding where we have selected certain pixels based just on their brightness.

Sometimes an image does not span the full range of available grey levels, for example an underexposed image will have no pixels with high values while an over-

**Fig. 12.6.** Street scene. **a** Histogram, **b** cumulative histogram before and after normalization



**Fig. 12.7.** Some monadic image operations, **a** original, **b** shadow regions, **c** histogram normalized, **d** posterization

The histogram of such an image will have gaps. If  $M$  is the maximum possible pixel value, and  $N < M$  is the maximum value in the image then the stretched image will have at most  $N$  unique pixel values, meaning that  $M - N$  values cannot occur.

exposed image will have no low values. We can apply a linear mapping to the grey-scale values

```
>> im = istretch(street);
```

which ensures that pixel values span the full range ▶ which is either [0, 1] or [0, 255] depending on the class of the image.

A more sophisticated version is histogram normalization or histogram equalization

```
>> im = inormhist(street);
```

which is shown in Fig. 12.7c and ensures that the cumulative distribution of pixel intensities is linear. We see that the grey levels in the shadow region have been raised and stretched out making the details in the shadowed area more visible. The cumulative histogram of the pixel values can be plotted using

```
>> ihist(street, 'cdf');
```

The cumulative distributions of the image before and after normalization are shown in Fig. 12.6b.

Operations such as `istretch` and `inormhist` can enhance the image from the perspective of a human observer, but it is important to remember that no new information has been added to the image. Subsequent image processing steps will not be improved.

As discussed in Sect. 10.3.4 the output of a camera is generally gamma encoded so that the pixel value is a non-linear function  $L^\gamma$  of the luminance sensed at the photosite. Such images can be gamma decoded by a non-linear monadic operation

```
>> im = igamma(street, 1/0.45);
```

which raises each pixel to the specified power, or

```
>> im = igamma(street, 'sRGB');
```

to decode images with the sRGB standard gamma encoding. The resulting *linear* image has greylevels, or tristimulus values, which are proportional to the luminance of the original scene.

Another simple non-linear monadic operation is posterization or banding. This pop-art effect is achieved by reducing the number of grey levels

```
>> idisp( street/64 )
```

as shown in Fig. 12.7d. Since integer division is used the resulting image has pixels with values in the range [0, 3] and therefore just four different shades of grey.

### 12.3 Diadic Operations

Diadic operations are shown schematically in Fig. 12.8. Two input matrices result in a single output matrix, and all three images are of the same size. Each output pixel is a function of the corresponding pixels in the two input images

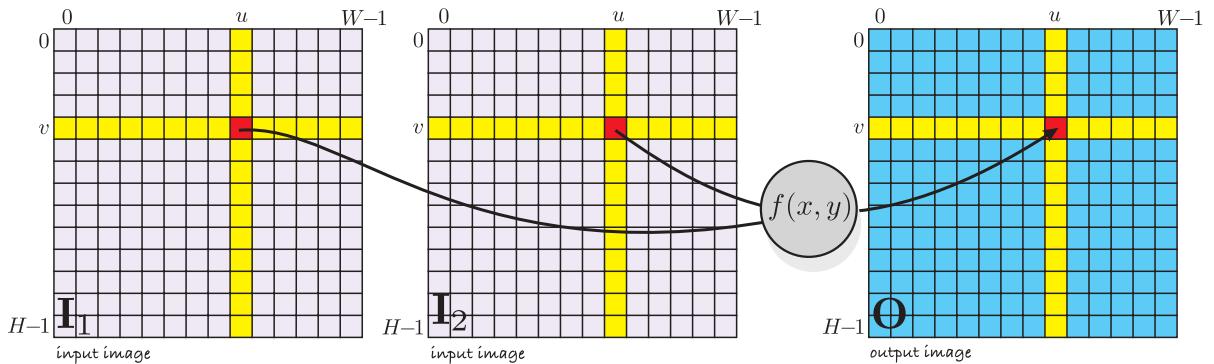
$$O[u, v] = f(I_1[u, v], I_2[u, v]), \quad \forall(u, v) \in I_1$$

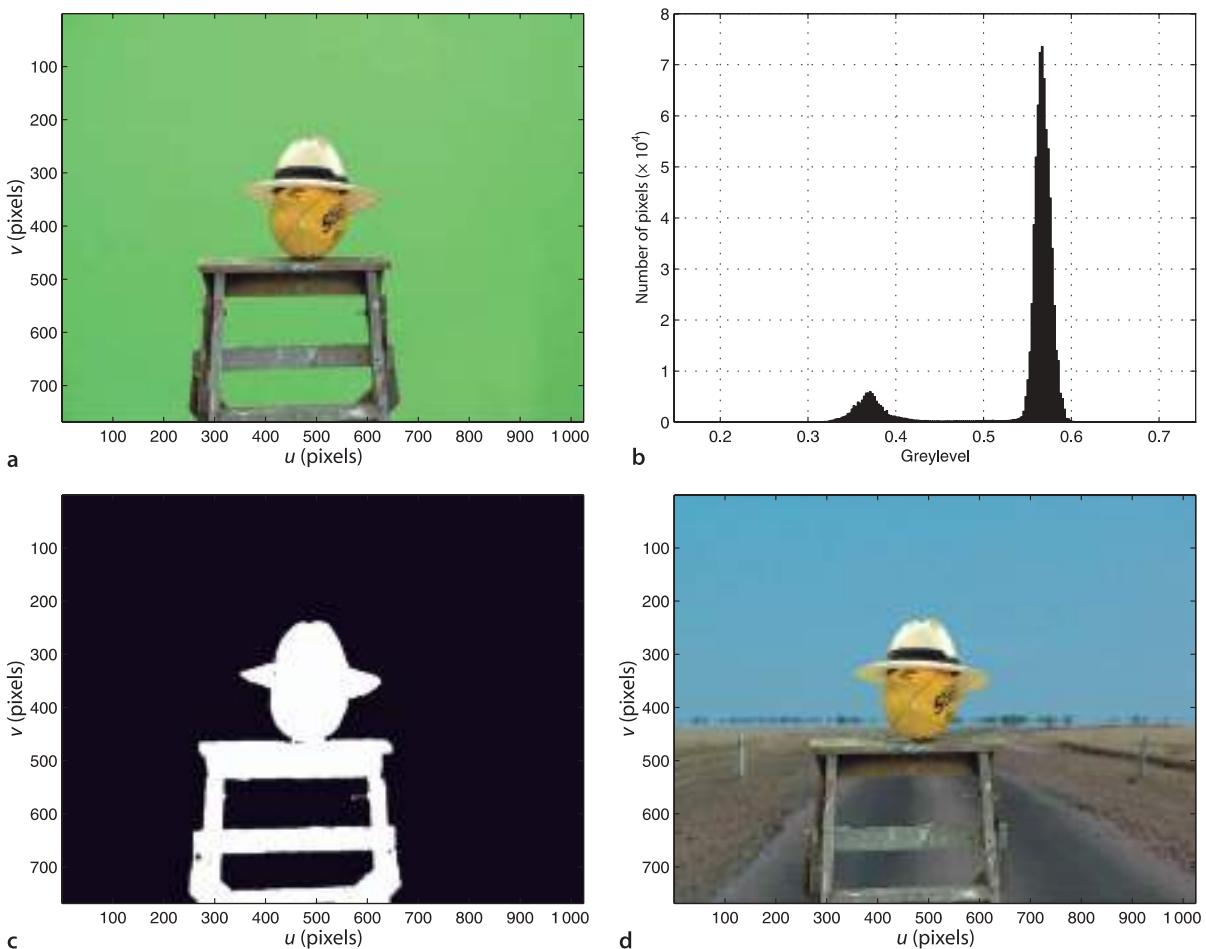
Examples of useful diadic operations include binary arithmetic operators such as addition, subtraction, element-wise multiplication, or builtin MATLAB® diadic matrix functions such as `max`, `min`, `bitand`, `atan2` etc.

Subtracting one `uint8` image from another results in another `uint8` image even though the result is potentially negative. MATLAB® quite properly clamps values to the interval [0, 255] so subtracting a larger number from a smaller number will result in zero not a negative value. With addition a result greater than 255 will be set to 255. To remedy this, the images should be first converted to signed integers using the MATLAB® function `cast` or to floating point values using the Toolbox function `idouble`.

We will illustrate diadic operations with two examples. The first example is a technique used on television to allow the image of a person to be superimposed on some background, for example a weather presenter superimposed on a weather map. The

**Fig. 12.8.** Diadic image processing operations. Each output pixel is a function of the two corresponding input pixels (shown in red)





**Fig. 12.9.** Chroma-keying. **a** The subject against a green background; **b** a histogram of green chromaticity values; **c** the computed mask image where true is white; **d** the subject *masked* into a background scene (photo courtesy of Fiona Corke)

subject is filmed against a blue or green background which makes it quite easy, using just the pixel values, to distinguish between background and the subject. We load an image of a subject taken in front of a green screen

```
>> subject = imread('greenscreen.jpg', 'double');
```

and this is shown in Fig. 12.9a. We compute the chromaticity coordinates Eq. 10.8

```
>> linear = igamma(subject, 'sRGB');
>> [r,g] = tristim2cc(linear);
```

after first converting the gamma encoded color image to linear tristimulus values. In this case  $g$  alone is sufficient to distinguish the background pixels. A histogram of values

```
>> ihist(g)
```

shown in Fig. 12.9b indicates a large population of pixels around 0.55 which is the background and another population which belongs to the subject. We can safely say that the subject corresponds to any pixel for which  $g < 0.45$ . We define a mask image

```
>> mask = g < 0.45;
>> idisp(mask)
```

where a pixel is true (equal to one) if it is part of the subject as shown in Fig. 12.9c. We need to apply this mask to all three so we replicate it

```
>> mask3 = icolor( idouble(mask) );
```

The image of the subject without the background is

```
>> idisp(mask3 .* subject);
```

Next we load the desired background image

```
>> bg = iread('road.png', 'double');
```

and scale and crop it to be the same size as our original image

```
>> bg = isamesize(bg, subject);
```

and display it with a *cutout* for the subject

```
>> idisp(bg .* (1-mask3))
```

Now we add the subject with no background, to the background with no subject to obtain the subject on the road

```
>> idisp(subject.*mask3 + bg.* (1-mask3));
```

which is shown in Fig. 12.9d. The technique will of course fail if the subject contains any colors that match the color of the background.► This example could be solved more compactly using the Toolbox per-pixel switching function `ipixswitch`

```
>> ipixswitch(mask, subject, bg);
```

where all arguments are images of the same width and height, and each output pixel is selected from the corresponding pixel in the second or third image according to the logical value of the corresponding pixel in the first image.

Distinguishing foreground objects from the background is an important problem in robot vision but the terms foreground and background are ill-defined and application specific. We can be almost guaranteed never to have the luxury of a special background as we did for the chroma-key example. We could instead take a picture of the scene without a foreground object present and consider this to be the background, but that requires that we have special knowledge about when the foreground object is not present. It also assumes that the background does not vary over time. Variation is a significant problem in real-world scenes where ambient illumination and shadows change over quite short time intervals, and the scene may be structurally modified over very long time intervals.

In the next example we process an image sequence and *estimate* the background even though there are a number of objects moving in the scene. We will use a recursive algorithm that updates the estimated background image  $\hat{B}$  at each time step based on the previous estimate and the current image

$$\hat{B}_{(k+1)} = \hat{B}_{(k)} + c(I_{(k)} - \hat{B}_{(k)})$$

where  $k$  is the time step and  $c(\cdot)$  is a monadic image saturation function

$$c(x) = \begin{cases} \sigma, & x > \sigma \\ x, & -\sigma \leq x \leq \sigma \\ -\sigma, & x < -\sigma \end{cases}$$

To demonstrate this we open a movie showing two people moving in the lobby of a building

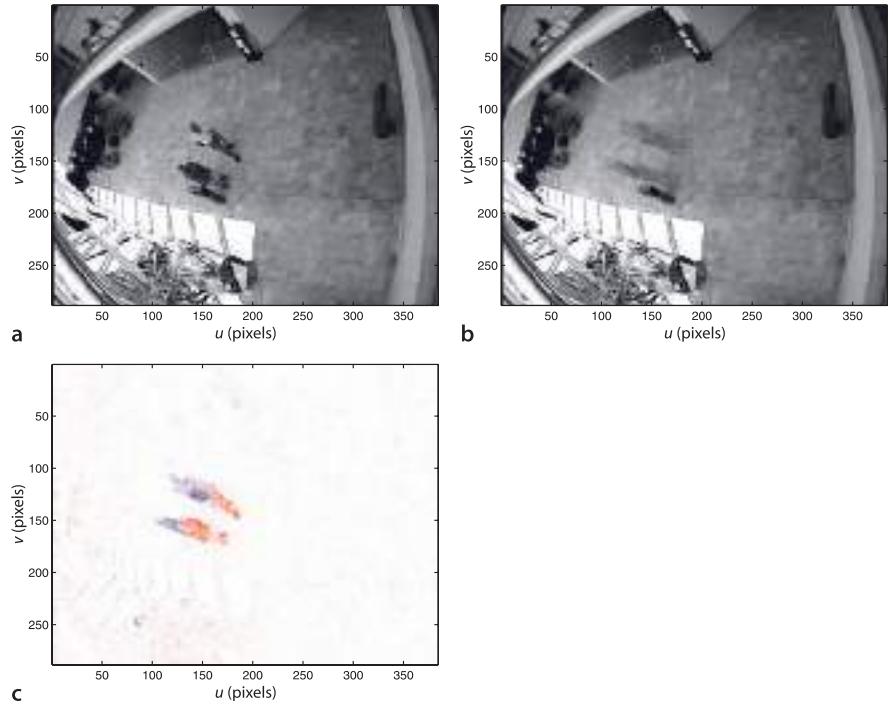
```
>> vid = Movie('LeftBag.mpg');
384 x 288 @ 24.999750, 1440 frames
```

and initialize the background to the first image in the sequence

```
>> bg = vid.grab();
```

then the main loop is

In the early days of television a blue screen was used. Today a green background is more popular because of problems that occur with blue eyes and blue denim clothing.

**Fig. 12.10.**

Example of image sequence analysis for the INRIA *LeftBag* image sequence at frame 250. **a** The current image; **b** the estimated background image; **c** the difference between the current and estimated background images where white is zero, red and blue are negative and positive values respectively and magnitude is indicated by color intensity (movie from the collection of the EC Funded CAVIAR project/ IST 2001 37540)

```

1 sigma = 2;
2 while 1
3   im = vid.grab;
4   if isempty(im), break; end
5   d = im-bg;
6   d = max(min(d, sigma), -sigma); % apply c(.)
7   bg = bg + d;
8   idisp(bg); drawnow
9 end

```

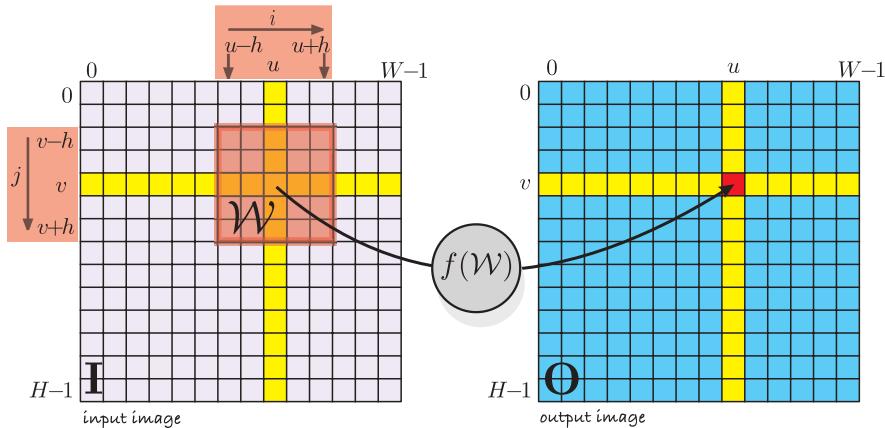
One frame from this sequence is shown in Fig. 12.10a. The estimated background image shown in Fig. 12.10b reveals the static elements of the scene and the two moving people have become a faint blur. Subtracting the scene from the estimated background creates an image where pixels are bright where they are different to the background as shown in Fig. 12.10c. Applying a threshold to the absolute value of this difference image shows the area of the image where there is motion. Of course if the people stay still long enough they will become part of the background.

## 12.4 Spatial Operations

Spatial operations are shown schematically in Fig. 12.11. Each pixel in the output image is a function of all pixels in a *region* surrounding the corresponding pixel in the input image

$$O[u, v] = f(I[u + i, v + j]), \quad \forall(i, j) \in \mathcal{W}, \quad \forall(u, v) \in I$$

where  $\mathcal{W}$  is known as the window, typically a  $w \times w$  square region with odd side length  $w = 2h + 1$  where  $h \in \mathbb{Z}^+$  is the half-width. In Fig. 12.11 the window includes all pixels in the red shaded region. Spatial operations are powerful because of the variety of possible functions  $f(\cdot)$ , linear or non-linear, that can be applied. The remainder of this section discusses linear spatial operators such as smoothing and edge detection, and some non-linear functions such as rank filtering and template matching. The following section covers a large and important class of non-linear spatial operators known as mathematical morphology.



**Fig. 12.11.**  
Spatial image processing operations. The red shaded region shows the window  $\mathcal{W}$  that is the set of pixels used to compute the output pixel (show in red)

### 12.4.1 Convolution

A very important linear spatial operator is convolution

$$O[u, v] = \sum_{(i,j) \in \mathcal{W}} I[u + i, v + j] K[i, j], \quad \forall (u, v) \in I$$

where  $K \in \mathbb{R}^{w \times w}$  is the convolution kernel. For every output pixel the corresponding window of pixels from the input image  $\mathcal{W}$  is multiplied element-wise with the kernel  $K$ . The centre of the window and kernel is considered to be coordinate  $(0, 0)$  and  $i, j \in [-h, h]$ . This can be considered as the weighted sum of pixels within the window where the weights are defined by the kernel  $K$ . As we will see convolution is the workhorse of image processing and the kernel  $K$  can be chosen to perform functions such as smoothing, gradient calculation or edge detection. Convolution is often written in operator form as

$$O = K \otimes I$$

Convolution is computationally expensive – an  $N \times N$  input image with a  $w \times w$  kernel requires  $w^2 N^2$  multiplication and additions. In the Toolbox convolution is performed using the function `iconv`

```
>> O = iconv(K, I);
```

If  $I$  has multiple color planes then so will the output image – each output color plane is the convolution of the corresponding input plane with the kernel  $K$ .

**Properties of convolution.** Convolution obeys the familiar rules of algebra, it is commutative

$$A \otimes B = B \otimes A$$

associative

$$A \otimes B \otimes C = (A \otimes B) \otimes C = A \otimes (B \otimes C)$$

distributive (superposition applies)

$$A \otimes (B + C) = A \otimes B + A \otimes C$$

linear

$$A \otimes (\alpha B) = \alpha(A \otimes B)$$

and shift invariant – if  $S(\cdot)$  is a spatial shift then

$$A \otimes S(B) = S(A \otimes B)$$

that is, convolution with a shifted image is the same as shifting the result of the convolution with the unshifted image.

### 12.4.1.1 Smoothing

Consider a kernel which is a square  $21 \times 21$  matrix containing equal elements

```
>> K = ones(21,21) / 21^2;
```

and of unit volume, that is, its values sum to one. The result of convolving an image with this kernel is an image where each output pixel is the mean of the pixels in a corresponding  $21 \times 21$  neighbourhood in the input image. As you might expect this averaging

```
>> lena = imread('lena.pgm', 'double');
>> idisp( iconv(K, lena) );
```

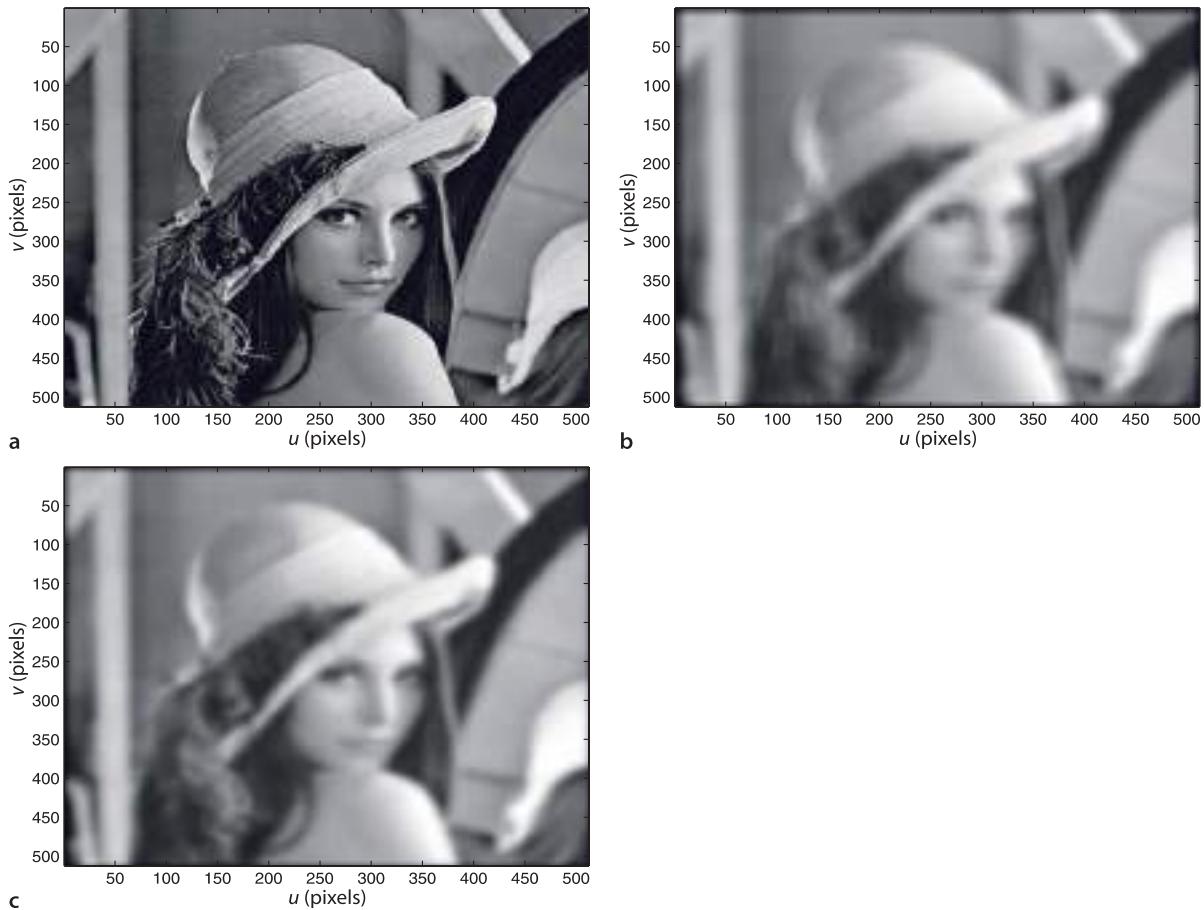
Defocus involves a kernel which is a 2-dimensional Airy pattern or sinc function. The Gaussian function is similar in shape, but is always positive whereas the Airy pattern has low amplitude negative going rings.

leads to smoothing, blurring or *defocus*◀ which we see in Fig. 12.12b. Looking very carefully we will see some faint horizontal and vertical lines – an artefact known as ringing. A more suitable kernel for smoothing is the 2-dimensional Gaussian function

$$G(u, v) = \frac{1}{2\pi\sigma^2} e^{-\frac{u^2+v^2}{2\sigma^2}} \quad (12.1)$$

which is symmetric about the origin and the volume under the curve is unity. The spread of the Gaussian is controlled by the standard deviation parameter  $\sigma$ . Applying this kernel to the image

```
>> K = kgauss(5);
>> idisp( iconv(K, lena) );
```



The Lena image became something of a standard for image processing research in the 1970s. It was digitized by image compression researchers at the University of Southern California Signal and Image Processing Institute (SIPI) in 1973 from a November 1972 Playboy centerfold. The model is Lena Soderberg (nee Sjööblom) who lives in Sweden. She is reported to be amused by this use of her picture and attended an imaging conference in 1997. See also <http://www.lenna.org>.



produces the result shown in Fig. 12.12c. Here we have specified the standard deviation of the Gaussian to be 5 pixels. The discrete approximation to the Gaussian is

```
>> about(K)
K [double] : 31x31 (7688 bytes)
```

a  $31 \times 31$  kernel. Smoothing can be achieved conveniently using the Toolbox function **isMOOTH**

```
>> idisp( ismooth(lena, 5) )
```

Blurring is a counter-intuitive image processing operation since we typically go to a lot of effort to obtain a clear and crisp image. To deliberately *ruin it* seems, at face value, somewhat reckless. However as we will see Gaussian smoothing turns out to be extremely useful.

The kernel is itself a matrix and therefore we can display it as an image

```
>> idisp( K );
```

which is shown in Fig. 12.13a. We clearly see the large value at the centre of the kernel and that it falls off smoothly in all directions. We can also display the kernel as a surface

```
>> surfl(-15:15, -15:15, K);
```

**How wide is my Gaussian?** When choosing a Gaussian kernel we need to consider the standard deviation, usually defined by the task, and the dimensions of the kernel  $\mathcal{W} \in \mathbb{R}^{w \times w}$  that contains the discrete Gaussian function. Computation time is proportional to  $w^2$  so ideally we want the window to be no bigger than it needs to be. The Gaussian decreases monotonically in all directions but never reaches zero. Therefore we choose the half-width  $h$  of the window such that value of the Gaussian is less than some threshold outside the  $w \times w$  convolution window.

At the edge of the window, a distance  $h$  from the centre, the value of the Gaussian will be  $e^{-h^2/2\sigma^2}$ . For  $\sigma=1$  and  $h=2$  the Gaussian will be  $e^{-2} \approx 0.14$ , for  $h=3$  it will be  $e^{-4.5} \approx 0.01$ , and for  $h=4$  it will be  $e^{-8} \approx 3.4 \times 10^{-4}$ . If  $h$  is not specified the Toolbox chooses  $h = 3\sigma$ . For  $\sigma=1$  that is a  $7 \times 7$  window which contains all values of the Gaussian greater than 1% of the peak value.

**Properties of the Gaussian.** The Gaussian function has some special properties. The convolution of two Gaussians is another Gaussian

$$\mathbf{G}(\sigma_1) \otimes \mathbf{G}(\sigma_2) = \mathbf{G}\left(\sqrt{\sigma_1^2 + \sigma_2^2}\right)$$

For the case where  $\sigma_1 = \sigma_2 = \sigma$  then

$$\mathbf{G}(\sigma) \otimes \mathbf{G}(\sigma) = \mathbf{G}(\sqrt{2}\sigma)$$

The 2-dimensional Gaussian is separable – it can be written as the product of two 1-dimensional Gaussians

$$\mathbf{G}(u, v) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{u^2}{2\sigma^2}} \times \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{v^2}{2\sigma^2}}$$

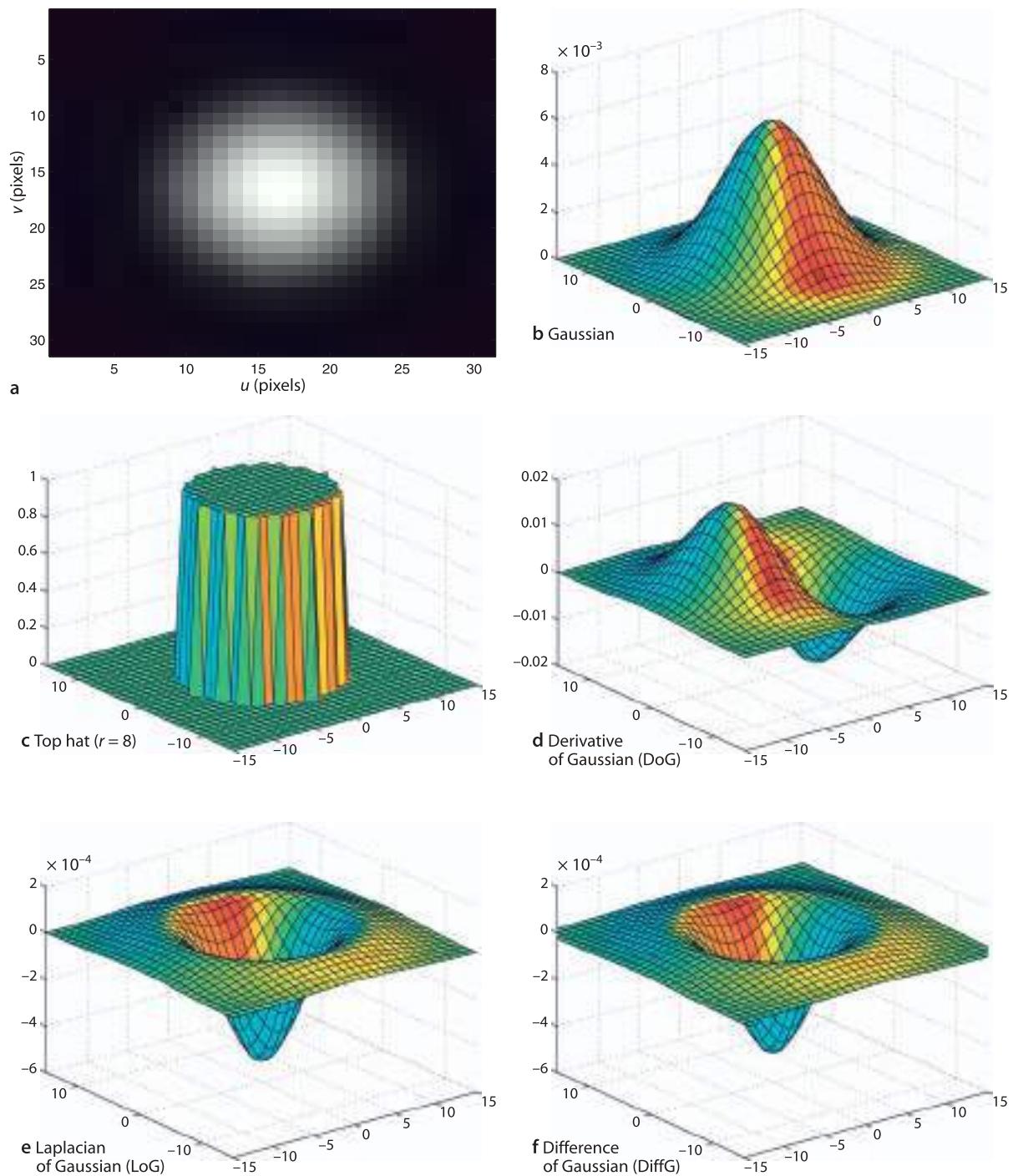
This implies that convolution with a 2-dimensional Gaussian can be computed by convolving each row with a 1-dimensional Gaussian, and then each column. The total number of operations is reduced to  $2wN^2$ , better by a factor of  $w$ . A Gaussian also has the same shape in the spatial and frequency domains.

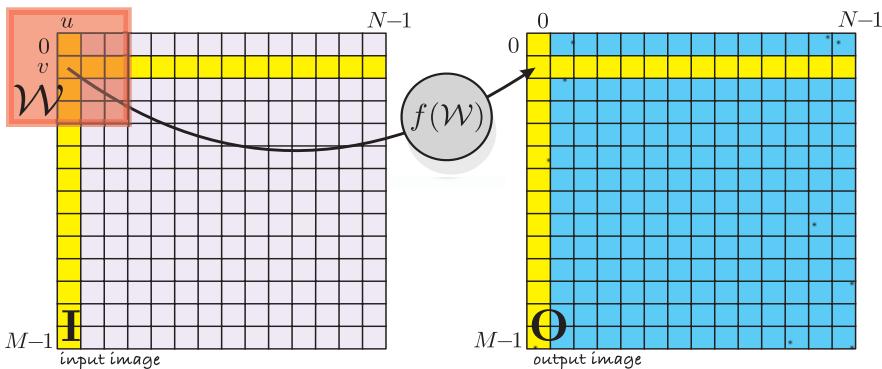
as shown in Fig. 12.13b. A crude approximation to the Gaussian is the top hat kernel which is cylinder with vertical sides rather than a smooth and gentle fall off in amplitude. The function `kcircle` creates a kernel which can be considered a unit height cylinder of specified radius

```
>> K = kcircle(8, 15);
```

**Fig. 12.13.** Gallery of commonly used convolution kernels.  $h = 15$ ,  $\sigma = 5$

as shown in Fig. 12.13c. The arguments specify a radius of 8 pixels within a window of half width  $h = 15$ .





**Fig. 12.14.**  
For the case where the window  $\mathcal{W}$  falls off the edge of the input image the output pixel at  $(u, v)$  is not defined. The hatched pixels in the output image are all those for which the output value is not defined

#### 12.4.1.2 Boundary Effects

A difficulty with convolution occurs when the window is close to the edge of the input image as shown in Fig. 12.14. In this case the output pixel is a function of a window that contains pixels *beyond the edge* of the input image. There are several common remedies to this problem. Firstly, we can assume the pixels beyond the image have a particular value. A common choice is zero and this is the default behaviour implemented by the Toolbox function `iconv`. We can see the effect of this in Fig. 12.12 where the borders of the smoothed image are dark due to the influence of these zeros.

Another option is to consider that the result is invalid when the window exceeds the boundary of the image. Invalid output pixels are shown hatched out in Fig. 12.14. The result is an output image that is  $(M - 2h) \times (N - 2h)$ . This option can be selected by passing the option '`valid`' to `iconv`.

#### 12.4.1.3 Edge Detection

Frequently we are interested in finding the edges of objects in a scene. Consider the image

```
>> castle = imread('castle_sign.jpg', 'double', 'grey');
```

shown in Fig. 12.15a. It is informative to look at the pixel values along a 1-dimensional profile through the image. A horizontal profile of the image at  $v = 360$  is

```
>> p = castle(360,:);
```

which is a vector that we can plot

```
>> plot(p);
```

against the horizontal coordinate  $u$  in Fig. 12.15b. The clearly visible tall spikes correspond to the white letters and other markings on the sign. Looking at one of the spikes more closely, Fig. 12.15c, we see the intensity profile across the vertical stem of the letter T. The background intensity  $\approx 0.3$  and the bright intensity  $\approx 0.9$  but will depend on lighting levels. However the very rapid increase over the space of just a few pixels is distinctive and a more reliable indication of an edge than any decision based on the actual grey levels.

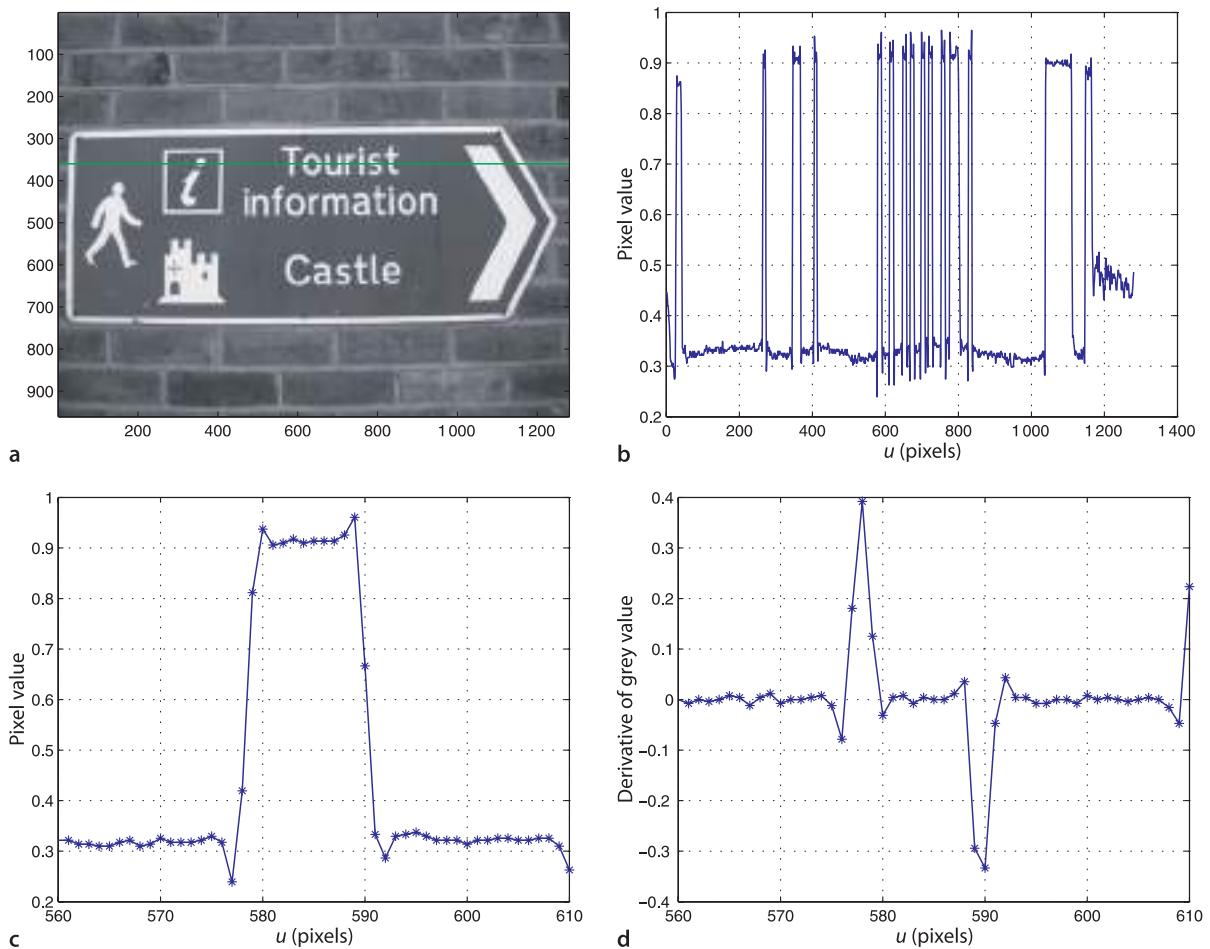
The first-order derivative along this cross-section is

$$p'[v] = p[v] - p[v - 1]$$

which can be computed using the MATLAB® function `diff`

```
>> plot(diff(p))
```

and is shown in Fig. 12.15d. The signal is nominally zero with clear non-zero responses at the edges of an object, in this case the edges of the stem of the letter T.



**Fig. 12.15.** Edge intensity profile.  
a Original image; b greylevel profile along horizontal line  $v = 360$ ;  
c close up view of the spike at  $u \approx 580$ ; d derivative of c (image from the ICDAR 2005 OCR dataset; Lucas 2005)

The derivative at point  $v$  can also be written as a *symmetrical* first-order difference

$$p'[v] = \frac{1}{2}(p[v+1] - p[v-1])$$

which is equivalent to convolution with the 1-dimensional kernel

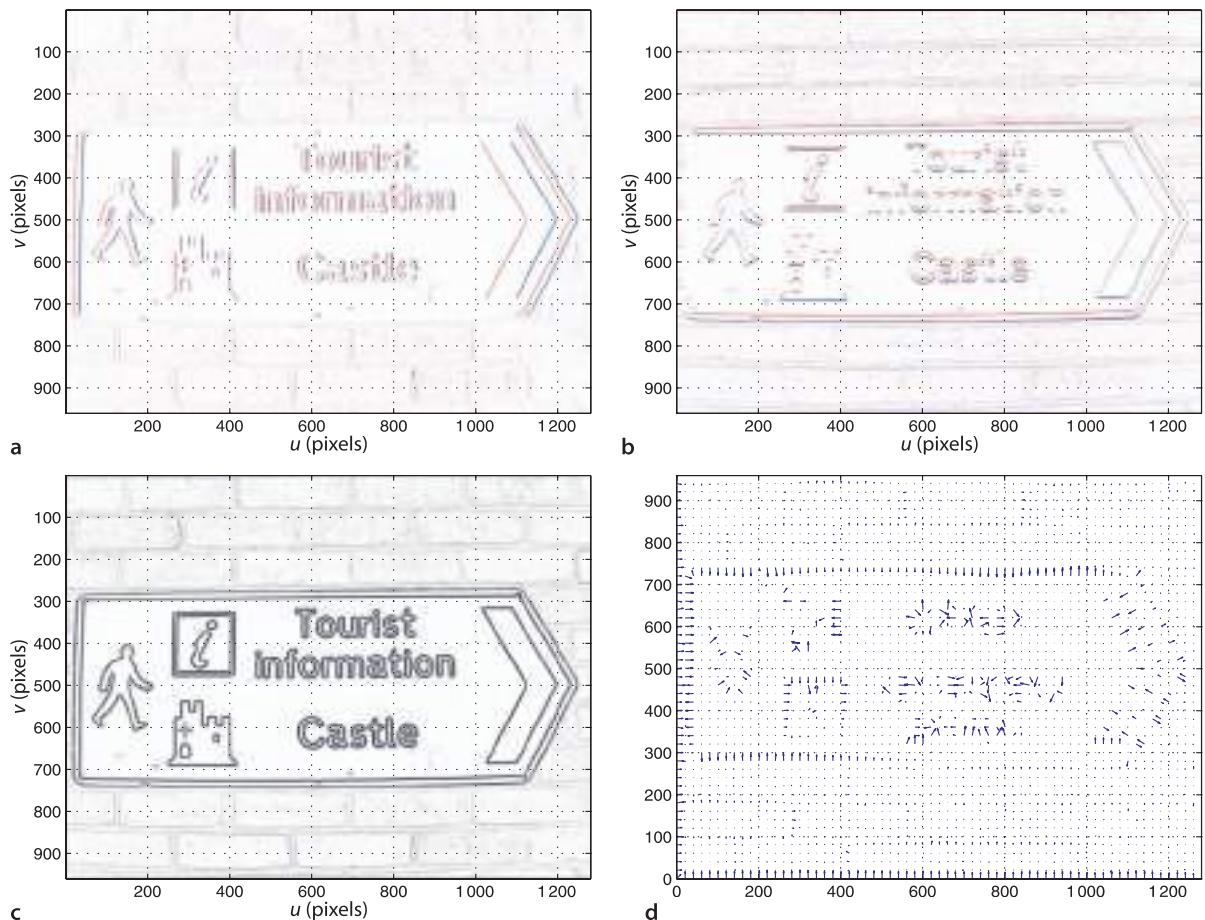
$$K = \begin{pmatrix} -\frac{1}{2} & 0 & \frac{1}{2} \end{pmatrix}$$

Convolving the image with this kernel

```
>> K = [-0.5 0 0.5];
>> idisp( iconv(castle, K), 'invsigned')
```

produces a result very similar to that shown in Fig. 12.16a in which vertical edges, high horizontal gradients, are clearly seen.

Since this kernel has signed values the result of the convolution will also be signed, that is, the gradient at a pixel can be positive or negative as shown in Fig. 12.15d. `idisp` always displays the minimum, most negative, value as black and the maximum, most positive, value as white. Zero would therefore appear as middle grey. The '`signed`' option to `idisp` uses red and blue shading to clearly indicate sign – zero is black, negative pixels are red, positive pixels are blue and the intensity of the color is proportional to pixel magnitude. The '`invsigned`' option is similar except that zero is indicated by white.



Many convolution kernels have been proposed for computing horizontal gradient. The most common is the Sobel kernel

```
>> Du = ksobel
Du =
-1     0     1
-2     0     2
-1     0     1
```

and each row is similar to the 1-dimensional kernel  $K$  defined above. The overall result is a weighted sum of the horizontal gradient for the current row, and the rows above and below. Convolving our image with this kernel

```
>> idisp( iconv(castle, Du), 'invsigned' )
```

generates the horizontal gradient image shown in Fig. 12.16a which highlights vertical edges. Vertical gradient is computed using the transpose of the kernel

```
>> idisp( iconv(castle, Du'), 'invsigned' )
```

and highlights horizontal edges<sup>►</sup> as shown in Fig. 12.16b. The notation used for gradients varies considerably in the literature. Most commonly the horizontal and vertical gradient are denoted respectively as  $\partial I / \partial u$ ,  $\partial I / \partial v$ ;  $\nabla_u I$ ,  $\nabla_v I$  or  $I_u$ ,  $I_v$ . In operator form this is written

$$I_u = D \otimes I$$

$$I_v = D^T \otimes I$$

where  $D$  is a derivative kernel such as Sobel.

**Fig. 12.16.** Edge gradient. **a**  $u$ -axis derivative; **b**  $v$ -axis derivative; **c** gradient magnitude; **d** gradient direction

Filters can be designed to respond to edges at any arbitrary angle. The Sobel kernel itself can be considered as an image and rotated using `irotate`. To obtain angular precision generally requires a larger kernel is required such as that generated by `kdgauss`.



**Carl Friedrich Gauss** (1777–1855) was a German mathematician who made major contributions to fields such as number theory, differential geometry, magnetism, astronomy and optics. He was a child prodigy, born in Brunswick, Germany, the only son of uneducated parents. At the age of three he corrected, in his head, a financial error his father had made, and made his first mathematical discoveries while in his teens.

Gauss was a perfectionist and a hard worker but not a prolific writer. He refused to publish anything he did not consider complete and above criticism. It has been suggested that mathematics could have been advanced by fifty years if he had published all of his discoveries. According to legend Gauss was interrupted in the middle of a problem and told that his wife was dying – he responded “Tell her to wait a moment until I am through”.

The normal distribution, or Gaussian function, was not one of his achievements. It was first discovered by de Moivre in 1733 and again by Laplace in 1778.

Taking the derivative of a signal accentuates high-frequency noise, and all images have noise as discussed on page 260. At the pixel level noise is a stationary random process – the values are not correlated between pixels. However the features that we are interested in such as edges have correlated changes in pixel value over a larger spatial scale as shown in Fig. 12.15c. We can reduce the effect of noise by smoothing the image before taking the derivative

$$\mathbf{I}_u = \mathbf{D} \otimes (\mathbf{G}(\sigma) \otimes \mathbf{I})$$

Instead of convolving the image with the Gaussian and *then* the derivative, we exploit the associative property of convolution to write

$$\nabla \mathbf{I} = \mathbf{D} \otimes (\mathbf{G}(\sigma) \otimes \mathbf{I}) = \underbrace{(\mathbf{D} \otimes \mathbf{G}(\sigma))}_{\text{DoG}} \otimes \mathbf{I}$$

We convolve the image with the *derivative of the Gaussian* (DoG) which can be obtained numerically by

```
>> Gu = iconv( Du, kgauss(sigma) );
```

or analytically by taking the derivative, in the  $u$ -direction, of the Gaussian Eq. 12.1 yielding

$$\mathbf{G}_u(u, v) = -\frac{u}{2\pi\sigma^2} e^{-\frac{u^2+v^2}{2\sigma^2}} \quad (12.2)$$

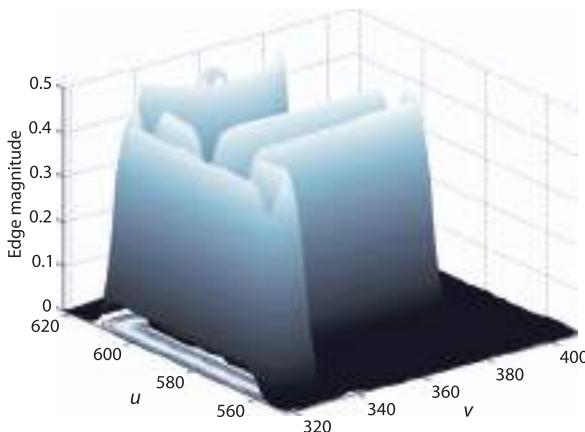
This is computed by the Toolbox function `kdgau`ss and is shown in Fig. 12.13d. The standard deviation  $\sigma$  controls the *scale* of the edges that are detected. For large  $\sigma$ , which implies increased smoothing, edges due to fine texture will be attenuated leaving only the edges of large features. This ability to find edges at different spatial scale is important and underpins the concept of scale space that we will return to in Sect. 13.3.2. Another interpretation of this operator is as a spatial *bandpass filter* since it is a cascade of a low-pass filter (smoothing) with a high-pass filter (differentiation).

Computing the horizontal and vertical components of gradient at each pixel

```
>> Iu = iconv( castle, kdgau(2) );
>> Iv = iconv( castle, kdgau(2)' );
```

allows us to compute the magnitude of the gradient at each pixel

```
>> m = sqrt( Iu.^2 + Iv.^2 );
```



**Fig. 12.17.**  
Close up of gradient magnitude around the letter T shown as a 3-dimensional surface

This *edge-strength* image shown in Fig. 12.16c reveals the edges very distinctly. The direction of the gradient at each pixel is

```
>> th = atan2( Iv, Iu);
```

and is best viewed as a sparse quiver plot

```
>> quiver(1:20:numcols(th), 1:20:numrows(th), ...
    Iu(1:20:end,1:20:end), Iv(1:20:end,1:20:end))
```

as shown in Fig. 12.16d. The edge direction plot is much noisier than the magnitude plot. Where the edge gradient is strong, on the border of the sign or the edges of letters, the direction is normal to the edge, but the fine-scale brick texture appears as almost random edge direction. The gradient images can be computed conveniently using the Toolbox function

```
>> [du,dv] = isobel( castle, kdgauss(2) );
```

where the last argument overrides the default Sobel kernel.

A well known and very effective edge detector is the Canny edge operator. It uses the edge magnitude and direction that we have just computed and performs two additional steps. The first is non-local maxima suppression. Consider the gradient magnitude image of Fig. 12.16c as a 3-dimensional surface where height is proportional to brightness as shown in Fig. 12.17. We see a series of hills and ridges and we wish to find the pixels that lie at the tops of hills or along *ridge lines*. By examining pixel values in a local neighbourhood *normal* to the edge direction, that is in the direction of the edge gradient, we can find the maximum value and set all other pixels to zero. The result is a set of non-zero pixels corresponding to peaks and ridge lines. The second step is hysteresis thresholding. For each non-zero pixel that exceeds the upper threshold a chain is created of adjacent pixels that exceed the lower threshold. Any other pixels are set to zero.

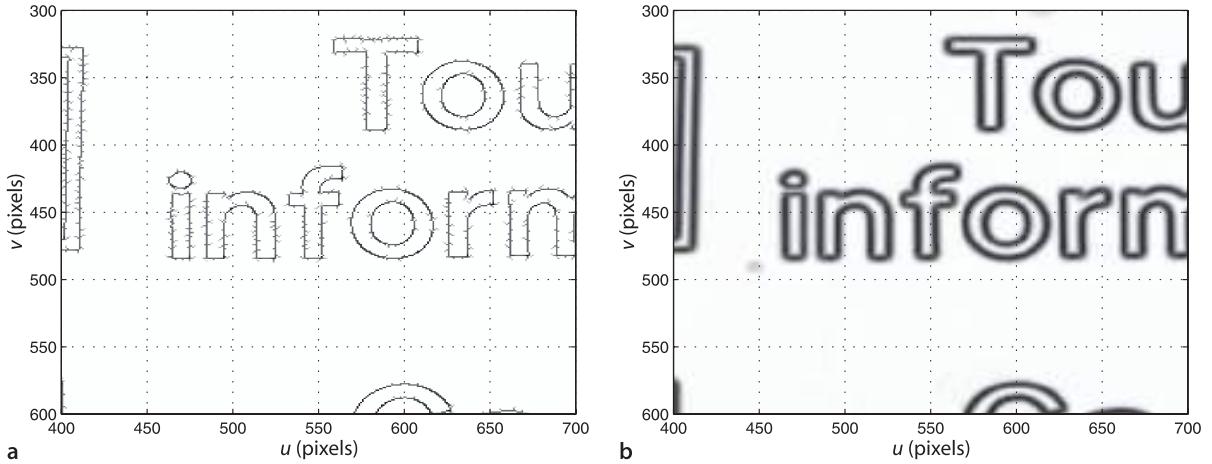
To apply the Canny operator to our example image is straightforward

```
>> edges = icanny(castle, 2);
```

Pierre-Simon Laplace (1749–1827) was a French mathematician and astronomer who consolidated the theories of mathematical astronomy in his five volume *Mécanique Céleste* (Celestial Mechanics). While a teenager his mathematical ability impressed d'Alembert who helped to procure him a professorship. When asked by Napoleon why he hadn't mentioned God in his book on astronomy he is reported to have said "Je n'avais pas besoin de cette hypothèse-là" ("I have no need of that hypothesis"). He became a count of the Empire in 1806 and later a marquis.

The Laplacian operator, a second-order differential operator, and the Laplace transform are named after him.





**Fig. 12.18.** Comparison of two edge operators: **a** Canny operator with default parameters; **b** Magnitude of derivative of Gaussian kernel ( $\sigma=2$ ). The  $|\text{DoG}|$  operator requires less computation than Canny but generates thicker edges. For both cases results are shown inverted, white is zero

and returns an image where the edges are marked by non-zero intensity values corresponding to gradient magnitude at that pixel as shown in Fig. 12.18a. We observe that the edges are much thinner than those for the magnitude of derivative of Gaussian operator which is shown in Fig. 12.18b. In this example  $\sigma=2$  for the derivative of Gaussian operation. The hysteresis threshold parameters can be set with optional arguments.

So far we have considered an edge as a point of high gradient, and non-maxima suppression has been used to *search* for the maximum value in local neighbourhoods. An alternative means to find the point of maximum gradient is to compute the second derivative and determine where this is zero. The Laplacian operator

$$\nabla^2 \mathbf{I} = \frac{\partial^2 \mathbf{I}}{\partial u^2} + \frac{\partial^2 \mathbf{I}}{\partial v^2} = \mathbf{I}_{uu} + \mathbf{I}_{vv} \quad (12.3)$$

is the sum of the second spatial derivative in the horizontal and vertical directions. For a discrete image this can be computed by convolution with the Laplacian kernel

```
>> L = klaplace()
L =
    0     1     0
    1    -4     1
    0     1     0
```

which is isotropic – it responds equally to edges in any direction. The second derivative is even more sensitive to noise than the first derivative and is again commonly used in conjunction with a Gaussian smoothed image

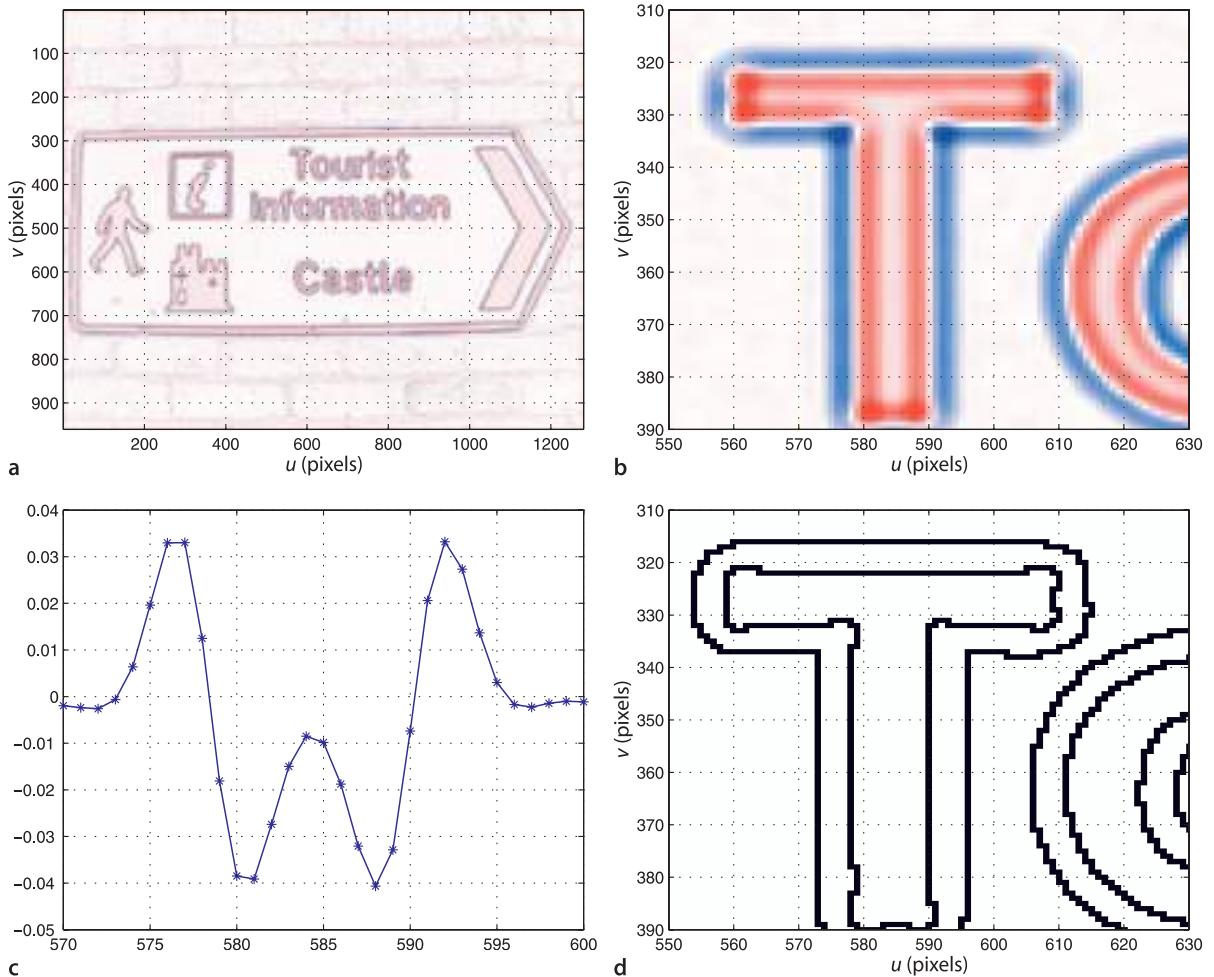
$$\nabla^2 \mathbf{I} = \mathbf{L} \otimes (\mathbf{G}(\sigma) \otimes \mathbf{I}) = \underbrace{(\mathbf{L} \otimes \mathbf{G}(\sigma)) \otimes \mathbf{I}}_{\text{LoG}} \quad (12.4)$$

which we combine into the Laplacian of Gaussian kernel (LoG), and  $\mathbf{L}$  is the Laplacian kernel given above. This can be written analytically as

$$\text{LoG}(u, v) = \frac{\partial^2 \mathbf{G}}{\partial u^2} + \frac{\partial^2 \mathbf{G}}{\partial v^2} \quad (12.5)$$

$$= \frac{1}{\pi \sigma^4} \left( \frac{u^2 + v^2}{2\sigma^2} - 1 \right) e^{-\frac{u^2+v^2}{2\sigma^2}} \quad (12.6)$$

which is known as the Marr-Hildreth operator or the *Mexican hat kernel* and is shown in Fig. 12.13e.



**Fig. 12.19.** Laplacian of Gaussian.  
**a** Laplacian of Gaussian; **b** closeup of **a** around the letter T where blue and red colors indicate positive and negative values respectively; **c** a horizontal cross-section of the LoG through the stem of the T; **d** closeup of the zero-crossing detector output at the letter T

We apply this kernel to our image by

```
>> lap = iconv( castle, klog(2) );
```

and the result is shown in Fig. 12.19a and b. The maximum gradient occurs where the second derivative is zero but a significant edge is a zero crossing from a strong positive value (blue) to a strong negative value (red). Consider the closeup view of the Laplacian of the letter T shown in Fig. 12.19b. We generate a horizontal cross-section of the stem of the letter T at  $v = 360$

**Difference of Gaussians.** The Laplacian of Gaussian can be approximated by the difference of two Gaussian functions

$$\text{DiffG}(u, v; \sigma_1, \sigma_2) = G(\sigma_1) - G(\sigma_2) = \frac{1}{2\pi\sigma_1^2\sigma_2^2} \left( \sigma_2^2 e^{-\frac{u^2+v^2}{2\sigma_1^2}} - \sigma_1^2 e^{-\frac{u^2+v^2}{2\sigma_2^2}} \right)$$

where  $\sigma_1 > \sigma_2$  and commonly  $\sigma_1 = 1.6\sigma_2$ . Figure 12.13e and f compares the LoG and DiffG kernels respectively.

This approximation is useful in scale-space sequences which will be discussed in Sect. 13.3.2. Consider an image sequence  $I(k)$  where  $I(k+1) = G(\sigma) \otimes I(k)$ , that is, the images are increasingly smoothed. The difference between any two images in the sequence is therefore equivalent to  $\text{DiffG}(\sqrt{2}\sigma, \sigma)$  applied to the original image.

```
>> p = lap(360,570:600);
>> plot(570:600, p, '-*');
```

which is shown in Fig. 12.19c. We see that the zero values of the second derivative lies *between* the pixels. A zero crossing detector selects pixels adjacent to the zero crossing points

```
>> zc = zcross(lap);
```

and this is shown in Fig. 12.19d. We see that the edges appear twice. Referring again to Fig. 12.19c we observe a weak zero crossing in the interval  $u \in [573, 574]$  and a much more definitive zero crossing in the interval  $u \in [578, 579]$ .

A fundamental limitation of all edge detection approaches is that intensity edges do not necessarily delineate the boundaries of objects. The object may have poor contrast with the background which results in weak boundary edges. Conversely the object may have a stripe on it which is not its edge. Shadows frequently have very sharp edges but are not real objects. Object texture will result in a strong output from an edge detector at points not just on its boundary, as for example with the bricks in Fig. 12.15b.

### 12.4.2 Template Matching

In our discussion so far we have used kernels that represent mathematical functions such as the Gaussian and its derivative and its Laplacian. We have also considered the convolution kernel as a matrix, a 3-dimensional shape and even an image as shown in Fig. 12.13a. In this section we will consider that the kernel is *an image* or a part of an image and such a kernel is referred to as a template. In template matching we wish to find which parts of the input image are most similar to the template.

Template matching is shown schematically in Fig. 12.20. Each pixel in the output image is given by

$$O[u, v] = s(T, W), \quad \forall(u, v) \in I$$

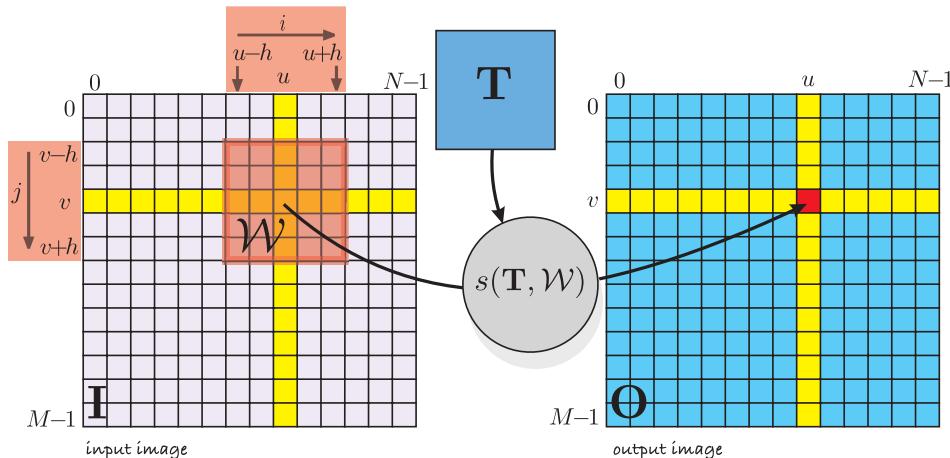
where  $T$  is the  $w \times w$  template, the pattern of pixels we are looking for, with odd side length  $w = 2h + 1$ , and  $W$  is the  $w \times w$  window centred at  $(u, v)$  in the input image. The function  $s(I_1, I_2)$  is a scalar measure that describes the *similarity* of two equally sized images  $I_1$  and  $I_2$ .

A number of common similarity measures<sup>4</sup> are given in Table 12.1. The most intuitive are computed simply by computing the pixel-wise difference  $T - W$  and taking

These measures can be augmented with a Gaussian weighting to deemphasize the differences that occur at the edges of the two windows.

**David Marr (1945–1980)** was a British neuroscientist and psychologist who synthesized results from psychology, artificial intelligence, and neurophysiology to create the discipline of Computational Neuroscience. He studied mathematics at Trinity College, Cambridge and his PhD in physiology was concerned with modeling the function of the cerebellum. His key results were published in three journal papers between 1969 and 1971 and formed a theory of the function of the mammalian brain much of which remains relevant today. In 1973 he was a visiting scientist in the Artificial Intelligence Laboratory at MIT and later became a professor in the Department of Psychology. His attention shifted to the study of vision and in particular the so-called early visual system.

He died of leukemia at age 35 and his book *Vision: A computational investigation into the human representation and processing of visual information* (Marr 2010) was published after his death.



**Fig. 12.20.**  
Spatial image processing operations. The red shaded region shows the window  $\mathcal{W}$  that is the set of pixels used to compute the output pixel (shown in red)

#### Sum of absolute differences

$$\text{SAD} \quad s = \sum_{(u,v) \in I} |I_1[u, v] - I_2[u, v]|$$

sad

$$\text{ZSAD} \quad s = \sum_{(u,v) \in I} |(I_1[u, v] - \bar{I}_1) - (I_2[u, v] - \bar{I}_2)|$$

zsad

#### Sum of squared differences

$$\text{SSD} \quad s = \sum_{(u,v) \in I} (I_1[u, v] - I_2[u, v])^2$$

ssd

$$\text{ZSSD} \quad s = \sum_{(u,v) \in I} ((I_1[u, v] - \bar{I}_1) - (I_2[u, v] - \bar{I}_2))^2$$

zssd

#### Cross correlation

$$\text{NCC} \quad s = \frac{\sum_{(u,v) \in I} I_1[u, v] \cdot I_2[u, v]}{\sqrt{\sum_{(u,v) \in I} I_1^2[u, v] \cdot \sum_{(u,v) \in I} I_2^2[u, v]}}$$

ncc

$$\text{ZNCC} \quad s = \frac{\sum_{(u,v) \in I} (I_1[u, v] - \bar{I}_1) \cdot (I_2[u, v] - \bar{I}_2)}{\sqrt{\sum_{(u,v) \in I} (I_1[u, v] - \bar{I}_1)^2 \cdot \sum_{(u,v) \in I} (I_2[u, v] - \bar{I}_2)^2}}$$

zncc

**Table 12.1.**

Similarity measures for two equal-sized image regions  $I_1$  and  $I_2$ . The Z-prefix indicates that the measure accounts for zero-offset or the difference in mean of the two images (Banks and Corke 2001).  $\bar{I}_1$  and  $\bar{I}_2$  are the mean of image regions  $I_1$  and  $I_2$  respectively. Toolbox functions are indicated in the last column

the sum of the absolute differences (SAD) or the sum of the squared differences (SSD). These metrics are zero if the images are identical and increase with dissimilarity. It is not easy to say what value of the measure constitutes a poor match but a ranking of similarity measures can be used to determine the *best* match.

More complex measures such as normalized cross-correlation yield a score in the interval  $[-1, +1]$  with  $+1$  for identical regions. In practice a value greater than 0.8 is considered to be a good match. Normalized cross correlation is however computationally more expensive – requiring multiplication, division and square root operations. Note that it is possible for the result to be undefined if the denominator is zero, which occurs if the the elements of either  $I_1$  or  $I_2$  are identical.

If  $I_2 \equiv I_1$  then it is easily shown that  $\text{SAD} = \text{SSD} = 0$  and  $\text{NCC} = 1$  indicating a perfect match. To illustrate we will create a  $51 \times 51$  template

```
>> T = iroi(lena, [240 290; 250 300]);
```

and evaluate the three common measures

```
>> sad(T, T)
ans =
    0
>> ssd(T, T)
ans =
    0
>> ncc(T, T)
ans =
    1
```

Now consider the case where the two images are of the same scene but one image is darker than the other – the illumination or the camera exposure has changed. In this case  $I_2 = \alpha I_1$  and now

```
>> sad(T, T*0.9)
ans =
    132.8090
>> ssd(T, T*0.9)
ans =
    7.3307
```

these measure indicate a high degree of disimilarity. However the normalized cross-correlation

```
>> ncc(T, T*0.9)
ans =
    1
```

is invariant to the change in intensity.

Next consider that the pixel values are also offset<sup>◀</sup> so that  $I_2 = \alpha I_1 + \beta$  and we find that

```
>> sad(T, T+0.1)
ans =
    260.1000
>> ssd(T, T+0.1)
ans =
    26.0100
>> ncc(T, T+0.1)
ans =
    0.9990
```

all measures indicate a degree of disimilarity. The problematic offset can be dealt with by first subtracting from each of  $T$  and  $\mathcal{W}$  their mean value

```
>> zsad(T, T+0.1)
ans =
    1.7300e-11
>> zssd(T, T+0.1)
ans =
    1.1507e-25
>> zncc(T, T+0.1)
ans =
    1.0000
```

and these measures indicate a high degree of similarity. The  $z$ -prefix denotes variants of the similarity measures described above that are invariant to intensity offset. Only the ZNCC measure

```
>> zncc(T, T*0.9+0.1)
ans =
    1.0000
```

is invariant to both gain and offset variation. All these methods will fail if the images have even a small change in relative rotation or scale.

Consider the problem from the well known children's book "Where's Wally" or "Where's Waldo" – the fun is trying to find Wally's face in a crowd

This could be due to an incorrect black level setting. A camera's black level is the value of a pixel corresponding to no light and is often  $>0$ .

```
>> crowd = imread('wheres-wally.png', 'double');
>> idisp(crowd)
```

Fortunately we know roughly what he looks like and the template

```
>> T = imread('wally.png', 'double');
>> idisp(T)
```

was extracted from a different image and scaled so that the head is approximately the same width as other heads in the crowd scene (around 21 pixel wide).

The similarity of our template `T` to every possible window location is computed by

```
>> S = isimilarity(T, crowd, @zncc);
```

using the matching measure ZNCC. The result

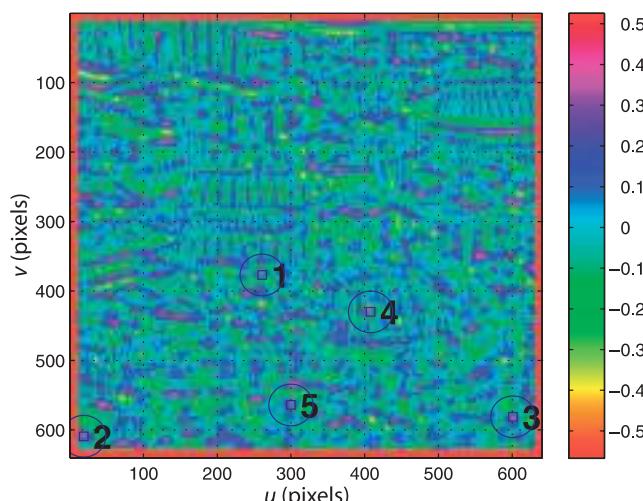
```
>> idisp(S, 'colormap', 'jet', 'bar')
```

is shown in Fig. 12.21 and the pixel color indicates the ZNCC similarity as indicated by the color bar. We can see a number of spots of high similarity (red) which are candidate positions for Wally. The peak values, with respect to a local  $3 \times 3$  window, are

```
>> [mx,p] = peak2(S, 1, 'npeaks', 5);
>> mx
mx =
    0.5258
    0.5230
    0.5222
    0.5032
    0.5023
```

in descending order. The second argument specifies the window half-width  $h = 1$  and the third argument specifies the number of features to return. The largest value 0.5258 is the similarity of the strongest match found. These matches occur at the coordinates  $(u, v)$  given by the first return value `p` and we can highlight these points on the scene

```
>> idisp(crowd);
>> plot_circle(p, 30, 'fillcolor', 'b', 'alpha', 0.3, 'edgecolor', 'none')
>> plot_point(p, 'sequence', 'bold', 'textsize', 24, 'textcolor', 'k')
```

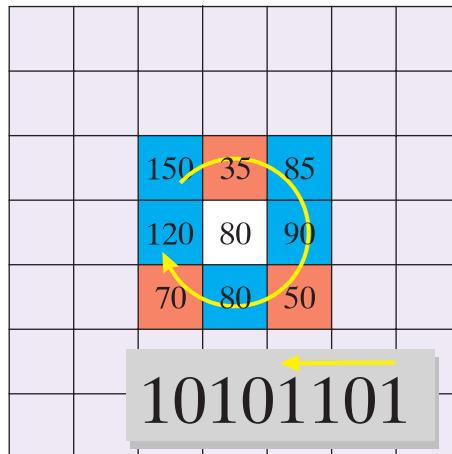


**Fig. 12.21.**  
Similarity image `S` with top five Wally candidates marked. The color bar indicate the similarity scale. Note the border of indeterminate values where the template window falls off the edge of the input image

**Fig. 12.22.**

Example of census and rank transform for a  $3 \times 3$  window. Pixels are marked red or blue if they are less than or greater than or equal to the centre pixel respectively. These boolean values are then packed into a binary word, in the direction shown, from least significant bit upwards.

The census value is  $10101101_2$  or decimal 173. The rank transform value is the total number of one bits and is 5



using transparent blue circles that are numbered sequentially. The best match at (261, 377) is in fact the correct answer – we found Wally! It is interesting to look at the other highly ranked candidates. Numbers two and three at the bottom of the image are people also wearing baseball caps who look quite similar.

There are some important points to note from this example. The images have quite low resolution and the template is only  $21 \times 25$  – it is a very crude likeness to Wally. The match is not a strong one – only 0.5258 compared to the maximum possible value of 1.0 and there are several contributing factors. The matching measure is not invariant to scale, that is, as the relative scale (zoom) changes the similarity score falls quite quickly. In practice perhaps a 10–20% change in scale between T and W can be tolerated. For this example the template was only approximately scaled. Secondly, not all Wallys are the same. Wally in the template is facing forward but the Wally we found in the image is looking to our left. Another problem is that the square template typically includes pixels from the background as well as the object of interest. As the object moves the background pixels may change, leading to a lower similarity score. This is known as the mixed pixel problem. Ideally the template should bound the object of interest as tightly as possible. In practice another problem arises due to perspective distortion. A square pattern of pixels in the centre of the image will appear keystone shaped at the edge of the image and thus will match less well with the square template.

Non-parametric similarity measures are more robust to the mixed pixel problem. Two common measures from this class are the census metric and the rank transform. For the census metric each window, template and candidate, is mapped to a bit string, and each bit corresponds to one pixel in that window as shown in Fig. 12.22. If a pixel is greater than the centre pixel its corresponding bit is set to one, else it is zero. For a  $w \times w$  window the string will be  $w^2 - 1$  bits long. ▶ The two bit strings are compared using a Hamming distance which is the number of bits that are different. This can be computed by counting the number of set bits in the exclusive-or of the two bit strings. Thus very few arithmetic operations are required compared to the more conventional methods – no square roots or division – and such algorithms are amenable to implementation in special purpose hardware or FPGAs. Another advantage is that intensities are considered relative to the centre pixel of the window making it invariant to overall changes in intensity or gradual intensity gradients.

The rank transform maps each window to a scalar which is the number of elements in the window that are greater than the centre pixel. This measure captures the essence of the region surrounding the centre pixel, and like the census measure it is invariant to overall changes in intensity since it is based on local relative grey scale

For a 32-bit integer `uint32` this limits the window to  $5 \times 5$  unless a sparse mapping is adopted (Humenberger et al. 2009). A 64-bit integer `uint64` supports a  $7 \times 7$  window.

values. The rank transform is typically used as a pre-processing step applied to each of the images before using a simple classical similarity measure such as SAD. The Toolbox function `isimilarity` supports these metrics using the '`census`' and '`rank`' options.

A common problem with template matching is that false matches can occur. In the example above the second candidate had a similarity score only 0.5% lower than the first, the fifth candidate was only than 5% lower. In practice a number of rules are applied before a match is accepted: the similarity must exceed some threshold and the first candidate must exceed the second candidate by some factor. Another approach is to bring more information to bear on the problem such as known motion of the camera or object. For example if we were tracking Wally from frame to frame in an image sequence then we would pick the best Wally closest to the previous location he was found. Alternatively we could create a motion model and assume he moves approximately the same distance and direction between successive frames. Then we would predict his future position and pick the Wally closest to that predicted position. However we would have to deal with practical difficulties such as Wally stopping, changing direction or becoming obscured.

### 12.4.3 Non-Linear Operations

Another class of spatial operations is based on non-linear functions of pixels within the window. For example

```
>> out = iwindow(lena, 3, 'var');
```

computes the variance of the pixels in *every*  $7 \times 7$  window. The arguments specify the window half-width  $h = 3$  and the builtin MATLAB® function `var`. The function is called with a  $49 \times 1$  vector argument comprising the pixels in the window arranged as a column vector and the function's return value becomes the corresponding output pixel value. This operation acts as an edge detector since it has a low value for homogeneous regions irrespective of their brightness. It is however computationally expensive because the `var` function is called over 260 000 times. Any MATLAB® function, builtin or your own M-file, that accepts a vector input and returns a scalar can be used in this way.

Rank filters sort the pixels within the window by value and return the specified element from the sorted list. The maximum value over a  $5 \times 5$  window about each pixel is the first ranked pixel in the window

```
>> mx = irank(lena, 1, 2);
```

where the arguments are the rank and the window half-width  $h = 2$ . The median over a  $5 \times 5$  window is the twelfth in rank

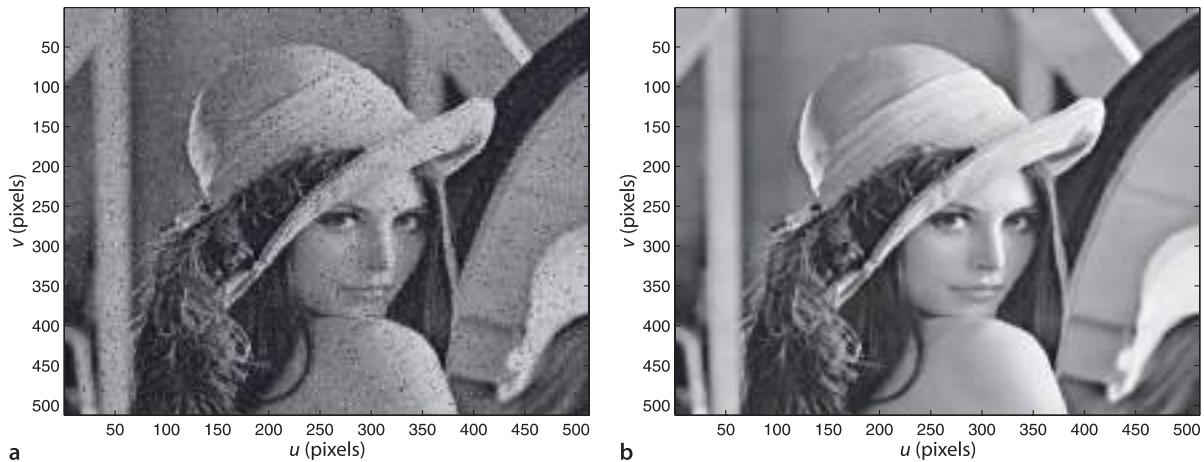
```
>> med = irank(lena, 12, 2);
```

and is useful as a filter to remove impulse-type noise. We can demonstrate this by adding impulse noise to a copy of an image

```
>> spotty = lena;
>> npix = prod(size(lena));
>> spotty(round(rand(5000,1)*(npix-1)+1)) = 0;
>> spotty(round(rand(5000,1)*(npix-1)+1)) = 1.0;
>> idisp(spotty)
```

and this is shown in Fig. 12.23a. We have set 5 000 random pixels to be zero, and another 5 000 random pixels to the maximum value. This type of noise is often referred to as impulse noise or salt and pepper noise. We apply a  $3 \times 3$  median filter

```
>> idisp( irank(spotty, 5, 1) )
```



**Fig. 12.23.** Median filter cleanup of impulse noise. **a** Corrupted image; **b** median filtered result

and the result shown in Fig. 12.23b is considerably improved. A similar effect could have been obtained by smoothing but that would tend to blur the image, median filtering has the advantage of preserving edges in the scene.

The third argument to `irank` can be a matrix instead of a scalar and this allows for some very powerful operations. For example

```
>> M = ones(3,3);
>> M(2,2) = 0
M =
    1     1     1
    1     0     1
    1     1     1
>> mxn = irank(lena, 1, M);
```

specifies the first in rank (maximum) over a *subset* of pixels from the window corresponding to the non-zero elements of `M`. In this case `M` specifies the eight neighbouring pixels but not the centre pixel. The result `mxn` is the maximum of the eight neighbours of each corresponding pixel in the input image. We can use this

```
>> idisp(lena > mxn)
```

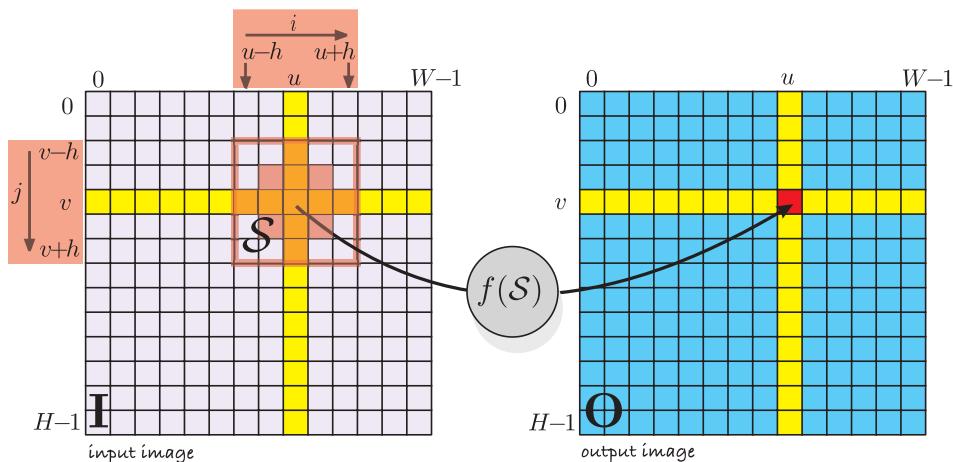
to display all those points where the pixel value is greater than its local neighbours and performs non-local maxima suppression. These correspond to local maxima, or peaks if the image is considered as a surface. This mask matrix is very similar to a structuring element which we will meet in the next section.

## 12.5 Mathematical Morphology

Mathematical morphology is a class of non-linear spatial operators shown schematically in Fig. 12.24. Each pixel in the output matrix is a function of a *subset* of pixels in a region surrounding the corresponding pixel in the input image

$$O[u, v] = f(I[u + i, v + j]), \quad \forall(i, j) \in S, \quad \forall(u, v) \in I \quad (12.7)$$

where  $S$  is the structuring element, typically a  $w \times w$  square region with odd side length  $w = 2h + 1$  where  $h \in \mathbb{Z}^+$  is the half-width. The structuring element is similar to the convolution kernel discussed previously except that the function  $f(\cdot)$  is applied only to a specified subset of pixels within the window. The selected pixels are those for which the corresponding values of the structuring element are non-zero – these are shown in red in Fig. 12.24. Mathematical morphology, as its name implies, is concerned with the form or *shape* of objects in the image.



**Fig. 12.24.**  
Morphological image processing operations. The operation is defined only for the selected elements (red) within the structuring element (red outlined square)

The easiest way to explain the concept is with a simple example, in this case a synthetic binary image created by the script

```
>> eg_morph1
>> idisp(im)
```

which is shown, repeated, down the column of Fig. 12.25. The structuring element is shown in red at the end of each row. If we consider the top most row, the structuring element is a square

```
>> S = ones(5,5);
```

and is applied to the original image using the minimum operation

```
>> mn = imorph(im, S, 'min');
```

and the result is shown in the second column. For each pixel in the input image we take the *minimum* of all pixels in the  $5 \times 5$  window. If *any* of those neighbours is zero the resulting pixel will be zero. The result is dramatic – two objects have disappeared entirely and the two squares have become separated and smaller. The two objects that disappeared were not *consistent* with the shape of the structuring element. This is where the connection to morphology or shape comes in – only shapes that could *contain* the structuring element will be present in the output image.

The structuring element could define a circle, an annulus, a 5-pointed star, a line segment 20 pixels long at  $30^\circ$  to the horizontal or a duck – the technique allows very powerful shape-based filters to be created. The second row shows the results for a larger square structuring element which has resulted in the complete elimination of the small square and the further reduction of the large square. The third row shows the results for a structuring element which is a horizontal line segment 14 pixel wide, and the only remaining shape is the long horizontal line.

The operation we just performed is often known as erosion since large objects are eroded or become smaller – in this case the  $5 \times 5$  structuring element has caused two pixels to be *shaved off* all the way around the perimeter of each shape. The small square, originally  $5 \times 5$ , is now only  $1 \times 1$ . If we repeated the operation the small square would disappear entirely, and the large square would be reduced even further.

The inverse operation is dilation which makes objects larger. In Fig. 12.25 we apply dilation to the second column results

```
>> mx = imorph(mn, S, 'max');
```

and the results are shown in the third column. For each pixel in the input image we take the *maximum* of all pixels in the  $5 \times 5$  window. If *any* of those neighbours is one the

The half width of the structuring element.

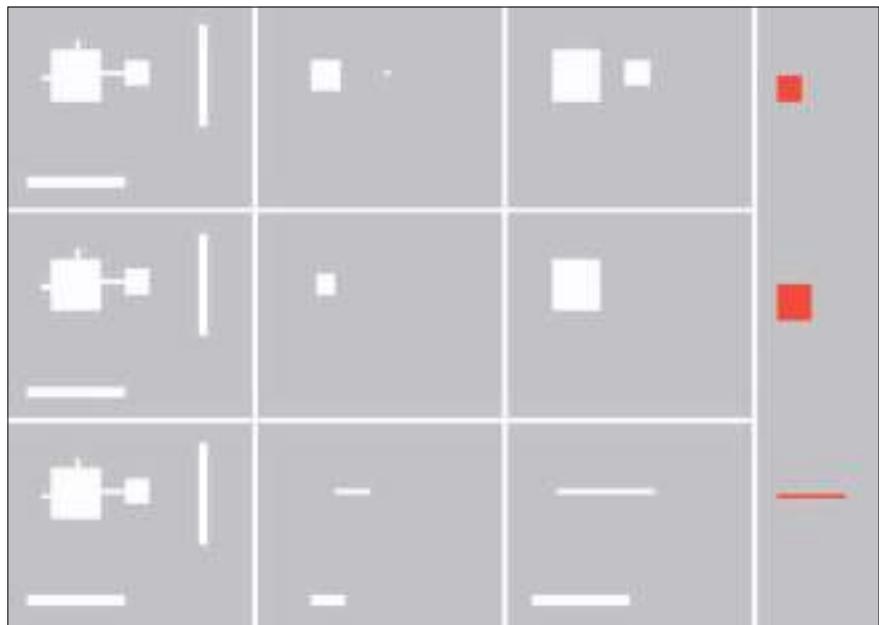


Fig. 12.25.

Mathematical morphology example. Pixels are either 0 (grey) or 1 (white). Each column corresponds to processing using the structuring element, shown at the end in red. The first column is the original image, the second column is after erosion by the structuring element, and the third column is after the second column is dilated by the structuring element

resulting pixel will be one. In this case we see that the two squares have returned to their original size, but the large square has lost its protrusions.

Morphological operations are often written in operator form. Erosion is

$$O = I \ominus S$$

where in Eq. 12.7  $f(\cdot) = \min(\cdot)$ , and dilation is

$$O = I \oplus S$$

where in Eq. 12.7  $f(\cdot) = \max(\cdot)$ .

Erosion and dilation are related by

$$A \oplus B = \overline{\overline{A} \ominus \overline{B}}$$

$B'[i, j] = B[j, i]$  where  $i = 0, j = 0$  is the centre of the structuring element.

where the bar denotes the logical complement of the pixel values and  $B'$  indicates the reflection of  $B$  about its centre. ▶ Essentially this states that eroding the white pixels is the same as dilating the dark pixels and vice versa. The morphological operations are associative

$$(A \oplus B) \oplus C = A \oplus (B \oplus C)$$

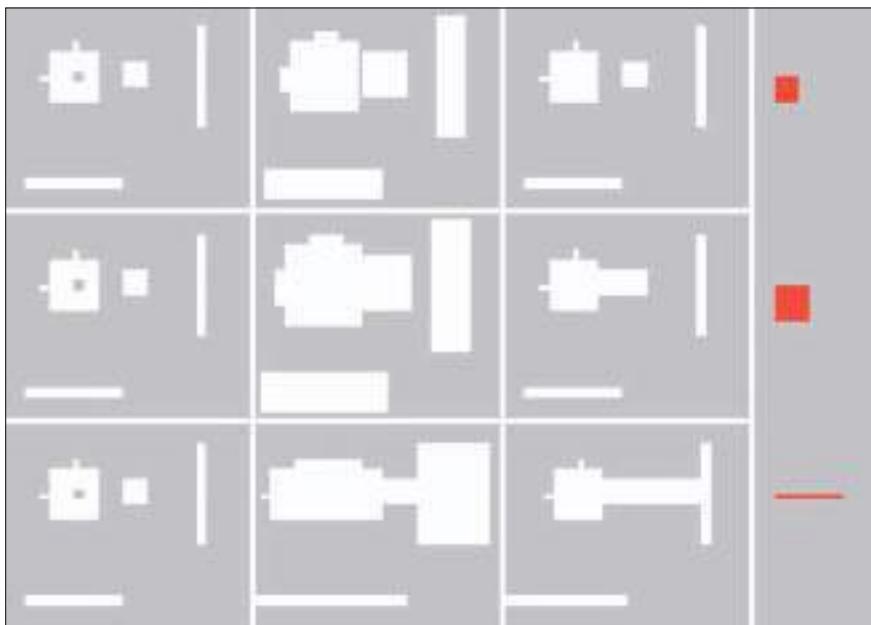
$$(A \ominus B) \ominus C = A \ominus (B \oplus C)$$

which means that successive erosion or dilation with a structuring element can be equivalent to the application of a single larger structuring element, but the former is computationally cheaper. ▶ These operations are also known as Minkowski subtraction and addition respectively. The shorthand functions

```
>> out = ierode(im, S);
>> out = idilate(im, S);
```

can be used instead of `imorph`.

For example a  $3 \times 3$  square structuring element applied twice is equivalent to  $5 \times 5$  square structuring element. The former involves  $2 \times (3 \times 3 \times N^2) = 18N^2$  operations whereas the latter involves  $5 \times 5 \times N^2 = 25N^2$  operations.



**Fig. 12.26.**  
Mathematical morphology example. Pixels are either 0 (grey) or 1 (white). Each row corresponds to processing using the structuring element, shown at the end in red. The first column is the original image, the second column is after dilation by the structuring element and the third column is eroded by the structuring element

The sequence of operations, erosion then dilation, is known as opening since it opens up gaps. In operator form it is written as

$$I \circ S = (I \ominus S) \oplus S$$

Not only has the opening selected particular shapes but it has also *cleaned up* the image: the squares have been separated and the protrusions on the large square have been removed since they are not consistent with the shape of the structuring element.

In Fig. 12.26 we perform the operations in the inverse order, dilation then erosion. In the first row no shapes have been lost, they grew then shrank, and the large square still has its protrusions. The hole has been filled since it is not consistent with the shape of the structuring element. In the second row, the larger structuring element has caused the two squares to join together. This sequence of operations is referred to as closing since it closes gaps and is written in operator form as

$$I \bullet S = (I \oplus S) \ominus S$$

Note that in the bottom row the two line segments have remained attached to the edge, this is due to the default behaviour in handling edge pixels.

**Dealing with edge pixels.** The problem of a convolution window near the edge of an input image was discussed on page 299. Similar problems exist for morphological spatial operations, and the Toolbox functions `imorph`, `irank` and `iwindow` support the option '`'valid'`' as does `iconv`. Other options cause the returned image to be the same size as the input image:

- '`'replicate'`' (default) the border pixel is replicated, that is, the value of the closest border pixel is used.
- '`'none'`' pixels beyond the border are not included in the set of pixels specified by the structuring element.
- '`'wrap'`' the image is assumed to wrap around, left to right, top to bottom.

These names make sense when considering what happens to white objects against a black background. For black objects the operations perform the inverse function.

Opening and closing<sup>4</sup> are implemented by the Toolbox functions `iopen` and `iclose` respectively. Unlike erosion and dilation repeated application of opening or closing is futile since those operations are idempotent

$$(I \circ S) \circ S = I \circ S$$

$$(I \bullet S) \bullet S = I \bullet S$$

### 12.5.1 Noise Removal

A common use of morphological opening is to remove noise in an image. The image

```
>> objects = imread('segmentation.png');
```

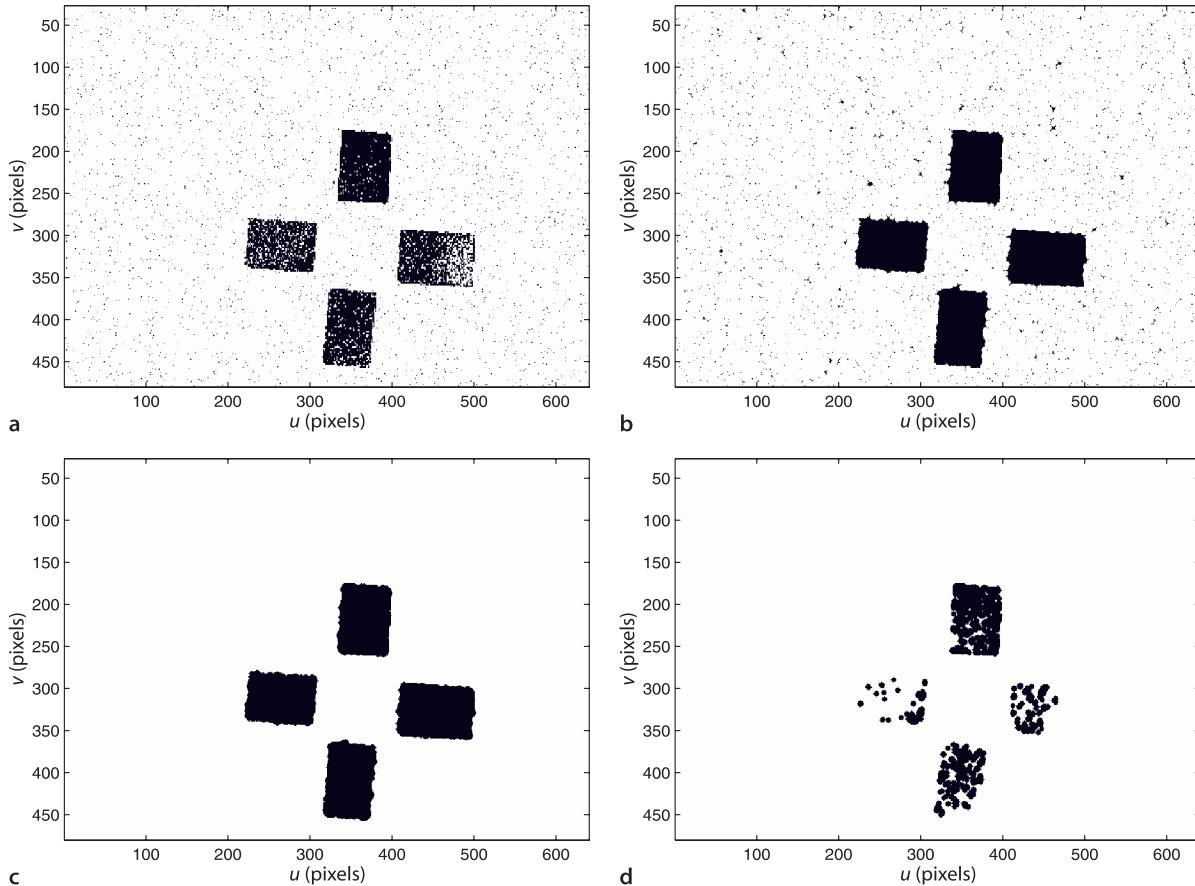
Image segmentation and binarization is discussed in Sect. 13.1.1.

shown in Fig. 12.27a is a noisy binary image from the output of a rather poor object segmentation operation.<sup>4</sup> We wish to remove the dark pixels that do not belong to the objects and we wish to fill in the holes in the four dark rectangular objects.

We choose a symmetric *circular* structuring element of radius 3

```
>> S = kcircle(3)
S =
    0     0     0     1     0     0     0
    0     1     1     1     1     1     0
    0     1     1     1     1     1     0
    1     1     1     1     1     1     1
    0     1     1     1     1     1     0
    0     1     1     1     1     1     0
    0     0     0     1     0     0     0
```

**Fig. 12.27.** Morphological cleanup. **a** Original image, **b** original after opening, **c** opening then closing, **d** closing then opening. Structuring element is a circle of radius 3



and apply a closing operation to fill the holes in the objects

```
>> closed = iclose(objects, S);
```

and the result is shown in Fig. 12.27b. The holes have been filled, but the noise pixels have grown to be small circles and some have agglomerated. We eliminate these by an opening operation

```
>> clean = iopen(closed, S);
```

and the result shown in Fig. 12.27c is a considerably cleaned up image. If we apply the operations in the inverse order, opening then closing

```
>> opened = iopen(objects, S);
>> closed = iclose(opened, S);
```

the results shown in Fig. 12.27d are much poorer. Although the opening has removed the isolated noise pixels it has removed large chunks of the targets which cannot be restored.

### 12.5.2 Boundary Detection

We can also use morphological operations to detect the edges of objects. Continuing the example from above and using the image `clean` shown in Fig. 12.27c we compute its erosion using a circular structuring element

```
>> eroded = imorph(clean, kcircle(1), 'min');
```

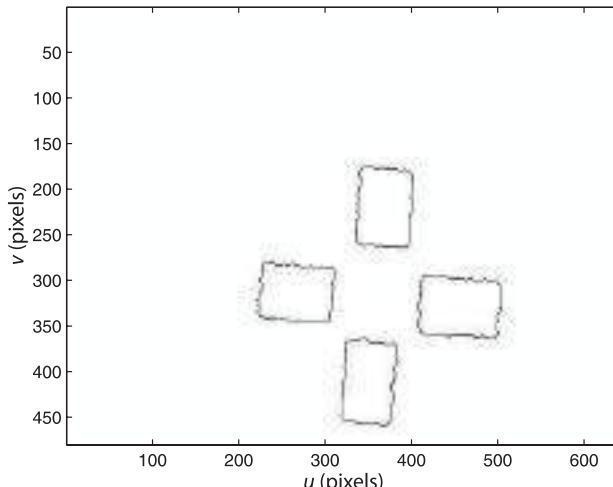
The objects in this image are slightly smaller since the structuring element has caused one pixel to be *shaved off* the outside of each object. Subtracting the eroded image from the original

```
>> idisp(clean-eroded)
```

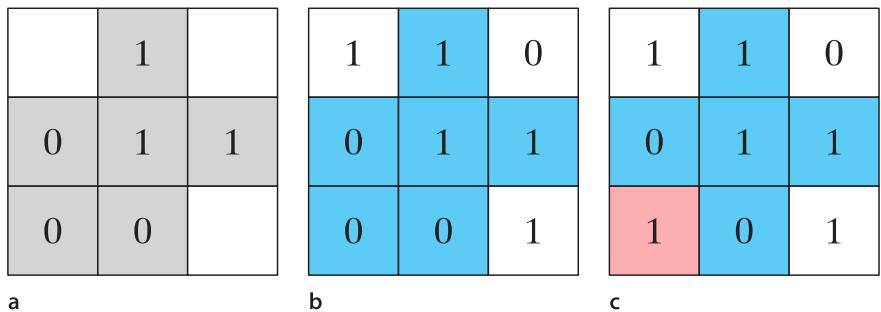
results in a layer of pixels around the edge of each object as shown in Fig. 12.28.

### 12.5.3 Hit and Miss Transform

The hit and miss transform uses a variation on the morphological structuring element. Its values are zero, one or *don't care* as shown in Fig. 12.29a. The zero and one



**Fig. 12.28.**  
Boundary detection by morphological processing. Results are shown inverted, white is zero

**Fig. 12.29.**

Hit and miss transform. **a** The structuring element has values of zero, one, or don't care which is shown in white; **b** an example of a hit; **c** an example of a miss, the pixel shown in red is inconsistent with the structuring element

pixels must exactly match the underlying image pixels in order for the result to be a one, as shown in Fig. 12.29b. If there is any mismatch of a one or zero as shown in Fig. 12.29c then the result will be zero. The Toolbox implementation is very similar to the morphological function, for example

```
out = hitmiss(image, S);
```

where the don't care elements of the structuring element are set to the special MATLAB® value `NaN`.

The hit and miss transform is used iteratively with a variety of structuring elements to perform operations such as skeletonization and linear feature detection. The skeleton of the objects is computed by

```
>> skeleton = ithin(clean);
```

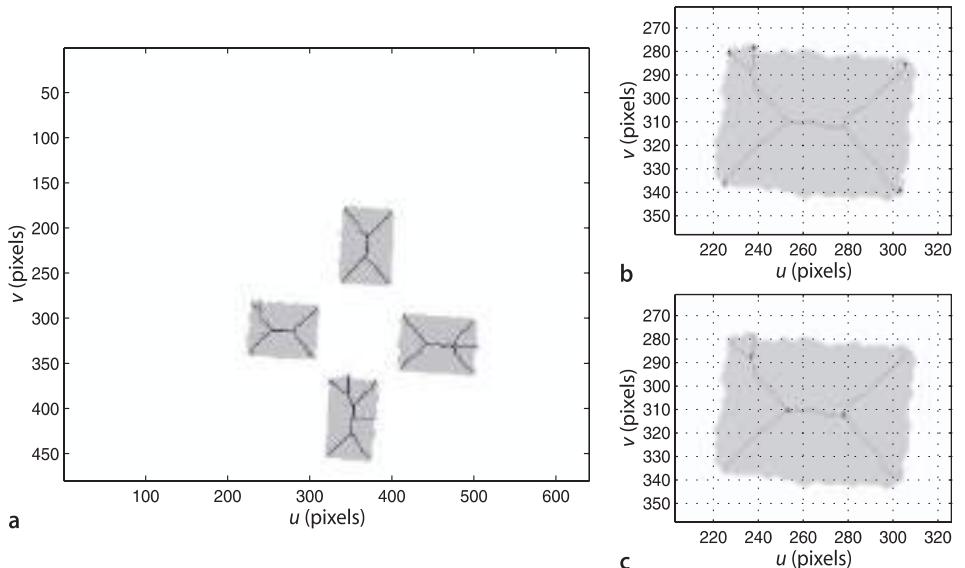
and is shown in Fig. 12.30a. The lines are a single pixel wide and are the edges of a generalized Voronoi diagram – they delineate sets of pixels according to the shape boundary they are closest to. We can then find the endpoints of the skeleton

```
>> ends = iendpoint(skeleton);
```

and also the triplepoints

```
>> joins = itriplepoint(skeleton);
```

which are points at which three lines join. These are shown in Fig. 12.30b and c respectively.

**Fig. 12.30.**

Hit and miss transform operations. **a** Skeletonization; **b** endpoint detection; **c** triple-point join detection. The images are shown inverted with the original binary image superimposed in grey. The end- and triple-points are shown as black pixels

## 12.6 Shape Changing

The final class of image processing operations that we will discuss are those that change the shape or size of an image.

### 12.6.1 Cropping

The simplest shape change of all is selecting a rectangular region from an image which is the familiar *cropping* operation. Consider the image

```
>> lena = imread('lena.pgm');
```

shown in Fig. 12.31a from which we interactively specify a region of interest or ROI

```
>> [eyes,roi] = iroi(lena);
>> idisp(eyes)
```

by clicking and dragging a selection box over the image. In this case we selected the eyes as shown in Fig. 12.31b. The corners of the selected region can be optionally returned and in this case was

```
>> roi
roi =
 239   359
 237   294
```

where the columns are the  $(u, v)$  coordinates for the top-left and bottom-right corners respectively. The rows are the  $u$ - and  $v$ -span respectively. The function can be used non-interactively by specifying a ROI

```
>> mouth = iroi(lena, [253, 330; 332, 370]);
```

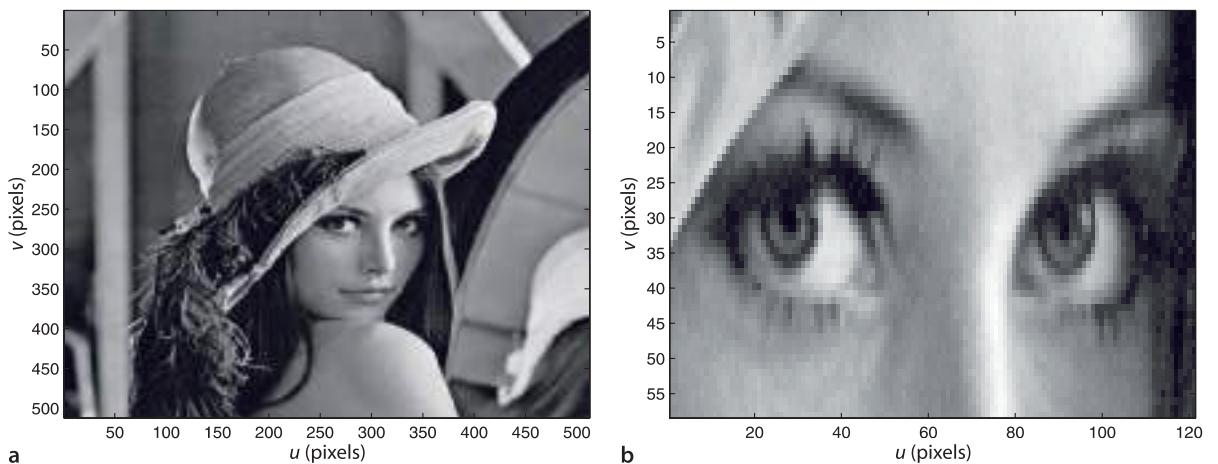
which in this case selects the mouth.

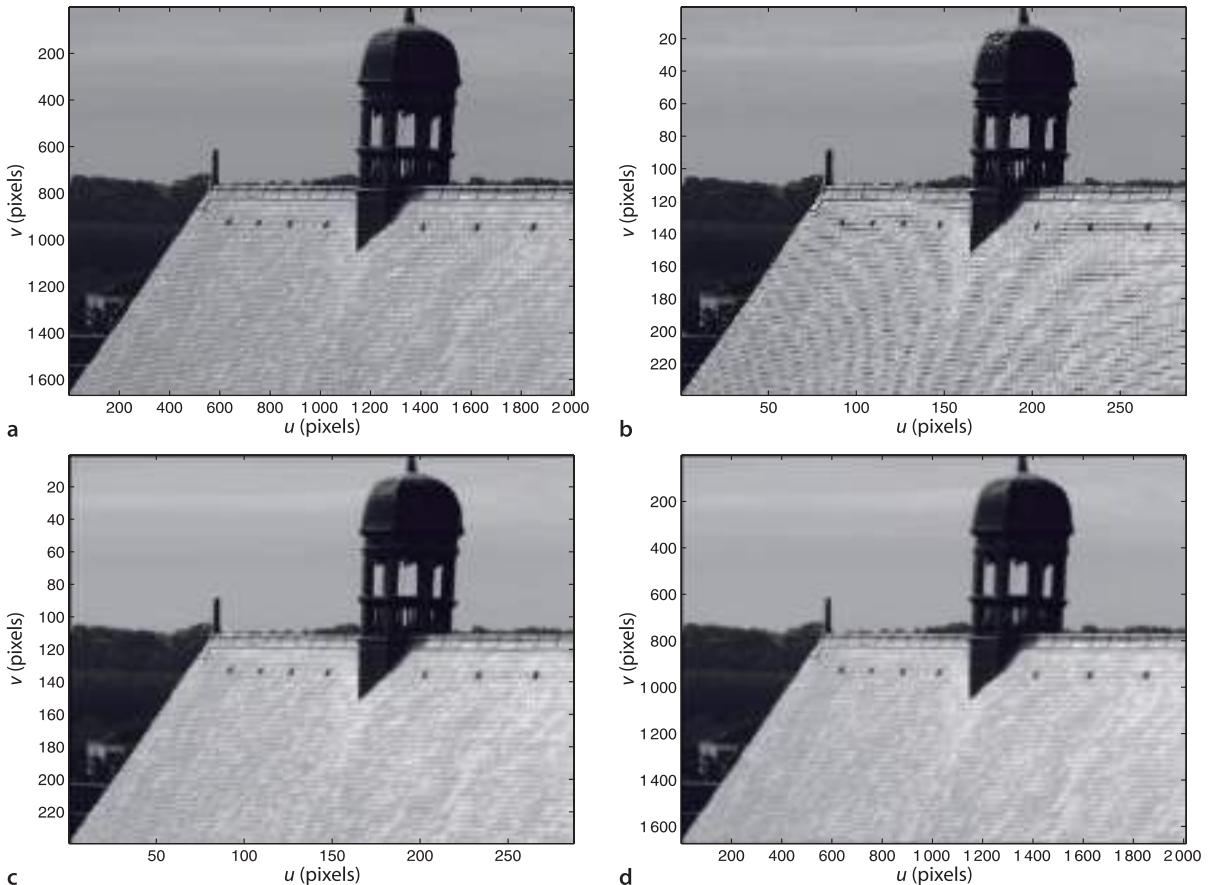
### 12.6.2 Image Resizing

Often we wish to reduce the dimensions of an image, perhaps because the large number of pixels results in long processing time or requires too much memory. We demonstrate this with a high-resolution image

```
>> roof = imread('roof.jpg', 'grey');
>> about(roof)
roof [uint8] : 1668x2009 (3351012 bytes)
```

**Fig. 12.31.** Example of region of interest or image cropping.  
a Original image, b selected region of interest





**Fig. 12.32.** Image scaling example. **a** Original image; **b** subsampled with  $m = 7$ , note the axis scaling; **c** subsampled with  $m = 7$  after smoothing; **d** image **c** restored to original size by pixel replication

which is shown in Fig. 12.32a. The simplest means to reduce image size is subsampling or decimation which selects every  $m^{\text{th}}$  pixel in the  $u$ - and  $v$ -direction, where  $m \in \mathbb{Z}^+$  is the subsampling factor. For example with  $m = 2$  an  $N \times N$  image becomes an  $N/2 \times N/2$  images which has one quarter the number of pixels of the original image.

For this example we will reduce the image size by a factor of seven in each direction

```
>> smaller = roif(1:7:end,1:7:end);
```

using standard MATLAB® indexing syntax to select every seventh row and column. The result is shown in Fig. 12.32b and we observe some pronounced curved lines on the roof which were not in the original image. These are artefacts of the sampling process. Subsampling reduces the spatial sampling rate of the image which can lead to spatial aliasing of high-frequency components due to texture or sharp edges. To ensure that the Shannon-Nyquist sampling theorem is satisfied an anti-aliasing low-pass spatial filter must be applied to reduce the spatial bandwidth of the image before it is subsampled.◀ This is another use for image blurring and the Gaussian kernel is a suitable low-pass filter for this purpose. The combined operation of smoothing and subsampling is implemented in the Toolbox by

```
>> smaller = idecimate(roof, 7);
```

and the results for  $m = 7$  are shown in Fig. 12.32c. We note that the curved line artefacts are no longer present.

The inverse operation is pixel replication, where each input pixel is replicated as an  $m \times m$  tile in the output image

```
>> bigger = ireplicate( smaller, 7 );
```

Any realizable low-pass filter has a finite response above its *cutoff frequency*. In practice the cutoff frequency is selected to be far enough below the theoretical cutoff that the filter's response at the Nyquist frequency is *sufficiently small*. As a rule of thumb for subsampling with  $m = 2$  a Gaussian with  $s = 1$  is used.

which is shown in Fig. 12.32d and appears a little *blocky* along the edge of the roof and along the skyline. The decimation stage removed 98% of the pixels and restoring the image to its original size has not added any new information.► However we could make the image easier on the eye by smoothing

```
>> smoother = ismooth( bigger, 4);
```

which will attenuate the high frequency information at the edges of the blocks.

We can perform the same function using the Toolbox function `iscale` which scales an image by an arbitrary factor  $m \in \mathbb{R}^+$  for example

```
>> smaller = iscale(lena, 0.1);
>> bigger = iscale(smaller, 10);
```

The second argument is the scale factor and if  $m < 1$  the image will be reduced,► and if  $m > 1$  it will be expanded.

Somewhat like the digital zoom function on a camera.

The image should be smoothed first using the '`smooth`' option to set the width of the Gaussian kernel.

### 12.6.3 Image Pyramids

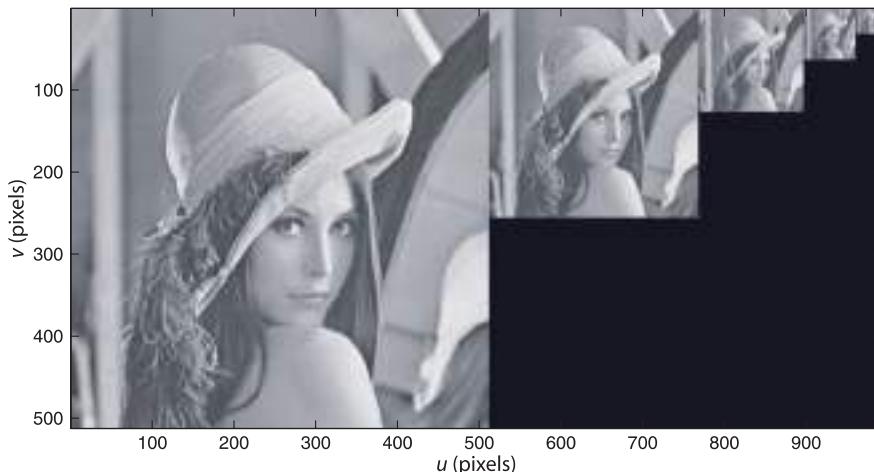
An important concept in computer vision, and one that we return to in the next chapter is scale space. The Toolbox function `ipyramid` returns a pyramidal decomposition of the input image

```
>> p = ipyramid(lena)
p =
  Columns 1 through 5
  [512x512 double]    [256x256 double]    [128x128 double]
  [64x64 double]      [32x32 double]

  Columns 6 through 10
  [16x16 double]    [8x8 double]    [4x4 double]
  [2x2 double]      [20.8959]
```

as a MATLAB® cell array containing images at successively lower resolutions. Note that the last element is the  $1 \times 1$  resolution version – a single grey pixel! These images are pasted into a composite image which is displayed in Fig. 12.33.

An image pyramid is the basis of many so-called coarse-to-fine strategies. Consider the problem of looking for a pattern of pixel values that represent some object of interest. The smallest image can be searched very quickly for the object since it has a very small number of pixels. The search is then refined using the next larger image but we now know which area of that larger image to search. The process is repeated until the object is located in the highest resolution image.



**Fig. 12.33.**  
Image pyramid, a succession of images each half (by side length) the resolution of the one to the left

#### 12.6.4 Image Warping

Image warping is a transformation of the pixel *coordinates* not their values. Warping can be used to scale image up or down in size, rotate an image or apply quite arbitrary shape changes. The coordinates of the pixel in the new view ( $u'$ ,  $v'$ ) are expressed as functions

$$u' = f_u(u, v), \quad v' = f_v(u, v) \quad (12.8)$$

of the coordinates in the original view.

Consider a simple example where the image is reduced in size by a factor of 4 in both directions and offset so that its origin, its top-left corner, is shifted to the coordinate (100, 200). We can express this concisely as

$$u' = u/4 + 100, \quad v' = v/4 + 200 \quad (12.9)$$

The coordinate matrices are such that  $U(u, v) = u$  and  $V(u, v) = v$  and are a common construct in MATLAB® see the documentation for `meshgrid`.

First we establish a pair of coordinate matrices that span the domain of the input image

```
>> [Ui,Vi] = imeshgrid(lena);
```

and another pair that span the domain of the output image which we choose arbitrarily to be  $400 \times 400$

```
>> [Uo,Vo] = imeshgrid(400, 400);
```

Now, for *every* pixel in the output image the corresponding coordinate in the input image is given by the inverse of the functions  $f_u$  and  $f_v$ . For our example the inverse of Eq. 12.9 is

$$u = 4(u' - 100), \quad v = 4(v' - 200) \quad (12.10)$$

which is implemented in matrix form in MATLAB® as

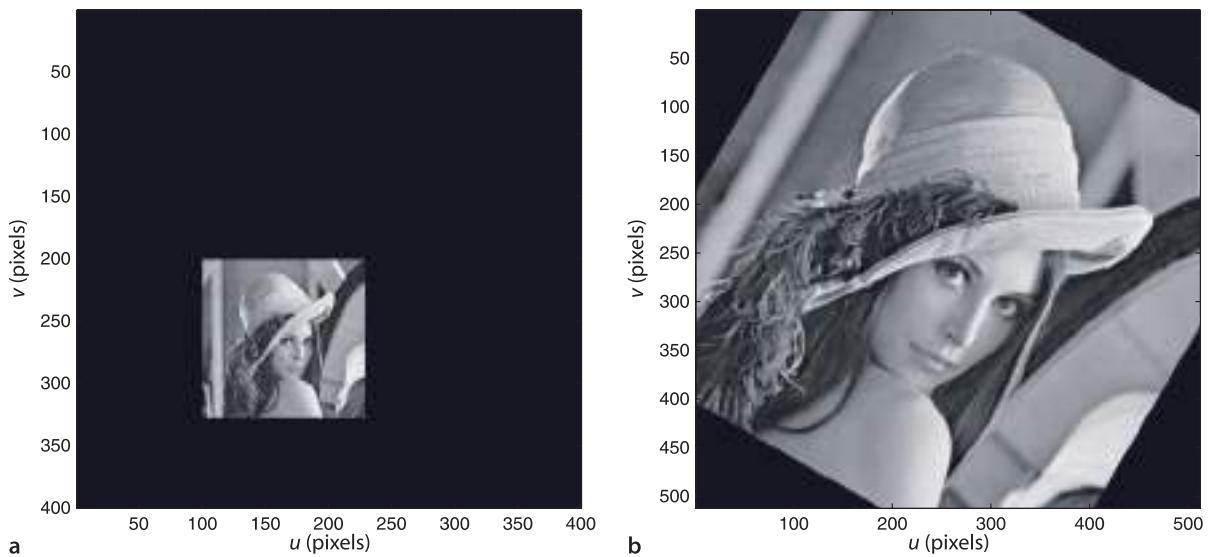
```
>> U = 4*(Uo-100); V = 4*(Vo-200);
```

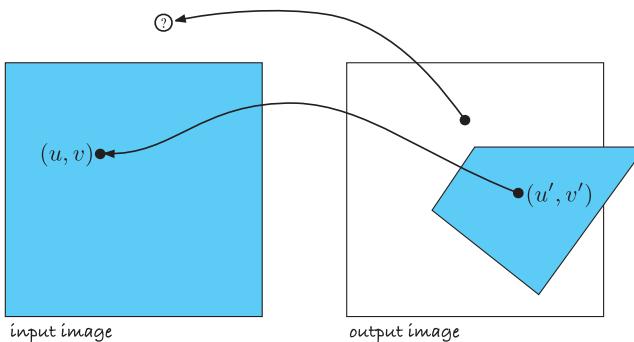
We can now warp the input image using the MATLAB® function `interp2`

```
>> lena_small = interp2(Ui, Vi, idouble(lena), U, V);
```

**Fig. 12.34.** Warped images. **a** Scaled and shifted; **b** rotated by 30° about its centre

and the result is shown in Fig. 12.34a. Note that `interp2` needs to work on a floating point image and we used `idouble` to convert the input image accordingly.



**Fig. 12.35.**

Coordinate notation for image warping. The pixel  $(u', v')$  in the output image is sourced from the pixel  $(u, v)$  in the input image as indicated by the arrow. The warped image is not necessarily polygonal, nor entirely within the second image

Some subtle things happen under the hood here. Firstly, while  $(u', v')$  are integer coordinates the input image coordinates  $(u, v)$  will not necessarily be integer. The pixel values must be interpolated<sup>10</sup> from neighbouring pixels in the input image<sup>11</sup>. Secondly, not all pixels in the output image have corresponding pixels in the input image as illustrated in Fig. 12.35. Fortunately for us `interp2` handles all these issues and pixels that do not exist in the input image are presented as black in the output image. In case of mappings that are extremely distorted it may be that many adjacent output pixels map to the same input pixel and this leads to pixelation or *blockyness* in the output image.

Now let's try something a bit more ambitious and rotate the image by  $30^\circ$  into an output image of the same size as the input image

```
>> [Uo, Vo] = imeshgrid(lena);
```

We want to rotate the image about its centre but since the origin of the input image is the top-left corner we must first change the origin to the centre, then rotate and then move the origin back to the top-left corner. The warp equation is therefore

$$\begin{pmatrix} u' \\ v' \end{pmatrix} = \underbrace{\begin{pmatrix} \cos \pi/6 & -\sin \pi/6 \\ \sin \pi/6 & \cos \pi/6 \end{pmatrix}}_{R(\pi/6)} \begin{pmatrix} u - u_c \\ v - v_c \end{pmatrix} + \begin{pmatrix} u_c \\ v_c \end{pmatrix} \quad (12.11)$$

where  $(u_c, v_c)$  is the coordinate of the image centre and  $R(\frac{\pi}{6})$  is a rotation matrix in  $SE(2)$ . This can be rearranged into the *inverse form* and implemented as

```
>> R = se2(0, 0, pi/6); uc = 256; vc = 256;
>> U = R(1,1)*(Uo-uc) + R(2,1)*(Vo-vc) + uc;
>> V = R(1,2)*(Uo-uc) + R(2,2)*(Vo-vc) + vc;
>> lena_rotated = interp2(Ui, Vi, idouble(lena), U, V);
```

and the result is shown in Fig. 12.34b. Note the direction of rotation – our definition of the  $x$ - and  $y$ -axes (parallel to the  $u$ - and  $v$ -axes respectively) is such that the  $z$ -axis is defined as being into the page making a clockwise rotation a positive angle. Also note that the corners of the original image have been lost, they fall outside the bounds of the output image.

The function `iscale` uses image warping to change image scale, and the function `irotate` uses warping to perform rotation. The example above could be achieved by

```
>> lena_rotated = irotate(lena, pi/6);
```

Finally we will revisit the lens distortion example from Sect. 11.1.1. The distorted image from the camera is the input image and will be warped to remove the distortion. We are in luck since the distortion model Eq. 11.13 is already in the inverse form. Recall that

$$u^d = u + \delta_u$$

$$v^d = v + \delta_v$$

where  $\delta_u$  and  $\delta_v$  are functions of  $(u, v)$ .

Different interpolation modes can be selected by a trailing argument to `interp2` but the default option is bilinear interpolation. A pixel at coordinate  $(u + \delta_u, v + \delta_v)$  where  $u, v \in \mathbb{Z}$  and  $\delta_u, \delta_v \in [0, 1]$  is a function of the pixels  $(u, v), (u + 1, v), (u, v + 1)$  and  $(u + 1, v + 1)$ .

The interpolation function acts as a weak anti-aliasing filter, but for very large reductions in scale the image should be smoothed first using a Gaussian kernel.

First we load the distorted image and build the coordinate matrices for the distorted and undistorted images

```
>> distorted = iread('Image18.tif', 'double');
>> [Ui,Vi] = imeshgrid(distorted);
>> Uo = Ui; Vo = Vi;
```

and then load the results of the camera calibration

```
>> load Bouguet
```

For readability we unpack the required parameters from the Calibration Toolbox variables `cc`, `fc` and `kc`

```
>> k = kc([1 2 5]); p = kc([3 4]);
>> u0 = cc(1); v0 = cc(2); fpix_u = fc(1); fpix_v = fc(2);
```

for radial and tangential distortion vectors, principal point and focal length in pixels.

Next we convert pixel coordinates to normalized image coordinates ▶

In units of metres with respect to the camera's principal point.

```
>> u = (Uo-u0) / fpix_u;
>> v = (Vo-v0) / fpix_v;
```

The radial distance of the pixels from the principal point is then

```
>> r = sqrt( u.^2 + v.^2 );
```

and the pixel coordinate errors due to distortion are

```
>> delta_u = u .* (k(1)*r.^2 + k(2)*r.^4 + k(3)*r.^6) + ...
2*p(1)*u.*v + p(2)*(r.^2 + 2*u.^2);
>> delta_v = v .* (k(1)*r.^2 + k(2)*r.^4 + k(3)*r.^6) + ...
p(1)*(r.^2 + 2*v.^2) + 2*p(1)*u.*v;
```

The distorted pixel coordinates in metric units are

```
>> ud = u + delta_u; vd = v + delta_v;
```

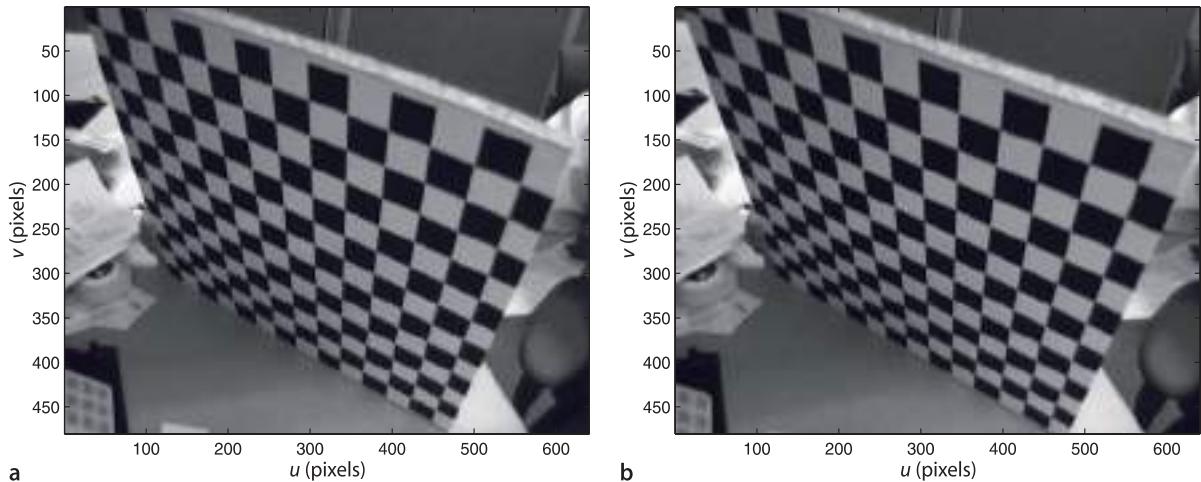
which we convert back to pixel coordinates

```
>> U = ud * fpix_u + u0;
>> V = vd * fpix_v + v0;
```

and finally apply the warp

```
>> undistorted = interp2(Ui, Vi, distorted, U, V);
```

The results are shown in Fig. 12.36. The change is quite subtle, but is most pronounced at the edges and corners of the image where  $r$  is the greatest.



**Fig. 12.36.** Warping to undistort an image. **a** Original distorted image; **b** corrected image. Note that the top edge of the target has become a straight lines (Example from Bouguet's Camera Calibration Toolbox, image number 18)

## 12.7 Wrapping Up

In this chapter we learnt how to acquire images from a variety of sources such as image files, movie files, video cameras and the internet, and load them into the MATLAB® workspace. We then discussed a large number of processing operations and a taxonomy of these is shown in Fig. 12.37. Operations on a single image include: unary arithmetic operations, type conversion, various color transformations and greylevel stretching; non-linear operations such as histogram normalization and gamma encoding or decoding; and logical operations such as thresholding. We also discussed operations on pairs of images such as green screening, background estimation and moving object detection.

The largest and most diverse class of operations were spatial operators. We discussed convolution which can be used to smooth an image and to detect edges. Smoothing is useful to reduce the effect of image noise in edge detection and also to low-pass filter an image prior to decimation. Non-linear operations can be used to perform template matching and to compute rank statistics over windows. A particular example of rank statistics is the median filter which was shown to reduce certain types of image noise. A related technique is mathematical morphology which can be used to filter images based on shape, to cleanup binary images and to perform skeletonization.

Finally we discussed shape changing operations such as regions of interest, scale changing and the problems that can arise due to aliasing, and generalized image warping which can be used for rotation or image undistortion. All these image processing techniques are the foundations of feature extraction algorithms that we discuss in the next chapter.

### Further Reading

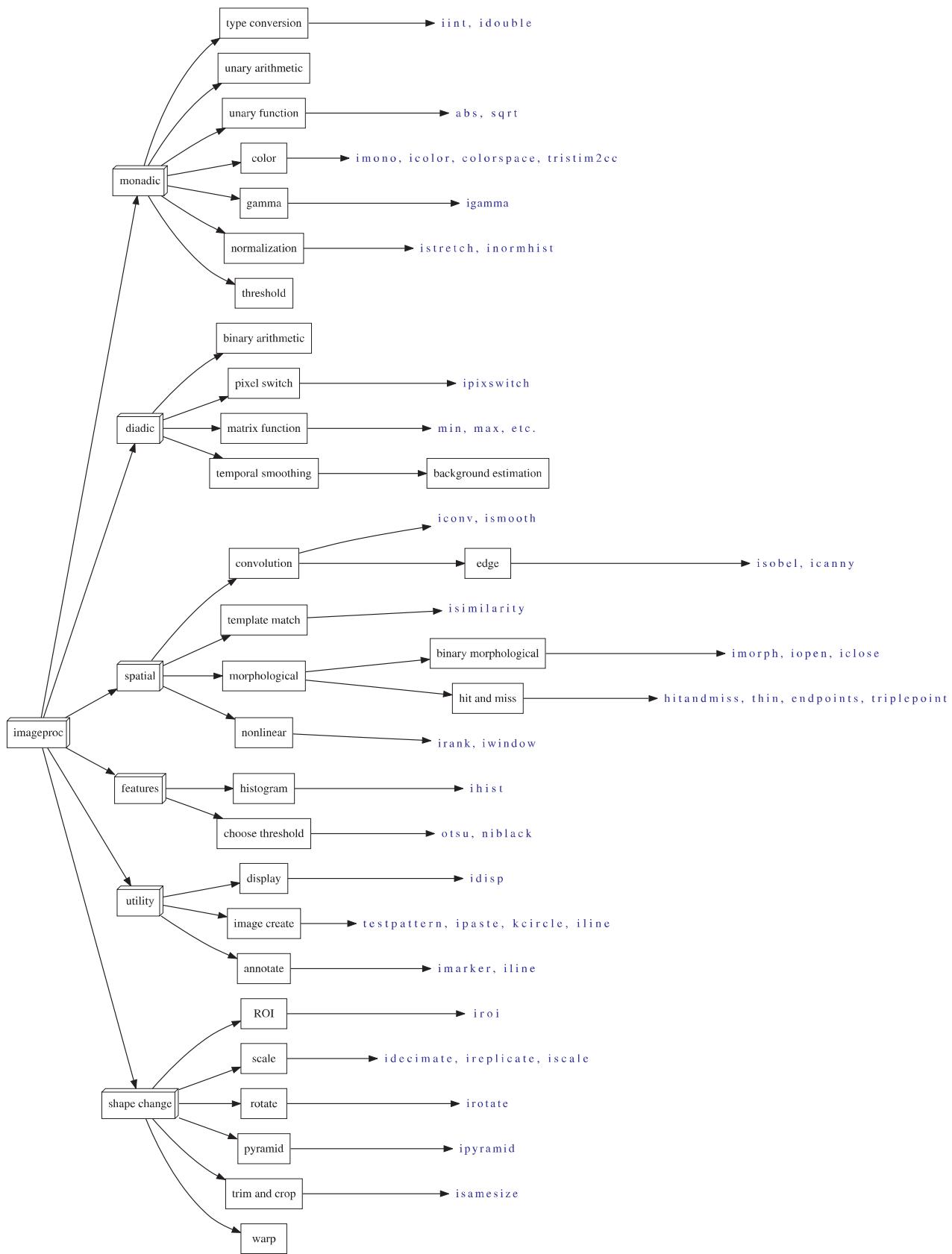
Image processing is a large field and this chapter has provided an introduction to many of the most useful techniques from a robotics perspective. The textbook by Gonzalez and Woods (2008) has a comprehensive discussion of image processing techniques as does Szeliski (2011). It expands on methods introduced in this chapter and covers additional topics such as greyscale morphology, image restoration, wavelet and frequency domain methods, and image compression. Online information about computer vision is available through CVonline at <http://homepages.inf.ed.ac.uk/rbf/CVonline>, and the material in this chapter is covered under the section *Image Transformations and Filters*.

Edge detection is a subset of image processing but one with huge literature of its own. Forsyth and Ponce (2002) have a comprehensive introduction to edge detection and a useful discussion on the limitations of edge detection. Nixon and Aguado (2008) also cover phase congruency approaches to edge detection and compare various edge detectors. The Sobel kernel for edge detection was described in an unpublished 1968 publication from the Stanford AI lab by Irwin Sobel and Jerome Feldman: *A 3 × 3 Isotropic Gradient Operator for Image Processing*. The Canny edge detector was originally described in Canny (1983, 1987).

Non-parametric measures for image similarity became popular in the 1990s with with a number of key papers such as Zabih and Woodfill (1994), Banks and Corke (2001), Bhat and Nayar (2002). The application to real-time image processing systems using high-speed logic such as FPGAs has been explored by several groups (Corke et al. 1999; Woodfill and Von Herzen 1997).

Mathematical morphology is another very large topic and we have only scraped the surface. Important techniques such as greyscale morphology and watersheds have not been covered. The pioneer of the field is Jean Serra and good starting points for further investigation are his book from 1983 (Serra 1983) or online tutorials at the Centre for Mathematical Morphology at [http://cmm.ensmp.fr/index\\_eng.html](http://cmm.ensmp.fr/index_eng.html). Another thorough treatment is the book by Soille (2003) while the book by Dougherty and Latufo (2003) is a more hands on tutorial approach. Gonzalez and Woods also has a useful discussion of greyscale morphology.

**Fig. 12.37.**  
Taxonomy of image processing algorithms discussed in this chapter



The approach to compute vision covered in this book is often referred to as bottom-up processing. This chapter has been about *low-level* vision techniques which are operations on pixels. The next chapter is about *high-level* vision techniques where sets of pixels are grouped and then described to represent objects in the scene.

---

### Sources of Image Data

There are thousands of online webcams as well as a number of sites that aggregate web cameras and provide lists categorized by location. Some of the content on these list pages can be rather dubious – so beware. Most of these sites do not connect you directly to the web camera so the URL of the camera is generally not available. Sites such as Opentopia <http://www.opentopia.com> do list a large number of web cameras and the actual URL for the camera is listed below the image on the line marked *Website*:. The root part of the URL (before the first single slash) is required for the `AxisWebCamera` class.

The motion sequence videos used in Sect. 12.3 are available from the CAVIAR project at <http://homepages.inf.ed.ac.uk/rbf/CAVIARDATA1> and are described in Fisher (2004).

A number of images are provided with the Toolbox in the `images` folder.

---

### MATLAB® Software Tools

The Image Processing Toolbox (IPT) can be purchased from The Mathwork Inc., the publishers of MATLAB. The companion to Gonzalez and Woods (2008) is their MATLAB® based book from 2009 (Gonzalez et al. 2009) which provides a detailed coverage of image processing using MATLAB® and includes functions that extend the IPT. These are provided for free as P-code format (no source or help available) or as M-files for purchase.

The Machine Vision Toolbox (MVTB) described in this book is open source. The MVTB functionality overlaps the IPT, but the MVTB contains functions not in the IPT and the IPT contains functions not in the MVTB. The image processing category at MATLAB® CENTRAL <http://www.mathworks.com/matlabcentral/fileexchange> lists more than 500 files.

---

### General Software Tools

There is a wealth of high quality software tools for image and video manipulation outside the MATLAB® environment. OpenCV at <http://opencv.org> is a mature open-source computer vision software project. The book by Bradski and Kaehler (2008) is a good introduction to the software and to computer vision in general.

ImageMagick <http://www.imagemagick.org> is a cross-platform collection of libraries and command-line tools for image format conversion (over 100 formats), manipulation and composition and the API has bindings for many languages. NetPBM <http://netpbm.sourceforge.net> can be built on most platforms and is also a collection of command-line tools for image format conversion (over 100 formats) and manipulation. The NetPBM utilities are an install option for many Linux distributions. Both packages are well suited for batch operations on large sets of images. The Toolbox function `pnmfilt` can be used to process an image through one of these external programs, for example

```
>> rlena = pnmfilt(lena, 'pnmrotate 30');
```

For video manipulation FFmpeg <http://www.ffmpeg.org> is an excellent and comprehensive cross-platform tool. It supports conversion between video formats as well as videos to still images and vice versa.

---

### Exercises

1. Become familiar with `idisp` for greyscale and color images. Explore pixel values in the image as well as the zoom, line and histogram buttons.
2. Grab some frames from the camera on your computer or from a movie file and display them.
3. Write a loop that grabs a frame from your camera and displays it. Add some effects to the image before display such as “negative image”, thresholding, posterization, false color, edge filtering etc.
4. Motion detection
  - a) Write a loop that performs background estimation using frames from your camera. What happens as you move objects in the scene, or let them sit there for a while? Explore the effect of changing the parameter  $\sigma$ .
  - b) Modify the *LeftBag* example on page 298 and highlight the moving people.
  - c) Combine motion detection and chroma-keying, put the moving people from the lobby into the desert.
5. Convolution
  - a) Compare the results of smoothing using a  $21 \times 21$  uniform kernel and a Gaussian kernel. Can you observe the ringing artefact in the former?
  - b) Why do we choose a smoothing kernel that sums to one?
  - c) Compare the performance of the simple horizontal gradient kernel  $K = (-0.5 \ 0 \ 0.5)$  with the Sobel kernel.
  - d) Investigate filtering with the Gaussian kernel for different values of  $\sigma$  and kernel size.
  - e) Create a  $31 \times 31$  kernel to detect lines at 60 deg.
  - f) Derive analytically the derivative of the Gaussian in the  $x$ -direction Eq. 12.2.
  - g) Derive analytically the derivative of the Laplacian of Gaussian Eq. 12.6.
  - h) Derive analytically the difference of Gaussian from page 310.
6. Show analytically the effect of an intensity scale error on the SSD and NCC similarity measures.
7. Template matching
  - a) Use `iroi` to select one of Lena’s eyes as a template. The template should have odd dimensions.
  - b) Use `isimilarity` to compute the similarity image. What is the best match and where does it occur? What is the similarity to the other eye? Where does the second best match occur and what is its similarity score?
  - c) Scale the intensity of the Lena image and investigate the effect on the peak similarity.
  - d) Add an offset to the intensity of the Lena image and investigate the effect on the peak similarity.
  - e) Repeat steps (c) and (d) for different similarity measures such as SAD, SSD, rank and census.
  - f) Scale the template size by different factors (use `iscale`) in the range 0.5 to 2.0 in steps of 0.05 and investigate the effect on the peak similarity. Plot peak similarity vs scale.
  - g) Repeat (f) for rotation of the template in the range  $-0.2$  to  $0.2$  rad in steps of 0.05.
8. Perform the sub-sampling example on page 325 and examine aliasing artefacts around sharp edges and the regular texture of the roof tiles. What is the appropriate smoothing kernel width for a decimation by  $M$ ?
9. Write a function to create Fig. 12.33 from the output of `ipyramid`.
10. Create a warp function that mimics your favourite funhouse mirror.
11. Warp the image to polar coordinates  $(r, \theta)$  with respect to the centre of the image, where the horizontal axis is  $r$  and the vertical axis is  $\theta$ .

# 13

# Image Feature Extraction

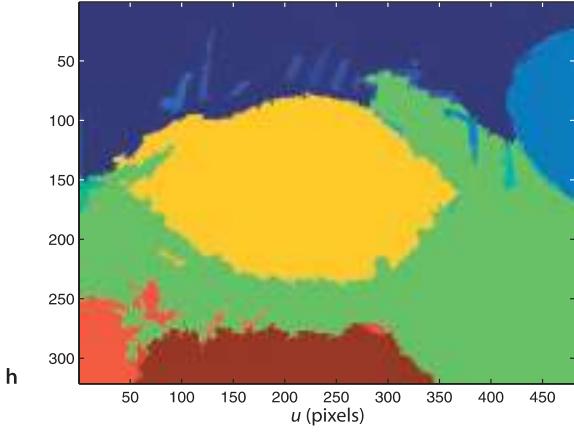
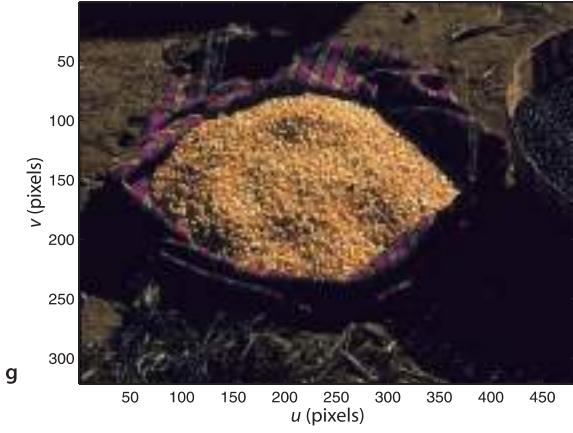
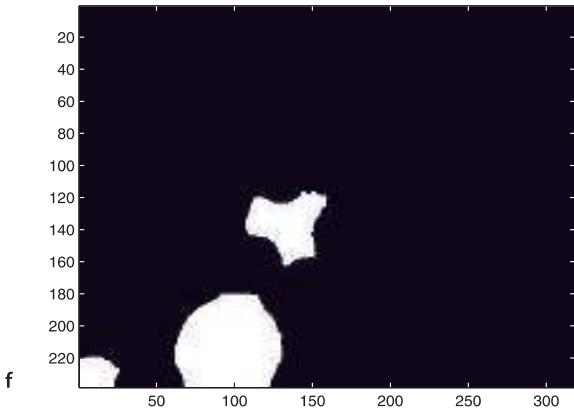
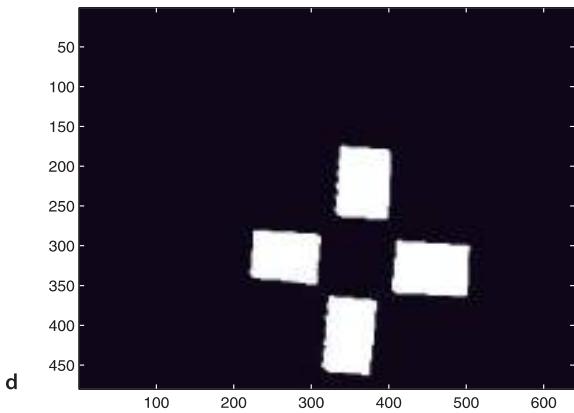
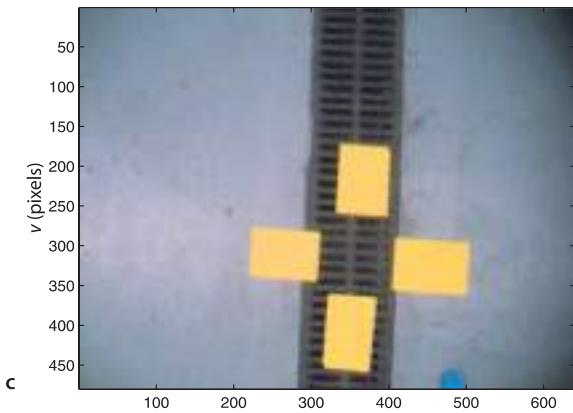
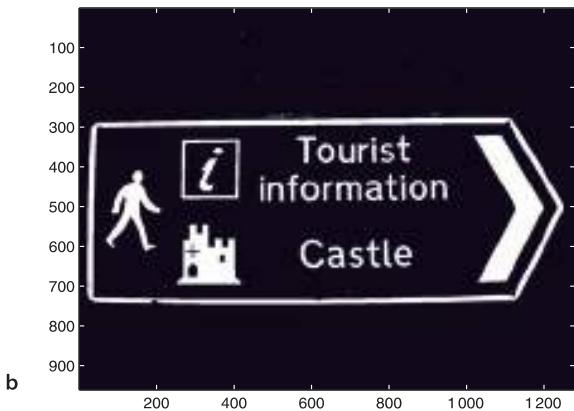


In the last chapter we discussed the acquisition and processing of images. We learnt that images are simply large arrays of pixel values but for robotic applications images have too much *data* and not enough *information*. We need to be able to answer pithy questions such as what is the pose of the object? what type of object is it? how fast is it moving? how fast am I moving? and so on. The answers to such questions are *measurements* obtained from the image and which we call image features. Features are the *gist* of the scene and the raw material that we need for robot control.

The image processing operations from the last chapter operated on one or more input images and returned another image. In contrast *feature extraction* operates on an image and returns one or more *image features*. Features are typically scalars (for example area or aspect ratio) or short vectors (for example the coordinate of an object or the parameters of a line). Image feature extraction is a necessary first step in using image data to control a robot. It is an *information concentration* step that reduces the data rate from  $10^6$ – $10^8$  bytes  $s^{-1}$  at the output of a camera to something of the order of tens of features per frame that can be used as input to a robot's control system.

In this chapter we discuss features and how to extract them from images. Drawing on image processing techniques from the last chapter we will discuss several classes of feature: regions, lines and interest points. Section 13.1 discusses region features which are contiguous groups of pixels that are homogeneous with respect to some pixel property. For example the set of pixels that represent a red object against a non-red background. Section 13.2 discusses line features which describe straight lines in the world. Straight lines are distinct and very common in man-made environments – for example the edges of doorways, buildings or roads. The final class of features are interest points which are discussed in Sect. 13.3. These are particularly distinctive *points* in a scene which can be reliably detected in different views of the same scene.

It is important to always keep in mind that image features are a summary of the information present in the pixels that comprise the image, and that the mapping from the world to pixels involves significant information loss. We typically counter this information loss by making assumptions based on our knowledge of the environment, but our system will only ever be as good as the validity of our assumptions. For example, we might use image features to describe the position and shape of a group of red pixels that correspond to a red object. However the size feature, typically the number of pixels, does not say anything about the size of the red object in the world – we need extra information such as the distance between the camera and the object and the camera's intrinsic parameters. We also need to assume that the object is not partially occluded – that would make the observed size less than the true size. Further we need to assume that the illumination is such that the chromaticity of the light reflected from the object is considered to be red. We might also find features in an image that do not correspond to a physical object – decorative markings, the strong edges of a shadow, or reflections in a window.



## 13.1 Region Features

Image segmentation is the process of partitioning an image into *application meaningful* regions as illustrated in Fig. 13.1. The aim is to segment or separate those pixels that represent objects of interest from all other pixels in the scene. This is one of the earliest approaches to scene understanding and while conceptually straightforward it is a very challenging problem. A key requirement is *robustness* which is how gracefully the method degrades as the underlying assumptions are violated, for example changing scene illumination or viewpoint.

Image segmentation is considered as three subproblems. The first is *classification* which is a decision process applied to each pixel and assigning it to one of  $C$  classes  $c \in \{0 \dots C - 1\}$ . Commonly we use  $C = 2$  which is known as binary classification or binarization and some examples are shown in Fig. 13.1a–f. The pixels have been classified as object ( $c = 1$ ) or not-object ( $c = 0$ ) which are displayed as white or black pixels respectively. The classification is *always application specific* – for example the object corresponds to pixels that are bright or yellow or red or moving. Figure 13.1h is a multi-level classification where  $C = 28$  and the pixel's class is reflected in its displayed color.

The underlying *assumption* in the examples of Fig. 13.1 is that regions are *homogeneous* with respect to some pixel characteristic. In practice we accept that this stage is imperfect and that pixels may be misclassified – subsequent processing steps will have to deal with this.

The second step in the segmentation process is *representation* where adjacent pixels of the same class are *connected* to form spatial sets  $S_1 \dots S_m$ . The sets can be represented by assigning a set label to each pixel or by a list of pixel coordinates that defines the boundary of the connected set. In the third and final step, the sets  $S_i$  are *described* in terms of scalar or vector-valued *features* such as size, position, and shape.

### 13.1.1 Classification

The pixel class is represented by an integer  $c \in \{0 \dots C - 1\}$  where  $C$  is the number of classes. In this section we discuss the problem of assigning each pixel to a class. In many of the examples we will use binary classification with just two classes corresponding to not-object and object, or background and foreground.

#### 13.1.1.1 Grey-Level Classification

A common rule for binary classification of pixels is

$$c[u, v] = \begin{cases} 0 & I[u, v] < t \\ 1 & I[u, v] \geq t \end{cases} \quad \forall (u, v) \in I$$

where the decision is based simply on the value of the pixel. This approach is called thresholding and  $t$  is referred to as the *threshold*.

Thresholding is very simple to implement. Consider the image

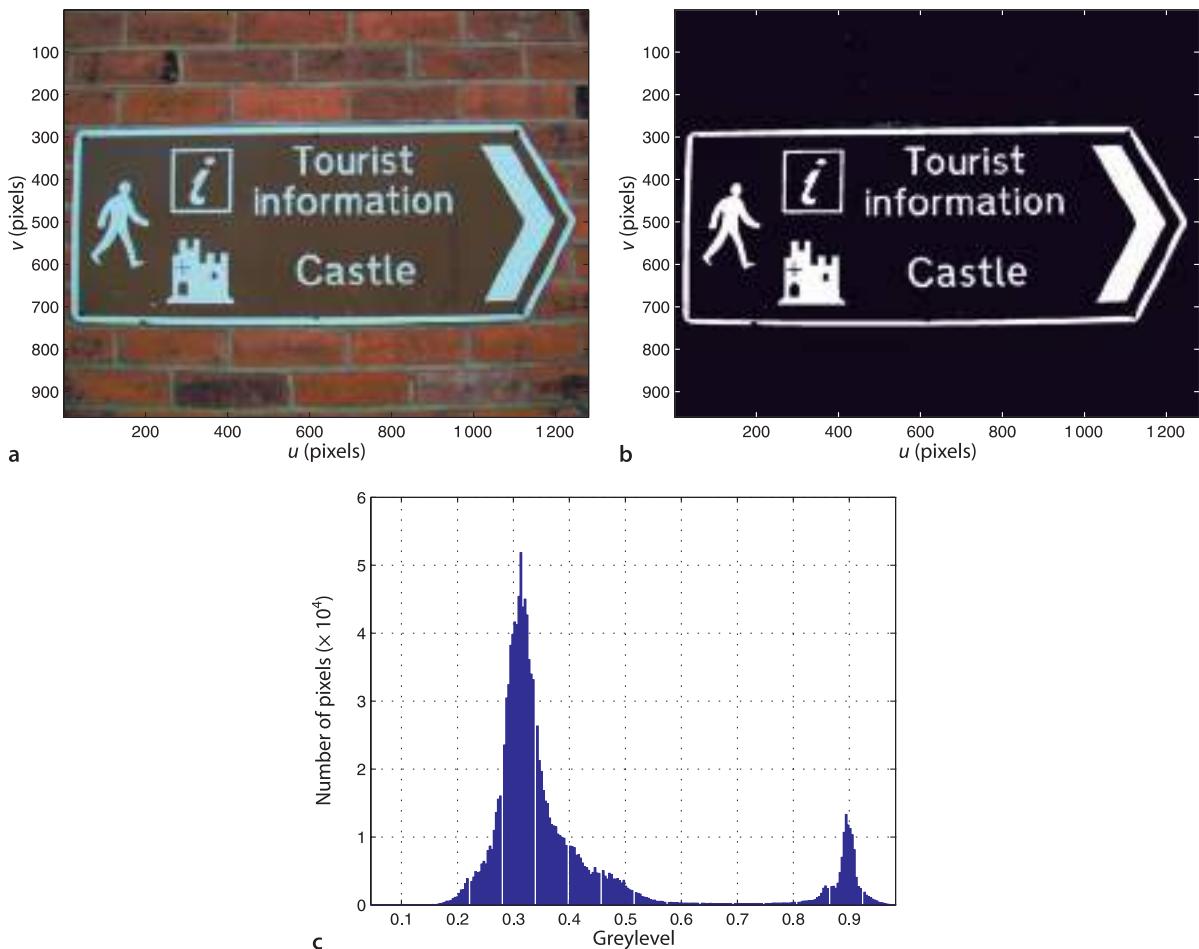
```
>> castle = imread('castle_sign.jpg', 'grey', 'double');
```

which is shown in Fig. 13.2a. The thresholded image

```
>> idisp(castle >= 0.6)
```

is shown in Fig. 13.2b. The pixels have been quite accurately classified as corresponding to white paint or not. This classification is based on the seemingly reasonable *assumption* that white paint is brighter than everything else in the image. In the early

**Fig. 13.1.** Examples of pixel classification. The left-hand column is the input image and the right-hand column is the classification. The classification is application specific and the pixels have been classified as either object (white) or not-object (black). The objects of interest are **a** the individual letters on the sign; **b** the yellow targets; **c** the red tomatoes. **d** is a multi-level segmentation where pixels have been assigned to 28 classes that represent locally homogeneous groups of pixels in the scene



**Fig. 13.2.** Binary classification.  
**a** Original image (image sourced from the ICDAR collection; Lucas 2005); **b** binary classification; **c** histogram of greyscale pixel values

days of computer vision, when computer power was limited, this approach was widely used – it was easier to contrive a world of white objects and dark backgrounds than to implement more sophisticated classification. In modern industrial inspection systems based on modest embedded computers this simple approach is still common – it works very well if the objects are on a conveyor belt of a suitable contrasting color or in silhouette at an inspection station. In the real world we generally have to work a little harder in order to achieve useful grey-level classification. An important question, and a hard one, is where did the threshold value of 0.6 come from? The most common approach is trial and error! The Toolbox function `ithresh`

```
>> ithresh(castle)
```

displays the image and a threshold slider that can be adjusted until a satisfactory result is obtained. However if the intensity of the image changed

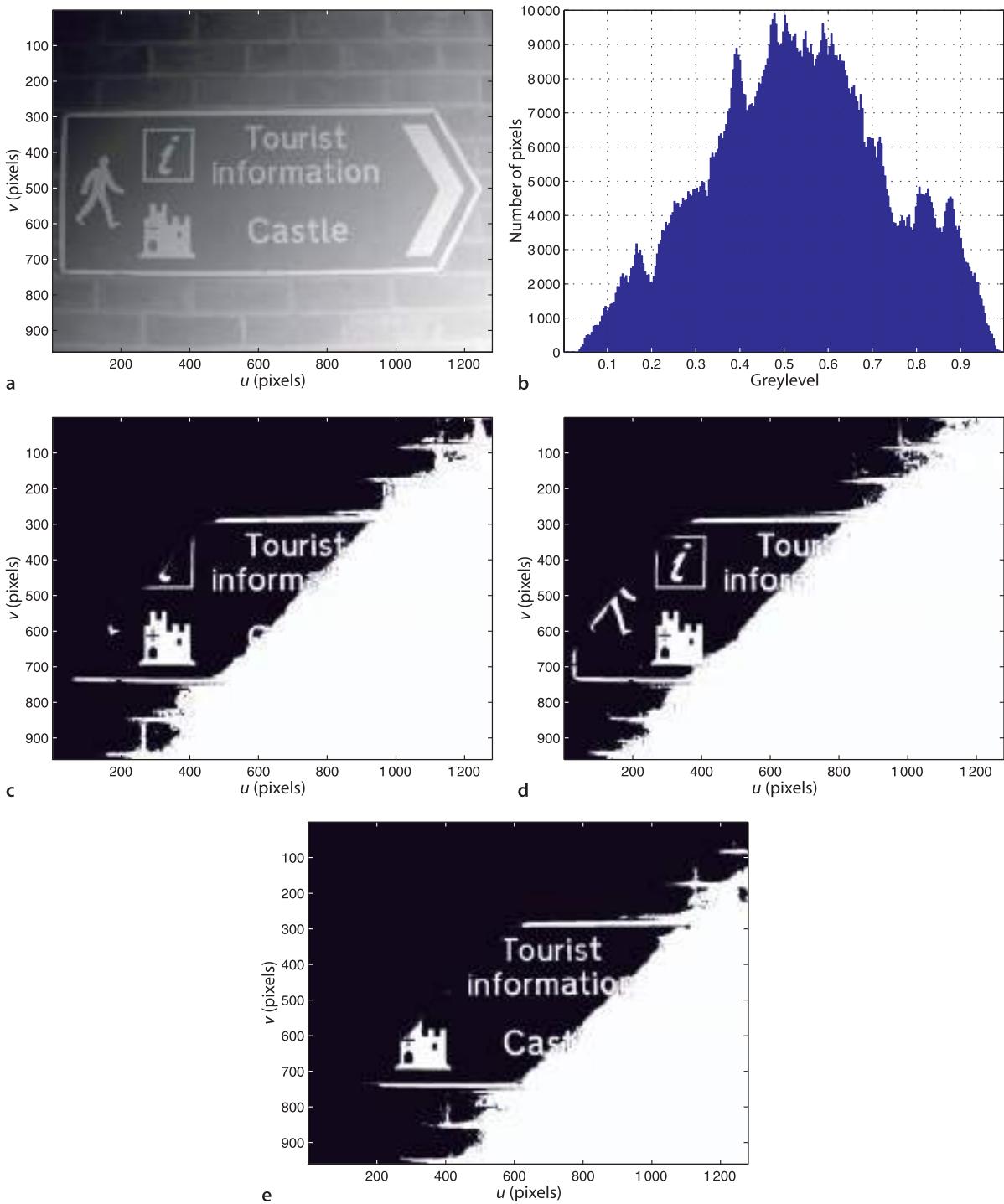
```
>> ithresh(castle*0.9)
```

a different threshold would be required.

A more principled approach than trial and error is to analyze the histogram of the image

```
>> ihist(castle, 'b');
```

which is shown in Fig. 13.2c. The histogram has two clearly defined peaks, a bimodal distribution, which correspond to two *populations* of pixels. The smaller peak around 0.9 corresponds to the pixels that are bright and it has quite a small range of



**Fig. 13.3.** Binary segmentation example. **a** Grey scale image with diagonal intensity gradient, **b** histogram, **c** thresholded with Otsu's threshold at 0.54, **d** thresholded at 0.50, **e** thresholded at 0.60

variation in value. The wider and taller peak around 0.3 corresponds to pixels in the darker background of the sign and the bricks, and has a much larger variation in brightness.

To separate the two classes of pixels we choose the decision boundary, the threshold-old, to lie in the valley between the peaks. In this regard the choice of  $t = 0.6$  is a good one. Since the valley is very wide we actually have quite a range of choice for the threshold, for example  $t = 0.7$  would also work well. The optimal threshold can be computed using Otsu's method

```
>> t = otsu(castle)
t =
0.6000
```

which separates an image into two classes of pixels in a way that minimizes the variance of values within each class and maximizes the variance of values between the classes – assuming that the histogram has just two peaks. Sadly, as we shall see, the real world is rarely this facilitating.

Consider the same image to which a not-unrealistic diagonal intensity gradient has been added

```
>> castle = imread('castle_sign2.png', 'double', 'grey');
```

which is shown in Fig. 13.3a. The histogram shown in Fig. 13.2b is radically different and much more complex. Unfortunately for us there are no longer two clearly distinct populations to separate. In this case Otsu's method computes a threshold of

```
>> t = otsu(castle)
t =
0.5412
```

and the results of applying this threshold is shown in Fig. 13.2c. The pixel classification is poor but the underlying assumption of Otsu's method, a bimodal distribution, is violated. Slightly lower and higher thresholds are shown in Fig. 13.2d and e respectively.

Thresholding-based techniques are notoriously brittle – a slight change in illumination of the scene means that the thresholds we chose would no longer be appropriate. In most real scenes there is no simple mapping from pixel values to particular objects – we cannot for example choose a threshold that would select a motorbike or a duck. Distinguishing an object from the background remains a hard computer vision problem.

One alternative is to choose a local rather than a global threshold. The Niblack algorithm is widely used in optical character recognition systems and computes a local threshold

$$t[u, v] = \mu(\mathcal{W}) + k\sigma(\mathcal{W})$$

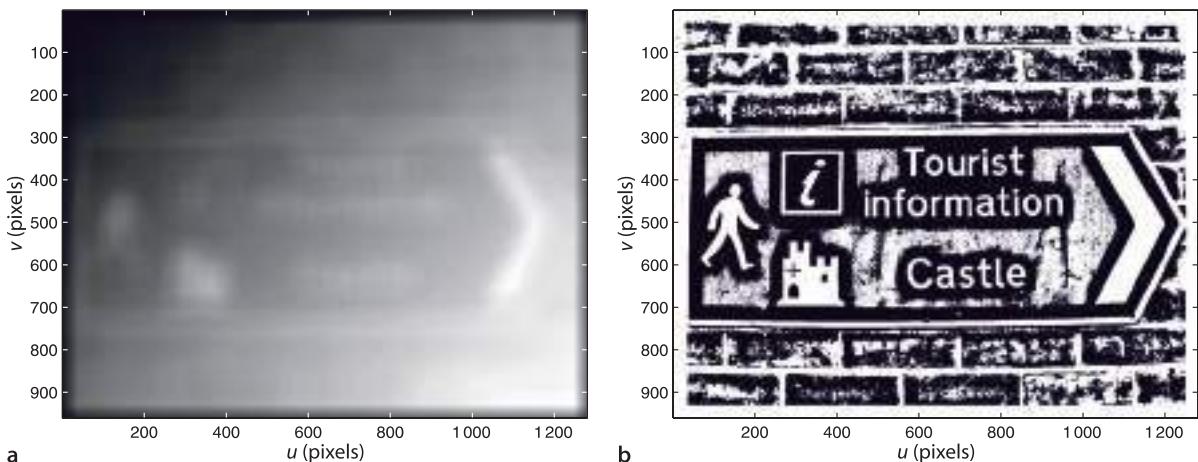
where  $\mathcal{W}$  is a region about the point  $(u, v)$  and  $\mu(\cdot)$  and  $\sigma(\cdot)$  are the mean and standard deviation respectively. The size of the window  $\mathcal{W}$  is a critical parameter and should be of a similar size to the objects we are looking for. For this example we make an assumption about the scene, that the characters are approximately 50–70 pixel tall, to choose a window half-width of 35 pixel

```
>> t = niblack(castle, -0.2, 35);
>> idisp(t)
```

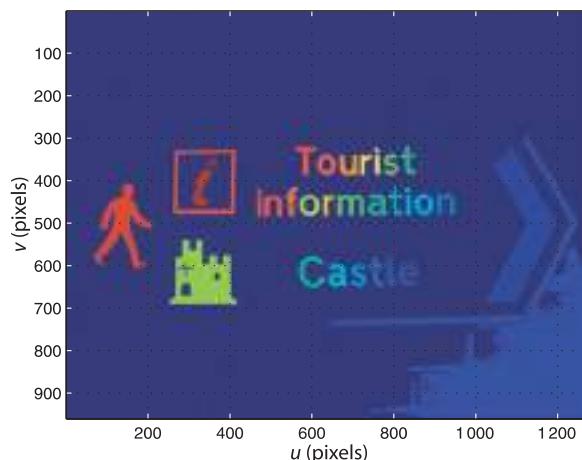
where  $k = -0.2$ . The resulting local threshold  $t$  is shown in Fig. 13.4a. We apply the threshold pixel-wise to the original image

```
>> idisp(castle >= t)
```

resulting in the classification shown in Fig. 13.4b. All the pixels belonging to the letters have been correctly classified but compared to Fig. 13.2c there are many false positives – non-object pixels classified as object. Later in this section we will discuss techniques to eliminate these false positives. Note that the classification process is no longer a function of just the input pixel, it is now a complex function of the pixel and its neighbours. While we no longer need to choose  $t$  we now need to choose the parameters  $k$  and window size, and again this is usually a trial and error process that can be made to work well for a particular type of scene.



**Fig. 13.4.** Niblack thresholding.  
a The local threshold displayed as an image; b the binary segmentation



**Fig. 13.5.**  
Segmentation using maximally stable extremal regions (MSER). The identified regions are uniquely color coded

The results shown in Fig. 13.2c to e are disappointing at first glance but we can see that every object is correctly classified at some, but not all, thresholds. In fact each object is correctly segmented for some *range* of thresholds and what we would like is the union of regions classified over the range of all thresholds. The maximally stable extremal region or MSER algorithm does exactly this. It is implemented by the Toolbox function `imser`

```
>> [mser,nsets] = imser(castle, 'light');
```

and for this image

```
>> nsets
nsets =
40
```

stable sets were found. The other return value is an image

```
>> idisp(mser, 'colormap', 'jet')
```

Although no explicit threshold has been given `imser` has a number of parameters and in this case their default values have given satisfactory results. See the online documentation for details of the parameters.

which is shown in Fig. 13.5 as a false color image. Each non-zero pixel corresponds to a stable set and the *label* assigned to that stable set which is displayed as a unique color. All the letters were correctly classified. The boundary has been partly misclassified as background, and part of it has been joined to the brick texture on the right hand side of the image. The option '`light`' indicates we are looking for light objects against a dark background. ▲

### 13.1.1.2 Color Classification

Color is a powerful cue for segmentation but roboticists tend to shy away from using it because of the problems with color constancy discussed in Sect. 10.3.1. In this section we consider two examples that use color images. The first is a navigation target for an indoor UAV

```
>> im_targets = imread('yellowtargets.png', 'gamma', 'sRGB', 'double');
```

shown in Fig. 13.6a and the second from the MIT Robotic Garden project

```
>> im_garden = imread('tomato_124.jpg', 'gamma', 'sRGB', 'double');
```

is shown in Fig. 13.7a. In both cases we were careful to apply gamma decoding to the color images since we will be performing colorimetric operations. Our objective is to determine the centroids of the yellow targets and the red tomatoes respectively. The initial stages of processing are the same for each image but we will illustrate the process in detail for the image of the yellow targets as shown in Fig. 13.6.

The Toolbox function `colorkmeans` first converts each color pixel to its *xy*-chromaticity coordinate – each color pixel is mapped to a point on the *xy*-chromaticity plane. Then the k-means algorithm is used to find clusters of points on the plane and each cluster corresponds to a group of pixels with a distinguishable color.<sup>►</sup> A limitation of k-means is that we must specify the number of clusters to find. We will use our knowledge that this particular scene has four differently colored elements: yellow targets, floor, metal drain cover and dark shadow. The pixels are clustered into *four* chromaticity classes ( $C = 4$ ) by

```
>> [cls, cxy,resid] = colorkmeans(im_targets, 4);
```

which returns the *xy*-chromaticity of each cluster

**k-means clustering** is an iterative algorithm for grouping  $n$ -dimensional points into  $k$  spatial clusters. Each cluster is defined by a centre point which is an  $n$ -vector  $c_k$ . At each iteration all points are assigned to the *closest* cluster centre, and then each cluster centre is updated to be the mean of all the points assigned to the cluster.

The algorithm is implemented by the Toolbox function `kmeans`. The distance metric used is Euclidean distance.<sup>►</sup> The k-means algorithm requires an initial estimate of the centre of each cluster and this can be provided in three ways. The initial cluster centres can be provided by the user, the option '`random`' will randomly select  $k$  of the provided points, and '`spread`' will choose  $k$  random points from within the hypercube spanned by the set of points. This random initialisation means the algorithm will return different results at each invocation.<sup>►</sup>

For example, choose 500 random 2-dimensional points

```
>> a = rand(2,500);
```

where `a` is a  $2 \times 500$  matrix with one point per column. We will cluster this data into three sets

```
>> [cls,centre,r] = kmeans(a, 3);
```

where `cls` is a 500-vector whose elements specify the class of the corresponding column of `a`. `centre` is a  $2 \times 3$  whose columns specify the centre of each 2-dimensional cluster.

```
>> r
r =
    4.3166
```

is the residual which is the norm of the distance of every point from its assigned cluster centroid.

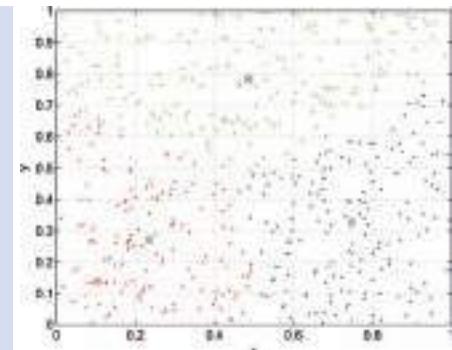
We plot the points in each cluster with the colors red, green and blue

```
>> plot_point(a(:,cls==1), 'r.')
>> plot_point(a(:,cls==2), 'g.');?>
plot_point(a(:,cls==3), 'b.')
```

and superimpose the centroids as well

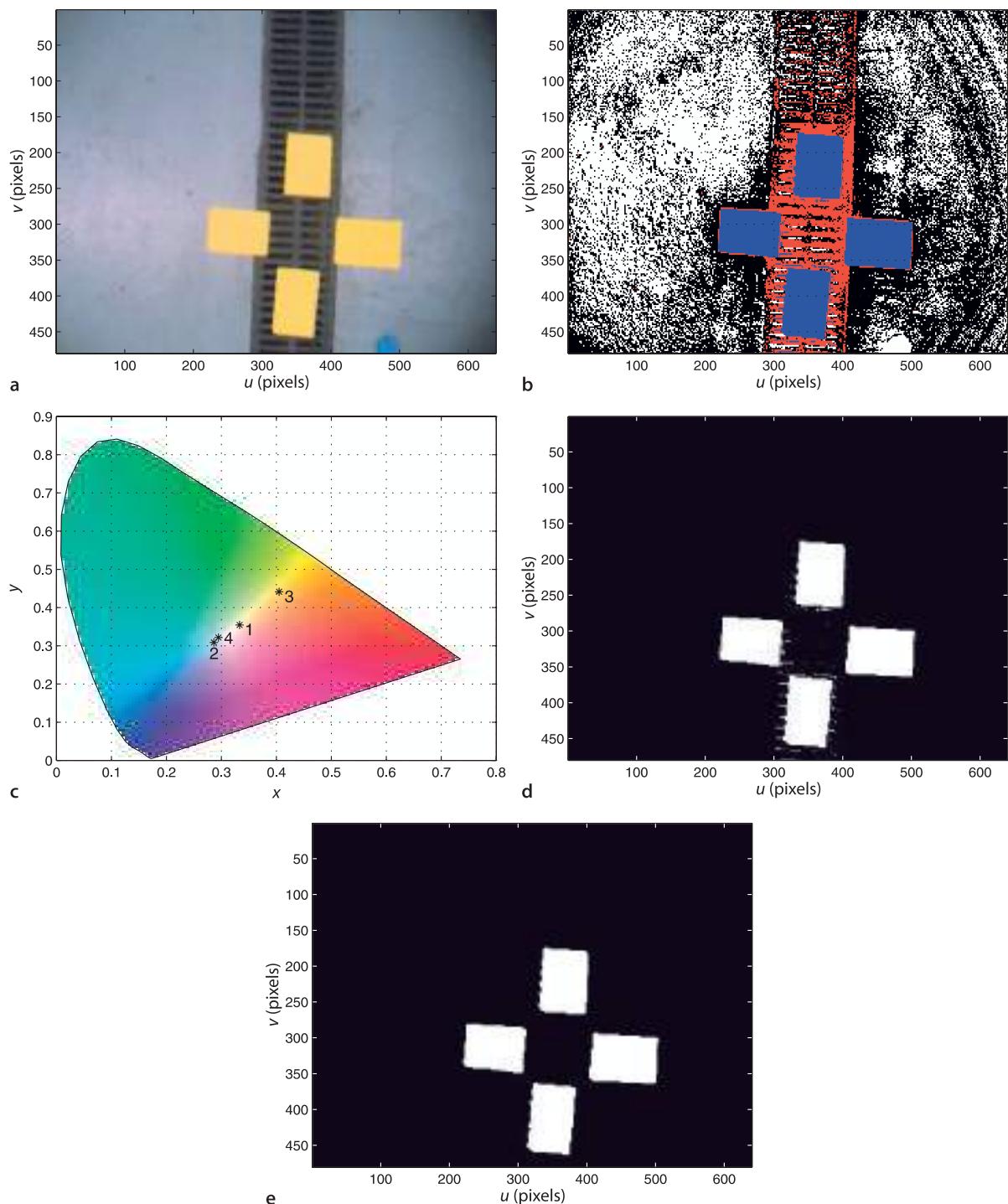
```
>> plot_point(centre, 'o'); plot_point(centre, 'x')
```

and we see that the points have been sensibly partitioned.



For a color space Euclidean distance may not be appropriate since human perception of color difference depends on distance in the *xy*-chromaticity plane and also the location within the plane.

The function `randinit` sets the MATLAB® random number generator to a known state and can be used to ensure repeatable results.



**Fig. 13.6.** Target image example.  
a Original image; b pixel classification ( $C = 4$ ) shown in false color;  
c cluster centres in the  $xy$ -chromaticity space; d all pixels of class  
 $c = 3$ ; e after morphological closing with a circular structuring el-  
ement (radius 2)

```
>> cxy
cxy =
0.3331    0.2862    0.4049    0.2947
0.3544    0.3088    0.4414    0.3215
```

which has one column per cluster. We can plot these cluster centres on the  $xy$ -plane

```
>> xycolorspace(cxy);
```

which is shown in Fig. 13.6c. We see that cluster 3 is the closest to yellow

```
>> colordname(cxy(:,3), 'xy')
ans =
'banana'
```

The residual

```
>> resid
resid =
6.2583
```

is the norm of the distance of every point from its assigned cluster centroid. Since the algorithm uses a random initialization we obtain different clusters and classification on every run, and therefore different residuals.

The function `colorkmeans` also returns the pixel classification which we can display as an image

```
>> idisp(cls, 'colormap', 'flag')
```

in false color as shown in Fig. 13.6b. The pixels in this image have values  $c = \{0, 1, 2, 3\}$  indicating which class the corresponding input pixels has been assigned to. If we use `idisp` to interactively probe the values we see that the yellow targets have indeed been assigned to class  $c = 3$  which is displayed as blue.

k-means clustering is computationally expensive and therefore not very well suited to real-time applications. However we can divide the process into a training phase and a classification phase. In the training phase a number of example images would be concatenated and passed to `colorkmeans` which would identify the centres of the clusters for each class. Subsequently we can assign pixels to their closest cluster relatively cheaply

```
>> cls = colorkmeans(im_targets, cxy);
```

The pixels belonging to class 3 can be selected

```
>> cls3 = (cls == 3);
```

which is a *logical image* that can be displayed

```
>> idisp(cls3)
```

as shown in Fig. 13.6d. All pixels of class 3 pixels are displayed as white<sup>►</sup> and correspond to the yellow targets in the original image. This binary image is a good classification but it is not perfect. It has some white pixels that do not belong to the target – these correspond to pixels that have a chromaticity close to yellow but which have low intensity and therefore do not look yellow.<sup>►</sup>

A morphological opening operation as discussed in Sect. 12.5 will eliminate these. We apply a symmetric structuring element of radius 2

```
>> targets_binary = iopen(cls3, kcircle(2));
```

and the result is shown in Fig. 13.6e. It shows a clean binary segmentation of the pixels into the two classes: target and not-target.

For the garden image we follow a very similar procedure. We classify the pixels into four clusters ( $C = 4$ ) based on our knowledge that the scene contains: tomatoes, light leaves, dark leaves and dark background

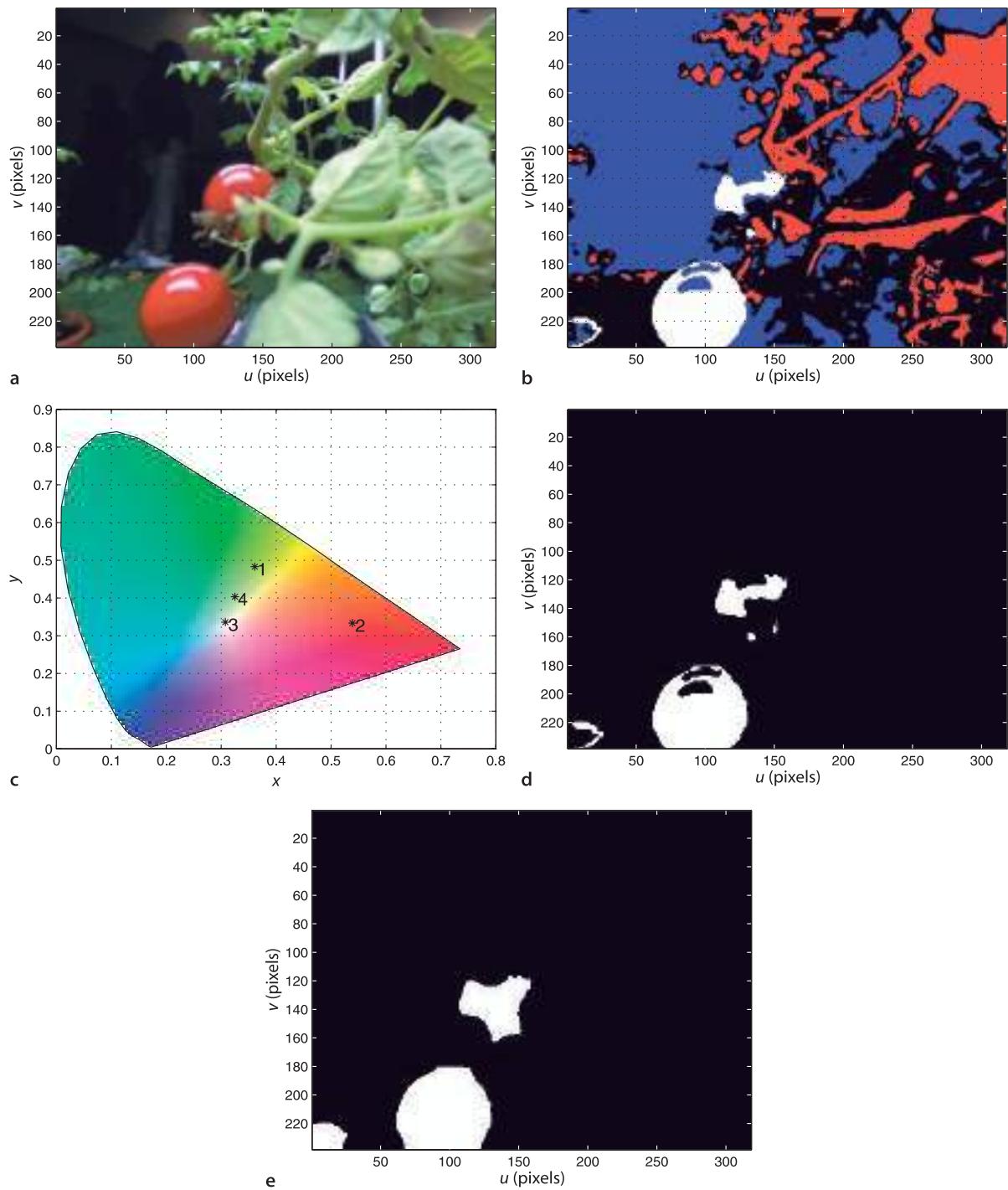
```
>> [cls, cxy] = colorkmeans(im_garden, 4);
>> cxy
cxy =
0.3610    0.5388    0.3076    0.3249
0.4829    0.3335    0.3359    0.4034
```

The pixel classes are shown in false color in Fig. 13.7b. Pixels corresponding to the tomato have been assigned to class  $c = 2$  which are displayed as white. The cluster centres are marked on the  $xy$ -chromaticity plane in Fig. 13.7c. The red pixels can be selected

One option is to run k-means a number of times, and take the cluster centres for which the residual is lowest.

MATLAB® represents the logical values *true* and *false* by the integer values *one* and *zero* respectively. These are displayed as *white* and *black* pixels respectively.

We encountered a similar issue with the flowers image in Sect. 10.3.5.



**Fig. 13.7.** Garden image example. a Original image (courtesy of Distributed Robot Garden project, MIT); b pixel classification ( $C = 4$ ) shown in false color; c cluster centres in the  $xy$ -chromaticity space; d all pixels of class  $c = 2$ ; e after morphological closing with a circular structuring element of radius 15

```
>> cls2 = (cls == 2);
```

and the resulting logical image is shown in Fig. 13.7d. This segmentation is far from perfect. Both tomatoes have *holes* due to specular reflection, in fact the top tomato is more hole than object. A few pixels at the bottom left have been erroneously classified as a tomato. We can improve the result by applying a morphological closing operation with a large circular kernel which is consistent with the shape of the tomato

**Specular reflection** is a mirror-like reflection from the surface of an object. The light does not penetrate the surface and scatter which is how an object ordinarily imparts color to reflected light. The color of the specular reflection is the color of the light source not the surface. A specular reflector such as a mirror obeys the rule where the angle of reflection of an outgoing light ray equals the angle of incidence of an incoming ray. A rough surface is a diffuse or Lambertian reflector and has the same apparent brightness at all viewing angles. Most real surfaces are a mixture of specular and Lambertian reflectors.

There are several ways to reduce the problem of specular reflection. Firstly, eliminate the light source that is specularly reflected. Secondly, use a diffuse light source near the camera, for instance a ring illuminator that fits around the lens of the camera. Thirdly, attenuate the specular reflection using a polarizing filter since light that is specularly reflected from a dielectric surface will be polarized.

```
>> tomatoes_binary = iclose(cls2, kcircle(15));
```

and the result is shown in Fig. 13.7e. The closing operation has somewhat restored the shape of the fruit but with the unwanted consequence that the group of misclassified pixels in the bottom-left corner have been enlarged. Nevertheless this image contains a workable classification of pixels into two classes: tomato and not-tomato.

The garden image illustrates two common real-world imaging artefacts: specular reflection and occlusion. The surface of the tomato is sufficiently shiny that the camera sees a reflection of the room light – these pixels are white rather than red.► The top tomato is also partly obscured by leaves and branches. Depending on how the application works this may or may not be a problem. Since the tomato cannot be reached from the direction the picture was taken, because of the occluding material, it might in fact be appropriate to not classify this as a tomato.

These examples have achieved a workable classification of the image pixels into object and not-object. The resulting groups of white pixels are commonly known as blobs. It is interesting to note that we have not specified any threshold or any definition of the object color, but we did have to specify the number of classes and determine which of those classes corresponded to the objects of interest.► We have also had to choose the sequence of image processing steps and the parameters for each of those steps, for example, the radius of the structuring element. Pixel classification is a difficult problem but we can get quite good results by exploiting knowledge of the problem, having a good collection of image processing tricks, and experience.

Observe that they have the same chromaticity as the black background, class 3 pixels, which are situated close to the white point on the xy-plane.

This is a relatively easy problem. The chromaticity of yellow is well known (we could use the `colorname` function to find it) so we could compute the distance between each cluster centre and the standard color and choose the cluster that is closest.

### 13.1.2 Representation

In the previous section we took complex greyscale or color images and processed them to produce binary or blob images. Representation is the subproblem of *connecting* adjacent pixels of the same class to form spatial sets  $S_1 \dots S_m$ .

Consider the small  $10 \times 8$  binary image

```
>> im = ilabeltest;
```

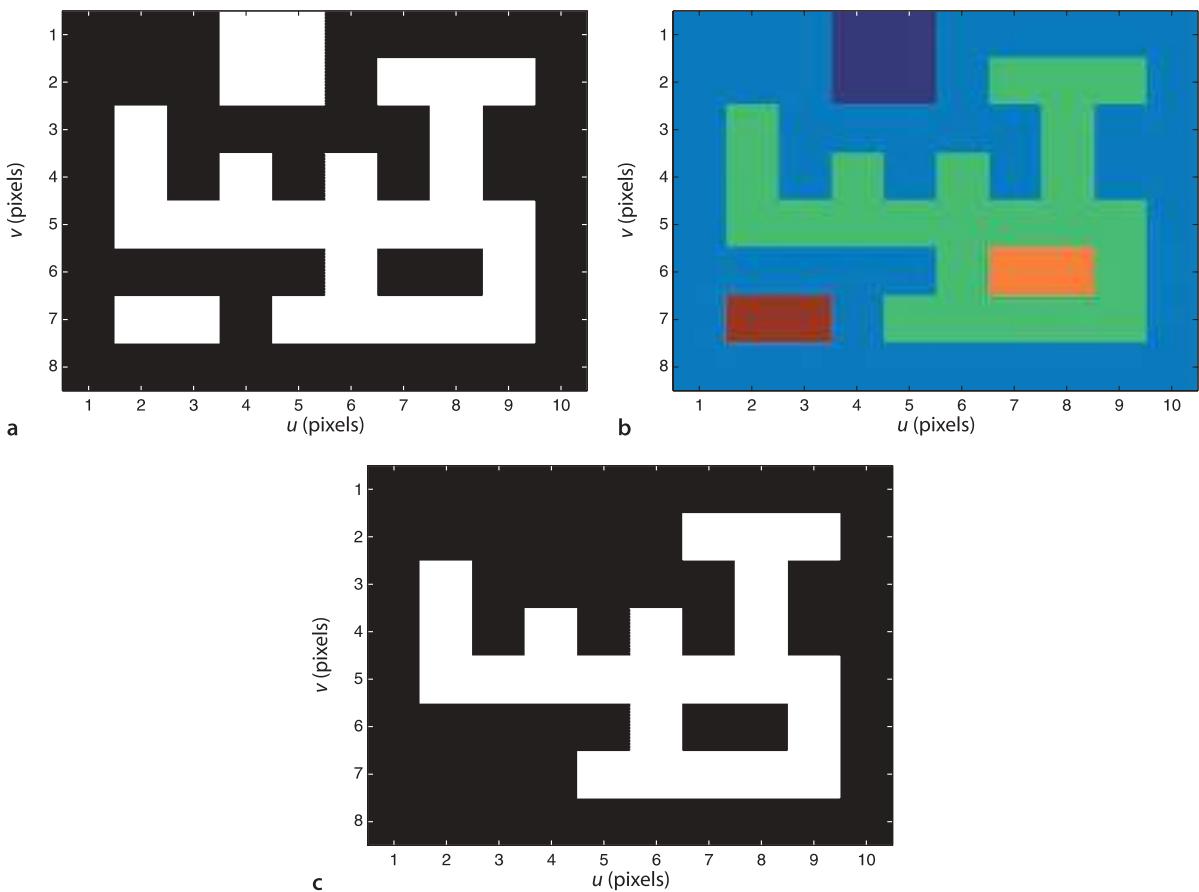
which is shown

```
>> idisp(im)
```

in Fig. 13.8a. We quickly identify three white blobs in this scene but what defines a blob? It is a set of pixels of the same class, white in this case, that are *connected* to each other. More formally we could say a blob is a spatially contiguous region of pixels of the same class. Blobs are also known as regions or connected components.

The Toolbox can perform connected component or connectivity analysis on this binary image

```
>> [label, m] = ilabel(im);
```



**Fig. 13.8.** Image labelling example. **a** Binary image; **b** labelled image; **c** all pixels with the label 3

The number of sets, or components, in this image is

```
>> m
m =
5
```

which are the three white blobs and the two black blobs (the background and the hole). These blobs are labeled from 1 to 5. The returned label matrix

```
>> label
label =
2 2 2 1 1 2 2 2 2 2
2 2 2 1 1 2 3 3 3 2
2 3 2 2 2 2 2 3 2 2
2 3 2 3 2 3 2 3 2 2
2 3 3 3 3 3 3 3 3 2
2 2 2 2 2 3 4 4 3 2
2 5 5 2 3 3 3 3 3 2
2 2 2 2 2 2 2 2 2 2
```

has the same size as the original image and each element contains the label  $s \in \{1 \dots m\}$  of the set to which the corresponding input pixel belongs. The label matrix can be displayed as an image in false color

```
>> idisp(label, 'colormap', 'jet')
```

as shown in Fig. 13.8b. Each connected region has a unique label and hence unique color. Looking at the elements of `label` shown above or by interactively probing the displayed label image using `idisp` we see that the background has been labelled as 2, the small square at the top is labelled 1, and the larger region is labelled 3 and the hole in that region is labelled 4.

We have seen a label image previously.  
The output of the MSER function in Fig. 13.5 is a label image.

To obtain an image containing just a particular blob is now very easy. To select all pixels belonging to region 3, the large irregular region, we create a logical image

```
>> reg3 = (label==3);
>> idisp(reg3)
```

which is shown in Fig. 13.8c.

The connectivity analysis can return additional output values

```
>> [label, m, parents, cls] = ilabel(im);
```

where the vector

```
>> parents'
ans =
    0     0     2     3     2
```

describes the topology or hierarchy of the regions. It indicates for example that region 4

```
>> parents(4)
ans =
    3
```

is enclosed by region 3 – region 3 is the parent of region 4. Regions 3 and 5 are enclosed by region 2 which is the background. Regions 1 and 2 have a parent of 0 which indicates that they touch the edge of the image and are not enclosed by any region. Each connected region contains pixels of a single class and the pixel class for each region is given by

```
>> cls'
ans =
    1     0     1     0     1
```

which indicates that regions 1, 3 and 5 comprise pixels of class 1 (white) and regions 2 and 4 comprise pixels of class 0 (black).►

In this example we have assumed 4-way connectivity, that is, pixels are connected into a region only through their north, south, east and west neighbours of the same class. The 8-way connectivity option allows connection via any of a pixel's eight neighbours of the same class.►

Returning now to the examples from the previous section. For the colored targets

```
>> targets_label = ilabel(targets_binary);
>> idisp(targets_label, 'colormap', 'jet');
```

and the garden image

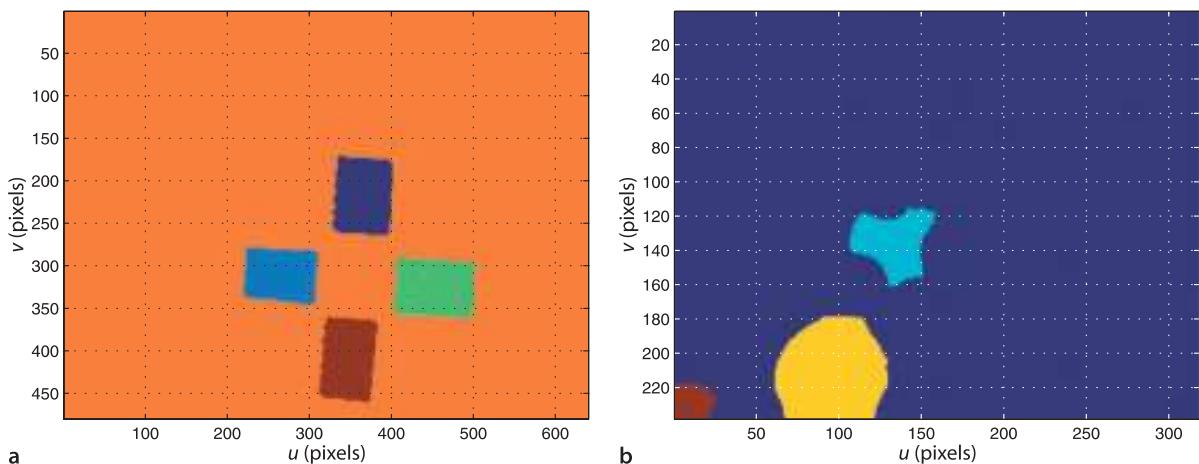
```
>> tomatoes_label = ilabel(tomatoes_binary);
>> idisp(tomatoes_label, 'colormap', 'jet');
```

the connected regions are shown in false color in Fig. 13.9.

We use the contraction `cls` since `class` is the name of a useful function in MATLAB®.

8-way connectivity can lead to surprising results. For example a black and white chequerboard would have just two regions, all white squares are one region and all the black squares another.

**Fig. 13.9.** Label images for the targets and garden examples in false color. The value of each pixel is the label of the spatially contiguous set to which the corresponding input pixel belongs



### 13.1.2.1 Graph-Based Segmentation

So far we have classified pixels based on some homogeneous characteristic of the object such as intensity or color. Consider now the complex scene

```
>> im = imread('58060.jpg');
```

The Berkeley segmentation site <http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds> hosts these images plus a number of different human-made segmentations.

shown in Fig. 13.10a. The Gestalt principle of emergence says that we identify objects as a whole rather than as a collection of parts – we see a bowl of grain rather than deducing a bowl of grain by recognising its individual components. However when it comes to a detailed pixel by pixel segmentation things become quite subjective – different people would perform the segmentation differently based on judgement calls about what is *important*. For example, should the colored stripes on the cloth be segmented? If segments represent real world objects, then the Gestalt view would be that the cloth should be just one segment. However the stripes are real, some effort was made to create them, so perhaps they should be segmented. This is why segmentation is a *hard* problem – humans cannot agree on what is correct. No computer algorithm could, or could be expected to, make this type of judgement.

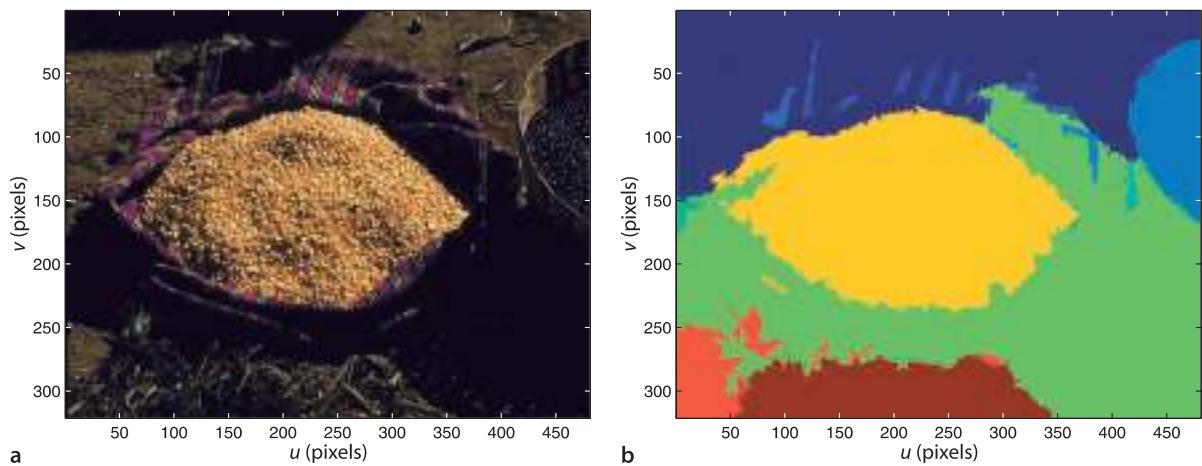
Nevertheless some recent algorithms can do a very impressive job on complex real world scenes. The image can be represented as a graph (see Appendix J) where each pixel is a vertex and has 8 edges connecting it to its neighbouring pixels. The weight of each edge is a non-negative measure of the dissimilarity between the two pixels – the absolute value of the difference in color. The algorithm starts with every vertex assigned to its own set. At each iteration the edge weights are examined and if the vertices are in different sets but the edge weight is below a threshold the two vertex sets are merged. The threshold is a function of the size of the set and a global parameter  $k$  which sets the scale of the segmentation – a larger value of  $k$  leads to a preference for larger connected components.

For the image discussed the graph-based segmentation is given by

```
>> [label, m] = igraphseg(im, 1500, 100, 0.5);
>> m
m =
28
>> idisp(label, 'colormap', 'jet')
```

**Fig. 13.10.** Complex segmentation example. **a** Original color image (image from the Berkeley Segmentation Dataset; Martin et al. 2001); **b** graph-based segmentation

where `label` is a matrix, shown in Fig. 13.10b, whose elements are the region label for the corresponding input pixels. The pixel classification step has been integrated into the representation step. The arguments are a scale parameter  $k = 1\,500$ , the minimum component size of 100 pixels, and the standard deviation for an initial Gaussian smoothing applied to the image.



### 13.1.3 Description

In the previous section we learnt how to find connected components in the image and how to isolate particular components such as shown in Fig. 13.8c. However this representation of the component is still just an image with logical pixel values rather than a concise *description* of its size, position and shape.

#### 13.1.3.1 Bounding Boxes

The simplest representation of size and shape is the bounding box – the smallest rectangle with sides parallel to the  $u$ - and  $v$ -axes that encloses the region. Returning to the targets image we can select all the pixels in region 5

```
>> reg5 = (targets_label == 5);
>> idisp(reg5);
```

and the resulting logical image is shown in Fig. 13.11b. The number of pixels in this region is simply the sum

```
>> sum(reg5(:))
ans =
6004
```

The coordinates of all the non-zero (object) pixels are

```
>> [v,u] = find(reg5);
```

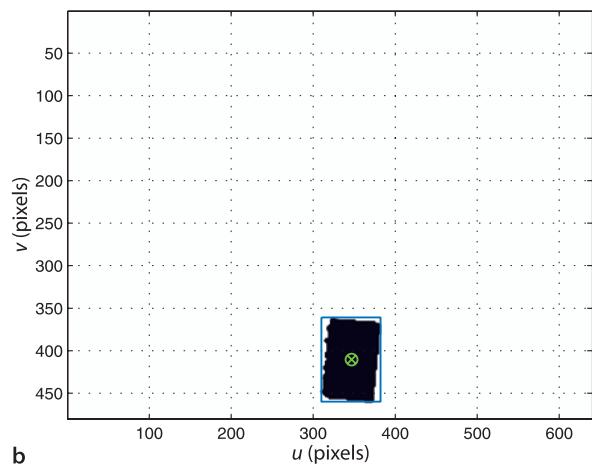
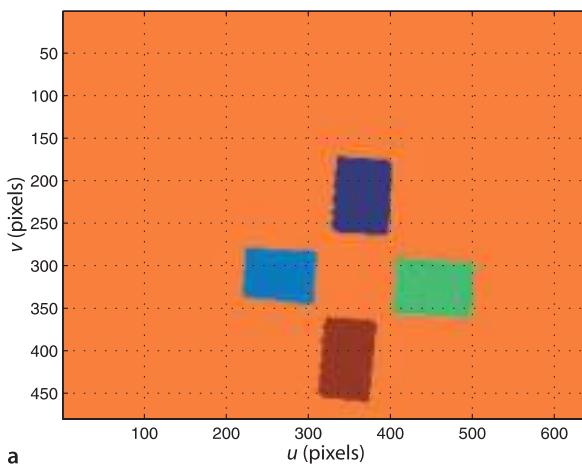
where  $u$  and  $v$  are each vectors of size

```
>> about(u)
u [double] : 6004x1 (48032 bytes)
```

The bounds of the region<sup>►</sup> are

```
>> umin = min(u)
umin =
310
>> umax = max(u)
umax =
382
>> vmin = min(v)
vmin =
361
>> vmax = max(v)
vmax =
460
```

This can be obtained more simply using the Toolbox function `ibbox`.



**Fig. 13.11.** Targets image. **a** Regions labels in false color; **b** region 1 (inverted, black is true) with bounding box and centroid marked

These bounds define a rectangle which we can superimpose on the image

```
>> plot_box(umin, vmin, umax, vmax, 'b')
```

as shown in Fig. 13.11b. The bounding box fits snugly around the blob and its centre could be considered as the centre of the blob. However the bounding box is not aligned with the blob, that is, its sides are not parallel with the sides of the blob. This means that as the blob rotates the size and shape of the bounding box would change even though the size and shape of the blob does not.

### 13.1.3.2 Moments

Moments are a rich and computationally cheap class of image features which can describe region size and location as well as shape. The moment of an image  $I$  is a scalar

$$m_{pq} = \sum_{(u,v) \in I} u^p v^q I[u,v] \quad (13.1)$$

where  $(p+q)$  is the *order* of the moment. The zeroth moment  $p=q=0$  is

$$m_{pq} = \sum_{(u,v) \in I} I[u,v] \quad (13.2)$$

and for a binary image where the background pixels are zero this is simply the number of non-zero (white) pixels – the area of the region.

Moments are calculated using the Toolbox function `mpq` and for region 5 of the target image the zeroth moment is

```
>> m00 = mpq(reg5, 0, 0)
m00 =
    6004
```

which is the area of the region in units of pixels.

Moments can be given a physical interpretation by regarding the image function as a mass distribution. Consider the region as being made out of thin metal plate where each pixel has one unit of area and one unit of mass. The total mass of the region is  $m_{00}$  and the centre of mass or centroid of the region is

$$u_c = \frac{m_{10}}{m_{00}}, \quad v_c = \frac{m_{01}}{m_{00}} \quad (13.3)$$

where  $m_{10}$  and  $m_{01}$  are the first-order moments. For our example the centroid of the target region is

```
>> uc = mpq(reg5, 1, 0) / m00
uc =
    346.7539
>> vc = mpq(reg5, 0, 1) / m00
vc =
    410.4446
```

which we can display

```
>> plot(uc, vc, 'gx'); plot(uc, vc, 'go');
```

as shown in Fig. 13.11b.

The central moments  $\mu_{pq}$  are computed with respect to the centroid

$$\mu_{pq} = \sum_{(u,v) \in I} (u - u_c)^p (v - v_c)^q I[u,v] \quad (13.4)$$

and are invariant to the position of the region. They are related to the moments  $m_{pq}$  by

$$\begin{aligned}\mu_{10} &= 0, & \mu_{01} &= 0 \\ \mu_{20} &= m_{20} - \frac{m_{10}^2}{m_{00}}, & \mu_{02} &= m_{02} - \frac{m_{01}^2}{m_{00}}, & \mu_{11} &= m_{11} - \frac{m_{10}m_{01}}{m_{00}}\end{aligned}\quad (13.5)$$

and are computed by the Toolbox function `upq`.

Using the thin plate analogy again, the inertia matrix of the region is

$$\mathbf{J} = \begin{pmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{pmatrix} \quad (13.6)$$

about axes parallel to the  $u$ - and  $v$ -axes and intersecting at the centroid of the region. The central second moments  $\mu_{20}, \mu_{02}$  are the moments of inertia and  $\mu_{11}$  is the product of inertia. The product of inertia is non-zero if the shape is asymmetric with respect to the region's axes.

The *equivalent ellipse* is the ellipse that has the same inertia matrix as the region. For our example

```
>> u20 = upq(reg5, 2, 0); u02 = upq(reg5, 0, 2); u11 = upq(reg5, 1, 1);
>> J = [ u20 u11; u11 u02]
J =
1.0e+06 *
2.0436   -0.2329
-0.2329   4.4181
```

The eigenvalues and eigenvectors of  $J$  are related to the radii of the ellipse and the orientation of its major and minor axes (see Appendix E). For this example the eigenvalues

```
>> lambda = eig(J)
lambda =
1.0e+06 *
2.0210
4.4408
```

are the principle moments of the region. The radii of the equivalent ellipse are

$$a = 2\sqrt{\frac{\lambda_1}{m_{00}}}, \quad b = 2\sqrt{\frac{\lambda_2}{m_{00}}} \quad (13.7)$$

or

```
>> ab = 2 * sqrt(lambda / m00)
ab =
36.6904
54.3880
```

in units of pixels. The ratio

```
>> ab(1)/ab(2)
ans =
0.6746
```

is the aspect ratio of the region and is a useful measure to characterise the shape of a region that is scale and rotation invariant.

The eigenvectors of  $J$  are the principal axes of the ellipse – the directions of its major and minor axes. The major, or principal, axis is the eigenvector  $v$  corresponding to the maximum eigenvalue. For our example this is

```
>> [x,lambda] = eig(J);
>> x
x =
-0.9954   -0.0962
-0.0962   0.9954
```

and since MATLAB® returns eigenvalues in ascending order  $v$  is always the *last* column of the returned eigenvector matrix

```
>> v = x(:,end);
```

The angle of this vector with respect to the horizontal axis is

$$\theta = \tan^{-1} \frac{v_y}{v_x}$$

and for our example this is

```
>> atan2( v(2), v(1) ) * 180/pi
ans =
95.5498
```

degrees which indicates that the major axis of the equivalent ellipse is approximately vertical. Finally we can superimpose the equivalent ellipse over the region

```
>> plot_ellipse(4*m00, [uc, vc], 'r');
```

and the result is shown in Fig. 13.12. The orientation and aspect ratio of the equivalent ellipse is a useful indicator of the region's shape and orientation.

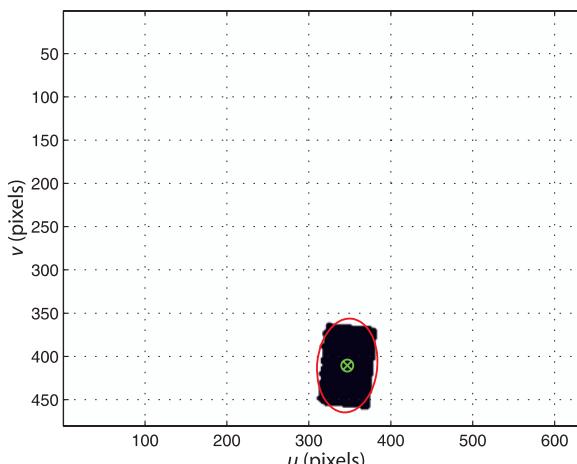
To summarize, we have created an image containing a spatially contiguous set of pixels corresponding to one of the objects in the scene that we segmented from the original color image. We have determined its area, a box that entirely contains it, the location of its centroid and its orientation.

The Toolbox provides a simpler way to do these useful things

```
>> blob = imoments(reg5)
blob =
area=6005, cent=(346.8,410.4), theta=1.67, a/b=0.675
```

which returns a `RegionFeature` object that contains many features describing this region including its area, its centroid, orientation and aspect ratio – the ratio of its minimum to maximum radius. These values are available as object properties, for example

```
>> blob.uc
ans =
346.7539
>> blob.theta
ans =
1.6677
>> blob.shape
ans =
0.6746
```



**Fig. 13.12.**

Equivalent ellipse and centroid for region 5 of the targets image

along with the zeroth- and first-order moments and the second-order central moments

```
>> blob.moments.m00
ans =
    6005
>> blob.moments.u11
ans =
 -2.3293e+05
```

The Toolbox provides a high-level function to compute features for *every* region in the image

```
>> f = iblobs(targets_binary)
f =
(1) area=5927, cent=(365.1,219.2), theta=-1.53, a/b=0.768, class=1,
    label=1, touch=0, parent=4
(2) area=5376, cent=(264.9,311.6), theta=0.10, a/b=0.707, class=1,
    label=2, touch=0, parent=4
(3) area=6315, cent=(452.3,327.1), theta=0.07, a/b=0.687, class=1,
    label=3, touch=0, parent=4
(4) area=283577, cent=(317.1,234.1), theta=-0.02, a/b=0.738, class=0,
    label=4, touch=1, parent=0
(5) area=6005, cent=(346.8,410.4), theta=-1.47, a/b=0.675, class=1,
    label=5, touch=0, parent=4
```

which returns a vector `f` of `RegionFeature` objects. The display method shows a summary of the region's properties and the number in parentheses indicates the index within the vector. Each `RegionFeature` object contains the area, bounding box, centroid, raw and central moments, and equivalent ellipse parameters as returned by `imoments` as well as additional properties such as the the class of the pixels within the region, the region label, the label of the parent region and whether or not the blob touches the edge.► Some examples of the properties of this class are

```
>> f(1).class
ans =
    1
>> f(1).parent
ans =
    4
>> f(1).touch
ans =
    0
>> f(1).umin
ans =
    328
>> f(1).shape
ans =
    0.7677
```

The moments and area calculations depend on knowledge of the shape of each pixel. In modern cameras pixels are square with typically more pixels in the horizontal than the vertical direction. Some vision system provide square images, for example  $512 \times 512$ , and the pixels can be wider than they are tall. In such a case the 'aspect' option should be provided to `iblobs` to define a non-unity pixel aspect ratio which is pixel height over pixel width.

The `RegionFeature` class also has plotting methods such as

```
>> f(1).plot_box('g')
```

which overlays the bounding box of feature `f(1)`, in green, on the current plot. Other plot methods include `plot_centroid` and `plot_ellipse`. All methods add to the current plot and can operate on a single object or a vector of objects, for example

```
>> f.plot_box('r:')
```

overlays the bounding box, in dotted red, for *all* blobs in `f`.

The `children` property is the inverse mapping of the `parent` property. It is a list of indices into the feature vector of `RegionFeature` objects which are children of this feature. For example the background blob, `f(4)`, has as its children

```
>> f(4).children
ans =
    1     2     3     5
blobs f(1), f(2), f(3) and f(5).
```

Importantly the function `iblobs` can perform filtering. For the garden image we might know something about the minimum and/or maximum size of a tomato so we can set bounds on the possible area

```
>> f = iblobs(tomatoes_binary, 'area', [1000, 5000])
f =
(1) area=1496, cent=(132.5,135.2), theta=0.21, a/b=0.871, class =1,
    label=2, touch=0, parent=1
(2) area=3380, cent=(95.8,210.9), theta=-0.31, a/b=0.886, class =1,
    label=3, touch=1, parent=0
```

which returns only blobs with an area between 1 000 and 5 000 pixels. For the tomato image we might wish to accept only blobs that do not touch the edge

```
>> f = iblobs(tomatoes_binary, 'touch', 0)
f =
(1) area=1496, cent=(132.5,135.2), theta=0.21, a/b=0.871, class =1,
    label=2, touch=0, parent=1
```

The filter rules can be cascaded, for example

```
>> f = iblobs(tomatoes_binary, 'touch', 0, 'area', [500 2000], 'class', 1);
```

and a blob must pass all rules in order to be accepted. Other filter parameters include pixel class, shape and aspect ratio and more details are provided in the online documentation.

### 13.1.3.3 Invariance

In order to recognize particular objects we would like to have some measure of shape that is invariant to the relative pose of the camera and the object. In this section we will be concerned only with planar objects that are fronto-parallel to the camera and which are subject to translation, rotation and scale change.

The shape of an object can be described very simply by the aspect ratio, the ratio of major to minor ellipse axis lengths  $a/b$ , and this is invariant to translation, rotation and scale. Another commonly used and intuitive shape feature is circularity which is defined as

$$\rho = \frac{4\pi m_{00}}{p^2} \quad (13.9)$$

where  $p$  is the region's perimeter length (discussed in the next section). Circularity has a maximum value of  $\rho=1$  for a circle, is  $\rho=\frac{\pi}{4}$  for a square and zero for an infinitely long line. Circularity is also invariant to translation, rotation and scale.

More complex ratios of moments can be used to form invariants for recognition of planar objects irrespective of position, orientation and scale. For example the image of Fig. 13.15a

```
>> im = iread('P2.png');
```

has two P-shaped regions

```
>> [f,L] = iblobs(im, 'class', 1);
>> f
f =
(1) area=1375, cent=(94.9,108.2), theta=1.98, a/b=0.648, color=1,
    label=2, touch=0, parent=1
(2) area=795, cent=(204.6,122.4), theta=2.76, a/b=0.645, color=1,
    label=4, touch=0, parent=1
```

**Moment invariants.** The normalized moments

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}}, \quad \gamma = \frac{1}{2}(p+q)+1 \quad \text{for } p+q=2,3,\dots \quad (13.8)$$

are invariant to translation and scale, and are computed by the Toolbox function `npcg`.

Third-order moments allow for the creation of quantities that are invariant to translation, scale and orientation within a plane. One such set of moments defined by Hu (1962) are

$$\begin{aligned}\phi_1 &= \eta_{20} + \eta_{02} \\ \phi_2 &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\ \phi_3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\ \phi_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\ \phi_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] \\ &\quad + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\ \phi_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\ &\quad + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\ \phi_7 &= (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] \\ &\quad + (3\eta_{12} - \eta_{30})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]\end{aligned}$$

and computed by the Toolbox function `humoments`.

	Translation	Rotation	Scale
Area	✓	✓	✗
Centroid	✗	✓	✓
Aspect ratio	✓	✓	✓
Orientation $\theta$	✓	✗	✓
Circularity	✓	✓	✓
Hu moments	✓	✓	✓

**Table 13.1.**  
Region features and their invariance to relative motion:  
translation, rotation about the object's centroid and scale factor

with labels 2 and 4. The second output argument is the label image. The moment invariants are

```
>> humoments(L==2)
ans =
    0.4181    0.0292    0.0230    0.0017    0.0000    0.0003    0.0000
>> humoments(L==4)
ans =
    0.4052    0.0280    0.0210    0.0014    0.0000    0.0002    0.0000
```

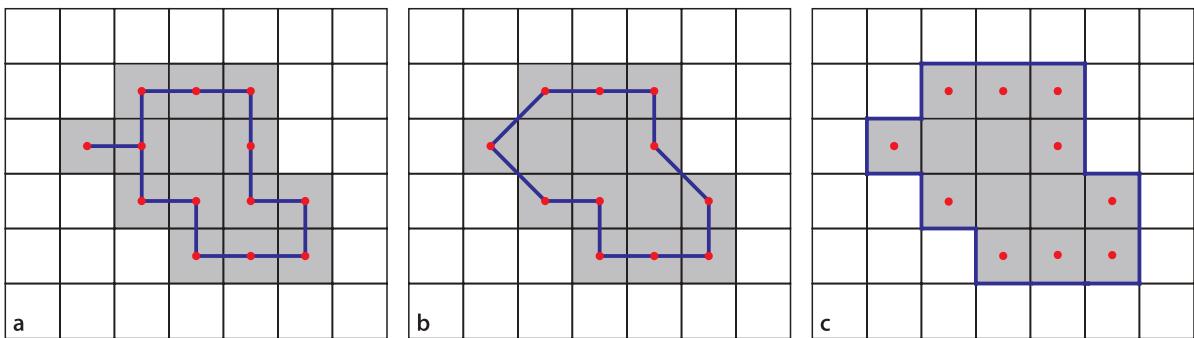
which are quite similar despite the different position, orientation and scale of the two shapes.► This shape descriptor can be considered as a point in 7-dimensional space, and similarity to other shapes can be defined in terms of Euclidean distance.

A summary of region features and their invariance properties is shown in Table 13.1.

In practice the discrete nature of the pixel data means that the invariance will only be approximate.

#### 13.1.3.4 Boundary Representation

A region can also be described by the shape of its boundary or perimeter. Figure 13.13 shows three common ways to represent the perimeter of a region. A chain code is a list of the outermost pixels of the region whose centre's are linked by short line segments. In the case of a 4-neighbour chain code the successive pixels must be adjacent and the perimeter segments have an orientation of  $k \times 90^\circ$ , where  $k \in \{0 \dots 3\}$ . With an 8-neighbour



**Fig. 13.13.** Boundary representations with region pixels shown in grey, perimeter segments shown in blue and the centre of boundary pixels marked by a red dot. **a** Chain code with 4 directions; **b** Freeman chain code with 8 directions; **c** crack code. The perimeter lengths for this example are respectively 14, 12.2 and 18 pixels

Since it is computationally more expensive.

chain code, or Freeman chain code, the perimeter segments have an orientation of  $k \times 45^\circ$ , where  $k \in \{0 \dots 7\}$ . The crack code has its segments in the cracks between the pixels on the edge of the region and the pixels outside the region. These have orientations of  $k \times 90^\circ$ , where  $k \in \{0 \dots 3\}$ .

The perimeter can be encoded as a list of pixel coordinates  $(u_i, v_i)$  or very compactly as a bit string using just 2 or 3 bits to represent  $k$  for each segment. These various representations are equivalent and any representation can be transformed to another.

Note that for chain codes the boundary follows a path that is on average half a pixel inside the true boundary and therefore underestimates the perimeter length. The error is most significant for small regions.

By default the Toolbox does not compute the boundary of objects, but this is enabled using the '`boundary`' option

```
>> f = iblobs(tomatoes_binary, 'boundary', 'class', 1)
f =
(1) area=1434, cent=(132.9,135.8), theta=-3.10, a/b=0.852, class=1,
label=2, touch=0, parent=1, perim=154, circ=0.760
(2) area=3380, cent=(95.9,210.9), theta=2.83, a/b=0.888, class=1,
label=3, touch=1, parent=0, perim=200, circ=1.062
(3) area=456, cent=(11.7,228.4), theta=-2.98, a/b=0.864, class=1,
label=4, touch=1, parent=0, perim=76, circ=0.992
```

The circularity parameter is seen to exceed one. This is due to the use of a 4-way chain code which slightly underestimates the perimeter length.

and we note that perimeter and circularity information are now displayed. We have used a blob filter here to select only blobs of class 1 (tomato colored) pixels. The boundary is a list of edge points represented as a matrix with one column per edge point. In this case there are

```
>> about(f(1).edge)
[double] : 2x154 (2464 bytes)
```

154 edge points and the first five points of the boundary are

```
>> f(1).edge(:,1:5)
ans =
142    142    141    140    139
116    117    118    119    120
```

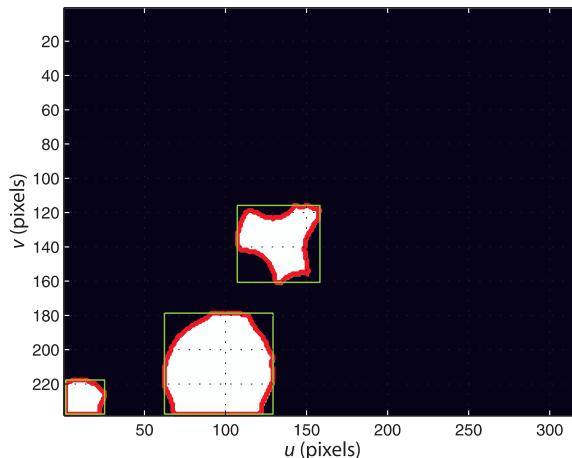
The boundary can be overlaid on the current plot using the object's `plot_boundary` method

```
>> f(1).plot_boundary('g')
```

in this case as a series of green dots. The plotting methods can be invoked on a feature vector

```
>> idisp(tomatoes_binary)
>> f.plot_boundary('r.')
>> f.plot_box('g')
```

which is shown in Fig. 13.14.



**Fig. 13.14.**  
Marked bounding boxes and  
perimeter for garden image

Every object has one external boundary, which may include the image border as is the case for the two lower blobs in Fig. 13.14. An object with holes has one internal boundary per hole but the Toolbox returns only the external boundary – the inner boundaries can be found as the external boundaries of the *holes* which are its child regions. The external boundary contains all the essential information about the shape of a region. In fact the moments can be computed given the boundary using the functions `mpq_poly`, `upq_poly` and `npq_poly` and assuming that the region has no holes.

Consider again the binary image

```
>> im = imread('P2.png');
```

shown in Fig. 13.15a which contains two similar shapes but with different translation, orientation and scale. The features of the white objects are

```
>> f = iblobs( im, 'boundary', 'class', 1)
f =
(1) area=1375, cent=(94.9,108.2), theta=-1.16, a/b=0.648, class=1,
    label=2, touch=0, parent=1, perim=204, circ=0.415
(2) area=795, cent=(204.6,122.4), theta=-0.38, a/b=0.645, class =1,
    label=4, touch=0, parent=1, perim=126, circ=0.629
>> f.plot()
```

and the two shapes have a similar aspect ratio. However the circularity, which should be the same, is quite different due to the underestimation of perimeter length as mentioned on page 357. The ratio of areas is

```
>> f(2).area / f(1).area
ans =
0.5782
```

which implies a reduction in scale by a factor of

```
>> sqrt(ans)
ans =
0.7604
```

The relative orientation of the two shapes is

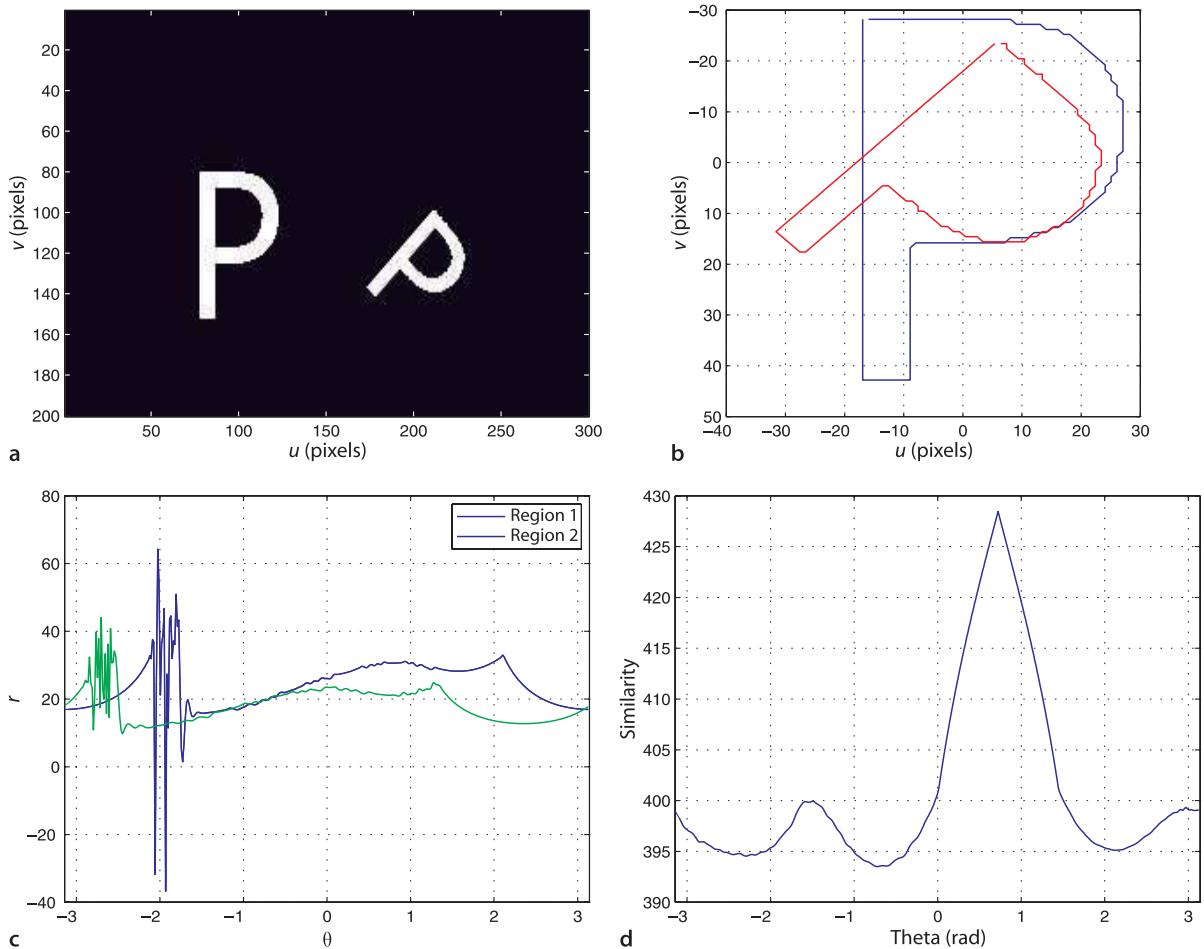
```
>> (f(2).theta - f(1).theta) * 180/pi
ans =
45.1267
```

degrees.

We can display the edge points with respect to the centroid

```
>> f(1).boundary();
```

which is shown in Fig. 13.15b for both regions. The optional return arguments provide the centroid-relative edge points in polar-coordinate form



**Fig. 13.15.** Region boundary matching. **a** The original binary image with object centroids marked; **b** object boundaries with respect to object centroids; **c** boundary signature, polar coordinate  $r$  versus  $\theta$ ; **d** correlation of the two boundary signatures

```
>> [r1,theta1] = f(1).boundary();
```

which is a unique signature for this region

```
>> plot(theta1, r1)
```

and shown in Fig. 13.15c. The angles are with respect to the top-leftmost point in the region and the polar coordinate vectors have 400 elements by default that span the interval  $[-\pi, \pi]$  rad. The shape profile for region 2

```
>> [r2, theta2] = f(2).boundary();
```

is also shown in Fig. 13.15c. The horizontal axis is an angle  $\theta \in \mathbb{S}$  so the graph is wrapped around a cylinder and the left- and right-hand edges are joined. We see that one profile is shifted horizontally (rotated) with respect to the other and the vertical magnitude is different due to the change in scale. If we normalize each profile by its maximum value then they will differ only by a horizontal shift. For each possible rotation of one profile with respect to the other we compute the similarity

```
>> [sim,scale] = boundmatch(r1, r2);
```

which we plot against angle

```
>> plot(theta1, sim)
```

and this is shown in Fig. 13.15d. A clear peak (strong similarity) indicates the relative rotation of

```
>> [s,th] = peak(sim, theta1, 'npeaks', 1);
>> th * 180/pi
ans =
41.4004
```

degrees which is in the ballpark of the known rotation and the previous estimate. The estimated scale ratio is

```
>> 1/scale
ans =
0.7651
```

Compared to moments the perimeter is a richer and higher-dimensional shape descriptor, in this case a 400-element vector. From the perimeter we can identify straight line segments and curves.

---

#### 13.1.4 Recap

We have discussed how to convert an input image, grey scale or color, into concise descriptors of regions within the scene. The criteria for what constitutes a region is *application specific*. For a tomato picking robot it would be round red regions, for landing a UAV it might be yellow targets on the ground.

The process outlined is the classical *bottom up* approach to machine vision applications and the key steps are:

1. Classify the pixels according to the application specific criterion, for example, redness, yellowness or motion. Each pixel is assigned a class  $c$ .
2. Group adjacent pixels of the same class into sets, and each pixel is assigned a label  $s$  indicating the set to which it has been assigned.
3. Describe the sets in terms of features derived from their spatial extent, moments, equivalent ellipse and boundary.

These steps are a progression from *low-level* to *high-level*. The low-level operations consider pixels in isolation, whereas the high-level is concerned with more abstract concepts such as size and shape. The MSER and graphcuts algorithms are powerful because they combines steps 1 and 2 and consider regions of pixels and localized differences in order to create a segmentation.

Importantly none of these steps need be perfect. Perhaps the first step has some false positives which will be small noise pixels that we can eliminate by morphological operations, or by rejecting them after connectivity analysis based on their size. The first step may also have false negatives, for example specular reflection and occlusion may cause some pixels to be classified incorrectly. In this case we need to develop some heuristics, for instance morphological processing to fill in the gaps in the fruit due to reflection. Another option is to oversegment the scene – increase the number of regions and use some application-specific knowledge to merge adjacent regions. For example a specular reflection colored region might be merged with surrounding regions to create a region corresponding to the whole fruit.

An alternative, and some would argue better, approach is to control the lighting so as to create the best possible image to start with. The stray light causing the unwanted reflection could be turned off or blocked, or a polarizing filter could be used to attenuate the reflection. Ideally the illumination would be diffuse lighting from behind the camera such as from a ring illuminator.

Domain knowledge is always a powerful tool. Given that we know the scene contains tomatoes and plants, the fact that we observe a large red region that is not circular, using our domain knowledge we can infer that the fruit is occluded. We therefore might command the robot to seek the fruit that is not occluded, and then to move to

another location where the occluded fruit might be accessible. Object segmentation remains one of the hardest aspects of machine vision and there is no silver bullet. It requires knowledge of image formation, fundamental image processing algorithms, insight, a good box of tools and patience.

## 13.2 Line Features

Lines are distinct visual features that are particularly common in man-made environments – for example the edges of roads, buildings and doorways. In Sect. 12.4.1.3 we discussed how image intensity gradients can be used to find edges within an image, and this section will be concerned with fitting line segments to such edges.

We will illustrate the principle using the very simple scene

```
>> im = imread('5points.png', 'double');
```

shown in Fig. 13.16a. Consider any one of these points – there are an infinite number of lines that pass through that point. If the point could vote for these lines, then each possible line passing through the point would receive one vote. Now consider another point that does the same thing, casting a vote for all the possible lines that pass through it. One line (the line that both points lie on) will receive a vote from each point – a total of two votes – while all the other possible lines receive either zero or one vote.

We need to describe each line in terms of a minimum number of parameters but the standard form  $v = mu + c$  is problematic for the case of vertical lines where  $m = \infty$ . Instead it is common to represent lines using the  $(\rho, \theta)$  parameterization

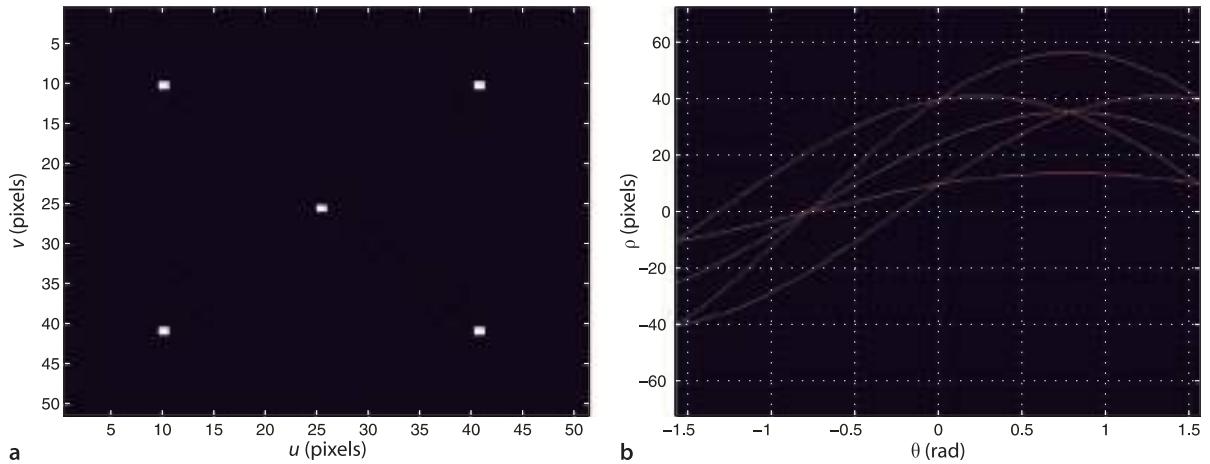
$$v = -u \tan \theta + \frac{\rho}{\cos \theta} \quad (13.10)$$

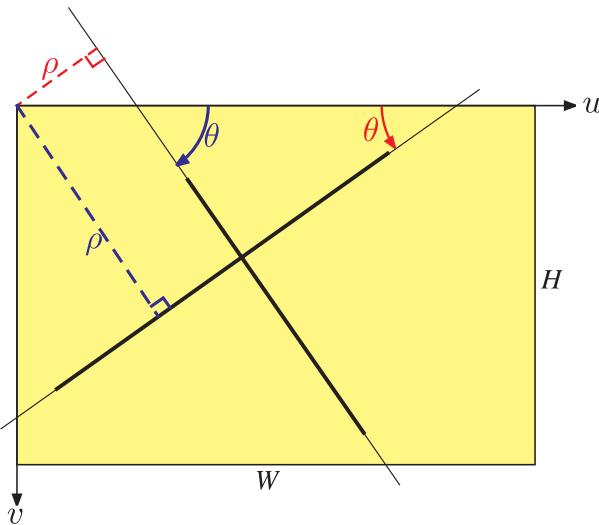
**Fig. 13.16.** Hough transform fundamentals. **a** Five points that define six lines; **b** the Hough accumulator array. The horizontal axis is an angle  $\theta \in \mathbb{S}$  so we can imagine the graph wrapped around a cylinder and the left- and right-hand edges joined. The sign of  $\rho$  also changes at the *join* so the the curve intersections on the left- and right-hand edges are equivalent

where  $\theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]$  is the angle from the horizontal axis to the perpendicular, and  $\rho \in [-\rho_{\min}, \rho_{\max}]$  is the perpendicular distance between the origin and the line. This is shown in Fig. 13.17. A horizontal line has  $\theta = 0$  and a vertical line has  $\theta = -\frac{\pi}{2}$ . Each line can therefore be considered a point  $(\rho, \theta)$  in the 2-dimensional space of all possible lines

In practice we cannot consider an infinite number of lines through each point, so we consider lines drawn from a finite set. The  $\theta\rho$ -space is quantized and a corresponding  $N_\theta \times N_\rho$  array  $A$  is used to tally the votes – the *accumulator array*. For a  $W \times H$  input image

$$\rho_{\max} = -\rho_{\min} = \max(W, H)$$





**Fig. 13.17.**  
 $(\theta, \rho)$  parameterization for two line segments. Positive quantities are shown in blue, negative in red

$A$  has  $N_\rho$  elements spanning the interval  $\rho \in [-\rho_{\max}, \rho_{\max}]$  and  $N_\theta$  elements spanning the interval  $\theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ . The indices of the array are integers  $(i, j) \subset \mathbb{Z}^2$  such that

$$\begin{aligned} i \in [1, N_\theta] &\mapsto \theta \in [-\frac{\pi}{2}, \frac{\pi}{2}] \\ j \in [1, N_\rho] &\mapsto \rho \in [-\rho_{\max}, \rho_{\max}] \end{aligned}$$

An edge point  $(u, v)$  votes for all lines that satisfy Eq. 13.10 which is all  $(i, j)$  pairs for which

$$\rho = u \sin \theta + v \cos \theta \quad (13.11)$$

and the elements  $A[i, j]$  are all incremented. For every  $i \in [1, N_\theta]$  the corresponding value of  $\theta$  is computed, then  $\rho$  is computed according to Eq. 13.11 and mapped to a corresponding integer  $j$ .

At the end of the process those elements of  $A$  with the largest number of votes correspond to dominant lines in the scene. For the example of Fig. 13.16a the resulting accumulator array is shown in Fig. 13.16b. Most of the array contains zero votes (black) and the red curves are trails of single votes corresponding to each of the five input points. These curves intersect and those points correspond to lines with more than one vote. We see four locations where two curves intersect, resulting in cells with two votes, and these correspond to the lines joining the four outside points of Fig. 13.16a. The horizontal axis represents angle  $\theta \in \mathbb{S}$  so the left- and right-hand ends are joined and  $d$  changes sign – the curve intersection points on the left- and right-hand sides of the array are equivalent. We also see two locations where three curves intersect, resulting in cells with three votes, and these correspond to the diagonal lines that include the middle point of Fig. 13.16a. This technique is known as the Hough transform.

Consider the more complex example of a solid square rotated counter-clockwise by 0.3 rad

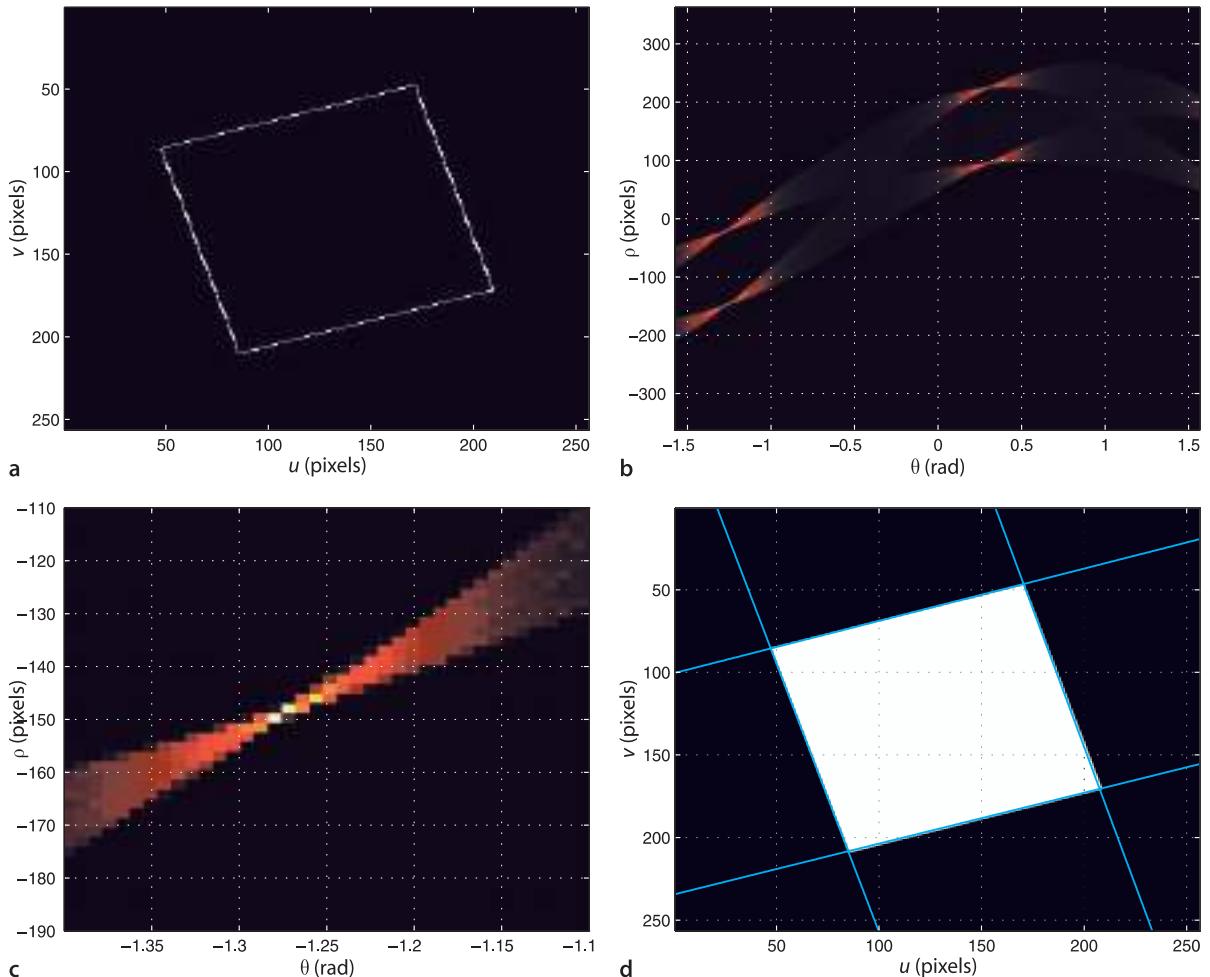
```
>> im = testpattern('squares', 256, 256, 128);
>> im = irotate(im, -0.3);
```

We compute the edge points

```
>> edges = icanny(im);
```

which are shown in Fig. 13.18a. The Hough transform is computed by

```
>> h = Hough(edges)
Hough: nd=401, ntheta=400, interp=3x3, distance=1
```



**Fig. 13.18.** Hough transform for a rotated square. **a** Edge image; **b** Hough accumulator; **c** closeup view of the Hough accumulator; **d** estimated lines overlaid on the original image

$\rho$  is symmetric about zero, so including zero this is an odd number of elements.  $\theta$  has a range of  $[-\frac{\pi}{2}, \frac{\pi}{2}]$ , it is asymmetric about zero and has an even number of elements.

and returns an instance of the `Hough` class. Its properties include the two-dimensional vote accumulator array `A` with `nd` elements in the row direction and `ntheta` elements in the column direction. By default the  $\theta\rho$ -plane is quantized into  $401 \times 400$  bins. The accumulator array can be visualized as an image

```
>> h.show();
```

which is shown in Fig. 13.18b. The four bright spots correspond to dominant edges in the input image. We can see that many other possible lines have received a small number of votes as well.

The next step is to find the peaks in the accumulator array

```
>> lines = h.lines()
lines =
theta=-1.27224, rho=-19.9157, strength=1
theta=-1.28025, rho=-150.009, strength=0.944099
theta=-1.25704, rho=-146.514, strength=0.801242
theta=0.298531, rho=224.022, strength=0.757764
theta=0.306266, rho=95.7803, strength=0.745342
theta=0.28664, rho=93.9642, strength=0.732919
theta=-1.24916, rho=-16.353, strength=0.68323
theta=0.279094, rho=222.237, strength=0.627329
theta=-1.29124, rho=-22.7355, strength=0.614907
theta=-1.29922, rho=-152.765, strength=0.565217
theta=0.322013, rho=225.884, strength=0.515528
```

which returns a vector of `LineFeature` objects corresponding to the lines with the most votes, as well as the number of votes associated with that line normalized with respect to the largest vote. If the function is called without output arguments the identified peaks are indicated on an image of accumulator vote strength.

Note that although the object has only four sides there are many more than four peaks in the accumulator array. We also note that the first and fourth peaks have quite similar line parameters, and this region of the accumulator is shown in more detail in Fig. 13.18c. We see several bright spots (high numbers of votes) that are close together and this is due to quantization effects. The concept of peak scale discussed on page 294 applies here and once again we apply non-local maxima suppression to eliminate smaller peaks in the neighbourhood of the maxima

```
>> h = Hough(edges, 'suppress', 5)
h =
Hough: nd=401, ntheta=400, interp=3x3, distance=5
```

In this case distance is five accumulator cells – the maxima suppresses smaller local maxima within a five cell radius. This leads to just four peaks

```
>> lines = h.lines()
lines =
theta=-1.27224, rho=-19.9157, strength=1
theta=-1.28025, rho=-150.009, strength=0.944099
theta=0.298531, rho=224.022, strength=0.757764
theta=0.306266, rho=95.7803, strength=0.745342
```

corresponding to the edges of the object.

Since the line parameters are quantized the `lines` method uses interpolation to refine the location of the peak (see Appendix K). By default interpolation is performed over a  $3 \times 3$  window centred on the local vote maxima. Once a peak has been found all votes within the suppression distance are zeroed so as to eliminate any close maxima and the process is repeated for all peaks in the voting array that exceed a specified fraction of the largest peak.►

The detected lines can be projected onto the original image

```
>> idisp(im);
>> h.plot('b')
```

and the result is shown in Fig. 13.18d.

A real image example is

```
>> im = iread('church.png', 'grey', 'double');
>> edges = icanny(im);
>> h = Hough(edges, 'suppress', 5);
>> lines = h.lines();
```

and the strongest ten lines

```
>> idisp(im);
>> lines(1:10).plot();
```

With no argument all peaks greater than '`houghThresh`' are displayed. This defaults to 0.5 but can be set by the '`houghThresh`' option to `Hough`.

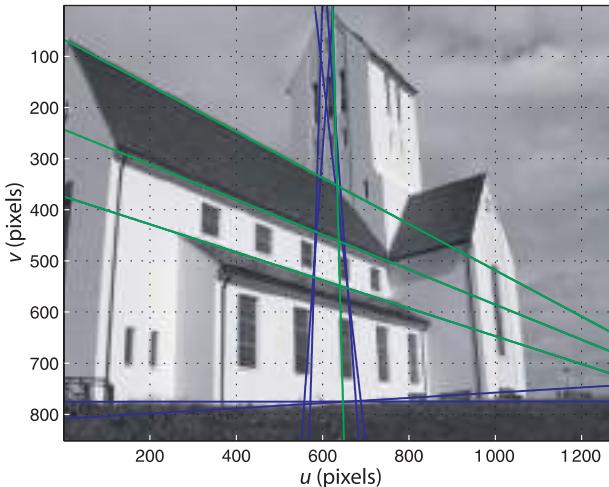
are shown in Fig. 13.19. Many strong lines in the image have been found, but the vertical lines caused by the window edges on the steeple suffer from angular quantization error over their short length and are not quite vertical.

Another measure of the importance of an edge can be found by reprojecting the line onto the edge image and counting the maximum number of *contiguous* edge pixels that lie along it

```
>> lines = lines.seglength(edges);
```

which returns a vector of `LineFeature` objects similar to that returned by the `lines` method but with the property `length` set to the maximum edge segment length

```
>> lines(1)
ans =
theta=0.000638093, rho=775.415, strength=1, length=47
```



**Fig. 13.19.**

Hough transform of a real image. The blue lines correspond to the ten strongest voting peaks. The overlaid green lines are those with an edge segment length of at least 100 pixels

in this case 47 pixel. An edge segment is defined as an almost contiguous group of edge pixels with no gap greater than five (by default) pixels. We can then choose all those Hough peaks corresponding to segments longer than 100 pixels

```
>> k = find( lines.length_v > 100);
```

and then highlight those lines in green

```
>> lines(k).plot('g')
```

as shown in Fig. 13.19. The three non-vertical lines converge on a perspective vanishing point.

The Hough transform is elegant in principle and in practice it can either work well or infuriatingly badly. It performs poorly when the scene contains a lot of texture or the edges are indistinct. Texture causes votes to be cast widely, but not uniformly, over the accumulator array which tends to mask the true peaks. Consequently a lot of experimentation is required for the parameters of the edge detector and the various thresholds within the Hough peak detector. The function `Hough` has many options which are described in the online documentation. The vote cast by each edge point can be one, or the edge strength at that point which emphasizes stronger edges. Edge strengths less than `edgeThresh` times the maximum edge strength are considered as zero. The `Hough` object can also be constructed from an array of edge coordinates and votes in which case the first argument is a  $2 \times N$  matrix of  $(u, v)$  coordinates, one per column, with equal votes. Alternatively if the input is a  $3 \times N$  matrix the third row is the strength of the vote to be cast.

The Hough transform estimates the direction of the line by fitting lines to the edge pixels. It ignores rich information about the direction of the edge at each pixel which was discussed on page 308. The consequence of not using all the information available to us is, ultimately, poorer estimation. There is little added expense in using the direction at each pixel since we have already computed the image gradients in order to evaluate edge magnitude.

### 13.3 Point Features

The final class of features that we will discuss are point features. These are visually distinct in the image and often called interest points, salient points, keypoints or commonly, but less precisely, corner points. We will first introduce some classical techniques for finding interest points and then discuss more recent scale-invariant techniques.

### 13.3.1 Classical Corner Detectors

We recall from Sect. 12.4.1.3 that a point on a line has a strong gradient in a direction normal to the line. However gradient *along* the line is low which means that a pixel on the line will look very much like its neighbours on the line. In contrast, an *interest point* is a point that has a high image gradient in orthogonal directions. It might be a single pixel that has a significantly different intensity to all of its neighbours or it might literally be a pixel on the corner of an object. Since corner points are quite distinct they have a much higher likelihood of being reliably detected in different views of the same scene. They are therefore key to multiview techniques such as stereo and motion estimation which we will discuss in the next chapter.

The earliest corner point detector was Moravec's *interest operator*, so called because it indicated points in the scene that were *interesting* from a tracking perspective. It was based on the intuition that if an image region  $\mathcal{W}$  is to be unambiguously located in another image it must be sufficiently different to all overlapping adjacent regions. Moravec defined the similarity between a region centred at  $(u, v)$  and an adjacent region, displaced by  $(\delta_u, \delta_v)$ , as

$$s(u, v, \delta_u, \delta_v) = \sum_{(i,j) \in \mathcal{W}} (I[u + \delta_u + i, v + \delta_v + j] - I[u + i, v + j])^2 \quad (13.12)$$

where  $\mathcal{W}$  is some local image region and typically an  $N \times N$  square window. This is the SSD similarity measure from Table 12.1 that we discussed previously. Similarity is evaluated for displacements in eight cardinal<sup>►</sup> directions  $(\delta_u, \delta_v) \in \mathcal{D}$  and the minimum value is the interest measure

$$C_M(u, v) = \min_{(\delta_u, \delta_v) \in \mathcal{D}} s(u, v, \delta_u, \delta_v) \quad (13.13)$$

The function  $C_M(\cdot)$  is evaluated for every pixel in the image and interest points are those where  $C_M$  is *high*. The main limitation of the Moravec detector is that it is non-isotropic since it examines image change, essentially gradient, in a limited number of directions. Consequently the detector can give a strong output for a point on a line, which is not desirable.

We can generalize the approach by defining the similarity as the weighted sum of squared differences between the image region and the displaced region as

$$s(u, v, \delta_u, \delta_v) = \sum_{(i,j) \in \mathcal{W}} W[i, j] \left( \underbrace{I[u + \delta_u + i, v + \delta_v + j]} - I[u + i, v + j] \right)^2$$

where  $W$  is a weighting matrix that emphasises points closer to the centre of the window  $\mathcal{W}$ . The indicated term can be approximated by a truncated Taylor series

$$I[u + \delta_u + i, v + \delta_v + j] \approx I[u + i, v + j] + I_u[u + i, v + j]\delta_u + I_v[u + i, v + j]\delta_v$$

where  $I_u$  and  $I_v$  are the horizontal and vertical image gradients respectively. We can now write

$$\begin{aligned} s(u, v, \delta_u, \delta_v) &= \sum_{(i,j) \in \mathcal{W}} W[i, j] (I_u[u + i, v + j]\delta_u + I_v[u + i, v + j]\delta_v)^2 \\ &= \delta_u^2 \sum_{(i,j) \in \mathcal{W}} W[i, j] I_u^2[u + i, v + j] + \delta_v^2 \sum_{(i,j) \in \mathcal{W}} W[i, j] I_v^2[u + i, v + j] \\ &\quad + \delta_u \delta_v \sum_{(i,j) \in \mathcal{W}} W[i, j] I_u[u + i, v + j] I_v[u + i, v + j] \end{aligned}$$

which can be written compactly as

N, NE, E, ... W, NW or  $i, j \in \{-1, 0, 1\}$ .

$$s(u, v, \delta_u, \delta_v) = (\delta_u \quad \delta_v) A \begin{pmatrix} \delta_u \\ \delta_v \end{pmatrix}$$

where

$$A = \begin{pmatrix} \Sigma W[i, j] I_u^2[u + i, v + j] & \Sigma W[i, j] I_u[u + i, v + j] I_v[u + i, v + j] \\ \Sigma W[i, j] I_u[u + i, v + j] I_v[u + i, v + j] & \Sigma W[i, j] I_v^2[u + i, v + j] \end{pmatrix}$$

If the weighting matrix is a Gaussian kernel  $W = G(\sigma_I)$  and we replace the summation by a convolution then

$$A = \begin{pmatrix} G(\sigma_I) \otimes I_u^2 & G(\sigma_I) \otimes I_u I_v \\ G(\sigma_I) \otimes I_u I_v & G(\sigma_I) \otimes I_v^2 \end{pmatrix} \quad (13.14)$$

which is a symmetric  $2 \times 2$  matrix referred to variously as the structure tensor, auto-correlation matrix or second moment matrix. It captures the intensity structure of the local neighbourhood and its eigenvalues provide a rotationally invariant description of the neighbourhood. The elements of the  $A$  matrix are computed from the image gradients, squared or multiplied, and then smoothed using a weighting matrix. This reduces noise and improves the stability and reliability of the detector. The gradient images  $I_u$  and  $I_v$  are typically calculated using a derivative of Gaussian kernel method (Sect. 12.4.1.3) with a smoothing parameter  $\sigma_D$ .

An interest point  $(u, v)$  is one for which  $s(\cdot)$  is high for *all* directions of the vector  $(\delta_u, \delta_v)$ . That is, in whatever direction we move the window it rapidly becomes dissimilar to the original region. If we consider the original image  $I$  as a surface the eigenvalues of  $A$  are the principal curvatures of the surface at that point. If both eigenvalues are small then the surface is flat, that is the image region has approximately constant local intensity. If one eigenvalue is high and the other low, then the surface is ridge shaped which indicates an edge. If both eigenvalues are high the surface is sharply peaked which we consider to be a corner. ▶

The Shi-Tomasi detector considers the strength of the corner, or *cornerness*, as the minimum eigenvalue

$$C_{ST}(u, v) = \min(\lambda_1, \lambda_2) \quad (13.15)$$

where  $\lambda_i$  are the eigenvalues of  $A$ . Points in the image for which this measure is high are referred to as “*good features to track*”. The Harris detector ▶ is based on this same insight but defines corner strength as

$$C_H(u, v) = \det(A) - k \text{tr}(A) \quad (13.16)$$

and again a large value represents a strong, distinct, corner. Since  $\det(A) = \lambda_1 \lambda_2$  and  $\text{tr}(A) = \lambda_1 + \lambda_2$  the Harris detector responds when both eigenvalues are large and elegantly avoids computing the eigenvalues of  $A$  which has a somewhat higher computational cost. ▶ A commonly used value for  $k$  is 0.04. Another variant is the Noble detector

$$C_N(u, v) = \frac{\det(A)}{\text{tr}(A)} \quad (13.17)$$

which is arithmetically simple but potentially singular.

Typically the corner strength is computed for every pixel resulting in a corner strength image. Then non-local maxima suppression is applied to only retain values that are greater than their immediate neighbours. A list of such points is created and sorted into descending corner strength. A threshold can be applied to only accept corners above a particular strength, or above a particular fraction of the strongest corner, or simply the strongest  $N$  corners.

Recall that image compression removes high-frequency detail from the image, and this is exactly what defines a corner. Ideally corner detectors should be applied to images that have not been compressed and decompressed.

Sometimes referred to in the literature as the Plessey corner detector.

Evaluating eigenvalues for a  $2 \times 2$  matrix involves solving a quadratic equation and therefore requires a square root operation.

The Toolbox provides a Harris corner detector which we will demonstrate using a real image

```
>> b1 = imread('building2-1.png', 'grey', 'double');
>> idisp(b1)
```

The Harris features are computed by

```
>> C = icorner(b1, 'nfeat', 200);
7712 corners found (0.8%), 200 corner features saved
```

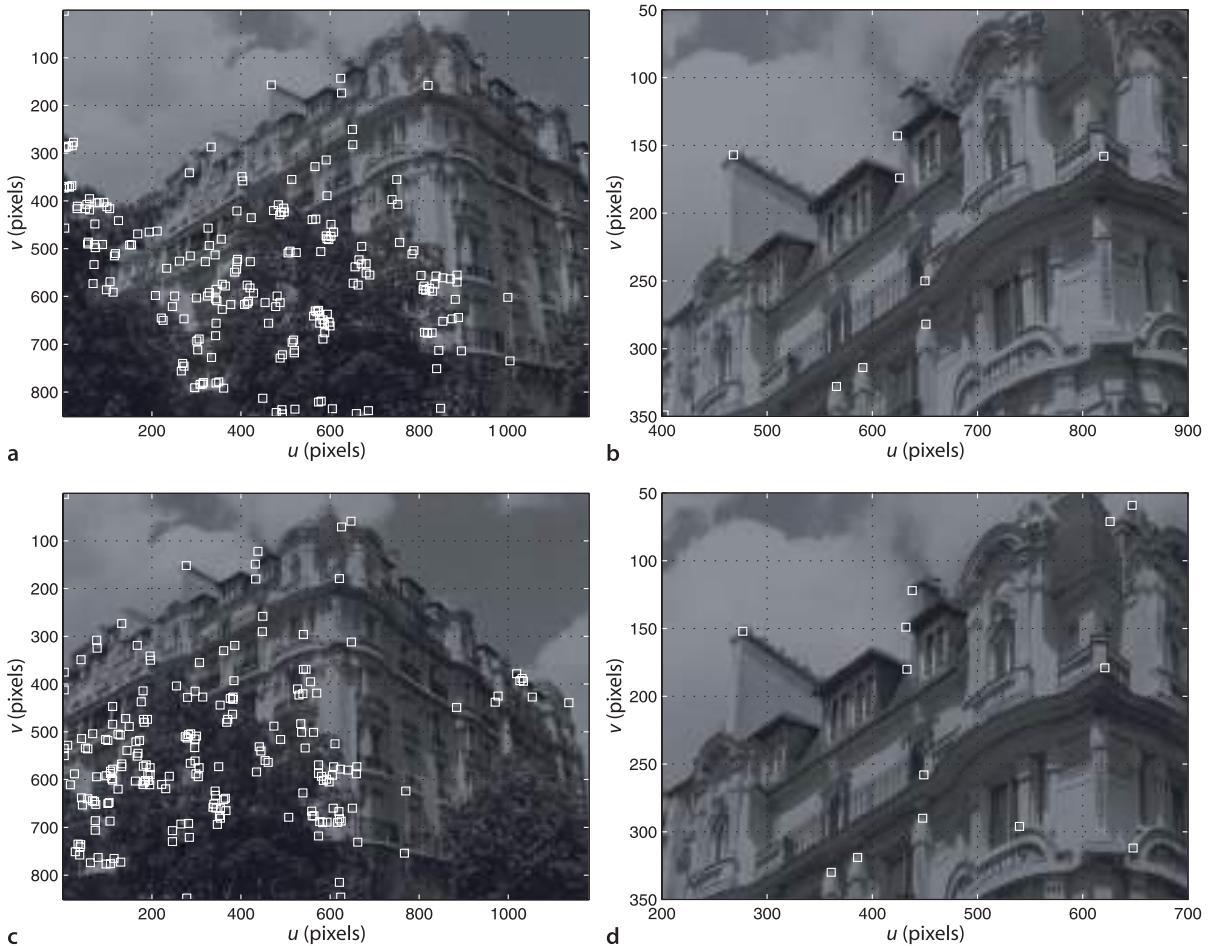
which returns a vector of `PointFeature` objects. The detector found over 7000 corners that were local maxima of the corner strength image and these comprised 0.8% of all pixels in the image. In this case we requested the 200 strongest corners. The vector contains the corners sorted by decreasing corner strength, and each `PointFeature` object contains the corner coordinate ( $u, v$ ), the corner strength and a descriptor which comprises the unique elements of the structure tensor in vector form ( $A_{11}, A_{22}, A_{12}$ ). The descriptor can be used as a simple signature of the corner to help match corresponding corners between different views.

The corners can be overlaid on the image as white squares

```
>> idisp(b1, 'dark');
>> C.plot('ws');
```

as shown in Fig. 13.20a. The 'dark' option to `idisp` reduces the brightness of the image to make the corner markers more visible. A closeup view is shown in Fig. 13.20b and we see the features are indeed often located on the corners of objects.

**Fig. 13.20.** Harris corner detector applied to two views of the same building. **a** View one; **b** zoomed in view one; **c** view two; **d** zoomed in view two. Notice that quite a number of the detected corners are attached to the same world features in the two views



Another approach to determining image curvature is to use the determinant of the Hessian (DoH). The Hessian is the matrix of second-order gradients at a point

$$H = \begin{pmatrix} I_{uu} & I_{uv} \\ I_{uv} & I_{vv} \end{pmatrix}$$

where  $I_{uu} = \partial^2 I / \partial u^2$ ,  $I_{vv} = \partial^2 I / \partial v^2$  and  $I_{uv} = \partial^2 I / \partial u \partial v$ . The determinant  $\det(H)$  is high when there is grey-level variation in two directions. However second derivatives accentuate image noise even more than first derivatives and the image must be smoothed first.

We also see that the corners tend to cluster unevenly, with a greater density in regions of high contrast and texture, and for some applications this can be problematic. To distribute corner points more evenly we can increase the distance used for non-local maxima suppression

```
>> C1 = icorner(b1, 'nfeat', 200, 'suppress', 10);
7422 corners found (0.7%), 200 corner features saved
```

by specifying a minimum distance between corners, in this case 10 pixel.

We can apply standard MATLAB® operations and syntax to vectors of `PointFeature` objects, for example

```
>> length(C)
ans =
200
```

and indexing

```
>> C(1:4)
ans =
(3,3), strength=0.00760968, descrip=(0.112697 0.112524 0.0551586)
(600,662), strength=0.0054555, descrip=(0.0910059 0.0716812 -0.00303761)
(24,277), strength=0.0039721, descrip=(0.0923745 0.0577764 -0.021521)
(54,407), strength=0.00393328, descrip=(0.0998847 0.0482311 0.00259472)
```

where the display method shows the essential properties of the feature. We can create expressions such as

```
>> C(1:5).strength
ans =
0.0076    0.0055    0.0040    0.0039    0.0038
>> C(1).u
ans =
3
```

To plot the coordinate of every fifth feature in the first 100 features is

```
>> C(1:5:100).plot()
```

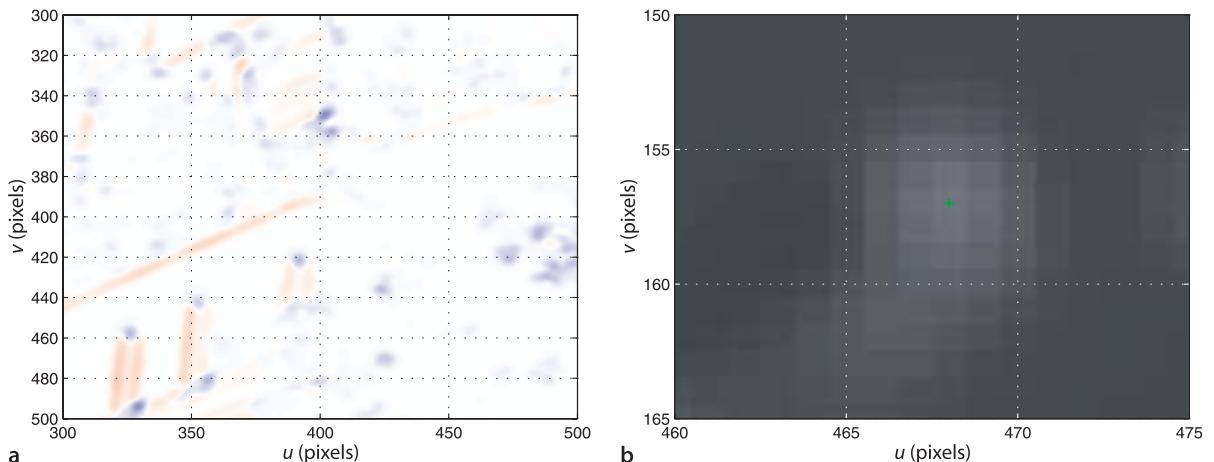
The corner strength is computed at each pixel and can be optionally returned

```
>> [C,strength] = icorner(b1, 'nfeat', 200);
7422 corners found (0.7%), 200 corner features saved
```

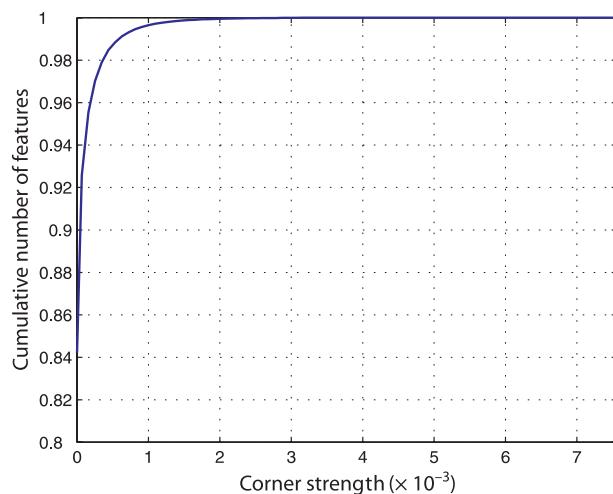
and displayed as an image

```
>> idisp(strength, 'invscaled')
```

which is shown in Fig. 13.21a. We observe that the corner strength function is strongly positive (blue) for corner features and strongly negative (red) for linear features. A zoomed in view is shown in Fig. 13.21b which indicates that the detected corner is at the top of a peak of *cornerness* that is several pixels wide. The detected corner is a local maxima but we could use the surrounding values to estimate its location to sub-pixel accuracy (see Appendix K). This involves additional computation but can be enabled using the option '`interp`'.



**Fig. 13.21.** Harris corner strength.  
a Zoomed view of corner strength displayed as an image (blue is positive, red is negative); b zoomed view of corner strength image (brightness proportional to strength) with detected feature shown



**Fig. 13.22.**  
Cumulative histogram of positive corner strengths

A cumulative histogram of the strength of the detected corners

```
>> ihist(strength, 'normcdf')
```

is shown in Fig. 13.22. The strongest corner has  $C_H \approx 0.007$  and more than 95% of corners exceed half this value.

Consider another image of the same building taken from a different location

```
>> b2 = iread('building2-2.png', 'grey', 'double');
```

and the detected corners

```
>> C2 = icorner(b2, 'nfeat', 200);
7666 corners found (0.8%), 200 corner features saved
>> idisp(b2,'dark')
>> C2.plot('ws');
```

are shown in Fig. 13.20c and d. For many useful applications in robotic vision – such as tracking, mosaicing and stereo vision that we will discuss in the next chapter – it is important that corner features are detected at the same world points irrespective of variation in illumination or changes in rotation and scale between the two views. From Fig. 13.20 we see that many, but not all, of the features are indeed *attached* to the same world feature in both views.

The Harris detector is computed from image gradients and is therefore robust to offsets in illumination, and the eigenvalues of the structure tensor  $A$  are invariant to rotation. However the detector is not invariant to changes in scale. As we *zoom in* the

gradients around the corner point become lower – the same change in intensity is spread over a larger number of pixels. This reduces the image curvature and hence the corner strength. The next section discusses a remedy for this using *scale-invariant* corner detectors.

For a color image the structure tensor is computed using the gradient images of the individual color planes which is slightly different to first converting the image to grey scale according to Eq. 10.10. In practice the use of color defies intuition – it makes surprisingly little difference for most scenes but adds significant computational cost. The `icorner` function accepts a large number of options to change:  $k$ , the derivative and smoothing kernel sizes  $\sigma_D$  and  $\sigma_b$ , absolute and/or relative corner strength threshold and enforcing a minimum distance between corners. The options '`st`' and '`noble`' allow computation of the corner measures Eq. 13.15 and Eq. 13.17 respectively. Details are provided in the online documentation.

### 13.3.2 Scale-Space Corner Detectors

The Harris corner detector introduced in the previous section works very well in practice but responds poorly to changes in scale. Unfortunately change in scale, due to changing camera to scene distance, is common in many real applications. We also notice that the Harris detector responds strongly to fine texture, such as the leaves of the trees in Fig. 13.20 but we would like to be able to detect features that are associated with larger-scale scene structure.

Figure 13.23 illustrates the fundamental principle of scale-space feature detection. We first load a synthetic image

```
>> im = imread('scale-space.png', 'double');
```

which is shown in Fig. 13.23a. The image contains four squares of different size:  $5 \times 5$ ,  $9 \times 9$ ,  $17 \times 17$  and  $33 \times 33$ . The scale-space sequence is computed by applying a Gaussian kernel with increasing  $\sigma$  that results in the regions becoming increasingly blurred and smaller regions progressively disappearing from view. At each step in the sequence the Gaussian-smoothed image is convolved with the Laplacian kernel Eq. 12.3 which results in a strong negative responses for these bright blobs.◀

With the Toolbox we compute the scale-space sequence by

```
>> [G,L,s] = iscalespace(im, 60, 2);
```

where the input arguments are the number of scale steps to compute, and the  $\sigma$  of the Gaussian kernel to be applied at each successive step. The function returns two 3-dimensional images, each a sequence of images where the last index corresponds to the scale. `G` is the image `im` at increasing levels of smoothing, `L` is the Laplacian of the smoothed images, and `s` is the corresponding scale. For example the fifth image in the Laplacian of Gaussian sequence is displayed by

```
>> idisp(L(:,:5), 'invsignd')
```

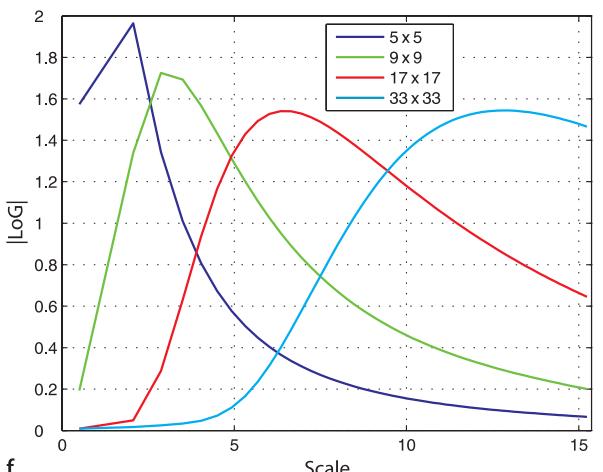
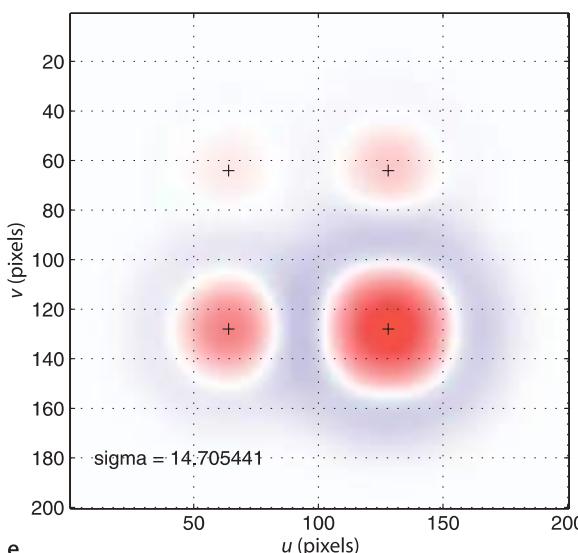
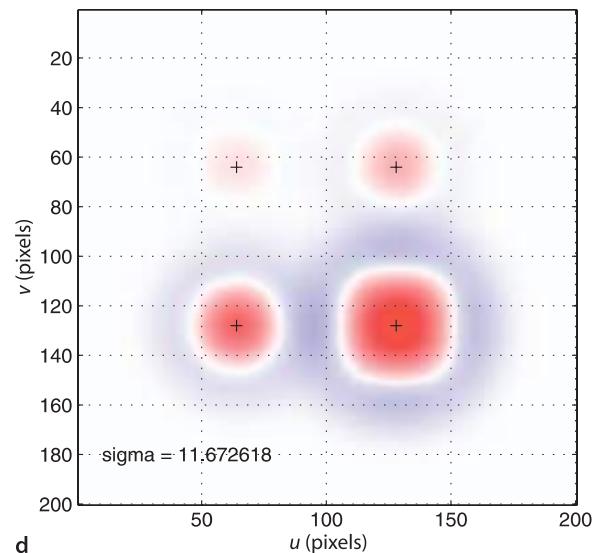
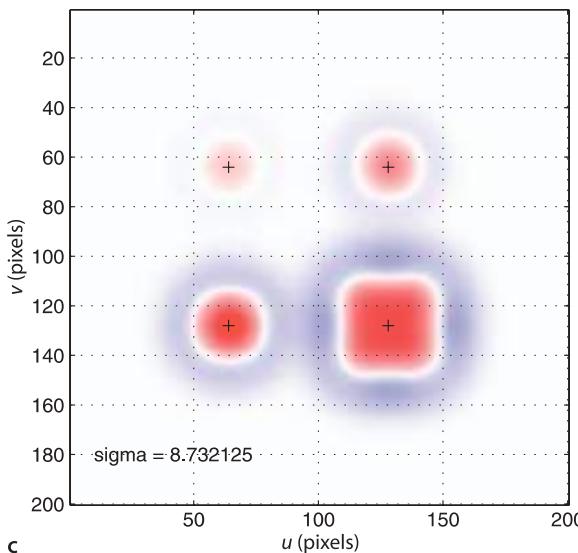
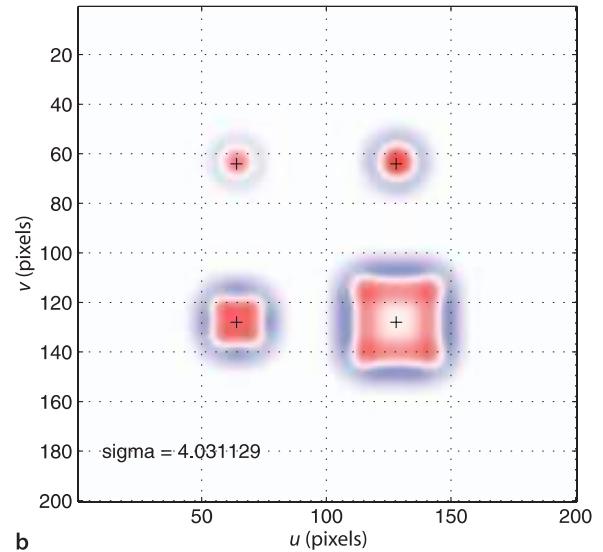
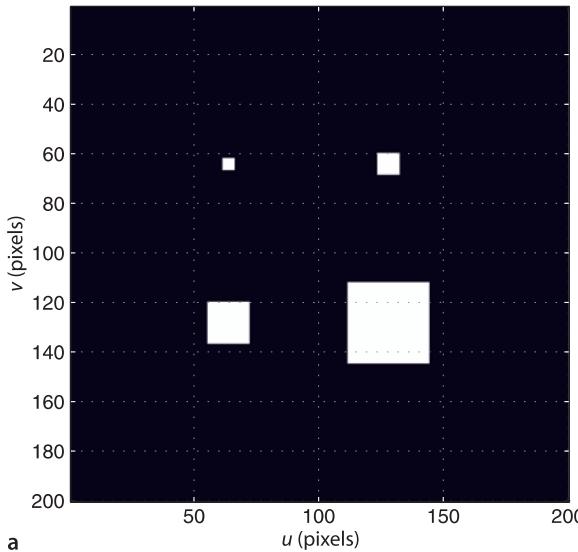
and has a scale of

```
>> s(5)
ans =
4.0311
```

Figures 13.23b–e show the Laplacian of Gaussian at four different points in the scale-space sequence.

Figure 13.23f shows the magnitude of the Laplacian of Gaussian response as a function of scale, taken at the points corresponding to the centre of each square in the input image. Each curve has a well defined peak, and the scale associated with the peak is proportional to the size of the region. This is the characteristic scale of the region.

We actually compute the difference of Gaussian approximation to the Laplacian of Gaussian, as illustrated in Fig. 13.26.



**Fig. 13.23.**

Scale-space example. **a** Synthetic image  $I$  with blocks of sizes  $5 \times 5$ ,  $9 \times 9$ ,  $17 \times 17$ , and  $33 \times 33$ ; **b–e** Normalized Laplacian of Gaussian  $\sigma^2 L \otimes G(\sigma) \otimes I$  for increasing values of scale,  $\sigma$  value indicated in lower left. False color is used: red is negative and blue is positive; **f** magnitude of Laplacian of Gaussian at centre of each square (indicated by '+') versus  $\sigma$

If we consider the 3-dimensional image  $L$  as a volume then a scale-space feature point is any pixel that is a 3D-maxima. That is, an element that is greater than its neighbours in *all three* dimensions – its spatial neighbours at the current scale and at the scale above and below. Such points are detected by the function `iscalemax`

```
>> f = iscalemax(L, s)
f =
(64,64), scale=2.91548, strength=1.96449
(128,64), scale=4.06202, strength=1.72512
(128,128), scale=18.1246, strength=1.54391
(64,128), scale=8.97218, strength=1.54057
(96,128), scale=15.5081, strength=0.345028
(97,128), scale=14.7139, strength=0.34459
```

which returns an array of `ScalePointFeature` objects which are a subclass of `PointFeature`. Each object has properties for the feature's coordinate, strength and scale. The features are arranged in order of decreasing strength and we see that four have significant strength and correspond to the four white objects. We can superimpose the detected features on the original image

```
>> idisp(im)
>> f(1:4).plot('g*')
```

and the result is shown in Fig. 13.24.

The scale associated with a feature can be easily visualized using circles of radius equal to the feature scale

```
>> f(1:4).plot_scale('r')
```

and the result is also shown in Fig. 13.24. We see that the identified features are located at the centre of each object and that the scale of the feature is related to the size of the object. The region within the circle is known as the support region of the feature.

For a real image

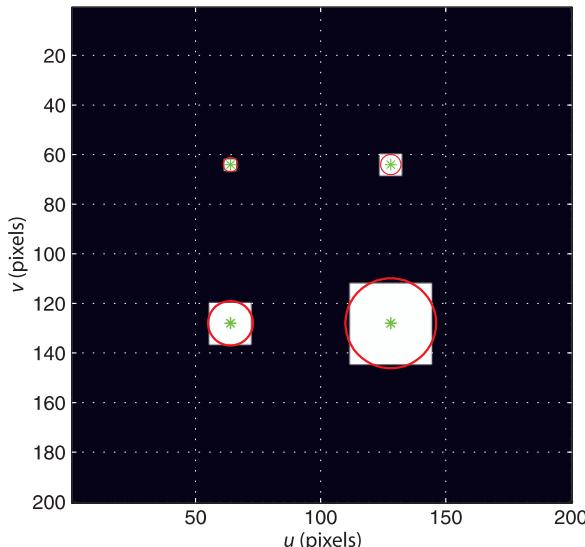
```
>> im = iread('lena.pgm', 'double');
```

we compute the scale-space in eight large steps with  $\sigma = 8$

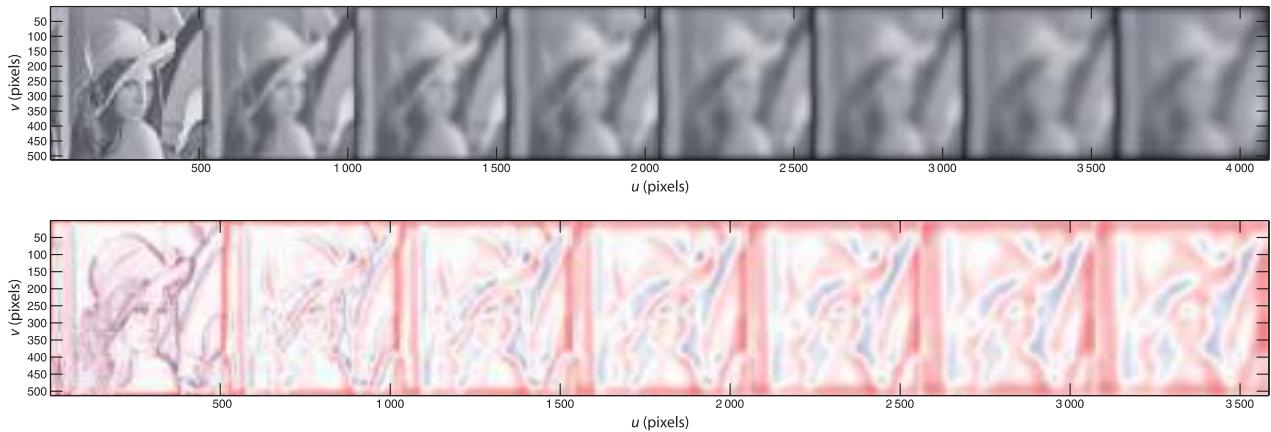
```
>> [G,L] = iscalespace(im, 8, 8);
```

which we can *flatten* and display

```
>> idisp(G, 'flatten', 'wide', 'square');
>> idisp(L, 'flatten', 'wide', 'square', 'invsigned');
```

**Fig. 13.24.**

Synthetic image with overlaid feature centre and scale indicator



as shown in Fig. 13.25. From left to right we see the eight levels of scale. The Gaussian sequence of images becomes increasing blurry. In the Laplacian of Gaussian sequence the dark eyes are strongly positive (blue) blobs at low scale and the light colored hat becomes a strongly negative (red) blob at high scale.

Convolving the original image with a Gaussian kernel of increasing  $\sigma$  results in the kernel size, and therefore the amount of computation, growing at each scale step. Recalling the properties of a Gaussian from page 302, a Gaussian convolved with a Gaussian is another wider Gaussian. Instead of convolving our original image with ever wider Gaussians, we can repeatedly apply the same Gaussian to the previous result. We also recall from page 310 that the LoG kernel is approximated by the difference of two Gaussians. Using the properties of convolution we can write

$$(G(\sigma_1) - G(\sigma_2)) \otimes I = G(\sigma_1) \otimes I - G(\sigma_2) \otimes I$$

where  $\sigma_1 > \sigma_2$ . The difference of Gaussian operator applied to the image is equivalent to the difference of the image at two different levels of smoothing. If we perform the smoothing by successive application of a Gaussian we have a sequence of images at increased levels of smoothing. The difference between successive steps in the sequence is therefore an approximation to the Laplacian of Gaussian. Figure 13.26 shows this in diagrammatic form.

### 13.3.2.1 Scale-Space Point Feature

The scale-space concepts just discussed underpin a number of popular feature detectors which find salient points within an image and determines their scale and also their orientation. The Scale-Invariant Feature Transform (SIFT) is based on the maxima in a difference of Gaussian sequence. The Speeded Up Robust Feature (SURF) is based on the maxima in an approximate Hessian of Gaussian sequence.

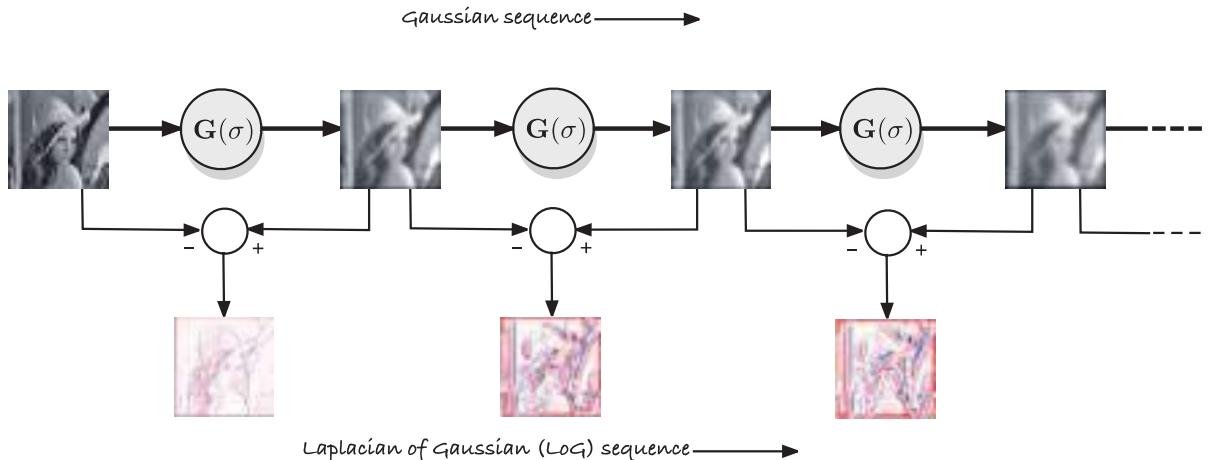
To illustrate we will compute the SURF features for the building image used previously

```
>> surf1 = isurf(bl)
surf1 =
4034 features (listing suppressed)
Properties: theta image_id scale u v strength descriptor
```

which returns an array of `SurfPointFeature` objects which are a subclass of `ScalePointFeature`. For example the first feature is

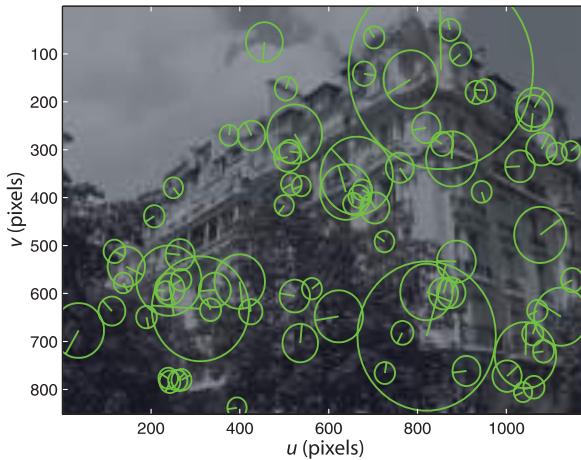
```
>> surf1(1)
ans = (117.587,511.978), theta=0.453513, scale=2.16257,
strength=0.0244179, descrip= ..
```

**Fig. 13.25.** Scale-space sequence for  $\sigma = 2$ , (top) Gaussian sequence, (bottom) Laplacian of Gaussian sequence



**Fig. 13.26.** Schematic for calculation of Gaussian and Laplacian of Gaussian scale-space sequence

**Fig. 13.27.**  
SURF descriptors showing the support region (scale) and orientation as a radial line



Each object includes the feature's coordinate (estimated to sub-pixel precision), scale, orientation, and a descriptor which is a 64-element vector. Orientation is defined by the dominant edge direction within the support region.

This image contains over 4 000 SURF features but we can show the first, and strongest, fifty features

```
>> idisp(b1, 'dark');
>> surf1(1:50:end).plot_scale('g', 'clock')
```

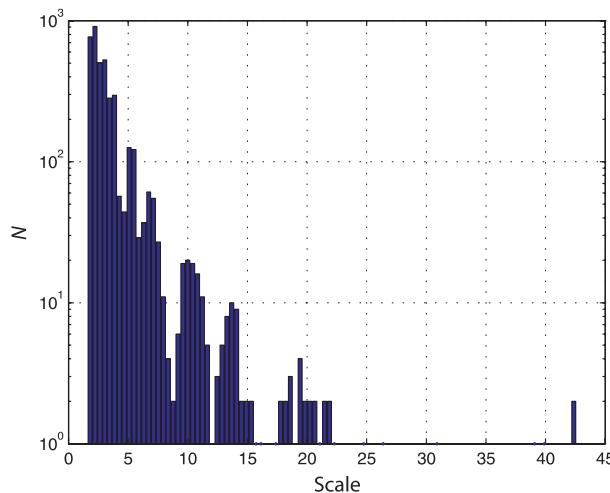
and the result is shown in Fig. 13.27. The `plot_scale` method draws a circle around the feature's location with a radius that indicates its scale – the size of the support region. The option '`clock`' draws a radial line which indicates the orientation of the SURF feature.

Feature scale varies widely and a histogram

```
>> hist(surf1.scale, 100);
```

shown in Fig. 13.28 indicates that there are many small features associated with fine image detail and texture. The bulk of the features have a scale less than 25 pixel but some have scales over 40 pixel. The `isurf` function accepts a number of options which are described in the online documentation.

The SURF algorithm is more than just a scale-invariant feature detector, it also computes a very robust *descriptor*. The descriptor is a 64-element vector that encodes the image gradient in sub-regions of the support region in a way which is invariant to brightness, scale and rotation. This enables feature descriptors to be unambiguously



**Fig. 13.28.**  
Histogram of feature scales shown with logarithmic vertical scale

matched to a descriptor of the same world point in another image even if their scale and orientation are quite different. The difference in position, scale and orientation of the matched features gives some indication of the relative camera motion between the two views. Matching features between scenes is crucial to the problems that we will address in the next chapter.

### 13.4 Wrapping Up

In this chapter we have discussed the extraction of features from an image. Instead of considering the image as millions of independent pixel values we succinctly describe regions within the image that correspond to distinct objects in the world. For instance we can find regions that are homogeneous with respect to intensity or color and describe them in terms of features such as a bounding box, centroid, equivalent ellipse, aspect ratio, circularity and perimeter shape. Features have invariance properties with respect to translation, rotation about the optical axis and scale which are important for object recognition. Straight lines are common visual features in man-made environments and we showed how to find and describe distinct straight lines in an image using the Hough transform.

We can also show how to find interest points that can reliably *associate* to particular points in the world irrespective of the camera view. These are key to techniques such as camera motion estimation, stereo vision, image retrieval, tracking and mosaicing that we will discuss in the next chapter.

### Further Reading

Region-based image segmentation and blob analysis are classical techniques covered in many books and papers. Gonzalez and Woods (2008) and Szeliski (2011) provide a thorough treatment of the methods introduced in this chapter, in particular thresholding and boundary descriptors. Otsu's algorithm for threshold determination was introduced in Otsu (1975), and the Niblack algorithm for adaptive thresholding was introduced in Niblack (1985). The book by Nixon and Aguado (2008) expands on material covered in this chapter and introduces techniques such as deformable templates and boundary descriptors. The Freeman chain code was first described in Freeman (1974).

In addition to region homogeneity based on intensity and color it is also possible to describe the texture of regions – a spatial pattern of pixel intensities whose statistics

can be described (Gonzalez and Woods 2008). Regions can then be segmented according to texture, for example a smooth road from textured grass.

Clustering of data is an important topic in machine learning (Bishop 2006). In this chapter we have used a simple implementation of k-means which is far from state-of-the-art in clustering, and requires the number of clusters to be known in advance. More advanced clustering algorithms are hierarchical and employ data structures such as kd-trees to speed the search for neighbouring points. The initialization of the cluster centres is also critical to performance. Forsyth and Ponce (2002) introduce more general clustering methods as well as graph-based methods for computer vision. The graph cuts algorithm for segmentation was described by Felzenszwalb and Huttenlocher (2004) and the Toolbox graph-cuts implementation is based on code by Pedro Felzenszwalb and available at <http://people.cs.uchicago.edu/~pff/segment>. The maximally stable extremal region (MSER) algorithm is described by Matas et al. (2004) and the Toolbox implementation is based on the work of Andrea Vedaldi and Brian Fulkerson which is available at <http://vlfeat.org>. The Berkeley Segmentation Dataset at <http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds> contains numerous complex real-world images each with several human-made segmentations.

This chapter has presented a classical *bottom up* approach for feature extraction, starting with pixels and working our way up to higher level concepts such as regions and lines. Forsyth and Ponce (2002) provide a good introduction to high-level vision using probabilistic techniques that can be applied to problems such as object recognition, for example face recognition, and image retrieval.

The Hough transform was first described in U.S. Patent 3,069,654 “Method and Means for Recognizing Complex Patterns” by Paul Hough, and its history is discussed in Hart (2009). The original application was automating the analysis of bubble chamber photographs and it used the problematic slope-intercept parametrization for lines. The currently known form with the  $(\rho, \theta)$  parameterization was first described in Duda and Hart (1972) as a “generalized Hough transform”. This paper is available at <http://www.ai.sri.com/pubs/files/tn036-duda71.pdf>. The Hough transform is covered in textbooks such as Gonzalez and Woods (2008) and Forsyth and Ponce (2002). The latter has a good discussion on shape fitting in general and estimators that are robust with respect to outlier data points. The basic Hough transform has been extended in many ways and there is a large literature. A useful review of the transform and its variants is presented in Leavers (1993). The transform can be generalized to other shapes (Ballard 1981) such as circles of a fixed size where votes are cast for the coordinates of the circle’s centre. For circles of unknown size a three-dimensional voting array is required for the circle’s centre and radius.

The literature on interest operators dates back to the early work of Moravec (1980) and the work of Förstner (Förstner and Gülich 1987; Förstner 1994). The Harris corner detector (Harris and Stephens 1988) became very popular for robotic vision application in the late 1980s since it was able to run in real-time on computers of the day and the features were quite stable (Tissainayagam and Suter 2004) from image to image. The Noble detector is described in Noble (1988). The work of Shi, Tomasi, Lucas and Kanade (Shi and Tomasi 1994; Tomasi and Kanade 1991) led to the Shi-Tomasi detector and the Kanade-Lucas-Tomasi (KLT) tracker. Good surveys of the relative performance of many corner detectors include those by Deriche and Giraudon (1993) and Mikolajczyk and Schmid (2004).

Scale-space concepts have long been known in computer vision. Koenderink (1984), Lindeberg (1993) and ter Haar Romeny (1996) are a readable introduction to the topic. Scale-space was applied to classic corner detectors creating hybrid detectors such as scale-Harris (Mikolajczyk and Schmid 2004). An important development in scale-space feature detectors was the scale-invariant feature transform (SIFT) introduced in the early 2000s by Lowe (2004) and was a significant improvement for applications such as tracking and object recognition. Unusually, and perhaps unfortunately, it was patented and could not be used in this book. Nature abhors a vacuum and an effective alternative called

Speeded Up Robust Features (SURF) was developed (Bay et al. 2008). The Toolbox function `isurf` wraps a MATLAB® implementation by Dirk-Jan Kroon and available at <http://www.mathworks.com/matlabcentral/fileexchange/28300-opensurf-including-image-warp>, which in turn is based on the OpenSurf implementations in C++ and C# by Chris Evans at <http://www.chrisevansdev.com/computer-vision-opensurf.html>. Other implementations are available from <http://www.vision.ee.ethz.ch/~surf> and GPU-based parallel implementations have been developed. The SIFT and SURF detectors do give different results and they are compared in Bauer et al. (2007). If you are interested in trying the SIFT detector you can use the Toolbox function `isift` which provides a similar interface to `isurf` and returns a feature vector of class `SiftPointFeature`. This function is a wrapper for the MATLAB® implementation from <http://www.vlfeat.org> which you will need to download and compile.

Many other feature detectors have been proposed recently. FAST by Rosten et al. (2010) is claimed to have very low computational requirements and high repeatability, and C and MATLAB® software resources are available at <http://mi.eng.cam.ac.uk/~er258/work/fast.html>. CenSurE by Agrawal et al. (Agrawal et al. 2008; Rosten et al. 2010) claims higher performance than SIFT, SURF and FAST at lower cost. BRIEF by Calonder et al. (2010) is not a feature detector but is a low cost and compact feature descriptor, just 256 bits instead of 64 floating point numbers per feature.

---

## Exercises

1. Greyscale classification
  - a) Experiment with `ithresh` on the images `castle_sign.png` and `castle_sign2.png`.
  - b) Experiment with the Niblack algorithm and vary the value of  $k$  and window size.
  - c) Apply `iblobs` to the output of the MSER segmentation. Develop an algorithm that uses the width and height of the bounding boxes to extract just those blobs that are letters.
  - d) The function `imser` has many parameters: `'Delta'`, `'MinDiversity'`, `'MaxVariation'`, `'MinArea'`, `'MaxArea'`. Explore the effect of adjusting these.
  - e) Apply the function `igraphcut` to the `castle_sign2.png` image. Understand and adjust the parameters to improve performance.
2. Color classification
  - a) Change  $k$ , the number of clusters, in the color classification examples. Is there a best value?
  - b) k-means with `'random'` or `'spread'` options performs a randomized initialization. Run k-means several times and determine how different the final clusters are.
  - c) Write a function that determines which of the clusters represents the targets, that is, the yellow cluster or the red cluster.
  - d) Apply the function `igraphcut` to the targets and garden image. How does it perform? Understand and adjust the parameters to improve performance.
  - e) Experiment with the parameters of the morphological “cleanup” used for the targets and garden images.
  - f) Write code that loops over images captured from your computer’s camera, applies a classification, and shows the result. The classification could be a greyscale threshold or color clustering to a pre-learnt set of color clusters (see `colorKmeans`).
3. Blobs. Create an image of an object with several holes in it. You could draw it and take a picture, export it from a drawing program, or write code to generate it.
  - a) Determine the outer, *inner* and total boundaries of the object.

- b) Place small objects within the holes in the objects. Write code to display the topological hierarchy of the blobs in the scene.
  - c) For the same shape at different scales plot how the circularity changes as a function of scale. Explain the shape of this curve?
  - d) Create a square object and plot the estimated and true perimeter as a function of the square's side length. What happens when the square is small?
  - e) Create an image of a simple scene with a number of different shaped objects. Using the shape invariant features (aspect ratio, circularity) to create a simple shape classifier. How well does it perform? Repeat using the Hu moment features.
  - f) Repeat the boundary matching example with some objects that you create. Modify the code to create a plot of edge-segment angle ( $k$ ) versus  $\theta$  and repeat the boundary matching example.
  - g) Another commonly used feature, not supported by the Toolbox, is the aligned rectangle. This is the smallest rectangle whose sides are aligned with the axes of the equivalent ellipse and which entirely contains the blob. The aspect ratio of this rectangle and the ratio of the blob's area to the rectangle's area are each scale and rotation invariant features. Write code to compute this rectangle, overlay the rectangle on the image, and compute the two features.
4. Hough transform
    - a) Experiment with varying the size of the Hough accumulator.
    - b) Experiment with using the Sobel edge operator instead of Canny.
    - c) Experiment with varying the parameters '`supress`', '`interpSize`', '`EdgeThresh`', '`houghThresh`'.
    - d) Apply the Hough transform to one of your own images.
    - e) Write code that loops over capturing images from your computer's camera, finding the two dominant lines and overlaying them on a window showing the image.
  5. Corner detectors
    - a) Experiment with the Harris detector by changing the parameters  $k$ ,  $\sigma_D$  and  $\sigma_I$ .
    - b) Compare the performance of the Harris, Noble and Shi-Tomasi corner detectors.
    - c) Implement the Moravec detector and compare to Harris detector.
    - d) Create a smoothed second derivative  $I_{uu}$ ,  $I_{vv}$  and  $I_{uv}$ .

# 14

# Using Multiple Images



Almost! We can determine the translation of the camera only up to an unknown scale factor, that is, the translation is  $\lambda \mathbf{t}$  where the direction of  $\mathbf{t}$  is known but  $\lambda$  is not.

In the previous chapter we learnt about corner detectors which find particularly distinctive *points* in a scene. These points can be reliably detected in different views of the same scene irrespective of viewpoint or lighting conditions. Such points are characterised by high image gradients in orthogonal directions and typically occur on the corners of objects. However the 3-dimensional coordinate of the corresponding world point was lost in the perspective projection process which we discussed in Chap. 11 – we mapped a 3-dimensional world point to a 2-dimensional image coordinate. All we know is that the world point lies along some ray in space corresponding to the pixel coordinate, as shown in Fig. 11.1. To recover the missing third dimension we need additional information. In Sect. 11.2.3 the additional information was camera calibration parameters plus a geometric object model, and this allowed us to estimate the object's 3-dimensional pose from the 2-dimensional image data.

In this chapter we consider an alternative approach in which the additional information comes from *multiple* views of the same scene. As already mentioned the pixel coordinates from a single view constrain the world point to lie along some ray. If we can locate the same world point in another image, taken from a different but known pose, we can determine another ray along which that world point must lie. The world point lies at the intersection of these two rays – a process known as triangulation or 3D reconstruction. Even more powerfully, if we observe sufficient points, we can estimate the 3D motion of the camera between the views as well as the 3D structure of the world. ▶

The underlying challenge is to find the same world point in multiple images. This is the *correspondence problem*, an important but non-trivial problem that we will discuss in Sect. 14.1. In Sect. 14.2 we revisit the fundamental geometry of image formation developed in Chap. 11. If you haven't yet read that chapter, or it's been a while since you read it, it would be helpful to (re)acquaint yourself with that material. We extend the geometry of single-camera imaging to the situation of multiple image planes and show the geometric relationship between pairs of images. Stereo vision is an important technique for robotic 3-dimensional perception and is discussed in some detail in Sect. 14.3. Information from two images of a scene, taken from different viewpoints, is combined to determine the 3-dimensional structure of the world. Section 14.4 introduces the topic of structure from motion where visual information from a sequence of images is used to determine the 3-dimensional structure of the world as well how the robot has moved through the world. The latter is known as visual odometry.

We finish this chapter, and this part of the book, with four application examples based on the concepts we have learned. Section 14.5 describes how we can transform an image with obvious perspective distortion into one without, effectively synthesizing the view from a virtual camera at a different location. Section 14.6 describes mosaicing which is the process of taking consecutive images from a moving camera

and *stitching* them together to form one large virtual image. Section 14.7 describes image retrieval which is the problem of finding which image in an existing set of images is most similar to some new image. This can be used by robot to determine whether it has visited a particular place, or seen the same object, before. Section 14.8 describes how we can process a sequence of images from a moving camera to locate consistent world points and to estimate the camera motion and 3-dimensional world structure.

## 14.1 Feature Correspondence

Correspondence is the problem of finding the pixel coordinates in two different images that correspond to the same point in the world.<sup>►</sup> Consider the pair of real images

```
>> im1 = imread('eiffel2-1.jpg', 'mono', 'double');
>> im2 = imread('eiffel2-2.jpg', 'mono', 'double');
```

shown in Fig. 14.1. They show the same scene viewed from two different positions using two different cameras – the pixel size, focal length and number of pixels for each image are quite different. The scenes are complex and we see immediately that determining correspondence is not trivial. More than half the pixels in each scene correspond to blue sky and it is impossible to match a blue pixel in one image to the corresponding blue pixel in the other – these pixels are insufficiently distinct. This situation is common and can occur with homogeneous image regions such as dark shadows, smooth sheets of water, snow or smooth man-made objects such as walls or the sides of cars.

The solution is to choose only those points that are distinctive. We can use the interest point detectors that we introduced in the last chapter to find Harris corner features

```
>> harris = icorner(im1, 'nfeat', 200);
>> idisp(im1); harris.plot('gs');
```

or SURF features<sup>►</sup>

```
>> sf = isurf(im1, 'nfeat', 200);
>> idisp(im1); sf.plot_scale('g');
```

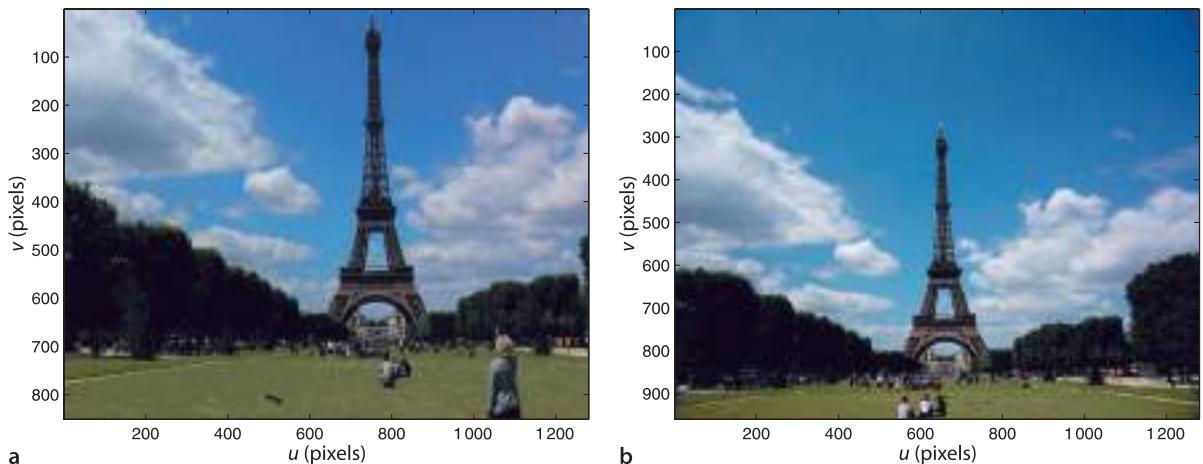
and these are shown in Fig. 14.2. We have simplified the problem – instead of millions of pixels we have just 200 distinctive points.

Consider the general case of two sets of features points:  $\{^1\mathbf{p}_i, i = 1 \dots N_1\}$  in the first image and  $\{^2\mathbf{p}_j, j = 1 \dots N_2\}$  in the second image. Since these are distinctive image points we would expect a significant number of points in image one would correspond to points found in image two. The problem is to determine which  $(^2u_j, ^2v_j)$ , if any, corresponds to each  $(^1u_i, ^1v_i)$ .

This is another example of the data association problem.

The SURF detector cannot process a color image, it converts it to greyscale. The Harris detector computes the squared gradients for the individual color planes separately and then combines them. All detectors can process an image sequence provided as a matrix of dimension greater than two. There is ambiguity between a color image and an image sequence of length three. If the image's third dimension is three it is deemed to be a color image, not a sequence. A four-dimensional image is unambiguous as a sequence of color images.

**Fig. 14.1.** Two views of the Eiffel tower. The images were captured approximately simultaneously using two different handheld digital cameras. **a** 7 Mpix camera with  $f=7.4$  mm; **b** 10 Mpix camera with  $f=5.2$  mm (photo by Lucy Corke). The images have quite different scale and the tower is 700 and 600 pixel tall in **a** and **b** respectively. The camera that captured image **b** is held by the person in the bottom-right corner of **a**



We cannot use the feature coordinates to determine correspondence – the features will have different coordinates in each image. For example in Fig. 14.1 we see that most features are lower in the right-hand image. We cannot use the intensity or color of the pixels either. Variations in white balance, illumination and exposure setting make it very unlikely that pixels that should correspond will have the same value. Even if intensity variation was eliminated there are likely to be tens of thousands of pixels in the other image with exactly the same intensity value – it is not sufficiently unique. We need some richer way of *describing* each feature.

In practice we describe the region of pixels *around* the corner point which provides a distinctive and unique description of the corner point and its immediate surrounds – the feature descriptor. In the Toolbox the feature descriptor for a corner point is a vector – the `descriptor` property of the `PointFeature` superclass. For the Harris corner feature the descriptor

```
>> harris(1).descriptor'
ans =
    0.0805    0.0821    0.0371
```

is a 3-vector that contains the unique elements of the structure tensor Eq. 13.14. This low-dimensional descriptor is computationally cheap since the elements were already computed in order to determine corner strength. These descriptor elements are gradients which have the advantage of being robust to offsets in image intensity. The similarity of two descriptors is based on Euclidean distance and is zero for a perfect match. For example, the similarity of corner features one and two is

```
>> harris(1).distance( harris(2) )
ans =
    0.0518
```

However it is difficult to know whether this value represents strong similarity or not since the units are somewhat arbitrary. Typically we would compare feature  ${}^1p_i$  with all features in the other image  $\{{}^2p_j, j = 1 \dots N_2\}$  and choose the one that is most similar. However a short descriptor vector like this is still insufficiently distinctive and prone to incorrect matching.

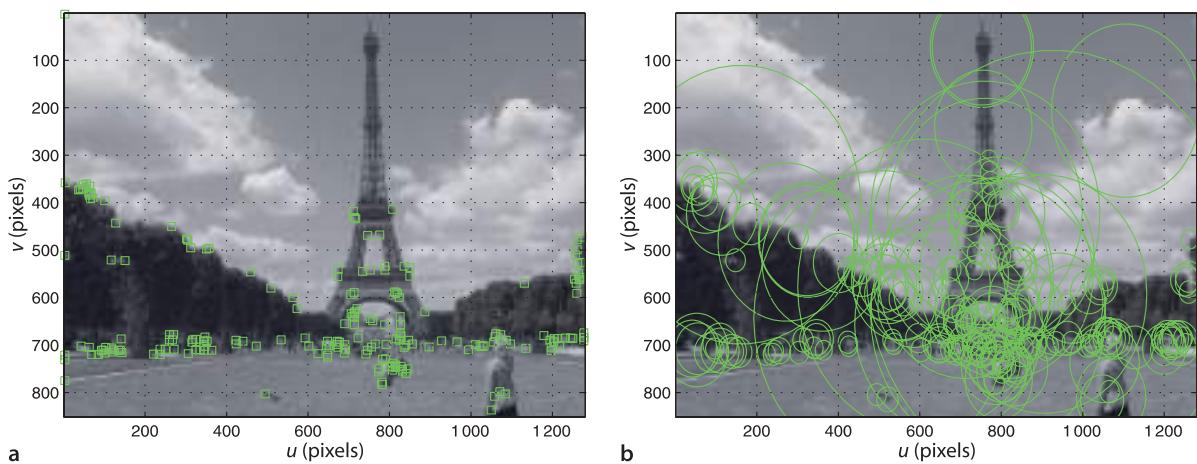
We can create a large descriptor vector by representing the square window around the feature point as a vector. For example

```
>> harris = icorner(im1, 'nfeat', 200, 'color', 'patch', 5)
```

creates a 121-element descriptor vector for each corner point from the window of specified half-width around the feature point – in this case an  $11 \times 11$  window. The pixel values are offset by the mean value and normalized to create a unit vector. We can rewrite the ZNCC similarity measure from Table 12.1 in 1-dimensional form as

If the world point is not visible in image two then the most similar feature will be an incorrect match.

**Fig. 14.2.** Corner features computed for Fig. 14.1a. **a** Harris corner features; **b** SURF corner features showing scale



$$\begin{aligned}
 s &= \frac{\sum_{i=1}^N (I_1[i] - \bar{I}_1) \cdot (I_2[i] - \bar{I}_2)}{\sqrt{\sum_{i=1}^N (I_1[i] - \bar{I}_1)^2} \cdot \sqrt{\sum_{i=1}^N (I_2[i] - \bar{I}_2)^2}} \\
 &= \underbrace{\frac{\sum_{i=1}^N (I_1[i] - \bar{I}_1)}{\sqrt{\sum_{i=1}^N (I_1[i] - \bar{I}_1)^2}}}_{d_1} \cdot \underbrace{\frac{\sum_{i=1}^N (I_2[i] - \bar{I}_2)}{\sqrt{\sum_{i=1}^N (I_2[i] - \bar{I}_2)^2}}}_{d_2}
 \end{aligned} \tag{14.1}$$

which we have factored into the dot product of a unit vector associated with each image patch. This normalized vector  $d_i$  can be used as the feature descriptor. Normalized cross-correlation is simply the dot product of two descriptors and the resulting similarity measure  $s \in [-1, 1]$  has some meaning. A perfect match is  $s = 1$  and  $s \geq 0.8$  is typically considered a good match. For the example above

```
>> harris(1).ncc( harris(2) )
ans =
-0.0292
```

the correlation score indicates a poor match. This descriptor is distinctive and invariant to changes in image intensity but is not invariant to scale or rotation. Other descriptors of the surrounding region that we could use include census and rank values as well as histograms of intensity or color. Histograms have the advantage of being invariant to rotation but they say nothing about the spatial relationship between the pixels, that is, the same pixel values in a completely different arrangement have the same histogram.

The SURF algorithm computes a 64-element descriptor► vector to describe the feature point in way that is scale and rotationally invariant and based on the pixels within the feature's support region. It is created from the image in the scale-space sequence corresponding to the feature's scale and rotated according to the feature's orientation. The vector is normalized to a unit vector to increase its invariance to changes in image intensity. Similarity between descriptors is based on Euclidean distance. This descriptor is quite invariant to image intensity, scale and rotation. SURF is both a corner detector and a descriptor, whereas the Harris operator is just a corner detector which must be used with one of a number of different descriptors.►

A 128-element vector can be created by passing the option '`extended`' to `isurf`.

For the remainder of this chapter we will use SURF features. They are computationally more expensive but pay for themselves in terms of the quality of matches between widely different views of the same scene. We compute SURF features for each image

```
>> s1 = isurf(im1)
s1 =
1288 features (listing suppressed)
Properties: theta image_id scale u v strength descriptor
>> s2 = isurf(im2)
s2 =
1426 features (listing suppressed)
Properties: theta image_id scale u v strength descriptor
```

which results in two vectors of `SurfPointFeature` objects. Many thousands of corner features were found in each image.

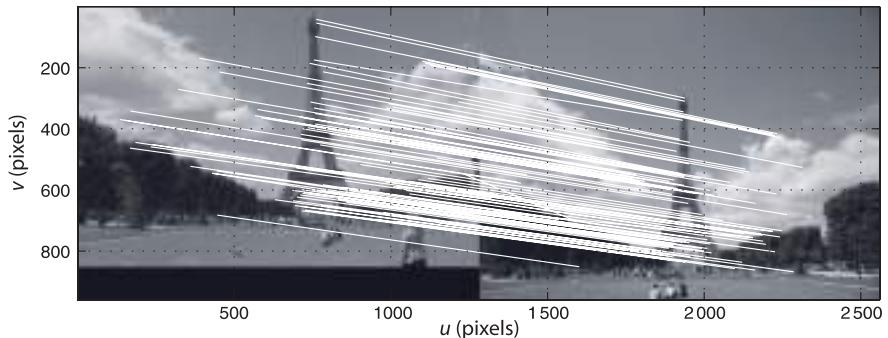
Next we match the two sets of SURF features based on the distance between the SURF descriptors

```
>> m = s1.match(s2)
m =
644 corresponding points (listing suppressed)
```

which results in a vector of `FeatureMatch` objects that represents 644 candidate-corresponding points. The first five candidate correspondences► are

It is conceivable to use the SURF descriptor with a Harris corner point.

We refer to them as candidates because although they are very likely to correspond this has not yet been confirmed.

**Fig. 14.3.**

Feature matching. Subset (100 out of 1664) of matches based on SURF descriptor similarity. We note that a few are clearly incorrect

```
>> m(1:5)
ans =
(819.56, 358.557) <-> (708.008, 563.342), dist=0.002137
(1028.3, 231.748) <-> (880.14, 461.094), dist=0.004057
(1027.6, 571.118) <-> (885.147, 742.088), dist=0.004297
(927.724, 509.93) <-> (800.833, 692.564), dist=0.004371
(854.35, 401.633) <-> (737.504, 602.187), dist=0.004417
```

which shows the feature coordinate in the first and second image, as well as the Euclidean distance between the two feature vectors. The matches are ordered by decreasing similarity, and a threshold on feature similarity has been applied.

We can overlay a subset of these matches on the original image pair

```
>> idisp({im1, im2})
>> m.subset(100).plot('w')
```

and the result is shown in Fig. 14.3. White lines connect the matched features in each image and the lines show a consistent pattern. Most of these connections seem quite sensible, but a few are quite obviously incorrect. Note that we passed a cell-array of images to `idisp` which it displays horizontally tiled as a single image. The `subset` method of the `FeatureMatch` class returns a vector with the specified number of `FeatureMatch` objects sampled evenly from the original vector. If all correspondences were shown we would just see a solid white mass.

The correspondences can be obtained via an optional return value

```
>> [m,corresp] = s1.match(s2);
>> corresp(:,1:5)
ans =
    215         389         357        1044        853
    246         418         312        1240        765
```

which is a matrix with one column per correspondence. The first column indicates that feature 215 in image one matches feature 246 in image two and so on. In terms of workspace variables this is `s1(215)` and `s2(246)`.

The Euclidean distance between the matched feature descriptors is given by the `distance` property and the distribution of these, with no thresholding applied, is

```
>> m2 = s1.match(s2, 'thresh', []);
>> ihist(m2.distance, 'normcdf')
```

shown in Fig. 14.4. It shows that 35% of all matches have descriptor distances below 0.05 which is the default threshold, whereas the maximum distance can be over ten times larger – such matches are less likely to be valid. Instead of a fixed threshold we could choose to take the matches with the  $N$  smallest distances

```
>> m = s1.match(s2, 'top', N);
```

or all those below a distance threshold

```
>> m = s1.match(s2, 'thresh', 0.04);
```

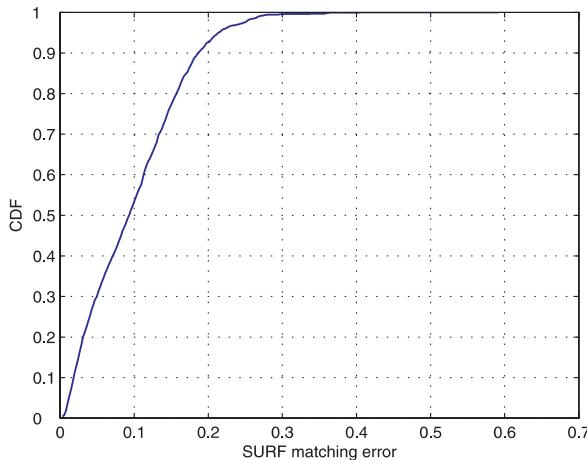


Fig. 14.4.

Cumulative distribution of feature distance

or all those below the median

```
>> m = s1.match(s2, 'median');
```

Feature matching is computationally expensive – it is an  $O(N^2)$  problem since every feature in one image must be compared with every feature in the other image.

Although the quality of matching shown in Fig. 14.3 looks quite good there are a few obviously incorrect matches in this small subset. We can discern a pattern in the lines joining the corresponding points, they are slightly converging and sloping down to the left. This pattern is a function of the relative pose between the two camera views, and understanding this is key to determining which of the candidate matches are correct. That is the topic of the next section.

## 14.2 Geometry of Multiple Views

We start by studying the geometric relationships between images of a single point  $P$  observed from two different viewpoints and this is shown in Fig. 14.5. This geometry could represent the case of two cameras simultaneously viewing the same scene, or one camera taking a picture from two different viewpoints.► The centre of each camera, the origins of {1} and {2}, plus the world point  $P$  defines a plane in space – the epipolar plane. The world point  $P$  is projected onto the image planes of the two cameras at pixel coordinates  ${}^1p$  and  ${}^2p$  respectively, and these points are known as conjugate points.

Assuming the point does not move.

Consider image one. The image point  ${}^1e$  is a function of the position of camera two. The image point  ${}^1p$  is a function of the world point  $P$ . The camera centre,  ${}^1e$  and  ${}^1p$  define the epipolar plane and hence the epipolar line  ${}^2\ell$  in image two. By definition the conjugate point  ${}^2p$  must lie on that line. Conversely  ${}^1p$  must lie along the epipolar line in image one  ${}^1\ell$  that is defined by  ${}^2p$  in image two.

This is a very fundamental and important geometric relationship – given a point in one image we know that its conjugate is constrained to lie along a line in the other image. We illustrate this with a simple example that mimics the geometry of Fig. 14.5

```
>> T1 = transl(-0.1, 0, 0) * trotz(0.4);
>> cam1 = CentralCamera('name', 'camera 1', 'default', ...
    'focal', 0.002, 'pose', T1)
```

which returns an instance of the `CentralCamera` class as discussed previously in Sect. 11.1. Similarly for the second camera

```
>> T2 = transl(0.1, 0, 0)*trotz(-0.4);
>> cam2 = CentralCamera('name', 'camera 2', 'default', ...
    'focal', 0.002, 'pose', T2);
```

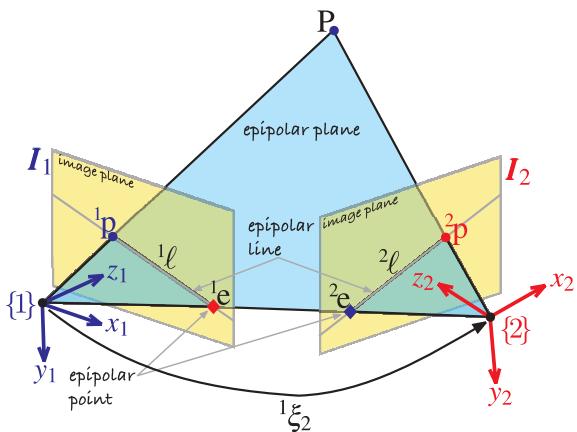


Fig. 14.5.

Epipolar geometry showing the two cameras with associated coordinate frames  $\{1\}$  and  $\{2\}$  and image planes. The world point  $P$  and the two camera centres form the epipolar plane, and the intersection of this plane with the image-planes form epipolar lines

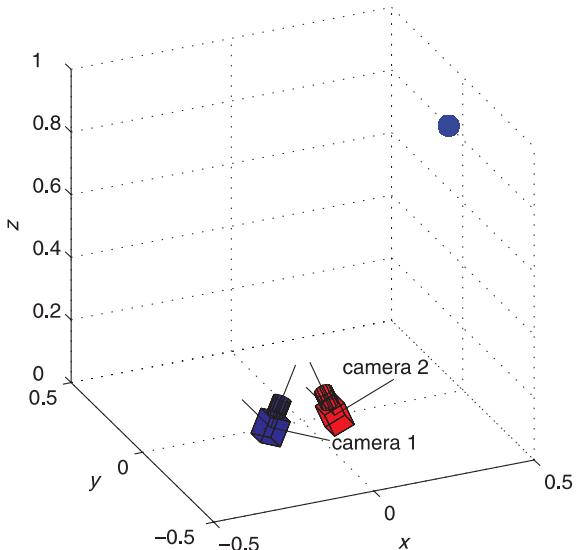


Fig. 14.6.

Simulation of two cameras and a target point. The origins of the two cameras are offset along the  $x$ -axis and the cameras are *verged*, that is, their optical axes intersect

and the pose of the two cameras is visualized by

```
>> axis([-0.5 0.5 -0.5 0.5 0 1])
>> cam1.plot_camera('color', 'b', 'label')
>> cam2.plot_camera('color', 'r', 'label')
```

which is also shown in Fig. 14.6. We define an arbitrary world point

```
>> P=[0.5 0.1 0.8]';
```

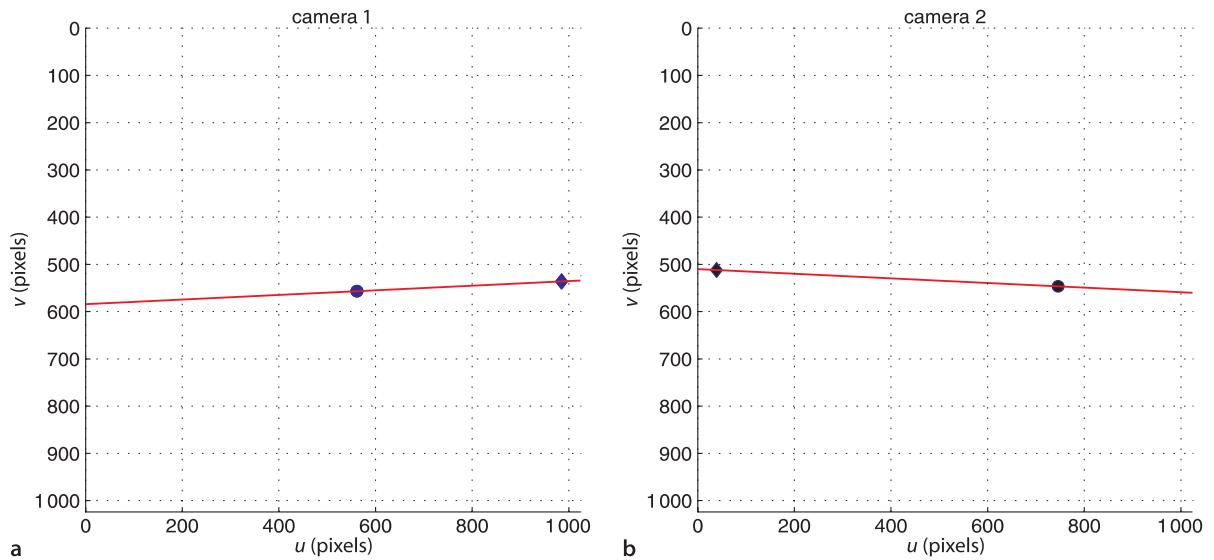
which we display as a small sphere

```
>> plot_sphere(P, 0.03, 'b');
```

which is shown in Fig. 14.6. We project this point to both cameras

```
>> p1 = cam1.plot(P)
p1 =
    561.6861
    532.6079
>> p2 = cam2.plot(P)
p2 =
    746.0323
    546.4186
```

and this is shown in Fig. 14.7. The epipoles are computed by projecting the centre of each camera to the other camera's image plane



```

>> cam1.hold
>> e1 = cam1.plot( cam2.centre, 'Marker', 'd', 'MarkerFaceColor', 'k')
e1 =
    985.0445  512.0000
>> cam2.hold
>> e2 = cam2.plot( cam1.centre, 'Marker', 'd', 'MarkerFaceColor', 'k')
e2 =
    38.9555  512.0000

```

and these are shown in Fig. 14.7 as a black ♦-marker.

### 14.2.1 The Fundamental Matrix

The epipolar relationship shown graphically in Fig. 14.5 can be expressed concisely and elegantly as

$${}^2\tilde{p}^T F {}^1\tilde{p} = 0 \quad (14.2)$$

where  ${}^1\tilde{p}$  and  ${}^2\tilde{p}$  are the image points  ${}^1p$  and  ${}^2p$  expressed in homogeneous form and  $F$  is a  $3 \times 3$  matrix known as the fundamental matrix.

We can group the last two terms

$${}^2\tilde{\ell} \simeq F {}^1\tilde{p} \quad (14.3)$$

which is the equation of a line, the epipolar line, along which conjugate point in image two must lie

**2D projective geometry in brief.** The projective plane  $\mathbb{P}^2$  is the set of all points  $(x_1, x_2, x_3)$ ,  $x_i \in \mathbb{R}$  and  $x_i$  not all zero. Typically the 3-tuple is considered a column vector. A point  $p = (u, v)$  is represented in  $\mathbb{P}^2$  by homogeneous coordinates  $\tilde{p} = (u, v, 1)$ . Scale is unimportant for homogeneous quantities and we express this as  $\tilde{p} \simeq \lambda \tilde{p}$  where the operator  $\simeq$  means equal up to a (possibly unknown) non-zero scale factor. A point in  $\mathbb{P}^2$  can be represented in non-homogeneous, or Euclidean, form  $p = (x_1/x_3, x_2/x_3)$  in  $\mathbb{R}^2$ . The homogeneous vector  $(u, v, f)$ , where  $f$  is the focal length in pixels, is a vector from the camera's origin that points toward the world point  $P$ . More details are given in Appendix I.

The Toolbox functions `e2h` and `h2e` convert between Euclidean and homogeneous coordinates for points (a column vector) or sets of points (a matrix with one column per point).

**Fig. 14.7.** Epipolar geometry simulation showing the virtual image planes of two Toolbox `CentralCamera` objects. The perspective projection of point  $P$  is a black circle, the projection of the other camera's centre is a black ♦-marker, and the epipolar line is shown in red

$${}^2\tilde{\mathbf{p}}^T {}^2\tilde{\ell} = 0 \quad (14.4)$$

This line is a function (Eq. 14.3) of the point  ${}^1\tilde{\mathbf{p}}$  in image one and is a powerful test as to whether or not a point in image two is a possible conjugate. Taking the transpose of both sides of Eq. 14.2 yields

$${}^1\tilde{\mathbf{p}}^T \mathbf{F}^T {}^2\tilde{\mathbf{p}} = 0 \quad (14.5)$$

from which we can write the epipolar line for camera one

$${}^1\tilde{\ell} = \mathbf{F}^T {}^2\tilde{\mathbf{p}} \quad (14.6)$$

in terms of a point viewed by camera two.

The fundamental matrix is a function of the camera parameters and the relative camera pose between the views

$$\mathbf{F} \approx \mathbf{K}^{-1} \mathbf{S}(t) \mathbf{R} \mathbf{K} \quad (14.7)$$

Note well that this is the inverse of what you might expect, camera two with respect to camera one.

where  $\mathbf{K}$  is the camera intrinsic matrix given in Eq. 11.7,  $\mathbf{S}(\cdot)$  is the skew-symmetric matrix, and  ${}^2\xi_1 \sim (\mathbf{R}, \mathbf{t})$  is the relative pose of camera one with respect to camera two. The fundamental matrix that relates the two views is returned by the method `F` of the `CentralCamera` class, for example

```
>> F = cam1.F( cam2 )
F =
    0    -0.0000    0.0010
   -0.0000     0    0.0019
    0.0010    0.0001   -1.0208
```

where the relative pose is from `CentralCamera` object `cam1` to `cam2`.

The fundamental matrix has some interesting properties. It is singular with a rank of two

```
>> rank(F)
ans =
    2
```

and has seven degrees of freedom. The epipoles are *encoded* in the null-space of the matrix. The epipole for camera one is the right null-space of  $\mathbf{F}$

```
>> null(F)'
ans =
-0.8873   -0.4612   -0.0009
```

in homogeneous coordinates or

```
>> e1 = h2e(ans)'
e1 =
  985.0445   512.0000
```

in Euclidean coordinates – the same as determined above using the `plot` function. The epipole for camera two is the left null-space of the transpose of the fundamental matrix

```
>> null(F)';
>> e2 = h2e(ans)'
e2 =
  38.9555   512.0000
```

The Toolbox can display epipolar lines using the `plot_epiline` methods of the `CentralCamera` class

```
>> cam2.plot_epiline(F, p1, 'r')
```

which is shown in Fig. 14.7 as a red line in the camera two image plane. We see, as expected, that the projection of  $\mathbf{P}$  lies on this epipolar line. The epipolar line for camera one is

```
>> cam1.plot_epiline(F', p2, 'r');
```

### 14.2.2 The Essential Matrix

The epipolar geometric constraint can also be expressed in terms of normalized image coordinates

$${}^2\tilde{x}^T E {}^1\tilde{x} = 0 \quad (14.8)$$

where  $E$  is the essential matrix and  ${}^1\tilde{x}$  and  ${}^2\tilde{x}$  are conjugate points in homogeneous normalized image coordinates.► This matrix is a simple function of the relative camera pose

$$E \approx S(t)R \quad (14.9)$$

where  ${}^2\xi_1 \sim (R, t)$  is the relative pose of camera one with respect to camera two. The essential matrix is singular, has a rank of two, and has two equal singular values► and one of zero. The essential matrix has only 5 degrees of freedom and is completely defined by 3 rotational and 2 translational► parameters. For pure rotation, when  $t = 0$ , the essential matrix is not defined.

We recall from Eq. 11.7 that  $\tilde{p} \simeq K\tilde{x}$  and substituting into Eq. 14.8 we write

$${}^2\tilde{p}^T \underbrace{K_2^{-T} E K_1^{-1}}_F {}^1\tilde{p} = 0 \quad (14.10)$$

Similarity to Eq. 14.2 yields a relationship between the two matrices

$$E \approx K_2^T F K_1 \quad (14.11)$$

in terms of the intrinsic parameters of the two cameras involved.► This is implemented by the `E` method of the `CentralCamera` class

```
>> E = cam1.E(F)
E =
    0   -0.0779      0
-0.0779      0   0.1842
    0   -0.1842   0.0000
```

where the intrinsic parameters of camera one (which is the same as camera two) are used.

Like the camera matrix in Sect. 11.2.2 the essential matrix can be *decomposed* to yield the relative pose  ${}^1\xi_2$  in homogeneous transformation form.► The inverse is not unique and in general there are two solutions

```
>> sol = cam1.invE(E)
sol(:,:,1) =
    1.0000    0.0000   -0.0000   -0.1842
    0.0000   -1.0000   -0.0000   -0.0000
   -0.0000    0.0000   -1.0000   -0.0779
    0         0         0     1.0000
sol(:,:,2) =
    0.6967    0.0000   -0.7174    0.1842
    0.0000    1.0000   0.0000    0.0000
    0.7174   -0.0000    0.6967    0.0779
    0         0         0     1.0000
```

which returns a 3-dimensional matrix where the last index is the solution number. The true relative pose from view two to view one is

```
>> inv(cam1.T) * cam2.T
ans =
    0.6967          0   -0.7174    0.1842
        0    1.0000          0          0
    0.7174          0    0.6967    0.0779
        0          0          0     1.0000
```

which indicates that, in this case, solution two is the correct one.

See page 254.

See Appendix D.

A 3-dimensional translation  $(x, y, z)$  with unknown scale can be considered as  $(x', y', 1)$ .

If both images were captured with the same camera then  $K_1 = K_2$ .

Although Eq. 14.9 is written in terms of  $\$(R, t) \sim {}^2\xi_1$  the Toolbox function returns  ${}^1\xi_2$ .

As observed by Hartley and Zisserman (2003, p 259) not even the sign of  $\mathbf{t}$  can be determined.

Unusually we have recovered the camera translation exactly but since  $E \simeq \lambda E$  the translational part of the homogeneous transformation matrix has an unknown scale factor. In this case it is correct because the essential matrix was determined directly from the relative pose between the cameras.

In this case we could choose the correct solution because we knew the pose of the two cameras, but how do we determine the correct solution in practice? One approach is to determine whether a world point is visible. Typically we would choose a point on the optical axis in front of the first camera

```
>> Q = [0 0 10]';
```

and its projection to the first camera

```
ans =
429.7889
512.0000
```

is a reasonable value. We can create an instance of the first camera with an arbitrary displacement using the `move` method

```
>> cam1.move(sol(:,:,1)).project(Q)
ans =
NaN
NaN
```

and the values of `NaN` indicate that the world point is behind the camera. The second solution

```
>> cam1.move(sol(:,:,2)).project(Q)
ans =
594.2111
512.0000
```

has a finite value and indicates that this solution is the valid one. We can perform this more compactly by providing a test point

```
>> sol = cam1.invE(E, Q)
sol =
0.6967 0.0000 -0.7174 0.1842
0.0000 1.0000 0.0000 0.0000
0.7174 -0.0000 0.6967 0.0779
0 0 0 1.0000
```

in which case only the valid solution is returned.

In summary these  $3 \times 3$  matrices, the fundamental and the essential matrix, encode the geometry of the two cameras. The fundamental matrix and a point in one image defines an epipolar line in the other image along which its conjugate points must lie. The essential matrix encodes the relative pose of the two camera's centres and the pose can be extracted, with two possible values, and with translation scaled by an unknown factor. In this example the fundamental matrix was computed from known camera motion and intrinsic parameters. The real world isn't like this – camera motion is difficult to measure and the camera may not be calibrated. Instead we can estimate the fundamental matrix directly from corresponding image points.

### 14.2.3 Estimating the Fundamental Matrix

Assume that we have  $N$  pairs of corresponding points in two views of the same scene  $(^1\mathbf{p}_i, ^2\mathbf{p}_i)$ ,  $i = 1 \dots N$ . To demonstrate this we create a set of twenty random point features (within a  $2 \times 2 \times 2$  m cube) whose center is located 3 m in front of the cameras

```
>> P = homtrans( transl(-1, -1, 2), 2*rand(3,20) );
```

and project these points onto the two camera image planes

```
>> p1 = cam1.project(P);
>> p2 = cam2.project(P);
```

If  $N \geq 8$  the fundamental matrix can be estimated from these two sets of corresponding points

```
>> F = fmatrix(p1, p2)
maximum residual 1.932e-29
F =
    0.0000   -0.0000    0.0239
   -0.0000   -0.0000    0.0460
    0.0239    0.0018  -24.4896
```

where the residual is the maximum value of the left-hand side of Eq. 14.2 and is ideally zero. The value here is not zero, but it is very small, and this is due to the accumulation of errors from finite precision arithmetic. The estimated matrix has the required rank property

```
>> rank(F)
ans =
    2
```

For camera two we can plot the projected points

```
>> cam2.plot(P);
```

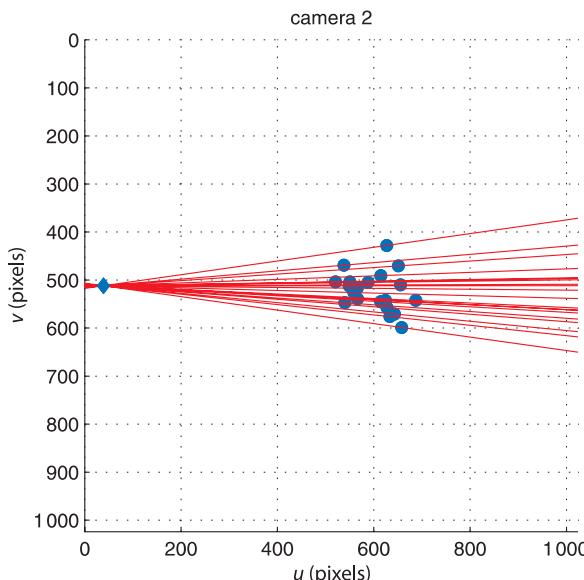
and overlay the epipolar lines generated by each point in image one

```
>> cam2.plot_epiline(F, p1, 'r')
```

which is shown in Fig. 14.8. We see a family or *pencil* of epipolar lines, and that every point in image two lies on an epipolar line. Note how the epipolar lines all converge on the epipole which is possible in this case▶ because the two cameras are verged as shown in Fig. 14.6.

To demonstrate the importance of correct point correspondence we will repeat the example above but introduce two *bad* data associations by swapping two elements in *p2*

```
>> p2(:,[8 7]) = p2(:,[7 8]);
```



The example has been contrived so that the epipoles lie within the images, that is, the each camera can see the centre of the other camera. A common imaging geometry is for the optical axes to be parallel, such as shown in Fig. 14.19 in which case the epipoles are at infinity (the third element of the homogeneous coordinate is zero) and all the epipolar lines are parallel.

**Fig. 14.8.**

A pencil of epipolar lines on the camera two image plane. Note how all epipolar lines pass through the epipole which is the projection of camera one's centre

### The fundamental matrix estimation

```
>> fmatrix(p1, p2)
maximum residual 236.2
ans =
    0.0000   -0.0000    0.0022
    0.0000    0.0000   -0.0023
   -0.0030    0.0014    1.0000
```

now has a very high residual – hundreds of pixels. This means that the point correspondence cannot be *explained* by the relationship Eq. 14.2.

If we knew the fundamental matrix we could test whether a pair of candidate corresponding points are in fact conjugates by measuring how far one is from the epipolar line defined by the other

```
>> epidist(F, p1(:,1), p2(:,1))
ans =
    2.3035e-13
>> epidist(F, p1(:,7), p2(:,7))
ans =
    18.8228
```

which shows that point 1 is a good fit, but point 7 (which we swapped with point 8), is a poor fit. However we have to first estimate the fundamental matrix and that requires that point correspondence is known. We break this deadlock with an ingenious algorithm called Random Sampling and Consensus or RANSAC.

The underlying principle is delightfully simple. Estimating a fundamental matrix requires eight points so we randomly choose eight candidate corresponding points (the sample) and estimate  $F$  to create a *model*. This model is tested against all the other candidate pairs and those that fit<sup>4</sup> vote for this model. The process is repeated a number of times and the model that had the most supporters (the consensus) is returned. Since the sample is small the chance that it contains all valid candidate pairs is high. The point pairs that support the model are termed inliers and those that do not are outliers.

RANSAC is remarkably effective and efficient at finding the inlier set, even in the presence of large numbers of outliers (more than 50%), and is applicable to a wide range of problems. Within the Toolbox we invoke RANSAC as a *driver* of the `fmatrix` function

```
>> [F,in,r] = ransac(@fmatrix, [p1; p2], 1e-6, 'verbose')
15 trials
2 outliers
3.48564e-29 final residual
```

and we obtain an excellent final residual. The set of inliers is also returned

```
>> in
in =
Columns 1 through 14
    1    2    3    4    5    6    9    10   11   12   13   14   15   16
Columns 15 through 18
    17   18   19   20
```

and the two incorrect associations, points 7 and 8, are notably absent from this list. The third parameter to `ransac` is the threshold  $t$  which is used to determine whether or not a point pair supports the model. If  $t$  is chosen to be too small RANSAC requires many more trials than its default maximum and this requires adjustment of additional parameters. Keep in mind also that the results of RANSAC will vary from run to run due to the random subsampling performed. Using RANSAC involve some trial and error to choose the correct threshold based on the final residual and the number of outliers. There are also a number of other options that are described in the online documentation.

We return now to the pair of images of the Eiffel tower shown in Fig. 14.3. When we left off at page 384 we had determined correspondence based on descriptor similarity

To within a defined threshold  $t$ . The Toolbox function `epidist` returns the distance between a point and an epipolar line.

but there were a number of clearly incorrect matches. RANSAC is available as a method `ransac` that operates on a vector of `FeatureMatch` objects

```
>> F = m.ransac(@fmatrix, 1e-4, 'verbose')
1527 trials
312 outliers
0.000140437 final residual
F =
    0.0000   -0.0000    0.0098
    0.0000    0.0000   -0.0148
   -0.0121    0.0129    3.6393
```

A small amount of trial and error was required to settle on the tolerance of  $10^{-4}$ . Making it smaller requires more RANSAC trials and requires raising the limit on maximum number of trials allowed but without any significant change in the result. It is also unrealistic to expect a very small residual since the real image data is subject to random error such as pixel noise and systematic error such as lens distortion.▶

RANSAC identified 312 outliers or incorrect data associations from the SURF feature matching stage which is nearly 50% of the *candidate* matches – the preliminary matching was worse than it looked. Running RANSAC has updated the elements of the `FeatureMatch` vector

```
>> m.show
ans =
1667 corresponding points
644 corresponding points
332 inliers (51.6%)
312 outliers (48.4%)
```

which displays the total number of inliers and outliers. Compared to page 384 the elements of the vector

```
>> m(1:5)
ans =
(819.56, 358.557) <-> (708.008, 563.342), dist=0.002137 +
(1028.3, 231.748) <-> (880.14, 461.094), dist=0.004057 -
(1027.6, 571.118) <-> (885.147, 742.088), dist=0.004297 +
(927.724, 509.93) <-> (800.833, 692.564), dist=0.004371 +
(854.35, 401.633) <-> (737.504, 602.187), dist=0.004417 +
```

now have a trailing plus or minus sign to indicate whether the corresponding match is an inlier or outlier. We can plot some of the inliers

```
>> idisp({im1, im2});
>> m.inlier.subset(100).plot('g')
```

or some of the outliers

```
>> idisp({im1, im2});
>> m.outlier.subset(100).plot('r')
```

and these are shown in Fig. 14.9.

An alternative way to create a `CentralCamera` object is from an image

```
>> cam = CentralCamera('image', im1);
```

The size of the pixel array is inferred from the image and the intrinsic parameters are set to default values. As before, we can overlay the epipolar lines computed from the corresponding points found in the second image

```
>> cam.plot_epiline(F', m.inlier.subset(20).p2, 'g');
```

and the result is shown in Fig. 14.10. The epipolar lines intersect at the epipolar point which we can clearly see is the projection of the second camera in the first image.▶ The epipole at

Lens distortion causes points to be displaced on the image plane and this violates the epipolar geometry. Images can be corrected as discussed in Sect. 12.6.4 but this is computationally expensive. A cheaper alternative is to correct the coordinates of the features by mapping them through the inverse distortion model Eq. 11.13.

We only plot a small subset of the epipolar lines since they are too numerous and would obscure the image.

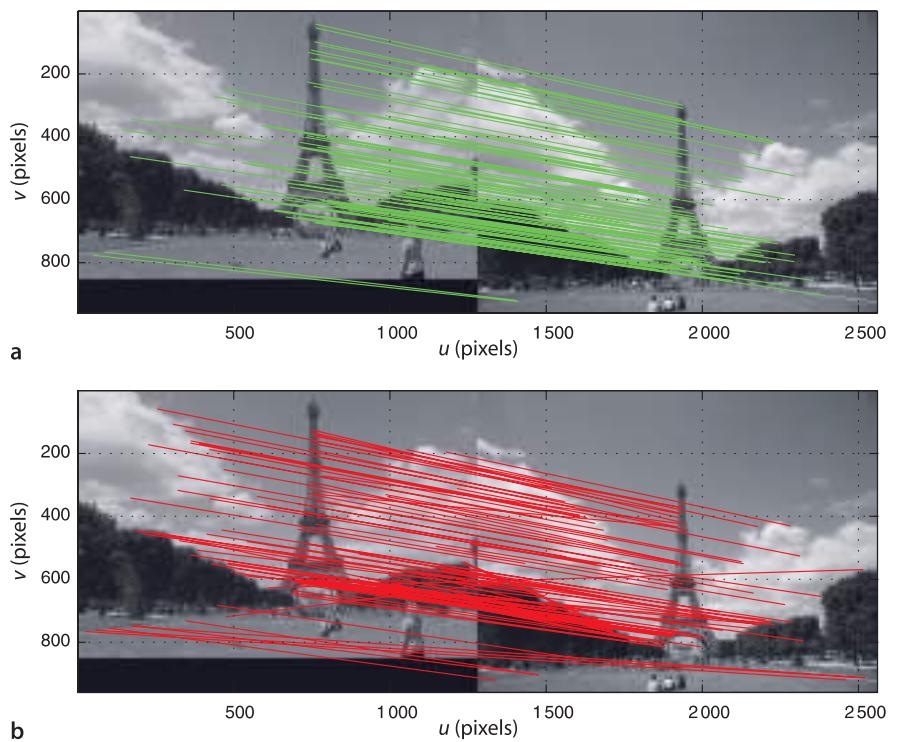
```

>> h2e( null(F) )
ans =
  1.0e+03 *
  1.0359
  0.6709
>> cam.plot(ans, 'bo')

```

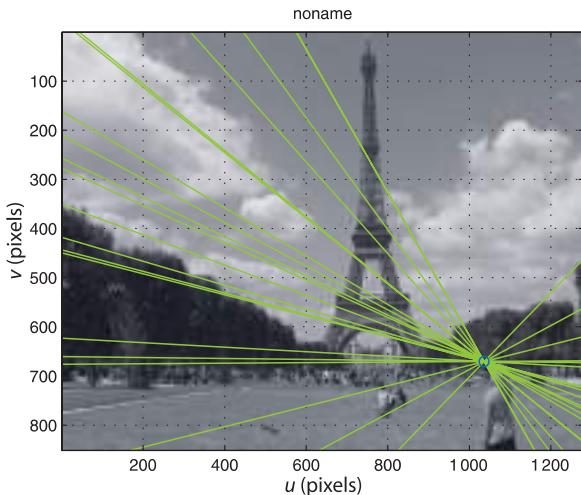
At the focal lengths used a 20 pix displacement on the image plane corresponds to a pointing error of less than 0.5°.

is also superimposed on the plot. With two handheld cameras and a common view we have been able to pinpoint the second camera in the first image. The result is not quite perfect – there is a horizontal offset of about 20 pixels which is likely to be due to a small pointing error in one or both cameras which were handheld and only approximately synchronized. ▶



**Fig. 14.9.**

Results of SURF feature matching after RANSAC. **a** Subset of all inlier matches; **b** subset of the outlier matches, some are quite visibly incorrect while others are more subtle



**Fig. 14.10.**

Image from Fig. 14.1a showing epipolar lines converging on the projection of the second camera's centre. In this case the second camera is clearly visible in the bottom right of the image

### 14.2.4 Planar Homography

In this section we will consider a camera viewing a group of world points  $P_i$  that lie on a plane. They are viewed by two different cameras and the projection in the cameras are  ${}^1p_i$  and  ${}^2p_i$  respectively which are related by

$${}^2\tilde{p}_i \simeq H {}^1\tilde{p}_i \quad (14.12)$$

where  $H$  is a non-singular  $3 \times 3$  matrix known as an homography, a planar homography, or the homography *induced* by the plane.

For example consider again the pair of cameras from page 387 now observing a  $3 \times 3$  grid of points

```
>> Tgrid = transl(0,0,1)*trotx(0.1)*trotz(0.2);
>> P = mkgrid(3, 1.0, 'T', Tgrid);
```

where `Tgrid` is the pose of the grid coordinate frame  $\{G\}$  and the grid points are centred in the frame's  $xy$ -plane. The points are projected to both cameras

```
>> p1 = cam1.plot(P, 'o');
>> p2 = cam2.plot(P, 'o');
```

and the images are shown in Fig. 14.11a and b respectively.

Just as we did for the fundamental matrix, if  $N \geq 8$  we can estimate the matrix `H` from two sets of corresponding points

```
>> H = homography(p1, p2)
H =
-0.4282 -0.0006 408.0894
-0.7030 0.3674 320.1340
-0.0014 -0.0000 1.0000
```

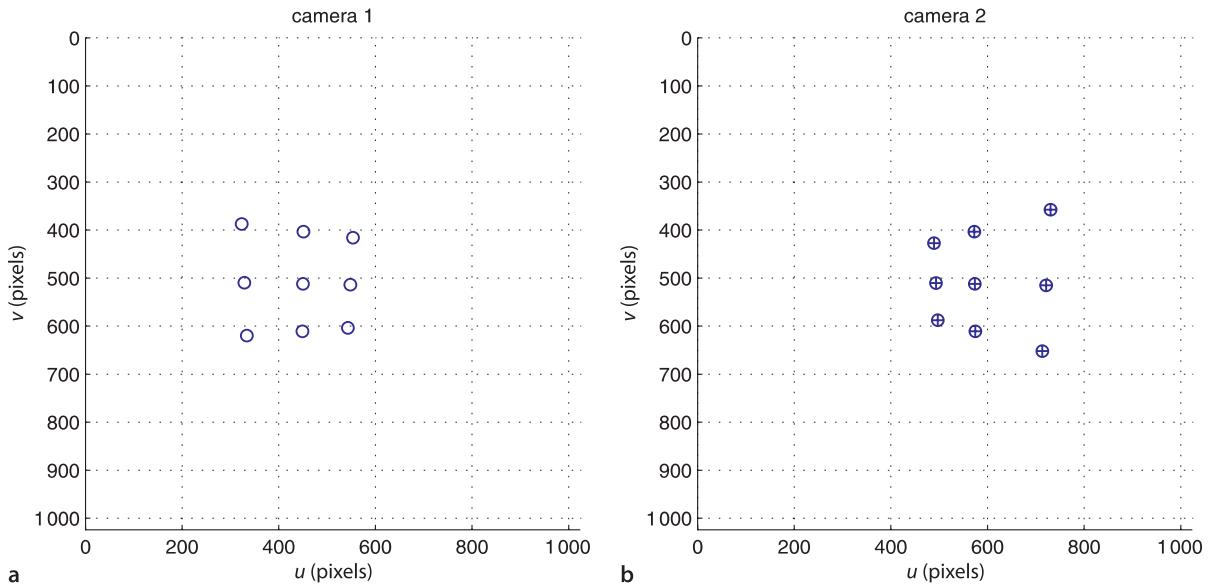
According to Eq. 14.12 we can predict the position of the grid points in image two from the corresponding image one coordinates

```
>> p2b = homtrans(H, p1);
```

which we can superimpose on image two as + symbols

```
>> cam2.hold()
>> cam2.plot(p2b, '+')
```

**Fig. 14.11.** Views of the oblique planar grid of points from two different view points. The grid points are projected as open circles. Plus signs in **b** indicate points transformed from the camera one image plane by the homography



This is shown in Fig. 14.11b and we see that the predicted points are perfectly aligned with the actual projection of the world points. The inverse of the homography matrix

$${}^1\tilde{p}_i \simeq H^{-1} {}^2\tilde{p}_i \quad (14.13)$$

performs the inverse mapping, from image two coordinates to image one

```
>> p1b = homtrans(inv(H), p2);
```

The fundamental matrix constrains the conjugate point to lie along a line but the homography tells us *exactly* where the conjugate point will be in the other image – provided that the points lie on a plane.

We can use this proviso to our advantage as a test for whether or not points lie on a plane. We will add some extra world points► to our example

```
>> Q = [
    -0.2302   -0.0545    0.2537
    0.3287    0.4523    0.6024
    0.4000    0.5000    0.6000  ];
```

which we plot in 3D

```
>> axis([-1 1 -1 1 0 2])
>> plot_sphere(P, 0.05, 'b')
>> plot_sphere(Q, 0.05, 'r')
>> cam1.plot_camera('color', 'b', 'label')
>> cam2.plot_camera('color', 'r', 'label')
```

and this is shown in Fig. 14.12. The new points, shown in red, are clearly not in the same plane as the original blue points. Viewed from camera one

```
>> p1 = cam1.plot([P Q], 'o');
```

as shown in Fig. 14.13a, these new points appear as an extra row in the grid of points we used above. However in the second view

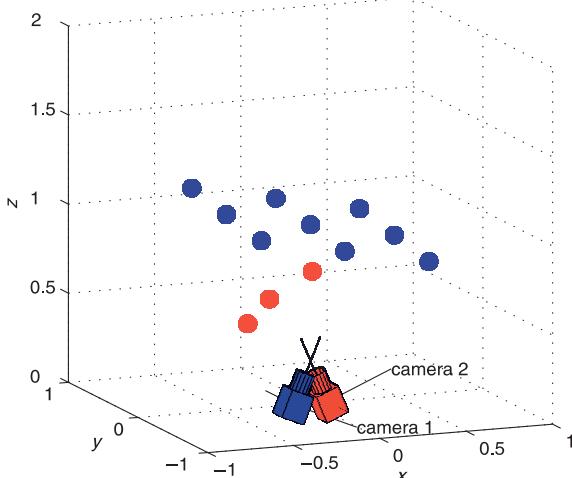
```
>> p2 = cam2.plot([P Q], 'o');
```

as shown in Fig. 14.13b these *out of plane* points no longer form a regular grid. If we apply the homography to the camera one image points

```
>> p2h = homtrans(H, p1);
```

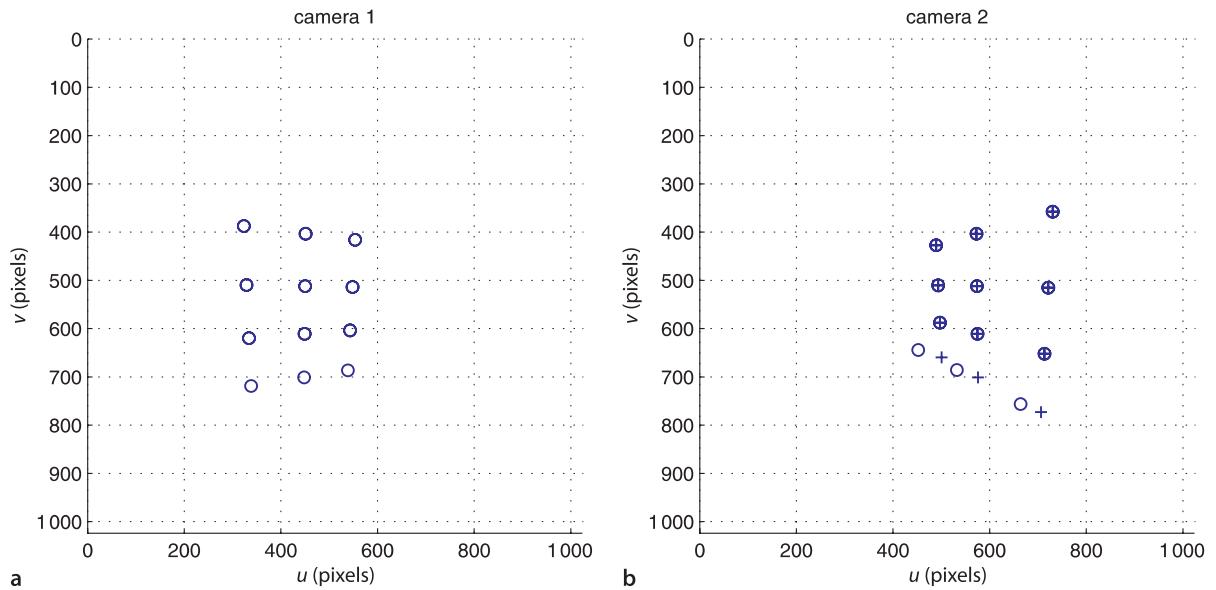
we find where they should be in camera two image if they belonged to the plane implicit in the homography

```
>> cam2.plot(p2h, '+')
```



**Fig. 14.12.**

World view of target points and two camera poses. Blue points lie in a planar grid, while the red points appear to lie in the grid from the viewpoint of camera one



We see that the original nine points overlap, but the three new points do not. We could make an automated test based on the prediction error

```
>> colnorm( homtrans(H, p1)-p2 )
ans =
Columns 1 through 9
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
    0.0000    0.0000    0.0000
Columns 10 through 12
    50.5969   46.4423   45.3836
```

which is large for these last three points – they do not belong to the plane that induced the homography.

In this example we estimated the homography based on two sets of corresponding points which were projections of known planar points. In practice we do not know in advance which points belong to the plane so we can again use RANSAC

```
>> [H,in] = ransac(@homography, [p1; p2], 0.1)
resid =
    4.0990e-13
H =
    -0.4282   -0.0006   408.0894
    -0.7030    0.3674   320.1340
    -0.0014   -0.0000    1.0000
in =
    1    2    3    4    5    6    7    8    9
```

which finds the homography that best explains the relationship between the sets of image points. It has also identified those points which support the homography and the three *out of plane points* are not on the inlier list.

The geometry related to the homography is shown in Fig. 14.14. We can express the homography in normalized image coordinates▶

**Fig. 14.13.** Views of the oblique planar grid of points from two different view points. The grid points are projected as open circles. Plus signs in **b** indicate points transformed from the camera one image plane by the homography. The bottom of row of points in each case are not co-planar with the other points

See page 254.

where  $H_E$  is the Euclidean homography which is written

$${}^2\tilde{x} \approx H_E {}^1\tilde{x} \quad (14.14)$$

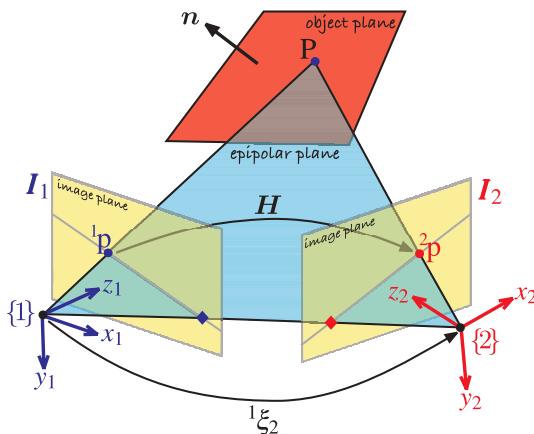


Fig. 14.14.

Geometry of homography showing two cameras with associated coordinate frames {1} and {2} and image planes. The world point  $P$  belongs to a plane with surface normal  $n$ .  $H$  is the homography, a  $3 \times 3$  matrix that maps  ${}^1p$  to  ${}^2p$

in terms of the motion  $(R, t) \sim {}^2\xi_1$  and the plane  $n^T P + d = 0$  with respect to frame {1}. The Euclidean and projective homographies are related by

$$H_E \simeq K^{-1}HK$$

where  $K$  is the camera parameter matrix.

As for the essential matrix the projective homography can be *decomposed* to yield the relative pose  ${}^1\xi_2$  in homogeneous transformation form as well as the normal to the plane. We use the `invH` method of the `CentralCamera` class

```
>> cam1.invH(H)
solution 1
    T = 0.82478   -0.01907   -0.56513   -0.01966
          0.01907   0.99980   -0.00591   -0.01917
          0.56513   -0.00591   0.82498   0.19911
          0.00000   0.00000   0.00000   1.00000
    n =  0.95519   0.00998   0.29582
solution 2
    T = 0.69671   0.00000   -0.71736   0.18513
          0.00000   1.00000   0.00000   -0.00000
          0.71736   -0.00000   0.69671   0.07827
          0.00000   0.00000   0.00000   1.00000
    n = -0.19676   -0.09784   0.97556
```

which returns a short structure array. Again there are multiple solutions and we need to apply additional information to determine the correct one. As usual the translational component of the transformation matrix has an unknown scale factor. We know from Fig. 14.12 that the camera motion is predominantly in the  $x$ -direction and that the plane normal is approximately parallel to the camera's optical- or  $z$ -axis and this knowledge helps us to choose solution two. The true transformation from camera one to two is

```
>> inv(T1)*T2
ans =
    0.6967      0   -0.7174   0.1842
    0   1.0000      0       0
    0.7174      0   0.6967   0.0779
    0       0       0   1.0000
```

The translation scale factor is quite close to one in this example, but in general it must be considered unknown.

and supports our choice. The pose of the grid with respect to camera one is

```
>> inv(T1)*Tgrid
ans =
    0.9797   -0.0389   -0.1968   -0.2973
    0.0198    0.9950   -0.0978       0
    0.1996    0.0920    0.9756   0.9600
    0       0       0   1.0000
```

and the third column is the grid's normal► which matches the estimated normal associated with solution two.

We can apply this technique to a pair of real images

```
>> im1=imread('garden-l.jpg', 'double');
>> im2=imread('garden-r.jpg', 'double');
```

shown in Fig. 14.15. We start by finding the SURF features

```
>> s1 = isurf(im1);
>> s2 = isurf(im2);
```

and the candidate corresponding points

```
>> m = s1.match(s2)
m =
323 corresponding points (listing suppressed)
```

then use RANSAC to find the set of corresponding points that best fits a plane in the world

```
>> [H,r] = m.ransac(@homography, 2)
H =
0.9966 0.0061 -15.6385
-0.0105 1.0081 -29.7619
-0.0000 0.0000 1.0000
r =
1.2228
```

The number of inlier and outlier points is

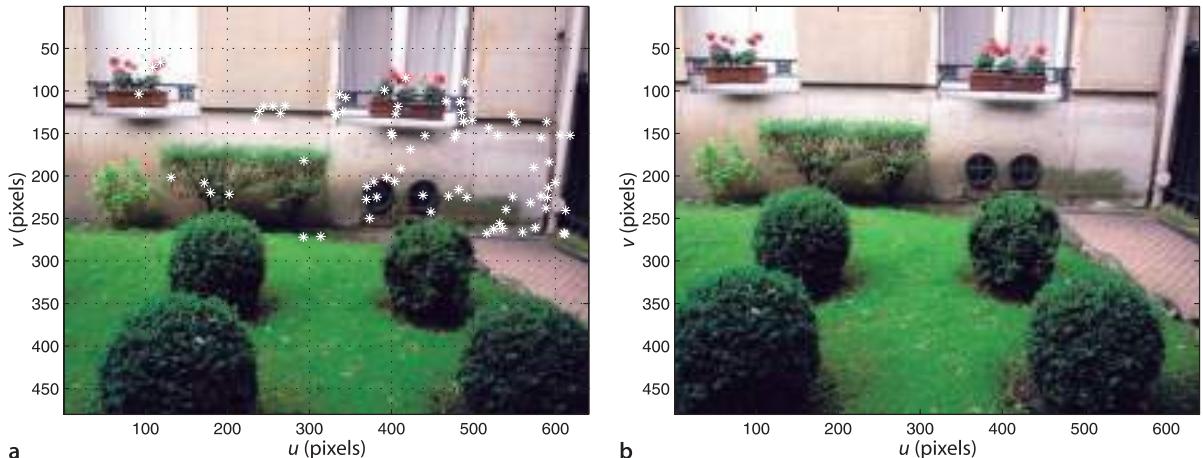
```
>> m.show
ans =
323 corresponding points
83 inliers (25.7%)
240 outliers (74.3%)
```

In this case the majority of point pairs do not fit the model, that is they do not belong to the plane that induces the homography  $H$ . However 83 points *do* belong to the plane and we can superimpose them on the figure

```
>> idisp(im1)
>> plot_point(m.inlier.pl, 'w*')
```

as shown in Fig. 14.15a. RANSAC has found a consensus which is the plane containing the wall. The tolerance was set to 4 pixel error since the dominant planes in this scene, the wall and the lawn, are only approximately planar. The lawn is quite large and contains many SURF feature points but the number of corresponding feature points, found

Since the points are in the  $xy$ -plane of the grid frame  $\{G\}$  the normal is the  $z$ -axis.



**Fig. 14.15.** Two pictures of a garden scene taken from different viewpoints. Image **b** approximately 30 cm to the right of image **a**. Image **a** has superimposed features that fit a plane. The camera was handheld with  $f = 5.2$  mm

by `match`, on the lawn is very low. If we remove the inlier points from the `FeatureMatch` vector, that is, we keep the outliers

```
>> m = m.outlier
```

and repeat the RANSAC homography estimation step we will find the next most dominant plane in the scene which turns out to be the top of the front bushes. Planes are very common in man-made environments and we will revisit homographies and their decomposition in Sect. 14.5.

### 14.3 Stereo Vision

Stereo vision is the technique of estimating the 3-dimensional structure of the world from two images taken from different viewpoints. We will discuss two approaches known as sparse and dense stereo respectively. Sparse stereo is a natural extension of what we have learned about feature matching and recovers the world coordinate  $(X, Y, Z)$  for each corresponding point pair. Dense stereo recovers the world coordinate  $(X, Y, Z)$  for *every pixel* in the image.

#### 14.3.1 Sparse Stereo

To illustrate sparse stereo we will return to the pair of garden images shown in Fig. 14.15. We have already found the SURF features so we will determine candidate matches based on descriptor similarity

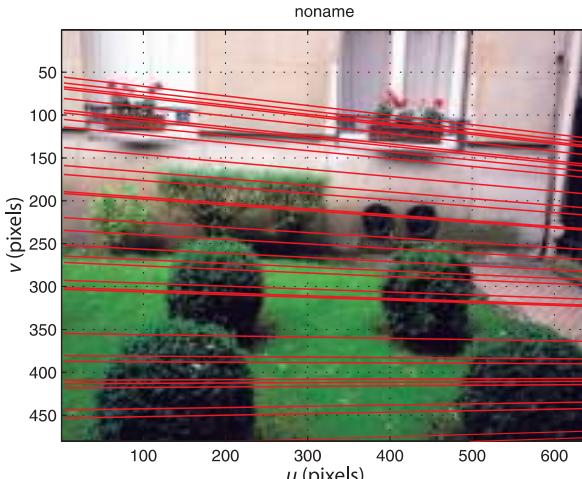
```
>> m = s1.match(s2)
m =
323 corresponding points (listing suppressed)
```

and then estimate the fundamental matrix

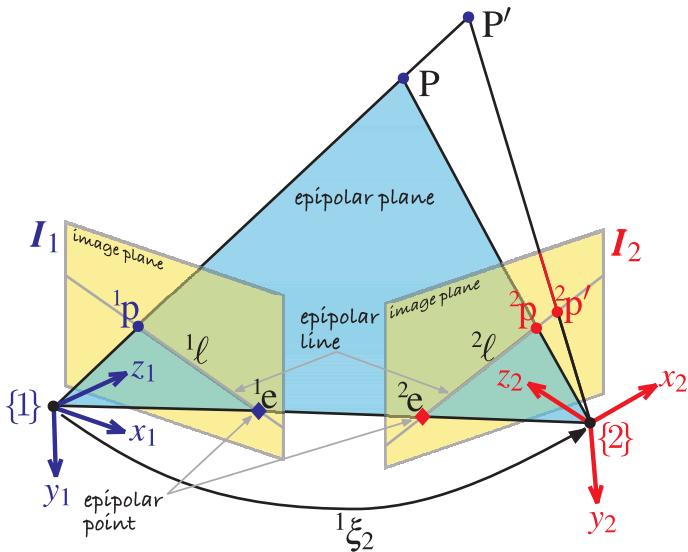
```
>> [F,r] = m.ransac(@fmatrix,1e-4, 'verbose');
77 trials
82 outliers
0.000131656 final residual
```

which captures the relative geometry of the two views. We can display the epipolar lines for a subset of points overlaid on the left-hand image

```
>> cam = CentralCamera('image', im1);
>> cam.plot_epiline(F', m.inlier.subset(30).p2, 'r');
```



**Fig. 14.16.**  
Image of Fig. 14.15a with epipolar lines for selected right image points superimposed



**Fig. 14.17.**  
Epipolar geometry for stereo vision. We can see clearly that as the depth of the world point increases, from  $P$  to  $P'$ , the projection moves along the epipolar line in the second image plane

which is shown in Fig. 14.16. In this case the epipolar lines are approximately horizontal and parallel which is expected for a camera motion that is a pure translation in the  $x$ -direction. Figure 14.17 shows the epipolar geometry for stereo vision. It is clear that as points move away from the camera,  $P$  to  $P'$  the conjugate points in the right-hand image moves to the right along the epipolar line.

The points  ${}^1p$  and  ${}^2p$  each define a ray in space which intersect at the world point. However to determine these rays we need to know the two poses of the camera and its intrinsic parameters. We can consider that the camera one frame  $\{1\}$  is the origin but the pose of camera two remains unknown. However we could estimate its pose by decomposing the essential matrix computed between between the two views. We have the fundamental matrix, but to determine the essential matrix according to Eq. 14.11 we need the camera's intrinsic parameters. With a little sleuthing we can find them!

The focal length used when the picture was taken is stored in the header of the EXIF-format file that holds the image and can be examined

```
>> [im1,tags] = imread('garden-1.jpg', 'double', 'mono');
```

where `tags` is a MATLAB® struct that contains various characteristics of the image as a structure of text strings. The element `DigitalCamera` describes the camera

```
>> tags.DigitalCamera
ans =
    ExposureTime: 0.0667
    FNumber: 3.3000
    .
    .
    .
    FocalLength: 5.2000
    .
    .
```

from which we determine the focal length is 5.2 mm.

The dimensions of the pixels  $\rho_w \times \rho_h$  are not included in the image header but some web-based research on this model camera gives the answer. This camera is reported to have an image sensor that measures  $7.18 \times 5.32$  mm and has  $3264 \times 2448$  pixel – both sets of numbers are consistent with a 4:3 aspect ratio. From this we can determine that the pixels are square and have a side length of  $2.2 \mu\text{m}$ . We create a `CentralCamera` object based on the known focal length, pixel size and image dimension

```

>> cam = CentralCamera('image', im1, 'focal', 5.2e-3, ...
    'sensor', [7.18e-3, 5.32e-3])
cam =
name: noname [central-perspective]
  focal length: 0.0052
  pixel size: (1.122e-05, 1.108e-05)
  principal pt: (320, 240)
  number pixels: 640 x 480
Tcam:
  1  0  0  0
  0  1  0  0
  0  0  1  0
  0  0  0  1

```

The effective pixel dimension is computed from the sensor dimensions and the pixels dimensions of the image. The image has been subsampled to  $640 \times 480$  pixels so the effective pixel size of  $11.2 \mu\text{m}$  is correspondingly larger. In the absence of any other information the principal point is assumed to be in the centre of the image.

The essential matrix is obtained by applying the camera intrinsic parameters to the fundamental matrix

```

>> E = cam.E(F)
E =
-0.0064   -0.5401    0.2240
  0.6914   -0.2065   -4.0732
 -0.1902    4.0445   -0.2493

```

and we then decompose it to determine the camera motion

```

>> sol = cam.invE(E, [0,0,10]')
sol =
  0.9995   -0.0011   -0.0331   -4.0549
  0.0029    0.9985    0.0554   -0.2324
  0.0329   -0.0555    0.9979   -0.6766
      0        0        0     1.0000
>> [R,t] = tr2rt(sol);

```

We chose a test point  ${}^1\mathbf{P} = (0, 0, 10)$ , a distant point along the optical axis, to determine the correct solution for the relative camera motion. Since the camera orientation was kept fairly constant the rotational part of the transformation is expected to be close to the identity matrix as we observe, and the actual rotation

```

>> tr2rpy(R, 'deg')
ans =
 -3.1781   -1.8940    0.0602

```

is just a couple of degrees of rotation about the  $x$ - and  $y$ -axes.

The estimated translation  $t$  from {1} to {2} has an unknown scale factor. Once again we bring in an extra piece of information – when we took the images the camera position changed by approximately 0.3 m in the positive  $x$ -direction. The estimated translation has the correct direction, dominant  $x$ -axis motion, but the sign and magnitude are quite wrong. We therefore scale the translation

```

>> t = 0.3 * t/t(1);
>> T = rt2tr(R, t)
T =
  0.9995   -0.0011   -0.0331    0.3000
  0.0029    0.9985    0.0554    0.0172
  0.0329   -0.0555    0.9979    0.0501
      0        0        0     1.0000

```

and we have  ${}^1\xi_2$  – the relative pose of camera two with respect to camera one represented as a homogeneous transformation.

Each point  $p$  in an image corresponds to a ray in space

$$\mathbf{P} = \alpha \mathbf{d} + \mathbf{P}_0, \quad \forall \alpha > 0$$

where  $\mathbf{P}_0$  is the centre of the camera and  $\mathbf{d}$  is the direction of the ray. If the camera matrix is  $C = (\Pi, c_4)$  where  $\Pi \in \mathbb{R}^{3 \times 3}$  and  $c_4 \in \mathbb{R}^3$  then the parameters of the ray are

$$\mathbf{d} = \Pi \tilde{\mathbf{p}}, \quad \mathbf{P}_0 = \Pi \mathbf{c}_4$$

Consider now the first corresponding point pair `m(1)`. The ray from camera one is

```
>> r1 = cam.ray(m(1).p1)
r1 =
d=(0.429152, -0.210292, 0.878411), P0=(0, 0, 0)
```

which is an instance of a `Ray3D` object with properties `P0` and `d` representing  $\mathbf{P}_0$  and  $\mathbf{d}$  respectively. The corresponding ray from the second camera is

```
>> r2 = cam.move(T).ray(m(1).p2)
r2 =
d=(0.38276, -0.21969, 0.897347), P0=(0.3, 0.017194, 0.0500546)
```

where the `move` method returns an instance copy of the `CentralCamera` object `cam` with the specified relative pose. The two rays intersect at

```
>> [x,e] = r1.intersect(r2);
>> x'
ans =
2.2031    -1.0796     4.5094
```

which is a point with a  $z$ -coordinate, or depth, of 4.51 m. Due to errors in the estimate of camera two's pose the two rays do not actually intersect, but their closest point is returned. At their closest point the rays are

```
>> e
e =
0.0049
```

around 5 cm apart. Considering the lack of rigour in this exercise, two handheld camera shots and only approximate knowledge of the magnitude of the camera displacement, the recovered depth information is quite remarkable.►

We draw a subset of twenty corresponding points from the inlier set

```
>> m2 = m.inlier.subset(20);
```

and then compute the rays in world space from each camera

```
>> r1 = cam.ray( m2.p1 );
>> r2 = cam.move(T).ray( m2.p2 );
```

which are each vectors of `Ray3D` objects. Their intersection points are

```
>> [P,e] = r1.intersect(r2);
```

where `P` is a matrix of closest points, one per column, and the last row

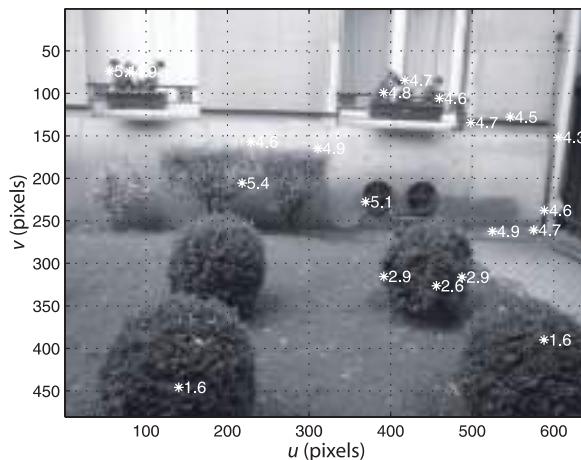
```
>> z = P(3,:);
```

is the depth coordinate. The columns of the vector `e` contains the distance between the rays at their closest points. We can superimpose the distance to each point on the image of the garden

```
>> idisp(im1)
>> plot_point(m.inlier.subset(20).p1, 'w*', 'textcolor', 'w', ...
'printf', {'%.1f', z});
```

which is shown in Fig. 14.18 and the feature markers are annotated with the estimated depth.

Even small errors in the estimated rotation between the camera poses will lead to large closing errors at distances of several metres. The closing error observed here would be induced by a rotational error of less than 1 deg.



**Fig. 14.18.**  
Image of Fig. 14.15a with depth  
of selected points indicated (units  
of metres)



**Fig. 14.19.**  
A stereo vision sensor which with PC-based software can compute depth maps at 25 frames per second. The cameras and lenses are identical and the relative pose is constant (image courtesy of Point Grey Research Inc.)

This is an example of stereopsis where we have used information from two overlapping images to infer the 3-dimensional position of points in the world. For obvious reasons the approach used here is referred to as sparse stereo because we only compute distance at a tiny subset of pixels in the image. More commonly the relative pose between the cameras would be known as would the camera intrinsic parameters.

### 14.3.2 Dense Stereo Matching

A stereo pair is taken by two cameras, generally with parallel optical axes, and separated by a known distance referred to as the camera baseline. Figure 14.19 shows a typical stereo camera system which simultaneously captures images from both cameras and transfers them to a host computer for processing.

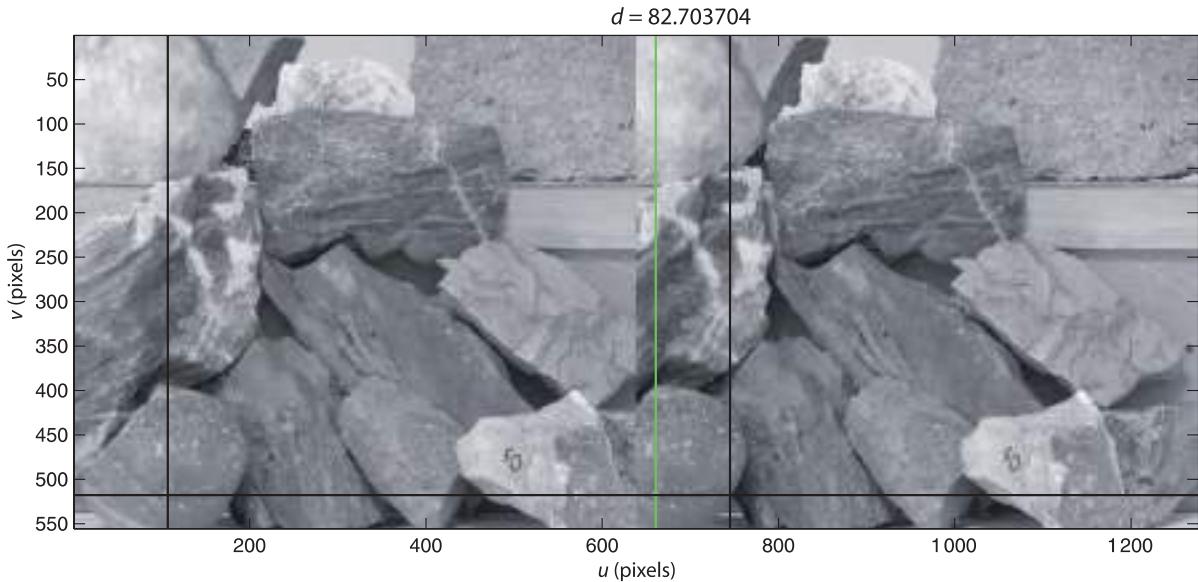
To illustrate we load an example stereo pair

```
>> L = iread('rocks2-l.png', 'reduce', 2);
>> R = iread('rocks2-r.png', 'reduce', 2);
```

We can interactively examine these two images together

```
>> stdisp(L, R)
```

as shown in Fig. 14.20. Clicking on a point in the left-hand image updates a pair of cross hairs that mark the *same* coordinate in the right-hand image. Clicking in the right-hand image sets another vertical cross hair and displays the difference between the horizontal coordinate of the two crosshairs. The cross hairs as shown are set to a small white spot on the front leftmost rock and we observe several things. Firstly the spot has the same vertical coordinate in both images, and this implies that the epipolar line is horizontal. Secondly, in the right-hand image the spot has moved to the left by 82.4 pixel. If we probed more points we would see that disparity is greater for points that are closer to the camera.



As shown in Fig. 14.17 the conjugate point in the right-hand image moves rightward along the epipolar line as the point depth increases. For the parallel-axis camera geometry the epipolar lines are parallel and horizontal, so conjugate points have the same  $v$ -coordinate. If the coordinates of two corresponding points are  $(^L u, ^L v)$  and  $(^R u, ^R v)$  then  $^R v = ^L v$ . The displacement along the horizontal epipolar line  $d = ^L u - ^R u$  where  $d \geq 0$  is called disparity.

The dense stereo process is illustrated in Fig. 14.21. For the pixel at  $(^L u, ^L v)$  in the left-hand image we know that its corresponding pixel is at some coordinate  $(^L u - d, ^L v)$  in the right-hand image where  $d \in [d_{\min}, d_{\max}]$ . To reliably find the corresponding point for a pixel in the left-hand image we create a  $W \times W$  pixel *template* region  $T$  about that pixel. As shown in Fig. 14.21 we *slide* the template window horizontally across the right-hand image. The position at which the template is most similar is considered to be the corresponding point from which disparity is calculated. Compared to other matching problems we have encountered this one is much simpler because there is no change in relative scale or orientation between the two images.

The epipolar constraint means that we only need to perform a 1-dimensional search for the corresponding point. The template is moved in  $D$  steps of 1 pixel in the range  $d_{\min} \dots d_{\max}$ . At each template position we perform a template matching operation, such as we discussed in Sect. 12.4.2, and for a  $W \times W$  window these have a computational cost of  $O(W^2)$ . The total cost of dense stereo matching is  $O(DW^2N^2)$  which is high but feasible in real time.

To perform stereo matching for the image pair in Fig. 14.20 using the Toolbox is quite straightforward

```
>> d = istereo(L, R, [40, 90], 3);
```

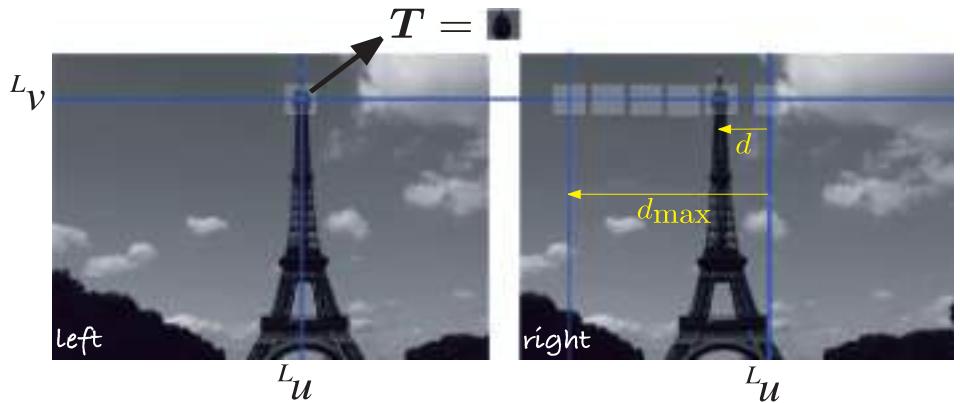
The result is a matrix the same size as  $L$  and the value of each element  $d[u, v]$ , or  $d(v, u)$  in MATLAB®, is the disparity at that coordinate in the left image. The corresponding pixel in the right image would be at  $(u - d[u, v], v)$ . We can display the disparity as a disparity image

```
>> idisp(d, 'bar')
```

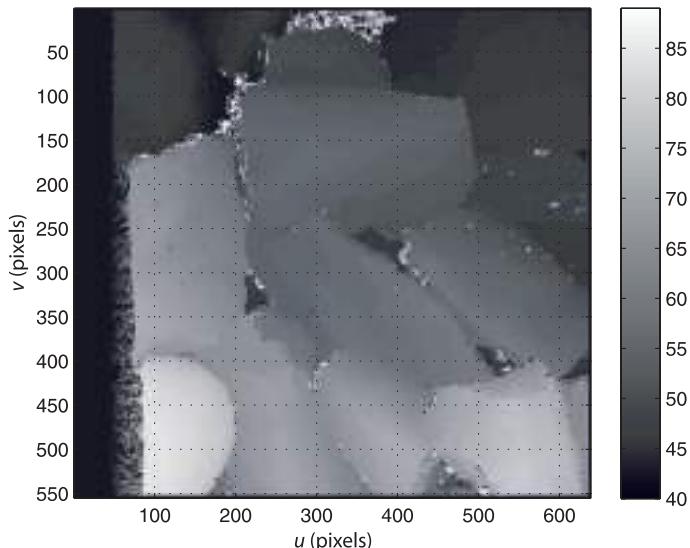
which is shown in Fig. 14.22. Disparity images have a distinctive ghostly appearance since all surface color and texture is absent. The third argument to `stereo` is the range of disparities to be searched, in this case from 40 to 90 pixel so the pixel values in the disparity image lie in the range [40, 90]. The disparity range was determined by examining some far and near points using `stdisp`.<sup>►</sup> The fourth argument to `istereo`

**Fig. 14.20.** The `stdisp` image browsing window. The black cross hair in the left-hand image has been positioned on a small white spot on the bottom-left rock. Another black cross hair is automatically positioned at the same coordinate in the right-hand image. Clicking on the corresponding point in the right-hand image sets the green cross-hair, and the panel at the top indicates a horizontal shift of 82.7 pixel to the left. This stereo image pair is from the Middlebury stereo database (Scharstein and Pal 2007). The focal length  $f/\rho$  is 3740 pixel, and the baseline is 160 mm. The images have been cropped so that the actual disparity should be offset by 274 pixel

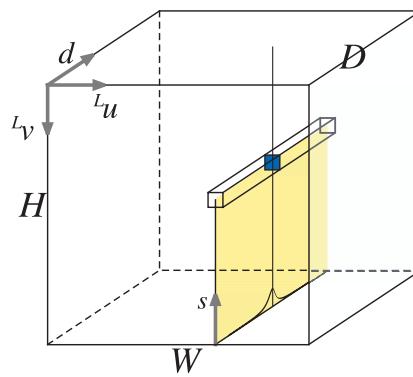
We could choose a range such as [0, 90] but this increases the search time: 91 disparities would have to be evaluated instead of 51. It also increases the possibility of matching errors.



**Fig. 14.21.**  
Stereo matching. A search centred at  $(u_r, v_r)$  in the right image is swept horizontally until it matches the template window  $T$  from the left image



**Fig. 14.22.**  
Disparity image for the rock pile stereo pair, where brighter means higher disparity or shorter range



**Fig. 14.23.**  
The disparity space image is a 3-dimensional image where element  $D(u, v, d)$  is the similarity between the support regions centered at  $(u_l, v_l)$  in the left image and  $(u_r - d, v_r)$  in the right image

is the half-width of the template, in this case we are using a  $7 \times 7$  window. By default `stereo` uses the ZNCC similarity measure.

In the disparity image we can clearly see that the rocks at the bottom of the pile have a larger disparity and are closer to the camera than those at the top. There are also some errors, such as the anomalous bright values around the edges of the rocks. These pixels are indicated as being nearer than they really are. The similarity score is set to `NaN` around the edge of the image where the similarity matching template falls off the edge of the image and to `Inf` for the case where the denominator of the ZNCC similarity metric (Table 12.1) is equal to zero. ▶ The values `NaN` and `Inf` are both displayed as black.

This occurs if all the pixels in either template have exactly the same value.

### 14.3.2.1 Stereo Failure Modes

The stereo function can also return the disparity space image (DSI)

```
>> [d,sim,DSI] = istereo(L, R, [40 90], 3);
```

which is an  $H \times W \times D$  matrix

```
>> about(DSI)
DSI [double] : 555x638x51 (144468720 bytes)
```

whose elements ( $v, u, d$ ) are the similarity measure between the templates centred at  $(u, v)$  in the left image and  $(u - d, v)$  in the right image.<sup>►</sup> The disparity image we saw earlier is simply the position of the maximum value in the  $d$ -direction evaluated at every pixel<sup>►</sup> and the matrix `sim` is the value of those maxima.

Each column in the  $d$ -direction gives the similarity measure versus disparity for the corresponding pixel in the left image. For the pixel at (138, 439) we can plot this

```
>> plot( squeeze(DSI(439,138,:)), 'o-' );
```

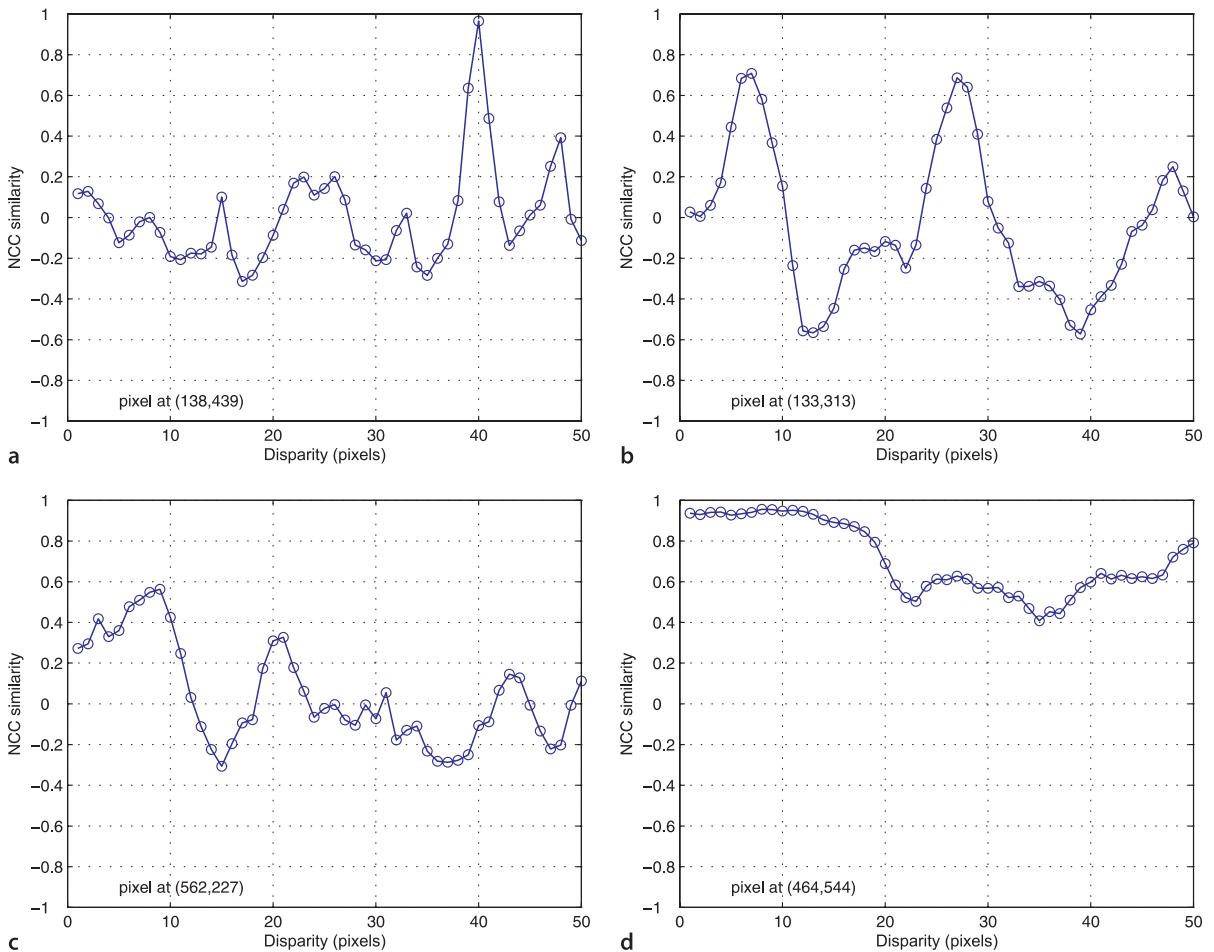
which is shown in Fig. 14.24a. We are using the ZNCC measure and we see a perfect match occurs at a disparity of 80 pixel, since the horizontal axis is  $d - d_{\min}$  and  $d_{\min} = 40$ .

Figure 14.24 shows some very typical plots of similarity metric as a function of disparity and this provides insight into the causes of error in stereo matching. Figure 14.24a shows a single unambiguous strong peak nearly equal to one which is the value for a perfect match. Fortunately this case is very common.

This is a large matrix (144 Mbyte) which is why the images were reduced in size when loaded.

This is a simplistic approach. A better approach is to apply regularization and estimate a function  $g(u, v)$  that fits the points of maximum similarity while maintaining smoothness and continuity.

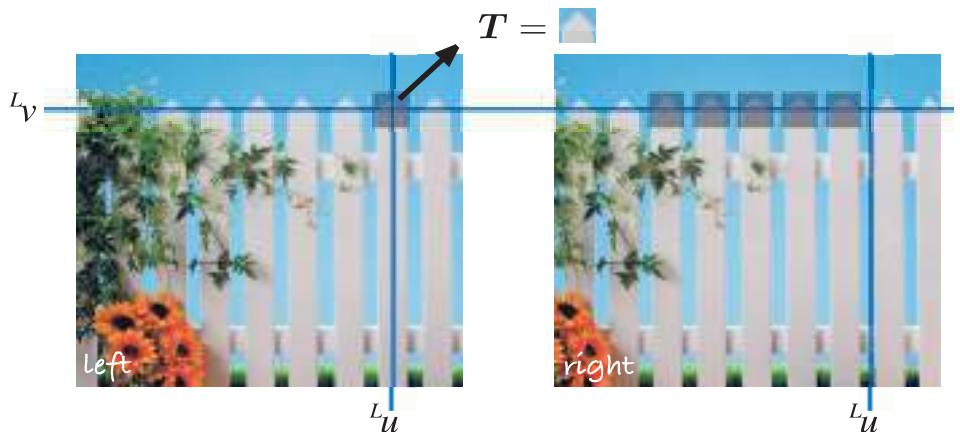
**Fig. 14.24.** Some typical ZNCC metric versus disparity curves.  
a Single strong peak; b multiple peaks; c weak peak; d broad peak



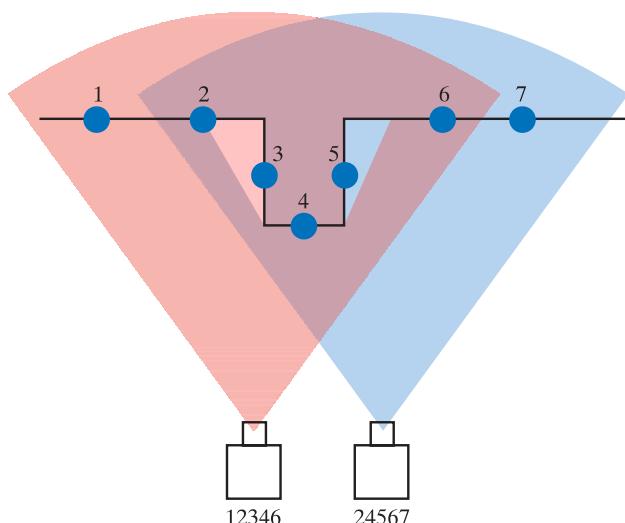
Multi-camera stereo, more than two cameras, is a powerful method to solve this ambiguity.

Two peaks of almost similar amplitude are shown in Fig. 14.24b. This means that the template pattern was found twice in the search region. This typically occurs when there are regular vertical features in the scene as is often the case in man-made scenes: brick walls, rows of windows, architectural features or a picket fence. The problem, illustrated in Fig. 14.25, is commonly known as the picket fence effect and more properly as spatial aliasing. There is no real cure for this problem<sup>14</sup> but we can detect its presence. The ambiguity ratio is the ratio of the height of second peak to the height of the first peak – a high-value indicate that the result is uncertain and should not be used. The chance of detecting incorrect peaks can be reduced by ensuring that the disparity range used in `iStereo` is as small as possible and this requires some knowledge of the expected range of objects.

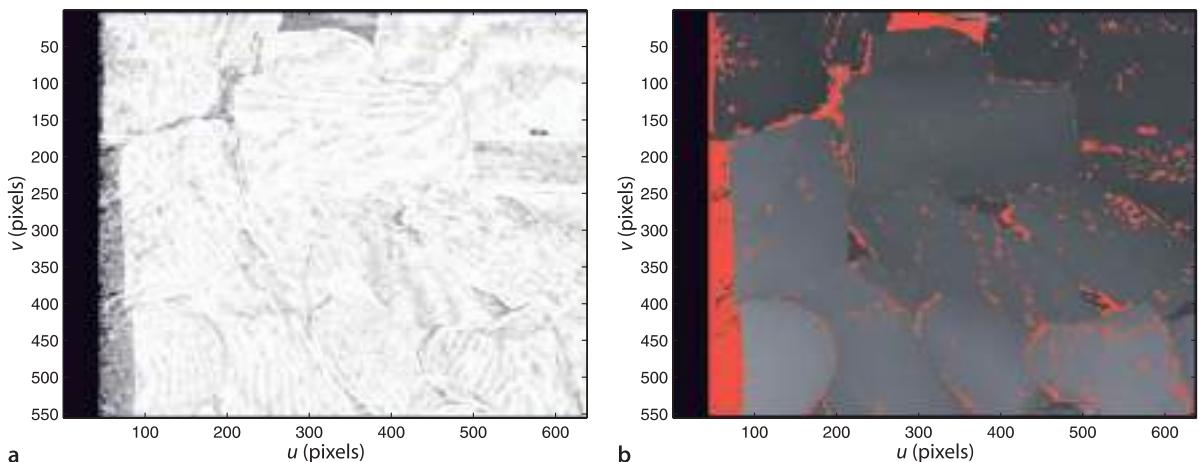
A weak peak is shown in Fig. 14.24c. This typically occurs when the corresponding scene point is not visible in the right-hand view due to occlusion or the missing parts problem. Occlusion is illustrated in Fig. 14.26 and it is clear that point 3 is only visible to the left camera. Dense stereo matching will attempt to find the best match in the right-hand image, but it will not find a point corresponding to 3. Even though the figure is an exaggerated depiction, real images suffer this problem where the depth changes rapidly. For example, this occurs at the edges of the rocks which is exactly where we observe the incorrect disparities in Fig. 14.22. The problem becomes more prevalent as the baseline increases. The problem also occurs when the corresponding point does not lie within the disparity search range, that is, the disparity search range is too small.



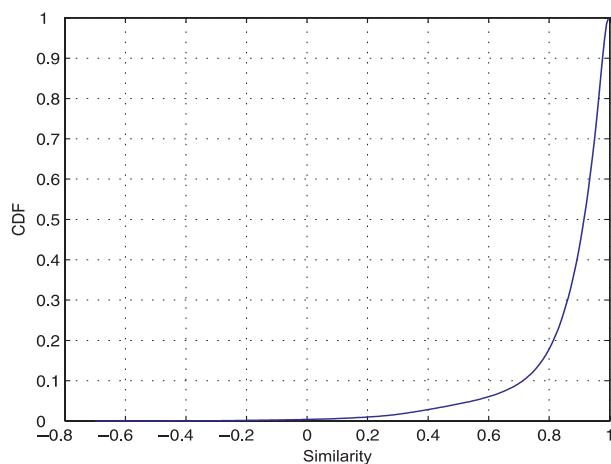
**Fig. 14.25.**  
Picket fence effect. The template will match well at a number of different disparities. This problem occurs in any scene with repeating patterns



**Fig. 14.26.**  
Occlusion in stereo vision. The field of view of the two cameras are shown as colored sectors. Points 1 and 7 fall outside the overlapping view area and are seen by only one camera each. Point 5 is occluded from the left camera and point 3 is occluded from the right camera. The order of points seen by the cameras is given



**Fig. 14.27.** Stereo template similarity. **a** Similarity image where brighter means higher similarity; **b** disparity image with pixels having low similarity score marked in red



**Fig. 14.28.**  
Cumulative probability of ZNCC scores. The probability of a score less than 0.9 is 45%

The problem cannot be cured but it can be detected. The simplest method is to apply a threshold to the similarity score and ignore those disparity results where similarity is low. The `istereo` function can return the value of the similarity score at the peak

```
>> [d,sim] = istereo(L, R, [40 90], 3);
>> idisp(sim)
```

and this is shown in Fig. 14.27a. We see that the erroneous disparity values correspond to low maximum similarity score. For example

```
>> ipixswitch(sim<0.7, 'red', d/90);
```

shown in Fig. 14.27b displays pixels with similarity  $s < 0.7$  as red. The distribution of maximum similarity scores

```
>> ihist(sim(isfinite(sim))), 'normcdf');
```

is shown in Fig. 14.28. We see that only 5% of pixels have a similarity score less than 0.5, and that 55% of pixels have a similarity score greater than 0.9.

A more powerful means to test for occlusion is to perform the matching in two directions which is known as the left-right consistency check. Starting with a pixel in the left-hand image the strongest match in the right-image is found. Then the strongest match to that pixel is found in the left-hand image. If this is the original left-hand image point the match is considered valid, otherwise it is discarded.

From Fig. 14.26 it is clear that pixels on the left-side of the left-hand image may not overlap at all with the right-hand image – point 1 is outside the field of view of the

right-hand camera. This is the reason for the large number of incorrect matches on the left hand side of the disparity image in Fig. 14.22. It is common practice to discard the  $d_{\max}$  left-most columns of the disparity image.

The final problem that can arise is a similarity function with a very broad peak as shown in Fig. 14.24d. The breadth makes it difficult to precisely estimate the maxima. This generally occurs when the template region has very low texture for example corresponding to the sky, dark shadows, sheets of water, snow, ice or smooth man-made objects. Simply put, in a region that is all grey, a grey template matches equally well with any number of grey candidate regions. One approach to detect this is to look at the variability of pixel values in the template using measures such as the difference between the maximum and minimum value or the variance of the pixel values. If the template has too little variance it is less likely to give a strong peak. Measures of peak sharpness can also be used to eliminate these cases and this is discussed in the next section.

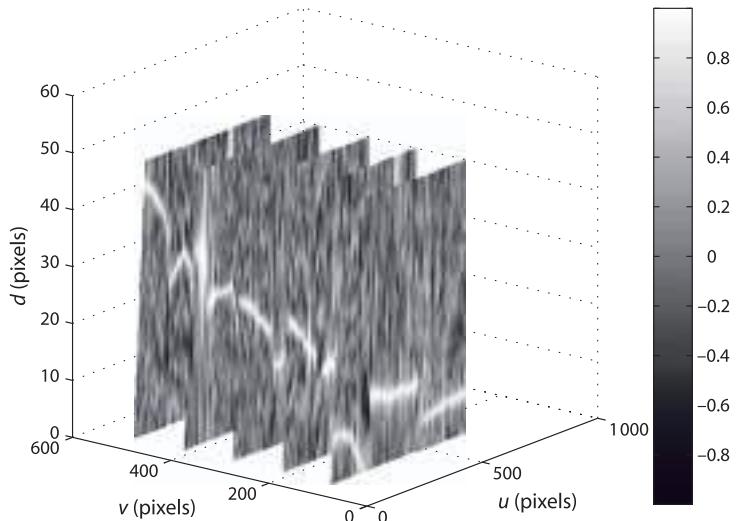
For the various problem cases just discussed disparity cannot be determined, but the problem can be detected. This is important since it allows those pixels to be marked as having no known range and this allows a robot path planner to be prudent with respect to regions whose 3-dimensional structure cannot be reliably determined.

The design of a stereo-vision system has three degrees of freedom. The first is the baseline distance between the cameras. As it increases the disparities become larger making it possible to estimate depth to greater precision, but the occlusion problem becomes worse. The disparity search range needs to be set carefully. If the maximum is too large the chance of spatial aliasing increases but if too small then points close to the camera will generate incorrect and weak matches. A large disparity range also increases the computation time. Template size involves a tradeoff between computation time and quality of the disparity image. A small template size can pick up fine depth structure but tends to give results that are much noisier since a small template is more susceptible to ambiguous matches. A large template gives a smoother disparity image but results in greater computation time. It also more likely that the template will contain pixels that belong to objects at different depths which is referred to as the mixed pixel problem. This tends to cause poor quality matching at the edges of objects, and the resulting disparity map appears blurred.

An alternative way to look at the failure modes is to use MATLAB's volume visualization functions to create horizontal slices through the disparity space image

```
>> slice(DSI, [], [100 200 300 400 500], []);
>> shading interp; colorbar
```

which is shown in Fig. 14.29. These are slices at constant  $v$ -coordinate, and within each of the  $ud$ -planes we see a bright path (high similarity values) that represents



**Fig. 14.29.**

The disparity space image is a 3-dimensional image where element  $D(u, v, d)$  is the similarity between the support regions centered at  $(^L u, ^L v)$  in the left image and  $(^L u - d, ^L v)$  in the right image

disparity  $d(u)$ . Note the significant discontinuities in the path for the plane at  $v = 100$  which correspond to sudden changes in depth. The planes at  $v = 200, 300, 400$  show that the path also fades away in places. In these regions the maximum similarity is low, there is no strong match in the right-hand image, and the most likely cause is occlusion.

### 14.3.3 Peak Refinement

The disparity at each pixel is an integer value  $d \in [d_{\min}, d_{\max}]$  at which the optimal similarity was found. Figure 14.24a shows a single unambiguous strong peak and we can use the disparity values adjacent to the peak to refine the estimate of the peak's position.► A parabola

$$s = Ad^2 + Bd + C \quad (14.15)$$

This two-dimensional peak refinement is discussed in Appendix K.

is fitted to the optimal disparity and its immediate neighbours. The optimal ZNCC similarity measure is a maxima which means that the parabola is inverted and  $A < 0$ . The interpolated maximum occurs when the derivative of Eq. 14.15 equals zero, from which we can estimate the horizontal position of the peak of the fitted parabola

$$\hat{d} = \frac{-B}{2A}$$

The  $A$  coefficient will be large for a sharp peak, and a simple threshold can be used to reject broad peaks, as we will discuss in the next section.

Disparity peak refinement is enabled with the '`interp`' option

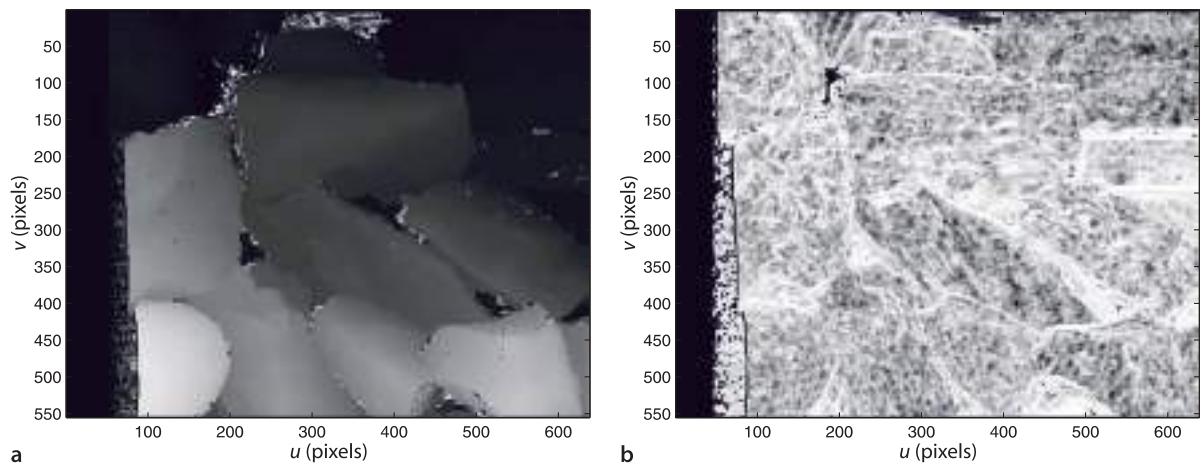
```
>> [di,sim,peak] = istereo(L, R, [40 90], 3, 'interp');
>> idisp(di)
```

and the resulting disparity image is shown in Fig. 14.30a. We see that it is much smoother than the one shown previously in Fig. 14.22. The additional optional output argument `peak` is a structure

```
>> peak
peak =
A: [555x638 double]
B: [555x638 double]
```

that contains the per-pixel values of the parabola coefficients. The  $A$  coefficient is shown as an image in Fig 14.30b.

**Fig. 14.30.a** Disparity image with peak refinement; **b** coefficient of  $d^2$  term at each peak. High values (bright) correspond to sharp peaks and occur where image texture is high. Broad peaks (dark) occur where image texture is low



#### 14.3.4 Cleaning up and Reconstruction

The result of stereo matching, such as shown in Fig. 14.22, has a number of imperfections for the reasons we have just described. For robotic applications such as path planning and obstacle avoidance it is important to know the 3-dimensional structure of the world, but it is also critically important to know what we don't know. Where reliable depth information from stereo vision is missing a robot should be prudent and treat it differently to free space. We use a number of simple measures to mark elements of the disparity image as being invalid or unreliable.

We start by creating a matrix `status` the same size as `d` and initialized to zero

```
>> status = zeros(size(d));
```

The elements are set to non-zero values if they correspond to specific failure conditions

```
>> [U,V] = imeshgrid(L);
>> status(isnan(d)) = 4;
>> status(U<=90) = 1;
>> status(sim<0.8) = 2;
>> status(peak.A>=-0.1) = 3;
```

We can display this matrix as an image

```
>> idisp(status)
>> colormap( colormap({'lightgreen', 'cyan', 'blue', 'orange', 'red'}))
```

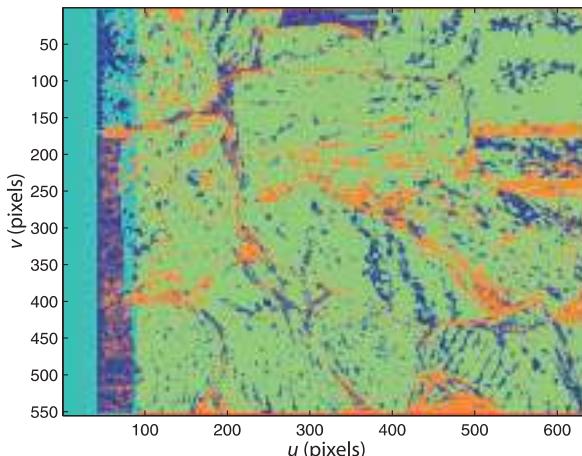
which is shown in Fig. 14.31. The colormap is chosen to display the status values as light green for a good stereo match, cyan if the disparity search range extends beyond the left edge of the right image, blue if the peak similarity is too small, orange if the peak is too broad, and red for `NaN` values where the search template would fall off the edge of the image. The good news is that there are a lot of light green pixels! In fact

```
>> sum(status(:)) / prod(size(status)) * 100
ans =
86.0674
```

nearly 90% of disparity values pass our battery of quality tests. The blue pixels, indicating weak similarity, occur around the edges of rocks and are due to occlusion. The orange pixels, indicating a broad peak, occur in areas that are fairly smooth, either deep shadow between rocks or the non-rock background.

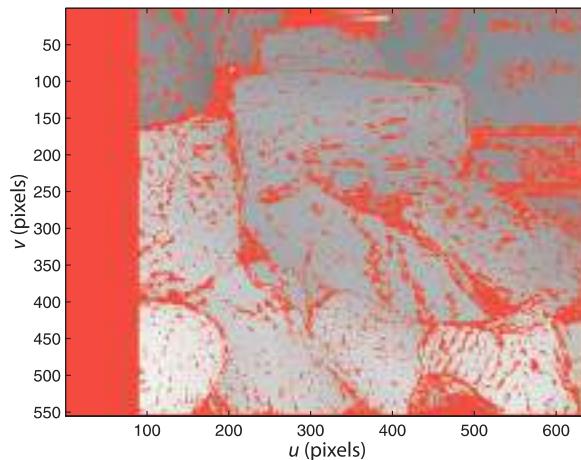
Earlier we created an interpolated disparity image `di` and now we will invalidate the disparity values that we have determined to be unreliable

```
>> di(status>0) = NaN;
```

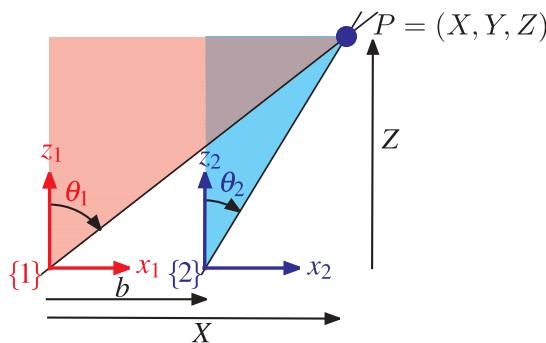


**Fig. 14.31.**

Stereo matching status on a per pixel basis. Good stereo match (green); disparity search range extends beyond the left edge of the right image (cyan); low maximum similarity (blue); no-sharp peak (orange); search template beyond edge of image (red)



**Fig. 14.32.**  
Interpolated disparity image  
with unreliable estimates  
indicated in red



**Fig. 14.33.**  
Stereo geometry for parallel  
camera axes.  $X$  and  $Z$  are  
measured with respect to  
camera one,  $b$  is the baseline

by setting them to the value `NaN`.<sup>►</sup> We can display this with the *unreliable* pixels marked in red by

```
>> ipixswitch(isnan(di), 'red', di/90);
```

which is shown in Fig. 14.32.<sup>►</sup> This is now in useful form for a robot – it contains disparity values interpolated to better than a pixel and all unreliable values are clearly marked.

The final step is to convert the disparity values in pixels to world coordinates in metres – a process known as 3D reconstruction. In the earlier discussion on sparse stereo we determined the world point from the intersection of two rays in 3-dimensional space. For a parallel axis stereo camera rig as shown in Fig. 14.19 the geometry is much simpler as illustrated in Fig. 14.33. For the red and blue triangles we can write

$$X = Z \tan \theta_1, \quad X - b = Z \tan \theta_2$$

where  $b$  is the baseline and the angles of the rays correspond to the horizontal image coordinate  ${}^i u$ ,  $i = \{L, R\}$

$$\tan \theta_i = \frac{\rho_u ({}^i u - u_0)}{f}$$

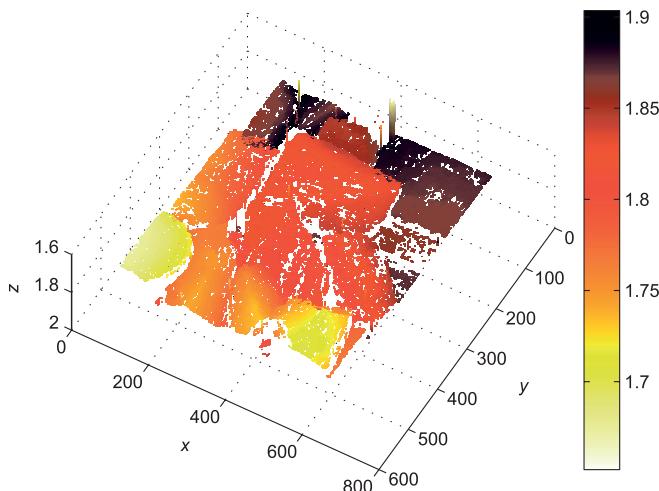
Substituting and eliminating  $X$  gives

$$Z = \frac{fb}{\rho_u ({}^L u - {}^R u)} = \frac{fb}{\rho_u d}$$

which shows that depth is inversely proportional to disparity  $d = {}^L u - {}^R u$  and  $d > 0$ .

The special floating point value `NaN` (for not a number) has the useful property that the result of any arithmetic operation involving `NaN` is always `NaN`. Many MATLAB® functions such as `max` or `min` ignore `NaN` values in the input matrix, and plotting and graphics functions do not display this value, leaving a hole in the graph or surface.

The division by 90 is to convert the floating point disparity values in the range [40, 90] into valid greyscale values in the range [0, 1].

**Fig. 14.34.**

3-dimensional reconstruction for parallel stereo cameras. Hotter colors indicate parts of the surface that are closer to the camera

We can also recover the  $X$ - and  $Y$ -coordinates

$$X = \frac{b(L_u - u_0)}{d}, \quad Y = \frac{b(L_v - v_0)}{d}$$

The images shown in Fig. 14.20, from the Middlebury dataset, were taken with a very wide camera baseline. The left edge of the left-image and the right edge of the right-image have no overlap and have been cropped. Cropping  $N$  pixels from the left of the left-hand image reduces the disparity by  $N$ . For this stereo pair the actual disparity must be increased by 274 to account for the cropping.

The true disparity is

```
>> di = di + 274;
```

and we compute the  $X$ -,  $Y$ - and  $Z$ -coordinate of each pixel as separate matrices to exploit MATLAB's efficient matrix operations

```
>> [U,V] = imeshgrid(L);
>> u0 = size(L,2)/2; v0 = size(L,1)/2;
>> b = 0.160;
>> X = b*(U-u0) ./ di; Y = b*(V-v0) ./ di; Z = 3740 * b ./ di;
```

which can be displayed as a surface

```
>> surf(Z)
>> shading interp; view(-150, 75)
>> set(gca,'ZDir', 'reverse'); set(gca,'XDir', 'reverse')
>> colormap(flipud(hot))
```

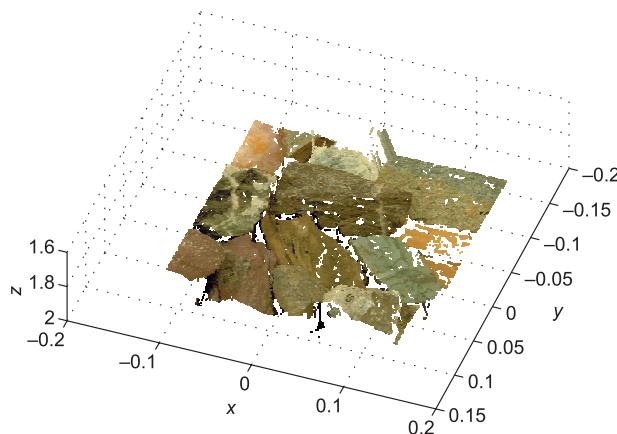
as shown in Fig. 14.34. This is somewhat unimpressive in print but by using the mouse to rotate the image using the MATLAB® figure toolbar *3D rotate* option the 3-dimensionality becomes quite clear. The axis reversals are required to have  $z$  increase from our viewpoint and to maintain a right-handed coordinate frame. There are many *holes* in this surface which are the `NaN` values we inserted to indicate unreliable disparity values.

### 14.3.5 3D Texture Mapped Display

For human, rather than robot, consumption it would be nice to enhance the surface representation so that it looks less ragged. We create a median filtered image

```
>> dimf = irank(di, 41, ones(9,9));
```

A process known as **vectorizing**. Using matrix and vector operations instead of **for** loops greatly increases the speed of MATLAB® code execution. See <http://www.mathworks.com/support/tech-notes/1100/1109.html> for details.



**Fig. 14.35.**  
3-dimensional reconstruction  
for parallel stereo cameras with  
image texture mapped onto the  
surface

where each output pixel is the median value over a  $9 \times 9$  window. This has *patched* many of the smaller holes but has the undesirable side effect of blurring the underlying disparity image. Instead we will keep the original interpolated disparity image and insert the median filtered values only where a `NaN` exists

```
>> di = ipixswitch(isnan(di), dimf, di);
```

We perform the reconstruction again

```
>> X = b*(U-u0) ./ di; Y = b*(V-v0) ./ di; Z = 3740 * b ./ di;
```

and plotting this as a surface displays a surface that looks significantly less ragged.

However we can do even better. We can *drape* the left-hand image over the 3-dimensional surface using a process called texture mapping. We reload the left-hand image, this time in color

```
>> Lcolor = iread('rocks2-1.png');
```

and render the surface with the image texture mapped

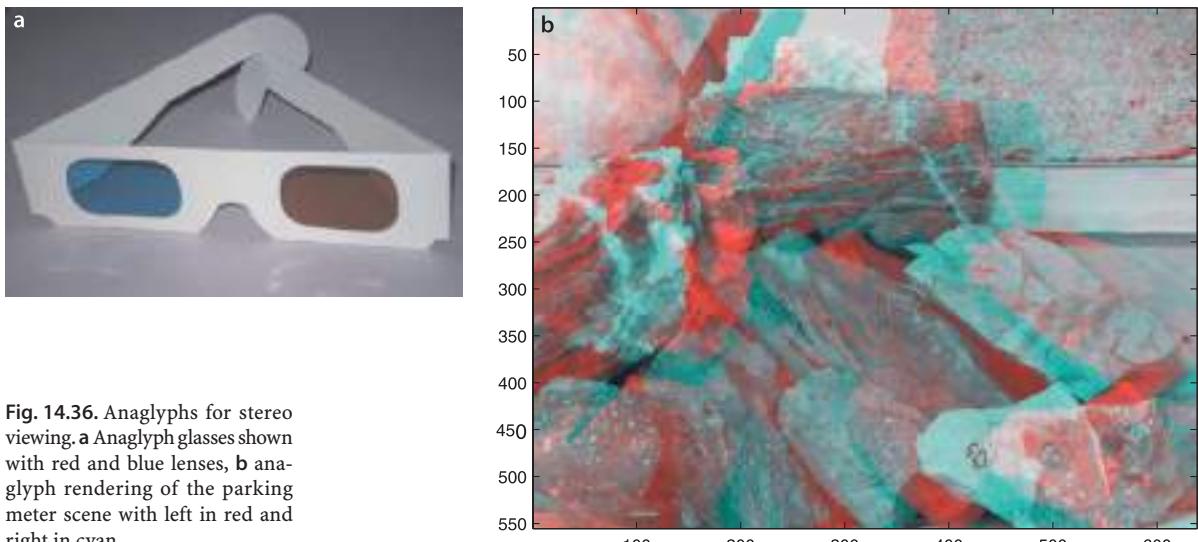
```
>> surface(X, Y, Z, Lcolor, 'FaceColor', 'texturemap', ...
    'EdgeColor', 'none', 'CDataMapping', 'direct')
>> xyzlabel
>> set(gca,'ZDir', 'reverse'); set(gca,'XDir', 'reverse')
>> view(-160, 75)
```

which creates the image shown in Fig. 14.35. Once again it is easier to get an impression of the 3-dimensionality by using the mouse to rotate the image using the MATLAB® figure toolbar *3D rotate* option.

### 14.3.6 Anaglyphs

Human stereo perception of depth works because each eye views the scene from a different viewpoint. If we look at a photograph of a 3D scene we still get a perception of depth, albeit reduced, because our brain uses many visual cues besides stereo to infer depth. Since the invention of photography in the 19<sup>th</sup> century people have been fascinated by 3D photographs and movies, and the recent introduction of 3D television is further evidence of this.

The key in all 3D display technologies to take the image from two cameras, with a similar baseline to the human eyes (approximately 8 cm) and present those images again to the corresponding eyes. Old fashioned stereograms required a binocular viewing device or could, with difficulty, be viewed by squinting at the stereo pair and crossing your eyes. More modern and convenient means of viewing stereo pairs are LCD shutter glasses or polarized glasses which allow full-color stereo movie viewing.



**Fig. 14.36.** Anaglyphs for stereo viewing. **a** Anaglyph glasses shown with red and blue lenses, **b** anaglyph rendering of the parking meter scene with left in red and right in cyan

**Anaglyphs.** The earliest developments occurred in France. In 1858 Joseph D'Almeida projected 3D magic lantern slide shows as red-blue anaglyphs and the audience wore red and blue goggles. Around the same time Louis Du Hauron created the first printed anaglyphs. Later, around 1890 William Friese-Green created the first 3D anaglyphic motion pictures using a camera with two lenses. Anaglyphic films called plasticons or plastigrams were a craze in the 1920s.

A century later stereo images from the surface of Mars are available on the web in anaglyph form at <http://marsprogram.jpl.nasa.gov/MPF/mpf/anaglyph-arc.html>.

An old but inexpensive method of viewing and distributing stereo information is through anaglyph images in which the left and right images are overlaid in different colors. Typically red is used for the left eye and cyan (greenish blue) for the right eye but many other color combinations are used. The red lens allows only the red part of the anaglyph image through to the left eye, while the cyan lens allows only the cyan parts of the image through to the right eye. The disadvantage is that only the scene intensity, not its color, can be portrayed. The big advantage of anaglyphs is that they can be printed on paper or imaged onto ordinary movie film and viewed with simple and cheap glasses such as those shown in Fig. 14.36a.

The rock pile stereo pair can be displayed as an anaglyph

```
>> anaglyph(L, R, 'rc')
```

which is shown in Fig. 14.36b. The argument '`rc`' indicates that left and right images are encoded in `red` and `cyan` respectively. Other color options include: `blue`, `green`, `magenta` and `orange`.

### 14.3.7 Image Rectification

The rock pile stereo pair of Fig. 14.20 has corresponding points on the same row in the left- and right-hand images. These are referred to as an epipolar-aligned image pair. Stereo cameras such as shown on page 405 are built with precision to ensure that the optical axes of the cameras are parallel and that the  $u$ - and  $v$ -axes of the two sensor chips are parallel. However there are limits to the precision of mechanical alignment and lens distortion will introduce error. Typically one or both images are warped to correct for these errors – a process known as rectification.

We will illustrate rectification using the garden stereo pair shown from Fig. 14.15

```
>> L = iread('garden-l.jpg', 'mono', 'double');
>> R = iread('garden-r.jpg', 'mono', 'double');
```

which we recall are far from being epipolar aligned. We first find the SURF features

```
>> sL = isurf(L);
>> sR = isurf(R);
```

and determine the candidate matches

```
>> m = sL.match(sR);
```

then determine the epipolar relationship

```
>> F = m.ransac(@fmatrix, 1e-4, 'verbose');
46 trials
81 outliers
9.60297e-05 final residual
```

The rectification step requires the fundamental matrix as well as the set of corresponding points which is embedded in the `FeatureMatch` object `m`

```
>> [Lr, Rr] = irectify(F, m, L, R);
```

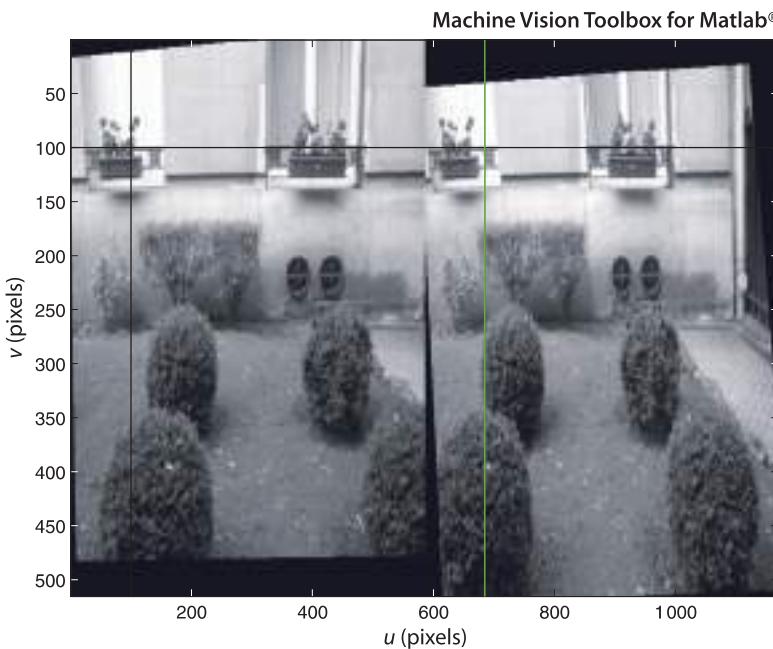
and returns rectified versions of the two input images. We display these using `stdisp`

```
>> stdisp(Lr, Rr)
```

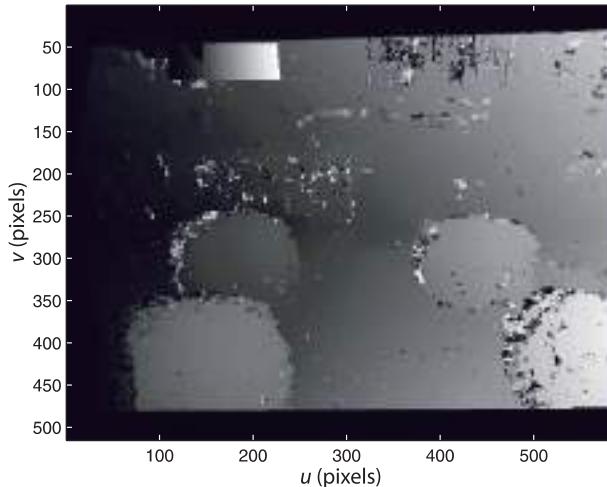
which is shown in Fig. 14.37. We see that corresponding points in the scene now have the same vertical coordinate. The function `irectify` works by computing unique homographies to warp the left and right images. As we have observed previously when warping images not all of the output pixels are mapped to the input images which results in undefined pixels which are displayed here as black. This pair of rectified images could now be used for dense stereo matching

```
>> d = istereo(Lr, Rr, [5 120], 4);
```

and the result is shown in Fig. 14.38. We have been able to create a dense 3-dimensional representation of the scene using just two images taken from a handheld camera.



**Fig. 14.37.**  
Rectified images of the garden

**Fig. 14.38.**

Dense stereo disparity image for the garden shows near objects as brighter than far objects

### 14.3.8 Plane Fitting

Stereo vision results in a set of 3-dimensional world points  $P_i$ , which are referred to as a point cloud. A common problem is fitting a plane to such a set of points. One simple and effective approach is to fit an ellipsoid to the data, and the ellipsoid will have one very small radius in the direction normal to the plane – that is, it will be an elliptical plate. The inertia matrix of the points can be calculated by

$$J = \sum_{i=1}^N \mathbf{x}_i \mathbf{x}_i^T \quad (14.16)$$

where  $\mathbf{x} = P_i - \bar{P}$  are the coordinates of the points with respect to the centroid of the points  $\bar{P} = \frac{1}{N} \sum_{i=1}^N P_i$ . The ellipsoid is centred at the centroid of the point cloud. The radii of the ellipsoid are the eigenvalues of  $J$  and the eigenvector corresponding to the smallest eigenvalue is the direction of the minimum radius which is the normal to the plane.

To illustrate this we create a  $10 \times 10$  grid of points in a plane

```
>> T = transl(1,2,3) * rpy2tr(0.3, 0.4, 0.5);
>> P = mkgrid(10, 1, T);
>> P = P + 0.02*randn(size(P));
```

with an arbitrary orientation represented by the homogeneous transformation  $T$ , and to which some Gaussian noise has been added.

The mean of the point cloud is

```
>> x0 = mean(P')
ans =
0.9980    1.9979    2.9973
```

and this is subtracted from all the data points

```
>> P = bsxfun(@minus, P, x0');
```

and then the moments are computed

```
>> J = 0;
>> for x = P, J = J + x*x'; end
>> J
J =
8.4143    1.0722   -3.3781
1.0722    9.6187    2.5577
-3.3781    2.5577    2.3441
```

The eigenvalues are

```
>> [x,lambda] = eig(J);
>> diag(lambda)'
ans =
    0.0357    9.9135   10.4279
```

and we see two large eigenvalues corresponding to the spread of points within the plane, and one eigenvalue which is the *thickness* of the plane. The eigenvector corresponding to the first, and smallest, eigenvalue is

```
>> n = x(:,1)';
n =
    0.3896   -0.2779    0.8781
```

which is the estimated normal to the plane.

The true direction of the plane's normal is given by the third column<sup>►</sup> of the transformation matrix

```
>> T
T =
    0.8083   -0.4416    0.3894    1.0000
    0.5590    0.7832   -0.2722    2.0000
   -0.1848    0.4377    0.8799    3.0000
        0         0         0    1.0000
```

and we see that it is very close to the estimated normal.

The equation of a plane is the set of points  $x$  such that

$$\mathbf{n}^T(\mathbf{x} - \mathbf{x}_0) = 0 \quad (14.17)$$

where  $\mathbf{n}$  is the normal and  $\mathbf{x}_0$  is the centroid.

Outlier data points are problematic with this type of estimator since they significantly bias the solution. A number of approaches are commonly used but a simple one is to modify Eq. 14.16 to include a weight

$$J = \sum_{i=1}^N w_i \mathbf{x}_i \mathbf{x}_i^T$$

which is inversely related to the distance of  $\mathbf{x}_i$  from the plane and solve iteratively. Initially all weights  $w_i = 1$ , and on subsequent iterations the weights<sup>►</sup> are set according to the distance of  $\mathbf{P}_i$  from the plane estimated at the previous step. Alternatively we could apply RANSAC by taking samples of three points to solve for Eq. 14.17. Appendix E has more details about ellipses.

Since the points lie in the frame's  $xy$ -plane, the normal is the frame's  $z$ -axis.

Commonly a Cauchy-Lorentz function  $w = r^2 / (x^2 + r^2)$  is used which is smooth over the range of point distance  $0 \rightarrow \infty$  and has a value of  $1/2$  when  $x = r$ .

### 14.3.9 Matching Sets of 3D Points

Consider a model of some object represented by a set of points in 2- or 3-dimensions with respect to the world frame. Now consider an example of that object somewhere in the world and we observe a set of 2- or 3-dimensional points on the object. The task is to determine the relative pose  $\xi$  that will transform the model points to the observed data points by matching the two sets of points.<sup>►</sup>

More formally, given two sets of points  $\mathbf{M}_i, \mathbf{D}_j \in \mathbb{R}^3, i, j \in 1 \dots N$  in the world frame determine the relative pose  $\xi$  such that

$$\mathbf{D}_i = \xi \cdot \mathbf{M}_i$$

The points  $\mathbf{M}_i$  are the *model* of a 3-dimensional object which we want to fit to the observed *data*  $\mathbf{D}_j$ .

To illustrate we will create a model which is a cloud of 100 random points in a  $1 \times 1 \times 1$  m cube centred at the origin<sup>►</sup>

The dual problem is that the camera has moved, not the object. The same technique can be applied to determine the camera motion.

The technique can work for 2- or 3-dimensional data but we will illustrate it here for 3-dimensional data.

```
>> M = rand(3,100) - 0.5;
```

The data is a copy of the model that has been rotated and translated

```
>> T_unknown = transl(1, 2, 3) * rpy2tr(0.2, 0.3, 0.4);
>> D = homtrans(T_unknown, M);
```

by an unknown relative pose  $\xi$ .

At first glance this looks like a problem where we need to establish correspondence between the points in the two sets but we will introduce an alternative approach called iterative closest point or ICP. For each data point  $D_i$ , the corresponding model point  $M_j$  is assumed to be the closest one, that is  $M_j$  which minimizes  $|D_i - M_j|$ . Correspondence is not unique and quite commonly several data points can be associated with a single model point, and consequently some data points  $D_j$  will be unpaired. This approach to correspondence is far from perfect but it is (generally) good enough to *improve* the alignment of the point clouds so that in the next iteration the computed correspondences will be a little more accurate.

The first step is to compute a translation that makes the centroids of the two point clouds coincident

$$\bar{M} = \frac{1}{N_M} \sum_{i=1}^{N_M} M_i$$

$$\bar{D} = \frac{1}{N_D} \sum_{i=1}^{N_D} D_i$$

from which we compute a displacement

$$t = \bar{D} - \bar{M}$$

Next we compute correspondence. For each data point  $D_i$  we find the closest model point  $M_j$ , and for this we use the Toolbox function `closest`

```
>> corresp = closest(D, M);
```

where `corresp(i)` is the column of `M` that corresponds to column `i` of `D`. The next step is to compute the  $3 \times 3$  moment matrix

$$W = \sum (M_i - \bar{M})(D_i - \bar{D})^T$$

This is the sum of a number of rank 1 matrices.

which encodes the rotation between the two point sets. ▶ The singular value decomposition is

$$W = U \Sigma V^T$$

from which the rotation matrix is determined to be

$$R = VU^T$$

The estimated relative pose between the two point clouds is  $\xi \sim (R, t)$ . The model points are transformed so that they are now closer to the data points

$$M_i^{(k+1)} = \xi^{(k)} \cdot M_i^{(k)}, i = 1 \dots N_M$$

and the process repeated until it converges. The correspondences used are unlikely to have all been correct and therefore the estimate of the relative orientation between the sets is only an approximation.

The Toolbox provides an implementation of ICP

```
>> [T,d] = icp(M, D, 'plot');
```

which returns the pose  $\xi$

```
>> trprint(T, 'rpy', 'radian')
t=(1 2 3),R=(0.2 0.3 0.4) rad
```

which is the exactly the unknown relative pose of the second point cloud that we chose above. The residual

```
>> d
d =
6.8437e-08
```

is the root mean square of the errors between the transformed model points and the data. The option '`plot`' shows the model and data points at each step as well as the closest-point correspondences. ICP is a popular algorithm because it is both fast and robust.

We can demonstrate the robustness of ICP by simulating some realistic sensor errors. Firstly we will randomly remove twenty points from the data

```
>> D(:,randi(100, 20,1)) = [];
```

which are points in the model not observed by the sensor. Then we will add ten spurious points that are not part of the model

```
>> D = [D rand(3,10)];
```

and finally we will add Gaussian noise with  $\sigma = 0.05$  to the data

```
>> D = D + randn(size(D)) * 0.05;
```

Now we fit the corrupted data to the model

```
>> [T,d] = icp(M, D, 'plot', 'distthresh', 1.5);
```

using an additional option to eliminate incorrect closest-point correspondences. The correspondences are established as described above and the median of the distances between the corresponding points is computed. In this case the correspondence is not made if the distance between the points is more than 1.5 times the median distance. The estimated pose  $\xi$  is now

```
>> trprint(T, 'rpy', 'radian')
t = (1.02161,1.98456,2.98566), R = (0.164436,0.299455,0.418697) rad
```

which is still close to the value computed for the ideal case but the residual

```
>> d
d =
0.6780
```

is higher since an exact fit between the model and noise corrupted data is no longer possible. ▶ ICP is fast and robust for modest sized point clouds but the correspondence determination is an  $O(N^2)$  problem which leads to computational bottlenecks for very large data sets. ▶

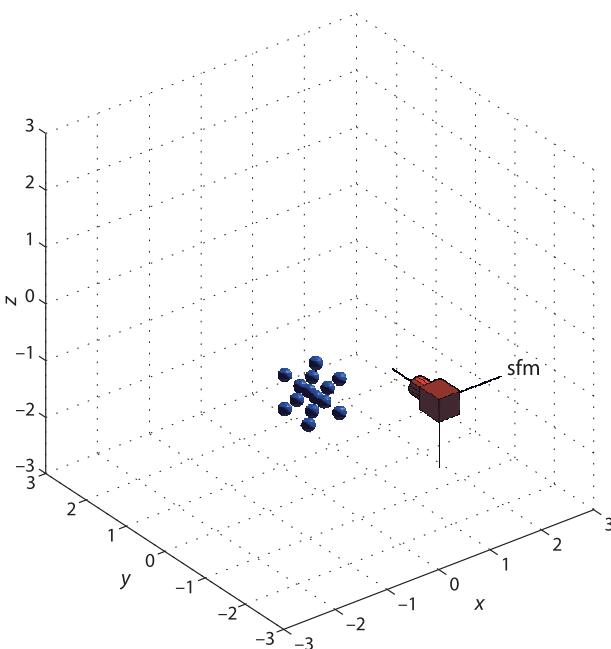
We would expect the residual to be approximately equal to  $\sqrt{N}\sigma$  where  $N$  is the number of corresponding points and  $\sigma$  is the standard deviation of the additive noise.

For large-scale problems the data would be kept in a *kd-tree* which reduces the time required to find the closest point.

## 14.4 Structure and Motion

In the sparse stereo example of Sect. 14.3.1 we estimated the *motion* of the camera, its change in pose, from corresponding image points. Using this as a baseline we were then able to estimate the 3-dimensional position of some points in the world, the *structure* of the scene.

For robotic applications we would like to perform these operations on the fly, that is, as each new observation is made we wish to update the estimates of the robot's pose and the structure of its world. We will illustrate this with an example using a sequence of image from a single moving camera.

**Fig. 14.39.**

Structure from motion example showing camera and world points

We create a central perspective camera

```
>> camera = CentralCamera('name', 'sfm', 'default');
```

and some points positioned at the vertices and face centres of a cube

```
>> cube = mkcube(0.6, 'facepoint')
```

A subsequent step estimates the fundamental matrix which requires at least eight points. A cube with only eight points leads to poor estimates of the fundamental matrix, hence the addition of six extra points in the centre of each face.

a total of 14 points. ▶ The camera moves in a circle of radius 3 m around the cube in the  $xy$ -plane while always facing the origin

```
>> nsteps = 50;
>> theta = tpoly(0, 2*pi, nsteps);
>> T = [];
>> for th=theta'
>> T = cat(3, T, trotz(-th) * transl(0, -3, 0) * trotx(-pi/2));
>> end
```

where  $T$  is the path, a sequence of camera poses, represented by a  $4 \times 4 \times 50$  matrix where the first two indices represent a homogeneous transformation and the last index corresponds to the camera viewpoint.

The image plane projection is

```
>> p = camera.plot(cube, 'Tcam', T);
```

which returns a  $2 \times 14 \times 50$  matrix where the second index corresponds to the world point, and the last index corresponds to the camera view.

At each step along the path we perform the following algorithm, illustrated here for step twenty

```
>> k = 20;
```

of the path. The corresponding points between the current and previous image are used to estimate the fundamental matrix

```
>> F = fmatrix( p(:,:,k-1), p(:,:,k) );
```

and then *upgrade* it to an essential matrix

```
>> E = camera.E(F);
```

and decompose that to an estimate of the motion from the last camera pose to the current

```
>> relpose = camera.invE(E, [0 0 10]');
solution 2 is good
relpose =
    0.9775   -0.0000    0.2108   -11.2645
    0.0000    1.0000    0.0000   -0.0000
   -0.2108   -0.0000    0.9775    1.2008
        0         0         0     1.0000
```

where we use a world point along the optical axis to determine which of the two solutions is valid.

The true relative pose determined from the known camera trajectory is

```
>> inv(T(:,:,k-1))*T(:,:,k)
ans =
    0.9775   0.0000    0.2108   -0.6324
   -0.0000    1.0000   -0.0000    0.0000
   -0.2108   -0.0000    0.9775    0.0674
        0         0         0     1.0000
```

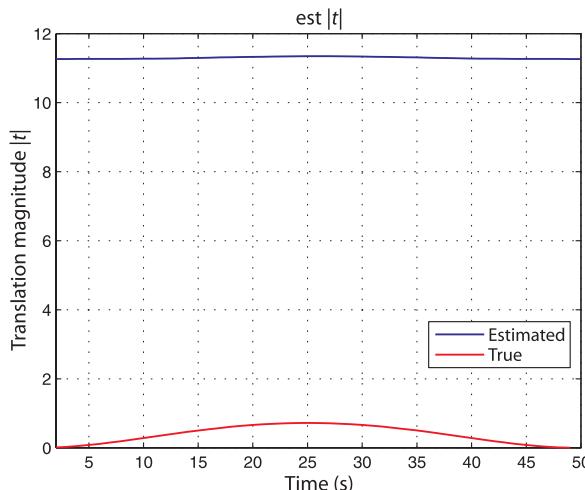
The rotational part of the estimated pose is accurate but the translational part suffers from the unknown scale factor problem that we have discussed previously. Figure 14.40 shows the magnitude of the estimated and true translation versus time. We see that the translation is always overestimated and that there is no obvious relationship to the true translation magnitude. ▶

Without knowledge of the scale factor we are somewhat stuck – we cannot estimate the incremental motion between views. As we have seen before the solution involves incorporating other sources of information. For example other sensors such as odometry or GPS can provide an estimate for the magnitude of the translational motion. The odometer is only required to provide the distance travelled, since the rotational component of the motion is determined without ambiguity. Alternatively we could use a stereo camera to provide the depth information directly and the problem at each time step is then determining the relative pose that aligns the 3D point clouds from the current and previous time step, which is can be solved using ICP. Another option, and the one we consider next is to consider that we know the height above ground of just one observed world point. In a robotic scenario we might know that the robot moves on a plane and that a particular feature point lies on the ground or the top of a doorway.

For this example we will assume that we know the height of point  $j$  and we arbitrarily choose

```
>> j = 1;
```

The true translation magnitude shows the smoothly changing velocity created by the `tpoly` function.



**Fig. 14.40.**  
Magnitude of camera translational motion at each time step as estimated from the essential matrix and the ground truth

and the point's world coordinate is

```
>> P(:,j)'
ans =
-0.3000 -0.3000 -0.3000
```

Since the camera lies in the *world* *xy*-plane with its *y*-axis pointing downward, then  ${}^C Y_j = 0.3$

```
>> Yj = 0.3;
```

This is the *only* extra piece of information that we need but it does require that this particular point can be found in all future images.

We will again illustrate the algorithm for step twenty of the path. We consider that {1} is the camera frame in the previous time step and frame {2} the current time step. A ray corresponding to the projection of the known point in {1} is

```
>> r1 = camera.ray( p(:,j,k-1) )
r1 =
d=(0.117002, 0.108684, 0.987167), p0=(0, 0, 0)
```

In vector form the plane is  $(0, 1, 0)$   
 $(X, Y, Z)^T = {}^C Y_j$  or in homogeneous  
form  $(0, 1, 0, -{}^C Y_j)(X, Y, Z, 1)^T = 0$ .

which intersects the plane  ${}^C Y = {}^C Y_j$  at the point $\blacktriangleleft$

```
>> P1 = r1.intersect([0 1 0 -Yj])
P1 =
0.3230
0.3000
2.7249
```

The projection of this world point in {2} which we observe with the camera is given by Eq. 11.5

$${}^2 \tilde{p} = C {}^2 T_1^{-1} \tilde{P}$$

which we can write in terms of an explicit homogeneous scale factor  $\lambda$

$$\begin{aligned} \lambda^2 \tilde{p} &= (\Pi \quad c_4) \begin{pmatrix} R & \sigma t \\ 0_{1 \times 3} & 1 \end{pmatrix} \begin{pmatrix} P \\ 1 \end{pmatrix} \\ &= \sigma \Pi t + \Pi R P + c_4 \\ &= \sigma a + b \end{aligned}$$

where  $\sigma$  is the unknown translational scale factor and  $a$  and  $b$  are vectors that are a function of the known camera matrix  $C$  and the estimated camera rotation  $R$ . The difference between the observed and actual image plane coordinate is known as reprojection error and its magnitude is

$$e = (\sigma a + b - \lambda^2 \tilde{p})^T (\sigma a + b - \lambda^2 \tilde{p})$$

To determine values of the translation scale factor  $\sigma$  and the homogeneous scale factor  $\lambda$  we minimize this error by taking the partial derivatives and setting them to zero

$$\frac{\partial e}{\partial \sigma} = 2(\sigma a + b - \lambda^2 \tilde{p}) \cdot a = 0$$

$$\frac{\partial e}{\partial \alpha} = -2(\sigma a + b - \lambda^2 \tilde{p}) \cdot {}^2 \tilde{p} = 0$$

which implies that the vector term in parentheses is orthogonal to both  $a$  and  ${}^2 \tilde{p}$ . This is the same as being proportional to their cross product

$$\sigma a + b - \lambda^2 \tilde{p} \propto a \times {}^2 \tilde{p}$$

and by introducing another scale factor  $\alpha$  we write this as an equality

$$\sigma a + b - \lambda^2 \tilde{p} = \alpha(a \times {}^2 \tilde{p})$$

that we rearrange as

$$(-\mathbf{a}^T \tilde{\mathbf{p}} - \mathbf{a} \times \tilde{\mathbf{p}}) \begin{pmatrix} \sigma \\ \lambda \\ \alpha \end{pmatrix} = \mathbf{b}$$

which is a linear equation that can be solved for the unknown parameters  $\sigma$ ,  $\lambda$  and  $\alpha$ . For this example

```
>> C = cam.C; PI = C(:,1:3); c4 = C(:,4);
>> [R,t] = tr2rt( inv(relpose) )
>> a = PI*t; b = PI*R*P1+c4;
>> p2 = e2h( p(:,j,k) );
```

and the parameters are estimated

```
>> phi = [-a p2 cross(a, p2)] \ b
phi =
    0.0561
    2.7991
    0
```

from which the displacement  ${}^1\xi_2$  can be computed

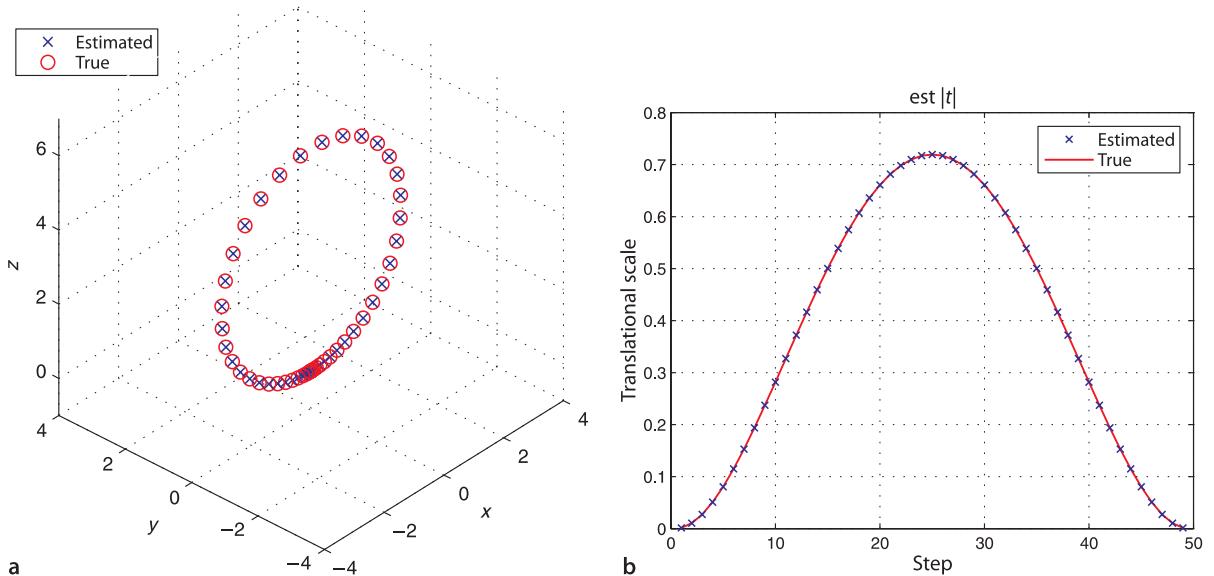
```
>> dT = inv( [R phi(1)*t; 0 0 0 1] )
dT =
    0.9775    0.0000    0.2108   -0.6324
   -0.0000    1.0000   -0.0000    0.0000
   -0.2108    0.0000    0.9775    0.0674
     0         0         0    1.0000
```

This is the same, to four significant figures, as the known camera motion shown earlier. The missing scale factor has been recovered using the known height above ground of a single observed point, known robot motion and a calibrated camera. The pose of the camera in the world frame is estimated by accumulating the frame-to-frame pose changes

$$\hat{T}(k) = \hat{T}(k-1) {}^1\hat{T}_2(k-1)$$

to obtain an estimate of the camera's pose at each time step with respect to the initial camera pose. Figure 14.41 shows the true and estimated position of camera's centre

**Fig. 14.41.** Structure from a motion with estimated scale from a single world point. **a** Estimated and true camera centre; **b** estimated and true translational scale factor



along the path as well as the estimated scale factor. This is a dead-reckoning approach to position estimation and, as we discussed in Sect. 6.1, the error in the estimate of each incremental motion will accumulate over time. The results for the case with noise added to the feature coordinates is shown in Fig. 14.42 and we see that the scale estimate is less accurate and that the dead-reckoned camera position has diverged from the true position. The full code for this example is available as `sfm1` in the examples directory.

Next we will estimate the structure of the scene with the sparse stereo approach we used previously in Sect. 14.3.1. The rays in 3-dimensional space corresponding to the projected points at the previous position are

```
>> r1 = camera.ray(p(:,:,k-1))
```

and at the current position they are

```
>> r2 = camera.move(dT).ray(p(:,:,k))
```

The rays intersect at

```
>> x = r1.intersect(r2)
```

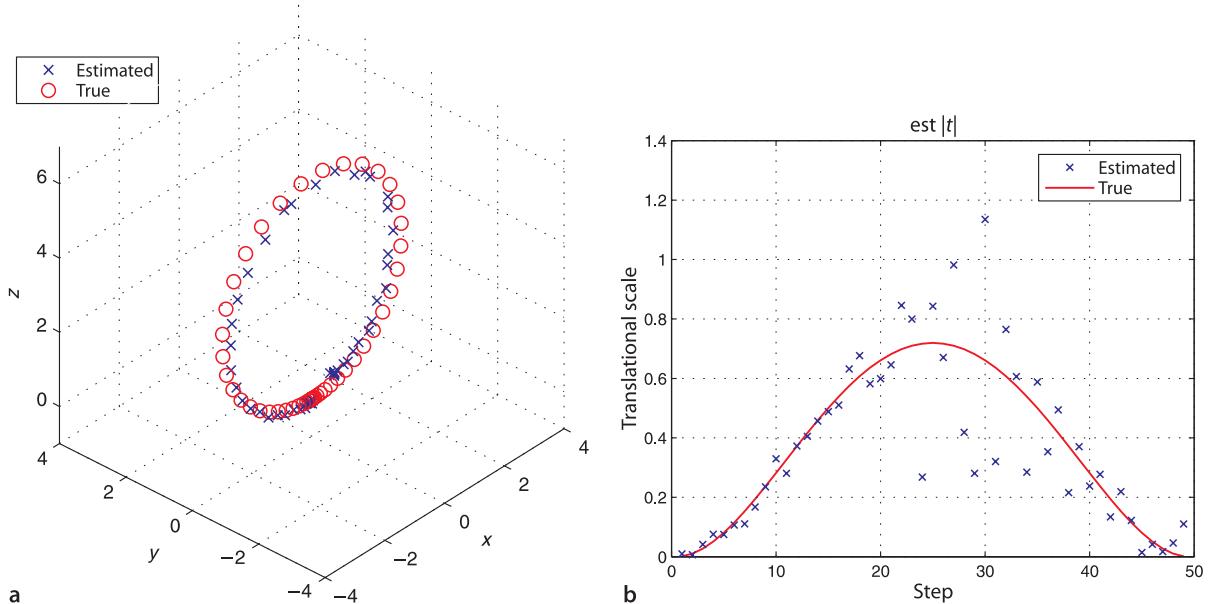
which is a matrix where each column is the 3-dimensional location of the point with respect to the previous camera pose. Using our estimate of the camera pose with respect to the world these points can be transformed into the world coordinate frame

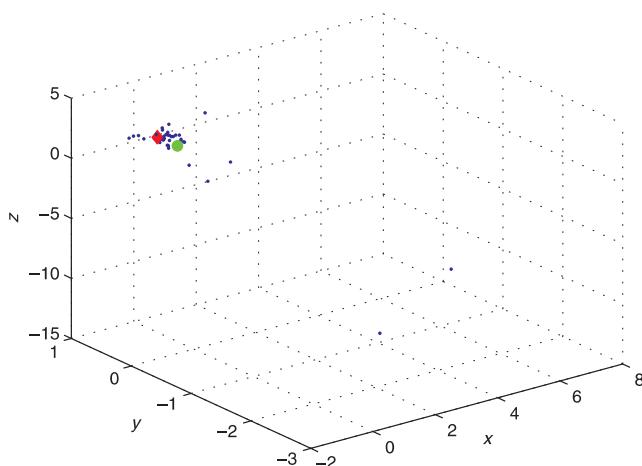
```
>> xc = homtrans(inv(T_est(:,:,k-1)), x)
```

This approach is simple but has a number of drawbacks. The biggest problem is that the 3-dimensional coordinates of every world point is estimated at every time step, and the quality of the estimates depends strongly on the relative motion between the camera and the world point. The triangulation involves three points, the two camera centres and the world point as shown in Fig. 14.17. The side of the triangle between the camera centres is the baseline, and if the baseline is small the estimates will be susceptible to noise. This will be the case here since the camera motion between images is small. However if the second camera centre is collinear with the first camera centre and the world point – the motion is along a line toward or away from the point – there is a zero base line and we can learn nothing about that point’s depth.

Figure 14.42 showed the effect of image noise on the estimate of camera position, and Fig. 14.43 shows the effect of this noise on the estimated location of a single world

**Fig. 14.42.** Structure from motion with estimated scale from single world point and pixel noise  $\sigma = 0.5$ . **a** Estimated and true camera centre; **b** estimated and true translational scale factor





**Fig. 14.43.**  
Estimated spread, pixel noise  $\sigma = 0.5$ . Blue dots are the pairwise estimated point position, red  $\diamond$ -marker is the true location of the world point (in camera frame), and the green circle is the mean of the estimated positions

point. We see that the estimates are scattered with some significant outliers that are in error by more than 1 m. One solution is to increase the effective baseline by triangulating between every  $N$  frames instead of between every consecutive frame. Another option is to apply some low-pass filter $\blacktriangleright$  to the estimates, and Fig. 14.43 shows that the mean of the estimates is close to the true position. $\blacktriangleright$  A better solution is to invert the problem and use an extended Kalman filter as we did for the localization problem of Sect. 6.1.2. $\blacktriangleright$  The filter's state comprises the pose of the camera and the world coordinates of landmarks. The observations are the pixel coordinates of the features. The difference between the observed and predicted pixel coordinates is the innovation which updates the state vector via the Kalman gain matrix. This framework allows for improved estimates of the world point locations since it incorporates measurements over many time steps and allows for explicit modeling of the uncertainty associated with the world points. The Kalman filter framework also allows for the incorporation of any number of other sensors such as odometry, GPS or laser range measurements which can help to resolve the scale problem. Some book keeping is required to keep track of features as they enter the camera view and later leave.

Such as a moving average filter, a Kalman filter or an  $\alpha-\beta$  tracking filter.

The errors do not appear to be zero-mean so a mean estimate would be biased.

The Kalman filter assumes that the noise is zero-mean and Gaussian, which is unlikely to be the case here.

## 14.5 Application: Perspective Correction

Consider the image

```
>> im = imread('notre-dame.jpg', 'double');
>> idisp(im)
```

shown in Fig. 14.44. The shape of the building is significantly distorted because the camera's optical axis was not normal to the plane of the building and we see evidence of perspective foreshortening or keystone distortion. We manually pick four points, clockwise from the bottom left, that are the corners of a large rectangle on the planar face of the building

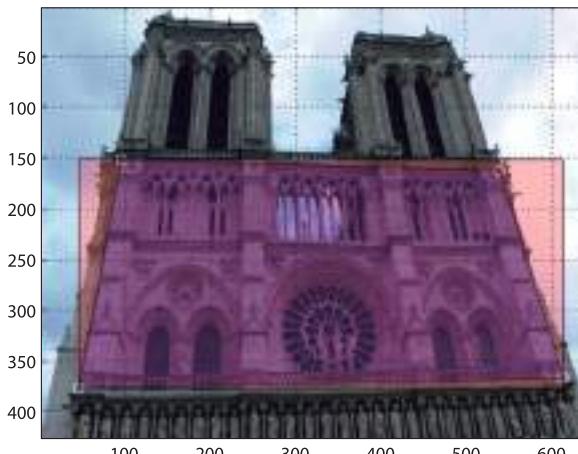
```
>> p1 = ginput(4)';
ans =
    44.1364    94.0065   537.8506   611.8247
   377.0654   152.7850   163.4019   366.4486
```

which has one column per point that contains the  $u$ - and  $v$ -coordinate. We overlay this on the image of the cathedral

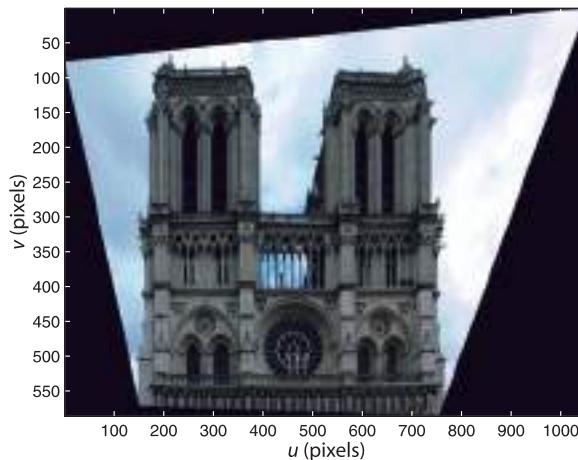
```
>> plot_poly(p1, 'wo', 'fill', 'b', 'alpha', 0.2);
```

with translucent blue fill. $\blacktriangleright$  We use the extrema of these points to define the vertices of a rectangle in the image

In computer graphics terminology alpha is the transparency of a surface, and varies from 0 completely transparent to 1 which is opaque.

**Fig. 14.44.**

Photograph taken from the ground shows the effect of foreshortening which gives the building a trapezoidal appearance (also known as keystone distortion). Four points on the approximately planar face of the building have been manually picked as indicated by the white O-markers (Notre Dame de Paris)

**Fig. 14.45.**

A fronto-parallel view synthesized from Fig. 14.44. The image has been transformed so that the marked points become the corners of a rectangle in the image

```
>> mn = min(p1');
>> mx = max(p1');
>> p2 = [mn(1) mx(2); mn(1) mn(2); mx(1) mn(2); mx(1) mx(2)]';
```

which we overlay on the image

```
>> plot_poly(p2, 'k', 'fill', 'r', 'alpha', 0.2)
```

in red.

The sets of points `p1` and `p2` are projections of world points that lie approximately in a plane so we can compute an homography

```
>> H = homography(p1, p2)
H =
    1.4003    0.3827   -136.5900
   -0.0785    1.8049   -83.1054
   -0.0003    0.0016    1.0000
```

An homography can also be computed from four lines in the plane, but this is not supported by the Toolbox.

that will transform the vertices of the blue trapezoid to the vertices of the red rectangle. ▶

$$\tilde{p}_2 \approx H\tilde{p}_1$$

That is, the homography maps image coordinates from the distorted keystone shape to an undistorted rectangular shape.

We can apply this homography to the coordinate of every pixel in an output image in order to warp the input image. We use the Toolbox generalized image warping function

```
>> homwarp(H, im, 'full')
```

and the result shown in Fig. 14.45 is a synthetic fronto-parallel view. This is equivalent to the view that would be seen by a camera high in the air with its optical axis normal to the face of the cathedral. However points that are not in the plane, such as the left-hand side of the right bell tower have been distorted. The black pixels in the output image are due to the corresponding pixel coordinates not being present in the input image. Note that with no output argument specified the warped image is displayed using `idisp`.

In addition to creating this synthetic view we can decompose the homography to recover the camera motion from the actual to the virtual viewpoint and also the surface normal of the cathedral. As we saw in Sect. 14.2.4 we need to determine the camera calibration matrix so that we can convert the projective homography into a Euclidean homography. We obtain the focal length from the header of the EXIF-format file that holds the image

```
>> [im,tags] = iread('notre-dame.jpg', 'double');
>> tags.DigitalCamera
ans =
    ExposureTime: 0.0031
    FNumber: 5.6000
    .
    .
    .
    FocalLength: 7.4000
    .
```

and the pixel dimensions are the same as for the example on page 402. We create a calibrated camera

```

>> cam = CentralCamera('image', im, 'focal', 7.4e-3, ...
    'sensor', [7.18e-3, 5.32e-3])
cam =
name: noname [central-perspective]
focal length: 0.0074
pixel size: (5.609e-06, 6.251e-06)
principal pt: (640, 425.5)
number pixels: 1280 x 851
Tcam:
    1  0  0  0
    0  1  0  0
    0  0  1  0
    0  0  0  1

```

Now we use the camera model to compute and decompose the Euclidean homography.

```

>> sol = cam.invH(H, 'verbose');
solution 1
    T =  0.98415   -0.15470    0.08667   0.23438
          0.15902    0.98623   -0.04534   0.92578
        -0.07846    0.05841    0.99520  -1.29375
         0.00000    0.00000    0.00000   1.00000
    n = -0.23041    0.88969    0.39416
solution 2
    T =  0.95800    0.23545    0.16369   0.07289
      -0.20987    0.18671    0.95974  -1.53146
       0.19541   -0.95378    0.22828   0.48488
         0.00000    0.00000    0.00000   1.00000
    n = -0.28109    0.11855    0.95233

```

which returns a structure array of two possible solutions. The coordinate frames for this example are sketched in Fig. 14.46 and shows the actual and virtual camera poses. In this case the second solution is the correct one since it represents considerable rotation about the  $x$ -axis. The camera translation vector, which is not to scale but has the

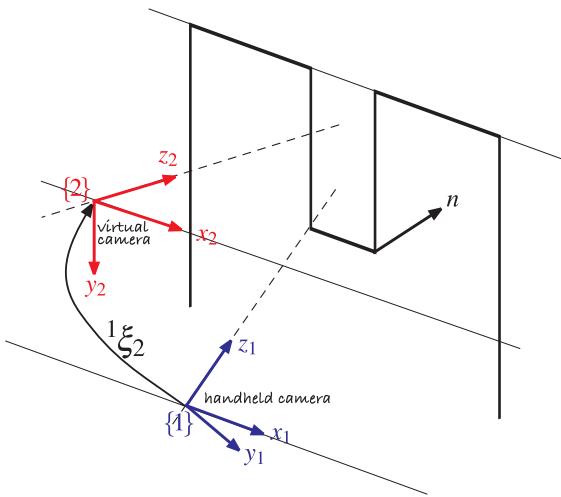


Fig. 14.46.

Notre-Dame example showing the two camera coordinate frames. The blue frame  $\{1\}$  is that of the camera that took the image, and the red frame  $\{2\}$  is the viewpoint for the synthetic fronto-parallel view

See Malis and Vargas (2007).

correct sign,  $\triangleright$  is dominantly in the negative  $y$ - and positive  $z$ -direction with respect to the frame  $\{1\}$ . The rotation matrix in XYZ-angle form

```
>> tr2rpy(sol(2).T, 'deg')
ans =
-76.6202    9.4210   -13.8081
```

indicates that the camera needs to be tilted downward (roll is rotation about the camera's  $x$ -axis) by 76 degrees to achieve the attitude of the virtual camera. The normal to the frontal plane of the church  $n$  is defined with respect to  $\{1\}$  and is essentially in the camera  $z$ -direction as expected.

## 14.6 Application: Mosaicing

Mosaicing or image stitching is the process of creating a large-scale composite image from a number of overlapping images. It is commonly applied to aerial and satellite images to create a seemingly continuous single picture of the earth's surface. It can also be applied to images of the ocean floor captured from downward looking cameras on an underwater robot. The panorama generation software supplied with digital cameras is another example of mosaicing.

The input to the mosaicing process is a sequence of overlapping images.  $\blacktriangleright$  It is not necessary to know the camera calibration parameters or the pose of the camera where the images were taken – the camera can rotate arbitrarily between images and the scale can change. However for the approach that we will use the scene is assumed to be planar which is reasonable for high-altitude photography where the vertical relief  $\blacktriangleleft$  is small.

We will illustrate our discussion with a real example using the pair of images

```
>> im1 = imread('mosaic/aerial2-1.png', 'double', 'grey');
>> im2 = imread('mosaic/aerial2-2.png', 'double', 'grey');
```

which are each  $1280 \times 1024$ . We create an empty composite image that is  $2000 \times 2000$

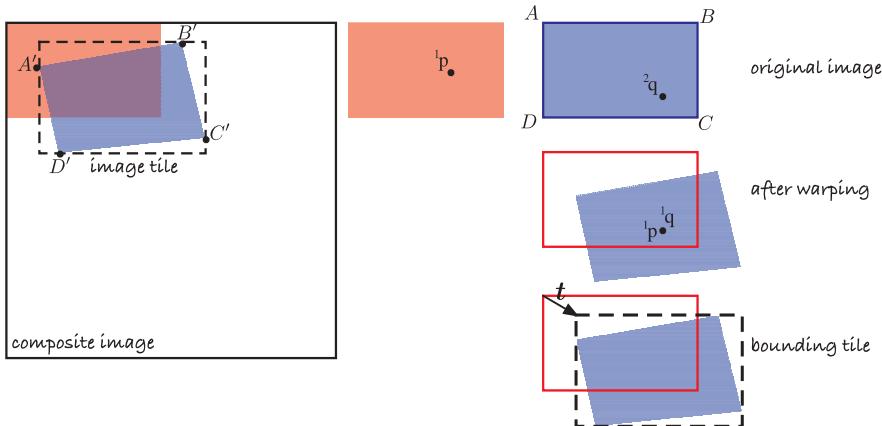
```
>> composite = zeros(2000,2000);
```

that will hold the mosaic. The essentials of the mosaicing process are shown in Fig. 14.47.

The first image is easy and we simply paste it into the top left corner

```
>> composite = ipaste(composite, im1, [1 1]);
```

of the composite image as shown in red in Fig. 14.47. The next image, shown in blue, is more complex and needs to be rotated, scaled and translated so that it correctly overlays the red image.



**Fig. 14.47.**  
The first image in the sequence is shown as red, the second as blue. The second image is warped into the image tile and then blended into the composite image

The first step is to identify common feature points which are known as tie points, and we use now familiar tools

```
>> surf = isurf(im1)
>> surf = isurf(im2)
>> m = surf.match(surf);
```

and then RANSAC to estimate the homography

```
>> [H,in] = m.ransac(@homography, 0.2)
```

Since we assume the features lie on a plane the homography maps  ${}^1\mathbf{p}$  to  ${}^2\mathbf{p}$ . Now we wish to map  ${}^2\mathbf{p}$  to its corresponding coordinate in the first image

$${}^1\mathbf{p} \approx H^{-1} {}^2\mathbf{p}$$

We do this for every pixel in the new image by warping

```
>> [tile,t] = homwarp(inv(H), im2, 'full', 'extrapval', 0);
```

As shown in Fig. 14.47 the warped blue image falls outside the bounds of the original blue image and the option '**full**' specifies that the returned image is the minimum containing rectangle of the warped image. This image is referred to as a *tile* and shown with a dashed black line. The vector  $t$  is returned by `homwarp` and gives the offset of the tile's coordinate frame with respect to the original image. In general not every pixel in the tile has a corresponding point in the input image and those pixels are set to zero, as specified by the fifth argument.

Now the tile has to be *blended* into the composite image

```
>> canvas = ipaste(canvas, tile, t, 'add');
```

and the result is shown in Fig. 14.48. We can clearly see several images overlaid and with excellent alignment. The non-mapped pixels in the warped image are set to zero so adding them causes no change to the existing pixel values in the composite image.

Simply *adding* the tile into the composite image means that overlapping pixels are necessarily brighter and a number of different strategies can be used to remedy this. We could instead set pixels in the composite image from the tile only if the composite image pixels have not yet been set. Conversely we could *always* set pixels in the composite image from the non-zero pixels in the tile. Alternatively we set the composite image pixels to the mean of the tile and the composite image. This requires that we tag the tile pixels that are not mapped

```
>> [tile,t] = homwarp(inv(H), im2, 'full', 'extrapval', NaN);
```

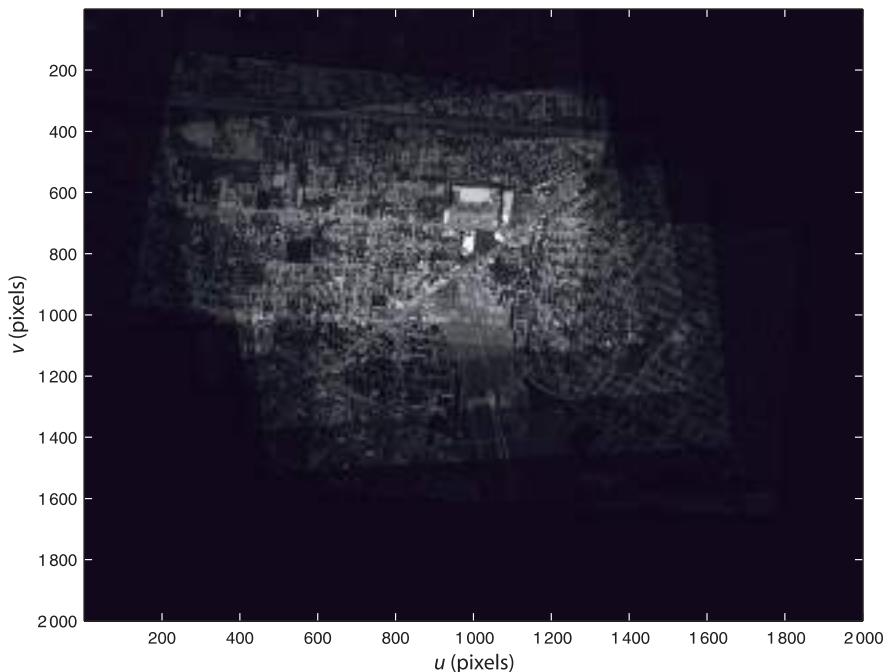
and then blend using the '**mean**' option

```
>> canvas = ipaste(canvas, tile, t, 'mean');
```

The bounding box of the tile is computed by applying the homography to the image corners  $A = (1, 1)$ ,  $B = (W, 1)$ ,  $C = (W, H)$  and  $D = (1, H)$ , where  $W$  and  $H$  are the width and height respectively, and finding the bounds in the  $u$ - and  $v$ -directions.

The default is NaN.

Which ignores any pixels with the value NaN.

**Fig. 14.48.**

Example image mosaic. At the bottom of the frame we can clearly see three overlapping views of the airport runway which shows good alignment between the frames

Google Earth provides an imperfect orthophoto. When looking at cities it is very common to see oblique views of buildings.

If the images were taken with the same exposure then the edges of the tiles would not be visible. If the exposures were different the two sets of overlapping pixels have to be analyzed to determine the average intensity offset and scale factor which can be used to correct the tile before blending – a process known as tone matching.

Finally, we need to consider the effect of points in the image that are not in the ground plane such as those on a tall building. An image taken from directly overhead will show just the roof of the building, but an image taken from further away will be an oblique view that shows the side of the building. In a mosaic we want to create the illusion that we are directly above every point in the image so we should not see the sides of any building. This type of image is known as an orthophoto and unlike a perspective view, where rays converge on the camera's focal point, the rays are all parallel which implies a viewpoint at infinity. At every pixel in the composite image we can choose a pixel from any of the overlapping tiles. To best approximate an orthophoto we should choose the pixel that is closest to overhead, that is, prior to warping the pixel was closest to the principal point.

In photogrammetry this type of mosaic is referred to as an uncontrolled digital mosaic since it does not use explicit control points – manually identified corresponding features in the images. The result is an orthophoto which has a viewpoint at infinity. The full code is given by `mosaic1` in the examples directory. The principles illustrated here can also be applied to the problem of image stabilization. The homography is used to map features in the new image to the location they had in the previous image.

## 14.7 Application: Image Matching and Retrieval

Given a set of images  $\{I_j, j = 1 \dots N\}$  and a new image  $I'$  the image matching problem is to determine  $j$  such that  $I'$  and  $I_j$  are most similar. This is a difficult problem when we consider the effect of changes in viewpoint and exposure. Pixel-level similarity measures such as SSD or ZNCC that we used previously are not suitable for this problem since quite small changes in viewpoint will result in almost zero similarity.

Image matching can be used by a robot to determine if it has visited a particular place before, or seen the same object before. If those previous images have some asso-

ciated semantic data such as the name of an object or the name of a place then by inference that semantic data applies to the new image. For example if a new image matches an existing image that has the semantic tag “lobby” then it implies the robot is seeing the same scene and is therefore in or close to, the lobby.

The particular technique that we will introduce is commonly referred to as “bag of words” and has become very popular in robotics in the last few years. It builds on techniques we have previously encountered such as SURF point features and k-means clustering.

We start by loading a set of twenty images

```
>> images = imread('campus/*.jpg', 'mono');
```

as a  $426 \times 640 \times 20$  array and for each of these we compute the SURF features

```
>> sf = isurf(images);
```

which returns a MATLAB® cell array whose elements are vectors of SURF features that correspond to the input images. For example

```
>> sf{1}
ans =
663 features (listing suppressed)
Properties: theta image_id scale u v strength descriptor
```

is a vector of 663 SURF feature objects corresponding to the first image in the sequence. The set of all SURF features across all images is

```
>> sf = [sf{:}]
sf =
17945 features (listing suppressed)
Properties: theta image_id scale u v strength descriptor
```

which is a vector of 17 945 SURF features objects.

Consider a particular SURF feature

```
>> sf(380)
ans =
(48.1448,219.771), theta=4.4736, image_id=1, scale=3.48386,
strength=0.000615617, descrip= ..
```

and we see the `SurfPointFeature` properties discussed earlier such as centroid, scale and orientation. The property `image_id` indicates that this feature was extracted from the first image in the original image sequence. We can display that image and superimpose the feature

```
>> idisp(images(:,:,1))
>> sf(380).plot('g+')
>> sf(380).plot_scale('g', 'clock')
```

which is shown in Fig. 14.49a. The support region for this feature

```
>> sf(380).support(images)
```

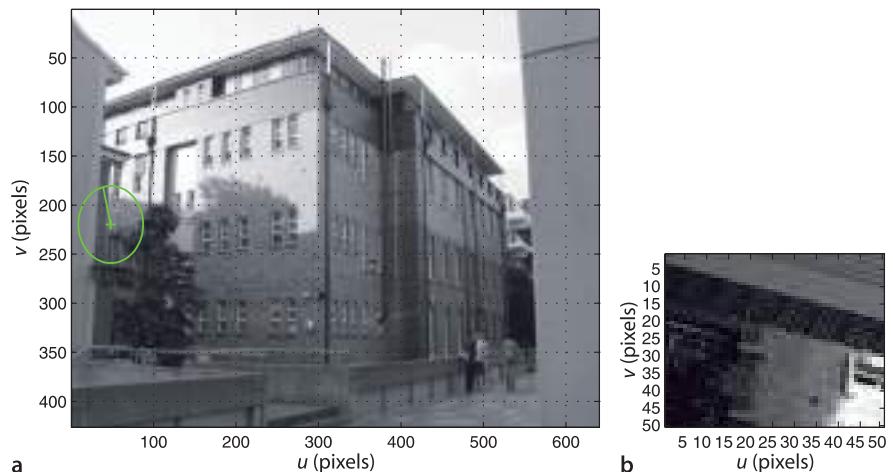
is shown in Fig. 14.49b. The support region shows bricks and the edge of a window. The `support` method uses the `image_id` property to determine which of the passed images contains the feature.

The key insight behind the bag of words technique is that many of these features will describe visually similar scene elements such as leaves, corners of windows, bricks, chimneys and so on. If we consider each SURF feature descriptor as a point in a 64-dimensional space then similar descriptors will form clusters, and this a k-means problem. To find 2 000 feature clusters

```
>> bag = BagOfWords(sf, 2000)
```

returns a `BagOfWords` object that contains the original features, the centre of each cluster, and various other information.► Each cluster is referred to as a visual word and is described by a 64-element SURF descriptor. The set of all visual words, 2 000 in this case, is a visual vocabulary. Just as a document comprises a set of words drawn

The `BagOfWords` class use the MEX-file k-means implementation from <http://www.vlfeat.org/>. This uses its own random number generator and to initialize it to a known state use `vl_twister('STATE', 0.0)`.



**Fig. 14.50.** Exemplars of visual word 1057 from the various images in which it appears. The annotation is of the form word/image

from some vocabulary, each image comprises a collection (or *bag*) of words drawn from the visual vocabulary.

The clustering step assigns a visual word index to every SURF feature. For the particular feature shown above

```
>> bag.words(380)
ans =
    1057
```

we find that the *k*-means clustering has assigned this image feature to word 1057 in the vocabulary – it is an instance of visual word 1057. That particular visual word appears

```
>> bag.ocurrence(1057)
ans =
    6
```

times across the set of images, and it appears at least once in each of the images

```
>> bag.contains(1057)
ans =
    1      2      3      4     14     16
```

We can display some of the different instances of word 1057 by

```
>> bag.exemplars(1057, images)
```

which is shown in Fig. 14.50. These exemplars actually look quite different, but we need to keep in mind that we are viewing them as patterns of pixels whereas the similarity is in terms of the descriptor. The exemplars do however share some dominant horizontal and vertical structure.

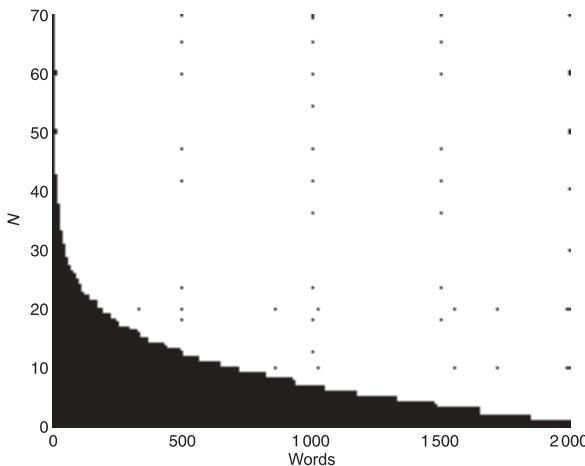
Visual words occur with quite different frequencies

```
>> [word,f] = bag.wordfreq()
```

where `word` is a vector containing all unique words and `f` are their corresponding frequencies. We can display these in descending order of frequency

```
>> bar( sort(f, 'descend') )
```

The descriptor comprises responses of Haar wavelet detectors computed over multiple windows within the support region.



**Fig. 14.51.**  
Histogram of the number of occurrences of each word (sorted). Note the small number of words that occur very frequently

which is shown in Fig. 14.51. Words that occur very frequently have less meaning or power to discriminate between images. They are analogous to English words that are considered stop words in text document retrieval.<sup>►</sup> The visual stop words are removed from the bag of words

```
>> bag.remove_stop(50)
Removing 1850 features associated with 50 most frequent words
>> bag
bag =
BagOfWords: 16095 features from 20 images
1950 words, 50 stop words
```

which leaves some 16 000 SURF features behind. This method performs relabelling so that word labels are now in the interval 1 … 1 950.

Our visual vocabulary comprises  $K$  visual words and in this case  $K = 1950$ . We apply a technique from text document retrieval and describe *each* image by a  $K$ -element vector

$$\mathbf{v}_i = (t_1, \dots, t_j, \dots, t_K)$$

whose elements describes the frequency of the corresponding visual words in the image.

$$t_j = \frac{n_{ij}}{n_i} \log \frac{N}{\underbrace{N_j}_{\text{idf}}} \quad (14.18)$$

where  $j$  is the visual word label,  $N$  is the total number of images in the database,  $N_j$  is the number of images which contain word  $j$ ,  $n_i$  is the number of words in image  $i$ , and  $n_{ji}$  is the number of times word  $j$  appears in image  $i$ . The inverse document frequency (idf) term is a weighting that reduces the significance of words that are common across all images and which are thus less discriminatory. The weighted word frequency vectors are a property of the `BagOfWords` object but can be accessed by

```
>> M = bag.wordvector;
```

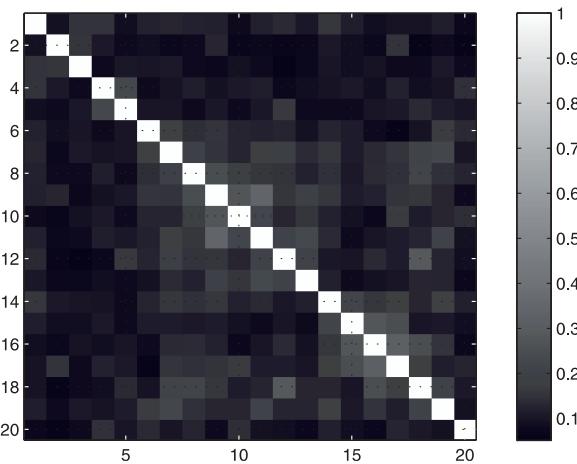
which is a  $1950 \times 20$  matrix and each column is a 1950-element vector that concisely describes the corresponding image.<sup>►</sup>

The similarity between two images is the cosine of the angle between their corresponding word-frequency vectors

$$s(\mathbf{v}_1, \mathbf{v}_2) = \frac{\mathbf{v}_1 \mathbf{v}_2^T}{|\mathbf{v}_1| |\mathbf{v}_2|}$$

Search engines ignore words such as 'a', 'and', 'the' and so on.

This might seem like a very large vector but it contains less than 1% of the number of elements of the original image.

**Fig. 14.52.**

Similarity matrix for 20 images where light colors indicate strong similarity. Element  $(i,j)$  indicates the similarity between image  $i$  and image  $j$

and is implemented by the `similarity` method. A value of one indicates maximum similarity. To compute the mutual similarity between two bags of words is simply

```
>> S = bag.similarity(bag)
```

which returns a  $20 \times 20$  similarity matrix where the elements  $S(i,j)$  indicates the similarity between the  $i^{\text{th}}$  column and  $j^{\text{th}}$  columns of  $M$ , or between image  $i$  and image  $j$ . Such a similarity matrix is best interpreted visually

```
>> idisp(S, 'bar')
```

which is shown in Fig. 14.52. The bright diagonal indicates, as a useful cross check, that image  $i$  is identical to image  $i$ .

Consider image 11 shown in Fig. 14.53a. Its similarity to other images is given by row, or column, 11 of the similarity matrix

```
>> s = S(:,11);
```

which we sort into descending order of similarity

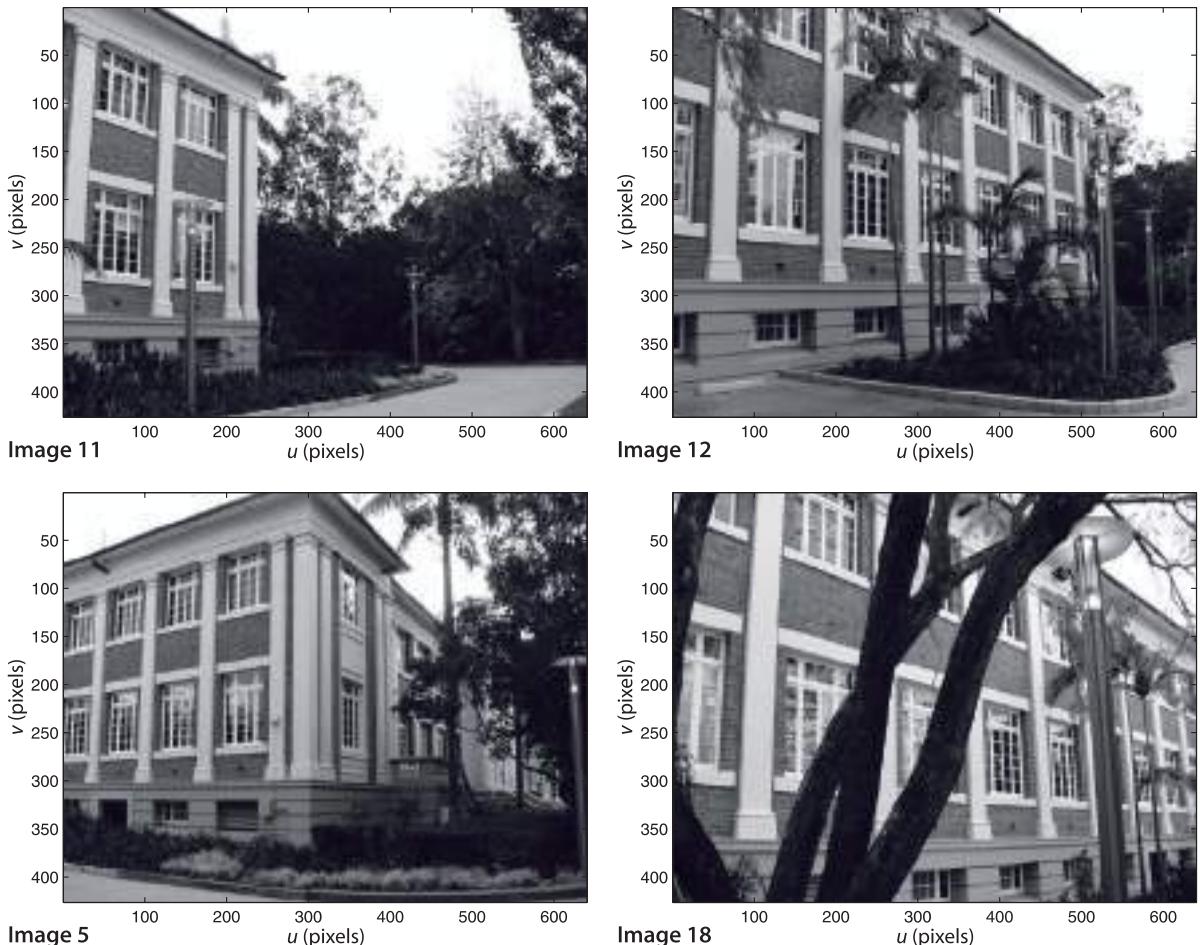
```
>> [z,k] = sort(s, 'descend');
>> [z k]
ans =
    1.0000    11.0000
    0.3448    9.0000
    0.2377   13.0000
    0.1948   10.0000
    0.1719    7.0000
    :
    :
```

where each row comprises the similarity measure and the corresponding image. Image 11 is identical to image 11 as expected, and in decreasing order of similarity we have images 9, 13, 10 and so on. These are shown in Fig. 14.53 and we see that the algorithm has recalled quite different views of the same building.

Now consider that we have some new images and we wish to determine which of the previous images is the most similar. Perhaps the robot has taken a picture and wishes to compare it to its database of existing images. The steps are broadly similar to the previous case

```
>> images2 = iread('campus/holdout/*.jpg', 'mono');
>> sf2 = isurf(images2)
```

but rather than perform clustering we want to assign the features to the existing set of visual words, that is, to determine the closest visual word for each of the new feature descriptors



**Fig. 14.53.** Image recall. Image 11 is the query, and in decreasing order of match quality we have recalled images 9, 13 and 10

```
>> bag2 = BagOfWords(sf2, bag)
bag2 =
BagOfWords: 4708 features from 5 images
1950 words, 50 stop words
```

This operation also removes any features words that were previously determined to be stop words, and computes the word frequency vectors according to Eq. 14.18.

Finally the similarity between the images in the two bags of words is

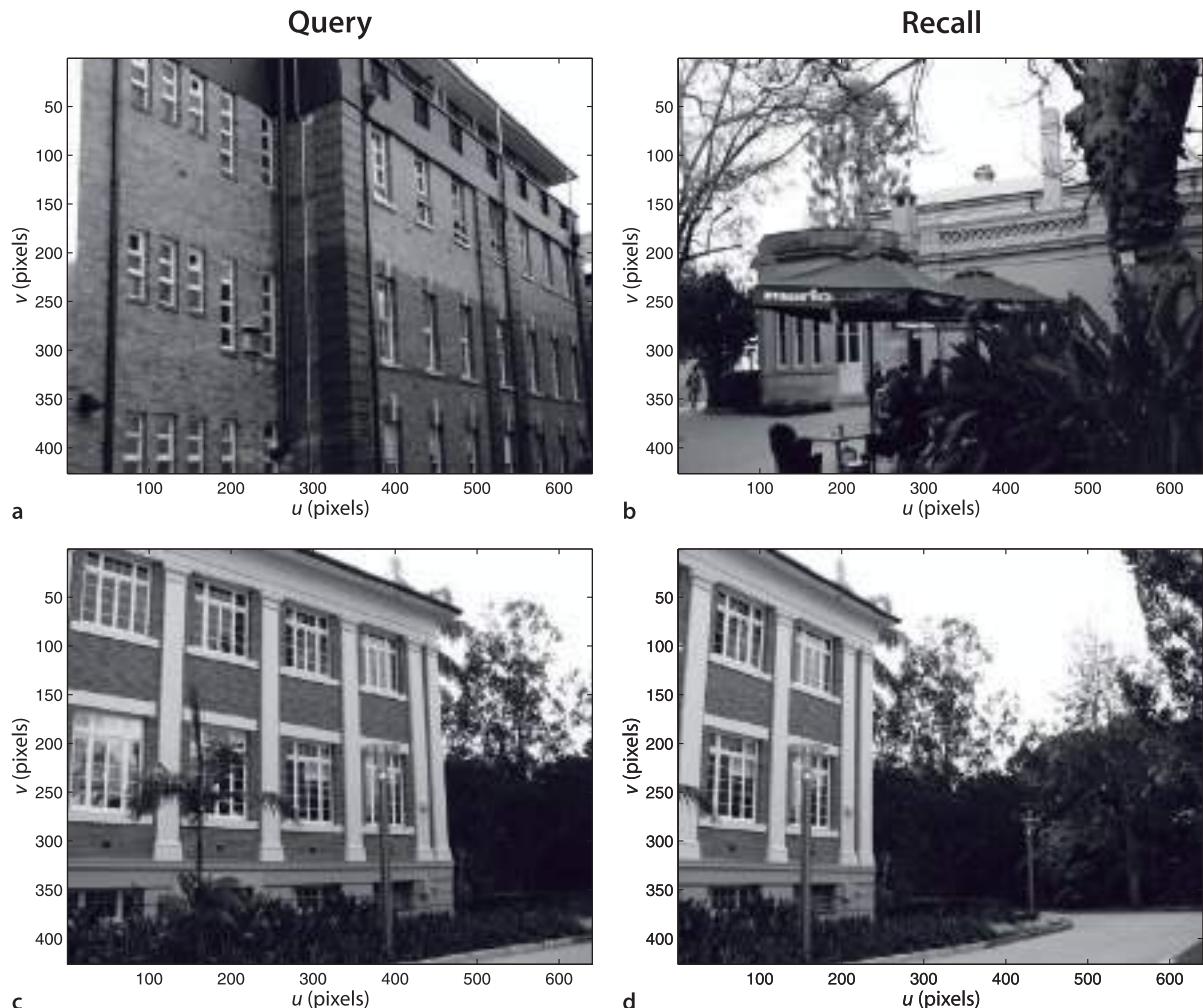
```
>> S2 = bag.similarity(bag2);
```

which returns a  $20 \times 5$  matrix where the elements  $S(i, j)$  indicates the similarity between the existing image  $i$  and new image  $j$ . The maxima in each column corresponds to the most similar image in the previously observed set

```
>> [z,i] = max(S2)
z =
    0.2751    0.690    0.5121    0.5017    0.3177
i =
    7     11     16     18     20
```

New image 1 best matches image 7 in the original sequence, new image 2 matches image 11 and so on. The new images and their closest existing images are shown in Fig. 14.54. The first recall has a low similarity score and is incorrect – it shows a very different building. However the two scenes do have some similar features such as windows, roof line and gutters.

Which requires the image-word statistics from the existing bag of words to compute the idf weighting terms.



**Fig. 14.54.** Image recall for new images. The new query images **a** and **c** recall the database images **b** and **d** respectively

## 14.8 Application: Image Sequence Processing

In this final section of the chapter, and this part of the book, we look at how to use the Toolbox for processing image sequences. We load a sequence of images taken from a car driving along a road

```
>> im = imread('bridge-1/*.png', 'roi', [20 750; 20 480]);
```

and the option '**roi**' selects a region of interest from each image to eliminate an irregular black border.► These images are unusual in having 16-bit pixels

```
>> about(im)
im [uint16] : 461x731x251 (169169482 bytes)
```

and the image **im** belongs to the class '**uint16**'. Since this sequence is already nearly 200 Mbyte we do not convert it to double precision since this would quadruple the amount of memory required.

The image sequence can be displayed as an animation

```
>> ianimate(im, 'fps', 10);
```

at 10 frames per second.

The black border is the result of image rectification.

For each frame we compute corner features

```
>> c = icorner(im, 'nfeat', 200, 'patch', 7);
```

and for a change we have used Harris corners since they are computationally cheaper. For this application the change in orientation and scale from frame to frame is small and Harris corner features are well suited for this purpose. The function returns a cell array with one element per input image, and each element is a vector of the 200 strongest Harris corner features per image. The 'patch' option specifies a  $15 \times 15$  local neighbourhood descriptor according to Eq. 14.1 which is a 225-element unit vector. The image sequence can be displayed as an animation with the features overlaid

```
>> ianimate(im, c, 'fps', 10);
```

at 10 frames per second and a single frame of this sequence is shown in Fig. 14.55. The features are associated with regions of high gradient such as the edges of trees, as well as the corners of signs and cars. Watching the animation we see that the corner features stick reliably to world points for many frames. The motion of features in the image is known as optical flow and is a function of the camera's motion through the world and the 3-dimensional structure of the world.►

The Toolbox class `Tracker` maintains a list of *tracks*, which are the motion of individual features over time

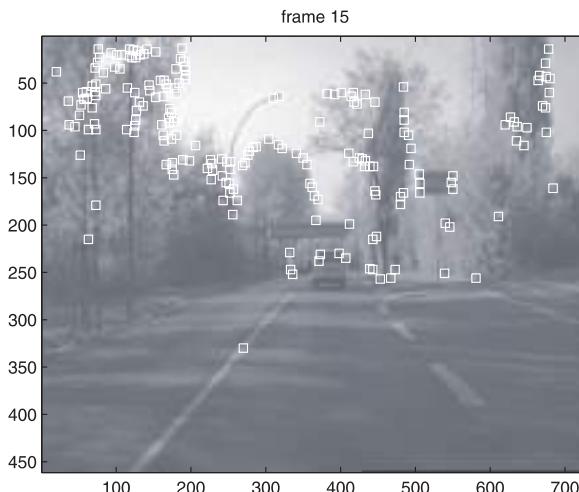
```
>> t = Tracker(im, c);
```

Figure 14.56a shows one frame from the operation of `Tracker` where the features are annotated with a unique identifier associated with a particular track. For each track the object maintains the track number, the current position of the feature, the feature descriptor and some statistics such as the number of times it has been seen and how many frames since it was last seen.

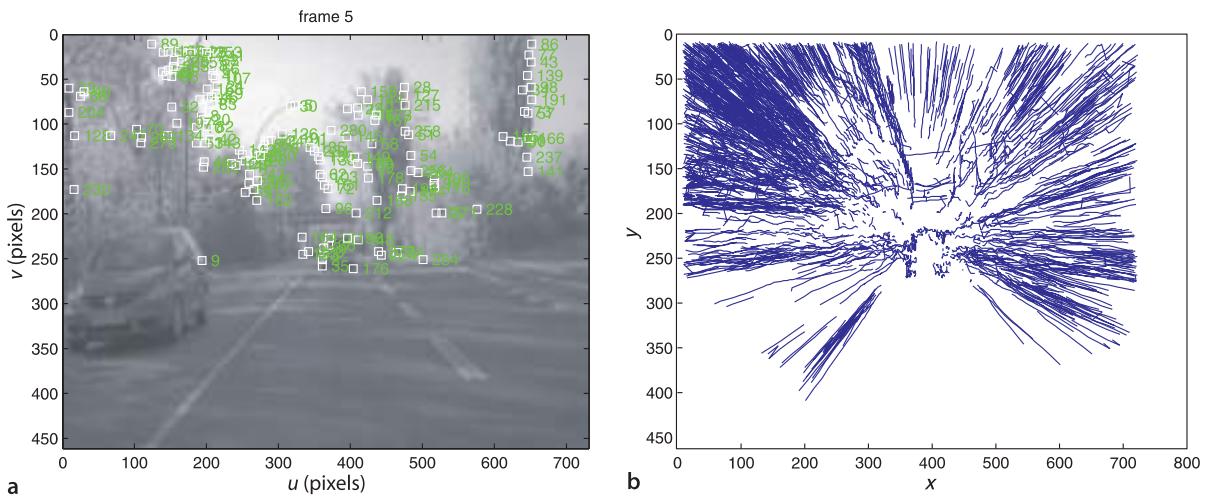
For each new frame every track finds a subset of features within a specified radius of its position in the previous frame. The descriptors of these features are compared with the track's feature descriptor from the previous frame, and if a match is found the track is updated. Any feature that is not claimed by a track is considered to be the start of a new track – a new entry is created in the table and a unique track identifier is assigned to it. If a track doesn't find a matching feature in the current frame a counter is incremented, and when it exceeds some threshold the track is retired.

The descriptor associated with the track is updated at each time step. In this way, even if the support region changes in appearance from frame to frame, we always compare new features to the most recent appearance of the tracked feature.

We will revisit optical flow in the next chapter.

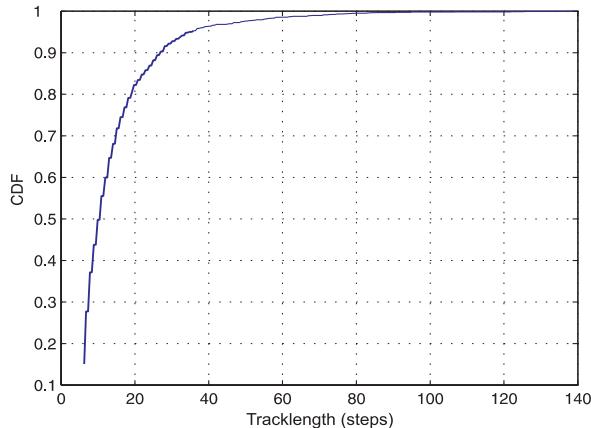


**Fig. 14.55.**  
Frame number 15 from the bridge-1 image sequence with overlaid features (image from enpeda. project, Klette et al. 2011)



**Fig. 14.56.** Temporal tracking of features. **a** Features with unique track identifiers shown; **b** all feature tracks

**Fig. 14.57.** Cumulative histogram of track length. 50% of the tracks are longer than 10 frames, and 5% are longer than 40 frames



Data about the tracks is stored as a property of the class. The feature trajectories can be displayed as individual lines

```
>> t.plot()
```

as shown in Fig. 14.56b. The motion of features outward from the centre of the image is very clear and the point from which features appear to expand is called the centre of expansion. In terms of 2-camera epipolar geometry the second camera is in front of the first camera, so the epipole will be close the middle of the image and the epipolar lines all pass through it. The corresponding points must lie on the epipolar lines so therefore move along these radiating lines, and the amount of motion is inversely related to the depth of the world point. In the next part of the book we will explore the relationship between feature motion and camera motion in more detail.

We see that there are some tracks that are not radial and these are due to the rather simplistic tracker which is not enforcing frame-to-frame epipolar constraints and which RANSAC could eliminate. One of the horizontal tracks is due to a car driving across a bridge which has violated our assumption that the world is rigid, that is, points in the world do not move with respect to each other. The number of frames over which features are tracked varies considerably. The method `tracklengths` returns a vector of the number of frames that each feature was tracked for

```
>> ihist( t.tracklengths(), 'normcdf' )
```

and Fig. 14.57 shows the distribution of feature track lengths. The majority are quite short, the mean is 14 frames, but the longest is 140 frames.

## 14.9 Wrapping Up

This chapter has covered many topics but the aim has been to demonstrate a multiplicity of concepts that are of use in real robotic vision systems. There have been two common threads through this chapter. The first has been the use of corner features to find distinctive points in images, and matching them to the same world point in another image. The second thread has been the use of additional sources of information to recover the depth of a point or translation scale that is lost in the perspective projection process.

We extended the geometry of single camera imaging to the case of two cameras and showed how corresponding points in the two images are constrained by the fundamental matrix. We showed how the fundamental matrix can be estimated from image data, the effect of incorrect data association, and how to overcome this using the RANSAC algorithm. Using camera intrinsic parameters the essential matrix can be computed and then decomposed to give the camera motion between the two view, but the translation has an unknown scale factor. With some extra information such as the magnitude of the translation the camera motion can be estimated completely. Given the camera motion then the 3-dimensional coordinates of points in the world can be estimated.

World points that lie on a plane induce an homography that is a linear mapping of image points between images. The homography can be used to detect points that do not lie in the plane and can be decomposed to give the camera motion between the two views (translation again has an unknown scale factor) and the normal to the plane.

If the fundamental matrix is known then a pair of overlapping images can be rectified to create an epipolar-aligned stereo pair and dense stereo matching can be used to recover the world coordinates for every point. Errors due to effects such as occlusion and lack of texture were discussed as were techniques to detect these situations.

These multi-view techniques were then used in a number of application examples such as perspective correction, mosaic creation, image retrieval and image sequence analysis.

## Further Reading

3-dimensional reconstruction and camera pose estimation has been studied by the photogrammetry community since the mid nineteenth century. 3-dimensional computer vision or *robot vision* has been studied by the computer vision and artificial intelligence communities since the 1960s. This book follows the language and nomenclature associated with the computer vision literature, but the photogrammetric literature can be comprehended with only a little extra difficulty.

Significant early work on multi-view geometry was conducted at laboratories such as Stanford, SRI International, MIT AI laboratory, CMU, JPL, INRIA, Oxford and ETL Japan in the 1980s and 1990s and led to a number of text books being published in the early 2000s.

The definitive references for multiple-view geometry are Hartley and Zisserman (2003) and Ma et al. (2003). These books present quite different approaches to the same body of material. The former takes a more geometric approach while the latter is more mathematical. Unfortunately they use quite different notation, and each differs from the notation used in this book – a summary of the important notational elements is given in Table 14.1. These books all cover feature extraction (using Harris corner features, since they were published before scale invariant feature detectors such as SIFT and SURF corner detectors were developed); the geometry of one, two and  $N$  views; fundamental and essential matrix; homographies; and the recovery of 3-dimensional scene structure and camera motion through offline batch techniques. Both provide the key algorithms in pseudo-code and have some supporting MATLAB® code on their associated web sites. The slightly earlier book by Faugeras et al. (2001) covers much of the same material using a fairly mathematical approach and with different

**Table 14.1.**  
Rosetta stone. Summary of notational differences between two other popular textbooks and this book

Object	Hartley and Zisserman 2003	Ma et al. 2003	This book
World point	$\mathbf{x}$	$P$	$\mathbf{P}$
Image point	$\mathbf{x}, \mathbf{x}'$	$x_1, x_2$	${}^1\mathbf{p}, {}^2\mathbf{p}$
$i^{\text{th}}$ image point	$\mathbf{x}_i, \mathbf{x}'_i$	$x_1^i, x_2^i$	${}^1\mathbf{p}_i, {}^2\mathbf{p}_i$
Camera motion	$\mathbf{R}, \mathbf{t}$	$R, T$	$\mathbf{R}, \mathbf{t}$
Normalized coordinates	$\mathbf{x}, \mathbf{x}'$	$x_1, x_2$	$(\bar{U}, \bar{V})$
Camera matrix	$\mathbf{P}$	$\Pi$	$\mathbf{C}$
Homogeneous quantities	$\mathbf{x}, \mathbf{X}$	$x, P$	$\tilde{\mathbf{p}}, \tilde{\mathbf{P}}$
Homogeneous equivalence	$\mathbf{x} = \mathbf{P} \mathbf{X}$	$\lambda x = \Pi P$ $x \sim \Pi P$	$\tilde{\mathbf{p}} \simeq \mathbf{C} \tilde{\mathbf{P}}$

notation again. The older book by Faugeras (1993) focuses on sparse stereo from line features. The recent book by Szeliski (2010) provides a very readable and deeper discussion of the topics in this chapter.

References related to SURF and other feature detectors were previously discussed on page 377. The performance of feature detectors and their matching performance is covered in Mikolajczyk and Schmid (2005) which reviews a number of different feature descriptors including spin images and local jets.◀

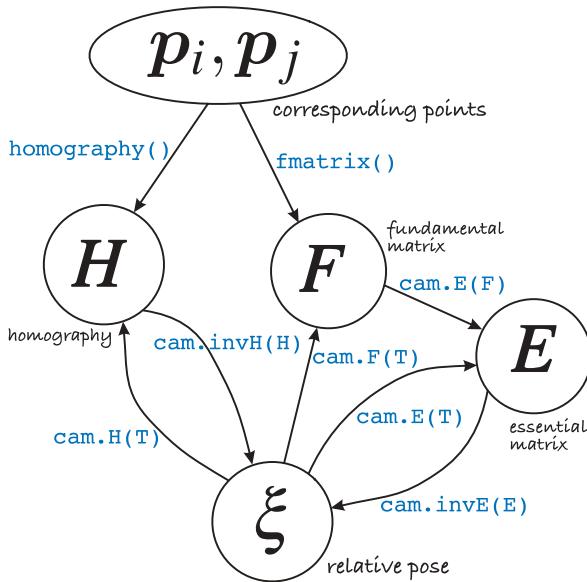
The RANSAC algorithm described by Fischler and Bolles (1981) is the workhorse of all the feature-based methods discussed in this chapter. A more recent development is PROSAC (Chum and Matas 2005) which exploits the ordering of corresponding points. Pilu (1997) discusses how SVD can be applied to a matrix formed from the distances between features to determine correspondence. Dellaert et al. (2000) describe a probabilistic approach to determining structure from a group of images not necessarily in order.

The term fundamental matrix was defined in the thesis of Luong (1992). The book by Xu and Zhang (1996) is a readable introduction to epipolar geometry. Epipolar geometry can also be formulated for non-perspective cameras in which case the epipolar line becomes an epipolar curve (Mičušík and Pajdla 2003; Svoboda and Pajdla 2002). For three views the geometry is described by the trifocal tensor  $\mathcal{T}$  which is a  $3 \times 3 \times 3$  tensor with 18 degrees of freedom that relates a point in one image to epipolar lines in two other images (Hartley and Zisserman 2003; Ma et al. 2003). An important early paper on epipolar geometry for an image sequence is Bolles et al. (1987).

The essential matrix was first described a decade earlier in a letter to Nature (Longuet-Higgins 1981) by the eminent theoretical chemist and cognitive scientist Christopher Longuet-Higgins (1923–2004). The paper describes a method of estimating the essential matrix from eight corresponding point pairs. The decomposition of the essential matrix was first described in Faugeras (1993, § 7.3.1) but is also covered in the texts Hartley and Zisserman (2003) and Ma et al. (2003). In this chapter we have estimated camera motion by first computing the essential matrix and then decomposing it. The first step requires at least eight pairs of corresponding points but algorithms such as Nistér (2003), Li and Hartley (2006) compute the motion directly from just five pairs of points. Decomposition of an homography is described by Faugeras and Lustman (1988), Hartley and Zisserman (2003), Ma et al. (2003), and the comprehensive technical report by Malis and Vargas (2007). The relationships between these matrices, camera motion, and the relevant Toolbox functions are summarized in Fig. 14.58.

Stereo cameras and stereo matching software are available today from several vendors and can provide high-resolution depth maps at more than 10 Hz on standard computers. A decade ago this was difficult and custom hardware including FPGAs was required to achieve real-time operation (Corke et al. 1999; Woodfill and Von Herzen 1997). The application of stereo vision for planetary rover navigation is discussed by

A jet is a vector of higher order derivatives such as  $I_{uuu} I_{vuv} I_{uvu} I_{uuu} I_{uvv} I_{uuv} I_{vvv}$  and so on (Mikolajczyk and Schmid 2005).



**Fig. 14.58.**  
Toolbox functions and camera object methods, and their inter-relationships

Matthies (1992). More than two cameras can be used, and multi-camera stereo was introduced by Okutomi and Kanade (1993) and provides robustness to problems such as the picket fence effect.

Brown et al. (2003) provide a readable review of stereo vision techniques with a focus on real-time issues. An old but clearly written book on the principles of stereo vision is Shirai (1987). Scharstein and Szeliski (2002) consider the stereo process as four steps: matching, aggregation, disparity computation and refinement. The cost and performance of different algorithms for each step are compared. The example in this chapter would be described as: NCC matching, box filter aggregation, winner takes all, and sub-pixel refinement. More sophisticated approaches are described that take similarity of neighbouring pixels into account using iterative regularization, dynamic programming or graph-based methods which can give improved performance at discontinuities and can explicitly model occlusion. However improving the smoothness and appearance of disparity data involves making assumptions about the world for which the images do not provide strong evidence, and should be used with caution for robotic control. The similarity of a stereo camera to our own two eyes is very striking, and while we do make strong use of stereo vision it is not the only technique we use to infer distance (Cutting 1997).

The ICP algorithm (Besl and McKay 1992) is used for a wide range of applications from robotics to medical imaging. ICP is fast but determining the correspondences via nearest neighbours is an expensive  $O(N^2)$  operation. Many variations have been developed that make the approach robust to outlier data and to improve computational speed for large datasets. Salvi et al. (2007) provide a recent review and comparison of some different algorithms. Determining the relative orientation between two sets of points is a classical problem and the SVD approach used here is described by Arun et al. (1987). Solutions based on quaternions and orthonormal rotation matrices were described by Horn (Horn et al. 1988; Horn 1987).

Structure from motion (SfM), the simultaneous recovery of world structure and camera motion, is a classical problem in computer vision. Two useful review papers are by Huang and Netravali (1994) which provides a taxonomy of approaches, and Jebara et al. (1999). Broida et al. (1990) describe an early recursive SfM technique for a monocular camera sequence using an EKF where each world point is represented by its  $(X, Y, Z)$  coordinate. McLauchlan provides a detailed description of a variable-length state estimator for SfM (McLauchlan 1999). Azarbayejani and Pentland (1995) present a recursive approach where each world point is parameterized by a scalar, its depth with respect to the first image. A more recent algorithm with bounded estimation error is described

by Chiuso et al. (2002) and also discusses the problem of scale variation. The MonoSlam system by Davison et al. (2007) is an impressive monocular SfM system that maintains a local map that includes features even when they are not currently in the field of view. The application of SfM to large-scale urban mapping is becoming increasing popular and Pollefeys et al. (2008) describe a system for offline processing of large image sets.

The offline SfM problem, in which a sequence of images is processed to recover the motion and structure, is not covered in this book. The approach typically involves estimating the camera matrix for each view, computing a projective reconstruction, and then upgrading it to a Euclidean reconstruction (Hartley and Zisserman 2003; Ma et al. 2003). A key part of these offline techniques is bundle adjustment which is an optimization process that adjusts the estimated camera poses and world points to minimize the error between estimated and actual image points. A good introduction to bundle adjustment is Triggs et al. (2000).

The SfM problem can be simplified by using stereo rather than monocular image sequences (Molton and Brady 2000; Zhang et al. 1992), or by incorporating inertial data (Strelow and Singh 2004). A related technique is visual odometry (VO) which is concerned only with recovering the camera motion. Nistér et al. (2006) describe a point feature-based system for monocular or stereo vision. Maimone et al. (2007) describe experience with stereo-camera VO on the Mars rover and Corke et al. (2004) describe catadioptric VO for a prototype planetary rover.

Mosaicing is a process as old as photography. In the past it was highly skilled and labour intensive requiring photographs, scalpels and sandpaper. The surface of the Moon and nearby planets was mosaiced manually in the 1960s using imagery sent back by robotic spacecraft. Today a number of high-quality mosaicing tools are available for creating panoramas, for example the Hugin open source project <http://hugin.sourceforge.net> and the proprietary AutoStitch.

The “bag of words” technique for image retrieval was first proposed by Sivic and Zisserman (2003) and has been used by many other researchers since. A notable extension for robotic applications is FABMAP (Cummins and Newman 2008) which explicitly accounts for the joint probability of feature occurrence and associates a probability with the image match.

Image sequence analysis is the core of many real-time robotic vision systems. Real-time feature tracking is described by Hager and Toyama (1998), Lucas and Kanade (1981) and is typically based on the computationally cheaper Harris detectors or the pyramidal Kanade-Lucas-Tomasi (KLT) tracker. SURF detectors are still too time consuming to use for this purpose although some C-based implementations and GPU implementations are capable of modest real-time performance.

---

## Resources

The field of computer vision has progressed through the availability of standard datasets. These have enabled researchers to quantitatively compare the performance of different algorithms on the same data. One of the earliest collections of stereo image pairs was the JISCT dataset (Bolles et al. 1993) named for the research groups that contributed to it: JPL, INRIA, SRI, CMU, and Teleos. It is available at <http://vasc.ri.cmu.edu/idb/html/jisct>. The more recent Middlebury dataset (Scharstein and Szeliski 2002) at <http://vision.middlebury.edu/stereo> provides an extensive collection of stereo images, at high resolution, taken at different exposure settings and including ground truth data. Stereo images from NASA’s Mars exploration rovers Spirit and Opportunity are available online at <http://marsrover.nasa.gov/gallery/3d>. These are in anaglyph format with left and right images encoded as red and cyan respectively. The red and blue color planes of the anaglyph image are rectified left and right images. Motion datasets include classic motion sequences of indoor scenes <http://vasc.ri.cmu.edu//idb/html/motion>, people moving inside a building <http://>

[homepages.inf.ed.ac.uk/rbf/CAVIARDATA1](http://homepages.inf.ed.ac.uk/rbf/CAVIARDATA1), traffic scenes [http://i21www.ira.uka.de/image\\_sequences](http://i21www.ira.uka.de/image_sequences), and from a moving vehicle <http://www.mi.auckland.ac.nz/EISATS>.

An implementation of the KLT feature tracker, in C, written by Stan Birchfield is available at <http://www.ces.clemson.edu/~stb/klt>. A GPU-based version of KLT, in C, by Christoper Zach is available at <http://www.cs.unc.edu/~cmzachopensource.html>. Pointers to SIFT and SURF implementations are given on page 377. The Epipolar Geometry Toolbox (Mariottini and Prattichizzo 2005) for MATLAB® by Gian Luca Mariottini and Domenico Prattichizzo is available at [http://egt.dii.unisi.it/#tth\\_sEc7](http://egt.dii.unisi.it/#tth_sEc7) and handles perspective and catadioptric cameras. Andrew Davison's MonoSLAM system for C and MATLAB® is available at <http://www.doc.ic.ac.uk/~ajd/software.html>.

There are several offline or batch SfM Toolboxes available. The Structure and Motion Toolkit in MATLAB® by Philip Torr (2002) is available at <http://www.mathworks.com/matlabcentral/fileexchange/4576-structure-and-motion-toolkit-in-MATLAB>. The *Structure from Motion Toolbox* by Vincent Ribaud is a collection of popular SfM algorithms (Rabaud, <http://vision.ucsd.edu/~vrabaud/toolbox>) and is available at <http://code.google.com/p/vincents-structure-from-motion-matlab-toolbox>. That in turn makes use of the Sparse Bundle Adjustment (SBA) tool by Lourakis and Argyros (2009). An alternative bundle adjustment package is Simple Sparse Bundle Adjustment (SSBA) by Christoper Zach which is available at <http://www.cs.unc.edu/~cmzachopensource.html>.

The fundamental matrix song can be found at <http://danielwedge.com/fmatrix/>.

## Exercises

1. Corner features. Examine the cumulative distribution of corner strength for Harris and SURF features. What is an appropriate way to choose strong corners for feature matching?
2. Feature matching. We could define the quality of descriptor-based feature matching in terms of the percentage of inliers after applying RANSAC.
  - a) Take any image. We will match this image against various transforms of itself to explore the robustness of SURF and Harris features. The transforms are: (a) scale the intensity by 70%; (b) add Gaussian noise with standard deviation of 0.05, 0.5 and 2 greylevels; (c) scale the size of the image by 0.9, 0.8, 0.7, 0.6 and 0.5; (d) rotate by 5, 10, 15, 20, 30, 40 degrees.
  - b) For the Harris detector compare the performance for the structure-tensor-based feature and the patch descriptor sizes of  $3 \times 3$ ,  $7 \times 7$  and  $11 \times 11$  and  $15 \times 15$ .
  - c) Try increasing the suppression radius for SURF and Harris corners. Does the lower density of matches improve the matching performance?
  - d) The Harris detector can process a color image. Does this lead to improved performance compared to the greyscale version of the same image?
  - e) Is there any correlation between outlier matches and strength of the corner features involved?
3. Write the equation for the epipolar line in image two, given a point in image one.
4. Show that the epipoles are the null-space of the fundamental matrix.
5. Can you determine the camera matrix  $C$  for camera two given the fundamental matrix and the camera matrix for camera one?
6. Estimating the fundamental matrix (page 391)
  - a) For the synthetic data example vary the number of points and the additive Gaussian noise and observe the effect on the residual.
  - b) For the Eiffel tower data observe the effect of varying the parameter to RANSAC. Repeat this with SURF features computed with a lower strength threshold (the default is 0.002).
  - c) What is the probability of drawing 8 inlier points in a random sample (without replacement)?

7. Essential matrix (page 390)
  - a) Create a set of corresponding points for a camera undergoing pure rotational motion, and compute the fundamental and essential matrix. Can you recover the rotational motion?
  - b) For a case of translational and rotational motion visualize both poses that result from decomposing the essential matrix. Sketch it or use `trplot`.
8. Sparse stereo (page 401)
  - a) The ray intersection method can return the closest distance between the rays (which is ideally zero). Plot a histogram of the closing error and compute the mean and maximum error.
  - b) The assumed camera translation magnitude was 30 cm. Repeat for 25 and 35 cm. Are the closing error statistics changed? Can you determine what translation magnitude minimizes this error?
9. Derive a relationship for depth in terms of disparity for the case of verged cameras. That is, cameras with their optical axes intersecting similar to the cameras shown in Fig. 14.6.
10. Stereo vision. Using the rock piles example (page 405)
  - a) Use `idisp` to zoom in on the disparity image and examine pixel values on the boundaries of the image and around the edges of rocks.
  - b) Experiment with different similarity measures and window sizes. What effects do you observe in the disparity image and computation time?
  - c) Experiment with changing the disparity range. Try `[50, 90]`, `[30, 90]`, `[40, 80]` and `[40, 100]`. What happens to the disparity image and why?
11. Using the rock piles example (page 405) obtain the disparity space image  $D$ 
  - a) For selected pixels  $(u, v)$  plot  $D(u, v, d)$  versus  $d$ . Look for pixels that have a sharp peak, broad peak and weak peak. Repeat this for stereo computed using ZSSD similarity. For a selected row  $v$  display  $D(u, v, d)$  as an image. What does this represent?
  - b) For a particular pixel plot  $s$  versus  $d$ , fit a parabola around the maxima and overlay this on the plot.
  - c) Use raw data from the DSI, find the second peak at each pixel and compute the ambiguity ratio
  - d) Display the epipolar lines on image two for selected points in image one.
12. Download an analglyph image and convert it into a pair of greyscale images, then compute dense stereo.
13. Epipolar geometry
  - a) Create two central cameras, one at the origin and the other translated in the  $x$ -direction. For a sparse fronto-parallel grid of world points display the family of epipolar lines in image two that correspond to the projected points in image one. Describe these epipolar lines? Repeat for the case where camera two is translated in the  $y$ - and  $z$ -axes and rotated about the  $x$ -,  $y$ - and  $z$ -axes. Repeat this for combinations of motion such as  $x$ - and  $z$ -translation or  $x$ -translation and  $y$ -rotation.
  - b) The garden example of Fig. 14.16 has epipolar lines that slope slightly downward. What does this indicate about the two camera views?
14. Homography (page 396)
  - a) Compute Euclidean homographies for translation in the  $x$ -,  $y$ - and  $z$ -directions and for rotation about the  $x$ -,  $y$ - and  $z$ -axes. Convert these to projective homographies and apply to a fronto-parallel grid of points. Is the resulting image motion what you would expect? Apply these homographies as a warp to a real image such as Lena.
  - b) Decompose the homography of Fig. 14.15, the garden image, to determine the plane of the wall with respect to the camera. You will need the camera intrinsic parameters.

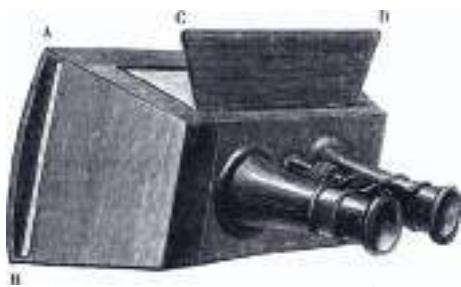
15. Plane fitting (page 419)
  - a) Test the robustness of the plane fitting algorithm to additive noise and outlier points.
  - b) Implement an iterative approach with weighting to minimize the effect of outliers.
  - c) Create a RANSAC-based plane fit algorithm that takes random samples of three points to solve for Eq. 14.17. Use the `fmatrix` and `homography` code to guide you. You will need to create a number of functions that are invoked by the `ransac_driver`.
16. ICP (page 420)
  - a) Run the ICP example on your computer and watch the animation.
  - b) Change the initial relative pose between the point clouds. Try some very large rotations.
  - c) Increase the noise added to the data points.
  - d) For the case where there are missing and/or spurious data points experiment with different values of the '`distthresh`' option.
17. Structure from motion (page 422)
  - a) Experiment with different levels of camera noise and see the effect on estimated camera position.
  - b) What is the effect of noise on estimated camera orientation?
  - c) Modify the simulation to model an odometer, based on the known distance travelled between frames, and use that information to correct the translational scale. Repeat with a systematic scale error in the odometer. Repeat with additive noise in the odometer.
  - d) Use a moving average filter to smooth the frame to frame world point estimates.
  - e) Use an extended Kalman filter to smooth the frame to frame world point estimates (challenging).
18. Perspective correction (page 428)
  - a) Create a virtual view looking downward at 45° to the front of the cathedral.
  - b) Create a virtual view from the original camera height but with the camera rotated 20° to the left.
  - c) Find another real picture with perspective distortion and attempt to correct it.
19. Mosaicing (page 431)
  - a) Run the example file `mosaic` and watch the whole mosaic being assembled.
  - b) Modify the way the tile is pasted into the composite image to use pixel averaging rather than addition.
  - c) Modify the way the tile is pasted into the composite image so that pixels closest to the principal point are used.
  - d) Run the software on a set of your own overlapping images and create a panorama.
20. Bag of words (page 433)
  - a) Examine the different support regions of different visual words using the `exemplars` method.
  - b) Investigate the effect of changing the number of stop words.
  - c) Investigate the effect of changing the size of the vocabulary. Try 1 000, 1 500, 2 500, 3 000.
  - d) Build a bag of words from a set of your own images.
21. Image sequence tracking (page 439)
  - a) Run the example on your computer.
  - b) Repeat using the default Harris feature descriptor, that is, without the '`patch`' option. What happens to the quality of the tracks?
  - c) Repeat with different patch sizes. What happens to the quality of the tracks?
  - d) Repeat using SURF rather than Harris corner features. Is the quality of the tracks improved? How has compute time changed?
  - e) Modify the tracker so that it keeps an estimate of the velocity of the feature in the image (based on its motion over two or more previous frames). Modify the association logic so that the search disk is centered on the predicted position of the feature rather than its previous position. The filter could be based on a simple constant-

velocity model, an  $\alpha - \beta$  tracker (use the `AlphaBetaFilter` class) or subclass the Toolbox Kalman filter (`KalmanFilter`) abstract superclass.

- f) At each frame estimate the fundamental matrix using RANSAC. Modify the association logic to use the epipolar line constraint.
- g) Estimate the essential matrix from frame to frame. The required camera calibration data is in the README file. Decompose the matrix to determine frame to frame change in pose. Integrate the change in orientation over time to provide a visual compass or gyroscope function. How could you recover the unknown scale factor on frame to frame translation? Can you plot the car's trajectory as seen from overhead?
- h) The bridge sequence was recorded in stereo and so far we have used just the left-hand camera. The folder `bridge-r` contains the corresponding right-hand images. Using the camera calibration data in the README file to perform a sparse stereo reconstruction for every image pair. Use ICP to determine the change in pose of the car from frame to frame.
- i) Perform a dense stereo reconstruction for every image pair.
- j) Compute Harris features for live video captured on your computer and overlay these on the captured frame. How many frames per second will this run on your computer? Use the `step` method of the `Tracker` class to track these live features.

# 15

# Vision-Based Control



The task in visual servoing is to control the pose of the robot's end-effector, relative to the target, using visual *features* extracted from the image. As shown in Fig. 15.1 the camera may be carried by the robot or fixed in the world. The configuration of Fig. 15.1a has the camera mounted on the robot's end-effector observing the target, and is referred to as end-point closed-loop or eye-in-hand. The configuration of Fig. 15.1b has the camera at a fixed point in the world observing both the target and the robot's end-effector, and is referred to as end-point open-loop. In the remainder of this book we will discuss only the eye-in-hand configuration.

The image of the target is a function of the relative pose  ${}^C\xi_T$ . Features such as coordinates of points, or the parameters of lines or ellipses are extracted from the image and these are also a function of the relative pose  ${}^C\xi_T$ .

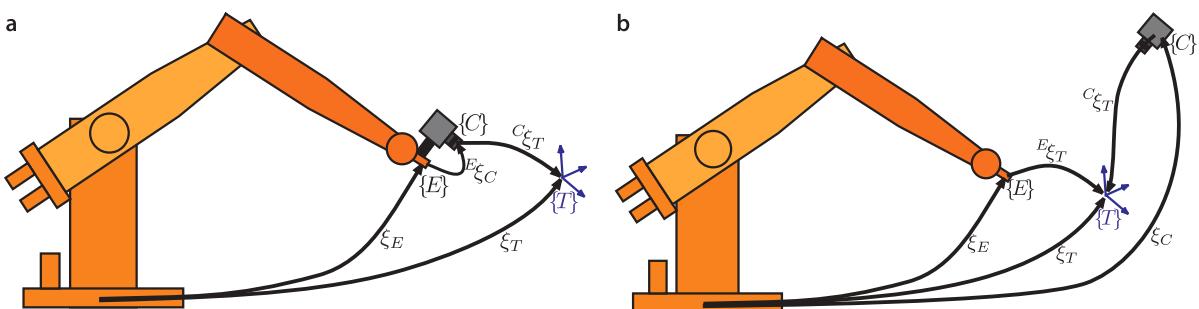
There are two fundamentally different approaches to visual servo control: Position-Based Visual Servo (PBVS) and Image-Based Visual Servo (IBVS). Position-based visual servoing, shown in Fig. 15.2a, uses observed visual features, a calibrated camera and a known geometric model of the target to determine the pose of the target with respect to the camera. The robot then moves toward that pose and the control is performed in task space which is commonly  $SE(3)$ . Good algorithms exist for pose estimation but it is computationally expensive and relies critically on the accuracy of the camera calibration and the model of the object's geometry. PBVS is discussed in Sect. 15.1.

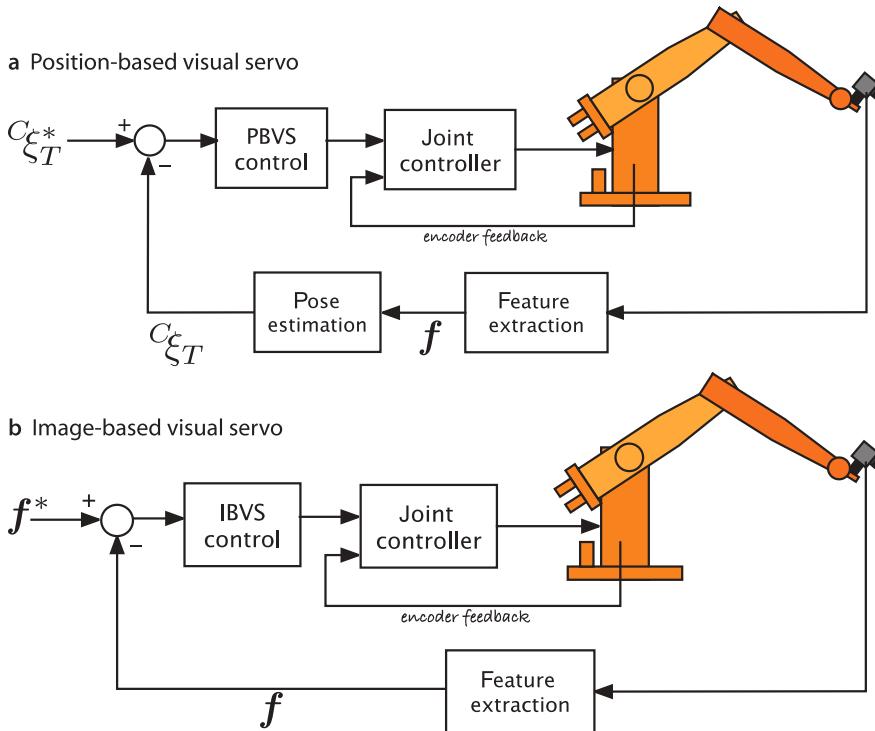
A **servo-mechanism**, or servo is an automatic device that uses feedback of error between the desired and actual position of a mechanism to drive the device to the desired position. The word servo is derived from the Latin root *servus* meaning slave and the first usage was by the Frenchman J.J.L. Farcot in 1868 – “Le Servomoteur” – to describe the hydraulic and steam engines used for steering ships.

Error in position is measured by a sensor then amplified to drive a motor that generates a force to move the device to reduce the error. Servo system development was spurred by WW II with the development of electrical servo systems for fire-control applications that used electric motors and electro-mechanical *amplidyne* power amplifiers. Later servo amplifiers used vacuum tubes and more recently solid state power amplifiers (motor drives). Today servomechanisms are ubiquitous and are used to position the read/write heads in optical and magnetic disk drives, the lenses in autofocus cameras, remote control toys, satellite-tracking antennas, automatic machine tools and robot joints.

“Servo” is properly a noun or adjective but has become a verb “to servo”. In the context of vision-based control we use the verb “visual servoing”.

**Fig. 15.1.** Visual servo configurations and relevant coordinate frames: world, end-effector  $\{E\}$ , camera  $\{C\}$  and target  $\{T\}$ . **a** End-point closed-loop configuration (eye-in-hand); **b** end-point open-loop configuration





**Fig. 15.2.**  
The two distinct classes of visual servo system

Image-based visual servoing, shown in Fig. 15.2b, omits the pose estimation step, and uses the image features directly. The control is performed in image coordinate space  $\mathbb{R}^2$ . The desired camera pose with respect to the target is defined *implicitly* by the image feature values at the goal pose. IBVS is a challenging control problem since the image features are a highly non-linear function of camera pose. IBVS is discussed in Sect. 15.2.

## 15.1 Position-Based Visual Servoing

In a PBVS system the pose of the target with respect to the camera  ${}^C\xi_T$  is estimated. The pose estimation problem was discussed in Sect. 11.2.3 and requires knowledge of the target's geometry, the camera's intrinsic parameters and the observed image plane features. The relationships between the poses is shown in Fig. 15.3. We specify the desired relative pose with respect to the target  ${}^C\xi_T^*$  and wish to determine the motion required to move the camera from its initial pose  $\xi_C$  to  $\xi_C^*$  which we call  $\xi_\Delta$ . The actual pose of the target  $\xi_T$  is not known. From the pose network we can write

$$\xi_\Delta \oplus {}^C\xi_T = {}^C\xi_T^*$$

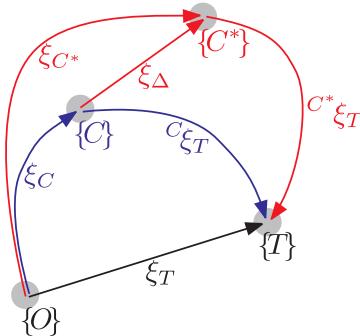
where  ${}^C\xi_T^*$  is the estimated pose of the target relative to the camera. We rearrange this as

$$\xi_\Delta = {}^C\xi_T \ominus {}^C\xi_T^*$$

which is the camera motion required to achieve the desired relative pose. The change in pose might be quite large so we do not attempt to make this movement in one step, rather we move to a point closer to  $\{\mathcal{C}\}$  by

$$\xi_C^{(k+1)} = \xi_C^{(k)} \oplus \lambda \xi_\Delta^{(k)}$$

which is a fraction  $\lambda \in (0, 1)$  of the translation and rotation required.

**Fig. 15.3.**

Relative pose network for PBVS example. Frame  $\{C\}$  is the current camera pose and frame  $\{C^*\}$  is the desired camera pose

Using the Toolbox we start by defining a camera with known parameters

```
>> cam = CentralCamera('default');
```

The target comprises four points that form a square of side length 0.5 m that lies in the  $xy$ -plane and centred at  $(0, 0, 3)$

```
>> P = mkgrid( 2, 0.5, 'T', transl(0,0,3) );
```

and we assume that this pose is unknown to the control system. The camera is at some pose  $T_c$  so the pixel coordinates of the world points are

```
>> p = cam.plot(P, 'Tcam', Tc)
```

from which the pose of the target with respect to the camera  $C\xi_T$  is estimated

```
>> Tc_t_est = cam.estpose(P, p);
```

The required motion  $\xi_\Delta$  is

```
>> Tdelta = TcStar_t * inv(Tc_t_est);
```

and the fractional motion toward the goal is given by

```
>> Tdelta = trinterp(eye(4,4), Tdelta, lambda);
```

giving the new value of the camera pose

```
>> Tc = trnorm(Tc * Tdelta);
```

where we ensure that the transformation remains a proper homogeneous transformation by normalizing it using `trnorm`. At each time step we repeat the process, moving a fraction of the required relative pose until the motion is complete. In this way even if the robot has errors and does not move as requested, or the target moves the motion computed at the next time step will account for that error.

For this example we choose the initial pose of the camera in world coordinates as

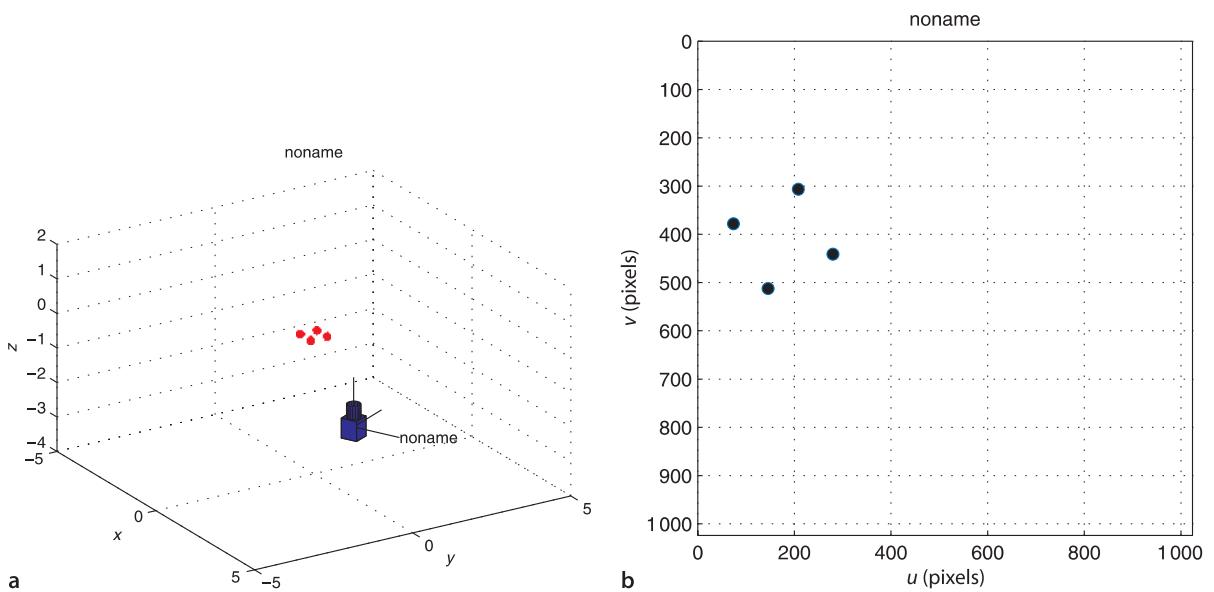
```
>> Tc0 = transl(1,1,-3)*trotz(0.6);
```

and the desired pose of the target with respect to the camera is

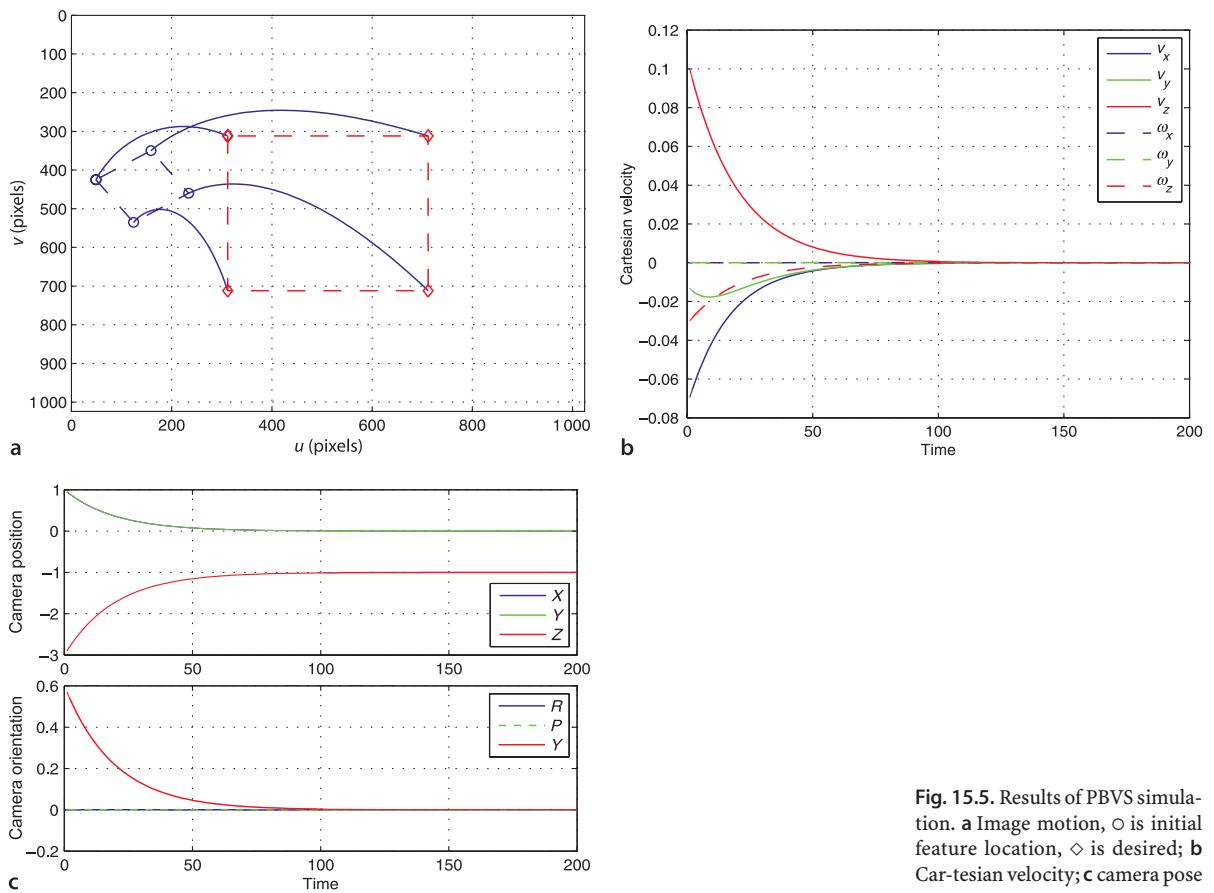
```
>> TcStar_t = transl(0, 0, 1);
```

which has the target 1 m in front of the camera and fronto-parallel to it. We create an instance of the `PBVS` class

```
>> pbvs = PBVS(cam, 'T0', Tc0, 'Tf', TcStar_t)
Visual servo object: camera=noname
  200 iterations, 0 history
P= -0.25      -0.25       0.25       0.25
      -0.25       0.25       0.25      -0.25
       0          0          0          0
Tc0: t = ( 1 1 -3 ), R = ( 34.3775deg |  0 0 1 )
Tc*_t: t = ( 0 0 1 ), R = nil
```



**Fig. 15.4.** Snapshot from the visual servo simulation. **a** An external view showing camera pose and features; **b** camera view showing current feature positions on the image plane



**Fig. 15.5.** Results of PBVS simulation. **a** Image motion,  $\circ$  is initial feature location,  $\diamond$  is desired; **b** Cartesian velocity; **c** camera pose

which is a subclass of the `VisualServo` class and implements the controller outlined above. The object constructor takes a `CentralCamera` object as its argument, and drives this camera to achieve the desired pose relative to the target. Many additional options can be passed to this class constructor. The display methods shows the coordinates of the world points, the initial camera pose, and the desired target relative pose. The simulation is run by

```
>> pbvs.run();
```

which repeatedly calls the `step` method to execute a single time step. The simulation animates both the image plane of the camera and the 3-dimensional visualization of the camera and the world points as shown in Fig. 15.4. The simulation completes after a defined number of iterations or when  $\xi_\Delta$  falls below some threshold.

The simulation results are stored within the object for later analysis. We can plot the path of the target features in the image, the Cartesian velocity versus time or Cartesian position versus time

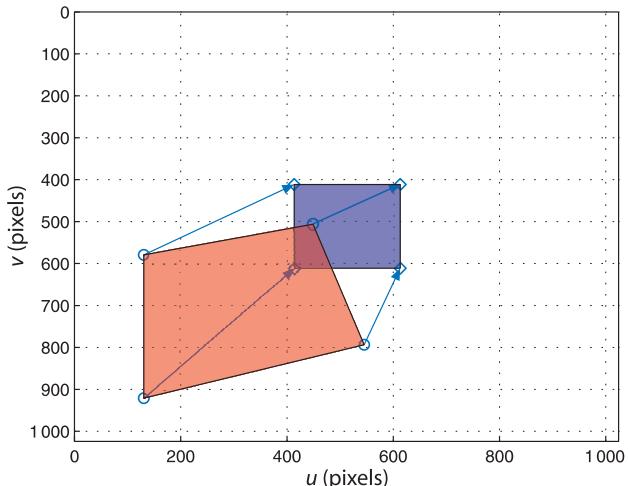
```
>> pbvs.plot_p();
>> pbvs.plot_vel();
>> pbvs.plot_camera();
```

which are shown in Fig. 15.5. We see that the feature points have followed a curved path in the image, and that the camera's translation and orientation have converged smoothly on the desired values.

## 15.2 Image-Based Visual Servoing

IBVS differs fundamentally from PBVS by not estimating the relative pose of the target. The relative pose is implicit in the values of the image features. Figure 15.6 shows two views of a square target. The view from the initial camera pose is shown in red and it is clear that the camera is viewing the target obliquely. The desired view is shown in blue where the camera is further from the target and its optical axis is normal to the plane of the target – a fronto-parallel view.

The control problem can be expressed in terms of image coordinates. The task is to move the feature points indicated by  $\circ$ -markers to the points indicated by  $\diamond$ -markers. The points may, but do not have to, follow the straight line paths indicated by the arrows. Moving the feature points in the image *implicitly* changes the pose – we have changed the problem from pose estimation to control of points on the image.



**Fig. 15.6.**

Two views of a square target. The blue shape is the desired view, and the red shape is the initial view

### 15.2.1 Camera and Image Motion

Consider the default camera

```
>> cam = CentralCamera('default');
```

and a world point at

```
>> P = [1 1 5]';
```

which has image coordinates

```
>> p0 = cam.project( P )
p0 =
    672
    672
```

Now if we displace the camera slightly in the  $x$ -direction the pixel coordinates will become

```
>> px = cam.project( P, 'Tcam', transl(0.1,0,0) )
px =
    656
    672
```

Using the camera coordinate conventions of Fig. 11.4, the camera has moved to the right so the image point has moved to the left. The sensitivity of image motion to camera motion is

```
>> ( px - p0 ) / 0.1
ans =
    -160
        0
```

which is an approximation to the derivative  $\partial\mathbf{p}/\delta_x$ . It shows that 1 m of camera motion would lead to  $-160$  pixel of feature motion in the  $u$ -direction. We can repeat this for  $z$ -axis translation

```
>> ( cam.project( P, 'Tcam', transl(0, 0, 0.1) ) - p0 ) / 0.1
ans =
    32.6531
    32.6531
```

which shows equal motion in the  $u$ - and  $v$ -directions. For  $x$ -axis rotation

```
>> ( cam.project( P, 'Tcam', trotx(0.1) ) - p0 ) / 0.1
ans =
    40.9626
    851.8791
```

the image motion is predominantly in the  $v$ -direction. It is clear that camera motion along and about the different degrees of freedom in  $SE(3)$  causes quite different motion of image points. Earlier we expressed perspective projection in functional form Eq. 11.10

$$\mathbf{p} = \mathcal{P}(\mathbf{P}, \mathbf{K}, \xi_C)$$

and its derivative with respect to camera pose  $\xi$  is

$$\dot{\mathbf{p}} = J_p(\mathbf{P}, \mathbf{K}, \xi_C)\nu$$

where  $\nu = (\nu_x, \nu_y, \nu_z, \omega_x, \omega_y, \omega_z) \in \mathbb{R}^6$  is the velocity of the camera, the spatial velocity, which we introduced in Sect. 8.1.  $J_p$  is a Jacobian-like object, but because we have taken the derivative with respect to a pose  $\xi \in SE(3)$  rather than a vector it is technically called an *interaction matrix*. However in the visual servoing world it is more commonly called an *image Jacobian* or a *feature sensitivity matrix*.

Consider a camera moving with a body velocity  $\nu = (v, \omega)$  in the world frame and observing a world point  $\mathbf{P}$  with camera relative coordinates  $\mathbf{P} = (X, Y, Z)$ . The velocity

of the point relative to the camera frame is

$$\dot{P} = -\omega \times P - v \quad (15.1)$$

which we can write in scalar form as

$$\begin{aligned}\dot{X} &= Y\omega_z Z - \omega_y - v_x \\ \dot{Y} &= Z\omega_x - X\omega_z - v_y \\ \dot{Z} &= X\omega_y - Y\omega_x - v_z\end{aligned}\quad (15.2)$$

The perspective projection Eq. 11.2 for normalized coordinates is

$$x = \frac{X}{Z}, \quad y = \frac{Y}{Z}$$

and the temporal derivative, using the quotient rule, is

$$\dot{x} = \frac{\dot{X}Z - X\dot{Z}}{Z^2}, \quad \dot{y} = \frac{\dot{Y}Z - Y\dot{Z}}{Z^2}$$

Substituting Eq. 15.2,  $X = xZ$  and  $Y = yZ$  we can write this in matrix form

$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} -\frac{1}{Z} & 0 & \frac{x}{Z} & xy & -(1+x^2) & y \\ 0 & -\frac{1}{Z} & \frac{y}{Z} & 1+y^2 & -xy & -x \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \quad (15.3)$$

which relates camera velocity to feature velocity in normalized image coordinates.

The normalized image-plane coordinates are related to the pixel coordinates by Eq. 11.7

$$u = \frac{f}{\rho_u}x + u_0, \quad v = \frac{f}{\rho_v}y + v_0$$

which we rearrange as

$$x = \frac{\rho_u}{f}\bar{u}, \quad y = \frac{\rho_v}{f}\bar{v} \quad (15.4)$$

where  $\bar{u} = u - u_0$  and  $\bar{v} = (v - v_0)$  are the pixel coordinates relative to the principal point. The temporal derivative is

$$\dot{x} = \frac{\rho_u}{f}\dot{\bar{u}}, \quad \dot{y} = \frac{\rho_v}{f}\dot{\bar{v}} \quad (15.5)$$

and substituting Eq. 15.4 and Eq. 15.5 into Eq. 15.3 leads to

$$\begin{pmatrix} \dot{\bar{u}} \\ \dot{\bar{v}} \end{pmatrix} = \underbrace{\begin{pmatrix} -\frac{f}{\rho_u Z} & 0 & \frac{\bar{u}}{Z} & \frac{\rho_u \bar{u} \bar{v}}{f} & -\frac{f^2 + \rho_u^2 \bar{u}^2}{\rho_u f} & \bar{v} \\ 0 & -\frac{f}{\rho_v Z} & \frac{\bar{v}}{Z} & \frac{f^2 + \rho_v^2 \bar{v}^2}{\rho_v f} & -\frac{\rho_v \bar{u} \bar{v}}{f} & -\bar{u} \end{pmatrix}}_{J_p} \begin{pmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \quad (15.6)$$

in terms of pixel coordinates *with respect to the principal point*. We can write this in concise matrix form as

$$\dot{p} = J_p \nu \quad (15.7)$$

where  $J_p$  is the  $2 \times 6$  *image Jacobian* matrix for a point feature.►

The Toolbox `CentralCamera` class provides the method `visjac_p` to compute the image Jacobian and for the example above it is

```
>> J = cam.visjac_p([672; 672], 5)
J =
 -160      0     32     32   -832    160
  0   -160     32    832    -32   -160
```

where the first argument is the coordinate of the point of interest with respect to the image, and the second argument is the depth of the point. The approximate values computed on page 460 appear as columns one, three and four respectively. Image Jacobians can also be derived for line and circle features and these are discussed in Sect. 15.3.

For a given camera velocity, the velocity of the point is a function of the point's coordinate, its depth and the camera parameters. Each column of the Jacobian indicates the velocity of a feature point with respect to the corresponding component of the velocity vector. The `flowfield` method of the `CentralCamera` class shows the feature velocity for a grid of points on the image plane for a particular camera velocity. For camera translational velocity in the  $x$ -direction the flow field is

```
>> cam.flowfield( [1 0 0 0 0 0] );
```

which is shown in Fig. 15.7a. As expected moving the camera to the right causes all the features points to move to the left. The motion of points on the image plane is known as optical flow and can be computed from image sequences as we showed in Sect. 14.8. For translation in the  $z$ -direction

```
>> cam.flowfield( [0 0 1 0 0 0] );
```

the points radiate outward from the principal point as shown in Fig. 15.7e. Rotation about the  $z$ -axis is

```
>> cam.flowfield( [0 0 0 0 0 1] )
```

causes the points to rotate about the principal point as shown in Fig. 15.7f.

Rotational motion about the  $y$ -axis is

```
>> cam.flowfield( [0 0 0 0 1 0] );
```

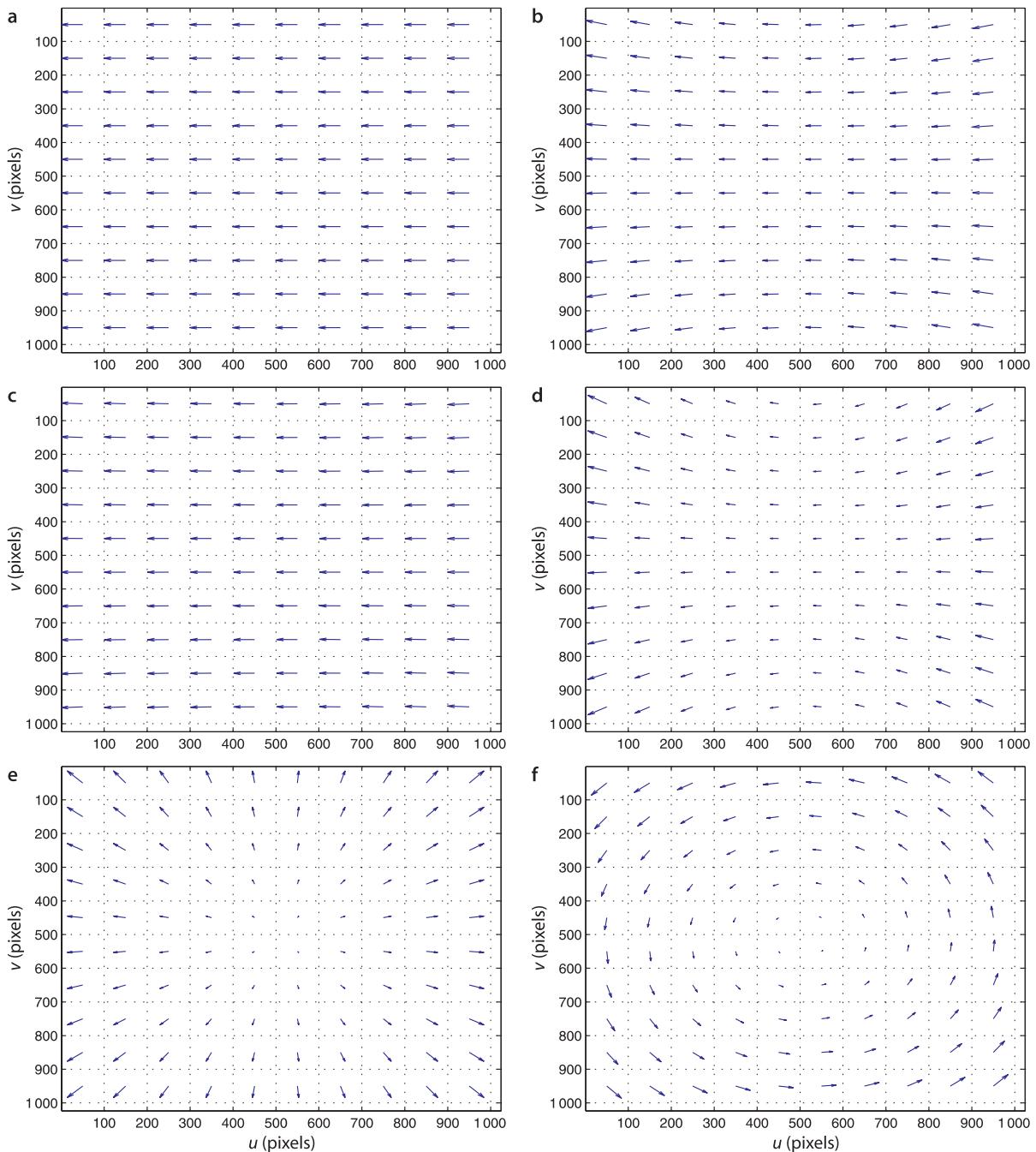
is shown in Fig. 15.7b and is very similar to the case of  $x$ -axis translation, with some small curvature for points far from the principal point. The reason for this is that the first and fifth column of the image Jacobian above are approximately equal which implies that translation in the  $x$ -direction causes almost the same image motion as rotation about the  $y$ -axis. You can easily demonstrate this equivalence by watching how the world moves if you translate your head to the right or rotate your head to the right – in both cases the world appears to move to the left. As the focal length increases the element  $J[2,5]$  becomes smaller and column five approaches a scalar multiple of column one. We can easily demonstrate this by increasing the focal length to  $f = 20$  mm (the default focal length is 8 mm) and the flowfield

```
>> cam.f = 20e-3;
>> cam.flowfield( [0 0 0 0 1 0] );
```

which is shown in Fig. 15.7c is almost identical to that of Fig. 15.7a. Conversely, for small focal lengths (wide-angle cameras) the image motion due to these camera motions will be more dissimilar

```
>> cam.f = 4e-3;
>> cam.flowfield( [0 0 0 0 1 0] );
```

This is commonly written in terms of  $u$  and  $v$  rather than  $\bar{u}$  and  $\bar{v}$  but we use the overbar notation to emphasize that the coordinates are with respect to the principal point, not the image origin which is typically in the top-left corner.



**Fig. 15.7.** Image plane velocity vectors for canonic camera velocities where all corresponding world points lie in a fronto-parallel plane. **a** x-axis translation; **b** y-axis rotation,  $f = 8$  mm; **c** y-axis rotation,  $f = 20$  mm; **d** y-axis rotation,  $f = 4$  mm; **e** z-axis translation; **f** z-axis rotation

and as shown in Fig. 15.7d the curvature is much more pronounced. The same applies for columns two and four except for a difference of sign – there is an equivalence between translation in the  $y$ -direction and rotation about the  $x$ -axis.

The Jacobian matrix has some interesting properties. It does not depend at all on the world coordinates  $X$  or  $Y$ , only on the image plane coordinates  $(u, v)$ . However the first three columns depend on the point's depth  $Z$  and reflects the fact that for a translating camera the image plane velocity is inversely proportional to depth. You can easily demonstrate this to yourself – translate your head sideways and observe that near objects move more in your field of view than distant objects. However, if you rotate your head all objects, near and far, move equally in your field of view.

The matrix has a rank of two,<sup>►</sup> and therefore has a null-space of dimension four. The null-space comprises a set of spatial velocity vectors that individually, or in any linear combination, cause *no motion* in the image. Consider the simple case of a point in front of the camera on the optical axis

```
>> J = cam.visjac_p([512; 512], 1)
```

The null-space of the Jacobian is

```
>> null(J)
ans =
    0         0      -0.7071       0
    0      0.7071        0       0
1.0000      0         0       0
    0      0.7071        0       0
    0         0      0.7071       0
    0         0        0      1.0000
```

The rank cannot be less than 2, even if  $Z \rightarrow \infty$ .

The first column indicates that motion in the  $z$ -direction, along the ray toward the point, results in no motion in the image. Nor does rotation about the  $z$ -axis, as indicated by column four. Columns two and three are more complex, combining rotation and translation. Essentially these exploit the image motion ambiguity mentioned above. Since  $x$ -axis translation causes the same image motion as  $y$ -axis rotation, column three indicates that if one is positive and the other negative the resulting image motion will be zero – that is translating left and rotating to the right.

We can consider the motion of two points by stacking the Jacobians

$$\begin{pmatrix} \dot{u}_1 \\ \dot{v}_1 \\ \dot{u}_2 \\ \dot{v}_2 \end{pmatrix} = \begin{pmatrix} J_{p_1} \\ J_{p_2} \end{pmatrix} \nu$$

to give a  $4 \times 6$  matrix which will have a null-space with two columns. One of these camera motions corresponds to rotation around a line joining the two points.

For three points

$$\begin{pmatrix} \dot{u}_1 \\ \dot{v}_1 \\ \dot{u}_2 \\ \dot{v}_2 \\ \dot{u}_3 \\ \dot{v}_3 \end{pmatrix} = \begin{pmatrix} J_{p_1} \\ J_{p_2} \\ J_{p_3} \end{pmatrix} \nu \quad (15.8)$$

the matrix will be non-singular so long as the points are not coincident or collinear.

### 15.2.2 Controlling Feature Motion

So far we have shown how points move in the image plane as a consequence of camera motion. As is often the case it is the inverse problem that is more useful – *what camera motion is needed in order to move the image features at a desired velocity?*

For the case of three points  $\{(u_i, v_i), i = 1 \dots 3\}$  and corresponding velocities  $\{(\dot{u}_i, \dot{v}_i)\}$  we can invert Eq. 15.8

$$\nu = \begin{pmatrix} J_{p_1} \\ J_{p_2} \\ J_{p_3} \end{pmatrix}^{-1} \begin{pmatrix} \dot{u}_1 \\ \dot{v}_1 \\ \dot{u}_2 \\ \dot{v}_2 \\ \dot{u}_3 \\ \dot{v}_3 \end{pmatrix} \quad (15.9)$$

and solve for the required camera velocity.

Given feature velocity we can compute the required camera motion, but how do we determine the feature velocity? The simplest strategy is to use a simple linear controller

$$\dot{\mathbf{p}}^* = \lambda(\mathbf{p}^* - \mathbf{p}) \quad (15.10)$$

that *drives* the features toward their desired values  $\mathbf{p}^*$  on the image plane. Combined with Eq. 15.9 we write

$$\boldsymbol{\nu} = \lambda \begin{pmatrix} J_{p_1} \\ J_{p_2} \\ J_{p_3} \end{pmatrix}^{-1} (\mathbf{p}^* - \mathbf{p})$$

We do require the depth  $Z$  of the point but we will come to that shortly.

That's it! This controller will drive the camera so that the feature points move toward the desired position in the image. It is important to note that nowhere have we required the pose of the camera or of the object, everything has been computed in terms of what can be measured on the image plane.

For the general case where  $N > 3$  points we can stack the Jacobians for all features and solve for camera motion using the pseudo-inverse

$$\boldsymbol{\nu} = \lambda \begin{pmatrix} J_1 \\ \vdots \\ J_N \end{pmatrix}^+ (\mathbf{p}^* - \mathbf{p}) \quad (15.11)$$

Note that it is possible to specify a set of feature point velocities which are inconsistent, that is, there is no possible camera motion that will result in the required image motion. In such a case the pseudo-inverse will find a solution that minimizes the norm of the feature velocity error.

For  $N \geq 3$  the matrix can be poorly conditioned if the points are nearly co-incident or collinear. In practice this means that some camera motions will cause very small image motions, that is, the motion has low perceptibility. There is strong similarity with the concept of manipulability that we discussed in Sect. 8.1.4 and we take a similar approach in formalizing it. Consider a camera spatial velocity of unit magnitude

$$\boldsymbol{\nu}^T \boldsymbol{\nu} = 1$$

and from Eq. 15.7 we can write the camera velocity in terms of the pseudo-inverse

$$\boldsymbol{\nu} = J_p^+ \dot{\mathbf{p}}$$

and substituting yields

$$\begin{aligned} \dot{\mathbf{p}}^T J_p^{+T} J_p^+ \dot{\mathbf{p}} &= 1 \\ \dot{\mathbf{p}}^T (J_p J_p^T)^{-1} \dot{\mathbf{p}} &= 1 \end{aligned}$$

which is the equation of an ellipsoid in the point velocity space. The eigenvectors of  $J_p J_p^T$  define the principal axes of the ellipsoid and the singular values of  $J_p$  are the radii. The ratio of the maximum to minimum radius is given by the condition number of  $J_p$  and indicates the anisotropy of the feature motion. A high value indicates that some of the points have low velocity in response to some camera motions. An alternative to stacking all the point feature Jacobians is to select just three that when stacked result in the best conditioned square matrix which can then be inverted.

Using the Toolbox we start by defining a camera

```
>> cam = CentralCamera('default');
```

The target comprises four points that form a square of side length 0.5 m that lies in the  $xy$ -plane and centred at  $(0, 0, 3)$

```
>> P = mkgrid( 2, 0.5, 'T', transl(0,0,3) );
```

and we assume that this pose is unknown to the control system. The desired position of the target features on the image plane are a  $400 \times 400$  square centred on the principal point

```
>> pStar = bsxfun(@plus, 200*[-1 -1 1 1; -1 1 1 -1], cam.pp');
```

which implicitly has the square target fronto-parallel to the camera.

The camera is at some pose  $T_c$  so the pixel coordinates of the world points are

```
>> p = cam.plot(P, 'Tcam', Tc)
```

from which we compute the image plane error▶

```
>> e = pStar - p;
```

and the stacked image Jacobian

```
>> J = visjac_p( ci, p, depth );
```

is a  $8 \times 6$  matrix in this case since  $p$  contains four points. The Jacobian does require the point depth which we do not know, so for now we will just choose a constant value. This is an important topic that we will address in Sect. 15.2.3.

The control law determines the required translational and angular velocity of the camera

```
>> v = lambda * pinv(J) * e;
```

where `lambda` is the gain, a positive number, and we take the pseudo-inverse of the non-square Jacobian to implement Eq. 15.11. The resulting velocity is expressed in the camera coordinate frame, and integrating it over a unit time step results in a displacement of the same magnitude. The camera pose is updated by

$$\xi_C(k+1) = \xi_C(k) \oplus \Delta^{-1}(\nu(k))$$

where  $\Delta^{-1}(\cdot)$  is defined by Eq. 3.12. Using the Toolbox this is implemented as

```
>> Tc = trnorm( Tc * delta2tr(v) );
```

where we ensure that the transformation remains a proper homogeneous transformation by normalizing it using `trnorm`.

For this example we choose the initial pose of the camera in world coordinates as

```
>> Tc0 = transl(1,1,-3)*trotz(0.6);
```

Similar to the PBVS example we create an instance of the `IBVS` class

```
>> ibvs = IBVS(cam, 'T0', Tc0, 'pstar', pStar)
```

which is a subclass of the `VisualServo` class and implements the controller outlined above. The option '`T0`' specifies the initial pose of the camera and '`pstar`' specifies the desired image coordinates of the features. The object constructor takes a `CentralCamera` object as its argument, and drives this camera to achieve the desired pose relative to the target. Many additional options can be passed to this class constructor. The display methods shows the coordinates of the world points, the initial absolute pose, the desired image plane feature coordinates. The simulation is run by

```
>> ibvs.run();
```

Note that papers based on the task function approach such as Espiau et al.(1992) write this as actual minus demand and write  $-\lambda$  in Eq. 15.11 to ensure negative feedback.

which repeatedly calls the `step` method to execute a single time step. The simulation animates both the image plane of the camera and the 3-dimensional visualization of the camera and the world points.

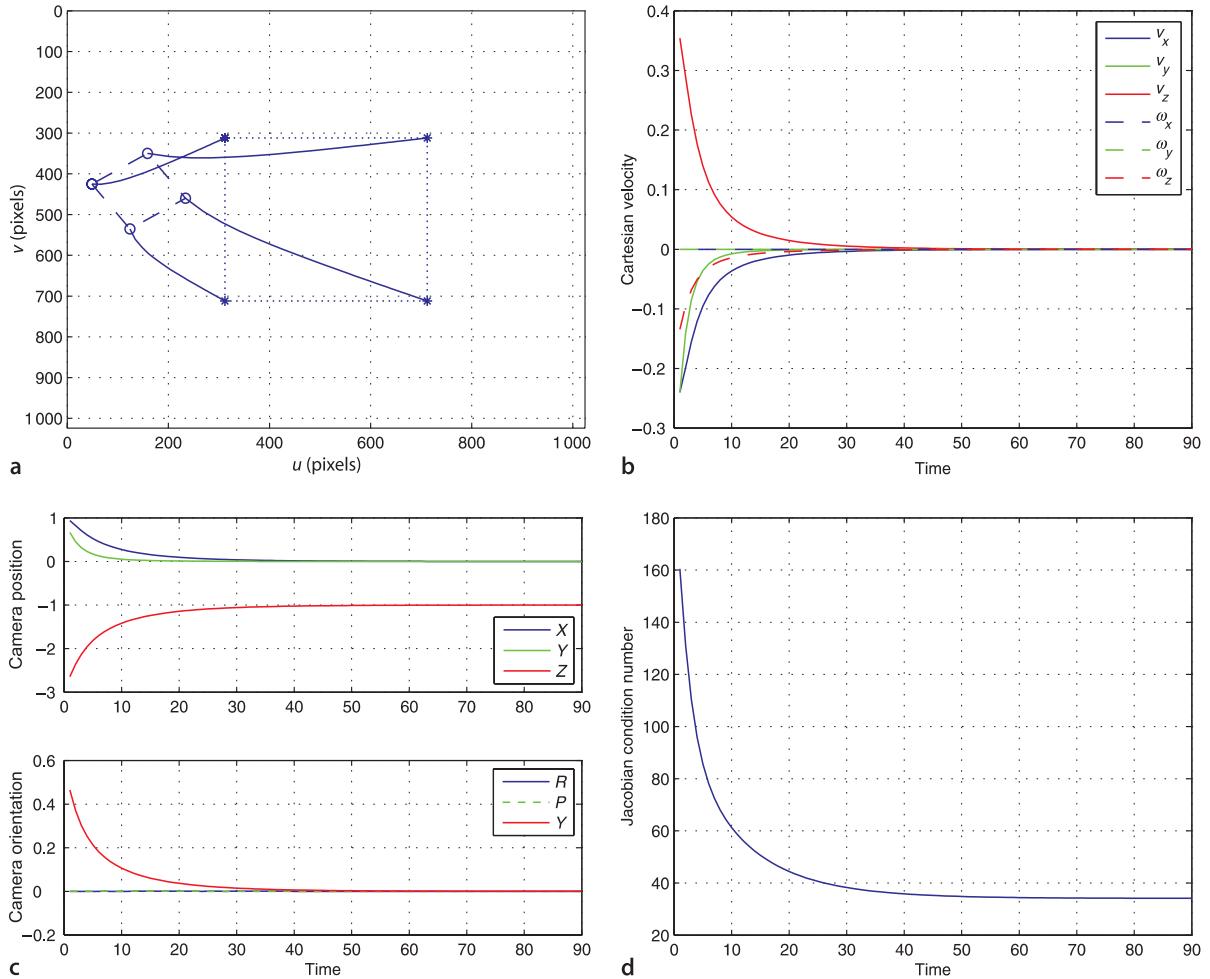
The simulation results are stored within the object for later analysis. We can plot the path of the target features on the image plane, the Cartesian velocity versus time or Cartesian position versus time

```
>> ibvs.plot_p();
>> ibvs.plot_vel();
>> ibvs.plot_camera();
>> ibvs.plot_jcond();
```

which are shown in Fig. 15.8. We see that the feature points have followed an almost straight-line path in the image, and the Cartesian position has changed smoothly toward the final value. The condition number of the image Jacobian decreases over the motion indicating that the Jacobian is becoming better conditioned, and this is a consequence of the features moving further apart.

How is  $p^*$  determined? The image points can be found by demonstration, by moving the camera to the desired pose and recording the observed image coordinates. Alternatively, if the camera calibration parameters and the target geometry are known the desired image coordinates can be computed for any specified goal pose. Note that this calculation, world point point projection, is computationally cheap and is performed only once before visual servoing commences.

**Fig. 15.8.** Results of IBVS simulation, created by `IBVS`. **a** Image plane motion, \* is desired,  $\circ$  is initial, **b** spatial velocity components; **c** camera pose; **d** image Jacobian condition number



The IBVS system can also be expressed in terms of a Simulink® model

```
>> sl_ibvs
```

which is shown in Fig. 15.9. The simulation is run by

```
>> r = sim('sl_ibvs')
```

and the camera pose, image plane feature error and camera velocity are animated. Scope blocks also plot the camera velocity and feature error against time. The initial pose of the camera is set by a parameter of the `pose` block, and the world points are parameters of the `camera` block. The `CentralCamera` object is a parameter to both the `camera` and visual `Jacobian` blocks.

The simulation results are stored in the simulation output object `r`. For example the camera velocity is the second recorded signal▶

```
>> t = r.find('tout');
>> v = r.find('yout').signals(2).values;
>> about(v)
v [double] : 501x6 (24048 bytes)
```

which has one row for every simulation step, and the columns are the camera spatial velocity components. We can plot camera velocity against time

```
>> plot(t, v)
```

The image plane coordinates are also logged

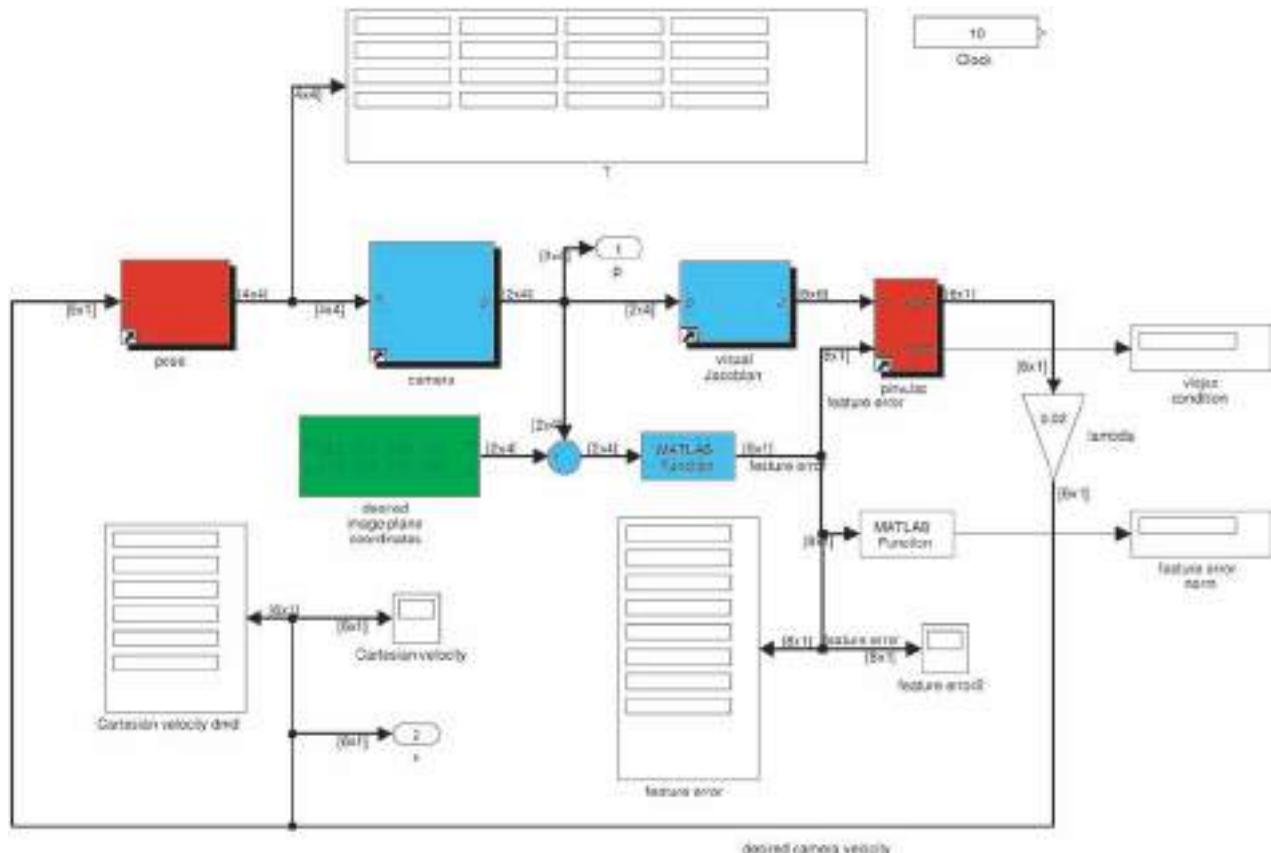
```
>> p = r.find('yout').signals(1).values;
>> about(p)
p [double] : 2x4x1001 (64064 bytes)
```

which can be plotted by

```
>> plot2(p)
```

It is connected to the Output block number 2.

**Fig. 15.9.** The Simulink® model `sl_ibvs` drives the feature points to the desired positions on the image plane. The initial camera pose is set in the `pose` block and the desired image plane points  $p^*$  are set in the green constant block



### 15.2.3 Depth

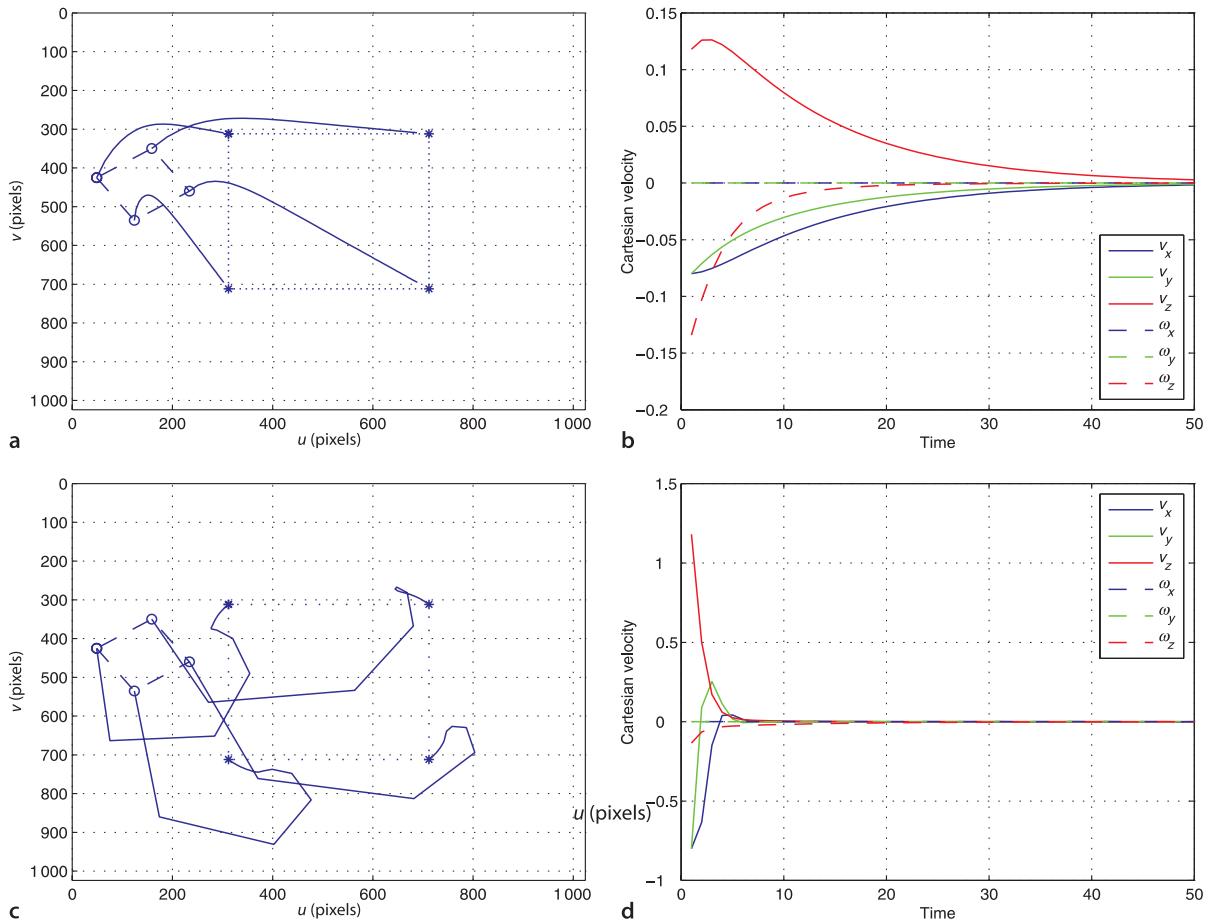
Computing the image Jacobian requires knowledge of the camera intrinsics, the principal point and focal length, but in practice it is quite tolerant to errors in these. The Jacobian also requires knowledge of  $Z_p$ , the distance to, or the depth of, each point. In the simulations just discussed we have assumed that depth is known – this is easy in simulation but not so in reality. Fortunately, in practice we find that IBVS is remarkably tolerant to errors in  $Z$ .

A number of approaches have been proposed to deal with the problem of unknown depth. The simplest is to just assume a constant value for the depth which is quite reasonable if the required camera motion is approximately in a plane parallel to the plane of the object points. To evaluate the performance of different constant estimates of point depth, we can compare the effect of choosing  $z = 1$  and  $z = 10$  for the example above

```
>> ibvs = IBVS(cam, 'T0', Tc0, 'pstar', pStar, 'depth', 1)
>> ibvs.run(50)
>> ibvs = IBVS(cam, 'T0', Tc0, 'pstar', pStar, 'depth', 10)
>> ibvs.run(50)
```

and the results are plotted in Fig. 15.10. We see that the image plane paths are no longer straight, because the Jacobian is now a poor approximation of the relationship between the camera motion and image feature motion. We also see that for  $Z = 1$  the convergence is much slower than for the  $Z = 10$  case. The Jacobian for  $Z = 1$  overestimates the optical flow, so the inverse Jacobian underestimates the required camera velocity. Nevertheless, for quite significant errors, the true depth is  $Z = 3$ , IBVS has converged. For the  $Z = 10$  case the displacement at each timestep is large leading to a very jagged path.

**Fig. 15.10.** Results of IBVS with different constant estimates of point depth: **a, b** Image and camera motion for  $Z = 1$ ; **c, d** Image and camera motion for  $Z = 10$



A second approach is to use standard computer vision techniques to estimate the value for  $Z$ . If the camera intrinsic parameters were known we could use sparse stereo techniques from consecutive camera positions to estimate the depth of each feature point.

A third approach is to estimate the value of  $Z$  online using measurements of robot and image motion. We can create a simple depth estimator by rearranging Eq. 15.6 into estimation form

$$\begin{aligned} \begin{pmatrix} \dot{u} \\ \dot{v} \end{pmatrix} &= \begin{pmatrix} -\frac{f}{\rho_u Z} & 0 & \frac{\bar{u}}{Z} \\ 0 & -\frac{f}{\rho_v Z} & \frac{\bar{v}}{Z} \end{pmatrix} \begin{pmatrix} \frac{\rho_u \bar{u} \bar{v}}{f} & -\frac{f^2 + \rho_u^2 \bar{u}^2}{\rho_u f} & \bar{v} \\ \frac{f^2 + \rho_v^2 \bar{v}^2}{\rho_v f} & -\frac{\rho_v \bar{u} \bar{v}}{f} & -\bar{u} \end{pmatrix} \begin{pmatrix} v \\ \omega \end{pmatrix} \\ &= \left( \frac{1}{Z} J_t \mid J_\omega \right) \begin{pmatrix} v \\ \omega \end{pmatrix} \\ &= \frac{1}{Z} J_t v + J_\omega \omega \end{aligned}$$

which we rearrange as

$$(J_t v) \frac{1}{Z} = \begin{pmatrix} \dot{u} \\ \dot{v} \end{pmatrix} - J_\omega \omega \quad (15.12)$$

The right-hand side is the observed optical flow from which the expected optical flow due to rotation of the camera is subtracted – a process referred to as derotating optical flow. The remaining optical flow, after subtraction, is only due to translation. Writing Eq. 15.12 in compact form

$$A\theta = b \quad (15.13)$$

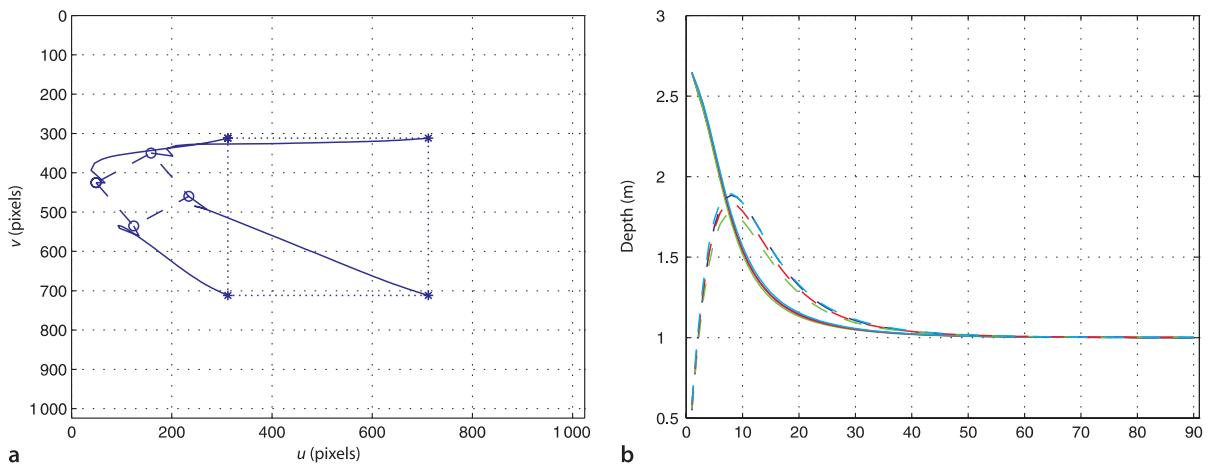
we have a simple linear equation with one unknown parameter  $\theta = 1/Z$  which can be solved using least-squares.

In our example we can enable this by

```
>> ibvs = IBVS(cam, 'T0', Tc0, 'pstar', pStar, 'depthest')
>> ibvs.run()
>> ibvs.plot_z()
>> ibvs.plot_p()
```

and the result is shown in Fig. 15.11. Figure 15.11b shows the estimated and true point depth versus time. The estimate depth was initially zero, a poor choice, but it has risen rapidly and then tracked the actual target depth and then tracked it accurately as the controller converges. Figure 15.11a shows the feature motion, and we see that the features initially move in the wrong direction because of the error in depth.

**Fig. 15.11.** IBVS with online depth estimator. **a** Feature paths; **b** comparison of estimated (dashed) and true depth (solid) for all four points



#### 15.2.4 Performance Issues

The control law for PBVS is defined in terms of the 3-dimensional workspace so there is no mechanism by which the motion of the image features is directly regulated. For the PBVS example shown in Fig. 15.5 the feature points followed a curved path on the image plane, and therefore it is possible that they could leave the camera's field of view. For a different initial camera pose

```
>> pbvs.T0 = transl(-2.1, 0, -3)*trotz(5*pi/4);
>> pbvs.run()
```

the result is shown in Fig. 15.12a and we see that two of the points move outside the image which would cause the PBVS control to fail. By contrast the IBVS control for the same initial pose

```
>> ibvs = IBVS(cam, 'T0', pbvs.T0, 'pstar', pStar, 'lambda',
0.002, 'niter', Inf, 'eterm', 0.5)
>> ibvs.run()
>> ibvs.plot_p();
```

gives the feature trajectories shown in Fig. 15.12b.

Conversely for image-based visual servo control there is no direct control over the Cartesian motion of the camera. This can sometimes result in surprising motion, particularly when the target is rotated about the z-axis

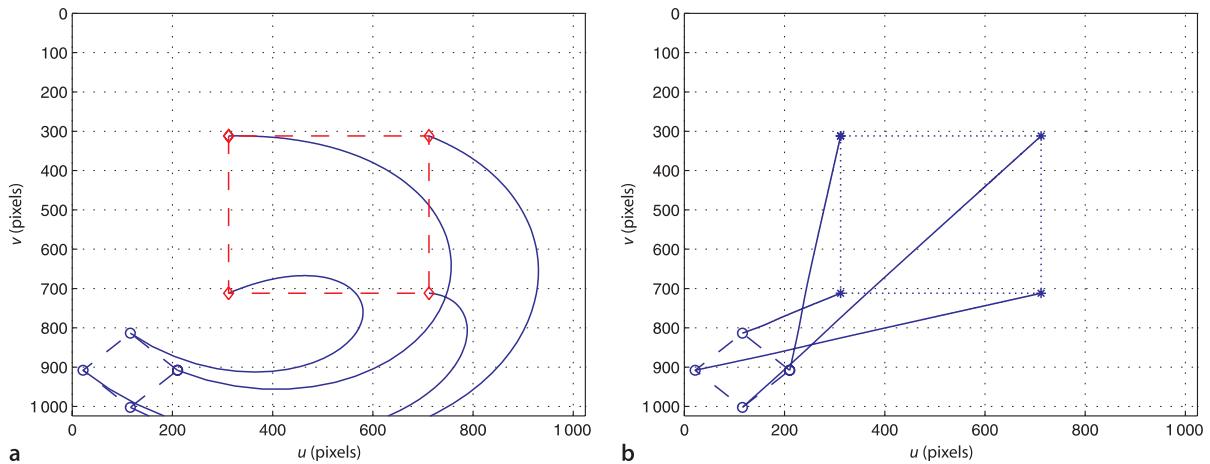
```
>> ibvs = IBVS(cam, 'T0', transl(0,0, -1)*trotz(1), 'pstar',
pStar);
>> ibvs.run()
>> ibvs.plot_camera
```

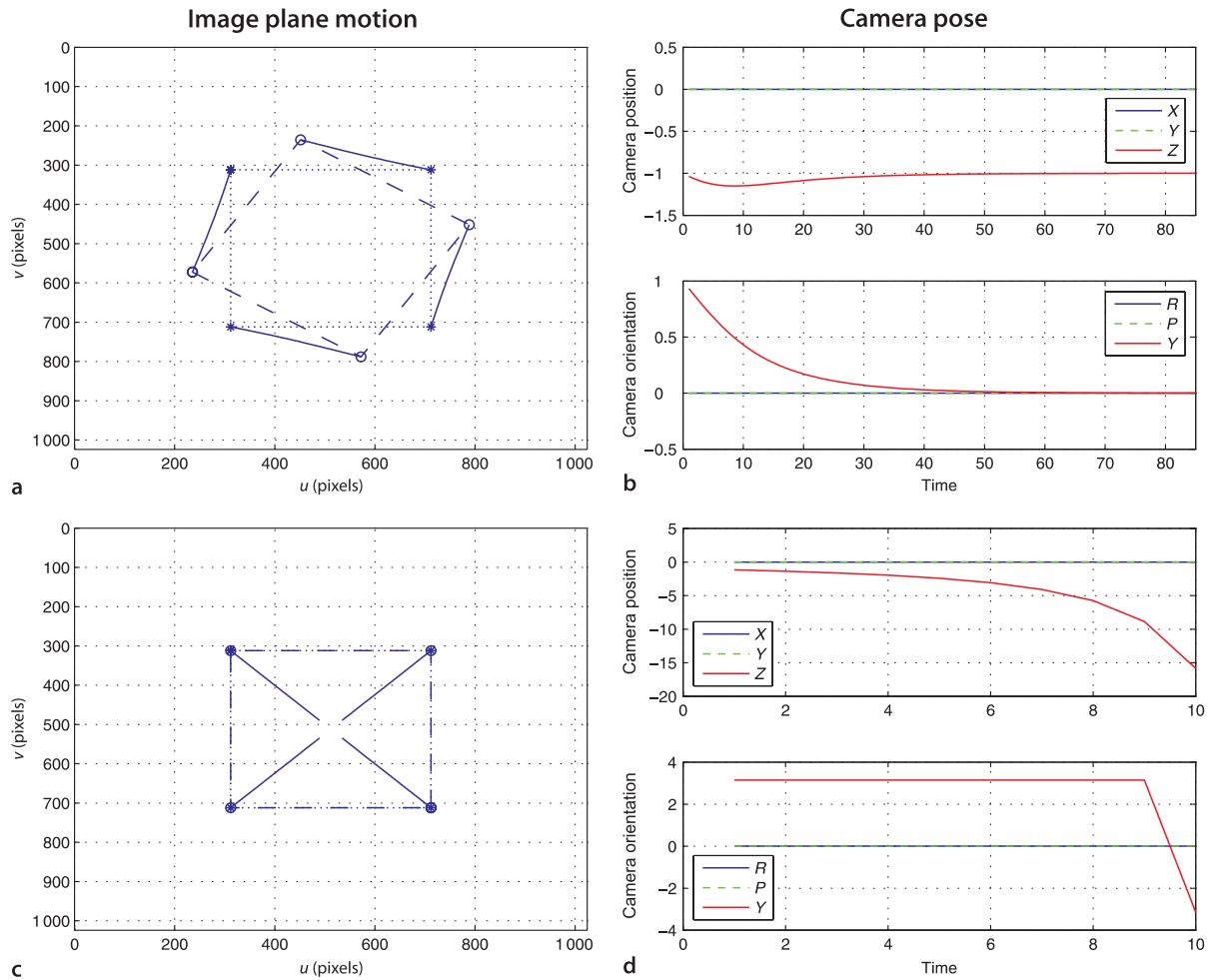
which is shown in Fig. 15.13(top). We see that the camera has performed an unnecessary translation along the z-axis – away from the target and back again. This phenomenon is termed camera retreat. The resulting motion is not time optimal and can require large and possibly unachievable camera motion. An extreme example arises for a pure rotation about the optical axis by  $\pi$  rad

```
>> ibvs = IBVS(cam, 'T0', transl(0,0, -1)*trotz(pi), ...
'pstar', pStar, 'niter', 10);
>> ibvs.run()
>> ibvs.plot_camera
```

which is shown in Fig. 15.13 (bottom). The feature points are, as usual, moving in a straight line toward their desired values, but for this problem the paths all pass through the origin which is a singularity and where IBVS will fail. The only way the target points can be at the origin in the image is if the camera is at negative infinity, and that is where it is headed!

**Fig. 15.12.** Image plane feature paths for **a** PBVS and **b** IBVS





**Fig. 15.13.** IBVS for pure target rotation about the optical axis. **a, b** for rotation of 1 rad; **c, d** for rotation of  $\pi$  rad

A final consideration is that the image Jacobian is a linearization of a highly nonlinear system. If the motion at each time step is large then the linearization is not valid and the features will follow curved rather than linear paths in the image, as we saw in Fig. 15.10. This can occur if the desired feature positions are a long way from the initial positions and/or the gain  $\lambda$  is too high. One solution is to limit the maximum norm of the commanded velocity

$$\nu = \begin{cases} \nu_{\max} \frac{\nu}{|\nu|} & |\nu| > \nu_{\max} \\ \nu & |\nu| \leq \nu_{\max} \end{cases}$$

The feature paths do not have to be straight lines and nor do the features have to move with asymptotic velocity – we have used these only for simplicity. Using the trajectory planning methods of Chap. 3 the features could be made to follow any arbitrary trajectory in the image and to have an arbitrary speed versus time profile.

In summary, IBVS is a remarkably robust approach to vision-based control. We have seen that it is quite tolerant to errors in the depth of points. We have also shown that it can produce less than optimal Cartesian paths for the case of large rotations about the optical axis. We will discuss remedies to these problems in the next chapter.

## 15.3 Using Other Image Features

So far we have considered only point features. In a real system we would use the feature extraction techniques discussed in Chap. 13 and the points would be the centroids of distinct regions, or Harris or SURF corner features. The points would then be used for pose estimation in a PBVS scheme, or directly in an IBVS scheme. For both PBVS or IBVS we need to solve the correspondence problem, that is, for each observed feature we must determine which desired image plane coordinate it corresponds to. IBVS can also be formulated to work with other image features such as lines, as found by the Hough transform, or the shape of an ellipse.

### 15.3.1 Line Features

For a line the Jacobian is written in terms of the  $(\rho, \theta)$  parameterization that we used for the Hough transform in Sect. 13.2

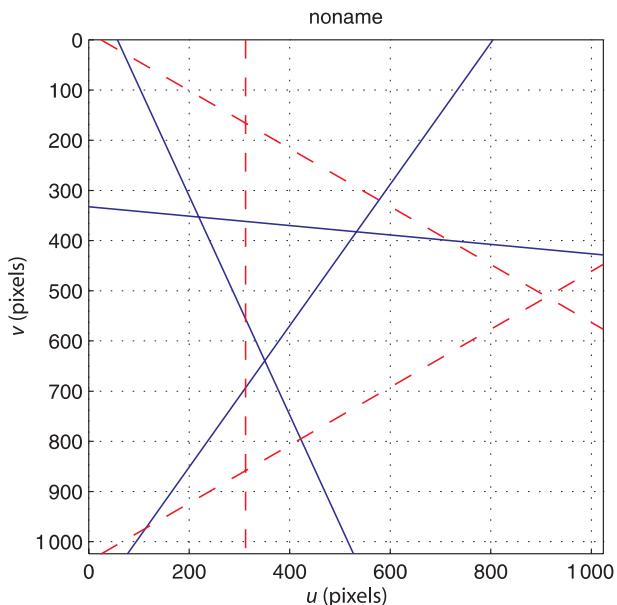
$$\begin{pmatrix} \dot{\theta} \\ \dot{\rho} \end{pmatrix} = J_l \nu$$

and the Jacobian is

$$J_l = \begin{pmatrix} \lambda_\theta \sin \theta & \lambda_\theta \cos \theta & -\rho \lambda_\theta & -\rho \sin \theta & -\rho \cos \theta & -1 \\ \lambda_\rho \sin \theta & \lambda_\rho \cos \theta & -\lambda_\rho \rho & -\cos \theta(1 + \rho^2) & \sin \theta(1 + \rho^2) & 0 \end{pmatrix}$$

where  $\lambda_\theta = (a \cos \theta - b \sin \theta) / d$  and  $\lambda_\rho = -(a \rho \sin \theta + b \rho \cos \theta + c) / d$ . The Jacobian describes how the line parameters change as a function of camera velocity. Just as the point feature Jacobian required some partial 3-dimensional knowledge, the point depth  $Z$ , the line feature Jacobian requires the equation of the plane  $aX + bY + cZ + d = 0$  that contains the line. Since each line is the intersection of two planes and therefore lies in two planes we choose the plane for which  $d \neq 0$ . Like a point feature, a line provides two rows of the Jacobian so we require a minimum of three lines in order to have a Jacobian of full rank. ▶

Interestingly a line feature provides two rows of the stacked Jacobian, yet two points which define a line would provide four rows.



**Fig. 15.14.**

IBVS using line features. The image plane showing the three current lines (solid) and desired (dashed)

We illustrate this with an example comprising three lines that all lie in the plane  $Z = 3$ , and we construct three points in that plane using the `circle` function with just three boundary points

```
>> P = circle([0 0 3], 0.5, 'n', 3);
```

and use the familiar `CentralCamera` class methods to project these to the image. For each pair of points we compute the equations of the line

$$\tan\theta = \frac{v_2 - v_1}{u_1 - u_2}, \rho = u_1 \sin\theta + v_1 \cos\theta$$

The simulation is run in familiar fashion

```
>> ibvs = IBVS_l(cam, 'example');
>> ibvs.run()
```

and a snapshot of results is shown in Fig. 15.14. Note that we need to perform correspondence between the observed and desired lines.

### 15.3.2 Circle Features

A circle in the world will be projected, in the general case, to an ellipse in the image which is described by

$$u^2 + E_1 v^2 - 2E_2 uv + 2E_3 u + 2E_4 v + E_5 = 0 \quad (15.14)$$

where  $E_i$  are parameters of the ellipse. The rate of change of the ellipse coefficients is related to camera velocity by

$$\begin{pmatrix} \dot{E}_1 \\ \dot{E}_2 \\ \vdots \\ \dot{E}_5 \end{pmatrix} = J_e(\mathbf{E}, \rho) \boldsymbol{\nu}$$

where the Jacobian is

$$J_e(\mathbf{E}, \rho) = \begin{pmatrix} 2bE_2 - 2aE_1 & 2E_1(b-aE_2) & 2bE_4 - 2aE_1E_3 & 2E_4 & 2E_1E_3 & -2E_2(E_1+1) \\ b-aE_2 & bE_2 - a(2E_2^2 - E_1) & a(E_4 - 2E_2E_3) + bE_3 & E_3 & 2E_2E_3 - E_4 & E_1 - 2E_2^2 - 1 \\ c-aE_3 & a(E_4 - 2E_2E_3) + cE_2 & cE_3 - a(2E_3^2 - E_5) & -E_2 & 1 + 2E_3^2 - E_5 & E_4 - 2E_2E_3 \\ E_3b + E_2c - 2aE_4 & E_4b + E_1c - 2aE_2E_4 & bE_5 + cE_4 - 2aE_3E_4 & E_5 - E_1 & 2E_3E_4 + E_2 & -2E_2E_4 - E_3 \\ 2cE_3 - 2aE_5 & 2cE_4 - 2aE_2E_5 & 2cA5 - 2aE_3E_5 & -2E_4 & 2E_3E_5 + 2E_3 & -2E_2E_5 \end{pmatrix}$$

This is different to the representation of an ellipse given in Appendix E, but the two forms are simply related by constant scale factors applied to the coefficients.

and where  $\rho = (\alpha, \beta, \gamma)$  defines a plane in world coordinates  $aX + bY + cZ + d = 0$  in which the ellipse lies and  $\alpha = -a/d$ ,  $\beta = -b/d$  and  $\gamma = -c/d$ . Just as was the case for point and line feature Jacobians we need to provide some depth information about the target. The Jacobian normally has a rank of five, but this drops to three when the projection is of a circle centred in the image plane, and a rank of two if the circle is a point.

An advantage of the ellipse feature is that the ellipse can be computed from the set of all boundary points without needing to solve the correspondence problem. The ellipse feature can also be computed from the moments of all the points within the ellipse boundary. We illustrate this with an example of a circle comprising ten points around its circumference

```
>> P = circle([0 0 3], 0.5, 'n', 10);
```

and the `CentralCamera` class projects these to the image plane.

```
>> p = cam.project(P, 'Tcam', Tc);
```

where  $T_c$  is the current camera pose and we convert to normalized image coordinates

```
>> p = homtrans( inv(cam.K), p );
```

The parameters of an ellipse are calculated using the methods of Appendix E

```
>> a = [y.^2; -2*x.*y; 2*x; 2*y; ones(1,numcols(x))]';  
>> b = -(x.^2)';  
>> E = a\b;
```

which returns a 5-vector of ellipse parameters. The image Jacobian for an ellipse feature is computed by a method of the `CentralCamera` class

```
>> J = cam.visjac_e(E, plane);
```

where the plane containing the circle must also be specified. For this example the plane is  $Z = 3$  so `plane = [0 0 1 -3]`.

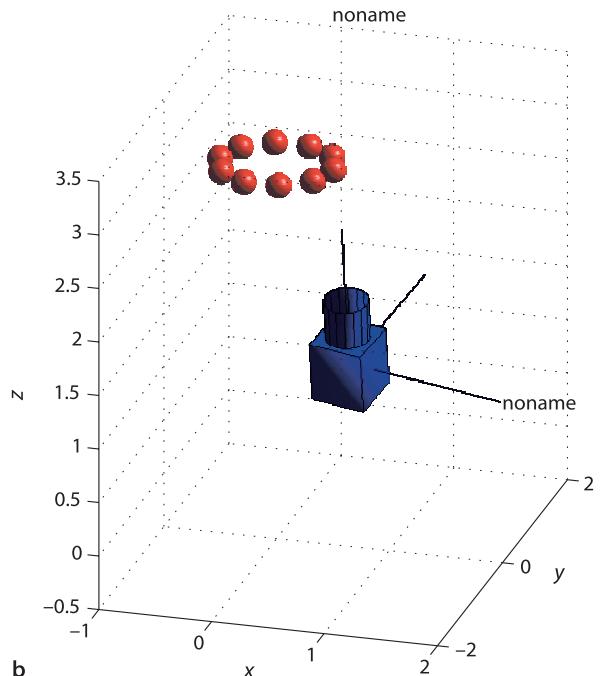
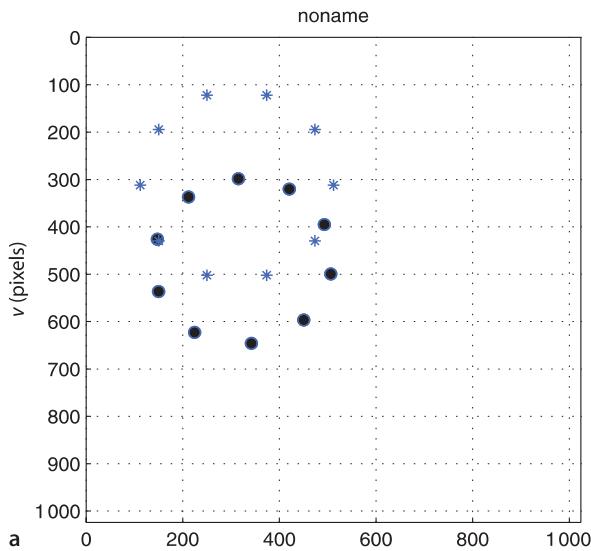
The Jacobian is  $5 \times 6$  and has a maximum rank of only 5 so we cannot uniquely solve for the camera velocity. We have at least two options. Firstly, if our final view is of a circle then we may not be concerned about rotation around the centre of the circle, and in this case we can delete the sixth column of the Jacobian to make it square and set  $\omega_z$  to zero. Secondly, and the approach taken in this example, is to combine the features for the ellipse and a single point

Here we arbitrarily choose the first point, any one will do.

$$\begin{pmatrix} \dot{E}_1 \\ \dot{E}_2 \\ \vdots \\ \dot{E}_5 \\ \dot{u}_1 \\ \dot{v}_1 \end{pmatrix} = \begin{pmatrix} J_e(E) \\ J_p(p_1) \end{pmatrix} \nu$$

**Fig. 15.15.** IBVS using ellipse feature. **a** The image plane showing the current points (solid) and demanded (\*); **b** a world view showing the points and the camera

and the stacked Jacobian is now  $7 \times 6$  and we can solve for camera velocity. As for the previous IBVS examples the desired velocity is proportional to the difference between the current and desired feature values



$$\begin{pmatrix} \dot{E}_1 \\ \dot{E}_2 \\ \vdots \\ \dot{E}_5 \\ \dot{u}_1 \\ \dot{v}_1 \end{pmatrix} = \lambda \begin{pmatrix} E_1^* - E_1 \\ E_2^* - E_2 \\ \vdots \\ E_5^* - E_5 \\ u_1^* - u_1 \\ v_1^* - v_1 \end{pmatrix}$$

The simulation is run in the now familiar fashion

```
>> ibvs = IBVS_e(cam, 'example');
>> ibvs.run()
```

and a snapshot of results is shown in Fig. 15.15.

## 15.4 Wrapping Up

In this chapter we have learnt about the fundamentals of vision-based robot control, and the fundamental techniques developed over two decades up to the mid 1990s. There are two distinct configurations. The camera can be attached to the robot observing the target, eye-in-hand, or fixed in the world observing both robot and target. Another form of distinction is the control structure: Position-Based Visual Servo (PBVS) and Image-Based Visual Servo (IBVS). The former involves pose estimation based on a calibrated camera and a geometric model of the target, while the latter performs the control directly in the image plane. Each approach has certain advantages and disadvantages. PBVS performs efficient straight-line Cartesian camera motion in the world but may cause image features to leave the image plane. IBVS always keeps features in the image plane but may result in trajectories that exceed the reach of the robot, particularly if it requires a large amount of rotation about the camera's optical axis. IBVS also requires a touch of 3-dimensional information, the depth of the feature points, but is quite robust to errors in depth and it is quite feasible to estimate the depth as the robot moves. IBVS can be formulated to work with not only point features, but also for lines and ellipses.

So far in our simulations we have determined the required camera velocity and moved the camera accordingly, without consideration of the mechanism to move it. In the next chapter we consider cameras attached to arm-type robots, mobile ground robots and flying robots.

### Further Reading

The tutorial paper by Hutchinson et al. (1996) was the first comprehensive articulation and taxonomy of the field. More recent articles by Chaumette and Hutchinson (2006) and Siciliano and Khatib (2008, § 24) provide excellent coverage of the fundamentals of visual servoing. Chapters on visual servoing are included in recent textbooks by Spong et al. (2006, § 12) and Siciliano et al. (2008, § 10).

The 1993 book edited by Hashimoto (1993) was the first collection of papers covering approaches and applications in visual servoing. The 1996 book by Corke (1996b) is now out of print but available free online and covers the fundamentals of robotics and vision for controlling the dynamics of an image-based visual servoing system. It contains an extensive, but dated, collection of references to visual servoing applications including industrial applications, camera control for tracking, high-speed planar micro-manipulator, road vehicle guidance, aircraft refuelling, and fruit picking. Another important collection of papers (Kriegman et al. 1998) stems from a 1998 workshop on the synergies between control and vision: how vision can be used for control and how



**Fig. 15.16.**

A 19 inch VMEbus rack of hardware image processing cards, capable of  $10 \text{ Mpix s}^{-1}$  throughput or framerate for  $512 \times 512$  images. Used by the author circa in the early 1990s

control can be used for vision. More recent algorithmic developments and application are covered in a collection of workshop papers by Chesi and Hashimoto (2010).

Visual servoing has a very long history – the earliest reference is by Shirai and Inoue (1973) who describe how a visual feedback loop can be used to correct the position of a robot to increase task accuracy. They demonstrated a system with a servo cycle time of 10 s, and this highlights a harsh reality for the field which has been the problem of real-time feature extraction. Until the late 1990s this required bulky and expensive special-purpose hardware such as that shown in Fig. 15.16. Other significant early work on industrial applications occurred at SRI International during the late 1970s (Hill and Park 1979; Makhlin 1985).

In the 1980s Weiss et al. (1987) introduced the classification of visual servo structures as either position-based or image-based. They also introduced a distinction between visual servo and dynamic look and move, the former uses only visual feedback whereas the latter uses joint feedback and visual feedback. This latter distinction is now longer in common usage and most visual servo systems today make use of joint-position *and* visual feedback. Weiss (1984) applied adaptive control techniques for IBVS of a robot arm without joint-level feedback, but the results were limited to low degree of freedom arms due to the low-sample rate vision processing available at that time. Others have looked at incorporating the manipulator dynamics Eq. 9.1 into controllers that command motor torque directly (Kelly 1996; Kelly et al. 2002a,b) but all still require joint angles in order to evaluate the manipulator Jacobian, and the joint rates to provide damping. Control and stability in closed-loop visual control systems was addressed by several researchers (Corke and Good 1992; Espiau et al. 1992; Papanikolopoulos et al. 1993) and feedforward predictive, rather than feedback, controllers were proposed by Corke (1994) and Corke and Good (1996).

Feddema (Feddema and Mitchell 1989; Feddema 1989) used closed-loop joint control to overcome problems due to low visual sampling rate and demonstrated IBVS for 4-DOF. Chaumette, Rives and Espiau (Chaumette et al. 1991; Rives et al. 1989) describe a similar approach using the task function method (Samson et al. 1990) and show experimental results for robot positioning using a target with four features. Feddema et al. (1991) describe an algorithm to select which subset of the available features give the best conditioned square Jacobian. Hashimoto et al. (1991) have shown that there are advantages in using a larger number of features and using a pseudo-inverse to solve for velocity.

It is well known that IBVS is very tolerant to errors in depth and its effect on control performance is examined in detail in Marey and Chaumette (2008). Feddema and Mitchell (1989) performed a partial 3D reconstruction to determine point depth based on observed features and known target geometry. Papanikolopoulos and Khosla (1993) described adaptive control techniques to estimate depth, as used in this chapter. Hosoda and Asada (1994), Jägersand et al. (1996) and Piepmeyer et al. (1999) have shown how the image Jacobian matrix itself can be estimated online from measurements of robot and image motion.

The most common image Jacobian is based on the motion of points in the image, but it can also be derived for the parameters of lines in the image plane (Chaumette 1990; Espiau et al. 1992) and the parameters of an ellipse in the image plane (Espiau et al. 1992). More recently moments have been proposed for visual servoing of planar scenes (Chaumette 2004; Tahri and Chaumette 2005).

The literature on PBVS is much smaller, but the paper by Westmore and Wilson (1991) is a good introduction. They use an EKF to implicitly perform pose estimation, the target pose is the filter state and the innovation between predicted and feature coordinates updates the target pose state. Hashimoto et al. (1991) present simulations to compare position-based and image-based approaches.

Visual servoing has been applied to a diverse range of problems that normally require human hand-eye skills such as ping-pong (Andersson 1989), juggling (Rizzi and Koditschek 1991) and inverted pendulum balancing (Dickmanns and Graefe 1988a; Andersen et al. 1993), catching (Sakaguchi et al. 1993; Buttazzo et al. 1993; Bukowski et al. 1991; Skofteland and Hirzinger 1991; Skaar et al. 1987; Lin et al. 1989), and controlling a labyrinth game (Andersen et al. 1993).

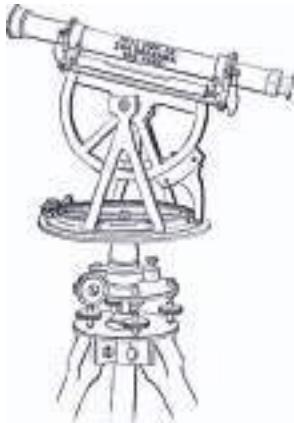
## Exercises

1. Position-based visual servoing
  - a) Run the PBVS example. Experiment with varying parameters such as the initial camera pose, the path fraction  $\lambda$  and adding pixel noise to the output of the camera.
  - b) Create a Simulink® model for PBVS.
  - c) Use a different camera model for the pose estimation (slightly different focal length or principal point) and observe the effect on final end-effector pose.
  - d) Implement an EKF based PBVS system as described in Westmore and Wilson (1991).
2. Optical flow fields
  - a) Plot the optical flow fields for cameras with different focal lengths.
  - b) Plot the flow field for some composite camera motions such as  $x$ - and  $y$ -translation,  $x$ - and  $z$ -translation, and  $x$ -translation and  $z$ -rotation.
3. For the case of two points the image Jacobian is  $4 \times 6$  and the nullspace has two columns. What camera motions do they correspond to?
4. Image-based visual servoing
  - a) Run the IBVS example, either command line or Simulink® version. Experiment with varying the gain  $\lambda$ . Remember that  $\lambda$  can be a scalar or a diagonal matrix which allows different gain settings for each degree of freedom.
  - b) Implement the function to limit the maximum norm of the commanded velocity.
  - c) Experiment with adding pixel noise to the output of the camera.
  - d) Experiment with different initial camera poses and desired image plane coordinates.
  - e) Experiment with different number of target points, from three up to ten. For the cases where  $N > 3$  compare the performance of the pseudo-inverse with just selecting a subset of three points (first three or random three). Can you design an algorithm that chooses a subset of points which results in the stacked Jacobian with the best condition number?
  - f) Create a set of desired image plane points that form a rectangle rather than a square. There is no perspective viewpoint from which a square appears as a rectangle. What does the IBVS system do?
  - g) Create a set of desired image plane points that cannot be reached, for example swap two adjacent world or image points. What does the IBVS system do?
  - h) Use a different camera model for the image Jacobian (slightly different focal length or principal point) and observe the effect on final end-effector pose.

- i) For IBVS we generally force points to move in straight lines but this is just a convenience. Use a trajectory generator to move the points from initial to desired position with some sideways motion, perhaps a half or full cycle of a sine wave. What is the effect on camera Cartesian motion?
- 5. Derive the image Jacobian for a pan/tilt camera head.
- 6. When discussing motion perceptibility we used the identity  $(J_p^+)^T J_p^+ = (J_p J_p^T)^{-1}$ . Prove this. Hint, use the singular value decomposition  $J = U \Sigma V^T$  and remember that  $U$  and  $V$  are orthogonal matrices.
- 7. End-point open-loop visual servo systems have not been discussed in this book. Consider a group of target points on the robot end-effector as well as the those on the target object, both being observed by a single camera (challenging).
  - a) Create an end-point open-loop PBVS system.
  - b) Use a different camera model for the pose estimation (slightly different focal length or principal point) and observe the effect on final end-effector relative pose.
  - c) Create an end-point open-loop IBVS system.
  - d) Use a different camera model for the image Jacobian (slightly different focal length or principal point) and observe the effect on final end-effector relative pose.
- 8. Run the line-based visual servo example.
- 9. Ellipse-based visual servo
  - a) Run the ellipse-based visual servo example.
  - b) Modify to servo five degrees of camera motion using just the ellipse parameters (without the point feature).
  - c) For an arbitrary shape we can compute its equivalent ellipse which is expressed in terms of an inertia matrix and a centroid. Determine the ellipse parameters of Eq. 15.14 from the inertia matrix and centroid. Create an ellipse-feature visual servo to move to a desired view of the arbitrary shape (challenging).

# 16

# Advanced Visual Servoing



This chapter builds on the previous one and introduces some advanced visual servo techniques and applications. Section 16.1 introduces a hybrid visual servo method that avoids some of the limitations of the IBVS and PBVS schemes described previously.

Wide-angle cameras such as fisheye lenses and catadioptric cameras have significant advantages for visual servoing. Section 16.2 shows how IBVS can be reformulated for polar rather than Cartesian image-plane coordinates. This is directly relevant to fisheye lenses but also gives improved rotational control when using a perspective camera. The unified imaging model from Sect. 11.4 allows most cameras (perspective, fisheye and catadioptric) to be represented by a spherical projection model, and Sect. 16.3 shows how IBVS can be reformulated for spherical coordinates.

The remaining sections present a number of application examples. These illustrate how visual servoing can be used with different types of cameras (perspective and spherical) and different types of robots (arm-type robots, mobile ground robots and flying robots). Section 16.4 considers a 6 degree of freedom robot arm manipulating the camera. Section 16.5 considers a mobile robot moving to a specific pose which could be used for navigating through a doorway or docking. Finally, Sect. 16.6 considers visual servoing of a quadcopter flying robot to hover at fixed pose with respect to a target on the ground.

## 16.1 XY/Z-Partitioned IBVS

In the last chapter we encountered the problem of camera retreat in an IBVS system. This phenomenon can be explained intuitively by the fact that our IBVS control law causes feature points to move in straight lines on the image plane, but for a rotating camera the points will naturally move along circular arcs. The linear IBVS controller dynamically changes the overall image scale so that motion along an arc appears as motion along a straight line. The scale change is achieved by  $z$ -axis translation.

Partitioned methods eliminate camera retreat by using IBVS to control some degrees of freedom while using a different controller for the remaining degrees of freedom. The XY/Z hybrid schemes consider the  $x$ - and  $y$ -axes as one group, and the  $z$ -axes as another group. The approach is based on a couple of insights. Firstly, and intuitively, the camera retreat problem is a  $z$ -axis phenomenon:  $z$ -axis rotation leads to unwanted  $z$ -axis translation. Secondly, from Fig. 15.7, the image plane motion due to  $x$ - and  $y$ -axis translational and rotation motion are quite similar, whereas the optical flow due to  $z$ -axis rotation and translation are radically different.

We partition the point-feature optical flow of Eq. 15.7 so that

$$\dot{p} = J_{xy} \nu_{xy} + J_z \nu_z \quad (16.1)$$

where  $\nu_{xy} = (v_x, v_y, \omega_x, \omega_y)$ ,  $\nu_z = (v_z, \omega_z)$ , and  $J_{xy}$  and  $J_z$  are respectively columns {1, 2, 4, 5} and {3, 6} of  $J_p$ . Since  $\nu_z$  will be computed by a different controller we can write Eq. 16.1 as

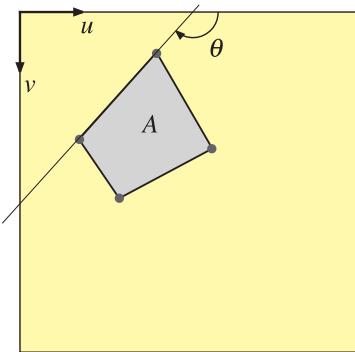
**Fig. 16.1.**

Image features for XY/Z partitioned IBVS control. As well as the coordinates of the four points (blue dots), we use the polygon area  $A$  and the angle of the longest line segment  $\theta$

$$\nu_{xy} = \lambda J_{xy}^+ (\dot{p}^* - J_z \nu_z) \quad (16.2)$$

where  $\dot{p}^*$  is the desired feature point velocity as in the traditional IBVS scheme Eq. 15.10.

The  $z$ -axis velocities  $\nu_z$  and  $\omega_z$  are computed directly from two additional image features  $A$  and  $\theta$  that are shown in Fig. 16.1. The first image feature  $\theta \in [0, 2\pi]$ , is the angle between the  $u$ -axis and the directed line segment joining feature points  $i$  and  $j$ . For numerical conditioning it is advantageous to select the longest line segment that can be constructed from the feature points, and allowing that this may change during the motion as the feature point configuration changes. The desired rotational rate is obtained using a simple proportional control law

$$\omega_z^* = \lambda_{\omega_z} (\theta^* \ominus \theta)$$

where the operator  $\ominus$  indicates modulo- $2\pi$  subtraction. As always with motion on a circle there are two directions to move to achieve the goal. If the rotation is limited, for instance by a mechanical stop, then the sign of  $\omega_z$  should be chosen so as to avoid motion through that stop.

The second image feature that we use is a function of the area  $A$  of the regular polygon whose vertices are the image feature points. The advantages of this measure are: it is a scalar; it is rotation invariant► thus decoupling camera rotation from  $Z$ -axis translation; and it can be cheaply computed. The area of the polygon is just the zeroth-order moment,  $m_{00}$  which can be computed using the Toolbox function `mpq_poly(p, 0, 0)`. The feature for control is the square root of area

$$\sigma = \sqrt{m_{00}}$$

which has units of length, in pixels. The desired camera  $z$ -axis translation rate is obtained using a simple proportional control law

$$\nu_z^* = \lambda_{\nu_z} (\sigma^* - \sigma) \quad (16.3)$$

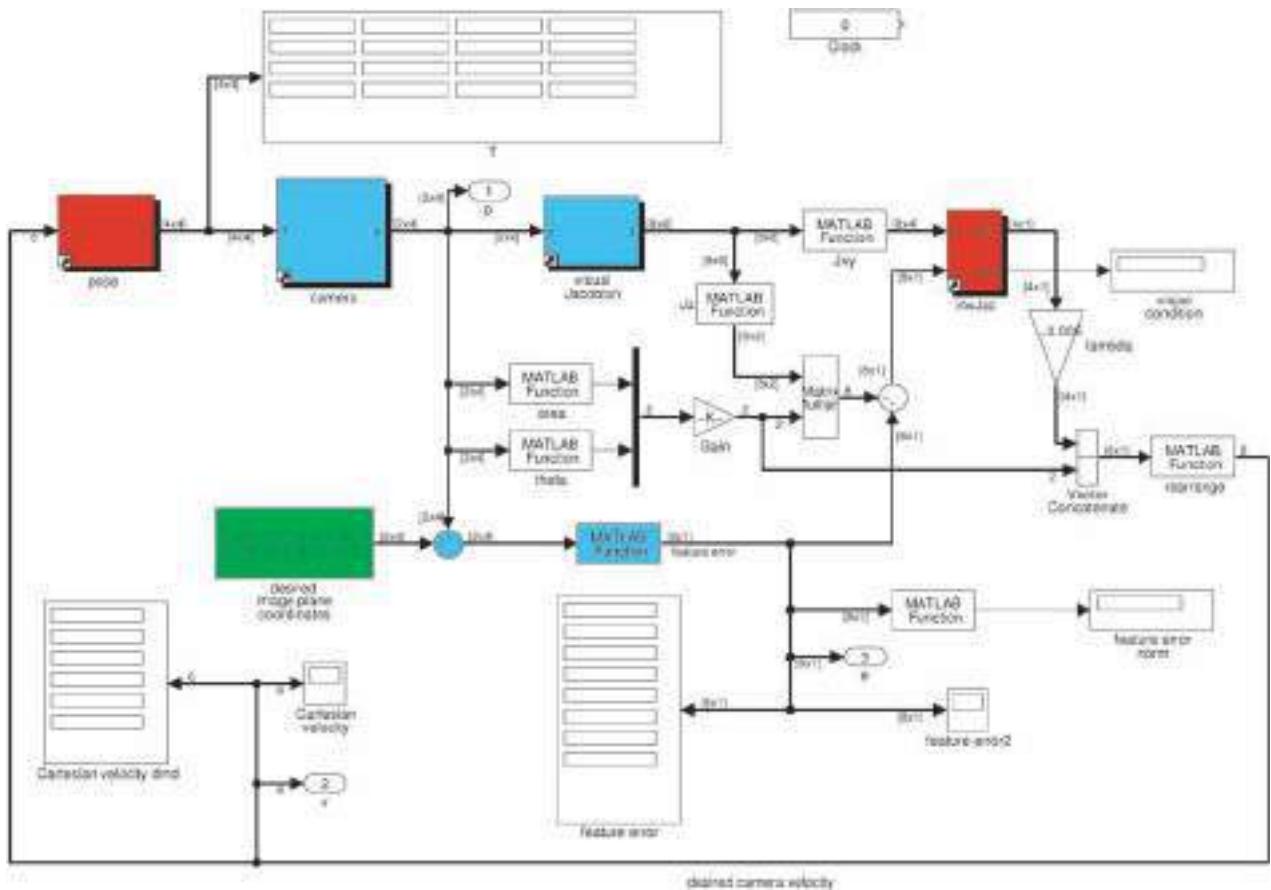
The features discussed above for  $z$ -axis translation and rotation control are simple and inexpensive to compute, but work best when the target normal is within  $\pm 40^\circ$  of the camera's optical axis. When the target plane is not orthogonal to the optical axis its area will appear diminished, due to perspective, which causes the camera to initially approach the target. Perspective will also change the perceived angle of the line segment which can cause small, but unnecessary,  $z$ -axis rotational motion.

The Simulink® model

```
>> sl_xyzpartitioned
```

is shown in Fig. 16.2. The initial pose of the camera is set by a parameter of the `pose` block. The simulation is run by

Rotationally invariant to rotation about the  $z$ -axis, not the  $x$ - and  $y$ -axes.



**Fig. 16.2.** The Simulink® model `sl_partitioned` is an XY/Z-partitioned visual servo scheme, an extension of the IBVS system shown in Fig. 15.9. The initial camera pose is set in the `pose` block and the desired image plane points  $p^*$  are set in the green constant block

```
>> sim('sl_partitioned')
```

and the camera pose, image plane feature error and camera velocity are animated. Scope blocks also plot the camera velocity and feature error against time.

If points are moving toward the edge of the field of view the simplest way to keep them in view is to move the camera away from the scene. We define a repulsive force that acts on the camera, pushing it away as a point approaches the boundary of the image plane

$$F_z(\mathbf{p}) = \begin{cases} \left( \frac{1}{d(\mathbf{p})} - \frac{1}{d_0} \right) \frac{1}{d^2(\mathbf{p})} & d(\mathbf{p}) \leq d_0 \\ 0 & d(\mathbf{p}) > d_0 \end{cases}$$

where  $d(p)$  is the shortest distance to the edge of the image plane from the image point  $p$ , and  $d_0$  is the width of the image zone in which the repulsive force acts, and  $\eta$  is a scalar gain coefficient. For a  $W \times H$  image

$$d(p) = \min\{u, v, W - u, H - v\} \quad (16.4)$$

The repulsion force is incorporated into the z-axis translation controller

$$\nu_z^* = \lambda_{\nu_z}(\sigma^* - \sigma) - \eta \sum_{i=1}^N F_z(p_i)$$

where  $\eta$  is a gain constant with units of damping. The repulsion force is discontinuous and may lead to chattering where the feature points oscillate in and out of the repulsive force – this can be remedied by introducing smoothing filters and velocity limiters.

## 16.2 IBVS Using Polar Coordinates

In Sect. 15.3 we showed image feature Jacobians for non-point features, but here we will show the point-feature Jacobian expressed in terms of a different coordinate system. In polar coordinates the image point is written  $\mathbf{p} = (r, \phi)$  where  $r$  is the distance of the point from the principal point

$$r = \sqrt{\bar{u}^2 + \bar{v}^2} \quad (16.5)$$

where we recall that  $\bar{u}$  and  $\bar{v}$  are the image coordinates with respect to the principal point rather than the image origin. The angle from the  $u$ -axis to a line joining the principal point to the image point is

$$\phi = \tan^{-1} \frac{\bar{v}}{\bar{u}} \quad (16.6)$$

The two coordinate representations are related by

$$\bar{u} = r \cos \phi, \bar{v} = r \sin \phi \quad (16.7)$$

and taking the derivatives with respect to time

$$\begin{pmatrix} \dot{\bar{u}} \\ \dot{\bar{v}} \end{pmatrix} = \begin{pmatrix} \cos \phi & -r \sin \phi \\ \sin \phi & r \cos \phi \end{pmatrix} \begin{pmatrix} \dot{r} \\ \dot{\phi} \end{pmatrix}$$

and inverting

$$\begin{pmatrix} \dot{r} \\ \dot{\phi} \end{pmatrix} = \begin{pmatrix} \cos \phi & \sin \phi \\ -\frac{1}{r} \sin \phi & \frac{1}{r} \cos \phi \end{pmatrix} \begin{pmatrix} \dot{\bar{u}} \\ \dot{\bar{v}} \end{pmatrix}$$

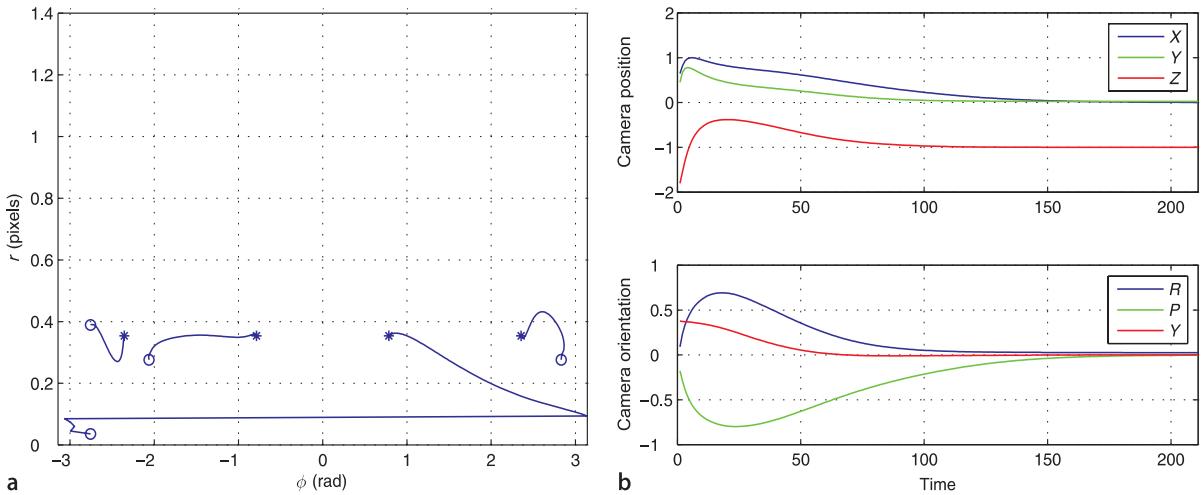
which we substitute into Eq. 15.6 along with Eq. 16.7 to write

$$\begin{pmatrix} \dot{r} \\ \dot{\phi} \end{pmatrix} = J_{p,p} \begin{pmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \quad (16.8)$$

where the feature Jacobian is

$$J_{p,p} = \begin{pmatrix} -\frac{f}{Z} \cos \phi & -\frac{f}{Z} \sin \phi & \frac{r}{Z} & \frac{f^2+r^2}{f} \sin \phi & -\frac{f^2+r^2}{f} \cos \phi & 0 \\ \frac{f}{rZ} \sin \phi & -\frac{f}{rZ} \cos \phi & 0 & \frac{f}{r} \cos \phi & \frac{f}{r} \sin \phi & -1 \end{pmatrix} \quad (16.9)$$

This Jacobian is unusual in that it has three constant elements. In the first row the zero indicates that radius  $r$  is invariant to rotation about the  $z$ -axis. In the second row the zero indicates that polar angle is invariant to translation along the optical axis (points move along radial lines), and the negative one indicates that the angle of a feature (with respect to the  $u$ -axis) decreases with positive camera rotation. As for the Cartesian point features, the translational part of the Jacobian (the first 3 columns) are proportional to  $1/Z$ . Note also that the Jacobian is undefined for  $r=0$ , that is for a point at the image centre. The interaction matrix is computed by the `vis_jac_p_polar` method of the `CentralCamera` class.



**Fig. 16.3.** IBVS using polar coordinates. **a** Feature motion in  $\phi$ - $r$ -space; **b** camera motion in Cartesian space

The desired feature velocity is a function of feature error

$$\dot{\mathbf{p}}^* = \lambda \begin{pmatrix} r^* - r \\ \phi^* \ominus \phi \end{pmatrix}$$

where  $\ominus$  is modulo- $2\pi$  subtraction for the angular component which is implemented by the Toolbox function `angdiff`. Note that  $|r| \gg |\phi|$  and should be normalized

$$r = \frac{\sqrt{\bar{u}^2 + \bar{v}^2}}{\sqrt{W^2 + H^2}}$$

so that  $r$  and  $\phi$  are of approximately the same order.

An example of IBVS using polar coordinates is implemented by the class `IBVS_polar`. We first create a canonic camera, ▶ that has normalized image coordinates

```
>> cam = CentralCamera('default')
>> Tc0 = transl(0, 0, -2)*trotz(1);
>> vs = IBVS_polar(cam, 'T0', Tc0, 'verbose')
```

and we run a simulation

```
>> vs.run()
```

The animation shows the feature motion in the image, and the camera and world points in a world view. The camera motion is quite different compared to the Cartesian IBVS scheme introduced in the previous chapter. For the previously problematic case of large optical-axis rotation the camera has simply moved toward the target and rotated. The features have followed straight line paths on the  $r\phi$ -plane. The performance of polar IBVS is the complement of Cartesian IBVS – it generates good camera motion for the case of large rotation, but poorer motion for the case of large translation.

The methods `plot_error`, `plot_vel` and `plot_camera` can be used to show data recorded during the simulation. An additional method

```
>> vs.plot_features()
```

displays the path of the features in  $\phi$ - $r$ -space and this is shown in Fig. 16.3 along with the camera motion which shows no sign of camera retreat.

The default camera parameters lead to large image coordinate values and hence to large values of  $r$ . The other feature coordinate is an angle  $\phi \in [-\pi, \pi]$  with a much lower value and this leads to numerical problems when solving for camera velocity.

### 16.3 IBVS for a Spherical Camera

In Sect. 11.3 we looked at non-perspective cameras such as the fisheye lens camera and the catadioptric camera. Given the particular projection equations we can derive an image-feature Jacobian from first principles. However the many different lens and mirror shapes leads to many different projection models and image Jacobians. Alternatively we can project the features from any type of camera to the sphere, Sect. 11.5.1, and derive an image Jacobian for visual servo control on the sphere.

The image Jacobian for the sphere is derived in a manner similar to the perspective camera in Sect. 15.2.1. Referring to Fig. 11.20 the world point  $P$  is represented by the vector  $P = (X, Y, Z)$  in the camera frame, and is projected onto the surface of the sphere at the point  $p = (x, y, z)$  by a ray passing through the centre of the sphere

$$x = \frac{X}{R}, \quad y = \frac{Y}{R}, \quad \text{and} \quad z = \frac{Z}{R} \quad (16.10)$$

where  $R = \sqrt{(X^2 + Y^2 + Z^2)}$  is the distance from the camera origin to the world point.

The spherical surface constraint  $x^2 + y^2 + z^2 = 1$  means that one of the Cartesian coordinates is redundant so we will use a minimal spherical coordinate system comprising the angle of colatitude

$$\theta = \sin^{-1} r, \quad \theta \in [0, \pi] \quad (16.11)$$

where  $r = \sqrt{x^2 + y^2}$ , and the azimuth angle (or longitude)

$$\phi = \tan^{-1} \frac{y}{x}, \quad \phi \in [-\pi, \pi] \quad (16.12)$$

which yields the point feature vector  $p = (\theta, \phi)$ .

Taking the derivatives of Eq. 16.11 and Eq. 16.12 with respect to time and substituting Eq. 15.2 as well as

$$X = R \sin \theta \cos \phi, \quad Y = R \sin \theta \sin \phi, \quad Z = R \cos \theta \quad (16.13)$$

we obtain, in matrix form, the spherical optical flow equation

$$\begin{pmatrix} \dot{\theta} \\ \dot{\phi} \end{pmatrix} = J_{p,s}(\theta, \phi, R) \begin{pmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \quad (16.14)$$

where the image feature Jacobian is

$$J_{p,s} = \begin{pmatrix} -\frac{\cos \phi \cos \theta}{R} & -\frac{\sin \phi \cos \theta}{R} & \frac{\sin \theta}{R} & \frac{\sin \phi}{\sin \theta} & -\cos \phi & 0 \\ \frac{\sin \phi}{R \sin \theta} & -\frac{\cos \phi}{R \sin \theta} & 0 & \frac{\cos \phi \cos \theta}{\sin \theta} & \frac{\sin \phi \cos \theta}{\sin \theta} & -1 \end{pmatrix} \quad (16.15)$$

There are similarities to the Jacobian derived for polar coordinates in the previous section. Firstly, the constant elements fall at the same place, indicating that colatitude is invariant to rotation about the optical axis, and that azimuth angle is invariant to translation along the optical axis but equal and opposite to camera rotation about the optical axis. As for all image Jacobians the translational sub-matrix (the first three columns) is a function of point depth  $1/R$ .

The Jacobian is not defined at the north and south poles where  $\sin \theta = 0$  and azimuth also has no meaning at these points. This is a singularity, and as we remarked in Sect. 2.2.1.3, in the context of Euler angle representation of orientation, this is a consequence of using a minimal representation. However, in general the benefits outweigh the costs for this application.

For control purposes we follow the normal procedure of computing one  $2 \times 6$  Jacobian, Eq. 8.2, for each of  $N$  feature points and stacking them to form a  $2N \times 6$  matrix

$$\begin{pmatrix} \dot{\theta}_1 \\ \dot{\phi}_1 \\ \vdots \\ \dot{\theta}_N \\ \dot{\phi}_N \end{pmatrix} = \begin{pmatrix} J_1 \\ J_2 \\ \vdots \\ J_N \end{pmatrix} \nu \quad (16.16)$$

The control law is

$$\nu = J^+ \dot{p}^* \quad (16.17)$$

where  $\dot{p}^*$  is the desired velocity of the features in  $\phi\theta$ -space. Typically we choose this to be proportional to feature error

$$\dot{p}^* = \lambda(\dot{p}^* \ominus p) \quad (16.18)$$

Note that motion on this plane is in general not a great circle on the sphere – only motion along lines of colatitude and the equator are great circles.

where  $\lambda$  is a positive gain,  $p$  is the current point in  $\phi\theta$ -coordinates, and  $\dot{p}^*$  the desired value. This results in locally linear motion of features within the feature space.  $\ominus$  denotes modulo subtraction and returns the smallest angular distance given that  $\theta \in [0, \pi]$  and  $\phi \in [-\pi, \pi]$ .

An example of IBVS using spherical coordinates (Fig. 16.4) is implemented by the class `IBVS_sph`. We first create a spherical camera

```
>> cam = SphericalCamera()
```

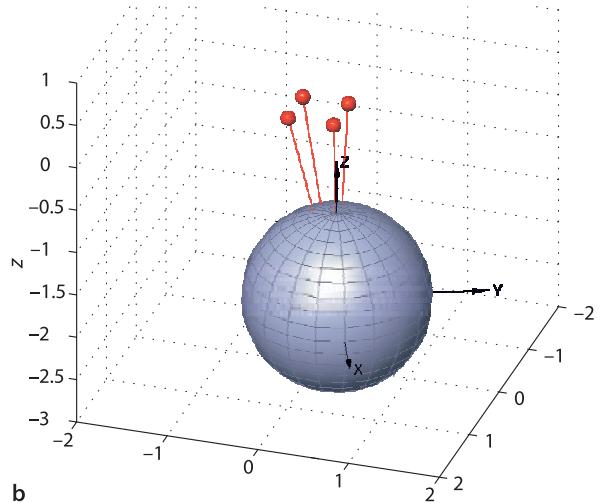
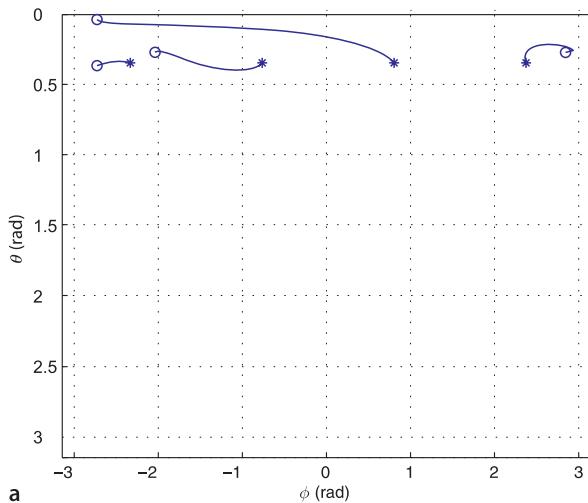
and then a spherical IBVS object

```
>> Tc0 = transl(0.3, 0.3, -2)*trotz(0.4);
>> vs = IBVS_sph(cam, 'T0', Tc0, 'verbose')
```

and we run a simulation for the IBVS failure case

```
>> vs.run()
```

**Fig. 16.4.** IBVS using spherical camera and coordinates. **a** Feature motion in  $\theta - \phi$  space; **b** four target points projected onto the sphere in its initial pose



The animation shows the feature motion on the  $\phi\theta$ -plane and the camera and world points in a world view. Spherical imaging has many advantages for visual servoing. Firstly, a spherical camera eliminates the need to explicitly keep features in the field of view which is a problem with both position-based visual servoing and some hybrid schemes. Secondly, we previously observed an ambiguity between the optical flow fields for  $R_x$  and  $-T_y$  motion (and  $R_y$  and  $-T_x$  motion) for small field of view. For IBVS with a long focal length this can lead to slow convergence and/or sensitivity to noise in feature coordinates. For a spherical camera, with the largest possible field of view, this ambiguity is reduced.

Spherical cameras do not yet exist but we can project features from one or more cameras of any type onto spherical image plane, and compute the control law in terms of spherical coordinates.

Provided that the world points are well distributed around the sphere.

## 16.4 Application: Arm-Type Robot

In this example the camera is carried by a 6-axis robot which can control all six degrees of camera motion. We will assume that the robot's joints are ideal velocity sources, that is, they move at precisely the velocity that was commanded. A modern robot is very close to this ideal, typically having high performance joint controllers using velocity and position feedback from encoders on the joints.

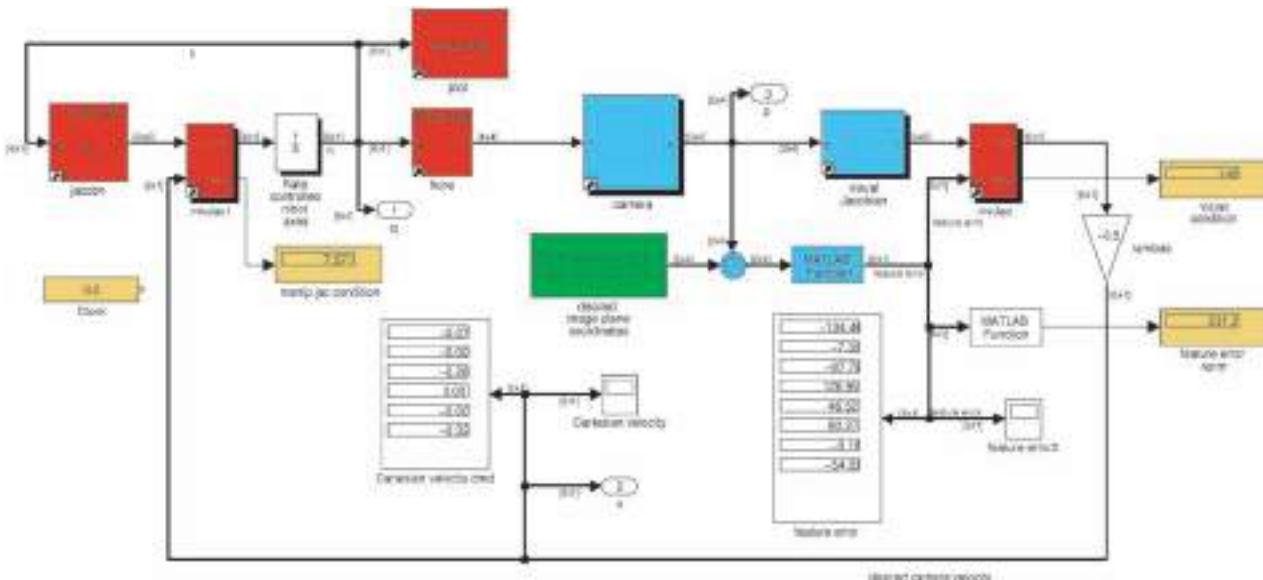
The nested control structure for a robot joint was discussed in Sect. 9.4.2. The inner velocity loop uses joint velocity feedback to ensure that the joint moves at the desired speed. The outer position loop uses joint position feedback to determine the joint speed required to follow the trajectory. In this visual servo system the position loop function is provided by the vision system. Vision sensors have a low sample rate compared to an encoder, typically 25 or 30 Hz, and often with a high latency of one or two sample times.

The Simulink® model of this eye-in-hand system

```
>> sl_arm_ibvs
```

is shown in Fig. 16.5. This is a complex example that simulates not only the camera and IBVS control but also the robot, in this case the ubiquitous Puma 560 from Part III of this book. The joint angles are the outputs of an integrator which represents the robot's velocity loops. These angles are input to a forward kinematics block which outputs the end-effector pose. A perspective camera with default parameters is mounted

**Fig. 16.5.** The Simulink® model `sl_arm_ibvs` drives a Puma robot arm that holds the camera using IBVS



on the robot's end-effector and its axes are aligned with the coordinate frame of the robot's end-effector. The camera's parameters include the `CentralCamera` object and the world coordinates of the target points which are the corners of a square in the  $yz$ -plane. The image features are used to compute a Jacobian with an assumed  $Z$  value for every point, and also to determine the feature error in image space. The image Jacobian is inverted and a gain applied to determine the spatial velocity of the camera. The inverse manipulator Jacobian maps this to joint rates which are integrated to determine joint angles. This closed loop system drives the robot to the desired pose with respect to a square target.

We run this model

```
>> r = sim('sl_ibvs')
```

which displays the robot moving and the image plane of a virtual camera. The signals at the various output blocks are stored in the object `r` and the joint angles at each time step, output port one, are

```
>> q = squeeze(out.find('yout').signals(1).values)';
>> about(q)
q [double] : 60x6 (2880 bytes)
```

Note that this model does not include any dynamics of the robot arm or the vision system. The joints are modelled as perfect velocity control devices, and the vision system is modelled as having no delay. This model could form the basis of more realistic system models that incorporate these real-world effects.

## 16.5 Application: Mobile Robot

In this section we consider a camera mounted on a mobile robot moving in a planar environment. We will first consider a holonomic robot, that is one that has an omnidirectional base and can move in any direction, and then extend the solution to a non-holonomic car-like base which touches on some of the issues discussed in Chap. 4. The camera observes two or more point landmarks that have known 3-dimensional coordinates, that is, they can be placed above the plane on which the robot operates. The visual servo controller will drive the robot until its view of the landmarks matches the desired view.

### 16.5.1 Holonomic Mobile Robot

For this problem we assume a central perspective camera fixed to the robot and a number of landmarks with known locations that are continuously visible to the camera as the robot moves along the path. The vehicle's coordinate frame is such that the  $x$ -axis is forward and the  $z$ -axis is upward.

We define a perspective camera

```
>> cam = CentralCamera('default', 'focal', 0.002);
```

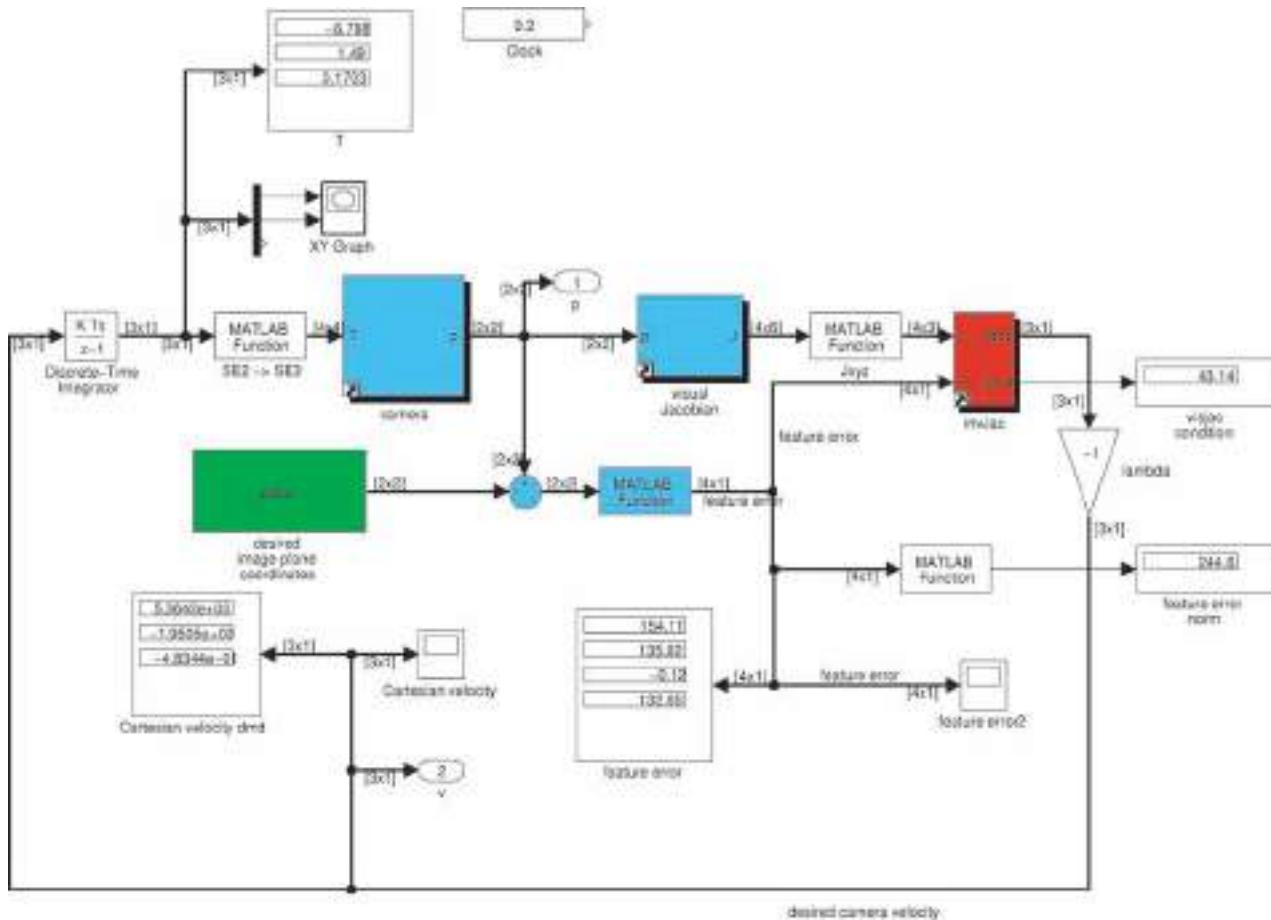
with a wide field of view so that it can keep the landmarks in view as it moves. The camera is mounted on the vehicle with an offset  ${}^V\xi_C$  of

```
>> T_vc = transl(0.2, 0.1, 0.3)*trotx(-pi/4);
```

relative to the vehicle coordinate frame. This is forward of the rear axle, to the left of the vehicle centre line, 30 cm above ground level, with its optical axis forward but pitched upward at  $45^\circ$ , and its  $x$ -axis pointing to the right of the vehicle. The two landmarks are 2 m above the ground and situated at  $x = 0$  and  $y = \pm 1$  m

```
>> P = [0 0; 1 -1; 2 2]
```

The desired vehicle position is with the centre of the rear axle at  $(-2, 0, 2)$ .



Since the robot operates in the  $xy$ -plane and can rotate only about the  $z$ -axis we can remove the columns from Eq. 15.6 that correspond to non-permissible motion and write

$$\begin{pmatrix} \dot{u} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} -\frac{f}{\rho_u Z} & 0 & \bar{v} \\ 0 & -\frac{f}{\rho_v Z} & -\bar{u} \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ \omega_z \end{pmatrix} \quad (16.19)$$

As for standard IBVS case we stack these Jacobians, one per landmark, and then invert the equation to solve for the vehicle velocity. Since there are only three unknown components of velocity, and each landmark contributes two equations, we need two or more feature points in order to solve for velocity.

The Simulink® model

```
>> sl_mobile_vs
```

is shown in Fig. 16.6 and is similar in principle to earlier models such as Fig. 16.5 and 15.9. The model is simulated by

```
>> r = sim('sl_mobile_vs')
```

and displays an animation of the vehicle's path in the  $xy$ -plane and the camera view. Results are stored in the simulation results object `r` and can be displayed as for previous examples. The parameters and camera are defined in the properties of the model's various blocks.

**Fig. 16.6.** The Simulink® model `sl_mobile_vs` drives a holonomic mobile robot to a pose using IBVS control

Simulink® menu File+Model Properties +Callbacks+PreLoadFcn. These commands are executed once when a model is loaded.

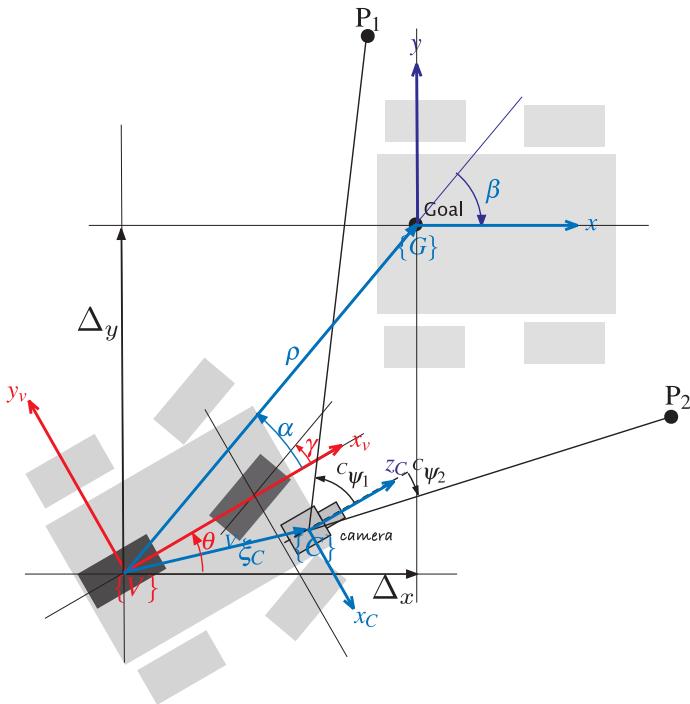


Fig. 16.7.

PBVS for non-holonomic vehicle (bicycle model) vehicle moving toward a goal pose:  $\rho$  is the distance to the goal,  $\beta$  is the angle of the goal vector with respect to the world frame, and  $\alpha$  is the angle of the goal vector with respect to the vehicle frame.  $P_1$  and  $P_2$  are landmarks which are at bearing angles of  $\psi_1$  and  $\psi_2$  with respect to the camera

### 16.5.2 Non-Holonomic Mobile Robot

The difficulties of servoing a hon-holonomic mobile robot to a pose were discussed earlier and a non-linear pose controller was introduced in Sect. 4.2.4. The notation for our problem is shown in Fig. 16.7 and once again we use a controller based on the polar coordinates  $\rho$ ,  $\alpha$  and  $\beta$ . For this control example we will use PBVS techniques to estimate the variables needed for control. We assume a central perspective camera that is fixed to the robot with a relative pose  ${}^V\xi_C$ , a number of landmarks with known locations that are continuously visible to the camera as it moves along the path, and that the vehicle's orientation  $\theta$  is also known, perhaps using a compass or some other sensor.

The Simulink® model

```
>> sl_drivepose_vs
```

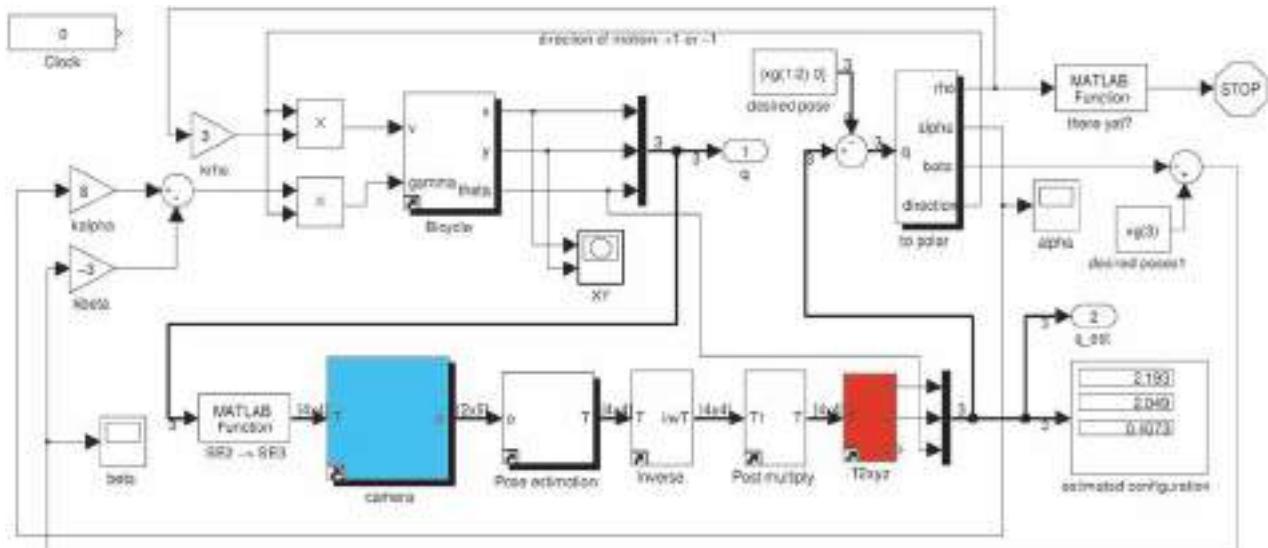
is shown in Fig. 16.8. The initial pose of the camera is set by a parameter of the `Bicycle` block. The view of the landmarks is simulated by the camera block and its output, the projected points, are input to a pose estimation block and the known locations of the landmarks are set as parameters. As discussed in Sect. 11.2.3 at least three landmarks are needed and in this example four landmarks are used. The output is the estimated pose of the landmarks with respect to the camera, but since the landmarks are defined in the world frame  $\{0\}$  the output is  ${}^C\xi_0$ . The vehicle pose in the world frame is obtained by a chain of simple transform operations  $\hat{\xi}_v = \ominus {}^C\xi_0 \ominus {}^V\xi_C$ . The  $x$ - and  $y$ -components of this transform are combined with estimated heading angle  $\hat{\theta}$  to yield an estimate of the vehicle's configuration  $(\hat{x}, \hat{y}, \hat{\theta})$  which is input to the pose controller. The remainder of the system is essentially the same as the example from Fig. 4.14.

The simulation is run by

```
>> r = sim('sl_ibvs')
```

and the camera pose, image plane feature error and camera velocity are animated. Scope blocks also plot the camera velocity and feature error against time. Results are stored in the simulation results object  $r$  and can be displayed as for previous examples.

In a real system heading angle would come from a compass, in this simulation we "cheat" and simply use the true heading angle.



**Fig. 16.8.** The Simulink® model `sl_drivepose_vs` drives a non-holonomic mobile robot to a pose (derived from Fig. 4.13)

## 16.6 Application: Aerial Robot

A spherical camera is particularly suitable for platforms that move in  $SE(3)$  such as aerial and underwater robots. In this example we consider a spherical camera attached to a quadcopter and we will use IBVS to servo the quadcopter to a particular pose with respect to four targets on the ground.

As we discussed in Sect. 4.3 the quadcopter is underactuated and we cannot independently control all 6 degrees of freedom in task space. We can control position ( $X, Y, Z$ ) and also yaw angle. Roll and pitch angle are manipulated to achieve translation in the horizontal plane and must be zero when the vehicle is in equilibrium. The Simulink® model

```
>> sl_quadcopter_vs
```

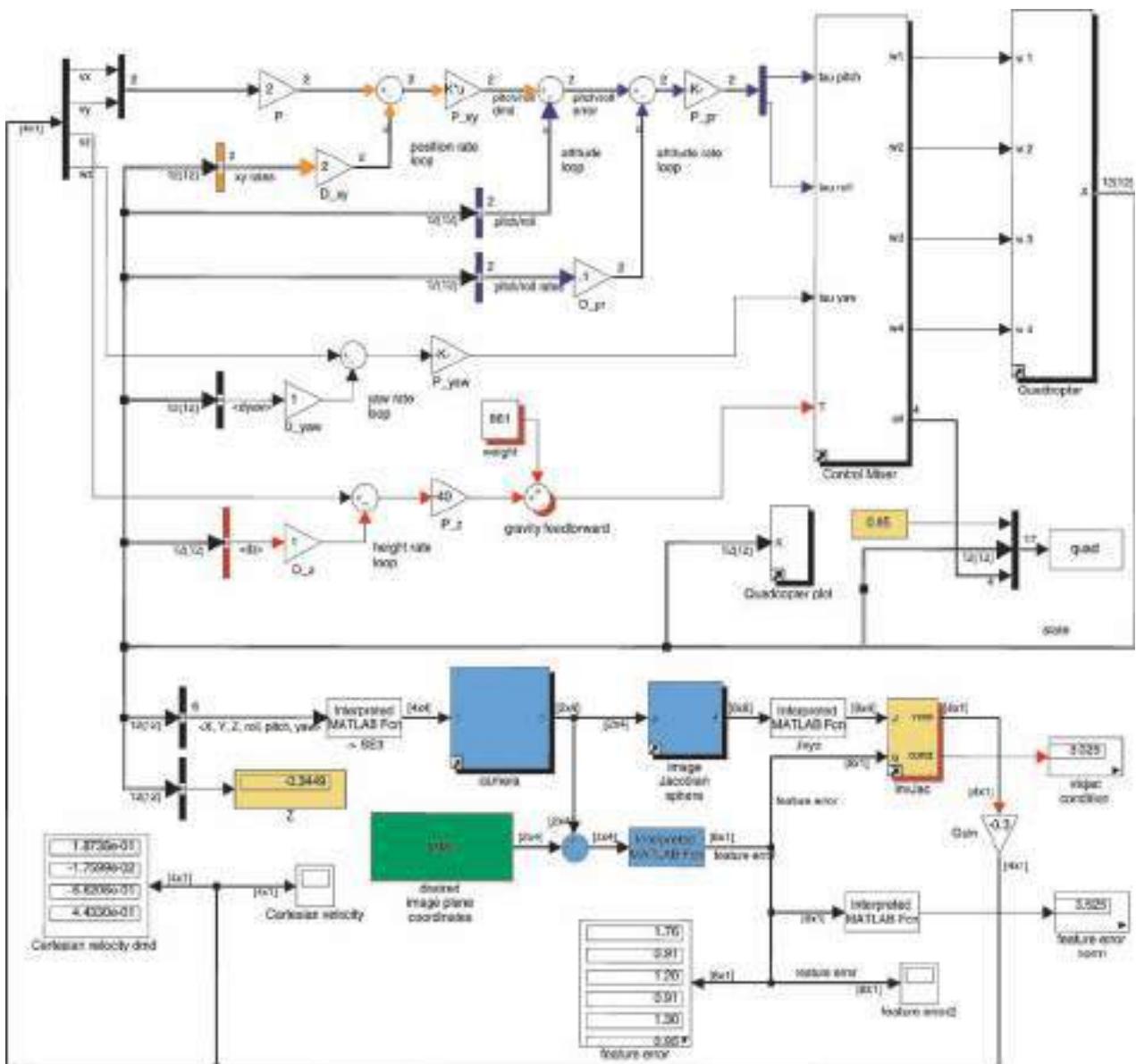
is shown in Fig. 16.9. The model is a hybrid of the quadcopter controller from Fig. 4.17 and the underactuated IBVS system of Fig. 16.6. There are however a number of key differences.

Firstly, in the quadcopter control of Fig. 4.17 we used a rotation matrix to map  $xy$ -error in the world frame to the pitch and roll demand of the flyer. This is not needed for the visual servo case since the  $xy$ -error is given in the camera, or vehicle, frame rather than the world frame. Secondly, like the mobile robot case the vehicle is underactuated, and here the Jacobian comprises only the columns corresponding to  $(v_x, v_y, v_z, \omega_z)$ . Thirdly, we are using a spherical camera, so a `SphericalCamera` object is passed to the camera and visual Jacobian blocks.

Fourthly there is a *derotation* block immediately following the camera. We recall how the quadcopter cannot translate without first tilting into the direction it wishes to translate, and this will cause the features to move in the image and increase the image feature error. For small amounts of roll and pitch this could be ignored but for aggressive manoeuvres it must be taken into account. We can use the image Jacobian to approximate the displacements in  $\theta$  and  $\phi$  as a function of displacements in camera roll and pitch angle which are rotations about the  $x$ - and  $y$ -axes respectively

$$\begin{pmatrix} \Delta\theta \\ \Delta\phi \end{pmatrix} = \begin{pmatrix} \cos\phi\cos\theta & \sin\phi\cos\theta \\ \sin\theta & \sin\theta \\ \sin\phi & -\cos\phi \end{pmatrix} \begin{pmatrix} \Delta\theta_R \\ \Delta\theta_P \end{pmatrix}$$

This is a first-order approximation to the feature motion.



**Fig. 16.9.** The Simulink® model `sl_quadcopter_vs`.IBVS with a spherical camera for hovering over a target

and these are subtracted from the features observed by the camera to give the features that would be observed by a camera in the vehicle's frame  $\{V\}$ .

Comparing Fig. 16.9 to Fig. 4.17 we see the same outermost *position loops* on  $x$ - and  $y$ -position, but the altitude and yaw loops are no longer required. The IBVS controller generates the required velocities for these degrees of freedom directly. Note that rate information is still required as input to the velocity loops and in a real robot this would be derived from an inertial measurement unit.

The simulation is run by

```
>> sim('sl_quadcopter_vs')
```

Simulink® menu File+Model Properties +Callbacks+InitFcn. These commands are always executed prior to the beginning of a simulation.

and the camera pose, image plane feature error and camera velocity are animated. Scope blocks also plot the camera velocity and feature error against time. The simulation results can be obtained from the simulation output object `out`. The initial pose of the camera is set in the model's properties ▶.

## 16.7 Wrapping Up

### Further Reading

A recent introduction to advanced visual servo techniques is their tutorial article Chaumette and Hutchinson (2007) and also briefly in Siciliano and Khatib (2008, § 24). Much of the interest in so-called hybrid techniques was sparked by Chaumette's paper (Chaumette 1998) which introduced the specific example that drives the camera of a point-based IBVS system to infinity for the case of target rotation by  $\pi$  about the optical axis. One of the first methods to address this problem was 2.5D visual servoing, proposed by Malis et al. (1999), which augments the image-based point features with a minimal Cartesian feature. Other notable early hybrid methods were proposed by Morel et al. (2000) and Deguchi (1998) which partitioned the image Jacobian into a translational and rotational part. An homography is computed between the initial and final view (so the target points must be planar) and then decomposed to determine a rotation and translation. Morel et al. combine this rotational information with translational control based on IBVS of the point features. Conversely, Deguchi et al. combine this translational information with rotational control based on IBVS. Since translation is only determined up to an unknown scale factor some additional means of determining scale is required.

Corke and Hutchinson (2001) presented an intuitive geometric explanation for the problem of the camera moving away from the target during servoing, and proposed a partitioning scheme split by axes:  $x$ - and  $y$ -translation and rotation in one group, and  $z$ -translation and rotation in the other. Another approach to hybrid visual servoing is to switch rapidly between IBVS and PBVS approaches (Gans et al. 2003). The performance of several partitioned schemes is compared by Gans et al. (2003).

The polar form of the image Jacobian for point features (Iwatsuki and Okiyama 2002a; Chaumette and Hutchinson 2007) handles the IBVS failure case nicely, but results in somewhat suboptimal camera translational motion (Corke et al. 2009) – the converse of what happens for the Euclidean formulation.

The Jacobian for a spherical camera is similar to the polar form. The two angle parameterization was first described in Corke (2010) and was used for control and for structure-from-motion estimation. There has been relatively little work on spherical visual servoing. Fomena and Chaumette (2007) consider the case for a single spherical target from which they extract features derived from the projection to the spherical imaging plane such as the center of the circle and its apparent radius. Tahri et al. (2009) consider spherical image features such as lines and moments. Hamel and Mahony (2002) describe kino-dynamic control of an underactuated aerial robot using point features.

The manipulator dynamics Eq. 9.1 and the perspective projection Eq. 11.2 are highly nonlinear and a function of the state of the manipulator and the target. Almost all visual servo systems consider that the robot is velocity controlled, and that the underlying dynamics are suppressed and linearized by tight control loops. As we learnt in Sect. 9.4 this is the case for arm-type robots and in the quadcopter example we used a similar nested control structure. This approach is driven by the short time constants of the underlying mechanism and the slow sample rate and latency of any visual control loop. Modern computers and high-speed cameras make it theoretically possible to do away with axis-level velocity loops but it is far simpler to use them.

Visual servoing of non-holonomic robots is non-trivial since Brockett's theorem (1983) shows that no linear time-invariant controller can control it. The approach used in this chapter was position based which is a minor extension of the pose controller introduced in Sect. 4.2.4. IBVS approaches have been proposed (Tsakiris et al. 1998; Masutani et al. 1994) but require that the camera is attached to the base by a robot with a small number of degrees of freedom. Mariottini et al. (1994, 2007) describe a two-step servoing approach where the camera is rigidly attached to the base and the epipoles

of the geometry defined by the current and desired camera views are explicitly servoed. Usher (Usher et al. 2003; Usher 2005) describes a switching control law that takes the robot onto a line that passes through the desired pose, and then along the line to the pose – experimental results on an outdoor vehicle are presented. The similarity between mobile robot navigation and visual servoing problem is discussed in Corke (2001).

---

## Exercises

1. XY/Z-partitioned IBVS (page 481)
  - a) Investigate the generated motion for different combinations of initial camera translation and rotation, and compare to the classical IBVS scheme of the last chapter.
  - b) Create a scenario where the features leave the image.
  - c) Add a repulsion field to ensure that the features remain within the image.
2. Investigate the performance of polar and spherical IBVS for different combinations of initial camera translation and rotation, and compare to the classical IBVS scheme of the last chapter.
3. Arm-robot IBVS example (page 488)
  - a) Add an offset (rotation and/or translation) between the end-effector and the camera. Your controller will need to incorporate an additional Jacobian (see Sect. 8.1.1) to account for this.
  - b) Add a discrete time sampler and delay after the camera block to model the camera's frame rate and image processing time. Investigate the response as the delay is increased, and the tradeoff between gain and delay. You might like to plot a discrete-time root locus diagram for this dynamic system.
  - c) Model a moving target. Hint use the `Camera2` block from the `roblox`s library. Show the tracking error, that is, the distance between the camera and the target.
  - d) Investigate feedforward techniques to improve the control (Corke 1996b). Hint, instead of basing the control on where the target was seen by the camera, base it on where it will be some short time into the future. How far into the future? What is a good model for this estimation? Check out the Toolbox class `AlphaBeta` for a simple to use tracking filter (challenging).
  - e) An eye-in-hand camera for a docking task might have problems as the camera gets really close to the target. How might you configure the targets and camera to avoid this?
4. Mobile robot visual servo (page 489)
  - a) For the holonomic and non-holonomic cases replace the perspective camera with a catadioptric camera.
  - b) For the holonomic case with a catadioptric camera, move the robot through a series of via points, each defined in terms of a set of desired feature coordinates.
  - c) For the non-holonomic case implement the pure pursuit and line following controllers from Chap. 4 but in this case using visual features. For pure pursuit consider the object being pursued carries one or two point features. For the line following case consider using one or two line features.
5. Display the feature flow fields, like Fig. 15.7, for the polar  $r - \phi$  and spherical  $\theta - \phi$  projections (Sect. 16.2 and 16.3). For the spherical case can you plot the flow vectors on the surface of a sphere?
6. Quadcopter
  - a) Replace the perspective camera with a spherical camera.
  - b) Create a controller to follow a series of point features rather than hover over a single point (challenging).
  - c) Create a controller to follow a series of point features rather than hover over a single point (challenging).
7. Implement the 2.5D visual servo scheme by Malis (1999) (challenging).