

A Pipelined Framework for Multiscale Image Comparison

by

David M. Martindale

B.Math University of Waterloo 1981

M.Math University of Waterloo 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

December 2002

© David M. Martindale, 2002

Abstract

In this thesis we present a pipelined framework and a family of algorithms for determining meaningful differences between two images. The pipeline is structured in a way that makes it easy to use only a portion of the pipeline when that is appropriate, and any module in the pipeline can be replaced in its entirety by a new module that uses a different algorithm, while requiring minimal or no changes to other modules. Many pieces of the pipeline can be used individually. Three key design goals were to facilitate comparison of real images to computer-generated images, to take into account the sensitivity of the human visual system, and to accept pairs of images with arbitrary size differences and moderate misalignment in translation, rotation, and field of view. Most existing approaches fail badly at this task, requiring the two images to be of the same size and perfectly aligned with each other.

The pipeline automatically aligns the two images as well as it can in the first four of its five phases. Images are first separated into octave-spaced bands of spatial frequencies, using the wavelet-based technique of Mallat and Zhong to find edges at multiple scales in each of the images. These edges are then matched using a graph-theoretic matching technique in the second phase, a least-squares fit is found for the alignment in the third phase, and the resulting transformation is applied using appropriate resampling in the fourth phase. The final phase of the pipeline computes difference measures for each spatial frequency band and three colour components, weighted according to the human contrast sensitivity function. The same Mallat-Zhong wavelet transform used for edge detection is used as the multi-band filter during comparison.

We describe the pipeline and each component in detail, and discuss alternate approaches for each of the phases and generalizations of the framework. The flexibility of the framework is demonstrated by examples. We show how to restructure the data flow to implement an improved iterative multiscale alignment technique, and we analyze a Laplacian-based edge detector that could be used instead of wavelets.

Contents

Abstract	iii
Contents	v
List of Tables	ix
List of Figures	xi
Acknowledgements	xiii
Dedication	xv
1 Introduction	1
1.1 Image Quality	1
1.2 Motivation	2
1.3 Images and Metrics	3
1.4 Summary of the Thesis	5
1.5 Digital Version of the Thesis	6
2 Background	7
2.1 Image Registration	7
2.1.1 Correlation and Sequential Search	7
2.1.2 Fourier methods	9
2.1.3 Hausdorff Distance	10
2.1.4 Mutual Information	10
2.1.5 Feature Mapping	10
2.2 Image Comparison	14
2.2.1 Mean Squared Error	15
2.2.2 Fourier Techniques	15
2.3 Related Literature	19
2.3.1 Rushmeier et al.	19
2.3.2 Other Image Quality related work	20
2.4 Image Querying	21

3	Design Overview	23
3.1	Pipeline	23
3.2	Multiresolution Issues	26
3.3	Matching Issues	28
3.4	Implementation	29
3.4.1	Libraries	30
3.5	Preview of the Following Chapters	31
4	Phase 1 — Feature Extraction	33
4.1	Wavelet Transform Choice	33
4.1.1	Output	34
4.1.2	To decimate, or not to decimate...	34
4.2	Method	38
4.2.1	Input	38
4.2.2	Wavelet Transform	39
4.2.3	Finding Maxima	40
4.2.4	Output	43
4.3	Alternate edge finder: the Laplacian	46
4.3.1	Algorithm	46
4.3.2	Evaluation	49
4.4	Mallat-Zhong Wavelet Transform Details	51
4.4.1	Definitions	52
4.4.2	One Dimensional Continuous Case	53
4.4.3	Two Dimensional Continuous Case	54
4.4.4	Dyadic Wavelet Transform	55
4.4.5	Discrete Dyadic Wavelet Transform — One Dimension	55
4.4.6	Discrete Dyadic Wavelet Transform — Two Dimensions	59
4.5	Implementation	59
4.6	Discussion	61
5	Phase 2 — Feature Matching	63
5.1	Method	63
5.1.1	Input	65
5.1.2	Level Alignment	66
5.1.3	Building the Graph	66
5.1.4	Matching	67
5.1.5	Alternative matching algorithms	68
5.1.6	Improving maximum-weight maximum-cardinality	72
5.1.7	Output	73
5.2	Performance	73
5.3	Implementation	74
5.3.1	Weight Scaling	75
5.4	Other Software	76
5.4.1	Implementation History	77

6	Phase 3 — Transformation Fitting	79
6.1	Models	79
6.2	Method	81
6.2.1	Transformation Math	81
6.2.2	Input	81
6.2.3	Least-Squares Fitting	82
6.2.4	Pruning	83
6.2.5	Output	91
6.3	Iterative Multiscale Fitting	91
6.3.1	Method	92
6.3.2	Synergy	94
6.4	Implementation	94
6.5	Performance	95
6.6	Discussion	95
6.6.1	Alternative Models	96
6.6.2	Alternative Least-Squares Methods	97
6.6.3	Techniques Related to Pruning	98
7	Phase 4 — Image Resampling	101
7.1	Background	101
7.1.1	Sampling	102
7.1.2	Resampling	103
7.2	Choosing the Image to Resample	104
7.3	Coordinate System	105
7.4	Filters	107
7.4.1	Evaluation	119
7.5	Implementation	124
7.5.1	Edge Handling	126
7.6	Performance	127
7.7	Discussion	127
8	Phase 5 — Image Comparison	129
8.1	CSF Models	130
8.2	Method	133
8.2.1	Initial Processing	133
8.2.2	Multiband Filtering	135
8.2.3	Displaying Differences	137
8.2.4	Calculating Distance	138
8.3	Spatial Frequency to Angular Frequency Conversion	142
8.4	Calculating Weights	143
8.4.1	Measuring Frequency Response	144
8.4.2	Fitting Weights	145
8.4.3	Simulating Frequency Response	146
8.5	Verification and Testing	152

8.6	Implementation	153
8.7	Performance	153
8.8	Discussion	153
9	Testing and Examples	155
9.1	Registration	155
9.2	Real vs. Computer-generated Images	158
9.3	Other Examples	162
10	Conclusions and Further Research	167
10.1	Further Research	168
	Bibliography	171

List of Tables

4.1	Mallat-Zhong wavelet transform filters	60
7.1	Resampling errors — many steps	123
7.2	Resampling errors — two steps	123
8.1	Fitted Weights	146
8.2	Model coefficients	149
8.3	Weights fitted using measured and simulated response	150

List of Figures

1.1	University of Aizu atrium	4
2.1	Two images with identical magnitude spectra	18
2.2	Two more images with identical magnitude spectra	18
3.1	Data flow diagram	24
4.1	Haar wavelet transform	36
4.2	Wavelet-processed disc image	41
4.3	Wavelet-processed “mandrill” image	42
4.4	Maxima at Multiple Scales 1	44
4.5	Maxima at Multiple Scales 2	45
4.6	Laplacian-processed disc image	47
4.7	The effect of thresholding	50
4.8	$\theta(x)$, $\psi(x)$, $\phi(x)$	58
5.1	Edge points in two images	64
5.2	Bipartite graph from edge points	65
5.3	Matching output: translation, levels 1-6	69
5.4	Matching output: rotation, levels 1-6	70
5.5	Matching output: scaling, levels 1-6	71
6.1	Fit: 0 iterations	86
6.2	Fit: 1 iteration	87
6.3	Fit: 2 iterations	88
6.4	Fit: 3 iterations	88
6.5	Fit: 4 iterations	89
6.6	Fit: 5 iterations	89
6.7	Fit: 6 iterations	90
6.8	Fit: 7 iterations	90
6.9	Multiscale fitting data flow diagram	93
7.1	Box filter in 1D and 2D	108
7.2	Triangle filter in 1D and 2D	110
7.3	Cubic polynomial filter, $\alpha = -0.5$	111
7.4	Cubic polynomial filter, $\alpha = -1.0$	112

7.5	Fifth-degree polynomial filter	114
7.6	Seventh-degree polynomial filter	115
7.7	Cubic B-spline filter alone	117
7.8	Cubic B-spline filter with preprocessing	118
7.9	Sinc filter	120
7.10	Lanczos-windowed sinc filter (2 lobes)	121
7.11	Lanczos-windowed sinc filter (4 lobes)	122
8.1	CSF function	131
8.2	Frequency Response of Wavelet Transform	136
8.3	Original Living Room Scenes	139
8.4	Level 1 and 3 Difference Images	140
8.5	Level 4 and 5 Difference Images	141
8.6	Luminance CSF and fitted frequency response	147
8.7	Modelled Frequency Response	149
8.8	Luminance CSF and two fitted frequency responses	150
8.9	Red-Green CSF and fitted frequency response	151
8.10	Blue-Yellow CSF and fitted frequency response	151
9.1	Test case: 50 degree rotation	156
9.2	Test case: 55 degree rotation	156
9.3	Test case: 58 pixel translation	157
9.4	Test case: scale by 1.45	157
9.5	Aizu atrium: photograph	159
9.6	Aizu atrium: artistic rendering	159
9.7	Aizu atrium: physical reflectance models	160
9.8	Aizu atrium: after alignment	161
9.9	Difference between aligned images	162
9.10	Original photos	164
9.11	Composite photo and mask	165
9.12	Components of the composite	166

Acknowledgements

The thesis was supervised by the late Dr. Alain Fournier. The supervisory committee comprised Dr. Ian Cumming, Dr. David Lowe, Dr. James Varah, and Dr. Kellogg Booth who served as acting supervisor during the final two years. Dr. Richard Bartels served as an unofficial member of the supervisory committee. Dr. David Kirkpatrick and Dr. Matthew Yedlin were University examiners and Dr. Holly Rushmeier was the external examiner. Dr. William Dunford chaired the oral examination.

Funding for the research was provided by the Natural Sciences and Engineering Research Council of Canada under a variety of grants and a Post Graduate Scholarship, the B.C. Advanced Systems Institute, the University of British Columbia, and the Media and Graphics Interdisciplinary Centre at UBC.

I would like to thank Alain Fournier for his initial help and inspiration, and I regret that he never saw its completion. Kellogg Booth took on the task of acting supervisor, a position with few of the benefits but all of the liabilities of supervising a PhD student, since I was not working in his area of interest and he had many other students already. I thank him for his advice and encouragement. I also thank all members of my examining committee, including the university and external examiners — they all provided comments and ideas that greatly improved the thesis.

I am particularly indebted to Richard Bartels, who spent more time reading early drafts of chapters than anyone else and who provided much good advice. He also gave support and encouragement during the sometimes-difficult process of writing. Finally, I thank my wife Chris Hitchcock for her support, her faith in me, and for saying “I told you so.” many fewer times than she was entitled to.

DAVID M. MARTINDALE

The University of British Columbia
December 2002

This thesis is dedicated to Alain Fournier, who inspired the work and who was my supervisor during the important initial stages of the research. During this period I received many email messages from him, which had the following .signature quote

I always wanted to be somebody. I should have been more specific.
(Lili Tomlin & Jane Wagner)

Alain didn't need to be more specific.

Chapter 1

Introduction

The research reported here addresses the problem of comparing two images to determine how similar one is to the other. Implicit in this is an assumption that one image may be better than another, although which is better may depend on the purposes for which the comparison is being performed. A flexible framework for performing image comparison is presented that allows for a variety of solutions that may depend on characteristics of the problem being solved. The framework is illustrated by a particular implementation, and some alternatives, to demonstrate the usefulness of the framework and the tradeoffs that can be made when configuring it.

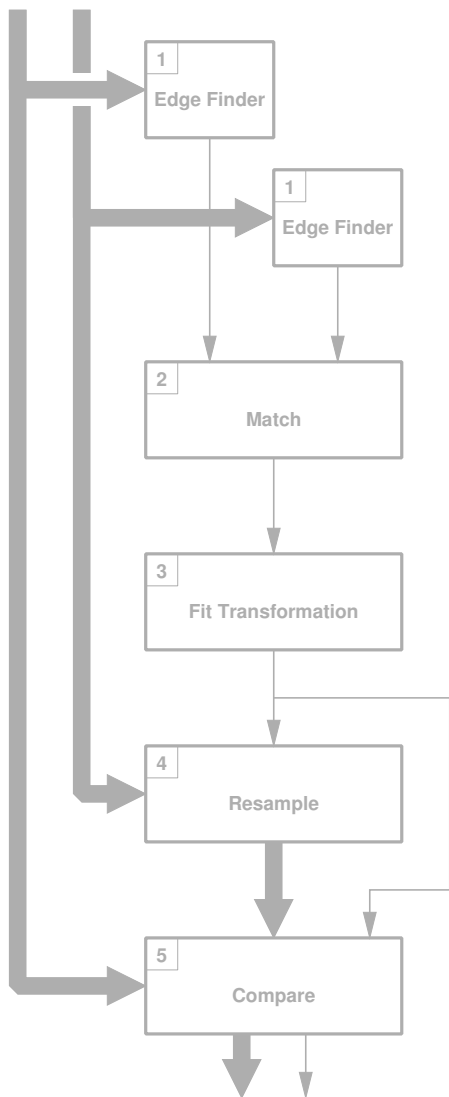
Three key design goals were to facilitate comparison of real images to computer-generated images, to take into account the sensitivity of the human visual system, and to accept pairs of images with arbitrary size differences and moderate misalignment in translation, rotation, and field of view. Most existing approaches fail badly at this task, requiring the two images to be of the same size and perfectly aligned with each other.

1.1 Image Quality

What does “image quality” mean? In fact, this term has multiple meanings depending on context.

In some contexts, “quality” is a property of a single image, and can be judged while looking at that image alone. One painting or photograph may be said to be of high quality if it meets some standard of artistic achievement or of technical competence. By nature, these judgements are somewhat subjective, and two experts may disagree greatly on the level of quality of a single image.

In other domains, the measure of “quality” may be some-



what more objective, depending on some statistical measure of properties of the image combined with a model of the value of these properties for a “good” image. For example, the Mitre IQM (Image Quality Measure) algorithm [50] is designed to judge the quality of scientific images like aerial photos or fingerprint records. It operates by analyzing the image power spectrum (the magnitude of the Fourier transform) in several ways that detect classes of common image degradations.

However, in this thesis, we make no attempt to measure the “quality” of a single image. We assume that we always have two images, with one being a *reference* image that is exactly what we want, and a second *test* image that may or may not be close to the reference image in appearance. If the test image looks nearly identical to the reference, we conclude that the test image is of high quality, while increasing difference is interpreted as reduction in quality.

This process of comparing two images to determine how similar they are is what we mean by an “image quality metric”.

1.2 Motivation

A substantial amount of the research in computer graphics over the last two decades has been devoted to creating ever more realistic images. Progress on several fronts has brought us to the point where sometimes we see computer-generated images that might be mistaken for real photographs. However, the “accuracy” of these images is still primarily judged by human viewers using their eyes and brains, not more objective measures.

In some circumstances, this is all that is needed. The ultimate aim of realistic image synthesis is to fool human viewers into believing that the scene is “real”, so human judgements about “realism” are quite appropriate. If a computer-generated image is presented as an isolated artistic image, all that is necessary is to convince the human viewer that the image is “real”, and the viewer’s perception of “realism” is all that matters.

On the other hand, there are situations where a more objective measure of accuracy is necessary. When a computer rendering is used to visualize the appearance of an office or entire building before it is built, and the architects make decisions about design based on this rendering, we would like the computer image to be as close as possible to a photograph of a real office built exactly to the current plans, down to the last detail of colour and shading. If we are going to insert computer-generated objects into real scenes photographed by a camera (e.g. movie special effects), we want the lighting and shading of the computer-generated components to be exactly as if they were real objects in the real scene.

For these situations, one might argue that it is not sufficient for a rendering system to produce images that look “realistic” when viewed alone. We would like to be able to verify the accuracy of our renderer by acquiring an actual photograph of a physical scene, then building a model of the same scene and rendering it under the same conditions (lighting, camera parameters) as when the photograph was taken, and then comparing the two images side-by-side. Looking for differences between two images is a much more sensitive and meaningful test than examining one image in isolation, and so this is how we should go about verifying the performance of our renderers before we trust them to generate images of things that do not physically exist.

Given a pair of images, we might consider using human observers to compare them. The human visual system is very good at ignoring essentially irrelevant differences such as:

- differences in physical image size

- small differences in field of view
- a camera that is a few degrees off level
- slight differences in overall brightness

Human viewers ignore these effects without thought, while looking for important differences in the *content* of the two images.

Unfortunately, human observers have some problems as well. They aren't repeatable; they may rate the amount of difference between the same pair of images differently on different occasions, influenced by a host of external factors like fatigue, weather, distractions, and so on. Humans also aren't necessarily available on demand to compare a pair of images at the time the comparison is needed — perhaps in the middle of the night for a long-running rendering job.

What we'd really like is a computer algorithm that is deterministic and repeatable, yet which mimics as many of the good attributes of human comparison as possible.

One place that such an image quality measure would be useful is in computer graphics, where some practitioners are concerned with producing extremely realistic-looking images. Computer graphics researchers have studied more realistic shading algorithms (Phong vs. Gouraud), then ray tracing, then radiosity and light transport techniques. In some cases, the computer algorithms are attempting to re-create real scenes or locations, and a well-taken photograph of the actual object or scene is the reference to which we compare the computer-generated scene.

Figure 1.1 shows an example: The image on the left is a digital camera photograph of an atrium at the University of Aizu. The image on the right is a computer-generated image, created from models of the room, the light output of the luminaires, and the light-reflecting characteristics of some of the surfaces. Ideally, the two would match.

In addition, an image difference measure is more widely useful. We might want to compare two or more computer-generated images of the same scene, rendered by different algorithms, to a reference photograph to see which technique produces more realistic images. Or we may simply wish to compare two images to see how alike they are in situations where there is no reference image at all.

Finally, it is important that the method be able to present to the user information about *how* and *where* two images differ. When comparing rendering algorithms, it is extremely useful to know not just that two images are different, but that (for example) the main source of difference is the rendering of the texture in a carpet. When comparing a continuous-tone version of an image with a bilevel version of the same image produced by dithering or halftoning techniques, we would expect to see large differences between the two images at the smallest spatial scales, since this is inherent in the techniques used to produce the bilevel image. Yet, if the images do match at larger spatial scales, the dithering/halftoning was successful, while differences at larger spatial scales indicate trouble. A single image difference number is not adequate in these circumstances.

1.3 Images and Metrics

The term *metric* has a well-defined mathematical meaning. If $M(A, B)$ is a metric for the “distance” between two images A and B , then M must satisfy the following axioms:

1. $M(A, B) = M(B, A)$

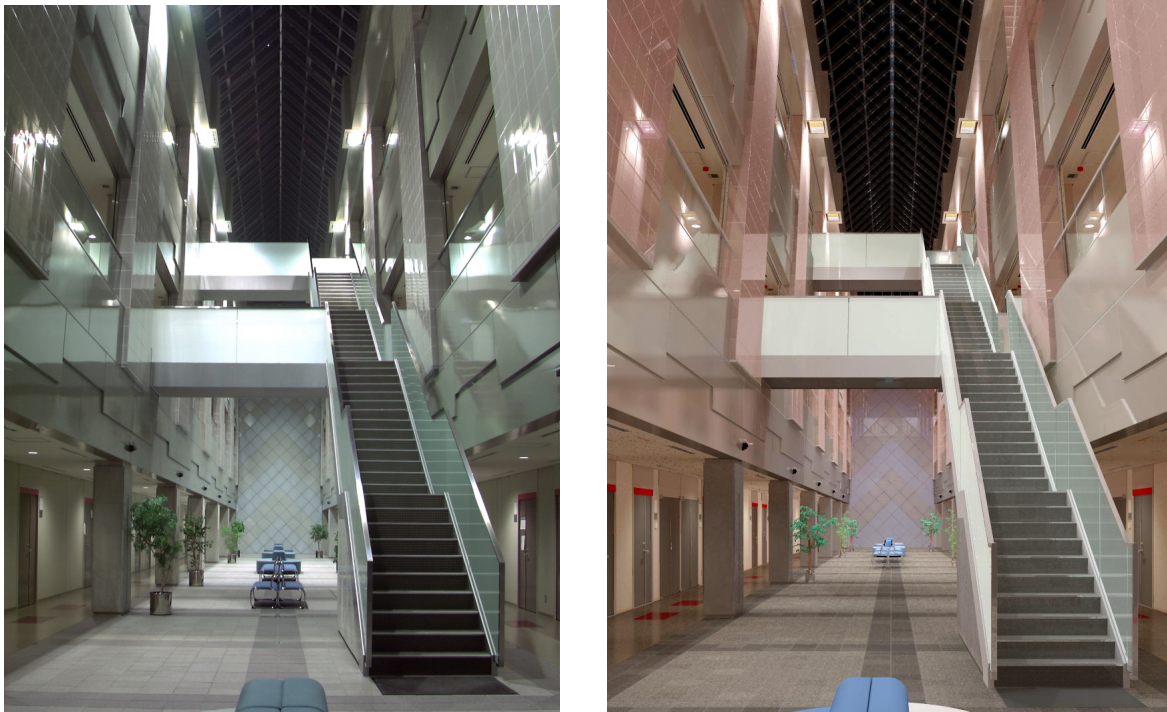


Figure 1.1: University of Aizu atrium; images by Karol Myszkowski and Frederic Drago [51]

2. $M(A, B) = 0$ if $A = B$
3. $M(A, B) \neq 0$ if $A \neq B$
4. $M(A, B) \leq M(A, C) + M(C, B)$

(Axioms 2 and 3 can be expressed as a single “if and only if” clause, but we want to refer to them separately.)

The first axiom simply says that the metric is symmetric, that the distance is the same no matter which of the images is first and which is second. We will arrange our algorithm so that this is true, at least as closely as we can make it. (Numerical error may prevent this from being exactly true.)

The second axiom requires that the distance be zero if an image is compared to itself. This will certainly be true of our algorithm.

The third axiom requires that the distance between all possible pairs of non-identical images be non-zero. This causes some problems. In the first place, there are some differences that we deliberately want to ignore as much as possible (the ones that human observers ignore), and we might on occasion succeed in ignoring these differences completely. We need to clarify that, for our purposes, a pair of images are considered to be “equal” if a normal human observer, viewing the pair of images so they subtend the viewing angle that we specified, cannot see any difference between the images. This is true even if the images are not bit-by-bit equal, as long as the differences are invisible.

Even if we do that, we need to prove that any possible difference between the two images, other than one we are ignoring, will cause the distance to be non-zero. Although our method is intended to behave in this way, we cannot prove that this is true in general. In fact, there is a simple counterexample: two images that differ only by a scale factor in intensity will compare equal, since we deliberately normalize the intensities. Yet if the intensity difference is sufficiently large, a human observer would see it.

(This visual definition of equal also causes a problem with the second axiom. Two images that are “visually equal”, but not bitwise equal, may not have exactly zero distance, though the distance will be small).

The fourth axiom is called the *triangle inequality* because of its application to the lengths of the sides of a triangle in plane geometry. In that domain, it says that the length of one side of a triangle is less than or equal to the sum of the lengths of the other two sides. In our context here, it says that the distance between two images A and B is always less than or equal to the distance between A and a third image C plus the distance between C and B .

Now, this is certainly sometimes true, but we cannot prove that it is always true for our algorithm. Thus, our algorithm is probably not a metric in the mathematical sense. It would be more accurate to call it an “image difference value” than an “image quality metric”. However, the latter term has already appeared in many papers, and seems to be embedded in the literature by now. So we will use “image quality metric” throughout the thesis to refer to our algorithm, while keeping in mind that it is not a true metric in the mathematical sense.

1.4 Summary of the Thesis

The principal contributions of this research are the design and implementation of a pipelined software architecture for image comparison, and the choices we made for each of the components. The pipeline is a flexible framework that supports easily configurable modules each of which performs a well defined phase in the overall process. The five phases of the pipeline are connected by simple, well defined interfaces that permit easy substitution of components.

The pipeline automatically aligns the two images as well as it can in the first four of its five phases. Images are first separated into octave-spaced bands of spatial frequencies, using the wavelet-based technique of Mallat and Zhong to find edges at multiple scales in each of the images. These edges are then matched using a graph-theoretic matching technique in the second phase, a least-squares fit is found for the alignment in the third phase, and the resulting transformation is applied using appropriate resampling in the fourth phase. The final phase of the pipeline computes difference measures for each spatial frequency band and three colour components, weighted according to the human contrast sensitivity function. The same Mallat-Zhong wavelet transform used for edge detection is used as the multi-band filter during comparison.

We describe the pipeline and each component in detail, and discuss alternate approaches for each of the phases and generalizations of the framework. The flexibility of the framework is demonstrated by examples. We show how to restructure the data flow to implement an improved iterative multiscale alignment technique, and we analyze a Laplacian-based edge detector that could be used instead of wavelets.

The thesis continues in Chapter 2 with background material and a short survey of the literature on image registration and image comparison, the two key elements of the problem we address. Chapter 3 follows with a high-level description of our framework, which is a pipelined software architecture that can be configured in a variety of ways to perform image comparison using a multiscale approach. The next five chapters describe each of phases of the pipeline in detail: edge finding, feature matching, transformation fitting, resampling, and image comparison. Chapter 9 returns to a discussion of the pipeline as a whole, and presents a number of tests that were performed to illustrate the performance characteristics of the pipeline under a variety of configurations. The final chapter highlights the contributions of the research and suggests future work.

Novel aspects of the research include the use of the Mallat-Zhong wavelet transform both for edge finding and for the final image comparison (this approach has not been taken before), an adaptive decimation scheme during edge finding (this optimizes the tradeoff between memory requirements and image match resolution), the use of bipartite graph matching algorithms for alignment (standard image registration approaches usually are designed for a different set of problems), the use of pruning during the least squares fitting to provide robustness (this adaptively throws out bad data and converges to the correct solution in many cases where a straightforward algorithm fails), the use of higher quality resampling filters than are usually employed in computer graphics (these are chosen to minimize image intensity distortion), and the use of a metric that compares both luminance and colour content (this takes into account the spatial and chromatic response sensitivity of human colour vision). Additional contributions are the ability to configure the pipeline for iterative multiscale matching and fitting (this extends the range of successful matching), and the design and analysis of a multiscale Laplacian-based edge finding algorithm (this is an alternative to the wavelet-based approach we recommend, though it may be better in some circumstances).

1.5 Digital Version of the Thesis

There will be a limited number of copies of this thesis on CD-ROM. The CD-ROM contains a black and white version of the thesis and a full colour version. The black and white version is identical to the printed version, and is on the CD-ROM for those who want to print their own copy with better quality than microfilm or photocopier allow. The digital colour version differs only in the figures, which provide a better illustration of some portions of the research than do the black and white figures. Interested readers are encouraged to view the colour version when possible.

Both versions are provided in Adobe PostScript and Adobe Acrobat PDF formats on the CD-ROM. If the CD-ROM is not available, contact the author for a URL to access this material via the World Wide Web.

Chapter 2

Background

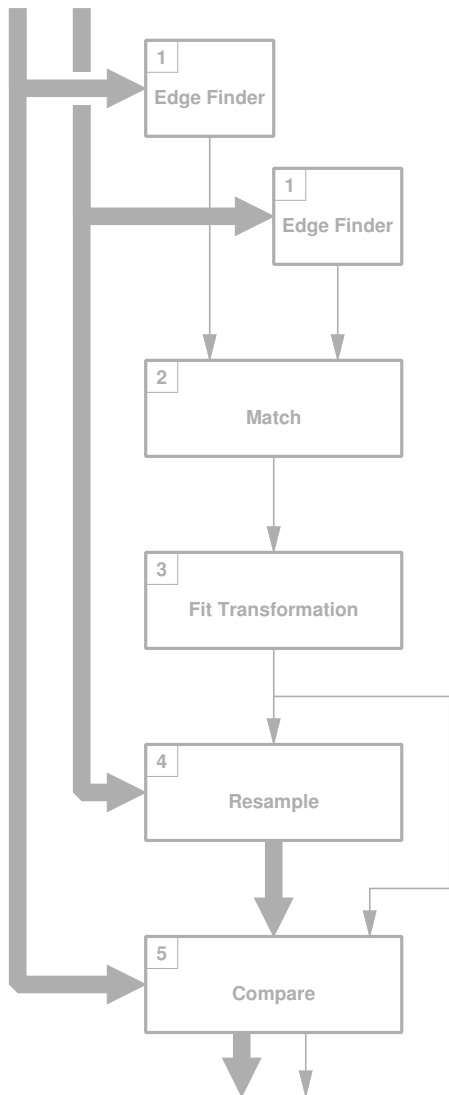
Our method operates by first aligning a pair of images as well as it can, and then comparing them. These two processes have been done many different ways, though usually in separate domains. The alignment process is usually referred to as “image registration” and is heavily used in medical imaging and computer vision. Image comparison methods are most often used in image coding, transmission, and compression.

2.1 Image Registration

Image registration is used in a wide range of fields, from photogrammetry through remote sensing, medical imaging, computer vision, and computer graphics. Naturally, a variety of techniques have been developed, many of them designed to solve a particular problem in a particular environment. Lisa Brown’s survey paper [4] covers the range of techniques available at that time, classifying them in several different ways. Zuk’s Master’s thesis [82] has an extensive discussion of the registration problem, particularly in a medical imaging context. Maintz and Viergever [39] provide a more recent survey of registration in medical imaging only. In this section, we introduce the major methods currently used for registration and discuss their applicability to the problem of comparing real and synthetic images.

2.1.1 Correlation and Sequential Search

One of the simplest ways to calculate the similarity of two images aligned in a particular way is to calculate their normalized cross-correlation. The result is maximized if the two images are identical in their area of overlap. If the two images are not aligned, but we suspect that a simple translation can bring them into alignment, we can calculate the correlation of the one image



with a translated version of the other image, for all possible translations within some maximum range. The particular (x, y) offset that yields the maximum correlation value is the translation that best aligns the two images. Some examples can be found in [55].

The cost of a single correlation is proportional to the number of pixels in the smaller of the images. The cost of using correlation to search for best alignment is proportional to the number of different offsets tried times the cost of a single correlation. Thus, direct correlation is reasonably inexpensive to use in matching a small “template” image to a small region in another image (that may be quite large), but it’s not practical to calculate the correlation of two moderate-size or large images for a large range of possible translations. There is a more efficient way of calculating large correlation results like this by multiplying the Fourier transforms of the two images, then taking the inverse Fourier transform.

However, simple correlation will not work well when the difference between the images is not just a translation. Any significant rotation or scale change will substantially reduce the correlation at the point of best translational alignment. If we are aligning small templates against a large image, we can deal with this by preparing multiple templates that cover a range of rotations or scale factors or both, then correlating all of the templates with the image all of the possible translational offsets. The more parameters we allow the misalignment to have, the larger the search space of possible transformations we must try and the faster the cost gets out of hand.

Conventional correlation methods calculate the correlation for all possible translational offsets as a single operation, then look at the results. Barnea and Silverman [1] speed up the process by using the sum of the absolute value of pixel differences as a metric instead of correlation, and then aborting the summation at any one particular offset if the sum exceeds a threshold. This can save a considerable amount of time.

Another way of managing the cost is to use a multiscale approach. A series of images of successively smaller sizes, sometimes called an “image pyramid”, is prepared from each input image. The coarsest images in the pyramid are aligned first. The result is only approximate because of the coarse resolution of the images, but this result (after adjusting for the difference in image size) is then used to restrict the range of search at the next-finer level of the pyramid. One recent example of this can be found in [70].

One interesting variant of correlation-based registration has been developed by Wolberg and Zokai [22]. They define a “log-polar transform”, a highly non-linear transform used to remap an image relative to its centre. They assign a standard polar coordinate frame to the input image, where every image feature is located at a position (r, θ) . In the transformed image, the same feature is mapped to a position proportional to $(\log r, \theta)$. Radial lines in the original image become horizontal lines in the log-polar image, while circles around the centre point of the original image become vertical lines in the log-polar image. The nature of this mapping is such that a rotation of the original image becomes a vertical translation of the log-polar image, while a scale change to the original image becomes a horizontal translation in the log-polar image. Thus, by converting two images into log-polar form and finding the location of peak correlation, Wolberg finds the rotation and scale factor that best aligns the two images.

Unfortunately, the log-polar transform depends on the assumed location of the centre of the two images before transformation. If the two centres are not located at the same place in the scene, there will be no clear peak in the correlation output, because rotation and scaling alone will not align the images. To overcome this, Wolberg explores a range of possible centre locations for the one image, performing the log-polar transform and correlation described above for each one. The highest correlation peak among all the correlation results is identified, with the location of the peak giving rotation and scale and the centre location used for that particular trial giving the best translational offset. As before, we can end up searching a large space of

possible transformations. Wolberg limits the cost by using a coarse-to-fine multiresolution approach. In addition, the result of the log-polar registration only needs to be approximately correct, since it provides the starting point for a full affine registration process using the Levenberg-Marquardt [36], [43] nonlinear least squares method. In this way, Wolberg obtains a correlation-based approach that can handle substantial scale changes and arbitrary rotation in addition to translation.

There is a substantial problem with correlation methods: they assume that the two images have nearly identical content. Multi-modal medical images come from different sensors (e.g. PET and MRI) and have poor correlation even when correctly aligned because of the considerable difference between the shading of the images. Similarly, a photograph and a computer-generated image of the same scene may have very different surface appearances. We also expect there to be small differences in camera position which will prevent any possible rigid transformation from giving perfect alignment. Thus, we do not believe that correlation-based methods can handle the range of differences we expect to find when comparing real and synthetic images.

2.1.2 Fourier methods

These methods begin by taking the Fourier transform of the two images. Simple transformations in the spatial domain have well-defined effects in the frequency domain, so looking for these frequency-domain effects tell us what spatial-domain changes produced them. Translation in the spatial domain becomes a phase shift proportional to frequency in the frequency domain. Rotation of the image rotates the Fourier transform by the same angle in the same direction. Scaling the image also scales the Fourier transform, but the spatial-domain and frequency-domain scale factors are the reciprocals of each other.

The simplest Fourier method, called phase correlation, was introduced by Kuglin and Hines [33]. If the two original images are substantially the same except for a translation, then the cross-power spectrum of their Fourier transforms is a signal whose phase is equal to the phase difference between the two Fourier transforms. If we take the inverse Fourier transform of the cross-power spectrum, we get an impulse whose location in the spatial domain is equal to the translational offset between the two images.

Because this method uses phase information from all frequencies, it is not perturbed much by different illumination conditions or even if the two images came from different sensors. However, it can only handle translation.

Various authors have proposed methods of extending the Fourier techniques to handle rotation as well as translation. These involve rotating the Fourier transform of one image until some measure indicates that the two images are close to being related by a pure translation. De Castro and Morandi [15] simply examine the inverse Fourier transform of the cross-power spectrum, looking for the angle that gives a result closest to an impulse.

Reddy and Chatterji [59] proposed a method to recover scaling in addition to translation and rotation. The magnitude spectrum of the Fourier transform is independent of phase in the frequency domain and thus independent of translation in the spatial domain. Reddy and Chatterji convert the two magnitude spectra into log-polar form and then use phase correlation to obtain the translation in this log-polar space. Translation in log-polar coordinates is equivalent to scaling and rotation in the Cartesian-coordinate frequency domain, which in turn is determined by scaling and rotation in the spatial domain. However, large scale factors (beyond about 1.8) alter the Fourier transform too much and the method fails.

We expect that we will sometimes have real and synthetic images that differ greatly in size, and where the transformation needed is more general than simply a combination of translation, rotation, and scaling.

Thus, we feel that Fourier methods are not suitable for our problem.

2.1.3 Hausdorff Distance

Given two sets of points A and B , the directed Hausdorff distance between A and B is (informally) the maximum distance that any point in A is from all points in B . More precisely, for each point p_i in A we find the minimum distance from p_i to any point in B , and call that distance d_i . Then the maximum over all of the d_i is the Hausdorff distance.

Huttenlocher, Klanderman, and Rucklidge [31] use this for registration. They first apply an edge detector to create a binary edge image, which is effectively a set of edge points. Then they use a search technique to minimize the Hausdorff distance between the two edge point sets. Their implementation handles only translation, though they discuss how to add rotation as well.

Unfortunately, this method seems to depend on a number of techniques for pruning the search space in order to get adequate performance, and the pruning methods get more complex when both rotation and translation are allowed. This does not seem general enough to be useful to us.

2.1.4 Mutual Information

Mutual information is a measure from information theory. Informally, it measures how well knowledge of one thing predicts the value of some other thing. If the “things” are the pixel intensities in a pair of images, then mutual information provides a similarity measure for the images. Thus, it can be used as an alternative error metric for the registration process.

Mutual information can be thought of as a nonlinear generalization of correlation. It successfully indicates the alignment of images with matching shapes but greatly different tonal scales, such as those obtained by different medical imaging sensors (e.g. CAT, MRI, and PET). In comparison, correlation gives a clear result only when both shape and brightness scales match. Given two images with a high correlation, inverting the tonal scale of one image (i.e. making it a negative) will cause their correlation to become low. Mutual information is not affected by the same change of intensity scale.

Viola’s Ph.D. thesis [74] and later collaboration [78] use mutual information to register 2D images to 3D models as well as registering multi-modal 2D medical images. Collignon’s Ph.D. thesis [10] uses a different way of estimating mutual information using a 2D histogram, and concentrates on registering 3D multi-modal medical images. Thevenaz and Unser [71] use mutual information as their similarity metric, but they also use a multiresolution decomposition of the input images based on cubic splines. All of these methods model the transformation as being affine or something more restrictive (i.e. just translation and rotation).

Methods based on mutual information have proved useful for aligning medical images from different sensors. They might also work well for real and computer-generated images with substantial shading differences. However, these techniques are relatively new and the optimization method is relatively expensive. We did not pursue this approach to our problem, though it might be interesting for future investigation.

2.1.5 Feature Mapping

This is really a large family of registration techniques, with many variations. We have chosen this framework for our own registration method. The general method works in three stages:

1. Identify significant features in each of the images.

2. Generate pairs of corresponding points.
3. Determine the transformation which best maps the points from the one image to the location of the corresponding points in the other image.

Sometimes the three stages take place in the sequence described with no feedback, but often the pairing and transformation-fitting stages are iterated. The transformation output from one iteration provides a starting point for the pairing stage of the following iteration, which should improve the quality of the pairing. This, in turn, may improve the transformation estimate further, and the whole process repeats until it stops improving.

Identifying Features

The features identified in the first step may simply be points, higher-level features like edges and corners and curves, or features with even more complex descriptions.

In photogrammetry, the features are traditionally hand-picked points in the images, but in most other registration work the features are automatically selected by some means. Sometimes in photogrammetry and medical imaging the control points are special markers inserted into the environment being measured (called intrinsic control points), but most of the time registration has to find features in the image data itself (called extrinsic control points).

Automated registration methods usually identify some particular sort of image feature in the hope that these are distinct features that will be found in both images. Edges are a commonly used feature, usually found with something like a Laplacian filter or a Canny [6] edge detector. Recently, a number of people have begun using a wavelet transform to extract “edges” from images at multiple scales. Le Moigne [35] uses a decimating wavelet transform to locate features in satellite imagery. Fonseca and Costa [20] use a decimating wavelet transform to filter the image into bands of different scales of detail. The wavelet transform is treated as a measure of the intensity gradient in the image (a vector), and edges are the set of points where the modulus of this vector reaches a local maximum. Hsu and Beuker [29] appear to use exactly the same edge-finder to begin with, but they compare edges found at multiple scales to discard insignificant or false edges.

Our own registration method has a number of things in common with the approaches above. We use a similar wavelet transform, but in a non-decimating form (for the first two levels at least) to provide better resolution for the features that are found. We also use the maximum of the modulus of the gradient vector as the location of the edge.

Zhang et al. [81] begin by finding corners (points of peak local image variation) instead of edges. Schmid and Mohr [66] begin with the same corner detector, then calculate a local descriptor for each corner based on the characteristics of the image around the corner as well. Lowe [37] developed a feature-detection method that operates at multiple scales, using a difference-of-Gaussian operator.

Pairing Corresponding Points

Hand-picked control points usually have their pairing explicitly specified as part of the selection process, but automatically-identified features must undergo some sort of process to identify corresponding pairs of points. Often this process is error-prone, so only a fraction of the correspondences is correct.

The pairing process usually begins by identifying plausible candidate pairs of points. Given a feature location in one image, all other features located within a maximum distance of this location in the other image are considered as possible candidates. (For example, Zhang et al. use half the image width as their limit.) For each such pair, a score of some sort is often calculated to determine whether a particular pair is likely to represent a true match or not.

Fonseca and Costa measure the closeness of match for a pair of points by calculating the cross-correlation of the pixels in the neighbourhood of the two points in the smoothed image output from the current level of the wavelet transform. Hsu and Beuker [29] also use correlation, but apparently based on the original images rather than wavelet-smoothed images. Zhang et al. [81] also use correlation, though the features they are comparing are corners instead of edges.

However, simple correlation only works when the two images present the same corner features at approximately the same scale and orientation. Correlation is also sensitive to changes in camera exposure or scene illumination. Schmid and Mohr [66] improve on correlation by calculating a local “image descriptor” for each corner based on an analysis of local image characteristics around the corner. The descriptor is calculated in a way that is invariant to rotation. Then they use a comparison of the descriptors to determine the likelihood of a potential match. Their method can handle scale changes too, but does so by calculating descriptors at many scales which multiplies the number of comparisons necessary.

Lowe [37] has developed a more general feature descriptor called a “SIFT key”. This descriptor contains a sample of the information in a small region around the location, derived in such a way that the key is invariant to translation, rotation, and scaling. The SIFT keys are also designed to be resistant to change from the introduction of noise or from contrast and brightness change. There is only a single descriptor for each feature regardless of scale, unlike the multiple descriptors at multiple scales of Schmid and Mohr. Lowe’s method can find matching features despite substantial rotation and scale changes between the two images.

Our own method of calculating closeness of match is based on comparing the wavelet transforms of the two images. We calculate a match score based on the similarity of the moduli of the gradients at the two points, the similarity of the arguments (orientations) of the gradients at the points, as well as the Euclidean distance between the location of the points. This is far less expensive than calculating cross-correlations for a large number of possible pairings of points, since the gradient vectors are already available from the edge-finding process. We calculate edges at multiple scales and align the scales between images so that we are always comparing similar-scale features regardless of the actual size of the two images. Our point similarity measure is only slightly affected by image brightness. It is somewhat affected by rotation, though less than correlation-based methods.

No matter how the similarity of pairs of points is determined, any method must ultimately decide which points are actually pairs. One very popular algorithm in computer vision is called Iterated Closest Point (ICP), which we will discuss at greater length in the “Iterative Matching and Fitting” section below. At its simplest, ICP matches points in one image with the closest available point in the other image. This is very prone to error if done just once, but iterating the process tends to find a set of points that are consistent with a transformation.

Fonseca and Costa match points by looking for point pairs where each point is the best available match for the other. Then they recursively check the consistency of the set of matches, deleting the least consistent one at each iteration. Hsu and Beuker calculate the displacement vector represented by each possible pair of points and then find a maximum-sized set of similar displacement vectors. (This assumes that any rotation or scale change is small.)

Zhang et al. use an iterative relaxation method that looks at how consistent each possible match is with other nearby matches, and whether the best match for a point is strongly better than the second-best match.

Our own approach to matching is unlike any of the methods above. We build a weighted bipartite graph representing the possible pairings of features and then use one of several standard algorithms from graph theory to find a matching of the points.

Transformation Fitting

The final stage is deriving the transformation from the matched points. This process needs a model of the transformation between the images, a measure of how well a particular instance of the model fits the data, and an optimization process which iterates toward the best transformation. In some respects, this stage is very similar to the entire process of correlation-based or mutual-information-based registration, but here the data being fitted is the location and other properties of features abstracted from the images, not pixel intensities directly from the images.

Point matching methods always seem to use a transformation model that allows for at least translation, rotation, and uniform scaling. Some allow general 2D affine transformations or projective transformations.

Whatever the transformation model, solving for the transformation parameters is complicated by the presence of incorrect matches in the data. A straightforward least-squares fit can produce an answer with substantial error if there are more than a few bad matches, and some method must be used to eliminate them. For example, Zhang et al. use Least Median of Squares estimation¹ to ignore bad matches while determining the geometry of the image pair. Our own approach uses an iterative technique to prune bad matches based on the size of residuals after least-squares fitting.

Fischler and Bolles [19] introduced the RANSAC (Random Sample Consensus) method to deal with data sets where the majority of matches may be incorrect. This method randomly selects a set of matches of the minimum size necessary to determine the transformation using the selected model, and then calculates the transformation from those points. (For example, two pairs of points determine a translate/rotate/scale transformation with four free parameters, three pairs of points determine a 2D affine transformation with six free parameters). RANSAC calculates the residuals of all point pairs assuming this transformation, and determines the set of points whose residuals are small under the transformation. If the size of this set is sufficiently large, it is called a consensus set, since all these points substantially agree on what the transformation should be. RANSAC re-calculates the transformation using all of the matches in the consensus set (but no others) and stops. On the other hand, if the set of data points with small residuals is too small, RANSAC randomly selects a new minimum subset of input points and starts the process again. If it exceeds some upper limit on the number of tries without ever finding a consensus set, RANSAC fails.

Iterative Matching and Fitting with Feedback

Many registration techniques perform both point matching and transformation fitting within a single iterative algorithm. These methods calculate an initial set of possible matched points, then determine an initial transformation based on those matches. The transformation is applied to the points, which brings the images into better (but still approximate) alignment. Repeating the matching process gives a better result, with more correct matches, since the correct pairs of points are closer together now. The incorrect matches are easier to find and eliminate, because their displacement vectors remain large while the displacement vectors

¹Section 6.6.3 has a discussion of the Least Median of Squares method.

of good matches have become smaller. Repeating the transformation fitting then gives a more accurate transformation. The whole process is repeated until some convergence criterion is satisfied.

A good example of this type of method is the Iterated Closest Point (ICP) algorithm used extensively in computer vision. It was developed by Besl and McKay [3] and Chen and Medioni [8]. It is usually used to match 2D or 3D points to a 3D model, but it should also work well for matching 2D points in two images. The basic method is very simple: For each point in the test image point set, find the closest point present in the model. (If we are registering two images, the “model” is just the point set from the other image.) Once all of the points are associated with a model point, fit a transformation matrix to the matched pairs. On the next iteration, all of the test image points are moved according to this transformation before the closest-point search is begun.

Many changes and optimizations are possible. The “closeness” measure may be simple Euclidean distance between the locations of the two points, or something more complex and more appropriate to the specific problem. The pairs may be examined and some discarded as unusable before the transformation-fitting step. Zhang [80] prunes the matched points using a distance threshold that is adaptive, depending on how close the algorithm is to completion and the size of the standard deviation of the distance between the points in a pair. Zhang also estimates the local tangent vector at each point along a curve, and rejects pairs of points if their tangent orientations are too different. Rusinkiewicz and Levoy [64] are interested in merging multiple 3D datasets, obtain by multiple passes from a 3D scanner, into a single 3D model. In this context, their paper gives a taxonomy of many ICP algorithm variants, analyzing the performance of the various choices in several different areas.

We do not use the ICP algorithm in our own registration method, though we do use something similar to it. We can iterate the match/fit process within a single level of detail, giving a method that is very close to ICP within that level. In addition, we normally use a multiscale approach to registration, where the transformation obtained at a coarser level of detail is used to prealign the edges found at the next finer level of detail before beginning the matching step at the finer level. A significant difference remains: our graph-theoretic matching technique ensures that any point in one image is matched to at most one point in the other image. In general, ICP may match multiple points in the “test” image to a single point in the “model” image.

Non-Global Transformations

All of the discussion of feature matching methods above assume that the goal is to find a single global transformation that applies to the entire image. This is not the only possible approach; there are registration methods that can warp the one image to align it with the other on a more local scale. This requires a way of specifying the distortion at a much finer level of control, which is often done using polynomial interpolation or a spline mesh or deformable elastic models. We will not discuss these methods other than to note that they exist, since finding the parameters for these models can be much more complex than with a global transformation. We decided to restrict ourselves to a global transformation for simplicity, and because we expect it to be able to model most of the differences between images in our application.

2.2 Image Comparison

Most image comparison techniques have been developed to compare an image before and after some process, and we wish to know how the process has affected the image. For example, analog transmission over a

noisy channel, or compression via some lossy method will change the image. Techniques implicitly assume that the two images are the same size (in pixels) and that they are perfectly aligned.

Thus, if you subtract one image from the other and the result is everywhere zero, they are identical and the process preserved the image perfectly. In general, though, the difference is not zero. There is a desire to know how much difference there is between the two images in this case, and image comparison methods provide different ways to answer that question.

2.2.1 Mean Squared Error

Perhaps the most widely used image difference metric is mean squared error. That is, for $M \times N$ -pixel images A and B the mean squared error (MSE) is given by

$$\frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} (A[x, y] - B[x, y])^2$$

The square root of this is called Root Mean Square Error (RMSE), and it is a true metric. (MSE itself is not a metric). MSE is actually a good thing to use if you're trying to measure the amount a video image is damaged by a noisy transmission channel, since the MSE is proportional to the power of the noise added to the signal during transmission. It has a clear physical meaning in this case.

Unfortunately, it has many problems. It requires two input images that are exactly the same size, and perfectly aligned with one another. A misalignment of just one pixel can cause a huge increase in MSE, particularly if the images have much fine (high spatial frequency) detail, yet a human observer would hardly notice a one-pixel shift between two images.

(Note: "spatial frequency" refers to how rapidly some quantity changes cyclically with respect to distance. The term "frequency" alone refers to how rapidly some quantity changes cyclically with respect to time. These measure different properties of a signal, but the same analysis tools can be used in both domains. Frequency is typically measured in units of cycles per second (Hz) or radians per second. Spatial frequency is typically measured in units of cycles per mm, cycles per image width, cycles per pixel, or radians per pixel.)

Another problem is that MSE has no frequency selectivity. Differences between images contribute to the MSE only according to the magnitude of the difference, without regard to spatial frequency. The human visual system (HVS) has many times better sensitivity to medium spatial frequencies (1-10 cycles per degree) than it does to very high or very low spatial frequencies, and an amount of error that is very visible at mid-frequencies will be completely invisible at low and high frequencies. (See Figure 8.1 for a graph of the human Contrast Sensitivity Function (CSF).) Thus, MSE performs very poorly at predicting whether a particular pair of images will appear different or not.

2.2.2 Fourier Techniques

Another very common metric (with many variations) is based on the Fourier transform. The process first computes the discrete Fourier transform (DFT) of both images.

$$\hat{A}[u, v] = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} A[x, y] e^{-2\pi i(ux/M + vy/N)} \quad 0 \leq u \leq M - 1, 0 \leq v \leq N - 1$$

(Note: there are many ways to define the Fourier transform and its inverse. In this definition, x and y are spatial position in pixels and u and v are spatial frequency in cycles per pixel. This is not the same as the definition in Section 4.4.1, which comes from the paper that is referenced there.)

This is usually done via some version of the Fast Fourier Transform (FFT) algorithm, which is fairly efficient; the cost is $O(MN \log(M) \log(N))$.

While some versions of the FFT can adapt to arbitrary image size, most FFT implementations require the image dimensions to both be a power of 2. Often the image must be square as well. Also, the Fourier transform, and thus the FFT, assume that the input image is actually periodic, and that the finite input sample is implicitly replicated infinitely many times in both directions. If the left edge of the image is quite different from the right edge of the image (it usually is) this effectively causes an abrupt discontinuity in sample values at the “wraparound” points. This in turn can create large magnitudes of the coefficients at all horizontal spatial frequencies that are really due to the discontinuity, not the image content. The same effect occurs with the top and bottom of the image and vertical spatial frequencies. There are techniques for dealing with this (called windowing) when analyzing one-dimensional time-domain signals, but they aren’t necessarily suitable for use with 2-dimensional images. However, this is often ignored.

Once the DFT of each image is calculated, the two transforms are compared coefficient by coefficient. The comparison is usually done by calculating the RMS difference. When done this way, this is also a true metric.

The Fourier transform has a very valuable property: it re-represents the image as the sum of a specific set of sinusoid functions. The actual frequency of each of these sinusoids is easily calculated from the array indices. First, we have to account for the fact that array indices greater than half the maximum actually represent negative frequencies, so we define

$$f_M(u) = \begin{cases} u & u \leq M/2 \\ M - u & u > M/2 \end{cases}$$

Then, the transform coefficients found in $\hat{A}[u, v]$ represent a spatial frequency of

$$s(u, v) = \sqrt{f_M(u)^2 + f_N(v)^2}$$

If we know the viewing conditions of the image, we can convert cycles per pixel into cycles per degree, and use a model of the human CSF to weight each DFT coefficient according to its visibility to the human eye.

In addition, the orientation of the sinusoids at this frequency is given by

$$\theta(u, v) = \tan^{-1}(v/u)$$

where a value of zero indicates a vertical pattern (i.e. the modulation is purely in the horizontal direction).

The human CSF is somewhat sensitive to orientation of the patterns viewed [65], [13] p. 185 so a more accurate model of human CSF perception will consider both frequency and orientation. Also, there are masking effects where one signal can hide the presence of another, but only when both frequency and orientation are similar.

Straightforward Fourier-based comparison is just as sensitive as MSE to a small misalignment between the two images. The Fourier transform coefficients $\hat{A}[u, v]$ are complex numbers, usually stored in rectangular form with real and imaginary components. If a coefficient is converted to polar form (r, θ) , the r component (*modulus*) determines the magnitude of this particular frequency of sinusoid. The θ component (*argument*) of the coefficient controls the phase of the sinusoid without changing its magnitude.

Now, if our original image A is shifted in any direction, the magnitude (modulus) of each DFT coefficient is completely unchanged, but the phase (argument) of each coefficient has a “phase shift” added to it. The phase shift is proportional to the size of the shift of the original image and the frequency of this term of the Fourier transform. For example, shifting the image by 0.25 pixel in the X direction causes the complex Fourier coefficient $\hat{A}[M/2, 0]$ to be rotated by 45 degrees. This particular coefficient represents a (horizontal) frequency of 0.5 cycles/pixel or a wavelength of two pixels/cycle. Coefficients for lower frequencies are rotated by smaller amounts.

So, if we have two images which are the same except that one has been shifted (translated) some distance relative to the other, the modulus of their Fourier transforms will be identical, while the phase (argument) of all of the coefficients will be different. If we simply subtract the two Fourier transforms component-by-component, we will calculate a substantial difference.

The set of magnitudes of all of the components of the Fourier transform is sometimes called the magnitude spectrum of the image. The fact that the magnitude spectrum of an image is unaffected by shifting the image suggests a technique for comparing images that is insensitive to translation: compare magnitude spectra while ignoring phase. A number of people have designed image similarity metrics using this.

(Note: for perfect equality, the translation must be a circular shift, with pixels that are shifted out of one side of the image being shifted back in on the opposite side of the image. But if the amount of translation is small compared to the image size, using an ordinary translation instead of a circular shift will produce only small errors.)

There is a considerable amount of information stored in the phase component of the Fourier transform. Consider Figures 2.1 and 2.2. The two images in Figure 2.1 have precisely the same Fourier magnitude spectrum, as do the two images in Figure 2.2. The right-hand images were produced by calculating the FFT of both images on the left, converting the Fourier transforms to polar form, interchanging the phase components (only) of the two Fourier transforms, converting the complex coefficients back to rectangular form, and then performing two inverse FFTs to convert back to the spatial domain. (Red, green, and blue were processed in three independent passes.)

As you can see, the two images in each figure are quite different. In fact, the two hybrid images on the right resemble the original image that their phase component was extracted from more strongly than they resemble the image that their magnitude component comes from. From this, we might be tempted to conclude that the phase component is perhaps even more important than the magnitude component of the Fourier transform.

In any case, it is clear that comparing only the magnitude spectra of these images would falsely conclude that two very different images were equivalent. This demonstrates that the difference of magnitude spectra is not a true metric, because it violates the third axiom of a metric: the metric must not be zero if the images are not identical. Moreover, falsely concluding that very different images are equal can be a very serious error for an image comparison metric. We would be much better off using a distance measure that may sometimes violate the fourth axiom of a metric (the triangle inequality) but that never violates the first three axioms. Then, at least, we would know that the result is zero only for identical images.

The Fourier transform is also sensitive to rotation and scaling of the source images. Rotating a source image also rotates its Fourier transform by the same angle. Scaling a source image also scales its Fourier transform, but the scale factor in the transform and the scale factor in the source image are reciprocals of each other.

Thus, a simple comparison of Fourier transforms requires source images that are just as perfectly



Figure 2.1: Two images with identical magnitude spectra



Figure 2.2: Two more images with identical magnitude spectra

matched as MSE requires. There are techniques to get around this limitation to some extent, with further complexity. However, the main advantage of Fourier techniques over MSE is that we can weight the components of the differences found according to their visibility to the human eye.

2.3 Related Literature

2.3.1 Rushmeier et al.

The inspiration for this thesis is the paper “Comparing Real and Synthetic Images: Some Ideas about Metrics” by Rushmeier, Ward, Piatko, Sanders, and Rust [63]. They introduce the problem of comparing real and synthetic images. They select a real room in a building to photograph, and measure it to build a geometric model of the room for computer rendering. They render some of these images and compare them to photographs taken with a camera that has a known response, using several comparison metrics. The images are approximately aligned by hand before comparison, but no algorithmic method is used to align the image pairs.

The paper also discusses the attributes that they feel a “metric” should have. If $M(A, B)$ is a metric measuring the distance between images A and B , then

1. $M(A, A) = 0$
2. $M(A, B) = M(B, A)$
3. $M(A, C)/M(A, B) \gg 1$ if A and B appear similar while A and C appear different
4. $M(A, B)/M(A, C) \simeq 1$ if A , B , and C all appear similar to one another
5. $M(A, B)/M(C, D) \simeq 1$ if the differences between A and B are similar to the differences between C and D

Note that the first two of these criteria are the same as the first and second axioms that define a metric in mathematics. However, none of these criteria guarantee that the third and fourth axioms of a metric will always be satisfied, and so even a “metric” that satisfies all of these criteria is not necessarily a mathematical metric. With that understood, we will proceed to refer to it as a metric anyway.

The paper also discusses some basic aspects of human perception that ought to be accounted for when computing the distance value in a metric. The authors note that

- The human visual system (HVS) responds to relative luminance differences within an image, not absolute luminances.
- The eye’s perception of “lightness” is a non-linear function of intensity (power).
- The sensitivity of the eye to intensity variations depends on the spatial frequency of the stimulus. It is maximum at a few cycles per degree, falling for frequencies both below and above that.

Rushmeier et al. then propose three different possible metrics. The first applies a nonlinear mapping to simulate brightness perception, then uses a FFT to analyze the frequency content of the images. Only the magnitude information from the FFT is used; the phase is discarded. Then a model of the human

Contrast Sensitivity Function (CSF) is used to weight the amount of signal at each frequency according to its visibility, and the mean squared difference is calculated and used as the distance.

The second method also does a FFT, but phase information is retained. The FFT magnitude is again scaled depending on the visibility of each frequency, but this time a 2D model of the CSF (created by fitting splines to real measurements) is used. The distance depends on both (scaled) magnitude and phase. There is no non-linear intensity mapping.

The third method uses a different non-linear intensity mapping. Then an FFT is done, retaining only magnitude. A third different CSF is used for weighting, and finally the mean squared difference is calculated.

The authors then describe a series of tests of the three metrics using real measured images of a room compared to several different simulated image with differing levels of realism (produced by more or less realistic lighting models), and random noise with statistics equal to the test image.

The paper notes that geometric misalignment between the images being compared impairs all three metrics. The one most affected is the second one, which makes use of Fourier transform phase information.

2.3.2 Other Image Quality related work

Daly’s “Visible Difference Predictor” (VDP) algorithm [13] and the Sarnoff Labs’ “Just Noticeable Difference” (JND) algorithm [11] include considerably more sophisticated models of the human visual system than we use in this thesis. Lubin’s “Visual Discrimination Model” (VDM) [38] is an earlier model from Sarnoff Labs.

Daly seems to be more oriented to high-quality still images. JND is designed for real-time video quality monitoring, and Tektronix has announced a quality measuring instrument that is based on the JND method.

Watson’s DVQ metric [77] is somewhat simpler, and seems to target compression artifacts in digital motion video equipment.

Karol Myszkowski and colleagues performed some tests to check the validity of Daly’s VDP metric in predicting the visibility of changes in rendered images [44]. They have also applied the VDP in the design of global illumination algorithms [75], [52]. It is interesting to note that the VDP metric was used only during the design stages, to choose the algorithms with best visual performance during each phase of rendering from among several available. The VDP is not used while actually rendering an image in their system because it is too expensive to calculate as part of rendering. We expect that our image difference metric is fast enough that it could be used within a global illumination renderer to evaluate the visibility of changes due to progression of the solution.

Sumanta Pattanaik and others at Cornell have built models of human perception to help control image synthesis [57], [18], and [25], or to alter image appearance to match human perception [54], [17]. Instead of beginning with images and calculating the visibility of differences, Ramasubramanian et al. [57] precalculate how much luminance difference can be tolerated at each pixel of an image given the image content. This allows image comparisons to be made in linear intensity space. In addition, the spatially-dependent and luminance-dependent parts of their vision model are separated so the expensive spatial calculations can be done just once. These are all adaptations designed to speed up the metric in the in the context of a rendering algorithm, where the metric is evaluated repeatedly.

Our own comparison method uses a less complex model of perception than Ramasubramanian et al., so it would miss some of the effects that their model captures (principally contrast masking). We have also designed our method for a one-time comparison. If someone wanted to incorporate our method into an

image renderer it appears that some of the same techniques used by Ramasubramanian could be used to speed up our method as well.

The earlier paper by Gaddipatti et al. [21] suggests using a simpler metric to control refinement during rendering. Their approach is similar to ours, using a wavelet transform to decompose an image into frequency bands then weighting the differences according to a model of the human CSF. However, our work improves on theirs in a number of respects: using a non-decimating wavelet transform with symmetric filters, more accurately modelling the CSF response, and processing colour. We discuss this at more length in Chapter 8.

Ann McNamara has done several studies of human perception of the difference between real and synthetic images [45], [46]. The knowledge gained from these studies may eventually be incorporated into computer rendering algorithms.

Peter Barten’s PhD thesis [2] discusses the link between the human CSF and image quality. He has developed his own quality metric called SQRI.

2.4 Image Querying

There has been some work in image processing using wavelets to help find “similar” images in a large collection or database of images. Although this work uses wavelets and compares images, it is quite different from what we are attempting to do.

The method in question is called “query by content” or “query by example”. The basic idea is that the user provides a low-resolution version of an image that they want, or a sketch of the sort of image they want. The database search engine compares this query image in some way to all of the images in the database, and returns a selection of images that are “similar” to the query.

One algorithm for performing this search is described in [32] and chapter 5 of [69]. It begins by pre-processing all of the images in the database. Each image has a wavelet transform applied, then all but the largest few wavelet coefficients are set to zero. (The book suggests retaining 40–60 non-zero coefficients per colour channel for 128 pixel square images.) Those largest coefficients are then quantized to one bit — their sign. This reduces each image to a “feature vector” that abstracts the most important image content. This process is done just once for each image in the database.

When a query image is presented for lookup, the algorithm calculates a feature vector for the query image in the same way. Then the image feature vector is compared to the feature vector of each image in the database, looking for images whose feature vectors have large wavelet coefficients in the same place and with the same sign as the vector for the query image. (The actual comparison is done more efficiently than this simplified description seems to suggest.) Each image gets a similarity rating, and the most similar images are displayed to the user.

In the image querying application, the wavelet transform is essentially used to compress the content of an image into as few bits as practical while remaining identifiably different from other images. The similarity measure used must be very fast, but it is of necessity only an approximate comparison. The query image, particularly if hand-drawn, may look very different than the desired database image yet the similarity measure should not reject the database image. At the same time, it’s acceptable for the similarity measure to match a number of database images since the final selection can be made by the user if more than one choice remains.

In our application, we use a wavelet transform to find edges. This is another form of image abstraction,

but we retain thousands of points per image because we want enough edge information to precisely align the images whenever that's possible. Our comparison measure can afford to be slow since it is only comparing two images, not checking every image in a database. Our comparison measure assumes that the two images should match very well, and calculates and describes in detail where they differ and by how much.

Chapter 3

Design Overview

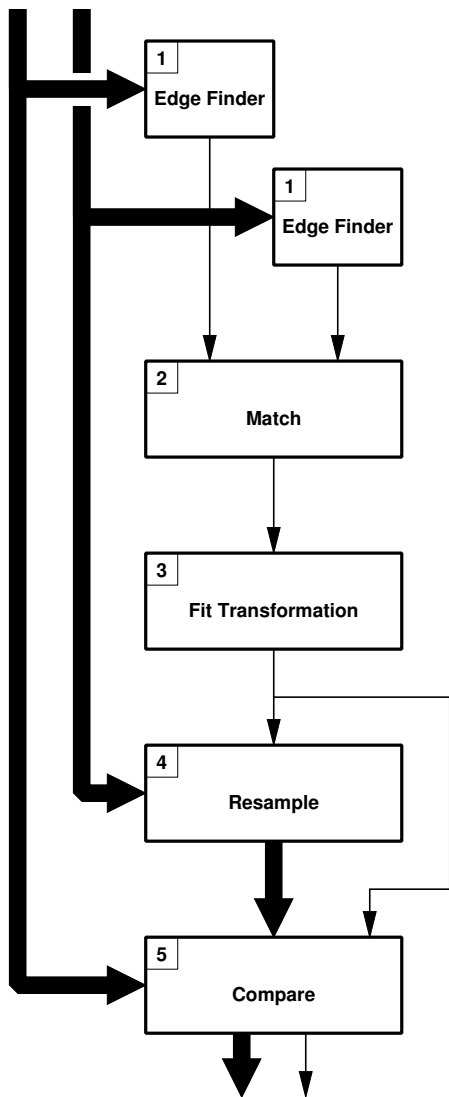
Our overall objectives are to

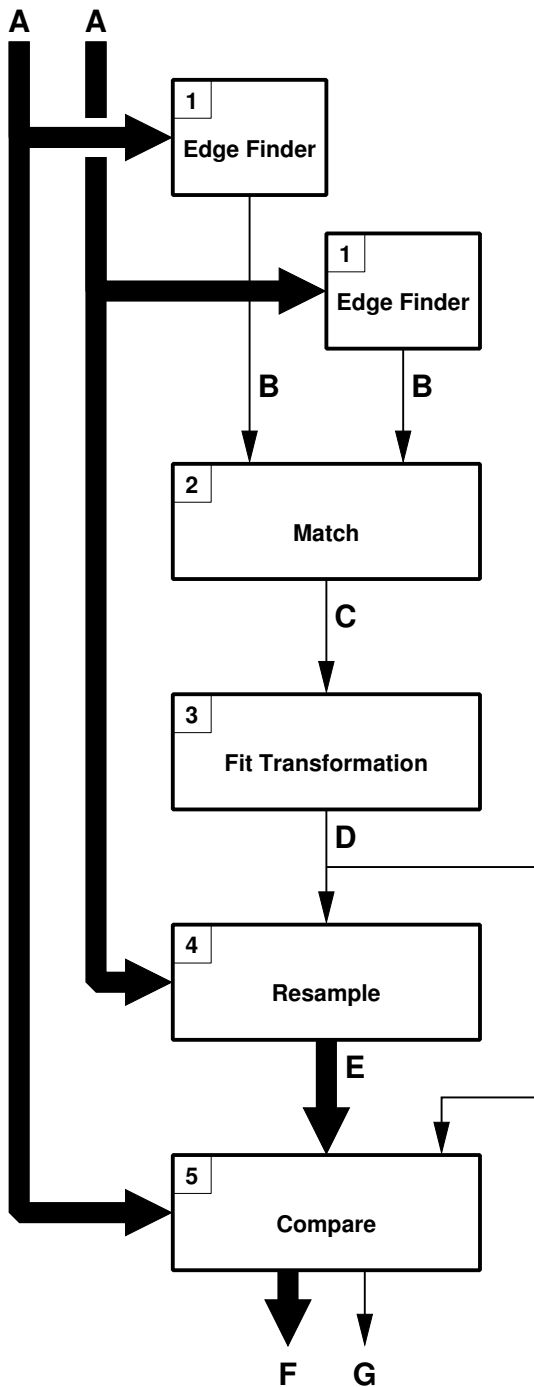
- Accept two images that show approximately the same field of view of the same scene, taken from approximately the same point of view. The actual image sizes (in pixels) may be very different. Overall scene brightness and lighting details may be quite different.
- Resample one or both images as required to provide two images of the same size that are better aligned with each other.
- Compare the two images and show the user where the remaining differences are, both in terms of their location in the image and their spatial frequency.
- Generate an “image distance” measure that tells the user how dissimilar the two images are. This measure will take into account some basic properties of the response of the human visual system.

Of course, we also need to evaluate how well our method works.

3.1 Pipeline

We propose to accomplish these objectives using a “pipeline” of software modules. Figure 3.1 provides a schematic overview of the pipeline with lines indicating data flow. The wide lines indicate image data, while the thin lines are numeric data.





Key to Flow Labels	
A	Original image (RGB or greyscale).
B	Edge location and intensity gradient data (4-tuples of data for each image scale).
C	Lists of matched points (pairs of (x,y) locations for each image scale).
D	Transformation specification (affine transformation expressed as 3x3 matrix).
E	Modified (resampled) image.
F	Map of differences (raster image).
G	Numerical results (image distance, etc).

Figure 3.1: Data flow diagram

The processing phases in the diagram are:

Phase 1: Edge Finder Read in one image, find “edge” features at multiple spatial scales. (This is applied to each input image individually.)

Phase 2: Match Read in the lists of features from two images and find a pairwise matching between features in the two images that maximizes a “quality of match” function we define.

Phase 3: Fit Transformation Calculate the geometric transformation that, when applied to the one image, aligns its features with the other image, as well as making it the same size.

Phase 4: Resample Resample one image according to this transformation, yielding two images that are the same size and aligned.

Phase 5: Compare Determine the remaining differences, show them to the user, and calculate a “distance” number.

This description implies that the algorithm forms a true pipeline, with each piece of information passing through each stage just once, and with no loops. That is the easiest way to understand the process initially, and in fact many moderate-sized images (up to about 512 pixels wide) can be handled this way. However, there are some circumstances in which a more complex data flow is used.

If the images are large or the alignment is poor, the matching process is too expensive if we apply it directly to the large images while allowing some reasonable amount of initial misregistration between them. The matching process looks for potential matching points within a region defined by a “bounding box”, and this must be large enough to include the largest distance between matching features that we expect to be present in the input images. We normally set this to 10% of the width of the images. Unfortunately, the number of edges in the graph, and thus the memory requirements of Phase 2, are roughly proportional to somewhere between the second and fourth power of the input image width. The actual exponent depends on which level of detail in the image the edge data came from. At the lowest level (finest detail), doubling the image dimensions tends to include additional edges in the image, and Phase 1 may find up to four times as many edge points. If both images are doubled in size, there are up to 16 times as many matches within any given bounding box size. At higher levels of detail, Phase 1 usually finds the same features in a large image as in a small image of the same subject, so there are few or no new features, but each feature has twice as many pixels defining it at the larger image size. Here, a doubling in image size creates about four times as many potential matches for a given bounding box size. Running time also increases along with edge count, though in a less predictable way. This power-law relationship between image size and matching cost means that it is not practical to register large images in a single pass.

If the images are poorly aligned, the matching process will produce many incorrect matches. The fitting process (Phase 3) employs a “pruning” technique to attempt to eliminate incorrect matches (see Chapter 6 for details), but it can be overwhelmed if the majority of the matches are incorrect.

To circumvent both of these problems, we use an iterative multiscale matching technique. We begin by matching only the edges from the coarsest level of detail from the two input images. This level generally contains relatively few edges, and may also be considerably smaller than the original image (see Section 4.1.2 for details), so the matching is tractable. We fit a transformation to the matched points. This transformation is then applied to the edge features found at the next lower level (finer detail) in one of the images before we perform the matching at that level.

The transformation found from the highest-level match is only an approximation to the correct transformation, particularly if the higher level edge information has been downsampled. However, it tends to move the two images in the correct direction, toward better alignment. When this transformation is applied before matching at the next finer level, it brings points which truly correspond with each other closer together. This yields a higher proportion of correctly-matched points in the matching process, which generally produces a more accurate transformation from the next level of transformation fitting. This new transformation is then applied to the next lower scale of data before matching, and so on. The iteration continues until we have matched and fitted the lowest level (finest detail) edges.

Because the iterative process “pulls” the images into alignment as it operates, we can gradually decrease the size of the bounding box used during matching. This keeps the matching process tractable at the lower levels where there are many more points. The process is also more powerful. We have seen test cases where the simple pipelined approach (match all levels, followed by fitting all levels of matching) failed to find the correct transformation, yet the iterative multiscale technique succeeded using exactly the same output from the Phase 1 edge finder.

Parts of this process are described throughout the text as necessary, and in particular in Section 6.3.

3.2 Multiresolution Issues

Several of the phases in the pipeline involve decomposing images into details at multiple spatial scales. This can be thought of as applying a series of bandpass filters to the image to separate it into multiple components, each limited to a certain scale of detail. This is often called a “multiresolution” representation of the image.

Common wavelet transforms (see [69]) divide the information content of an image into frequency bands that are spaced a factor of 2 apart (octave spaced). Typically, for an image that is M pixels wide, a wavelet transform will give us $\log_2(M)$ “levels” of detail.

Level 1 contains all the highest spatial frequency information, the finest detail in the image. To a first approximation, it contains information at spatial frequencies from 0.5 cycles/pixel (the highest frequency that can be represented in a sampled image) down to 0.25 cycles/pixel. Level 2 contains the information at approximately 0.25 to 0.125 cycles/pixel, and so on. In general, level N carries spatial frequencies in the range 2^{-N} to $2^{-(N+1)}$ cycles/pixel.

There are several reasons for using a multiresolution decomposition. First, the human visual system (HVS) is more sensitive to information at some spatial frequencies and less sensitive to other frequencies. Phase 5 needs to be able to separate information by spatial frequency in order to weight it by visibility in the calculation of image distance. This could be done using the DFT, but the DFT loses all information about where differences occurred in the image, and we need the location information as well. A wavelet transform provides coarser frequency resolution than the DFT, but there is still enough precision to provide a good approximation to the CSF we desire, while also providing the spatial location of differences.

Second, the fine detail (high spatial frequencies) may be greatly different between the two images, depending on their origins. If the one image comes from digitized film or video, it will contain low-amplitude high-frequency noise everywhere due to film grain and/or electronic noise. Computer-generated images (CGI) generally lack this noise, though they may have other noise due to stochastic sampling techniques used during rendering. When comparing digitized film to a rendered image, we would expect a substantial difference between the images at the finest spatial scales, but at coarser scales the images should be substantially the same. If the algorithm displays differences separated by spatial scale, we can verify that all the

differences are at the finest scales.

Images produced by halftoning, dithering, and some non-photorealistic rendering (NPR) techniques quite deliberately introduce a large amount of high-frequency structure to the image to compensate for lack of intensity resolution, or just for visual effect. If these techniques are operating properly, all of the differences between the original and the processed image should be confined to the finest spatial scales, where they are least visible. Again, by making Phase 5 display differences according to frequency, we can verify this.

However, the multiresolution representation is useful in Phases 1 and 2 as well. If we can separate the information in the image into bands of spatial scale, we can ignore scales where the correlation between the images is poor, while using the coarser, more robust features for alignment. It doesn't matter whether the high-frequency differences are noise or are deliberate patterns introduced by some processing technique, we still want to be able to ignore them and use the larger-scale features for alignment. The octave-spaced frequency bands provided by a wavelet transform provide enough frequency discrimination for this. At the same time, the wavelet transform retains the spatial information that we need to do alignment.

Third, the two images that will be compared may differ greatly in size (dimensions in pixels) while still portraying the same scene. For example, suppose that one image is 250 pixels wide while the other is 1000 pixels wide. A feature that occupies 5 pixels in the first image will be about 20 pixels in size in the second. The finest detail in the large image will be completely missing from the small one. When matching features between these two images, we need to compare things that are about the same fraction of the image width in size, not things that are the same number of pixels wide.

This size difference is particularly common when dealing with halftoning, dithering, and NPR techniques, because they are often used to take a moderate-resolution continuous-tone image (e.g. 100-200 pixels per inch (PPI), 8 bits/pixel) and re-represent it for printing on a higher-resolution device with bilevel output (e.g. a laser or inkjet printer that writes at 600 or 1200 PPI but with only 1 bit/pixel). We would like to be able to compare the original continuous-tone image to the higher-resolution bilevel image.

Because of the factor of 4 difference in pixels per image width in our example images, information with a frequency of f cycles/pixel in the small image will have a frequency of $f/4$ cycles/pixel in the large image. Thus, we can compare information at similar visual scales between the two images by comparing level 1 of a multiresolution decomposition of the small image to level 3 from the large image, comparing level 2 to level 4, 3 to 5, and so on. (Figure 4.4 show an example of this.)

Of course, the ratio of image sizes won't always be a power of 2, so we can't always do the Phase 2 matching with spatial scales that match exactly. However, we can always select the alignment between levels in Phase 2 so that the spatial scales being matched differ by at most a factor of $\sqrt{2}$.

Thus, Phase 1 does a multiresolution decomposition of the input images and finds features at each scale separately. Phase 2 aligns levels between the two images based on their relative sizes, and then finds a "best fit" matching at each scale. Phase 3 combines information from all levels to produce a single geometric transformation. Phase 5 again does a multiresolution decomposition of the two images in order to calculate distance and display differences according to scale.

We also make use of multiresolution information in a different method of matching and transformation fitting that goes outside the simple "pipeline" model we've been discussing. We can perform matching and fitting one level of scale at a time, starting from the highest level (largest features). After fitting a transformation based on this level, we use it to prealign the edge data from the next lower level. This gives a better matching than we would have obtained without the prealignment step. The process iterates until all

levels are done. (We can also iterate within one level, though this is usually not needed). For full details, see Section 6.3.

3.3 Matching Issues

Perhaps the most novel aspect of the image alignment portion of this thesis is the use of a graph-theoretic matching algorithm to find corresponding features in the two images, rather than one of the more conventional methods (e.g. correlation, Fourier techniques).

This was originally suggested by Vishwa Ranjan’s PhD thesis [58], which represents 3D objects via a “union of spheres” representation. His work first converts 3D objects from their initial form (volumetric data (voxels), range data (e.g. laser scanner output), or a polygon mesh representing the object surface) into the more abstract representation of a set of intersecting spheres. Then he compares and registers 3D objects by generating a matching between the sets of spheres that represent two objects.

This thesis uses a similar approach. We first convert a raster image, which is in concept a 2D function sampled on a regular grid, into the more abstract representation of a set of (multiscale) edges. Then we register two images by comparing and matching features between the edge representations of the two images.

The primary advantage of this approach is its generality. The graph-theoretic matching algorithm used (in Phase 2) has no *a priori* assumptions about the likely form of the transformation needed to align one image with the other. It isn’t limited to considering translation only, or an affine transformation only, or any other such restricted domain. It simply finds the best global matching of points in one image to points in the other image, based on the similarity of several properties of the points being paired.

Of course, when we reach Phase 3 in the pipeline, we must use some model of the transformation in order to fit its parameters to the matched pairs of points found — we can’t put off choosing a model forever. However, we might conceivably have more than one model available in Phase 3, and be prepared to choose one model over another based on the size of the residuals after fitting each model. This is plausible because Phase 3 is much less expensive (in terms of compute time) than Phase 2 matching, and because the Phase 2 output does not favour any particular transformation model (so Phase 2 does not need to be repeated when the model changes).

In contrast, most standard image registration techniques have assumptions about the range of transformations that are possible built into the heart of their algorithms. Switching assumptions means switching to a different algorithm entirely, or doing some sort of exhaustive search. Most standard techniques cannot handle images of widely different scales. (For example, simple correlation is very fast for finding a translation, but cannot handle scale changes. Converting the images to log-polar form and then using correlation can handle scale changes and rotation about the image centre easily, but cannot handle rotation or scale changes relative to an arbitrary centre without exhaustive search.)

On the other hand, the very generality of graph-theoretic matching is also the major disadvantage of this technique. Every pair of points is considered in isolation, without regard to whether the match being considered for a particular pair of points is at all similar to matches already made or being considered for other nearby points in the two images.

Any transformation that we will consider has the property that points which are near each other in one image will also be near each other in the other image. For every pair of points that are matched, we could calculate a “match vector” that shows how far and in what direction the point in the first image must move to end up at the corresponding location in the second image. If all of our matchings were correct (or as

correct as they can be given the ± 0.5 pixel uncertainty of the integer sampling grid), then all of the match vectors associated with points in a small area would be very similar, and the length and orientation of these vectors would change only gradually and smoothly across the area of the image. Most standard registration techniques make use of this either explicitly or implicitly, while our matching algorithm does not. This is a waste of valuable information.

In addition, the graph-theoretic matching algorithm is relatively expensive, and when applied naively its cost is proportional to the image size raised to some power. (See Chapter 5 for more detailed information.) We employ several techniques to deal with this.

3.4 Implementation

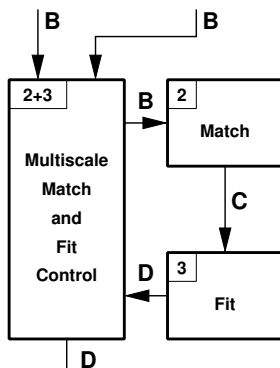
Note: This section discusses a specific implementation of software that tests the ideas described in this thesis. It may be skipped on first reading, particularly if the reader is not interested in implementation details.

The test implementation is primarily written in the C and C++ languages. Some of the higher-level control is provided by shell scripts. The system is designed to be easily restructured. In particular, each “phase” of processing described earlier in this chapter is implemented by a separate executable program (or sometimes more than one). These are made as independent of each other as possible. All of the data passed between phases is either passed as numeric data in an ASCII text file, or as a raster image.

This design makes it easy to try two different approaches for one phase of the pipeline, without affecting the other phases. It makes it easy to write different phases in different languages (although only C and C++ have actually been used). It is also easy to edit the data files during testing using an ordinary text editor, to display the data using graphing software like `gnuplot`, or analyze it using `awk` or `perl`.

For example, when we tried using an alternate implementation of the Phase 1 edge detector (see Section 4.3), only a few command names in shell scripts needed to be changed to use the new edge detector in place of the old one.

Another example: The iterative multiresolution matching/fitting process that is described in Section 6.3 was added after the “straight through” pipeline version of the software had already been in use for some time. We did this by adding one new module, marked “2+3” in the revised flow diagram below. (This is an excerpt from Figure 6.9.)



The new module simply replaces Phases 2 and 3 as far as the rest of the software is concerned. It accepts exactly the same edge data from Phase 1 as the original Phase 2, and provides exactly the same transformation data to Phase 4 as the original Phase 3 did. Yet the new module required little additional code to be written, because the matching and fitting processes are still performed by passing the data to the original, unchanged Phase 2 and Phase 3 modules. (See Section 6.3 for more detail.)

This modularity also makes it easy to use a subset of the full method when that makes sense. Performing Phases 1-3 only produces a transformation matrix, which is useful if you just want to know the amount of misalignment of the images. Using Phases 1-4 gives you two images that are the same size and aligned with each other, which may be very useful for photo editing and compositing applications. If the input images are already aligned, we may skip Phases 1-4 and pass the images directly

to Phase 5.

Within the thesis programs, image data is always handled in floating-point form. This allows us to handle high-dynamic range images without clipping of the intensity range. We can use high-quality resampling filters that have many terms, small coefficients, and negative coefficients without causing any numerical problems. We don't need to worry about negative values or overflow during our processing. The only real negative aspect of this choice is the extra memory needed.

3.4.1 Libraries

The software builds on top of several libraries.

Several of the processing phases (1, 4, and 5) must read and write images. We use a library and a collection of image tools collectively called “`dimg`” (Dave’s Image package), which was previously written by the author of the thesis. This package is quite flexible, and provides a number of functions that are needed by the thesis software that are not present in publicly-available image storage libraries (e.g. `PBMPLUS`, `libtiff`, or `pnglib`).

For the purposes of this thesis, the most important feature of the library component of `dimg` is its ability to read images stored using any of several high-dynamic-range encoding methods: floating point, Cineon 10-bit logarithmic, Pixar 12-bit linear, as well as the more common linear and “gamma-corrected” encodings in any width up to 16 bits per sample. In addition, the library makes it easy to convert an image using any of these supported encodings to any other encoding. All of the thesis software uses floating point image storage internally, so it can handle images from rendering software (e.g. Radiance) or high-dynamic-range photography without discarding highlight or shadow detail.

The `dimg` package also includes about 35 utility programs that perform basic operations like viewing images, cropping them, converting to and from other formats, resizing and rotating and reflecting images, and so on. Almost all of these tools can handle high-dynamic-range images in all of the above encodings without losing information.

On the other hand, the `dimg` package is reasonably large — about 30,000 lines of code, twice the size of the thesis code itself. Some of the features of `dimg` are not used by the thesis code at all. For any particular real-world application of the thesis methods, where the set of file formats and image encodings that needed to be supported was known and small, `dimg` could be replaced by a much smaller set of functions specific to those files.

Phase 2 uses a large graph algorithm package called LEDA, which was obtained from its developers under an academic source licence. For more details on LEDA licensing and alternatives, see Chapter 5.

We have written a small C++ library that handles 2D affine transformations using homogeneous coordinates. There are C++ classes for the 3×3 matrix and the 3-element vector that are needed, and only the first two rows of each are stored because the last row is constant. There are operations for initializing the matrix, multiplying two matrices, multiplying a matrix by a vector on the right, inverting a matrix, and printing a matrix. There is also a routine that decomposes an affine transformation into the product of a translation, a rotation, a uniform scaling, and a residual that is made as close to the identity as possible.

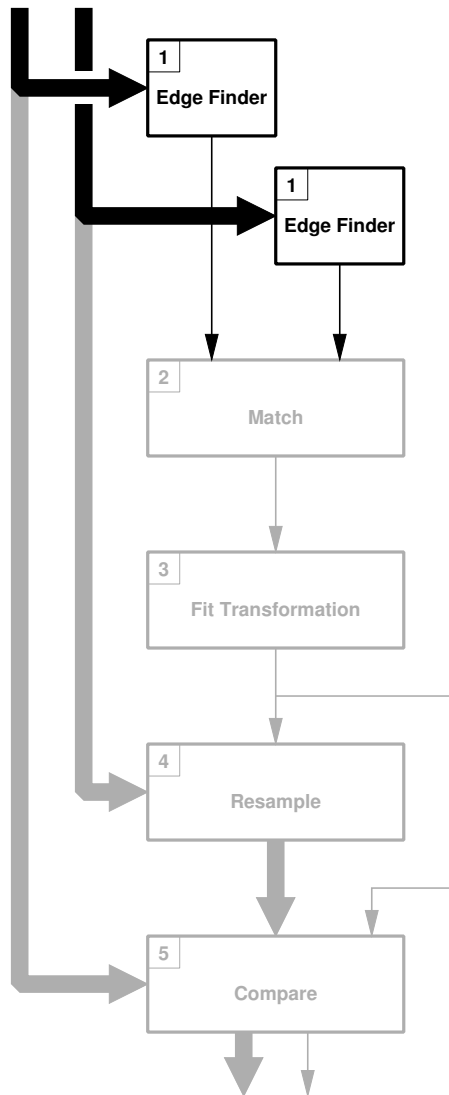
We also make use of a small subset of the library routines from Numerical Recipes in C++ [56]. Specifically, we use `gaussj` for inverting matrices, `select` for finding the Nth largest element of a set, `svdcmp` and `svbksb` for least-squared data fitting, and `pythag` which is used by the SVD routines. This software is available from Numerical Recipes Software.

3.5 Preview of the Following Chapters

The following five chapters describe in detail the operation of each of the five pipeline phases. At the end of each chapter, an “Implementation” section will discuss the test implementation of the methods of the chapter. There may also be a “Discussion” section that may discuss alternative ways of carrying out this phase, approaches that were tried and abandoned, or any other information that seems useful to another researcher interested in duplicating this work or learning from it. However, these sections are optional, in the sense that the information in them is not essential to understanding the thesis. Some readers may prefer to skip these sections on their first reading.

Chapter 4

Phase 1 — Feature Extraction



The task of this first phase of the processing pipeline is to identify a certain set of “significant” features in any single raster image. The location and some other properties of the features will be written out to a data file, ready for Phase 2 to attempt to match features between two such feature lists.

We will discuss two different algorithms for this phase. The primary one is based on a particular wavelet transform, while the other uses a Laplacian operator.

4.1 Wavelet Transform Choice

We want to find features at multiple scales (levels of detail) in the image, for reasons given in the previous chapter. To do this, we use a particular wavelet transform (and some associated processing) developed by Stephane Mallat and Sifen Zhong, described in “Characterization of Signals from Multiscale Edges” [41]. In that paper, the authors describe an image compression process that finds edges at multiple scales, records only the position of the edges plus the values of the wavelet transform at the edges (discarding all other data), and then approximately reconstructs the original images from this edge data. In this thesis, we use their algorithm only to the point of finding the edges. (We will discuss an alternative edge-finding method in Section 4.3.)

The Mallat-Zhong algorithm begins by applying a wavelet transform to the input image. The wavelet transform is an unusual one, with a number of useful features. We provide more detail on the transform in Section 4.4, but we’ll note two important ones here.

4.1.1 Output

The wavelet (detail) images produced at each level are in the form we want for further processing. The smoothing function used by Mallat and Zhong is a cubic spline function designed to have a shape much like a Gaussian. The wavelet function used by Mallat and Zhong is the derivative of the smoothing function, so it is approximately the derivative of a Gaussian. In other words, it is very similar to a Canny edge detector. Like the Canny edge detector, a local maximum in the absolute value of the detail function indicates the centre of an edge. (In effect, the centre is defined to be where the rate of change of intensity is the greatest.)

In the 2D case, we have two detail images, one containing X-direction detail and one with Y-direction detail. To a first approximation, these are actually first partial derivatives of intensity (within a certain bandwidth) with respect to X and Y. In other words, these are the two components of a gradient vector. The method for finding edges from gradients is described below. Most other wavelet transforms do not yield a gradient.

The Mallat-Zhong wavelet transform uses a smoothing filter (H) that is symmetric and a detail filter (G) that is anti-symmetric. Because of this, it does not cause image features to “drift” in location between the levels of the transform. Some other wavelet transforms (e.g. Daubechies [14]) use non-symmetric filters. These non-symmetric filters may not matter in applications where the data passes through both forward and inverse wavelet transforms, since the distortion in the forward transform is exactly reversed by the inverse transform. However, we use the forward transform only, and we may end up comparing features from different levels of the wavelet transform if image sizes differ, so any wavelet transform that causes an apparent edge position shift between levels is undesirable for our purposes.

4.1.2 To decimate, or not to decimate...

In any wavelet transform, at each level of the transform the input signal is convolved with a smoothing function to yield a low-pass (smoothed) result, and convolved with a wavelet function to yield a high-pass (detail) result. In the continuous domain, the smoothing function and the wavelet function applied at the next level are dilated by a factor of two compared to this level.

However, the wavelet transform is normally used via a discrete version of the continuous mathematics, called the Discrete Wavelet Transform (DWT). Instead of convolving a continuous input signal with continuous smoothing or wavelet functions, we do a discrete convolution of a sampled version of the input signal with a finite impulse response (FIR) filter that is generated from the smoothing and wavelet functions.

Also, a common wavelet transform is designed to decimate both the low-pass and detail results by a factor of two after the convolution. Thus, the two components produced at this level of the transform fit into the space originally occupied by the input signal, and no additional space is needed. If the wavelet transform is properly designed, this decimation by two does not lose any information, in the sense that the original signal can be exactly reconstructed at its original length from the two half-length products.

In terms of the dimensions of the original image, the smoothing and wavelet filters double in length at each successive level of the transform. (More precisely, the filters always have the same number of non-zero values, but the effective spacing between the non-zero samples doubles each level). The smoothing filter acts as a low-pass filter, and this doubling of the filter width for each level means that the upper frequency limit for the information contained in the output is reduced by a factor of two for each level.

Since the signal has been compressed by a factor of two due to the decimation step in the DWT implementation, the 2X “dilation” requires the smoothing and wavelet functions to remain the same size relative

to the sample spacing, not double in size. Thus, the common DWT algorithm convolves with the same pair of filters at each level. The implementation is simple and straightforward.

Because of this decimation by two, the input data normally needs to be a power of two in length. In the case of a 2D image, both dimensions need to be a power of two, though they do not necessarily need to be the same.

With a common wavelet transform, each subsequent level of the transform takes as its input only the smoothed output of the previous level, so each level works on only half as much data as the previous one. The total cost of a one-dimensional common wavelet transform with an input signal of size N is limited to $2N$ convolutions, making the entire algorithm $O(N)$.

There are two ways of extending the one-dimensional wavelet transform to two dimensions. In one, the so-called *standard* construction, we apply the full 1D wavelet transform to completion (all levels) on every row of the image individually. Then we apply the 1D wavelet transform to completion on every column of the result. (Or we can do columns first, followed by rows — it doesn't matter.) If we started with an image that is $M \times N$ pixels in size, the total number of (1D) convolutions needed is $4MN$. This is $O(MN)$.

The *nonstandard* construction is slightly faster. In this, we apply one level of the 1D wavelet transform to every row in the image. Then we apply one level of wavelet transform to every column in the result. When we're done, we have one smoothed subimage that is $M/2 \times N/2$ pixels in size, just $1/4$ the area of the original, and we apply the next level of the 2D transform only to this subimage. The other $3/4$ of the original area is occupied by three wavelet transform products, containing horizontally-oriented, vertically-oriented, and diagonally-oriented detail information. With this construction, the total number of convolutions is $4MN/3$. This is also $O(MN)$, but with a smaller multiplier.

The nonstandard construction would be most useful for an edge detector, because the “detail” images all retain the same aspect ratio as the original image, and there are a smaller number of them.

Figure 4.1 shows a 2D Haar wavelet transform applied to the image of a sphere using the standard and nonstandard construction. (Compare these to Figure 4.2, which shows the Mallat-Zhong wavelet transform process.)

The Mallat-Zhong wavelet transform does not fit this pattern. Their discrete wavelet transform does not decimate the smoothed and detail information after the convolution step. Thus, the new smoothed output can be stored in place of the input signal, but the detail output needs new memory equal in size to the input. In the case of a 2D transform, the convolution with the wavelet function is done to all the rows of the input image yielding an X-direction detail output. Then the wavelet function is convolved with all of the columns of the input image to yield a Y-direction detail output. Thus, each level of the 2D transform produces two new detail images the size of the original input image, plus a smoothed image that can fit back into the original image space.

Also, since there is no decimation of the data between levels, the smoothing and wavelet filters are actually dilated by a factor of two for each level of the transform. This is done at level l by inserting $2^{l-1} - 1$ zeros in between each original coefficient of the filters. Because the inserted values are zero, the convolutions don't become any more expensive, but the filters are wider in support. Note that the effect of filter spreading in the Mallat-Zhong transform is the same as that of decimation in the traditional DWT: the cutoff frequency for the information in each level is reduced by a factor of two per level. (This filter spreading is the identifying characteristic of a type of wavelet transform referred to as the “algorithme à trous”, after the zeros (holes) that are inserted in the filters).

Normally, the Mallat-Zhong wavelet transform is carried to $L = \log_2(\max(M, N))$ levels. If that

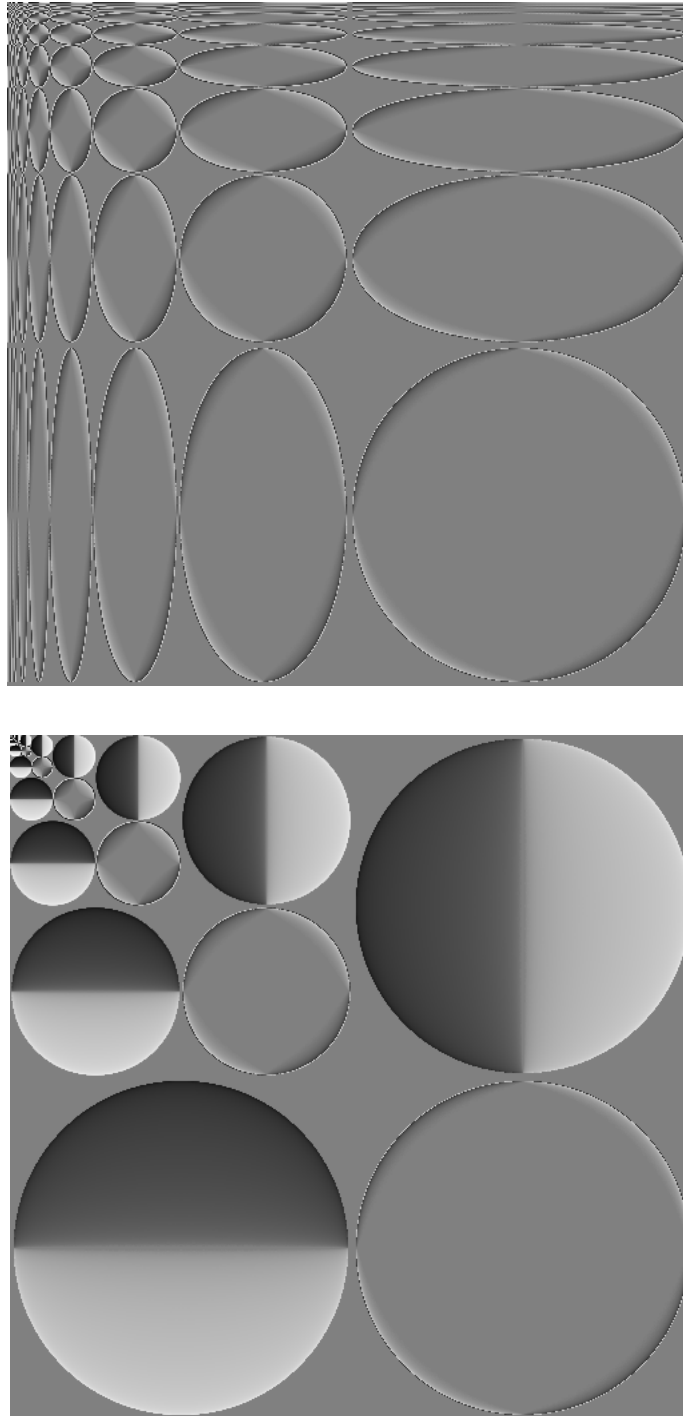


Figure 4.1: Haar wavelet transform, standard (top) and nonstandard (bottom) construction

is done, we need a total of $(2L + 1)$ times the original image space for all the detail images plus the final smoothed image. We perform $4MN$ convolutions at each level, for a total cost that is approximately $4MN \log_2(M)$ (assuming that $M \geq N$). This is definitely more expensive than the standard discrete wavelet transform, though no more expensive than a FFT.

This extra cost in time and space is sometimes worthwhile. Each of the detail images, at all levels of the transform, has the full resolution of the original input image. Thus, the image features we find are located with a precision of one pixel in the original image, no matter what the level. With a standard DWT, each higher level provides position information with half the resolution of the level below, and the information at the higher levels quickly becomes useless for precise alignment. Although the higher levels of the Mallat-Zhong transform isolate physically large features in the image, they do so with half-pixel precision. In comparison, a standard DWT would provide only one-pixel precision in the location of edges at its lowest level, two-pixel precision at the next level, et cetera. Even the lowest level of output does not provide full input-image precision.

On the other hand, retaining full-size images through all the levels of the transform is not necessary if we use the iterative multiscale matching and fitting process described in Section 6.3. In that case, the higher-level edge information need only provide approximate alignment information that is later refined by information from lower levels. In fact, it is desirable to reduce the amount of higher-level edge information by decimating the images in order to speed up the matching. Still, we want to retain full input-image resolution to at least the second level of detail. (Level 1 is often very noisy and in those cases level 2 provides better matching information.)

To deal with this, we have developed a method of performing the wavelet transformation that is a generalization of both the DWT and “algorithme à trous” methods.

We start by observing that one thing is invariant in both wavelet transform methods: the effective spacing of the non-zero samples in the smoothing and wavelet filters is $2^{(l-1)}$ samples (in terms of input image samples), where l is the current level of the wavelet transform. The DWT achieves this by decimating the image at each level while the algorithme à trous spreads the filters instead, but both maintain that relationship at all levels.

The core of our wavelet transform algorithm can do both. It can use any (power-of-two) filter spreading factor at any level of the transform, and it can also either decimate the output of the smoothing and wavelet filters or leave them at the same size. An initialization process calculates filter spreading factors and per-level decimation flags in such a way that exactly one of the two techniques is used to double the effective filter width between adjacent levels. In this way, we maintain the same effective factor-of-two cutoff frequency change between levels regardless of exactly what pattern of decimation vs. filter spreading we use.

In practice, our usual policy is to retain full image size to level 2 of the transform at least, and then begin decimation. We continue decimation at each level as long as the width and height of the image remain 50 pixels or greater. We return to constant-size mode to keep the images from becoming smaller than that, in order to prevent the highest-level images from becoming too small to provide useful position information.

It is worth noting that this “hybrid” method cannot be a true wavelet transform any more, in the sense of containing all the information in the original image and being invertible. Mallat and Zhong have proved that the pure implementation of their transform, with no decimation, is a proper wavelet transform (with plenty of redundancy). However, when the same filters are used with decimation, some information is necessarily lost. The output of one stage of the transform with decimation yields three images, each one-quarter the size of the input image from the previous level. With only three-quarters as many independent values in the

output as the input, information has been lost. However, we are using this method as an edge detector, not an invertible transform, so this is not a problem.

The memory and computation time requirements of this method lie somewhere between those of the DWT and the standard Mallat-Zhong transform, depending on what fraction of the total number of stages involve decimation.

When we are using iterative multiscale matching, the Phase 1 edge detector remains an independent pipeline phase. However, we normally use decimation during Phase 1 in this case. When we are using the simpler “straight through” pipeline for matching and fitting, we normally operate Phase 1 without decimation. (This case is only used for small images.) There are two other configurations possible, but they are not normally used.

4.2 Method

In concept, the control flow in Phase 1 is extremely simple: read in the image, perform all levels of the wavelet transform, identify edges, and write out the edge information. The actual implementation deviates from this in order to use less memory, but we will ignore that here.

4.2.1 Input

The first step is to read the input file into allocated memory, converting it from whatever encoding was used in the file to a floating point value linearly proportional to intensity. Values are scaled so that black is represented by zero and “reference white” for the scene is represented by 1.0.

The program can read RGB colour images, but will convert them to monochrome before doing the feature extraction. In this phase, we care only about extracting the spatial structure of the image, so colour information is irrelevant. The monochrome conversion is done with the common NTSC luminance formula

$$Y = 0.299R + 0.587G + 0.114B$$

Technically, these specific weights are only precisely correct for the red, green, and blue phosphor chromaticities and white point specified in the NTSC television standard, and they’re likely at least somewhat wrong for any RGB image found today. Also, we apply the formula to RGB values that are linearly proportional to intensity, while NTSC television encoders apply them to gamma-corrected values — in this case our use is closer to correct than what the TV encoders do. But it doesn’t matter; this is simply an expeditious way of converting a RGB image to a monochrome one before further processing, and it works fine for this application.

Optionally, we can convert the input image from its usual linear form to logarithmic form (pixel values proportional to logarithm of intensity). This causes image structures in the shadows of the image to contribute as much to the features extracted as highlight areas.¹

To see why this is useful, consider an object whose surface reflectance varies by a factor of 2, between 45% and 90% reflectance. If this object appears in a bright well-lit area of the scene, the actual pixel values in memory may range between 0.5 and 1.0. If this same object is in a shadow area of the scene, the light falling on its surfaces may be a factor of ten less, and the object ends up with pixel values in memory that range between 0.05 and 0.1. The object in bright light has ten times as much difference between its parts,

¹Thanks to John Canny for suggesting this.

in absolute light terms, as the same object in shadow. The wavelet transform is linear, so ten times as much signal amplitude at the input to the transform results in ten times as much signal amplitude in the output. Since we will ignore maxima below a certain threshold (see below), this process favours bright areas of the image for finding features and tends to suppress information in the shadows.

By taking the logarithm of intensity, a fixed intensity difference in reflected light produces the same difference between pixel values no matter what the intensity of the illumination on that area of the scene. In the case of our example, a 2:1 intensity ratio between two areas in the input images will become a pixel value difference of $\log(2)$ in the log images for both highlight and shadow areas.

However, this isn't the default because photographic images often have noisy shadow areas, and it is better not to give shadows equal weight with these images. So the log conversion is an option.

4.2.2 Wavelet Transform

Once we've read in the image, we perform a slightly modified version of the Mallat-Zhong discrete wavelet transform.

Like the Discrete Fourier transform, the math of the Mallat-Zhong wavelet transform assumes an image that extends to infinity in both X and Y directions. In practice, real images are finite in size. We must provide some way of finding the value of pixels outside the original source image.

The Fourier transform handles this by assuming that the infinite-size image is actually periodic, and the finite source image is a sample of exactly one period of the infinite image. In effect, the infinite-size image is simply the infinite plane tiled with copies of the source image, all oriented the same. Unfortunately, this produces apparent discontinuities at the edges of the tiles.

The Discrete Cosine Transform (DCT) deals with this slightly differently. It assumes that the infinite plane consists of copies of the source image and its X- and Y-reflections arranged so that each edge of each tile is always identical to the immediately adjacent edge of the adjacent tile. This produces an infinite-sized image with no edge discontinuities. (It also produces an image that is symmetric about the X and Y axes, an even function. If you do a Fourier Transform of such a function, all of the sine terms are zero and only the cosine terms remain. This is one way of deriving the DCT.)

Mallat and Zhong recommend extending the image by reflection in the same manner as the DCT. In fact, they actually suggest making the image twice the dimensions in memory (four times the number of pixels) and then treating it as periodic. This simplifies addressing, but the wavelet transform then takes four times as much memory and time. This approach is necessary if you want to do an inverse Mallat-Zhong wavelet transform, because the intermediate images produced during the inverse transform are not mirror-symmetric in any simple way. Since there are three images at one level (the smoothed image plus two wavelet images), these images are stored in floating-point form, and they contain four times as many pixels because of the reflection, so we end up with 48 bytes of data per input pixel per level of the wavelet transform.

However, we only use the forward transform in this thesis, and it is possible to perform it using only enough memory for the original-size input image. This reduces the memory needed to 12 bytes per input pixel per level. The effect of reflected copies of the input image extending to infinity is produced by an algorithm that "folds" pixel addresses that are outside of the source image back to the corresponding pixel address in the source image. This address pattern is calculated once and stored in a lookup table, so it doesn't add too much cost to the convolution operations.

A simpler if less elegant method is simply clamping the pixel addresses to the boundaries of the source image, rather than folding them. This is equivalent to replicating the first and last row of the source image

to infinity in the Y direction, and replicating the first and last column to infinity in the X direction. It is not as nice in theory as a mirror image, but it does prevent edge discontinuities, and it is cheap to implement. In practice, it works almost as well.

Finally, under some conditions we decimate the smoothed and wavelet data by a factor of two. See Section 4.1.2 above for details.

Figure 4.2 shows the progress of the wavelet transform applied to a simple disc test image, and Figure 4.3 shows the result of transforming a more complex sample image. The first column of images shows the smoothed images ($S_{2^{j+1}}^d f$ in the Mallat-Zhong paper’s notation) for successively larger values of j . The extreme upper left image is the original input image $S_1^d f$. The second and third columns are the X- and Y-wavelet images $W_{2^{j+1}}^{1,d} f$ and $W_{2^{j+1}}^{2,d} f$. The remaining columns will be described in the next sections. Higher levels of the transform appear lower in the images.

Note: the standard mandrill image is 512×512 pixels, but the last row is all zeros (black). This black edge is virtually invisible to the naked eye just looking at the image, but the discontinuity it produces in intensity represents a high-amplitude high-frequency edge that shows up clearly when the image is passed through the wavelet transform. (That is how we discovered the row of zeros in the first place — we thought there was a bug in the Phase 1 code, but it was really a bug in the input). For most purposes, we just discard that final row, leaving the image slightly non-square — that doesn’t bother any of the thesis code.

4.2.3 Finding Maxima

Once the wavelet transform has been done, we find edges at each scale. Because of the way the wavelet functions were designed, the X- and Y-wavelet images that have been calculated at each level of the transform are the partial derivatives of the original image smoothed by a smoothing function at a particular scale. The two partial derivatives give us the gradient of the smoothed image. (The gradient is a vector-valued function.)

First, we convert the gradient into polar form, calculating the length (modulus) of the gradient and its orientation. Intuitively, the orientation of the gradient tells us the direction of steepest “ascent” (most rapid change from dark to light), while the modulus of the gradient vector tells us just how fast intensity is changing in that direction.

Again referring to Figures 4.2 and 4.3, the fourth column of images shows the modulus of the gradient while the fifth column shows the orientation of the gradient. The orientation is coded using either intensity or hue (depending on whether you are looking at a B&W or colour version of the figure), and the disc at the top of the column is a key that shows the display corresponding to each direction. Think of the gradient vector having its origin in the centre of the disc, with the end of the vector pointing at the colour for that orientation of the vector. Thus, a gradient pointing to the positive X axis is displayed as either mid-grey or magenta (purple).

Now, the gradient achieves its maximum value in a local area in the “middle” of an edge in the smoothed image. More precisely, a maximum in the gradient occurs at an inflection point in the intensity function (which is the image). So, to find edges, we look for the locations where the gradient modulus is greater than its two neighbour pixels in the direction of fastest change of intensity. (By neighbour, we mean one of the 8 pixels that are immediately adjacent to this one.)

Intuitively, if the gradient modulus values were used to define a 3D surface, with surface height at each (x, y) location corresponding to the modulus of the gradient at that location, then what we are trying to find is “ridge lines” on the surface. A point on the ridge line has the property that it is higher than either of its

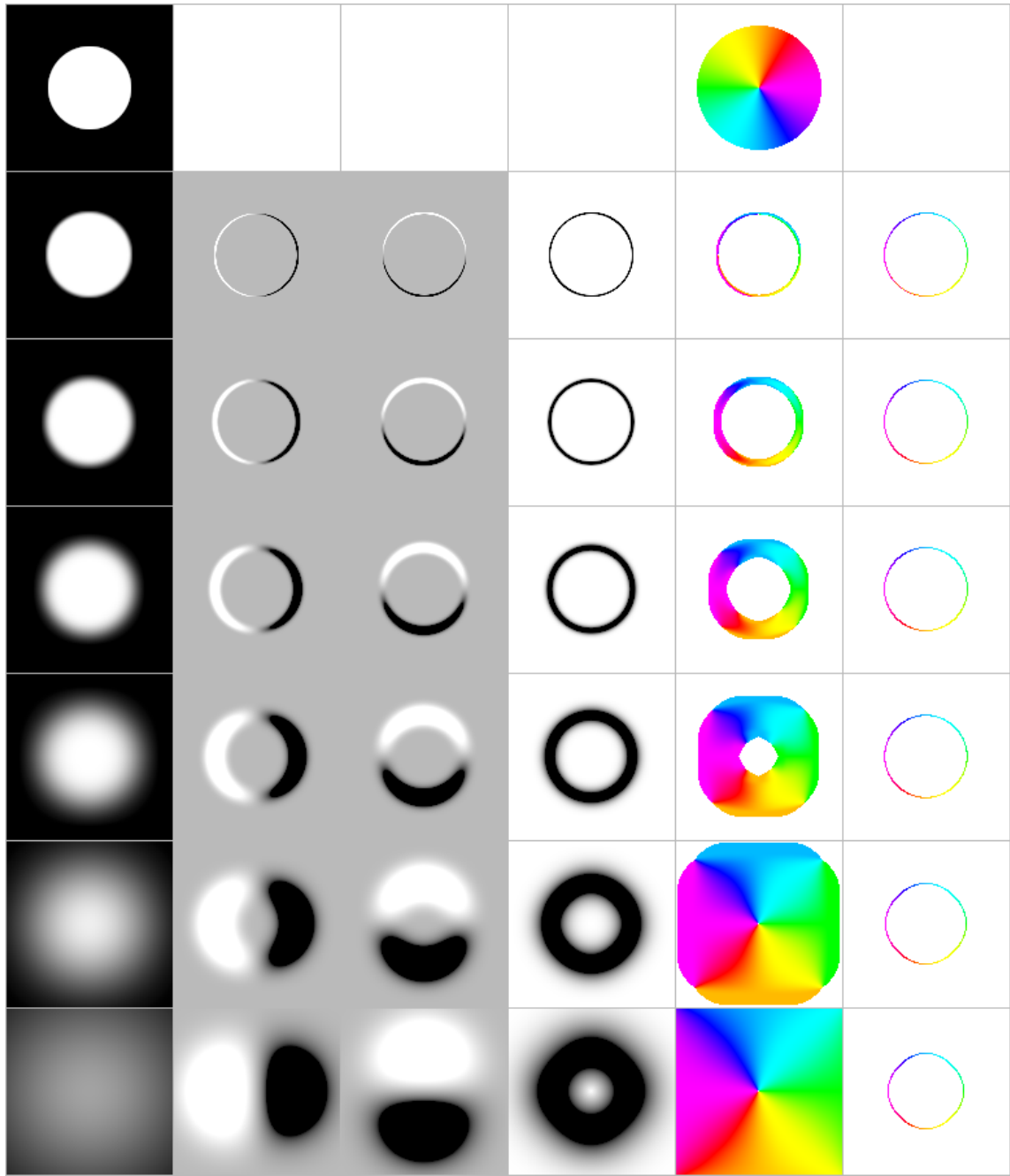


Figure 4.2: Wavelet-processed disc image

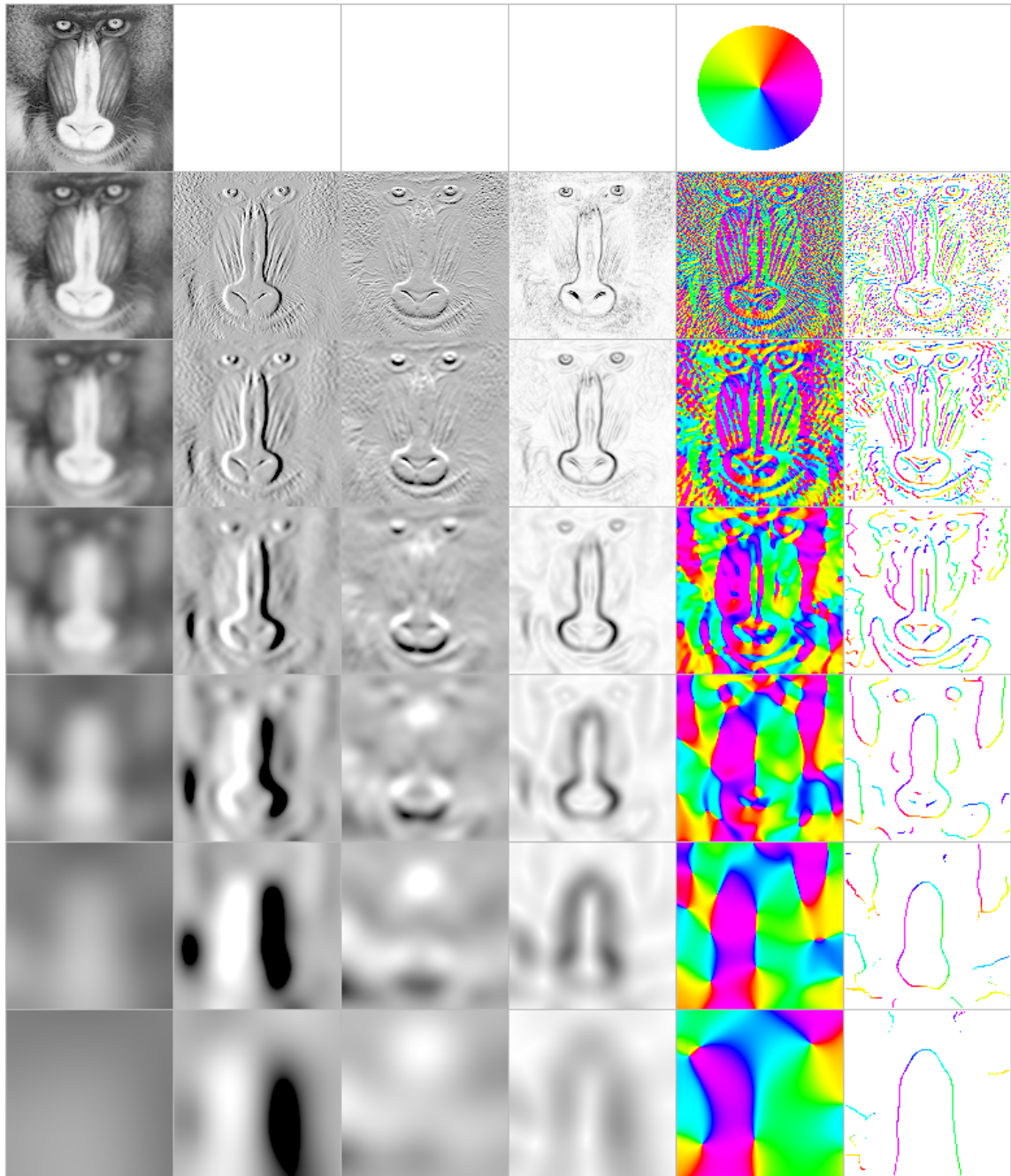


Figure 4.3: Wavelet-processed “mandrill” image

neighbour pixels in the direction of steepest descent (i.e. down the flanks of the ridge), even though it is generally not higher than one or both of its neighbours along the ridge top.

For everywhere but the outermost pixels, the algorithm determines the orientation of the gradient at that location. A line through this point at that angle will pass between two neighbour pixels on one side, and between two other neighbour pixels on the other side. On each side, we determine the modulus of those two neighbouring pixels, and then use linear interpolation to estimate the value of the modulus of the gradient at the point where our imaginary line crosses the centre of the adjacent row or column.

Once we have interpolated the two “neighbour” modulus values, we compare them to the modulus value at this pixel. If this pixel’s gradient modulus is greater than or equal to both of its “neighbour” moduli, and strictly greater than at least one, we declare this point to be a local maximum, corresponding to an edge in the smoothed image. To avoid cluttering the data with noise, we discard maxima whose modulus is below some fixed threshold, even if it is higher than its neighbours.

In Figures 4.2 and 4.3, the sixth column shows the edges found at each level. In the case of the round disc test image, note how accurately the edges found follow the size and shape of the object in the original image across many scales. This will be important when comparing images of greatly different size.

Figure 4.4 shows the maxima pixels found in the “mandrill” image at two different scales, 128×128 and 512×512 . (The 128-pixel results are magnified by a factor of four relative to the 512-pixel results in order to show features at the same size.)

Note how well the features at level l in the left image match the features at level $l + 2$ in the right image. This means that we should get good results when matching maxima found at this pair of levels, despite the fact that one source image was four times as large as the other. (Even larger size ratios work, but it becomes difficult to display them on paper.) These sample images were created with no decimation, so although the two images show almost the same contours, there are roughly four times as many points in the edge data for the larger image. This means that about three-quarters of the points from the larger image will remain unmatched in Phase 2. This is not a problem, but it does mean some effort was wasted.

If we turn on decimation starting at level 3 (our usual way of using it), the process will reduce the size of all of the higher levels of edge data. In combination with the way that levels are aligned when the two images are different sizes, the decimation process reduces all of the higher-level images that are compared to approximately the same size (within a factor of $\sqrt{2}$). This provides approximately the same number of edge points from both images at most levels, no matter how different the original image sizes are.

Figure 4.5 shows the edges found for the same pair of source images as the previous figure, but with decimation turned on. The difference in image size is now only a factor of two for the lowest-level pair, and the other pairs are all the same size.

4.2.4 Output

At this point, we need to write out the maxima points found at each level of the wavelet transform. However, first we check to see if we found too many points.

The Phase 2 matching algorithm requires time and space that is proportional to the number of input points raised to some power, and too many data points will make it run too slowly or fail entirely. Thus, we need to limit the number of points per transform level. The limit can be set by the user, but our experience suggests that values around 5000 seem to be a good compromise.

Suppose our limit on points per level is L , and the number of points we have found at this level is P . If $P > L$, we need to discard $P - L$ points. We could just do this randomly, but instead we arrange to keep



Figure 4.4: Mandrill128 levels 1-4; Mandrill512 levels 3-6

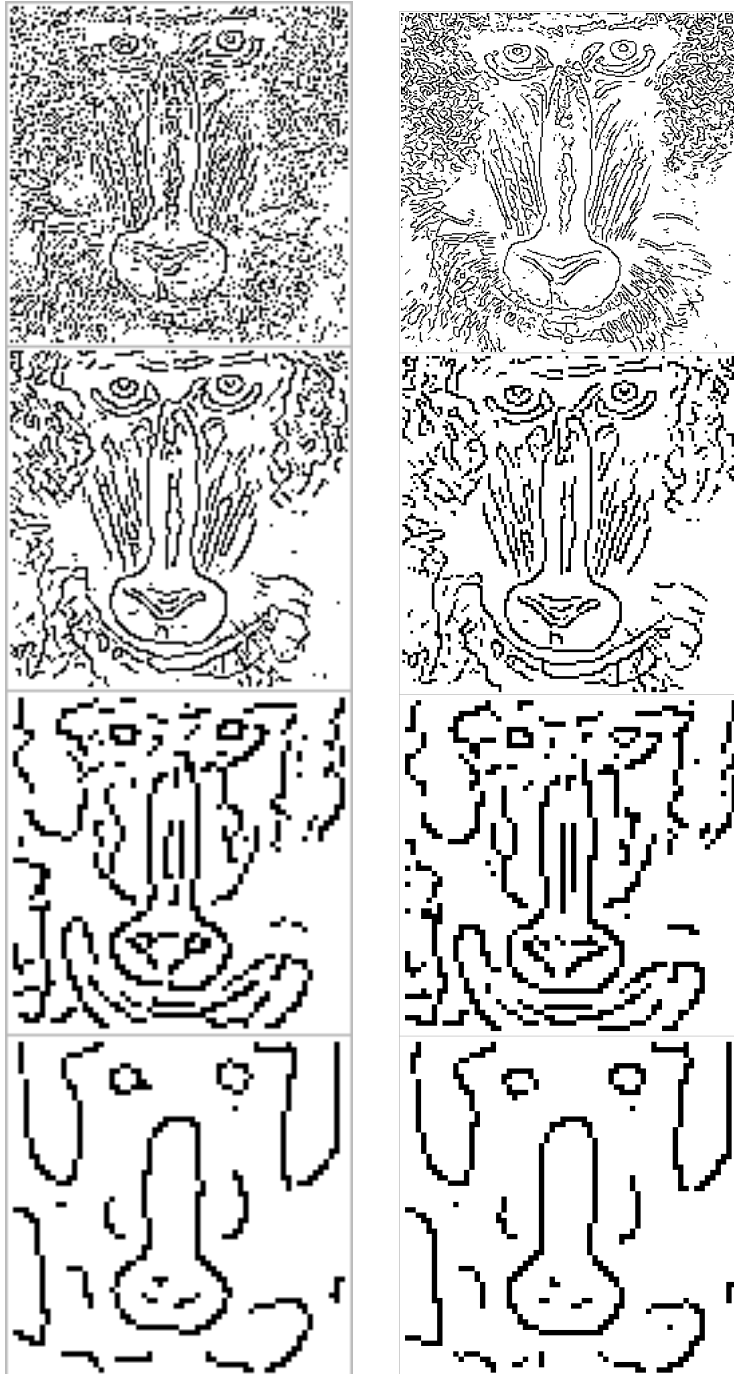


Figure 4.5: Mandrill128 levels 1-4; Mandrill512 levels 3-6, with decimation

the points that have the L largest gradient moduli, discarding the points with the $P - L$ smallest moduli.

Once we have our final list of maxima, we write them out to a data file. For each point, we write its position within the image, the modulus of the gradient at that point, and the orientation of the gradient. All of these pieces of information will be used in the next phase.

Optionally, the images shown in Figures 4.2 and 4.3 are also generated and written out.

4.3 Alternate edge finder: the Laplacian

We also implemented a second edge extractor based on the Laplacian, to see how well our (somewhat unusual) wavelet-based edge detector compares to an algorithm based on a more common edge detection operator.

We kept the two algorithms as similar as possible. In particular, we built a multiscale edge-finder like the Mallat-Zhong wavelet transform. (The Laplacian is most often used only to find edges at a single scale.) The multiscale decomposition of the image still uses the same 4×4 H filter, so the “frequency response” of the smoothing filter is identical for both algorithms. The Laplacian-based algorithm can also decimate the image (or not) between levels, with the same behaviour as the wavelet version. The only significant difference between the two edge extractors is the high-pass filter used, and the method of identifying edge pixels.

4.3.1 Algorithm

The high-pass filter we use is a 2D variant of the Laplacian whose kernel² is

$$\frac{1}{6} \begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix}$$

This is applied by direct 2D convolution to the input image, giving a single output image. This is faster than the equivalent step of the Mallat-Zhong wavelet transform, which generates two separate directional derivative images.

Figure 4.6 shows the results of a simple version of the Laplacian method applied to the disc test image. The first column is the successively-smoothed input image; this is identical to the first column of the wavelet transform output. The second column shows the Laplacian, encoded so that zero is mid-grey while positive values are brighter and negative values are darker.

Once we’ve calculated the Laplacian, we need to identify which pixels lie along edges. The Laplacian calculates the second derivative of intensity, so edges occur at zero crossings in the Laplacian. (The middle of an edge is where the intensity is changing the fastest, so the first derivative of intensity is at a maximum and the second derivative of intensity is zero.)

Initially, we looked for edges by identifying all pixels where the Laplacian has both positive and negative values within the 3×3 groups of pixels centred on the current pixel. This has the problem that edges were drawn as anywhere from two to four pixels thick. This has a bad effect on the matching, since the number of edge points (which turn into graph nodes) more than doubled. Ideally, every edge in the image should be converted into a single-pixel-thick line. The wavelet method does a good job of this, but we can’t duplicate

²kernel courtesy of Dr. Matt Yedlin

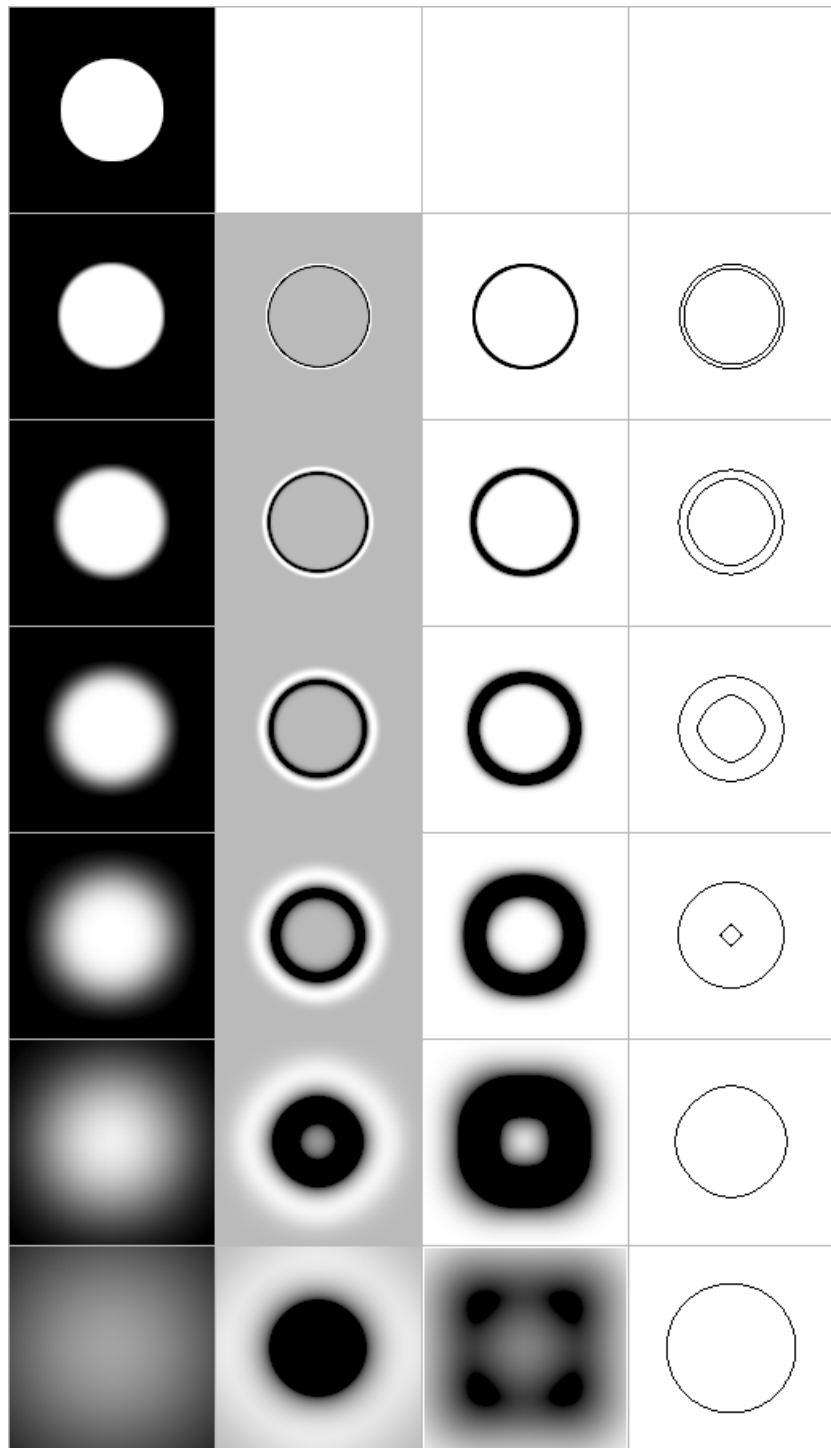


Figure 4.6: Laplacian-processed disc image

its method directly because the Laplacian does not give us gradient direction in any easy-to-extract form. Instead, we modified our rule for “edge points”: Now we declare a point to belong to an edge if and only if the Laplacian is non-negative at this pixel and the number of neighbour pixels with non-negative Laplacian values is between three and six inclusive. The first restriction eliminates half of the points near the edge, and the second restriction eliminates points where the edge is too far from the centre of the 3×3 neighbourhood.

However, there were still problems. Some of these are fundamental problems with using the second derivative. With a first-derivative edge detector, the absolute value of the derivative is large wherever intensity is changing rapidly (i.e. an edge). The centre of the peak tells us the centre of the edge, while the magnitude of the peak gives us some idea of the amount of intensity change across the edge. A simple threshold applied to the magnitude of the first derivative can be used to eliminate insignificant edges. In comparison, the Laplacian is zero at an edge, so the value of the Laplacian provides no information about the significance of the corresponding edge in the image. Worse, the second derivative is zero at local minima in the absolute value of the first derivative. These occur at locations where the intensity variation is least in the original image, which is not an edge. Thus, an edge detector based on the Laplacian alone tends to find “false edges” in areas of minimum variation, as well as true edges. You can see this in the rightmost column of Figure 4.6 — there are false edges inside the area of the disc at most levels.

So, we need some additional method to get rid of these false edges. A Web search found a web page at Rice University [9] where the authors noted the problem of false edges when using the Laplacian. Their solution was to calculate the local variance of the original image (not of the Laplacian of the image) and compare it to a threshold. Real edges are accompanied by a large variance in the image and are accepted, while false edges correspond to a small variance and are rejected.

We added essentially this solution to our Laplacian edge detector, although for reasons that will be apparent later we used the standard deviation rather than variance. The standard deviation is computed on the same 3×3 neighbourhood as the Laplacian. Done at the same time, it adds little extra cost. The third column of Figure 4.6 shows the variance in the smoothed input image, coded so that white means zero while increasing blackness means increasing standard deviation. (However, the fourth column in this figure was computed with the threshold set to zero, effectively disabling the variance test.) This threshold provides an effective way to eliminate false edges; we will see an example soon.

For use in our method, any edge detector should provide some additional information about the edges it finds in order to aid the matching process. The wavelet edge detector supplies the local intensity gradient vector at every edge point, and passes that to Phase 2. The Laplacian does not calculate gradient, but we wanted to provide some sort of numerical value that provides an indication of edge “strength”. We tried several alternative measures, but once we had the intensity variance of the input image available, we decided to use the standard deviation (square root of variance) as our edge “magnitude” measure. We used standard deviation simply because we wanted a measure proportional to edge magnitude, not the square of edge magnitude.

This is actually rather similar to what the wavelet method outputs. The standard deviation of intensity in a localized area of the image is roughly proportional to the rate of change of intensity over that small area. In other words, it is roughly proportional to the absolute value of the gradient vector at that location. We tried running a few test images (mandrill, peppers, Lenna) through both algorithms and looked at the size of their outputs, and then scaled the standard deviations by a factor of 3.6 to make their mean approximately the same as the mean gradient magnitude output by the wavelet method with the same images. This makes it easier to substitute the Laplacian method for the wavelet method without changing other software.

Then we had to determine a suitable value for the threshold. For the disc test image, with its flat top and infinitely steep sides, this is easy. Testing shows that any threshold value of $1e-40$ or larger eliminates the false edge in the flat top, while the threshold can be as large as 1.0 without losing the true edge. With a different disc image, using a linear ramp as its edge rather than an abrupt edge, the $1e-40$ threshold still eliminates the false edge, but any threshold larger than 0.1 starts losing the real edge. Experiments with photographic images suggest that a threshold of about 0.1 is in fact a good choice — it rejects false edges, and some true but uninteresting edges, while keeping most of the important true edges.

The wavelet method also has a magnitude threshold built in. It is not as important as the threshold in the Laplacian method, because the wavelet method does not produce false edges even when the threshold is set to zero. However, we normally use it to eliminate the less important edges in an image. It turns out that a threshold value of 0.1 gives approximately the same amount of clutter reduction for the wavelet method as it does for the Laplacian method (probably because we scaled the standard deviation to have about the same mean on the same image).

Figure 4.7 shows the effect of a threshold setting of 0.1 on the output of both methods. The first and third columns are the edges reported by the wavelet method, while the second and fourth columns come from the Laplacian method. The left two columns show the results with no thresholding applied, while the right two columns show the edges pruned by the threshold. As you can see, the two methods provide approximately the same amount of edge detail at each level with the same threshold.

Because the two edge finders were designed to be so similar, we can almost select one at random and use it in the pipeline. However, there is one significant difference in their output that other phases need to know about. The wavelet method outputs four numbers per edge point: the relative (x,y) position, plus the 2D gradient vector at that point. The Laplacian method outputs the point position in the same way, but no gradient vector is available. There is only a scalar “magnitude” number based on the standard deviation of intensity in the neighbourhood of this pixel. Thus, the output format from Phase 1 has to distinguish between the scalar and vector forms of additional data.

Phase 2, the matching software, also needs to be aware of whether it is receiving vector or scalar extra data. Most important, the algorithm that supervises the iterative multiscale matching must know whether the data is scalar or vector. (See Section 6.3 for more details.)

4.3.2 Evaluation

It is interesting to compare what the two edge finding algorithms base their decisions on. The wavelet-based algorithm calculates gradient, a first derivative measure, then determines the modulus of that gradient. The local maximum value of the modulus is used to identify the location of edges, while a threshold applied to the modulus is used to (optionally) eliminate less significant edges. The Laplacian-based method calculates the standard deviation of intensity in a small window, which is a scalar value related to the first derivative of intensity. This value is also used to eliminate less significant edges. But the location of the edges is determined using the Laplacian, a second-derivative measure. Because of the “false edges” problem of the Laplacian, the thresholding based on the standard deviation of intensity is mandatory, not optional.

Because the outputs of the two edge finders look so similar, you might expect them to work equally well. This is not the case. In experiments we have done, Phase 2 has always run faster when working from the output of the wavelet edge finder than from the Laplacian method’s output. We believe the major difference is the vector gradient information in the wavelet method output. Images often have multiple edges near each other, particularly in the lower levels of the output. If you examine the gradient vector orientation in

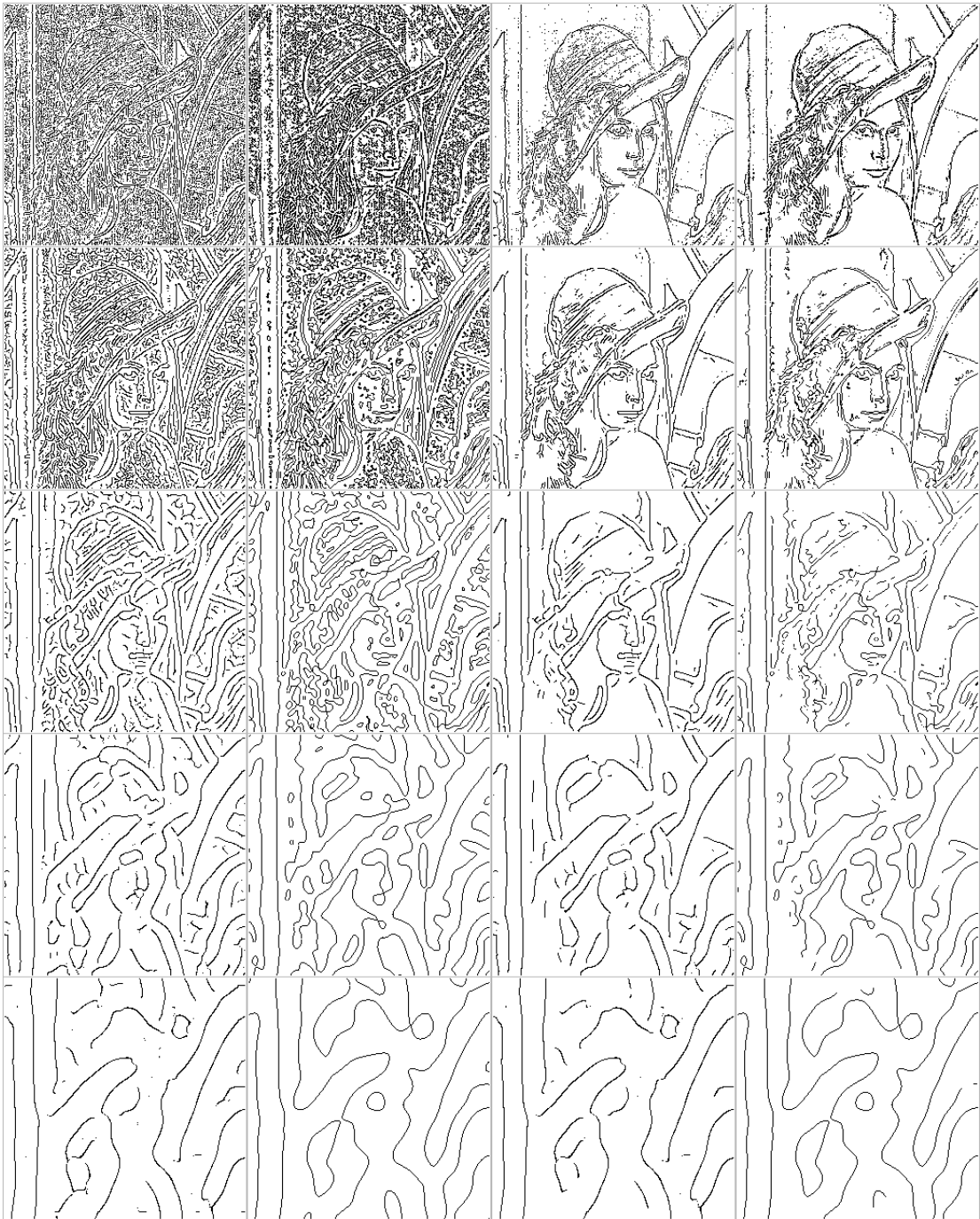


Figure 4.7: The effect of thresholding

the output data, you will often find that nearby parallel edges have gradient vectors that are oriented almost 180 degrees apart. When the matching phase builds its graph, it will assign a lower weight to any edges representing a match between two such points, since the matching is unlikely to be a good one. Thus, many potential matches are rejected outright or downgraded because of the vector data, leaving fewer potential matches for the algorithm to consider. In contrast, the scalar “magnitude” provided by the Laplacian method cannot distinguish between the two nearby edges, leaving more possible choices for the matching algorithm.

For example, we tried a test case involving two copies of the mandrill image, one scaled, rotated, and translated slightly from the other. Both were 250 pixels square. Matching using the wavelet method output took about 135 seconds, while the Laplacian output took about 375 seconds — three times as long. (The Laplacian edge-finder itself is slightly faster than the wavelet version, by less than a second in this example). For level 1 of the matching alone, with 5001 edge points in the data (limited by the maximum point cutoff), the graph built from the wavelet edge data had only 73% as many edges as the Laplacian-based graph.

In addition, visual examination of the matches found by Phase 2 shows that there is a larger fraction of incorrect matches when the input data comes from the Laplacian version of Phase 1. In at least two test cases, the Phase 3 fitting process converged to the correct transformation matrix when the pipeline began with the wavelet edge finder, while Phase 3 could not find the right transformation when the Laplacian edge finder was used. The input images were identical, and all other test conditions (thresholds, etc.) were as similar as we could make them.

While these are anecdotal results, it is significant that we have never seen the overall system produce better alignment when it uses the Laplacian edge detector, while we have several times seen markedly worse results with the Laplacian detector. In addition, in our tests the matching process has always taken considerably longer when starting from Laplacian-based edges.

Despite the comments above, the Laplacian-based edge finder does work reasonably well. With suitable test images, the system as a whole finds the appropriate transformation and brings the images into alignment.

However, the wavelet-based method works either equally well or better on every test that we’ve done, so it is what we normally use.

4.4 Mallat-Zhong Wavelet Transform Details

In this section, we will attempt to explain how the Mallat-Zhong discrete wavelet transform is designed. This section is not essential to understanding this chapter of the thesis, and could be skipped on a first reading. On the other hand, anyone who is planning to read or use the C language code for our test implementation should read at least this section and probably the entire Mallat-Zhong paper.

We will not attempt to present the level of detail found in Mallat and Zhong’s paper [41]. Instead, this section is intended as a tutorial introduction to the part of the Mallat-Zhong paper that defines the wavelet transform. There is really no information here that cannot also be found in the paper, but it is spread between two places (Section III and Appendices A-D) in the paper and the notation can be difficult to figure out. We have used the same notation as Mallat and Zhong whenever possible, so this section can be read alongside their paper without conflict.

Much of this same material also appears in Mallat’s book [40], in sections 5.5 and 6.3. However, the scaling of the filters in the book is different from the ones in the Mallat-Zhong paper, and some of the notation is slightly different. We have used the filters from the 1992 paper in our algorithm, so in this section we will aim for consistency with the paper, not the book.

We will begin with a few definitions. Then we will discuss the one-dimensional continuous version of the wavelet transform, then show how it extends to two dimensions, how it can be restricted to power-of-two scaling factors, and then how the discrete version of it is built. Then the fast wavelet transform (Mallat-Zhong's version) will be described.

4.4.1 Definitions

Many wavelet transforms are defined in terms of the convolution operator. Continuous-domain convolution in 1D and 2D are defined by

$$f * g(x) = \int_{-\infty}^{\infty} f(u)g(x-u) du$$

$$f * g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(u, v)g(x-u, y-v) dudv$$

A 2D function $g(x, y)$ is said to be separable if there exist two 1D functions g_1 and g_2 such that $g(x, y) = g_1(x)g_2(y)$ for all x and y . If $g(x, y)$ is separable, then a 2D convolution involving g can be separated into two successive 1D convolutions with g_1 and g_2 :

$$f * g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(u, v)g_1(x-u)g_2(y-v) dudv$$

$$= \int_{-\infty}^{\infty} \left[\int_{-\infty}^{\infty} f(u, v)g_1(x-u) du \right] g_2(y-v) dv$$

The inner integral takes a horizontal “slice” from $f(u, v)$ (by making v constant while u is the variable of integration) and convolves it with g_1 . Then the result, which is a 1D function of v , is convolved with g_2 . Mallat and Zhong use the notation $f * (g_1, g_2)$ for this separable convolution operation.

The discrete versions of these convolution operators are:

$$f * g(x) = \sum_{i=-\infty}^{\infty} f(i)g(x-i)$$

$$f * g(x, y) = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} f(i, j)g(x-i, y-j)$$

$$f * (g_1, g_2)(x, y) = \sum_{j=-\infty}^{\infty} \left[\sum_{i=-\infty}^{\infty} f(i, j)g_1(x-i) \right] g_2(y-j)$$

Although the sums above are shown with infinite limits, generally discrete signals are finite in extent, with an assumed value of zero everywhere outside some finite domain. Thus, we need only calculate the sum for values of i (and j , in 2D) for which all the functions have parameter values that are within their domains. It is easy to calculate the limits of the resulting finite summations.

In the 1D case, if the length of the sequences f and g are M and N , with $M \gg N$, then the cost of computing $f * g(x)$ for all useful values of x is $O(N(M+N))$. In the 2D case, if the two images are $M \times M$ and $N \times N$ in size, the total cost is $O(N^2(M+N)^2)$. The cost of the separable case is

$O(MN(M + N) + N(M + N)^2)$. In practice, this is about a factor of $N/2$ cheaper than the general 2D convolution.

Mallat and Zhong denote the Fourier transform of $f(x)$ by $\hat{f}(\omega)$ and define it and its inverse by

$$\begin{aligned}\hat{f}(\omega) &= \int_{-\infty}^{\infty} f(x)e^{-i\omega x} dx \\ f(x) &= \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega)e^{i\omega x} d\omega\end{aligned}$$

In 2D these become

$$\begin{aligned}\hat{f}(\omega_x, \omega_y) &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y)e^{-i(\omega_x x + \omega_y y)} dx dy \\ f(x, y) &= \frac{1}{(2\pi)^2} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \hat{f}(\omega_x, \omega_y)e^{i(\omega_x x + \omega_y y)} d\omega_x d\omega_y\end{aligned}$$

(Note that there are many possible consistent definitions of the Fourier transform and its inverse. In this pair, x and y are spatial positions in pixels, mm, or some other unit of distance. The frequencies $(\omega, \omega_x, \omega_y)$ are in radians per pixel, radians per mm, or radians per unit distance (depending on the interpretation of x). If you let $f = \omega/(2\pi)$, f is in more conventional spatial frequency units of cycles per pixel, cycles per mm, or cycles per unit distance. This definition of the Fourier transform is different from the one used in Section 2.2.2, which expresses frequency in cycles/pixel directly. In this section we are consistent with Mallat and Zhong.)

4.4.2 One Dimensional Continuous Case

In the continuous version of the M-Z wavelet transform, the smoothing function $\theta(x)$ is a cubic spline function that is approximately Gaussian in shape. The wavelet function $\psi^a(x)$ is the derivative of $\theta(x)$.

$$\psi^a(x) = \frac{d\theta(x)}{dx}$$

(The superscript a merely identifies this as the first derivative. The paper also defines a $\psi^b(x)$ that is the second derivative of $\theta(x)$.) Thus, $\psi^a(x)$ is a quadratic spline that is shaped much like the derivative of a Gaussian. Both functions are zero everywhere outside the interval $x = [-1, 1]$, so it is efficient to convolve the signal (image) with them. Figure 4.8 below shows the specific $\theta(x)$ and $\psi^a(x)$ used in the paper.

The (continuous) wavelet transform is computed by convolution with dilated copies of $\psi^a(x)$. Mallat and Zhong use a subscript to indicate dilation by a scaling factor s . For any function $\xi(x)$

$$\xi_s(x) = \frac{1}{s}\xi\left(\frac{x}{s}\right)$$

Thus, the wavelet transform of $f(x)$ at the scale s and position x is defined by

$$W_s^a f(x) = f * \psi_s^a(x) = \int_{-\infty}^{\infty} f(u)\psi_s^a(x - u) du$$

where $*$ indicates convolution.

But note that

$$\frac{d\theta_s(x)}{dx} = \frac{d}{dx} \left(\frac{1}{s} \theta \left(\frac{x}{s} \right) \right) = \frac{1}{s^2} \frac{d\theta \left(\frac{x}{s} \right)}{dx} = \frac{1}{s} \psi_s^a(x)$$

and so

$$W_s^a f(x) = f * \left(s \frac{d\theta_s(x)}{dx} \right) (x) = s \frac{d}{dx} (f * \theta_s)(x)$$

In other words, $W_s^a f(x)$ is the derivative of the signal smoothed by convolution with the smoothing function $\theta(x)$ dilated by the scale factor s . Since $\theta(x)$ was designed to approximate a Gaussian function in shape, the wavelet transform is in fact an approximation to the “derivative of Gaussian” edge detector applied to the signal. With such an edge detector, an edge is indicated by a local maxima in the absolute value of the detector’s output. (This is also similar to a Canny [6] edge detector, particularly for larger values of its x_m parameter.)

4.4.3 Two Dimensional Continuous Case

With a two-dimensional input signal (image), the wavelet transform becomes two-dimensional. There is now a two-dimensional smoothing function $\theta(x, y)$, and the smoothing operation becomes a 2D convolution. However, the derivative of the smoothing function is now a gradient, which is a 2D vector function, not a scalar function. To deal with this, we define two distinct wavelet functions that are the partial derivatives of $\theta(x, y)$ in the X and Y directions:

$$\psi^1(x, y) = \frac{\partial \theta(x, y)}{\partial x} \quad \text{and} \quad \psi^2(x, y) = \frac{\partial \theta(x, y)}{\partial y}$$

(Note that the superscripts 1 and 2 are just superscripts indicating X- and Y-direction partial derivatives. They are not exponents.)

The smoothing and wavelet functions can be scaled simultaneously in X and Y by a factor s :

$$\xi_s(x, y) = \frac{1}{s^2} \xi \left(\frac{x}{s}, \frac{y}{s} \right)$$

Then for any given scaling factor s , the wavelet transform has two (scalar) components:

$$\begin{aligned} W_s^1 f(x, y) &= f * \psi_s^1(x, y) \\ W_s^2 f(x, y) &= f * \psi_s^2(x, y) \end{aligned}$$

In the same way as for the one-dimensional case,

$$\begin{aligned} W_s^1 f(x, y) &= f * \left(s \frac{\partial \theta_s(x, y)}{\partial x} \right) (x, y) = s \frac{\partial}{\partial x} (f * \theta_s)(x, y) \\ W_s^2 f(x, y) &= f * \left(s \frac{\partial \theta_s(x, y)}{\partial y} \right) (x, y) = s \frac{\partial}{\partial y} (f * \theta_s)(x, y) \end{aligned}$$

Then these two scalar wavelet transform components define a vector with the property

$$\begin{bmatrix} W_s^1 f(x, y) \\ W_s^2 f(x, y) \end{bmatrix} = s \begin{bmatrix} \frac{\partial}{\partial x} (f * \theta_s)(x, y) \\ \frac{\partial}{\partial y} (f * \theta_s)(x, y) \end{bmatrix} = s \vec{\nabla} (f * \theta_s)(x, y)$$

In other words, the wavelet transform components are the X- and Y-components of the gradient of the image smoothed by $\theta_s(x, y)$.

The gradient vector points in the direction for which the directional derivative at this point in the image has the largest magnitude. The length (modulus) of the gradient vector is proportional to the absolute value of the directional derivative at this point.

4.4.4 Dyadic Wavelet Transform

For efficiency in computation, Mallat and Zhong restrict the values of the scaling factor s to integer powers of 2. In Section IIIA of their paper [41], they demonstrate that no information is lost by doing this — that the original (continuous) signal can be exactly reconstructed from (continuous) wavelet transforms calculated at only these values of s . In this form, there is still an infinite number of wavelet transforms involved; $2^{-\infty} \leq s \leq 2^{\infty}$. The authors call this the dyadic wavelet transform.

The dyadic wavelet transform can be inverted by convolving each wavelet transform with a suitably scaled reconstruction wavelet function $\chi(x)$ and summing:

$$f(x) = \sum_{j=-\infty}^{\infty} W_{2^j} f * \chi_{2^j}(x)$$

The reconstruction wavelet $\chi(x)$ is any function whose Fourier transform satisfies:

$$\sum_{j=-\infty}^{\infty} \hat{\psi}(2^j \omega) \hat{\chi}(2^j \omega) = 1$$

Thus, it is intimately related to $\psi(x)$ and $\theta(x)$.

A similar construction is used for the 2D case. $\theta(x)$ becomes a 2D smoothing function $\theta(x, y)$, and we end up with two wavelets $\psi^1(x, y)$ and $\psi^2(x, y)$ and two reconstruction wavelets $\chi^1(x, y)$ and $\chi^2(x, y)$ that operate in orthogonal directions.

4.4.5 Discrete Dyadic Wavelet Transform — One Dimension

To apply the M-Z wavelet transform to images, we need a discrete version of the transform. Mallat and Zhong describe how to construct one such discrete transform. The mathematical underpinnings are in Section IIIB of their paper, while the details of the construction of the particular wavelets chosen are in Appendix A of the paper. We will not repeat the entire construction here, but will try to give the flavour of how it works and some of the implications of this design.

The 1D construction starts out defining the function $\phi(x)$, a smoothing function that replaces $\theta(x)$ in the discrete case. A series of special constraints are placed on the Fourier transforms of $\phi(x)$, $\psi(x)$, and $\chi(x)$. These functions can be interpreted as discrete FIR filters that are applied to the sampled data. The constraints are described in terms of constraints on functions $H(\omega)$, $G(\omega)$, and $K(\omega)$, which can be regarded as the transfer functions of the filters $\phi(x)$, $\psi(x)$, and $\chi(x)$.

Then Mallat and Zhong introduce a family of functions that satisfy all of the necessary constraints on $H(\omega)$, $G(\omega)$, and $K(\omega)$, and select a particular set of three members of that family. They are:

$$\begin{aligned} H(\omega) &= e^{i\omega/2} (\cos(\omega/2))^3 \\ G(\omega) &= 4ie^{i\omega/2} \sin(\omega/2) \\ K(\omega) &= \frac{1 - |H(\omega)|^2}{G(\omega)} \end{aligned}$$

Performing inverse Fourier transforms on these three functions yields the impulse response of three filters H, G, and K. Because we are now working in discrete space, these are discrete filters. Due to the choice of the transfer functions, the filters are finite and in fact quite compactly supported. The non-zero coefficients of these filters are:

$$\begin{aligned} H &= \frac{1}{8} [1 \quad 3 \quad 3 \quad 1] \\ G &= [2 \quad -2] \\ K &= \frac{1}{128} [-1 \quad -7 \quad -22 \quad 22 \quad 7 \quad 1] \end{aligned}$$

(Note that the order of coefficients above is reversed from that shown in Table 1 of the Mallat-Zhong paper [41]. The coefficients above are actual impulse responses that will be convolved with the input data. The coefficients in Table 1 in the paper have been reversed in sequence so their order matches that of the input data values, without the negative indexing used in convolution.)

(Also, note that there is a typographic error in two of the values in Table 1 of [41]: 0.054685 should be 0.0546875. It is clear that something is wrong because the composition of the forward and inverse wavelet transform, using the coefficient set in the paper, is not quite the identity transformation, so we don't quite get the original image back. With the error corrected as described above, the forward and inverse wavelet transforms are truly inverses of each other (if performed with exact rational arithmetic). Locating the bad coefficient is easy; all of the other coefficients are integer multiples of 1/128.)

The continuous-domain functions $\phi(x)$, $\psi(x)$, and $\theta(x)$ are obtained by first deriving their Fourier transforms:

$$\begin{aligned} \hat{\phi}(\omega) &= \left(\frac{\sin(\omega/2)}{\omega/2} \right)^3 \\ \hat{\psi}(\omega) &= i\omega \left(\frac{\sin(\omega/4)}{\omega/4} \right)^4 \\ \hat{\theta}(\omega) &= \left(\frac{\sin(\omega/4)}{\omega/4} \right)^4 \end{aligned}$$

Then we take the inverse Fourier transform to obtain these piecewise quadratic or cubic splines:

$$\begin{aligned} \theta(x) &= \left. \begin{array}{ll} 0 & x \leq -1 \\ \frac{8}{3}x^3 + 8x^2 + 8x + \frac{8}{3} & -1 < x \leq \frac{-1}{2} \\ -8x^3 - 8x^2 + \frac{4}{3} & \frac{-1}{2} < x \leq 0 \\ 8x^3 - 8x^2 + \frac{4}{3} & 0 < x \leq \frac{1}{2} \\ -\frac{8}{3}x^3 + 8x^2 - 8x + \frac{8}{3} & \frac{1}{2} < x \leq 1 \\ 0 & 1 < x \end{array} \right\} \\ \psi(x) &= \left. \begin{array}{ll} 0 & x \leq -1 \\ 8x^2 + 16x + 8 & -1 < x \leq \frac{-1}{2} \\ -24x^2 - 16x & \frac{-1}{2} < x \leq 0 \\ 24x^2 - 16x & 0 < x \leq \frac{1}{2} \\ -8x^2 + 16x - 8 & \frac{1}{2} < x \leq 1 \\ 0 & 1 < x \end{array} \right\} \end{aligned}$$

$$\phi(x) = \left\{ \begin{array}{ll} 0 & x \leq \frac{-3}{2} \\ \frac{1}{2}x^2 + \frac{3}{2}x + \frac{9}{8} & \frac{-3}{2} < x \leq \frac{-1}{2} \\ -x^2 + \frac{3}{4} & \frac{-1}{2} < x \leq \frac{1}{2} \\ \frac{1}{2}x^2 - \frac{3}{2}x + \frac{9}{8} & \frac{1}{2} < x \leq \frac{3}{2} \\ 0 & \frac{3}{2} < x \end{array} \right\}$$

Figure 4.8 shows graphs of these functions. Two of them can also be seen in Figure 1 of the Mallat-Zhong paper.

Note: there is a factor of four difference in the magnitude of $\psi(x)$ between Figure 4.8 here and Figure 1 in the paper. We believe the one in the paper is simply wrong. The function $\psi(x)$ is supposed to be the derivative of $\theta(x)$, and that is true of the $\psi(x)$ defined and shown here, but not the one graphed in Figure 1 of the paper. The paper never explicitly shows the formula for $\psi(x)$, only its Fourier transform $\hat{\psi}(\omega)$. The inverse Fourier transform of $\hat{\psi}(\omega)$ is the spline $\psi(x)$ shown above, which is graphed correctly in Figure 4.8. In addition, $\psi(-0.5) = 2$ and $\psi(0.5) = -2$, which matches the coefficients of the G filter that is derived from $\psi(x)$.

In Mallat's 1998 book, he reformulates this wavelet transform in a way that makes $\psi(x)$ 1/4 as large as it is in the 1992 paper, as well as being reflected in the Y axis. See equation 5.83 and Figure 5.6 in the book for this altered definition. It is our guess that when the authors were writing the 1992 paper, they were already changing this particular wavelet transform, and inadvertently included a graph of the "new" $\psi(x)$ that had been scaled but not reflected yet.

In any case, it doesn't really matter for our purposes. The G filter is applied just once to produce each level of the wavelet transform, so the effect of scaling $\psi(x)$ is to multiply the entire wavelet transform by a constant. Our algorithms primarily depend on the relative size of the coefficients; their absolute size is nearly irrelevant. The only change is that the default value of the minimum-magnitude threshold in the code would need to be scaled.

The 1D fast wavelet transform algorithm is described in Appendix B of the Mallat-Zhong paper. We begin with our original signal, and call it the "smoothed" signal at level 1: $S_1^d f(x)$. (The superscript "d" simply means "discrete". The subscript is the smoothing scale of the signal.) Then for additional scales 2^{j+1} , $j \geq 0$ we calculate the wavelet transform $W_{2^{j+1}}^d f$ using the filter G_j and the new smoothed signal $S_{2^{j+1}}^d f$ using the filter H_j . The filters G_j and H_j are simply G and H widened by inserting $2^j - 1$ zeroes between each non-zero coefficient. The important steps are:

$$\begin{aligned} W_{2^{j+1}}^d f &= S_{2^j}^d f * G_j \\ S_{2^{j+1}}^d f &= S_{2^j}^d f * H_j \end{aligned}$$

The algorithm is repeated for $J = \lceil \log_2(N) \rceil$ steps. In principle, the iteration can be continued forever, but if N (the number of points in the input data) is a power of 2, the smoothed signal $S_{2^j}^d f$ becomes constant when $j \geq J$. If N is not a power of 2, this is no longer precisely true, but the smoothed signal becomes almost devoid of information anyway, and we might as well stop.

The algorithm in the paper also scales the wavelet amplitude by a factor $1/\lambda_j$, but we do not need to do that because we don't compare wavelet amplitudes between scales.

Note that the filter that gives the smoothed signal $S_{2^{j+1}}^d f$ applied to the input data values is not merely the filter H_j , it is $H_j * H_{j-1} * H_{j-2} * \dots * H_0$. In effect, the smoothing filter at each level is defined by the iterative convolution of the H filter at that scale with the H filter at all smaller scales. Similarly, the filter that gives the wavelet transform signal $W_{2^{j+1}}^d f$ when applied to the input data values is not merely G_j , it is

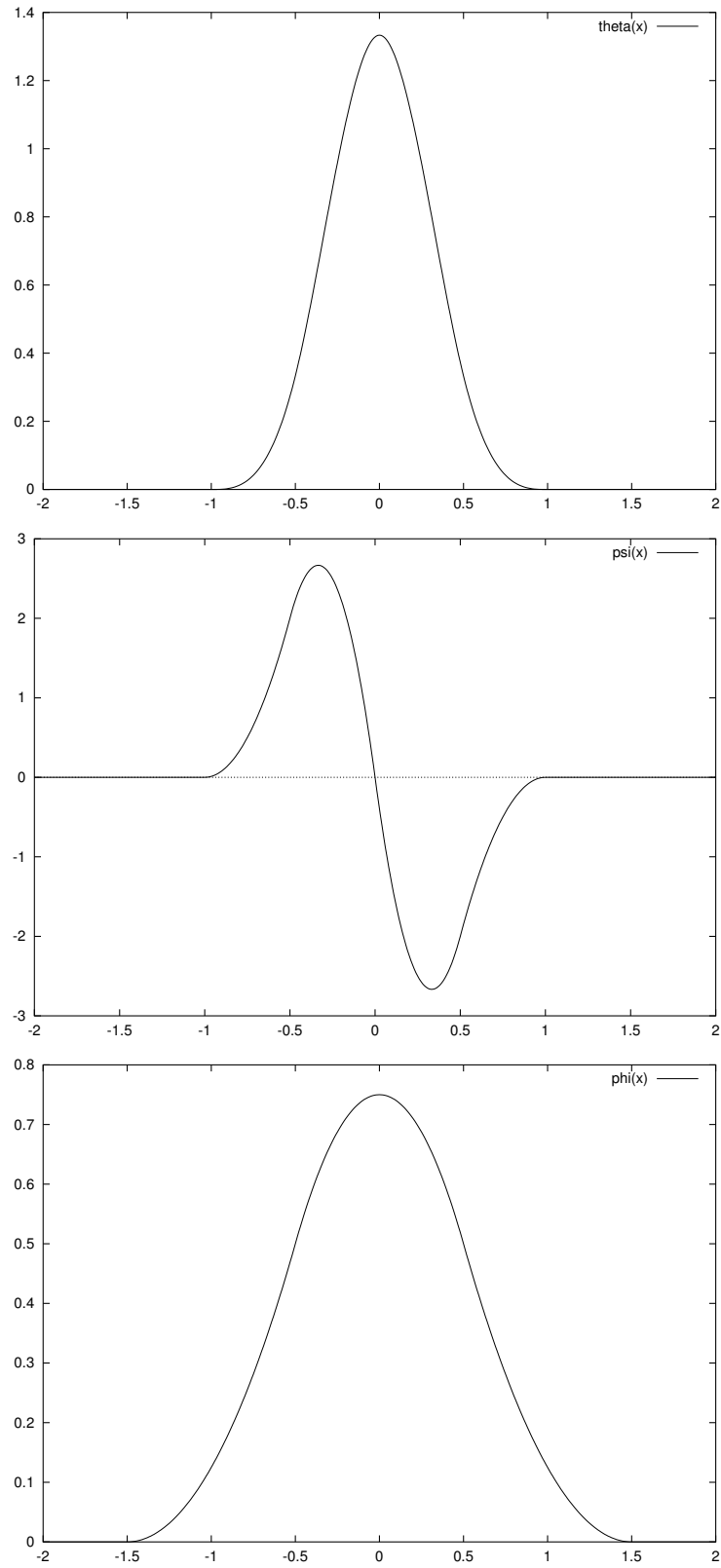


Figure 4.8: $\theta(x)$, $\psi(x)$, $\phi(x)$

the convolution of G at the current scale with H at all smaller scales: $G_j * H_{j-1} * H_{j-2} * \dots * H_0$. So the wavelet functions themselves are defined iteratively.

4.4.6 Discrete Dyadic Wavelet Transform — Two Dimensions

The two-dimensional version of the fast wavelet transform is built in a similar way, though there are some additional complications. The two wavelets $\psi^1(x, y)$ and $\psi^2(x, y)$ are separable functions defined as the product of a 1D wavelet and a (scaled) 1D smoothing function. The reconstruction wavelets $\chi^1(x, y)$ and $\chi^2(x, y)$ have transfer functions specified by two functions $K(\omega)$ and $L(\omega)$, instead of just $K(\omega)$. (More details are in Appendix C of the paper.)

We end up obtaining the same H , G , and K filters as in the one-dimensional case, but a new L filter appears:

$$L = \frac{1}{128} [1 \quad 6 \quad 15 \quad 84 \quad 15 \quad 6 \quad 1]$$

In the 2D fast wavelet transform algorithm, the two components of the wavelet transform are generated similarly: the X-component is produced by applying the G filter to the rows of the previous-level smoothed image, while the Y-component is produced by applying the G filter to its columns. The new smoothed image results from applying the H filter to both rows and columns. The important steps in the algorithm are:

$$\begin{aligned} W_{2^{j+1}}^{1,d} f &= S_{2^j}^d f * (G_j, D) \\ W_{2^{j+1}}^{2,d} f &= S_{2^j}^d f * (D, G_j) \\ S_{2^{j+1}}^d f &= S_{2^j}^d f * (H_j, H_j) \end{aligned}$$

Recall that $A * (G, H)$ denotes the separable convolution of the rows and columns (respectively) of the 2D image A with the 1D filters G and H . The filter D is the Dirac delta function, which is the identity filter. (I.e. convolving with D does not change the data, so we may skip that particular convolution step.)

Again, the smoothing filter and wavelet filter that act to generate the two wavelet images and the smoothed image at each level of the transform are really defined by the iterative convolution of filters at all scales between the current level and level 0.

Table 4.1 shows the effective smoothing filters and wavelet filters used at each scale of the M-Z wavelet transform when applied to a 31×31 pixel image. Note that the vertical scale factor is set for each graph individually, so the vertical scales are not comparable, particularly from one transform level to the next.

4.5 Implementation

There is a single executable program named “maxima” that performs the Phase 1 computations using the wavelet transform. A similar program named “lmaxima” uses the Laplacian method instead. Both of these are built from two major source files, one which performs the actual transforms, and one that does higher-level housekeeping.

Rather than performing the operations in the obvious order (do transform, find edges, write them out), both programs interleave these three operations to perform them one level at a time. This saves a substantial amount of memory.

This phase is fairly fast, in both implementations. Finding maxima at all levels of the 512-wide mandrill image takes about 2.3 seconds on a Pentium III 700 MHz machine. The Laplacian version takes about 1.8 seconds.

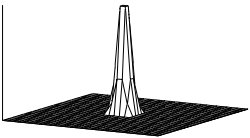
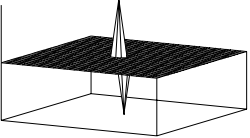
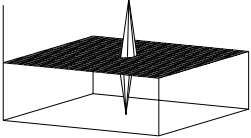
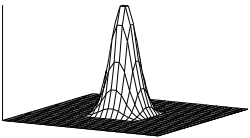
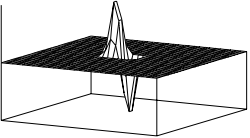
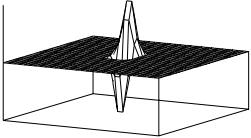
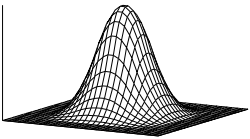
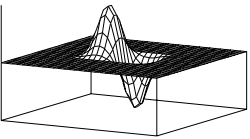
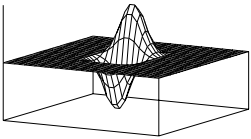
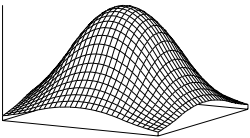
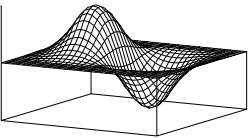
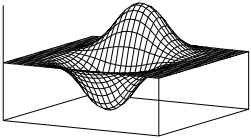
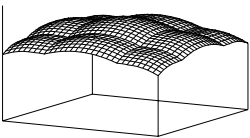
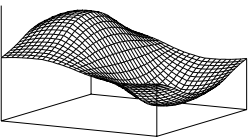
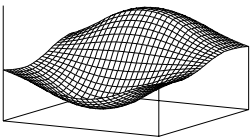
Level	Smoothing	X-wavelet	Y-wavelet
1			
2			
3			
4			
5			

Table 4.1: Mallat-Zhong wavelet transform filters

4.6 Discussion

Ultimately, this phase of the process needs to find edges at multiple scales, and any combination of algorithms that does so could be used. We tried doing so with the Laplacian method. However, it appears that the gradient vector data produced by the wavelet transform is more valuable during the matching process than we first appreciated.

Figure 4.4 above shows how well the features that are found match between images that are a factor of four different in size (a factor of 16 difference in the number of pixels). We expect this to be true for any two images when the ratio of their sizes is exactly a power of two.

However, when the ratio is not a power of two, the match will not be so good — the levels of detail found in one of the images will be somewhere between the levels illustrated in the figures. We might get better matching of features if we ensured that the two images were exactly a power of two different in size. This can be done without significant loss of information by enlarging the smaller of the two images by a factor that ranges between one and two. We have not tried this yet.

As explained in Section 4.2.4 above, we discard some of the maxima found if there are “too many” of them. It is important to limit the number of points per level to prevent the Phase 2 matching algorithm from taking far longer than necessary. And, after all, the matches are ultimately used to fit a global transformation with only six parameters. It is not necessary to have thousands of points as input to the Phase 3 fitting; a dozen well-chosen points would do. In fact, human-assisted image registration often starts with a relatively small number of hand-picked pairs of points.

However, we don’t have a human eye guiding the process. What we really want is a sufficient number of matched pairs of points well-distributed over the area of the image. If the image is small enough, we simply keep all of the maxima points found, but if there are too many, we need to prune them somehow.

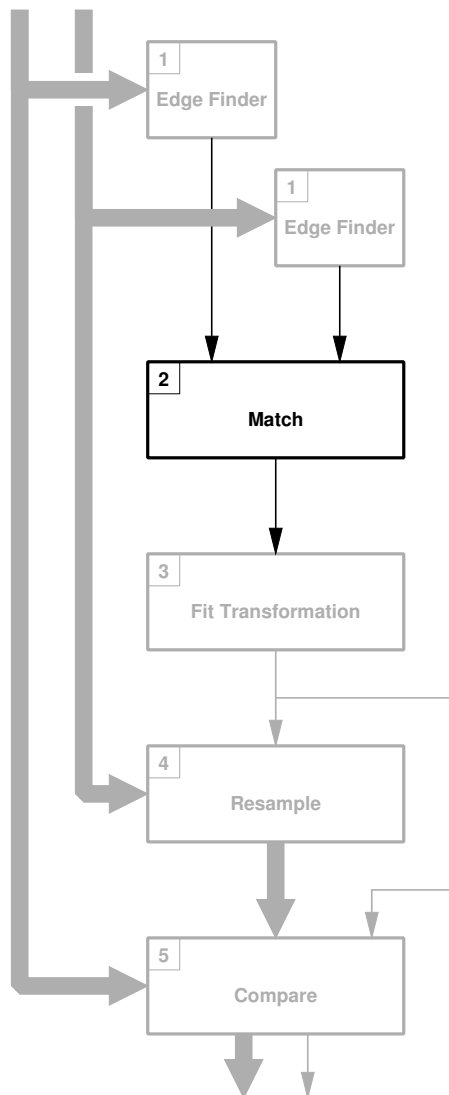
Keeping the maxima with the largest gradient magnitude doesn’t necessarily give us points from all of the area of the image — it is entirely possible that all of the points retained might be from a small region. On the other hand, the “keep the largest” algorithm does have a good chance of retaining approximately the same maxima from both images, which gives the matching phase a reasonable chance of being able to find good matches for most points.

We might also consider pruning the data by eliminating points at random. This would retain whatever spatial distribution of points that we started with. Or, we might consider selectively deleting points from denser areas first, using some sort of criterion involving minimum distance to nearest neighbour. This would tend to even out the density of the data. However, these methods would be used on two input images individually, with nothing to ensure that the remaining points were actually the same image features. Thus, this would reduce the odds of Phase 2 being able to find the best-matching pairs of points, and the output of Phase 2 would have many fewer exact matches. This is not good.

To do a better job of “thinning” the data points, we would need something that tends to preserve the same image features from both images, while being sensitive to local point density. One possibility would be to start with a binary image (0/1 only) showing where maxima were and were not found. Applying a low-pass filter to this would yield a blurred image where local intensity corresponds to local maxima density. This could then be scaled and used as a threshold in deciding which maxima to keep. In this scheme, the threshold would be high in dense areas and low in sparse areas, tending to keep some data points from all areas of the image, yet always keeping the most prominent maxima from any given region. This would be an interesting method to try in the future.

Chapter 5

Phase 2 — Feature Matching



The task of Phase 2 is to read in the list of significant features in two images (produced by applying Phase 1 to each of the images independently). Then it matches points in the two lists that seem most likely to correspond to each other (in the sense of possibly representing the same visual feature in both images).

The matching is done separately for each level of the wavelet transform, since each level extracts information from the original image at a different spatial scale. When the two images are of different sizes (in pixels), the wavelet transform levels are aligned so the levels being compared differ by a ratio of at most $\sqrt{2}$ in size.

5.1 Method

The central method used by this program comes from graph theory, where it is called “bipartite graph matching”. A general graph $G = (V, E)$ consists of a set V of vertices and a set E of edges. Each edge in E is incident to exactly two vertices in V . If a graph G is *bipartite*, then the set of vertices V can be partitioned into two sets V_1 and V_2 such that every edge in E has one endpoint in V_1 and the other endpoint in V_2 .

A *matching* M of G is a subset of the edges E such that no two edges share a common endpoint. The cardinality $|M|$ is the number of edges in M . A *maximum cardinality* matching is a matching whose cardinality is at least as large as all other possible matchings. In other words, it contains the maximum possible number of edges (and thus vertices) of G .

In a *weighted* graph, each edge has a real number called its *weight* or *cost* associated with it. The *weight* of a matching M is simply the sum of the weights associated with all of the edges in M . A *maximum weight* matching is a matching whose weight

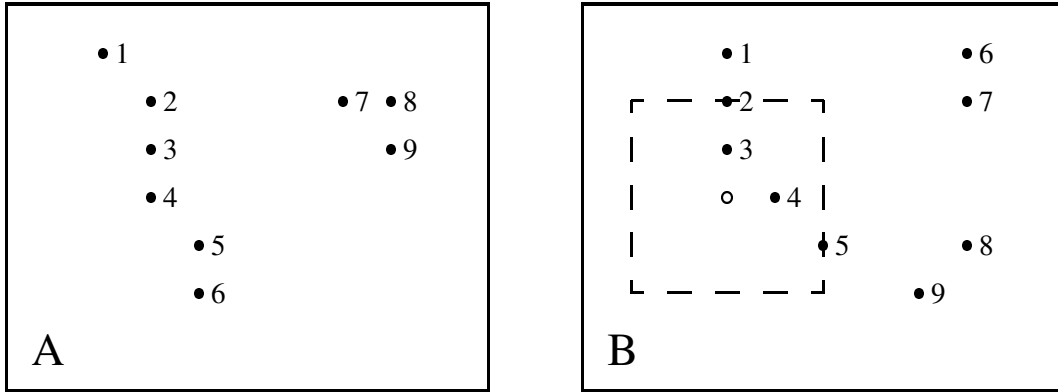


Figure 5.1: Edge points in two images

is at least as large as all other possible matchings. Note that a maximum weight matching is generally not a maximum cardinality matching, and vice versa.

To solve our particular problem of matching image features, we proceed as follows: We begin with a set of edge points passed to us from Phase 1 for each of the two original images. Figure 5.1 shows two tiny example images “A” and “B” each with a number of edge points. We create a graph containing one vertex for every one of these edge points. All of the points from image A form one partition of the graph, while all of the points from image B make up the other partition.

Then, for every edge point in one image, we scan through all of the edge points from the other image that might plausibly be where that particular image 1 point has moved to in image 2. In the most general case, this might be all of the edge points in image 2, though we usually confine our search to a box of a certain size around the location of the current point in image 1 on the assumption that the displacement of points between the two images is within some smaller bound. Figure 5.1 shows the bounding box for point A4: The open circle in image B is the location of point A4 in the coordinate system of image B. The dashed-line square is a bounding box that is plus or minus two pixels from this location. Thus, points B2, B3, B4, and B5 are all plausible matches for A4.

For every such plausible pair of points, we add an edge to the graph, with the endpoints of the edge being the vertices for these two points. Figure 5.2 shows the graph corresponding to the images in Figure 5.1. Note that each point in image A is connected to all of the points in image B that are “sufficiently close”, but no other points in B. Of course, this example is extremely simplified so you can see what is happening. In actual use, there are thousands of points in each image. The bounding box is typically 20% of the image width in size, not just four pixels, and each node in one partition is connected to several hundred nodes in the other partition.

The weight of each edge in the graph is a number calculated to reflect how likely the two points are to represent the same image feature, taking into account the distance between the two points, the difference in gradient magnitude, and the difference in gradient orientation. Smaller distance, more equal magnitude, and more equal orientation all yield a larger weight. (These weights are not shown in the figure.)

This graph is obviously bipartite by its construction. We apply a maximum weight bipartite matching algorithm to the graph we have built. The matching produced is, in some sense, a global optimum matching of features in one image with similar features in the other image (dependent of course on the weight function we used).

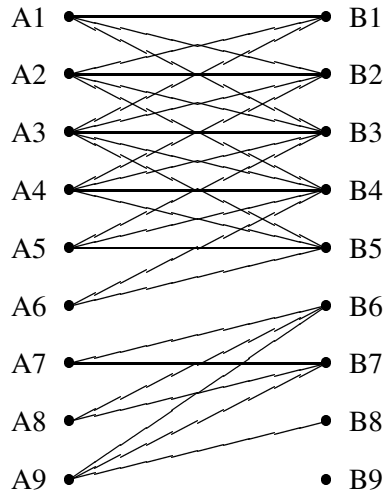


Figure 5.2: Bipartite graph from edge points

The edge detector output contains edges found at multiple levels of detail in the image. If we are using a “straight through” pipeline without iterative multiscale matching, the Phase 2 algorithm reads in all levels of data, but performs the graph matching one level at a time.

If we are making use of iterative multiscale matching (see Section 6.3), the multiscale controller software splits the edge data into individual levels, and prealigns it before matching. In this case, the matching program is called once for each level of data, and there is only one level of data present in its input file each time.

5.1.1 Input

The matching algorithm begins by reading in two data files that were produced by the Phase 1 edge-finding algorithm. For each level of detail, there is a list of all of the edge points that were deemed significant. Each such point is described by its (x,y) location in the image, plus the value of the gradient vector at that point.

While the points are being read, the (x,y) coordinates are normalized. If the image is $M \times N$ pixels in size, both X and Y coordinates are divided by a scale factor of $\max(M - 1, N - 1)$. This yields normalized X and Y coordinates that are always in the range $[0, 1]$ no matter what the original size of the image. This is important because we may be comparing two images with greatly different sizes (in pixels). As long as the two images have the same field of view, corresponding points in the two images will end up with very close normalized coordinates. Note that the same scale factor is used for both X and Y coordinates. This is important to preserve shapes.

The gradient vectors are read in Cartesian form, but the algorithm converts them to polar form for internal use. If the edge data file came from the Laplacian-based edge finder, there is a scalar value instead of the gradient vector. This is simply turned into a vector whose length is the scalar value and whose angle is zero. From this point forward there is no difference in the treatment of data from the two different edge finders.

5.1.2 Level Alignment

Once both input files have been read, the algorithm looks at the sizes of the two images to decide which levels from the wavelet transform should be compared. If both images are approximately the same size (in pixels), then equal levels will be compared: level 1 to level 1, level 2 to level 2, etc. But if the images are of substantially different sizes, level 1 from the smaller image is expected to contain approximately the same edges as level 2, 3, or higher from the larger image. (See Figure 4.4 for an example of how edges match across levels for images of different sizes.)

To do this, we assume that the horizontal field of view of the two images is identical. If the widths of the two images are M_1 and M_2 , we calculate an offset $O = \text{round}(\log_2(M_2/M_1))$. This tells us that level l from the first image should be matched against level $l + O$ in the second image. Done this way, the spatial scales being compared differ by a factor that is between $1/\sqrt{2}$ and $\sqrt{2}$.

Note: If we are using iterative multiscale matching, the level alignment is handled by the software that supervises the process, not the matching algorithm.

5.1.3 Building the Graph

Once we have identified the data from each image that will be matched on this pass, we build a bipartite graph that describes the edge points and their relationships.

We create a graph data structure, then add all of the edge points as nodes in the graph. All nodes that came from points in the left (respectively right) image are also entered into a separate left (right) node list. This is used to keep track of the bipartition of the nodes.

Then we add edges. In concept, we could build a complete bipartite graph, with every node on the one side of the graph connected to all nodes on the other side. However, even with the limitation on the number of points that can be output at one level from Phase 1, we may have 5000-10000 nodes in each half of the graph for larger input images. A complete graph with this many nodes has 25-100 million edges. This would consume hundreds of megabytes of memory, and also slow down the matching substantially. Thus, we want to reduce the number of edges as much as we can.

We do this by assuming that the two images are not too dissimilar. More precisely, we assume that the (normalized) coordinates of an image feature in one image are within some small fraction of the image width of the coordinates of the same feature within the other image. The default threshold is 10% of the image width or height, whichever is greater. This can be made larger (at the cost of slower matching) if the images are suspected of being more badly misaligned. Conversely, the threshold may be made very small if the images are known to be close to aligned already. In the case of iterative multiscale matching, this window starts out large for the first matches based on the high level (large structure) edges, and becomes smaller as the iteration proceeds on the assumption that the images are becoming better aligned.

Thus, for every edge point in the one image, we search the other image for points that are within our threshold distance. To keep the test cheap, we test against a square window whose width is twice the threshold, rather than a circle whose radius is the threshold.

For each point inside the window, we calculate a suitable edge weight. If a maximum point in the first image is located at the (normalized) coordinates (x_1, y_1) , and the gradient there has a magnitude of r_1 at an orientation of θ_1 degrees, then we calculate three “distance” values:

$$\begin{aligned} D &= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \\ R &= |\ln(r_2) - \ln(r_1)| = |\ln(r_2/r_1)| \end{aligned}$$

$$A = |(\theta_2 - \theta_1 + 180) \bmod 360 - 180|/180$$

D is just the Euclidean distance between the two points (in normalized coordinates), and is zero when the features are in exactly the same location relative to the boundaries of the images. R is the absolute value of the difference in the logarithms of the gradient magnitudes (or the logarithm of their ratio, which is equivalent). This is zero if the magnitudes are equal, and gives the same value for the same amount of relative change in magnitude. A is the angular difference between the two gradient orientations, taken in whichever direction gives the minimum angle, normalized to $[0, 1]$. (Perhaps this deserves more explanation: The difference between the two orientations is some arbitrary angle. That angle $\bmod 360$ gives us a canonical angle in $[0, 360]$. However, values greater than 180 degrees represent angles that are less than 180 degrees when measured in the other direction, and we want the smaller number. Adding and subtracting 180 changes the range of canonical angles to $[-180, 180]$. Then we take the absolute value and divide by 180 to normalize the result.)

Note that all of these values are symmetric — it doesn't matter which is image 1 and which is image 2. Then we calculate the edge weight as a weighted sum of these differences:

$$E = 1 - (w_d D + w_r R + w_a A)$$

The three terms on the right are all subtracted from the sum because the weight must decrease when D , R , or A increase. The three weighting factors w_d , w_r , and w_a determine the relative importance of the three distances in the overall edge weight. We have been using $w_d = 2.5$, $w_r = 0.75$, and $w_a = 0.75$. We based those choices on some thought about what would be a “poor” match in position, magnitude, or orientation, and set the weights so that a poor match in any one category is sufficient to reduce the overall edge weight to zero. We haven't experimented with different sets of weights, though that would obviously be an interesting thing to investigate.

At first, it might seem that you could multiply all three weights by a constant and get the same result from matching, since the relative importance of the three components of the weight would remain the same. However, the size of the three weights compared to the constant “1” in the equation is quite important. We normally find a maximum weight matching, and such a matching will never contain an edge whose weight is negative (since omitting it would increase the weight of the matching, and it is always possible to delete an edge from a matching). Knowing this, the graph creation algorithm immediately discards any potential edge whose weight would be negative if it was included. Thus, the sizes of w_d , w_r , and w_a effectively set a threshold for how large the distances D , R , and A can be before an edge is entirely eliminated from consideration in the matching because it is “too unlikely”.

Sufficiently small values of the weights will prune no edges at all. Larger weight values prune more of the “unlikely” edges, saving more space and time. Obviously, it is possible to go too far — we don't want to remove too many edges that would in fact be good match candidates. With the weight values listed above, our algorithm typically discards one third or fewer of the potential edges that have passed the bounding box test.

5.1.4 Matching

When construction of the graph is complete, we use a standard algorithm to perform the matching.

The following figures show some sample matching output. We started with a pair of 125-pixel-square mandrill images, one offset horizontally and vertically from the other. (This low resolution source image

allows individual vectors to be seen in the printed images.) Figures 4.3 and 4.4 show approximately the set of edges that were found in this source image. Then we matched the edge points from the two images. Figure 5.3 displays the results one level at a time.

In the diagrams, each matched pair of points is drawn as a vector. The vector extends from the location of the point in the first image to the location of the matched point in the second image. Match vectors are drawn with a colour that interpolates from red to green; red indicates the end belonging to image 1 and green indicates image 2. Thus, you can tell the direction of the vectors. The blue points are vectors whose start and end position is the same. These are mostly edge artifact features in this example, since all real image features are moved by a translation.

Note that the level 1 match output contains many incorrect matches, particularly in the areas containing the mandrill's head fur and beard. These areas in the level 1 output from Phase 1 are very noisy, with many small features crowded together and little coherent structure. It is not surprising that the matching algorithm makes many mistakes here. However, at higher levels, the image becomes less cluttered, containing fewer but better-defined features. The matching works very well, particularly in levels 3-6. Finally, at the highest two levels, there is little real information left in the images, and the features present are often distorted by edge effects.

Figures 5.4 and 5.5 shows more matching output. The first is from a pair of mandrill images where one was rotated five degrees. The second shows a 5% difference in image scale. Note a similar pattern in the matching output: the first level or two are noisy with many bad matches, the middle levels are all pretty good, and the upper two levels are poor again.

5.1.5 Alternative matching algorithms

In fact, we have tried three somewhat different matching algorithms: maximum cardinality, maximum weight, and maximum weight maximum cardinality.

One alternative is maximum-cardinality (MC) matching. This finds a matching with the maximum number of edges, without regard to the edge weights. Our usual choice is a maximum-weight (MW) matching algorithm. It simply finds a matching such that there is no other matching with a greater sum of edge weights. The result is generally not a maximum cardinality matching. Finally, we have tried maximum-weight maximum-cardinality (MWMC) matching. This is effectively a hybrid of the other two: it always finds a maximum-cardinality matching, but of all the possible MC matchings it selects one with maximum possible edge weight.

For "typical" photographic images like the mandrill, peppers, or Lenna, the MW matching method works best. By this, we mean that a higher percentage of the matched points that are passed on to Phase 3 are actually correct, regardless of the total number of points. The larger the proportion of correct matches, the more likely Phase 3 will be able to converge to the correct transformation. For one example, we started with a pair of 500 pixel square images, both derived from the same original. The difference between the two image involved a 5% scale change, a 5 degree rotation, and a small translation. We declare that a pair of points has been "correctly matched" if, after applying the known transformation being tested, the residual distance between the pair of points is one pixel spacing or less. MC matching produced a total of 30980 matches at all levels, of which only 1.7% were correct. MW matching yielded 30256 matches, with 26% correct. MWMC matching produced 30980 matches, with 20% correct.

Of course, no one would expect pure MC matching to perform well in this application, since it completely ignores the edge weights that carry information about how likely a match is to be correct. It is

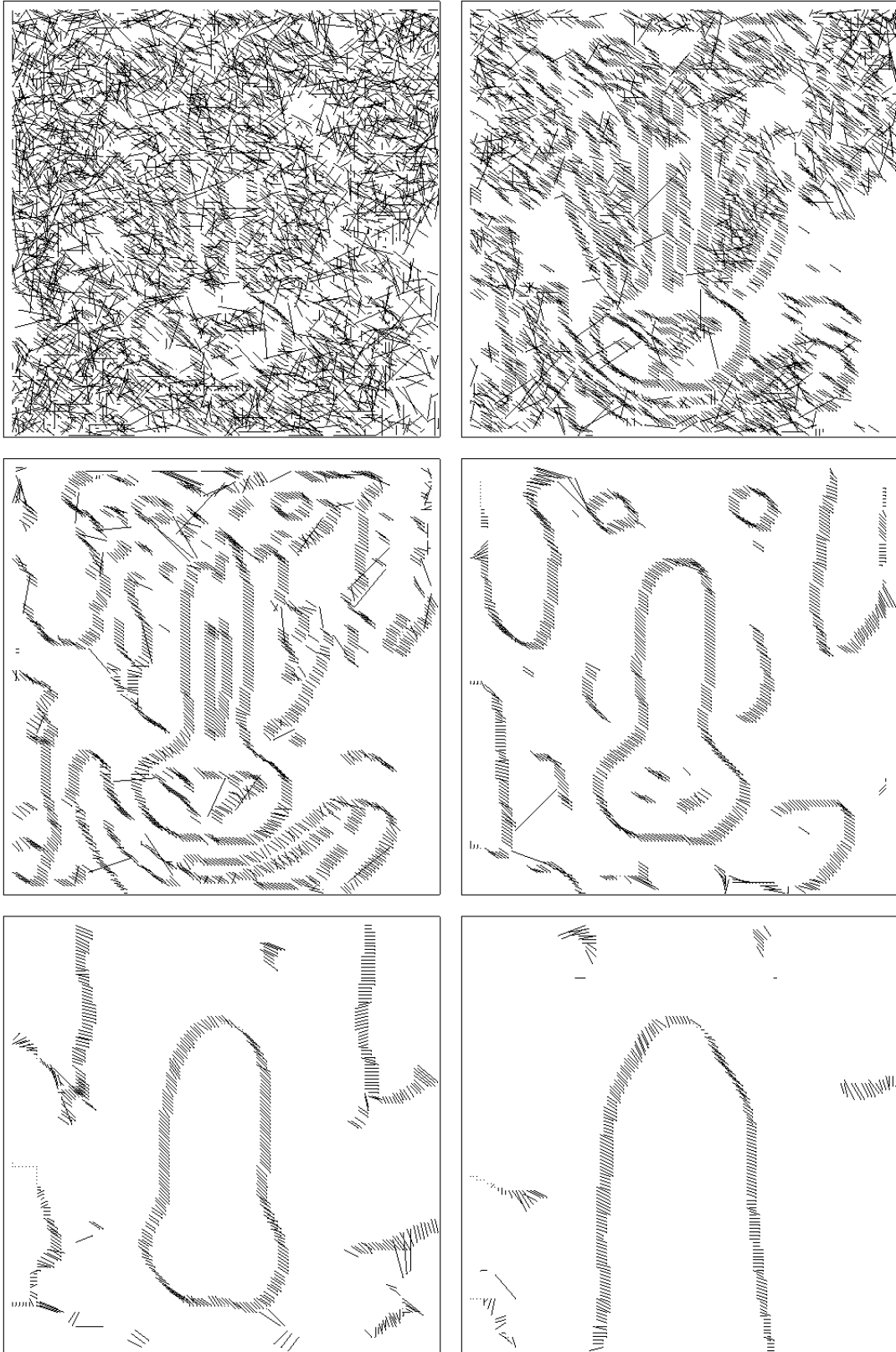


Figure 5.3: Matching output: translation, levels 1-6

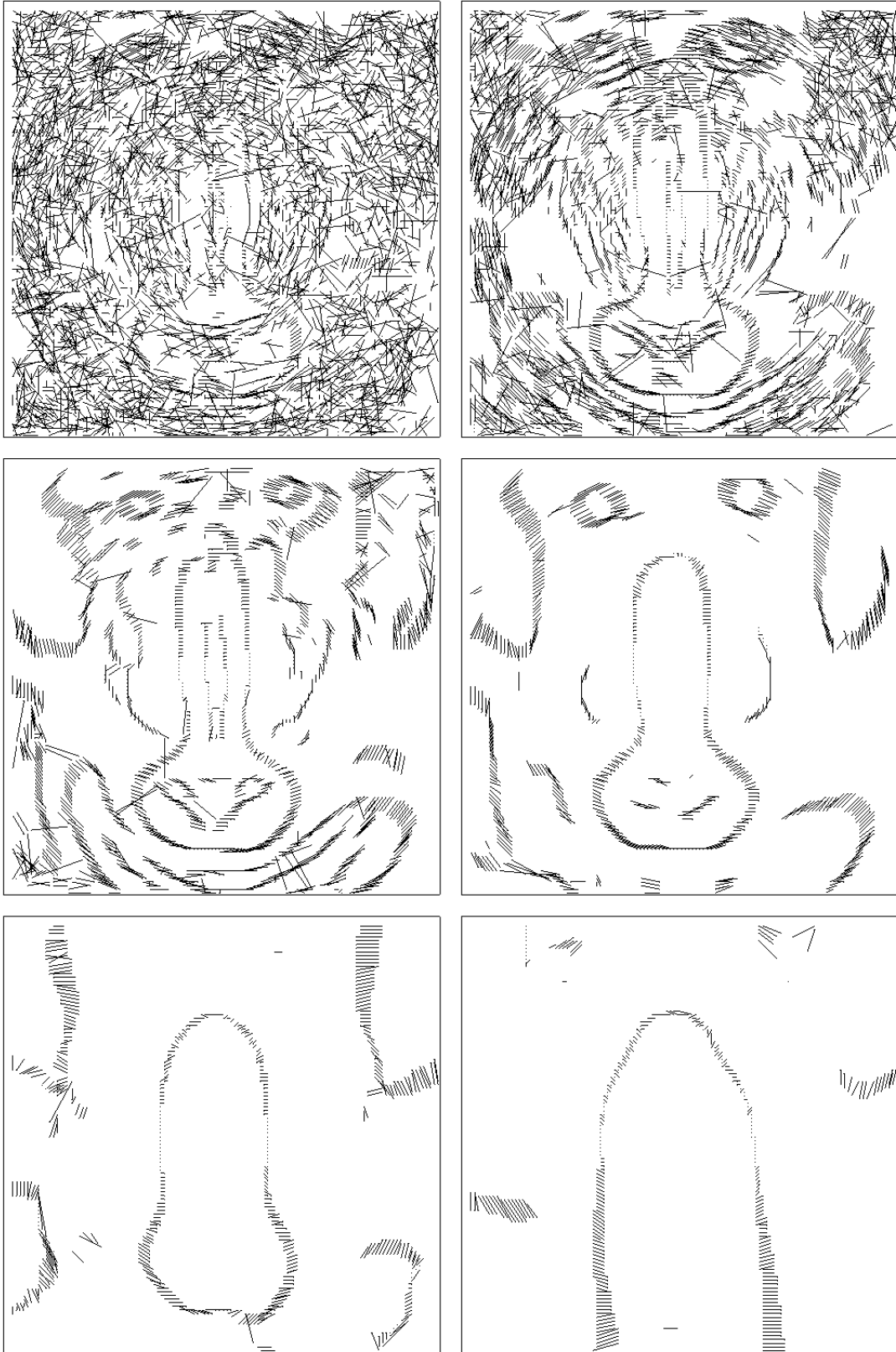


Figure 5.4: Matching output: rotation, levels 1-6

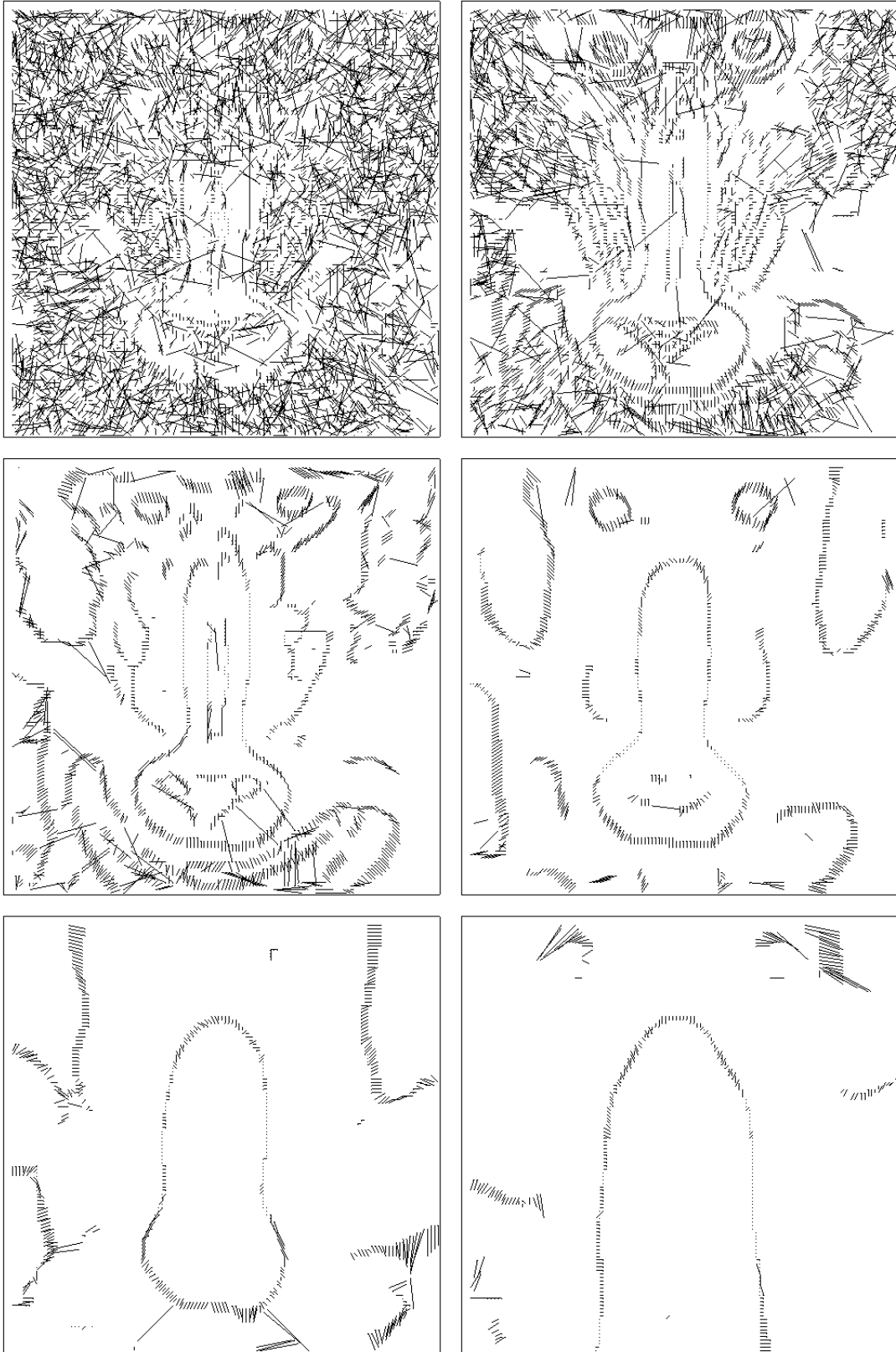


Figure 5.5: Matching output: scaling, levels 1-6

included just for comparison. There is a substantial difference in running time as well. With the particular implementations we used (see below for details), running time for this example was 2.1 minutes for MC matching, 6.8 minutes for MW matching, and 22.2 minutes for MWMC matching.

Now, it is definitely not always true that MW matching gives the best results. Consider the example¹ of an image containing nothing but a horizontal line, say about 60 pixels long. Shift this image horizontally by 10 pixels, find edges, and match. For the moment, consider only the lowest level (finest scale). The edges in the two images overlap for 50 pixels of their length, while each image has 10 pixels that do not overlap edge pixels in the other image. The MW matching algorithm proceeds to match the 50 overlapping points each with the point in the same location in the other image, leaving 10 unmatched points at the end of the line in both images. Essentially, it is worthwhile “abandoning” those points on the end to get the higher edge weights of the matched points in the middle. Fitting a transformation to these matches will yield the identity transformation, which is quite wrong.

In contrast, MC matching matches all of the points. Though it may match pairs of points essentially randomly, any transformation that results from fitting these matches must be closer to the truth than the identity transformation. The MWMC algorithm also matches all of the points. Some of the matches are too long and some too short, but the errors average out and the first pass of least-squares fitting gives a nearly perfect answer.

5.1.6 Improving maximum-weight maximum-cardinality

The example points out that MWMC sometimes gives better results because it “tries harder” to match all of the points. But it usually performs worse on photographic images. Can we improve it?

Examining a graphical display of the matches found by the MWMC algorithm shows a number of really bad matches (very long match distance) where one of the two points is close to the edge of its image. In many of these cases, this point in one image has no truly corresponding point in the other image due to differences in the cropping of the two images. Thus, no good matches for these points existed. MW matching was free to leave these points unmatched, but MWMC matching was bound to match them to something if possible, with predictable results.

We decided to try to alleviate this problem. For each point in an image that is within some small distance of the edge of that image, we add a “dummy” node to the other partition of the graph. Then we link the new dummy node to the graph node corresponding to the point with an edge that has zero weight. We do this for both input images. Then we run the MWMC matching algorithm as before. When we write out the data, we only output matches where both vertices are actual points in the two images. A point that matched with a dummy node is discarded.

The effect is that each of these near-boundary points now have a hidden dummy node that wants to “capture” it. In fact, the MWMC matching does match most of these edge points with their dummy node and they vanish from the matching output. However, a few of these nodes are matched with a point in the other image when it is advantageous to do so. When we applied this to the test case described above, we got 27448 matches, so about 11% of the former matches were captured by the dummy nodes. Surprisingly, the ratio of correct matches among the remaining ones did not improve — it was still 20%. Looking at the diagrams of the matching output, the edge points have nearly vanished, but there are just as many poorly-matched pairs in the interior of the image.

¹suggested by Dr. David Kirkpatrick

There are a number of ways we should be able to improve on this. Instead of providing one dummy node for every near-edge point, we could provide only (for example) 50% as many dummy nodes as edge points, and provide an edge from every edge point to all of these nodes. The MWMC matching algorithm would then be forced to choose at most half of the edge points to be captured, rather than capturing all of them.

We could extend extra edges from some number of dummy nodes to all real-point nodes in the graph, enabling poor matches to vanish from the interior of the image as well. However, it would be impractical to connect all of thousands of original nodes to each of hundreds of new dummy nodes — that’s just too many new edges. We would have to do something like connecting each original node to a few dummy nodes, selected at random from the pool of dummy nodes.

Matching is already the most expensive stage of the entire algorithm, and MWMC matching typically takes several times longer than MW matching already. We are reluctant to pursue solutions that would slow Phase 2 even further.

In addition, MW matching actually does work even for the example of the single horizontal line that we used above. It is true that it does a poor job of matching the level 1 data without help, but at higher levels the horizontal line fattens into a structure that looks rather like a cucumber, with two sides and two ends. The MW matching algorithm does a fine job of matching the edges from this higher-level appearance. If we are also using iterative multiscale matching, the nearly correct transformation found from these higher-level features is applied to the level-1 data before matching, eliminating almost all of the 10-pixel offset before matching. Then even level 1 is matched correctly after all, and we find the correct transformation.

5.1.7 Output

Once the matching is complete at any one level, we must write out the matches found. The matching algorithm returns a list of edges that defines the matching. We scan through this list of edges, and identify the nodes that the edges are incident with. Then we write out the two points (location and gradient) to the output file. (If we are using dummy nodes in the matching process, we only write out matched nodes when both nodes are real image points.)

5.2 Performance

This is the slowest phase of the pipeline, sometimes by a lot.

The time and space used by Phase 2 is highly dependent on the number of vertices and edges in the graph. A threshold in Phase 1 sets the maximum number of points that will be output per level, which directly controls the number of vertices in the graph. The number of edges is controlled both by the number of vertices, and by the size of the “window” used to determine how far away we will look for matching points.

For example, suppose we start with a pair of 500×500 pixel mandrill images that are slightly different. If we make set the maximum-points threshold in Phase 1 to 10000, and make the maximum match length in Phase 2 to 10% of the image width, the matching process (all levels) takes 252 megabytes of memory and 27.7 minutes of processor time (Pentium III 700 MHz). If we reduce the match length limit by a factor of two, matching uses 92 MB of memory and about 8.4 minutes. The actual output is nearly the same in both cases, so it is clearly useful to keep the match length limit as small as practical (while still having it large enough to include the range of disparity expected between the images).

Reducing the number of vertices helps even more. If we set the maximum-point threshold in Phase 1 to 5000 points instead of 10000, and reset the match length limit to 10%, the matching uses 80 MB of memory and takes 5.8 minutes. If we also reduce the match length limit to 5% of image width, the cost drops to 31 MB of memory and 2.0 minutes.

Of course, with half as many input points, the output has only about half as many matches. But 5000 points at each of several levels (about 30000 matches in total) is still more than enough data points to determine a global transformation between the two images in the next phase of the pipeline. After all, we're fitting an affine transformation with only six free parameters. We could reduce the maximum-point limit still further.

Iterative multiscale matching improves the situation considerably. We normally have Phase 1 decimate the edge images starting at level 3 when multiscale matching is in use. Thus, levels 1 and 2 are at full input image resolution, but the size of the higher-level images decrease by a factor of 2 per level. This reduces the number of edge points by a factor of about 4 per level. As a result, the highest several levels can be matched and fitted in a fraction of a second per level.

The lowest levels benefit too. As the multiscale matching process proceeds, the transformation data from the higher levels is used to prealign the lower-level data, allowing the size of the matching window to be gradually reduced. The two lower levels typically use a window of 2% of the image width (the minimum allowed) though the highest level begins with a window of 10% of image width. This factor of 25 reduction in window area greatly reduces the number of edges in the graph, which speeds up matching. The matching and least-squares fitting with a 10000-point limit take a total of 53.6 seconds for all eight levels. With a 5000-point limit, the total matching time drops to 10.2 seconds. (The reported times do not include the time needed for the least-squares fitting step or any of the multiscale fitting overhead. These additional activities add only 2–3 seconds in these two examples.)

5.3 Implementation

There is a program named `match` that performs all of the Phase 2 operations described above.

The actual matching algorithm we use comes from the LEDA software library. LEDA stands for Library of Efficient Data types and Algorithms. It was developed to provide combinatorial and geometric data types and algorithms for researchers. The LEDA book [47] describes it at length. We use only a small subset of what it provides, specifically a few bipartite graph matching routines and their supporting data structures.

Since we use a LEDA library routine to do the matching, we have to build the data structures it needs. In particular, we need a `GRAPH` structure for the graph itself, a pair of `list<node>` node lists to define the bipartition of the vertices, and an `edge_array<double>` containing the edge weights. We actually store the latter in the edge nodes in the graph data structure.

When we are about to start building the graph for one level of matching, we begin by looking at the number of maxima points from the two images at this level. The matching algorithm runs faster if the “left” side of the bipartite graph has fewer nodes than the “right” side, so we simply arrange a pair of pointers so that the “left” one always points to the image with fewer points at this level. From our point of view the matching problem is symmetric, so it makes no difference which image is used for the “left” side, and we might as well make the algorithm run as fast as we can.

The code of `match` can be conditionally compiled to use one of three different LEDA subroutines for the matching. By default, we use `MAX_WEIGHT_BIPARTITE_MATCHING_T`, which generates a simple

maximum-weight matching. Optionally, we can use either `MAX_CARD_BIPARTITE_MATCHING`, which generates a maximum cardinality matching, or `MWMCB_MATCHING_T`, which generates a maximum-weight maximum-cardinality matching.

All of the methods above use double-precision floating-point weights, perturbed as necessary by `MWBM_SCALE_WEIGHTS`. The code of `match` has a fourth option as well: maximum-weight bipartite matching using integer weights. The scaling from floating-point to integer weights is done as described in the next section. This special integer-weight version produces results that appear to be as good as the floating-point version of the same algorithm, but it uses less memory and is 10-15% faster. Thus, it is the default.

At each level, the LEDA matching algorithm returns a list of edges. For each edge in the list, we look up the two vertices that are adjacent to the edge. The vertices in turn provide pointers back to the original edge point data records read from the input file. Each of these records gets a pointer to each other. When we're done, each point from either image that was matched to a point from the other image now has a pointer to that point. Now the LEDA graph memory can be freed, and we go on to process the next level.

5.3.1 Weight Scaling

Before we can start the matching algorithm itself, we need to do a bit of “tweaking” of the edge weights. The problem is that the matching algorithm we use operates by calculating and comparing weight values, and the algorithm correctness depends on those arithmetic operations being performed exactly, with no overflow or roundoff error. Without this being guaranteed, the algorithm may not terminate, or it may produce incorrect answers. Section 7.2 of the LEDA book contains a thorough explanation of the edge weight scaling process; what follows here is an abbreviated treatment.

One possible solution is to use exact computer arithmetic. LEDA provides an `integer` data type that gives exact results without overflow (using as much memory as necessary), a `rational` data type for arbitrary-precision rational numbers, and a `bigfloat` type for floating point with an arbitrary (but predetermined) number of mantissa bits. Any of these could be made to work, but their variable-size character makes operations considerably slower than the built-in machine arithmetic types. LEDA provides a better method for our purposes.

An analysis of the weighted matching algorithms shows that the only arithmetic operations used on the weights are addition, subtraction, and comparison. If the weights are integer values, there can be no rounding errors introduced during the computation, so the only possible problem is overflow. The LEDA authors have proved that the largest calculated result that can occur during the algorithm is no larger than 3 times the maximum absolute value of the input weights. If the maximum representable integer is $2^{31} - 1$, then if we can guarantee that $W \leq (2^{31} - 1)/3$ for all edge weights W , no overflow is possible and the algorithm will work properly. If the maximum absolute value of any edge weight is C , we can safely scale all of our floating-point weights E to produce new integer weights E' using

$$E' = \lfloor \frac{2^{31} - 1}{3C} E + 0.5 \rfloor$$

(In our case, $C = 1$ because of the way we calculate E .)

These scaled integer weights will lose some of the precision of the floating-point edge weights, but with up to 29 significant bits remaining it will be a very unusual event for two weights that are unequal in floating-point form to become equal in integer form. Thus, we are unlikely to create many situations where two edges that should have different weights are made equal via this process.

However, it is not necessary to use integers; we can use a similar technique to make floating-point arithmetic error-free. An IEEE-format double-precision floating point number can store any integer whose absolute value is up to $2^{53} - 1$ as an exact value with no rounding. Let us define a scale factor S which is a power of two, and then scale all of the weights using it:

$$\begin{aligned} S &= 2^{\lfloor 53 - \log_2(3C) \rfloor} \\ E' &= \text{sign}(E) \lfloor S|E| \rfloor \end{aligned}$$

(We use truncation toward zero rather than rounding because rounding requires an extra mantissa bit and we're already using all of them.) If we do this, all of the scaled weights E' are integers that can be stored exactly in a C `double`, and all sums and differences that may be calculated in the bipartite matching algorithm are also exactly representable, so the arithmetic done is all free of rounding error. (The rules of IEEE arithmetic specify this.)

However, since the scale factor S is a power of two, dividing by S simply subtracts some integer value from the exponent part of the dividend, while leaving the mantissa untouched. We can generate scaled weights \hat{E} using

$$\hat{E} = \text{sign}(E) \lfloor S|E| \rfloor / S$$

The mantissas of E' and \hat{E} are identical, and the exponents differ by a constant ($\log_2(S)$).

Note that these weights \hat{E} are nearly equal to the original weights E numerically, which is convenient. Yet they are just as immune to rounding errors inside the matching algorithms as the integer weight values (as long as the arithmetic is done according to IEEE rules).

Intuitively, the process described forces all mantissa bits that represent values less than $1/S$ to zero in all edge weights. S is chosen so that no possible calculation produces a result where one or more “one” bits are shifted out and either lost or rounded, so the arithmetic within the matching algorithm is exact. S is also chosen to retain as many mantissa bits as possible given this constraint. In this case, up to 51 mantissa bits may be retained, considerably better than we could do using 32-bit integer weights.

It would be possible to use exactly the same scaling process to make single-precision floating-point weights error-free. However, single-precision IEEE float has only 24 bits of mantissa precision, while integers provide 31 useful bits in the same amount of memory, so it is always better to use `int` instead of `float`.

Essentially, the process described converts the original weighted matching problem into a slightly different matching problem that the matching algorithm is known to work correctly on, by perturbing the weights. In general, the result of the modified problem might not be exactly the same as the true result of the original problem, but we don't expect the difference to be noticeable in this application.

LEDA provides a routine named `MWBM_SCALE_WEIGHTS` to do the `double` version of the scaling described above. If we want to do the integer version of the scaling, we must do it ourselves. We use both in different circumstances.

5.4 Other Software

There are also two auxiliary programs that are part of Phase 2, named `mview` and `mhist`. The `mview` program is used to visualize the output of `match`; it reads in the output file and displays the matching in a workstation window as a series of vectors. (It produced the displays in Figure 5.3.) The `mhist` program

calculates a histogram of match vector lengths. Both of these are straightforward and we won't describe them further.

5.4.1 Implementation History

Our first implementation of the matching phase did not use LEDA. Because the idea for using graph-theoretic matching came from Vishwa Ranjan's thesis [58], we started by using the same matching algorithm. This is the "CSA" algorithm (the name means Cost Scaling Algorithm) of Kennedy and Goldberg [23].

Unfortunately, this turned out to be quite unsuitable for our purposes. CSA was really designed to solve minimum-weight assignment problems. An assignment is a perfect matching (all vertices are matched) in a bipartite graph. If a graph does in fact have an assignment, the CSA algorithm will find the minimum-cost assignment, and do so faster than most other alternatives. However, if the graph does not have an assignment, the algorithm may never terminate, or it may terminate having found a matching that is not a minimum-cost matching. (CSA was really designed for use on artificial data sets designed for benchmarking matching algorithms. These data sets are known to have an assignment by their construction. CSA was never designed to handle arbitrary real-world data.)

To make CSA work in matching sets of image features, we had to build graphs that were guaranteed to have a matching. This means that both vertex partitions must be the same size, so we had to add dummy vertices to whichever image had fewer real data points. Then, to ensure the existence of a perfect matching, we built a complete bipartite graph: an edge connects every possible pair of vertices. This ate memory at an alarming rate — a 256-pixel-wide image might need one gigabyte of memory for matching.

In addition, CSA is fundamentally a maximum-cardinality matching algorithm and it had the problem of generating really bad matches for features that appear in only one image, just like the LEDA maximum weight maximum cardinality algorithm.

We eventually abandoned CSA in favour of the LEDA package. This allowed us to build graphs with a relatively sparse set of edges using the maximum match length "window". Now each vertex is typically connected to about 2-5% of the vertices in the other partition (with the default 10% length limit) instead of all of them. Essentially, we only include edges that have some reasonable probability of being part of the solution. Also, switching to maximum weight matching got rid of the most of the bad matches for near-edge features, improving the quality of matches for our purposes.

Unfortunately, LEDA has its own problems. It is a large and very complex system written using many of the more complex features of the C++ language, making it difficult to find and fix bugs.

We originally tried using a binary distribution of LEDA, but it did not link properly with our program. We then obtained a source-code distribution of LEDA, and it worked fine — at first. After an operating system upgrade including a compiler upgrade, LEDA would no longer compile. The problem turned out to be the authors using a behaviour of the preprocessor that the C++ standard explicitly says is undefined, but which happened to work with most compilers. The new compiler conformed more strictly to the language specification, and rejected the old LEDA code. We ended up rewriting a header file to conform to standard practice; there was no real reason why the nonstandard (and more complex) scheme originally used was needed.

On another occasion, we found that a particular routine was always returning an exit status indicating that it had not modified any data, even when we knew it must have done so. This turned out to be a simple bug of overwriting input data with output data before comparing them for equality.

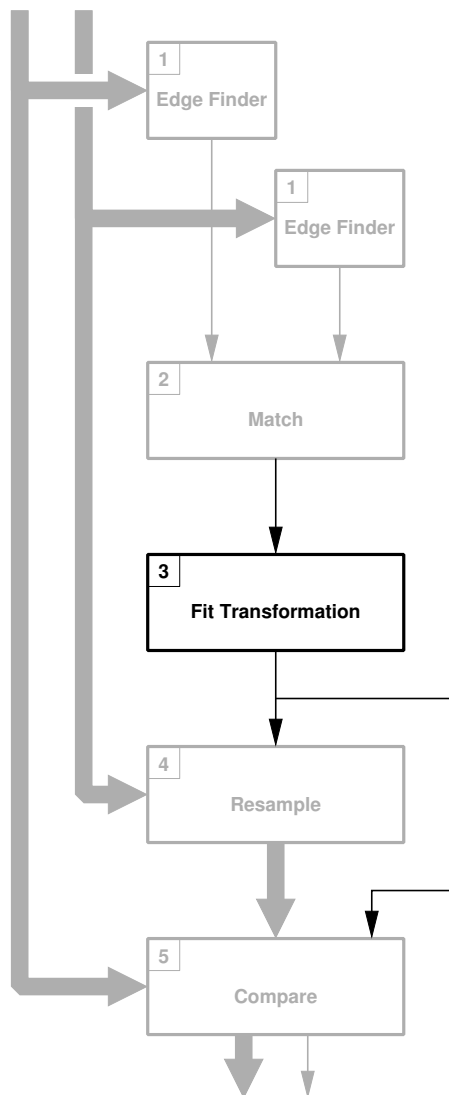
Part of the problem is that LEDA was originally an academic research project but it has become commercial software. When we began using it, there was free distribution to other researchers, a mailing list for bug reports, and other accoutrements of research software. Later, LEDA was transferred to a commercial entity. It is no longer available for free to anyone, support costs money, and the old bug reporting/support mailing list has vanished. So we ended up finding and fixing these problems independently.

In this thesis, we use only a tiny fraction of what LEDA can do, and most of its power is wasted on this rather simple application. If someone else wanted to use this thesis software for anything, even further research, they would have to pay for a binary or source licence. (The LEDA code used in this thesis is licenced only to the author, and we can neither use the software commercially nor transfer the licence to another researcher.) A binary licence now costs a nominal amount of money, but a source licence costs more than US\$10,000.

If another researcher wanted to use or extend this thesis, we would recommend that they simply search for a free or inexpensive implementation of a maximum-weight matching algorithm, instead of continuing to use LEDA. A custom implementation would likely be smaller and faster than the LEDA one.

Chapter 6

Phase 3 — Transformation Fitting



Phase 2 has provided a list of “matching” points in our two input images. Some of the matched points are, in fact, correct matches: the two points are the same visible image feature in both digital images (with some fractional-pixel uncertainty added by the discrete sampling grid in the two digital images). Other matches are incorrect — the Phase 2 algorithm simply picked the wrong pair of points.

The task of Phase 3 is to estimate the geometric transformation needed to map one image onto the other, aligning features as accurately as possible, in the face of input data that is a mixture of correct and incorrect matches.

Ultimately, we need a mathematical model of the possible transformation, with some number of parameters that control how it operates. We then estimate the values of those parameters based on the data passed from Phase 2. The ultimate value of the transformation we obtain depends at least as much on whether the mathematical model chosen can actually model the difference between the images as it does on the quality of the matching data.

6.1 Models

There are several fundamental choices that need to be made before writing any software. Brown’s “A Survey of Image Registration Techniques” [4] reviews the possibilities. We could try to find a single global transformation that maps the entire image, or instead use a mesh of localized transformations that can vary across the extent of the image. Geometric differences due to camera misalignment or non-ideal lens imaging behaviour are global effects and can (in principle, at least) be handled by a global transformation. On the other hand, localized errors (e.g.

a computer model of a room that simply places one object in a different location than the real object in the real room) are poorly handled by a global transformation.

For simplicity, we chose the global transformation approach, recognizing that this limits the range of image differences that we can hope to match. This allows us to obtain a transformation that is useful in some cases, and allows us to test the entire system as a whole, without spending a great deal of time on Phase 3 alone. At the same time, we recognize that a global fit will sometimes have problems, so this should be considered a prototype of the fitting step, not the best possible implementation.

Once we've settled on finding a global transformation, we must pick a model for that transformation. Again, Brown covers the range of choices. We could assume that the transformation is a rigid 2D transformation composed of only translation, rotation, and uniform scaling. This can handle cases where the two cameras (real or virtual) were misaligned by a rotation around the optical axis or had a slightly different field of view, and errors in alignment of the optical axis of the lens with the centre of the image.

Or, we could assume a general affine transformation. This allows translation and rotation too, as well as independent scaling in the X and Y directions. Some digital cameras do have non-square pixels, so the non-uniform scaling is a useful additional degree of freedom. It can also model reflection about an arbitrary line and a shear, although these do not seem useful for our purposes. One very useful attribute is the ability to represent any possible affine transformation with a single matrix. The mapping itself is a linear function.

A perspective transformation is even more useful, since it allows compensating for a change in the location of the camera lens. However, this introduces several new problems. The perspective transformation requires a knowledge of the three-dimensional position of features in the image, which is very difficult to obtain when all you have to work from is a single 2D image. In some cases, if the one image is the result of a 3-D rendering algorithm, it is possible to have the renderer save a depth map along with the visible image. If multiple images of the same scene are available, taken from different viewpoints, stereo vision algorithms can recover 3-D depth information. It is also common for a foreground object to hide some portion of background objects in real scenes. If we are trying to calculate what the scene looks like from a slightly different viewpoint, there will be pixels in the output image for which we have no valid data in the input image, because that area of the object was hidden.

The projective transformation lies between the complexity of the affine and perspective transformations. The mapping in a projective transformation is done by rational functions (i.e. the ratio of two polynomials). A projective transformation can handle the effect of moving the camera's viewpoint when the scene is planar, or nearly so (e.g. an aerial photograph). It does so without needing explicit depth information for every point in the image.

We have decided that, at least for our initial prototype, we want to do our fitting using linear least squares methods. Linear least squares fitting is well understood, stable, robust, and relatively fast. The "linear" in "linear least squares" means that the result is modelled as a linear combination of some set of basis functions, not that the basis functions themselves must be linear functions of the input variables. Thus, the basis functions may be nonlinear, but the output must be modelled as a weighted sum of terms. Due to this constraint, we can't work with the projective transformation model, since its mappings are determined by the ratio of two polynomials.

So, we chose to assume an affine transformation, which is the most general transformation that we can still solve using linear least squares. (See the Discussion section for further comments about this choice, and alternatives.)

6.2 Method

6.2.1 Transformation Math

We have chosen an affine transformation model. This can be expressed as multiplying a 2-vector (the original position) by a 2×2 matrix (for the rotation/scale/skew), followed by adding another 2-element vector for the translation. If (x, y) are the coordinates of some feature in the input image and (u, v) are the coordinates of the same feature in the output image, the general affine transformation is given by

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} a_{02} \\ a_{12} \end{bmatrix}$$

However, it is simpler to borrow a technique from 3-D computer graphics and use homogeneous coordinates. We represent a 2D position as a 3-vector whose final component is always 1, and an arbitrary affine transformation can then be represented by a single 3×3 matrix:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The a_{ij} parameters with the same name have the same values in both of these formulations. Thus, the rotation/scale/skew effects are determined by the upper left 2×2 submatrix, while the right-hand column's first two entries determine the translation.

Note that it is equally valid to adopt the convention that vectors are rows, not columns. Current computer graphics fashion is to use column vectors, but the row-vector form is closer to the design matrix that we'll see in a moment. The row-vector form of the transformation model is:

$$\begin{bmatrix} u & v & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} a_{00} & a_{10} & 0 \\ a_{01} & a_{11} & 0 \\ a_{02} & a_{12} & 1 \end{bmatrix}$$

Note the transposition of the 3×3 matrix.

It is convenient having all the parameters in a single matrix, but the real advantage of homogeneous coordinates comes when you want to calculate the effect of the composition of two different transformations. To do so, you simply multiply the two 3×3 matrices in the correct order (right to left for column vectors, left to right for row vectors).

6.2.2 Input

Our algorithm begins by reading in the output file produced by Phase 2. The data points from each wavelet transform level are read into separate arrays, so that the fitting can be done at each level individually if desired.

For each match record, we save only the (x, y) location of the points that were matched. Length and orientation of the gradient are discarded. We offset the coordinates by half the width and height of the image that the point belongs to, effectively putting the origin of our coordinate system in the exact centre of each image. We do this because if any rotation is present, it is far more likely to be rotation around the centre (or a point near the centre) of the image than it is to be rotation around the upper left corner. A coordinate system with the origin in the centre lets us easily read any translation present independent of the rotation.

6.2.3 Least-Squares Fitting

The least-squares fitting technique we will use is a standard method, described in many numerical analysis textbooks, using the Singular Value Decomposition (SVD). Since we are using a standard technique, we will touch on the details of this technique only lightly, leaving a detailed explanation to the textbooks. For more details, look at [24], [34], and [56].

The SVD is the method of choice for this sort of problem. It is robust in the face of singular data — it never fails due to singular data, and (used properly) doesn't give unstable answers when given near-singular data. The wavelet transform has few significant edges present at the highest levels. We have seen cases where the maxima found were all located in a single column or a single row of the image. This produces a singular system.

In addition, the SVD is well-behaved numerically. The worst-case error in the output is proportional to the condition number of the \mathbf{A} matrix plus the condition number squared times the residual. It is also slower than other methods, but not enough to bother us in this application.

Looking at the matrix equation, we can see that the equation for the u component of the output of the transformation is $u = a_{00}x + a_{01}y + a_{02}$. In least-squares terminology, the a_{ij} are the parameters that we want to fit. The three basis functions we are using are simply x , y , and 1.

To solve for the parameters, we start by building the design matrix. This is a matrix with one row for every data point we have, and one column for every basis function. Each entry in the design matrix \mathbf{A} is the value of the appropriate basis function (determined by column) evaluated at the i th abscissa where i is the row number. With the basis functions we are using, this is just x_i , y_i , and 1. We also build a vector \mathbf{b} whose i th entry is the ordinate, the u coordinate from the i th data record. Finally, we have a vector \mathbf{x} consisting of the three parameters we wish to solve for. We can write the matrix equation that we want to solve as $\mathbf{Ax} = \mathbf{b}$. Expanded, it is:

$$\begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} a_{00} \\ a_{01} \\ a_{02} \end{bmatrix} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ \vdots \end{bmatrix}$$

Although this is written as an equation, in general there is no possible \mathbf{x} vector that makes the two sides equal. We can have equality only if every observation fits the model perfectly for some \mathbf{x} vector, and that is very unlikely with real measurements of any kind.

Instead, with real data, we must expect there to be a non-zero difference between the predicted ordinate given by \mathbf{Ax} and the actual measurements \mathbf{b} . This difference is called the residual, and defined by $\mathbf{r} = \mathbf{Ax} - \mathbf{b}$. We “solve” the system by finding the \mathbf{x} vector which minimizes the length of the residual vector $|\mathbf{r}|$. In other words, the solution minimizes the sum of the squares of the residual values for each data point.

Now, so far we've only set up a least-squares problem for one of the two output coordinates, and three of the six parameters. The v output coordinate depends on the parameters a_{10} , a_{11} , and a_{12} in exactly the same way that u depends on a_{00} , a_{01} , and a_{02} , with exactly the same design matrix \mathbf{A} . So although this can be thought of as two separate least-squares problems, we can also write it as a single matrix system by making \mathbf{x} and \mathbf{b} two-column matrices containing the two sets of parameters and the two sets of ordinates.

Our system now looks like:

$$\begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} a_{00} & a_{10} \\ a_{01} & a_{11} \\ a_{02} & a_{12} \end{bmatrix} = \begin{bmatrix} u_0 & v_0 \\ u_1 & v_1 \\ u_2 & v_2 \\ u_3 & v_3 \\ \vdots & \vdots \end{bmatrix}$$

The solution is obtained in three stages. The first stage uses the SVD algorithm to decompose the \mathbf{A} matrix into three matrices \mathbf{U} , \mathbf{W} , and \mathbf{V} : $\mathbf{A} = \mathbf{U}\mathbf{W}\mathbf{V}^T$. If \mathbf{A} is $M \times N$, then \mathbf{U} is $M \times N$, \mathbf{W} is $N \times N$, and \mathbf{V} is $N \times N$. \mathbf{U} and \mathbf{V} have columns that are orthonormal, while \mathbf{W} is diagonal.

The second stage examines the diagonal elements in \mathbf{W} , called the singular values. If any of the singular values are zero, or excessively small, this indicates that the basis functions are nearly linearly dependent at the data points used, and this fact is noted. Also, excessively small singular values are set to zero, which improves the quality of the eventual solution.

The final stage uses back-substitution to solve for the \mathbf{x} matrix, given the data matrix \mathbf{b} and the \mathbf{U} , \mathbf{W} , and \mathbf{V} matrices produced by the SVD.

6.2.4 Pruning

The description above discusses solving a single least-squares fitting problem given a single set of data. What we actually do is somewhat more complex.

The input data always includes a significant number of “bad” matches. We think of a “good” match as being a pair of points that are actually the same visual feature in the two scenes being compared, plus or minus the inevitable 0.5 pixel error due to the integer pixel grid. All other matches are “bad”; the points matched do not correspond to each other visually. In test images, the fraction of good matches can sometimes be below 20%, though sometimes it is more than 50%. The usual predominance of bad matches means that a straightforward least-squares (LS) fit using all the data points may result in a transformation that is far from accurate.

If we were optimistic, we might hope that the bad matches are essentially random and their errors would tend to cancel each other during the LS process. However, that is not true. If a point is not matched to its correct mate, it is more likely to be matched to an incorrect point that is closer than the correct point, compared to one which is further than the correct point. Because of the maximum match length window used in Phase 2 and the fact that the weight of an edge is affected by distance, Phase 2 disproportionately prefers too-short errors to too-long errors. In turn, this means that a LS fit using all the data is very likely to produce a transformation that underestimates the amount of motion required, whatever the motion is (rotation, translation, or scaling).

The solution to this is to prune away most of the bad matches, leaving most of the good ones. We do this using an iterative approach which usually proceeds automatically, though it can be supervised by a human operator. (Although we developed this method independently, it has similarities to some published statistical analysis techniques intended for data with many outliers. See the Discussion section below for more details.)

The algorithm uses an “active” flag for each data record (i.e. each pair of matched points). Active points participate in the LS fitting step, while inactive points do not. This is an outline of the algorithm:

1. Mark all points as active initially.
2. Perform a LS fit for the transformation matrix using only data that is currently marked active.
3. Given this transformation matrix, calculate the residuals for all data records, active and inactive.
4. Calculate the standard deviation of the X- and Y-direction residuals for the active data only. Use these to define an “acceptance region” as an ellipse centred on the origin whose semi-axis lengths are twice the X and Y standard deviation. In other words, if σ_x and σ_y are the standard deviations of the X and Y residuals, then any point with residuals (r_x, r_y) is within the ellipse if $(r_x/(2\sigma_x))^2 + (r_y/(2\sigma_y))^2 \leq 1$.
5. (Manual control:) Display a plot of the residuals of the data (active and inactive), the standard deviations, and the elliptical acceptance region to a person who is controlling the pruning. If the operator decides that the pruning process is not converging usefully, or that it has already converged to a satisfactory degree, they instruct the program to break out of this loop.

(Automatic control:) A heuristic algorithm decides whether to break out or continue for another iteration. The decision is based on the size of the residuals of the active matches, the fraction of the original data that is still active, and an iteration count limit.
6. If continuing, prune the data: Make a pass through all the data records, marking them active if their residuals are within the elliptical region and inactive otherwise.
7. Go to step 2 for the next pass.

The basic idea behind the algorithm is very simple: Some fraction of the data consists of “good” matches. If we somehow were given the perfect transformation matrix immediately, the residuals for all of the good points would be very small, within $[-0.5, 0.5]$ in X and Y. Even when the transformation matrix is close to the correct one the good data points are likely to have small residuals, and most good matches will have residuals that cluster near the origin.

Meanwhile, the rest of the data consists of “bad” matches. When projected by the perfect transformation matrix, these matches will have residuals that range from relatively small (but more than 0.5 pixels) to very large. Even with a less-than-perfect transformation, many of these points will still have large residuals.

Thus, if we perform a least-squares fit with all the matches participating, we expect to find a transformation that isn’t too far from correct. The good matches will probably have small residuals and there will be a tight cluster of them near the origin. If we graph the residuals, we can expect to see this cluster. The bad matches will have a much wider range of residuals. If the data fits this pattern, then there will be some points outside the “two sigma” ellipse, and it is nearly certain that most of these are from bad matches. Then, when we perform one pruning step, we eliminate these particularly bad matches, while keeping all (or nearly all) of the good matches. The next LS fitting step will be more strongly influenced by the cluster of good matches, and they should have smaller residuals and cluster even closer to the origin on the next iteration. In turn, the standard deviations of the residuals should be reduced, resulting in more bad matches being outside the ellipse. They will be pruned on the next iteration. If all goes well, we simultaneously converge on including mostly good matches and excluding bad ones, while refining the transformation matrix.

Note that during pruning (step 6), the algorithm always looks at all of the data, not merely the currently-active ones. Sometimes some good matches will have large residuals at first because the initial transformation matrix is poor. These data values will be marked inactive because of the large residuals. However, if

the algorithm converges toward the correct transformation, the residuals for these good but inactive matches will become small. Once the residuals drop below the current pruning boundary, the points will become active and participate in the fitting once again.

Now, we can't guarantee that the process will always work the way we've described. We have not proved that it will always converge, or that when it converges the results are always correct. But we can look at these issues a little more closely.

First, suppose we have a set of data where the "matches" consist of random pairs of points with no actual correlation at all. The LS fit will find no particularly interesting transformation, and the plot of the residuals will also be approximately random. In one dimension, if we have data that has a uniform random distribution over $[-L, L]$, the standard deviation σ of that data is $L/2$. If we set a threshold of 2σ for accepting data points, the threshold will be at L and we will keep all the original data points, pruning none. Thus, the algorithm will never converge. (In two dimensions, the fact that we use an elliptical pruning boundary instead of a rectangular one means that we will prune a few points even with a uniform distribution, so we will get slow convergence.)

However, with any distribution that shows a central tendency, with a higher probability of points near the middle than toward the edges, the standard deviation of the residuals will be less than half the distance to the largest residual, there will be points located outside the 2σ boundary, and pruning will take place. So, as long as the distribution of residuals has a central cluster with less dense outlying points, we can expect progress from each iteration.

On the other hand, if there is no affine transformation that adequately models the transformation between the two images (i.e. we picked the wrong model, or the images are too unlike to be aligned with any model), then we can not expect to see any central cluster in the residuals, and the algorithm will fail. (But there really wasn't any correct answer available, given our assumptions, in this case.) If the number of good matches is too small a fraction of the total, the initial LS fit may yield nothing like the correct transformation, the good matches may not form a cluster at all, and the pruning will not have anything to guide it so the final result is likely to be bad.

Example

In practice, the algorithm appears to work very well in cases where there was in fact an affine transformation between the two images. For example, Figures 6.1-6.8 show an example case where the transformation was a rotation by 5 degrees. (The images are a 500×500 mandrill, basically the same example used near the end of the previous chapter.) The "perfect" transformation matrix is

$$\begin{bmatrix} 0.9962 & -0.0872 & 0.0 \\ 0.0872 & 0.9962 & 0.0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 6.1 shows the initial display when `fit` is applied to this data. The window title bar tells us the name of the file we are working on. It also tells us that we are using all 30324 data points in the fit.

The grey grid over the entire plot area is drawn with a one-pixel distance between grid lines, which gives us a scale reference. The plot is automatically scaled to show all the active points, so we'll see the grid spacing change as we "zoom in" as the residuals get smaller. The two black lines in the centre of the grid are the axes. In this first display, the residuals are all within a range of about 40 pixels from the origin.

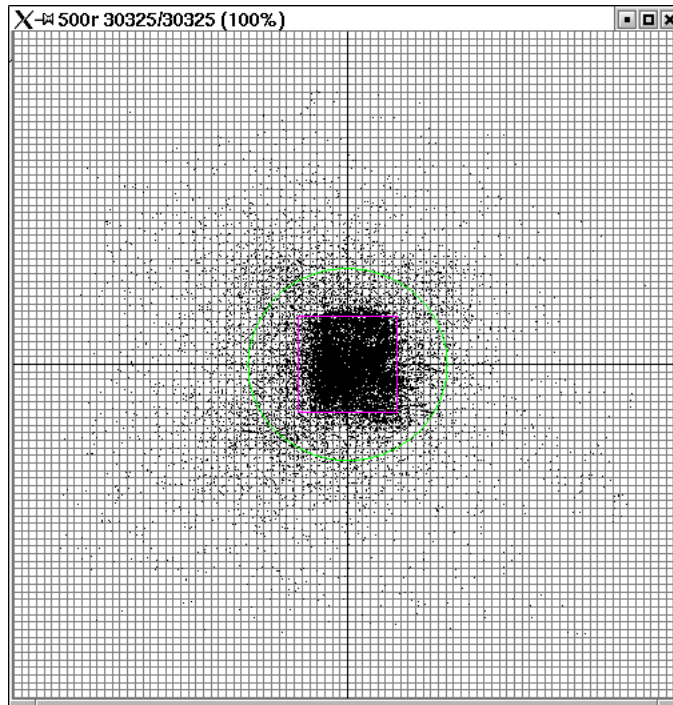


Figure 6.1: Fit: 0 iterations

The black points on the graph are the residuals themselves. They are clearly denser near the origin, with a very dense cloud having a radius of about 8 pixels in the centre. The rectangle near the centre is drawn at plus and minus one standard deviation of the active points, to show us how large σ is. The ellipse is the boundary of the 2σ “acceptance region”. The points inside this ellipse will remain active if the operator decides to perform a pruning step. (Some of these features are difficult to see in Figure 6.1, but they will become more visible during later iterations.)

The initial transformation matrix from the LS fit is

$$\begin{bmatrix} 0.9959 & -0.0640 & -1.0018 \\ 0.0557 & 0.9976 & 0.1285 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 6.2 shows the result of one pruning step followed by re-fitting of the transformation. The title bar tells us that 87% of the matches are still active, so the pruning step eliminated 13%. The scale is different — we have zoomed in by more than a factor of two. The dense cloud of residuals now has a radius of about 4 pixels, indicating that the transformation is better, with smaller residuals. In fact, the new transformation matrix is

$$\begin{bmatrix} 0.9960 & -0.0777 & -0.3660 \\ 0.0704 & 0.9963 & 0.4137 \\ 0 & 0 & 1 \end{bmatrix}$$

The black dots are the residuals of the active matches, as before. You can also see some less dark dots near the outer edge; they are the residuals from inactive matches. The rectangle and inner ellipse have the same meaning as before, but they are now smaller because the standard deviation of the new residuals is

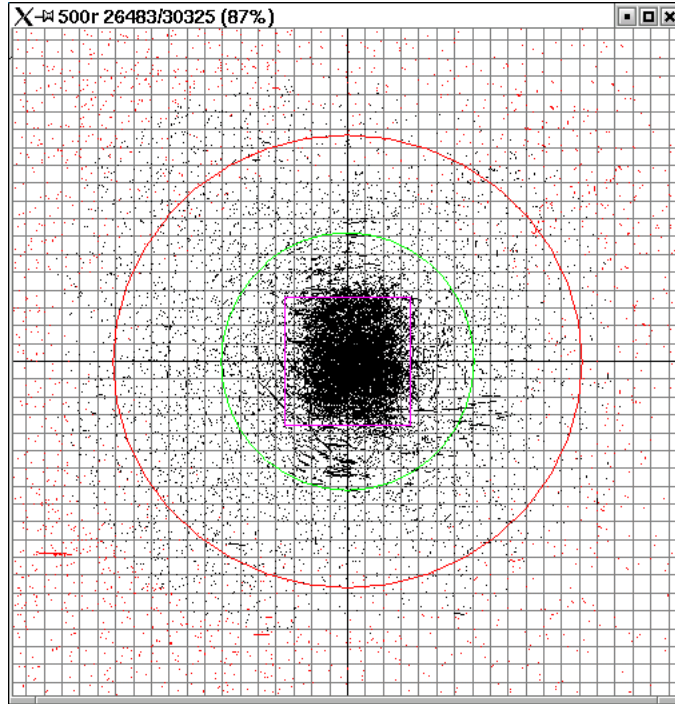


Figure 6.2: Fit: 1 iteration

about half what it was before. There is also a new outer ellipse, which simply shows us the size of the inner ellipse from the previous stage (i.e. the pruning boundary).

Note that, roughly speaking, black (active) points are inside the outer ellipse while grey (inactive) points are outside it. However, it is not true for all points, because we are using a different transformation now and some points will have larger residuals than before while others will be smaller.

Figure 6.3 shows the display after a second pruning step. Now 75% of the original data is still active. The dense cloud of points continues to shrink, and the rectangle and ellipses along with it. The new transformation matrix is

$$\begin{bmatrix} 0.9957 & -0.0844 & -0.0293 \\ 0.0798 & 0.9964 & 0.5550 \\ 0 & 0 & 1 \end{bmatrix}$$

Figures 6.4-6.8 show the effects of five more iterations of the algorithm. You can see the fraction of active points continue to drop, down to 32% in the final one. The central dense cloud gets down to about a one-pixel radius and then stays about that size while the algorithm continues to tighten the ellipses.

At this point, all of the active points have residual vectors less than one pixel in length, so it is time to quit. The final transformation matrix obtained is

$$\begin{bmatrix} 0.9962 & -0.0872 & 0.0161 \\ 0.0865 & 0.9962 & 0.4801 \\ 0 & 0 & 1 \end{bmatrix}$$

This is a rotation of 4.98 degrees, a translation of 0.5 pixel, horizontal and vertical scaling of 0.9999 and 1.0001, and a tiny skew. The translation is the largest error.

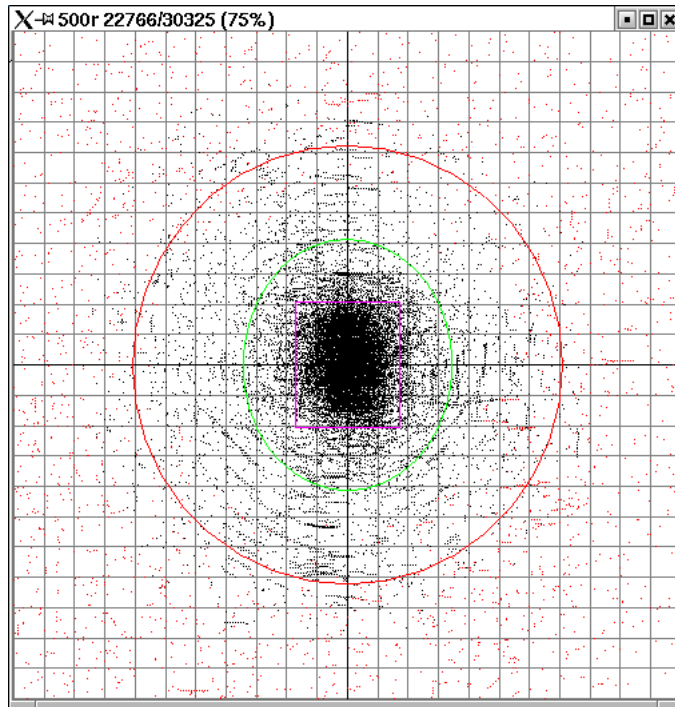


Figure 6.3: Fit: 2 iterations

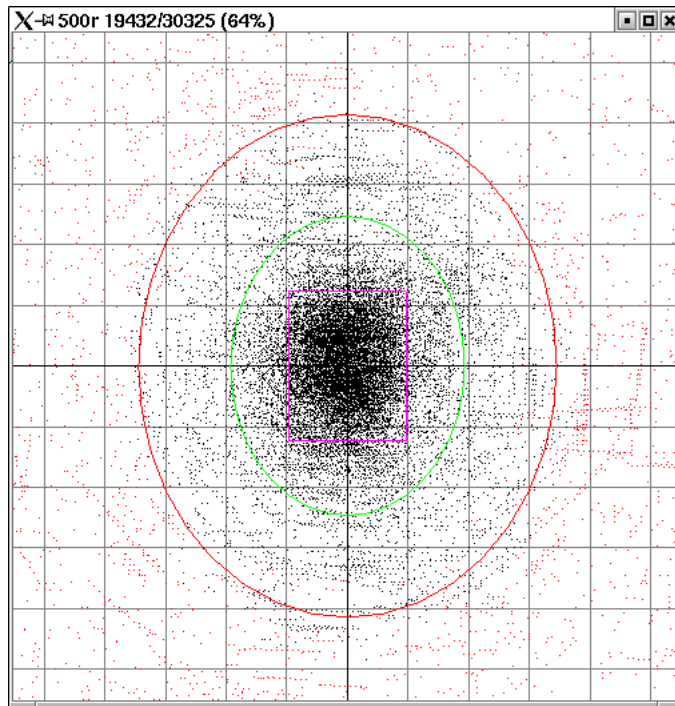


Figure 6.4: Fit: 3 iterations

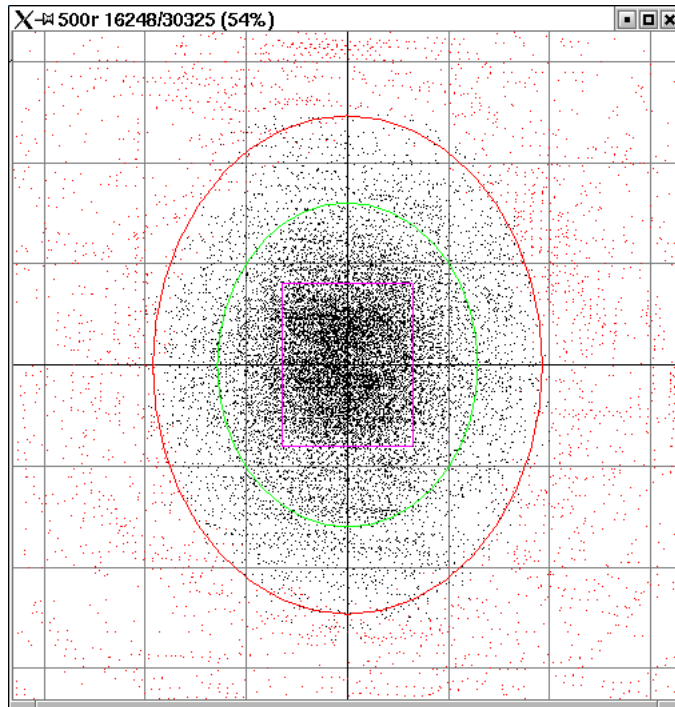


Figure 6.5: Fit: 4 iterations

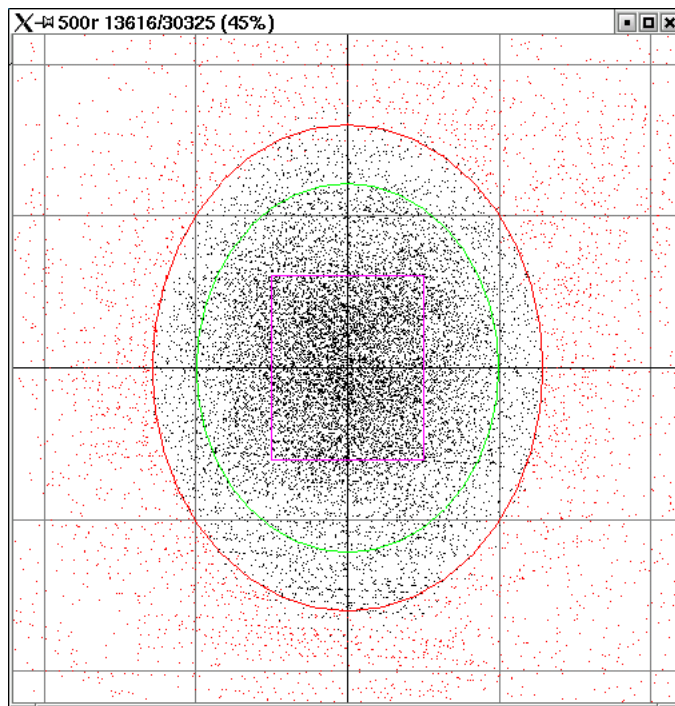


Figure 6.6: Fit: 5 iterations

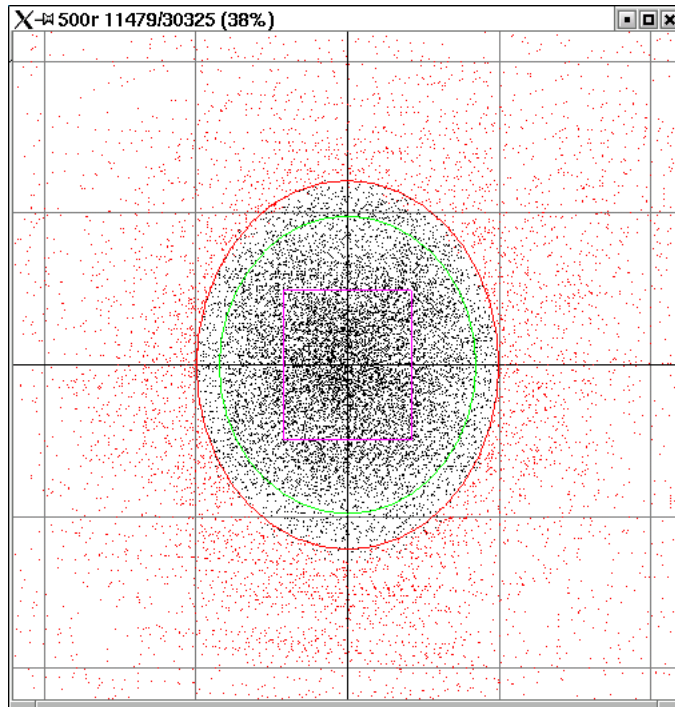


Figure 6.7: Fit: 6 iterations

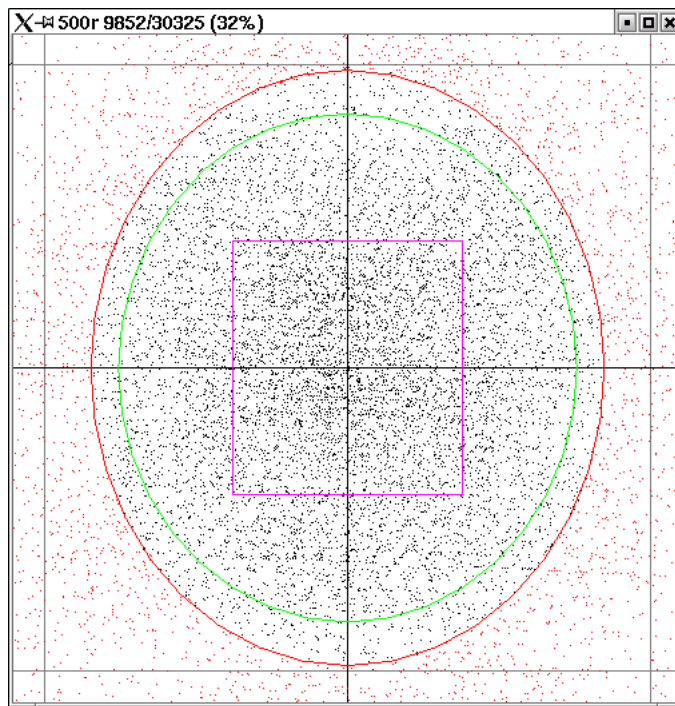


Figure 6.8: Fit: 7 iterations

Modes

The fitting algorithm actually has three modes of operation. The example above showed the results of manually-controlled iteration, fitting match data from all levels simultaneously. The final transformation matrix written out is just the one produced by the final iteration.

In a second mode, the algorithm processes data from only one level of the wavelet transform at a time. The operator looks at the data from only that level, does a number of pruning cycles until the data seems to have converged, then goes on to the next level and repeats the process there. When the operator has finished examining all levels, the program gathers all of the active data from all of the levels and does one final global LS fit based on it. This final all-levels transformation is what is written out.

During the examination of one level, the operator may decide that the data is just too noisy at that level (typically level 1) and that the whole level should be discarded from the fitting process. One command allows doing this. Also, if the LS fitting process discovers a singularity in the design matrix, the whole level is automatically disabled. (This might happen at the highest levels due to edge artifacts and lack of interesting data points.)

The third mode, which is the default, operates without human intervention. It is suitable when the user has some expectation that the data will be well-behaved. Automatic fitting always uses all levels of the matching data. It proceeds until the maximum residual size for the active points drops below a threshold (0.8 pixels), until the percentage of points that are still active drops below a threshold (25%), or until it reaches an iteration limit (15).

6.2.5 Output

Once a transformation matrix has been determined for the active data at all levels, it is normally written out into a file, along with a header that describes the size of images that the transformation applies to.

Optionally, the transformation matrix can be “regularized” first. The general affine transform matrix, which has six degrees of freedom, is converted into the product of two matrices. The first matrix consists of a rotation, uniform scaling, and translation (four degrees of freedom) which is as close as possible to the original affine matrix. The second matrix is a “residual” transformation containing any skew or non-uniform scaling. If this residual matrix is sufficiently close to the identity matrix, then the first matrix is a good model for the transformation, and we write it out instead of the affine matrix. This may be useful when you know that there should be no skew or non-uniform scaling.

6.3 Iterative Multiscale Fitting

So far, we have mostly discussed the various Phases of our method as if they formed a true pipeline, as shown in Figure 3.1, and on the first page of each chapter since. In fact, the method can operate that way, particularly for small images. But for larger images, we must take a different approach to keep the running time moderate.

The fundamental idea is to use feedback in the matching and fitting process. We start at the highest level (largest scale features) in the edge data, and match just that one level. Next we fit a transformation to the matched points. Then we use that transformation to preadjust the edge data from the next lower scale level before matching it. This normally brings the points closer to alignment before the matching, so there are a higher proportion of good matches in the output of Phase 2. This in turn increases the likelihood of Phase 3

converging, and of it providing a better estimate of the actual transformation. This transformation is fed back to adjust the next lower level, and we continue the iteration until we've completed the lowest level.

We can also iterate within one level as well, giving a more accurate estimate of the transformation at that level before proceeding to the next. This normally isn't necessary to get the correct final transformation, so we avoid it to save time.

Figure 6.9 shows how the system data flow is altered when we are using iterative multiscale fitting; compare it to Figure 3.1. Essentially, we have a new "match/fit controller" phase that is inserted into the pipeline in place of Phases 2 and 3. It accepts the same edge data from Phase 1 and supplies the same transformation data to Phase 4 as the replaced phases did, so the rest of the pipeline need not be aware of the change.

6.3.1 Method

The new controller is actually rather simple. We read in all of the data from Phase 1 for both images. We calculate how the levels of the two images must be aligned during the comparison. (This function is described in Section 5.1.2 since it is part of Phase 2 in the pipeline model, but it must be moved to this point in the iterative model). A "current transformation" matrix is initialized to the identity matrix. Then for each level of comparison, starting at the highest, the method does this:

1. Apply the current transformation to the points from image 1 only for the current level only. (Both the point position and the gradient vector are updated.)
2. Write out the (modified) image 1 data and the (unmodified) image 2 data from this level only into two temporary files. The files are constructed to look like Phase 1 output, but have only one level present.
3. Run the Phase 2 matching program on these data files, producing a matching output file that contains only one level.
4. Run the Phase 3 fitting program on that file, producing a transformation matrix file.
5. Read the transformation matrix file. Apply this incremental transformation to the data points from image 1 at the current level.
6. Multiply the old current transformation matrix by this new matrix to obtain a new current transformation matrix. (Multiplying the transformation applied to the data before matching by the transformation obtained from fitting gives us the best estimate to this point of the overall transformation needed to align the images.)
7. If we have not exhausted the iteration count for this level, return to step 2 for another intra-level iteration.
8. If we are not at the lowest level, decrease the current level and return to step 1.

Note that most of the computation is done by the existing Phase 2 and Phase 3 programs. We "trick" them into processing only the set of data that we want by passing only that much to them.

The only substantial computation performed in the controller is adjusting the edge data. At the beginning of each level, we have a transformation matrix that we would like to apply to that image before matching. In

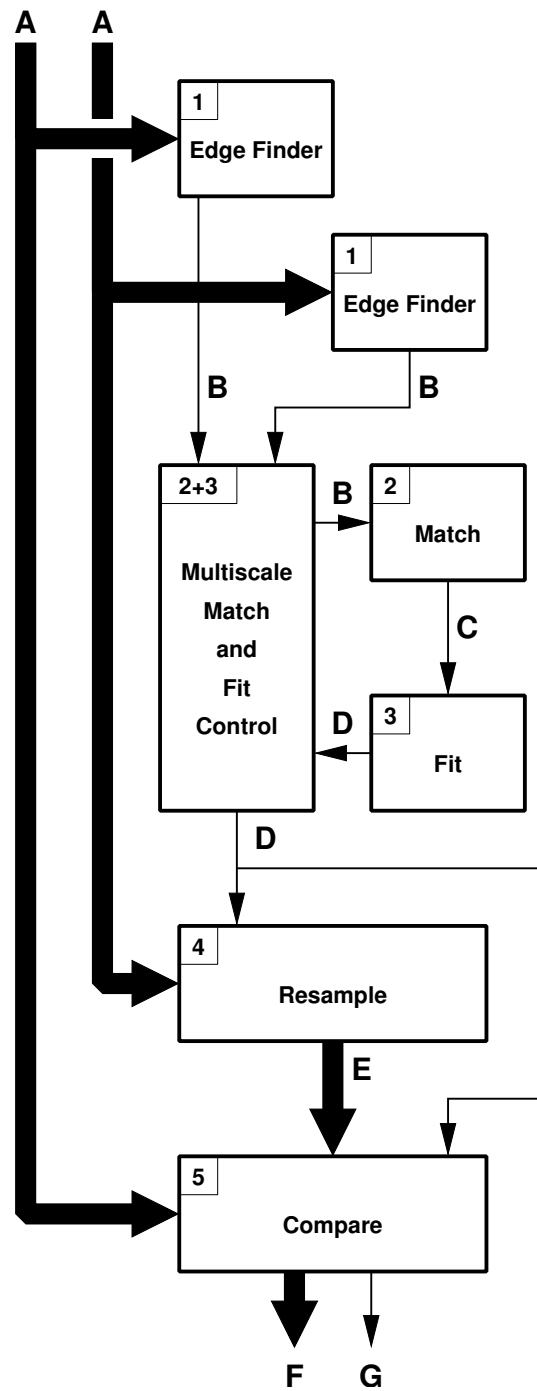


Figure 6.9: Multiscale fitting data flow diagram

concept, we could actually resample the original image according to the transformation (using the Phase 4 code), then pass it through Phase 1 to get new edge data for this level, but that would be quite expensive. Instead, we take the edge data produced by Phase 1 operating on the original unmodified image, and adjust the data to what it would have been if the image had been resampled. This is inexpensive. Similarly, iteration within a level is done by adjusting the existing data using the incremental transformation obtained from Phase 3 fitting.

Moving the location of the edge points is done simply by multiplying by the current transformation matrix. A bit of calculus shows us that when a function is transformed by an affine transform, the gradient at the new location is altered in a way that can be calculated by multiplying the old gradient by the transpose of the inverse of the transformation. (This uses only the upper left 2×2 submatrix, since the gradient is a vector not a point).

If the data came from the Laplacian edged detector instead of the wavelet-based one, we don't have a real gradient vector to transform. Instead, we calculate the effect of the transformation on the length of two unit gradient vectors along the X and Y axes, take the geometric mean of those two size changes, and apply that to each "magnitude" value supplied by the Laplacian edge finder.

The controller algorithm also modifies the matching "window" size as it iterates. The window starts out at the normal size (usually 10% of the image width) for the first pass. Each successive iteration reduces this by a factor of 0.7, until a lower limit of 2% is reached. Thus, the Phase 2 matching is done with a much "tighter" window at the lower levels. This is possible because most of the alignment has already been done at that point.

6.3.2 Synergy

The iterative multiscale method works well. In our tests so far, it has never failed to converge to a good estimate of the transformation in circumstances where the pipeline model did converge. On the other hand, there have been several test cases where the pipelined model did not converge but the iterative model was able to converge to the right answer.

Using this method also allows us to accelerate Phase 1 by decimation between levels. This results in higher-level images that are much smaller than they would be if Phase 1 retained full size all the way through the levels. This saves some time during Phase 1. The smaller data sets at the higher levels also substantially speed up the matching. The resolution of the data is much coarser when decimation is used, but then in iterative fitting the higher levels need only provide approximate alignment. More precise alignment is provided by the higher-resolution lower level data that will be used at the end.

The combination of decimation during Phase 1 and the iterative multiscale version of Phase 2-plus-3 make it practical to work on quite large images. We have tested images up to 2048 pixels square, but in principle even larger images should work.

6.4 Implementation

The simple pipelined version of Phase 3 is implemented by a single program named `fit`.

The SVD algorithm we use is `svdcmp` from Numerical Recipes in C++ [56]. The solution is obtained using the `svbksb` routine from Numerical Recipes, calling it for each column in `x` and `b`.

We use double precision versions of these two routines because of the large number of data points we may have. In "fit all levels simultaneously" mode, we could have up to 100,000 points being fitted. There

is a rule of thumb for least squares fitting using the SVD that says the “tolerance” value (the ratio between the largest singular value and the smallest non-zero one allowed) should be the number of data points times machine epsilon. This is based on the worst-case buildup of roundoff error. In single precision, epsilon is about $1e-7$, so with $1e5$ data points we end up with a tolerance of 0.01. This is large enough to get some data sets treated as rank-deficient when they really are not. By using double precision, with epsilon of about $1e-16$, we will always have a tolerance of $1e-10$ or smaller, which should never be a problem.

The manual-control interface provides an arbitrary number of levels of “undo”, so it is easy for the operator to perform many pruning operations, then back up all the way to the beginning, then redo the pruning again. This is useful when trying to find the best place to stop. If the program is operating in the mode where levels are treated separately, there is a full undo history kept for each level, so you can work on one level, switch to a different level for a while, then return to the previous level and still undo operations there.

The multiscale matching/fitting controller is implemented in a program called `imatch`.

A utility program called `tinvert` calculates the inverse of a transformation file. It swaps the input and output image sizes and inverts the transformation matrix. This is needed because `fit` always calculates the transformation needed to resample the first image to align with the second image, but in some cases it is better to resample the second image to align with the first one instead. This is discussed in more detail in Section 7.2.

6.5 Performance

The current software is subjectively quite fast. Although we may be fitting 50000 data points or more (in default all-levels mode), one iteration of the pruning takes only a fraction of a second. In manual mode, this includes the pruning itself, calculating a new LS fit, calculating new residuals for all points, and refreshing the display of all these things. The operator seldom has a sense of waiting for the computer between iterations.

In automatic mode, the total elapsed time to fit 30325 matches from a pair of 500×500 images using 9 pruning iterations is about 1.9 seconds (PIII 700 MHz).

With iterative multiscale matching and fitting, there are only 18553 matches total across all levels, because of the decimation used in Phase 1. The total fitting time for eight levels of data is 0.76 seconds.

6.6 Discussion

When using iterative multiscale fitting, we have the potential for two different levels of iteration within a single level of detail from the images. The fitting program internally uses iterative pruning to eliminate outliers, while the multiscale controller can iterate the match/fit process multiple times per level. The iterative pruning acts on an existing set of matching data, so it is comparatively fast. Iterating the whole match/fit process is a more powerful method of converging on a solution than the pruning iteration, because the additional matching attempts should create a larger proportion of correct matches as the images become better aligned. However, the matching step is many times as expensive as the internal pruning iteration.

We routinely make use of the iterative pruning method, but usually do not use iterative matching within a level (except during testing) because of its cost.

6.6.1 Alternative Models

Lenses on physical cameras often have some amount of geometric distortion. The most common form of this is barrel or pincushion distortion. It would be useful if we could find and remove this when comparing images from a physical camera to ones from a computer renderer (renderer cameras usually have distortionless “optics”).

Barrel and pincushion distortion are nothing more than a change in magnification that depends on the distance of a point from the optical axis of the lens. If we align the origin of our coordinate system with the lens axis and use polar coordinates for position, it has a particularly simple form. For any normal lens (i.e. not a fisheye or anamorphic design, where distortion is very large but deliberate) we should be able to model the distortion characteristics of the lens with a polynomial. Let us assume a third degree polynomial; the extension to higher degree will be obvious. Then, for any feature in the scene that would have been imaged at a point (r, θ) by a pinhole “lens”, our distorting lens will image it at a point (r', θ) , where

$$r' = r + a_2 r^2 + a_3 r^3$$

Note that θ does not change at all. There is no constant coefficient in the expression, since anything that is right on the optical axis (where $r = 0$) is going to stay there. There is also no coefficient in the first-degree term, since any non-unity value there gives simple magnification, which is not distortion. Only the higher-degree coefficients a_2 and a_3 are adjustable, and they control distortion. If both are positive, we get pincushion distortion. Negative values give barrel distortion. If the coefficients have opposite signs, the behaviour can be more complex.

For our purposes, it is more convenient to convert the expression above into a radial scale function $s(r)$:

$$s(r) = \frac{r'}{r} = 1 + a_2 r + a_3 r^2$$

We can use this scale function to apply the distortion directly in Cartesian coordinates, without converting to polar form and back:

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ u &= s(r)x = x + a_2 r x + a_3 r^2 x \\ v &= s(r)y = y + a_2 r y + a_3 r^2 y \end{aligned}$$

Thus, it shouldn't be difficult to modify the resampling algorithm (Phase 4) to include lens distortion compensation either before or after the affine transformation.

Actually determining the distortion coefficients, though, is best done in polar coordinates. The expression for $s(r)$ is in the form of a polynomial, and should be easily handled using linear least squares methods. However, this would need to be done in a second pass, separate from the affine transform fitting. We have not attempted to do this yet.

In many situations involving a physical camera, it would likely be more practical to calibrate the camera for lens distortion in advance by photographing a test chart with straight lines, then (automatically or manually) finding a good fit for the distortion parameters a_2 and a_3 . Once this information is obtained, all images from that camera and lens could have their distortion removed in a resampling step before the image is passed to Phase 1. In other words, it is easier to measure distortion under controlled conditions, when it isn't confounded with all of the other variations between a pair of images.

The assumption of an affine transformation is probably the Achilles heel of this thesis as a whole. There are many differences that can occur between a real and synthetic image which simply can't be modelled by an affine transformation. For example, a difference in camera position (other than rotation around the optical axis), or modelling errors that cause small shifts in object location, produce non-affine changes to the image. An obvious area of further investigation is the use of more general transformation models. For example, remote sensing applications often use a pair of bivariate polynomials to specify the mapping. A spline mesh could also be used, giving the advantage of local control (adjusting one control point affects only a small number of local spline patches, not the entire image). These allow areas of the image which are distant from each other to be reshaped in different ways by the transformation, while still maintaining continuity of the transformation.

However, these more general transformations are usually fitted to "clean" datasets of matching points. It isn't clear whether or how well they would work with the output of the automated Phase 2 matching. With many more free parameters to fit, there is much more freedom for the transformation to adapt itself to erroneous data. Once it has done that, it may be difficult or impossible to use a method based on residuals to prune the bad data.

Also, more general transformation models, either global or local, are more likely to have nonlinear equations that cannot be handled using linear least squares methods. There are more general nonlinear least-squares algorithms available (e.g. Levenberg-Marquardt [36] [43]), but they are considerably slower and may not find the best possible solution if started far from it. It is not clear that interactive pruning would still be practical with a nonlinear least-squares fitting algorithm.

6.6.2 Alternative Least-Squares Methods

At one point, we experimented with using the "normal equations" to solve the system instead of building the design matrix. The normal equations method scans through the data building a covariance matrix. Then a system of equations needs to be solved, but the system is only 3×3 in size. This is very very fast.

Unfortunately, the normal equations method has a worst-case error proportional to the square of the condition number of the matrix, much worse than using SVD with the design matrix (where error is proportional to the condition number when the residual is small). Using normal equations in double precision yields the same accuracy as the SVD in single precision, but we are already using double precision for the SVD. With a suitable compiler and library we could use quad precision (128 bit), but few machines have hardware support for this.

In addition, the SVD can be relied upon to detect cases where the design matrix is nearly singular, and do something reasonable. The normal equations method will not necessarily detect a problem, and can produce very wrong answers because the condition number of the problem is so high.

Normal equations are a better choice when the number of data points being fitted are moderate and double precision arithmetic is adequate. They would be fine if the problem construction guaranteed that the covariance matrix was never close to singular. And normal equations are very useful in applications like real-time tracking where the answer is needed very rapidly.

On the other hand, we have a large number of data points, the design matrix can be near-singular, and we do not need anything faster than the SVD. The automatic-fitting time of a few seconds is completely overshadowed by almost every other phase of processing. In the manual-fit case, the user isn't aware of any significant delay.

There is yet another standard method for solving LS problems, using the QR algorithm. It begins with

the same \mathbf{A} and \mathbf{b} matrices as the SVD method, but the \mathbf{A} matrix is decomposed into two matrices \mathbf{Q} and \mathbf{R} instead of the three matrices produced by the SVD algorithm.

The QR technique is just as accurate as the SVD technique, and it is faster than SVD. However, it can't handle a design matrix that is singular or nearly so, and this can happen with our data. The SVD method handles the occasional rank-deficient matrix without difficulty, and we really don't need the extra speed.

6.6.3 Techniques Related to Pruning

The iterative pruning technique described in Section 6.2.4 attempts to find and remove most of the poor matches from the data so the least squares (LS) fit is based primarily on the good matches. It turns out to be similar to a couple of techniques in the "Robust Regression" statistics literature.

Rousseeuw and colleagues [60], [61], [62] have described the Least Median of Squares (LMS) and Least Trimmed Squares (LTS) methods. Recall that standard least squares solves for the value of the \mathbf{x} vector that minimizes the sum of the squares of all of the residuals. The LMS method attempts to find the value of the \mathbf{x} vector that minimizes the median of all of the residuals. As a result, it ignores large residuals for up to 50% of the input data. (It doesn't necessarily minimize the residuals of the data points with smaller than median residuals either.) A variant, called Least Quantile of Squares (LQS), minimizes the k th largest residual, where k is usually between $0.5N$ and $0.75N$. In contrast, the LTS method attempts to minimize the sum of the squares of the k smallest residuals instead of just the k th one.

These are both called *positive breakdown* estimators, because the fit is unaffected by the introduction of some fraction of non-representative data points to the dataset.

The authors' first implementation of these techniques is in an algorithm called PROGRESS. It begins by selecting a large number of subsets of the data at random. Each subset has as many data points as the length of the \mathbf{x} vector (i.e. the number of unknowns) and solves the system exactly for those data points. Then the residuals of all data points are calculated and the value of the objective function (the k th smallest residual for LQS, the sum of the smallest k residuals for LTS) is saved. When sufficiently many subsets have been tried, the one that gave the smallest value of the objective function is declared to be the solution. However, PROGRESS is slow. With a 3-variable fit and tens of thousands of data points, it would find 1500 subsets of 3 points, calculate residuals for all data points, and calculate the objective function for each of these sets of residuals. (LMS estimators based on these random sampling techniques are built into the commercial S-Plus and SAS/IML statistical packages.)

This method might work for our data. We have only three unknowns to fit (at one time), so PROGRESS would select 3-tuples from the data. Given enough random samples, there would be a good chance that at least one of them would have three "good" matches, and calculate the right transformation. If there were 50% or more "good" matches in the dataset, then either the LQS or LTS objective functions would be very small (on the order of 0.5 pixel) if $k = N/2$. However, if only 20% of the matches are "good", minimizing the median residual or the sum of the smallest half of the residuals would likely not give as good a fit as one based on looking at only the smallest 20% of the residuals. Our technique is capable of concentrating on that 20% and ignoring the other 80%.

The RANSAC algorithm [19] is another approach to finding a transformation when the fraction of good data is substantially less than 50%. We discussed its operation in Section 2.1.5. RANSAC is principally different from PROGRESS in that it stops when it finds the first adequate solution rather than always trying many solutions and keeping the best. RANSAC has a number of tunable parameters that need to be set based on some knowledge of the input data.

In 1999, Rousseeuw and Van Driessen introduced FAST-LTS to handle LTS fitting of larger data sets more rapidly [62]. (It handles LTS only, not LMS/LQS.) The key to the new algorithm is the “C-step” (C stands for Concentration). It takes as input a subset (k in size) of the dataset with N points, and calculates a new subset of the same size which is guaranteed to have a LS fit with equal or smaller residuals. It operates in this way:

1. Perform LS fitting for the unknowns on the old subset of data points.
2. Compute the residuals for all data points with these fitted values.
3. Sort the absolute values of the residuals into ascending sequence.
4. Define a new subset consisting of the data points with the first k sorted residuals.
5. Calculate the sum of the squares of the residuals of the points in this subset; this is our objective function.

The C-step is iterated until the objective function stops changing. Since there are a finite number of k -subsets of the data, we must reach this point eventually. The authors note that it often takes fewer than 10 iterations in practice.

Note how closely the C-step matches the inner loop of our own algorithm. The C-step’s subset corresponds to our active point flag, which also defines a subset of the initial data points. LS fitting is based on the subset only, while residuals are calculated for all data points. The only difference is that in the C-step, the fraction k/N of the data points to select is predetermined, and remains the same for each iteration, while our method tends to reduce k at each stage by eliminating points whose residuals are sufficiently larger than the standard deviation of the currently-active subset.

The authors of FAST-LTS note that when the C-step iteration stops converging, the minimum found is only a local minimum, not necessarily a global minimum. To make it likely to find a global minimum, the algorithm selects multiple k -subsets of the data and lets the C-step converge for each of them, then selects the one with the smallest objective function. These are found as in PROGRESS: select subsets with as many points as unknowns, find the exact fit to those points, and calculate residuals for all data points from this fit. The difference is that, rather than merely evaluating the objective function and going on to the next random subset, FAST-LTS selects the k points with the smallest residuals and then iterates the C-step until convergence. So now we don’t need to have one random selection that falls exactly at the location of the global minimum in order to find that minimum, which was the case in PROGRESS. All that is necessary is for one of the random selections to fall in the same “valley” in the objective function that contains the global minimum, and the C-step will find it by iteration.

FAST-LTS also saves time by stopping after only two C-steps if the objective function is not converging rapidly for a particular subset. Thus, it spends its time refining subsets that are probably uncontaminated, not ones that are likely contaminated. Still, to keep the cost down, the algorithm manages a complex system of nested subsets of subsets when N is large.

Our own algorithm does not attempt to use multiple subsets. It starts with a fit of the entire dataset and then reduces the subset size based on residuals. It is certainly possible that our approach could fail to converge, or converge to an inappropriate answer, if there are too few “good” matches among the data. In that case the FAST-LTS approach seems like a good solution.

However, in our experience, this has not been a problem. In cases where there was a global affine transformation that did model the difference between the images, our method of starting with the entire dataset has always found it. In cases where our method did not converge (it was obvious from the graphical display of residuals that it had not), the real problem has been that the difference between the two images was not representable as an affine transformation. So, if we want to improve the performance of Phase 3, we need to look first at using more general models.

There is another technique named Iteratively Reweighted Least Squares (IRLS) [28]. Instead of all residuals contributing equally to the objective function that we are trying to minimize, each data point has a weight associated with it. The larger the weight, the more influence that point has in the LS solution. The weights are initially set to unity. On each iteration, a weighted LS fit is done using the current weights, and the residuals are computed. Then each point gets a new weight calculated by applying a reweighting function to the point's residual. Then we go back to the top of the loop for another LS fit. The process stops when two successive residuals sets are nearly the same.

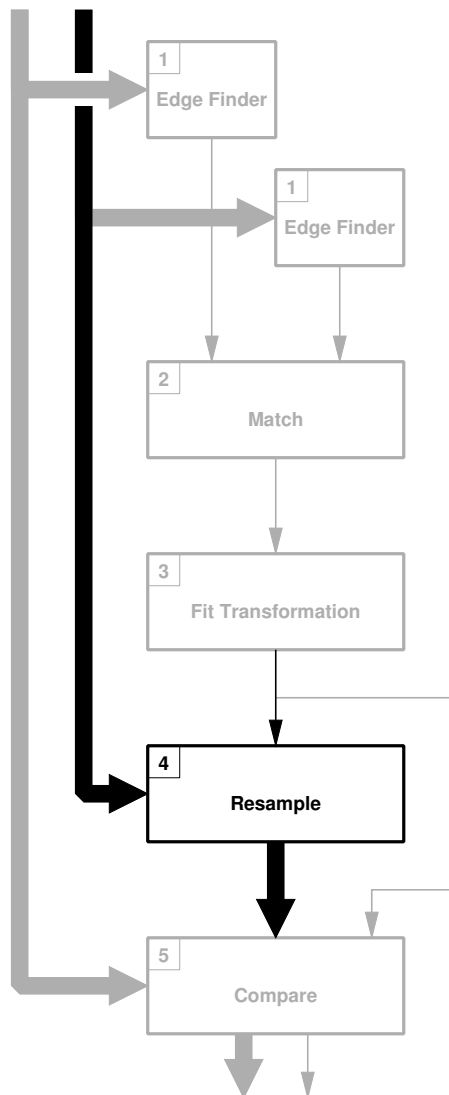
There are several different reweighting functions suggested for this algorithm. As you would expect, all have the property that small residuals produce larger weights, and larger residuals produce smaller weights. In some cases, there is an abrupt cutoff. The "bisquare" function returns zero (effectively removing a point from the LS fit entirely) if the residual is more than a certain threshold larger than the median residual. Smaller residuals return a value whose size depends on the residual.

The "Talworth" reweighting function always returns either zero or one. Effectively it excludes data points with large residuals and includes points with small residuals, without attempting to weight them further. The boundary between the two cases is set at a constant times the median residual.

The significant difference between IRLS using the Talworth function and our own pruning algorithm is that we use a threshold based on the standard deviation of the residuals, one that is calculated only from the active points, not all points.

Chapter 7

Phase 4 — Image Resampling



Phase 3 has determined, as well as it can, an affine transformation that maps points in one image to their corresponding locations in the other image. The task of Phase 4 is to actually “resample” one of the original images according to this transformation, creating a new image which is the same size as the other original image, and which is now aligned with it.

The resampling process itself should be done as accurately as possible, retaining as much information from the original image while minimizing the addition of any errors or artifacts. To the greatest extent possible, we want any differences between the images found in Phase 5 to be due to real differences between the original images, not side effects of the resampling process.

7.1 Background

Nothing in this chapter, or the software it describes, is novel. The actual process of resampling an image in order to warp, rotate, or resize it has been extensively studied, and there is a great deal of literature describing it. All of our filters have been taken from that literature. There is an introduction by Ken Turkowski in *Graphics Gems* [72]. Wolberg’s book *Digital Image Warping* [79] provides more extensive coverage of the subject. In this chapter, we will assume that the reader already has a working knowledge of standard image resampling, and we will only discuss what is unusual about our particular method or implementation.

We will begin, however, with a whirlwind tour of sampling and resampling, to introduce terminology and context. For more detail, see [53] or [55].

7.1.1 Sampling

The real world is continuous, with nearly infinitely small detail visible in any direction (if you have the optics capable of resolving it). Similarly, computer-modelled worlds sometimes have infinitely fine detail, in this case not even limited by the wavelength of light. A digital image of either of these things, on the other hand, is a discrete set of samples of the light reflected or emitted by a scene toward the camera. The location of these samples typically forms a rectangular grid, though other sampling patterns are possible. When we move from the continuous to the discrete, information is necessarily lost. The Nyquist sampling theorem says that the highest spatial frequency that can be faithfully reproduced under perfect conditions is 0.5 cycles per pixel. This applies in the horizontal and vertical directions; the limit is a factor of $\sqrt{2}$ greater in the diagonal direction. Information at spatial frequencies higher than this is not merely lost in the sampling process (that would be acceptable) — it is aliased to lower frequencies. Aliasing produces the appearance of image content that is really not present in the original continuous image.

To avoid aliasing, the original scene must be low-pass filtered before the sampling step to remove the troublesome fine detail (high spatial frequencies) that would be aliased when sampled. The filter that does this is called the *sampling prefilter* or *anti-aliasing filter*.

In the case of images acquired by a digital camera, the lens itself act as an initial low-pass filter. Sometimes a special crystalline filter with a sharper low-pass response is used between the lens and sensor. The CCD or CMOS sensor photosites have non-zero area and also provide some averaging that reduces their high frequency response. (Note that all of these filters are physical devices operating on light in the continuous domain.)

Ideally, the combined effect of all of these would be to reduce the response of the system to zero for frequencies above the 0.5 cycles/pixel limit, while maintaining near 100% response to all lower spatial frequencies. In practice, this is impossible. If all frequencies above the limit are reduced to insignificance, and there is no chance of aliasing at all, some lower frequencies will be attenuated and the image may appear less than fully sharp. Less low-pass filtering yields sharper images, but at the risk of aliasing artifacts. Some professional cameras allow the user to remove the crystalline low-pass filter so they can make their own tradeoff between two different filter responses.

With computer-generated imagery (CGI), anti-aliasing is accomplished by a variety of methods. Sometimes, no filtering is done at all — each pixel is calculated by shooting a single ray into the scene and finding the (infinitely small) point or points in the scene which affect that ray. Sometimes, multiple rays per pixel are shot and averaged, providing some low-pass filtering, along with some noise. Sometimes a pixel is treated as a square with area, and all of the polygons which appear inside this square contribute colour according to their area; this is equivalent to filtering with a “box filter” before point sampling. None of these approaches could be described as a good low-pass filter, and so computer-generated images usually have some aliasing artifacts when high-frequency, high-contrast detail is present in the modelled scene.

No matter how the image is acquired or generated, if aliasing has occurred then the alias of the high-frequency detail appears as some frequency below the 0.5 cycles/pixel cutoff. It becomes indistinguishable from true image information, and cannot be separated later.

Looking at sampling in the frequency domain, instead of in the spatial domain, the original image has a spectrum that extends off to infinity in all directions. The anti-aliasing filter attenuates all the high frequencies, leaving only frequency content that is within a box around the origin. The limits of the box are at ± 0.5 cycles/pixel in both X and Y. The act of sampling the prefiltered image replicates this spectrum across the frequency plane, with copies of it centred at every integer (x, y) location. (These copies are

called *images* of the original spectrum.) However, if the anti-aliasing was done properly, none of the copies overlap. When the sampled image is converted back to continuous form, the original (filtered) spectrum can be separated from all of its images. If the anti-aliasing did not remove frequencies above 0.5 cycles/pixel, then some of the high frequencies belonging to adjacent images of the spectrum will overlap the original spectrum centred at $(0, 0)$. This overlap of spectrum content is aliasing.

7.1.2 Resampling

Now, suppose we want to warp an image to align it with another. Or perhaps we want to make it larger or smaller, or rotate it. For any of these operations, we can easily describe the mapping from the source image to the new image using a mathematical function, usually a continuous one. Once we've done that, for each output image sample point location (i.e. for each output pixel) we can easily calculate the coordinates of the corresponding point in the input image. Unfortunately, this generally won't be one of the discrete points for which we already have a sample value. What we would really like is to go back and sample the original scene with a new grid of sample points that are not aligned with the original sample points.

Unfortunately, we don't have the original scene any more; we only have a frequency-limited and discretely-sampled version of it. The best we can do is convert the discrete version of the image back into a continuous approximation to the original scene, and then sample this approximation at the new sample grid points to give a new discrete version of the approximation. This process is called resampling.

The first step is to create a continuous 2D function from the discrete image. This is done by passing the sampled image through a *reconstruction* or *anti-imaging* filter. This filter needs to pass the spectrum of the original image, which is located near the origin in the frequency domain, with as little attenuation as possible. At the same time, the anti-imaging filter must strongly attenuate all of the other images of the spectrum that were introduced by sampling. Thus, we want a filter whose gain is near 1.0 from zero up near 0.5 cycles/pixel, but whose gain is near zero at any frequency above 0.5 cycles/pixel.

Once the anti-imaging filter has given us a continuous function, resampling proceeds like sampling. We first use an anti-aliasing filter to remove any frequencies that are too high for the new sampling grid. Then we can sample the filtered image, giving us a new discrete image.

In concept, we always use both the anti-imaging filter and the anti-aliasing filter during resampling. Each is a low-pass filter with its cutoff frequency near 0.5 cycles/pixel. However, since the input and output sample spacings are (in general) different, and each filter's frequency limits are relative to the sample frequency in a different image, the absolute frequencies involved are different. For example, if the input image is 1000 pixels wide, the maximum representable frequency is 0.5 cycles/pixel or 500 cycles per image width. This is the cutoff frequency for the anti-imaging filter. If the output image is to be 700 pixels wide, the maximum unaliased frequency in the output is 350 cycles per image width (0.5 cycles/pixel in output pixel terms), and the anti-aliasing filter must remove any high-frequency content above that limit.

However, it is usually possible to eliminate one of these two filters. The effect of applying two successive filters in the spatial domain can be calculated by multiplying their frequency responses in the frequency domain. The result is (nearly) equal to the narrower of the two original filters. This tells us that we can simply use the lower-cutoff filter alone. If the image is being enlarged (the output sample points are closer together than the input sample points), the frequency cutoff for the anti-aliasing filter is higher than that of the anti-imaging filter. Since a properly-designed anti-imaging filter won't generate or pass any frequencies above its own cutoff, the anti-aliasing filter does nothing useful, and we can eliminate it. The resampling process becomes one of reconstruction by the anti-imaging filter followed directly by sampling.

On the other hand, if the image is being reduced in size, the anti-aliasing filter has a lower cutoff frequency than the anti-imaging filter. We can ignore the anti-imaging filter, since the anti-aliasing filter will also perform the reconstruction and anti-image filtering that is needed.

Thus, in many cases, we can decide in advance that we need only one of the filters. On the other hand, sometimes we need both. If the transformation is a polynomial-controlled warp, some regions of the image may be enlarged while others are reduced in size, so we have to switch between the two filters depending on where we are in the image (or always use both). If we're rotating the image by 45 degrees but are keeping it the same size, the anti-imaging filter and anti-aliasing filter both need the same cutoff frequency, but their box-shaped 2D frequency responses need to be rotated 45 degrees relative to each other. In this case, we theoretically ought to use both filters, and end up with a frequency response whose passband is octagon-shaped. (More practically, we could use a filter with a circular passband instead of octagonal, and ignore the slight loss of high frequencies.)

7.2 Choosing the Image to Resample

If we label the two original images A and B , and present them to Phase 2 (matching) in that order, then the transformation calculated by Phase 3 will always describe how to transform image A into a new image \hat{A} which should align with image B . However, we could equally well use the inverse of the transformation to resample image B to give a new image \hat{B} that aligns with image A .

In practice, we always want to choose whichever option involves scaling an image up in size. When an image is enlarged, the information in the original image is sampled more densely and (in theory at least) no information is lost by this process. The new image has the capacity to represent additional fine detail that is simply not used, since this detail is lacking in the original smaller image.

On the other hand, when an image is reduced in size, the information is sampled more coarsely, and there is usually some fine detail in the image at spatial frequencies which are above 0.5 cycles per pixel at the new sample rate. This information cannot be represented accurately at the new sample rate; if it is not removed from the image during the resampling process it will be aliased to a lower frequency and appear as false image content. Normally, resampling employs an anti-aliasing filter to remove most of the troublesome fine detail so it cannot be aliased. However, for our purposes, both possible outcomes are undesirable. If the anti-aliasing filter removes fine detail from the image, we have lost some of the real information content of the original image, which may cause us to declare the pair of images to be closer than they really are. If any high-frequency content remains and is aliased to a different frequency during resampling, we have introduced a difference (or a fake match) that is not present in the original image.

The only way to avoid these problems of size reduction (downsampling) is to avoid reduction, and instead apply the inverse transformation to the other input image. This necessarily involves enlarging (up-sampling) instead.

We can determine which image to resample by looking at the first two columns of the transformation matrix from Phase 3. The first column, interpreted as a vector, tells us what the horizontal unit vector $(1, 0)$ in image A maps to in image B . Similarly, the second column of the matrix tells us what the vertical unit vector $(0, 1)$ in image A maps to in image B . If the magnitude of both of these vectors is greater than 1.0, then the transformation enlarges the image in all directions, and we choose to resample image A to produce A' . On the other hand, if both vectors are less than 1.0, then the transformation reduces image scale (downsamples) in all directions. In this case, we choose to resample image B to produce B' using the

inverse of the calculated transformation. This now involves only upsampling.

If both vectors have a magnitude of exactly 1.0, then the transformation involves no resizing at all, and it doesn't matter which image we choose to resample, so we pick A arbitrarily. If only one vector has a magnitude of 1.0, then we use the size of the remaining vector to determine which image to resample, just as if both vectors were larger or smaller than 1.0.

In practice, all of the policies above can be implemented by simply calculating the magnitude of the two vectors, determining their geometric mean, and then choosing to resample image A if the result is greater than or equal to 1.0 (selecting image B otherwise). This works whenever we know that we have one of these three cases.

Finally, it is possible for one vector to be larger than 1.0 while the other is smaller. This means that the transformation involves upsampling in one direction and downsampling in the other direction, which may or may not be a problem. For the images we expect to work with, the X and Y scale factors should be very close to each other, within a few percent at most, and if this is true then both vectors must be close to 1.0 if their actual values bracket the value 1.0. If we use the "geometric mean magnitude" criterion in the previous paragraph, we will choose the direction of resampling that keeps the downsampling factor as close to 1.0 as possible. If this is in fact within a few percent of 1.0, then there are likely to be no visible unpleasant side effects from the downsampling, and we can safely ignore the problem.

On the other hand, if the two vector magnitudes are substantially different, this may indicate a more complex problem. It is possible that one of the images was sampled at different rates in X and Y , giving pixels with a non-square aspect ratio (this is common in digitized video images). Such an image will require substantially different scale factors in X and Y to align with a square-pixel image. The best way to handle this is to first resample the image with non-square pixels in one direction to make the pixel aspect ratio square, or approximately so. The resampling direction should be chosen so that the operation is upsampling, not downsampling.

For example, digital NTSC video cameras provide a 720×480 pixel image although the overall image has an aspect ratio of 1.33. This is because the vertical sample pitch is about 13% larger than the horizontal sample pitch in these cameras (for historical reasons). To compare an image from such a camera to a CGI image, or even to rotate the video image without distortion, the video image must be resampled to have square pixels (or, more accurately, equal horizontal and vertical sample pitch). This can be done by resampling the image to either 720×540 or 640×480 pixels. The former involves upsampling, and will not lose image information. The latter involves downsampling, and should be avoided.

After applying the inverse of this scale change to the transformation matrix, the result is used to control a more general resampling step. Generally, this will mean that both original images will end up being resampled to give \hat{A} and \hat{B} , but downsampling will be avoided.

Since we always arrange to upsample our images in Phase 4, we can make use of a resampling algorithm that provides only an anti-imaging filter; we don't need an anti-aliasing filter. (We also assume that any rotation we do will be small, so we don't need to worry about the 45 degree rotation case mentioned at the end of the previous section.)

7.3 Coordinate System

Most of the time, digital images can be regarded as a 2D or 3D array of pixels, with the location of each pixel being described by two or three integers (the array subscripts). We may be vaguely aware that these

pixels are supposed to represent samples from a continuous domain, but we can often ignore this and treat pixel locations as integers. In resampling, we convert from the discrete to the continuous and back to the discrete using a different sampling grid. Sample point locations are real numbers, not integers, and we need a well-defined consistent correspondence between the integer pixel coordinates and the real sample point locations in order to do the math.

The simplest convention is to say that images are composed of nothing but samples, and that a pixel is nothing but the value of a sample. In particular, pixels have no area of their own. It is then natural to specify that the sample point locations occur at integer addresses in the real-number address space. (For example, adjacent samples would be at X addresses of 0.0, 1.0, 2.0,)

There are a number of consequences of this: If an image file is 100 pixels wide, this means we really have 100 point samples that actually span a width of 99 times the pixel spacing. Continuous-domain X addresses within the image fall in the range $[0, 99]$. If we decide to upscale the image by a factor of exactly 2.0, meaning the new sample locations are half as far apart, we will produce an image with X addresses in the range $[0, 198]$, and the new image will have 199 samples per row — not 200.

In addition, when we double the image scale like this, $1/4$ of the new sample points are co-located with the input image sample points, and we can simply use the input pixel values (provided we are using a reconstruction filter function that interpolates the original data). The remaining $3/4$ of the new points must be calculated, but their locations are all either collinear with the input sample point rows and columns, or equidistant between two such rows or columns. The first and last output rows and columns of the new image are coincident with the first and last rows and columns of the input image.

This convention is mathematically simple, but it does not correspond well with many computer graphics input and output devices. With these devices, a pixel occupies a finite area — on screen, on film, in a CCD sensor, etc. Often such devices can operate at multiple resolutions, provided by splitting or combining pixels in integer ratios. For example, some CCDs provide lower resolution by “binning”, where the charge from an $N \times N$ group of sensing elements is combined before A/D (analog to digital) conversion. Frame buffers may provide hardware “zoom” by dividing pixel clock rates by some integer factor N , making a single value in memory occupy N times as much width and height on screen. As the pixel size changes, the location of the visual centre of the pixel also changes, and it is necessary to change the location of the continuous-domain sample points to avoid image shifts with resolution change.

For these reasons, it is sometimes better to use a different model. In this model, we assume that pixels actually have width and height. They are usually, but not always, square. Each pixel is associated with a point sample of a continuous function, and the location of the point sample is in the exact centre of the pixel’s area. The usual convention for assigning real addresses to these features is to declare that the boundaries between pixels occur at positions which are integers, while the sample point locations occur at integers plus 0.5.

In this convention, if an image is 100 pixels wide, it occupies an area that is 100 times the pixel spacing (not 99 times). Continuous-domain X addresses within the image are in $[0, 100]$. The sample point locations are 0.5, 1.5, 2.5, ..., 99.5. If we decide to upsample the image by a factor of exactly 2.0, the output image will be 200 pixels wide (not 199). None of the output sample locations will be coincident with the input samples; the output sampling grid rows and columns are aligned $1/4$ and $3/4$ pixel distance between the rows and columns of the input image sample grid. Thus, reusing input pixel values is not possible.

Despite the slightly more complex addressing, we use the latter convention for pixel and sample addresses. This seems to be consistent with the OpenGL [67] model of pixel and sample location. It is also

consistent with the way Paul Heckbert's `zoom` program [27] does pixel computations. The code for `zoom` describes the difference between pixel location and sample location using two different coordinate systems that are offset by a half-pixel, rather than declaring that samples are located at integers plus 0.5, but the effect is the same.

Of course, there are yet other consistent ways of describing the mapping between integer and continuous coordinates and the issue of image dimensions. Any definition can yield the correct results during resampling, as long as the code uses it consistently.

In this thesis, we make one modification to the convention described above: we put the origin of the coordinate system in the centre of the image. Thus, our 100-pixel-wide sample image would actually have continuous-domain addresses in $[-50, 50]$, not $[0, 100]$. This causes the programs that work with coordinates to do a little extra work, but makes it much easier for a human user to specify a rotation around the centre of the image. It also make the rightmost column in an affine transformation matrix indicate the translation of the centre of the image instead of the translation of the upper left corner, which is useful. All of the thesis software which deals with transformation matrices is consistent in this interpretation.

7.4 Filters

The heart of a resampling program is the anti-imaging (or anti-aliasing) filter used. Almost any image resizing tool provides a few standard filters, and our resampling tools include them too.

The simplest interpolation method of all is “nearest neighbour” resampling, because the method simply copies whichever input sample is closest to the correct position for each output pixel. No filtering is done at all, when either enlarging or reducing image scale. Nearest-neighbour resampling almost never gives a faithful reconstruction of the image, and often looks terrible. The output has abrupt discontinuities in intensity, and features can shift by up to one-half pixel from where they should be.

A closely related method is interpolation using a “box filter”. The box filter is simply a square one unit high, with a width of one unit, centred around the origin. The version for 2D filtering is a cube with all edges one unit long. The same method is also sometimes called zero-order polynomial interpolation or zero-order spline interpolation; several different more general interpolation methods have this as their degenerate case. When upsampling, the box filter is identical to the nearest-neighbour method. When downsampling a box filter will average several input pixels to calculate one output pixel, but nearest-neighbour continues to copy just one input pixel to each output pixel.

Figure 7.1 shows the box filter. The graph on the left is a plot of the single-variable function, while the one on the right is the 2D separable box filter kernel formed by applying the function on the left in both directions. (The lines on the surface below the 2D filter are contour lines.)

In the frequency domain, the frequency response of the box filter is a sinc function, which has prominent sidelobes at high frequencies that are supposed to be blocked. It is certainly not suitable for our use, which is resampling images that will be compared.

The next simplest method is bilinear interpolation. One way of implementing it is to find the four input pixels that surround the location of the current output pixel (mapped back into input image coordinates), then use the position of the output point within the square of input points to linearly interpolate first horizontally and then vertically from the four neighbour points. This is referred to as using a triangle filter or a tent filter, because the linear interpolation process in 1D is equivalent to convolution with a triangle of height one and base width two. It may also be called first-order polynomial interpolation or first-order spline

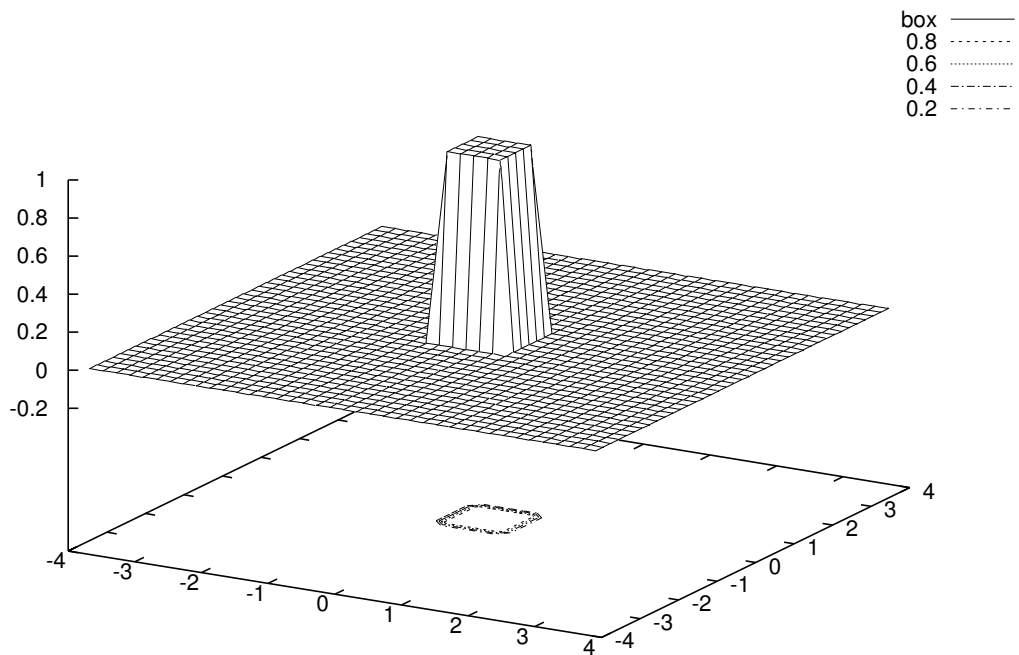
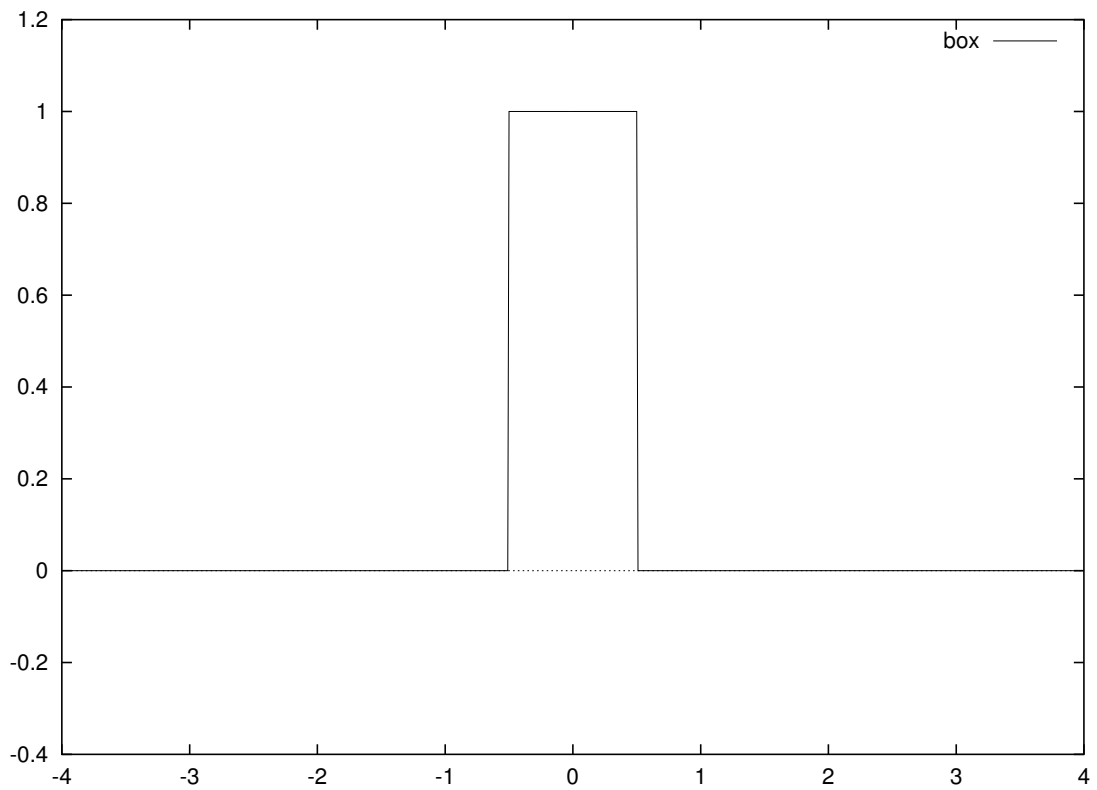


Figure 7.1: Box filter in 1D and 2D

interpolation. (It is also worth noting that the triangle filter is simply the result of convolving two unit box filters.) Figure 7.2 shows the triangle filter.

Linear interpolation performs much better than a box filter or nearest-neighbour. The result of applying the reconstruction filter is now continuous (C^0 continuity). However, the first derivative (as well as all higher orders) is discontinuous across input image row and column boundaries, and this is sometimes visible in the output. (The human eye is a good detector of a discontinuous first derivative of intensity.) The apparent position shifts of nearest-neighbour are gone. The sidelobes in the frequency response are much better attenuated. However, this still isn't good enough.

You might expect the next filter to be based on some sort of quadratic. However, even-degree piecewise polynomials do not lend themselves well to creating interpolation kernels, and they are essentially never used. (The basic problem is that an even-degree polynomial system is supported on an odd-length interval, so the knots between the polynomials occur at integer plus one-half positions, not integers.)

The next common interpolation technique is called “cubic convolution”. The filter kernel is

$$h(x) = \begin{cases} (\alpha + 2)|x|^3 - (\alpha + 3)|x|^2 + 1 & 0 \leq |x| < 1 \\ \alpha|x|^3 - 5\alpha|x|^2 + 8\alpha|x| - 4\alpha & 1 \leq |x| < 2 \\ 0 & 2 \leq |x| \end{cases}$$

This is basically a cubic spline approximation to the sinc function. We assume that the filter kernel will be formed from four segments, each one unit long, over the domain $[-2, 2]$. Further, it must be symmetric, so there are really only two segments to define. To force the reconstruction convolution to interpolate the original input data samples, we require $h(0) = 1$ and $h(x) = 0$ for $x = 1$ and 2 . We also require that the first derivative $h'(x)$ be continuous at $x = 0, 1,$ and 2 . Solving these constraints yields the polynomials above, which still have one free parameter α . Because of the constraints, the reconstructed function will have C^1 continuity (for any value of α).

There are several possible choices for α . If we select $\alpha = -0.5$, we get the best approximation to the original function in the sense that the Taylor series expansion of the interpolated function matches the original function in as many terms as possible. In fact, the interpolated function will exactly reconstruct up to a quadratic polynomial. The value $\alpha = -0.5$ also makes the Fourier transform of the filter have a second derivative of zero at zero frequency. In other words, the frequency response is maximally flat. It also turns out that this value of α is equivalent to interpolating the data points with a Catmull-Rom spline [7].

Another popular choice is to make $\alpha = -1$. This makes the slope at $x = 1$ equal to the slope of the sinc function. This produces some high-frequency sharpening, which is useful in some circumstances. (NASA mandated this particular cubic polynomial for interpolating remote-sensing data.) Finally, $\alpha = -0.75$ makes the spline have second-derivative continuity at $x = 1$. This provides mild sharpening, much less visible than the previous case. Adobe Photoshop's “bicubic” resampling filter appears to be either identical or very similar to this case. Our `diresample` and `discale` programs provide all three choices for cubic polynomial resampling.

Figures 7.3 and 7.4 show the cubic polynomial filter for $\alpha = -0.5$ and $\alpha = -1.0$.

Few image resizing programs provide all of the above filters. Cubic convolution provides visually pretty good results in most circumstances. However, we wanted to do better yet if we could, since there is always some loss in resampling.

Meijering, Zuiderveld, and Viergever [49] describe a way to extend polynomial interpolation to higher orders. It is basically a family of interpolation functions derived using similar methods for all. A D th-degree (D is always odd) function is constructed from $D + 1$ segments each of degree D , and it approximates a

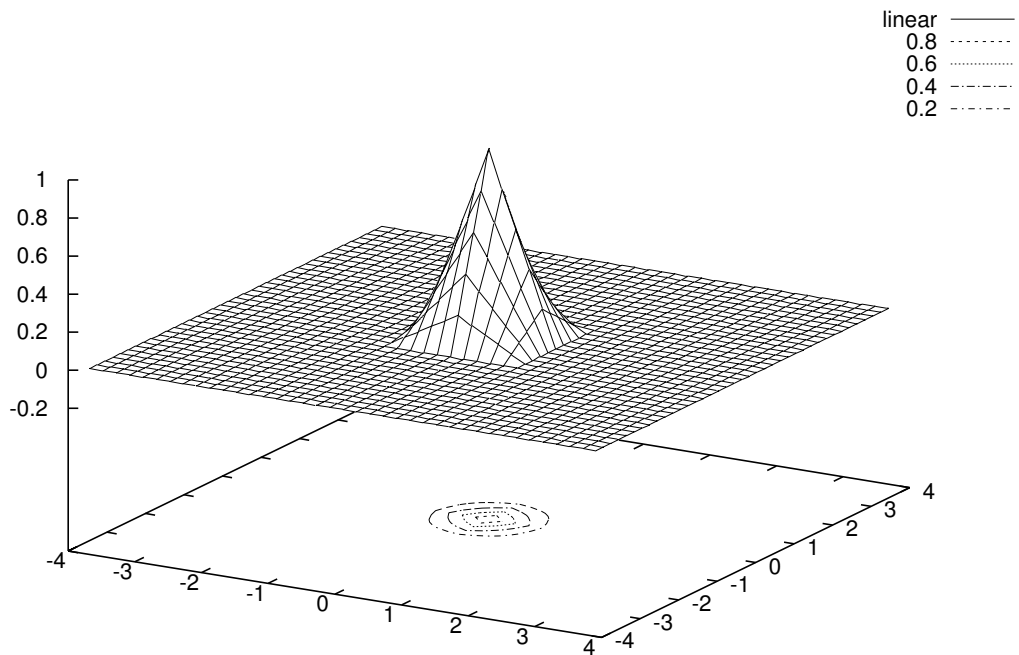
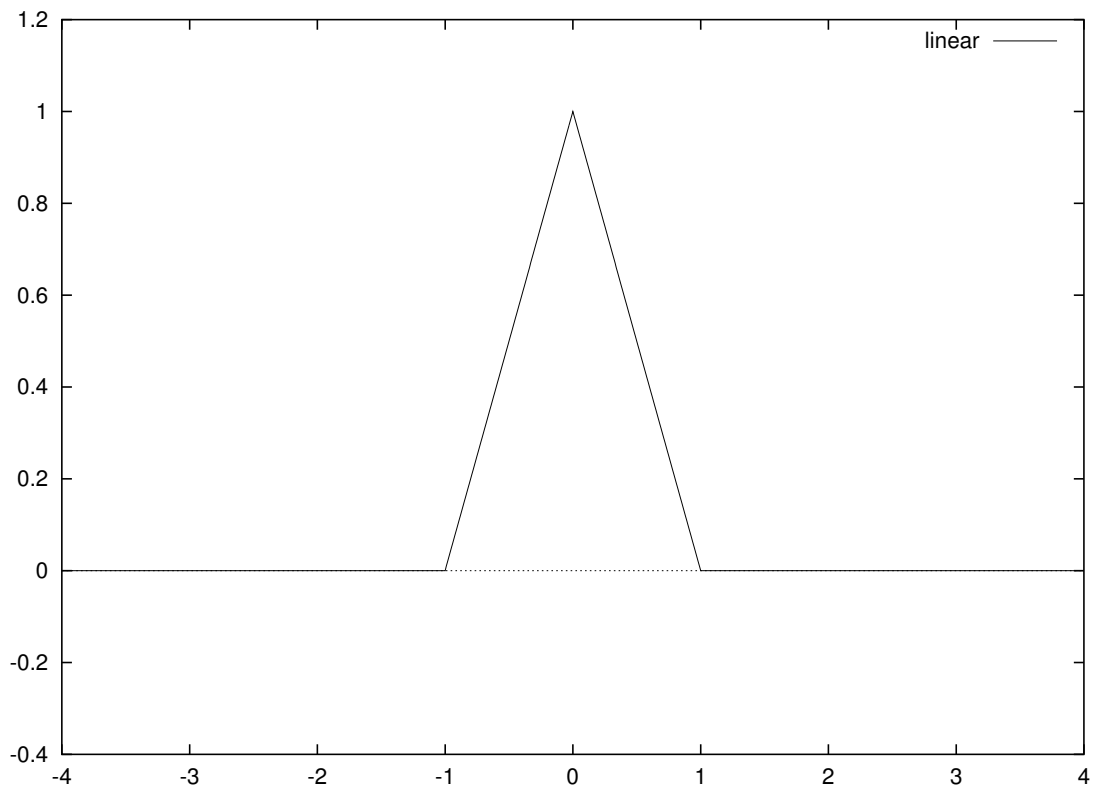


Figure 7.2: Triangle filter in 1D and 2D

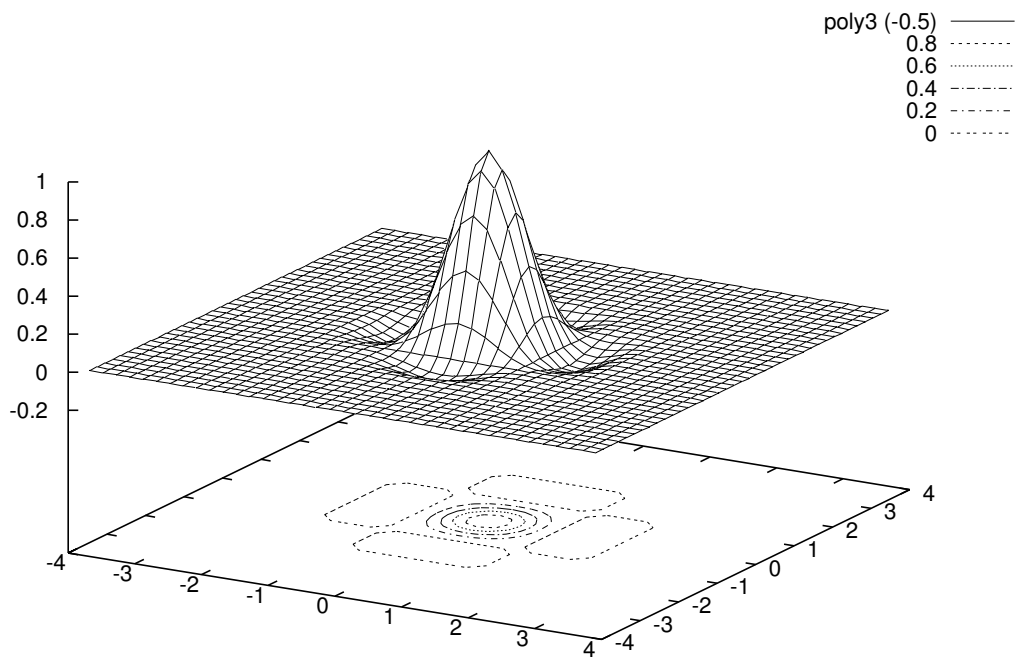
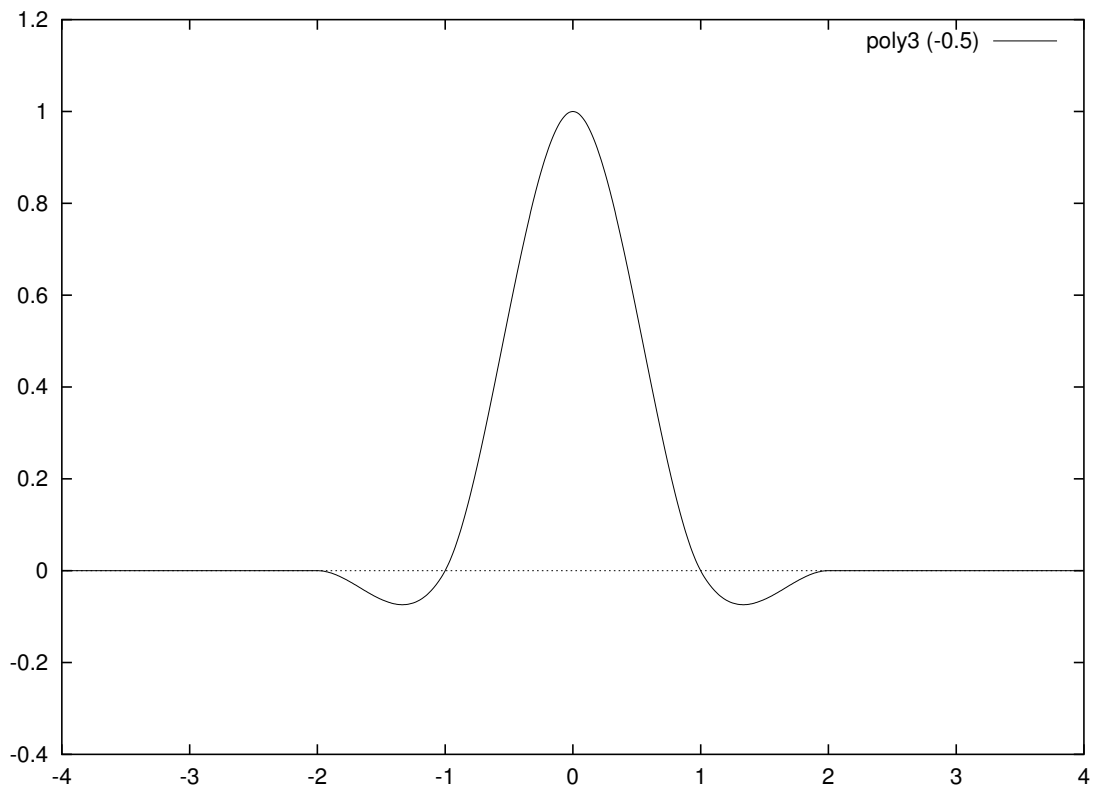


Figure 7.3: Cubic polynomial filter, $\alpha = -0.5$

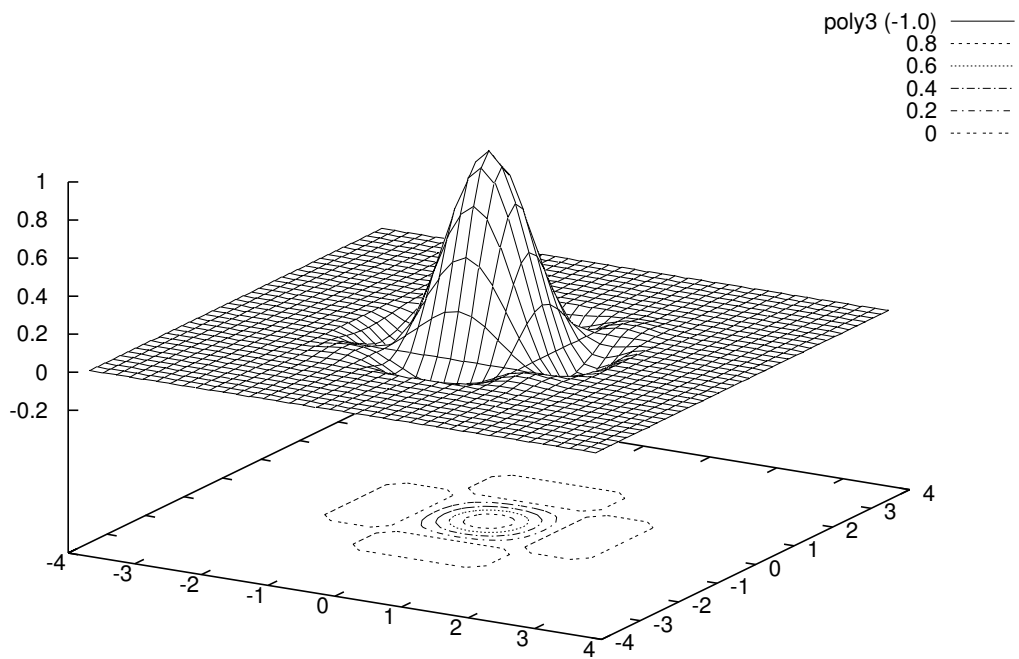
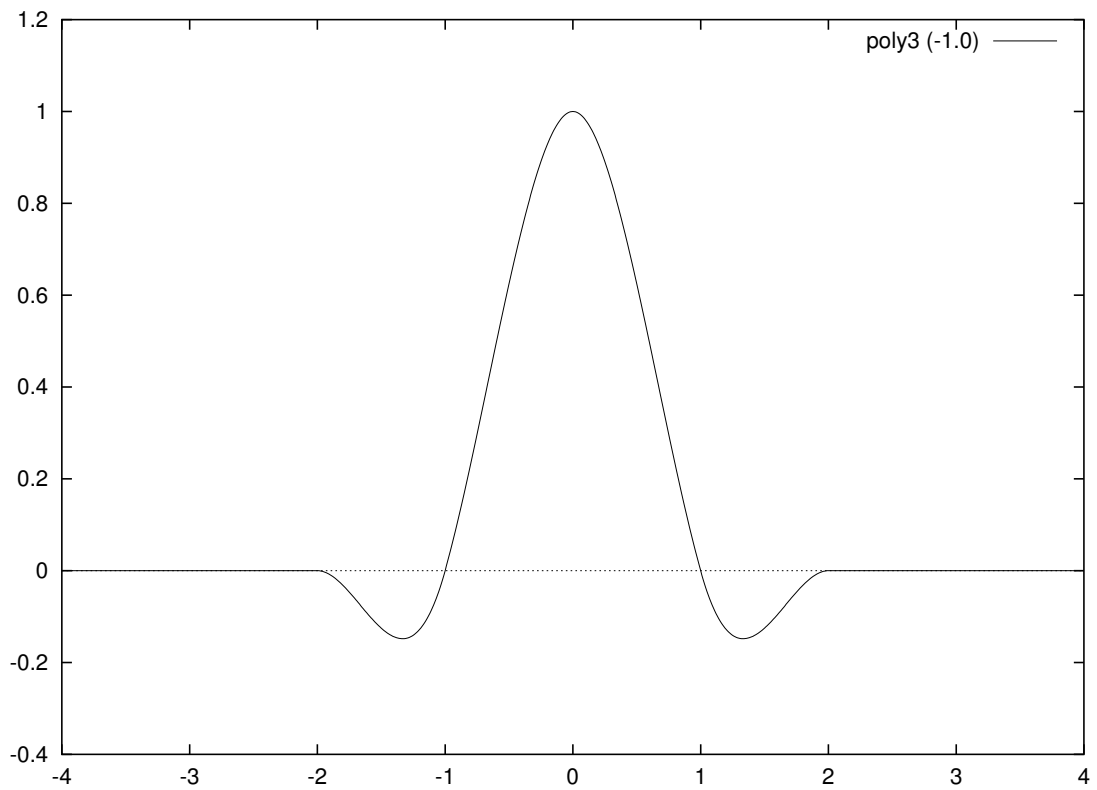


Figure 7.4: Cubic polynomial filter, $\alpha = -1.0$

sinc function over the domain $[-\frac{D-1}{2}, \frac{D+1}{2}]$. The polynomials are constrained by requiring $h(0) = 1$ and $h(x) = 0$ for $x = 1, 2, \dots$ as before. We now require continuity of all derivatives up to $(D - 2)$ th at the knots. This yields a system of polynomials with one degree of freedom, so there is still a free parameter α . This can be constrained by choosing one of three criteria: maximally flat Fourier transform, slope equal to the sinc function at $x = 1$, or one extra degree of derivative continuity at $x = 1$. (Note that the last choice gives the extra continuity at only one knot, not all knots, so it seems pretty useless.) The paper referenced above lists all of the coefficients for the 5th and 7th-degree polynomial systems, and we have implemented them in our software. The original paper gives only the value of α for maximal flatness of the frequency response, but a later paper [48] gives α values for all three constraints.

We have implemented 5th and 7th-degree polynomial resampling with all three choices of constraints in our resamplers. We made one significant change: The original code defines each polynomial in terms of the actual offset x , which increases by one for each successive polynomial segment. Thus, the first segment polynomial is evaluated over $[0, 1]$, the second over $[1, 2]$, and so on. In the case of the 7th degree system, the fourth polynomial is used over $[3, 4]$, and 4^7 is a large number. The polynomial ends up with large coefficients which cause considerable numerical cancellation during evaluation of the polynomial, particularly when using single-precision floating point. For example, the original form of the final segment in the 7th degree system is

$$f(x) = \alpha x^7 - 27\alpha x^6 + 312\alpha x^5 - 2000\alpha x^4 + 7680\alpha x^3 - 17664\alpha x^2 + 22528\alpha x - 12288\alpha$$

(Note that $|f(x)| \leq 1$ over the domain of interest!)

We performed a simple change of variable for each polynomial except the first so that each segment is evaluated over the domain $[0, 1]$. (This is the normal practice with splines.) For the polynomial shown above, the change of variable is $t = x - 3$, and the polynomial becomes

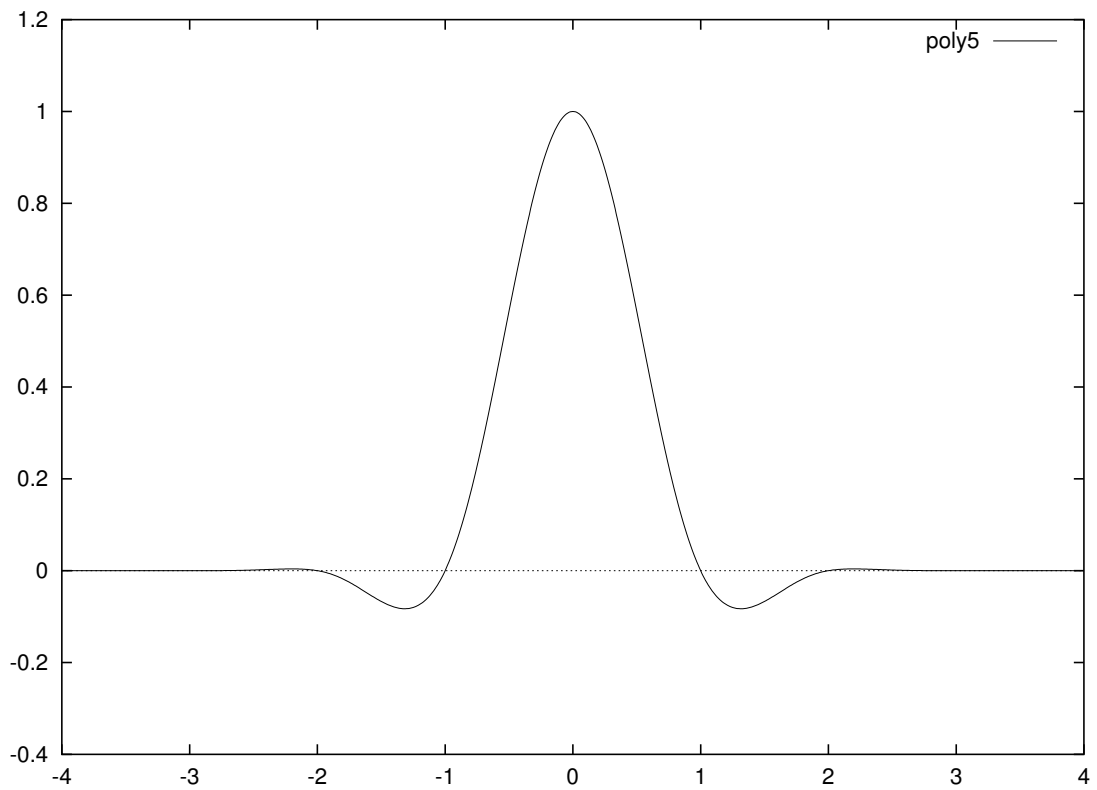
$$f(t) = \alpha t^7 - 6\alpha t^6 + 15\alpha t^5 - 20\alpha t^4 + 15\alpha t^3 - 6\alpha t^2 + \alpha t + 0$$

Figures 7.5 and 7.6 shows the fifth and seventh-degree polynomial filters. (Both have α chosen for flattest response.)

We tried using the different polynomial systems to resample test images. As you might expect, the cubic polynomial system produces considerably better results than linear (triangle filter) interpolation. The resampled images appeared both sharper and smoother. However, 5th and 7th degree polynomial interpolation almost never produced any visible improvement over the cubic polynomial results with our test images, while taking considerably longer to calculate.

Meijering et al. [48] analyzed the root mean square error (RMSE) produced by a large range of interpolation techniques, including all the useful ones above, applied to a sampling of medical images. They found that, for any given order of interpolating function (and thus cost), B-spline interpolation gave less error over the range of samples than any other technique tested. So we added quadratic and cubic B-spline interpolation to our resamplers.

Now, cubic B-spline interpolation is almost the same as polynomial interpolation; in both cases the interpolation function is a piecewise polynomial of the same degree. The only difference is the use of different basis functions. The B-spline basis functions are actually just a box filter convolved with itself a total of four times (for a cubic B-spline). The interpolated result has second-derivative (C^2) continuity, so it is “smoother” than the C^1 continuity of polynomial interpolation. However, there is a problem: A B-spline interpolated function does not pass through its control points. If you use a B-spline directly for image resampling, the result has considerably softened fine detail.



poly5 —
 0.8 - - - -
 0.6 - · - · -
 0.4 - - - -
 0.2 - · - · -
 0 - - - -

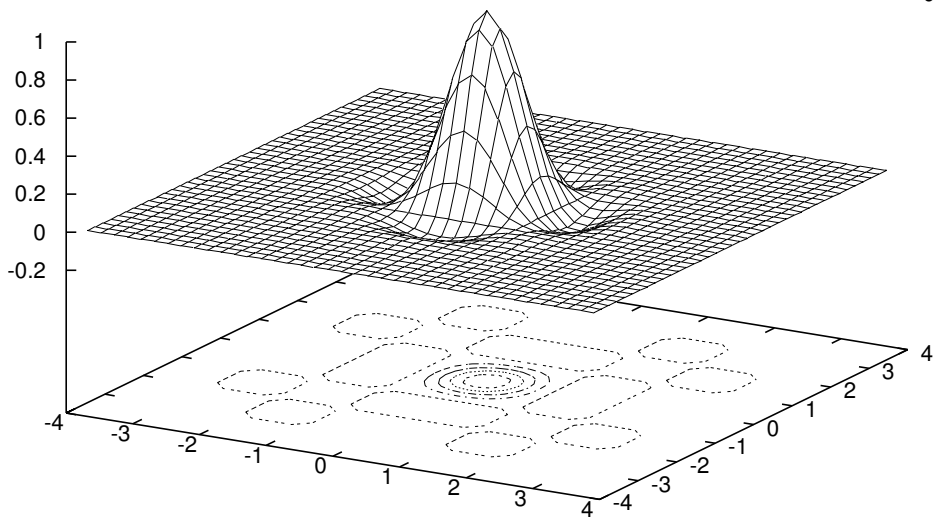


Figure 7.5: Fifth-degree polynomial filter

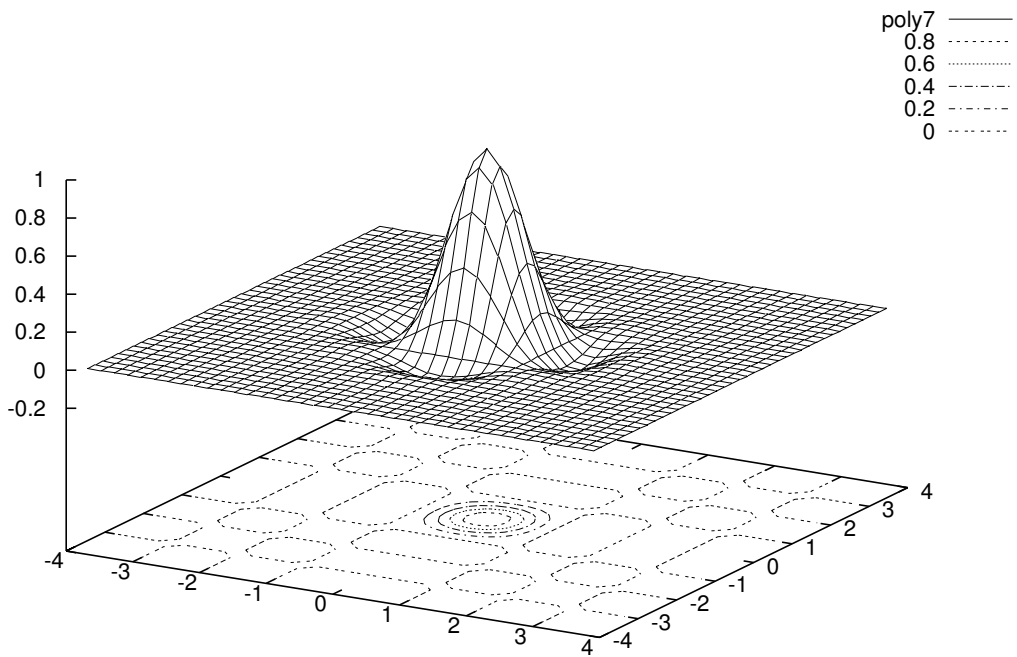
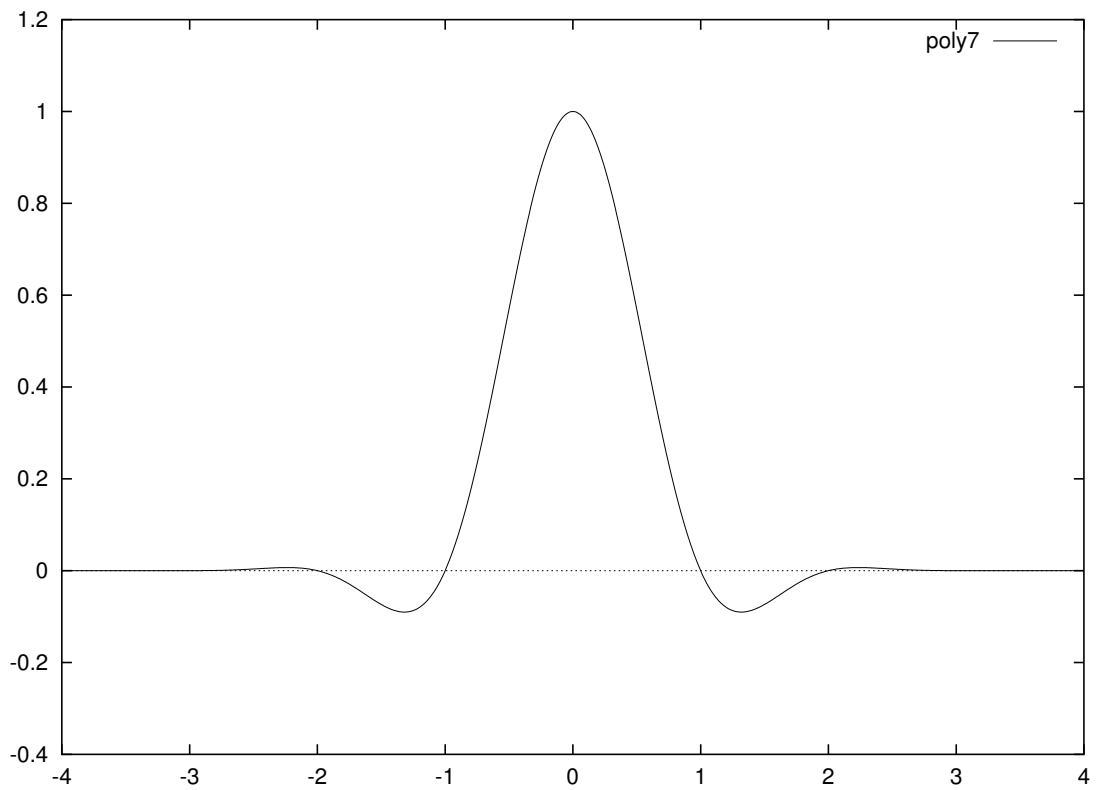


Figure 7.6: Seventh-degree polynomial filter

This can be fixed by a preprocessing step. Essentially, we need to find a set of control points for the B-spline that cause it to interpolate (pass through) the original data points, which are the pixel intensities. We can do this as a preprocessing step before the actual interpolation. The preprocessing is separable, so with a 2D image we only need to apply a 1D version of the algorithm to each row and each column of the input image. One method for the preprocessing step, described in Wolberg’s book [79], involves setting up a system of linear equations and solving it. For a cubic B-spline, the system of equations is tridiagonal, which can be solved in linear time. We implemented this in our code, using a modified version of the Numerical Recipes `tridag` routine. Michael Unser [73] describes an entirely different way to do the preprocessing, using a pair of specially-designed high-pass filters that are run over the rows and columns of the input image. This has ultimately the same effect, and may cost less than the matrix method.

Although the B-spline interpolation step itself uses a filter kernel of limited support, the preprocessing step involves a global solution to a system of equations (or a infinite impulse response filter, with Unser’s method). Thus, every output pixel depends on all pixels in the input image. This is the cost of the additional degree of continuity that B-spline interpolation gives. Because of this, the prefiltering step requires that the entire input image be read into memory before beginning to produce the output image (unless you’re willing to make multiple passes over the image). This is no problem for `dirsample`, since it reads the input image as a matter of course. However, `discale` is careful to minimize the number of input image rows in memory, so it cannot use this interpolation technique.

Figure 7.7 shows the cubic B-spline filter alone; note that it does not go through $(0, 1)$ like all of the other interpolation functions. Figure 7.8 shows the effective filter formed by the preprocessing step followed by interpolating with the B-spline filter. (Note that this filter has infinite support; it is not limited to the $[-4, 4]$ domain shown.)

There is another candidate for “best” reconstruction filter. The theoretically perfect filter is the sinc function:

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$$

This function is considered ideal because its Fourier transform is a box, so it passes the original signal frequencies with no attenuation, while totally eliminating all frequency-domain images produced by the sampling process. However, the sinc function has infinite extent in the spatial domain, so it is impossible to use directly. It can be truncated after a certain distance from the origin, but that may produce artifacts in the result.

A better way to use the sinc as a reconstruction filter is to window it, multiplying it by a second function that has a value of 1 at $x = 0$ but which gradually tapers to zero some distance from the origin. There are a variety of windows one might choose; Turkowski recommends the Lanczos-windowed sinc, and Wolberg describes it as well. The N -lobed Lanczos window is given by

$$L_N(x) = \begin{cases} \text{sinc}(\frac{x}{N}) & 0 \leq |x| < N \\ 0 & N \leq |x| \end{cases}$$

and the complete Lanczos-windowed sinc with N lobes is

$$L_N \text{sinc}(x) = \begin{cases} \text{sinc}(x) \text{sinc}(\frac{x}{N}) & 0 \leq |x| < N \\ 0 & N \leq |x| \end{cases}$$

This filter has the advantage of being truly variable-sized; N can be any integer ≥ 2 . It is more expensive to evaluate the filter coefficients themselves than it is for a polynomial, so we build a lookup table to speed up the process of obtaining coefficients.

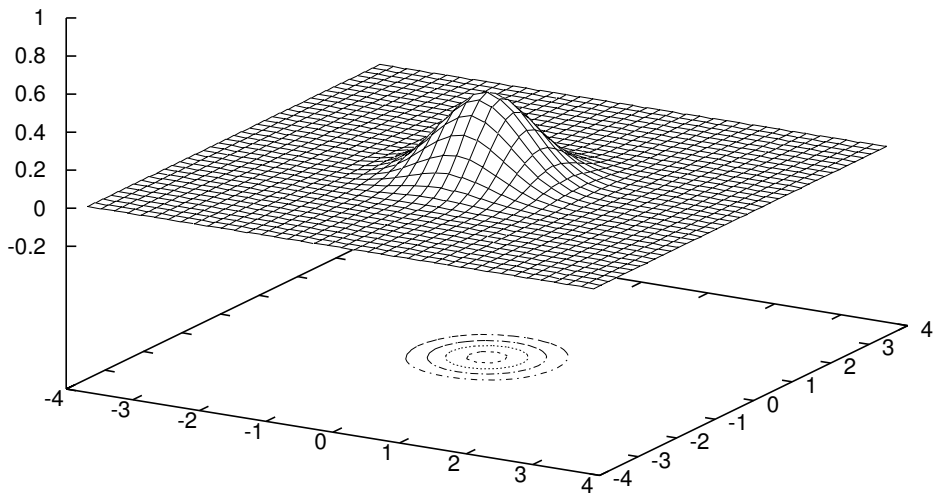
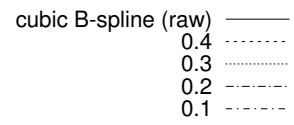
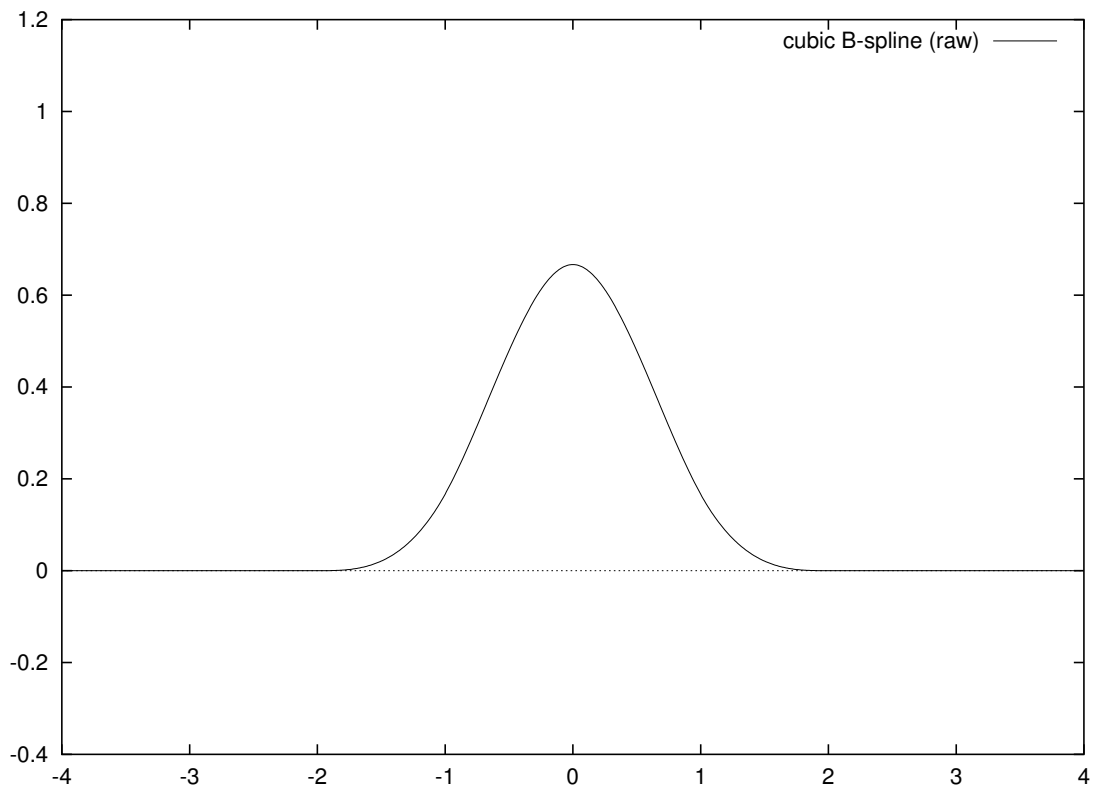


Figure 7.7: Cubic B-spline filter alone

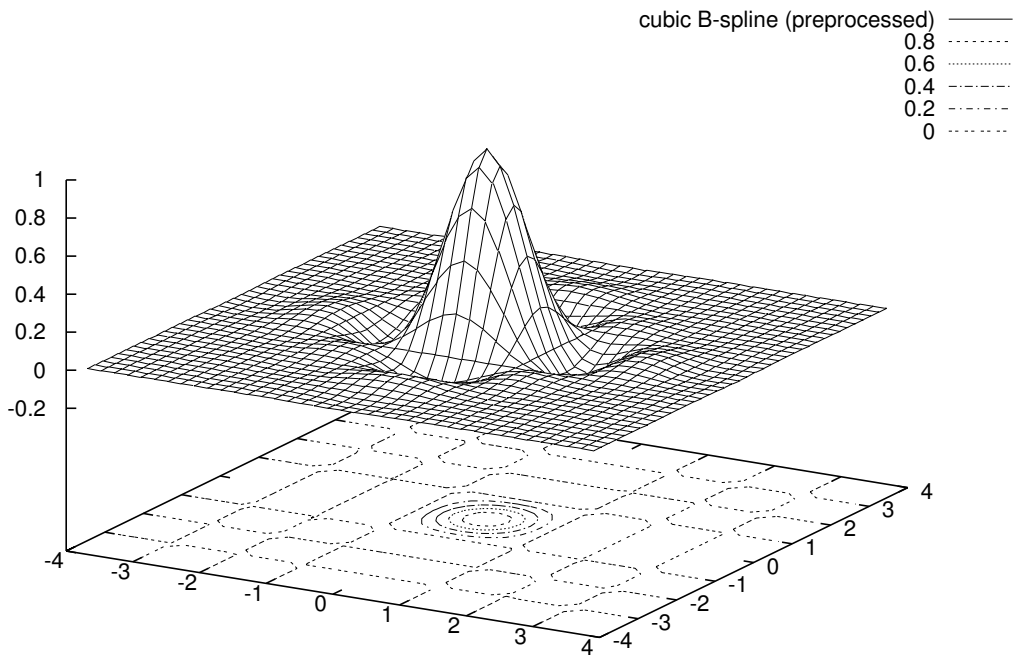
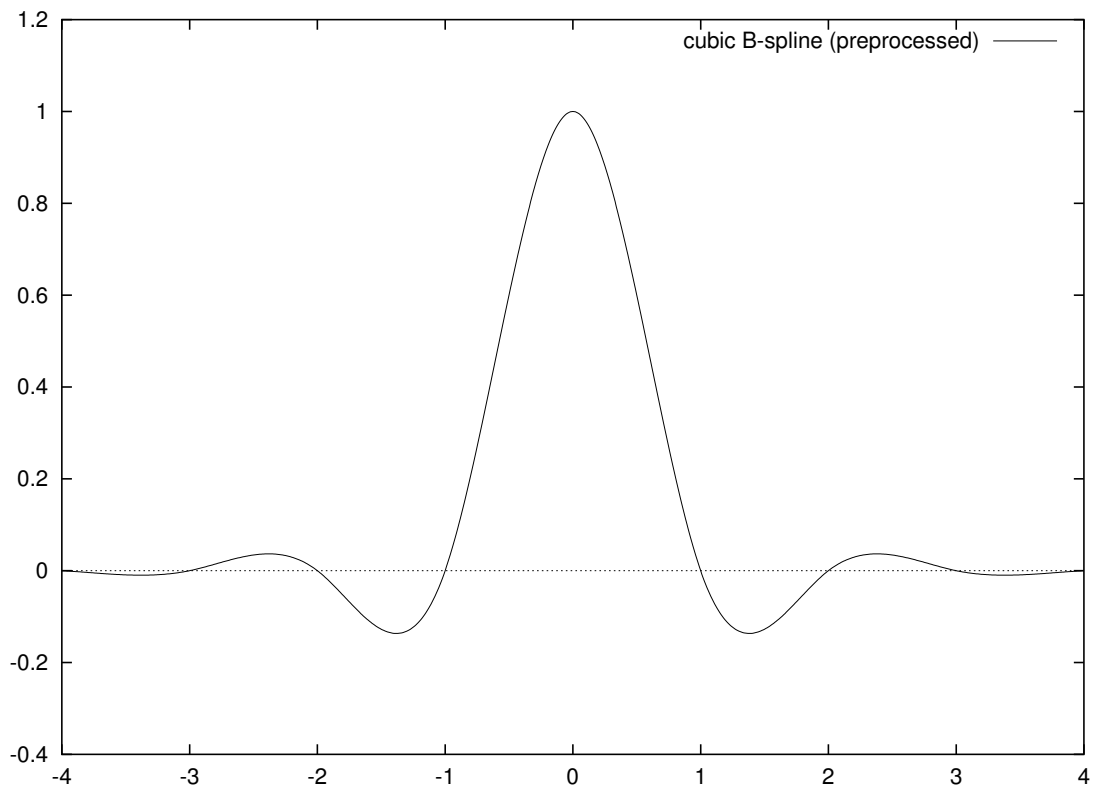


Figure 7.8: Cubic B-spline filter with preprocessing

However, there are other problems with the Lanczos-windowed sinc, and with windowed sinc functions in general. The sinc function itself has an integral of 1.0, but this attribute is lost when it is windowed. The discrete counterpart of this property is that any set of coefficients derived from the filter should sum to 1.0, no matter what the offset between the sample points and the origin of the filter. This is “automatically” true of the triangle, polynomial, and spline filters at their natural size because of their design, but it is not true of a windowed sinc, so the normalization of the sum to 1.0 must be done explicitly.

Figure 7.9 shows the plain unwindowed sinc filter. Figure 7.10 is a Lanczos-windowed sinc with two lobes. Figure 7.11 is the 4-lobed version. (Note that the sinc filter has infinite support, while the two windowed sinc functions are zero outside the area shown.)

The Lanczos-windowed sinc allows us to get arbitrarily close to the “ideal” box-shaped frequency response simply by increasing N . However, the “ideal” response isn’t always ideal for our purposes. It is a fundamental fact of filtering that the squarer the response of a filter in the frequency domain, the worse the filter overshoots in the spatial domain when it encounters an abrupt edge — particularly if the input was not well anti-aliased at its origin. Worse, the overshoot “rings” with a little sine wave that dies out as you move away from the edge, but the wider the filter the further from the edge the ringing appears.

When an image will be resampled many times, it is important to minimize the alteration of its high-frequency content. In these circumstances, a wide Lanczos-windowed sinc would be a good choice. When an image will be resampled only once, that isn’t necessarily true. Subjectively, we might prefer an image with slightly more high-frequency loss (e.g. the result of the B-spline filter) if it avoids ringing artifacts.

7.4.1 Evaluation

In cases where what we really want is a pair of aligned images, the output of Phase 4 is actually the output from the whole process, and it is sensible to choose a resampling filter based on subjective judgements of the image it produces.

However, when we’re using the full pipeline to compare images, the output of Phase 4 forms one input to Phase 5, and what we really want is the resampling technique that introduces the least error into the comparison. It is more useful to use an objective method of performance to evaluate the resampling methods in this case.

Ideally, we could use the Phase 5 algorithm itself to tell us how much difference there is between a “before and after” image pair, and which resampling filter gives the least visible error. However, Phase 5 takes some perceptual effects into account, and its results depend on the assumed viewing distance of the image. We want a resampling filter that works well for all image size and viewing distance combinations, so the frequency-selective filtering of Phase 5 is not appropriate.

Instead, we use simple Mean Squared Error (MSE) to measure interpolation error.

We first tried resampling some test images many times, with a range of rotations and translations. The rotation test rotates the test image by angles of 0.7, 3.2, 6.5, 9.3, 12.1, 15.2, 18.4, 21.3, 23.7, 26.6, 29.8, 32.9, 35.7, 38.5, 41.8, and 44.3 degrees. The sum of these angles is 360 degrees, so the final result image should (ideally) be identical to the original. The translation test translates the test image by 0.01, 0.04, 0.07, 0.11, 0.15, 0.18, 0.21, 0.24, 0.26, 0.29, 0.32, 0.35, 0.39, 0.43, 0.46, and 0.49 pixels in one direction. The sum of these displacements is 4 pixels. Then we translate the image by -4 pixels (which is lossless), bringing it back to its original position for comparison with the original. (This particular set of test values is from [48].)

Table 7.1 shows the results of these tests on a couple of standard 512-pixel images. The numbers are RMS error expressed as a percentage of the distance between black and white. This is calculated after

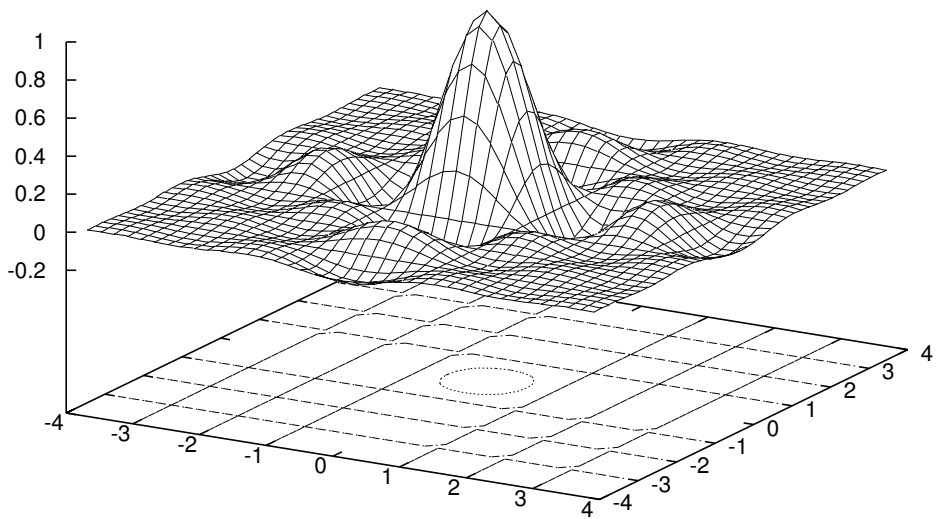
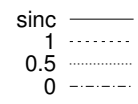
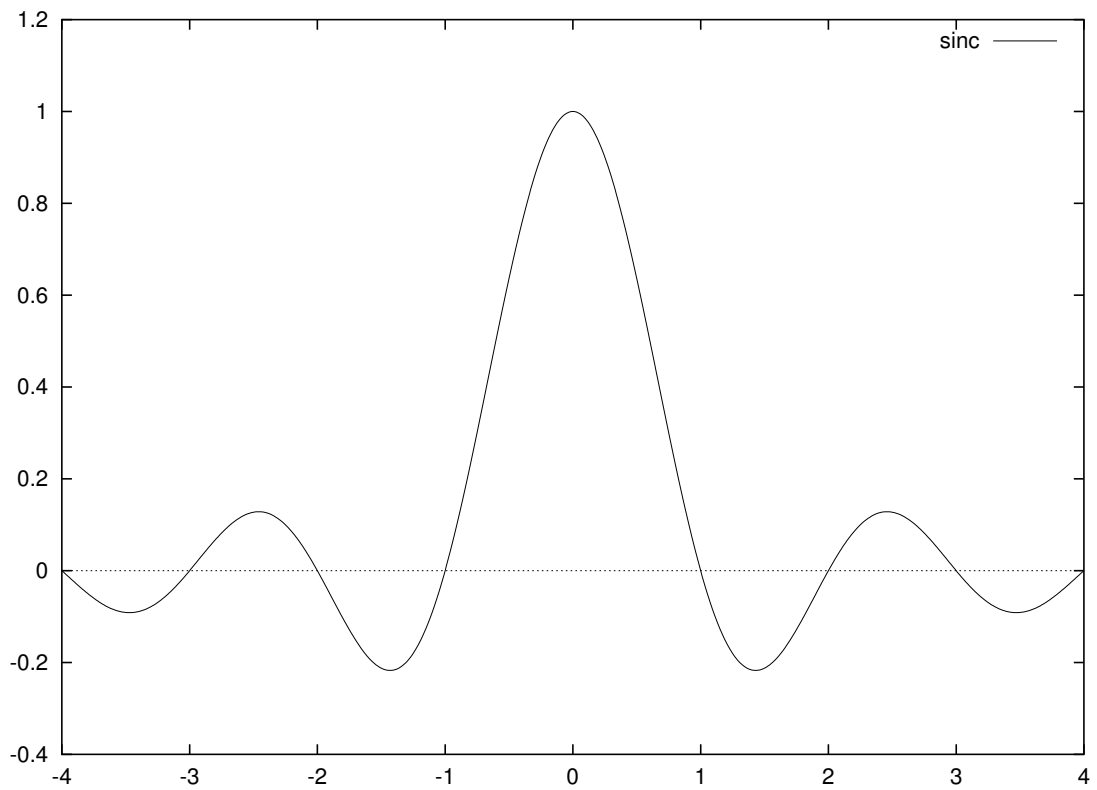


Figure 7.9: Sinc filter

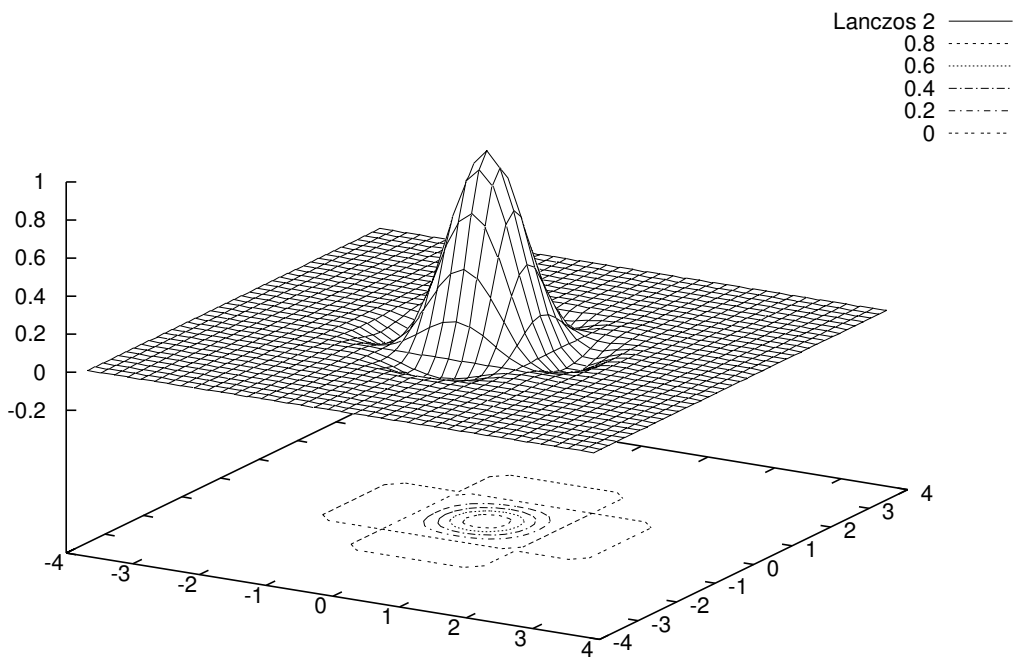
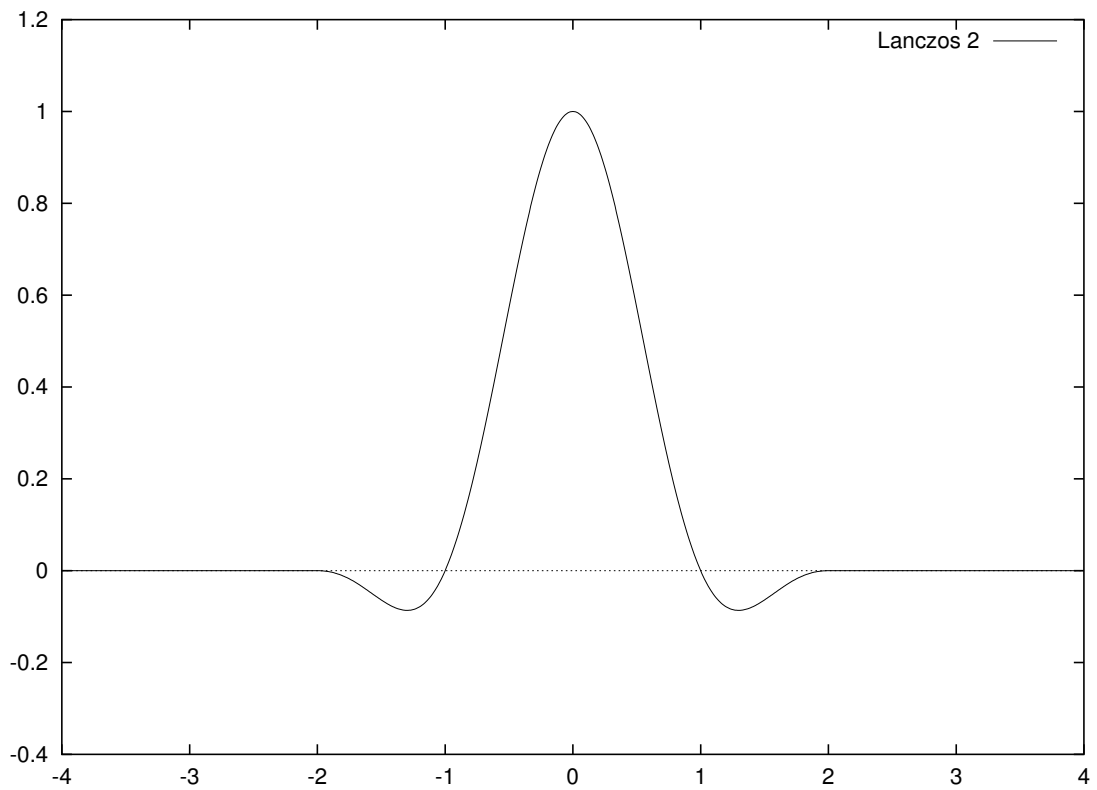


Figure 7.10: Lanczos-windowed sinc filter (2 lobes)

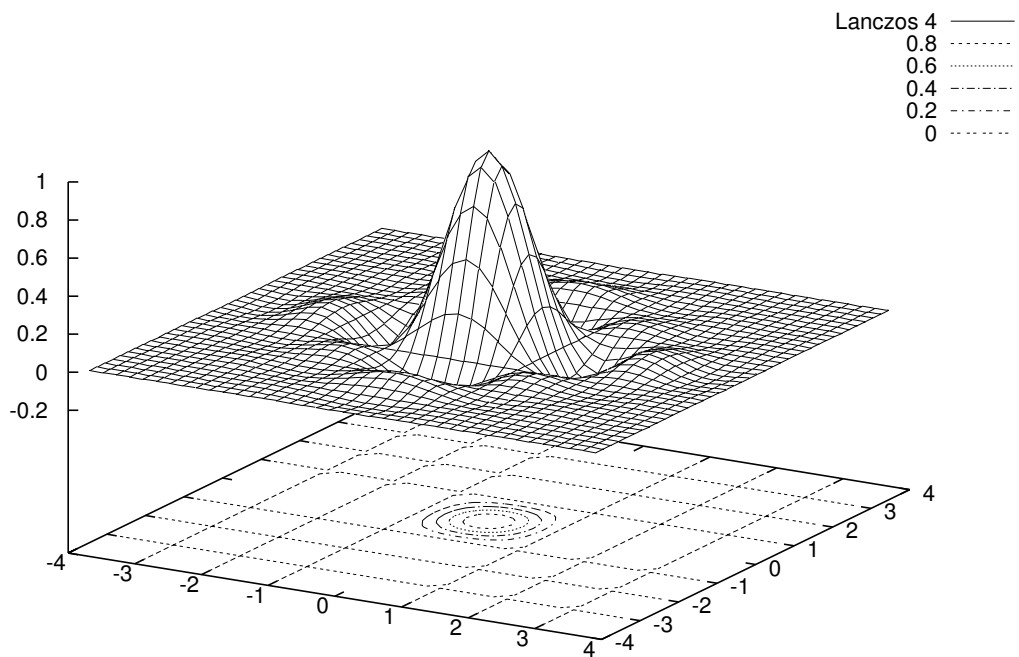
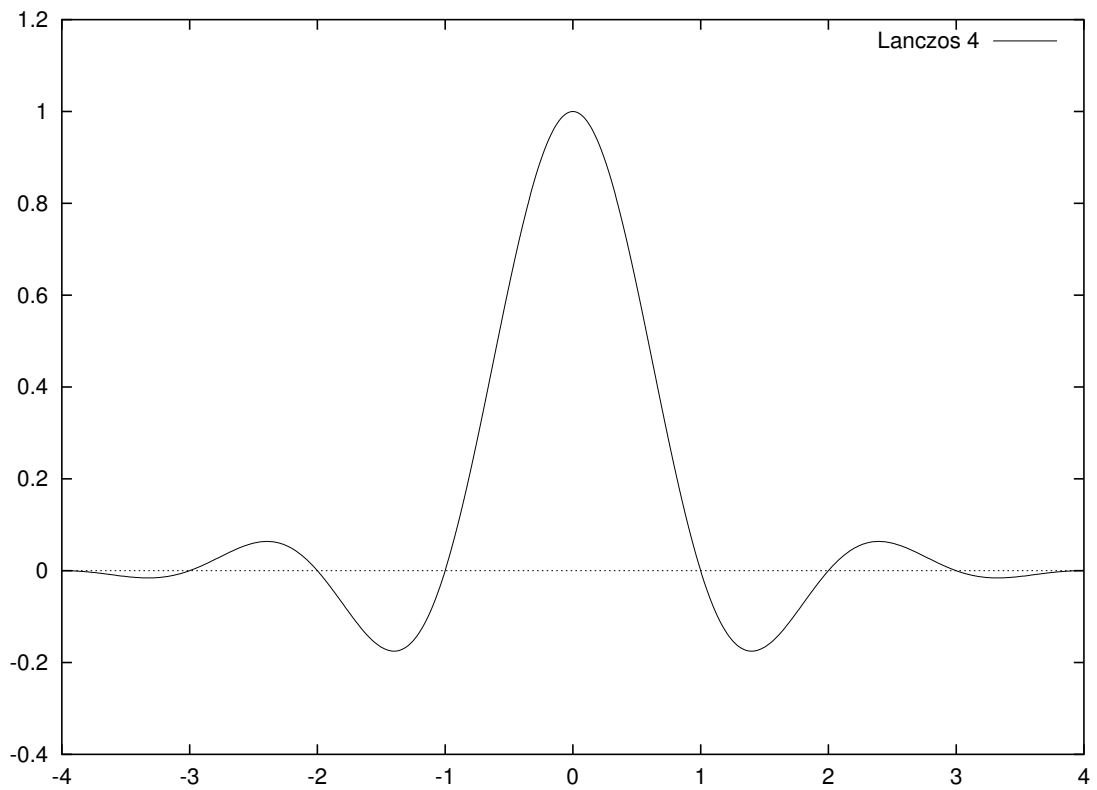


Figure 7.11: Lanczos-windowed sinc filter (4 lobes)

filter	mandrill		Lenna		peppers	
	rotate	translate	rotate	translate	rotate	translate
nearest	11.50	15.00	6.94	10.40	5.59	9.04
linear	8.21	6.98	4.51	3.10	3.56	2.83
poly3	5.96	5.49	2.74	2.01	2.34	2.10
poly5	5.82	5.34	2.66	1.93	2.29	2.04
poly7	5.66	5.17	2.58	1.85	2.24	1.99
cubic B-spline	4.53	4.07	2.03	1.38	1.88	1.65
Lanczos2	5.92	5.33	2.72	2.03	2.33	2.10
Lanczos3	4.23	3.93	1.91	1.40	1.78	1.63
Lanczos4	3.58	3.25	1.62	1.13	1.60	1.40
Lanczos8	2.68	2.11	1.24	0.75	1.34	1.06
Lanczos16	2.34	1.46	1.11	0.55	1.26	0.57

Table 7.1: Resampling errors — many steps

filter	mandrill	Lenna	peppers
nearest	1.63	0.83	0.69
linear	4.13	1.95	1.74
poly3	2.61	1.18	1.2
poly5	2.51	1.14	1.17
poly7	2.44	1.10	1.15
cubic B-spline	1.91	0.88	0.97
Lanczos2	2.57	1.17	1.19
Lanczos3	1.84	0.85	0.95
Lanczos4	1.56	0.73	0.84
Lanczos8	1.06	0.51	0.52
Lanczos16	0.79	0.40	0.33

Table 7.2: Resampling errors — two steps

converting the gamma-corrected pixel values to a value proportional to intensity.

We can see that the Lanczos-windowed sinc with 3 lobes (this gives a 6×6 filter kernel) is better than all of the other methods shown, except for one case where it is nearly equal to the B-spline. As we increase the number of lobes further, the error continues to go down, but we need to pick some point to stop. (We also tried a Kaiser-windowed sinc, but it was generally worse than a Lanczos-windowed sinc of the same size, except for the shortest length).

It would also be very interesting to know how much error arises in a single resampling step. It is difficult to measure this as a practical test, though, since it takes at least two resampling steps to move an image and then bring it back to where it was, or to take it to an orientation from where the image can be losslessly returned to its original form. Also, two equal size but opposite-direction rotations or translations might have errors that cancel. So we did two rotations by two different angles (3.14 and 86.86 degrees) that add to 90 degrees. Then we returned the image to its original orientation by a lossless permutation step, and compared it to the original. The results are shown in Table 7.2.

It is interesting to note that 16 rotations generate only about twice the RMS error that 2 rotations did.

Based on these tests, we have selected Lanczos4 (8×8 kernel) as our standard for Phase 4 resampling.

It gives less error than all the shorter filters at not much more cost. Lanczos8 is better yet, but not by that much, and twice as expensive.

We've also done some visual verification of the output. To check a pair of images for geometric alignment accuracy, we usually use a method that converts both images to monochrome, then creates a new RGB image whose red and blue components come from one input image while the green component comes from the other input image. If the two images are misaligned, magenta or green colour fringes appear on the edges of objects. Perfect alignment of the two input images gives an image with no colour. This test can reveal geometric errors of one pixel or less with a high contrast image, as well as showing the direction of misalignment. When this test is applied to the Lanczos4-resampled test images after 16 rotations, there is absolutely no visible colour. Thus, the geometric calculations are accurate.

We also have another test that is more sensitive, but harder to interpret. For this test, we simply subtract the one image from the other and add an offset of 0.5 (half-intensity grey). With a high-contrast image like the mandrill, this test will show misalignment of 0.1 pixel or even less as a sort of "bas-relief" effect in the output image. This test also shows any interpolation errors. The only differences found are differences in high-frequency content primarily in the whisker and hair areas of the image, but they are not biased in any one direction. Thus, we conclude that the geometric accuracy is very good, but we can see some interpolation errors.

7.5 Implementation

Logically, we could have a single resampling program that performed all of the resampling operations that we could possibly want. However, different operations are controlled more easily using different user interfaces, which allows different implementation tradeoffs to be made. In reality, we use two different programs for high-quality image resampling: `diresample` and `discale`.

The more general of these is `diresample`. The mapping from input to output pixel locations is specified by an arbitrary affine transformation. More than a dozen different reconstruction (anti-imaging) filters are provided for the user to choose among. The output image may be the same size as the original, or it may be automatically sized to avoid discarding any input pixels, or its size may be explicitly specified.

Because the output image may have any orientation with respect to the input image, including rotations and mirror reflection, it would be difficult for `diresample` to keep track of exactly which input image rows are needed for each output row. Indeed, in some cases every input row is referenced for every output row. To avoid trying to manage this, `diresample` simply reads the entire input image into memory before beginning any resampling. The internal representation uses floating point, which typically takes four times as many bytes as the disk storage form of the image. Any image that is too large to fit in physical memory may result in very slow operation and a great deal of paging. Fortunately, typical memory sizes are now large enough that images up to about 6000 pixels wide can be handled without difficulty.

Again because the output image rows may be oriented arbitrarily with respect to the input image rows, the coefficients of the anti-imaging filter are (in general) different for every output pixel, and have to be calculated individually. This takes considerably longer than simply convolving with precalculated filter coefficients, particularly if the filters involve trig functions rather than merely evaluating a polynomial. In some cases, `diresample` calculates a lookup table of anti-imaging filter shape to reduce some of this cost, but `diresample` is still about half as fast as `discale` for operations that both programs can perform.

The code of `diresample` provides only an anti-imaging filter, aligned properly with the input image

sample grid. It cannot be rescaled and used as an anti-aliasing filter. Thus, `diresample` is suitable only for same-size resampling or upsampling, not downsampling. However, that is all we need for Phase 4 of the thesis.

The actual transformation is specified by six numeric values, which are the first two rows of an affine transformation matrix using homogeneous coordinates. The third row never changes, so it does not have to be specified. This is general, but not very intuitive for a human being to use. For general use, the auxiliary program `dimatrix` can be used to generate the matrix for an arbitrary sequence of rotation, translation, uniform scaling, and reflection operations.

However, `diresample`'s interface isn't convenient for use within the processing pipeline of this thesis. Phase 3 writes out a transformation file that contains both the matrix itself and also the sizes of the two images (for error checking, and to crop the output image correctly). We provide another program named `resample` to perform Phase 4. It reads in the transformation file, checks that the image input file is the correct resolution for use with this transformation, and then calls `diresample` to do the real work.

The other program that does resampling is `discale`. As its name suggests, it only handles resizing images, but it has a number of advantages over `diresample` for that task. It is faster and uses less memory.

Since no rotations are possible, the rows and columns of the output sampling grid are parallel to the rows and columns of the input sampling grid. This allows `discale` to precalculate all of the filter coefficients needed for horizontal interpolation at each X position, and then reuse those coefficients for every line of output. Similarly, the vertical filter coefficients can be calculated once for each row of the output, and then reused in every column. This is responsible for most of the speed difference between the two programs.

Again because the sampling grids are aligned, a single row of the output image depends only on a small number of input rows (the number is equal to the width of the anti-imaging filter used). The `discale` program allocates only enough memory for this many rows of the input image, and reads in new lines (discarding unnecessary ones) only as necessary. Thus, it can handle almost arbitrarily large images in little memory. (However, this prevents using B-spline reconstruction filters because they require multiple passes over the data — see more detail below.)

Unlike `diresample`, `discale` is designed to handle downsampling properly. Assume that the anti-imaging filter in use has a natural width of N pixels (e.g. $N = 4$ for cubic polynomial interpolation), and that the image scale factor is S where values greater than 1 imply enlarging the image. When $S \geq 1$, the anti-imaging filter operates at its natural size in input image coordinates, spanning $N \times N$ input pixels, while its effective size in output image pixels is $NS \times NS$. When $S < 1$ we now need an anti-aliasing filter that is wider (has a lower cutoff frequency) than the anti-imaging filter. The `discale` program rescales the filter so it is $N/S \times N/S$ input pixels in size, which makes it a constant $N \times N$ output pixels in extent. The amplitude of the filter is also scaled by s to keep the sum of its coefficients equal to 1.0. The anti-imaging filter has effectively turned into an anti-aliasing filter, even though the algorithm is still applying it in input image coordinates.

Using the anti-imaging filter as an anti-aliasing filter works in `discale` because the input and output pixel grids are different by at most a scale and a translation. When an anti-aliasing filter is needed (during downsampling), the anti-aliasing filter has a fixed size and shape and alignment relationship with the output sample grid. If the anti-aliasing filter is warped by the inverse of the image translation, it is still the same shape in input space, and it is not rotated with respect to the input pixel grid. In fact, the only difference between an anti-aliasing filter warped into input space and an anti-imaging filter is that the anti-aliasing filter is scaled wider, and it is centred on a particular output pixel rather than a particular input pixel. Thus, only

a slight change in the resampling code is necessary to support downsampling as well as upsampling.

In contrast, `dirsample` allows an arbitrary affine transformation between input and output. When a nice rectangular anti-aliasing filter is mapped from output space back into input space, the result is not necessarily rectangular, and its edges are not generally aligned with the input sample grid's rows and columns. So the anti-imaging filter cannot simply be used as an anti-aliasing filter.

It is possible to rework the resampling algorithm to filter in output space, where the anti-aliasing filter is rectangular and aligned with the sample grid, to handle downsampling. However, we don't need this in Phase 4, so `dirsample` does not yet provide for this case.

Except for the differences noted above, `dirsample` and `discale` are very similar programs. They both provide the same set of reconstruction filters and most of the same options. Except where specifically noted, the rest of this section applies to both programs equally.

7.5.1 Edge Handling

During resampling, the algorithm will sometimes reference a pixel address in the input image coordinate system that is outside the input image itself. With `discale`, the edges of the output image are constrained to lie somewhere within the input image, so you cannot "look" beyond the boundaries of the input image. But the anti-aliasing or anti-imaging filter has a non-zero size, and even though the sample location of the output pixel must be inside the input image, up to half the width or height of the filter may extend beyond the input image. At a corner of the input image, fully 3/4 of the area of the filter may actually be outside of the input image. To the viewer's eye, these output pixels are still within the input image, and they should faithfully reflect what was there in the input image. We need to invent some value for the "missing" input pixels, even though we have no real data for them, so we can properly calculate the output pixels.

The simplest approach is to assume that all pixels outside the input image are zero (black). This is simple to implement and efficient, but it creates an abrupt artificial discontinuity in intensity where none existed in the original scene. This sharp edge will provoke overshoot from any filter more complex than the triangle filter, and with sharp-cutoff filters like the Lanczos-windowed sinc there will be a band of ringing that extends several pixels into the inside of the source image. This approach is best avoided.

Another method is to treat the input image as being periodic, with the left edge logically adjacent to the right edge, and top to bottom. This effectively tiles the infinite plane with repeated copies of the image, all in the same orientation. The discrete Fourier transform makes exactly this assumption about its input. This is better than the "black border" approach, since there will be less discontinuity across edges. However, few images are truly periodic, and there is often a large difference in brightness between top and bottom of an image, or (less often) between right and left. This remaining discontinuity can still stimulate filters to overshoot and ring.

A better method yet is to extend the image by reflection. At the left edge of the image, we assume there is a plane that either passes through the sample points of the left-hand column of pixels, or is located one half pixel further left. We then reflect the entire image in this plane. We assume similar reflection planes at the other three boundaries. This has the effect of tiling the infinite plane with copies of the input image, but rotated and reflected as required so that the images always match along their boundaries. (The two choices of location for the plane cause the edge rows and columns to be either duplicated or not at these boundaries.)

The advantage of this method is that it completely eliminates discontinuity in intensity at the edges (though in general the derivatives will be discontinuous there). This sort of tiling is implicitly built into the mathematics of the discrete cosine transform. However, it is moderately complex to implement properly.

A simpler method is to simply “extend” or copy the first and last row and column out to infinity. This also eliminates the edge discontinuity, but is simple to implement, and seems to work about as well in practice. This is the approach that we normally use in both `discale` and `diresample`.

Note: the discussion above assumes that the same anti-imaging or anti-aliasing filter is used across the entire image area. There are interpolation methods which use different basis functions at the edges of the source image to avoid referencing pixels outside the image. We do not use these methods.

In `diresample`, a more extreme instance of the problem often appears. Because the transformation may include rotation and skew, and because the boundaries of the output image are arbitrary, it is often true that some pixels in the output image correspond to sample locations that are outside the input image. In this case, this pixel’s sample point will appear to the viewer to be located beyond the input image, and there is some question about what is most appropriately shown in the output image.

By default, `diresample` simply sets such pixels to zero. This clearly indicates that these pixels do not contain valid image data. (Note that setting a pixel to zero in the output does not cause any filter problems, since this is done after all filtering.) The other approach (controlled by a command-line option) is to do nothing special at all, and compute the pixel like any other. This will allow the viewer to see the edge rows and columns extending off to infinity, which may be visually disturbing. However, this is useful if the image will be resampled later.

7.6 Performance

The `discale` program is quite fast. It takes about 1.25 seconds (Pentium III 700 MHz) to resample a 1024×1024 pixel colour image to the same size, and slightly less for any smaller output size. In the other direction, it takes about 3.8 seconds to double the dimensions (4 times as many pixels) and 16.1 seconds to quadruple the dimensions (16 times as many pixels). For larger images yet, the cost is nearly linearly proportional to the number of output pixels. (These numbers are for the cubic polynomial filter.)

The `diresample` program is slower because it can’t reuse results as extensively. It takes about 2.0 seconds to rotate the same test image, keeping the same image size, using the same filter. If we use the (better) cubic B-spline filter with preprocessing, the time rises to about 3.4 seconds. Enlarging to twice and four times the dimensions takes 7.7 and 30.9 seconds using the cubic polynomial filter, 9.1 and 31.8 seconds using the cubic B-spline filter.

(The cost of the preprocessing step for the B-spline filter is proportional to the number of input pixels. The actual interpolation cost for both filters is the same, and is almost linearly proportional to the number of output pixels.)

These two programs are among the fastest of the processing steps in the complete pipeline, so selecting a slower but better-quality filter has only an insignificant effect on the total time for our method.

7.7 Discussion

As we said, at the moment `diresample` and its helper `resample` can only perform affine transformations specified in the form of a matrix. However, very little of the code inside these programs is specific to this transformation model. It would be straightforward to replace the geometric mapping with something more complex, for example a projective or perspective transform, or a general warp controlled by a fitted polynomial or a spline mesh.

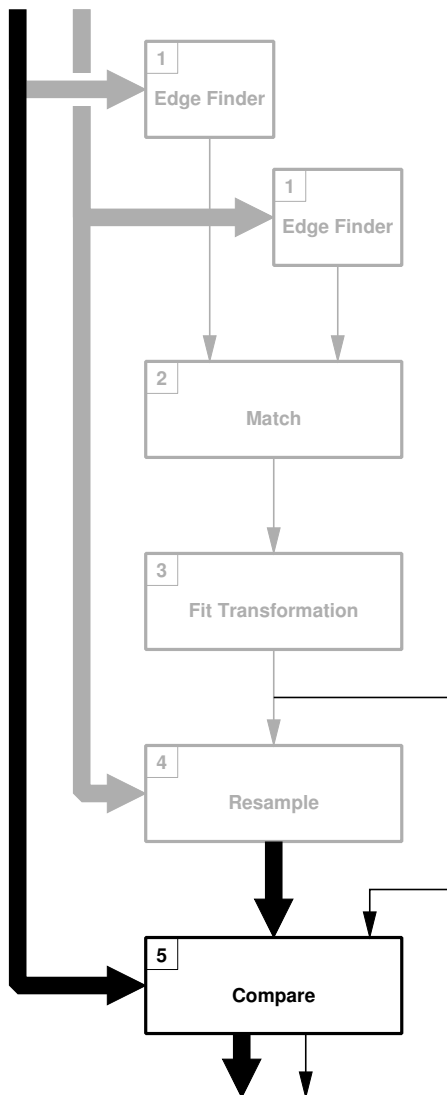
There are two factors that would cause complications: First, the transformation needs to be relatively easily invertible. The algorithm actually maps points from the output image back to the input image, and it would be undesirable (but still possible) if we had to use numerical root-finding techniques to invert the transformation.

Second, and much more serious, is the fact that the software currently provides only anti-imaging (reconstruction) filtering. This works properly only when the image scale remains the same or is enlarged (upsampled). Accepting more general transformations means that we could easily encounter a situation where part of the image is upsampled while another part of the image is downsampled. This, in turn, means that we need to provide an anti-aliasing filter for downsampling, and be able to switch from anti-imaging to anti-aliasing filters in the midst of a scanline.

(There is a workaround for this problem using the existing software: Make the output image for the warping step sufficiently large that all portions of the image are being upsampled, and then perform the warp. Then downsample this image to the output size originally desired using `discale`. This should give good quality, but is unnecessarily expensive.)

Chapter 8

Phase 5 — Image Comparison



The previous phases have delivered two images of exactly the same size that are aligned as accurately as possible. Now we need to compare the content of the images, and show the user where the images differ. We also want to produce a single number which is our “image distance” measure.

There is considerable latitude in what we could do at this stage. Since we are starting with aligned images of the same size, we could use virtually any image comparison technique that has been used over the years (see Section 2.2).

We could use Mean Squared Error (MSE). Since the images have been aligned, MSE of this pair of images is much more meaningful than if we had used the unaligned original images. We could produce a “map” showing where the differences are large and where they are small, which is useful to the user in determining what part of the image is wrong. On the other hand, MSE weights all image differences strictly according to magnitude, with no recognition that the human eye is much more sensitive to some scales of detail than others, so the number we get will not be a very accurate measure of the visibility of the error.

We could perform Fourier transforms on both images, and then calculate the difference in the transforms. This makes it very easy to weight differences according to how visible they are likely to be (as a function of spatial frequency), so our distance number will be more meaningful. However, we no longer have any sort of map of where the major differences between the images are.

Instead, we do a wavelet decomposition of the two images (using the same wavelet transform as in Phase 1). This divides the spatial frequency content of the images into octave-spaced (power of 2) bands, which is nowhere near as precise as the Fourier transform, but good enough for us to model reasonably

the human eye’s sensitivity to information of different spatial frequency. At the same time, it leaves us with positional information about the differences as well. It ends up not being that much more expensive to compute than the previous alternatives, either.

We should point out here that there are difference measures which include more sophisticated models of the human visual system (HVS). One of these is the “Visible Difference Predictor” of Scott Daly [13]. Sarnoff Labs also has a competing model called JND [11]. These algorithms can model the change of appearance with light level, contrast sensitivity changes as a function of orientation of features, and masking effects. Masking refers to an effect whereby strong image content can hide the presence of noise or other errors in an image, but only when the content has nearly the same frequency and orientation as the noise. It would be possible to replace the entirety of our Phase 5 with one of these algorithms, if that is necessary for some application, since we can deliver a pair of equal-size aligned images to such an algorithm. However, Daly’s algorithm seems to process only luminance, ignoring colour. It also uses a CSF model that is not specific to a particular viewing distance, but attempts to model a range of distances (with attendant loss of accuracy). The JND model does process the colour components of the images, but it seems to be strongly oriented toward measuring video image quality, particularly compressed digital video quality.

Instead, we have set our sights on a middle path, something that is more useful than simple MSE, yet not a lot more expensive.

8.1 CSF Models

As mentioned above, we wish to model the fact that the human visual system is more sensitive to some spatial frequencies than others. This is usually done with the aid of the Contrast Sensitivity Function (CSF).

This is measured by showing a test pattern (usually consisting of sine or square wave modulation of intensity) to subjects, and changing the contrast of the pattern until the “just perceptible” contrast level is found. If I_{max} and I_{min} are the maximum and minimum light intensity found in the test pattern, then contrast is defined by

$$C = \frac{I_{max} - I_{min}}{I_{max} + I_{min}}$$

Note that whenever I_{min} is zero (i.e. the pattern reaches full black), contrast is 1.0 by definition (as long as I_{max} is non-zero).

It may be easier to understand the definition above intuitively if the numerator and denominator in this expression are each divided by two. The numerator becomes $(I_{max} - I_{min})/2$, which is the peak (not peak-to-peak) amplitude of the modulation in the “signal”. The denominator becomes $(I_{max} + I_{min})/2$, which is just the mean intensity of the signal (if the modulation is symmetric — it normally is). So contrast is just the ratio of the modulation amplitude to the average intensity.

The contrast measurement is repeated at many different spatial frequencies to map out the minimum perceptible contrast as a function of frequency. The CSF is then the reciprocal of the minimum perceptible contrast function. Thus, if our eyes can see half the amplitude of modulation at a frequency f_1 as we can at some other frequency f_2 , then $CSF(f_1) = 2 CSF(f_2)$.

Figure 8.1 shows three different CSF functions. The function that reaches the highest peak and extends furthest to the right is representative of the human luminance CSF, often referred to as simply CSF. It is measured using a target with black and white bars, or sinusoidally varying white light intensity. The other two functions represent the colour CSF of the human visual system (HVS). They are measured using test

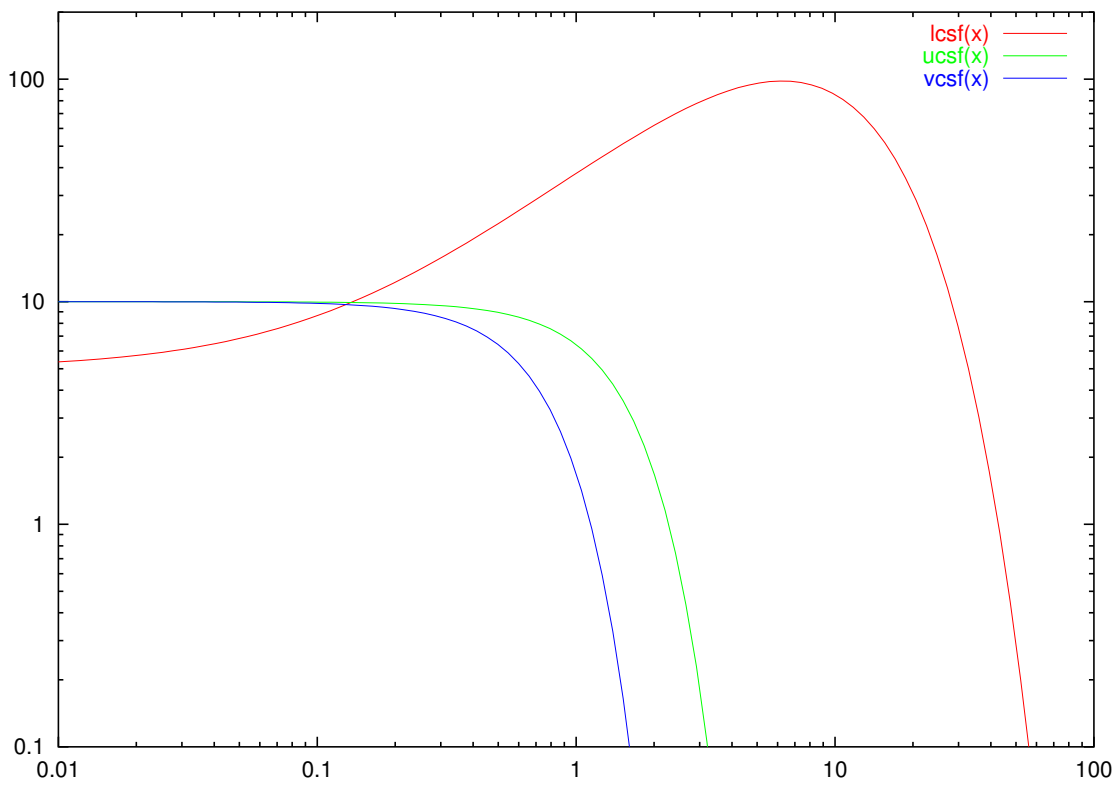


Figure 8.1: CSF function

patterns that are carefully designed to change colour but keep luminance constant, since any luminance change would make the patterns much easier to see. Of the two colour CSFs, the one that extends further to the right is labelled “ucsf(x)” and indicates the HVS sensitivity to red-green changes. The remaining one, labelled “vcsf(x)”, indicates sensitivity to blue-yellow changes. (We use “u” and “v” in the names because those colour combinations are approximately aligned with colour axes with those names in the colour space to which we will eventually convert the images.)

Note that both scales are logarithmic. The X axis is spatial frequency in cycles per degree (CPD). Each textbook seems to show a slightly different graph of the luminance CSF, but the general shape is always similar. (See [76] pp. 201–237, [2] pp. 38–60, [16] p. 31, [13] p. 185, [65].)

The luminance CSF peaks at a value of about 100–250 at a frequency somewhere between 5 and 10 cycles/degree. This means that at that optimum frequency, the smallest modulation we can see is about 0.5%. If the target is alternating light and dark bars, the light bars have only 1% higher intensity than the dark ones. (If the peak modulation is 0.5%, the peak-to-peak modulation is 1%). Below the frequency of peak sensitivity, the CSF falls off slowly. However, above the peak frequency the CSF drops rapidly. When the CSF reaches 1, that means a 100% contrast pattern is just barely visible, and this happens around 50–60 CPD (1 cycle per arc minute) for someone with normal vision.

To make use of the CSF in our algorithm, we need a model of it. Mannos and Sakrison [42] created just such a model by performing experiments on a set of human subjects, then empirically fitting a mathematical function to the measured data. The function is

$$\text{CSF}(f) = 2.6 * (0.0192 + 0.144f) \exp(-(0.144f)^{1.1})$$

The Mannos/Sakrison function is normalized so its peak is (almost) 1.0. Figure 8.1 actually shows this function multiplied by 100, to give values more representative of the measured (unnormalized) contrast thresholds actually found.

Rushmeier et al. [63] use the Mannos/Sakrison CSF model in their “Model 1” image metric. Gaddipatti et al. [21] also use the same model in their work, though it is written slightly differently. We use it for our own Phase 5 algorithm, too.

Colour CSF models are harder to obtain. The two shown in Figure 8.1 were obtained by hand-fitting a Gaussian function to the colour CSF graphs found in Figure 1-18 on page 31 of Fairchild’s book [16]. If you look at Fairchild’s graph, you will see that his luminance CSF looks somewhat different than mine. This is because I have used the Mannos/Sakrison function for the shape of my luminance CSF rather than trying to fit Fairchild’s. However, I have scaled the Mannos/Sakrison CSF so that it has the same maximum amplitude as the one in the Fairchild graph, so amplitude comparisons are correct.

It is worth noting that the domain of the CSF function is in units of cycles per degree. The spatial frequencies that we have been discussing in the context of sampling, resampling, and the wavelet transform are in units of cycles per pixel. Converting from one to the other requires making assumptions about image size and viewing distance. This will be covered in more detail later in this chapter.

Note the large difference in the highest visible frequency between the three functions. The HVS response to luminance information drops to nothing around 60 cycles/degree while the two colour CSFs have upper limits at about 2 and 4 cycles/degree. Above those limits, we see only the luminance component of whatever image we are looking at. Fine-scale colour differences without corresponding luminance changes simply disappear.

The eye/brain system is very nonlinear in its response, so a particular CSF applies only at a particular overall illumination level (see [76] p. 230). The curves above apply to bright conditions, and we assume

that digital images are usually viewed under such conditions, so it is appropriate for our purposes. We make no attempt to model the change in the CSF at low illumination levels.

Also, this discussion has assumed that the same CSF applies to any orientation of the test pattern. It does not; the visual system is more sensitive to patterns that have their lines horizontal or vertical than it is to patterns with lines on a 45 degree diagonal. A better model of the CSF would treat it as a 2D function of horizontal and vertical frequency; see [65] and [13]. However, the effect is small. The method we use does not distinguish between different orientations of a signal, so we would not be able to use the additional information in any case without changing to an orientation-sensitive filter.

8.2 Method

Here is a brief overview of our method for this phase: We read in the two images and convert them via an intermediate space from the *RGB* colour space into CIE $L^*u^*v^*$ which is designed to be more perceptually uniform than *RGB*. The three colour components of each image are filtered into octave-spaced frequency bands. We compare the components of the image band-by-band, calculating a numerical “distance” measure for each colour channel and band. We optionally generate a “difference map” image that shows where the two images differ, in both frequency and spatial terms. Finally, we weight the differences measured in each band according to the visibility of those differences and generate a single “image distance” number.

8.2.1 Initial Processing

Our first task is to read in the two images. Any non-linear encoding used in the image file is decoded, so the floating-point values in memory are linearly proportional to light intensity in the scene. We immediately convert colour images from the *RGB* colour space to the standard linear CIE *XYZ* colour space. We assume that the *RGB* colour space is really the standardized *sRGB* space [68]. This is increasingly true of digital camera images today, and a reasonable guess for images that have no colour space information in them. (However, input images that contain real colour space data such as an embedded ICC profile really should be converted to *XYZ* using that data instead.)

If one of the two input images is monochrome, or if a user of the software explicitly requests it, we switch to monochrome-only mode. In this mode, only the Y component of a colour image is retained after conversion to *XYZ*. When the input image is already monochrome, we use it as Y directly. (The *sRGB* to *XYZ* transformation is scaled in such a way that Y equals the original intensity when $R = G = B$).

When used as part of the pipeline, Phase 5 is normally passed a copy of the transformation matrix that was used by Phase 4 to resample one of the images. Because of our desire to always upsample images (described in Section 7.2) either the first or second input image has been resampled (sometimes both) and the resampling process normally results in some areas of the output image that are blank because they are beyond the bounds of the original image. The transformation matrix file tells us which pixels in the image are filler because of this. We use that information matrix to build a “mask” of pixels that are invalid in at least one of the two input images. If both images were resampled, we create a mask that is the logical OR of the masks from the two images. These “masked out” pixels will not participate in the image comparison.

Once the invalid pixel mask is built, we normalize the two images so they have the same mean luminance. The HVS tends to ignore minor variations in intensity scaling (exposure variations in photographic terms) and this step eliminates most of the effect of any such variations.

We first scan through each image, determining their mean and maximum luminance. These calculations skip any masked pixels so they are based on the same content in both images. (We use only the Y channels to calculate the mean). Then we calculate how much to scale the intensity of each image to make their mean luminance the same. Normally, we increase the luminance of whichever image is darker. However, if both images have a maximum luminance of 1.0 or less (i.e. they are not high-dynamic-range images), we may reduce the scale factors in order to prevent the maximum luminance of either image exceeding 1.0, which helps when displaying them later. This may result in one image being made brighter and the other made darker, but they will end up with the same mean luminance. All three channels of one image are scaled equally in XYZ space, which brightens or darkens the image but does not change contrast or colour.

There is another difference we need to remove: Although the validity mask can prevent us referring to invalid pixels at this stage, we are about to perform a wavelet transform on the image. The values stored in these invalid pixels will affect the values of valid pixels in the wavelet transform up to a distance equal to the filter kernel width. Since the effective filter kernel doubles in size for each successive level of the wavelet transform, differences in a small number of invalid pixels near the edge of the images ultimately cause many of the pixels in the highest level of the wavelet transform to be different. To avoid this “error spreading” effect, we must force invalid pixels to be identical in both images before beginning the wavelet transform. We do this by setting all invalid pixels in both images to a neutral grey whose intensity matches the mean intensity of the freshly-normalized images.

Finally, we do another colour space conversion, from linear XYZ to nonlinear CIE $L^*u^*v^*$ space. $L^*u^*v^*$ is designed to be approximately perceptually uniform. This means that if there is a just noticeable difference between two colours, the Euclidean distance between the location of these two colours in $L^*u^*v^*$ space is approximately the same anywhere in the space (i.e. for any choice of the two colours). In comparison, RGB and XYZ are highly non-uniform in this respect, so $L^*u^*v^*$ is a better space for comparison.

L^* is a nonlinear “lightness” value that is a better representation of how the eye perceives brightness. CIE L^* is defined by

$$L^* = \begin{cases} 116\left(\frac{Y}{Y_n}\right)^{\frac{1}{3}} - 16 & \frac{Y}{Y_n} \geq 0.008856 \\ 903.3\left(\frac{Y}{Y_n}\right) & \frac{Y}{Y_n} < 0.008856 \end{cases}$$

where Y_n is the Y value of the “reference white” for the scene. This function is approximately the cube root of the ratio between a given intensity and the intensity of reference white. The cube root function is slightly scaled and translated and then “spliced” to a linear segment at very low intensity values. The splicing is done to give function value continuity at the splice point. A cube root function alone has the problem that its slope is infinite at $Y = 0$, and the addition of the linear portion avoids that. However, the function as a whole is not far from a cube root. (The functions for calculating $L^*u^*v^*$ come from Sections 8.6-8.8 of [30]. Also, [26] and [12] discuss colour transforms.)

In our case, intensities are implicitly scaled so that $Y_n = 1$, so the denominators in the expression vanish. Also, L^* is defined to have a value of 100 at the reference white, while we prefer to have the reference white remain 1.0. Thus, we actually calculate $L^*/100$.

Rushmeier et al. [63] and Gaddipatti et al. [21] convert intensity to L^* in their algorithms, using a simplified version of the function above (without the linear segment).

Obtaining u^* and v^* is more complex. First we normalize XYZ to intensity-independent xyz :

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

At this point, z is redundant ($z = 1 - x - y$) and is discarded. Now we calculate u' and v' (defined by the CIE in 1976) by:

$$u' = \frac{4x}{-2x + 12y + 3} \quad v' = \frac{9y}{-2x + 12y + 3}$$

Note that u' and v' are independent of light intensity; they are a pure colour measure. These are now scaled by the nonlinear L^* , since perceived colour difference depends on lightness, to give u^* and v^* :

$$u^* = 13L^*(u' - u'_n) \quad v^* = 13L^*(v' - v'_n)$$

where u'_n and v'_n are the u' and v' for the scene's reference white. They are calculated from the xy chromaticity of reference white. Here we use the D_{65} white, which is specified by the *sRGB* image format, which in turn is borrowed from the ITU BT.709 standard [5].

Now we can calculate the colour difference ΔE_{uv}^*

$$\Delta E_{uv}^* = \sqrt{(\Delta L^*)^2 + (\Delta u^*)^2 + (\Delta v^*)^2}$$

Because we actually calculate $L^*/100$, and because u^* and v^* are linear functions of L^* , the u^* , v^* , and ΔE^* we calculate are all a factor of 100 smaller than the standard definitions. This does not cause any problems.

Rushmeier et al. [63] and Gaddipatti et al. [21] work with monochrome information only, so they only need L^* .

8.2.2 Multiband Filtering

Now we filter each image into a sequence of frequency bands, in order to separate information at different spatial frequencies. In colour mode, we filter the three colour components individually. We use the same Mallat-Zhong wavelet transform that we used in Phase 1. However, here we're effectively using it as just a bank of bandpass filters; we're not trying to find edges.

This wavelet transform is better for this purpose than most other wavelet transforms, for many of the same reasons we like it for Phase 1. Each frequency channel (one level of the wavelet transform) is an image the same size as the original image, so they can easily be presented side by side. The filters used are symmetric or anti-symmetric around their centres, so they don't distort or shift the image in the process of filtering it (except for a half-pixel shift because the level-1 filters are even length). The smoothing filter is a cubic spline shaped like a Gaussian, so the shape of the frequency response of the filter is well-behaved.

Gaddipatti et al. [21] also use a wavelet transform as a bandpass filter, again so that they can apply a CSF model to weight image content according to visibility. However, they use one of the Daubechies family of wavelet transforms [14]. These transforms decimate by a factor of 2 after each level, so higher levels are successively smaller in size. This is fine for numerically calculating a difference, but not very useful for displaying the difference to a human viewer. Also, the Daubechies wavelets are neither symmetric nor smooth, so they suffer more aliasing when used as filters.

We can numerically measure the frequency response of this wavelet transform used as a filter, using a method that will be described in Section 8.4.1. Figure 8.2 is a graph of the response for a 1024 pixel wide image, so there are 10 bands (levels) present. Level 1 contains the highest-frequency information and its curve is the rightmost one. Other levels appear in sequence from right to left. The horizontal scale reads frequency in cycles per pixel. The vertical scale is mean square signal after filtering.

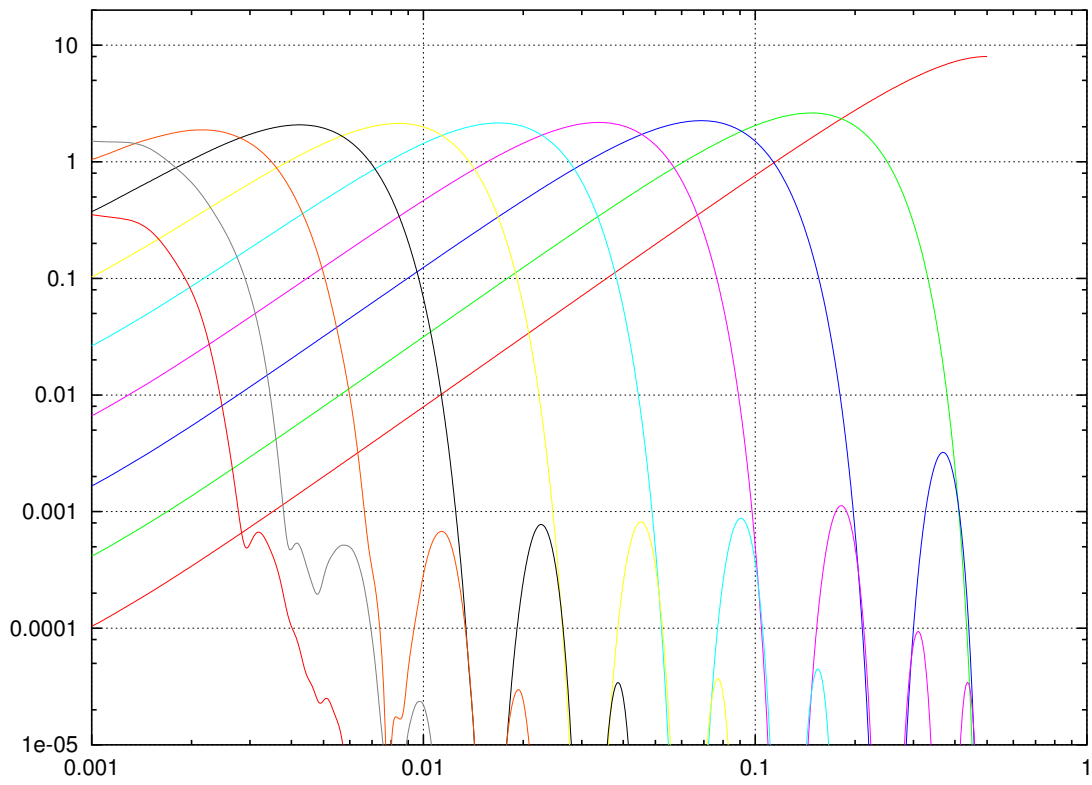


Figure 8.2: Frequency Response of Wavelet Transform

Note that levels 2 through 8 have almost exactly the same shape of response curve. Level 1 is unusual because it is produced by the high-pass G filter only; all other levels are the product of one or more instances of the low-pass H filter combined with the G filter. Levels 9 and 10 are unusual partly because of the effect of the image edge on such a wide filter (The H filter used in producing the level 10 image is wider than the source image) and partly due to problems measuring such low frequencies (at the left edge of the graph, the frequency is one cycle per image width).

(Note that here we are only talking about the response of the wavelet transform as a filter. The process that formed the digital image involved a low-pass filter of some sort, and this limits the frequency response of the entire system. Thus, even level 1 has a bandpass response with respect to the original scene, not high-pass.)

It is worth noting that the frequency response in Figure 8.2 applies exactly only to image content that is oriented either horizontally or vertically, not diagonally. With horizontal or vertical content, the information appears in either the X-wavelet or Y-wavelet at the appropriate levels, but not both. Signals (modulation) with other orientations appear in both wavelet outputs with a frequency that is lower than its true frequency. (Remember that the X-wavelet is really the partial derivative of the signal, after filtering, in the X direction). Suppose we have a signal with frequency f whose orientation is θ , where $\theta = 0$ means that the wavefront is vertical. The wavelet transform will “see” this as two signals, an X-direction signal with frequency $f \cos(\theta)$ and a Y-direction signal with frequency $f \sin(\theta)$. So diagonally-oriented signals are reported as if they had a lower frequency, but possibly a higher amplitude (because of the contribution to both X and Y wavelets), than is actually the case. This is a fundamental limitation of the Mallat-Zhong wavelet transform used as a filter — it provides no angular discrimination.

The small “bumps” along the bottom are unwanted responses. For example, if you find the level 2 curve and follow it to the bottom of the graph near the lower right corner, the bump immediately to the left of it is actually part of the level 3 curve, the one adjacent to the left. However, the peak of these spurious responses is 1/1000 as large as the main response at that frequency (30 dB down in engineering terminology). These responses are too small to be visible in the pictorial output, and we don’t expect them to have any significant effect on the numerical output either.

8.2.3 Displaying Differences

After filtering the two images, we subtract the result at each level to find the difference. (In colour mode, we calculate three differences). As explained in Chapter 4, the X- and Y-direction wavelet transform data at each level is effectively a gradient vector for the filtered image. When subtracting these, we do a vector subtraction, and the result is a vector. The length (modulus) of the difference vector is interpreted as the magnitude of the difference between the images — this is the main value we want. The orientation (argument) of the difference vector doesn’t seem to have any particularly useful meaning.

Once we’ve calculated the differences between the two images for each frequency band, we generate a “difference display” for the user to look at. From this display, the user can see where the important differences are. At present, we only display luminance differences in this way. We have not decided how best to incorporate colour differences.

For each frequency band, we display the two (luminance) images that are input to this level of the wavelet transform for reference. In between these two, we display an image of the difference.

The outer two reference images have had all of the low-pass filtering that will be used at this level already applied, so any high-frequency detail that is not going to participate in the difference at this level is

not present. However, the high-pass G filter has not been applied yet, so low-frequency content that will not be used is still visible. We could have displayed the two bandpass images (after the G filter) as the reference images, but we feel that they make a less useful reference for the user. Images stripped of low-frequency content look strange; for example, look at the fourth column in Figure 4.3. The fact that the G filter is a differentiator makes it worse. So, we use the low-pass images for reference.

The next several figures show an example. The test images that we have used in previous chapters (mostly the mandrill rotated or scaled or translated) don't make a very interesting example here, since the two images align nearly perfectly and there is very little to look at in the difference maps. Instead, we took a couple of photographs of a (messy) living room scene taken with a handheld camera. The camera moved some between exposures, and the two images cannot be perfectly aligned, so there are residual differences to look at. Figure 8.3 shows the two original images. Figure 8.4 shows level 1 (top) and level 3 (bottom) difference images. (Level 2 isn't sufficiently different from either of these to be interesting). From them, you can see that most of the difference at these scales comes from the lower left corner, where alignment is poorest. Still, most of the image matches well.

Figure 8.5 shows levels 4 and 5 of the difference. There is something interesting happening here — the largest difference in level 4 is near the upper right, on the fireplace mantel. It is even more obvious in the level 5 image; whatever it is totally dominates the difference at that scale. If we go back to the original scenes, we can see that someone removed an object (a stereo speaker) between the two photographs. Although this didn't result in much difference in the finer scales, the object is the right size to appear in level 5, and there it is very obvious. (The missing object is also the largest difference in level 6, but it is not as dramatically larger than everything else as it is in level 5.)

We believe that, in most cases, showing the user where the original images are different, including a set of displays that function as a “difference map”, is actually the most useful product of the entire system. The single “image distance” number that we calculate necessarily discards almost all this information. Daly's Visible Difference Predictor [13] can provide a visual display of where the images are sufficiently different for the differences to be visible, but his difference display does not separate differences according to spatial scale.

8.2.4 Calculating Distance

At first glance, calculating a numerical “image distance” is easy. Within a single level (frequency band), we add up the sum of the absolute value of the differences at each pixel and then divide by the number of pixels (so the result is independent of image size). Then we calculate a weighted sum of these mean differences, with the weights being proportional to the relative visibility of the frequencies in each band, to produce a final single number.

However, “adding” the differences shouldn't be done as a simple sum, because of the way signals of different frequencies add. Suppose we have a single sine wave of some arbitrary frequency and amplitude 1.0. The mean is zero, which isn't very interesting. The mean absolute value is $2/\pi$ or 0.6366, which is more promising (particularly for us, where the wavelet transform is a vector and we take the modulus of the vector, thus effectively calculating the absolute value of the signal). If you add another sine wave of the same frequency and phase, you really have just a single sine wave of double the amplitude, and the mean absolute value also doubles. However, if you add together two sine wave of the same amplitude but different frequencies, you get a more complex waveform. The peak amplitude is 2.0, but the mean absolute value increases by a factor of only 1.25. (This is true for two unrelated frequencies; it isn't usually true when the



Figure 8.3: Original Living Room Scenes



Figure 8.4: Level 1 and 3 Difference Images



Figure 8.5: Level 4 and 5 Difference Images

frequencies are related by a rational factor). Mean absolute value clearly isn't a very good way of summing the contributions of signals at different frequencies.

Instead, the mean square signal is normally used. A single sine wave with amplitude 1.0 has a mean square of 0.5. The sum of two sine waves, each with amplitude 1.0, gives a complex waveform whose mean square is 1.0, as long as the two frequencies are different. Adding three signals of different frequency gives a composite signal whose mean square is three times that of the individual signals, and so on. This is what we need.

However, mean square has an oddity: when you double the amplitude of a single sine wave, the mean square becomes four times as large. This is really a fundamental difference between adding signals of the same vs. different frequencies that can't be avoided. To help deal with this, the root mean square (RMS) value of a composite signal is often used.

In the physical world, if the signal is actually a voltage or current waveform, these measures have physical meaning. The mean square of the waveform is proportional to the mean power carried by that waveform. (Because of this, you will sometimes find signal squared referred to as "power" in signal processing texts, even when the signal isn't actually a voltage or current.) The RMS value of the waveform is an equivalent steady-state voltage or current for the waveform. For example, if a complex waveform has a RMS voltage of 100 V, then the average power it delivers to the load is the same as a constant 100 V DC (direct current) power source. In contrast, the mean absolute value of a waveform really isn't useful for very much. In the physical world, adding multiple signals of different frequencies really does give a signal whose power is the sum of the individual signal powers. The (RMS) voltage of the sum is not the sum of the individual voltages, it is the square root of the sum of the squares of the individual voltages. (This is a consequence of the definition of RMS voltage as being the DC-equivalent voltage). On the other hand, doubling the amplitude of a single-frequency signal really does deliver four times the power to the load, and the voltage is doubled.

Now, we're not processing voltages, we're dealing with light intensity. Our "signal" is not even linearly related to light intensity, but is rather the cube root of intensity to model perceptual effects. Still, it seems clear that we should calculate the "size" of the difference between the two images in a single frequency band using the mean square, since the difference image is generally composed of many signals with many frequencies.

Similarly, when we combine contributions from different frequency bands, normally different frequencies are present and we should be calculating a weighted sum of squares. (The actual values we use for the weights is the subject of Section 8.4 below.) Adding squared signals is also the correct way to add the differences from the three colour channels within a single level, since that is how the colour space was intended to be used (see the definition of ΔE^* above).

We report squared differences for each level and colour channel to the user, as well as a per-level sum and a per-channel weighted sum. This is consistent with other error measures that report mean squared differences. However, when we report the final overall total "distance" we display both mean square and RMS values. The RMS value is our "image distance", since it has a chance of satisfying the triangle inequality (the fourth axiom of a metric).

8.3 Spatial Frequency to Angular Frequency Conversion

We wish to weight the "image distance" values calculated for each frequency band according to how visible that band of frequencies is to the human eye. Our first problem is that the wavelet transform (or any similar

approach based on a set of digital filters) yields frequency bands aligned according to frequency in cycles per pixel. For example, in a power-of-two frequency subdivision, the highest-frequency band will always represent approximately 0.25 to 0.5 cycles per pixel. However, this is a variable number of cycles per image width or image height, since the image size in pixels varies from one image to the next. When images are printed or displayed on a screen, this becomes cycles per unit length, depending on the physical size of the paper or screen. Then the viewer looks at the paper or screen from some viewing distance, and only then can the signal be expressed as cycles per degree at the viewer's eye.

Note: the cycles per degree (CPD) actually changes across the span of the image, since off-axis pixels are foreshortened and this increases the apparent CPD. In the following, we will work in terms of CPD at the centre of the image, implicitly assuming that the viewer's eye is located on the centre axis of the print or screen, and the viewer is looking at the centre of the image. We will also assume that the image is displayed flat, not on a cylindrical or spherical surface. The calculations for off-axis viewing and non-flat images aren't particularly difficult, but they would have negligible effect on the results for normal viewing angles, so we ignore them.

The human CSF measures response in units of cycles per degree, so we need to convert between cycles per pixel and cycles per degree to use it, and the conversion necessarily involves knowing both the image size in pixels and the viewing angle subtended by the image at the eye. The viewing angle, in turn, could be specified directly, or computed from the image physical size and viewing distance. We use the viewing angle directly. By default, we assume a horizontal field of view (HFOV) of 28.1 degrees, which is the angular width of a common 4 × 6 inch print viewed from a distance of 12 inches. The user may specify any plausible HFOV for different viewing conditions.

If F_H is the horizontal field of view, W is the width of the image in pixels, f_p is a frequency in cycles per pixel, and f_d is a frequency in cycles per degree, we can convert f_p to f_d using the following steps:

$$\begin{aligned} a &= \tan(F_H/2) \\ b &= \frac{2a}{W} \\ c &= \frac{1}{\tan^{-1}(b)} \\ f_d &= cf_p \end{aligned}$$

$F_H/2$ is the angle subtended by half the screen width at the viewer's eye. The tangent of this angle is the ratio of half the screen width to the viewing distance. Dividing this quantity by the number of pixels in half the image width gives us the ratio of a single pixel width to the viewing distance — in other words the tangent of the angle subtended by a single pixel at the centre of the screen. Taking the arctan gives us the degrees per pixel at the centre of the screen. The reciprocal of that is pixels per degree. Multiplying a frequency in cycles per pixel by pixels per degree gives us cycles per degree. Now we can find the value of the CSF for any frequency in cycles/pixel, under the given viewing conditions.

8.4 Calculating Weights

With frequency conversion dealt with, how should we calculate the weights for the wavelet transform levels? Gaddipatti et al. [21] use a particularly simple approach: They (in effect) assume that their wavelet transform provides perfect separation between frequency bands, so any particular frequency in the image will appear in exactly one level of the wavelet transform. They assume that each level of the wavelet transform captures

exactly one octave of frequency space, and that the frequency-space boundaries between the filter bands occur at exact integer powers of 0.5 cycles/pixel. In other words, they assume that the first wavelet transform level yields exactly (and only) content at frequencies from 0.25 to 0.5 cycles/pixel, the second level contains 0.125 to 0.25 cycles/pixel, and so on. (If you look at the Gaddipatti paper, the boundaries are labelled π , $\pi/2$, $\pi/4$, etc. These are frequencies in radians/pixel, equivalent to 0.5, 0.25, 0.125, ... cycles/pixel.)

Gaddipatti et al. also assume that their wavelet transform has equal amplitude response at all levels — that no matter which band of the filter a signal appears in, its amplitude will be the same. Given all these assumptions, the weight applied to each level of the transform can be obtained simply by determining the mean amplitude of the CSF function over the frequency range spanned by this level of the transform. This is shown in Figure 1(a) in the paper. These weights can't be precomputed, since a change in image size or HFOV moves the locations of the frequency band boundaries in the CSF. But once the size and viewing conditions are known, it takes only a few quick integrations to determine the weights.

However, these are pretty optimistic assumptions. No wavelet transform is going to provide perfect frequency separation, since the filters used have compact support. We don't know whether the particular wavelet transform that is used in the Gaddipatti paper actually meets the other two assumptions, but they are certainly not true of the Mallat-Zhong wavelet transform. Thus, whether or not this simple method does a good job calculating weights for the application in the paper, it is not a good choice for us.

8.4.1 Measuring Frequency Response

To do better, we begin by measuring the actual frequency response of the Mallat-Zhong wavelet transform. We calculate it in 1D, since for horizontal and vertical information the signal is effectively constant to the wavelet transform filters operating in one of the two orientations (this is a consequence of all filters being separable).

We fill a buffer with a discrete sine wave at some specific frequency. Then we multiply the signal by a function designed to reduce its amplitude smoothly to zero at the ends, eliminating the transient in the signal that would occur except when the buffer is an integral number of wavelengths long. We use the Hann (or “Hanning”) window. If our buffer contains N samples indexed from 0 to $N - 1$, the Hann window for N samples is defined by

$$\text{Hann}_N(x) = \begin{cases} \frac{1}{2}(1 - \cos(2\pi\frac{x}{N-1})) & 0 \leq x \leq N - 1 \\ 0 & \text{otherwise} \end{cases}$$

We rescale these values so that the mean square of the window values is 1.0; this prevents the windowing operation from changing the overall mean squared signal. Then, as each different frequency of sine wave is generated, we multiply it by the window before performing the wavelet transform.

Windowing is described in *Discrete-Time Signal Processing* by Oppenheim and Schaffer [53], and in *Digital Image Warping* by Wolberg [79]. It is often described as a way of shortening infinite-length functions like $\text{sinc}(x)$ for use as a digital filter kernel, but it is just as useful for removing boundary effects from synthetic signals before filtering. Pratt [55] discusses the necessity of windowing signals before Fourier transform filtering in Section 9.4.2 of his book.

We perform all levels of the Mallat-Zhong wavelet transform. Then we measure the level of that signal in the output at each level. We actually measure the mean square signal, since it is squared signal that we will be summing later. We repeat this for a wide variety of frequencies, typically from one cycle per image

width ($1/W$ cycles per pixel) up to 0.5 cycles per pixel. These frequencies are equally spaced in the log frequency domain by default. The result is the frequency response shown earlier in Figure 8.2.

As we noted before, the lowest and highest frequency bands have quite different frequency response curve shape from the rest of the bands. There is quite a bit of overlap between bands; at any given frequency (except for the top two octaves) there are three or four bands with significant response to that frequency. Note that the “crossover frequencies” where there is a change in which band has the greatest response occur at frequencies of 0.182, 0.091, 0.0455, 0.02275, etc. cycles/pixel rather than the 0.25, 0.125, 0.0625 sequence that one might have expected — nearly half an octave shift.

Finally, note that the shape of the individual curves is quite asymmetric, with a long straight constant-slope left flank and a steep right flank. This is because the curve is actually due to the sequential application of two very different filters, and the response is the product of their individual responses. The left flank is due to the response of the G filter, which is basically a differentiator. Used as a filter, a differentiator has a straight-line response of 6 dB per octave. In other words, every time the frequency doubles, the output signal amplitude doubles, and the squared output signal is four times as large. This gives a straight line on a log-log graph such as Figure 8.2.

The right flank of the frequency response is due to the low-pass H filter. Since the H filter is roughly Gaussian in shape, its frequency response should also be roughly Gaussian (the Fourier transform of a Gaussian is a Gaussian). It certainly looks Gaussian, on a graph with linear axes (not shown).

8.4.2 Fitting Weights

Now that we have the measured frequency response of the Mallat-Zhong wavelet transform “filter”, we can proceed with determining the per-level weights. We do this by least-squares fitting, using essentially the same algorithm as we used in Phase 3.

For an image whose largest dimension is W , our wavelet transform has $L = \lceil \log_2(W) \rceil$ levels. If we start with a signal of frequency f , and perform the wavelet transform, we calculate mean square signal values $S_i(f)$, $i = 1, \dots, L$ for each level. Our overall single-number “signal strength” value $S(f)$ is going to be a weighted sum of the S_i values:

$$S(f) = \sum_{i=1}^L W_i S_i(f)$$

Ideally, $S(f)$ would be equal to $CSF(f)$, so we need to determine the set of weights W_i that minimizes the difference between $S(f)$ and $CSF(f)$.

This can be expressed as a linear least squares problem. The “basis functions” are simply the measured frequency responses of the wavelet transform filters S_i evaluated at the set of frequencies that we’ve used. The right hand side is a set of samples of $CSF^2(f)$ evaluated at that same set of frequencies. (Contrast is a signal amplitude measure, and CSF is expressed in terms of contrast. If we scale amplitudes in proportion to CSF, we need to scale mean squared signal in proportion to CSF^2 .)

In other words, if the problem is written in matrix form as $Ax = b$, the A matrix is simply the table of signal output vs. transform level and frequency that was used to create Figure 8.2. The b vector is the corresponding set of $CSF^2(f)$ values, and the x vector is the set of W_i we’re solving for. When we are working in colour, the b vector simply expands to a three-column matrix, with one column containing each of luminance, red-green, and blue-yellow CSF values. Similarly, the x vector becomes a three-column matrix, with one column giving the weights for each colour component.

Level	L^*	u^*	v^*
1	622.0	0.02	0.01
2	3490.0	0.29	-0.11
3	50.5	-1.25	0.44
4	319.0	8.23	-1.36
5	-50.5	25.50	8.31
6	62.5	23.90	25.70
7	-8.0	24.30	23.30
8	30.6	25.70	26.60
9	-63.7	7.77	4.83

Table 8.1: Fitted Weights

We solve this system using the SVD, as before. For an image 1024 pixels wide and the default 28.1 degree field of view, we obtain the set of weights in Table 8.1. Note how far some of these weights are from resembling heights of the CSF function — some of them are actually negative! (Since the weights may be negative, we should more properly call them factors or coefficients, not weights).

Figure 8.6 shows the desired (lightness) CSF for the particular viewing conditions we assumed, and the frequency response that we obtain using the factors in Table 8.1. The fit is quite good except at the very highest frequencies. There isn't anything that the least-squares fitting can do about that; only two bands of the filter (1 and 2) have any output at those frequencies and their shapes interact to produce the shape shown at the upper end. Adjusting factors can move the anomaly up or down, but not change its shape.

It is worth noting that calculating the frequency response data is fairly expensive. We generate something like 1000 sine waves of various frequencies and as long as the image is wide, then perform all levels of the wavelet transform, and measure the output signal. However, once we've calculated this for a given image width, it is fast to recalculate the factors if the field of view angle changes. The data in the A matrix uses frequencies expressed in cycles per pixel, and that doesn't change with FOV. The b vector contains CSF values, and those do change because the pixels per degree changes when FOV changes. Thus, we just need to build a new b vector and redo the least-squares fit, which is fast.

With per-level factors in hand, we simply weight the mean square difference from each level and colour channel by the appropriate weight before adding them to the grand total.

8.4.3 Simulating Frequency Response

As noted above, actually measuring the frequency response of the wavelet transform is moderately expensive. We can't just precompute this once and store it, since the number of levels, and the shape of the response of the two highest levels, depends on the image width. However, when we look at how the response graph changes as a function of the image width, all of the changes happen at the highest levels. As the width increases, the X axis of the graph (log frequency) extends further leftward. The highest level present (level 10 in Figure 8.2) increases in amplitude and changes shape to look more like the lower levels. After a factor of 2 increase in resolution, a new level appears, level 11, looking exactly like level 10 did when we started. Reducing resolution, the inverse changes happen — the highest level changes shape, shrinks and ultimately vanishes. All throughout this, all but the highest two levels or so remain almost completely unaffected.

This observation suggests that one could store a table of frequency response for a particular image size, and then alter it via a simple algorithm to approximate the correct response at any other image size.

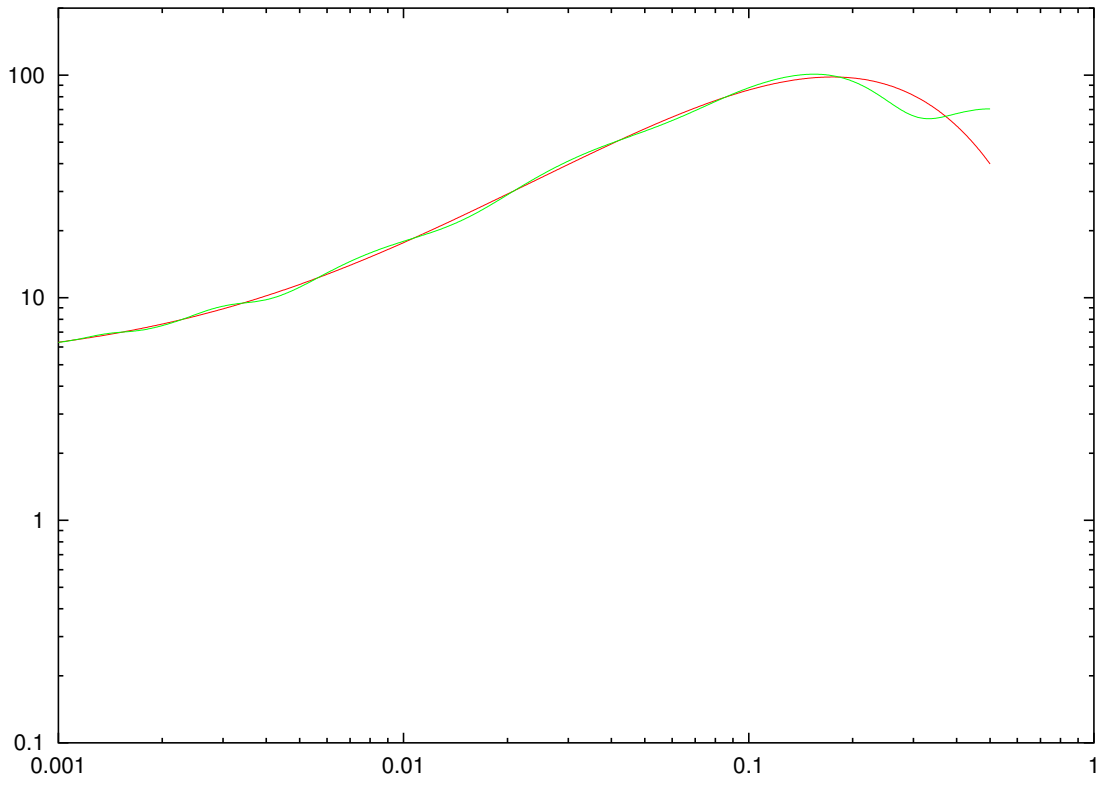


Figure 8.6: Luminance CSF and fitted frequency response

However, we took a different approach. As we observed before, the response of most of the filters is a composite of that of the G (high-pass differentiator) and H (low-pass) filters. We assumed that we could find a simple model for these filter's transfer functions, and fit a few parameters to reproduce all of the frequency response functions.

The G filter behaves like a differentiator. A simple analog electronics differentiator consisting of one resistor and one capacitor has an amplitude response given by

$$R_G(f) = \frac{1}{\sqrt{1 + (f_1/f)^2}}$$

where the parameter f_1 is the filter "corner frequency".

We supposed that the H filter has a Gaussian transfer function. Thus, it should be able to be modelled by

$$R_H(f) = \exp\left(-\left(\frac{f}{f_2}\right)^2\right)$$

where the parameter f_2 is a frequency scaling adjustment. When the H filter responses are calculated (without the G filter) and plotted on the right sort of graph (log negative log amplitude vs. log frequency), we get a long straight section with slope 2, confirming the exponent of 2 in the model above. However, the slope increases to near 3 at the right end of each line, and that is where the composite response is mostly controlled by the H filter, so we're likely better to use an exponent of 3 rather than 2 in R_H .

Now, we suppose that the overall transfer function is given by the product of these two transfer functions plus an additional amplitude scaling factor A , so our model of the overall transfer function for one level of the wavelet transform is

$$R(f) = AR_G(f)R_H(f) = A\frac{1}{\sqrt{1 + (f_1/f)^2}} \exp\left(-\left(\frac{f}{f_2}\right)^3\right)$$

However, that is a model for the amplitude response, and we actually measure squared signal, so we need to square the transfer functions too. Our new model is

$$R^2(f) = AR_G^2(f)R_H^2(f) = A\frac{1}{1 + (f_1/f)^2} \exp\left(-2\left(\frac{f}{f_2}\right)^3\right)$$

We simply hand-fitted a set of ten of these functions, one per level, to the set of data for a 1024-pixel wide image. Level 1 is unusual because it doesn't use the H filter, so the effect of the Gaussian is simply eliminated for that level. Table 8.2 shows the fitted coefficients. To adjust this model for different resolutions, there is a set of simple scaling rules for deleting levels or adding them, adjusting frequencies and amplitudes. Figure 8.7 shows the modelled frequency response for an image width of 1024 pixels at the default viewing distance. Compare this to the measured response shown in Figure 8.2.

With this simple model of the frequency response, it is very fast to calculate a set of responses to a full set of frequencies, then do the least-squares fit to obtain the factors we need. We use this approach in Phase 5, instead of actually calculating the frequency response for the viewing distance specified.

Table 8.3 shows a set of factors for the lightness channel calculated using the modelled frequency response, compared to the factors generated from the measured response (repeated from Table 8.1). Figure 8.8 shows the frequency response curve generated by the two sets of fitted factors, as well as the CSF curve desired.

Figures 8.9 and 8.10 show the frequency response that results from a least-squares fit to our red-green and blue-yellow CSF models. The flat portion of the CSF on the left is matched very well, but there are

Level	A	f_1	f_2
1	19.845	0.5	0
2	9.2450	0.174	0.256
3	7.0313	0.075	0.122
4	6.4800	0.0355	0.0605
5	6.2130	0.0173	0.0304
6	5.9500	0.0084	0.0153
7	5.4450	0.00395	0.0077
8	4.5000	0.00185	0.0041
9	2.2000	0.0004	0.0024
10	1.0400	0.00095	0.00182

Table 8.2: Model coefficients

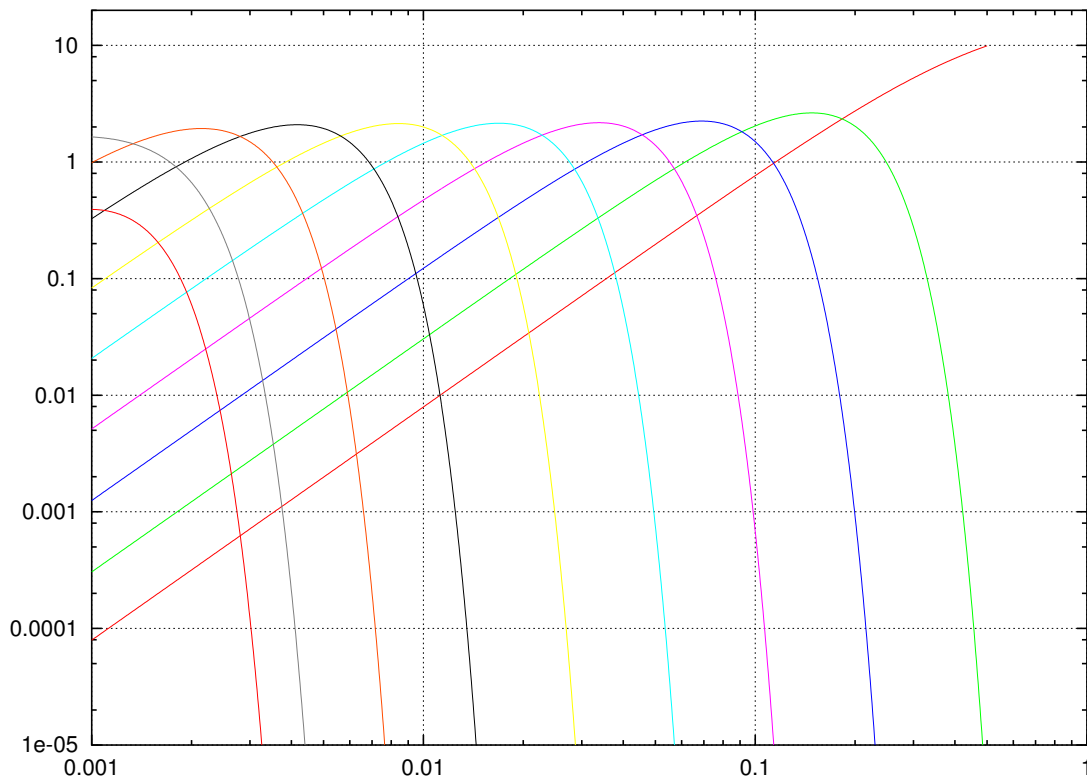


Figure 8.7: Modelled Frequency Response

Level	Measured	Modelled
1	622.0	556.0
2	3490.0	3550.0
3	50.5	46.6
4	319.0	335.0
5	-50.5	-57.9
6	62.5	64.2
7	-8.0	-5.9
8	30.6	19.3
9	-63.7	6.2

Table 8.3: Weights fitted using measured and simulated response

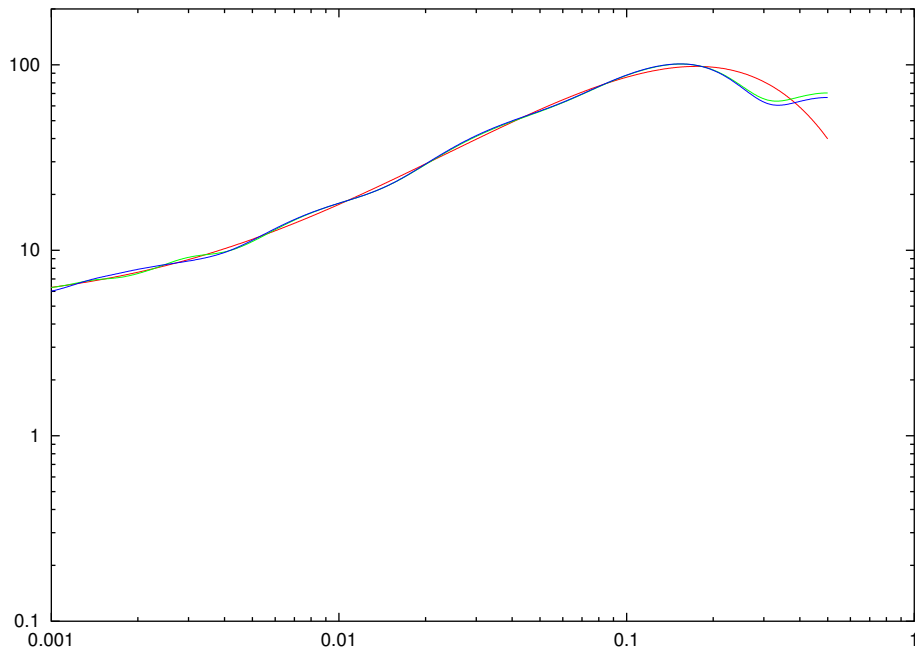


Figure 8.8: Luminance CSF and two fitted frequency responses

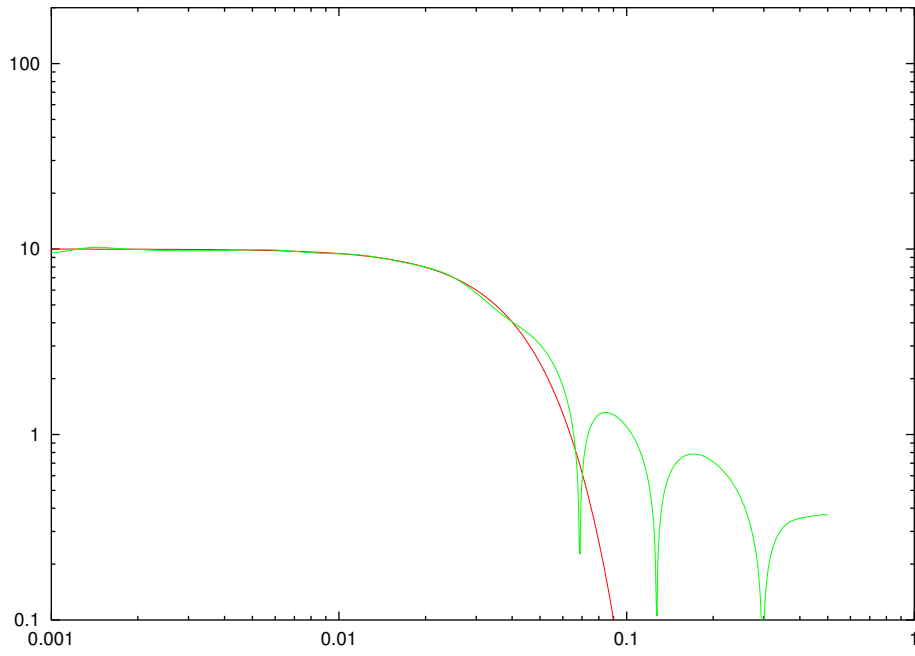


Figure 8.9: Red-Green CSF and fitted frequency response

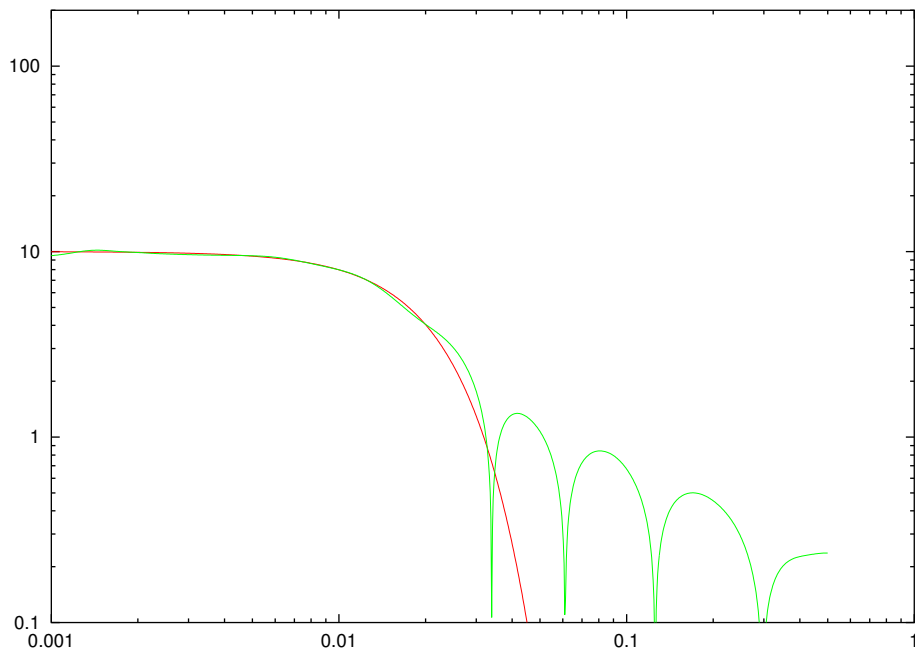


Figure 8.10: Blue-Yellow CSF and fitted frequency response

some significant variations from the desired shape in the area where the CSF is falling rapidly. The errors are mostly in the direction of responding too much to these high frequencies rather than responding too little. This will cause our “distance” to be somewhat affected by high-frequency colour information that is in fact not visible. Still, the unwanted responses are less than $1/5$ the amplitude of the main lobe.

8.5 Verification and Testing

In the Introduction, we listed the axioms that define a metric. We wish to determine how close our Phase 5 algorithm comes to being a metric. Here are the axioms again:

1. $M(A, B) = M(B, A)$
2. $M(A, B) = 0$ if $A = B$
3. $M(A, B) \neq 0$ if $A \neq B$
4. $M(A, B) \leq M(A, C) + M(C, B)$

The method was designed to make the first axiom always true; the internal processing of the two images is identical up to the point where wavelet transform coefficients are subtracted from each other. Swapping the order of the images should negate the vector that results from the subtraction. The only thing we do with the vector is calculate its modulus, which is unaffected by negation of the vector. So, by design, the distance value should be the same regardless of the order of the input images. We verified that this was in fact true for a few randomly-selected pairs of test images. In all cases, all of the numerical distance results (at all levels and for all colour channels) were identical when the image order was reversed.

The second axiom should also be true by design. Again, both images are processed in exactly the same way up to the point of subtraction. If both images are the same, the values subtracted will be identical and the subtraction will yield zero. This will happen at all pixels in all levels in all colours, resulting in a final distance of zero. Testing with a few randomly-selected images shows that this is how the software behaves.

In the case of the third axiom, we do not want it to be literally true. In cases where the only difference between two images is fine detail that cannot be seen at the given image size and viewing distance, we want the distance to be reported as zero, even though the images are not identical. However, if we reinterpret the “not equal” in the axiom to mean “perceptibly different”, then we can satisfy it. Basically, if the two images are perceptibly different at all, our distance measure should not be zero.

The authors of the Mallat-Zhong wavelet transform have proved that it is invertible. Thus, if there are two different images fed to our algorithm, there will be some difference in the output of the wavelet transform at some level (otherwise information would have been lost, and the transform is not invertible). Now, in fact, we only make use of the wavelet outputs at the multiple levels; we do not compare the highest-level smoothed image and so we do ignore some of the information. A DC level shift (an offset added to all pixels) is the only change that one can make that does not affect any of the wavelet outputs, where its only effect is on the smoothed image. Any varying signal that is added, no matter how slowly varying, will cause a change in one of the wavelet levels. If we look at the Phase 5 algorithm as a whole, a DC level shift in one image will trigger a rescaling of all intensities in one image due to the normalization code, so they will not compare equal after all. So, any difference in the images will appear as a difference in the wavelet output. The squared differences that are summed are always positive, so a difference at just one pixel will cause the summed difference at that level to be non-zero, in any colour channel.

These per-level and per-colour sums of squared differences are then added according to the calculated weighting factors. If there is some spatial frequency where the overall output of the system is zero, then the distance will be zero for a non-zero difference in the images at that frequency. An examination of the fitted frequency response graphs (Figures 8.8 through 8.10) shows that the response is non-zero in the region where the CSF functions are of significant size. Changing the angle of view or the image resolution generates a different set of weighting factors, but the statement above appears to remain true over a wide range of values — the response is non-zero in the areas where it is supposed to be non-zero. Thus, any difference in the images that is visible should result in a non-zero distance measure.

We cannot prove that our Phase 5 algorithm satisfies the fourth axiom, called the “triangle inequality”. On the other hand, we have not been able to find an example of the results violating the axiom either. We took a random collection of images and calculated a complete matrix of pair-wise distance values. Then we checked the fourth axiom for all possible permutations of three images out of the set — it was true in all cases. We have tried adding sine waves and white noise to images in varying amounts and calculating the distance from the original. We have checked the effect of adding sine waves and noise on top of each other to an image. We have not yet found a violation of the triangle inequality. Nevertheless, this does not prove that we have a metric.

8.6 Implementation

All of the work in Phase 5 is performed by a single program named `wdiff`.

A utility program called `response` calculates the actual frequency response of the wavelet transform, the response of the H filters only, and the overall response for the system using either of two sets of fitted factors. All of the graphs in the second half of this chapter were produced with data from `response`. However, `response` has no role in comparing images; it was only used to help design `wdiff`.

Both `wdiff` and `response` use functions found in a couple of other files in the same directory. One contains the CSF models and the mapping from cycles per pixel to cycles per degree. The other one implements the model of Mallat-Zhong wavelet transform frequency response described in Section 8.4.3.

8.7 Performance

Running `wdiff` on a pair of 500-square images takes about 13.7 seconds of time (Pentium III 700 MHz). If we disable the “difference map” image output, because we only want the numerical distance results, execution time is reduced to 7.8 seconds. If we request monochrome-only processing, those times drop to 8.8 and 3.0 seconds.

Calculating numbers only on a pair of 2272×1704 pixel digital camera images (the fireplace example from early in this chapter) takes about 270 seconds. The cost of this phase is approximately $O(D^2 \log_2(D))$, where $D = \max(\text{width}, \text{height})$.

8.8 Discussion

As noted, we currently only display differences in the luminance component of colour images, though we calculate colour differences and evaluate them numerically. The individual u^* and v^* channels aren’t very meaningful on their own. Perhaps a 2D colour difference display would work: the original images could be

displayed with colour but constant luminance, and the 2D difference in colour (called ΔC_{uv}^*) displayed in the middle using intensity, or intensity combined with colour. Another possibility is to simply calculate the 3-D difference ΔE_{uv}^* and display that in the centre between full-colour versions of the original image. We should build prototypes of several interfaces and try them.

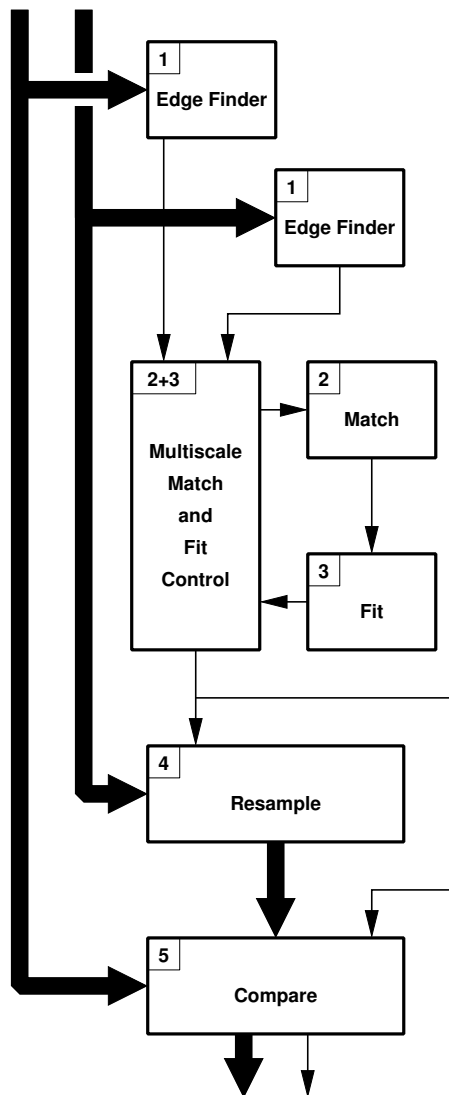
At the moment, `wdiff` simply writes out the “difference map” as a large single image file, and we use other tools (the GIMP image editor) to examine and scroll around the large image. This works well for original images up to about 300 pixels across, since the two originals and the difference map will fit across the width of the screen while being displayed at full resolution. However, above this size, you start losing part of one reference image or the other, and with images above 1000 pixels or so you can’t even see the same region in the difference map and one reference image at the same time. It would obviously be useful to have a “difference browser” application that displays three windows which always show the same portion of the difference image and two reference images. Scrolling one would scroll all three together. It should also be able to zoom in and out on the images and flip between levels, again keeping all three windows synchronized.

In retrospect, the Mallat-Zhong wavelet transform may not have been the best choice of a multiband filter for use in this phase. The G filter is designed as a differentiator, and while that is important for the edge-finding function of Phase 1, it doesn’t do us any good in this phase. In fact, it has a number of problems: the asymmetric shape of the frequency response, the fact that the wavelet transform is vector-valued, and the excessively strange appearance of the wavelet transform due to differentiation.

If we were to re-implement this phase, we would look for an octave-spaced filter bank where the output in each band is a single image (not two), it is simply bandpass filtered (not differentiated), and where the filter passbands have a symmetric shape. One advantage of our pipeline model is that this change would have no effect on any of the other phases.

Chapter 9

Testing and Examples



To this point, we have done some testing of the function of each of the phases in isolation. Now we will test the operation of the first four phases as a unit. The operation of the fifth phase is naturally independent, and we have already done more extensive testing of its operation in Chapter 8.

Then we will try applying the entire system to a pair of real and synthetic images. We'll also demonstrate an application where the first four phases are useful on their own, without the fifth.

9.1 Registration

The first four phases of our method act to align a pair of images. Our testing is designed to verify that the method will work for images that are reasonably well aligned (a pair that a human observer would agree are substantially similar images), as well as explore the limits of the system.

Our first set of tests was of the ability to align images where one has been rotated. Our first test image was a single letter "R" in white on a black background. This provides good contrast, but no information content in the outer portions of the image. We tried rotating this by a series of angles spaced every five degrees, then comparing it to the original unrotated image. Performance was poor in this test; our method managed to do a fair job of aligning the images after five degrees of rotation, but ten degrees was too much for it to cope with. Iterative multiscale matching wasn't noticeably better with this subject.

Next we tried a photographic image with plenty of edges to work with: the mandrill. We started with a 256-square mandrill image and rotated it by some angle between zero and 90 degrees. Then we cropped the rotated image to 160 pixels square to elim-

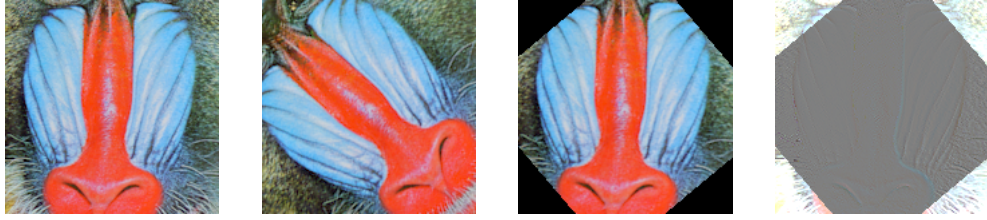


Figure 9.1: Test case: 50 degree rotation

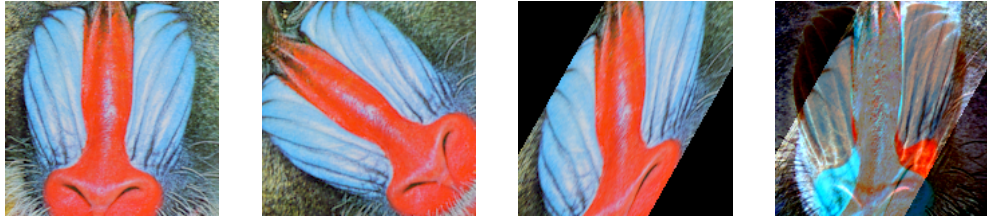


Figure 9.2: Test case: 55 degree rotation

inate any black corners. We cropped the unrotated reference image to the same size. Figure 9.1 shows an example of the two source images. With the default matching settings and using the straight-through pipeline processing mode, our method provided excellent alignment for rotations up to 15 degrees, but failed beyond that.

This is not surprising. The default matching settings assume that the “cameras” that captured the two images are nearly level, within 5 or at most 10 degrees of each other. The default window used in matching assumes that matching points are within 10% of the image width of each other. Rotations of more than 10 degrees move points near the edge of the image further than this amount. So we tried increasing the matching window size, at the same time switching to the iterative multiresolution mode of operation to keep the matching time short. With a matching window size of 20% of image width, our method handled rotations up to 40 degrees. Increasing the window size to 30% gave us good performance up to 50 degrees, but even a 100% matching limit did not improve the results any further.

Figure 9.1 shows the 50 degree test case. The first two images are the inputs. The third image is the effect of the registration process on the second image. The fourth image is the result of subtracting the third image from the original. There is only a tiny misregistration remaining, under a pixel. Numerically, the transformation matrix that the method fitted can be decomposed into a 49.94 degree rotation, a scaling of 1.0004, and a translation of less than 0.1 pixel.

In contrast, Figure 9.2 shows the 55 degree test case. Here, the calculated transformation did rotate the mandrill’s nose back to the vertical position, but there is now a strong skew component as well.

Then we tried iterating the match/fit process three times at each level before proceeding to the next lower level. This allowed us to align the 55 degree test case (as well as all lesser rotation tests), but it failed at 60 degrees. Thus, iteration within a level improved the matching range only slightly.

Next we tested translation on the mandrill. (We used multiscale matching for this and all following tests.) With a sufficiently large window, matching worked up to a translation offset of 58 pixels. At this point, there was only about 65% overlap of the two images. Figure 9.3 shows the largest successful test offset. The recovered transformation matrix was nearly perfect (there was a relative error of about $1e-7$ in the coefficients). For this test, iterating the matching within one level did not improve the match range at



Figure 9.3: Test case: 58 pixel translation



Figure 9.4: Test case: scale by 1.45

all.

Interestingly, at offsets of 60 pixels through 100 pixels our algorithm generated transformation matrices that involve a reflection in the Y axis plus other minor adjustments. In retrospect, this isn't surprising — the mandrill image is nearly symmetric, and a reflection in Y gives a pair of images where all of the features in one image are near similar features in another image. The correct transformation, involving only a translation, has only 40–60% overlap between the two images, leaving large areas of each with nothing to match. If the test image was less symmetric, reflection would have been a less attractive alternative and the algorithm might have found the correct translation with larger offsets. This is obviously image-dependent.

This also suggests that sometimes an affine transformation is too general, and we would be better to restrict the transformation to a combination of translation, rotation, and scaling only in some cases.

Our next test used a 256-pixel mandrill that was scaled larger and then cropped to the original size, so it has a smaller field of view. Then we compared it to the original and tried to bring the original to the same size. Figure 9.4 shows the largest successful scale change, a factor of 1.45. The recovered transformation is a scale of 1.4496, no rotation, and a translation of less than 0.1 pixel. (The largest successful match was actually at a scale of 1.47 using intra-level iteration, and at 1.43 without it — very little change.) Zea.

Registration works a less well when the images are of drastically different sizes. One test with a 4:1 size difference (512 and 128 pixels square) produced worst-case registration errors of about 2-3 pixels at the scale of the larger image. (This was at the edge, due to a rotation error — the alignment was better in the centre). However, this error is less than one pixel distance in the smaller of the images, which seems like pretty good performance after all.

Size differences that are a power of two are a particularly favourable case for the method, because it allows the level alignment technique used on different-sized images (see Section 4.2.3) to pick wavelet

transform levels that have nearly identical content. For a more difficult test, we used images with a 3:1 size difference, which is near the worst case size ratio (any odd power of $\sqrt{2}$). Alignment was somewhat worse, with 3-4 pixel differences at the scale of the larger image, which is somewhat over 1 pixel at the scale of the small image.

All of the above tests were with image pairs where it was possible to match the images near-perfectly. Both images came from the same source image, so if the registration algorithm calculated the perfect transformation matrix, the only possible difference between the two aligned images would be residual errors from resampling.

With real-world images, such perfect matching often isn't possible. If a camera moves between successive exposures, there is some chance that the resulting change in the image cannot be described by an affine transformation. In fact, only a few special cases of camera motion do yield affine transformations in the 2D images. With a hand-held camera, the transformation is virtually guaranteed to be non-affine. This can be seen in Figure 8.3, which is a pair of hand-held camera images. After alignment, the images match quite well near the centre but less well near the edges, particularly the lower left corner. This is visible in the difference images in Figure 8.4.

9.2 Real vs. Computer-generated Images

Now we turn to an example of real and computer generated images. These are images of an atrium at the University of Aizu. There are photographs and several computer-generated images with approximately the same lens (or simulated lens) position. The light output and pattern of the luminaires (artificial light sources) was measured and modelled, and actual surface reflectance characteristics (BRDF) of some of the surfaces were measured. Figure 9.5 is a photograph of the atrium, Figure 9.6 is a computer rendering with an artist's choice of material properties, and Figure 9.7 is a computer rendering using measured BRDF data. We applied our method to the three possible pairings of these three images.

Unfortunately, the computer-simulated camera lens is slightly closer to the scene than the real camera lens. You can see this if you look at the alignment between the doorways on the lower floor walls at the extreme lower left and right, and the two large concrete columns in the foreground. Because of this, no 2-D transformation can align the photograph with the rendered images. Figure 9.8 shows the photograph resampled to align with the rendered image, since the rendered image is larger. I have manually brightened the photograph to show more detail in this figure. If you could see the two images overlaid, you would see that the registration of the long stairway and the elaborate glass ceiling is nearly perfect, but that there are substantial differences in the position of objects close to the camera. Interestingly, the right edge of the door in the lower left corner of the image is perfectly aligned — it was likely one of the edges that participated in the final transformation.

Since the registration is not perfect, the “image distance” is dominated by the geometric errors, not the content. However, the two computer-generated images fail to align with the photograph in virtually the same way, so perhaps the difference in “distance” is partially due to content. Our distance measure says that the BRDF-based image shown in Figure 9.8 is more like the photograph than the “artistic” image in Figure 9.5, with distance scores of 12.2 and 10.8 respectively.

On the other hand, we have done better with the two computer-generated images. You might imagine that they would already be aligned, since they were probably produced by the same software, but they are not. They appear to share the same camera viewpoint, but the field of view or simulated lens focal length is



Figure 9.5: Aizu atrium: photograph



Figure 9.6: Aizu atrium: artistic rendering



Figure 9.7: Aizu atrium: physical reflectance models



Figure 9.8: Aizu atrium: after alignment



Figure 9.9: Difference between aligned images

different, and the two images are slightly different sizes. It would take some determined work to manually align the pair of images. But our method aligns them automatically. It turns out that a scaling of 1.037 is needed to align them. Alignment is essentially perfect — if there are any errors, they are less than a pixel in size. Figure 9.9 shows the result of subtracting the two images. This clearly shows tone and colour differences between the two images due to material property differences, but no alignment errors. Our distance measure also says that these two images are very like each other; their distance score is 4.02.

9.3 Other Examples

In this section, we present an examples of the use of the registration component of the system alone, without Phase 5.

Figure 9.10 shows a group picture taken a Christmas time. Unfortunately, the subjects were having too much fun and not cooperating with the us, the photographer. One subject is looking away from the camera in the first photo, while a different one is doing so in the second. We didn't obtain any images that show everyone at their best. (Note: the images are 2272×1704 pixels.)

We would like to create a composite image that includes people from the two different images, but this is difficult because the two images are not aligned. It is easy to align a pair of images in Adobe Photoshop if only a translation is required; that can be done interactively. But if the alignment requires rotation or scaling, it becomes a trial-and-error process. So, we use our method to bring the two images into alignment.

Fortunately, there is enough general clutter in the background that our method finds lots of edges and can determine a suitable transformation despite the fact that all of the people had moved in between images. Then we resample one image so both are aligned. Once that is done, it is fairly simple to hand-paint a mask used to composite the two images into a single one. Figures 9.11 and 9.12 show the final composite image, the mask that was used to create it, and the two original images under the control of the mask.

Note that the mask is quite crude — all we had to do was paint the separating line that included each character entirely from one image or the other. The dividing line between the two images goes right through a CRT monitor and a binder along the wall behind the people, yet the transition is invisible in the composite image. This is only possible because of the perfect alignment of the “clutter horizon” in the image, provided by our method.



Figure 9.10: Original photos

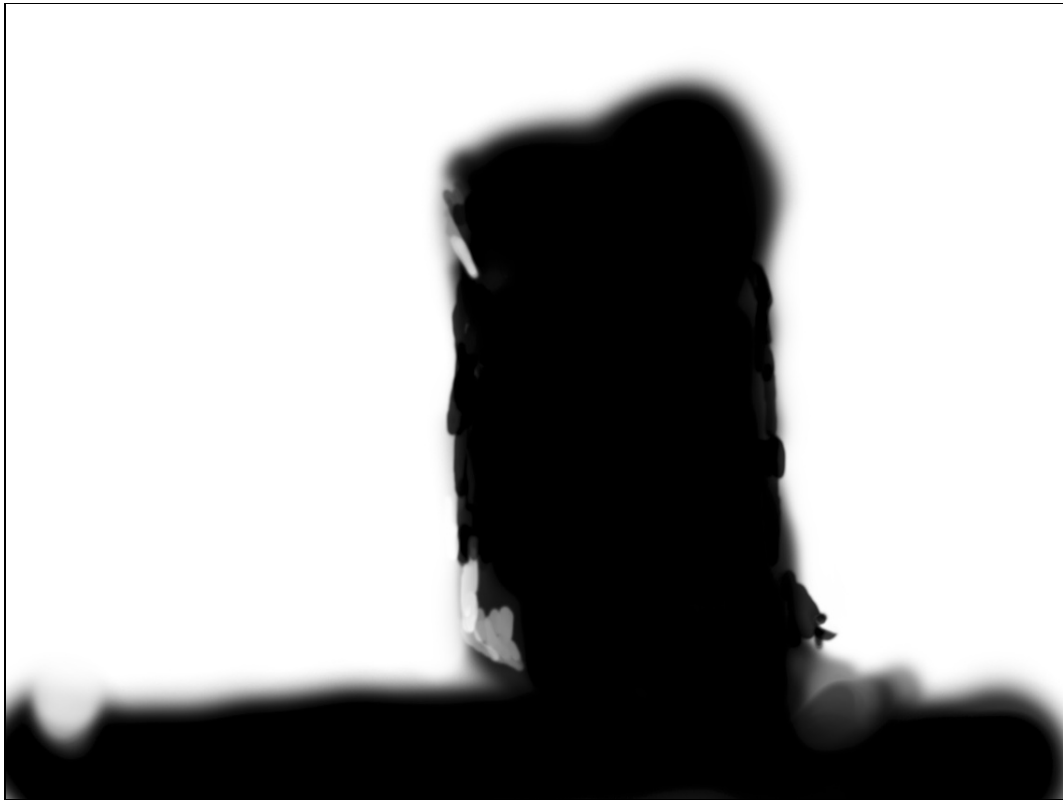


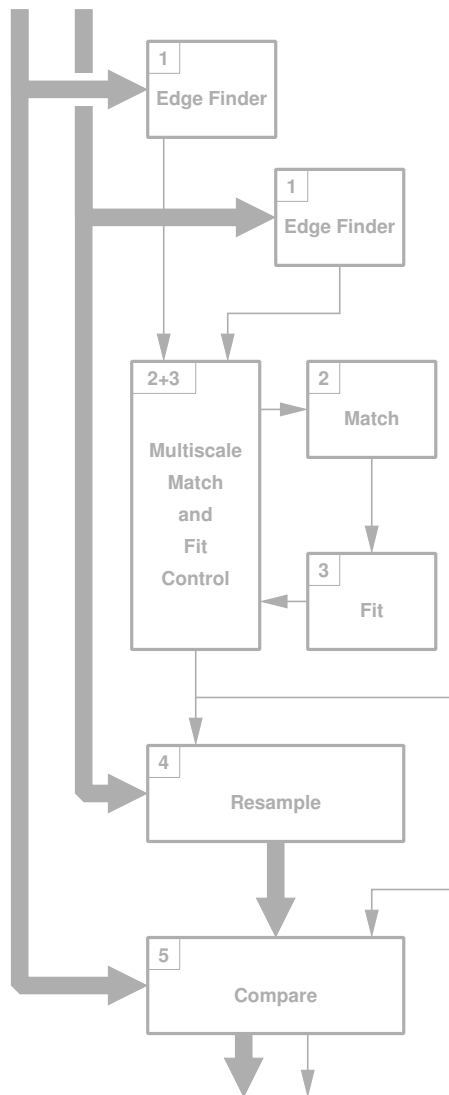
Figure 9.11: Composite photo and mask



Figure 9.12: Components of the composite

Chapter 10

Conclusions and Further Research



We have described an algorithm that accepts two images and compares them. The images are aligned before comparison based on the content of the images, with features being identified and matched at multiple spatial scales.

The method of finding features at multiple scales in the input images uses a published but still unusual method involving a wavelet transform. This wavelet transform calculates the intensity gradient of the image at multiple levels of detail. The gradient information is used to assist the matching phase that follows, as well as identifying edges. We have also built an alternate edge-finder based on the Laplacian operator for comparison, although it does not provide as much information to the matching stage.

Both edge-finding modules use a novel technique for providing the transition between one spatial level of detail and the next. This technique generalizes and unifies the filter scaling methods used in both the common wavelet transform and the Mallat-Zhong wavelet transform. This technique lets us retain spatial detail at full input image resolution for the finest several levels of detail, while permitting us to decimate the image at coarser levels of detail. The latter is important for handling large images.

The match-finding method is unusual in that it uses a graph matching algorithm that knows nothing specifically about images.

The transformation fitting algorithm uses an interesting iterative pruning technique for eliminating badly matched points and converging on a cluster of well-matched points that will provide the transformation we are seeking if we can isolate them. It commonly finds the correct transformation when as little as 20% of the input points are actually correct matches. The pruning technique has much in common with several “robust statistics”

methods, but most of these methods do not obtain the correct answer with input data where the fraction of correct points is below 50%.

After alignment, the remaining differences between the images are measured and displayed to the user. The algorithm also produces a number which may be interpreted as a visual “distance” between the two images.

This distance measure takes into account some of the properties of the human visual system. If one of the two images is considered to be a “perfect” or “canonical” image, the distance between the two images is a measure of the “image quality” of the other image. (Of course, there are many uses for image comparison when neither image is perfect.)

In addition to using the entire algorithm to calculate image distance, some portions of the algorithm can be useful alone. We expect that the ability to align a pair of images will be useful in photography (e.g. creating a composite image from multiple source images) and computer graphics (e.g. putting computer-generated objects or characters into real-world images).

The difference measurement portion of the algorithm (Phase 5) should be useful on its own in applications where the image pair is already aligned, but where the cost of really sophisticated algorithms (e.g. Daly’s) is not warranted. There are possible applications in image compression, control of successive refinement in rendering algorithms, etc.

The system as a whole is designed to be modular, and it has lived up to that aim in practice. It is easy to use a subset of the full processing pipeline. We have demonstrated satisfactory operation of the system using two different edge detection algorithms. It was easy to add the iterative multiscale matching and fitting technique (see Section 6.3) long after the system was originally designed with no changes to existing software and minimal new code. This is now the default mode of operation.

10.1 Further Research

When the iterative multiscale matching and fitting algorithm is asked to iterate within a single level of detail, it simply iterates a fixed number of times per level. It would be better to have the method automatically decide when to terminate the intra-level iteration based on how rapidly the transformation is changing.

Similarly, the multiscale matching/fitting method narrows the matching “window” between levels according to a fixed pattern. It should be possible to provide more intelligent adjustment of the window sized by examining the size of the residuals after fitting.

We should compare our graph-based matching method with a more conventional technique such as Iterated Closest Point.

Phase 5 needs a way of displaying a “difference map” of the colour differences found in a way that is meaningful to a human user. (At the moment, it displays differences in lightness only.)

The greatest limitation of the current method is its assumption of an affine transformation. Most of the ways in which one can acquire a pair of images of the same scene do not guarantee that the lens (or simulated lens) positions are matched well enough for an affine transformation to fully align the images. It would be very desirable to have the system support a more general transformation model.

If the image comparison software was used in an environment with a particular 3D renderer, it should be possible for the comparison software to obtain the 3D scene data that created the image. Alternately, if the renderer saved a depth map (e.g. Z-buffer contents) at the point the computer-generated image was created, it should be possible to reconstruct the 3D geometry of the visible surfaces in the computer-generated scene.

If the other image is a photograph, this depth information is not available directly, but matching edges and surfaces between the 2D photograph and the 3D computer-generated scene may allow approximate recovery of depth in the photograph. This should provide enough information to derive a perspective transformation, and to align the two images using it. A perspective transformation can deal with a shift in lens position.

It would be interesting to compare this software's calculation of image distance to the judgement of human viewers, particularly the relative visibility of varying amounts of image degradation due to noise, blur, etc.

Bibliography

- [1] D.I. Barnea and H.F. Silverman. A class of algorithms for fast image registration. *IEEE Trans. Computers*, C-21(2):179–186, 1972.
- [2] Peter G. J. Barten. *Contrast sensitivity of the human eye and its effects on image quality*. SPIE Optical Engineering Press, Bellingham, WA, 1999.
- [3] P.J. Besl and N.D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, February 1992.
- [4] L. G. Brown. A survey of image registration techniques. *Computing Surveys*, 24(4):325–376, 1992.
- [5] Itu-r bt.709-3, parameter values for the hdtv standards for production and international programme exchange. International Telecommunications Union.
- [6] J.F. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986.
- [7] Edwin E. Catmull and Raphael J. Rom. A class of local interpolating splines. In Robert E. Barnhill and Richard F. Riesenfeld, editors, *Computer aided geometric design*, pages 317–326. Academic Press, New York, 1974.
- [8] Y. Chen and G.G. Medioni. Object modelling by registration of multiple range images. *Image and Vision Comp.*, 10(3):145–155, 1992.
- [9] R. Claypoole, J. Lewis, S. Bhashyam, and K. Kelly. Laplacian edge detection. Retrieved 15 Dec. 2002 from <http://www.owl.net.rice.edu/~elec539/Projects97/morphjrks/laplacian.html>, 1997.
- [10] A. Collignon. *Multi-Modality Medical Image Registration by Maximization of Mutual Information*. PhD thesis, Katholieke Universiteit Leuven, 1998.
- [11] Subcommittee T1A1 Committee T1. Objective perceptual video quality measurement using a jnd-based full reference technique. Technical Report T1.TR.PP.75–2001, ATIS (Alliance for Telecommunications Industry Solutions), 2001.
- [12] Conrac Corporation. *Raster Graphics Handbook (second edition)*. Van Nostrand Reinhold, New York, 1985.
- [13] Scott Daly. The visible difference predictor: An algorithm for the assessment of image fidelity. In Andrew B Watson, editor, *Digital images and human vision*, pages 179–206. MIT Press, Cambridge, Mass., 1993.

- [14] I. Daubechies. *Ten Lectures on Wavelets*, volume 61 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, PA, 1992.
- [15] E. DeCastro and C. Morandi. Registration of translated and rotated images using finite Fourier transforms. *IEEE Trans. Patt. Anal. Machine Intell.*, 9(5):700–703, 1987.
- [16] Mark D. Fairchild. *Color appearance models*. Addison–Wesley, Reading, Mass., 1998.
- [17] J.A. Ferwerda, S.N. Pattanaik, P. Shirley, and D.P. Greenberg. A model of visual adaptation for realistic image synthesis. In *Proceedings of SIGGRAPH’96*, pages 249–258, 1996.
- [18] J.A. Ferwerda, S.N. Pattanaik, P. Shirley, and D.P. Greenberg. A model of visual masking for computer graphics. In *Proceedings of SIGGRAPH’97*, pages 143–152, 1997.
- [19] M.A. Fischler and R.C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.
- [20] L. Fonseca and M. Costa. Automatic registration of satellite images. In *Brazilian Symposium on Graphic Computation and Image Processing*, pages 219–226. IEEE Computer Society, 1997.
- [21] A. Gaddipatti, R. Machiraju, and R. Yagel. Steering image generation with wavelet based perceptual metric. *Comput. Graph. Forum*, 16(3):241–252, 1997.
- [22] Siavash Zokai George Wolberg. Robust image registration using log-polar transform. In *Proc. of IEEE intl. conf. on image proc.*, 2000.
- [23] A. V. Goldberg and R. Kennedy. An efficient cost scaling algorithm for the assignment problem. *Math. Program.*, 71(2):153–177, 1995.
- [24] G.H. Golub and C.F. Van Loan. *Matrix Computations, third edition*. Johns Hopkins University Press, Baltimore, Maryland, 1996.
- [25] D. Greenberg, K. Torrance, P. Shirley, J. Arvo, J. Ferwerda, S. Pattanaik, E. Lafortune, B. Walter, and B. Foo, S. and Trumbore. A framework for realistic image synthesis. In *Proceedings of SIGGRAPH’97*, pages 477–494, 1997.
- [26] Roy Hall. *Illumination and color in computer generated imagery*. Monographs in visual communication. Springer-Verlag, New York, 1989.
- [27] Paul Heckbert. Zoom [software]. Retrieved 15 Dec. 2002 from <http://www-2.cs.cmu.edu/~ph/src/zoom>, 16 May 1999.
- [28] R.M. Heiberger and R.A. Becker. Design of an S function for robust regression using iteratively reweighted least squares. *J. Comp. and Graph. Stat.*, 1(3):181–196, 1992.
- [29] Chiou-Ting Hsu and Rob A. Beuker. Multiresolution feature-based image registration. In *Proc. Visual Communications and Image Processing 2000*, volume 4067, pages 1490–1498. SPIE, June 2000.
- [30] R. W. G. Hunt. *The reproduction of colour*. Fountain Press, Kingston-Upon-Thames, 5th edition, 1995.

- [31] D. Huttenlocher, D. Klanderman, and A. Rucklidge. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, September 1993.
- [32] C.E. Jacobs, A. Finkelstein, and D.H. Salesin. Fast multiresolution image querying. In *Proceedings of SIGGRAPH '95*, pages 177–184, 1995.
- [33] C.D. Kuglin and D.C. Hines. The phase correlation image alignment method. In *Proc. IEEE 1975 Intl. Conf. on Cybernetics and Society*, pages 163–165, 1975.
- [34] C.L. Lawson and R.J. Hanson. *Solving least squares problems*. Prentice-Hall series in automatic computation. Prentice-Hall, Englewood Cliffs, N.J., 1974.
- [35] J Le Moigne. Parallel registration of multi-sensor remotely sensed imagery using wavelet coefficients. In *Proc. 1994 SPIE Wavelet Applications Conf.*, pages 432–443. SPIE, 1994.
- [36] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quart. Applied Mathematics*, 2:164–168, 1944.
- [37] D.G. Lowe. Object recognition from local scale-invariant features. In *Proc. of the International Conference on Computer Vision ICCV, Corfu*, pages 1150–1157, 1999.
- [38] J. Lubin. A visual discrimination model for imaging system design and evaluation. In E. Peli, editor, *Vision Models for Target Detection and Recognition*, pages 245–283. World Scientific, 1995.
- [39] J. Maintz and M. Viergever. A survey of medical image registration. *Medical Image Analysis*, 2(1):1–36, 1998.
- [40] S Mallat. *A wavelet tour of signal processing*. Academic Press, San Diego, CA, 1998.
- [41] S. Mallat and S. Zhong. Characterization of signals from multiscale edges. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(7):710–732, 1992.
- [42] J.L. Mannos and D.J. Sakrison. The effects of a visual fidelity criterion on the encoding of images. *IEEE Transactions on Information Theory*, IT-20(4):525–536, 1974.
- [43] D.W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *J. Soc. Industrial and Applied Mathematics*, 11(2):431–441, 1963.
- [44] W. Martens and K Myszkowski. Psychophysical validation of the Visible Differences Predictor for global illumination applications. In *IEEE Visualization '98 (Late Breaking Hot Topics)*, pages 49–52, 1998.
- [45] A. McNamara. *Comparing real and synthetic scenes using human judgements of lightness*. PhD thesis, University of Bristol, Bristol, 2000.
- [46] A. McNamara. Visual perception in realistic image synthesis. *Comput. Graph. Forum*, 20(4):211–224, 2001.
- [47] K. Mehlhorn and S. Nher. *LEDA: a platform for combinatorial and geometric computing*. Cambridge University Press, Cambridge, UK, 1999.

- [48] E. H. W. Meijering, W. J. Niessen, and M. A. Viergever. Quantitative evaluation of convolution-based methods for medical image interpolation. *Med. Image Anal.*, 5(2):111–126, 2001.
- [49] E. H. W. Meijering, K. J. Zuiderveld, and M. A. Viergever. Image reconstruction by convolution with symmetrical piecewise nth-order polynomial kernels. *IEEE Trans. Image Process.*, 8(2):192–201, 1999.
- [50] Mitre Corporation. Image quality evaluation. Retrieved 15 Dec. 2002 from <http://www.mitre.org/technology/mtf>, 24 March 2002.
- [51] K. Myszkowski and F. Drago. Validation proposal for global illumination and rendering techniques. Retrieved 15 Dec. 2002 from <http://www.mpi-sb.mpg.de/resources/atrium>, (n.d.).
- [52] K. Myszkowski and T. L. Kunii. A case study towards validation of global illumination algorithms: progressive hierarchical radiosity with clustering. *Visual Computer*, 16(5):271–288, 2000.
- [53] Alan V. Oppenheim and Ronald W. Schaffer. *Discrete-time signal processing*. Prentice Hall, Englewood Cliffs, New Jersey, 2nd edition, 1989.
- [54] S.N. Pattanaik, J.A. Ferwerda, M.D. Fairchild, and D.P. Greenberg. A multiscale model of adaptation and spatial vision for realistic image display. In *Proceedings of SIGGRAPH'98*, pages 287–298, 1998.
- [55] William K. Pratt. *Digital image processing*. Wiley, New York, 2nd edition, 1991.
- [56] William H. Press. *Numerical recipes in C++ : the art of scientific computing*. Cambridge University Press, Cambridge [Cambridgeshire] ; New York, 2nd edition, 2002.
- [57] M. Ramasubramanian, S.N. Pattanaik, and D.P. Greenberg. A perceptually based physical error metric for realistic image synthesis. In *Proceedings of SIGGRAPH'99*, pages 73–82, 1999.
- [58] V. Ranjan. *A union of spheres representation for 3D objects*. PhD thesis, University of British Columbia, Vancouver, 1996.
- [59] B.S. Reddy and B.N. Chatterji. An fft-based technique for translation, rotation, and scale-invariant image registration. *IEEE Trans. Patt. Anal. Machine Intell.*, 16(12):1266–1270, 1996.
- [60] P.J. Rousseeuw and Mia Hubert. Recent developments in PROGRESS. In Yadolah Dodge, editor, *LI-statistical procedures and related topics*, Lecture notes-monograph series ; v. 31, pages xxix, 498. Institute of Mathematical Statistics, Hayward, Calif., 1997.
- [61] P.J. Rousseeuw and S. Van Aelst. Positive-breakdown robust methods in computer vision. *Computing Science in Statistics*, 31:451–460, 1999.
- [62] P.J. Rousseeuw and K Van Driessen. Computing lts regression for large data sets. Technical report, University of Antwerp, 1999.
- [63] H. Rushmeier, G. Ward, C. Piatko, P. Sanders, and B. Rust. Comparing real and synthetic images: some ideas about metrics. In *Sixth Eurographics Workshop on Rendering*, pages 82–91, Dublin, Ireland, 1995.

- [64] S. Rusinkiewicz and M. Levoy. Efficient variants of the icp algorithm. In *Proc. 3D Digital Imaging and Modeling*, pages 145–152, 2001.
- [65] B. Rust and H. Rushmeier. A new representation of the contrast sensitivity function for human vision. In *Proceedings of the International Conference on Imaging Science, Systems, and Technology (CISST'97)*, pages 1–15, 1997.
- [66] C. Schmid and R. Mohr. Local grayvalue invariants for image retrieval. *IEEE PAMI*, 19(5):530–534, 1997.
- [67] D. Shreiner, editor. *OpenGL Reference Manual, Third Edition*. Addison-Wesley, Reading, Mass., 2000.
- [68] M. Stokes, M. Anderson, S. Chandrasekar, and R. Motta. A standard default color space for the internet - sRGB. Retrieved 15 Dec. 2002 from <http://www.w3.org/Graphics/Color/sRGB.html>, 5 Nov. 1996.
- [69] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. *Wavelets for computer graphics: theory and applications*. Morgan Kaufmann Publishers, Inc, San Francisco, 1996.
- [70] P. Thévenaz, U.E. Ruttimann, and M. Unser. A pyramid approach to subpixel registration based on intensity. *IEEE Transactions on Image Processing*, 7(1):27–41, January 1998.
- [71] P. Thévenaz and M. Unser. Optimization of mutual information for multiresolution image registration. *IEEE Transactions on Image Processing*, 9(12):2083–2099, December 2000.
- [72] Ken Turkowski. Filters for common resampling tasks. In Andrew S. Glassner, editor, *Graphics gems*, pages xxix, 833. Academic Press, Boston, 1990.
- [73] M. Unser. Splines — a perfect fit for signal and image processing. *IEEE Signal Process. Mag.*, 16(6):22–38, 1999.
- [74] P.A. Viola. *Alignment by Maximization of Mutual Information*. PhD thesis, Massachusetts Institute of Technology, 1995.
- [75] V. Volevich, K. Myszkowski, A. Khodulev, and E. A. Kopylov. Using the Visual Differences Predictor to improve performance of progressive global illumination computation. *ACM Trans. Graph.*, 19(2):122–161, 2000.
- [76] B.A. Wandell. *Foundations of Vision*. Sinauer Associates, Sunderland, MA, 1995.
- [77] A. B. Watson, J. Hu, and J. F. McGowan. Digital video quality metric based on human vision. *J. Electron. Imaging*, 10(1):20–29, 2001.
- [78] W. Wells, P. Viola, H. Atsumi, S. Nakajima, and R. Kikinis. Multi-modal volume registration by maximization of mutual information. *Medical Image Analysis*, 1(1):35–51, 1996.
- [79] George Wolberg. *Digital image warping*. IEEE Computer Society Press, Los Alamitos, Calif., 1990.
- [80] Z. Zhang. Iterative point matching for registration of free-form curves. Technical Report 1658, INRIA, 1992.

- [81] Z. Zhang, R. Deriche, O. Faugeras, and Q.-T. Luong. A robust technique for matching two uncalibrated images through the recovery of the unknown epipolar geometry. Technical Report 2273, INRIA, 1994.
- [82] T.D. Zuk. The registration of multimodality medical scans. Master's thesis, University of British Columbia, Vancouver, 1993.