

HIPR

Hypermedia Image Processing Reference

Robert B Fisher
Ashley Walker

Simon Perkins
Erik Wolfart

JOHN WILEY & SONS LTD
Chichester . New York . Brisbane . Toronto . Singapore

Contents

I	User Guide	5
1	Introduction to HIPR	6
1.1	Welcome to HIPR!	6
1.2	What is HIPR?	7
1.3	Guide to Contents	8
2	How to Use HIPR	11
2.1	General Overview	11
2.2	Hypermedia Basics	12
2.3	How to Use the Worksheets	15
2.4	Getting the Most Out of HIPR	18
2.5	Technical Support	19
3	Advanced Topics	20
3.1	Directory Structure of HIPR	20
3.2	Images and Image Formats	22
3.3	Filename Conventions	23
3.4	Producing the Hardcopy Version of HIPR	26
3.5	Installation Guide	28
3.6	Making Changes to HIPR	34
3.7	Changing the Default Image Format	36
3.8	Adding Local Information	37
3.9	Editing HTML and L ^A T _E X Directly	38
4	Local Information	40
II	Image Processing Operator Worksheets	41
5	Image Arithmetic	42
5.1	Pixel Addition	43
5.2	Pixel Subtraction	45

5.3	Pixel Multiplication and Scaling	48
5.4	Pixel Division	50
5.5	Blending	53
5.6	Logical AND/NAND	55
5.7	Logical OR/NOR	58
5.8	Logical XOR/XNOR	60
5.9	Invert/Logical NOT	63
5.10	Bitshift Operators	65
6	Point Operations	68
6.1	Thresholding	69
6.2	Adaptive Thresholding	72
6.3	Contrast Stretching	75
6.4	Histogram Equalization	78
6.5	Logarithm Operator	82
6.6	Exponential/'Raise to Power' Operator	85
7	Geometric Operations	89
7.1	Geometric Scaling	90
7.2	Rotate	93
7.3	Reflect	95
7.4	Translate	97
7.5	Affine Transformation	100
8	Image Analysis	104
8.1	Intensity Histogram	105
8.2	Classification	107
8.3	Connected Components Labeling	114
9	Morphology	117
9.1	Dilation	118
9.2	Erosion	123
9.3	Opening	127
9.4	Closing	130
9.5	Hit-and-Miss Transform	133
9.6	Thinning	137
9.7	Thickening	142
9.8	Skeletonization/Medial Axis Transform	145
10	Digital Filters	148
10.1	Mean Filter	150

10.2 Median Filter	153
10.3 Gaussian Smoothing	156
10.4 Conservative Smoothing	161
10.5 Crimmins Speckle Removal	164
10.6 Frequency Filter	167
10.7 Laplacian/Laplacian of Gaussian	173
10.8 Unsharp Filter	178
11 Feature Detectors	183
11.1 Roberts Cross Edge Detector	184
11.2 Sobel Edge Detector	188
11.3 Canny Edge Detector	192
11.4 Compass Edge Detector	195
11.5 Zero Crossing Detector	199
11.6 Line Detection	202
12 Image Transforms	205
12.1 Distance Transform	206
12.2 Fourier Transform	209
12.3 Hough Transform	214
13 Image Synthesis	220
13.1 Noise Generation	221
III Other User Information and Resources	224
A A to Z of Image Processing Concepts	225
A.1 Binary Images	225
A.2 Color Images	225
A.3 8-bit Color Images	226
A.4 24-bit Color Images	226
A.5 Color Quantization	227
A.6 Convolution	227
A.7 Distance Metrics	229
A.8 Dithering	230
A.9 Edge Detectors	230
A.10 Frequency Domain	232
A.11 Grayscale Images	232
A.12 Image Editing Software	233
A.13 Idempotence	233

A.14 Isotropic Operators	233
A.15 Kernel	233
A.16 Logical Operators	234
A.17 Look-up Tables and Colormaps	235
A.18 Masking	235
A.19 Mathematical Morphology	236
A.20 Multi-spectral Images	237
A.21 Non-linear Filtering	237
A.22 Pixels	238
A.23 Pixel Connectivity	238
A.24 Pixel Values	239
A.25 Primary Colors	240
A.26 RGB and Colorspaces	240
A.27 Spatial Domain	240
A.28 Structuring Elements	241
A.29 Wrapping and Saturation	241
B Common Software Implementations	244
B.1 Visilog	244
B.2 Khoros	246
B.3 Matlab Image Processing Toolbox	249
B.4 HIPS	251
C HIPRscript Reference Manual	253
D Bibliography	267
E Acknowledgements	270
F The HIPR Copyright	272
G User License	274
H About the Authors	280
I The Image Library	281
I.1 Introduction	281
I.2 Image Catalogue by Subject/Type	282
J Order form	314
Index	315

Part I

User Guide

Chapter 1

Introduction to HIPR

1.1 Welcome to HIPR!

You are looking at HIPR — The Hypermedia Image Processing Reference, a new source of on-line assistance for users of image processing everywhere. If you are a new user then this section is intended to help you explore the facilities of HIPR and so enable you to start using it effectively as quickly as possible.

Hypermedia vs Hardcopy

There are in fact two versions of HIPR — a *hypermedia* version which must be viewed on a computer screen, and a *hardcopy* (paper and ink) version which you can read like any other book. The first part of this welcome is an introduction to using the hypermedia version, for those who have not used hypermedia documents before.

Moving Around in Hypermedia Documents

Note that most of this section is not relevant to the hardcopy version of HIPR.

If you are viewing the hypermedia version of HIPR, then what you are seeing now is the Welcome Page of HIPR. You are viewing it with the aid of a piece of software called a *hypermedia browser*, probably one called **Netscape** although others can be used just as easily. The central portion of the screen contains this text and around its edges are various other buttons and menus that will be explained later. In fact you will probably not be able to see the whole Welcome Page since it is quite large. To see more of the page, look to the left or right of the text for a *scroll-bar*. Clicking in this with the left mouse button at different points along its length will cause different parts of the Welcome Page to be displayed. Try clicking in the different parts of the bar with different mouse buttons to see what effect they have. When you are happy with this method of moving around within a page, return to this point again.

You may also be able to move around a page by pressing keyboard keys, which you might prefer. If you are using Netscape or Mosaic then <Space> will scroll the page forward one screenfull, and <BackSpace> or <Delete> will scroll backwards.

The Welcome Page is just one of many pages that make up HIPR. These pages are linked together using *hyperlinks*. A hyperlink usually appears as a highlighted word or phrase that refers to another part of HIPR, often to a place where an explanation of that word or phrase may be found. The magic of hyperlinks is that simply clicking on this highlighted text with the mouse takes you to the bit of HIPR that is being referred to. This is one of the most powerful features of HIPR, since it allows rapid cross-references and explanations to be checked with the minimum of effort. In Netscape, hyperlinks appear underlined by default.

Note that if you are reading the hardcopy version of HIPR, then hyperlinks simply appear in parentheses as a cross-reference to the relevant page.

If you are using the hypermedia version of HIPR then you can try this out right now. For instance, this link (p.6) merely takes you to the top of the Welcome Page. You can return here after trying out the link using the scrollbar. You could have got there in the first place simply by using the scroll-bar, but sometimes a hyperlink is more convenient even for just moving around within a page. On the other hand this link (p.1) (don't follow it until you've read the rest of this paragraph!) takes you to the Top-Level Page of HIPR, which is where you will usually enter HIPR when you start using it for real. Near the top of that page is a hyperlink titled 'Welcome to HIPR!' which will bring you back here. Try it.

Hyperlinks don't have to be words or phrases — they can also be images. For instance if you go to the bottom of this page you will see a small icon with a picture of a house in it. Clicking on this will take you to the Top-Level Page again. Incidentally, this button, which appears at the bottom of almost every page in HIPR, is a good way of reorienting yourself if you get 'lost in hyperspace'.

Hyperlinks don't always just take you to another chunk of text. Sometimes they cause other sorts of information to be displayed such as pictures, or even movies. They can also cause sound clips to be played.

Once you have mastered moving around within a page using the scroll-bar or short-cut keys, and moving around between pages using hyperlinks, you know all that you need to start exploring HIPR by yourself without getting lost.

What Next?

HIPR includes a complete user guide that contains much more information than this Welcome Page, and you should familiarize yourself with the most important parts of this next. In the hardcopy version this means at least the first two chapters, while in the hypermedia version, the same material is found in the sections titled 'What is HIPR?', 'Guide to Contents' and 'How to Use HIPR'. If you are using the hypermedia version then the *User Guide* can be accessed from the Top-Level Page of HIPR (p.1).

1.2 What is HIPR?

Description

The Hypermedia Image Processing Reference (HIPR) was developed at the Department of Artificial Intelligence in the University of Edinburgh in order to provide a set of computer-based tutorial materials for use in taught courses on image processing and machine vision.

The package provides on-line reference and tutorial information on a wide range of image processing operations, extensively illustrated with actual digitized images, and bound together in a hypermedia format for easy browsing, searching and cross-referencing. Amongst the features offered by HIPR are:

- Reference information on around 50 of the most common classes of image processing operations in use today.
- Detailed descriptions of how each operation works.
- Guidelines for the use of each operation, including their particular advantages and disadvantages, and suggestions as to when they are appropriate.
- Example input and output images for each operation illustrating typical results. The images are viewable on screen and are also available to the student as an image library for further exploration using an image processing package.

- A large number of student exercises.
- Encyclopedic glossary of common image processing concepts and terms, cross-referenced with the image processing operation reference.
- Bibliographic information.
- Tables of equivalent operators for several common image processing packages: VISILOG, Khoros, the Matlab image processing toolbox and HIPS.
- Software and detailed instructions for editing and extending the structure of HIPR.

Motivations

The motivation behind HIPR is to bridge the gap between image processing textbooks which provide good technical detail, but do not generally provide very high quality or indeed very many example images; and image processing software packages which readily provide plenty of interactivity with real images and real computers, but often lack much in the way of a tutorial component.

By providing example input and output images for all the image processing operations covered, and making these easily available to the student through the use of hypermedia, HIPR presents image processing in a much more ‘hands on’ fashion than is traditional. It is the authors’ belief that this approach is essential for gaining real understanding of what can be done with image processing. In addition, the use of hypertext structure allows the reference to be efficiently searched, and cross-references can be followed at the click of a mouse button. Since the package can easily be provided over a local area network, the information is readily available at any suitably equipped computer connected to that network.

Another important goal of the package was that it should be usable by people using almost any sort of computer platform, so much consideration has been given to portability issues. The package should be suitable for many machine architectures and operating systems, including UNIX workstations, PC/Windows and Apple Macintosh.

1.3 Guide to Contents

HIPR is split into five main parts. The ordering of these parts differs slightly between the hypermedia and hardcopy versions of HIPR, but their content is very similar.

User Guide

The user guide provides a wealth of information about how to use, install and extend the HIPR package. It also describes in detail the structure of the package, and some of the motivations and philosophy behind the design. In this section:

Welcome to HIPR! (p.6) Where to start if you’re completely new to HIPR.

What is HIPR? (p.7) Introduction to the motivation and philosophy behind HIPR and a brief overview of the structure.

Guide to Contents (p.8) What you’re reading.

How to Use HIPR (p.11)

General Overview (p.11) General background information about how HIPR is organized.

Hypermedia Basics (p.12) An introduction to hypermedia and using hypermedia browsers.

How to Use the Worksheets (p.15) Detail instructions for using the worksheets contained in the *Image Processing Operations* section of HIPR.

Getting the Most out of HIPR (p.18) How to use HIPR effectively, illustrated with examples of typical tasks that users might use HIPR for.

Advanced Topics (p.20)

The Directory Structure of HIPR (p.20) How the files that make up HIPR are arranged into various sub-directories.

Images and Image Formats (p.22) Brief description of the image library and an explanation of the image format used.

Filename Conventions (p.23) Describes the naming conventions used for the various types of files found in the HIPR distribution.

Producing the Hardcopy Version of HIPR (p.26) How to print out and/or regenerate the hardcopy version of HIPR.

Installation Guide (p.28) Instructions for installing HIPR on your system.

Making changes to HIPR (p.34) If HIPR doesn't quite suit your needs you can make modifications to it. This and the following sections outline how it is done.

Local Information (p.40) This is a convenient place for the maintainer of your HIPR system to add local information about the particular image processing setup you use.

Image Processing Operator Worksheets

The bulk of HIPR is in this section, which consists of detailed descriptions of around 50 of the most commonly found image processing operations. The operations are grouped into nine categories:

Image Arithmetic (p.42) Applying the four standard arithmetic operations of addition, subtraction, multiplications and division to images. Also Boolean logical operations on images.

Point Operations (p.68) Operations that simply remap pixel values without altering the spatial structure of an image.

Geometric Operations (p.89) Altering the shape and size of images.

Image Analysis (p.104) Statistical and other measures of image attributes.

Morphology (p.117) Operations based on the shapes of features in images.

Digital Filters (p.148) Largely operations that can be implemented using convolution (p.227).

Feature Detectors (p.183) Operations designed to identify and locate particular image features such as edges or corners.

Image Transforms (p.205) Changing the way in which an image is represented, *e.g.* representing an image in terms of the spatial frequency components it contains.

Image Synthesis (p.220) Generating artificial images and adding artificial features to images.

The Image Library

All of the images used in HIPR are catalogued and described in this section.

Other User Information and Resources

Additional reference information, including particularly the *HIPR A to Z of Image Processing*.

A to Z of Common Image Processing Concepts (p.225) A comprehensive introductory level glossary of common image processing terms.

Common Software Implementations (p.244) Tables of equivalent operator names for several common image processing packages.

HIPRscript Reference Manual (p.253) Describes the markup language that defines this package.

Bibliography (p.267) Useful general references and texts for image processing and machine vision.

Acknowledgements (p.270) Our thanks to our many helpers.

The HIPR Copyright (p.272) Sets out the conditions of use of HIPR.

The User Licence (p.274) The legal contract for the use of HIPR.

About the Authors (p.280) That's us...

The Index

The main index for all of HIPR, and a very useful place to start looking for information. The hypertext version includes 'hyperlinks' to each indexed item.

Chapter 2

How to Use HIPR

2.1 General Overview

HIPR is a reference package for image processing, particularly designed for use in conjunction with image processing courses. Its key feature is that it is available in an easy-to-use hypermedia format, complete with on-line example images to illustrate the effects of many different image processing operations. This section introduces some of the main concepts of HIPR.

The Most Important Bits of HIPR

Some sections of HIPR are of particular importance. If you intend to learn how to use HIPR effectively then these are the sections that you should first familiarize yourself with:

How to Use HIPR This section of HIPR provides all the background information that you will need in order to use HIPR effectively, including a detailed description of the format of the image processing reference worksheets, and instructions on navigating around HIPR.

Image Processing Operations This is the core section of HIPR, providing reference ‘worksheets’ on a large number of image processing operations in common usage today. The structure of these worksheets is described in detail in the section on *How to Use the Worksheets* (p.15).

The A to Z The *A to Z of Common Image Processing Concepts* provides essential backup to the reference information contained in the *Image Processing Operations* section. This section provides background and tutorial information that describes jargon and concepts used in the operator worksheets, with which it is extensively cross-referenced.

The Index Unsurprisingly perhaps, the index is often the most useful starting place for an information search. However, it is particularly useful when using the hypermedia version of HIPR, because the index entries are actually *hot-links* that can be followed with simple mouse clicks.

Of course, the remaining sections of HIPR are also important, but the above four are the most relevant to beginning users.

Hypermedia vs. Hardcopy

There are two versions of HIPR: a *hypermedia* version and a *hardcopy* version. The former must be viewed on a computer screen using a special piece of software known as a *hypermedia browser*,

the latter is simply a traditional ink on paper document. They are virtually identical in content, but there are differences in the ease of use of each version for different tasks.

The hypermedia version of HIPR provides functionality such as following links between reference sections at the click of a mouse button and easy searching of documents. It also has the advantage that, if used on a local area network, the reference information is available on any computer connected to that network, and to any number of users at the same time. Additionally, the example images that accompany operator worksheets can be displayed easily on screen by simply clicking on appropriate links in the hypertext.

The hardcopy version of HIPR obviously does not have these advantages, but nevertheless, there are times when it is nice to have a printed version of the reference. For one thing, the hardcopy reference does not occupy any space on the screen when you are working on something else.

Whether reading HIPR in hypermedia or hardcopy form, the overall structure of the reference is the same, consisting of an introductory section, a large image processing operator reference section, and finally a series of appendices and an index. A more detailed overview of the structure of HIPR is given in the Guide to Contents (p.8).

If you are using the hypermedia version of HIPR then you should be familiar with the operation of your hypertext browser. Some useful hints and tips are given in the section on Hypermedia Basics.

2.2 Hypermedia Basics

What is Hypermedia?

Hypermedia is a relatively new term created to describe the fusion of two other new technologies: *multimedia* and *hypertext*.

Multimedia refers to the capabilities of modern computers to provide information to a user in a number of different forms (sometimes called *modalities*) including images, graphics, video and audio information, in addition to the standard textual output of older computers.

Hypertext refers to the idea of linking different documents together using *hyperlinks*. A hyperlink often appears in a hypertext document as a piece of highlighted text. The text usually consists of a word or phrase that the user might require further information on. When the user activates the hyperlink, typically by clicking on it using a mouse, the user's view of the document is changed so as to show more information on the word or phrase concerned. This may mean that a different document is displayed on screen, perhaps positioned so that the relevant piece of text is at the top of the viewing screen; or alternatively, the original text might 'unfold' to include some extra paragraphs providing the required information. The exact effect varies from implementation to implementation. Through the use of hyperlinks, many documents or parts of documents can be combined together to make a larger hypertext document. The hyperlinks make it very easy to follow cross-references between documents and so to look up related information. This often makes hypertext documents more suitable as reference manuals than conventional text manuals which must be accessed in serial fashion.

Hypermedia documents are simply hypertext documents with multimedia capabilities in addition. HIPR is a hypermedia document. Its basic structure is that of a hypertext document for ease of cross-referencing and information finding, but it also includes links to a library of images illustrating the effects of image processing operations, a multimedia capability. Of course, hypermedia documents in general can include many more types of multimedia information than simply images.

Hypermedia in Practice

In practice, in order to read a hypermedia document using a computer, a user needs two things. The first is the hypermedia document itself. This will be supplied in some machine-readable way (in a disk file typically) in a format that encodes the information content and hyperlink structure of

the document. The second thing the user needs is some way of viewing the document in a human-readable way. This is usually done by a program called a hypermedia *browser*. The browser displays (or plays in the case of audio information) the information contained within the document to the user and also handles the processing of hyperlinks when required.

The hypermedia format used by HIPR is known as HTML (HyperText Markup Language), which is a language that has achieved great popularity recently since it is a cornerstone of the *World Wide Web*. The World Wide Web (or WWW) is a hypermedia system on a global scale, that links together documents and information all over the world via a collection of computer networks known as the Internet. Documents available via WWW are all encoded in HTML format and so a browser is necessary to display them on a computer screen. The most popular HTML browsers in use at the time of writing are *Mosaic*, available from the National Center for Supercomputing Applications (NCSA) in Illinois, and *Netscape*, available from Netscape Communications. HIPR is intended to work particularly well with Netscape, but it will work almost as well with any other graphical HTML browser (such as Mosaic). Note that only the HTML files are supplied with HIPR — it is necessary to obtain and install Netscape or a similar viewer yourself separately. Fortunately, an increasingly large number of computer networks in universities and businesses have Netscape installed already, in order to access WWW, and the same setup can be used for viewing HIPR. Details of how to obtain Netscape if your system does not already have it are given in the *Installation Guide* (p.28).

Using HIPR with Netscape

The preferred HTML browser for use with HIPR is Netscape

Netscape is widely used to access the World Wide Web and is one of the better graphical HTML browsers available. It also has the advantage that it is available free from Netscape Communications and for a number of popular computer platforms. However, there are several other HTML browsers available and most of the graphically based ones should work just as well. For instance, Mosaic was one of the first good graphical browsers, and until recently was by far the most popular one. It is similar in appearance to Netscape (it was largely written by the same people) but it currently lacks some of the advanced presentation features available with Netscape. We assume that you will be using Netscape in this text, but most of the comments apply to Mosaic and other graphical browsers equally well.

If you are using the hypermedia version of HIPR and have managed to get this far, then you already know how to follow hyperlinks in text, and how to scroll the screen, but to summarize:

Hyperlinks appear as highlighted text. Their exact appearance varies according to how your browser is set up, but typically in Netscape they will appear as underlined text, possibly in a different color from normal text. To follow a hyperlink simply point at the link using the mouse and press the left mouse button once. The document at the other end of the hyperlink will then be displayed.

Some hyperlinks (known as ‘imagelinks’), cause images to be displayed when they are clicked on. In the hypermedia version of HIPR These image links appear as small pictures called thumbnails. Clicking on the image causes the full sized image to be displayed. In the hardcopy version of HIPR, imagelinks are simply printed as a filename in a **typewriter** font. The filename is the name of a file in the **images** sub-directory of HIPR that contains the full-sized image being referred to.

Note: On some Netscape setups, when the user clicks on an imagelink, the current document will be replaced in the Netscape window with the full-sized image. The disadvantage of this is that it makes it very difficult to read the text describing the image while viewing that image! One simple solution is to force Netscape to open a new full window when displaying the image (*e.g.* by clicking on the imagelink with the middle mouse button on UNIX systems). However, a full Netscape window is a rather cumbersome method of displaying images and so a better way is to instruct Netscape to display images using a specialized external viewer. Ask the person who installed HIPR on your system if it is possible to set this up. Details on how to do this are provided in the *Installation Guide* in the section on Using External Image Viewers (p.30).

If the displayed document is more than one screen long then a dark vertical bar with arrows at top and bottom will appear to the left or right of the the main document view. Clicking within this bar with the left or middle mouse button 'scrolls' the document up or down so that you can see different parts of it. Experiment with this to get an idea of how it works.

Possible Problems

Monochrome vs Grayscale vs Color Displays

Since HIPR contains large numbers of images and graphics, many of which are in color, a color display is needed to show HIPR off at its best. However, HIPR can be displayed reasonably well with Netscape on grayscale or even monochrome ('black and white') screens. Most of the practical examples of image processing in HIPR use grayscale images for demonstration purposes, and so these will display perfectly well on grayscale screens. There may however be a few cases where it is difficult to differentiate between colors on a grayscale screen that are clearly distinct on a color screen. On monochrome screens, Netscape will use 'dithering' to approximate grayscales. This means that some example images (particularly those containing lots of fine detail) will not show up very well.

Netscape Bugs

It has come to our attention recently that version 1.1N of Netscape running on Sun workstations using Solaris 2.3 or 2.4 has a bug which may cause HIPR to crash occasionally. This seems to occur particularly on screens containing lots of images. The problem only occurs when HIPR is being accessed in local mode *i.e.* with a URL that begins `file://localhost/...` Therefore one way to avoid the problem is to access the pages via your WWW server (if you have one) using a URL that begins `http://...` Ask your system administrator to set this up. Note that this will slow down access of HIPR pages slightly. Alternatively, and preferably, the problem seems to have been fixed in the subsequent release of version 2.0 of Netscape, so you may prefer to download that from the usual FTP sites. See the *Installation Guide* (p.28) for details.

Essential Hints and Tips

- If you have an internet connection, then extensive help on Netscape may be found by clicking on the **Help** menu at the top of the screen, and then selecting **Handbook**.
- The **Back** button at the top of the screen can be used to take you back to the previous document after following a hyperlink. If you follow multiple successive hyperlinks, then Netscape remembers each one, and so successive use of the **Back** button will take you back along the 'chain'.
- The **Forward** button reverses the action of **Back**.
- At the top and/or bottom of many HIPR pages are 'navigation buttons'. The use of these is explained in the section on Worksheet Structure (p.17).
- It is possible to search for an item of text within a document by selecting **Find...** from the **Edit** menu, and entering information into the pop-up search window. Note that only the currently displayed document is searched. It is not possible to directly search the whole of HIPR in one go. However, it *is* possible to search within the HIPR index (p.314) which is almost as useful.

Other Useful Hints and Tips

- There are several 'accelerator keys' that duplicate mouse actions, and that can make Netscape faster to use. The most important ones are:

<**Space**> Scroll forward

-
- <BackSpace> Scroll backwards
 - <Alt>-<Left> Go back to previous document
 - <Alt>-<Right> Go forward to next document
 - <Alt>-h Display a list of previously visited documents and allow the user to jump back to one
 - <Alt>-f Pop up the search window

- When the mouse pointer is positioned over a hyperlink (before any buttons are pressed), the *URL* (Uniform Resource Locator) of the destination document is displayed at the bottom of the screen. The most important bit of this is the text after the final '/' which is the name of the destination document. This information is particularly useful when following a link that leads to an image since it gives the name of the image file within the HPR images directory (p.22).
- By default, Netscape keeps track of which documents you have visited before, and highlights hyperlinks leading to these documents slightly differently from normal. Typically, such hyperlinks will have a differently colored underline, or a dashed underline. This option can be disabled (consult the Netscape documentation for details).

2.3 How to Use the Worksheets

The Role of the Worksheets

The worksheets which make up the middle section of HPR are probably the most important part of the reference. They provide detailed information and advice covering most of the image processing operations found in most image processing packages. Generally, each worksheet describes one operator. However, in addition, many worksheets also describe similar operators or common variants of the main operator. And since different implementations of the same operator often work in slightly different ways, we attempt to describe this sort of variation as well.

The worksheets assume a basic knowledge of a few image processing concepts. However, most terms that are not explained in the worksheets are cross-referenced (via hyperlinks where applicable) to explanations in the A to Z (p.225) or elsewhere. This means that the worksheets are not swamped with too much beginner level material, but that at the same time such material is easily available to anyone who needs it.

Some of the worksheets also assume some mathematical knowledge, particularly in the descriptions of *how* the various operators work. However this is rarely important for understanding *why* you might use the operator.

Worksheet Organization

The worksheets are divided into nine categories:

- Image Arithmetic
- Point Operations
- Geometric Operations
- Image Analysis
- Morphology
- Digital Filters
- Feature Detectors

-
- Image Transforms
 - Image Synthesis

These categories are arranged in very approximate order of increasing difficulty (so that the easiest and often most useful categories come first). The categories are largely independent however, and may be tackled in any order.

Within each category, the individual worksheets are also arranged in approximate order of increasing difficulty and decreasing usefulness. The worksheet ordering is slightly more important than is the case with categories, since later worksheets tend to assume some understanding of earlier worksheets. However, as usual, any references to information contained in earlier worksheets will take the form of hyperlinks that can be quickly followed if necessary.

The Elements of a Worksheet

Each worksheet nominally consists of the same set of sections, although some of them are omitted on some worksheets. The sections are:

Common Names

The main heading of each worksheet gives what we believe is the most appropriate name for the operator concerned. This is usually the commonest name for the operator, but is sometimes chosen to fit in with other operator names. The purpose of the Common Names section is to list alternative names for the same or very similar operators.

Brief Description

This section provides a short one or two paragraph layperson's description of what the operator does.

How it Works

Unsurprisingly, this section explains how the operator concerned actually works. Typically, the section first describes the theory behind the operator, before moving onto details of how the operation is implemented.

Guidelines for Use

This is one of the more important parts of the worksheets, and often the largest. This section provides advice on how to use an operator, illustrated with examples of what the operator can do, and examples of what can go wrong. An attempt is made to provide guidelines for deciding *when* it is appropriate to use a particular operator, and for choosing appropriate parameter settings for its use.

The Guidelines section contains *imagelinks* like this one: `bld1heq2` which represent example images. In the hardcopy version of HIPR these simply appear as a filename in **typewriter font** that refers to an image in the images directory (p.22). However, in the hypermedia version of HIPR, imagelinks appear as small pictures (known as thumbnails) which when clicked on cause the corresponding full-sized image to be displayed. Try it.

The Guidelines section often provides worked through examples of common image processing tasks that illustrate the operator being described.

Common Variants

This section is optional, and describes related operators that are not sufficiently different from the current operator to merit a worksheet of their own, but have not been adequately covered in the rest of the current worksheet.

Exercises

Exercises are provided to test understanding of the topics discussed on the worksheet. A proportion of the questions involve practical exercises for which the use of an image processing package is required. Suggestions for suitable test images from the image library (p.22) are also given.

References

This section lists bibliographic references in a number of popular image processing textbooks for the operator concerned.

Local Information

This section is provided to allow the person in charge of installing HIPR to add information specific to the local installation. Suitable information would include details about which operators in local image processing packages correspond to the operator described. More details are given in the section on adding local information (p.37).

Navigation Buttons

At the top of almost every page in the hypermedia version of HIPR appear up to four *navigation buttons*. On pages that occupy more than about a screenful, the buttons are duplicated at the bottom of the page. These navigation buttons help the user navigate around the worksheets quickly, and have the following functions:

Home Go to the top-level page.

Left Arrow Go left one page, when in a linear series of topics. Note that this is not the same as the **Back** button described elsewhere (p.14).

Right Arrow Go right one page, when in a linear series of topics. Note that this is not the same as the **Forward** button described elsewhere (p.14).

Up Arrow Go up one level

To understand the operation of the navigation buttons, refer to Figure 2.1 which shows part of the structure of HIPR.

As the figure shows, the structure of HIPR is somewhat like the root system of a plant (or a tree turned upside-down), with each node branching out into finer detailed nodes. With this picture in mind it should be fairly easy to see how the various navigation buttons work.

Note that the left and right arrow buttons are *not* equivalent to the **Back** and **Forward** buttons provided by Netscape (at the top left of the screen). The **Back** button simply reverses the effect of the last link followed (no matter whether it was via a navigation button or via a hyperlink in the text of a worksheet). The **Forward** button can only be used after the **Back** button has been used, in which case it undoes the backwards jump.

It is possible, by following too many hyperlinks in succession, to become 'lost in hyperspace', *i.e.* to become confused as to where you are in the HIPR structure. In this case it is quite a good idea

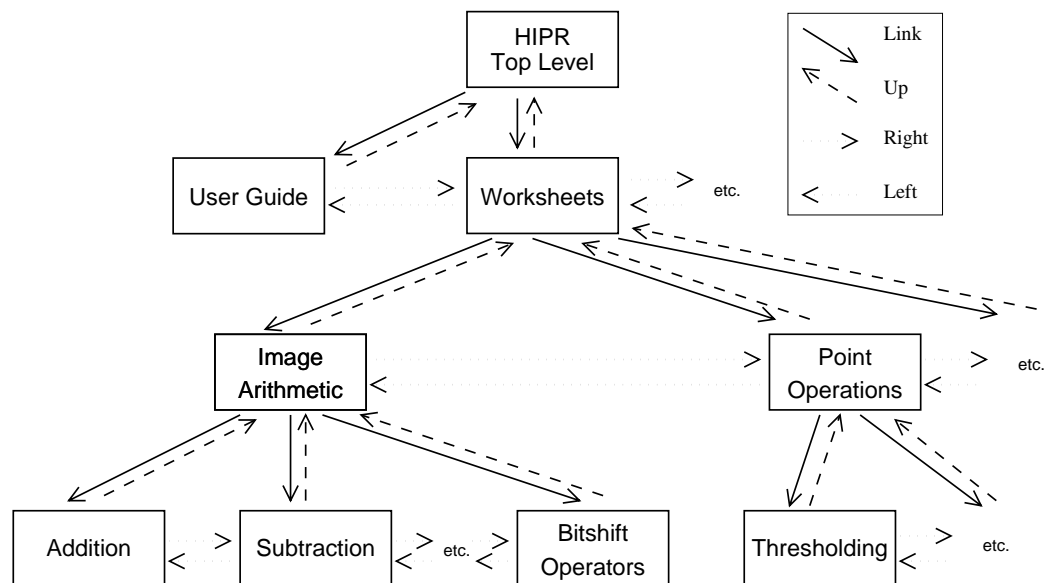


Figure 2.1: The structure of part of the worksheet section of HIPR. The arrows show possible transitions between HIPR pages, and the arrow type indicates how this transition is achieved.

to press the **Back** button repeatedly until you return to somewhere you recognize. Alternatively, just hit the **Home** HIPR navigation button to get back to the top level again.

2.4 Getting the Most Out of HIPR

The best way to find out how to use HIPR effectively is to play around with it for a while. The workings are for the most part fairly intuitive and simple. However, before doing this, users might find it helpful to scan the following examples which attempt to illustrate how HIPR can be used in common real-life situations.

You will probably find it handy to work through the examples yourself in order to make them clearer. Most of the examples start from the top level worksheet, so you should jump there (p.1). Note that if you are using Netscape or Mosaic, then click on this link with the middle mouse button to produce a second window in which you can work while keeping this information visible in the first window.

Examples

Q: *I know that I am supposed to use the Canny edge detector for a problem, and I know that it's a sort of feature detector, but I don't know what the 'Gaussian width' parameter that the algorithm asks for is, or how varying it affects the output. Can HIPR help?*

A: Since you know that Canny is a feature detector, you can jump straight to the *Feature Detectors* section of HIPR from the top level. After doing this you will see that the Canny edge detector is one of the feature detectors listed and so you can click on that to bring up the Canny worksheet. The 'Guidelines for Use' section mentions that the effect of the Canny operator is controlled by three parameters, one of which looks like the 'Gaussian width' parameter. The 'Guidelines' section explains the effect of varying this, while the 'How it Works' section explains more about what it actually does.

Q: *I have heard about the Prewitt edge detector and I would like to find out more about it, but I*

can't find it listed in the operator worksheets section. Where can I find some information?

A: Not every operator has a worksheet named after it. This is because there are a lot of operators that do very similar things, and often several alternative names for each operator. However, these operators are usually mentioned as variants within another worksheet and if so they will be cross-referenced in the index. So if you jump to the Index from the top level and *search* for the string 'Prewitt' (using Alt-F in Netscape for example), you will find references to two slightly different Prewitt operators. Clicking on either will take you to the relevant worksheet that describes them.

Q: *I have been trying to use Histogram Equalization to enhance some images. On some pictures the result is definitely clearer, but on others the detail that I am interested in disappears. What is happening?*

A: The place to find out about how to use an operator in practice is the 'Guidelines for Use' section of each worksheet. In this case, going to the Histogram Equalization worksheet (under Point Operations from the top level), you will find that the 'Guidelines' section explains why this enhancement technique sometimes gives unexpected results on images with large areas of fairly uniform background.

Q: *I have just installed HIPR and I would like to add some information saying where I can be contacted. How do I do this?*

A: The section on *Adding Local Information* (p.37) describes how to do this. Briefly though, you would simply put the information you want to include in the *General Local Information* section.

Q: *I am using the Khoros image processing package, and I want to find out if it has a Laplacian of Gaussian operator. How can I find this out?*

A: The appendix on *Common Implementations* at the end of HIPR has tables listing equivalent operators for several popular image processing packages. In this case you would find that, the appropriate convolution filter is produced using `vmarr` (to produce a 'Marr filter').

2.5 Technical Support

While every effort has been made in the development of HIPR to create a package that is simple to modify, and to give pointers to sources of the tools you may need to achieve this, it is impossible to guarantee that every problem you may encounter will have been anticipated. The variety of architectures on which HIPR can be used make it extremely difficult to do this.

Unfortunately, neither the authors nor Wiley can provide direct technical support to anyone making modifications to HIPR. An alternative source of possible help and advice has been created in the form of a mailing list to which you can post questions and queries about HIPR. The authors of HIPR monitor this list and will attempt to provide answers based on their own particular experiences with the system. The list will be used to disseminate news about changes, upgrades and bug-fixes to HIPR, so we strongly recommend that all installers of HIPR subscribe to the list. We would also welcome any general comments about the product which you should like to make.

To subscribe to the mailing list send an e-mail to:

`hipr-users-request@dai.ed.ac.uk`

containing just the word: 'subscribe' on a line by itself in the BODY of the message (not the subject). You will be added to the list and you will then receive detailed information on how to use the list to ask questions.

There is also a WWW page for HIPR users at:

http://www.dai.ed.ac.uk/staff/personal_pages/rbf/HIPR/hipr_users.htm

or you can contact Wiley directly at this email address: `hipr@wiley.co.uk`

Chapter 3

Advanced Topics

3.1 Directory Structure of HIPR

This section describes the organization of the files and directories that make up the installed HIPR system. It is useful to have read this section before looking at the other advanced sections. *Note that, if you are using a Macintosh computer, then a ‘directory’ is the same thing as a ‘folder’.*

Overall Structure

HIPR consists of more than 2000 separate files, divided over nine main sub-directories. These sub-directories all branch off from the *HIPR root directory*, which by default is called simply `hipr`. Each sub-directory contains a different sort of file as described below. This overall structure is shown graphically in Figure 3.1.

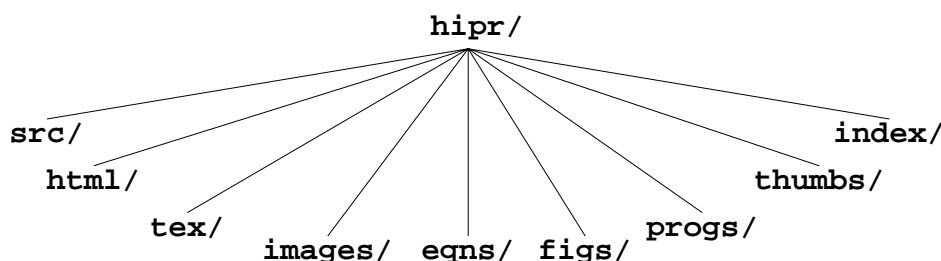


Figure 3.1: Directory structure of HIPR. Note that the directory names are all shown with a trailing slash (/) which is a UNIX convention for denoting directories (as opposed to data files). The slash is not part of the actual name.

Each of the sub-directories will now be described briefly in turn. Note that the names we mention are the default ones — your system installer may have chosen to change them.

html

Contains the HTML (HyperText Markup Language) files which define the hypermedia version of HIPR. When using a hypermedia browser such as Netscape to look at HIPR, you are actually looking primarily at the files in this directory.

tex

Contains the hardcopy version of HIPR. As explained in the *Generating Hardcopy* (p.26) section, the hardcopy is printed using PostScript files, which are originally generated from L^AT_EX source files. This directory contains both L^AT_EX and PostScript versions.

src

As detailed in *Making Changes to HIPR* (p.34), both HTML files and L^AT_EX source files are generated from common sources files written in a special HIPR format known as *HIPRscript*. These source files are stored in this directory.

eqns

Contains GIF image files of all the equations used in HIPR. These are necessary because at the time of writing, Netscape is unable to render mathematics directly and so equations are included as inline images.

figs

Contains GIF files and equivalent PostScript files for all the figures used in HIPR. The GIF files are used by Netscape, and the PostScript files are used by L^AT_EX.

images

All the full sized images that are used as examples of image processing operations are stored in this directory.

thumbs

For each of the full sized images, there is a 'thumbnail' sized version that is stored in this directory. The thumbnails are used in the hypermedia version of HIPR for imagelinks to the full sized images.

progs

This directory contains the Perl scripts that are used to regenerate HTML and L^AT_EX files from the HIPR sources files. It also contains several UNIX scripts that can be used for regenerating figures, equations and thumbnail images. See *HIPRscript Reference Manual* (p.253) for details.

index

In the course of generating HTML and L^AT_EX files from HIPR source files, the above Perl scripts write information about index entries to this directory, from where it can be gathered in order to produce the *HIPR Index* (p.314).

3.2 Images and Image Formats

The HIPR Image Library

One of the major reasons for using HIPR is the fact that every image processing operation described in the package is extensively illustrated using good quality on-line digitized images. Viewing these images on a computer screen gives a much more realistic impression of what real image processing is like than is provided by looking at typical pictures in image processing textbooks.

In addition, every single image is stored in the *HIPR Image Library*, from where they can be used as input to real image processing packages. This extends the teaching potential of HIPR in a number of ways:

- Students can compare the output of their own image processing package's operators with the example output images included with HIPR to see how different implementations differ slightly in functionality.
- They can also experiment with the effect of changing parameters to operators to see how this changes the output. Suggestions of interesting things to try are given in the student exercises section of many of the worksheets.
- Image processing operators can be tried out on additional images which are *not* used as test examples for that operator in HIPR.

And we are sure that users of image processing will find many other uses for such a large and varied collection of images. For example, as test images to new image processing algorithms.

Viewing Images with HIPR

As explained in earlier introductory sections (p.16), references to images in HIPR are made using imagelinks. To summarize briefly, in the hypermedia version of HIPR, these appear as small inline images known as thumbnails. Clicking on a thumbnail causes a full sized version of that image to be displayed. In the hardcopy version, imagelinks appear simply as a filename in **typewriter font**.

This is an example of an imagelink: **ape1**.

Viewing images with the hypermedia version of HIPR is easy — just click on the relevant imagelink. If you wish to view the images outside of this environment then things are slightly more tricky. You must use a piece of software called an image viewer and point it at the actual image file containing the image you wish to view. The details of how to do this vary considerably depending upon the machine architecture on which you are using HIPR, and so we cannot tell you exactly how to go about this. Typically though, you would proceed in one of two ways:

For the purposes of these examples assume that the image viewer is called 'viewimage' and the image is called 'test.gif'.

- If you are using a *command-line based system* (*i.e.* one where you type commands at the computer and hit ENTER to get them executed) such as MS-DOS or a UNIX shell, then you would type something like:

```
viewimage test.gif
```

and press ENTER to display the image.

- If you are using a windows system that doesn't provide a command-line prompt, then you would probably double-click on an icon representing the **viewimage** program, and you would then use that program's file browser to select the **test.gif** image to be displayed. Alternatively (*e.g.* on Macintoshes) you might just double-click on the icon representing the image

file itself and the windows system would automatically start up an appropriate viewer to display that image.

Your system supervisor or course administrator should be able to advise you on the best way to do this. You may find more information in the Local Information section (p.40) of HIPR.

The above discussion assumes that you know where the image file to be displayed actually is and what it is called. The location of images is discussed in the next section.

Two Types of Images

The images in HIPR can be divided into two different categories: *raw* images and *processed* images.

Raw images are simply images that have been digitized for storage on a computer, but as yet have had no (or very little) image processing done to them. Raw images are what are produced by devices such as video cameras and scanners.

Processed images are simply raw images that have had at least one stage of image processing applied to them.

This terminology is of relevance to image file naming conventions (p.25) and to the *Image Library Catalogue* (p.281).

Image Directories and Formats

Note that if you are using a Macintosh computer, then a 'directory' is the same thing as a 'folder'.

All the example images used in HIPR are stored in the same directory: the `images` directory. This directory is just one of several which make up the HIPR system, all of which branch off from the HIPR root directory. This directory structure is explained in more detail in the section on *Directory Structure* (p.20).

To get to the images directory from the `html` directory (which is where the HTML files for the hypermedia version of HIPR are stored), first go up one directory to the HIPR root directory. Within this directory will be a sub-directory called `images`. This is the directory where all the image files will be found.

Again, your system supervisor should be able to help. Alternatively you might find details in HIPR's Local Information section (p.40).

Within the image directory, the images are, by default, stored in GIF (**G**raphic **I**nterchange **F**ormat) files. This is a very common and convenient image format that is understood by many image viewers and image processing packages. However, your system supervisor *may* have chosen to use a different format if, for instance, the particular image processing package you use does not accept GIF. Converting the image files to a different format is described in the *Installation Guide* (p.31).

3.3 Filename Conventions

HIPR consists of a very large number of separate files (more than 2000 in fact), spread over nine main directories. In order to help keep track of this multitude we have adopted a standard naming convention for files. The purpose of the convention is to allow the user to ascertain as far as possible what a particular file is, merely by looking at its filename.

In coming up with a convention we ran into some difficulties. The main problem was the requirement to maintain portability to MS-DOS with its severe restriction on filename length. Specifically, an MS-DOS file can only consist of a *maximum* of eight characters, plus, optionally, a full-stop (period) and a three character *filename extension*. Because of this limit many filenames are much

more terse and cryptic than we would like, which makes understanding the convention we use even more important.

The following sections detail the various categories of files within HIPR and the naming conventions that apply.

HIPR Source Files

Found in the `src` sub-directory.

The textual information in HIPR is split into many small chunks, each of which correspond to a 'page' in the hypermedia version. Each page has a name which is intended to convey its meaning in eight characters or less. For instance the name of this page is `filename`. Then the filename of the HIPR source file corresponding to that page is formed by adding the extension `.hpr`. Hence the HIPR source file for this page is called `filename.hpr`.

Many pages, particularly worksheet pages, also have a local information section which is *included* into the main file from a separate file. These included files have names formed by adding `.loc` to the page name.

HTML Files

Found in the `html` sub-directory.

For every HIPR source file there is a corresponding HTML file with a similar name, but with the extension `.htm` instead of `.hpr`.

So the HTML file corresponding to this page is `filename.htm`.

The `html` directory also includes a few small GIF image files that are used as decorations in the hypermedia version of HIPR.

L^AT_EX Files

Found in the `tex` sub-directory.

Similarly, for every HIPR source file there is a corresponding L^AT_EX file with a similar name, but with the extension `.tex` instead of `.hpr`.

The L^AT_EX file corresponding to this page is `filename.tex`.

There is also a file called `hipr_top.ps` which is a PostScript copy of the hardcopy version of HIPR. See *Producing the Hardcopy Version of HIPR* (p.26) for details.

There may also be in this directory various files with the extensions `.aux`, `.log`, `.toc`, `.bbl`, `.blg` and `.dvi`. These are simply side products of the L^AT_EX generation process.

Equation Files

Found in the `eqns` sub-directory.

Equation image files for use with the hypermedia version of HIPR begin with the letters `eqn...`, and have the extension `.gif`. The second part of the main name normally gives some indication in which worksheet the equation is used. There may also be an index number if there is more than one equation used in a that worksheet. For example, `eqnrob1.gif` is the first equation found in the Roberts Cross (p.184) worksheet.

Some very common equations that are used in more than one worksheet have names that indicate what they are rather than to which worksheet they belong to. So, for instance, `eqntheta.gif` is an equation for the Greek letter θ .

This technique of including equations in the hypermedia version of HIPR as inline images is necessary because, at the time of writing, Netscape cannot generate mathematical symbols itself.

Figure Files

Found in the `figs` sub-directory.

All figures and diagrams for use with HIPR are included in two different forms — an *encapsulated PostScript* form for inclusion into the hardcopy version, and a GIF version for inclusion into the hypermedia version. The PostScript files have the extension `.eps` and the GIF files have the extension `.gif`. The rest of the filename is the same for both formats, and is chosen to describe the figure. There are no special conventions for choosing this name, although the name often reflects the name of the worksheet that includes the figure. For instance the figure used in the section on directory structure (p.20) is contained in files called `direct.gif` and `direct.eps`.

Image Files

Found in the `images` sub-directory.

Images have one of the more complicated naming conventions — this is in an attempt to convey as much information as possible in the filename, while conforming to the MS-DOS constraint that filenames can only be 8 characters long. Images fall into two categories: raw and processed, with slightly different naming conventions. See the section on *Images* (p.23) for an explanation of these terms.

By default images are stored in GIF format and have the extension `.gif`. However, your system installer may have changed this (see the *Installation Guide* (p.28) for details) and so another extension may be used.

For raw images, the main part of the image filename (*i.e.* before the extension) is simply a four-character identifier for that image. Typically this identifier takes the form of a three-letter abbreviation indicating the type of image involved, plus a single digit differentiating different images within that category. For example `scr1.gif` is a picture of a jumble of objects, including principally a screwdriver. `fce3.gif` is a picture of a face, and the third example in the face series.

For processed images, the main part of the image filename is split into two four-character halves. The first half is simply the four-character name of the raw image from which the processed image was produced. The second half describes the image processing operation most relevant to the production of the image. As with raw image names, this description consists of a three-letter abbreviation indicating the type of image processing operation, and then a final single digit which differentiates between different examples of that operation applied to the raw image concerned.

Some examples might make this clearer:

`wdg2.gif` is a raw image showing the second of a family of two dimensional silhouetted ‘widgets’.

`wdg2thr1.gif` is the same image after it has been thresholded (p.69) using one set of parameters.

`wdg2thr2.gif` is the image after thresholding with a slightly different set of parameters.

`wdg2sob1.gif` is the result of applying the Sobel edge detector (p.188) to `wdg2.gif`.

`cln1.gif` is a raw image containing a picture of a clown.

`cln1sob1.gif` is the result of applying the Sobel edge detector to `cln1.gif`.

This convention is a little confusing at first, but fortunately you will not have to worry about it most of the time, since in the hypermedia version of HIPR at least, images are displayed merely by clicking on their ‘thumbnails’. In addition, the *Image Library Catalogue* (p.281) lists all the images in HIPR in a much more user-friendly fashion.

Thumbnail Files

Found in the `thumbs` sub-directory.

For every image in the `images` sub-directory, there is a corresponding *thumbnail* image. These thumbnails are just small versions of the GIF files in the `images` directory and are GIF files themselves. They have similar names to their corresponding images, except that they have the upper-case extension `.GIF` instead of `.gif`. This is to avoid confusion with the corresponding full image. Note that on operating systems that ignore case in filenames (such as MS-DOS), this difference will not be apparent. As an example, the thumbnail associated with the image `wdg2thr1.gif` is called `wdg2thr1.GIF`.

Index Files

Found in the `index` sub-directory.

Index files are written by the HIPR generating programs in order to keep track of index entry information. They are not normally of interest to the user and should not be altered. There is one index file for each HIPR source file, with a similar name, except that they have the extension `.idx` instead of `.hpr`.

Program Files

Found in the `progs` sub-directory.

The HTML and \LaTeX files are generated from HIPR source files using *Perl scripts*. These scripts are called `hiprgen.pl` and `hiprindx.pl`.

3.4 Producing the Hardcopy Version of HIPR

As mentioned in earlier introductory sections, HIPR can be used in two main forms — on-line, using a hypermedia browser such as Netscape, or as hardcopy, *i.e.* to be read like a book. You may already have some copies of the hardcopy version, but there will almost certainly be times when you want to generate more. This section describes how you go about doing that.

Printing HIPR on a PostScript Compatible Printer

The complete text of HIPR is included in the distribution package as a PostScript file. The file is called `hipr_top.ps` and can be found in the `tex` sub-directory of the `hipr` root directory. This file can simply be sent straight to any PostScript compatible printer for printing. The method of doing this varies tremendously from system to system, so consult your own system's documentation for details.

Obviously this method is only applicable if you have a PostScript printer (many laser printers are PostScript compatible, however, or can be made so relatively easily). If you do not have access to such a printer, then you might be able to arrange with a third party to print the file — for instance many small-scale printing shops provide this service. Alternatively, you may be able to use \LaTeX to regenerate the printable file in a format more suited to your particular printer (you will probably still need a laser or inkjet printer however).

Regenerating HIPR Hardcopy using \LaTeX

The hardcopy version of HIPR is also included in the form of \LaTeX source files. As with the PostScript file, these are found in the `tex` sub-directory of the `hipr` root directory.

Note that we do not have the facilities to provide support for people with problems installing \LaTeX , or running \LaTeX on the HIPR \LaTeX source files. This facility is provided ‘as is’ and is under no guarantee. If you just want to print out hardcopy, then we really suggest that you find someone who knows how to print the PostScript file included with HIPR. Having said that, the following sections do attempt to provide some guidelines as to what can commonly go wrong, and what to do about it.

\LaTeX (usually pronounced ‘lay-tek’ — the last ‘X’ is actually a Greek ‘Chi’) is a freely available (and free) typesetting program that is widely used in many academic institutions and elsewhere, around the world. The program works on input files containing text that has been ‘marked up’ using \LaTeX formatting commands. You do not need to be able to understand these in order to run \LaTeX on the HIPR \LaTeX files, but essentially they consist of instructions and suggestions as to how the text should be formatted and presented so as to look good on paper. The end result of ‘running’ \LaTeX on these files is to produce a printable file representing the marked up text.

Versions of \LaTeX are available for most modern computer platforms. Information on obtaining it for your system is given in the *Installation Guide* (p.31).

If you have \LaTeX properly installed on your system then generating a printable file is very simple. There are basically three stages.

1. Run \LaTeX on the file `hipr_top.tex` in the `tex` sub-directory of the `hipr` root directory. This will produce a number of files, amongst them a *DVI file* which will probably be called `hipr_top.dvi`. This file contains the important bit.
2. Convert the DVI file to a printable file. Usually, you convert the DVI file to a PostScript file, but it is possible to convert it to other printable formats as well, suitable for non-PostScript compatible printers. The program you use to do this varies tremendously from computer to computer, and even on a single computer system, there may be several slightly different conversion programs available. You will have to consult the documentation that comes with the version of \LaTeX for your system for details. As an example, a popular DVI-to-PostScript conversion program used on UNIX systems is called `dvips`.
3. Print the printable file. Again the method of doing this varies from system to system, so consult the documentation that came with your computer system and printer for details.

For example, the following sequence of commands would do the job on my UNIX system. (Assume that I have already entered the `tex` directory.)

```
latex hipr_top
dvips hipr_top
lpr hipr_top.ps
```

Other things to note:

- When \LaTeX runs, it may generate lots of messages, including several warning messages about ‘Overfull \vboxes’ and the like. These are mostly just \LaTeX being fussy and can safely be ignored.
- However if \LaTeX stops with a question mark and reports an error, then you are in trouble. If you know \LaTeX then you may be able to look at the source files and fix the problem yourself. If not, then the most likely reason for failure is that you have not properly installed either HIPR or \LaTeX . Here are some things to try:
 - \LaTeX needs to be run from within the `tex` sub-directory ideally (this may not apply to some window based systems).

-
- It uses files with extension `.tex` from the `tex` sub-directory, and files with extension `.eps` from the `figs` sub-directory. If the L^AT_EX error message is claiming that it cannot find one of these files, then check to see that that file exists and is readable (*i.e.* not protected in some way).
 - The HIPR L^AT_EX source files require the `epsf` and `fancyheadings` L^AT_EX macros to be installed. See the *Installation Guide* for details.
 - You may have incorrectly installed some other part of the L^AT_EX system. For instance some fonts may not be available. Check with the L^AT_EX documentation.
 - If you have been modifying bits of HIPR (as explained in *Making Changes* (p.34)), then you may have caused an incorrect piece of L^AT_EX to be generated. Most such errors are picked up during the HIPR regeneration process, but there may be some errors that get through. If possible, try to undo the changes that you have made until you get back to a stage where everything works. Then progress forward again and try to find a different way of doing what you were trying to do originally.
- When L^AT_EX has finished, it may print a message that goes something like:

Label(s) may have changed. Re-run to get cross-references right.

If this happens, just re-run the L^AT_EX program until the message goes away.

3.5 Installation Guide

Note that this section is really only relevant to the person responsible for installing and maintaining HIPR on your system. Note also that the FTP addresses given on this page are correct at the time of writing but cannot be guaranteed to remain so.

Be sure to read the architecture-specific instructions (in `README_xxx`, `xxx=UNIX`, etc) for unpacking HIPR's files before working through these instructions.

Unpacking the Core HIPR Distribution

The HIPR software is distributed in a variety of ways. You may have received it on computer tape, or on CD-ROM, or perhaps you have downloaded it directly over the Internet from an authorized FTP site. Each of these distribution methods requires slightly different approaches to unpacking and installing the files on your system. For each distribution type, the unpacking procedure is described in a text file called `README_xxx` that will be included in unpacked, uncompressed form in the same place as rest of the HIPR package. Consult this file for the basic installation procedure.

No matter which distribution you start with, the end result of the unpacking process is the same — you should end up with the HIPR directory structure as described in *The Directory Structure of HIPR* (p.20) installed somewhere on your filesystem. Note that if you wish HIPR to be accessible from your local area network, then you should make sure that you install the HIPR root directory in a location that is visible from that network.

Copyright Note

You should be careful when installing HIPR on your system, that it cannot be retrieved by remote sites via the Internet from your machines. This would break the conditions of the HIPR copyright (p.272). In particular, you should make sure that HIPR is *not* accessible via the World Wide Web. If you choose to run HIPR from a Web server at your site then you must ensure that the server is configured so that only users on your local area network can access HIPR. You should also not place HIPR upon FTP servers or other publicly accessible archives.

Starting up HIPR

Once you have unpacked the core distribution, you are almost ready to begin using HIPR. If you have a graphical HTML browser (*e.g.* Netscape) already installed on your system, then simply point it at the file `hipr_top.htm` in the `html` sub-directory. How you do this depends upon exactly where you have inserted the HIPR directories into your system. For instance, if you are using a UNIX system, and the `hipr` root directory has been copied to `/usr/docs`, then you would give Netscape the following URL (Uniform Resource Locator):

```
file://localhost/usr/docs/hipr/html/hipr_top.htm
```

Note that, if possible, you should *not* access HIPR via your system's World Wide Web server (if one is installed), *i.e.* you should not use a URL beginning with `http://...` This method may work, but it will be much slower than access in local file mode using the `file://localhost...` URL. In addition, you may be breaking the HIPR copyright (p.272) if HIPR is available over the World Wide Web to other sites.

If you wish to print out the hardcopy version of HIPR, then consult the section on *Producing the Hardcopy Version of HIPR* (p.26).

HTML Browsers

The above section assumes that you have a HTML/WWW browser such as Netscape already installed on your system. If you do not, then you will have to get one in order to use the online version of HIPR.

The recommended HTML browser for use with HIPR is Netscape. This is most easily available via FTP (File Transfer Protocol) from the Netscape Communications' FTP server and is free for non-commercial use (consult the license that comes with it for details). The FTP server contains binary executables for Netscape on a variety of computer platforms: PCs running Microsoft Windows, Apple Macintoshes and a large number of different UNIX systems running X Windows.

To use FTP you first need a connection to the Internet and appropriate FTP software. Your Internet connection provider will be able to tell you how to get the FTP software if you don't have it. Consult your documentation for details of how to use this software. Then:

1. Connect to `ftp.netscape.com` using your FTP software (*e.g.* type `ftp ftp.netscape.com`). If you are in Europe you may find it faster to connect to `ftp.enst.fr`.
2. Log onto the remote FTP server, by giving `ftp` as your login name and your e-mail address as the password.
3. Change to the `/pub/navigator/2.02` sub-directory on the remote machine (*e.g.* type `cd /pub/navigator/2.02`). Newer versions may also be provided in other directories by the time you read this (for instance, a beta test release of version 3.0 is also available). (On `ftp.enst.fr` you need to go to `/pub/netscape/navigator/2.02`.)
4. Change to the sub-directory dealing with the version of Netscape you require (*e.g.* type `cd unix`).
5. Retrieve the `README` file contained in that directory and follow the instructions contained within for downloading and unpacking the Netscape executable (*e.g.* start by typing `get README`). Note that you will have to put your FTP program into binary mode for transferring the executable.

If you do not have an Internet connection then you will have to obtain the software from your local computer vendor. The disadvantage of this is that you may have to pay for a commercial version of something that is free over the Internet. Unfortunately we cannot supply Netscape in the HIPR distribution, since this would violate its distribution license.

A Note on Browser Compatibility

It is unfortunately not the case that all browsers, even all graphical browsers, are the same. The HTML language is in a process of evolution, so there exist browsers which support features that have not yet become standard HTML, and there exist browsers that do not yet completely support the latest standard HTML definition. Netscape, the standard browser for use with HIPR, is one of the more feature-laden browsers around, and we have made use of a few of these features (such as centered text) even though they are not universally implemented yet. Other browsers, such as Mosaic, do not, at the time of writing, support some of these features. Therefore certain bits of HIPR may look odd when viewed with Mosaic, although all parts of HIPR should be adequately displayed.

For a summary of some other problems that may occur with Netscape and with browsers in general, see the section on *Possible Problems* (p.14) in the introduction to hypermedia browsers.

Using External Image Viewers

Netscape is able to display the GIF images that make up the HIPR image library by itself. However, by default, such images are displayed in the same Netscape window that was previously showing the hyperlink to that image. This means that when the user clicks on an image thumbnail in order to display the full image, the full sized image *replaces* the thumbnail (and all the surrounding explanatory text) in the Netscape window. Which makes it very difficult to read the text describing the image while viewing that image!

We have found that a better approach is to get Netscape to spawn off an *external image viewer* to display the full sized GIFs. This allows the user to continue to read any explanatory text, while the full-sized image is displayed in a compact separate window.

The **Help** menu of Netscape provides information on obtaining a suitable viewer and on configuring Netscape to use that viewer. However, we present here a brief summary of how this would be done on a typical UNIX system.

1. First obtain a suitable image viewer. An excellent viewer for X-Window systems is **xv**, obtainable via FTP from `ftp.x.org` (or a suitable mirror site) in the `R5contrib` directory.
2. Under the **Options** menu on Netscape select **Preferences...** and then **Helpers**. The two files you are interested in are the global and local *mailcap* files. They tell Netscape how to deal with data of different types. Note down the locations of these files.
3. Now you must edit one of these files to tell Netscape to use an external viewer to display GIF files. If you edit the global file, then the changes you make will apply to everyone who uses the same global mailcap file (typically everyone else on the same LAN). If you edit the local file, then the changes will only affect the user who owns that local file. Entries in the local file have a higher priority than entries in the global file, where there is a clash.
4. Assuming you will be using **xv** as the viewer, insert the following line into the appropriate mailcap file:

```
image/gif; xv %s
```

5. Restart Netscape. The browser should now use **xv** to display GIF files. Note that this does not affect the display of inline GIFs within HTML files.

Image Processing Software

While strictly not part of HIPR, it is extremely useful to have an image processing software package available on your system. A huge variety of these are available, and if you are using HIPR at all, it is likely that you already have one installed on your system. We list here just a couple of packages that are available free via the Internet.

Khoros Khoros is a very powerful (and very large) visual programming environment that includes extensive facilities for image processing and image manipulation. For more information, use Netscape to connect to:

`http://www.khoros.unm.edu/`

NIH Image A free image processing package available for the Apple Macintosh. Available via FTP from `zippy.nimh.nih.gov` or `alw.nih.gov` in the `/pub/image` sub-directory.

Additional information on obtaining some common image processing packages is given in the appendix on *Common Software Implementations* (p.244).

Image Format Converters

Many image processing packages will perform image conversion for you so check to see if your image processing package can help. Failing that, if you wish to convert the GIF images in the HIPR image library into different formats then you must obtain some image conversion software. Again, we list only a few popular packages that are available for free over the Internet.

PBMplus Toolkit This toolkit is a very popular set of programs that allow you to convert between just about any formats you like, via an intermediate format. It is available via FTP from `export.lcs.mit.edu` in the directory `/pub/contrib/pbmplus` or from `ftp.ee.lbl.gov`.

Utah Raster Tools Another very popular conversion toolkit. It can be obtained via FTP from `wuarchive.wustl.edu` in the directory `/graphics/graphics/packages/urt`.

Image Alchemy An MS-DOS version of this is available via FTP from `wuarchive.wustl.edu` in the directory `/msdos/graphics`.

This information represents just a small part of the *alt.binaries.pictures* FAQ (Frequently Asked Questions). The full version which lists many more utilities can be obtained via FTP from `rtfm.mit.edu` in the directory `/pub/usenet/alt.binaries.pictures.d` as `/alt.binaries.pictures_FAQ_-_OS_specific_info`.

Obtaining L^AT_EX and Associated Software

L^AT_EX is a typesetting program, extremely popular in academic circles, that relies on a *markup language* in order to allow the user to specify the structure of a document without having to worry about its appearance too much. You need to install it (if you don't have it already) only if you wish to *regenerate* the hardcopy version of HIPR, perhaps after making some changes to it. Note that you don't need L^AT_EX if all you want to do is print out the distributed hardcopy version since there is a ready made PostScript file for this purpose included in the HIPR distribution. Nor do you need it to use the online version of HIPR. See the section on *Producing the Hardcopy Version of HIPR* (p.26) for more information on how L^AT_EX is used with HIPR.

L^AT_EX is itself a large collection of programs and utilities and the installation process is unfortunately not simple. It is however available for free and for a wide variety of machine architectures. To obtain L^AT_EX you will need to connect via FTP to your nearest *CTAN site* (Comprehensive TeX Archive Network). The three major CTAN sites are:

- `ftp.dante.de`
- `ftp.tex.ac.uk`
- `pip.shsu.edu`

There is a slightly user-friendly front end to these archives which can be accessed using Netscape at

<http://jasper.ora.com/ctan.html>

Note that \LaTeX itself is ‘just’ a collection of *macros* which sits on top of another program called *TeX*, which you will also probably have to retrieve from the same archive.

There are some other programs which you will also need to obtain from the same archive. Firstly, you require a program to translate the DVI files that \LaTeX outputs into something you can print, normally PostScript. The recommended program to do this is called *dvips*. You also need to make sure that you get hold of the *epsf* and *fancyheadings* macro packages for use with \LaTeX , or you will find that things don’t work. Finally you may find it convenient to obtain a DVI previewer which will allow you to display the \LaTeX output on screen without printing it. A common previewer for UNIX systems running X-Windows is called *xdvi*.

Finally, you should be aware that there is a relatively new version of \LaTeX called *LaTeX2e*. This should be compatible with the \LaTeX files contained in HIPR. If it isn’t then use the older version of \LaTeX .

Extra Requirements for HIPRscript

The *HIPRscript Reference Manual* (p.253) describes in detail how it is possible to make changes to the core HIPR documentation by editing the appropriate HIPRscript source files and then regenerating HTML and \LaTeX output files. To do this you need to have the language *Perl* installed on your system, so that you can run the HIPRscript translation program *hiprgen.pl*. Perl is a very widely available, fast and extremely portable language and for those reasons *hiprgen.pl* was written in that language. It is significantly more portable than ‘C’ for instance.

Perl is obtainable via FTP from a great many sites and for a variety of machines. There are currently two versions available: Version 4 (patchlevel 36), also known as Version 4.036; and Version 5. Either version should work with *hiprgen.pl*.

If you have a UNIX or VMS machine then you should obtain Perl from one of the following sites (this list was taken from the *comp.lang.perl* FAQ).

Site	Directory
-----	-----
North America:	
ftp.netlabs.com	/pub/outgoing/perl[VERSION]/
ftp.cis.ufl.edu	/pub/perl/src/[VERSION]/
prep.ai.mit.edu	/pub/gnu/perl5.000.tar.gz
ftp.uu.net	/languages/perl/perl5.000.tar.gz
ftp.khoros.unm.edu	/pub/perl/perl5.000.tar.gz
ftp.cbi.tamucc.edu	/pub/duff/Perl/perl5.000.tar.gz
ftp.metronet.com	/pub/perl/sources/
genetics.upenn.edu	/perl5/perl5_000.zip
Europe:	
ftp.cs.ruu.nl	/pub/PERL/perl5.0/perl5.000.tar.gz
ftp.funet.fi	/pub/languages/perl/ports/perl5/perl5.000.tar.gz
ftp.zrz.tu-berlin.de	/pub/unix/perl/perl5.000.tar.gz
src.doc.ic.ac.uk	/packages/perl5/perl5.000.tar.gz
Australia:	
sungear.mame.mu.oz.au	/pub/perl/src/5.0/perl5.000.tar.gz
South America (mirror of prep.ai.mit.edu:/pub/gnu):	

```
ftp.inf.utfsm.cl      /pub/gnu/perl5.000.tar.gz
```

If you have a non-UNIX or VMS machine, then use one of the following sites:

Machine/OS Site Directory

```
-----
MS-DOS ftp.ee.umanitoba.ca /pub/msdos/perl/perl4
MS-DOS ftp.einet.net /pub/perl5
MS-DOS ftp.khoros.unm.edu /pub/perl/msdos
MS-DOS ftp.ee.umanitoba.ca /pub/msdos/perl/perl5
Windows/NT ftp.cis.ufl.edu
Macintosh nic.switch.ch /software/mac/perl
Macintosh ftp.maths.tcd /pub/Mac/perl-4.035
OS/2 ftp.cis.ufl.edu /pub/perl/src/os2
Amiga ftp.wustl.edu /pub/aminet/dev/lang
Amiga ftp.doc.ic.ac.uk /pub/aminet/dev/lang
```

When installing Perl, you may find it useful to consult the *comp.lang.perl* FAQ which is available from that newsgroup on USENET, or via Netscape at:

```
ftp://rtfm.mit.edu/pub/usenet-by-hierarchy/news/answers/perl-faq
```

Configuring HIPRscript for your System

Before you can use HIPRscript, you may have to make some minor modifications to `hiprgen.pl` and `hiprindx.pl` in the `progs` sub-directory. Most notably, on non-UNIX systems, you should change the pathnames that allow HIPRscript to locate the other HIPR directories from the `src` sub-directory.

If you edit the `hiprgen.pl`, you will see at the top a list of variable definitions like this:

```
# Directory where various files are to be found. Note that trailing slash
# IS required!

$IMAGE_DIR = '../images/';
$TEX_DIR = '../tex/';
$HTML_DIR = '../html/';
$EQNS_DIR = '../eqns/';
$FIGS_DIR = '../figs/';
$THUMB_DIR = '../thumbs/';
$INDEX_DIR = '../index/';
```

These variables by default hold UNIX relative pathnames from `src` to the other HIPR sub-directories. If these are not appropriate for your system then you must change them to suit. For instance, on MS-DOS systems, you would change this section to:

```
# Directory where various files are to be found. Note that trailing slash
# IS required!

$IMAGE_DIR = '..\images\';
$TEX_DIR = '..\tex\';
$HTML_DIR = '..\html\';
$EQNS_DIR = '..\eqns\';
$FIGS_DIR = '..\figs\';
$THUMB_DIR = '..\thumbs\';
$INDEX_DIR = '..\index\';
```

The change here being that the UNIX '/' directory separator has become a '\' suitable for MS-DOS. Simply change these pathnames to something more suitable for your system. Before you do this, however, check to see that Perl doesn't automatically work out what you mean without you having to change anything (Perl was developed on UNIX systems and may understand the meaning of UNIX pathname syntax on other systems).

Once you have changed `hiprgen.pl`, edit `hiprindx.pl` and make the same changes there (note that fewer lines will need to be altered).

There are some other things you may like to change in the `hiprgen.pl`, but we recommend that you don't unless you know Perl and have a good idea what you are doing. No other changes should be necessary anyway. The comments in the `hiprgen.pl` file describe what the other introductory parts of the file do, if you're interested.

Automatic Generation of Figures, Equations and Thumbnails

Figures, equations and `\imagerref` tags all require the presence of external files as described in the *HIPRscript Reference Manual* (p.265). If you have a UNIX system, then it is possible (although fiddly) to get HIPRscript to generate some of these files automatically. To do so you will need to do several things:

- The shell scripts `hfig2gif`, `heqn2gif` and `himg2thm` should all be copied from the `progs` directory to a directory somewhere on the path defined by the `$PATH` environment variable. Alternatively you can just add the full pathname of the `progs` sub-directory onto the end of the `$PATH` environment variable, which saves having to copy any files.
- The file `pstoppm.ps` should also be copied from the `progs` sub-directory and put somewhere appropriate. It doesn't really matter where, although you should note that it is not a directly executable file (it is actually a PostScript program for use with GhostScript).
- The shell scripts `hfig2gif` and `heqn2gif` should be edited to reflect the location of `pstoppm.ps`. Instructions on doing this are given in those files.
- You need to have the PostScript previewer *GhostScript* installed on your system. You can obtain information about GhostScript on the World Wide Web by pointing Netscape at
`http://www.cs.wisc.edu/ghost/index.html`
- You also need to have the `giftrans` program installed and on your path. This can be obtained via FTP from `ftp.rz.uni-karlsruhe.de` in the directory `/pub/net/www/tools`.
- You must have L^AT_EX installed on your system as described above.
- You must have the PBMplus library installed on your system as described above.
- Finally, edit the `hfig2gif`, `heqn2gif` and `himg2thm` scripts so that the names of the various utilities used by them correspond to the names of utilities installed on your system.

If you have done all of this, then edit `hiprgen.pl` as described in the *HIPRscript Reference Manual* (p.265) in order to enable automatic equation, figure and/or thumbnail generation, and test it out. Good luck!

The UNIX scripts have been tried out on Sun SparcStations running both SunOS 4.1.3 and Solaris 2.3. They should however work on most other varieties of UNIX without any changes.

3.6 Making Changes to HIPR

Note that this section is really only relevant to the person responsible for installing and maintaining HIPR on your system.

Introduction

The HIPR system is intended to be immediately usable by a wide variety of people using a wide variety of different computer systems and image processing software. Given this fact, it is almost inevitable that there will be things that you don't like about it! Perhaps the images are in the wrong format, or perhaps you would like to add some extra information to the worksheets for students to read. Perhaps you would even like to add an extra worksheet yourself, explaining some unusual operator which is particularly relevant to your situation. HIPR allows you to do all of this — if you're prepared to put in a little work.

WARNING!

Note that the facilities described here for altering HIPR to further suit your needs are provided 'as is'. We are not able to offer technical support to anyone having difficulties making any of the modifications we discuss. While we have tried to make the task as simple and robust as possible, the large number of different architectures on which HIPR can be used make it impossible to describe every situation. As a result you make any such changes entirely at your own risk. For your own protection we recommend that you maintain a backup copy of the original HIPR installation which you can turn back to should any attempted modification go wrong.

How HIPR is Generated

As mentioned elsewhere (p.11), the HIPR package is provided in two forms: a hypermedia version and a hardcopy version. The hypermedia version consists of a collection of HTML files (HyperText Markup Language) and associated GIF and image files, that can be displayed using a graphical HTML browser such as Netscape. The hardcopy version is supplied in PostScript form, which in turn is generated from L^AT_EX source files (also supplied) as described in the section on *Generating Hardcopy* (p.26).

Since both hypermedia and hardcopy versions ought to contain exactly the same up-to-date text, it would be a time-consuming and error-prone process if one had to make separate changes to both versions when any change to the information content of HIPR were made. Therefore, HIPR is initially written in a specialist intermediate language known as *HIPRscript*. The HIPRscript files are then automatically translated into both HTML and L^AT_EX versions (and the L^AT_EX version is subsequently turned into PostScript for printing). The HIPRscript source files are contained in the `src` sub-directory. The programs to perform the conversion are contained in the `progs` directory and are written in a language called *Perl*.

The completely professional way to make *major* changes to HIPR is therefore to first edit the HIPRscript files and then to run the conversion program. This ensures that both hypermedia and hardcopy versions contain up-to-date material. It also helps enforce a certain consistency of style.

However, running the HIPRscript conversion programs requires that a number of additional utilities be installed on your system (as detailed in the relevant section of the *Installation Guide* (p.32)), and these may prove to be difficult or time-consuming to set up, particularly on non-UNIX machines. Fortunately, many changes can be made without resorting to the full machinery of HIPRscript. The following sections describe a number of common ways in which you might want to change the HIPR system, and how you should go about doing it in the easiest way possible. If, having read these sections, you decide that you want to delve even further into the deeper mysteries of HIPRscript, then the HIPRscript Reference Manual (p.253) should contain all you need.

3.7 Changing the Default Image Format

The example image library supplied with HIPR contains over 700 images, all in the GIF (Graphic Interchange Format) image format. GIF was chosen because it is widely used, efficient to store (it is a compressed format) and also readily converted into other formats using standard image manipulation tools.

The image viewer supplied with your hypermedia browser will almost certainly be able to display GIF images, since GIF is a very popular format on the World Wide Web, and the browsers were initially developed for that environment.

In addition it is highly likely that your image processing software will be able to read in GIF images directly since they are such a common format, so you may be able to use the image library as a starting point for further exploration of image operators without altering the image files at all.

In an ideal world that would be that. However, in practice there are multitudes of different image formats in use today, and there do exist some image processing packages that are not able to import GIF images directly. For these reasons it is sometimes necessary to convert images from GIF format into some other format. There are three main approaches that you could take.

Firstly, if HIPR works fine as a reference manual using GIF images and you only want to use a few images for input into your image processing software, then it is easiest to keep the GIF images as they are for use with HIPR and just make duplicate versions of the images you are interested in, in a different format as necessary. Software to perform this conversion is widely available, and may even be included with your image processing package. Some suggestions for converters are given in the *Installation Guide* (p.31). No messing around with HIPRscript is required for this approach.

Secondly, if you have the disk space, you might just consider making a *duplicate* image directory containing copies of all the image files in the required new format. The hypermedia version of HIPR will continue to use the GIF images, but for further exploration you can direct students to use the alternate image directory. Since converting over 700 images by hand is very tedious, it is a good idea to automate the conversion process by writing some sort of *script* or *batch file*. Since the details of how you would do this are extremely software- and machine-dependent we cannot tell you exactly what to do here.

Finally, and only pursue this route if you are aware that it involves a major amount of work, and if you *really* don't have enough filespace to maintain a duplicate image library in a different format, you can choose to convert *all* the images into another format. The disadvantage of doing that is that you also have to change every single image reference in the HIPR HTML files to reflect the new file suffix of your images (which will presumably no longer be *.gif*). In addition, you must ensure that your hypermedia browser is able to display that image file format and knows how to do so (consult your Netscape documentation for details of how to link image viewers to file suffixes).

If you want to take this approach then do the following:

1. First check that all the HIPRscript utilities are installed correctly and that you are comfortable making small changes to HIPR using the HIPRscript programs as detailed under *Making Changes using HIPRscript* (p.253).
2. Make sure that your hypermedia browser is capable of displaying the image format you would like to use. Normally this will involve making sure that Netscape recognizes the new image file extension as an image file and knows which viewer to spawn in order to display it. Check your browser documentation for details. Look particularly for information on *mailcap* and *MIME types*.
3. Convert all the images into the new format. This will probably involve a change to the filename extensions of the images, but you should keep the rest of the filename the same. *e.g.* if you were turning the images from GIF into TIFF format, you would probably change the filename *wdg2.gif* into *wdg2.tiff* or *wdg2.tif*. As mentioned above you may find it useful to find some way of automating the conversion process. Be very careful that while converting

images you don't accidentally lose some! It is a very good idea to maintain a backup of all the GIF files somewhere until the conversion job is entirely finished.

4. Now edit the file `hiprgen.pl` in the `progs` sub-directory. Near the top of the file should be a line saying:

```
$IMAGE_EXT = 'gif'
```

Change this to reflect the new image filename extension. Do not include the '.' here. Do not change anything else. Save the file.

5. Now regenerate every HTML file in HIPR. See *Making Changes using HIPRscript* (p.253) for details.
6. Test to see that it works! If it doesn't then we are afraid that you are on your own. In the worst case, however, it should be possible to return to where you started from by simply reversing the above steps.

3.8 Adding Local Information

Introduction

HIPR contains a large amount of information about image processing expressed in fairly general terms. However, image processing is very much a practical subject and the details of how you actually *do* a particular image processing task depend on the details of the system and software you are using. Since we cannot possibly know all the details of your particular system, HIPR provides a convenient way of allowing HIPR maintainers to add *local information* into HIPR. Local information can be used to describe such things as how to start up the local image processing package, or who to ask for help. It can also be inserted into a particular operator's worksheet in order to highlight peculiarities of the local implementation of that operator. These are only a few of the ways in which local information can be used.

Where to Add Local Information

There are two main places where local information can be added. Information that applies to HIPR in general, *e.g.* how to start up image processing packages, or who to contact for help, should be added to the introductory section entitled *Local Information* (p.40). On the other hand local information that pertains to a particular operator worksheet should be added to the *Local Information* section at the bottom of that worksheet.

The HTML and L^AT_EX files for *general* local information are generated from a HIPRscript file called `local.hpr`. This file is in fact just a wrapper for a file called `local.loc` which is where the actual information should go. The `local.loc` file is also contained in the `src` sub-directory.

Specific local information for each operator is also contained in files with a `.loc` extension. There is one such file for each operator, so that for instance, the local information for the morphological dilation (p.118) worksheet is contained in a file called `dilate.loc`.

In general therefore, local information sections in HIPR are generated from source files contained in the `src` sub-directory with the extension `.loc`. These files are just standard HIPRscript files with a non-standard filename extension (all other HIPRscript files have the extension `.hpr`). The HIPRscript translation process takes this information and inserts it into appropriate HTML and L^AT_EX files.

Therefore, one way of adding local information is to edit the appropriate `.loc` files, and then regenerate the corresponding HTML and L^AT_EX files using `hiprgen.pl`, as described in *Making Changes Using HIPRscript* (p.253).

An alternative way is just to edit the appropriate \LaTeX and HTML files directly to add the information. This has the disadvantage that two sets of files must be edited to keep hypermedia and hardcopy version up to date with one another, and also, if at some time in the future you do decide to use HIPRscript, your changes will be overwritten. See *Editing \LaTeX and HTML Files Directly* (p.38) for more details.

Both these methods have problems, so to make things easier we have provided a third way to add local information, as described in the next section.

Local .txt files

Since using HIPRscript is not trivial, and editing long HTML and \LaTeX files by hand may be difficult and tedious, we have set things up to make it easy to add information in a limited way without having to run HIPRscript or edit HTML or \LaTeX files directly.

By default, each of the standard supplied .loc files contains little more than a link to a plain text file with a .txt extension. There is one such file for each operator, so that for instance the dilation (p.118) worksheet mentioned above is associated with a file called `dilate.txt`. The .txt files are all contained in a sub-directory of the `src` directory, called `local`.

The .txt files are included directly by both \LaTeX and HTML files, so they cannot contain any \LaTeX , HTML or HIPRscript tags — they can only contain plain text.

Adding local information using these files is simple. Just edit the appropriate file and add whatever information you require, in plain text. It is not necessary to regenerate any HTML or \LaTeX files for these changes to become visible. The disadvantage of this method is that the advanced markup available using HIPRscript is not available. To use this you must edit the .loc files and then regenerate \LaTeX and HTML files as described in *Making Changes Using HIPRscript* (p.253).

3.9 Editing HTML and \LaTeX Directly

Why Direct Edit?

HIPR consists of hypermedia and hardcopy versions, realized by HTML and \LaTeX files respectively. Both these file types are originally generated from common source files written in HIPRscript. Using HIPRscript source files ensures that both HTML and \LaTeX files contain the same information and in addition halves the amount of work required to make changes. Therefore, if at all possible, it is usually easier to make changes to HIPR by first modifying the HIPRscript source files and then regenerating the corresponding HTML and \LaTeX files using the translation programs supplied with HIPR.

However, installing the extra utilities necessary to get the HIPRscript translation working is not trivial, and in fact is very difficult on non-UNIX machines. In addition, whereas many people already understand HTML and \LaTeX , HIPRscript is a new language which you must learn in order to use. For these two reasons therefore, it is sometimes easier to make changes to HIPR by directly editing the HTML and \LaTeX files themselves. This is particularly true if the changes involved are small or you don't anticipate having to make changes very often.

Hints and Tips

Obviously, before you can make changes to HTML and/or \LaTeX files, it is necessary to know a bit about those languages. HTML manuals are readily available on the World Wide Web. If you are connected to the internet, then simply clicking on the **Help** menu of your browser should provide access to such a manual. \LaTeX manuals are more conventionally found in hardcopy form. The standard textbook is Leslie Lamport's book *\LaTeX* , published by Addison-Wesley (ISBN 0-201-15790-X).

HIPR is split up into a large number of files. Before you can make a change to the way HIPR looks it is necessary to work out which of those files contains the information you want to change. The easiest way to do this is by using the hypermedia version of HIPR. Simply go to the page you intend to modify and read off the name of the file in the message window showing you the current URL. The last bit of this gives the name of the relevant HTML file in the `html` sub-directory. The corresponding \LaTeX file will have the file extension `.tex` and will be found in the `tex` sub-directory.

We recommend that when you make changes to HIPR you change both hypermedia and hardcopy versions in order to avoid confusion. This involves making similar changes to corresponding HTML and \LaTeX files.

When you make changes to HIPR by directly editing the HTML and \LaTeX files, you are leaving the original HIPRscript files unchanged. Therefore if at some point in the future you regenerate HIPR from the HIPRscript files, your changes may be overwritten. For this reason, if you think you might be using HIPRscript at some time in the future, then you should probably start using it straight away.

Figures are included as inline GIF files in HTML, but as encapsulated PostScript in \LaTeX . Therefore there are two picture files corresponding to each figure. They will both be found in the `figs` sub-directory and will have similar names, but the GIF file will have the extension `.gif`, whereas the PostScript file will have the extension `.eps`. If you wish to add or replace a figure then you should ensure that the picture is present in both formats.

It is not really practical to alter equations in HTML by direct editing. This is because most current browsers cannot display the necessary symbols, and so equations are included as inline graphics. These graphics are generated directly from the HIPRscript by the translation programs. Therefore if you want to add or modify equations that contain non-ASCII symbols, then you will find it easiest to use HIPRscript.

Chapter 4

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

Part II

Image Processing Operator Worksheets

Chapter 5

Image Arithmetic

Image arithmetic applies one of the standard arithmetic operations or a logical operator to two or more images. The operators are applied in a pixel-by-pixel fashion which means that the value of a pixel in the output image depends only on the values of the corresponding pixels in the input images. Hence, the images normally have to be of the same size. One of the input images may be a constant value, for example when adding a constant offset to an image.

Although image arithmetic is the most simple form of image processing, there is a wide range of applications. A main advantage of arithmetic operators is that the process is very simple and therefore fast.

In many applications the processed images are taken from the same scene at different points of time, as, for example, in reduction of random noise by adding (p.43) successive images of the same scene or motion detection by subtracting (p.45) two successive images.

Logical operators (p.234) are often used to combine two (mostly binary) images. In the case of integer images, the logical operator is normally applied in a bitwise fashion. Then we can, for example, use a binary mask (p.235) to select a particular region of an image.

5.1 Pixel Addition

Brief Description

In its most straightforward implementation, this operator takes as input two identically sized images and produces as output a third image of the same size as the first two, in which each pixel value (p.239) is the sum of the values of the corresponding pixel from each of the two input images. More sophisticated versions allow more than two images to be combined with a single operation.

A common variant of the operator simply allows a specified constant to be added to every pixel.

How It Works

The addition of two images is performed straightforwardly in a single pass. The output pixel values are given by:

$$Q(i, j) = P_1(i, j) + P_2(i, j)$$

Or if it is simply desired to add a constant value C to a single image then:

$$Q(i, j) = P_1(i, j) + C$$

If the pixel values in the input images are actually vectors rather than scalar values (*e.g.* for color images (p.225)) then the individual components (*e.g.* red, blue and green components (p.240)) are simply added separately to produce the output value.

If the image format being used only supports, say 8-bit integer pixel values (p.232), then it is very easy for the result of the addition to be greater than the maximum allowed pixel value. The effect of this depends upon the particular implementation. The overflowing pixel values might just be set to the maximum allowed value, an effect known as saturation (p.241). Alternatively the pixel values might wrap around from zero again. If the image format supports pixel values with a much larger range, *e.g.* 32-bit integers or floating point numbers, then this problem does not occur so much.

Guidelines for Use

Image addition crops up most commonly as a sub-step in some more complicated process rather than as a useful operator in its own right. As an example we show how addition can be used to overlay the output from an edge detector (p.230) on top of the original image after suitable masking (p.235) has been carried out.

The image `wdg2` shows a simple flat dark object against a light background. Applying the Canny edge detector (p.192) to this image, we obtain `wdg2can1`. Suppose that our task is to overlay this edge data on top of the original image. The image `wdg2add2` is the result of straightforwardly adding the two images. Since the sum of the edge pixels and the underlying values in the original is greater than the maximum possible pixel value, these pixels are (in this implementation) wrapped around. Therefore these pixels have a rather low pixel value and it is hard to distinguish them from the surrounding pixels. In order to avoid the pixel overflow we need to *replace* pixels in the original image with the corresponding edge data pixels, at every place where the edge data pixels are non-zero. The way to do this is to mask off a region of the original image before we do any addition.

The mask is made by thresholding (p.69) the edge data at a pixel value of 128 in order to produce `wdg2thr1`. This mask is then inverted (p.63) and subsequently ANDed (p.55) with the original image to produce `wdg2and1`.

Finally, the masked image is added to the unthresholded edge data to produce `wdg2add1`. This image now clearly shows that the Canny edge detector has done an extremely good job of localizing

the edges of the original object accurately. It also shows how the response of the edge detector drops off at the fuzzier left hand edge of the object.

Other uses of addition include adding a constant offset to all pixels in an image so as to brighten that image. For example, adding a constant value of *50* to `egg1` yields `egg1add1`. It is important to realize that if the input images are already quite bright, then straight addition may produce a pixel value overflow. Image `egg1add2` shows the results of adding *100* to the above image. Most of the background pixels are greater than the possible maximum (*255*) and therefore are (with this implementation of addition) wrapped (p.241) around from zero. If we implement the operator in such a way that pixel values exceeding the maximum value are set to *255* (*i.e.* using a hard limit) we obtain `egg1add4`. This image looks more natural than the wrapped around one. However, due to the saturation (p.241), we lose a certain amount of information, since all the values exceeding the maximum value are set to the same graylevel.

In this case, the pixel values should be scaled down (p.48) before addition. The image `egg1add3` is the result of scaling the original with *0.8* and adding a constant value of *100*. Although the image is brighter than the original, it has lost contrast due to the scaling. In most cases, scaling (p.48) the image with a factor larger than *1* without using addition at all provides a better way to brighten an image, as it increases the image contrast. For comparison, `egg1sca1` is the original image multiplied with *1.3*.

Blending (p.53) provides a slightly more sophisticated way of merging two images which ensures that saturation cannot happen.

When adding color images it is important to consider how the color information has been encoded. The section on 8-bit color images (p.226) describes the issues to be aware of when adding such images.

Exercises

1. Add the above Canny edge image (p.192) to its original, using different implementation's of *pixel addition* which handle the pixel overflow in different ways. Which one yields the best results for this implementation?
2. Use skeletonization (p.145) to produce a skeleton of `art7`. Add the skeleton to the original. Which problems do you face and how might they be solved?
3. Add a constant value of *255* to `eir1`. Use two different implementations, one wrapping around from zero all pixel values exceeding the maximum value and one using a hard limit of *255*. Comment on the results.

References

- A. Marion** *An Introduction to Image Processing*, Chapman and Hall, 1991, pp 242 - 244.
D. Vernon *Machine Vision*, Prentice-Hall, 1991, pp 51 - 52.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

5.2 Pixel Subtraction

The pixel subtraction operator takes two images as input and produces as output a third image whose pixel values (p.239) are simply those of the first image minus the corresponding pixel values from the second image. It is also often possible to just use a single image as input and subtract a constant value from all the pixels. Some versions of the operator will just output the absolute difference between pixel values, rather than the straightforward signed output.

How It Works

The subtraction of two images is performed straightforwardly in a single pass. The output pixel values are given by:

$$Q(i, j) = P_1(i, j) - P_2(i, j)$$

Or if the operator computes absolute differences between the two input images then:

$$Q(i, j) = |P_1(i, j) - P_2(i, j)|$$

Or if it is simply desired to subtract a constant value C from a single image then:

$$Q(i, j) = P_1(i, j) - C$$

If the pixel values in the input images are actually vectors rather than scalar values (*e.g.* for color images (p.225)) then the individual components (*e.g.* red, blue and green components (p.240)) are simply subtracted separately to produce the output value.

Implementations of the operator vary as to what they do if the output pixel values are negative. Some work with image formats that support negatively-valued pixels, in which case the negative values are fine (and the way in which they are displayed will be determined by the display colormap (p.235)). If the image format does not support negative numbers then often such pixels are just set to zero (*i.e.* black typically). Alternatively, the operator may ‘wrap’ (p.241) negative values, so that for instance -30 appears in the output as 226 (assuming 8-bit pixel values (p.232)).

If the operator calculates absolute differences and the two input images use the same pixel value type, then it is impossible for the output pixel values to be outside the range that may be represented by the input pixel type and so this problem does not arise. This is one good reason for using absolute differences.

Guidelines for Use

Image subtraction is used both as a sub-step in complicated image processing sequences, and also as an important operator in its own right.

A common use is to subtract background variations in illumination from a scene so that the foreground objects in it may be more easily analyzed. For instance, `son1` shows some text which has been badly illuminated during capture so that there is a strong illumination gradient across the image. If we wish to separate out the foreground text from the background page, then the obvious method for black on white text is simply to threshold (p.69) the image on the basis of intensity. However, simple thresholding fails here due to the illumination gradient. A typical failed attempt looks like `son1thr1`.

Now it may be that we cannot adjust the illumination, but we can put different things in the scene. This is often the case with microscope imaging, for instance. So we replace the text with a sheet of white paper and without changing anything else we capture a new image, as shown in `son2`. This image is the *lightfield*. Now we can subtract the lightfield image from the original image to attempt to eliminate variation in the background intensity. Before doing that an offset of 100 is added (p.43) to the first image to in order avoid getting negative numbers and we also use 32-bit

integer pixel values (p.239) to avoid overflow problems. The result of the subtraction is shown in `son1sub1`. Note that the background intensity of the image is much more uniform than before, although the contrast in the lower part of the image is still poor. Straightforward thresholding can now achieve better results than before, as shown in `son1thr3`, which is the result of thresholding at a pixel value of 80. Note that the results are still not ideal, since in the poorly lit areas of the image the contrast (*i.e.* difference between foreground and background intensity) is much lower than in the brightly lit areas, making a suitable threshold difficult or impossible to find. Compare these results with the example described under pixel division (p.50).

Absolute image differencing is also used for change detection. If the absolute difference between two frames of a sequence of images is formed, and there is nothing moving in the scene, then the output will mostly consist of zero value pixels. If however, there is movement going on, then pixels in regions of the image where the intensity changes spatially, will exhibit significant absolute differences between the two frames.

As an example of such change detection, consider `scr1`, which shows an image of a collection of screws and bolts. The image `scr2` shows a similar scene with one or two differences. If we calculate the absolute difference between the frames as shown in `scr1sub1`, then the regions that have changed become clear. The last image here has been contrast-stretched (p.75) in order to improve clarity.

Subtraction can also be used to estimate the temporal derivative of intensity at each point in a sequence of images. Such information can be used, for instance, in optical flow calculations.

Simple subtraction of a constant from an image can be used to darken an image, although scaling (p.48) is normally a better way of doing this.

It is important to think about whether negative output pixel values can occur as a result of the subtraction, and how the software will treat pixels that do have negative values. An example of what may happen can be seen in `son1sub2`, which is the above *lightfield* directly subtracted from the text images. In the implementation of *pixel subtraction* which was used, negative values are wrapped around (p.241) starting from the maximum value. Since we don't have exactly the same reflectance of the paper when taking the images of the lightfield and the text, the difference of pixels belonging to background is either slightly above or slightly below zero. Therefore the wrapping results in background pixels with either very small or very high values, thus making the image unsuitable for further processing (for example, thresholding (p.69)). If we alternatively set all negative values to zero, the image would become completely black, because subtracting the pixels in the lightfield from the pixels representing characters in the text image yields negative results, as well.

In this application, a suitable way to deal with negative values is to use absolute differences, as can be seen in `son1sub3` or as a gamma corrected (p.85) version in `son1sub4`. Thresholding this image yields similar good results as the earlier example.

If negative values are to be avoided then it may be possible to first add (p.43) an offset to the first input image. It is also often useful if possible to convert the pixel value type to something with a sufficiently large range to avoid overflow, *e.g.* 32-bit integers or floating point numbers.

Exercises

1. Take images of your watch at two different times, without moving it in between, and use subtraction to highlight the difference in the display.
2. Use `art4` to investigate the following method for edge detection (p.230). First apply erosion (p.123) to the image and then subtract the result from the original. What is the difference in the edge image if you use dilation (p.118) instead of erosion? What effects have size and form of the structuring element (p.241) on the result. How does the technique perform on grayscale images (p.232)?

References

- A. Jain** *Fundamentals of Digital Image Processing*, Prentice Hall, 1989, pp 240 - 241.
- R. Gonzales and R. Woods** *Digital Image Processing*, Addison wesley, 1992, pp 47 - 51, 185 - 187.
- R. Boyle and R. Thomas** *Computer Vision: A First Course*, Blackwell Scientific Publications, 1988, p 35.
- A. Marion** *An Introduction to Image Processing*, Chapman and Hall, 1991, pp 238 - 241.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, pp 52 - 53.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

5.3 Pixel Multiplication and Scaling

Brief Description

Like other image arithmetic operators, multiplication comes in two main forms. The first form takes two input images and produces an output image in which the pixel values (p.239) are just those of the first image, multiplied by the values of the corresponding values in the second image. The second form takes a single input image and produces output in which each pixel value is multiplied by a specified constant. This latter form is probably the more widely used and is generally called *scaling*.

This *graylevel* scaling should not be confused with geometric scaling (p.90).

How It Works

The multiplication of two images is performed in the obvious way in a single pass using the formula:

$$Q(i, j) = P_1(i, j) \times P_2(i, j)$$

Scaling by a constant is performed using:

$$Q(i, j) = P_1(i, j) \times C$$

Note that the constant is often a floating point number, and may be less than one, which will reduce the image intensities. It may even be negative if the image format supports that.

If the pixel values are actually vectors rather than scalar values (*e.g.* for color images (p.225)) then the individual components (*e.g.* `ref{rgb}`{red, blue and green components}) are simply multiplied separately to produce the output value.

If the output values are calculated to be larger than the maximum allowed pixel value, then they may either be truncated at that maximum value, or they can ‘wrap around’ (p.241) and continue upwards from the minimum allowed number again.

Guidelines for Use

There are many specialist uses for scaling. In general though, given a scaling factor greater than one, scaling will brighten an image. Given a factor less than one, it will darken the image. Scaling generally produces a much more natural brightening/darkening effect than simply adding (p.43) an offset to the pixels, since it preserves the relative contrast of the image better. For instance, `pum1dim1` shows a picture of model robot that was taken under low lighting conditions. Simply scaling every pixel by a factor of 3, we obtain `pum1mul1`, which is much clearer. However, when using pixel multiplication, we should make sure that the calculated pixel values don’t exceed the maximum possible value. If we, for example, scale the above image by a factor of 5 using a 8-bit representation (p.232), we obtain `pum1mul2`. All the pixels which, in the original image, have a value greater than 51 exceed the maximum value and are (in this implementation) wrapped around (p.241) from 255 back to 0.

The last example shows that it is important to be aware of what will happen if the multiplications result in pixel values outside the range that can be represented by the image format being used. It is also very easy to generate very large numbers with pixel-by-pixel multiplication. If the image processing software supports it, it is often safest to change to an image format with a large range, *e.g.* floating point, before attempting this sort of calculation.

Scaling is also often useful prior to other image arithmetic in order to prevent pixel values going out of range, or to prevent integer quantization ruining the results (as in integer image division (p.50)).

Pixel-by-pixel multiplication is generally less useful, although sometimes a binary image (p.225) can be used to multiply another image in order to act as a mask (p.235). The idea is to multiply

by 1 those pixels that are to be preserved, and multiply by zero those that are not. However for integer format images it is often easier and faster to use the logical operator AND (p.55) instead.

Another use for pixel by pixel multiplication is to filter images in the frequency domain. We illustrate the idea using the example of `hse1`. First, we obtain `hse1fou1` by applying the Fourier transform (p.209) to the original image, and then we use pixel multiplication to attenuate certain frequencies in the Fourier domain. In this example we use a simple lowpass filter which (as a scaled version) can be seen in `hse1msk3`. The result of the multiplication is shown in `hse1fou2`. Finally, an inverse Fourier transform is performed to return to the spatial domain. The final result `hse1fil1` shows the smoothing effect of a lowpass filter. More details and examples are given in the worksheets dealing with frequency filtering (p.167).

Exercises

1. Overlay `tol1` and its skeleton (p.145) `tol1sk11` using pixel addition (p.43) (the skeleton was derived from `tol1thr1`, which was produced by thresholding the input image at 110). Use image multiplication (p.48) to scale the images prior to the addition in order to avoid the pixel values being out of range. What effect does this have on the contrast of the input images.
2. Use thresholding (p.69) to segment the simple image `wdg4` into foreground and background. Use scaling to set the foreground pixel value to 2, and the background pixel value to 0. Then use pixel-by-pixel multiplication to multiply this image with the original image. What has this process achieved and why might it be useful?

References

- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 2.
A. Marion *An Introduction to Image Processing*, Chapman and Hall, 1991, p 244.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

5.4 Pixel Division

Brief Description

The image division operator normally takes two images as input and produces a third whose pixel values are just the pixel values (p.239) of the first image divided by the corresponding pixel values of the second image. Many implementations can also be used with just a single input image, in which case every pixel value in that image is divided by a specified constant.

How It Works

The division of two images is performed in the obvious way in a single pass using the formula:

$$Q(i, j) = P_1(i, j) \div P_2(i, j)$$

Division by a constant is performed using:

$$Q(i, j) = P_1(i, j) \div C$$

If the pixel values are actually vectors rather than scalar values (*e.g.* for color images (p.225)) than the individual components (*e.g.* red, blue and green components (p.240)) are simply divided separately to produce the output value.

The division operator may only implement integer division, or it may also be able to handle floating point division. If only integer division is performed, then results are typically rounded down to the next lowest integer for output. The ability to use images with pixel value types other than simply 8-bit integers (p.232) comes in very handy when doing division.

Guidelines for Use

One of the most important uses of division is in change detection, in a similar way to the use of subtraction (p.45) for the same thing. Instead of giving the absolute change for each pixel from one frame to the next, however, division gives the fractional change or ratio between corresponding pixel values (hence the common alternative name of *ratioing*). The images `scr1` and `scr2` are of the same scene except two objects have been slightly moved between the exposures. Dividing the former by the latter using a floating point pixel type and then contrast stretching (p.75) the resulting image yields `scr1div1`. After the division, pixels which didn't change between the exposures have a value of 1, whereas if the pixel value increased after the first exposure the result of the division is clustered between 0 and 1, otherwise it is between 1 and 255 (provided the pixel value in the second image is not smaller than 1). That is the reason why we can only see the new position of the moved part in the contrast-stretched image. The old position can be visualized by histogram equalizing (p.78) the division output, as shown in `scr1div2`. Here, high values correspond to the new position, low values correspond to the old position, assuming that the intensity of the moved object is lower than the background intensity. Intermediate graylevels in the equalized image correspond to areas of no change. Due to noise, the image also shows the position of objects which were not moved.

For comparison, the absolute difference (p.45) between the two images, as shown in `scr1sub1`, produces approximately the same pixel values at the old and the new position of a moved part.

Another application for pixel division is to separate the actual reflectance of an object from the unwanted influence of illumination. This image `son1` shows a poorly illuminated piece of text. There is a strong illumination gradient across the image which makes conventional foreground/background segmentation using standard thresholding (p.69) impossible. The image `son1thr1` shows the result of straightforward intensity thresholding at a pixel value of 128. There is no global threshold value that works over the whole of the image.

Suppose that we cannot change the lighting conditions, but that we can take several images with different items in the viewfield. This situation arises quite a lot in microscopy, for instance. We choose to take a picture of a blank sheet of white paper which should allow us to capture the incident illumination variation. This *lightfield* image is shown in `son2`.

Now, assuming that we are dealing with a flat scene here, with points on the surface of the scene described by coordinates x and y , then the reflected light intensity $B(x,y)$ depends upon the reflectance $R(x,y)$ of the scene at that point and also on the incident illumination $I(x,y)$ such that:

$$B(x,y) \propto I(x,y) \times R(x,y)$$

Using subscripts to distinguish the blank (lightfield) image and the original image, we can write:

$$\frac{B_{orig}(x,y)}{B_{blank}(x,y)} \propto \frac{I_{orig}(x,y) \times R_{orig}(x,y)}{I_{blank}(x,y) \times R_{blank}(x,y)}$$

But since $I(x,y)$ is the same for both images, and assuming the reflectance of the blank paper to be uniform over its surface, then:

$$\frac{B_{orig}(x,y)}{B_{blank}(x,y)} \propto R_{orig}(x,y)$$

Therefore the division should allow us to segment the letters out nicely. In image `son1div1` we see the result of dividing the original image by the lightfield image. Note that floating point format images were used in the division, which were then normalized (p.75) to 8-bit integers (p.232) for display. Virtually all the illumination gradient has been removed. The image `son1thr2` shows the result of thresholding (p.69) this image at a pixel value of 160. While not fantastic, with a little work using morphological operations (p.236), the text could become quite legible. Compare the result with that obtained using subtraction (p.45).

As with other image arithmetic operations (p.42), it is important to be aware of whether the implementation being used does integer or floating point arithmetic. Dividing two similar images, as done in the above examples, results mostly in very small pixel values, seldom greater than 4 or 5. To display the result, the image has to be normalized to 8-bit integers. However, if the division is performed in an integer format the result is quantized before the normalization, hence a lot of information is lost. Image `scr1div3` shows the result of the above change detection if the division is performed in integer format. The maximum result of the division was less than 3, therefore the integer image contains only three different values, *i.e.* 0, 1 and 2 before the normalization. One solution is to multiply the first image (the numerator image) by a scaling factor (p.48) before performing the division. Of course this is not generally possible with 8-bit integer images since significant scaling will simply saturate all the pixels in the image. The best method is, as was done in the above examples, to switch to a non-byte image type, and preferably to a floating point format. The effect is that the image is not quantized until the normalization and therefore the result does contain more graylevels. If floating point cannot be used, then use, say, 32-bit integers, and scale up the numerator image before dividing.

Exercises

1. Take two images of an (analogue) watch at different times, without moving it in between. Use division to detect the change in the image and compare the result with the one achieved with pixel subtraction (p.45). How easily can one detect small motions (*i.e.* minutes hand vs seconds hand)?
2. Describe a possible application where determining the *percentage* change between two images of similar scenes using ratioing is a better idea than determining the difference between the images using subtraction.

References

- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap 2.
A. Marion *An Introduction to Image Processing*, Chapman and Hall, 1991, p 244.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

5.5 Blending

Brief Description

This operator forms a blend of two input images of the same size. Similar to pixel addition (p.43), the value of each pixel in the output image is a linear combination of the corresponding pixel values (p.239) in the input images. The coefficients of the linear combination are user-specified and they define the ratio by which to scale (p.48) each image before combining them. These proportions are applied such that the output pixel values do not exceed the maximum pixel value.

How It Works

The resulting image is calculated using the formula

$$Q(i, j) = X \times P_1(i, j) + (1 - X) \times P_2(i, j)$$

P_1 and P_2 are the two input images. In some applications P_2 can also be a constant, thus allowing a constant offset value to be added to a single image.

X is the blending ratio which determines the influence of each input image in the output. X can either be a constant factor for all pixels in the image or can be determined for each pixel separately using a mask. The size of the mask must then be identical with the size of the images.

Some implementations only support graylevel images. If multi-spectral images (p.237) are supported the calculation is done for each band separately.

Guidelines for Use

Image blending is used for similar applications as image addition (p.43) with the difference that we don't have to worry whether the values of the output image exceed the allowed maximum. In most cases the operator is a part of some more complicated process. As an example we use image blending to overlay the output of an edge detector (p.230) on top of the original image, (compare with the results achieved with image addition).

The image `wdg2` shows a simple flat dark object against a light background. Applying the Canny edge detector (p.192) to this image we obtain `wdg2can1`. We get `wdg2b1d1` if we apply the blending operator with $X = 0.5$, where the original image is P_1 and the edge image is P_2 . The result clearly shows the disadvantage of image blending over image addition: since each of the input images is scaled with 0.5 before they are added up, the contrast of each image is halved. That is why it is hard to see the difference between the object and the background of the original image. If the contrast in one image is more important than the other, we can improve the result by choosing a blending ratio other than 0.5, thus keeping more of the contrast in the image where it is needed. To get `wdg2b1d2` the same two images as above were blended with $X=0.7$.

The bad result in the first example is mainly due to the low initial contrast in the input images. So, we will have a better result if the input images are of high contrast. To produce `wdg2str1`, the input images were contrast-enhanced with contrast stretching (p.75) and then blended with $X = 0.5$. Although this already yields a better result, we still lose some contrast with respect to the original input images.

To maintain the full contrast in the output image we can define a special mask. The mask is made by thresholding (p.69) the edge image at a pixel value of 128 and setting the non-zero values to one. Now, we blend the graylevel edge image (now corresponding to P_1) and the original image using the thresholded image as a blending mask $X(i,j)$. The image `wdg2b1d3` shows the result, which is identical to the one achieved with image addition (p.43), but now achieved via a slightly simpler process.

Blending can also be used to achieve nice effects in photographs. We obtained `moo1b1d1` by blending `moo1` with the resized version of `fce6` using the $X=0.5$.

Exercises

1. Examine the effects of using blending ratios other than 0.5 when blending `moo1` and `fce6`.
2. Take an image and add a constant value (e.g. 100) using image blending and image addition. Comment on the differences of the results.
3. Produce a skeleton from `art6` using skeletonization (p.145). Assess the result by combining the two images using the blending operator.

References

E. Davies *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 2.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

5.6 Logical AND/NAND

Brief Description

AND and NAND are examples of logical operators (p.234) having the truth-tables shown in Figure 5.1.

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

NAND

Figure 5.1: Truth-tables for AND and NAND.

As can be seen, the output values of NAND are simply the inverse of the corresponding output values of AND.

The AND (and similarly the NAND) operator typically takes two binary (p.225) or integer graylevel images (p.232) as input, and outputs a third image whose pixel values are just those of the first image, ANDed with the corresponding pixels from the second. A variation of this operator takes just a single input image and ANDs each pixel with a specified constant value in order to produce the output.

How It Works

The operation is performed straightforwardly in a single pass. It is important that all the input pixel values being operated on have the same number of bits in them or unexpected things may happen. Where the pixel values (p.239) in the input images are not simple 1-bit numbers, the AND operation is normally (but not always) carried out individually on each corresponding bit in the pixel values, in bitwise fashion (p.235).

Guidelines for Use

The most obvious application of AND is to compute the intersection of two images. We illustrate this with an example where we want to detect those objects in a scene which did not move between two images, *i.e.* which are at the same pixel positions in the first *and* the second image. We illustrate this example using `scr3` and `scr4`. If we simply AND the two graylevel images in a bitwise fashion we obtain `scr3and1`. Although we wanted the moved object to disappear from the resulting image, it appears twice, at its old and at its new position. The reason is that the object has rather low pixel values (similar to a logical 0) whereas the background has a high values (similar to a logical 1). However, we normally associate an object with logical 1 and the background with logical 0, therefore we actually ANDed the negatives of two images, which is equivalent to NOR (p.58) them. To obtain the desired result we have to invert (p.63) the images before ANDing them, as it was done in `scr3and2`. Now, only the object which has the same position in both images is highlighted. However, ANDing two graylevel images might still cause problems, as it is

not guaranteed that ANDing two high pixel values in a bitwise fashion yields a high output value (for example, 128 AND 127 yields 0). To avoid these problems, it is best to produce a binary versions from the grayscale images using thresholding (p.69). `scr3thr1` and `scr4thr1` are the thresholded versions of the above images and `scr3and3` is the result of ANDing their negatives.

Although ANDing worked well for the above example, it runs into problems in a scene like `pap1`. Here, we have two objects with the average intensity of one being higher than the background and the other being lower. Hence, we can't produce a binary image containing both objects using simple thresholding. As can be seen in the following images, ANDing the grayscale images is not successful either. If in the second scene the light part was moved, as in `pap2`, then the result of ANDing the two images is `pap1and1`. It shows the desired effect of attenuating the moved object. However, if the second scene is somehow like `pap3`, where the dark object was moved, we obtain `pap1and2`. Here, the old and the new positions of the dark object are visible.

In general, applying the AND operator (or other logical operators) to two images in order to detect differences or similarities between them is most appropriate if they are binary or can be converted into binary format using thresholding.

As with other logical operators, AND and NAND are often used as sub-components of more complex image processing tasks. One of the common uses for AND is for masking (p.235). For example, suppose we wish to selectively brighten a small region of `car1` to highlight a particular car. There are many ways of doing this and we illustrate just one. First a paint program (p.233) is used to identify the region to be highlighted. In this case we set the region to black as shown in `car1msk1`. This image can then be thresholded (p.69) to just select the black region, producing the mask shown in `car1thr1`. The mask image has a pixel value of 255 (11111111 binary) in the region that we are interested in, and zero pixels (00000000 binary) elsewhere. This mask is then bitwise ANDed with the original image to just select out the region that will be highlighted. This produces `car1and1`. Finally, we brighten this image by scaling (p.48) it by a factor of 1.1, dim the original image using a scale factor of 0.8, and then add (p.43) the two images together to produce `car1add1`.

AND can also be used to perform so called *bit-slicing* on an 8-bit image. To determine the influence of one particular bit on an image, it is ANDed in a bitwise fashion with a constant number, where the relevant bit is set to 1 and the remaining 7 bits are set to 0. For example, to obtain the bit-plane 8 (corresponding to the most significant bit) of `ape1`, we AND the image with 128 (10000000 binary) and threshold (p.69) the output at a pixel value of 1. The result, shown in `ape1and8`, is equivalent to thresholding the image at a value of 128. Images `ape1and7`, `ape1and6` and `ape1and4` correspond to bit-planes 7, 6 and 4. The images show that most image information is contained in the higher (more significant) bits, whereas the less significant bits contain some of the finer details and noise. The image `ape1and1` shows bit-plane 1.

Exercises

1. NAND `cir2` and `cir3`. Compare the result with the result of ANDing the negatives (p.63) of the two input images.
2. AND `scr3thr1` and `scr4thr1` as well as the negatives (p.63) of `pap1` and `pap2`. Compare the results with the ones obtained in the previous section.
3. Extract all 8 bit planes from `pen1` and `str1`. Comment on the number of visually significant bits in each image.
4. What would be the effect of ANDing an 8-bit graylevel image (p.232) with a constant value of 240 (11110000 in binary)? Why might you want to do this?
5. What would be the effect of ANDing an 8-bit graylevel image with a constant value of 15 (00001111 in binary)? Why might you want to do this? Try this out on `ball` and comment on what you see.

References

E. Davies *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 2.

R. Gonzalez and R. Woods *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 47 - 51, 171 - 172.

A. Jain *Fundamentals of Digital Processing*, Prentice Hall, 1989, pp 239 - 240.

B. Horn *Robot Vision*, MIT Press, 1986, pp 47 - 48.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

5.7 Logical OR/NOR

Brief Description

OR and NOR are examples of logical operators (p.234) having the truth-tables shown in Figure 5.2.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

OR

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

NOR

Figure 5.2: Truth-tables for OR and NOR.

As can be seen, the output values of NOR are simply the inverses of the corresponding output values of OR.

The OR (and similarly the NOR) operator typically takes two binary (p.225) or graylevel (p.232) images as input, and outputs a third image whose pixel values are just those of the first image, ORed with the corresponding pixels from the second. A variation of this operator takes just a single input image and ORs each pixel with a specified constant value in order to produce the output.

How It Works

The operation is performed straightforwardly in a single pass. It is important that all the input pixel values being operated on have the same number of bits in them or unexpected things may happen. Where the pixel values (p.239) in the input images are not simple 1-bit numbers, the OR operation is normally (but not always) carried out individually on each corresponding bit in the pixel values, in bitwise fashion (p.235).

Guidelines for Use

We can illustrate the function of the OR operator using `scr3` and `scr4`. The images show a scene with two objects, one of which was moved between the exposures. We can use OR to compute the *union* of the images, *i.e.* highlighting all pixels which represent an object either in the first *or* in the second image. First, we threshold (p.69) the images, since the process is simplified by use binary input. If we OR the resulting images `scr3thr1` and `scr4thr1` we obtain `scr3or2`. This image shows only the position of the object which was at the same location in both input images. The reason is that the objects are represented with logically 0 and the background is logically 1. Hence, we actually OR the background which is equivalent to NANDing the objects. To get the desired result, we first have to invert (p.63) the input images before ORing them. Then, we obtain `scr3or1`. Now, the output shows the position of the stationary object as well as that of the moved object.

As with other logical operators, OR and NOR are often used as sub-components of more complex image processing tasks. OR is often used to merge two images together. Suppose we want to

overlay `wdg2` with its histogram (p.105), shown in `wdg2hst1`. First, an image editor (p.233) is used to enlarge the histogram image until it is the same size as the grayscale image as shown in `wdg2hst2`. Then, simply ORing the two gives `wdg2or1`. The performance in this example is quite good, because the images contain very distinct graylevels. If we proceed in the same way with `bld1` we obtain `bld1or1`. Now, it is difficult to see the characters of the histogram (which have high pixel values) at places where the original image has high values, as well. Compare the result with that described under XOR (p.60).

Note that there is no problem of overflowing pixel values with the OR operator, as there is with the addition operator (p.43).

ORing is usually safest when at least one of the images is binary, *i.e.* the pixel values are 0000... and 1111... only. The problem with ORing other combinations of integers is that the output result can fluctuate wildly with a small change in input values. For instance 127 ORed with 128 gives 255, whereas 127 ORed with 126 gives 127.

Exercises

1. NOR `cir2` and `cir3` and AND their negatives (p.63). Compare the results.
2. Why can't you use thresholding (p.69) to produce a binary image containing both objects of `pap2` and `pap3`? Use graylevel ORing to combine the two images. Can you detect all the locations of the objects in the two images? What changes if you invert (p.63) the images before combining them.
3. In the example above, how could you make the histogram appear in black instead of white? Try it.
4. Summarize the conditions under which you would use OR to combine two images rather than, say, addition (p.43) or blending (p.53).

References

R. Gonzalez and R. Woods *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 47 - 51, 171 - 172.

E. Davies *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 2.

B. Horn *Robot Vision*, MIT Press, 1986, pp 47 - 48.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

5.8 Logical XOR/XNOR

Brief Description

XOR and XNOR are examples of logical operators (p.234) having the truth-tables shown in Figure 5.3.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

XOR

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

XNOR

Figure 5.3: Truth-tables for XOR and XNOR.

The XOR function is only true if just one (and only one) of the input values is true, and false otherwise. XOR stands for *eXclusive OR*. As can be seen, the output values of XNOR are simply the inverse of the corresponding output values of XOR.

The XOR (and similarly the XNOR) operator typically takes two binary (p.225) or graylevel images (p.232) as input, and outputs a third image whose pixel values (p.239) are just those of the first image, XORed with the corresponding pixels from the second. A variation of this operator takes a single input image and XORs each pixel with a specified constant value in order to produce the output.

How It Works

The operation is performed straightforwardly in a single pass. It is important that all the input pixel values being operated on have the same number of bits in them, or unexpected things may happen. Where the pixel values in the input images are not simple 1-bit numbers, the XOR operation is normally (but not always) carried out individually on each corresponding bit in the pixel values, in bitwise fashion (p.235).

Guidelines for Use

We illustrate the function of XOR using `scr3` and `scr4`. Since logical operators work more reliably with binary input we first threshold (p.69) the two images, thus obtaining `scr3thr1` and `scr4thr1`. Now, we can use XOR to detect changes in the images, since pixels which didn't change output 0 and pixels which did change result in 1. The image `scr3xor1` shows the result of XORing the thresholded images. We can see the old and the new position of the moved object, whereas the stationary object almost disappeared from the image. Due to the effects of noise, we can still see some pixels around the boundary of the stationary object, *i.e.* pixels whose values in the original image were close to the threshold.

In a scene like `pap1`, it is not possible to apply a threshold in order to obtain a binary image, since one of the objects is lighter than the background whereas the other one is darker. However, we can combine two grayscale images by XORing them in a bitwise fashion. `pap3` shows a scene where the

dark object was moved and in `pap2` the light object changed its position. XORing each of them with the initial image yields `pap1xor1` and `pap1xor2`, respectively. In both cases, the moved part appears at the old as well as at the new location and the stationary object almost disappears. This technique is based on the assumption that XORing two similar grayvalues produces a low output, whereas two distinct inputs yield a high output. However, this is not always true, *e.g.* XORing 127 and 128 yields 255. These effects can be seen at the boundary of the stationary object, where the pixels have an intermediate graylevel and might, due to noise (p.221), differ slightly between two of the images. Hence, we can see a line with high values around the stationary object. A similar problem is that the output for the moved pen is much higher than the output for the moved piece of paper, although the contrast between their intensities and that of the background value is roughly the same. Because of these problems it is often better to use image subtraction (p.45) or image division (p.50) for change detection.

As with other logical operators, XOR and XNOR are often used as sub-components of more complex image processing tasks. XOR has the interesting property that if we XOR A with B to get Q , then the bits of Q are the same as A where the corresponding bit from B is zero, but they are of the opposite value where the corresponding bit from B is one. So for instance using binary notation, 1010 XORed with 1100 gives 0110. For this reason, B could be thought of as a *bit-reversal mask*. Since the operator is symmetric, we could just as well have treated A as the mask and B as the original.

Extending this idea to images, it is common to see an 8-bit XOR image mask (p.235) containing only the pixel values 0 (00000000 binary) and 255 (11111111 binary). When this is XORed pixel-by-pixel with an original image it reverses the bits of pixels values where the mask is 255, and leaves them as they are where the mask is zero. The pixels with reversed bits normally ‘stand out’ against their original color and so this technique is often used to produce a cursor that is visible against an arbitrary colored background. The other advantage of using XOR like this is that to undo the process (for instance when the cursor moves away), it is only necessary to repeat the XOR using the same mask and all the flipped pixels will become unflipped. Therefore it is not necessary to explicitly store the original colors of the pixels affected by the mask. Note that the flipped pixels are not always visible against their unflipped color — light pixels become dark pixels and dark pixels become light pixels, but middling gray pixels become middling gray pixels!

The image `wdg2` shows a simple graylevel image. Suppose that we wish to overlay this image with its histogram (p.105) shown in `wdg2hst1` so that the two can be compared easily. One way is to use XOR. We first use an image editor (p.233) to enlarge the histogram until it is the same size as the first image. The result is shown in `wdg2hst2`. To perform the overlay we simply XOR this image with the first image in bitwise fashion to produce `wdg2xor1`. Here, the text is quite easy to read, because the original image consists of large and rather light or rather dark areas. If we proceed in the same way with `bld1` we obtain `bld1xor1`. Note how the writing is dark against light backgrounds and light against dark backgrounds and hardly visible against gray backgrounds. Compare the result with that described under OR (p.58). In fact XORing is not particularly good for producing easy to read text on gray backgrounds — we might do better just to add a constant offset to the image pixels that we wish to highlight (assuming wraparound under addition overflow) — but it is often used to quickly produce highlighted pixels where the background is just black and white or where legibility is not too important.

Exercises

1. XOR `cir2` and `cir3`. Compare the result with the output of XORing their negatives (p.63). Do you see the same effect as for other logical operators?
2. Use the technique discussed above to produce a cursor on `fce1`. Place the cursor on different location of the image and examine the performance on a background with high, low, intermediate and mixed pixel values.

References

R. Gonzalez and R. Woods *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 47 - 51.

E. Davies *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 2.

B. Horn *Robot Vision*, MIT Press, 1986, pp 47 - 48.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

5.9 Invert/Logical NOT

Brief Description

Logical NOT or *invert* is an operator which takes a binary (p.225) or graylevel image (p.232) as input and produces its photographic negative, *i.e.* dark areas in the input image become light and light areas become dark.

How It Works

To produce the photographic negative of a binary image we can employ the logical NOT operator. Its truth-table is shown in Figure 5.4.

A	Q
0	1
1	0

NOT

Figure 5.4: Truth-table for logical NOT.

Each pixel in the input image having a logical 1 (often referred to as foreground) has a logical 0 (associated with the background in the output image and *vice versa*). Hence, applying logical NOT to a binary image changes its polarity (p.225).

The logical NOT can also be used for a graylevel image being stored in *byte* pixel format (p.239) by applying it in a bitwise (p.234) fashion. The resulting value for each pixel is the input value subtracted from 255:

$$Q(i, j) = 255 - P(i, j)$$

Some applications of *invert* also support *integer* or *float* pixel format. In this case, we can't use the logical NOT operator, therefore the pixel values of the inverted image are simply given by

$$Q(i, j) = -P(i, j)$$

If this output image is normalized (p.75) for an 8-bit display, we again obtain the photographic negative of the original input image.

Guidelines for Use

When processing a binary image with a logical (p.234) or morphological (p.117) operator, its polarity (p.225) is often important. Hence, the logical NOT operator is often used to change the polarity of a binary image as a part of some larger process. For example, if we OR (p.58) `cir2neg1` and `cir3neg1` the resulting image, `cir2or2`, shows the *union* of the background, because it is represented with a logical 1. However, if we OR `cir2` and `cir3` which are the inverted versions of the above image we obtain `cir2or1`. Now, the result contains the union of the two circles.

We illustrate another example of the importance of the polarity of a binary image using the dilation (p.118) operator. Dilation expands all white areas in a binary image. Hence, if we dilate `art4`, the object, being represented with a logical 1, grows and the holes in the object shrink. We obtain `art4di11`. If we dilate `art4neg1` which was obtained by applying logical NOT to the original image, we get `art4di12`. Here, the background is expanded and the object became smaller.

Invert can be used for the same purpose on grayscale images, if they are processed with a morphological or logical operator.

Invert is also used to print the photographic negative of an image or to make the features in an image appear clearer to a human observer. This can, for example, be useful for medical images, where the objects often appear in black on a white background. Inverting the image makes the objects appear in white on a dark background, which is often more suitable for the human eye. From the original image `ce17` of a tissue slice, we obtain the photographic negative `ce17neg1`.

Exercises

1. Apply the erode (p.123) operator to `art1` and `art3`. Which polarity of the image allows you to suppress the circles?
2. Compare the results of ORing (p.58) `scr3` and `scr4` and ORing their photographic negatives.
3. Take the photographic negative of `egg1`. Does it improve the visibility of the features in the image?

References

- A. Jain** *Fundamentals of Digital Image Processing*, Prentice Hall, 1989, p 238.
R. Gonzales and R. Woods *Digital Image Processing*, Addison Wesley, 1992, pp 47 - 51.
E. Davies *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 2.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

5.10 Bitshift Operators

Brief Description

The bitshift operator works on images represented in byte or integer pixel format (p.239), where each pixel value is stored as a *binary* number with a fixed amount of bits. Bitshifting shifts the binary representation of each pixel to the left or to the right by a pre-defined number of positions. Shifting a binary number by one bit is equivalent to multiplying (p.48) (when shifting to the left) or dividing (p.50) (when shifting to the right) the number by 2.

How It Works

The operation is performed straightforwardly in a single pass. If the binary representation of a number is shifted in one direction, we obtain an empty position on the opposite side. There are generally three possibilities of how to fill in this empty position: we can pad the empty bits with a 0 or a 1 or we can wrap around the bits which are shifted out of the binary representation of the number on the other side. The last possibility is equivalent to *rotating* the binary number.

The choice of technique used depends on the implementation of the operator and on the application. In most cases, bitshifting is used to implement a fast multiplication or division. In order to obtain the right results for this application, we have to pad the empty bits with a 0. Only in the case of dividing a negative number by a power of 2, do we need to fill the left bits with a 1, because a negative number is represented as the *two's-complement* of the positive number, *i.e.* the sign bit is a 1. The result of applying bitshifting in this way is illustrated in the following formula:

$$\begin{aligned} \text{Shifting } i \text{ bits to the right} &\Leftrightarrow Q(i, j) = P(i, j) \div 2^i \\ \text{Shifting } i \text{ bits to the left} &\Leftrightarrow Q(i, j) = P(i, j) \times 2^i \end{aligned}$$

An example is shown in Figure 5.5.

If bitshifting is used for multiplication, it might happen that the result exceeds the maximum possible pixel value. This is the case when a 1 is shifted out of the binary representation of the pixel value. This information is lost and the effect is that the value is wrapped around (p.241) from zero.

Guidelines for Use

The main application for the bitshift operator is to divide or multiply an image by a power of 2. The advantage over the normal pixel division (p.50) and pixel multiplication (p.48) operators is that bitshifting is computationally less expensive.

For example, if we want to add (p.43) two images we can use bitshifting to make sure that the result will not exceed the maximum pixel value. We illustrate this example using `tol1` and `tol1sk11` where the latter is the skeleton (p.145) gained from the thresholded (p.69) version of the former. To better visualize the result of the skeletonization we might want to overlay these two images. However, if we add them straightforwardly we obtain pixel values greater than the maximum value. First shifting both images to the right by one bit yields `tol1shi1` and `tol1sk12`, which then can be added without causing any overflow problems. The result can be seen in `tol1add1`. Here, we can see that shifting a pixel to the right does, as a normal pixel division, decrease the contrast in the image.

On the other hand, shifting the binary representation of a pixel to the left increases the image contrast, like the pixel multiplication. For example, `pum1dim1` is an image taken under poor lighting conditions. Shifting each pixel in the image to the left by one bit, which is identical to multiplying it with 2, yields `pum1shi1`. Although the operator worked well in this example, we have to be aware that the result of the multiplication might exceed the maximum pixel value. Then, the effect for the pixel value is that it is wrapped around (p.241) from 0. For example, if we shift each

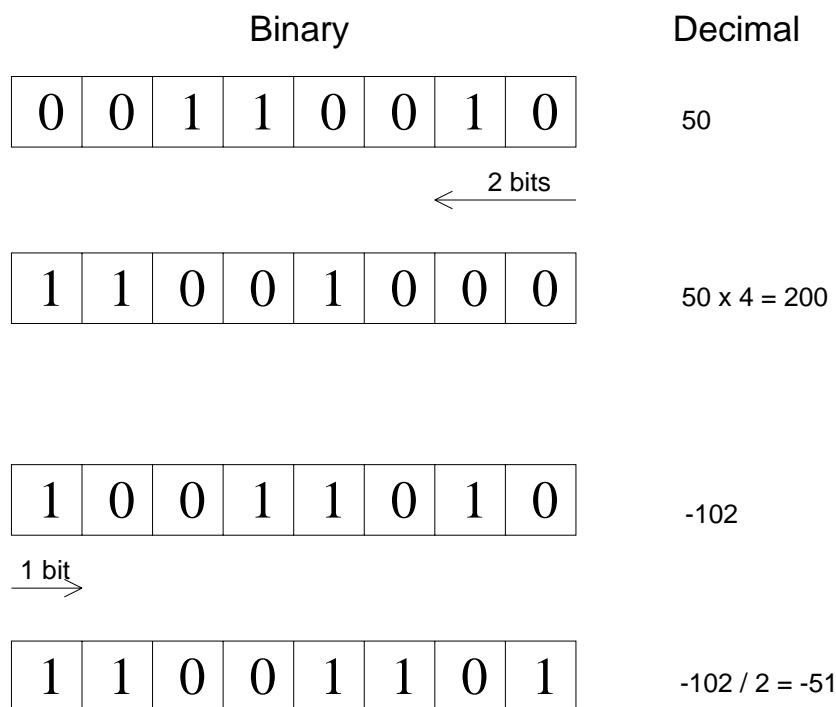


Figure 5.5: Examples for using bitshifting for multiplication and division. Note that the bottom example uses a signed-byte convention where a byte represents a number between -128 and $+127$

pixel in the above image by two bits, at some pixels a 1 is shifted out of the binary representation of the image, resulting in a loss of information. This can be seen in `pum1shi2`. In general, we should make sure that the values in the input image are sufficiently small or we have to be careful when we interpret the resulting image. Alternatively, we can change the pixel value (p.239) format prior to applying the bitshift operator, *e.g.* change from *byte* format to *integer* format.

Although multiplication and division are the main applications for bitshifting it might also be used for other, often very specialized, purposes. For example, we can store two 4 -bit images in a byte array if we shift one of the two images by 4 bits and mask out the unused bits. Using the logical OR operator (p.58) we can combine the two images into one without losing any information. Sometimes it might also be useful to rotate the binary representation of each bit, apply some other operator to the image and finally rotate the pixels back to the initial order.

Exercises

1. Use pixel addition (p.43) to overlay `wdg2` and its edge image `wdg2can1`. Apply the bitshift operator to the original image in order to increase its contrast. Convert the image into an integer format prior to the shifting to preserve all image information. Compare the result of the addition with the one you get without the bitshifting.
2. What is the result of dividing -7 (binary: `1001`) by 2 using bitshifting. What is the result of dividing $+7$ (binary: `0111`) by 2 ?

References

- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 2.
- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 50 - 51.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

Chapter 6

Point Operations

Single-point processing is a simple method of image enhancement. This technique determines a pixel value in the enhanced image dependent only on the value of the corresponding pixel in the input image. The process can be described with the *mapping function*

$$s = M(r)$$

where r and s are the pixel values in the input and output images, respectively. The form of the mapping function M determines the effect of the operation. It can be previously defined in an ad-hoc manner, as for thresholding (p.69) or gamma correction (p.87), or it can be computed from the input image, as for histogram equalization (p.78). For example, a simple mapping function is defined by the thresholding operator:

$$s = \begin{cases} 0 & \text{if } r < T \\ L - 1 & \text{if } r > T \end{cases}$$

The corresponding graph is shown in Figure 6.1.

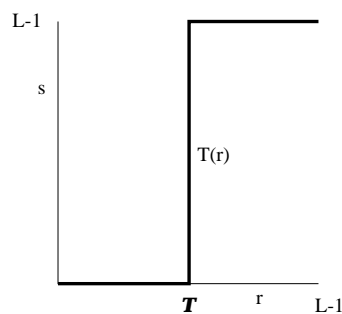


Figure 6.1: Graylevel transformation function for thresholding.

Point operators are also known as *LUT-transformations*, because the mapping function, in the case of a discrete image, can be implemented in a look-up table (LUT) (p.235).

A subgroup of the point processors is the set of anamorphosis operators. This notion describes all point operators with a strictly increasing or decreasing mapping function. Examples include the logarithm operator (p.82), exponential operator (p.85) and contrast stretching (p.75), to name just a few.

An operator where M changes over the image is adaptive thresholding (p.72). This is not a pure point operation anymore, because the mapping function, and therefore the output pixel value, depends on the local neighborhood of a pixel.

6.1 Thresholding

Brief Description

In many vision applications, it is useful to be able to separate out the regions of the image corresponding to objects in which we are interested, from the regions of the image that correspond to background. Thresholding often provides an easy and convenient way to perform this segmentation on the basis of the different intensities or colors in the foreground and background regions of an image.

In addition, it is often useful to be able to see what areas of an image consist of pixels whose values lie within a specified range, or *band* of intensities (or colors). Thresholding can be used for this as well.

How It Works

The input to a thresholding operation is typically a grayscale (p.232) or color image (p.225). In the simplest implementation, the output is a binary image (p.225) representing the segmentation. Black pixels correspond to background and white pixels correspond to foreground (or *vice versa*). In simple implementations, the segmentation is determined by a single parameter known as the *intensity threshold*. In a single pass, each pixel in the image is compared with this threshold. If the pixel's intensity (p.239) is higher than the threshold, the pixel is set to, say, white in the output. If it is less than the threshold, it is set to black.

In more sophisticated implementations, multiple thresholds can be specified, so that a *band* of intensity values can be set to white while everything else is set to black. For color (p.225) or multi-spectral images (p.237), it may be possible to set different thresholds for each color channel, and so select just those pixels within a specified cuboid in RGB space (p.240). Another common variant is to set to black all those pixels corresponding to background, but leave foreground pixels at their original color/intensity (as opposed to forcing them to white), so that that information is not lost.

Guidelines for Use

Not all images can be neatly segmented into foreground and background using simple thresholding. Whether or not an image can be correctly segmented this way can be determined by looking at an intensity histogram (p.105) of the image. We will consider just a grayscale histogram here, but the extension to color is trivial.

If it is possible to separate out the foreground of an image on the basis of pixel intensity, then the intensity of pixels within foreground objects must be distinctly different from the intensity of pixels within the background. In this case, we expect to see a distinct peak in the histogram (p.105) corresponding to foreground objects such that thresholds can be chosen to isolate this peak accordingly. If such a peak does not exist, then it is unlikely that simple thresholding will produce a good segmentation. In this case, adaptive thresholding (p.72) may be a better answer.

Figure 6.2 shows some typical histograms along with suitable choices of threshold.

The histogram for image `wdg2` is `wdg2hst1`. This shows a nice bi-modal distribution — the lower peak represents the object and the higher one represents the background. The picture can be segmented using a single threshold at a pixel intensity value of 120. The result is shown in `wdg2thr3`.

The histogram for image `wdg3` is `wdg3hst1`. Due to the severe illumination gradient across the scene, the peaks corresponding to foreground and background have run together and so simple thresholding does not give good results. Images `wdg3thr1` and `wdg3thr2` show the resulting bad segmentations for single threshold values of 80 and 120 respectively (reasonable results can be achieved by using adaptive thresholding (p.72) on this image).

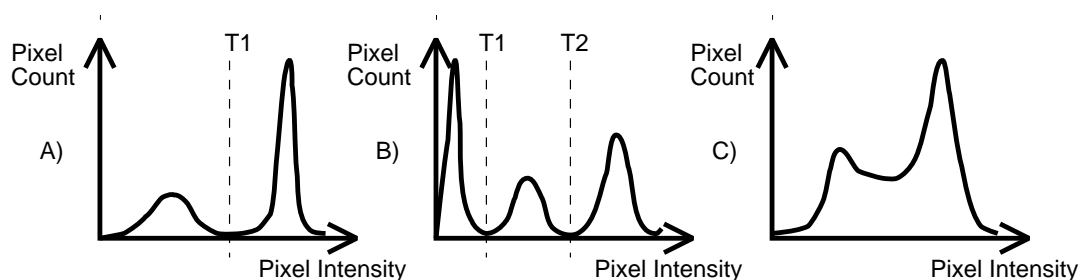


Figure 6.2: **A)** shows a classic bi-modal intensity distribution. This image can be successfully segmented using a single threshold $T1$. **B)** is slightly more complicated. Here we suppose the central peak represents the objects we are interested in and so threshold segmentation requires two thresholds: $T1$ and $T2$. In **C)**, the two peaks of a bi-modal distribution have run together and so it is almost certainly not possible to successfully segment this image using a single global threshold

Thresholding is also used to filter the output of or input to other operators. For instance, in the former case, an edge detector (p.230) like Sobel (p.188) will highlight regions of the image that have high spatial gradients. If we are only interested in gradients above a certain value (*i.e.* sharp edges), then thresholding can be used to just select the strongest edges and set everything else to black. As an example, `wdg2sob2` was obtained by first applying the Sobel operator to `wdg2` to produce `wdg2sob1` and then thresholding this using a threshold value of 60.

Thresholding can be used as preprocessing to extract an interesting subset of image structures which will then be passed along to another operator in an image processing chain. For example, image `ce14` shows a slice of brain tissue containing nervous cells (*i.e.* the large gray blobs, with darker circular nuclei in the middle) and *glia* cells (*i.e.* the isolated, small, black circles). We can threshold this image so as to map all pixel values between 0 and 150 in the original image to foreground (*i.e.* 255) values in the binary image, and leave the rest to go to background, as in `ce14thr1`. The resultant image can then be connected-components-labeled (p.114) in order to count the total number of cells in the original image, as in `ce14lab1`. If we wanted to know how many nerve cells there are in the original image, we might try applying a double threshold in order to select out just the pixels which correspond to nerve cells (and therefore have middle level grayscale intensities) in the original image. (In remote sensing and medical terminology, such thresholding is usually called *density slicing*.) Applying a threshold band of 130 - 150 yields `ce14thr2`. While most of the foreground of the resulting image corresponds to nerve cells, the foreground features are so disconnected (because nerve cell nuclei map to background intensity values along with the glia cells) that we cannot apply connected components labeling. Alternatively, we might obtain a better assessment of the number of nerve cells by investigating some attributes (*e.g.* size, as measured by a distance transform (p.206)) of the binary image containing both whole nerve cells and glia. In reality, sophisticated modeling and/or pattern matching is required to segment such an image.

Exercises

1. How would you set up the lighting for a simple scene containing just flat metal parts viewed from above so as to ensure the best possible segmentation using simple thresholding?
2. In medical imagery of certain mouse nervous tissue, healthy cells assume a medium graylevel intensity, while dead cells become dense and black. The images `c1a3`, `c1b3` and `c1c3` were each taken on a different day during an experiment which sought to quantify cell death. Investigate the intensity histogram (p.105) of these images and choose a threshold which allows you to segment out the dead cells. Then use connected components labeling (p.114) to count the number of dead cells on each day of the experiment.
3. Thresholding is often used in applications such as remote sensing where it is desirable to select

out, from an image, those regions whose pixels lie within a specified range of pixel values. For instance, it might be known that wheat fields give rise to a particular range of intensities (in some spectral band) that is fairly unusual elsewhere. In the multi-spectral image `aer1`, assume that wheat fields are visible as yellow patches. Construct a set of thresholds for each color channel which allow you to segment out the wheat fields (note, you may need to reset your display).

4. How should the intensity threshold be chosen so that a small change in this threshold value causes as little change as possible to the resulting segmentation? Think about what the intensity histogram must look like at the threshold value.
5. Discuss whether you expect thresholding to be of much use in segmenting natural scenes.

References

- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 4.
- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, Chap. 7.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, pp 49 - 51, 86 - 89.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

6.2 Adaptive Thresholding

Brief Description

Thresholding is used to segment an image by setting all pixels whose intensity values are above a threshold to a foreground value and all the remaining pixels to a background value.

Whereas the conventional thresholding (p.69) operator uses a global threshold for all pixels, adaptive thresholding changes the threshold dynamically over the image. This more sophisticated version of thresholding can accommodate changing lighting conditions in the image, *e.g.* those occurring as a result of a strong illumination gradient or shadows.

How It Works

Adaptive thresholding typically takes a grayscale (p.232) or color (p.225) image as input and, in the simplest implementation, outputs a binary image (p.225) representing the segmentation. For each pixel in the image, a threshold has to be calculated. If the pixel value is below the threshold it is set to the background value, otherwise it assumes the foreground value.

There are two main approaches to finding the threshold: (i) the *Chow and Kanenko* approach and (ii) *local* thresholding. The assumption behind both methods is that smaller image regions are more likely to have approximately uniform illumination, thus being more suitable for thresholding. Chow and Kanenko divide an image into an array of overlapping subimages and then find the optimum threshold for each subimage by investigating its histogram. The threshold for each single pixel is found by interpolating the results of the subimages. The drawback of this method is that it is computationally expensive and, therefore, is not appropriate for real-time applications.

An alternative approach to finding the local threshold is to statistically examine the intensity values of the local neighborhood of each pixel. The statistic which is most appropriate depends largely on the input image. Simple and fast functions include the *mean* of the *local* intensity distribution,

$$T = \text{mean}$$

the *median* value,

$$T = \text{median}$$

or the mean of the minimum and maximum values,

$$T = \frac{\text{max} - \text{min}}{2}$$

The size of the neighborhood has to be large enough to cover sufficient foreground and background pixels, otherwise a poor threshold is chosen. On the other hand, choosing regions which are too large can violate the assumption of approximately uniform illumination. This method is less computationally intensive than the Chow and Kanenko approach and produces good results for some applications.

Guidelines for Use

Like global thresholding (p.69), adaptive thresholding is used to separate desirable foreground image objects from the background based on the difference in pixel intensities of each region. Global thresholding uses a fixed threshold for all pixels in the image and therefore works only if the intensity histogram (p.105) of the input image contains neatly separated peaks corresponding

to the desired subject(s) and background(s). Hence, it cannot deal with images containing, for example, a strong illumination gradient.

Local adaptive thresholding, on the other hand, selects an individual threshold for each pixel based on the range of intensity values in its local neighborhood. This allows for thresholding of an image whose global intensity histogram doesn't contain distinctive peaks.

A task well suited to local adaptive thresholding is in segmenting text from the image `son1`. Because this image contains a strong illumination gradient, global thresholding produces a very poor result, as can be seen in `son1thr1`.

Using the *mean* of a 7×7 neighborhood, adaptive thresholding yields `son1adp1`. The method succeeds in the area surrounding the text because there are enough foreground and background pixels in the local neighborhood of each pixel; *i.e.* the mean value lies between the intensity values of foreground and background and, therefore, separates easily. On the margin, however, the *mean* of the local area is not suitable as a threshold, because the range of intensity values within a local neighborhood is very small and their *mean* is close to the value of the center pixel.

The situation can be improved if the threshold employed is not the *mean*, but $(\text{mean} - C)$, where C is a constant. Using this statistic, all pixels which exist in a uniform neighborhood (*e.g.* along the margins) are set to background. The result for a 7×7 neighborhood and $C=7$ is shown in `son1adp2` and for a 75×75 neighborhood and $C=10$ in `son1adp3`. The larger window yields the poorer result, because it is more adversely affected by the illumination gradient. Also note that the latter is more computationally intensive than thresholding using the smaller window.

The result of using the median (p.153) instead of the *mean* can be seen in `son1adp4`. (The neighborhood size for this example is 7×7 and $C = 4$). The result shows that, in this application, the median is a less suitable statistic than the mean.

Consider another example image containing a strong illumination gradient `wdg3`. This image can not be segmented with a global threshold, as shown in `wdg3thr1`, where a threshold of 80 was used. However, since the image contains a large object, it is hard to apply adaptive thresholding, as well. Using the $(\text{mean} - C)$ as a local threshold, we obtain `wdg3adp1` with a 7×7 window and $C = 4$, and `wdg3adp2` with a 140×140 window and $C = 8$. All pixels which belong to the object but do not have any background pixels in their neighborhood are set to background. The latter image shows a much better result than that achieved with a global threshold, but it is still missing some pixels in the center of the object. In many applications, computing the mean of a neighborhood (for each pixel!) whose size is of the order 140×140 may take too much time. In this case, the more complex *Chow and Kanenko* approach to adaptive thresholding would be more successful.

If your image processing package does not contain an adaptive threshold operator, you can simulate the effect with the following steps:

1. Convolve the image with a suitable statistical operator, *i.e.* the *mean* or *median*.
2. Subtract the original from the convolved image.
3. Threshold the difference image with C .
4. Invert the thresholded image.

Exercises

1. In the above example using `son1`, why does the *mean* produce a better result than the *median*? Can you think of any example where the *median* is more appropriate?
2. Think of an appropriate statistic for finding dark cracks on a light object using adaptive thresholding.
3. If you want to recover text from an image with a strong illumination gradient, how does the local thresholding method relate to the technique of removing the illumination gradient

using pixel subtraction (p.45)? Compare the results achieved with adaptive thresholding, pixel subtraction and pixel division (p.50).

References

E. Davies *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, pp 91 - 96.

R. Gonzales and R. Woods *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 443 - 452.

A. Jain *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986, p 408.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

6.3 Contrast Stretching

Brief Description

Contrast stretching (often called normalization) is a simple image enhancement technique that attempts to improve the contrast in an image by ‘stretching’ the range of intensity values it contains to span a desired range of values, *e.g.* the the full range of pixel values (p.239) that the image type concerned allows. It differs from the more sophisticated histogram equalization (p.78) in that it can only apply a *linear* scaling function to the image pixel values. As a result the ‘enhancement’ is less harsh. (Most implementations accept a graylevel image (p.232) as input and produce another graylevel image as output.)

How It Works

Before the stretching can be performed it is necessary to specify the upper and lower pixel value limits over which the image is to be normalized. Often these limits will just be the minimum and maximum pixel values that the image type concerned allows. For example for 8-bit graylevel images the lower and upper limits might be 0 and 255. Call the lower and the upper limits a and b respectively.

The simplest sort of normalization then scans the image to find the lowest and highest pixel values currently present in the image. Call these c and d . Then each pixel P is scaled using the following function:

$$P_{out} = (P_{in} - c) \left(\frac{b - a}{d - c} \right) + a$$

The problem with this is that a single outlying pixel with either a very high or very low value can severely affect the value of c or d and this could lead to very unrepresentative scaling. Therefore a more robust approach is to first take a histogram (p.105) of the image, and then select c and d at, say, the 5th and 95th percentile in the histogram (that is, 5% of the pixel in the histogram will have values lower than c , and 5% of the pixels will have values higher than d). This prevents outliers affecting the scaling so much.

Another common technique for dealing with outliers is to use the intensity histogram to find the most popular intensity level in an image (*i.e.* the histogram peak) and then define a *cutoff fraction* which is the minimum fraction of this peak magnitude below which data will be ignored. In other words, all intensity levels with histogram counts below this cutoff fraction will be discarded (driven to intensity value 0) and the remaining range of intensities will be expanded to fill out the full range of the image type under consideration.

Some implementations also work with color images (p.225). In this case all the channels will be stretched using the same offset and scaling in order to preserve the correct color ratios.

Guidelines for Use

Normalization is commonly used to improve the contrast in an image without distorting relative graylevel intensities too significantly.

We begin by considering an image `wom1` which can easily be enhanced by the most simple of contrast stretching implementations because the intensity histogram (p.105) forms a tight, narrow cluster between the graylevel intensity values of 79 - 136, as shown in `wom1hst1`. After contrast stretching, using a simple linear interpolation between $c = 79$ and $d = 136$, we obtain `wom1str1`. Compare the histogram of the original image with that of the contrast-stretched version `wom1hst2`.

While this result is a significant improvement over the original, the enhanced image itself still appears somewhat flat. Histogram equalizing (p.78) the image increases contrast dramatically, but

yields an artificial-looking result `wom1heq1`. In this case, we can achieve better results by contrast stretching the image over a more narrow range of graylevel values from the original image. For example, by setting the cutoff fraction parameter to 0.03, we obtain the contrast-stretched image `wom1str2` and its corresponding histogram `wom1hst3`. Note that this operation has effectively spread out the information contained in the original histogram peak (thus improving contrast in the interesting face regions) by pushing those intensity levels to the left of the peak down the histogram x -axis towards 0. Setting the cutoff fraction to a high value, *e.g.* 0.8, yields the contrast stretched image `wom1str3`. As shown in the histogram `wom1hst4`, most of the information to the left of the peak in the original image is mapped to 0 so that the peak can spread out even further and begin pushing values to its right up to 255.

As an example of an image which is more difficult to enhance, consider `moo2` which shows a low contrast image of a lunar surface.

The image `moo2hst2` shows the intensity histogram of this image. Note that only part of the y -axis has been shown for clarity. The minimum and maximum values in this 8-bit image are 0 and 255 respectively, and so straightforward normalization to the range 0 - 255 produces absolutely no effect. However, we *can* enhance the picture by ignoring all pixel values outside the 1% and 99% percentiles, and only applying contrast stretching to those pixels in between. The outliers are simply forced to either 0 or 255 depending upon which side of the range they lie on.

`moo2str1` shows the result of this enhancement. Notice that the contrast has been significantly improved. Compare this with the corresponding enhancement achieved using histogram equalization (p.78).

Normalization can also be used when converting from one image type (p.239) to another, for instance from floating point pixel values to 8-bit integer pixel values. As an example the pixel values in the floating point image might run from 0 to 5000. Normalizing this range to 0-255 allows easy conversion to 8-bit integers. Obviously some information might be lost in the compression process, but the relative intensities of the pixels will be preserved.

Exercises

1. Derive the scaling formula given above from the parameters a , b , c and d .
2. Suppose you had to normalize an 8-bit image to one in which the pixel values were stored as 4-bit integers. What would be a suitable destination range (*i.e.* the values of a and b)?
3. Contrast-stretch the image `sap1`. (You must begin by selecting suitable values for c and d .) Next, edge-detect (*i.e.* using the Sobel (p.188), Roberts Cross (p.184) or Canny (p.192) edge detector) both the original and the contrast stretched version. Does contrast stretching increase the number of edges which can be detected?
4. Imagine you have an image taken in low light levels and which, as a result, has low contrast. What are the advantages of using contrast stretching to improve the contrast, rather than simply scaling (p.48) the image by a factor of, say, three?

References

- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, pp 26 - 27, 79 - 99.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989, Chap. 7, p 235.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, p 45.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

6.4 Histogram Equalization

Brief Description

Histogram modeling techniques (*e.g.* histogram equalization) provide a sophisticated method for modifying the dynamic range and contrast of an image by altering that image such that its intensity histogram (p.105) has a desired shape. Unlike contrast stretching (p.75), histogram modeling operators may employ *non-linear* and *non-monotonic* transfer functions to map between pixel intensity values (p.239) in the input and output images. Histogram equalization employs a monotonic, non-linear mapping which re-assigns the intensity values of pixels in the input image such that the output image contains a uniform distribution of intensities (*i.e.* a flat histogram). This technique is used in image comparison processes (because it is effective in detail enhancement) and in the correction of non-linear effects introduced by, say, a digitizer or display system.

How It Works

Histogram modeling is usually introduced using continuous, rather than discrete, process functions. Therefore, we suppose that the images of interest contain continuous intensity levels (in the interval $[0,1]$) and that the transformation function f which maps an input image $A(x, y)$ onto an output image $B(x, y)$ is continuous within this interval. Further, it will be assumed that the transfer law (which may also be written in terms of intensity density levels, *e.g.* $D_B = f(D_A)$) is single-valued and monotonically increasing (as is the case in histogram equalization) so that it is possible to define the inverse law $D_A = f^{-1}(D_B)$. An example of such a transfer function is illustrated in Figure 6.3.

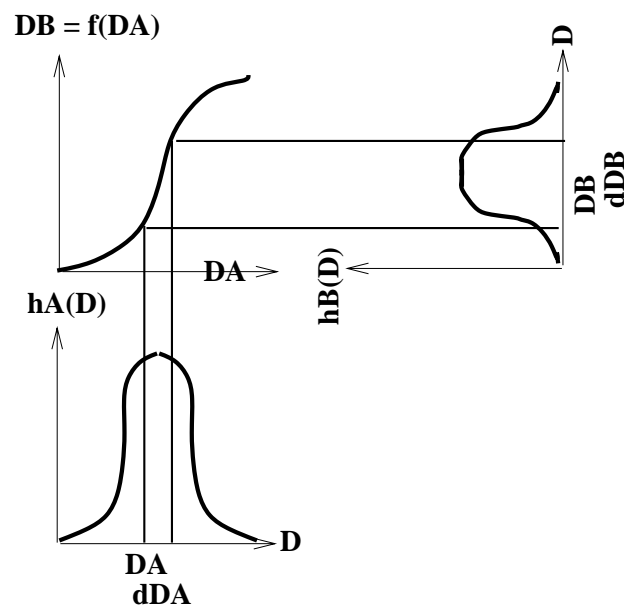


Figure 6.3: A histogram transformation function.

All pixels in the input image with densities in the region D_A to $D_A + dD_A$ will have their pixel values re-assigned such that they assume an output pixel density value in the range from D_B to $D_B + dD_B$. The surface areas $h_A(D_A)dD_A$ and $h_B(D_B)dD_B$ will therefore be equal, yielding:

$$h_B(D_B) = h_A(D_A) \div d(D_A)$$

where $d(x) = \frac{df(x)}{dx}$.

This result can be written in the language of probability theory if the histogram h is regarded as a continuous probability density function p describing the distribution of the (assumed random) intensity levels:

$$p_B(D_B) = p_A(D_A) \div d(D_A)$$

In the case of histogram equalization, the output probability densities should all be an equal fraction of the maximum number of intensity levels in the input image D_M (where the minimum level considered is 0). The transfer function (or point operator) necessary to achieve this result is simply:

$$d(D_A) = D_M * p_A(D_A)$$

Therefore,

$$f(D_A) = D_M \int_0^{D_A} p_A(u) du = D_M * F_A(D_A)$$

where $F_A(D_A)$ is simply the cumulative probability distribution (*i.e.* cumulative histogram) of the original image. *Thus, an image which is transformed using its cumulative histogram yields an output histogram which is flat!*

A digital implementation of histogram equalization is usually performed by defining a transfer function of the form:

$$f(D_A) = \max(0, \text{round}[D_M * n_k / N^2] - 1)$$

where N is the number of image pixels and n_k is the number of pixels at intensity level k or less.

In the digital implementation, the output image will not necessarily be fully equalized and there may be 'holes' in the histogram (*i.e.* unused intensity levels). These effects are likely to decrease as the number of pixels and intensity quantization levels in the input image are increased.

Guidelines for Use

To illustrate the utility of histogram equalization, consider `mo02` which shows an 8-bit grayscale image (p.232) of the surface of the moon. The histogram `mo02hst2` confirms what we can see by visual inspection: this image has poor dynamic range. (Note that we can view this histogram as a description of pixel probability densities by simply scaling the vertical axis by the total number of image pixels and normalizing the horizontal axis using the number of intensity density levels (*i.e.* 256). However, the shape of the distribution will be the same in either case.)

In order to improve the contrast of this image, without affecting the structure (*i.e.* geometry) of the information contained therein, we can apply the histogram equalization operator. The resulting image is `mo02heq1` and its histogram is shown `mo02hst1`. Note that the histogram is not flat (as in the examples from the continuous case) but that the dynamic range and contrast have been enhanced. Note also that when equalizing images with narrow histograms and relatively few gray levels, increasing the dynamic range has the adverse effect of increasing visual graininess. Compare this result with that produced by the linear contrast stretching (p.75) operator `mo02str1`.

In order to further explore the transformation defined by the histogram equalization operator, consider the image of the Scott Monument in Edinburgh, Scotland `bld1`. Although the contrast on the building is acceptable, the sky region is represented almost entirely by light pixels. This causes most histogram pixels `bld1hst1` to be pushed into a narrow peak in the upper graylevel region. The histogram equalization operator defines a mapping based on the cumulative histogram `bld1cuh1` which results in the image `bld1heq1`. While histogram equalization has enhanced the

contrast of the sky regions in the image, the picture now looks artificial because there is very little variety in the middle graylevel range. This occurs because the transfer function is based on the shallow slope of the cumulative histogram in the middle graylevel regions (*i.e.* intensity density levels 100 - 230) and causes many pixels from this region in the original image to be mapped to similar graylevels in the output image.

We can improve on this if we define a mapping based on a sub-section of the image which contains a better distribution of intensity densities from the low and middle range graylevels. If we crop the image so as to isolate a region which contains more building than sky `bld1crp1`, we can then define a histogram equalization mapping for the whole image based on the cumulative histogram `bld1cuH2` of this smaller region. Since the cropped image contains a more even distribution of dark and light pixels, the slope of the transfer function is steeper and smoother, and the contrast of the resulting image `bld1heq2` is more natural. This idea of defining mappings based upon particular sub-sections of the image is taken up by another class of operators which perform *Local Enhancements* as discussed below.

Common Variants

Histogram Specification

Histogram equalization is limited in that it is capable of producing only one result: an image with a uniform intensity distribution. Sometimes it is desirable to be able to control the shape of the output histogram in order to highlight certain intensity levels in an image. This can be accomplished by the histogram specialization operator which maps a given intensity distribution $a(x, y)$ into a desired distribution $c(x, y)$ using a histogram equalized image $b(x, y)$ as an intermediate stage.

The first step in histogram specialization, is to specify the desired output density function and write a transformation $g(c)$. If g^{-1} is single-valued (which is true when there are no unfilled levels in the specified histogram or errors in the process of rounding off g^{-1} to the nearest intensity level), then $c = g^{-1}(b)$ defines a mapping from the equalized levels of the original image, $b(x, y) = f[a(x, y)]$. It is possible to combine these two transformations such that the image need not be histogram equalized explicitly:

$$c = g^{-1}[f(a)]$$

Local Enhancements

The histogram processing methods discussed above are global in the sense that they apply a transformation function whose form is based on the intensity level distribution of an entire image. Although this method can enhance the overall contrast and dynamic range of an image (thereby making certain details more visible), there are cases in which enhancement of details over small areas (*i.e.* areas whose total pixel contribution to the total number of image pixels has a negligible influence on the global transform) is desired. The solution in these cases is to derive a transformation based upon the intensity distribution in the local neighborhood of every pixel in the image.

The histogram processes described above can be adapted for local enhancement. The procedure involves defining a neighborhood around each pixel and, using the histogram characteristics of this neighborhood, to derive a transfer function which maps that pixel into an output intensity level. This is performed for each pixel in the image. (Since moving across rows or down columns only adds one new pixel to the local histogram, updating the histogram from the previous calculation with new data introduced at each motion is possible.) Local enhancement may also define transforms based on pixel attributes other than histogram, *e.g.* intensity mean (to control variance) and variance (to control contrast) are common.

Exercises

- Suppose that you have a 128×128 square pixel image with an 8 gray level (p.232) intensity range, within which the lighter intensity levels predominate as shown in the table below. **A)** Sketch the histogram (p.105) (number of pixels vs gray level) to describe this distribution. **B)** How many pixels/gray levels would there be in an equalized version of this histogram? **C)** Apply the discrete transformation described above and plot the new (equalized) histogram. (How well does the histogram approximate a uniform distribution of intensity values?)

Gray Level	Number of Pixels
0	34
1	50
2	500
3	1500
4	2700
5	4500
6	4000
7	3100

- Suppose you have equalized an image once. Show that a second pass of histogram equalization will produce exactly the same result as the first.
- Interpreting images derived by means of a non-monotonic or non-continuous mapping can be difficult. Describe the effects of the following transfer functions:
 - f has a horizontal plateau,
 - f contains a vertical jump,
 - f has a negative slope.
 (Hint: it can be useful to sketch the curve, as in Figure 6.3, and then map a few points from histogram A to histogram B.)
- Apply local histogram equalization to the image `b1d1`. Compare this result with those derived by means of the global transfer function shown in the above examples.
- Apply global and local histogram equalization to the montage image `soi1`. Compare your results.

References

- R. Boyle and R. Thomas** *Computer Vision: A First Course*, Blackwell Scientific Publications, 1988, pp 35 - 41.
- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, Chap. 4.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986, pp 241 - 243.
- A. Marion** *An Introduction to Image Processing*, Chapman and Hall, 1991, Chap. 6.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

6.5 Logarithm Operator

Brief Description

The dynamic range of an image can be compressed by replacing each pixel value (p.239) with its logarithm. This has the effect that low intensity pixel values are enhanced. Applying a pixel logarithm operator to an image can be useful in applications where the dynamic range may too large to be displayed on a screen (or to be recorded on a film in the first place).

How It Works

The logarithmic operator is a simple point processor (p.68) where the *mapping function* is a logarithmic curve. In other words, each pixel value is replaced with its logarithm. Most implementations take either the *natural logarithm* or the *base 10 logarithm*. However, the basis does not influence the shape of the logarithmic curve, only the scale of the output values which are scaled (p.48) for display on an 8-bit system. Hence, the basis does not influence the degree of compression of the dynamic range. The logarithmic mapping function is given by

$$Q(i, j) = c \log(|P(i, j)|)$$

Since the logarithm is not defined for 0, many implementations of this operator add the value 1 to the image before taking the logarithm. The operator is then defined as

$$Q(i, j) = c \log(1 + |P(i, j)|)$$

The scaling constant c is chosen so that the maximum output value is 255 (providing an 8-bit format (p.232)). That means if R is the value with the maximum magnitude in the input image, c is given by

$$c = \frac{255}{\log(1 + |R|)}$$

The degree of compression (which is equivalent to the curvature of the mapping function) can be controlled by adjusting the range of the input values. Since the logarithmic function becomes more linear close to the origin, the compression is smaller for an image containing small input values. The *mapping function* is shown for two different ranges of input values in Figure 6.4.

Guidelines for Use

The most common application for the dynamic range compression is for the display of the Fourier Transform (p.209). We will illustrate this using `cln1`. The maximum magnitude value of its Fourier Transform is 7.9×10^6 , and the second largest value is approximately 10 times smaller. If we simply linearly scale (p.48) this image, we obtain `cln1fur1`. Due to the large dynamic range, we can only recognize the largest value in the center of the image. All remain values appear as black on the screen. If we instead apply the logarithmic operator to the Fourier image, we obtain `cln1fur2`. Here, smaller pixel values are enhanced and therefore the image shows significantly more details.

The logarithmic operator enhances the low intensity pixel values, while compressing high intensity values into a relatively small pixel range. Hence, if an image contains some important high intensity information, applying the logarithmic operator might lead to loss of information. For example, `stp1fur1` is the linearly scaled (p.48) Fourier Transform of `stp1`. The image shows one bright spot in the center and two darker spots on the diagonal. We can infer from the image that these three frequencies are the main components of the image with the DC-value having the largest magnitude. Applying the logarithmic transform to the Fourier image yields `stp1fur2`. Here, we can see that the image contains many more frequencies. However, it is now hard to tell which are

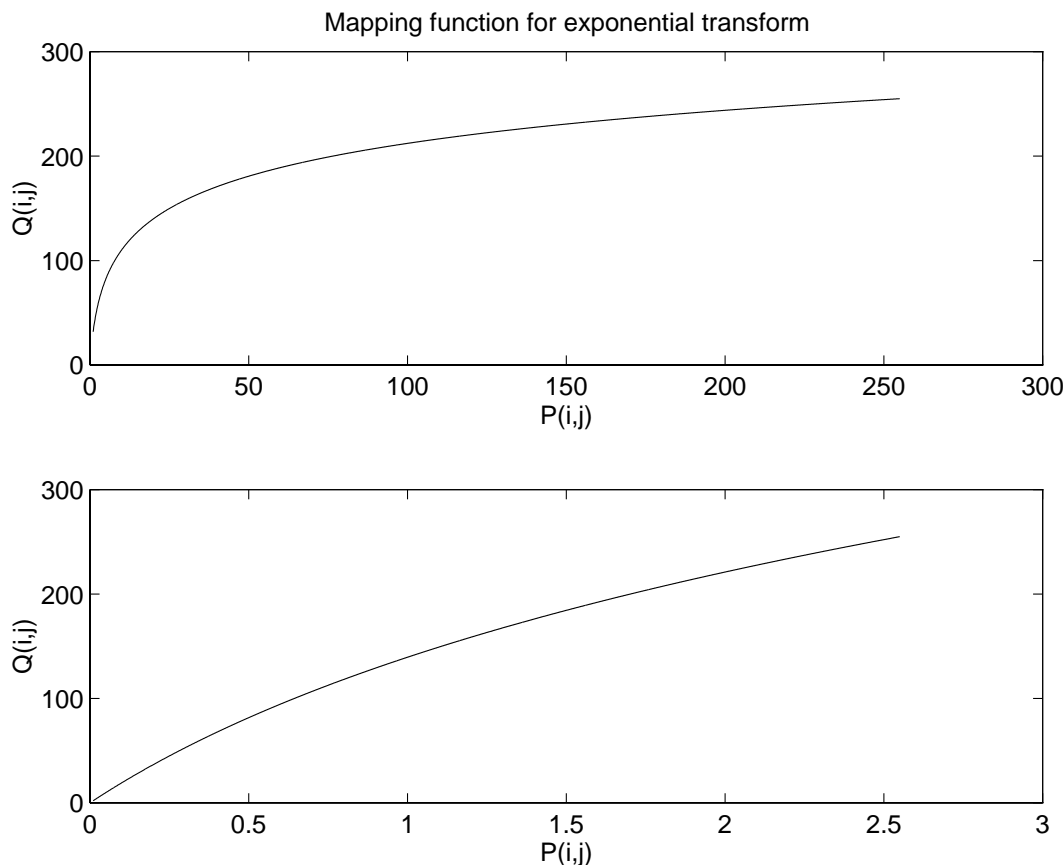


Figure 6.4: Logarithmic mapping functions at two different scales.

the dominating ones, since all high magnitudes are compressed into a rather small pixel value range. The magnitude of compression is large in this case because there are extremely high intensity values in the output of the Fourier Transform (in this case up to 7×10^6). We can decrease the compression rate by scaling (p.48) down the Fourier image before applying the logarithmic transform. Image `stp1fur6` is the result of first multiplying each pixel with 0.0001 and then taking its logarithm. Now, we can recognize all the main components of the Fourier image and can even see the difference in their intensities.

Thus, a logarithmic transform is appropriate when we want to enhance the low pixel values at the expense of loss of information in the high pixel values. For example, the man in `man8` was photographed in front of a bright background. The dynamic range of the film material is too small, so that the graylevels on the subject's face are clustered in a small pixel value range. A logarithmic transform spreads them over a wider range, while the higher values are compressed. The result can be seen in `man8log1`.

On the other hand, applying a logarithmic transform to `svs1` is less appropriate, because most of its details are contained in the high pixel values. Applying the logarithmic operator yields `svs1log1`. This image shows that a lot of information is lost during the transform.

Common Variants

The logarithmic operator is a member of the family of anamorphosis operators (p.68), which are *LUT transformations* with a strictly increasing or decreasing *mapping function*.

An anamorphosis operator which is similar to the logarithmic transform is the *square-root* operator.

Its *mapping function* is defined as

$$Q(i, j) = \sqrt{P(i, j)}$$

Both operators increase the contrast of low pixel values at the cost of the contrast of high pixel values. Hence, both are suitable to enhance details contained in the high values. However, they produce slightly different enhancements, since the shapes of their curves are not identical.

Exercises

1. Apply the logarithmic operator to `wom2`. Does this process improve the image. What is the reason? What is the result using `fce4`?
2. Is it generally a good idea to apply the logarithmic operator to astronomical images? Try it on `str2`.

References

R. Gonzalez and R. Woods *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 167 - 168.

A. Jain *Fundamentals of Digital Processing*, Prentice Hall, 1989, p 240.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

6.6 Exponential/‘Raise to Power’ Operator

Brief Description

The exponential and ‘raise to power’ operators are two anamorphosis (p.68) operators which can be applied to grayscale images (p.232). Like the logarithmic transform (p.82), they are used to change the dynamic range of an image. However, in contrast to the logarithmic operator, they enhance high intensity pixel values.

How It Works

The exponential operator is a point process (p.68) where the mapping function is an exponential curve. This means that each pixel intensity value in the output image is equal to a basis value raised to the value of the corresponding pixel value (p.239) in the input image. Which basis number is used depends on the desired degree of compression of the dynamic range. In order to enhance the visibility of a normal photograph, values just above 1 are suitable. For display, the image must be scaled (p.48) such that the maximum value becomes 255 (assuming an 8-bit display). The resulting image is given by

$$Q(i, j) = c b^{P(i, j)}$$

where P and Q are the input and output images, respectively, b is the basis and c is the scaling factor. As we can see from the formula, $Q=0$ yields $P=c$. To avoid the resulting offset, many implementations subtract 1 from the exponential term before the scaling. Hence, we get the following formula:

$$Q(i, j) = c (b^{P(i, j)} - 1)$$

Figure 6.5 shows the mapping function for $b = 10$ and $b = 1.01$.

We can see from the figure that the curvature of the *base 10* exponential function is far too high to enhance the visibility of normal image. We can control the curvature of the mapping function either by choosing the appropriate basis or by scaling down the image prior to applying the exponential operator. This is because, over a low range of input values, an exponential function with a high basis has the same shape as an exponential function with smaller basis over a larger range of input values.

In the case of the ‘raise to the power’ operator, the pixel intensity values in the input image act as the basis which is raised to a (fixed) power. The operator is defined by the following formula:

$$Q(i, j) = c P(i, j)^r$$

If $r > 1$, the ‘raise to power’ operator is similar to the exponential operator in the sense that it increases the bandwidth of the high intensity values at the cost of the low pixel values. However, if $r < 1$, the process enhances the low intensity value while decreasing the bandwidth of the high intensity values. This is similar to the effect achieved with the logarithmic transform (p.82).

Examples for $r = 0.5$, $r=2$ and $r=6$ can be seen in Figure 6.6.

Guidelines for Use

The exponential operator is the dual of the logarithmic transform (p.82). Hence, one application of the exponential operator is to undo changes originating from the logarithmic operator. In this case, the basis for the exponential operator should be the same as it was for the logarithmic transform.

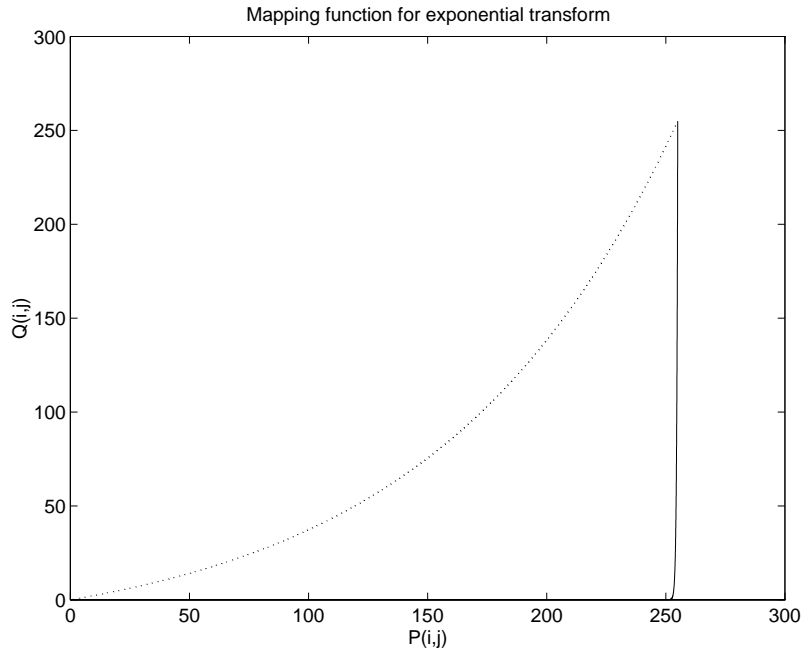


Figure 6.5: Exponential mapping functions for $b = 10$, $c = 2.55 \times 10^{253}$ (solid line) and $b = 1.01$, $c = 20.2$ (dotted line).

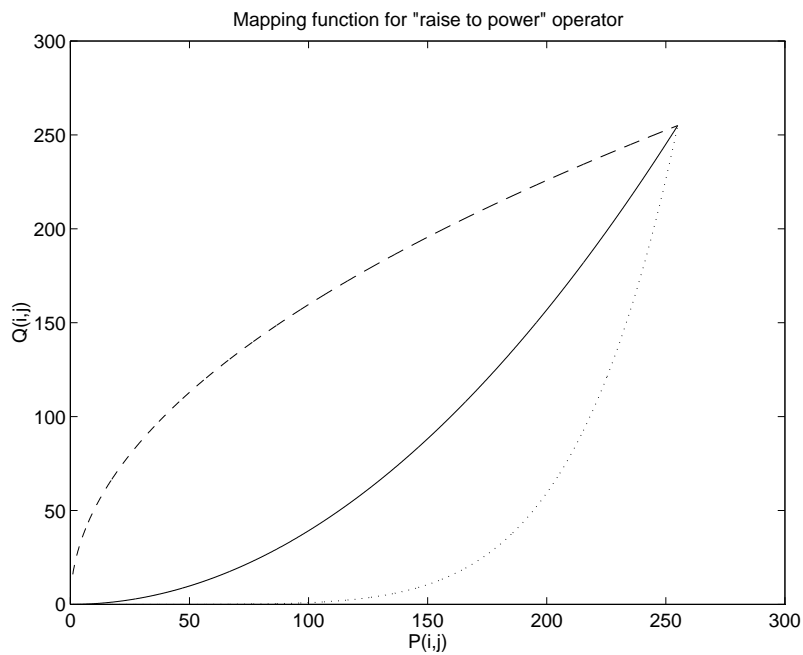


Figure 6.6: Mapping function of 'raise to power' operator for $r=0.5$ (dashed line), $r = 2$ (solid line) and $r=6$ (dotted line).

For example, `man8log1` was obtained from `man8` by applying a base 10 logarithm. If we need to restore the original version, we can do this by replacing each pixel with 10 to the power of its value. However, we have to be careful which image to use as input. If we take the above scaled and quantized image, the result of the exponential operator (after normalizing to the range 0 - 255) is

`man8exp1`. The reason for this is summarized in the equation below:

$$Q2(i, j) = 10^{Q1(i, j)} = 10^{k \log(P(i, j))} = P(i, j)^k$$

where P is the original image, $Q1$ is the image after the logarithmic transform and $Q2$ is the final result.

The effect can be seen in the shape of the *base 10* exponential function, as shown in Figure 6.5. To get the original image we need to apply the exponential operator to the unscaled image, containing the values directly after the logarithmic operation, *i.e.* in this case the log values will all be smaller than 3. If we store the unscaled image in a floating-point format (p.239) and then apply the exponential operator we obtain the correct original image: `man8exp2`.

Like the logarithmic operator, the exponential and ‘raise to power’ operators might be used to compress the dynamic range or generally enhance the visibility of an image. Because they can be used to enlarge the range of high intensity pixel values relative to the range of the low intensity values, these operators are suitable for enhancement of images containing details represented by a rather narrow range of high intensity pixel values. For example, in `fce4`, we might be interested in enhancing the details of the face, because it is slightly over-exposed. Applying an exponential transform with the base *1.005* yields `fce4exp1`. We can see that the contrast in the high pixel values has been increased at the cost of the contrast in the low pixel values. This effect is magnified if we increase the basis to *1.01*. As can be seen in `fce4exp2`, the resulting image now appears to be too dark.

For comparison, `fce4pow1` and `fce4pow2` show the original image after a ‘raise to power’ operation with $r = 1.5$ and $r = 2.5$, respectively. We can see that both exponential and ‘raise to power’ operator perform similarly at this (rather low) rate of compression. One difference, however, is that the ‘raise to power’ operator does not depend on the scale of the image. The exponential operator, on the other hand, compresses the dynamic range more if the image contains pixels with very high intensity values (*e.g.* an image in floating point format (p.239) with high pixel intensity values).

The exponential transform is not always appropriate to enhance the visibility of an image. For example, `wom2` is an under-exposed image and we would like to enhance the contrast of the low pixel values. The exponential operator, however, makes the situation even worse, as can be seen in `wom2exp1`. In this example, the logarithmic operator (p.82) would be more suitable.

The ‘raise to power’ operator is also known as gamma correction. In this case, the operator is used to correct for the non-linear response of a photographic film. The relation between the exposure of the film E and the resulting intensity value in the image I can be approximated, over a wide range of E , with the ‘raise to power’ operator:

$$I = E^\gamma$$

where γ is called the *gamma of the film*. A typical value for gamma lies between *0.7* and *1.0*. The gamma correction usually takes the gamma value as parameter and performs a ‘raise to power’ transform with $r=1/\textit{gamma}$ so that the relation between the initial exposure of the film and the intensity value in the corrected image becomes linear.

Exercises

1. Summarize the conditions under which the exponential and ‘raise to power’ operators are appropriate.
2. Apply the exponential operator to `leg1` and `trn1`. Which of the two images has improved? Try the logarithmic operator for comparison.
3. Apply the exponential and the ‘raise to power’ operator to `svs1`. What are the effects? Can you see any differences between the two resulting images?

References

D. Ballard and C. Brown *Computer Vision*, Prentice-Hall, 1982, p 45.

R. Gonzalez and R. Woods *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 72, 162 - 163.

A. Jain *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986, pp 232 - 234, 273.

A. Marion *An Introduction to Image Processing*, Chapman and Hall, 1991, pp 113 - 114.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

Chapter 7

Geometric Operations

A geometric operation maps pixel information (*i.e.* the intensity values at each pixel location (x_1, y_1)) in an input image to another location (x_2, y_2) in an output image. For basic operators described in this package, these functions are first order polynomials which take the form:

$$\begin{vmatrix} x_2 \\ y_2 \end{vmatrix} = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \times \begin{vmatrix} x_1 \\ y_1 \end{vmatrix} + \begin{vmatrix} b_1 \\ b_2 \end{vmatrix}$$

Translation (p.97) can be accomplished by specifying values for the B matrix, while scaling (p.90), rotation (p.93) and reflection (p.95) are defined by instantiating variables in the A matrix. A general affine transformation (p.100) (which uses both matrices) can be used to perform a combination of these operations at once, and is often used in applications, *e.g.* remote sensing, where we wish to correct for geometric distortions in the image introduced by perspective irregularities. Geometric operators are also used to improve the visualization of the image, *e.g.* zooming an interesting region of the image, or as a part of an image processing chain where, *e.g.* translation is required in order to register two images.

7.1 Geometric Scaling

Brief Description

The scale operator performs a geometric transformation which can be used to shrink or zoom the size of an image (or part of an image). Image reduction, commonly known as *subsampling*, is performed by replacement (of a group of pixel values (p.239) by one arbitrarily chosen pixel value from within this group) or by *interpolating* between pixel values in a local neighborhoods. Image zooming is achieved by *pixel replication* or by interpolation. Scaling is used to change the visual appearance of an image, to alter the quantity of information stored in a scene representation, or as a low-level preprocessor in multi-stage image processing chain which operates on features of a particular scale. Scaling is a special case of affine transformation (p.100).

How It Works

Scaling compresses or expands an image along the coordinate directions. As different techniques can be used to subsample and zoom, each is discussed in turn.

Figure 7.1 illustrates the two methods of sub-sampling. In the first, one pixel value within a local neighborhood is chosen (perhaps randomly) to be representative of its surroundings. (This method is computationally simple, but can lead to poor results if the sampling neighborhoods are too large.) The second method interpolates between pixel values within a neighborhood by taking a statistical sample (such as the mean) of the local intensity values.

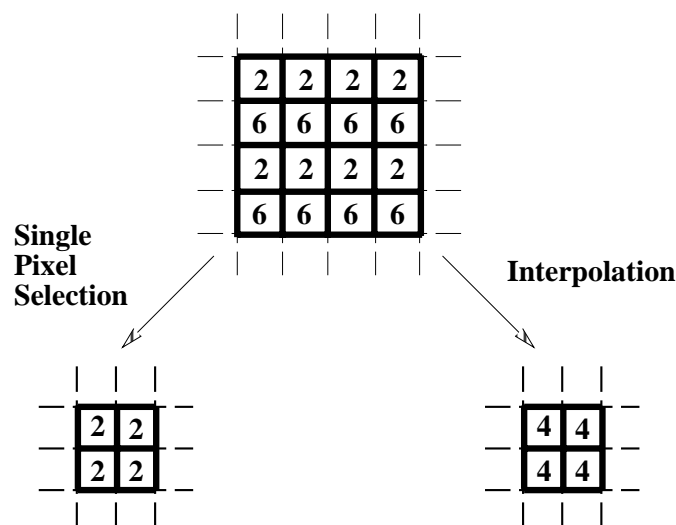


Figure 7.1: Methods of subsampling. a) Replacement with upper left pixel. b) Interpolation using the mean value.

An image (or regions of an image) can be zoomed either through pixel replication or interpolation. Figure 7.2 shows how pixel replication simply replaces each original image pixel by a group of pixels with the same value (where the group size is determined by the scaling factor). Alternatively, interpolation of the values of neighboring pixels in the original image can be performed in order to replace each pixel with an expanded group of pixels. Most implementations offer the option of increasing the actual dimensions of the original image, or retaining them and simply zooming a portion of the image within the old image boundaries.

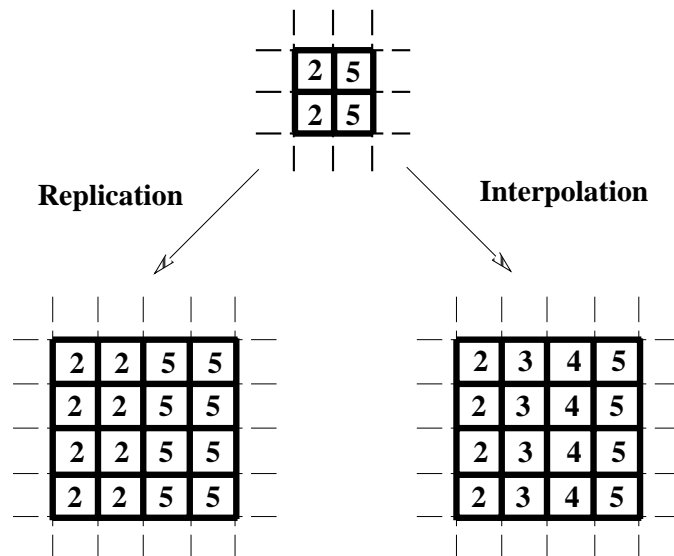


Figure 7.2: Methods of zooming. a) Replication of a single pixel value. b) Interpolation.

Guidelines for Use

Image sampling is a fundamental operation of any image processing system. In a digital computer, images are represented as arrays of finite size numbers. The size of the image array is determined by the number of sampling points (p.238) measured. The information at each sampling position in the array represents the average irradiance over the small sampling area and is *quantized* into a finite number of bits.

How many sampling points and quantization levels are required to make a good approximation to the continuous image? The *resolution* of the image increases as we increase the number of sampling and quantization levels. However, large values for these parameters also increase the image storage space and processing requirements. Therefore, the decision of how to set these parameters must involve striking a compromise between these competing objectives. Fortunately, sampling theory gives us some boundaries within which to make an otherwise largely qualitative decision. If we model images as bandlimited signals, Fourier analysis tells us that sampling at frequencies greater than twice the image bandwidths allows us to recover images without error (by lowpass filtering (p.148) the sampled image). However if the sampling frequencies are below this *Nyquist* limit, then the spectrum will be distorted and information will be lost. This phenomenon is known as *aliasing*.

We can witness this effect by subsampling the test image `tst2`. The effect of reducing the sampling grid size by a half is shown in `tst2sca1`. The maximum number of intensity levels, or quantization levels, is held constant in this example. Also, since the display area used in each example is the same, pixels in the lower resolution images are duplicated in order to fill out the display field. Reducing the resolution by a factor of 4, 8, and 16 are shown in `tst2sca2`, `tst2sca3` and `tst2sca4`, respectively.

Next we consider the different methods of implementing subsampling using a series of examples based on the basic image `wat1str1`. This image is first subsampled using (i) pixel replacement, as shown in `wat1psh1` and (ii) pixel interpolation, as shown in `wat1pin1`. (In each case the image is reduced by a factor of 4.) Notice how the latter produces a smoother image with less loss of information on the upper watch face.

We can zoom in on a region of the watch image using both (i) pixel replication `wat1exp1` and (ii) interpolation `wat1pin2`. Again the interpolation method produced a slightly smoother result, but the differences in quality are rather insignificant as the neighborhood size used was small (*i.e.*

2) to keep the computation time down when processing such a large image. (Note, the zooming facilities used here produced different sized outputs; *i.e.* pixel replication created an image which was increased in size by the scale factor and pixel interpolation cropped the zoomed image to remain within the boundaries of the original image. To aid viewing, the pixel replicated image was manually cropped to cut it down to the approximate size of the interpolated image.)

Zooming another, smaller image `cam1` shows more clearly the effects of increasing the neighborhood size. Using a scaling factor of 3, pixel replication produces `cam1exp1` and pixel interpolation yields `cam1pin1`. At this scale factor, we begin to see the undesirable effects of block edge features appearing in the replicated image.

Scaling algorithms are implemented on hardware (*e.g.* a zooming facility is provided on most frame grabbers) as well as software. Subsampling has utility in applications where information can be reduced without losing the main characteristics of the image, while image expansion is often used to examine details and local image information.

Exercises

1. Using the binary images `art5`, `art6inv1` and `art7`, make a collage by scaling each image so that the object in the second image can fit within the object in the first image, and the third within the second.
2. Explore the effects of subsampling noisy images. Using the image `fce5noi3` – which has been contaminated with salt and pepper noise (p.221) and `fce5noi5` – which has been corrupted by Gaussian noise (p.221). Subsample each image using both the pixel replacement and pixel interpolation methods. Does one algorithm cope with noise better than the other? Which type of noise is more corrupting to the subsampled image.
3. Using objects of different sizes, determine the scale of subsampling at which the objects are no longer visible in the output images. Experiment with images `bri1` and `mof1`. Repeat this analysis for both types of subsampling.

References

- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, pp 465 - 469, 470.
- R. Gonzalez and P. Wintz** *Digital Image Processing, 2nd edition*, Addison-Wesley Publishing Company, 1987, pp 22 - 26.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986, Chap. 4.
- A. Marion** *An Introduction to Image Processing*, Chapman and Hall, 1991, Chap. 5.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

7.2 Rotate

Brief Description

The rotation operator performs a geometric transform which maps the position (x_1, y_1) of a picture element (p.238) in an input image onto a position (x_2, y_2) in an output image by rotating it through a user-specified angle θ about an origin O . In most implementations, output locations (x_2, y_2) which are outside the boundary of the image are ignored. Rotation is most commonly used to improve the visual appearance of an image, although it can be useful as a preprocessor in applications where directional operators are involved. Rotation is a special case of affine transformation (p.100).

How It Works

The rotation operator performs a transformation of the form:

$$x_2 = \cos(\theta) * (x_1 - x_0) - \sin(\theta) * (y_1 - y_0) + x_0$$

$$y_2 = \sin(\theta) * (x_1 - x_0) + \cos(\theta) * (y_1 - y_0) + y_0$$

where (x_0, y_0) are the coordinates of the center of rotation (in the input image) and θ is the angle of rotation. Even more than the translate (p.97) operator, the rotation operation produces output locations (x_2, y_2) which do not fit within the boundaries of the image (as defined by the dimensions of the original input image). In such cases, destination elements which have been mapped outside the image are ignored by most implementations. Pixel locations out of which an image has been rotated are usually filled in with black pixels.

The rotation algorithm, unlike that employed by translation (p.97), can produce coordinates (x_2, y_2) which are not integers. In order to generate the intensity (p.239) of the pixels at each integer position, different heuristics (or *re-sampling* techniques) may be employed. For example, two common methods include:

- Allow the intensity level at each integer pixel position to assume the value of the nearest non-integer neighbor (x_2, y_2) .
- Calculate the intensity level at each integer pixel position based on a weighted average of the n nearest non-integer values. The weighting is proportional to the distance or pixel overlap of the nearby projections.

The latter method produces better results but increases the computation time of the algorithm.

Guidelines for Use

A rotation is defined by an angle θ and an origin of rotation O . For example, consider the image `art4ctr1` whose subject is centered. We can rotate the image through 180 degrees about the image (and circle) center at $(x = 150, y = 150)$ to produce `art4rot1`.

If we use these same parameter settings but a new, smaller image, such as the 222×217 size artificial, black-on-white image `art1`, we achieve poor results, as shown in `art1rot1`, because the specified axis of rotation is sufficiently displaced from the image center that much of the image is swept off the page. Likewise, rotating this image through a θ value which is not an integer multiple of 90 degrees (*e.g.* in this case θ equals 45 degrees) rotates part of the image off the visible output and leaves many empty pixel values, as seen in `art1rot2`. (Here, non-integer pixel values were re-sampled using the first technique mentioned above.)

Like translation (p.97), rotation may be employed in the early stages of more sophisticated image processing operations. For example, there are numerous directional operators in image processing

(e.g. many edge detection (p.230) and morphological operators (p.236)) and, in many implementations, these operations are only defined along a limited set of directions: 0, 45, 90, *etc.* A user may construct a hybrid operator which operates along any desired image orientation direction by first rotating an image through the desired direction, performing the edge detection (or erosion (p.123), dilation (p.118), *etc.*), and then rotating the image back to the original orientation. (See Figure 7.3.)

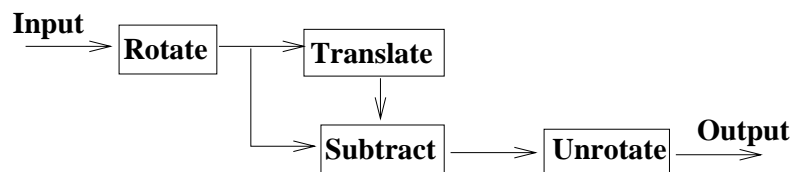


Figure 7.3: A variable-direction edge detector.

As an example, consider `art2` whose edges were detected by the directional operator defined using translation (p.97) giving `art2sub1`. We can perform edge detection along the opposite direction to that shown in the image by employing a 180 degree rotation in the edge detection algorithm. The result is shown in `art2trn2`. Notice the slight degradation of this image due to rotation re-sampling.

Exercises

1. Consider images `wdg4` and `wdg5` which contain L-shaped parts of different sizes. **a)** Rotate and translate (p.97) one of the images such that the bottom left corner of the “L” is in the same position in both images. **b)** Using a combination of histogramming (p.105), thresholding (p.69) and pixel arithmetic (e.g. pixel subtraction (p.45)) determine the approximate difference in size of the two parts.
2. Make a collage based on a series of rotations and pixel additions (p.43) of image `art9`. You should begin by centering the propeller in the middle of the image. Next, rotate the image through a series of 45 degree rotations and add each rotated version back onto the original. (Note: you can improve the visual appearance of the result if you scale (p.48) the intensity values of each rotated propeller a few shades before adding it onto the collage.)
3. Investigate the effects of re-sampling when using rotation as a preprocessing tool in an image erosion application. First erode the images `wdg2` and `b1b1` using a 90 degree directional erosion (p.123) operator. Next, rotate the image through 90 degrees before applying the directional erosion operator along the 0 degree orientation. Compare the results.

References

- D. Ballard and C. Brown** *Computer Vision*, Prentice-Hall, 1982, Appendix 1.
B. Horn *Robot Vision*, MIT Press, 1986, Chap. 3.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

7.3 Reflect

Brief Description

The reflection operator geometrically transforms an image such that image elements, *i.e.* pixel values (p.239), located at position (x_1, y_1) in an original image are reflected about a user-specified image *axis* or image *point* into a new position (x_2, y_2) in a corresponding output image. Reflection is mainly used as an aid to image visualization, but may be used as a preprocessing operator in much the same way as rotation (p.93). Reflection is a special case of affine transformation (p.100).

How It Works

Reflection can be performed about an image axis or a point in the image. In the case of the former, some commonly used transformations are the following:

- Reflection about a vertical axis of abscissa x_0 in the input image:

$$x_2 = -x_1 + (2 * x_0)$$

$$y_2 = y_1$$

- Reflection about a horizontal axis of ordinate y_0 :

$$x_2 = x_1$$

$$y_2 = -y_1 + (2 * y_0)$$

- Reflection about an axis oriented in any arbitrary direction θ , and passing through (x_0, y_0) :

$$x_2 = x_1 + 2 * \Delta * (-\sin(\theta))$$

$$y_2 = y_1 + 2 * \Delta * (\cos(\theta))$$

where $\Delta = (x_1 - x_0) * \sin(\theta) - (y_1 - y_0) * \cos(\theta)$.

Note that if (x_0, y_0) is not in the center of the input image, part of the image will be reflected out of the visible range of the image. Most implementations fill in image areas out of which pixels have been reflected with black pixels.

- From this discussion, it is easy to see that horizontal and vertical reflection about a point (x_0, y_0) in the input image are given by:

$$x_2 = -x_1 + (2 * x_0)$$

$$y_2 = -y_1 + (2 * y_0)$$

Guidelines for Use

The simplest reflection we can define reflects an image about an axis located in the center of an image. For example, we can reflect `art4ctr1` about a vertical axis in the center of the image to produce `art4ref1`. Similarly, `art7ref1` shows the reflection of `art7` about a horizontal axis passing through the image center.

Reflection about a point in the center of the image maps `wat1str1` into `wat1ref1`. This result, of course, could also be achieved by rotating the image through 180 degrees about its center.

A popular application for reflection is symmetry analysis. For example, consider the image `ape1`. A quick examination of this face might lead us to believe that the left and right halves were mirror images of each other. However, if we reflect this image (about a carefully selected axis running vertically between the eyes) and then create two new images such that (i) the first contains the original left half of the face, joined in the middle to a reflection of the left half and (ii) the second contains a similar description of the right half of the face, we see that this is not the case. A comparison of the left `ape1ref1` and right `ape1ref2` reflection images reveals differences in the fur color, eye shape/expression, nose orientation, whisker alignment, *etc.*

Exercises

1. Consider image `art9`. What sort of reflection might have produced `art9ref1`?
2. Using images `art2`, `son1div1` and `moo1`, compare the reflection and rotation operators in terms of their computational speed and the quality of the resultant image.
3. Perform a symmetry analysis (as in the example above) of the images `wom1str2` and `wom3`. Alignment of the axis of reflection with the center of the face is tricky. You might want to consider putting it at a position equi-distant between both eyes.
4. How might one inspect symmetric objects using reflection? Try your answer on an image containing a square with a corner missing).

References

- D. Ballard and C. Brown** *Computer Vision*, Prentice-Hall, 1982, Appendix 1.
B. Horn *Robot Vision*, MIT Press, 1986, Chap. 3.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

7.4 Translate

Brief Description

The translate operator performs a geometric transformation which maps the position of each picture element (p.238) in an input image into a new position in an output image, where the dimensionality of the two images often is, but need not necessarily be, the same. Under translation, an image element located at (x_1, y_1) in the original is shifted to a new position (x_2, y_2) in the corresponding output image by displacing it through a user-specified translation (β_x, β_y) . The treatment of elements near image edges varies with implementation. Translation is used to improve visualization of an image, but also has a role as a preprocessor in applications where registration of two or more images is required. Translation is a special case of affine transformation (p.100).

How It Works

The translation operator performs a transformation of the form:

$$\begin{aligned}x_2 &= x_1 + \beta_x \\y_2 &= y_1 + \beta_y\end{aligned}$$

Since the dimensions of the input image are well defined, the output image is also a discrete space of finite dimension. If the new coordinates (x_2, y_2) are outside the image, the translate operator will normally ignore them, although, in some implementations, it may link the higher coordinate points with the lower ones so as to wrap the result around back onto the visible space of the image. Most implementations fill the image areas out of which an image has been shifted with black pixels.

Guidelines for Use

The translate operator takes two arguments, (β_x, β_y) , which specify the desired horizontal and vertical pixel displacements, respectively. For example, consider the artificial image `art4ctr1`, in which the subject's center lies in the center of the 300×300 pixel image. We can naively translate the subject into the lower, right corner of the image by defining a mapping (*i.e.* a set of values) for (β_x, β_y) which will take the subject's center from its present position at $x = 150, y = 150$ to an output position of $(x' = 300, y' = 300)$, as shown in `art4trn1`. In this case, information is lost because pixels which were mapped to points outside the boundaries defined by the input image were ignored. If we perform the same translation, but wrap the result, all the intensity information is retained, giving image `art4trn2`.

Both of the mappings shown above disturb the original geometric structure of the scene. It is often the case that we perform translation merely to change the position of a scene object, not its geometric structure. In the above example, we could achieve this effect by translating the circle center to a position located at the lower, right corner of the image *less* the circle radius `art4trn3`. At this point, we might build a *collage* by adding (p.43) another image(s) whose subject(s) has been appropriately translated, such as in `art7trn1`, to the previous result. This simple collage is shown in `art7add1`.

Translation has many applications of the cosmetic sort illustrated above. However, it is also very commonly used as a preprocessor in application domains where registration of two or more images is required. For example, feature detection (p.183) and spatial filtering (p.148) algorithms may calculate gradients in such a way as to introduce an offset in the positions of the pixels in the output image with respect to the corresponding pixels from the input image. In the case of the Laplacian of Gaussian (p.173) spatial sharpening filter, some implementations require that the filtered image be translated by half the width of the Gaussian kernel with which it was convolved (p.227) in order to bring it into alignment with the original. Likewise, the unsharp filter (p.178), requires translation to achieve a re-registration of images. The result of subtracting (p.45) the smoothed

version of the image `wdg1` away from the original image (after translating the smoothed image by the offset induced by the filter before we subtract to re-align the two images) yields the edge image `wdg1usp2`.

We again view the effects of mis-alignment if we consider translating `art2` by one pixel in the x and y directions and then subtracting this result from the original. The resulting image, shown in `art2sub1`, contains a description of all the places (along the direction of translation) where the intensity gradients are different; *i.e.* it highlights edges (and noise (p.221)). The image `cln1` was used in examples of edge detection (p.230) using the Roberts Cross (p.184), Sobel (p.188) and Canny (p.192) operators. Compare this result to the translation-based edge-detector illustrated here `cln1trn1`. Note that if we increase the translation parameter too much, *e.g.*, by 6 pixels in each direction, as in `cln1trn2`, edges become severely mis-aligned and blurred.

Exercises

1. Investigate which arguments to the translation operator could perform the following translations: **a)** `art2` into `art2trn1`. **b)** `art3` into `art3trn1`.
2. We can create more interesting artificial images by combining the translate operation with other operators. For example, `art7` has been translated and then pixel added (p.43) back onto itself to produce `art7add2`. **a)** Produce an artificial image of this sort using `art6`. **b)** Combine `art5` and `art7` using translation and pixel addition into a collage.
3. Describe how you might derive a simple isotropic (p.233) edge detector (p.230) using a series of translation and subtraction (p.45) operations.
4. Would it be possible to make a simple sharpening filter (p.178) based on translation and pixel addition (p.43) or subtraction (p.45)? On what types of images might such a filter work?
5. How could one use translation to implement convolution (p.227) with the kernel shown in Figure 7.4.

0	-1	0
-1	4	-1
0	-1	0

Figure 7.4: Convolution kernel.

Can one implement every convolution using this approach?

References

- D. Ballard and C. Brown** *Computer Vision*, Prentice-Hall, 1982, Appendix 1.
B. Horn *Robot Vision*, MIT Press, 1986, Chap. 3.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

7.5 Affine Transformation

Brief Description

In many imaging systems, detected images are subject to geometric distortion introduced by perspective irregularities wherein the position of the camera(s) with respect to the scene alters the apparent dimensions of the scene geometry. Applying an affine transformation to a uniformly distorted image can correct for a range of perspective distortions by transforming the measurements from the ideal coordinates to those actually used. (For example, this is useful in satellite imaging where geometrically correct ground maps are desired.)

An affine transformation is an important class of linear 2-D geometric transformations which maps variables (*e.g.* pixel intensity values (p.239) located at position (x_1, y_1) in an input image) into new variables (*e.g.* (x_2, y_2) in an output image) by applying a linear combination of translation (p.97), rotation (p.93), scaling (p.90) and/or shearing (*i.e.* non-uniform scaling in some directions) operations.

How It Works

In order to introduce the utility of the affine transformation, consider the image `prt3`, wherein a machine part is shown lying in a fronto-parallel plane. The circular hole of the part is imaged as a circle, and the parallelism and perpendicularity of lines in the real world are preserved in the image plane. We might construct a model of this part using these primitives; however, such a description would be of little use in identifying the part from `prt4`. Here the circle is imaged as an ellipse, and orthogonal world lines are not imaged as orthogonal lines.

This problem of perspective can be overcome if we construct a shape description which is *invariant* to perspective projection. Many interesting tasks within model based computer vision can be accomplished without recourse to Euclidean shape descriptions (*i.e.* those requiring absolute distances, angles and areas) and, instead, employ descriptions involving *relative* measurements (*i.e.* those which depend only upon the configuration's intrinsic geometric relations). These relative measurements can be determined directly from images. Figure 7.5 shows a hierarchy of planar transformations which are important to computer vision.

The transformation of the part face shown in the example image above is approximated by a planar *affine* transformation. (Compare this with the image `prt5` where the distance to the part is not large compared with its depth and, therefore, parallel object lines begin to converge. Because the scaling varies with depth in this way, a description to the level of *projective* transformation is required.) An affine transformation is equivalent to the composed effects of translation (p.97), rotation (p.93), isotropic scaling (p.90) and shear.

The general affine transformation is commonly written in homogeneous coordinates as shown below:

$$\begin{vmatrix} x_2 \\ y_2 \end{vmatrix} = A \times \begin{vmatrix} x_1 \\ y_1 \end{vmatrix} + B$$

By defining only the B matrix, this transformation can carry out pure translation (p.97):

$$A = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}, B = \begin{vmatrix} b_1 \\ b_2 \end{vmatrix}$$

Pure rotation (p.93) uses the A matrix and is defined as:

$$A = \begin{vmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{vmatrix}, B = \begin{vmatrix} 0 \\ 0 \end{vmatrix}$$

Similarly, pure scaling (p.90) is:

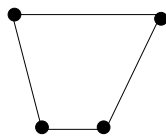
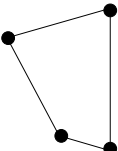

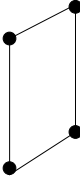
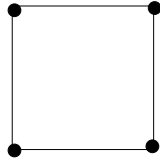
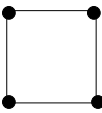
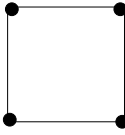
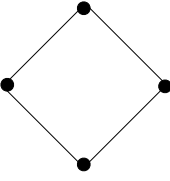
Transformation	Before	After
Projective		
Affine		
Similarity		
Euclidean		

Figure 7.5: Hierarchy of plane to plane transformation from Euclidean (where only rotations and translations are allowed) to Projective (where a square can be transformed into any more general quadrilateral where no 3 points are collinear). Note that transformations lower in the table inherit the invariants of those above, but because they possess their own groups of definitive axioms as well, the converse is not true.

$$A = \begin{vmatrix} a & 0 \\ 0 & a \end{vmatrix}, B = \begin{vmatrix} 0 \\ 0 \end{vmatrix}$$

(Note that several different affine transformations are often combined to produce a resultant transformation. The order in which the transformations occur is significant since a translation followed by a rotation is not necessarily equivalent to the converse.)

Since the general affine transformation is defined by 6 constants, it is possible to define this transformation by specifying the new output image locations (x_2, y_2) of any three input image coordinate (x_1, y_1) pairs. (In practice, many more points are measured and a least squares method is used to find the best fitting transform.)

Guidelines for Use

Most implementations of the affine operator allow the user to define a transformation by specifying to where 3 (or less) coordinate pairs from the input image (x_1, y_1) re-map in the output image (x_2, y_2) . (It is often the case, as with the implementation used here, that the user is restricted to re-mapping *corner coordinates* of the input image to *arbitrary new coordinates* in the output image.)

Once the transformation has been defined in this way, the re-mapping proceeds by calculating, for each output pixel location (x_2, y_2) , the corresponding input coordinates (x_1, y_1) . If that input point is outside of the image, then the output pixel is set to the background value. Otherwise, the value of (i) the input pixel itself, (ii) the neighbor nearest to the desired pixel position, or (iii) a bilinear interpolation of the neighboring four pixels is used.

We will illustrate the operation of the affine transformation by applying a series of special-case transformations (*e.g.* pure translation (p.97), pure rotation (p.93) and pure scaling (p.90)) and then some more general transformations involving combinations of these.

Starting with the 256×256 binary (p.225) artificial image `rlf1`, we can apply a translation using the affine operator in order to obtain the image `rlf1aff1`. In order to perform this pure translation, we define a transformation by re-mapping a single point (*e.g.* the input image lower-left corner $(0, 0)$) to a new position at $(64, 64)$.

A pure rotation requires re-mapping the position of two corners to new positions. If we specify that the lower-left corner moves to $(256, 0)$ and the lower-right corner moves to $(256, 256)$, we obtain `rlf1aff2`. Similarly, reflection (p.95) can be achieved by swapping the coordinates of two opposite corners, as shown in `rlf1aff3`.

Scaling (p.90) can also be applied by re-mapping just two corners. For example, we can send the lower-left corner to $(64, 64)$, while pinning the upper-right corner down at $(256, 256)$, and thereby uniformly shrink the size of the image subject by a quarter, as shown in `rlf1aff5`. Note that here we have also translated the image. Re-mapping any 2 points can introduce a combination of translation, rotation and scaling.

A general affine transformation is specified by re-mapping 3 points. If we re-map the input image so as to move the lower-left corner up to $(64, 64)$ along the 45 degree oblique axis, move the upper-right corner down by the same amount along this axis, and pin the lower-right corner in place, we obtain an image which shows some shearing effects `rlf1aff4`. Notice how parallel lines remain parallel, but perpendicular corners are distorted.

Affine transformations are most commonly applied in the case where we have a detected image which has undergone some type of distortion. The geometrically correct version of the input image can be obtained from the affine transformation by re-sampling the input image such that the information (or intensity) at each point (x_1, y_1) is mapped to the correct position (x_2, y_2) in a corresponding output image.

One of the more interesting applications of this technique is in remote sensing. However, because most images are transformed before they are made available to the image processing community, we will demonstrate the affine transformation with the terrestrial image `rot1str1`, which is a contrast-stretched (p.75) (cutoff fraction = 0.9) version of `rot1`. We might want to transform this image so as to map the door frame back into a rectangle. We can do this by defining a transformation based on a re-mapping of the (i) upper-right corner to a position 30% lower along the y -axis, (ii) the lower-right corner to a position 10% lower along the x -axis, and (iii) pinning down the upper-left corner. The result is shown in `rot1aff1`. Notice that we have defined a transformation which works well for objects at the depth of the door frame, but nearby objects have been distorted because the affine plane transformation cannot account for distortions at widely varying depths.

It is common for imagery to contain a number of perspective distortions. For example, the original image `boa1` shows both affine and projective type distortions due to the proximity of the camera with respect to the subject. After affine transformation, we obtain `boa1aff1`. Notice that the front face of the captain's house now has truly perpendicular angles where the vertical and horizontal members meet. However, the far background features have been distorted in the process and, furthermore, it was not possible to correct for the perspective distortion which makes the bow appear much larger than the hull,

Exercises

1. It is not always possible to accurately represent the distortion in an image using an affine

transformation. In what sorts of imaging scenarios would you expect to find non-linearities in a scanning process and/or differences in along-scans vs across-scans?

2. Apply an affine transformation to the image `hse1`. **a)** Experiment with different combinations of basic translation (p.97), rotation (p.93) and scaling (p.90) and then apply a transform which combines several of these operations. **b)** Rotate a translated version of the image and compare your result with the result of translating a rotated version of the image.

References

- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986, p 321.
- B. Horn** *Robot Vision*, MIT Press, 1986, pp 314 - 315.
- D. Marr** *Vision*, Freeman, 1982, p 185.
- A. Zisserman** *Notes on Geometric and Invariance in Vision*, British Machine Vision Association and Society for Pattern Recognition, 1992, Chap. 2.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

Chapter 8

Image Analysis

Many of the other categories of image processing operators demonstrated in HIPR transform an input image to form a new image without attempting to extract usable global information from the image. The operators included in this section do extract globally useful information, such as:

- *Pixel Value Distribution* (p.105), the number of pixels having each value.
- *Classified Pixels* (p.107), the category of scene entity that the corresponding pixel is taken from.
- *Connected Components* (p.114), the groups of pixels all of which have the same label or classification.

It is possible to represent some of this information as an image (*i.e.* an image whose pixels are the category or index number of the scene structure), but the information need not always be so represented, nor even necessarily be representable as an image. For example, while not included here, one might analyze a binary image (p.225) of an isolated object to determine the area or various moments of the shape. Another example would be to link together connected edge (p.230) fragments to make a list of edge pixels.

This category of operation is often considered part of the middle level image interpretation (*i.e.* a signal-to-symbol transformations or feature extraction), and whose results might ultimately be used in higher level image interpretation (*i.e.* symbol-to-symbol transformations such as scene description, object location, *etc.*).

8.1 Intensity Histogram

Brief Description

In an image processing context, the histogram of an image normally refers to a histogram of the pixel intensity values (p.239). This histogram is a graph showing the number of pixels (p.238) in an image at each different intensity value found in that image. For an 8-bit grayscale image (p.232) there are 256 different possible intensities, and so the histogram will graphically display 256 numbers showing the distribution of pixels amongst those grayscale values. Histograms can also be taken of color images (p.225) — either individual histograms of red, green and blue channels can be taken, or a 3-D histogram can be produced, with the three axes representing the red, blue and green (p.240) channels, and brightness at each point representing the pixel count. The exact output from the operation depends upon the implementation — it may simply be a picture of the required histogram in a suitable image format, or it may be a data file of some sort representing the histogram statistics.

How It Works

The operation is very simple. The image is scanned in a single pass and a running count of the number of pixels found at each intensity value is kept. This is then used to construct a suitable histogram.

Guidelines for Use

Histograms have many uses. One of the more common is to decide what value of threshold to use when converting a grayscale image (p.232) to a binary (p.225) one by thresholding (p.69). If the image is suitable for thresholding then the histogram will be *bi-modal* — *i.e.* the pixel intensities will be clustered around two well-separated values. A suitable threshold for separating these two groups will be found somewhere in between the two peaks in the histogram. If the distribution is not like this then it is unlikely that a good segmentation can be produced by thresholding.

The intensity histogram for the input image `wdg2` is `wdg2hst1`. The object being viewed is dark in color and it is placed on a light background, and so the histogram exhibits a good bi-modal distribution. One peak represents the object pixels, one represents the background. The histogram `wdg2hst3` is the same, but with the *y*-axis expanded to show more detail. It is clear that a threshold value of around 120 should segment the picture nicely, as can be seen in `wdg2thr2`.

The histogram of image `wdg3` is `wdg3hst1`. This time there is a significant incident illumination gradient across the image, and this blurs out the histogram. The bi-modal distribution has been destroyed and it is no longer possible to select a single global threshold that will neatly segment the object from its background. Two failed thresholding segmentations are shown in `wdg3thr1` and `wdg3thr2` using thresholds of 80 and 120, respectively.

It is often helpful to be able to adjust the scale on the *y*-axis of the histogram manually. If the scaling is simply done automatically, then very large peaks may force a scale that makes smaller features indiscernible.

The histogram is used and altered by many image enhancement operators. Two operators which are closely connected to the histogram are contrast stretching (p.75) and histogram equalization (p.78). They are based on the assumption that an image has to use the full intensity range to display the maximum contrast. Contrast stretching takes an image in which the intensity values don't span the full intensity range and stretches its values linearly. This can be illustrated with `cla3`. Its histogram, `cla3hst1` shows that most of the pixels have rather high intensity values. Contrast stretching the image yields `cla3str1` which has a clearly improved contrast. The corresponding histogram is `cla3hst2`. If we expand the *y*-axis, as was done in `cla3hst3`, we can see that now the pixel values are distributed over the entire intensity range. Due to the discrete character of

the pixel values, we can't increase the number of distinct intensity values. That is the reason why the stretched histogram shows the gaps between the single values.

The image `wom2` also has low contrast. However, if we look at its histogram, `wom2hst1`, we see that the entire intensity range is used and we therefore cannot apply contrast stretching. On the other hand, the histogram also shows that most of the pixels values are clustered in a rather small area, whereas the top half of the intensity values is used by only a few pixels. The idea of histogram equalization (p.78) is that the pixels should be distributed evenly over the whole intensity range, *i.e.* the aim is to transform the image so that the output image has a *flat* histogram. The image `wom2heq1` results from the histogram equalization and `wom2hst2` is the corresponding histogram. Due to the discrete character of the intensity values, the histogram is not entirely flat. However, the values are much more evenly distributed than in the original histogram and the contrast in the image was essentially increased.

Exercises

1. Suppose that you had a scene of three objects of different distinct intensities against an extremely bright background. What would the corresponding histogram look like?
2. How could you get a program to automatically work out the ideal threshold for an image from its histogram? What do you think might be the problems?
3. If there is a very high peak right at the top end of the histogram, what does this suggest?

References

R. Boyle and R. Thomas *Computer Vision: A First Course*, Blackwell Scientific Publications, 1988, Chap. 4.

E. Davies *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 4.

A. Marion *An Introduction to Image Processing*, Chapman and Hall, 1991, Chap. 5.

D. Vernon *Machine Vision*, Prentice-Hall, 1991, p 49.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

8.2 Classification

Brief Description

Classification includes a broad range of decision-theoretic approaches to the identification of images (or parts thereof). All classification algorithms are based on the assumption that the image in question depicts one or more features (*e.g.*, geometric parts in the case of a manufacturing classification system, or spectral regions in the case of remote sensing, as shown in the examples below) and that each of these features belongs to one of several distinct and exclusive classes. The classes may be specified *a priori* by an analyst (as in *supervised classification*) or automatically clustered (*i.e.* as in *unsupervised classification*) into sets of prototype classes, where the analyst merely specifies the number of desired categories. (Classification and *segmentation* have closely related objectives, as the former is another form of component labeling (p.114) that can result in segmentation of various features in a scene.)

How It Works

Image classification analyzes the numerical properties of various image features and organizes data into categories. Classification algorithms typically employ two phases of processing: *training* and *testing*. In the initial training phase, characteristic properties of typical image features are isolated and, based on these, a unique description of each classification category, *i.e.* *training class*, is created. In the subsequent testing phase, these feature-space partitions are used to classify image features.

The description of training classes is an extremely important component of the classification process. In supervised classification, *statistical* processes (*i.e.* based on an *a priori* knowledge of probability distribution functions) or *distribution-free* processes can be used to extract class descriptors. Unsupervised classification relies on *clustering* algorithms to automatically segment the training data into prototype classes. In either case, the motivating criteria for constructing training classes is that they are:

- *independent*, *i.e.* a change in the description of one training class should not change the value of another,
- *discriminatory*, *i.e.* different image features should have significantly different descriptions, and
- *reliable*, all image features within a training group should share the common definitive descriptions of that group.

A convenient way of building a parametric description of this sort is via a feature vector (v_1, v_2, \dots, v_n) , where n is the number of attributes which describe each image feature and training class. This representation allows us to consider each image feature as occupying a point, and each training class as occupying a sub-space (*i.e.* a representative point surrounded by some spread, or deviation), within the n -dimensional classification space. Viewed as such, the classification problem is that of determining to which sub-space class each feature vector belongs.

For example, consider an application where we must distinguish two different types of objects (*e.g.* bolts and sewing needles) based upon a set of two attribute classes (*e.g.* *length* along the major axis and *head diameter*). If we assume that we have a vision system capable of extracting these features from a set of training images, we can plot the result in the 2-D feature space, shown in Figure 8.1.

At this point, we must decide how to numerically partition the feature space so that if we are given the feature vector of a test object, we can determine, quantitatively, to which of the two classes it belongs. One of the most simple (although not the most computationally efficient) techniques is to employ a supervised, distribution-free approach known as the *minimum (mean) distance classifier*. This technique is described below.

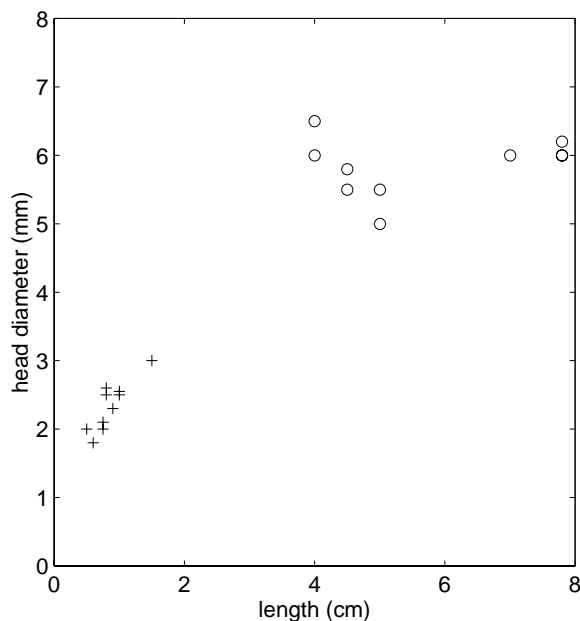


Figure 8.1: Feature space: + sewing needles, o bolts.

Minimum (Mean) Distance Classifier

Suppose that each training class is represented by a prototype (or *mean*) vector:

$$m_j = 1/N_j \sum_{x \in \omega_j} x \text{ for } j = 1, 2, \dots, M$$

where N_j is the number of training pattern vectors from class ω_j . In the example classification problem given above, $m_{needle} = [0.86, 2.34]^T$ and $m_{bolt} = [5.74, 5.85]^T$ as shown in Figure 8.2.

Based on this, we can assign any given pattern x to the class of its closest prototype by determining its proximity to each m_j . If Euclidean distance (p.229) is our measure of proximity, then the distance to the prototype is given by

$$D_j(x) = \|x - m_j\| \text{ for } j = 1, 2, \dots, M$$

It is not difficult to show that this is equivalent to computing

$$d_j(x) = x^T m_j - 1/2(m_j^T m_j) \text{ for } j = 1, 2, \dots, M$$

and assign x to class ω_j if $d_j(x)$ yields the largest value.

Returning to our example, we can calculate the following decision functions:

$$d_{needle}(x) = 0.86x_1 + 2.34x_2 - 3.10$$

$$d_{bolt}(x) = 5.74x_1 + 5.85x_2 - 33.59$$

Finally, the *decision boundary* which separates class ω_i from ω_j is given by values for x for which

$$d_i(x) - d_j(x) = 0$$

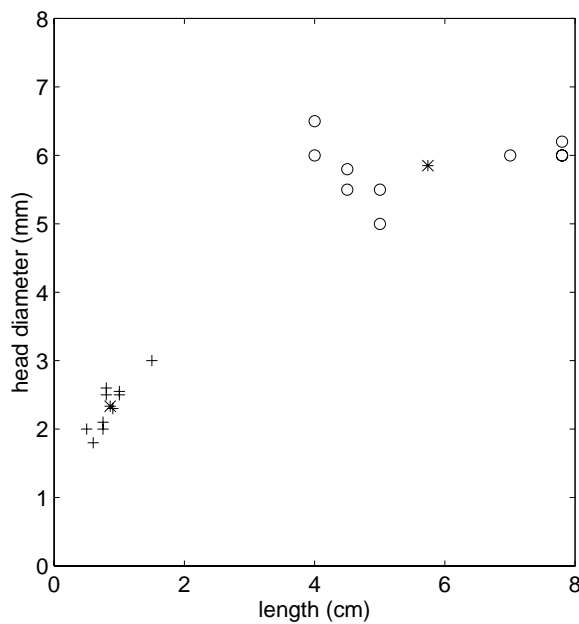


Figure 8.2: Feature space: + sewing needles, o bolts, * class mean

In the case of the needles and bolts problem, the decision surface is given by:

$$d_{needle/bolt}(x) = -4.88x_1 - 3.51x_2 + 30.49 = 0$$

As shown in Figure 8.3, the surface defined by this decision boundary is the perpendicular bisector of the line segment joining m_i and m_j .

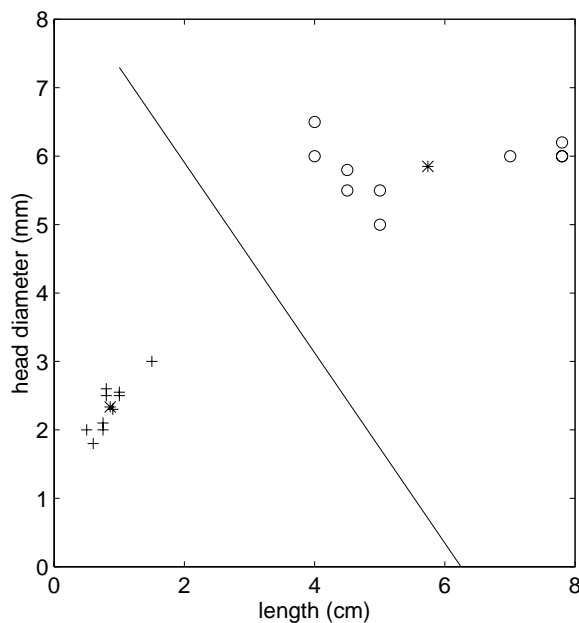


Figure 8.3: Feature space: + sewing needles, o bolts, * class mean, line = decision surface

In practice, the minimum (mean) distance classifier works well when the distance between means

is large compared to the spread (or randomness) of each class with respect to its mean. It is simple to implement and is guaranteed to give an error rate within a factor of two of the ideal error rate, obtainable with the statistical, supervised *Bayes' classifier*. The Bayes' classifier is a more informed algorithm as the frequencies of occurrence of the features of interest are used to aid the classification process. Without this information the minimum (mean) distance classifier can yield biased classifications. This can be best combatted by applying training patterns at the natural rates at which they arise in the raw training set.

Guidelines for Use

To illustrate the utility of classification (using the minimum (mean) distance classifier), we will consider a remote sensing application. Here, we have a collection of multi-spectral images (p.237) (*i.e.* images containing several bands, where each band represents a single electro-magnetic wavelength or frequency) of the planet Earth collected from a satellite. We wish to classify each image pixel (p.238) into one of several different classes (*e.g.* water, city, wheat field, pine forest, cloud, *etc.*) on the basis of the spectral measurement of that pixel.

Consider a set of images of the globe (centered on America) which describe the visible `bvs1` and infra-red `bir1` spectrums, respectively. From the histograms of the visible band image `bvs1hst1` and infra-red band image `bir1hst1`, we can see that it would be very difficult to find a threshold (p.69), or decision surface, with which to segment the images into training classes (*e.g.* spectral classes which correspond to physical phenomena such as cloud, ground, water, *etc.*). It is often the case that having a higher dimensionality representation of this information (*i.e.* using one 2-D histogram instead of two 1-D histograms) facilitates segmentation of regions which might overlap when projected onto a single axis, as shown for some hypothetical data in Figure 8.4.

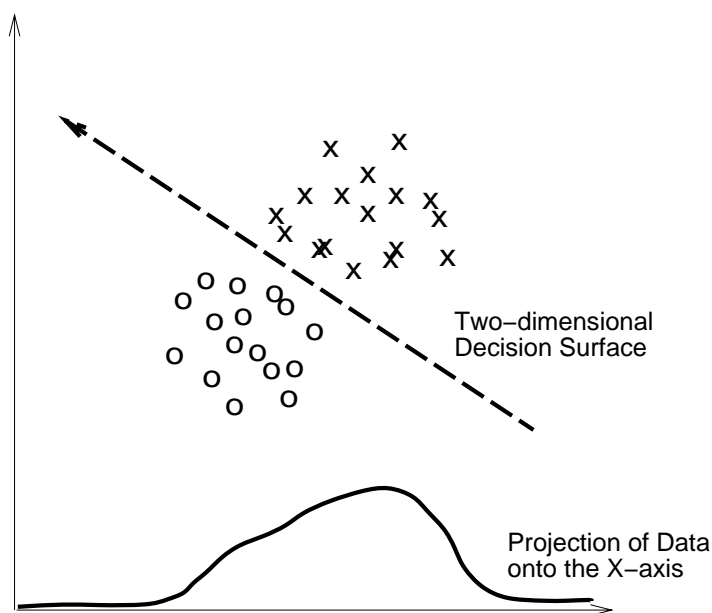


Figure 8.4: 2-D feature space representation of hypothetical data. (The projection of the data onto the X-axis is equivalent to a 1-D histogram.)

Since the images over America are registered, we can combine them into a single two-band image and find the decision surface(s) which divides the data into distinct classification regions in this higher dimensional representation. To this aim, we use a *k-means algorithm* to find the training classes of the 2-D spectral images. (This algorithm converts an input image into vectors of equal size (where the size of each vector is determined by the number of spectral bands in the input image) and then determines the *k* prototype mean vectors by minimizing of the sum of the squared

distances from all points in a class to the class center m_i .)

If we choose $k=2$ as a starting point, the algorithm finds two prototype mean vectors, shown with a * symbol in the 2-D histogram `bvi1tdh1`. This figure also shows the linear decision surface which separates out our training classes.

Using two training classes, such as those found for the image over America, we can classify a similar multi-spectral image of Africa `avs1` (visible) and `air1` (infra-red) to yield the result: `avi2cls1`. (Note that the image size has been scaled (p.90) by a factor of two to speed up computation, and a border has been placed around the image to mask out any background pixels.) We can see that one of the classes created during the training process contains pixels corresponding to land masses over north and south Africa, whereas the pixels in the other class represent water or clouds.

Classification accuracy using the minimum (mean) distance classifier improves as we increase the number of training classes. The images `avi2cls4` and `avi2cls5` show the results of the classification procedure using $k=4$ and $k=6$ training classes. The equivalent with a color assigned to each class is shown in `avi2cls2` and `avi2cls3` for $k=4$ and $k=6$, respectively. Here we begin to see the classification segmenting out regions which correspond to distinct physical phenomena.

Common Variants

Classification is such a broad ranging field, that a description of all the algorithms could fill several volumes of text. We have already discussed a common supervised algorithm, therefore in this section we will briefly consider a representative unsupervised algorithm. In general, unsupervised clustering techniques are used less frequently, as the computation time required for the algorithm to learn a set of training classes is usually prohibitive. However, in applications where the features (and relationships between features) are not well understood, clustering algorithms can provide a viable means for partitioning a sample space.

A general clustering algorithm is based on a *split* and *merge* technique, as shown in Figure 8.5. Using a *similarity measure* (e.g. the dot product of two vectors, the weighted Euclidean distance, etc.), the input vectors can be partitioned into subsets, each of which should be sufficiently distinct. Subsets which do not meet this criterion are merged. This procedure is repeated on all of the subsets until no further splitting of subsets occurs or until some *stopping criteria* is met.

Exercises

1. In the classification of natural scenes, there is often the problem that features we want to classify occur at different scales. For example, in constructing a system to classify trees, we have to take into account that trees close to the camera will appear large and sharp, while those at some distance away may be small and fuzzy. Describe how one might overcome this problem.
2. The following table gives some training data to be used in the classification of flower types. Petal length and width are given for two different flowers. Plot this information on a graph (utilizing the same scale for the *petal length* and *petal width* axes) and then answer the questions below.

Petal Length	Petal Width	Class
4	3	1
4.5	4	1
3	4	1
6	1	2
7	1.5	2
6.5	2	2

- a) Calculate the mean, or prototype, vectors m_i for the two flower types described above.
- b) Determine the decision functions d_i for each class. c) Determine the equation of the boundary (i.e. $d_{12} = d_1(x) - d_2(x)$) and plot the decision surface on your graph. d) Notice

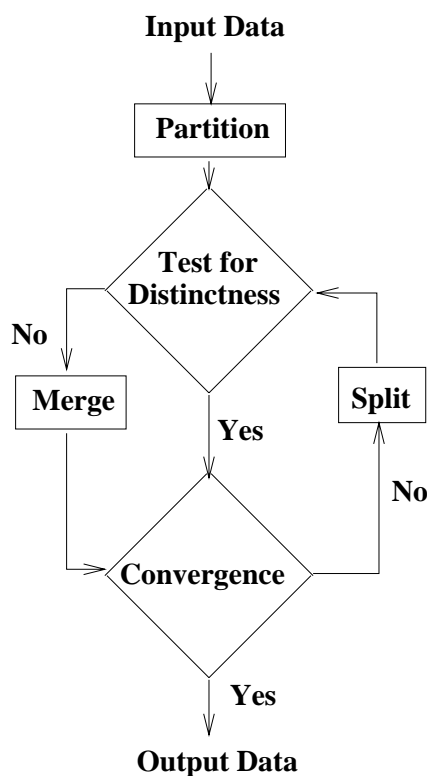


Figure 8.5: General clustering algorithm

that substitution of a pattern from class ω_1 into your answer from the previous section yields a positive valued $d_{12}(x)$, while a pattern belonging to the class ω_2 yields a negative value. How would you use this information to determine a new pattern's class membership?

3. Experiment with classifying some remotely sensed images: *e.g.* `evs1` and `eir1` are the visible and infra-red images of Europe, `uvs1` and `uir1` are those of the United Kingdom and `svs1` and `sir1` are those of Scandinavia. Begin by combining the two single-band spectral images of Europe into a single multi-band image (p.237). (You may want to scale (p.90) the image so as to cut down the processing time.) Then, create a set of training classes, where k equals 6,8,10... (Remember that although the accuracy of the classification improves with greater numbers of training classes, the computational requirements increase as well.) Then try classifying all three images using these training sets.

References

- T. Avery and G. Berlin** *Fundamentals of Remote Sensing and Airphoto Interpretation*, Maxwell Macmillan International, 1985, Chap. 15.
- D. Ballard and C. Brown** *Computer Vision*, Prentice-Hall, Inc., 1982, Chap. 6.
- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 18.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986, Chap. 9.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, Chap. 6.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

8.3 Connected Components Labeling

Brief Description

Connected components labeling scans an image and groups its pixels (p.238) into components based on pixel connectivity (p.238), *i.e.* all pixels in a connected component share similar pixel intensity values (p.239) and are in some way connected with each other. Once all groups have been determined, each pixel is labeled with a graylevel or a color (color labeling) according to the component it was assigned to.

Extracting and labeling of various disjoint and connected components in an image is central to many automated image analysis applications.

How It Works

Connected component labeling works by scanning an image, pixel-by-pixel (from top to bottom and left to right) in order to identify connected pixel regions, *i.e.* regions of adjacent pixels which share the same set of intensity values V . (For a binary image $V=\{1\}$; however, in a graylevel image V will take on a range of values, for example: $V=\{51, 52, 53, \dots, 77, 78, 79, 80\}$.)

Connected component labeling works on binary (p.225) or graylevel images (p.232) and different measures of connectivity (p.238) are possible. However, for the following we assume binary input images and δ -connectivity. The connected components labeling operator scans the image by moving along a row until it comes to a point p (where p denotes the pixel to be labeled at any stage in the scanning process) for which $V=\{1\}$. When this is true, it examines the four neighbors of p which have already been encountered in the scan (*i.e.* the neighbors (i) to the left of p , (ii) above it, and (iii and iv) the two upper diagonal terms). Based on this information, the labeling of p occurs as follows:

- If all four neighbors are 0, assign a new label to p , else
- if only one neighbor has $V=\{1\}$, assign its label to p , else
- if one or more of the neighbors have $V=\{1\}$, assign one of the labels to p and make a note of the equivalences.

After completing the scan, the equivalent label pairs are sorted into equivalence classes and a unique label is assigned to each class. As a final step, a second scan is made through the image, during which each label is replaced by the label assigned to its equivalence classes. For display, the labels might be different graylevels or colors.

Guidelines for Use

To illustrate connected components labeling, we start with a simple binary image (p.225) containing some distinct artificial objects `art8`. After scanning this image and labeling the distinct pixels classes with a different grayvalue, we obtain the labeled output image `art8lab1`. Note that this image was scaled (p.48), since the initial grayvalues ($1 - 8$) would all appear black on the screen. However, the pixels initially assigned to the lower classes (1 and 2) are still indiscernible from the background. If we assign a distinct color to each graylevel we obtain `art8lab2`.

The full utility of connected components labeling can be realized in an image analysis scenario wherein images are pre-processed via some segmentation (*e.g.* thresholding (p.69)) or classification (p.107) scheme. One application is to use connected components labeling to count the objects in an image. For example, in the above simple scene the δ objects yield δ different classes.

If we want to count the objects in a real world scene like `c1c3`, we first have to threshold the image in order to produce a binary input image (the implementation being used only takes binary

images). Setting all values above a value of *150* to zero yields `c1c3thr1`. The white dots correspond to the black, dead cells in the original image. The connected components of this binary image can be seen in `c1c3lab1`. The corresponding colormap (p.235) shows that the highest value is *163*, *i.e.* the output contains *163* connected components. In order to better see the result, we now would like to assign a color to each component. The problem here is that we cannot find *163* colors where each of them is different enough from all others to be distinguished by the human eye. Two possible ways to assign the colors are as follows:

- We only use a few colors (*e.g.* *8*) which are clearly different from each other and assign each graylevel of the connected component image to one of these colors. The result can be seen in `c1c3lab2`. We can now easily distinguish two different components, provided that they were not assign the same color. However, we lose a lot of information because the (in this case) *163* graylevels are reduced to *8* different colors.
- We can assign a different color to each grayvalue, many of them being quite similar. A typical result for the above image would be `c1c3lab3`. Although, we sometimes cannot tell the border between two components when they are very close to each other, we do not lose any information.

More sophisticated techniques combine both methods. They make sure that two nearby components always have distinct colors by taking into account the colors of the neighbors of the component which is going to be assigned a color.

If we compare the above color-labeled images with the original, we can see that the number of components is quite close to the number of dead cells in the image. However, we obtain a small difference since some cells merge together into one component or dead cells are suppressed by the threshold.

We encounter greater problems when trying to count the number of turkeys in `tur1gry1`. Labeling the thresholded image `tur1thr1` yields `tur1lab1` as a graylevel or `tur1lab2` as a color labeled version. Although we managed to assign one connected component to each turkey, the number of components (*196*) does not correspond to the number of turkeys.

The two last examples showed that the connected component labeling is the easy part of the automated analysis process, whereas the major task is to obtain a good binary image which separates the objects from the background.

Finally, we consider the problem of labeling data output from a classification (p.107) processes. We can classify multi-spectral images (p.237) (*e.g.* a two-band image consisting of `avs2` (visible range) and `air2` (infra-red range)) in order to find *k* groupings of the data based on the pixel intensities clusters. This result is shown in the image `avi2cls1`, where the multi-spectral image was classified into two groups. If we now apply connected components labeling, connected geographic regions which belong to the same intensity classes can be labeled. The result contains 49 different components, most of them being only a few pixels large. The color labeled version can be seen in `avi2lab1`. One could now use this image to further investigate the regions, *e.g.* if some components changed their size compared to a reference image or if other regions merged together.

Common Variants

A collection of morphological operators (p.117) exists for extracting connected components and labeling them in various ways. A simple method for extracting connected components of an image combines dilation (p.118) and the mathematical intersection operation. The former identifies pixels which are part of a continuous region sharing a common set of intensity values $V=\{I\}$ and the latter eliminates dilations centered on pixels with $V=\{0\}$. The structuring element (p.241) used defines the desired connectivity.

More sophisticated variants of this include a set of *geodesic* functions for measuring the exact shape of distinct objects in an image. These operators are based on the notion of geodesic distance *d* which is defined as the shortest distance (p.229) between two points located within an image object

such that the entire path between the points is included in the object. One way to obtain this information is to apply a series of dilations of size 1. (Distance (p.206) measuring operators are described in fuller detail elsewhere.)

For example, consider the image `tol1crp1` which shows a triangular block. Applying a geodesic operator to the image produces a labeled image `b1d1lab1` wherein the graylevel intensity labeling across the surface of the block encodes geodesic distance, *i.e.* light pixels represent larger distances.

Exercises

1. How would the scanning algorithm described above label an object containing a hole? How would the morphological approach? Investigate how your implementation handles the image `cir1`.
2. Apply connected components labeling to an image counting problem. Starting from `pen1`, produce a suitable binary image (p.225) (*i.e.* threshold (p.69) the image) and then apply connected components labeling with the aim of obtaining a distinct label for each penguin. (Note, this may require some experimentation with threshold values.)
3. The remote sensing example given in the test used a rather convoluted set of operations (e.g., classification (p.107), thresholding (p.69) and then labeling). See if you can obtain similar results by simply thresholding one of the original images, such as `avs1` and/or `air1`, and then applying labeling directly.
4. Classifying (p.107) the above two-band satellite image of Africa using the classification parameter $k=4$ yields `avi2c1s5`. Use labeling to identify each connected component in this image. If your implementation of the operator does not support graylevel images use thresholding (p.69) to produce four binary images, each containing one of the four classes. Then apply connected component labeling to each of the binary images.
5. Try using thresholding and connected components analysis to segment the image `aer1` into urban and rural areas. You might investigate thresholding within a particular color band(s) to create two binary files containing a description of (i) rural areas, *i.e.* fields, trees, hills, *etc.* around the image perimeter and (ii) urban areas, *i.e.* roads, houses, *etc.* in the image interior.

References

- T. Avery and G. Berlin** *Fundamentals of Remote Sensing and Airphoto Interpretation*, Maxwell Macmillan International, 1985, Chap. 15.
- D. Ballard and C. Brown** *Computer Vision*, Prentice-Hall, 1982, Chap. 2.
- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 6.
- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, Chap. 2.
- B. Horn** *Robot Vision*, MIT Press, 1986, Chap. 4.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, pp 34 - 36.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

Chapter 9

Morphology

Morphological operators often take a binary image (p.225) and a structuring element (p.241) as input and combine them using a set operator (intersection, union, inclusion, complement). They process objects in the input image based on characteristics of its shape, which are encoded in the structuring element. The mathematical details are explained in *Mathematical Morphology* (p.236).

Usually, the structuring element is sized 3×3 and has its origin at the center pixel. It is shifted over the image and at each pixel of the image its elements are compared with the set of the underlying pixels. If the two sets of elements match the condition defined by the set operator (*e.g.* if the set of pixels in the structuring element is a subset of the underlying image pixels), the pixel underneath the origin of the structuring element is set to a pre-defined value (0 or 1 for binary images). A morphological operator is therefore defined by its structuring element and the applied set operator.

For the basic morphological operators the structuring element (p.241) contains only foreground pixels (*i.e.* ones) and 'don't care's'. These operators, which are all a combination of erosion (p.123) and dilation (p.118), are often used to select or suppress features of a certain shape, *e.g.* removing noise from images or selecting objects with a particular direction.

The more sophisticated operators take zeros as well as ones and 'don't care's' in the structuring element. The most general operator is the hit and miss (p.133), in fact, all the other morphological operators can be deduced from it. Its variations are often used to simplify the representation of objects in a (binary) image while preserving their structure, *e.g.* producing a skeleton of an object using skeletonization (p.145) and tidying up the result using thinning (p.137).

Morphological operators can also be applied to graylevel images (p.232), *e.g.* to reduce noise (p.221) or to brighten the image. However, for many applications, other methods like a more general spatial filter (p.148) produces better results.

9.1 Dilation

Brief Description

Dilation is one of the two basic operators in the area of mathematical morphology (p.236), the other being erosion (p.123). It is typically applied to binary images (p.225), but there are versions that work on grayscale images (p.232). The basic effect of the operator on a binary image is to gradually enlarge the boundaries of regions of foreground pixels (p.238) (*i.e.* white pixels, typically). Thus areas of foreground pixels grow in size while holes within those regions become smaller.

How It Works

Useful background to this description is given in the mathematical morphology (p.236) section of the Glossary.

The dilation operator takes two pieces of data as inputs. The first is the image which is to be dilated. The second is a (usually small) set of coordinate points known as a structuring element (p.241) (also known as a *kernel* (p.233)). It is this structuring element that determines the precise effect of the dilation on the input image.

The mathematical definition of dilation for *binary* images is as follows:

Suppose that X is the set of Euclidean coordinates corresponding to the input binary image, and that K is the set of coordinates for the structuring element.

Let Kx denote the translation of K so that its origin is at x .

Then the dilation of X by K is simply the set of all points x such that the intersection of Kx with X is non-empty.

The mathematical definition of grayscale dilation is identical except for the way in which the set of coordinates associated with the input image is derived. In addition, these coordinates are 3-D rather than 2-D.

As an example of binary dilation, suppose that the structuring element is a 3×3 square, with the origin at its center, as shown in Figure 9.1. Note that in this and subsequent diagrams, foreground pixels are represented by 1's and background pixels by 0's.

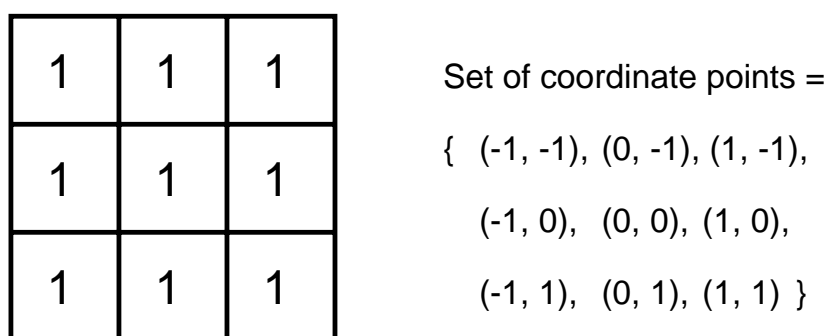


Figure 9.1: A 3×3 square structuring element

To compute the dilation of a binary input image by this structuring element, we consider each of the *background* pixels in the input image in turn. For each background pixel (which we will call the *input pixel*) we superimpose the structuring element on top of the input image so that the origin of the structuring element coincides with the input pixel position. If *at least one* pixel in the structuring element coincides with a foreground pixel in the image underneath, then the input

pixel is set to the foreground value. If all the corresponding pixels in the image are background, however, the input pixel is left at the background value.

For our example 3×3 structuring element, the effect of this operation is to set to the foreground color any background pixels that have a neighboring foreground pixel (assuming 8-connectedness (p.238)). Such pixels must lie at the edges of white regions, and so the practical upshot is that foreground regions grow (and holes inside a region shrink).

Dilation is the *dual* of erosion (p.123) *i.e.* dilating foreground pixels is equivalent to eroding the background pixels.

Guidelines for Use

Most implementations of this operator expect the input image to be binary, usually with foreground pixels at pixel value 255, and background pixels at pixel value 0. Such an image can often be produced from a grayscale image using thresholding (p.69). It is important to check that the polarity (p.225) of the input image is set up correctly for the dilation implementation being used.

The structuring element may have to be supplied as a small binary image, or in a special matrix format, or it may simply be hardwired into the implementation, and not require specifying at all. In this latter case, a 3×3 square structuring element is normally assumed which gives the expansion effect described above. The effect of a dilation using this structuring element on a binary image is shown in Figure 9.2.

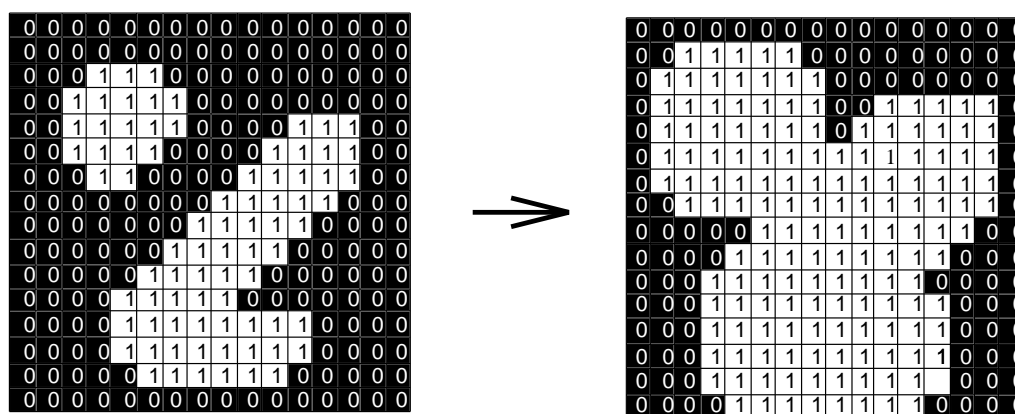


Figure 9.2: Effect of dilation using a 3×3 square structuring element

The 3×3 square is probably the most common structuring element used in dilation operations, but others can be used. A larger structuring element produces a more extreme dilation effect, although usually very similar effects can be achieved by repeated dilations using a smaller but similarly shaped structuring element. With larger structuring elements, it is quite common to use an approximately disk shaped structuring element, as opposed to a square one.

The image `wdg2thr3` shows a thresholded (p.69) image of `wdg2`. The basic effect of dilation on the binary is illustrated in `wdg2dil1`. This image was produced by two dilation passes using a disk shaped structuring element of 11 pixels radius. Note that the corners have been rounded off. In general, when dilating by a disk shaped structuring element, convex boundaries will become rounded, and concave boundaries will be preserved as they are.

Dilations can be made directional by using less symmetrical structuring elements. *e.g.* a structuring element that is 10 pixels wide and 1 pixel high will dilate in a horizontal direction only. Similarly, a 3×3 square structuring element with the origin in the middle of the top row rather than the center, will dilate the bottom of a region more strongly than the top.

Grayscale dilation with a flat disk shaped structuring element will generally brighten the image.

Bright regions surrounded by dark regions grow in size, and dark regions surrounded by bright regions shrink in size. Small dark spots in images will disappear as they are ‘filled in’ to the surrounding intensity value. Small bright spots will become larger spots. The effect is most marked at places in the image where the intensity changes rapidly and regions of fairly uniform intensity will be largely unchanged except at their edges. Figure 9.3 shows a vertical cross-section through a graylevel image and the effect of dilation using a disk shaped structuring element.

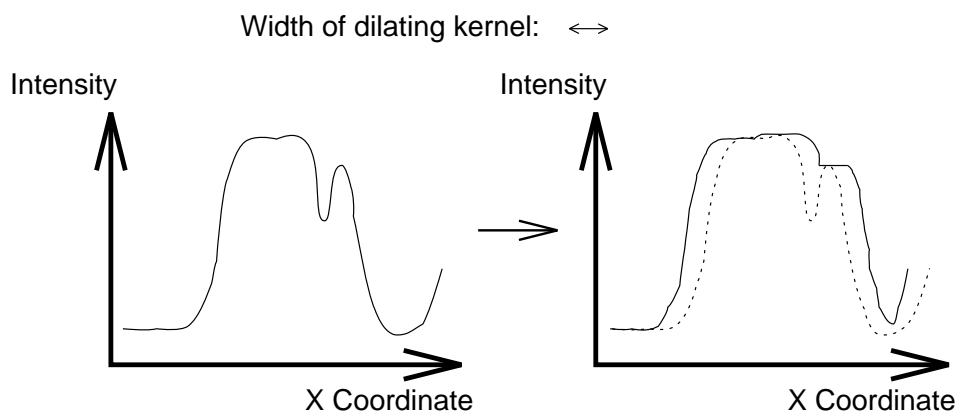


Figure 9.3: Graylevel dilation using a disk shaped structuring element. The graphs show a vertical cross-section through a graylevel image (p.232).

The image `blb1di11` shows the basic effects of graylevel dilation. This was produced from `blb1` by two erosion passes using a 3×3 flat square structuring element. The highlights on the bulb surface have increased in size and have also become squared off as an artifact of the structuring element shape. The dark body of the cube has shrunk in size since it is darker than its surroundings, while within the outlines of the cube itself, the darkest top surface has shrunk the most. Many of the surfaces have a more uniform intensity since dark spots have been filled in by the dilation. The effect of five passes of the same dilation operator on the original image is shown in `blb1di12`.

There are many specialist uses for dilation. For instance it can be used to fill in small spurious holes (‘pepper noise’ (p.221)) in images. The image `fce5noi2` shows an image containing pepper noise, and `fce5di11` shows the result of dilating this image with a 3×3 square structuring element. Note that although the noise has been effectively removed, the image has been degraded significantly. Compare the result with that described under closing (p.130).

Dilation can also be used for edge detection (p.230) by taking the dilation of an image and then subtracting (p.45) away the original image, thus highlighting just those new pixels at the edges of objects that were added by the dilation. For example, starting with `wdg2thr3` again, we first dilate it using 3×3 square structuring element, and then subtract away the original image to leave just the edge of the object as shown in `wdg2ded1`.

Finally, dilation is also used as the basis for many other mathematical morphology operators, often in combination with some logical operators (p.234). A simple example is *region filling* which is illustrated using `reg1`. This image and all the following results were zoomed (p.90) with a factor of 16 for a better display, *i.e.* each pixel during the processing corresponds to a 16×16 pixel square in the displayed images. Region filling applies logical NOT (p.63), logical AND (p.55) and dilation iteratively. The process can be described by the following formula:

$$X_k = \text{dilate}(X_{k-1}, J) \cap A_{not}$$

where X_k is the region which after convergence fills the boundary, J is the structuring element and A_{not} is the negative of the boundary. This combination of the dilation operator and a logical operator is also known as *conditional dilation*.

Imagine that we know X_0 , *i.e.* one pixel which lies inside the region shown in the above image, *e.g.* `reg1fst1`. First, we dilate the image containing the single pixel using a structuring element as shown in Figure 9.1, resulting in `reg1dil1`. To prevent the growing region from crossing the boundary, we AND it with `reg1neg1` which is the negative of the boundary. Dilating the resulting image, `reg1and1`, yields `reg1dil2`. ANDing this image with the inverted boundary results in `reg1and2`. Repeating these two steps until convergence, yields `reg1and3`, `reg1and4`, `reg1and5`, `reg1and6` and finally `reg1and7`. ORing (p.58) this image with the initial boundary yields the final result, as can be seen in `reg1fill1`.

Many other morphological algorithms make use of dilation, and some of the most common ones are described here (p.117). An example in which dilation is used in combination with other morphological operators is the pre-processing for automated character recognition described in the thinning (p.137) section.

Exercises

1. What would be the effect of a dilation using the cross-shaped structuring element shown in Figure 9.4?

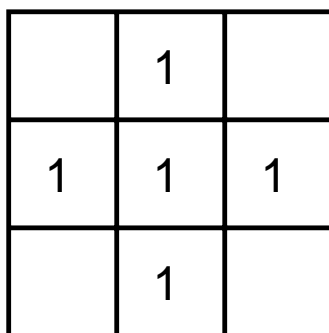


Figure 9.4: Cross-shaped structuring element

2. What would happen if the boundary shown in the region filling example is disconnected at one point? What could you do to fix that problem?
3. What would happen if the boundary in the region filling example is δ -connected (p.238)? What should the structuring element look like in this case?
4. How might you use conditional dilation to determine a connected component (p.238) given one point of this component?
5. What problems occur when using dilation to fill small noisy holes in objects?

References

- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 518 - 519, 549.
- R. Haralick and L. Shapiro** *Computer and Robot Vision*, Vol. 1, Chap. 5, Addison-Wesley Publishing Company, 1992.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986, p 384.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, pp 63 - 66, 76 - 78.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

9.2 Erosion

Brief Description

Erosion is one of the two basic operators in the area of mathematical morphology (p.236), the other being dilation (p.118). It is typically applied to binary images (p.225), but there are versions that work on grayscale images (p.232). The basic effect of the operator on a binary image is to erode away the boundaries of regions of foreground pixels (p.238) (*i.e.* white pixels, typically). Thus areas of foreground pixels shrink in size, and holes within those areas become larger.

How It Works

Useful background to this description is given in the mathematical morphology (p.236) section of the *Glossary*.

The erosion operator takes two pieces of data as inputs. The first is the image which is to be eroded. The second is a (usually small) set of coordinate points known as a structuring element (p.241) (also known as a *kernel* (p.233)). It is this structuring element that determines the precise effect of the erosion on the input image.

The mathematical definition of erosion for *binary* images is as follows:

Suppose that X is the set of Euclidean coordinates corresponding to the input binary image, and that K is the set of coordinates for the structuring element.

Let Kx denote the translation of K so that its origin is at x .

Then the erosion of X by K is simply the set of all points x such that Kx is a subset of X .

The mathematical definition for grayscale erosion is identical except in the way in which the set of coordinates associated with the input image is derived. In addition, these coordinates are 3-D rather than 2-D.

As an example of binary erosion, suppose that the structuring element is a 3×3 square, with the origin at its center as shown in Figure 9.5. Note that in this and subsequent diagrams, foreground pixels are represented by 1's and background pixels by 0's.

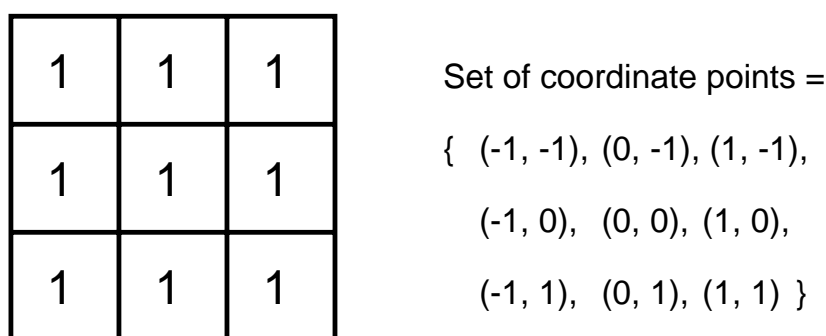


Figure 9.5: A 3×3 square structuring element

To compute the erosion of a binary input image by this structuring element, we consider each of the *foreground* pixels in the input image in turn. For each foreground pixel (which we will call the *input pixel*) we superimpose the structuring element on top of the input image so that the origin of the structuring element coincides with the input pixel coordinates. If for *every* pixel in the structuring element, the corresponding pixel in the image underneath is a foreground pixel,

then the input pixel is left as it is. If any of the corresponding pixels in the image are background, however, the input pixel is also set to background value.

For our example 3×3 structuring element, the effect of this operation is to remove any foreground pixel that is not completely surrounded by other white pixels (assuming 8-connectedness (p.238)). Such pixels must lie at the edges of white regions, and so the practical upshot is that foreground regions shrink (and holes inside a region grow).

Erosion is the *dual* of dilation (p.118), *i.e.* eroding foreground pixels is equivalent to dilating the background pixels.

Guidelines for Use

Most implementations of this operator will expect the input image to be binary, usually with foreground pixels at intensity value 255, and background pixels at intensity value 0. Such an image can often be produced from a grayscale image using thresholding (p.69). It is important to check that the polarity (p.225) of the input image is set up correctly for the erosion implementation being used.

The structuring element may have to be supplied as a small binary image, or in a special matrix format, or it may simply be hardwired into the implementation, and not require specifying at all. In this latter case, a 3×3 square structuring element is normally assumed which gives the shrinking effect described above. The effect of an erosion using this structuring element on a binary image is shown in Figure 9.6.

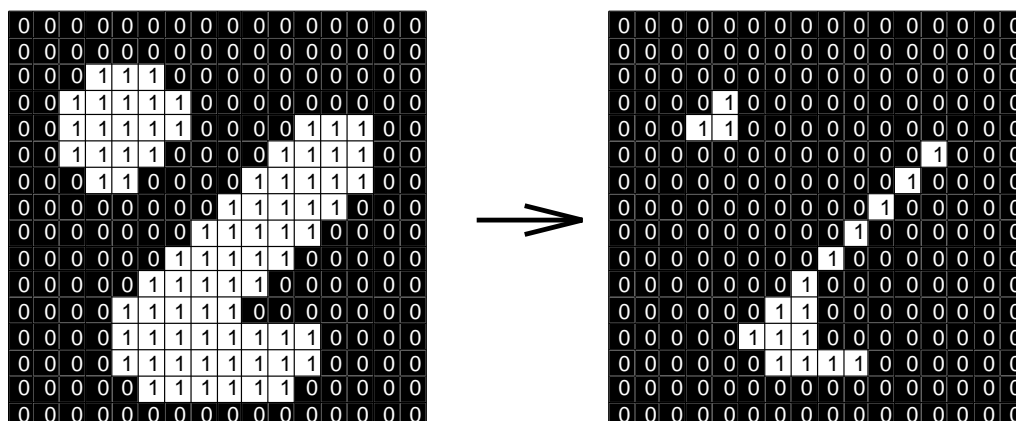


Figure 9.6: Effect of erosion using a 3×3 square structuring element

The 3×3 square is probably the most common structuring element used in erosion operations, but others can be used. A larger structuring element produces a more extreme erosion effect, although usually very similar effects can be achieved by repeated erosions using a smaller similarly shaped structuring element. With larger structuring elements, it is quite common to use an approximately disk shaped structuring element, as opposed to a square one.

The image `wdg2ero1` is the result of eroding `wdg2thr3` four times with a disk shaped structuring element 11 pixels in diameter. It shows that the hole in the middle of the image increases in size as the border shrinks. Note that the shape of the region has been quite well preserved due to the use of a disk shaped structuring element. In general, erosion using a disk shaped structuring element will tend to round concave boundaries, but will preserve the shape of convex boundaries.

Erosions can be made directional by using less symmetrical structuring elements. For example, a structuring element that is 10 pixels wide and 1 pixel high will erode in a horizontal direction only. Similarly, a 3×3 square structuring element with the origin in the middle of the top row rather than the center, will erode the bottom of a region more severely than the top.

Grayscale erosion with a flat disk shaped structuring element will generally darken the image. Bright regions surrounded by dark regions shrink in size, and dark regions surrounded by bright regions grow in size. Small bright spots in images will disappear as they are eroded away down to the surrounding intensity value, and small dark spots will become larger spots. The effect is most marked at places in the image where the intensity changes rapidly, and regions of fairly uniform intensity will be left more or less unchanged except at their edges. Figure 9.7 shows a vertical cross-section through a graylevel image and the effect of erosion using a disk shaped structuring element. Note that the flat disk shaped kernel causes small peaks in the image to disappear and valleys to become wider.

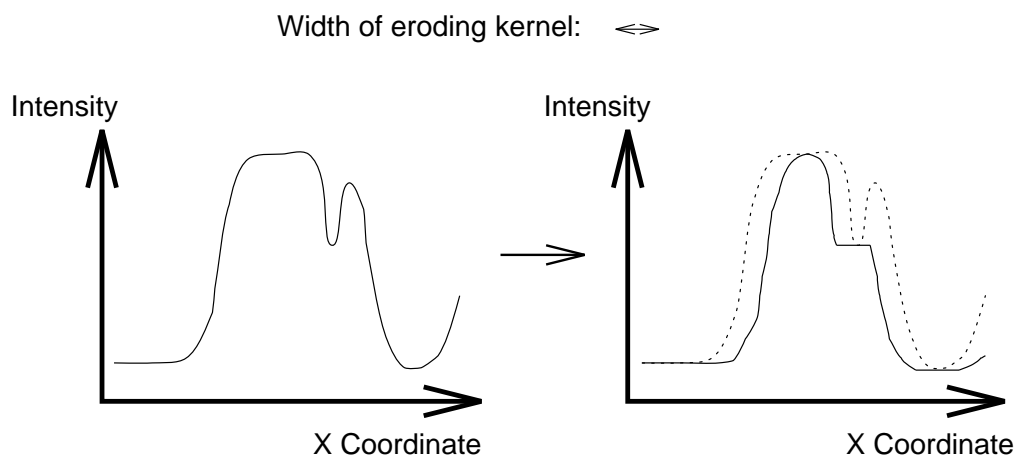


Figure 9.7: Graylevel erosion using a disk shaped structuring element. The graphs show a vertical cross-section through a graylevel image (p.232).

The image `blb1ero1` illustrates graylevel erosion. It was produced from `blb1` by two erosion passes using a 3×3 flat square structuring element. Note that the highlights have disappeared, and that many of the surfaces seem more uniform in appearance due to the elimination of bright spots. The body of the cube has grown in size since it is darker than its surroundings. The effect of five passes of the same erosion operator on the original image is shown in `blb1ero2`.

There are many specialist uses for erosion. One of the more common is to separate touching objects in a binary image so that they can be counted using a labeling algorithm (p.114). The image `mon1` shows a number of dark disks (coins in fact) silhouetted against a light background. The result of thresholding (p.69) the image at pixel value 90 yields `mon1thr1`. It is required to count the coins. However, this is not going to be easy since the touching coins form a single fused region of white, and a counting algorithm would have to first segment this region into separate coins before counting, a non-trivial task. The situation can be much simplified by eroding the image. The image `mon1ero1` shows the result of eroding twice using a disk shaped structuring element 11 pixels in diameter. All the coins have been separated neatly and the original shape of the coins has been largely preserved. At this stage a labeling algorithm (p.114) can be used to count the coins. The relative sizes of the coins can be used to distinguish the various types by, for example, measuring the area of each distinct region.

The image `mon1ero2` is derived from the same input picture, but a 9×9 square structuring element is used instead of a disk (the two structuring elements have approximately the same area). The coins have been clearly separated as before, but the square structuring element has led to distortion of the shapes, which in some situations could cause problems in identifying the regions after erosion.

Erosion can also be used to remove small spurious bright spots ('salt noise' (p.221)) in images. The image `fce5noi1` shows an image with salt noise, and `fce5ero1` shows the result of erosion with a 3×3 square structuring element. Note that although the noise has been removed, the rest of the image has been degraded significantly. Compare this with the same task using opening (p.127).

We can also use erosion for edge detection (p.230) by taking the erosion of an image and then subtracting (p.45) it away from the original image, thus highlighting just those pixels at the edges of objects that were removed by the erosion. An example of a very similar technique is given in the section dealing with dilation (p.119).

Finally, erosion is also used as the basis for many other mathematical morphology operators.

Exercises

1. What would be the effect of an erosion using the cross-shaped structuring element shown in Figure 9.8?

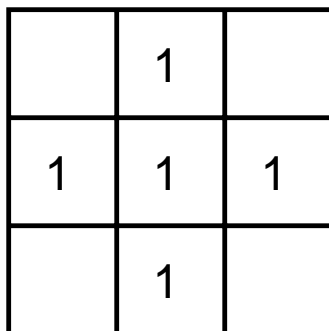


Figure 9.8: Cross-shaped structuring element

2. Is there any difference in the final result between applying a 3×3 square structuring element twice to an image, and applying a 5×5 square structuring element just once to the image? Which do you think would be faster and why?
3. When using large structuring elements, why does a disk shaped structuring element tend to preserve the shapes of convex objects better than a square structuring element?
4. Use erosion in the way described above to detect the edges of `wdg2thr3`. Is the result different to the one obtained with dilation (p.118)?

References

- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 518, 512, 550.
- R. Haralick and L. Shapiro** *Computer and Robot Vision*, Vol. 1, Chap. 5, Addison-Wesley Publishing Company, 1992.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986, p 384.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, pp 63 - 66, 76 - 78.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

9.3 Opening

Brief Description

Opening and closing (p.130) are two important operators from mathematical morphology (p.236). They are both derived from the fundamental operations of erosion (p.123) and dilation (p.118). Like those operators they are normally applied to binary images (p.225), although there are also graylevel (p.232) versions. The basic effect of an opening is somewhat like erosion in that it tends to remove some of the foreground (bright) pixels from the edges of regions of foreground pixels. However it is less destructive than erosion in general. As with other morphological operators, the exact operation is determined by a structuring element (p.241). The effect of the operator is to preserve *foreground* regions that have a similar shape to this structuring element, or that can completely contain the structuring element, while eliminating all other regions of foreground pixels.

How It Works

Very simply, an opening is defined as an erosion followed by a dilation *using the same structuring element for both operations*. See the sections on erosion (p.123) and dilation (p.118) for details of the individual steps. The opening operator therefore requires two inputs: an image to be opened, and a structuring element.

Graylevel opening consists simply of a graylevel erosion followed by a graylevel dilation.

Opening is the *dual* of closing, *i.e.* opening the foreground pixels with a particular structuring element is equivalent to closing the background pixels with the same element.

Guidelines for Use

While erosion can be used to eliminate small clumps of undesirable foreground pixels, *e.g.* ‘salt noise’ (p.221), quite effectively, it has the big disadvantage that it will affect *all* regions of foreground pixels indiscriminately. Opening gets around this by performing both an erosion and a dilation on the image. The effect of opening can be quite easily visualized. Imagine taking the structuring element and sliding it around *inside* each foreground region, without changing its orientation. All pixels which can be covered by the structuring element with the structuring element being entirely within the foreground region will be preserved. However, all foreground pixels which cannot be reached by the structuring element without parts of it moving out of the foreground region will be eroded away. After the opening has been carried out, the new boundaries of foreground regions will all be such that the structuring element fits inside them, and so further openings with the same element have no effect. The property is known as *idempotence* (p.233). The effect of an opening on a binary image using a 3×3 square structuring element is illustrated in Figure 9.9.

As with erosion and dilation, it is very common to use this 3×3 structuring element. The effect in the above figure is rather subtle since the structuring element is quite compact and so it fits into the foreground boundaries quite well even before the opening operation. To increase the effect, multiple erosions are often performed with this element followed by the same number of dilations. This effectively performs an opening with a larger square structuring element.

Consider `art3` which is a binary image containing a mixture of circles and lines. Suppose that we want to separate out the circles from the lines, so that they can be counted. Opening with a disk shaped structuring element 11 pixels in diameter gives `art3opn1`. Some of the circles are slightly distorted, but in general, the lines have been almost completely removed while the circles remain almost completely unaffected.

The image `art2` shows another binary image. Suppose that this time we wish to separately extract the horizontal and vertical lines. The result of an opening with a 3×9 vertically oriented structuring element is shown in `art2opn1`. The image `art2opn2` shows what happens if we use a

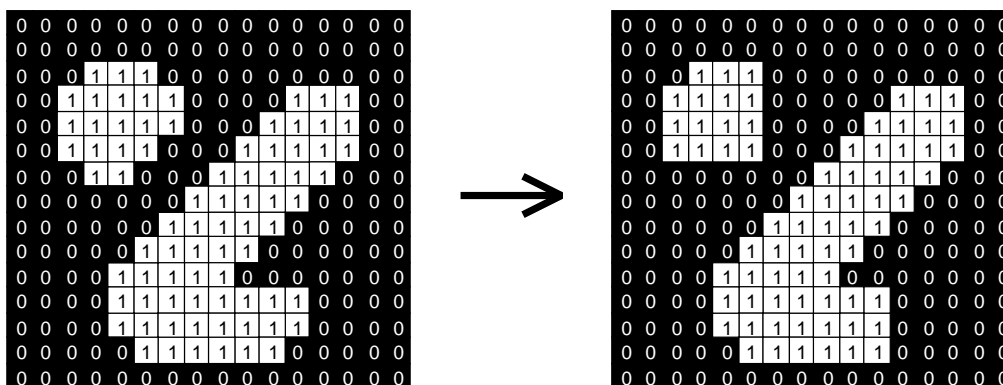


Figure 9.9: Effect of opening using a 3×3 square structuring element

9×3 horizontally oriented structuring element instead. Note that there are a few glitches in this last image where the diagonal lines cross vertical lines. These could easily be eliminated, however, using a slightly longer structuring element.

Unlike erosion and dilation, the position of the origin of the structuring element does not really matter for opening and closing the result is independent of it.

Graylevel opening can similarly be used to select and preserve particular intensity patterns while attenuating others. As a simple example we start with `ape1` and then perform graylevel opening with a flat 5×5 square structuring element to produce `ape1opn1`. The important thing to notice here is the way in which bright features smaller than the structuring element have been greatly reduced in intensity, while larger features have remained more or less unchanged in intensity. Thus the fine grained hair and whiskers in the image have been much reduced in intensity, while the nose region is still at much the same intensity as before. Note that the image does have a more matt appearance than before since the opening has eliminated small specularities and texture fluctuations.

Similarly, opening can be used to remove ‘salt noise’ (p.221) in images. The image `fce5noi1` shows an image containing salt noise, and `fce5opn1` shows the result of opening with a 3×3 square structuring element. The noise has been entirely removed with relatively little degradation of the underlying image. However, if the noise consists of dark points (*i.e.* ‘pepper noise’ (p.221)) as it can be seen in `fce5noi2`, graylevel opening yields `fce5opn2`. Here, no noise has been removed. At some places where two nearby noise pixels have merged into one larger point, the noise level has even been increased. In this case of ‘pepper noise’, graylevel closing (p.130) is a more appropriate operator.

As we have seen, opening can be very useful for separating out particularly shaped objects from the background, but it is far from being a universal 2-D object recognizer/segmenter. For instance if we try and use a long thin structuring element to locate, say, pencils in our image, any one such element will only find pencils at a particular orientation. If it is necessary to find pencils at other orientations then differently oriented elements must be used to look for each desired orientation. It is also necessary to be very careful that the structuring element chosen does not eliminate too many desirable objects, or retain too many undesirable ones, and sometimes this can be a delicate or even impossible balance.

Consider, for example, `ce14`, which contains two kinds of cell: small, black ones and larger, gray ones. Thresholding (p.69) the image at a value of `210` yields `ce14thr3`, in which both kinds of cell are separated from the background. We want to retain only the large cells in the image, while removing the small ones. This can be done with straightforward opening. Using a `11` pixel circular structuring element yields `ce14opn1`. Most of the desired cells are in the image, whereas none of the black cells remained. However, we cannot find any structuring element which allows us to detect the small cells and remove the large ones. Every structuring element that is small

enough to allow the dark cells remain in the image would not remove the large cells, either. This is illustrated in `ce14opn2`, which is the result of applying a 7 pixel wide circular structuring element to the thresholded image.

Common Variants

It is common for opening to be used in conjunction with closing to achieve more subtle effects, as described in the section on closing (p.132).

Exercises

1. Apply opening to `ce14` using square structuring elements (p.241) of increasing size. Compare the results obtained with the different sizes. If your implementation of the operator does not support graylevel opening, threshold (p.69) the input image.
2. How can you detect the small cells in the above example `ce14` while removing the large cells? Use the closing operator with structuring elements at different sizes in combination with some logical operator.
3. Describe two 2-D object shapes (different from the ones shown in the image below) `art1` between which simple opening *could* distinguish, when the two are mixed together in a loose flat pile. What would be the appropriate structuring elements to use?
4. Now describe two 2-D shapes that opening *couldn't* distinguish between.
5. Can you explain why the position of the origin within the structuring element does not affect the result of the opening, when it *does* make a difference for both erosion and dilation?

References

R. Haralick and L. Shapiro *Computer and Robot Vision*, Vol. 1, Addison-Wesley Publishing Company, 1992, Chap. 5, pp 174 - 185.

D. Vernon *Machine Vision*, Prentice-Hall, 1991, pp 78 - 79.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

9.4 Closing

Brief Description

Closing is an important operator from the field of mathematical morphology (p.236). Like its dual operator opening (p.127), it can be derived from the fundamental operations of erosion (p.123) and dilation (p.118). Like those operators it is normally applied to binary images (p.225), although there are graylevel (p.232) versions. Closing is similar in some ways to dilation in that it tends to enlarge the boundaries of foreground (bright) regions in an image (and shrink background color holes in such regions), but it is less destructive of the original boundary shape. As with other morphological operators (p.117), the exact operation is determined by a structuring element (p.241). The effect of the operator is to preserve *background* regions that have a similar shape to this structuring element, or that can completely contain the structuring element, while eliminating all other regions of background pixels.

How It Works

Closing is opening performed in reverse. It is defined simply as a dilation followed by an erosion *using the same structuring element for both operations*. See the sections on erosion (p.123) and dilation (p.118) for details of the individual steps. The closing operator therefore requires two inputs: an image to be closed and a structuring element.

Graylevel closing consists straightforwardly of a graylevel dilation followed by a graylevel erosion.

Closing is the dual of opening, *i.e.* closing the foreground pixels with a particular structuring element, is equivalent to closing the background with the same element.

Guidelines for Use

One of the uses of dilation is to fill in small background color holes in images, *e.g.* ‘pepper noise’ (p.221). One of the problems with doing this, however, is that the dilation will also distort *all* regions of pixels indiscriminately. By performing an erosion on the image after the dilation, *i.e.* a closing, we reduce some of this effect. The effect of closing can be quite easily visualized. Imagine taking the structuring element and sliding it around *outside* each foreground region, without changing its orientation. For any background boundary point, if the structuring element can be made to touch that point, without any part of the element being inside a foreground region, then that point remains background. If this is not possible, then the pixel is set to foreground. After the closing has been carried out the background region will be such that the structuring element can be made to cover any point in the background without any part of it also covering a foreground point, and so further closings will have no effect. This property is known as *idempotence* (p.233). The effect of a closing on a binary image using a 3×3 square structuring element is illustrated in Figure 9.10.

As with erosion and dilation, this particular 3×3 structuring element is the most commonly used, and in fact many implementations will have it hardwired into their code, in which case it is obviously not necessary to specify a separate structuring element. To achieve the effect of a closing with a larger structuring element, it is possible to perform multiple dilations followed by the same number of erosions.

Closing can sometimes be used to selectively fill in particular background regions of an image. Whether or not this can be done depends upon whether a suitable structuring element can be found that fits well inside regions that are to be preserved, but doesn’t fit inside regions that are to be removed.

The image `art4` is an image containing large holes and small holes. If it is desired to remove the small holes while retaining the large holes, then we can simply perform a closing with a disk-shaped structuring element with a diameter larger than the smaller holes, but smaller than the large holes.

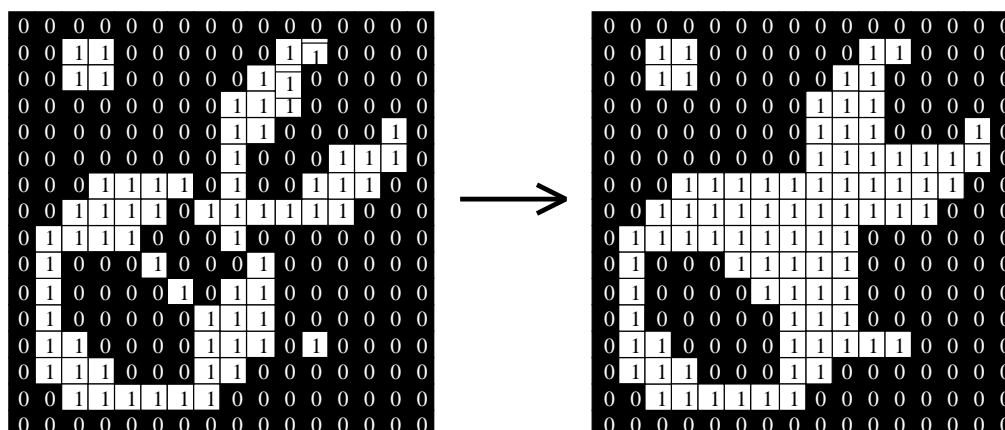


Figure 9.10: Effect of closing using a 3×3 square structuring element

The image `art4clo1` is the result of a closing with a 22 pixel diameter disk. Note that the thin black ring has also been filled in as a result of the closing operation.

In real world applications, closing can, for example, be used to enhance binary images (p.225) of objects obtained from thresholding (p.69). Consider that we want compute the skeleton (p.145) of `phn1`. To do this we first need to transform the graylevel (p.232) image into a binary image. Simply thresholding the image at a value of *100* yields `phn1thr1`. We can see that the threshold classified some parts of the receiver as background. The image `phn1clo1` is the result of closing the thresholded, image with a circular structuring element of size *20*. The merit of this operator becomes obvious when we compare the skeletons of the two binary images. The image `phn1ske1` is the skeleton of the image which was only thresholded and `phn1ske2` is the skeleton of the image produced by the closing operator. We can see that the latter skeleton is less complex and it better represents the shape of the object.

Unlike erosion and dilation, the position of the origin of the structuring element does not really matter for opening and closing. The result is independent of it.

Graylevel closing can similarly be used to select and preserve particular intensity patterns while attenuating others.

The image `ape1` is our starting point.

The result of graylevel closing with a flat 5×5 square structuring element is shown in `ape1clo1`. Notice how the dark specks in between the bright spots in the hair have been largely filled in to the same color as the bright spots, while the more uniformly colored nose area is largely the same intensity as before. Similarly the gaps between the white whiskers have been filled in.

Closing can also be used to remove ‘pepper noise’ (p.221) in images.

The image `fce5noi2` is an image containing pepper noise.

The result of a closing with a 3×3 square structuring element is shown in `fce5clo1`. The noise has been completely removed with only a little degradation to the underlying image. If, on the other hand, the noise consists of bright spots (*i.e.* ‘salt noise’ (p.221)), as can be seen in `fce5noi1`, closing yields `fce5clo2`. Here, no noise has been removed. The noise has even been increased at locations where two nearby noise pixels have merged together into one larger spot. Compare these results with the ones achieved on the same image using opening (p.127).

Although closing can sometimes be used to preserve particular intensity patterns in an image while attenuating others, this is not always the case. Some aspects of this problem are discussed under opening (p.127).

Common Variants

Opening and closing are themselves often used in combination to achieve more subtle results. If we represent the closing of an image f by $C(f)$, and its opening by $O(f)$, then some common combinations include:

Proper Opening $\text{Min}(f, O(C(f)))$

Proper Closing $\text{Max}(f, C(O(f)))$

Automedian Filter $\text{Max}(O(C(O(f))), \text{Min}(f, C(O(C(f))))$

These operators are commonly known as *morphological filters*.

Exercises

1. Use closing to remove the lines from `pcb2`, whereas the circles should remain. Do you manage to remove all the lines?
Now use closing to remove the circles while keeping the lines. Is it possible to achieve this with only one structuring element (p.241)?
2. Can you use closing to remove certain features (*e.g.* the diagonal lines) from `shu2`? Try it out.
3. Combine closing and opening (p.127) to remove the ‘salt’n’pepper’ noise (p.221) from `fce5noi3`.

References

- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 524, 552.
- R. Haralick and L. Shapiro** *Computer and Robot Vision*, Vol. 1, Addison-Wesley Publishing Company, 1992, pp 174 - 185.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986, p 387.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, pp 78 - 79.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

9.5 Hit-and-Miss Transform

Brief Description

The hit-and-miss transform is a general binary morphological operation that can be used to look for particular patterns of foreground and background pixels in an image. It is actually the basic operation of binary morphology since almost all the other binary morphological operators can be derived from it. As with other binary morphological operators it takes as input a binary image (p.225) and a structuring element (p.241), and produces another binary image as output.

How It Works

The structuring element used in the hit-and-miss is a slight extension to the type that has been introduced for erosion (p.123) and dilation (p.118), in that it can contain both foreground and background pixels (p.238), rather than just foreground pixels, *i.e.* both ones and zeros. Note that the simpler type of structuring element used with erosion and dilation is often *depicted* containing both ones and zeros as well, but in that case the zeros really stand for ‘don’t care’s’, and are just used to fill out the structuring element to a convenient shaped kernel, usually a square. In all our illustrations, these ‘don’t care’s’ are shown as blanks in the kernel in order to avoid confusion. An example of the extended kind of structuring element is shown in Figure 9.11. As usual we denote foreground pixels using ones, and background pixels using zeros.

	1	
0	1	1
0	0	

Figure 9.11: Example of the extended type of structuring element used in hit-and-miss operations. This particular element can be used to find corner points, as explained below.

The hit-and-miss operation is performed in much the same way as other morphological operators, by translating the origin of the structuring element to all points in the image, and then comparing the structuring element with the underlying image pixels. If the foreground and background pixels in the structuring element *exactly match* foreground and background pixels in the image, then the pixel underneath the origin of the structuring element is set to the foreground color. If it doesn’t match, then that pixel is set to the background color.

For instance, the structuring element shown in Figure 9.11 can be used to find right angle convex corner points in images. Notice that the pixels in the element form the shape of a bottom-left convex corner. We assume that the origin of the element is at the center of the 3×3 element. In order to find all the corners in a binary image we need to run the hit-and-miss transform four times with four different elements representing the four kinds of right angle corners found in binary images. Figure 9.12 shows the four different elements used in this operation.

After obtaining the locations of corners in each orientation, We can then simply OR (p.58) all these images together to get the final result showing the locations of all right angle convex corners in any orientation. Figure 9.13 shows the effect of this corner detection on a simple binary image.

Implementations vary as to how they handle the hit-and-miss transform at the edges of images

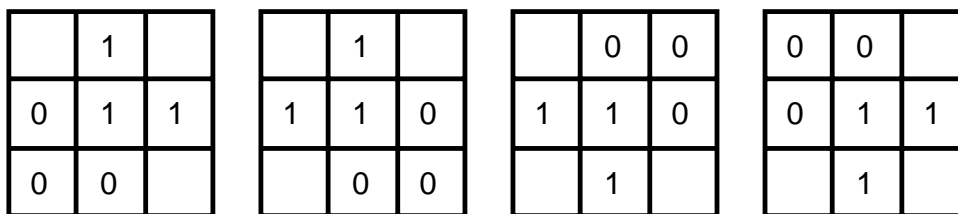


Figure 9.12: Four structuring elements used for corner finding in binary images using the hit-and-miss transform. Note that they are really all the same element, but rotated by different amounts.

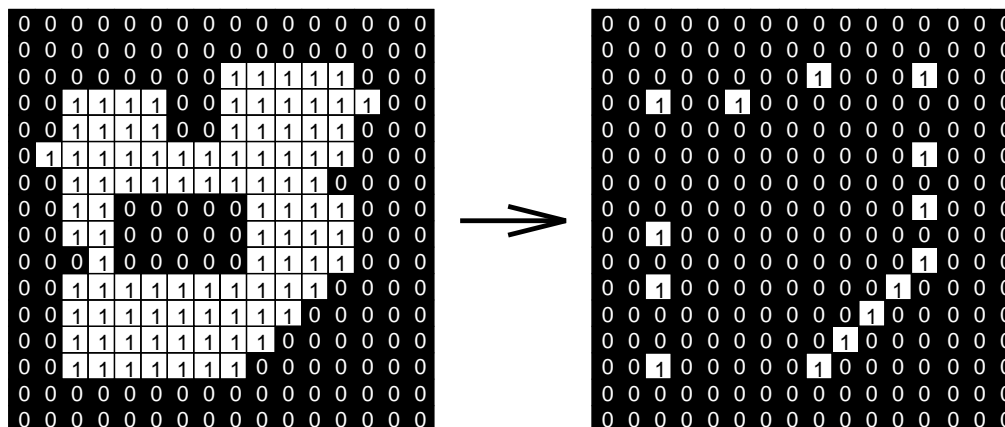


Figure 9.13: Effect of the hit-and-miss based right angle convex corner detector on a simple binary image. Note that the ‘detector’ is rather sensitive.

where the structuring element overlaps the edge of the image. A simple solution is to simply assume that any structuring element that overlaps the image does not match underlying pixels, and hence the corresponding pixel in the output should be set to zero.

The hit-and-miss transform has many applications in more complex morphological operations. It is being used to construct the thinning (p.137) and thickening (p.142) operators, and hence for all applications explained in these worksheets.

Guidelines for Use

The hit-and-miss transform is used to look for occurrences of particular binary patterns in fixed orientations. It can be used to look for several patterns (or alternatively, for the same pattern in several orientations as above) simply by running successive transforms using different structuring elements, and then ORing (p.58) the results together.

The operations of erosion (p.123), dilation (p.118), opening (p.127), closing (p.130), thinning (p.137) and thickening (p.142) can all be derived from the hit-and-miss transform in conjunction with simple set operations.

Figure 9.14 illustrates some structuring elements that can be used for locating various binary features.

We illustrate two of these applications on an image skeleton (p.145).

We start with `art7sk11` which is the skeleton of `art7`.

The image `art7ham1` shows the triple points (*i.e.* points where three lines meet) of the skeleton.

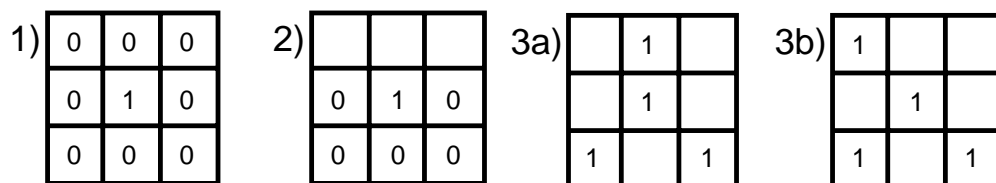


Figure 9.14: Some applications of the hit-and-miss transform. **1** is used to locate isolated points in a binary image. **2** is used to locate the end points on a binary skeleton (p.145) Note that this structuring element must be used in all its rotations so four hit-and-miss passes are required. **3a** and **3b** are used to locate the triple points (junctions) on a skeleton. Both structuring elements must be run in all orientations so eight hit-and-miss passes are required.

Note that the hit-and-miss transform itself merely outputs single foreground pixels at each triple point (the rest of the output image being black). To produce our example here, this image was then dilated (p.118) once using a cross-shaped structuring element in order to mark these isolated points clearly, and this was then ORed (p.58) with the original skeleton in order to produce the overlay.

The image `art7ham2` shows the end points of the skeleton. This image was produced in a similar way to the triple point image above, except of course that a different structuring element was used for the hit-and-miss operation. In addition, the isolated points produced by the transform were dilated with a square in order to mark them, rather than with a cross.

The successful use of the hit-and-miss transform relies on being able to think of a relatively small set of binary patterns that capture all the possible variations and orientations of a feature that is to be located. For features larger than a few pixels across this is often not feasible.

Exercises

1. How can the hit-and-miss transform be used to perform erosion?
2. How can the hit-and-miss transform, together with the NOT (p.63) operation, be used to perform dilation?
3. What is the smallest number of different structuring elements (p.241) that you would need to use to locate all foreground points in an image which have at least one foreground neighbor, using the hit-and-miss transform? What do the structuring elements look like?

References

- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, p 528.
- R. Haralick and L. Shapiro** *Computer and Robot Vision*, Vol. 1, Addison-Wesley Publishing Company, 1992, Chap 5, pp 168 - 173.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989, Chap. 9.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, p 75.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

9.6 Thinning

Brief Description

Thinning is a morphological operation that is used to remove selected foreground pixels from binary images (p.225), somewhat like erosion (p.123) or opening (p.127). It can be used for several applications, but is particularly useful for skeletonization (p.145). In this mode it is commonly used to tidy up the output of edge detectors (p.230) by reducing all lines to single pixel thickness. Thinning is normally only applied to binary images, and produces another binary image as output.

The thinning operation is related to the hit-and-miss transform (p.133), and so it is helpful to have an understanding of that operator before reading on.

How It Works

Like other morphological operators, the behavior of the thinning operation is determined by a structuring element (p.241). The binary structuring elements used for thinning are of the extended type described under the hit-and-miss transform (p.133) (*i.e.* they can contain both ones and zeros).

The thinning operation is related to the hit-and-miss transform and can be expressed quite simply in terms of it. The thinning of an image I by a structuring element J is:

$$\text{thin}(I, J) = I - \text{hit-and-miss}(I, J)$$

where the subtraction is a *logical subtraction* defined by $X - Y = X \cap \text{NOT } Y$.

In everyday terms, the thinning operation is calculated by translating the origin of the structuring element to each possible pixel position (p.238) in the image, and at each such position comparing it with the underlying image pixels. If the foreground and background pixels in the structuring element *exactly match* foreground and background pixels in the image, then the image pixel underneath the origin of the structuring element is set to background (zero). Otherwise it is left unchanged. Note that the structuring element must always have a one or a blank at its origin if it is to have any effect.

The choice of structuring element determines under what situations a foreground pixel will be set to background, and hence it determines the application for the thinning operation.

We have described the effects of a single pass of a thinning operation over the image. In fact, the operator is normally applied repeatedly until it causes no further changes to the image (*i.e.* until *convergence*). Alternatively, in some applications, *e.g.* *pruning*, the operations may only be applied for a limited number of iterations.

Thinning is the dual of thickening (p.142), *i.e.* thickening the foreground is equivalent to thinning the background.

Guidelines for Use

One of the most common uses of thinning is to reduce the thresholded (p.69) output of an edge detector such as the Sobel operator (p.188), to lines of a single pixel thickness, while preserving the full length of those lines (*i.e.* pixels at the extreme ends of lines should not be affected). A simple algorithm for doing this is the following:

Consider all pixels on the boundaries of foreground regions (*i.e.* foreground points that have at least one background neighbor). Delete any such point that has more than one foreground neighbor, as long as doing so does not *locally disconnect* (*i.e.* split into two) the region containing that pixel. Iterate until convergence.

This procedure erodes away the boundaries of foreground objects as much as possible, but does not affect pixels at the ends of lines.

This effect can be achieved using morphological thinning by iterating until convergence with the structuring elements shown in Figure 9.15, and all their 90° rotations ($4 \times 2 = 8$ structuring elements in total).

In fact what we are doing here is determining the *octagonal skeleton* of a binary shape — the set of points that lie at the centers of octagons that fit entirely inside the shape, and which touch the boundary of the shape at at least two points. See the section on skeletonization (p.145) for more details on skeletons and on other ways of computing it. Note that this skeletonization method is guaranteed to produce a connected skeleton.

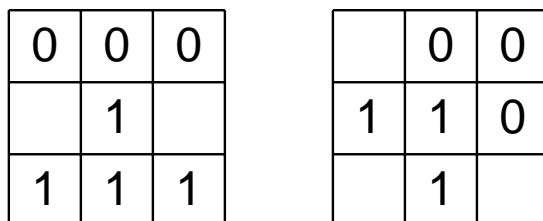


Figure 9.15: Structuring elements for skeletonization by morphological thinning. At each iteration, the image is first thinned by the left hand structuring element, and then by the right hand one, and then with the remaining six 90° rotations of the two elements. The process is repeated in cyclic fashion until none of the thinnings produces any further change. As usual, the origin of the structuring element is at the center.

Figure 9.16 shows the result of this thinning operation on a simple binary image.

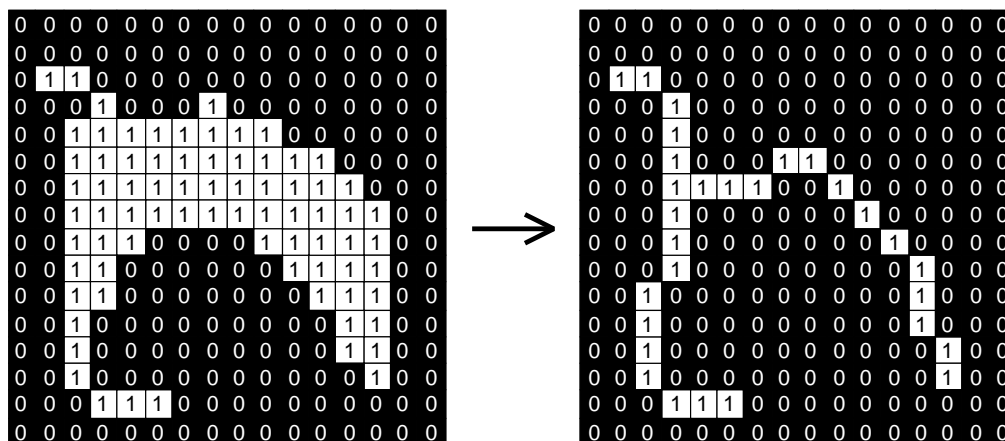


Figure 9.16: Example skeletonization by morphological thinning of a simple binary shape, using the above structuring elements. Note that the resulting skeleton is connected.

Note that skeletons produced by this method often contain undesirable short spurs produced by small irregularities in the boundary of the original object. These *spurs* can be removed by a process called pruning, which is in fact just another sort of thinning. The structuring element for this operation is shown in Figure 9.17, along with some other common structuring elements.

Note that many implementations of thinning have a particular structuring element ‘hardwired’ into them (usually the skeletonization structuring elements), and so the user does not need to be concerned about selecting one.

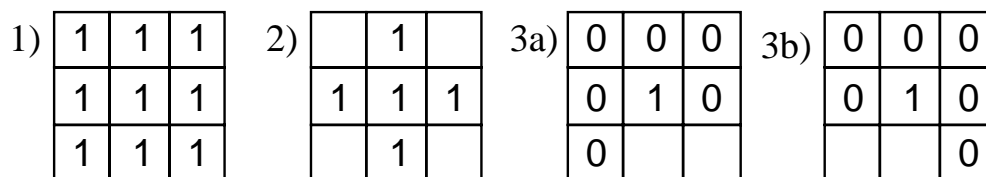


Figure 9.17: Some applications of thinning. **1** simply finds the boundary of a binary object, *i.e.* it deletes any foreground points that don't have at least one neighboring background point. Note that the detected boundary is 4-connected (p.238). **2** does the same thing but produces an 8-connected (p.238) boundary. **3a** and **3b** are used for pruning. At each thinning iteration, each element must be used in each of its four 90° rotations. Pruning is normally carried out for only a limited number of iterations to remove short spurs, since pruning until convergence will actually remove all pixels except those that form closed loops.

The image `wdg2sob1` is the result of applying the Sobel operator (p.188) to `wdg2`. Note that the detected boundaries of the object are several pixels thick.

We first threshold the image at a graylevel value (p.239) of 60 producing `wdg2sob2` in order to obtain a binary image (p.225).

Then, iterating the thinning algorithm until convergence, we get `wdg2thn1`. The detected lines have all been reduced to a single pixel width. Note however that there are still one or two 'spurs' present, which can be removed using pruning.

The image `wdg2thn2` is the result of pruning (using thinning) for five iterations. The spurs are now almost entirely gone.

Thinning is often used in combination with other morphological operators to extract a simple representation of regions. A common example is the automated recognition of hand-written characters. In this case, morphological operators are used as pre-processing to obtain the shapes of the characters which then can be used for the recognition. We illustrate a simple example using `txt3crp1`, which shows a Japanese character. Note that this and the following images were zoomed (p.90) by a factor of 4 for a better display. Hence, a 4×4 pixel square here corresponds to 1 pixel during the processing. Since we want to work on binary images (p.225), we start off by thresholding (p.69) the image at a value of 180, obtaining `txt3thr1`. A simple way to obtain the skeleton (p.145) of the character is to thin the image with the structuring elements shown in Figure 9.18 until convergence. The result is shown in `txt3thn1`.

The character is now reduced to a single pixel-wide line. However, the line is broken at some locations, which might cause problems during the recognition process. To improve the situation we can first dilate (p.118) the image to connect the lines before thinning it. Dilating the image twice with a 3×3 square structuring element yields `txt3dil1`, then the result of the thinning is `txt3thn2`. The corresponding images for three dilations are `txt3dil2` and `txt3thn3`. Although the line is now connected the process also had negative effects on the skeleton: we obtain spurs on the end points of the lines and the skeleton changes its shape at high curvature locations. We try to prune the spurs by thinning the image using the structuring elements shown in Figure 9.18.

Pruning the image which was obtained after 2 dilations and thinning yields `txt3prn1`, using two iterations for each orientation of the structuring element. For the example obtained after 3 dilations we get `txt3prn2` using 4 iterations of pruning. The spurs have now disappeared, however, the pruning has also suppressed pixels at the end of correct lines. If we want to restore these parts of the image, we can combine the dilation (p.118) operator with a logical AND (p.55) operator. First, we need to know the end points of the skeleton so that we know where to start the dilation. We find these by applying a hit-and-miss (p.133) operator using the structuring element shown in Figure 9.18. The end points of the latter of the two pruned, images are shown in `txt3end1`. Now, we dilate this image using a 3×3 structuring element. ANDing it with the thinned, but not pruned image prevents the dilation from spreading out in all direction, hence it limits the dilation

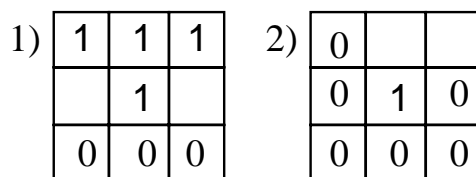


Figure 9.18: Shows the structuring elements used in the character recognition example. **1** shows the structuring element used in combination with thinning to obtain the skeleton. **2** was used in combination with thinning to prune the skeleton and with the hit-and-miss operator to find the end points of the skeleton. Each structuring element was used in each of its 45° rotations.

along the original character. This process is known as *conditional* dilation. After repeating this procedure 5 times, we obtain `txt3end2`. Although one of the parasitic branches has disappeared, the ones appearing close to the end of the lines remain.

Our final step is to OR (p.58) this image with the pruning output thus obtaining `txt3mor1`. This simple example illustrates that we can successfully apply a variety of morphological operators to obtain information about the shape of a character. However, in a real world application, more sophisticated algorithms and structuring elements would be necessary to get good results.

Thinning more complicated images often produces less spectacular results.

For instance `cln1sob1` is the output from the Sobel operator (p.188) applied to `cln1`.

The image `cln1sob3` is the same image thresholded (p.69) at a graylevel value of 200.

And `cln1thn1` is the effect of skeletonization by thinning. The result is a lot less clear than before. Compare this with the results obtained using the Canny operator (p.192).

Exercises

1. What is the difference of a thinned line obtained from the slightly different skeleton structuring elements in Figure 9.15 and Figure 9.18?
2. The conditional dilation in the character recognition example ‘followed’ the original character not only towards the initial end of the line but also backwards. Hence it also might restore unwanted spurs which were located in this direction. Can you think of a way to avoid that using a second condition?
3. Can you think of any situation in the character recognition example, in which the pruning structuring element shown in Figure 9.18 might cause problems?
4. Find the boundaries of `wdg2` using morphological edge detection (p.230). First threshold (p.69) the image, then apply thinning using the structuring element shown in Figure 9.17. Compare the result with `wdg2thn2` which was obtain using the Sobel operator (p.188) and morphological post-processing (see above).
5. Compare and contrast the effect of the Canny operator with the combined effect of Sobel operator plus thinning and pruning.
6. If an edge detector has produced long lines in its output that are approximately x pixels thick, what is the longest length spurious spur (prune) that you could expect to see after thinning to a single pixel thickness? Test your estimate out on some real images.
7. Hence, approximately how many iterations of pruning should be applied to remove spurious spurs from lines that were thinned down from a thickness of x pixels?

References

R. Gonzalez and R. Woods *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 518 - 548.

E. Davies *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, pp 149 - 161.

R. Haralick and L. Shapiro *Computer and Robot Vision*, Vol 1, Addison-Wesley Publishing Company, 1992, Chap. 5, pp 168 - 173.

A. Jain *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989, Chap. 9.

D. Vernon *Machine Vision*, Prentice-Hall, 1991, Chap. 4.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

9.7 Thickening

Brief Description

Thickening is a morphological operation that is used to *grow* selected regions of foreground pixels in binary images (p.225), somewhat like dilation (p.118) or closing (p.130). It has several applications, including determining the approximate *convex hull* of a shape, and determining the *skeleton by zone of influence*. Thickening is normally only applied to binary images, and it produces another binary image as output.

The thickening operation is related to the hit-and-miss transform (p.133), and so it is helpful to have an understanding of that operator before reading on.

How It Works

Like other morphological operators, the behavior of the thickening operation is determined by a structuring element (p.241). The binary structuring elements used for thickening are of the extended type described under the hit-and-miss transform (p.133) (*i.e.* they can contain both ones and zeros).

The thickening operation is related to the hit-and-miss transform and can be expressed quite simply in terms of it. The thickening of an image I by a structuring element J is:

$$\text{thicken}(I, J) = I \cup \text{hit-and-miss}(I, J)$$

Thus the thickened image consists of the original image plus any additional foreground pixels switched on by the hit-and-miss transform.

In everyday terms, the thickening operation is calculated by translating the origin of the structuring element to each possible pixel position (p.238) in the image, and at each such position comparing it with the underlying image pixels. If the foreground and background pixels in the structuring element *exactly match* foreground and background pixels in the image, then the image pixel underneath the origin of the structuring element is set to foreground (one). Otherwise it is left unchanged. Note that the structuring element must always have a zero or a blank at its origin if it is to have any effect.

The choice of structuring element determines under what situations a background pixel will be set to foreground, and hence it determines the application for the thickening operation.

We have described the effects of a single pass of a thickening operation over the image. In fact, the operator is normally applied repeatedly until it causes no further changes to the image (*i.e.* until *convergence*). Alternatively, in some applications, the operations may only be applied for a limited number of iterations.

Thickening is the dual of thinning (p.137), *i.e.* thinning the foreground is equivalent to thickening the background. In fact, in most cases thickening is performed by thinning the background.

Guidelines for Use

We will illustrate thickening with two applications, determining the *convex hull*, and finding the *skeleton by zone of influence* or SKIZ.

The convex hull of a binary shape can be visualized quite easily by imagining stretching an elastic band around the shape. The elastic band will follow the convex contours of the shape, but will 'bridge' the concave contours. The resulting shape will have no concavities and contains the original shape. Where an image contains multiple disconnected shapes, the convex hull algorithm will determine the convex hull of each shape, but will not connect disconnected shapes, unless their convex hulls happen to overlap (*e.g.* two interlocked 'U'-shapes).

An approximate convex hull can be computed using thickening with the structuring elements shown in Figure 9.19. The convex hull computed using this method is actually a ‘45° convex hull’ approximation, in which the boundaries of the convex hull must have orientations that are multiples of 45°. Note that this computation can be *very* slow.

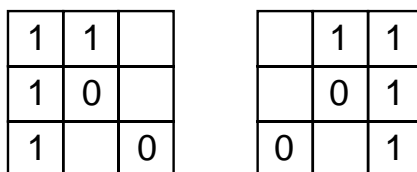


Figure 9.19: Structuring elements for determining the convex hull using thickening. During each iteration of the thickening, each element should be used in turn, and then in each of their 90° rotations, giving 8 effective structuring elements in total. The thickening is continued until no further changes occur, at which point the convex hull is complete.

The image `art8` is an image containing a number of cross-shaped binary objects.

Applying the 45° convex hull algorithm described above results in `art8thk1`. This process took a considerable amount of time — over 100 thickening passes with *each* of the eight structuring elements!

Another application of thickening is to determine the skeleton by zone of influence, or *SKIZ*. The *SKIZ* is a skeletal structure that divides an image into regions, each of which contains just one of the distinct objects in the image. The boundaries are drawn such that all points within a particular boundary are closer to the binary object contained within that boundary than to any other. As with normal skeletons (p.145), various possible distance metrics (p.229) can be used. The *SKIZ* is also sometimes called the *Voronoi diagram*.

One method of calculating the *SKIZ* is to first determine the skeleton of the background, and then prune this until convergence to remove all branches except those forming closed loops, or those intersecting the image boundary. Both of these concepts are described (applied to foreground objects) under thinning (p.137). Since thickening is the dual of thinning, we can accomplish the same thing using thickening. The structuring elements used in the two processes are shown in Figure 9.20.

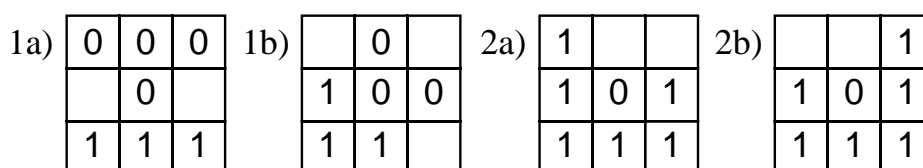


Figure 9.20: Structuring elements used in determining the *SKIZ*. **1a** and **1b** are used to perform the skeletonization of the background. Note that these elements are just the duals of the corresponding skeletonization by thinning elements. On each thickening iteration, each element is used in turn, and in each of its 90° rotations. Thickening is continued until convergence. When this is finished, structuring elements **2a** and **2b** are used in similar fashion to prune the skeleton until convergence and leave behind the *SKIZ*.

We illustrate the *SKIZ* using the same starting image as for the convex hull.

`art8thk2` shows the image after the skeleton of the background has been found. `art8thk3` is the same image after pruning until convergence. This is the *SKIZ* of the original image.

Since the *SKIZ* considers each foreground pixel as an object to which it assigns a zone of influence, it is rather sensitive to noise (p.221). If we, for example, add some ‘salt noise’ (p.221) to the above

image, we obtain `art8noi1`. The SKIZ of that image is given by `art8thk4`. Now, we not only have a zone of influence for each of the crosses, but also for each of the noise points.

Since thickening is the dual to thinning (p.137), it can be applied for the same range of tasks as thinning. Which operator is used depends on the polarity (p.225) of the image, *i.e.* if the object is represented in black and the background is white, the thickening operator thins the object.

Exercises

1. What would the convex hull look like if you used the structuring element (p.241) shown in Figure 9.21? Determine the convex hull of `art8` using this structuring element and compare it with the result obtained with the structuring element shown in Figure 9.19.

1		
1	0	
1		

Figure 9.21: Alternative structuring element to determine convex hull. This structuring element is used together with its 90° rotations.

2. Why is finding the approximate convex hull using thickening so slow?
3. Can you think of (or find out about) any uses for the SKIZ?
4. Use thickening and other morphological operators (p.117) (*e.g.* erosion (p.123) and opening (p.127)) to process `hse4`. Reduce all lines to a single pixel width and try to obtain their maximum length.

References

- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 518 - 548.
- R. Haralick and L. Shapiro** *Computer and Robot Vision*, Vol 1, Addison-Wesley Publishing Company, 1992, Chap. 5, pp 168 - 173.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989, Chap. 9.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

9.8 Skeletonization/Medial Axis Transform

Brief Description

Skeletonization is a process for reducing foreground regions in a binary image (p.225) to a skeletal remnant that largely preserves the extent and connectivity of the original region while throwing away most of the original foreground pixels. To see how this works, imagine that the foreground regions in the input binary image are made of some uniform slow-burning material. Light fires simultaneously at all points along the boundary of this region and watch the fire move into the interior. At points where the fire traveling from two different boundaries meets itself, the fire will extinguish itself and the points at which this happens form the so called ‘quench line’. This line is the skeleton. Under this definition it is clear that thinning (p.137) produces a sort of skeleton.

Another way to think about the skeleton is as the loci of centers of bi-tangent circles that fit entirely within the foreground region being considered. Figure 9.22 illustrates this for a rectangular shape.

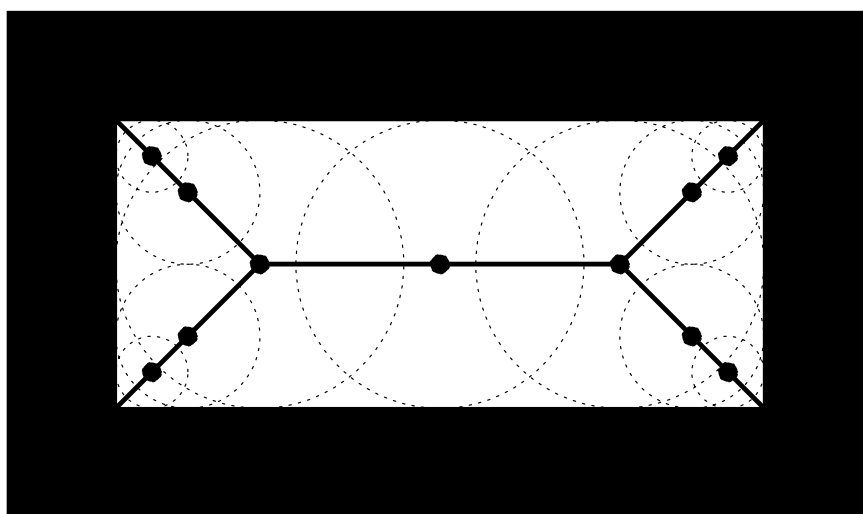


Figure 9.22: Skeleton of a rectangle defined in terms of bi-tangent circles.

The terms medial axis transform (MAT) and skeletonization are often used interchangeably but we will distinguish between them slightly. The skeleton is simply a binary image showing the simple skeleton. The MAT on the other hand is a graylevel image (p.232) where each point on the skeleton has an intensity which represents its distance to a boundary in the original object.

How It Works

The skeleton/MAT can be produced in two main ways. The first is to use some kind of morphological thinning (p.137) that successively erodes away pixels from the boundary (while preserving the end points of line segments) until no more thinning is possible, at which point what is left approximates the skeleton. The alternative method is to first calculate the distance transform (p.206) of the image. The skeleton then lies along the *singularities* (*i.e.* creases or curvature discontinuities) in the distance transform. This latter approach is more suited to calculating the MAT since the MAT is the same as the distance transform but with all points off the skeleton suppressed to zero.

Note: The MAT is often described as being the ‘locus of local maxima’ on the distance transform. This is not really true in any normal sense of the phrase ‘local maximum’. If the distance transform is displayed as a 3-D surface plot with the third dimension representing the grayvalue (p.239), the MAT can be imagined as the ridges on the 3-D surface.

Guidelines for Use

Just as there are many different types of distance transform (p.206) there are many types of skeletonization algorithm, all of which produce slightly different results. However, the general effects are all similar, as are the uses to which the skeletons are put.

The skeleton is useful because it provides a simple and compact representation of a shape that preserves many of the topological and size characteristics of the original shape. Thus, for instance, we can get a rough idea of the length of a shape by considering just the end points of the skeleton and finding the maximally separated pair of end points on the skeleton. Similarly, we can distinguish many qualitatively different shapes from one another on the basis of how many ‘triple points’ there are, *i.e.* points where at least three branches of the skeleton meet.

In addition, to this, the MAT (not the pure skeleton) has the property that it can be used to exactly reconstruct the original shape if necessary.

As with thinning, slight irregularities in a boundary will lead to spurious spurs in the final image which may interfere with recognition processes based on the topological properties of the skeleton. Despurring or pruning (p.137) can be carried out to remove spurs of less than a certain length but this is not always effective since small perturbations in the boundary of an image can lead to large spurs in the skeleton.

Note that some implementations of skeletonization algorithms produce skeletons that are not guaranteed to be continuous, even if the shape they are derived from is. This is due to the fact that the algorithms must of necessity run on a discrete grid. The MAT is actually the locus of slope discontinuities in the distance transform.

Here are some example skeletons and MATs produced from simple shapes. Note that the MATs have been contrast-stretched (p.75) in order to make them more visible.

Starting with `art5`. Skeleton is `art5sk11`, MAT is `art5mat1`.

Starting with `art6`. Skeleton is `art6sk11`, MAT is `art6mat1`.

Starting with `art7`. Skeleton is `art7sk11`, MAT is `art7mat1`.

Starting with `wdg2thr3`. Skeleton is `wdg2sk11`, MAT is `wdg2mat1`.

The skeleton and the MAT are often very sensitive to small changes in the object. If, for example, the above rectangle changes to `art5cha1`, the corresponding skeleton becomes `art5ske3`. Using a different algorithm which does not guarantee a connected skeleton yields `art5ske2`. Sometimes this sensitivity might be useful. Often, however, we need to extract the binary image (p.225) from a grayscale image (p.232). In these cases, it is often difficult to obtain the ideal shape of the object so that the skeleton becomes rather complex. We illustrate this using `phn1`. To obtain a binary image we threshold (p.69) the image at a value of *100*, thus obtaining `phn1thr1`. The skeleton of the binary image, shown in `phn1ske1`, is much more complex than the one we would obtain from the ideal shape of the telephone receiver. This example shows that simple thresholding is often not sufficient to produce a useful binary image. Some further processing might be necessary before skeletonizing the image.

The skeleton is also very sensitive to noise (p.221). To illustrate this we add some ‘pepper noise’ (p.221) to the above rectangle, thus obtaining `art5noi1`. As can be seen in `art5ske5`, the corresponding skeleton connects each noise point to the skeleton obtained from the noise free image.

Common Variants

It is also possible to skeletonize the background as opposed to the foreground of an image. This idea is closely related to the dual of the distance transform mentioned in the thickening (p.142) worksheet. This skeleton is often called the *SKIZ* (Skeleton by Influence Zones).

Exercises

1. What would the skeleton of a perfect circular disk look like?
2. Why does the skeleton of `wdg2thr3` look so strange? Can you say anything general about the effect of holes in a shape on the skeleton of that shape?
3. Try to improve the binary image (p.225) of the telephone receiver so that its skeleton becomes less complex and better represents the shape of the receiver.
4. How can the MAT be used to reconstruct the original shape of the region it was derived from?
5. Does a skeleton always go right to the edge of the shape it represents?

References

- D. Ballard and C. Brown** *Computer Vision*, Prentice-Hall, 1982, Chap. 8.
- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, pp 149 - 161.
- R. Haralick and L. Shapiro** *Computer and Robot Vision*, Vol. 1, Addison-Wesley Publishing Company, 1992, Chap. 5.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989, Chap. 9.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

Chapter 10

Digital Filters

In image processing filters are mainly used to suppress either the high frequencies in the image, *i.e.* smoothing the image, or the low frequencies, *i.e.* enhancing or detecting edges in the image.

An image can be filtered either in the frequency (p.232) or in the spatial (p.240) domain.

The first involves transforming the image into the frequency domain, multiplying it with the frequency filter (p.167) function and re-transforming the result into the spatial domain. The filter function is shaped so as to attenuate some frequencies and enhance others. For example, a simple lowpass function is 1 for frequencies smaller than the *cut-off frequency* and 0 for all others.

The corresponding process in the spatial domain (p.240) is to convolve (p.227) the input image $f(i,j)$ with the filter function $h(i,j)$. This can be written as

$$g(i,j) = h(i,j) \odot f(i,j)$$

The mathematical operation is identical to the multiplication in the frequency space, but the results of the digital implementations vary, since we have to approximate the filter function with a discrete and finite kernel (p.233).

The discrete convolution can be defined as a ‘*shift and multiply*’ operation, where we shift the kernel over the image and multiply its value with the corresponding pixel values of the image. For a square kernel (p.233) with size $M \times M$, we can calculate the output image with the following formula:

$$g(i,j) = \sum_{m=-\frac{M}{2}}^{\frac{M}{2}} \sum_{n=-\frac{M}{2}}^{\frac{M}{2}} h(m,n) f(i-m, j-n)$$

Various standard kernels exist for specific applications, where the size and the form of the kernel determine the characteristics of the operation. The most important of them are discussed in this chapter. The kernels for two examples, the mean (p.150) and the Laplacian (p.173) operator, can be seen in Figure 10.1.

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	0	-1	0
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	-1	4	-1
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	0	-1	0
Mean			Laplacian		

Figure 10.1: Convolution kernel for a mean filter and one form of the discrete Laplacian.

In contrast to the frequency domain, it is possible to implement non-linear filters (p.237) in the spatial domain. In this case, the summations in the convolution function are replaced with some kind of non-linear operator:

$$g(i, j) = O_{m,n}[h(m, n) f(i - m, j - n)]$$

For most non-linear filters the elements of $h(i, j)$ are all 1. A commonly used non-linear operator is the median (p.153), which returns the 'middle' of the input values.

10.1 Mean Filter

Brief Description

Mean filtering is a simple, intuitive and easy to implement method of *smoothing* images, *i.e.* reducing the amount of intensity variation between one pixel and the next. It is often used to reduce noise in images.

How It Works

The idea of mean filtering is simply to replace each pixel value in an image with the mean (‘average’) value of its neighbors, including itself. This has the effect of eliminating pixel values which are unrepresentative of their surroundings. Mean filtering is usually thought of as a convolution filter (p.227). Like other convolutions it is based around a kernel (p.233), which represents the shape and size of the neighborhood to be sampled when calculating the mean. Often a 3×3 square kernel is used, as shown in Figure 10.2, although larger kernels (*e.g.* 5×5 squares) can be used for more severe smoothing. (Note that a small kernel can be applied more than once in order to produce a similar but not identical effect as a single pass with a large kernel.)

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

Figure 10.2: 3×3 averaging kernel often used in mean filtering

Computing the straightforward convolution of an image with this kernel carries out the mean filtering process.

Guidelines for Use

Mean filtering is most commonly used as a simple method for reducing noise (p.221) in an image.

We illustrate the filter using `fce5`.

The image `fce5noi4` shows the original corrupted by Gaussian noise (p.221) with a mean of zero and a standard deviation (σ) of 8.

The image `fce5mea3` shows the effect of applying a 3×3 mean filter. Note that the noise is less apparent, but the image has been ‘softened’. If we increase the size of the mean filter to 5×5 , we obtain an image with less noise and less high frequency detail, as shown in `fce5mea6`.

The same image more severely corrupted by Gaussian noise (with a mean of zero and a σ of 13) is shown in `fce5noi5`.

The image `fce5mea4` is the result of mean filtering with a 3×3 kernel.

An even more challenging task is provided by `fce5noi3`. It shows an image containing ‘salt and pepper’ shot noise (p.221).

The image `fce5mea1` shows the effect of smoothing the noisy image with a 3×3 mean filter. Since the shot noise pixel values are often very different from the surrounding values, they tend to significantly distort the pixel average calculated by the mean filter.

Using a 5×5 filter instead gives `fce5mea2`. This result is not a significant improvement in noise reduction and, furthermore, the image is now very blurred.

These examples illustrate the two main problems with mean filtering, which are:

- A single pixel with a very unrepresentative value can significantly affect the mean value of all the pixels in its neighborhood.
- When the filter neighborhood straddles an edge, the filter will interpolate new values for pixels on the edge and so will blur that edge. This may be a problem if sharp edges are required in the output.

Both of these problems are tackled by the median filter (p.153), which is often a better filter for reducing noise than the mean filter, but it takes longer to compute.

In general the mean filter acts as a lowpass frequency filter (p.167) and, therefore, reduces the spatial intensity derivatives present in the image. We have already seen this effect as a ‘softening’ of the facial features in the above example. Now consider the image `sta2` which depicts a scene containing a wider range of different spatial frequencies. After smoothing once with a 3×3 mean filter we obtain `sta2mea1`. Notice that the low spatial frequency information in the background has not been affected significantly by filtering, but the (once crisp) edges of the foreground subject have been appreciably smoothed. After filtering with a 7×7 filter, we obtain an even more dramatic illustration of this phenomenon in `sta2mea2`. Compare this result to that obtained by passing a 3×3 filter over the original image three times in `sta2mea3`.

Common Variants

Variations on the mean smoothing filter discussed here include *Threshold Averaging* wherein smoothing is applied subject to the condition that the center pixel value is changed only if the difference between its original value and the average value is greater than a preset threshold. This has the effect that noise is smoothed with a less dramatic loss in image detail.

Other convolution filters that do not calculate the mean of a neighborhood are also often used for smoothing. One of the most common of these is the Gaussian smoothing filter (p.156).

Exercises

1. The mean filter is computed using a convolution. Can you think of any ways in which the special properties of the mean filter kernel can be used to speed up the convolution? What is the *computational complexity* of this faster convolution?
2. Use an edge detector on the image `bri2` and note the strength of the output. Then apply a 3×3 mean filter to the original image and run the edge detector again. Comment on the difference. What happens if a 5×5 or a 7×7 filter is used?
3. Applying a 3×3 mean filter twice does not produce quite the same result as applying a 5×5 mean filter once. However, a 5×5 convolution kernel *can* be constructed which is equivalent. What does this kernel look like?
4. Create a 7×7 convolution kernel which has an equivalent effect to three passes with a 3×3 mean filter.
5. How do you think the mean filter would cope with Gaussian noise which was not symmetric about zero? Try some examples.

References

R. Boyle and R. Thomas *Computer Vision: A First Course*, Blackwell Scientific Publications, 1988, pp 32 - 34.

E. Davies *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 3.

D. Vernon *Machine Vision*, Prentice-Hall, 1991, Chap. 4.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

10.2 Median Filter

Brief Description

The median filter is normally used to reduce noise in an image, somewhat like the mean filter (p.150). However, it often does a better job than the mean filter of preserving useful detail in the image.

How It Works

Like the mean filter (p.150), the median filter considers each pixel in the image in turn and looks at its nearby neighbors to decide whether or not it is representative of its surroundings. Instead of simply replacing the pixel value with the *mean* of neighboring pixel values, it replaces it with the *median* of those values. The median is calculated by first sorting all the pixel values from the surrounding neighborhood into numerical order and then replacing the pixel being considered with the middle pixel value. (If the neighborhood under consideration contains an even number of pixels, the average of the two middle pixel values is used.) Figure 10.3 illustrates an example calculation.

123	125	126	130	140
122	124	126	127	135
118	120	150	125	134
119	115	119	123	133
111	116	110	120	130

Neighborhood values:

115, 119, 120, 123, 124,
125, 126, 127, 150

Median value: 124

Figure 10.3: Calculating the median value of a pixel neighborhood. As can be seen, the central pixel value of 150 is rather unrepresentative of the surrounding pixels and is replaced with the median value: 124. A 3×3 square neighborhood is used here — larger neighborhoods will produce more severe smoothing.

Guidelines for Use

By calculating the median value of a neighborhood rather than the mean filter (p.150), the median filter has two main advantages over the mean filter:

- The median is a more robust average than the mean and so a single very unrepresentative pixel in a neighborhood will not affect the median value significantly.
- Since the median value must actually be the value of one of the pixels in the neighborhood, the median filter does not create new unrealistic pixel values when the filter straddles an edge. For this reason the median filter is much better at preserving sharp edges than the mean filter.

The image `fce5noi4` shows an image that has been corrupted by Gaussian noise (p.221) with mean 0 and standard deviation (σ) 8. The original image is `fce5` for comparison. Applying a

3×3 median filter produces `fce5med2`. Note how the noise has been reduced at the expense of a slight degradation in image quality. The image `fce5noi5` has been corrupted by even more noise (Gaussian noise with mean 0 and σ 13), and `fce5med3` is the result of 3×3 median filtering. The median filter is sometimes not as subjectively good at dealing with large amounts of Gaussian noise as the mean filter (p.150).

Where median filtering really comes into its own is when the noise produces extreme ‘outlier’ pixel values, as for instance in `fce5noi3` which has been corrupted with ‘salt and pepper’ noise (p.221), *i.e.* bits have been flipped with probability 1%. Median filtering this with a 3×3 neighborhood produces `fce5med1`, in which the noise has been entirely eliminated with almost no degradation to the underlying image. Compare this with the similar test on the mean filter (p.150).

Consider another example wherein the original image `sta2` has been corrupted with higher levels (*i.e.* $p=5\%$ that a bit is flipped) of salt and pepper noise (p.221) `sta2noi1`. After smoothing with a 3×3 filter, most of the noise has been eliminated `sta2med1`. If we smooth the noisy image with a larger median filter, *e.g.* 7×7 , all the noisy pixels disappear, as shown in `sta2med2`. Note that the image is beginning to look a bit ‘blotchy’, as graylevel regions are mapped together. Alternatively, we can pass a 3×3 median filter over the image three times in order to remove all the noise with less loss of detail `sta2med3`.

In general, the median filter allows a great deal of high spatial frequency detail to pass while remaining very effective at removing noise on images where less than half of the pixels in a smoothing neighborhood have been effected. (As a consequence of this, median filtering can be less effective at removing noise from images corrupted with Gaussian noise (p.221).)

One of the major problems with the median filter is that it is relatively expensive and complex to compute. To find the median it is necessary to sort all the values in the neighborhood into numerical order and this is relatively slow, even with fast sorting algorithms such as *quicksort*. The basic algorithm can, however, be enhanced somewhat for speed. A common technique is to notice that when the neighborhood window is slid across the image, many of the pixels in the window are the same from one step to the next, and the relative ordering of these with each other will obviously not have changed. Clever algorithms make use of this to improve performance.

Exercises

1. Using the image `bri2`, explore the effect of median filtering with different neighborhood sizes.
2. Compare the relative speed of mean (p.150) and median filters using the same sized neighborhood and image. How does the performance of each scale with size of image and size of neighborhood?
3. Unlike the mean filter (p.150), the median filter is non-linear. This means that for two images $A(x)$ and $B(x)$:

$$\text{median}[A(x) + B(x)] \neq \text{median}[A(x)] + \text{median}[B(x)]$$

Illustrate this to yourself by performing smoothing and pixel addition (p.43) (in the order indicated on each side of the above equation!) to a set of test images. Carry out this experiment on some simple images, *e.g.* `stp1` and `stp2`.

References

- R. Boyle and R. Thomas** *Computer Vision: A First Course*, Blackwell Scientific Publications, 1988, pp 32 - 34.
- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 3.
- A. Marion** *An Introduction to Image Processing*, Chapman and Hall, 1991, p 274.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, Chap. 4.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

10.3 Gaussian Smoothing

Brief Description

The Gaussian smoothing operator is a 2-D convolution operator (p.227) that is used to ‘blur’ images and remove detail and noise. In this sense it is similar to the mean filter (p.150), but it uses a different kernel (p.233) that represents the shape of a Gaussian (‘bell-shaped’) hump. This kernel has some special properties which are detailed below.

How It Works

The Gaussian distribution in 1-D has the form:

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$$

where σ is the standard deviation of the distribution. We have also assumed that the distribution has a mean of zero (*i.e.* it is centered on the line $x=0$). The distribution is illustrated in Figure 10.4.

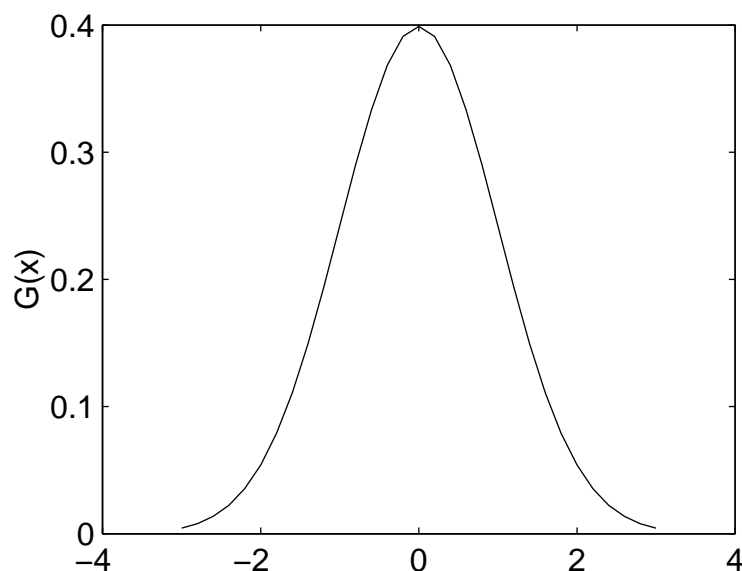


Figure 10.4: 1-D Gaussian distribution with mean 0 and $\sigma=1$

In 2-D, an isotropic (*i.e.* circularly symmetric) Gaussian has the form:

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

This distribution is shown in Figure 10.5.

The idea of Gaussian smoothing is to use this 2-D distribution as a ‘point-spread’ function, and this is achieved by convolution. Since the image is stored as a collection of discrete pixels we need to produce a discrete approximation to the Gaussian function before we can perform the convolution. In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution kernel, but in practice it is effectively zero more than about three standard deviations from the mean, and so we can truncate the kernel at this point. Figure 10.6 shows a suitable integer-valued convolution kernel that approximates a Gaussian with a σ of 1.4.

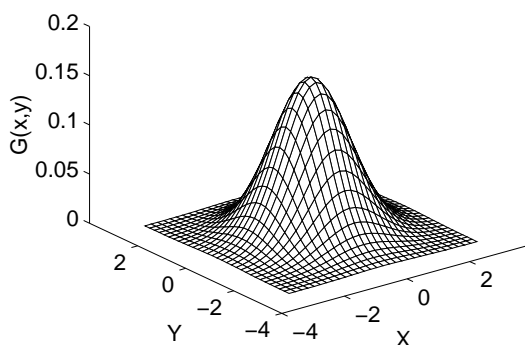


Figure 10.5: 2-D Gaussian distribution with mean (0,0) and $\sigma=1$

$$\frac{1}{115}$$

2	4	5	4	2
4	9	12	9	4
5	12	15	12	5
4	9	12	9	4
2	4	5	4	2

Figure 10.6: Discrete approximation to Gaussian function with $\sigma=1.4$

Once a suitable kernel has been calculated, then the Gaussian smoothing can be performed using standard convolution methods (p.227). The convolution can in fact be performed fairly quickly since the equation for the 2-D isotropic Gaussian shown above is separable into x and y components. Thus the 2-D convolution can be performed by first convolving with a 1-D Gaussian in the x direction, and then convolving with another 1-D Gaussian in the y direction. (The Gaussian is in fact the *only* completely circularly symmetric operator which can be decomposed in such a way.) Figure 10.7 shows the 1-D x component kernel that would be used to produce the full kernel shown in Figure 10.6. The y component is exactly the same but is oriented vertically.

A further way to compute a Gaussian smoothing with a large standard deviation is to convolve an image several times with a smaller Gaussian. While this is computationally complex, it can have applicability if the processing is carried out using a hardware pipeline.

The Gaussian filter not only has utility in engineering applications. It is also attracting attention from computational biologists because it has been attributed with some amount of biological plausibility, *e.g.* some cells in the visual pathways of the brain often have an approximately Gaussian response.

$$\frac{1}{10.7} \begin{bmatrix} 1.3 & 3.2 & 3.8 & 3.2 & 1.3 \end{bmatrix}$$

Figure 10.7: One of the pair of 1-D convolution kernels used to calculate the full kernel shown in Figure 10.6 more quickly.

Guidelines for Use

The effect of Gaussian smoothing is to blur an image, in a similar fashion to the mean filter (p.150). The degree of smoothing is determined by the standard deviation of the Gaussian. (Larger standard deviation Gaussians, of course, require larger convolution kernels in order to be accurately represented.)

The Gaussian outputs a ‘weighted average’ of each pixel’s neighborhood, with the average weighted more towards the value of the central pixels. This is in contrast to the mean filter’s uniformly weighted average. Because of this, a Gaussian provides gentler smoothing and preserves edges better than a similarly sized mean filter.

One of the principle justifications for using the Gaussian as a smoothing filter is due to its *frequency response*. Most convolution-based smoothing filters act as lowpass frequency filters (p.167). This means that their effect is to remove low spatial frequency components from an image. The frequency response of a convolution filter, *i.e.* its effect on different spatial frequencies, can be seen by taking the Fourier transform (p.209) of the filter. Figure 10.8 shows the frequency responses of a 1-D mean filter with width 7 and also of a Gaussian filter with $\sigma = 3$.

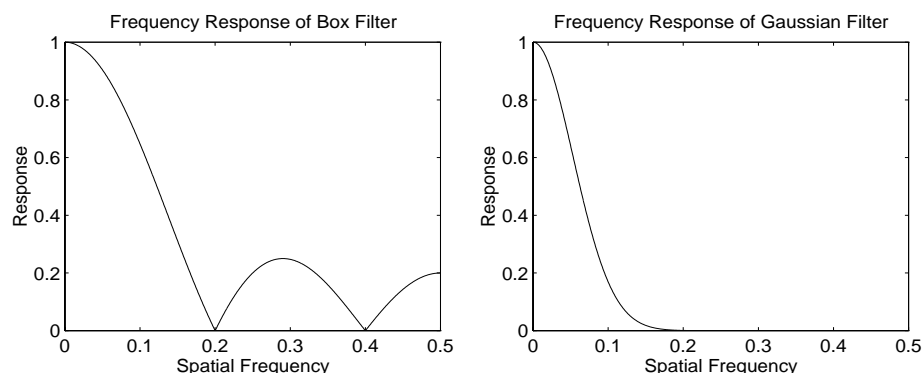


Figure 10.8: Frequency responses of Box (*i.e.* mean) filter (width 7 pixels) and Gaussian filter ($\sigma = 3$ pixels). The spatial frequency axis is marked in cycles per pixel, and hence no value above 0.5 has a real meaning.

Both filters attenuate high frequencies more than low frequencies, but the mean filter exhibits oscillations in its frequency response. The Gaussian on the other hand shows no oscillations. In fact, the shape of the frequency response curve is itself (half a) Gaussian. So by choosing an appropriately sized Gaussian filter we can be fairly confident about what range of spatial frequencies are still present in the image after filtering, which is not the case of the mean filter. This has consequences for some edge detection techniques, as mentioned in the section on zero crossings (p.199). (The Gaussian filter also turns out to be very similar to the optimal smoothing filter for edge detection under the criteria used to derive the Canny edge detector (p.192).)

We use `ben2` to illustrate the effect of smoothing with successively larger and larger Gaussian

filters.

The image `ben2gau1` shows the effect of filtering with a Gaussian of $\sigma = 1.0$ (and kernel size 5×5).

The image `ben2gau2` shows the effect of filtering with a Gaussian of $\sigma = 2.0$ (and kernel size 9×9).

The image `ben2gau3` shows the effect of filtering with a Gaussian of $\sigma = 4.0$ (and kernel size 15×15).

We now consider using the Gaussian filter for noise reduction. For example, consider the image `fce5noi4` which has been corrupted by Gaussian noise (p.221) with a mean of zero and $\sigma = 8$. Smoothing this with a 5×5 Gaussian yields `fce5gsm1`. (Compare this result with that achieved by the mean (p.150) and median (p.153) filters.)

Salt and pepper noise (p.221) is more challenging for a Gaussian filter. Here we will smooth the image `sta2noi2`, which has been corrupted by 1% salt and pepper noise (*i.e.* individual bits have been flipped with probability 1%). The image `sta2gsm1` shows the result of Gaussian smoothing (using the same convolution as above). Compare this with the original `sta2`. Notice that much of the noise still exists and that, although it has decreased in magnitude somewhat, it has been smeared out over a larger spatial region. Increasing the standard deviation continues to reduce/blur the intensity of the noise, but also attenuates high frequency detail (*e.g.* edges) significantly, as shown in `sta2gsm2`. This type of noise is better reduced using median filtering (p.153), conservative smoothing (p.161) or Crimmins Speckle Removal (p.164).

Exercises

1. Starting from the Gaussian noise (p.221) (mean 0, $\sigma = 13$) corrupted image `fce5noi5`, compute both mean filter (p.150) and Gaussian filter smoothing at various scales, and compare each in terms of noise removal vs loss of detail.
2. At how many standard deviations from the mean does a Gaussian fall to 5% of its peak value? On the basis of this suggest a suitable square kernel size for a Gaussian filter with $\sigma = s$.
3. Estimate the frequency response for a Gaussian filter by Gaussian smoothing an image, and taking its Fourier transform (p.209) both before and afterwards. Compare this with the frequency response of a mean filter (p.150).
4. How does the time taken to smooth with a Gaussian filter compare with the time taken to smooth with a mean filter (p.150) *for a kernel of the same size*? Notice that in both cases the convolution can be speeded up considerably by exploiting certain features of the kernel.

References

- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, pp 42 - 44.
- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, p 191.
- R. Haralick and L. Shapiro** *Computer and Robot Vision*, Addison-Wesley Publishing Company, 1992, Vol. 1, Chap. 7.
- B. Horn** *Robot Vision*, MIT Press, 1986, Chap. 8.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, pp 59 - 61, 214.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

10.4 Conservative Smoothing

Brief Description

Conservative smoothing is a noise reduction technique that derives its name from the fact that it employs a simple, fast filtering algorithm that sacrifices noise suppression power in order to preserve the high spatial frequency detail (*e.g.* sharp edges) in an image. It is explicitly designed to remove *noise spikes* — *i.e.* isolated pixels of exceptionally low or high pixel intensity (*e.g.* salt and pepper noise (p.221)) and is, therefore, less effective at removing *additive noise* (*e.g.* Gaussian noise (p.221)) from an image.

How It Works

Like most noise filters, conservative smoothing operates on the assumption that noise has a high spatial frequency (p.240) and, therefore, can be attenuated by a local operation which makes each pixel's intensity roughly consistent with those of its nearest neighbors. However, whereas mean filtering (p.150) accomplishes this by averaging local intensities and median filtering (p.153) by a non-linear rank selection technique, conservative smoothing simply ensures that each pixel's intensity is bounded within the *range* of intensities defined by its neighbors.

This is accomplished by a procedure which first finds the *minimum* and *maximum* intensity values of all the pixels within a windowed region around the pixel in question. If the intensity of the central pixel lies within the intensity range spread of its neighbors, it is passed on to the output image unchanged. However, if the central pixel intensity is greater than the maximum value, it is set equal to the maximum value; if the central pixel intensity is less than the minimum value, it is set equal to the minimum value. Figure 10.9 illustrates this idea.

123	125	126	130	140
122	124	126	127	135
118	120	150	125	134
119	115	119	123	133
111	116	110	120	130

Neighborhood values:

115, 119, 120, 123, 124,
125, 126, 127, 150

Max: 127, Min: 115

Figure 10.9: Conservatively smoothing a local pixel neighborhood. The central pixel of this figure contains an intensity spike (intensity value 150). In this case, conservative smoothing replaces it with the maximum intensity value (127) selected amongst those of its 8 nearest neighbors.

If we compare the result of conservative smoothing on the image segment of Figure 10.9 with the result obtained by mean filtering (p.150) and median filtering (p.153), we see that it produces a more subtle effect than both the former (whose central pixel value would become 125) and the latter (124). Furthermore, conservative smoothing is less corrupting at image edges than either of these noise suppression filters.

Guidelines for Use

Images are often corrupted by noise from several sources, the most frequent of which are *additive noise* (e.g. Gaussian noise (p.221)) and *impulse noise* (e.g. salt and pepper noise (p.221)). Linear filters, such as the mean filter (p.150), are the primary tool for smoothing digital images degraded by additive noise. For example, consider the image `fce5noi5` which has been corrupted with Gaussian noise with mean 0 and deviation 13. The image `fce5mea4` is the result after mean filtering (p.150) with a 3×3 kernel. Comparing this result with the original image `fce5`, it is obvious that in suppressing the noise, edges were blurred and detail was lost.

This example illustrates a major limitation of linear filtering, namely that a weighted average smoothing process tends to reduce the magnitude of an intensity gradient. Rather than employing a filter which inserts intermediate intensity values between high contrast neighboring pixels, we can employ a non-linear noise suppression technique, such as the median filtering (p.153) or conservative smoothing, to preserve spatial resolution by re-using pixel intensity values already in the original image. For example, consider `fce5med3` which is the Gaussian noise corrupted image considered above passed through a median filter (p.153) with a 3×3 kernel. Here, noise is dealt with less effectively, but detail is better preserved than in the case of mean filtering (p.150).

If we classify smoothing filters along this *Noise Suppression vs Detail Preservation* continuum, conservative smoothing would be rated near the tail end of the former category. The image `fce5csm2` shows the same image conservatively smoothed, using a 3×3 neighborhood. Maximum high spatial frequency (p.240) detail is preserved, but at the price of noise suppression. Conservative smoothing is unable to reduce much Gaussian noise as individual noisy pixel values do not vary much from their neighbors.

The real utility of conservative smoothing (and median filtering (p.153)) is in suppressing salt and pepper (p.221), or impulse, noise. A linear filter cannot totally eliminate impulse noise, as a single pixel which acts as an intensity spike can contribute significantly to the weighted average of the filter. Non-linear filters can be robust to this type of noise because single outlier pixel intensities can be eliminated entirely.

For example, consider `fce5noi3` which has been corrupted by 1% salt and pepper noise (p.221) (i.e. bits have been flipped with probability 1%). After mean filtering (p.150), the image is still noisy, as shown in `fce5mea1`. After median filtering (p.153), all noise is suppressed, as shown in `fce5med1`. Conservative smoothing produces an image which still contains some noise in places where the pixel neighborhoods were contaminated by more than one intensity spike `fce5csm1`. However, no image detail has been lost; e.g. notice how conservative smoothing is the only operator which preserved the reflection in the subject's eye.

Conservative smoothing works well for low levels of salt and pepper noise (p.221). However, when the image has been corrupted such that more than 1 pixel in the local neighborhood has been effected, conservative smoothing is less successful. For example, smoothing the image `sta2noi1` which has been infected with 5% salt and pepper noise (i.e. bits flipped with probability 5%), yields `sta2csm1`. The original image is `sta2`. Compare this result to that achieved by smoothing with a 3×3 median filter (p.153) `sta2med1`. You may also compare the result achieved by conservative smoothing to that obtained with 10 iterations of the Crimmins Speckle Removal (p.164) algorithm `sta2crm1`. Notice that although the latter is effective at noise removal, it smoothes away so much detail that it is of little more general utility than the conservative smoothing operator on images badly corrupted by noise.

Exercises

1. Explore the effects of conservative smoothing on images corrupted by increasing amounts of Gaussian noise. At what point does the algorithm become incapable of producing significant noise suppression?
2. Compare conservative smoothing with Crimmins Speckle Removal (p.164) on an image which is corrupted by low levels (e.g. 0.1%) of salt and pepper noise (p.221). Use the image

wom1noi1 and use the original wom1str2 as a benchmark for assessing which algorithm reduces the most noise while preserving image detail. (Note, you should not need more than 8 iterations of Crimmins to clean up this image.)

3. When low-pass filtering (p.167) (e.g. by smoothing with a mean filter (p.150)), the magnitudes of intensity gradients in the original image decrease as the size of the kernel increases. Consider the effects of increasing the neighborhood size used by the conservative smoothing algorithm. Does this trend exist? Could repeated calls to the conservative smoothing operator yield increased smoothing?
4. Conservative smoothing is a morphological operator (p.236). Viewed as such, we can define other neighborhoods (or structuring elements (p.241)) besides the square configurations used in the examples. Consider the effects of conservatively smoothing an edge (of different orientations) using the structuring elements from Figure 10.10.

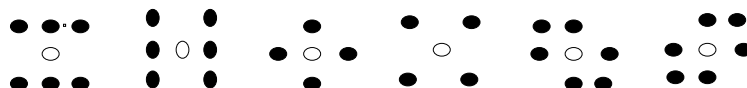


Figure 10.10: Six different structuring elements, for use in exercise 3. These local neighborhoods can be used in conservative smoothing by moving the central (white) portion of the structuring element over the image pixel of interest and then computing the maximum and minimum (and, hence the range of) intensities of the image pixels which are covered by the blackened portions of the structuring element. Using this range, a pixel can be conservatively smoothed as described in this worksheet.

References

- R. Boyle and R. Thomas** *Computer Vision: A First Course*, Blackwell Scientific Publications, 1988, pp 32 - 34.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986, Chap. 7.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, Chap. 4.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

10.5 Crimmins Speckle Removal

Brief Description

Crimmins Speckle Removal reduces speckle from an image using the Crimmins *complementary hulling algorithm*. The algorithm has been specifically designed to reduce the intensity of salt and pepper noise (p.221) in an image. Increased iterations of the algorithm yield increased levels of noise removal, but also introduce a significant amount of blurring of high frequency (p.240) details.

How It Works

Crimmins Speckle Removal works by passing an image through a speckle removing filter which uses the *complementary hulling technique* to reduce the speckle index of that image. The algorithm uses a non-linear noise reduction technique which compares the intensity of each pixel in an image with those of its 8 nearest neighbors and, based upon the relative values, increments or decrements the value of the pixel in question such that it becomes more representative of its surroundings. The noisy pixel alteration (and detection) procedure used by Crimmins is more complicated than the ranking procedure used by the non-linear median filter (p.153). It involves a series of pairwise operations in which the value of the ‘middle’ pixel within each neighborhood window is compared, in turn, with each set of neighbors (N-S, E-W, NW-SE, NE-SW) in a search for intensity spikes. The operation of the algorithm is illustrated in Figure 10.11 and described in more detail below.

For each iteration and for each pair of pixel neighbors, the entire image is sent to a *Pepper Filter* and *Salt Filter* as shown above. In the example case, the Pepper Filter is first called to determine whether each image pixel is *darker than*, *i.e.* by more than 2 intensity levels, its northern neighbors. Comparisons where this condition proves true cause the intensity value of the pixel under examination to be incremented twice (*lightened*), otherwise no change is effected. Once these changes have been recorded, the entire image is passed through the Pepper Filter again and the same series of comparisons are made between the current pixel and its southern neighbor. This sequence is repeated by the Salt Filter, where the conditions *lighter than* and *darken than* are, again, instantiated using 2 intensity levels.

Note that, over several iterations, the effects of smoothing in this way propagate out from the intensity spike to infect neighboring pixels. In other words, the algorithm smoothes by reducing the magnitude of a locally inconsistent pixel, as well as increasing the magnitude of pixels in the neighborhood surrounding the spike. It is important to notice that a *spike* is defined here as a pixel whose value is more than 2 intensity levels different from its surroundings. This means that after 2 iterations of the algorithm, the immediate neighbors of such a spike may themselves become *spikes* with respect to pixels lying in a wider neighborhood.

Guidelines for Use

We begin examining the Crimmins Speckle Removal algorithm using the image `wom1str2`, which is a contrast-stretched (p.75) version of `wom1`. We can corrupt this image with a small amount (*i.e.* $p=0.1\%$ that a bit is flipped) of salt and pepper noise (p.221) `wom1noi1` and then use several iterations of the Crimmins Speckle Removal algorithm to clean it up. The results after 1, 4 and 8 iterations of the algorithm are: `wom1crm1`, `wom1crm2`, `wom1crm3`. It took 8 iterations to produce the relatively noise-free version that is shown in the latter image.

In this case, it is instructive to examine the images where significant noise still exists. For example, we can quantify what we see qualitatively (that the intensity of the speckle noise is decreasing with increased iterations of the algorithm) by measuring the intensity values of a particular (arbitrarily chosen) noisy pixel in each of the noisy images. If we zoom (p.90) a small portion of (i) the original noisy corrupted image `wom1crp1`, (ii) the speckle filtered image after 1 iteration `wom1crp2` and (iii) 4 iterations `wom1crp3`, we find that the pepper intensity spike just under the eye takes on intensity values 51, 67, and 115 respectively. This confirms what we would expect from an algorithmic

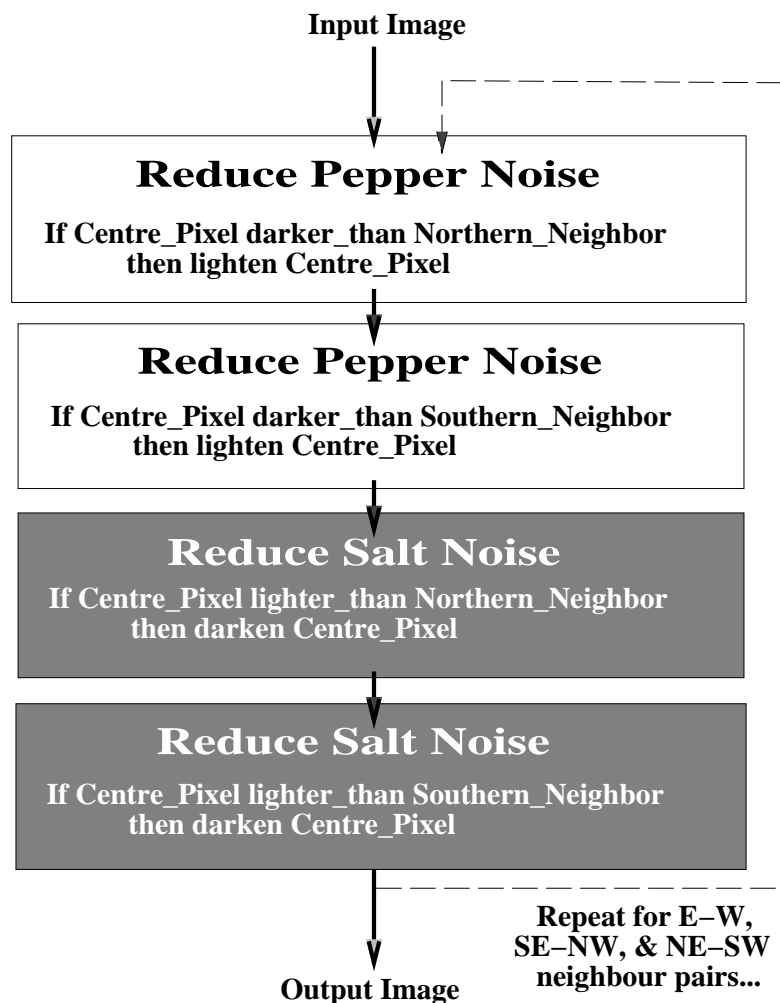


Figure 10.11: Crimmins Speckle Removal Algorithm.

analysis: each iteration of Crimmins Speckle Removal reduces the magnitude of a noise spike by 16 intensity levels (2 for each of 8 neighbor directions).

We can also see from this example that a noisy spike (*i.e.* any pixel whose magnitude is different than its neighbors by more than 2 levels) is reduced by driving its pixel intensity value towards those of its neighbors and driving the neighboring values towards that of the spike (although the latter phenomena occurs rather more slowly). By increasing the number of iterations of the algorithm, we increase the extent of this effect, and hence, incur blurring. (If we keep increasing the number of iterations, we would obtain an image with very little contrast, as all sharp gradients will be smoothed down to a magnitude of 2 intensity levels.)

An extreme example of this can be demonstrated using the image `fce5noi3` which has been corrupted by $p=1\%$ (that a bit is flipped) salt and pepper noise (p.221). The original is `fce5`. In order to remove all the noise, as shown in `fce5crm1`, 13 iterations of Crimmins Speckle Removal (p.164) are required. Much detail has been sacrificed. We can obtain better performance out of Crimmins Speckle Removal if we use fewer iterations of the algorithm. However, because this algorithm reduces noise spikes by a few intensity levels at each iteration, we can only expect to remove noise over few iterations if the noise has a similar intensity value(s) to those of the underlying image. For example, applying 8 iterations of Crimmins Speckle Removal to the face corrupted with 5% salt noise, as shown in `fce5noi8`, yields `fce5crm3`. Here the snow has been removed from the light regions on the subject's face and sweater, but remains in areas where the background is dark.

The foregoing discussion has pointed to the fact that the Crimmins Speckle Removal algorithm is most useful on images corrupted by noise whose values are not more than a couple intensity levels different from those in the underlying image. For example, we can use Crimmins to smooth the Gaussian noise (p.221) corrupted image (zero mean and $\sigma=8$) `fce5noi4`. The result, after only 2 iterations, is shown in `fce5crm4`.

Below results are tabulated for other smoothing operators applied to this noisy image.

The results of filtering this image are given below:

`fce5mea3` smoothed with a 3×3 kernel mean filter (p.150),

`fce5mea6` smoothed with a 5×5 kernel mean filter (p.150),

`fce5gsm1` smoothed with a 5×5 kernel Gaussian smoothing (p.156) filter,

`fce5med2` smoothed with a 3×3 kernel median filter (p.153),

If we allow a little noise in the output of the Crimmins filter (though not as much as we see in some of the above filter outputs), we can retain a good amount of detail, as shown in `fce5crm5`. If you now return to examine the cropped and zoomed versions of the first series of examples in this worksheet, you can see the Gaussian noise components being smoothed away after very few iterations (*i.e.* long before the more dramatic noise spikes are reduced).

Exercises

1. How does the Crimmins algorithm reduce the spatial extent of pixel alteration in the region around an intensity spike? (In other words, when the algorithm finds an isolated pepper spike against a uniform light background, how do the conditions within the algorithmic specification given above limit the amount of darkening that affects pixels outside the local neighborhood of the spike?)
2. Investigate the effects of Crimmins Speckle Removal on the image `wom1` which has poor contrast and a limited dynamic range centered in the middle of the grayscale spectrum. First filter a $p=3\%$ salt and peppered (p.221) version of this image. Then take the resultant image and contrast stretch (p.75) it using a cutoff frequency of 0.03. Compare your result to `wom1crm4` which was filtered (and noise corrupted, using $p=3\%$) *after* contrast stretching. It took 11 iterations to produce the latter. Why did it take fewer filtering iterations to remove the noise in your result? Why doesn't your result look as good?
3. Corrupt the image `fce5` with Gaussian noise (p.221) with a large σ and then filter it using Crimmins Speckle removal. Compare your results with that achieved by mean filtering (p.150), median filtering (p.153), and conservative smoothing (p.161).

References

T. Crimmins *The Geometric filter for Speckle Reduction*, Applied Optics, Vol. 24, No. 10, 15 May 1985.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

10.6 Frequency Filter

Brief Description

Frequency filters process an image in the frequency domain (p.232). The image is Fourier transformed (p.209), multiplied with the filter function and then re-transformed into the spatial domain (p.240). Attenuating high frequencies results in a smoother image in the spatial domain, attenuating low frequencies enhances the edges.

All frequency filters can also be implemented in the spatial domain and, if there exists a simple kernel for the desired filter effect, it is computationally less expensive to perform the filtering in the spatial domain. Frequency filtering is more appropriate if no straightforward kernel can be found in the spatial domain, and may also be more efficient.

How It Works

Frequency filtering is based on the Fourier Transform (p.209). (For the following discussion we assume some knowledge about the Fourier Transform, therefore it is advantageous if you have already read the corresponding worksheet.) The operator usually takes an image and a filter function in the Fourier domain. This image is then multiplied with the filter function in a pixel-by-pixel fashion:

$$G(k, l) = F(k, l)H(k, l)$$

where $F(k, l)$ is the input image in the Fourier domain, $H(k, l)$ the filter function and $G(k, l)$ is the filtered image. To obtain the resulting image in the spatial domain, $G(k, l)$ has to be re-transformed using the inverse Fourier Transform.

Since the multiplication in the Fourier space is identical to convolution (p.227) in the spatial domain, all frequency filters can in theory be implemented as a spatial filter. However, in practice, the Fourier domain filter function can only be approximated by the filtering kernel in spatial domain.

The form of the filter function determines the effects of the operator. There are basically three different kinds of filters: *lowpass*, *highpass* and *bandpass* filters. A low-pass filter attenuates high frequencies and retains low frequencies unchanged. The result in the spatial domain is equivalent to that of a smoothing filter (p.148); as the blocked high frequencies correspond to sharp intensity changes, *i.e.* to the fine-scale details and noise in the spatial domain image.

A highpass filter, on the other hand, yields edge enhancement or edge detection in the spatial domain, because edges contain many high frequencies. Areas of rather constant graylevel consist of mainly low frequencies and are therefore suppressed.

A bandpass attenuates very low and very high frequencies, but retains a middle range band of frequencies. Bandpass filtering can be used to enhance edges (suppressing low frequencies) while reducing the noise at the same time (attenuating high frequencies).

The most simple lowpass filter is the *ideal lowpass*. It suppresses all frequencies higher than the *cut-off frequency* D_0 and leaves smaller frequencies unchanged:

$$H(k, l) = \begin{cases} 1 & \text{if } \sqrt{k^2 + l^2} < D_0 \\ 0 & \text{if } \sqrt{k^2 + l^2} > D_0 \end{cases}$$

In most implementations, D_0 is given as a fraction of the highest frequency represented in the Fourier domain image.

The drawback of this filter function is a ringing effect that occurs along the edges of the filtered spatial domain image. This phenomenon is illustrated in Figure 10.12, which shows the shape of the one-dimensional filter in both the frequency and spatial domains for two different values of D_0 . We obtain the shape of the two-dimensional filter by rotating these functions about the *y-axis*. As

mentioned earlier, multiplication in the Fourier domain corresponds to a convolution in the spatial domain. Due to the multiple peaks of the ideal filter in the spatial domain, the filtered image produces ringing along intensity edges in the spatial domain.

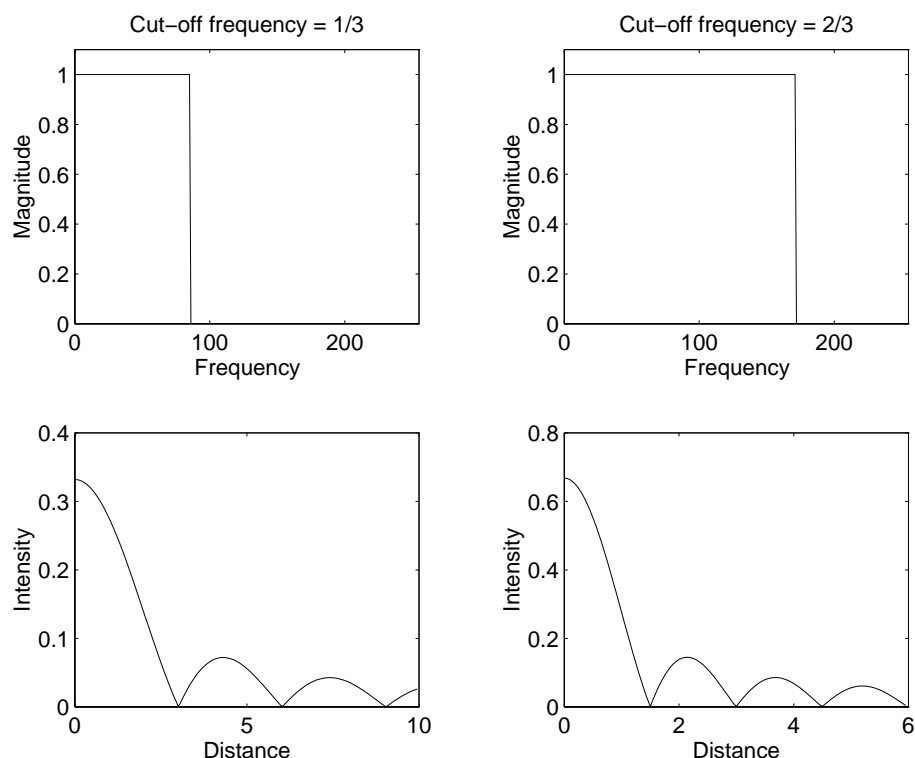


Figure 10.12: Ideal lowpass in frequency and spatial domain.

Better results can be achieved with a *Gaussian* shaped filter function. The advantage is that the Gaussian has the same shape in the spatial and Fourier domains and therefore does not incur the ringing effect in the spatial domain of the filtered image. A commonly used discrete approximation to the Gaussian is the *Butterworth filter*. Applying this filter in the frequency domain shows a similar result to the Gaussian smoothing (p.156) in the spatial domain. One difference is that the computational cost of the spatial filter increases with the standard deviation (*i.e.* with the size of the filter kernel), whereas the costs for a frequency filter are independent of the filter function. Hence, the spatial Gaussian filter is more appropriate for narrow lowpass filters, while the Butterworth filter is a better implementation for wide lowpass filters.

The same principles apply to highpass filters. We obtain a highpass filter function by inverting the corresponding lowpass filter, *e.g.* an ideal highpass filter blocks all frequencies smaller than D_0 and leaves the others unchanged.

Bandpass filters are a combination of both lowpass and highpass filters. They attenuate all frequencies smaller than a frequency D_0 and higher than a frequency D_1 , while the frequencies between the two cut-offs remain in the resulting output image. We obtain the filter function of a bandpass by multiplying the filter functions of a lowpass and of a highpass in the frequency domain, where the cut-off frequency of the lowpass is higher than that of the highpass.

Instead of using one of the standard filter functions, we can also create our own filter mask, thus enhancing or suppressing only certain frequencies. In this way we could, for example, remove periodic patterns with a certain direction in the resulting spatial domain image.

Guidelines for Use

Frequency domain filters are most commonly used as lowpass filters. We will demonstrate this performance with `c1n1`. Corrupting this image with Gaussian noise with a zero mean and a standard deviation of 8 yields `c1n1noi1`. We can reduce this type of noise using a lowpass filter, because noise consists largely of high frequencies, which are attenuated by a lowpass filter.

The image `c1n1low1` is the result of applying an ideal lowpass filter to the noisy image with the cut-off frequency being $\frac{1}{3}$. Although we managed to reduce the high frequency noise, this image is of no practical use. We lost too many of the fine-scale details and the image exhibits strong ringing due to the shape of the ideal low pass filter.

Applying the same filter with a cut-off frequency of 0.5 yields `c1n1low2`. Since this filter keeps a greater number of frequencies, more details remain in the output image. The image is less blurred, but also contains more noise. The ringing is less severe, but still exists.

Better results can be achieved with a Butterworth filter. We obtain `c1n1low3` with a cut-off frequency of $\frac{1}{3}$. This image doesn't show any visible ringing and only little noise. However, it also lost some image information, *i.e.* the edges are blurred and the image contains less details than the original.

In order to retain more details, we increase the cut-off frequency to 0.5, as can be seen in `c1n1low4`. This image is less blurred, but also contains a reasonable amount of noise. In general, when using a lowpass filter to reduce the high frequency noise, we have to compromise some desirable high frequency information if we want to smooth away significant amounts of noise.

The ringing effect originating from the shape of the ideal lowpass can be better illustrated using the following artificial image. The image `art5` is a binary image of a rectangle. Filtering this image with an ideal lowpass filter (cut-off frequency $\frac{2}{3}$) yields `art5low1`. The ringing is already recognizable in this image but is much more obvious in `art5low3` which is obtained after a histogram equalization (p.78). The effect gets even worse if we block more of the frequencies contained in the input image. In order to obtain `art5low2` we used a cut-off frequency of $\frac{1}{3}$. Apart from the (desired) smoothing the image also contains a severe ringing which clearly visible even without histogram equalization. We can also see that the cut-off frequency directly corresponds to the frequency of the ringing, *i.e.* as we double the cut-off frequency, we double the distance between two rings. The image `art5low4` has been filtered with a Butterworth filter using a cut-off frequency of $\frac{2}{3}$. In contrast to the above examples, this image doesn't exhibit any ringing.

We will illustrate the effects of highpass frequency filtering using `c1n1` as well. As a result of attenuating (or blocking) the low frequencies, areas of constant intensity in the input image are zero in the output of the highpass filter. Areas of a strong intensity gradient, containing the high frequencies, have positive and negative intensity values in the filter output. In order to display the image on the screen, an offset is added to the output in the spatial domain and the image intensities are scaled. This results in a middle grayvalue for low frequency areas and dark and light values for the edges. The image `c1n1high1` shows the output of a Butterworth highpass with the cut-off frequency being 0.5. An alternative way to display the filter output is to take the absolute value of the filtered spatial domain image. If we apply this method to the clown image (and threshold (p.69) the result with 13) we obtain `c1n1high2`. This image may be compared with `c1n1sob1`, which is an edge image produced by the Sobel operator (p.188) and, thus, shows the absolute value of the edge magnitude. We can see that the Sobel operator detects the edges better than the highpass filter. In general, spatial filters are more commonly used for edge detection while frequency filters are more often used for high frequency emphasis. Here, the filter doesn't totally block low frequencies, but magnifies high frequencies relative to low frequencies. This technique is used in the printing industries to crispen image edges.

Frequency filters are quite useful when processing parts of an image which can be associated with certain frequencies. For example, in `hse1` each part of the house is made of stripes of a different frequency and orientation. The corresponding Fourier Transform (after histogram equalization (p.78) can be seen in `hse1fou1`. We can see the main peaks in the image corresponding to the periodic patterns in the spatial domain image which now can be accessed separately. For example, we can

smooth the vertical stripes (*i.e.* those components which make up the wall in the spatial domain image) by multiplying the Fourier image with the frequency mask `hse1msk1`. The effect is that all frequencies within the black rectangle are set to zero, the others remain unchanged. Applying the inverse Fourier Transform and normalizing (p.75) the resulting image yields `hse1fil2` in the spatial domain. Although the image shows some regular patterns in the formerly constant background, the vertical stripes are almost totally removed whereas the other patterns remained mostly unchanged.

We can also use frequency filtering to achieve the opposite effect, *i.e.* finding all features in the image with certain characteristics in the frequency domain. For example, if we want to keep the vertical stripes (*i.e.* the wall) in the above image, we can use `hse1msk2` as a mask. To perform the frequency filtering we transform both the image of the house and the mask into the Fourier domain where we multiply the two images with the effect that the frequencies occurring in the mask remain in the output while the others are set to zero. Re-transforming the output into the spatial domain and normalizing it yields `hse1fil3`. In this image, the dominant pattern is the one defined by the mask. The pixel values are the highest at places which were composed of this vertical pattern in the input image and are zero in most of the background areas. It is now possible to identify the desired area by applying a threshold (p.69), as can be seen in `hse1fil4`. To understand this process we should keep in mind that a multiplication in the Fourier domain is identical to a convolution (p.227) in the spatial domain.

Frequency filters are also commonly used in image reconstruction. Here, the aim is to remove the effects of a non-ideal imaging system by multiplying the image in the Fourier space with an appropriate function. The easiest method, called inverse filtering, is to divide the image in the Fourier space with the optical transfer function (OTF). We illustrate this technique, also known as deconvolution, using `brg1`. We simulate a non-ideal OTF by multiplying the Fourier Transform of the image with the Fourier Transform of a *Gaussian* image with a standard deviation of 5. Re-transforming the result into the spatial domain yields the blurred image `brg1blu1`. We can now reconstruct the original image using inverse filtering by taking the Fourier Transform of the blurred image and dividing it by the Fourier Transform of the Gaussian kernel, which was used to initially blur the image. The reconstructed image is shown in `brg1dec1`.

Although we obtain, in the above case, exactly the original image, this method has two major problems. First, it is very sensitive to noise (p.221). If we, for example, add 0.1% spike noise to the *blurred* image, we obtain `brg1blu2`. Inverse filtering the image (as described above) using this image in order to de-blur yields the low contrast result `brg1dec2`. (Note that doing contrast enhancement to emphasize the original image features can produce an image very similar to the original, except for a loss of fine details). The situation can be slightly improved if we ignore all values of the Fourier space division in which the divisor (*i.e.* the value of the OTF) is below a certain threshold. The effect of using a threshold of 3 can be seen in `brg1dec3`. However, if we increase the threshold we have to discard more of the Fourier values and therefore lose more image information. Hence, we will be less successful in reconstructing the original image.

The second problem with this image restoration method is that we need to know the OTF which corrupted the image in the first place. If we, for example, blur the image by convolving it with the Gaussian image in the spatial domain, we obtain `brg1blu3`. Although this should theoretically be the same image as obtained from the multiplication in the Fourier Space, we obtain small differences due to quantization errors and effects around the border of an image when convolving it in spatial domain. Reconstructing the original image by dividing the blurred image in the Fourier space with the Fourier Transform of the Gaussian yields `brg1dec4` or `brg1dec5` if we use a minimum OTF threshold of 5.

We face a similar problem if we want to deconvolve a real blurred image like `orn1crp1`. Since we do not know the transfer function which caused the blurring, we have to estimate it. The images `orn1dec1` and `orn1dec2` are the results of estimating the OTF with a Gaussian image with a standard deviation of 3 and 10, respectively and applying an inverse filtering with a minimum OTF threshold of 10. We can see that the image improved only very little, if at all.

Due to the above problems, in most practical cases more sophisticated reconstruction methods are used. For example, *Wiener filtering* and *Maximum Entropy filtering* are two techniques that are

based on the same principle as inverse filtering, but produce better results on real world images.

Finally, frequency filtering can also be used for *pattern matching*. For example, we might want to find all locations in `txt2` which are occupied by a certain letter, say *X*. To do this, we need an image of an isolated *X* which can act as a mask, in this case `txt2msk1`. To perform the pattern matching, we transform both image and mask into the Fourier space and multiply them. We apply the inverse Fourier Transform to the resulting Fourier image and scale (p.48) the output to obtain `txt2fil1` in the spatial domain. This image is (theoretically) identical to the result of convolving (p.227) image and mask in the spatial domain. Hence, the image shows high values at locations in the image which match the mask well. However, apart from the *X*, these are also other letters like the *R* and the *K* which match well. In fact, if we threshold the image at 255, as can be seen in `txt2fil2`), we have two locations indicating an *X*, one of them being incorrect.

Since we multiplied the two complex Fourier images, we also changed the phase of the original text image. This results in a constant shift between the position of the letter in the original and its response in the processed image. The example shows that this straightforward method runs into problems if we want to distinguish between similar patterns or if the mask and the corresponding pattern in the data differ slightly. Another problem includes the fact that this operation is neither *rotation*- nor *scale*-invariant. (Note that we also run into these problems if we implement the operation as a simple convolution in the spatial domain.) The size of the pattern determines whether it is better to perform the matching in the spatial or frequency domain. In our case (the letter was approximately 10×20 pixels); it is substantially faster to do the matching in the frequency domain.

The above method might be modified in the following way: instead of multiplying the Fourier Transforms of the image and the mask as a first step, we threshold (p.69) the Fourier image of the mask to identify the most important frequencies which make up the letter *X* in the spatial domain. For example, scaling (p.48) the Fourier magnitude of the above mask to 255 and thresholding it at a value of 10 yields all the frequencies with at least 4% of the peak magnitude, as can be seen in `txt2fur1`. Now, we multiply this modified mask with the Fourier image of the text, thus retaining only frequencies which also appear in the letter *X*. Inverse Fourier Transforming this image yields `txt2fil3`. We can see that the *X* is the letter which preserved its shape the best and also has higher intensity values. Thresholding this image yields `txt2fil4` which correctly identifies the position of the *X*.

Exercises

1. Apply median (p.153), mean (p.150) and Gaussian smoothing (p.156) to `cln1noi1` and compare the results with the images obtained via lowpass filtering.
2. Add ‘salt and pepper’ noise (p.221) to `fce5` and then enhance the resulting image using a lowpass filter. Which method would be more suitable and why?
3. Remove single parts from `hse1` (e.g. a window, the roof or the wall) by creating an appropriate mask and multiplying it with the Fourier Transform (p.209) of the image.

References

- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 9.
- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, Chap. 4.
- R. Hamming** *Digital Filters*, Prentice-Hall, 1983.
- B. Horn** *Robot Vision*, MIT Press, 1986, Chap. 6.
- IEEE Trans. Circuits and Systems** *Special Issue on Digital Filtering and Image Processing*, Vol. CAS-2, 1975.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986, Chap. 8.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

10.7 Laplacian/Laplacian of Gaussian

Brief Description

The Laplacian is a 2-D isotropic (p.233) measure of the 2nd spatial derivative (p.240) of an image. The Laplacian of an image highlights regions of rapid intensity change and is therefore often used for edge detection (see zero crossing edge detectors (p.199)). The Laplacian is often applied to an image that has first been smoothed with something approximating a Gaussian smoothing filter (p.156) in order to reduce its sensitivity to noise, and hence the two variants will be described together here. The operator normally takes a single graylevel image as input and produces another graylevel image as output.

How It Works

The Laplacian $L(x,y)$ of an image with pixel intensity values $I(x,y)$ is given by:

$$L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

This can be calculated using a convolution filter (p.227).

Since the input image is represented as a set of discrete pixels, we have to find a discrete convolution kernel that can approximate the second derivatives in the definition of the Laplacian. Three commonly used small kernels are shown in Figure 10.13.

0	1	0	1	1	1	-1	2	-1
1	-4	1	1	-8	1	2	-4	2
0	1	0	1	1	1	-1	2	-1

Figure 10.13: Three commonly used discrete approximations to the Laplacian filter. (Note, we have defined the Laplacian using a negative peak because this is more common; however, it is equally valid to use the opposite sign convention.)

Using one of these kernels, the Laplacian can be calculated using standard convolution methods.

Because these kernels are approximating a second derivative measurement on the image, they are very sensitive to noise. To counter this, the image is often Gaussian smoothed (p.156) before applying the Laplacian filter. This pre-processing step reduces the high frequency noise components prior to the differentiation step.

In fact, since the convolution operation is associative, we can convolve the Gaussian smoothing filter with the Laplacian filter first of all, and then convolve this hybrid filter with the image to achieve the required result. Doing things this way has two advantages:

- Since both the Gaussian and the Laplacian kernels are usually much smaller than the image, this method usually requires far fewer arithmetic operations.
- The LoG ('Laplacian of Gaussian') kernel can be precalculated in advance so only one convolution needs to be performed at run-time on the image.

The 2-D LoG function centered on zero and with Gaussian standard deviation σ has the form:

$$\text{LoG}(x, y) = -\frac{1}{\pi\sigma^4} \left[1 - \frac{x^2 + y^2}{2\sigma^2} \right] e^{-\frac{x^2 + y^2}{2\sigma^2}}$$

and is shown in Figure 10.14.

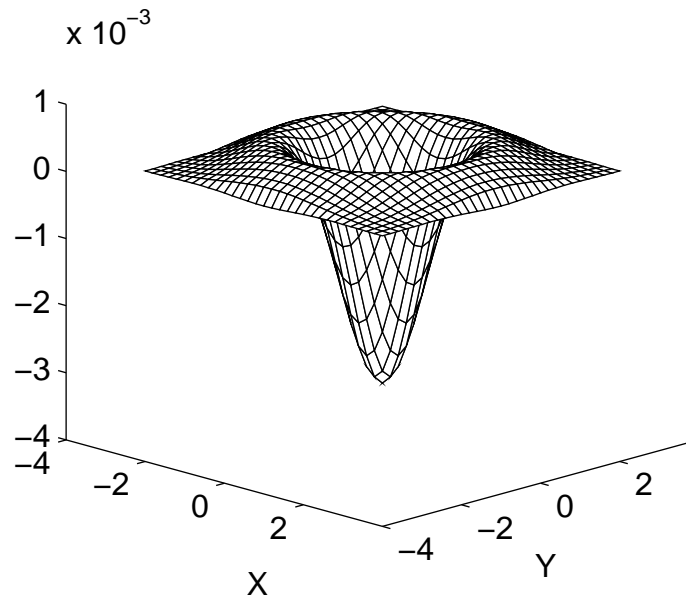


Figure 10.14: The 2-D Laplacian of Gaussian (LoG) function. The x and y axes are marked in standard deviations (σ).

A discrete kernel that approximates this function (for a Gaussian $\sigma = 1.4$) is shown in Figure 10.15.

0	0	3	2	2	2	3	0	0
0	2	3	5	5	5	3	2	0
3	3	5	3	0	3	5	3	3
2	5	3	-12	-23	-12	3	5	2
2	5	0	-23	-40	-23	0	5	2
2	5	3	-12	-23	-12	3	5	2
3	3	5	3	0	3	5	3	3
0	2	3	5	5	5	3	2	0
0	0	3	2	2	2	3	0	0

Figure 10.15: Discrete approximation to LoG function with Gaussian $\sigma = 1.4$

Note that as the Gaussian is made increasingly narrow, the LoG kernel becomes the same as the simple Laplacian kernels shown in Figure 10.13. This is because smoothing with a very narrow Gaussian ($\sigma < 0.5$ pixels) on a discrete grid has no effect. Hence on a discrete grid, the simple Laplacian can be seen as a limiting case of the LoG for narrow Gaussians.

Guidelines for Use

The LoG operator calculates the second spatial derivative of an image. This means that in areas where the image has a constant intensity (*i.e.* where the intensity gradient is zero), the LoG response will be zero. In the vicinity of a change in intensity, however, the LoG response will be positive on the darker side, and negative on the lighter side. This means that at a reasonably sharp edge between two regions of uniform but different intensities, the LoG response will be:

- zero at a long distance from the edge,
- positive just to one side of the edge,
- negative just to the other side of the edge,
- zero at some point in between, on the edge itself.

Figure 10.16 illustrates the response of the LoG to a step edge.

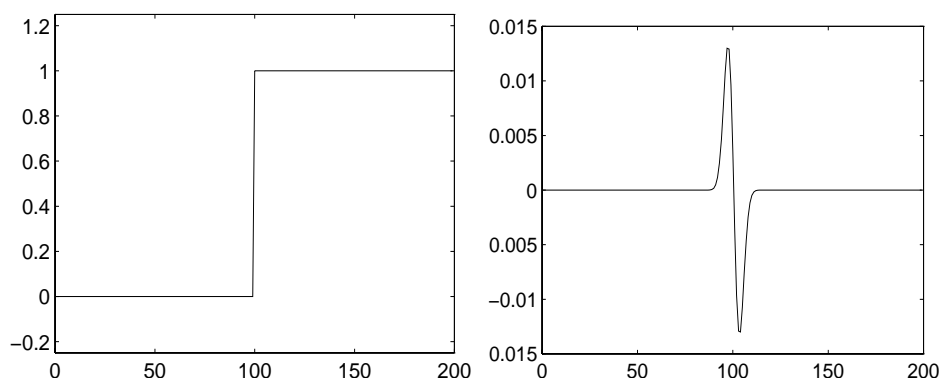


Figure 10.16: Response of 1-D LoG filter to a step edge. The left hand graph shows a 1-D image, 200 pixels long, containing a step edge. The right hand graph shows the response of a 1-D LoG filter with Gaussian $\sigma = 3$ pixels.

By itself, the effect of the filter is to highlight edges in an image.

For example, `wdg4` is a simple image with strong edges.

The image `wdg4log1` is the result of applying a LoG filter with Gaussian $\sigma = 1.0$. A 7×7 kernel was used. Note that the output contains negative and non-integer values, so for display purposes the image has been normalized (p.75) to the range 0 - 255.

If a portion of the filtered, or gradient, image is added (p.43) to the original image, then the result will be to make any edges in the original image much sharper and give them more contrast. This is commonly used as an enhancement technique in remote sensing applications.

To see this we start with `fce2`, which is a slightly blurry image of a face.

The image `fce2log1` is the effect of applying an LoG filter with Gaussian $\sigma = 1.0$, again using a 7×7 kernel.

Finally, `fce2log2` is the result of combining (*i.e.* subtracting (p.45)) the filtered image and the original image. Note that the filtered image had to be suitable scaled (p.48) before combining in order to produce a sensible enhancement. Also, it may be necessary to translate (p.97) the filtered image by half the width of the convolution kernel in both the x and y directions in order to register the images correctly.

The enhancement has made edges sharper but has also increased the effect of noise. If we simply filter the image with a Laplacian (*i.e.* use a LoG filter with a very narrow Gaussian) we obtain

`fce2lap2`. Performing edge enhancement using this sharpening image yields the noisy result `fce2lap1`. (Note that unsharp filtering (p.178) may produce an equivalent result since it can be defined by adding (p.43) the negative Laplacian image (or any suitable edge image) onto the original.) Conversely, widening the Gaussian smoothing component of the operator can reduce some of this noise, but, at the same time, the enhancement effect becomes less pronounced.

The fact that the output of the filter passes through zero at edges can be used to detect those edges. See the section on zero crossing edge detection (p.199).

Note that since the LoG is an isotropic filter (p.233), it is not possible to directly extract edge orientation information from the LoG output in the same way that it is for other edge detectors such as the Roberts Cross (p.184) and Sobel (p.188) operators.

Convolving with a kernel such as the one shown in Figure 10.15 can very easily produce output pixel values that are much larger than any of the input pixels values, and which may be negative. Therefore it is important to use an image type (*e.g.* floating point) that supports negative numbers and a large range in order to avoid overflow or saturation. The kernel can also be scaled down by a constant factor in order to reduce the range of output values.

Common Variants

It is possible to approximate the LoG filter with a filter that is just the difference of two differently sized Gaussians. Such a filter is known as a *DoG* filter (short for ‘Difference of Gaussians’).

As an aside it has been suggested (Marr 1982) that LoG filters (actually DoG filters) are important in biological visual processing.

An even cruder approximation to the LoG (but much faster to compute) is the DoB filter (‘Difference of Boxes’). This is simply the difference between two mean filters (p.150) of different sizes. It produces a kind of squared-off approximate version of the LoG.

Exercises

1. Try the effect of LoG filters using different width Gaussians on the image `ben2`. What is the general effect of increasing the Gaussian width? Notice particularly the effect on features of different sizes and thicknesses.
2. Construct a LoG filter where the kernel size is much too small for the chosen Gaussian width (*i.e.* the LoG becomes truncated). What is the effect on the output? In particular what do you notice about the LoG output in different regions each of uniform but different intensities?
3. Devise a rule to determine how big an LoG kernel should be made in relation to the σ of the underlying Gaussian if severe truncation is to be avoided.
4. If you were asked to construct an edge detector that simply looked for peaks (both positive and negative) in the output from an LoG filter, what would such a detector produce as output from a single step edge?

References

- R. Haralick and L. Shapiro** *Computer and Robot Vision*, Vol. 1, Addison-Wesley Publishing Company, 1992, pp 346 - 351.
- B. Horn** *Robot Vision*, MIT Press, 1986, Chap. 8.
- D. Marr** *Vision*, Freeman, 1982, Chap. 2, pp 54 - 78.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, pp 98 - 99, 214.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

10.8 Unsharp Filter

Brief Description

The unsharp filter is a simple sharpening operator which derives its name from the fact that it enhances edges (and other high frequency components in an image) via a procedure which subtracts an unsharp, or smoothed, version of an image from the original image. The unsharp filtering technique is commonly used in the photographic and printing industries for crispening edges.

How It Works

Unsharp masking produces an edge image $g(x, y)$ from an input image $f(x, y)$ via

$$g(x, y) = f(x, y) - f_{smooth}(x, y)$$

where $f_{smooth}(x, y)$ is a smoothed version of $f(x, y)$. (See Figure 10.17.)

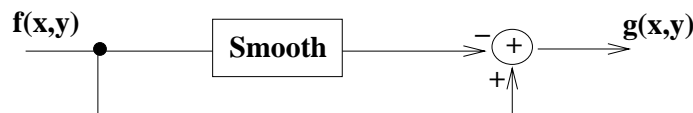


Figure 10.17: Spatial sharpening.

We can better understand the operation of the unsharp sharpening filter by examining its frequency response characteristics. If we have a signal as shown in Figure 10.18(a), subtracting away the low-pass component of that signal (as in Figure 10.18(b)), yields the highpass, or ‘edge’, representation shown in Figure 10.18(c).

This edge image can be used for sharpening if we add it back into the original signal, as shown in Figure 10.19.

Thus, the complete unsharp sharpening operator is shown in Figure 10.20.

We can now combine all of this into the equation:

$$f_{sharp}(x, y) = f(x, y) + k * g(x, y)$$

where k is a scaling constant. Reasonable values for k vary between 0.2 and 0.7, with the larger values providing increasing amounts of sharpening.

Guidelines for Use

The unsharp filter is implemented as a window-based operator, *i.e.* it relies on a convolution (p.227) kernel to perform spatial filtering. It can be implemented using an appropriately defined lowpass filter (p.167) to produce the smoothed version of an image, which is then pixel subtracted (p.45) from the original image in order to produce a description of image edges, *i.e.* a highpassed image.

For example, consider the simple image object `wdg1`, whose strong edges have been slightly blurred by camera focus. In order to extract a sharpened view of the edges, we smooth this image using a mean filter (p.150) (kernel size 3×3) and then subtract the smoothed result from the original image. The resulting image is `wdg1usp2`. (Note, the gradient image contains positive and negative values and, therefore, must be normalized (p.75) for display purposes.)

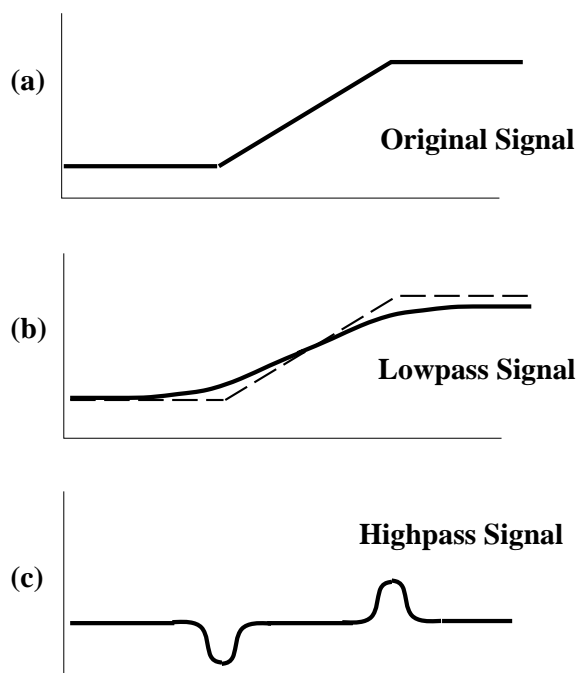


Figure 10.18: Calculating an edge image for unsharp filtering.

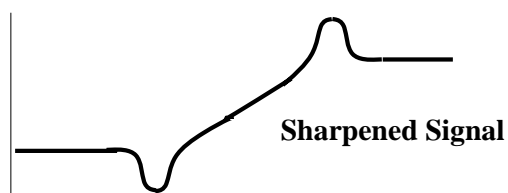


Figure 10.19: Sharpening the original signal using the edge image.

Because we subtracted all low frequency components from the original image (*i.e.*, we highpass filtered (p.167) the image) we are left with only high frequency edge descriptions. Normally, we would require that a sharpening operator give us back our original image with the high frequency components enhanced. In order to achieve this effect, we now add (p.43) some proportion of this gradient image back onto our original image. The image `wdg1usp3` has been sharpened according to this formula, where the scaling (p.48) constant k is set to 0.7.

A more common way of implementing the unsharp mask is by using the negative Laplacian operator to extract the highpass information directly. See Figure 10.21.

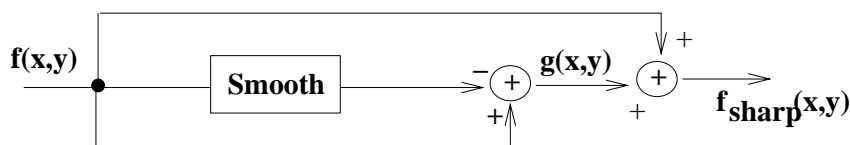


Figure 10.20: The complete unsharp filtering operator.

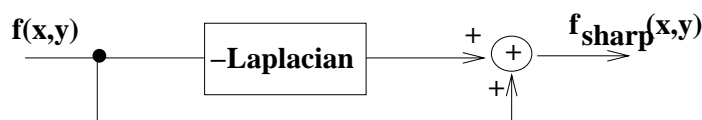


Figure 10.21: Spatial sharpening, an alternative definition.

Some unsharp masks for producing an edge image of this type are shown in Figure 10.22. These are simply negative, discrete Laplacian filters. After convolving an original image with a kernel such as one of these, it need only be scaled (p.48) and then added (p.43) to the original. (Note that in the Laplacian of Gaussian (p.173) worksheet, we demonstrated edge enhancement using the correct, or *positive*, Laplacian and LoG kernels. In that case, because the kernel peak was positive, the edge image was subtracted (p.45), rather than added (p.43), back into the original.)

0	-1	0
-1	4	-1
0	-1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

1	-2	1
-2	4	-2
1	-2	1

Figure 10.22: Three discrete approximations to the Laplacian filter.

With this in mind, we can compare the unsharp and Laplacian of Gaussian (p.173) filters. First, notice that the gradient images produced by both filters (*e.g.* `wdg1usp2` produced by unsharp and `wdg4log1` produced by LoG) exhibit the side-effect of *ringing*, or the introduction of additional intensity image structure. (Note also that the rings have opposite signs due to the difference in signs of the kernels used in each case.) This ringing occurs at high contrast edges. Figure 10.23 describes how oscillating (*i.e.* positive, negative, positive, *etc.*) terms in the output (*i.e.* ringing) are induced by the oscillating terms in the filter.

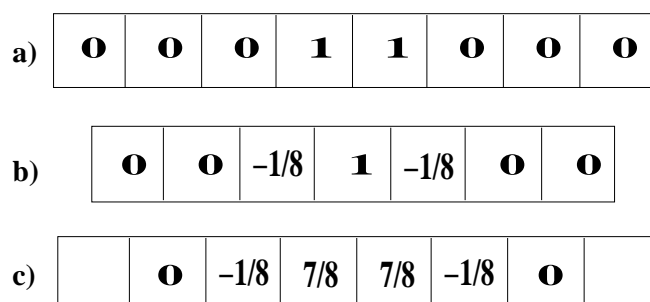


Figure 10.23: Ringing effect introduced by the unsharp mask in the presence of a 2 pixel wide, high intensity stripe. (Gray levels: -1=Dark, 0=Gray, 1=Bright.) **a)** 1-D input intensity image slice. **b)** Corresponding 1-D slice through unsharp filter. **c)** 1-D output intensity image slice.

Another interesting comparison of the two filters can be made by examining their edge enhancement capabilities. Here we begin with reference to `fce2`. The image `fce2log2` shows the sharpened version produced by a 7×7 Laplacian of Gaussian (p.173). The image `fce2lap1` is that due to unsharp sharpening with an equivalently sized Laplacian (p.173). In comparing the unsharp mask defined using the Laplacian with the LoG, it is obvious that the latter is more robust to noise,

as it has been designed explicitly to remove noise before enhancing edges. Note, we can obtain a slightly less noisy, but also less sharp, image using a smaller (*i.e.* 3×3) Laplacian kernel, as shown in `fce2usp1`.

The unsharp filter is a powerful sharpening operator, but does indeed produce a poor result in the presence of noise. For example, consider `fce5noi4` which has been deliberately corrupted by Gaussian noise (p.221). (For reference, `fce5mea3` is a mean filtered (p.150) version of this image.) Now compare this with the output of the unsharp filter `fce5usp1` and with the original image `fce5`. The unsharp mask has accentuated the noise.

Common Variants

Adaptive Unsharp Masking

A powerful technique for sharpening images in the presence of low noise levels is via an adaptive filtering algorithm. Here we look at a method of re-defining a highpass filter (such as the one shown in Figure 10.24) as the sum of a collection of edge directional kernels.

$$\frac{1}{16} \begin{array}{|c|c|c|} \hline -1 & -2 & -1 \\ \hline -2 & 12 & -2 \\ \hline -1 & -2 & -1 \\ \hline \end{array}$$

Figure 10.24: Sharpening filter.

This filter can be re-written as $1/16$ times the sum of the eight edge sensitive kernels shown in Figure 10.25.

$$\begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline -2 & 2 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline -1 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline 0 & -2 & 0 \\ \hline 0 & 2 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline 0 & 0 & -1 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 2 & -2 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & -1 \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 2 & 0 \\ \hline 0 & -2 & 0 \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline -1 & 0 & 0 \\ \hline \end{array}$$

Figure 10.25: Sharpening filter re-defined as eight edge directional kernels

Adaptive filtering using these kernels can be performed by filtering the image with each kernel, in turn, and then summing those outputs that exceed a threshold. As a final step, this result is added to the original image. (See Figure 10.26.)

This use of a threshold makes the filter adaptive in the sense that it overcomes the directionality of any single kernel by combining the results of filtering with a selection of kernels — each of which is tuned to an edge direction inherent in the image.

Exercises

1. Consider the image `ben2`, which, after unsharp sharpening (using a mean smoothing filter,

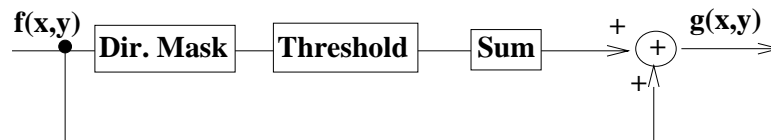


Figure 10.26: Adaptive sharpening.

with kernel size 3×3) becomes `ben2usp1`. **a)** Perform unsharp sharpening on the raw image using a Gaussian filter (p.156) (with the same kernel size). How do the sharpened images produced by the two different smoothing functions compare? **b)** Try re-sharpening this image using a filter with larger kernel sizes (*e.g.* 5×5 , 7×7 and 9×9). How does increasing the kernel size affect the result? **c)** What would you expect to see if the kernel size were allowed to approach the image size?

2. Sharpen the image `grd1`. Notice the effects on features of different scale.
3. What result would you expect from an unsharp sharpening operator defined using a smoothing filter (*e.g.* the median (p.153)) which does not produce a lowpass image.
4. Enhance the edges of the 0.1% salt and pepper noise (p.221) corrupted image `wom1noi1` using both the unsharp and Laplacian of Gaussian (p.173) filters. Which performs best under these conditions?
5. Investigate the response of the unsharp masking filter to edges of various orientations. Some useful example images include `art2`, `wdg2` and `cmp1`. Compare your results with those produced by adaptive unsharp sharpening.

References

- R. Haralick and L. Shapiro** *Computer and Robot Vision*, Addison-Wesley Publishing Company, 1992.
- B. Horn** *Robot Vision*, MIT Press, 1986, Chap. 6.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989, Chap. 7.
- R. Schalkoff** *Digital Image Processing and Computer Vision*, John Wiley & Sons, 1989, Chap. 4.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

Chapter 11

Feature Detectors

The operators included in this section are those whose purpose is to identify meaningful image features on the basis of distributions of pixel graylevels. The two categories of operators included here are:

- *Edge Pixel Detectors*, that assign a value to a pixel in proportion to the likelihood that the pixel is part of an image edge (p.230) (*i.e.* a pixel that is on the boundary between two regions of different intensity values).
- *Line Pixel Detectors*, that assign a value to a pixel in proportion to the likelihood that the pixel is part of a image line (*i.e.* a dark narrow region bounded on both sides by lighter regions, or *vice-versa*).

Detectors for other features can be defined, such as circular arc detectors in intensity images (or even more general detectors, as in the generalized Hough transform (p.214)), or planar point detectors in range images, *etc.*

Note that the operators in this section merely identify pixels likely to be part of such a structure. To actually extract the structure from the image it is then necessary to group together image pixels (which are usually adjacent). A verification stage also commonly occurs after the grouping stage (*e.g.* an examination of the residuals of the best fit line through a point set) and then the extraction of parameters of the structure (*e.g.* the equation of the line through the point set).

11.1 Roberts Cross Edge Detector

Brief Description

The Roberts Cross operator performs a simple, quick to compute, 2-D spatial gradient measurement on an image. It thus highlights regions of high spatial frequency (p.232) which often correspond to edges. In its most common usage, the input to the operator is a grayscale image, as is the output. Pixel values at each point in the output represent the estimated absolute magnitude of the spatial gradient of the input image at that point.

How It Works

In theory, the operator consists of a pair of 2×2 convolution kernels (p.227) as shown in Figure 11.1. One kernel is simply the other rotated by 90° . This is very similar to the Sobel operator (p.188).

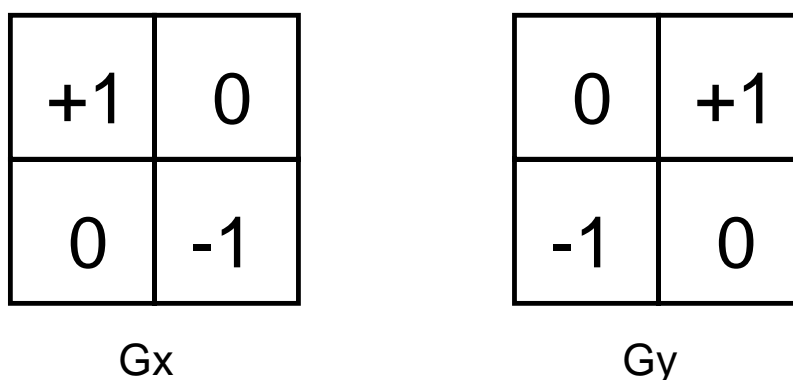


Figure 11.1: Roberts Cross convolution kernels

These kernels are designed to respond maximally to edges running at 45° to the pixel grid, one kernel for each of the two perpendicular orientations. The kernels can be applied separately to the input image, to produce separate measurements of the gradient component in each orientation (call these Gx and Gy). These can then be combined together to find the absolute magnitude of the gradient at each point and the orientation of that gradient. The gradient magnitude is given by:

$$|G| = \sqrt{Gx^2 + Gy^2}$$

although typically, an approximate magnitude is computed using:

$$|G| = |Gx| + |Gy|$$

which is much faster to compute.

The angle of orientation of the edge giving rise to the spatial gradient (relative to the pixel grid orientation) is given by:

$$\theta = \arctan(Gy/Gx) - 3\pi/4$$

In this case, orientation 0 is taken to mean that the direction of maximum contrast from black to white runs from left to right on the image, and other angles are measured clockwise from this.

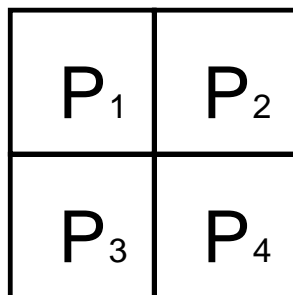


Figure 11.2: Pseudo-convolution kernels used to quickly compute approximate gradient magnitude

Often, the absolute magnitude is the only output the user sees — the two components of the gradient are conveniently computed and added in a single pass over the input image using the pseudo-convolution operator shown in Figure 11.2.

Using this kernel the approximate magnitude is given by:

$$|G| = |P_1 - P_4| + |P_2 - P_3|$$

Guidelines for Use

The main reason for using the Roberts Cross operator is that it is very quick to compute. Only four input pixels need to be examined to determine the value of each output pixel, and only subtractions and additions are used in the calculation. In addition there are no parameters to set. Its main disadvantages are that since it uses such a small kernel, it is very sensitive to noise. It also produces very weak responses to genuine edges unless they are very sharp. The Sobel operator (p.188) performs much better in this respect.

We use `cln1` to illustrate the effect of the operator.

The image `cln1rob1` is the corresponding output from the Roberts Cross operator. The gradient magnitudes output by the operator have been multiplied (p.48) by a factor of 5 to make the image clearer. Note the spurious bright dots on the image which demonstrate that the operator is susceptible to noise. Also note that only the strongest edges have been detected with any reliability.

The image `cln1rob2` is the result of thresholding (p.69) the Roberts Cross output at a pixel value of 80.

We can also apply the Roberts Cross operator to detect depth discontinuity edges in *range images*. In the range image `ufo2`, the distance from the sensor to the object is encoded in the intensity value of the image. Applying the Roberts Cross yields `ufo2rob1`. The operator produced a line with high intensity values along the boundary of the object. On the other hand, intensity changes originating from depth discontinuities within the object are not high enough to output a visible line. However, if we threshold (p.69) the image at a value of 20, all depth discontinuities in the object produce an edge in the image, as can be seen in `ufo2rob2`.

The operator's sensitivity to noise can be demonstrated if we add noise (p.221) to the above range image. The image `ufo2noi1` is the result of adding Gaussian noise with a standard deviation of 8, `ufo2rob3` is the corresponding output of the Roberts Cross operator. The difference to the previous image becomes visible if we again threshold the image at a value of 20, as can be seen in `ufo2rob4`. Now, we not only detect edges corresponding to real depth discontinuities, but also some noise points. We can show that the Roberts Cross operator is more sensitive to noise than, for example, the Sobel operator (p.188) if we apply the Sobel operator to the same noisy image. In that case, we can find a threshold which removes most of the noise pixels while keeping all edges of the object. Applying a Sobel edge detector to the above noisy image and thresholding the output at a value of 150 yields `ufo2sob4`.

The previous examples contained sharp intensity or depth changes, which enabled us (in the noise-free case) to detect the edges very well. The image `ufo1` is a range image where the depth values change much more slowly. Hence, the edges in the resulting Roberts Cross image, `ufo1rob1`, are rather faint. Since the intensity of many edge pixels in this image is very low, it is not possible to entirely separate the edges from the noise. This can be seen in `ufo1rob2`, which is the result of thresholding (p.69) the image at a value of `30`.

The effects of the shape of the edge detection kernel on the edge image can be illustrated using `hse1`. Applying the Roberts Cross operator yields `hse1rob1`. Due to the different width and orientation of the lines in the image, the response in the edge image varies significantly. Since the intensity steps between foreground and background are constant in all patterns of the original image, this shows that the Roberts Cross operator responds differently to different frequencies and orientations.

If the pixel value type being used only supports a small range of integers (*e.g.* 8-bit integer images), then it is possible for the gradient magnitude calculations to overflow the maximum allowed pixel value. In this case it is common to simply set those pixel values to the maximum allowed value. In order to avoid this happening, image types that support a greater range of pixel values, *e.g.* floating point images, can be used.

There is a slight ambiguity in the output of the Roberts operator as to which pixel in the output corresponds to which pixel in the input, since technically the operator measures the gradient intensity at the point where four pixels meet. This means that the gradient image will be shifted by half a pixel in both *x* and *y* grid directions.

Exercises

1. Why does the Roberts Cross' small kernel size make it very sensitive to noise in the image?
2. Apply the Roberts Cross operator to `ren1`. Can you obtain an edge image that contains only lines corresponding to the contours of the object? Compare with the results obtained with the Sobel (p.188) and Canny (p.192) operators.
3. Compare the result of applying the Roberts Cross operator to `hse1` with the one of using the Sobel operator (p.188).
4. Compare the performance of the Roberts Cross with the Sobel operator in terms of noise rejection, edge detection and speed.
5. Under what situations might you choose to use the Roberts Cross rather than the Sobel? And under what conditions would you avoid it?

References

- R. Boyle and R. Thomas** *Computer Vision: A First Course*, Blackwell Scientific Publications, 1988, pp 50 - 51.
- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 5.
- L. Roberts** *Machine Perception of 3-D Solids*, Optical and Electro-optical Information Processing, MIT Press 1965.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, Chap. 5.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

11.2 Sobel Edge Detector

Brief Description

The Sobel operator performs a 2-D spatial gradient measurement on an image and so emphasizes regions of high spatial frequency (p.232) that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image.

How It Works

In theory at least, the operator consists of a pair of 3×3 convolution kernels (p.227) as shown in Figure 11.3. One kernel is simply the other rotated by 90° . This is very similar to the Roberts Cross (p.184) operator.

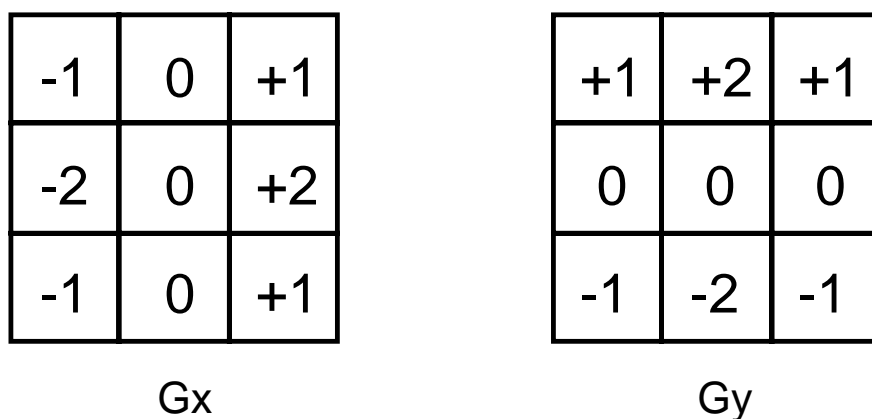


Figure 11.3: Sobel convolution kernels

These kernels are designed to respond maximally to edges running vertically and horizontally relative to the pixel grid, one kernel for each of the two perpendicular orientations. The kernels can be applied separately to the input image, to produce separate measurements of the gradient component in each orientation (call these Gx and Gy). These can then be combined together to find the absolute magnitude of the gradient at each point and the orientation of that gradient. The gradient magnitude is given by:

$$|G| = \sqrt{Gx^2 + Gy^2}$$

Typically, an approximate magnitude is computed using:

$$|G| = |Gx| + |Gy|$$

which is much faster to compute.

The angle of orientation of the edge (relative to the pixel grid) giving rise to the spatial gradient is given by:

$$\theta = \arctan(Gy/Gx)$$

In this case, orientation 0 is taken to mean that the direction of maximum contrast from black to white runs from left to right on the image, and other angles are measured anti-clockwise from this.

Often, this absolute magnitude is the only output the user sees — the two components of the gradient are conveniently computed and added in a single pass over the input image using the pseudo-convolution operator shown in Figure 11.4.

Using this kernel the approximate magnitude is given by:

$$|G| = |(P_1 + 2 \times P_2 + P_3) - (P_7 + 2 \times P_8 + P_9)| + |(P_3 + 2 \times P_6 + P_9) - (P_1 + 2 \times P_4 + P_7)|$$

P_1	P_2	P_3
P_4	P_5	P_6
P_7	P_8	P_9

Figure 11.4: Pseudo-convolution kernels used to quickly compute approximate gradient magnitude

Guidelines for Use

The Sobel operator is slower to compute than the Roberts Cross operator, but its larger convolution kernel smooths the input image to a greater extent and so makes the operator less sensitive to noise. The operator also generally produces considerably higher output values for similar edges, compared with the Roberts Cross.

As with the Roberts Cross operator, output values from the operator can easily overflow the maximum allowed pixel value for image types that only support smallish integer pixel values (*e.g.* 8-bit integer images). When this happens the standard practice is to simply set overflowing output pixels to the maximum allowed value. The problem can be avoided by using an image type that supports pixel values with a larger range.

Natural edges in images often lead to lines in the output image that are several pixels wide due to the smoothing effect of the Sobel operator. Some thinning (p.137) may be desirable to counter this. Failing that, some sort of hysteresis ridge tracking could be used as in the Canny operator (p.192).

The image `cln1sob1` shows the results of applying the Sobel operator to `cln1`. Compare this with the equivalent Roberts Cross output `cln1rob1`. Note that the spurious noise that afflicted the Roberts Cross output image is still present in this image, but its intensity relative to the genuine lines has been reduced, and so there is a good chance that we can get rid of this entirely by thresholding (p.69). Also, notice that the lines corresponding to edges have become thicker compared with the Roberts Cross output due to the increased smoothing of the Sobel operator.

The image `wdg2` shows a simpler scene containing just a single flat dark object against a lighter background. Applying the Sobel operator produces `wdg2sob1`. Note that the lighting has been carefully set up to ensure that the edges of the object are nice and sharp and free of shadows.

The Sobel edge detector can also be applied to *range images* like `ufo2`. The corresponding edge image is `ufo2sob1`. All edges in the image have been detected and can be nicely separated from the background using a threshold of `150`, as can be seen in `ufo2sob2`.

Although the Sobel operator is not as sensitive to noise (p.221) as the Roberts Cross (p.184) operator, it still amplifies high frequencies. The image `ufo2noi2` is the result of adding Gaussian noise with a standard deviation of `15` to the original image. Applying the Sobel operator yields `ufo2sob5` and thresholding the result at a value of `150` produces `ufo2sob6`. We can see that the noise has increased during the edge detection and it is no longer possible to find a threshold which removes all noise pixels and at the same time retains the edges of the objects.

The object in the previous example contains sharp edges and its surface is rather smooth. Therefore, we could (in the noise-free case) easily detect the boundary of the object without getting any erroneous pixels. A more difficult task is to detect the boundary of `ren1`, because it contains many fine depth variations (*i.e.* resulting in intensity changes in the image) on its surface. Applying the Sobel operator straightforwardly yields `ren1sob1`. We can see that the intensity of many pixels on the surface is as high as along the actual edges. One reason is that the output of many edge

pixels is greater than the maximum pixel value and therefore they are ‘cut off’ at 255. To avoid this overflow we scale (p.48) the range image by a factor 0.25 prior to the edge detection and then normalize (p.75) the output, as can be seen in `ren1sob2`. Although the result has improved significantly, we still cannot find a threshold so that a closed line along the boundary remains and all the noise disappears. Compare this image with the results obtained with the Canny (p.192) edge detector.

Common Variants

A related operator is the Prewitt gradient edge detector (not to be confused with the Prewitt *compass* edge detector (p.195)). This works in a very similar way to the Sobel operator but uses slightly different kernels, as shown in Figure 11.5. This kernel produces similar results to the Sobel, but is not as isotropic (p.233) in its response.

-1	0	+1
-1	0	+1
-1	0	+1

G_x

+1	+1	+1
0	0	0
-1	-1	-1

G_y

Figure 11.5: Masks for the Prewitt gradient edge detector.

Exercises

1. Experiment with thresholding (p.69) the example images to see if noise can be eliminated while still retaining the important edges.
2. How does the Sobel operator compare with the Roberts Cross operator in terms of noise rejection, edge detection and speed?
3. How well are the edges located using the Sobel operator? Why is this?
4. Apply the Sobel operator to `ufo1` and see if you can use thresholding (p.69) to detect the edges of the object without also selecting noisy pixels. Compare the results with those obtained with the Roberts Cross (p.184) operator.
5. Under what conditions would you want to use the Sobel rather than the Roberts Cross operator? And when would you not want to use it?

References

- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison Wesley, 1992, pp 414 - 428.
- R. Boyle and R. Thomas** *Computer Vision: A First Course*, Blackwell Scientific Publications, 1988, pp 48 - 50.
- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 5.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, Chap. 5.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

11.3 Canny Edge Detector

Brief Description

The Canny operator was designed to be an optimal edge detector (according to particular criteria — there are other detectors around that also claim to be optimal with respect to slightly different criteria). It takes as input a gray scale image, and produces as output an image showing the positions of tracked intensity discontinuities.

How It Works

The Canny operator works in a multi-stage process. First of all the image is smoothed by Gaussian convolution (p.156). Then a simple 2-D first derivative operator (somewhat like the Roberts Cross (p.184)) is applied to the smoothed image to highlight regions of the image with high first spatial derivatives. Edges give rise to ridges in the gradient magnitude image. The algorithm then tracks along the top of these ridges and sets to zero all pixels that are not actually on the ridge top so as to give a thin line in the output, a process known as *non-maximal suppression*. The tracking process exhibits hysteresis controlled by two thresholds: $T1$ and $T2$, with $T1 > T2$. Tracking can only begin at a point on a ridge higher than $T1$. Tracking then continues in both directions out from that point until the height of the ridge falls below $T2$. This hysteresis helps to ensure that noisy edges are not broken up into multiple edge fragments.

Guidelines for Use

The effect of the Canny operator is determined by three parameters — the width of the Gaussian kernel (p.233) used in the smoothing phase, and the upper and lower thresholds used by the tracker. Increasing the width of the Gaussian kernel reduces the detector's sensitivity to noise, at the expense of losing some of the finer detail in the image. The localization error in the detected edges also increases slightly as the Gaussian width is increased.

Usually, the upper tracking threshold can be set quite high, and the lower threshold quite low for good results. Setting the lower threshold too high will cause noisy edges to break up. Setting the upper threshold too low increases the number of spurious and undesirable edge fragments appearing in the output.

One problem with the basic Canny operator is to do with Y-junctions *i.e.* places where three ridges meet in the gradient magnitude image. Such junctions can occur where an edge is partially occluded by another object. The tracker will treat two of the ridges as a single line segment, and the third one as a line that approaches, but doesn't quite connect to, that line segment.

We use the image `c1n1` to demonstrate the effect of the Canny operator on a natural scene.

Using a Gaussian kernel with standard deviation 1.0 and upper and lower thresholds of 255 and 1, respectively, we obtain `c1n1can1`. Most of the major edges are detected and lots of details have been picked out well — note that this may be too much detail for subsequent processing. The 'Y-Junction effect' mentioned above can be seen at the bottom left corner of the mirror.

The image `c1n1can2` is obtained using the same kernel size and upper threshold, but with the lower threshold increased to 220. The edges have become more broken up than in the previous image, which is likely to be bad for subsequent processing. Also, the vertical edges on the wall have not been detected, along their full length.

The image `c1n1can3` is obtained by lowering the upper threshold to 128. The lower threshold is kept at 1 and the Gaussian standard deviation remains at 1.0. Many more faint edges are detected along with some short 'noisy' fragments. Notice that the detail in the clown's hair is now picked out.

The image `c1n1can4` is obtained with the same thresholds as the previous image, but the Gaussian

used has a standard deviation of 2.0. Much of the detail on the wall is no longer detected, but most of the strong edges remain. The edges also tend to be smoother and less noisy.

Edges in artificial scenes are often sharper and less complex than those in natural scenes, and this generally improves the performance of any edge detector.

The image `wdg2` shows such an artificial scene, and `wdg2can1` is the output from the Canny operator.

The Gaussian smoothing in the Canny edge detector fulfills two purposes: first it can be used to control the amount of detail that appears in the edge image and second, it can be used to suppress noise.

To demonstrate how the Canny operator performs on noisy images we use `ufo2noi2`, which contains Gaussian noise with a standard deviation of 15. Neither the Roberts Cross (p.184) nor the Sobel (p.188) operator are able to detect the edges of the object while removing all the noise in the image. Applying the Canny operator using a standard deviation of 1.0 yields `ufo2can1`. All the edges have been detected and almost all of the noise has been removed. For comparison, `ufo2sob6` is the result of applying the Sobel operator and thresholding (p.69) the output at a value of 150.

We use `ren1` to demonstrate how to control the details contained in the resulting edge image. The image `ren1can1` is the result of applying the Canny edge detector using a standard deviation of 1.0 and an upper and lower threshold of 255 and 1, respectively. This image contains many details; however, for an automated recognition task we might be interested to obtain only lines that correspond to the boundaries of the objects. If we increase the standard deviation for the Gaussian smoothing to 1.8, the Canny operator yields `ren1can2`. Now, the edges corresponding to the unevenness of the surface have disappeared from the image, but some edges corresponding to changes in the surface orientation remain. Although these edges are ‘weaker’ than the boundaries of the objects, the resulting pixel values are the same, due to the saturation (p.241) of the image. Hence, if we scale down (p.48) the image before the edge detection, we can use the upper threshold of the edge tracker to remove the weaker edges. The image `ren1can3` is the result of first scaling the image with 0.25 and then applying the Canny operator using a standard deviation of 1.8 and an upper and lower threshold of 200 and 1, respectively. The image shows the desired result that all the boundaries of the objects have been detected whereas all other edges have been removed.

Although the Canny edge detector allows us to find the intensity discontinuities in an image, it is not guaranteed that these discontinuities correspond to actual edges of the object. This is illustrated using `prt2`. We obtain `prt2can1` by using a standard deviation of 1.0 and an upper and lower threshold of 255 and 1, respectively. In this case, some edges of the object do not appear in the image and many edges in the image originate only from reflections on the object. It is a demanding task for an automated system to interpret this image. We try to improve the edge image by decreasing the upper threshold to 150, as can be seen in `prt2can2`. We now obtain most of the edges of the object, but we also increase the amount of noise. The result of further decreasing the upper threshold to 100 and increasing the standard deviation to 2 is shown in `prt2can3`.

Common Variants

The problem with Y-junctions mentioned above can be solved by including a model of such junctions in the ridge tracker. This will ensure that no spurious gaps are generated at these junctions.

Exercises

1. Adjust the parameters of the Canny operator so that you can detect the edges of `ufo2noi2` while removing *all* of the noise.
2. What effect does increasing the Gaussian kernel size have on the magnitudes of the gradient maxima at edges? What change does this imply has to be made to the tracker thresholds when the kernel size is increased?

3. It is sometimes easier to evaluate edge detector performance after thresholding (p.69) the edge detector output at some low gray scale value (*e.g.* 1) so that all detected edges are marked by bright white pixels. Try this out on the third and fourth example images of the clown mentioned above. Comment on the differences between the two images.
4. How does the Canny operator compare with the Roberts Cross (p.184) and Sobel (p.188) edge detectors in terms of speed? What do you think is the slowest stage of the process?
5. How does the Canny operator compare in terms of noise rejection and edge detection with other operators such as the Roberts Cross and Sobel operators?
6. How does the Canny operator compare with other edge detectors on simple artificial 2-D scenes? And on more complicated natural scenes?
7. Under what situations might you choose to use the Canny operator rather than the Roberts Cross or Sobel operators? In what situations would you definitely not choose it?

References

- R. Boyle and R. Thomas** *Computer Vision: A First Course*, Blackwell Scientific Publications, 1988, p 52.
- J. Canny** *A Computational Approach to Edge Detection*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 8, No. 6, Nov. 1986.
- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 5.
- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, Chap. 4.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

11.4 Compass Edge Detector

Brief Description

Compass Edge Detection is an alternative approach to the differential gradient edge detection (see the Roberts Cross (p.184) and Sobel (p.188) operators). The operation usually outputs two images, one estimating the local edge gradient magnitude and one estimating the edge orientation of the input image.

How It Works

When using compass edge detection the image is convolved (p.227) with a set of (in general δ) convolution kernels, each of which is sensitive to edges in a different orientation. For each pixel the local edge gradient *magnitude* is estimated with the maximum response of all δ kernels at this pixel location:

$$|G| = \max(|G_i| : i = 1 \text{ to } n)$$

where G_i is the response of the kernel i at the particular pixel position and n is the number of convolution kernels. The local edge *orientation* is estimated with the orientation of the kernel that yields the maximum response.

Various kernels can be used for this operation; for the following discussion we will use the Prewitt kernel. Two templates out of the set of δ are shown in Figure 11.6:

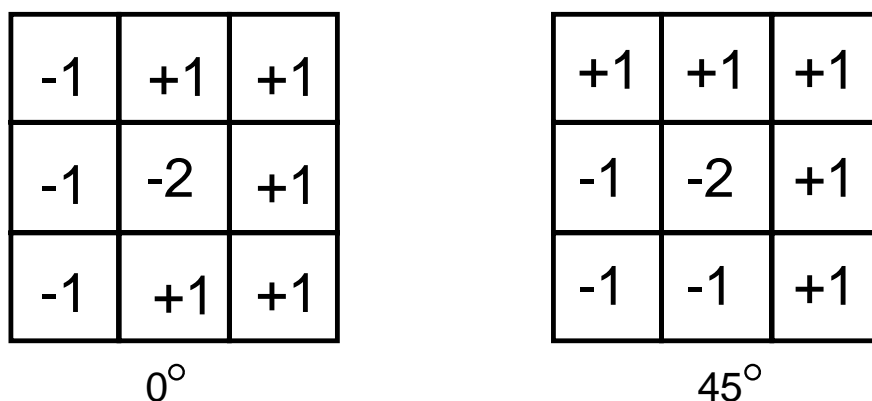


Figure 11.6: Prewitt compass edge detecting templates sensitive to edges at 0° and 45° .

The whole set of δ kernels is produced by taking one of the kernels and rotating its coefficients circularly. Each of the resulting kernels is sensitive to an edge orientation ranging from 0° to 315° in steps of 45° , where 0° corresponds to a vertical edge.

The maximum response $|G|$ for each pixel is the value of the corresponding pixel in the output magnitude image. The values for the output orientation image lie between 1 and δ , depending on which of the δ kernels produced the maximum response.

This edge detection method is also called *edge template matching*, because a set of edge templates is matched to the image, each representing an edge in a certain orientation. The edge magnitude and orientation of a pixel is then determined by the template that matches the local area of the pixel the best.

The compass edge detector is an appropriate way to estimate the magnitude *and* orientation of an edge. Although differential gradient edge detection needs a rather time-consuming calculation to estimate the orientation from the magnitudes in the x- and y-directions, the compass edge detection obtains the orientation directly from the kernel with the maximum response. The compass operator

is limited to (here) 8 possible orientations; however experience shows that most direct orientation estimates are not much more accurate.

On the other hand, the compass operator needs (here) 8 convolutions for each pixel, whereas the gradient operator needs only 2, one kernel being sensitive to edges in the vertical direction and one to the horizontal direction.

The result for the edge magnitude image is very similar with both methods, provided the same convolving kernel is used.

Guidelines for Use

If we apply the Prewitt Compass Operator to `cln1` we get two output images. The image `cln1cmp1` shows the local edge magnitude for each pixel. We can't see much in this image, because the response of the Prewitt kernel is too small. Applying histogram equalization (p.78) to this image yields `cln1cmp4`. The result is similar to `cln1sob2`, which was processed with the Sobel (p.188) differential gradient edge detector and histogram equalized.

The edges in the image can be rather thick, depending on the size of the convolving kernel used. To remove this unwanted effect some further processing (e.g. thinning (p.137)) might be necessary.

The image `cln1cmp2` is the graylevel orientation image that was contrast-stretched (p.75) for a better display. That means that the image contains 8 graylevel values between 0 and 255, each of them corresponding to an edge orientation. The orientation image as a color labeled image (containing 8 colors, each corresponding to one edge orientation) is shown in `cln1cmp3`.

The orientation of strong edges is shown very clearly, as for example at the vertical stripes of the wall paper. On a uniform background without a noticeable image gradient, on the other hand, it is ambiguous which of the 8 kernels will yield the maximum response. Therefore a uniform area results in a random distribution of the 8 orientation values.

A simple example of the orientation image is obtained if we apply the Compass Operator to `wdg2`. Each straight edge of the square yields a line of constant color (or graylevel). The circular hole in the middle, on the other hand, contains all 8 orientations and is therefore segmented in 8 parts, each of them having a different color. Again, the image is displayed as a normalized graylevel image `wdg2cmp2` and as a colored label image `wdg2cmp3`.

The image `stc1` is an image containing many edges with gradually changing orientation. Applying the Prewitt compass operator yields `stc1cmp1` for the edge magnitude and `stc1cmp2` for the edge orientation. Note that, due to the distortion of the image, all posts along the railing in the lower left corner have a slightly different orientation. However, the operator classifies them in only 3 different classes, since it assigns the same orientation label to edges when the orientation varies within 45° .

Another image suitable for edge detection is `bok1`. The corresponding output of the compass edge detector is `bok1cmp1` and `bok1cmp2` for the magnitude and orientation, respectively. Like the previous image, this image contains little noise and most of the resulting edges correspond to boundaries of objects. Again, we can see that most of the roughly vertical books were assigned the same orientation label, although the orientation varies by some amount.

We demonstrate the influence of noise (p.221) on the compass operator by adding Gaussian noise with a standard deviation of 15 to the above image. The image `bok1noi1` shows the noisy image. The Prewitt compass edge detector yields `bok1cmp3` for the edge magnitude and `bok1cmp4` for the edge orientation. Both images contain a large amount of noise and most areas in the orientation image consist of a random distribution of the 8 possible values.

Common Variants

As already mentioned earlier, there are various kernels that can be used for Compass Edge Detection. The most common ones are shown in Figure 11.7:

	0°	45°																		
Sobel	<table border="1"> <tr><td>-1</td><td>0</td><td>1</td></tr> <tr><td>-2</td><td>0</td><td>2</td></tr> <tr><td>-1</td><td>0</td><td>1</td></tr> </table>	-1	0	1	-2	0	2	-1	0	1	<table border="1"> <tr><td>0</td><td>1</td><td>2</td></tr> <tr><td>-1</td><td>0</td><td>1</td></tr> <tr><td>-2</td><td>-1</td><td>0</td></tr> </table>	0	1	2	-1	0	1	-2	-1	0
-1	0	1																		
-2	0	2																		
-1	0	1																		
0	1	2																		
-1	0	1																		
-2	-1	0																		
Kirsch	<table border="1"> <tr><td>-3</td><td>-3</td><td>5</td></tr> <tr><td>-3</td><td>0</td><td>5</td></tr> <tr><td>-3</td><td>-3</td><td>5</td></tr> </table>	-3	-3	5	-3	0	5	-3	-3	5	<table border="1"> <tr><td>-3</td><td>5</td><td>5</td></tr> <tr><td>-3</td><td>0</td><td>5</td></tr> <tr><td>-3</td><td>-3</td><td>-3</td></tr> </table>	-3	5	5	-3	0	5	-3	-3	-3
-3	-3	5																		
-3	0	5																		
-3	-3	5																		
-3	5	5																		
-3	0	5																		
-3	-3	-3																		
Robinson	<table border="1"> <tr><td>-1</td><td>0</td><td>1</td></tr> <tr><td>-1</td><td>0</td><td>1</td></tr> <tr><td>-1</td><td>0</td><td>1</td></tr> </table>	-1	0	1	-1	0	1	-1	0	1	<table border="1"> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>-1</td><td>0</td><td>1</td></tr> <tr><td>-1</td><td>-1</td><td>0</td></tr> </table>	0	1	1	-1	0	1	-1	-1	0
-1	0	1																		
-1	0	1																		
-1	0	1																		
0	1	1																		
-1	0	1																		
-1	-1	0																		

Figure 11.7: Some examples for the most common compass edge detecting kernels, each example showing two kernels out of the set of eight.

For every template, the set of all eight kernels is obtained by shifting the coefficients of the kernel circularly.

The result for using different templates is similar; the main difference is the different scale in the magnitude image. The advantage of Sobel and Robinson kernels is that only 4 out of the 8 magnitude values must be calculated. Since each pair of kernels rotated about 180° opposite is symmetric, each of the remaining four values can be generated by inverting the result of the opposite kernel.

Exercises

1. Compare the performance of the different kernels by applying them to `stc1`.
2. Compare the magnitude edge image of the book shelf with and without noise. Can you find a threshold (p.69) that retains all important edges but removes the noise?
3. Produce an image containing 8 edge orientations from `wdg2` (e.g. by rotating (p.93) the image about 45° and blending (p.53) it with the original). Then apply the compass edge operator to the resulting image and examine the edge orientation image. Do the same with an image containing 12 different edge orientations.
4. Take the orientation image obtained in exercise 2 and mask out the pixels not corresponding to a strong edge using the thresholded edge magnitude image as a mask.

References

E. Davies *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, pp 101 - 110.

R. Gonzalez and R. Woods *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, p 199.

D. Vernon *Machine Vision*, Prentice-Hall, 1991, Chap. 5.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

11.5 Zero Crossing Detector

Marr edge detector, Laplacian of Gaussian edge detector

Brief Description

The zero crossing detector looks for places in the Laplacian (p.173) of an image where the value of the Laplacian passes through zero — *i.e.* points where the Laplacian changes sign. Such points often occur at ‘edges’ in images — *i.e.* points where the intensity of the image changes rapidly, but they also occur at places that are not as easy to associate with edges. It is best to think of the zero crossing detector as some sort of feature detector rather than as a specific edge detector (p.230). Zero crossings always lie on closed contours, and so the output from the zero crossing detector is usually a binary image with single pixel thickness lines showing the positions of the zero crossing points.

The starting point for the zero crossing detector is an image which has been filtered using the Laplacian of Gaussian (p.173) filter. The zero crossings that result are strongly influenced by the size of the Gaussian used for the smoothing stage of this operator. As the smoothing is increased then fewer and fewer zero crossing contours will be found, and those that do remain will correspond to features of larger and larger scale in the image.

How It Works

The core of the zero crossing detector is the Laplacian of Gaussian (p.173) filter and so a knowledge of that operator is assumed here. As described there, ‘edges’ in images give rise to zero crossings in the LoG output. For instance, Figure 11.8 shows the response of a 1-D LoG filter to a step edge in the image.

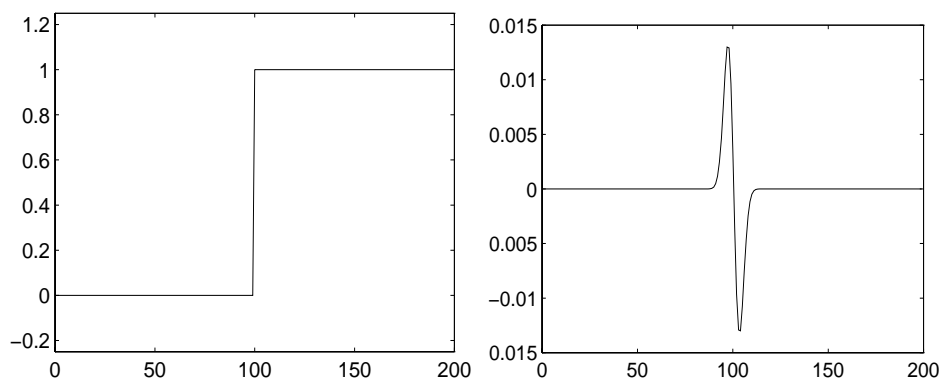


Figure 11.8: Response of 1-D LoG filter to a step edge. The left hand graph shows a 1-D image, 200 pixels long, containing a step edge. The right hand graph shows the response of a 1-D LoG filter with Gaussian standard deviation 3 pixels.

However, zero crossings also occur at any place where the image intensity gradient starts increasing or starts decreasing, and this may happen at places that are not obviously edges. Often zero crossings are found in regions of very low gradient where the intensity gradient wobbles up and down around zero.

Once the image has been LoG filtered, it only remains to detect the zero crossings. This can be done in several ways.

The simplest is to simply threshold (p.69) the LoG output at zero, to produce a binary image where the boundaries between foreground and background regions represent the locations of zero

crossing points. These boundaries can then be easily detected and marked in single pass, *e.g.* using some morphological operator (p.117). For instance, to locate all boundary points, we simply have to mark each foreground point that has at least one background neighbor.

The problem with this technique is that will tend to bias the location of the zero crossing edge to either the light side of the edge, or the dark side of the edge, depending on whether it is decided to look for the edges of foreground regions or for the edges of background regions.

A better technique is to consider points on both sides of the threshold boundary, and choose the one with the lowest absolute magnitude of the Laplacian, which will hopefully be closest to the zero crossing.

Since the zero crossings generally fall in between two pixels in the LoG filtered image, an alternative output representation is an image grid which is spatially shifted half a pixel across and half a pixel down, relative to the original image. Such a representation is known as a *dual lattice*. This does not actually localize the zero crossing any more accurately, of course.

A more accurate approach is to perform some kind of interpolation to estimate the position of the zero crossing to sub-pixel precision.

Guidelines for Use

The behavior of the LoG zero crossing edge detector is largely governed by the standard deviation of the Gaussian used in the LoG filter (p.173). The higher this value is set, the more smaller features will be smoothed out of existence, and hence fewer zero crossings will be produced. Hence, this parameter can be set to remove unwanted detail or noise as desired. The idea that at different smoothing levels different sized features become prominent is referred to as 'scale'.

We illustrate this effect using `cln1` which contains detail at a number of different scales.

The image `cln1log1` is the result of applying a LoG filter with Gaussian standard deviation 1.0. Note that in this and in the following LoG output images, the true output contains negative pixel values. For display purposes the graylevels have been offset so that displayed graylevel 128 corresponds to an actual value of zero, and rescaled (p.48) to make the image variation clearer. Since we are only interested in zero crossings this rescaling is unimportant.

The image `cln1zer1` shows the zero crossings from this image. Note the large number of minor features detected, which are mostly due to noise or very faint detail. This smoothing corresponds to a fine 'scale'.

The image `cln1log2` is the result of applying a LoG filter with Gaussian standard deviation 2.0 and `cln1zer3` shows the zero crossings. Note that there are far fewer detected crossings, and that those that remain are largely due to recognizable edges in the image. The thin vertical stripes on the wall, for example, are clearly visible.

Finally, `cln1log3` is the output from a LoG filter with Gaussian standard deviation 3.0. This corresponds to quite a coarse 'scale'. The image `cln1zer4` is the zero crossings in this image. Note how only the strongest contours remain, due to the heavy smoothing. In particular, note how the thin vertical stripes on the wall no longer give rise to many zero crossings.

All edges detected by the zero crossing detector are in the form of closed curves in the same way that contour lines on a map are always closed. The only exception to this is where the curve goes off the edge of the image.

Since the LoG filter is calculating a second derivative of the image, it is quite susceptible to noise, particularly if the standard deviation of the smoothing Gaussian is small. Thus it is common to see lots of spurious edges detected away from any obvious edges. One solution to this is to increase the smoothing of the Gaussian to preserve only strong edges. Another is to look at the gradient of the LoG at the zero crossing (*i.e.* the third derivative of the original image) and only keep zero crossings where this is above a certain threshold. This will tend to retain only the stronger edges, but it is sensitive to noise, since the third derivative will greatly amplify any high frequency noise in the image.

The image `cln1zer2` is similar to the image obtained with a standard deviation of 1.0, except that the zero crossing detector has been told to ignore zero crossings of shallow slope (in fact it ignores zero crossings where the pixel value difference across the crossing in the LoG output is less than 40). As a result, fewer spurious zero crossings have been detected. Note that, in this case, the zero crossings do not necessarily form closed contours.

Marr (1982) has suggested that human visual systems use zero crossing detectors based on LoG filters at several different scales (Gaussian widths).

Exercises

1. Compare the output from the zero crossing edge detector with that from the Roberts Cross (p.184), Sobel (p.188) and Canny (p.192) edge detectors, for edge detection, noise rejection and edge localization.
2. Take a simple image containing step edges such as `t1s1`, and see what happens to the locations of zero crossings as the level of smoothing is increased. Do they keep the same positions?
3. Comment on the way in which zero crossings disappear as smoothing is increased.
4. Try and develop an algorithm which can work out which side (positive or negative) of a particular discrete zero crossing is closer to the genuine zero crossing, and hence which should be marked as part of the zero crossing contour. Think about various possible 3×3 neighborhoods.
5. Think of an interpolation method which would allow you to estimate the zero crossing location between two pixels to sub-pixel precision.

References

- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, p 442.
- D. Marr** *Vision*, Freeman, 1982, pp 54 - 78.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, Chap. 5.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

11.6 Line Detection

Brief Description

While edges (p.230) (*i.e.* boundaries between regions with relatively distinct graylevels) are by far the most common type of discontinuity in an image, instances of thin lines in an image occur frequently enough that it is useful to have a separate mechanism for detecting them. Here we present a convolution (p.227) based technique which produces an image description of the thin lines in an input image. Note that the Hough transform (p.214) can be used to detect lines; however, in that case, the output is a *parametric* description of the lines in an image.

How It Works

The line detection operator consists of a convolution kernel (p.227) tuned to detect the presence of lines of a particular width n , at a particular orientation θ . Figure 11.9 shows a collection of four such kernels, which each respond to lines of single pixel width at the particular orientation shown.

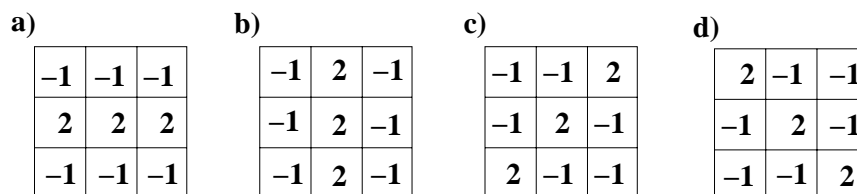


Figure 11.9: Four line detection kernels which respond maximally to horizontal, vertical and oblique (+45 and -45 degree) single pixel wide lines.

If R_i denotes the response of kernel i , we can apply each of these kernels across an image, and for any particular point, if $|R_i| > |R_j|$ for all $j \neq i$ that point is more likely to contain a line whose orientation (and width) corresponds to that of kernel i . One usually thresholds (p.69) R_i to eliminate weak lines corresponding to edges and other features with intensity gradients which have a different scale than the desired line width. In order to find complete lines, one must join together line fragments, *e.g.*, with an *edge tracking* operator.

Guidelines for Use

To illustrate line detection, we start with the artificial image `art2`, which contains thick line segments running horizontally, vertically and obliquely across the image. The result of applying the line detection operator, using the horizontal convolution kernel shown in Figure 11.9.a, is `art21dh1`. (Note that this gradient image has been *normalized* for display.) There are two points of interest to note here.

1. Notice that, because of way that the oblique lines (and some ‘vertical’ ends of the horizontal bars) are represented on a square pixel grid, *e.g.* `art2crp1` shows a zoomed (p.90) region containing both features, the horizontal line detector responds to more than high spatial intensity horizontal line-like features, *e.g.* `art2crp2`.
2. On an image such as this one, where the lines to be detected are wider than the kernel (*i.e.* the image lines are five pixels wide, while the kernel is tuned for a single width pixel), the line detector acts like an edge detector: the edges of the lines are found, rather than the lines themselves.

This latter fact might cause us to naively think that the image which gave rise to `art21dh1` contained a series of parallel lines rather than single thick ones. However, if we compare this result

to that obtained by applying the line detection kernel to an image containing lines of a single pixel width, we find some consistent differences. For example, we can skeletonize (p.145) the original `art2sk11` (so as to obtain a representation of the original wherein most lines are a single pixel width), apply the horizontal line detector `art21dh2`, and then threshold the result `art21dh3`. If we then threshold the original line detected image at the same pixel value, we obtain the null image `art21dh4`. Thus, the R_i values corresponding to the true, single pixel *lines* found in the skeletonized version are stronger than those R_i values corresponding to *edges*. Also, if we examine a cropped and zoomed (p.90) version of the line detected raw image `art2crp3` and the skeletonized line detected image `art2crp4` we see that the single pixel width lines are distinguished by a region of minimal response on either side of the maximal response values coincident with the pixel location of a line. One can use this signature to distinguish lines from edges.

The results of line detecting (and then normalizing) the skeletonized version of this image with single pixel width convolution kernels of different θ are `art21dv2` for a vertical kernel, `art21dp2` for the oblique 45 degree line, and `art21dn2` for the oblique 135 degree line. The thresholded versions are `art21dv1`, `art21dp1`, and `art21dn1`, respectively. We can add these together to produce a reasonably faithful binary representation of the line locations `art2add1`.

It is instructive to compare the two operators under more realistic circumstances, *e.g.* with the natural image `brg2`. After converting this to a grayscale image `brg3` and applying the Canny operator (p.192), we obtain `brg3can1`. Applying the line detector yields `brg3lda1`. We can improve this result by using a trick employed by the Canny operator (p.192). By smoothing (p.150) the image before line detecting, we obtain the cleaner result `brg3add2`. However, even with this preprocessing, the line detector still gives a poor result compared to the edge detector. This is true because there are few single pixel width lines in this image, and therefore the detector is responding to the other high spatial frequency image features (*i.e.* edges, thick lines and noise). (Note that in the previous example, the image contained the feature that the kernel was tuned for and therefore we were able to threshold away the weaker kernel response to edges.) We could improve this result by increasing the width of the kernel or geometrically scaling (p.90) the image.

Exercises

1. Consider the basic image `rob1`. We can investigate the scale of features in the image by applying line detection kernels of different widths. For example, after convolving with a single pixel horizontal line detecting kernel we discover that only the striped shirt of the bank robber contains single pixel width lines. The normalized result is shown in `rob1ldh1` and after thresholding (p.69) (at a value of 254), we obtain `rob1ldh2`. **a)** Perform the same analysis on the image `hse1` using different width kernels to extract the different features (*e.g.* roof, windows, doors, *etc.*). Threshold your result so that the final images contain a binary description of just the feature of interest. **b)** Try your kernels on other architectural drawings such as `hse2` and `hse4`.
2. Investigate a line detection algorithm which might extract the tail feathers of the peacock in `pea1`. You will most likely need to apply some smoothing as a first step and you may then want apply several different kernels and add the results together. Compare your final result with an edge detection algorithm, *e.g.* Roberts cross (p.184), Sobel (p.188), Compass (p.195) and/or Canny (p.192) edge detector.

References

- D. Vernon** *Machine Vision*, Prentice-Hall, 1991, Chap. 5.
- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 415 - 416.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

Chapter 12

Image Transforms

Most of the image processing operators demonstrated in HIPR transform an input image to form a new image. However, the operators demonstrated in this section produce output images whose character is generally quite different from the character of the input images. This difference might be in the geometry of the information in the image or the nature of the information itself. Of course, the size of the output image might be quite different from that of the input image (and the values at each pixel might be different, as in the complex number output of the Fourier transform).

The usual purpose of applying a transformation is to help make more obvious or explicit some desired information. In the case of the operators in this group, the main effect is to make explicit:

- *Distance Transform*, the thickness of image features
- *Fourier Transform*, the spatial frequency composition of the image
- *Hough Transform*, the parameters of geometric shapes.

The transformation is often followed by a thresholding (p.69) operation, which is intended to select the most prominent or relevant features. It may then be possible to apply an inverse transform, for the purpose of reconstructing the geometry of the original image, except with the desired features explicit or enhanced.

12.1 Distance Transform

Brief Description

The distance transform is an operator normally only applied to binary images. The result of the transform is a graylevel image that looks similar to the input image, except that the graylevel intensities of points inside foreground regions are changed to show the distance to the closest boundary from each point.

One way to think about the distance transform is to first imagine that foreground regions in the input binary image are made of some uniform slow burning inflammable material. Then consider simultaneously starting a fire at all points on the boundary of a foreground region and letting the fire burn its way into the interior. If we then label each point in the interior with the amount of time that the fire took to first reach that point, then we have effectively computed the distance transform of that region. Figure 12.1 shows a distance transform for a simple rectangular shape.

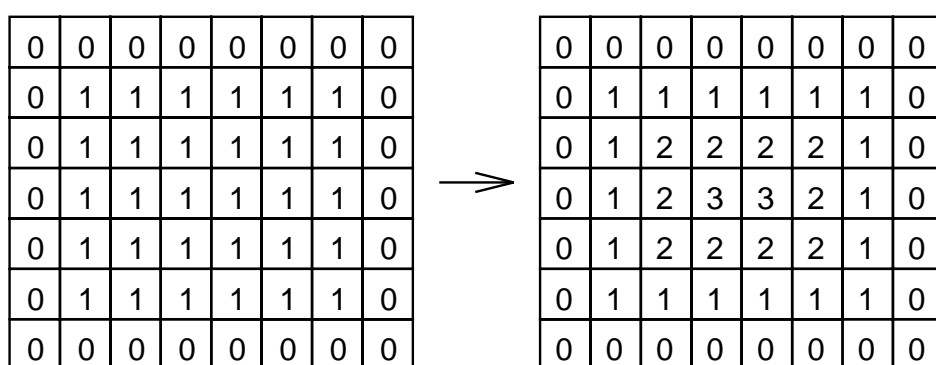


Figure 12.1: The distance transform of a simple shape. Note that we are using the ‘chessboard’ distance metric (p.229).

There is a dual to the distance transform described above which produces the distance transform for the background region rather than the foreground region. It can be considered as a process of inverting the original image and then applying the standard transform as above.

How It Works

There are several different sorts of distance transform, depending upon which distance metric (p.229) is being used to determine the distance between pixels. The example shown in Figure 12.1 uses the ‘chessboard’ distance metric but both the Euclidean and ‘city block’ metrics can be used as well.

Even once the metric has been chosen, there are many ways of computing the distance transform of a binary image. One intuitive but extremely inefficient way of doing it is to perform multiple successive erosions (p.123) with a suitable structuring element (p.241) until all foreground regions of the image have been eroded away. If each pixel is labeled with the number of erosions that had to be performed before it disappeared, then this is just the distance transform. The actual structuring element that should be used depends upon which distance metric has been chosen. A 3×3 square element gives the ‘chessboard’ distance transform, a cross shaped element gives the ‘city block’ distance transform, and a disk shaped element gives the Euclidean distance transform. Of course it is not actually possible to generate a good disk shaped element on a discrete grid on a small scale, but there are algorithms that vary the structuring element on each erosion so as to approximate a circular element.

The distance transform can be calculated much more efficiently using clever algorithms in only two

passes (*e.g.* Rosenfeld and Pfaltz 1968). This algorithm, which is based on recursive morphology, will not be described here.

Guidelines for Use

The distance transform is very closely linked to both the medial axis transform and to skeletonization (p.145). It can also be used to derive various other symmetries from binary shapes. As such it is usually only used as a step on the way to producing these end products (and in fact is often only produced in theory rather than in practice).

Here we illustrate the Euclidean distance transform with some examples.

The binary image `art5` becomes `art5dst1` when a distance transform is applied (scaled (p.48) by a factor of 5).

Similarly, `art6` becomes `art6dst1` (scaled by a factor of 3).

And finally, `art7` becomes `art7dst1` (scaled by a factor of 4).

The distance transform is sometimes very sensitive to small changes in the object. If, for example, we change the above rectangle to `art5cha2`, which contains a small black region in the center of the white rectangle, then the distance transform becomes `art5dst3` (after brightening the image by a factor of 6). This can be of advantage when we want to distinguish between similar objects like the two different rectangles above. However, it can also cause problems when trying to classify objects into classes of roughly the same shape. It also makes the distance transform very sensitive to noise (p.221). For instance, if we add some ‘pepper noise’ to the above rectangle, as in `art5noi1`, the distance transform yields `art5dst2` (brightened by a factor of 15).

An example of applying the distance transform to a real world image is illustrated with `phn1`. To obtain a binary input image, we threshold (p.69) the image at a value of 100, as shown in `phn1thr1`. The scaled (factor 6) distance transform is `phn1dst1`. Although the image gives a rough measure for the width of the object at each point, it is quite inaccurate at places where the object is incorrectly segmented from the background.

The last three examples show that it is important that the binary input image is a good representation of the object that we want to process. Simple thresholding (p.69) is often not enough. It might be necessary to further process the image before applying the distance transform.

Exercises

1. Try to obtain a better binary image of the telephone receiver so that the distance transform gives a better result. Consider applying some other morphological operators (p.117) (*e.g.* closing (p.130)) to the thresholded image.
2. Imagine representing the distance transform in 3-D, *i.e.* displaying the distance to the nearest boundary point on a third axis. What shape is the Euclidean distance transform of a circle?
3. Discuss the differences between the distance transforms using ‘city block’, ‘chessboard’ and Euclidean distance metrics. Under what situations are they different from each other? How do they vary on the example images above?
4. Why might you choose to use one distance metric over another?

References

A. Jain *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989, Chap. 9.

R. Haralick and L. Shapiro *Computer and Robot Vision*, Vol. 1, Addison-Wesley Publishing Company, 1992, Chap. 5.

A. Rosenfeld and J. Pfaltz *Distance Functions in Digital Pictures*, Pattern Recognition, Vol. 1, 1968, pp 33 - 61.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

12.2 Fourier Transform

Brief Description

The Fourier Transform is an important image processing tool which is used to decompose an image into its sine and cosine components. The output of the transformation represents the image in the *Fourier* or frequency domain (p.232), while the input image is the spatial domain (p.240) equivalent. In the Fourier domain image, each point represents a particular frequency contained in the spatial domain image.

The Fourier Transform is used in a wide range of applications, such as image analysis, image filtering, image reconstruction and image compression.

How It Works

As we are only concerned with digital images, we will restrict this discussion to the *Discrete Fourier Transform* (DFT).

The DFT is the sampled Fourier Transform and therefore does not contain all frequencies forming an image, but only a set of samples which is large enough to fully describe the spatial domain image. The number of frequencies corresponds to the number of pixels in the spatial domain image, *i.e.* the image in the spatial and Fourier domain are of the same size.

For a square image of size $N \times N$, the two-dimensional DFT is given by:

$$F(k, l) = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f(i, j) e^{-i2\pi(\frac{ki}{N} + \frac{lj}{N})}$$

where $f(i, j)$ is the image in the spatial domain and the exponential term is the basis function corresponding to each point $F(k, l)$ in the Fourier space. The equation can be interpreted as: the value of each point $F(k, l)$ is obtained by multiplying the spatial image with the corresponding base function and summing the result.

The basis functions are sine and cosine waves with increasing frequencies, *i.e.* $F(0, 0)$ represents the DC-component of the image which corresponds to the average brightness and $F(N-1, N-1)$ represents the highest frequency.

In a similar way, the Fourier image can be re-transformed to the spatial domain. The inverse Fourier transform is given by:

$$f(i, j) = \frac{1}{N^2} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} F(k, l) e^{i2\pi(\frac{ki}{N} + \frac{lj}{N})}$$

To obtain the result for the above equations, a double sum has to be calculated for each image point. However, because the Fourier Transform is *separable*, it can be written as

$$F(k, l) = \frac{1}{N} \sum_{j=0}^{N-1} P(k, j) e^{-i2\pi \frac{lj}{N}}$$

where

$$P(k, j) = \frac{1}{N} \sum_{i=0}^{N-1} f(i, j) e^{-i2\pi \frac{ki}{N}}$$

Using these two formulas, the spatial domain image is first transformed into an intermediate image using N one-dimensional Fourier Transforms. This intermediate image is then transformed into the final image, again using N one-dimensional Fourier Transforms. Expressing the two-dimensional

Fourier Transform in terms of a series of $2N$ one-dimensional transforms decreases the number of required computations.

Even with these computational savings, the ordinary one-dimensional DFT has N^2 complexity. This can be reduced to $N \log_2 N$ if we employ the *Fast Fourier Transform* (FFT) to compute the one-dimensional DFTs. This is a significant improvement, in particular for large images. There are various forms of the FFT and most of them restrict the size of the input image that may be transformed, often to $N = 2^n$ where n is an integer. The mathematical details are well described in the literature.

The Fourier Transform produces a complex number valued output image which can be displayed with two images, either with the *real* and *imaginary* part or with *magnitude* and *phase*. In image processing, often only the magnitude of the Fourier Transform is displayed, as it contains most of the information of the geometric structure of the spatial domain image. However, if we want to re-transform the Fourier image into the correct spatial domain after some processing in the frequency domain, we must make sure to preserve both magnitude and phase of the Fourier image.

The Fourier domain image has a much greater range than the image in the spatial domain. Hence, to be sufficiently accurate, its values are usually calculated and stored in float values.

Guidelines for Use

The Fourier Transform is used if we want to access the geometric characteristics of a spatial domain image. Because the image in the Fourier domain is decomposed into its sinusoidal components, it is easy to examine or process certain frequencies of the image, thus influencing the geometric structure in the spatial domain.

In most implementations the Fourier image is shifted in such a way that the DC-value (*i.e.* the image mean) $F(0,0)$ is displayed in the center of the image. The further away from the center an image point is, the higher is its corresponding frequency.

We start off by applying the Fourier Transform of `c1n1`. The magnitude calculated from the complex result is shown in `c1n1fur1`. We can see that the DC-value is by far the largest component of the image. However, the dynamic range of the Fourier coefficients (*i.e.* the intensity values in the Fourier image) is too large to be displayed on the screen, therefore all other values appear as black. If we apply a logarithmic transformation (p.82) to the image we obtain `c1n1fur2`. The result shows that the image contains components of all frequencies, but that their magnitude gets smaller for higher frequencies. Hence, low frequencies contain more image information than the higher ones. The transform image also tells us that there are two dominating directions in the Fourier image, one passing vertically and one horizontally through the center. These originate from the regular patterns in the background of the original image.

The phase of the Fourier transform of the same image is shown in `c1n1fur3`. The value of each point determines the phase of the corresponding frequency. As in the magnitude image, we can identify the vertical and horizontal lines corresponding to the patterns in the original image. The phase image does not yield much new information about the structure of the spatial domain image; therefore, in the following examples, we will restrict ourselves to displaying only the magnitude of the Fourier Transform.

Before we leave the phase image entirely, however, note that if we apply the inverse Fourier Transform to the above magnitude image while ignoring the phase (and then histogram equalize (p.78) the output) we obtain `c1n1fil1`. Although this image contains the same frequencies (and amount of frequencies) as the original input image, it is corrupted beyond recognition. This shows that the phase information is crucial to reconstruct the correct image in the spatial domain.

We will now experiment with some simple images to better understand the nature of the transform. The response of the Fourier Transform to periodic patterns in the spatial domain images can be seen very easily in the following artificial images.

The image `stp2` shows 2 pixel wide vertical stripes. The Fourier transform of this image is shown in `stp2fur1`. If we look carefully, we can see that it contains 3 main values: the DC-value and,

since the Fourier image is symmetrical to its center, two points corresponding to the frequency of the stripes in the original image. Note that the two points lie on a horizontal line through the image center, because the image intensity in the spatial domain changes the most if we go along it horizontally.

The distance of the points to the center can be explained as follows: the maximum frequency which can be represented in the spatial domain are one pixel wide stripes.

$$f_{max} = \frac{1}{1 \text{ pixel}}$$

Hence, the two pixel wide stripes in the above image represent

$$f = \frac{1}{2 \text{ pixel}} = \frac{f_{max}}{2}$$

Thus, the points in the Fourier image are halfway between the center and the edge of the image, *i.e.* the represented frequency is half of the maximum.

Further investigation of the Fourier image shows that the magnitude of other frequencies in the image is less than $\frac{1}{100}$ of the DC-value, *i.e.* they don't make any significant contribution to the image. The magnitudes of the two minor points are each two-thirds of the DC-value.

Similar effects as in the above example can be seen when applying the Fourier Transform to `stp1`, which consists of diagonal stripes. In `stp1fur1`, showing the magnitude of the Fourier Transform, we can see that, again, the main components of the transformed image are the DC-value and the two points corresponding to the frequency of the stripes. However, the logarithmic transform of the Fourier Transform, `stp1fur2`, shows that now the image contains many minor frequencies. The main reason is that a diagonal can only be approximated by the square pixels of the image, hence, additional frequencies are needed to compose the image. The logarithmic scaling makes it difficult to tell the influence of single frequencies in the original image. To find the most important frequencies we threshold (p.69) the original Fourier image at level 13. The resulting Fourier image, `stp1fur3`, shows all frequencies whose magnitude is at least 5% of the main peak. Compared to the original Fourier image, several more points appear. They are all on the same diagonal as the three main components, *i.e.* they all originate from the periodic stripes. The represented frequencies are all multiples of the basic frequency of the stripes in the spatial domain image. This is because a rectangular signal, like the stripes, with the frequency f_{rect} is a composition of sine waves with the frequencies $f_{sine} = n \times f_{rect}$, known as the harmonics of f_{rect} . All other frequencies disappeared from the Fourier image, *i.e.* the magnitude of each of them is less than 5% of the DC-value.

A Fourier-Transformed image can be used for frequency filtering (p.167). A simple example is illustrated with the above image. If we multiply the (complex) Fourier image obtained above with an image containing a circle (of $r = 32$ pixels), we can set all frequencies larger than f_{rect} to zero as shown in the logarithmic transformed image `stp1fur5`. By applying the inverse Fourier Transform we obtain `stp1fil1`. The resulting image is a lowpass filtered version of the original spatial domain image. Since all other frequencies have been suppressed, this result is the sum of the constant DC-value and a sine-wave with the frequency f_{rect} . Further examples can be seen in the worksheet on frequency filtering (p.167).

A property of the Fourier Transform which is used, for example, for the removal of additive noise (p.221), is its *distributivity over addition*. We can illustrate this by adding (p.43) the complex Fourier images of the two previous example images. To display the result and emphasize the main peaks, we threshold the magnitude of the complex image, as can be seen in `stp1fur4`. Applying the inverse Fourier Transform to the complex image yields `stp1fil2`. According to the distributivity law, this image is the same as the direct sum of the two original spatial domain images.

Finally, we present an example (*i.e.* text orientation finding) where the Fourier Transform is used to gain information about the geometric structure of the spatial domain image. Text recognition using image processing techniques is simplified if we can assume that the text lines are in a predefined direction. Here we show how the Fourier Transform can be used to find the initial orientation of the text and then a rotation (p.93) can be applied to correct the error. We illustrate this technique using `son3`, a binary image of English text. The logarithm of the magnitude of its Fourier transform

is `son3fur2`, and `son3fur4` is the thresholded magnitude of the Fourier image. We can see that the main values lie on a vertical line, indicating that the text lines in the input image are horizontal. If we proceed in the same way with `son3rot1`, which was rotated about 45° , we obtain `son3fur1` and `son3fur3` in the Fourier space. We can see that the line of the main peaks in the Fourier domain is rotated according to rotation of the input image. The second line in the logarithmic image (perpendicular to the main direction) originates from the black corners in the rotated image.

Although we managed to find a threshold which separates the main peaks from the background, we have a reasonable amount of noise in the Fourier image resulting from the irregular pattern of the letters. We could decrease these background values and therefore increase the difference to the main peaks if we were able to form solid blocks out of the text-lines. This could, for example, be done by using a morphological operator (p.117).

Common Variants

Another sinusoidal transform (*i.e.* transform with sinusoidal base functions) related to the DFT is the *Discrete Cosine Transform* (DCT). For an $N \times N$ image, the DCT is given by

$$C(k, n) = \alpha(k, n) \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f(i, j) \cos\left(\frac{(2i+1)k\pi}{2N}\right) \cos\left(\frac{(2j+1)n\pi}{2N}\right)$$

with

$$\alpha(k, n) = \begin{cases} \frac{1}{N} & \text{for } k, n = 0 \\ \frac{2}{N} & \text{for } k, n = 1, 2, \dots, N-1 \end{cases}$$

The main advantages of the DCT are that it yields a real valued output image and that it is a fast transform. A major use of the DCT is in image compression — *i.e.* trying to reduce the amount of data needed to store an image. After performing a DCT it is possible to throw away the coefficients that encode high frequency components that the human eye is not very sensitive to. Thus the amount of data can be reduced, without seriously affecting the way an image looks to the human eye.

Exercises

1. Take the Fourier Transforms of `stp1` and `stp2` and add them using `blend` (p.53). Take the inverse Fourier Transform of the sum. Explain the result.
2. Using a paint program (p.233), create an image made of periodical patterns of varying frequency and orientation. Examine its Fourier Transform and investigate the effects of removing or changing some of the patterns in the spatial domain image.
3. Apply the mean (p.150) operator to `stp2` and compare its Fourier Transform before and after the operation.
4. Add different sorts of noise (p.221) to `cln1` and compare the Fourier Transforms with `cln1fur2`.
5. Use the `open` (p.127) operator to transform the text lines in the above images into solid blocks. Make sure that the chosen structuring element (p.241) works for all orientations of text. Compare the Fourier Transforms of the resulting images with the transforms of the unprocessed text images.
6. Investigate if the Fourier Transform is distributive over multiplication. To do so, multiply `stp1` with `stp2` and take the Fourier Transform. Compare the result with the multiplication of the two direct Fourier Transforms.

References

- D. Ballard and C. Brown** *Computer Vision*, Prentice-Hall, 1982, pp 24 - 30.
- R. Gonzales, R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, pp 81 - 125.
- B. Horn** *Robot Vision*, MIT Press, 1986, Chaps 6, 7.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989, pp 15 - 20.
- A. Marion** *An Introduction to Image Processing*, Chapman and Hall, 1991, Chap. 9.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

12.3 Hough Transform

Brief Description

The Hough transform is a technique which can be used to isolate features of a particular shape within an image. Because it requires that the desired features be specified in some parametric form, the *classical* Hough transform is most commonly used for the detection of regular curves such as lines, circles, ellipses, *etc.* A *generalized* Hough transform can be employed in applications where a simple analytic description of a feature(s) is not possible. Due to the computational complexity of the generalized Hough algorithm, we restrict the main focus of this discussion to the classical Hough transform. Despite its domain restrictions, the classical Hough transform (hereafter referred to without the *classical* prefix) retains many applications, as most manufactured parts (and many anatomical parts investigated in medical imagery) contain feature boundaries which can be described by regular curves. The main advantage of the Hough transform technique is that it is tolerant of gaps in feature boundary descriptions and is relatively unaffected by image noise.

How It Works

The Hough technique is particularly useful for computing a global description of a feature(s) (where the number of solution classes need not be known *a priori*), given (possibly noisy) local measurements. The motivating idea behind the Hough technique for line detection is that each input measurement (*e.g.* coordinate point) indicates its contribution to a globally consistent solution (*e.g.* the physical line which gave rise to that image point).

As a simple example, consider the common problem of fitting a set of line segments to a set of discrete image points (*e.g.* pixel locations output from an edge detector). Figure 12.2 shows some possible solutions to this problem. Here the lack of *a priori* knowledge about the number of desired line segments (and the ambiguity about what constitutes a line segment) render this problem under-constrained.

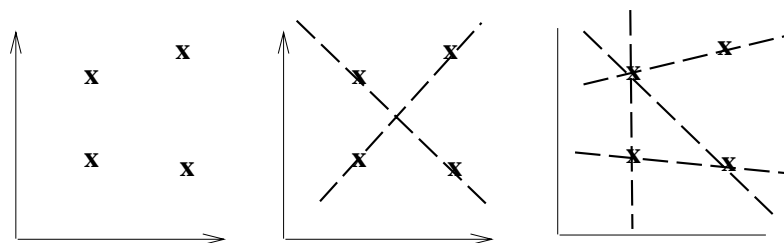


Figure 12.2: **a)** Coordinate points. **b)** and **c)** Possible straight line fittings.

We can analytically describe a line segment in a number of forms. However, a convenient equation for describing a set of lines uses *parametric* or *normal* notion:

$$x \cos \theta + y \sin \theta = r$$

where r is the length of a normal from the origin to this line and θ is the orientation of r with respect to the X-axis. (See Figure 12.3.) For any point (x, y) on this line, r and θ are constant.

In an image analysis context, the coordinates of the point(s) of edge segments (*i.e.* (x_i, y_i)) in the image are known and therefore serve as constants in the parametric line equation, while r and θ are the unknown variables we seek. If we plot the possible (r, θ) values defined by each (x_i, y_i) , points in cartesian image space map to curves (*i.e.* sinusoids) in the polar Hough parameter space. This *point-to-curve* transformation is the Hough transformation for straight lines. When viewed

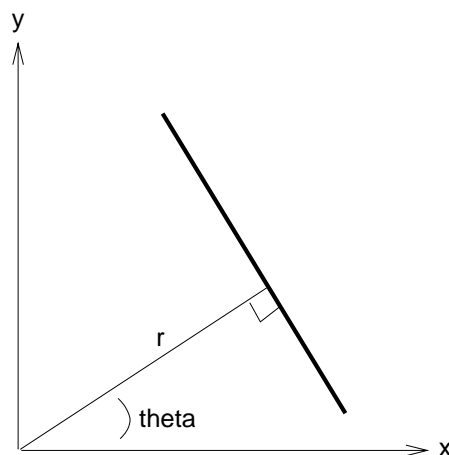


Figure 12.3: Parametric description of a straight line.

in Hough parameter space, points which are collinear in the cartesian image space become readily apparent as they yield curves which intersect at a common (r, θ) point.

The transform is implemented by quantizing the Hough parameter space into finite intervals or *accumulator cells*. As the algorithm runs, each (x_i, y_i) is transformed into a discretized (r, θ) curve and the accumulator cells which lie along this curve are incremented. Resulting peaks in the accumulator array represent strong evidence that a corresponding straight line exists in the image.

We can use this same procedure to detect other features with analytical descriptions. For instance, in the case of *circles*, the parametric equation is

$$(x - a)^2 + (y - b)^2 = r^2$$

where a and b are the coordinates of the center of the circle and r is the radius. In this case, the computational complexity of the algorithm begins to increase as we now have three coordinates in the parameter space and a 3-D accumulator. (In general, the computation and the size of the accumulator array increase polynomially with the number of parameters. Thus, the basic Hough technique described here is only practical for simple curves.)

Guidelines for Use

The Hough transform can be used to identify the parameter(s) of a curve which best fits a set of given edge points. This edge description is commonly obtained from a feature detecting operator such as the Roberts Cross (p.184), Sobel (p.188) or Canny (p.192) edge detector and may be noisy, *i.e.* it may contain multiple edge fragments corresponding to a single whole feature. Furthermore, as the output of an edge detector defines only *where* features are in an image, the work of the Hough transform is to determine both *what* the features are (*i.e.* to detect the feature(s) for which it has a parametric (or other) description) and *how many* of them exist in the image.

In order to illustrate the Hough transform in detail, we begin with the simple image of two occluding rectangles, `sqr1`. The Canny edge detector (p.192) can produce a set of boundary descriptions for this part, as shown in `sqr1can1`. Here we see the overall boundaries in the image, but this result tells us nothing about the identity (and quantity) of feature(s) within this boundary description. In this case, we can use the Hough (line detecting) transform to detect the eight separate straight line segments of this image and thereby identify the true geometric structure of the subject.

If we use these edge/boundary points as input to the Hough transform, a curve is generated in polar (r, θ) space for each edge point in cartesian space. The accumulator array, when viewed

as an intensity image, looks like `sqr1hou1`. Histogram equalizing (p.78) the image allows us to see the patterns of information contained in the low intensity pixel values, as shown in `sqr1hou2`. Note that, although r and θ are notionally polar coordinates, the accumulator space is plotted rectangularly with θ as the abscissa and r as the ordinate. Note that the accumulator space wraps around at the vertical edge of the image such that, in fact, there are only 8 real peaks.

Curves generated by collinear points in the gradient image intersect in peaks (r, θ) in the Hough transform space. These intersection points characterize the straight line segments of the original image. There are a number of methods which one might employ to extract these bright points, or *local maxima*, from the accumulator array. For example, a simple method involves thresholding (p.69) and then applying some thinning (p.137) to the isolated clusters of bright spots in the accumulator array image. Here we use a *relative threshold* to extract the unique (r, θ) points corresponding to each of the straight line edges in the original image. (In other words, we take only those local maxima in the accumulator array whose values are equal to or greater than some fixed percentage of the global maximum value.)

Mapping back from Hough transform space (*i.e. de-Houghing*) into cartesian space yields a set of line descriptions of the image subject. By overlaying this image on an inverted (p.63) version of the original, we can confirm the result that the Hough transform found the 8 true sides of the two rectangles and thus revealed the underlying geometry of the occluded scene `sqr1hou3`. Note that the accuracy of alignment of detected and original image lines, which is obviously not perfect in this simple example, is determined by the quantization of the accumulator array. (Also note that many of the image edges have several detected lines. This arises from having several nearby Hough-space peaks with similar line parameter values. Techniques exist for controlling this effect, but were not used here to illustrate the output of the standard Hough transform.)

Note also that the lines generated by the Hough transform are infinite in length. If we wish to identify the actual line segments which generated the transform parameters, further image analysis is required in order to see which portions of these infinitely long lines actually have points on them.

To illustrate the Hough technique's robustness to noise, the Canny edge description has been corrupted by 1% salt and pepper noise (p.221) `sqr1can2` before Hough transforming it. The result, plotted in Hough space, is `sqr1hou4`. De-Houghing this result (and overlaying it on the original) yields `sqr1hou5`. (As in the above case, the relative threshold is 40%.)

The sensitivity of the Hough transform to gaps in the feature boundary can be investigated by transforming the image `sqr1can3`, which has been edited using a paint program (p.233). The Hough representation is `sqr1hou6` and the de-Houghed image (using a relative threshold of 40%) is `sqr1hou7`. In this case, because the accumulator space did not receive as many entries as in previous examples, only 7 peaks were found, but these are all structurally relevant lines.

We will now show some examples with natural imagery. In the first case, we have a city scene where the buildings are obstructed in fog, `sff1sca1`. If we want to find the true edges of the buildings, an edge detector (*e.g.* Canny (p.192)) cannot recover this information very well, as shown in `sff1can1`. However, the Hough transform can detect some of the straight lines representing building edges within the obstructed region. The histogram equalized (p.78) accumulator space representation of the original image is shown in `sff1hou1`. If we set the relative threshold to 70%, we get the following de-Houghed image `sff1hou2`. Only a few of the long edges are detected here, and there is a lot of duplication where many lines or edge fragments are nearly colinear. Applying a more generous relative threshold, *i.e.* 50%, yields `sff1hou3` yields more of the expected lines, but at the expense of many spurious lines arising from the many colinear edge fragments.

Our final example comes from a remote sensing application. Here we would like to detect the streets in the image `urb1` of a reasonably rectangular city sector. We can edge detect the image using the Canny edge detector (p.192) as shown in `urb1can1`. However, street information is not available as output of the edge detector alone. The image `urb1hou1` shows that the Hough line detector is able to recover some of this information. Because the contrast in the original image is poor, a limited set of features (*i.e.* streets) is identified.

Common Variants

Generalized Hough Transform

The generalized Hough transform is used when the shape of the feature that we wish to isolate does not have a simple analytic equation describing its boundary. In this case, instead of using a parametric equation of the curve, we use a look-up table to define the relationship between the boundary positions and orientations and the Hough parameters. (The look-up table values must be computed during a preliminary phase using a prototype shape.)

For example, suppose that we know the shape and orientation of the desired feature. (See Figure 12.4.) We can specify an arbitrary reference point (x_{ref}, y_{ref}) within the feature, with respect to which the shape (*i.e.* the distance r and angle of normal lines drawn from the boundary to this reference point ω) of the feature is defined. Our look-up table (*i.e.* *R-table*) will consist of these distance and direction pairs, indexed by the orientation ω of the boundary.

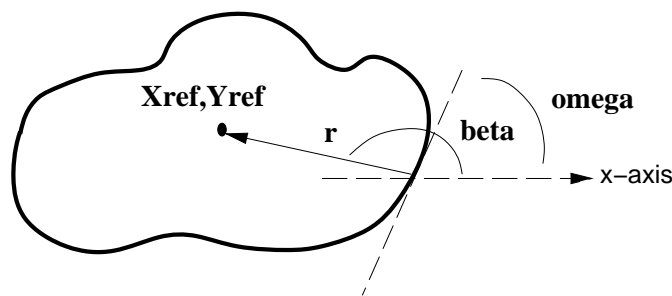


Figure 12.4: Description of R-table components.

The Hough transform space is now defined in terms of the possible positions of the shape in the image, *i.e.* the possible ranges of (x_{ref}, y_{ref}) . In other words, the transformation is defined by:

$$x_{ref} = x + r \cos(\beta)$$

$$y_{ref} = y + r \sin(\beta)$$

(The r and β values are derived from the R-table for particular known orientations ω .) If the orientation of the desired feature is unknown, this procedure is complicated by the fact that we must extend the accumulator by incorporating an extra parameter to account for changes in orientation.

Exercises

1. Find the Hough line transform of the objects shown in Figure 12.5.

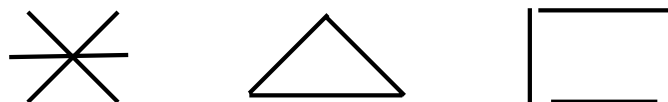


Figure 12.5: Features to input to the Hough transform line detector.

2. Starting from the basic image `art5`, create a series of images with which you can investigate the ability of the Hough line detector to extract occluded features. For example, begin using translation (p.97) and image addition (p.43) to create an image containing the original image

overlapped by a translated copy of that image. Next, use edge detection (p.230) to obtain a boundary description of your subject. Finally, apply the Hough algorithm to recover the geometries of the occluded features.

3. Investigate the robustness of the Hough algorithm to image noise. Starting from an edge detected version of the basic image `wdg3`, try the following: **a)** Generate a series of boundary descriptions of the image using different levels of Gaussian noise (p.221). How noisy (*i.e.* broken) does the edge description have to be before Hough is unable to detect the original geometric structure of the scene? **b)** Corrode the boundary descriptions with different levels of salt and pepper noise (p.221). At what point does the combination of broken edges and added intensity spikes render the Hough line detector useless?
4. Try the Hough transform line detector on the images: `pea1`, `pdc1` and `arp1`. Experiment with the Hough circle detector on `alg1`, `cel5`, `rck3` and `tom2`.
5. One way of reducing the computation required to perform the Hough transform is to make use of gradient information which is often available as output from an edge detector. In the case of the Hough circle detector, the edge gradient tells us in which direction a circle must lie from a given edge coordinate point. (See Figure 12.6.)

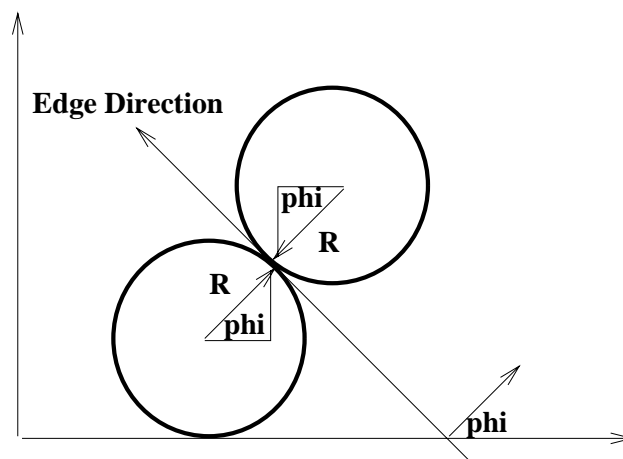


Figure 12.6: Hough circle detection with gradient information.

- a)** Describe how you would modify the 3-D circle detector accumulator array in order to take this information into account. **b)** To this algorithm we may want to add gradient magnitude information. Suggest how to introduce *weighted* incrementing of the accumulator.
6. The Hough transform can be seen as an efficient implementation of a generalized matched filter strategy. In other words, if we created a template composed of a circle of 1's (at a fixed r) and 0's everywhere else in the image, then we could convolve it with the gradient image to yield an accumulator array-like description of all the circles of radius r in the image. Show formally that the basic Hough transform (*i.e.* the algorithm with no use of gradient direction information) is equivalent to template matching.
7. Explain how to use the generalized Hough transform to detect octagons.

References

- D. Ballard and C. Brown** *Computer Vision*, Prentice-Hall, 1982, Chap. 4.
- R. Boyle and R. Thomas** *Computer Vision: A First Course*, Blackwell Scientific Publications, 1988, Chap. 5.

A. Jain *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989, Chap. 9.

D. Vernon *Machine Vision*, Prentice-Hall, 1991, Chap. 6.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

Chapter 13

Image Synthesis

Image synthesis is the process of creating new images from some form of image description. The kinds of images that are typically synthesized include:

- *Test Patterns*, Scenes with simple two dimensional geometric shapes.
- *Image Noise*, Images containing random pixel values, usually generated from specific parametrized distributions.
- *Computer Graphics*, Scenes or images based on geometric shape descriptions. Often the models are three-dimensional, but may also be two-dimensional.

Synthetic images are often used to verify the correctness of operators by applying them to known images. They are also often used for teaching purposes, as the operator output on such images is generally 'clean', whereas noise and uncontrollable pixel distributions in real images make it harder to demonstrate unambiguous results. The images could be binary, gray level or color.

13.1 Noise Generation

Brief Description

Real world signals usually contain departures from the ideal signal that would be produced by our model of the signal production process. Such departures are referred to as *noise*. Noise arises as a result of unmodelled or unmodellable processes going on in the production and capture of the real signal. It is not part of the ideal signal and may be caused by a wide range of sources, *e.g.* variations in the detector sensitivity, environmental variations, the discrete nature of radiation, transmission or quantization errors, *etc.* It is also possible to treat irrelevant scene details as if they are image noise (*e.g.* surface reflectance textures). The characteristics of noise depend on its source, as does the operator which best reduces its effects.

Many image processing packages contain operators to artificially add noise to an image. Deliberately corrupting an image with noise allows us to test the resistance of an image processing operator to noise and assess the performance of various noise filters.

How It Works

Noise can generally be grouped into two classes:

- independent noise.
- noise which is dependent on the image data.

Image independent noise can often be described by an additive noise model, where the recorded image $f(i,j)$ is the sum of the *true* image $s(i,j)$ and the noise $n(i,j)$:

$$f(i, j) = s(i, j) + n(i, j)$$

The noise $n(i,j)$ is often *zero-mean* and described by its variance σ_n^2 . The impact of the noise on the image is often described by the *signal to noise ratio* (SNR), which is given by

$$SNR = \frac{\sigma_s}{\sigma_n} = \sqrt{\frac{\sigma_f^2}{\sigma_n^2} - 1}$$

where σ_s^2 and σ_f^2 are the variances of the true image and the recorded image, respectively.

In many cases, additive noise is evenly distributed over the frequency domain (p.209) (*i.e.* *white noise*), whereas an image contains mostly low frequency information. Hence, the noise is dominant for high frequencies and its effects can be reduced using some kind of lowpass filter. This can be done either with a frequency filter (p.167) or with a spatial filter (p.148). (Often a spatial filter is preferable, as it is computationally less expensive than a frequency filter.)

In the second case of *data-dependent noise* (*e.g.* arising when monochromatic radiation is scattered from a surface whose roughness is of the order of a wavelength, causing wave interference which results in image *speckle*), it is possible to model noise with a multiplicative, or non-linear, model. These models are mathematically more complicated; hence, if possible, the noise is assumed to be data independent.

Detector Noise

One kind of noise which occurs in all recorded images to a certain extent is *detector noise*. This kind of noise is due to the discrete nature of radiation, *i.e.* the fact that each imaging system is recording an image by counting photons. Allowing some assumptions (which are valid for many applications) this noise can be modeled with an independent, additive model, where the noise $n(i,j)$ has a zero-mean Gaussian distribution described by its standard deviation (σ), or variance. (The

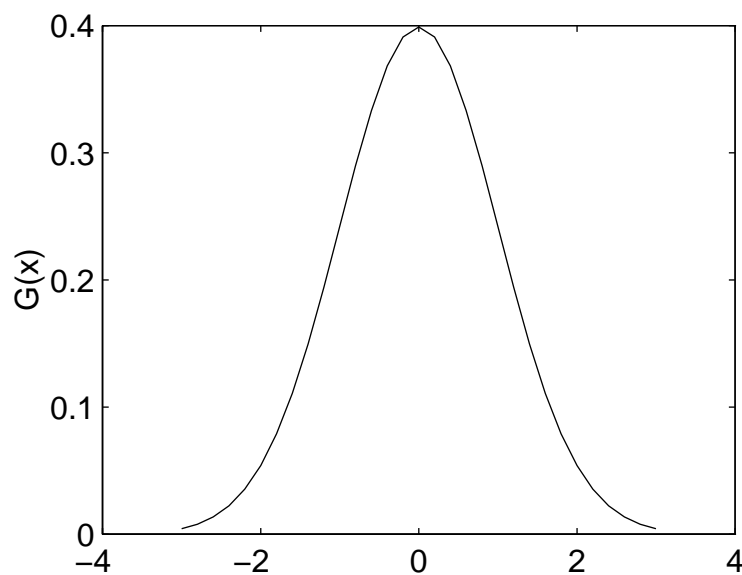


Figure 13.1: 1-D Gaussian distribution with mean 0 and standard deviation 1

1-D Gaussian distribution has the form shown in Figure 13.1.) This means that each pixel in the noisy image is the sum of the true pixel value and a random, Gaussian distributed noise value.

Salt and Pepper Noise

Another common form of noise is *data drop-out* noise (commonly referred to as *intensity spikes*, *speckle* or *salt and pepper noise*). Here, the noise is caused by errors in the data transmission. The corrupted pixels are either set to the maximum value (which looks like snow in the image) or have single bits flipped over. In some cases, single pixels are set alternatively to zero or to the maximum value, giving the image a ‘salt and pepper’ like appearance. Unaffected pixels always remain unchanged. The noise is usually quantified by the percentage of pixels which are corrupted.

Guidelines for Use

In this section we will show some examples of images corrupted with different kinds of noise and give a short overview of which noise reduction operators are most appropriate. A fuller discussion of the effects of the operators is given in the corresponding worksheets.

Gaussian Noise

We will begin by considering additive noise with a Gaussian distribution. If we add Gaussian noise with σ values of 8, we obtain the image `fce5noi4`. Increasing σ yields `fce5noi5` and `fce5noi6` for $\sigma=13$ and 20. Compare these images to the original `fce5`.

Gaussian noise can be reduced using a spatial filter. However, it must be kept in mind that when smoothing an image, we reduce not only the noise, but also the fine-scaled image details because they also correspond to blocked high frequencies. The most effective basic spatial filtering techniques for noise removal include: mean filtering (p.150), median filtering (p.153) and Gaussian smoothing (p.156). Crimmins Speckle Removal (p.164) filter can also produce good noise removal. More sophisticated algorithms which utilize statistical properties of the image and/or noise fields exist for noise removal. For example, adaptive smoothing algorithms may be defined which adjust the filter response according to local variations in the statistical properties of the data.

Salt and Pepper Noise

In the following examples, images have been corrupted with various kinds and amounts of drop-out noise. In `fce5noi3`, pixels have been set to 0 or 255 with probability $p=1\%$. In `fce5noi7` pixel bits were flipped with $p=3\%$, and in `fce5noi8` 5% of the pixels (whose locations are chosen at random) are set to the maximum value, producing the snowy appearance.

For this kind of noise, conventional lowpass filtering, *e.g.* mean filtering (p.150) or Gaussian smoothing (p.156) is relatively unsuccessful because the corrupted pixel value can vary significantly from the original and therefore the mean can be significantly different from the true value. A median filter (p.153) removes drop-out noise more efficiently and at the same time preserves the edges and small details in the image better. Conservative smoothing (p.161) can be used to obtain a result which preserves a great deal of high frequency detail, but is only effective at reducing low levels of noise.

Exercises

1. The image `che1noi1` is a binary chessboard image with 2% of drop-out noise. Which operator yields the best results in removing the noise?
2. The image `che1noi2` is the same image corrupted with Gaussian noise with a variance of 180. Is the operator used in *Exercise 1* still the most appropriate? Compare the best results obtained from both noisy images.
3. Compare the images achieved by median filter (p.153) and mean filter (p.150) filtering `fce5noi5` with the result that you obtain by applying a frequency lowpass filter (p.167) to the image. How does the mean filter relate to the frequency filter? Compare the computational costs of mean, median and frequency filtering.

References

- R. Gonzales and R. Woods** *Digital Image Processing*, Addison Wesley, 1992, pp 187 - 213.
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice Hall, 1989, pp 244 - 253, 273 - 275.
- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, pp 29 - 30, 40 - 47, 493.
- B. Horn** *Robot Vision*, MIT Press, 1986, Chap. 2.
- A. Marion** *An Introduction to Image Processing*, Chapman and Hall, 1991, Chap. 5.

Local Information

Information relevant to your local image processing setup can be added here by the person who maintains your HIPR system. This is merely the default message.

More general advice about the local HIPR installation is available in the *Local Information* (p.40) introductory section.

Part III

Other User Information and Resources

Appendix A

A to Z of Image Processing Concepts

A.1 Binary Images

Binary images are images whose pixels (p.238) have only two possible intensity values (p.239). They are normally displayed as black and white. Numerically, the two values are often 0 for black, and either 1 or 255 for white.

Binary images are often produced by thresholding (p.69) a grayscale (p.232) or color image (p.225), in order to separate an object in the image from the background. The color of the object (usually white) is referred to as the *foreground color*. The rest (usually black) is referred to as the *background color*. However, depending on the image which is to be thresholded, this *polarity* might be inverted, in which case the object is displayed with 0 and the background is with a non-zero value.

Some morphological (p.117) operators assume a certain polarity of the binary input image so that if we process an image with inverse polarity the operator will have the opposite effect. For example, if we apply a closing (p.130) operator to a black text on white background, the text will be opened (p.127).

A.2 Color Images

It is possible to construct (almost) all visible colors by combining the three primary colors (p.240) red, green and blue, because the human eye has only three different color receptors, each of them sensible to one of the three colors. Different combinations in the stimulation of the receptors enable the human eye to distinguish approximately *350000* colors. A RGB (p.240) color image is a multi-spectral (p.237) image with one band for each color red, green and blue, thus producing a weighted combination of the three primary colors for each pixel.

A full 24-bit color (p.226) image contains one 8-bit value for each color, thus being able to display $2^{24} = 16,777,216$ different colors.

However, it is computationally expensive and often not necessary to use the full 24-bit image to store the color for each pixel. Therefore, the color for each pixel is often encoded in a single byte, resulting in an 8-bit color (p.226) image. The process of reducing the color representation from 24-bits to 8-bits, known as color quantization (p.227), restricts the number of possible colors to 256. However, there is normally no visible difference between a 24-color image and the same image displayed with 8 bits. An 8-bit color images are based on colormaps (p.235), which are *look-up tables* taking the 8-bit pixel value as index and providing an output value for each color.

A.3 8-bit Color Images

Full RGB (p.240) color requires that the intensities of three color components be specified for each and every pixel. It is common for each component intensity to be stored as an 8-bit integer, and so each pixel requires 24 bits to completely and accurately specify its color. If this is done, then the image is known as a 24-bit color image (p.226). However there are two problems with this approach:

- Storing 24 bits for every pixel leads to very large image files that with current technology are cumbersome to store and manipulate. For instance a 24-bit 512×512 image takes up 750KB in uncompressed form.
- Many monitor displays use colormaps (p.235) with 8-bit index numbers, meaning that they can only display 256 different colors at any one time. Thus it is often wasteful to store more than 256 different colors in an image anyway, since it will not be possible to display them all on screen.

Because of this, many image formats (*e.g.* 8-bit GIF and TIFF) use 8-bit colormaps (p.235) to restrict the maximum number of different colors to 256. Using this method, it is only necessary to store an 8-bit index into the colormap for each pixel, rather than the full 24-bit color value. Thus 8-bit image formats consist of two parts: a colormap describing what colors are present in the image, and the array of index values for each pixel in the image.

When a 24-bit full color image is turned into an 8-bit image, it is usually necessary to throw away some of the colors, a process known as color quantization (p.227). This leads to some degradation in image quality, but in practice the observable effect can be quite small, and in any case, such degradation is inevitable if the image output device (*e.g.* screen or printer) is only capable of displaying 256 colors or less.

The use of 8-bit images with colormaps does lead to some problems in image processing. First of all, each image has to have its own colormap, and there is usually no guarantee that each image will have exactly the same colormap. Thus on 8-bit displays it is frequently impossible to correctly display two different color images that have different colormaps *at the same time*. Note that in practice 8-bit images often use reduced size colormaps with less than 256 colors in order to avoid this problem.

Another problem occurs when the output image from an image processing operation contains different colors to the input image or images. This can occur very easily, as for instance when two color images are added together (p.43) pixel-by-pixel. Since the output image contains different colors from the input images, it ideally needs a new colormap, different from those of the input images, and this involves further color quantization which will degrade the image quality. Hence the resulting output is usually only an approximation of the desired output. Repeated image processing operations will continually degrade the image colors. And of course we still have the problem that it is not possible to display the images simultaneously with each other on the same 8-bit display.

Because of these problems it is to be expected that as computer storage and processing power become cheaper, there will be a shift away from 8-bit images and towards full 24-bit image processing.

A.4 24-bit Color Images

Full RGB (p.240) color requires that the intensities of three color components be specified for each and every pixel. It is common for each component intensity to be stored as an 8-bit integer, and so each pixel requires 24 bits to completely and accurately specify its color. Image formats that store a full 24 bits to describe the color of each and every pixel are therefore known as *24-bit color images*.

Using 24 bits to encode color information allows $2^{24} = 16,777,216$ different colors to be represented, and this is sufficient to cover the full range of human color perception fairly well.

The term 24-bit is also used to describe monitor displays that use 24 bits per pixel in their display memories, and which are hence capable of displaying a full range of colors.

There are also some disadvantages to using 24-bit images. Perhaps the main one is that it requires three times as much memory, disk space and processing time to store and manipulate 24-bit color images as compared to 8-bit color images (p.226). In addition, there is often not much point in being able to store all those different colors if the final output device (*e.g.* screen or printer) can only actually produce a fraction of them. Since it is possible to use colormaps (p.235) to produce 8-bit color images (p.226) that look almost as good, at the time of writing 24-bit displays are relatively little used. However it is to be expected that as the technology becomes cheaper, their use in image processing will grow.

A.5 Color Quantization

Color quantization is applied when the color information of an image is to be reduced. The most common case is when a 24-bit color (p.226) image is transformed into an 8-bit color (p.226) image.

Two decisions have to be made:

1. which colors of the larger color set remain in the new image, and
2. how are the discarded colors mapped to the remaining ones.

The simplest way to transform a 24-bit color image into 8 bits is to assign 3 bits to red and green and 2 bits to blue (blue has only 2 bits, because of the eye's lower sensitivity to this color). This enables us to display 8 different shades of red and green and 4 of blue. However, this method can yield only poor results. For example, an image might contain different shades of blue which are all clustered around a certain value such that only one shade of blue is used in the 8-bit image and the remaining three blues are not used.

Alternatively, since 8-bit color images are displayed using a colormap (p.235), we can assign any arbitrary color to each of the 256 8-bit values and we can define a separate colormap for each image. This enables us perform a color quantization adjusted to the data contained in the image. One common approach is the *popularity algorithm*, which creates a histogram (p.105) of all colors and retains the 256 most frequent ones. Another approach, known as the *median-cut algorithm*, yields even better results but also needs more computation time. This technique recursively fits a box around all colors used in the RGB colorspace (p.240) which it splits at the median value of its longest side. The algorithm stops after 255 recursions. All colors in one box are mapped to the centroid of this box.

All above techniques restrict the number of displayed colors to 256. A technique of achieving additional colors is to apply a variation of *half-toning* used for gray scale (p.232) images, thus increasing the color resolution at the cost of spatial resolution. The 256 values of the colormap are divided into four sections containing 64 different values of red, green, blue and white. As can be seen in Figure A.1, a 2×2 pixel area is grouped together to represent one composite color, each of the four pixels displays either one of the primary colors (p.240) or white. In this way, the number of possible colors is increased from 256 to 64^4 .

A.6 Convolution

Convolution is a simple mathematical operation which is fundamental to many common image processing operators. Convolution provides a way of 'multiplying together' two arrays of numbers, generally of different sizes, but of the same dimensionality, to produce a third array of numbers

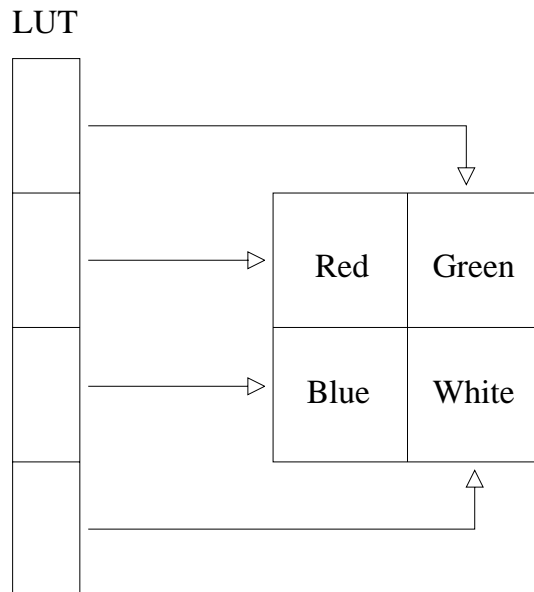


Figure A.1: A 2×2 pixel area displaying one composite color.

of the same dimensionality. This can be used in image processing to implement operators whose output pixel values are simple linear combinations of certain input pixel values.

In an image processing context, one of the input arrays is normally just a graylevel image. The second array is usually much smaller, and is also two-dimensional (although it may be just a single pixel thick), and is known as the kernel (p.233). Figure A.2 shows an example image and kernel that we will use to illustrate convolution.

I₁₁	I₁₂	I₁₃	I₁₄	I₁₅	I₁₆	I₁₇	I₁₈	I₁₉
I₂₁	I₂₂	I₂₃	I₂₄	I₂₅	I₂₆	I₂₇	I₂₈	I₂₉
I₃₁	I₃₂	I₃₃	I₃₄	I₃₅	I₃₆	I₃₇	I₃₈	I₃₉
I₄₁	I₄₂	I₄₃	I₄₄	I₄₅	I₄₆	I₄₇	I₄₈	I₄₉
I₅₁	I₅₂	I₅₃	I₅₄	I₅₅	I₅₆	I₅₇	I₅₈	I₅₉
I₆₁	I₆₂	I₆₃	I₆₄	I₆₅	I₆₆	I₆₇	I₆₈	I₆₉

K₁₁	K₁₂	K₁₃
K₂₁	K₂₂	K₂₃

Figure A.2: An example small image (left) and kernel (right) to illustrate convolution. The labels within each grid square are used to identify each square.

The convolution is performed by sliding the kernel over the image, generally starting at the top left corner, so as to move the kernel through all the positions where the kernel fits entirely within the boundaries of the image. (Note that implementations differ in what they do at the edges of images, as explained below.) Each kernel position corresponds to a single output pixel, the value of which is calculated by multiplying together the kernel value and the underlying image pixel value for each of the cells in the kernel, and then adding all these numbers together.

So, in our example, the value of the bottom right pixel in the output image will be given by:

$$O_{57} = I_{57}K_{11} + I_{58}K_{12} + I_{59}K_{13} + I_{67}K_{21} + I_{68}K_{22} + I_{69}K_{23}$$

If the image has M rows and N columns, and the kernel has m rows and n columns, then the size of the output image will have $M - m + 1$ rows, and $N - n + 1$ columns.

Mathematically we can write the convolution as:

$$O(i, j) = \sum_{k=1}^m \sum_{l=1}^n I(i+k-1, j+l-1)K(k, l)$$

where i runs from 1 to $M - m + 1$ and j runs from 1 to $N - n + 1$.

Note that many implementations of convolution produce a larger output image than this because they relax the constraint that the kernel can only be moved to positions where it fits entirely within the image. Instead, these implementations typically slide the kernel to all positions where just the top left corner of the kernel is within the image. Therefore the kernel ‘overlaps’ the image on the bottom and right edges. One advantage of this approach is that the output image is the same size as the input image. Unfortunately, in order to calculate the output pixel values for the bottom and right edges of the image, it is necessary to *invent* input pixel values for places where the kernel extends off the end of the image. Typically pixel values of zero are chosen for regions outside the true image, but this can often distort the output image at these places. Therefore in general if you are using a convolution implementation that does this, it is better to clip the image to remove these spurious regions. Removing $n - 1$ pixels from the right hand side and $m - 1$ pixels from the bottom will fix things.

Convolution can be used to implement many different operators, particularly spatial filters and feature detectors. Examples include Gaussian smoothing (p.156) and the Sobel edge detector (p.188).

A.7 Distance Metrics

It is often useful in image processing to be able to calculate the distance between two pixels in an image, but this is not as straightforward as it seems. The presence of the pixel grid makes several so-called *distance metrics* possible which often give different answers to each other for the distance between the same pair of points. We consider the three most important ones.

Euclidean Distance

This is the familiar straight line distance that most people are familiar with. If the two pixels that we are considering have coordinates (x_1, y_1) and (x_2, y_2) , then the Euclidean distance is given by:

$$D_{Euclid} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

City Block Distance

Also known as the Manhattan distance. This metric assumes that in going from one pixel to the other it is only possible to travel directly along pixel grid lines. Diagonal moves are not allowed. Therefore the ‘city block’ distance is given by:

$$D_{City} = |x_2 - x_1| + |y_2 - y_1|$$

Chessboard Distance

This metric assumes that you can make moves on the pixel grid as if you were a King making moves in chess, *i.e.* a diagonal move counts the same as a horizontal move. This means that the metric is given by:

$$D_{Chess} = \max(|x_2 - x_1|, |y_2 - y_1|)$$

Note that the last two metrics are usually much faster to compute than the Euclidean metric and so are sometimes used where speed is critical but accuracy is not too important.

A.8 Dithering

Dithering is an image display technique that is useful for overcoming limited display resources. The word *dither* refers to a random or semi-random perturbation of the pixel values.

Two applications of this techniques are particularly useful:

Low quantization display: When images are quantized to a few bits (*e.g.* 3) then only a limited number of graylevels are used in the display of the image. If the scene is smoothly shaded, then the image display will generate rather distinct boundaries around the edges of image regions when the original scene intensity moves from one quantization level to the next. To eliminate this effect, one dithering technique adds random noise (with a small range of values) to the input signal before quantization into the output range. This randomizes the quantization of the pixels at the original quantization boundary, and thus pixels make a more gradual transition from neighborhoods containing 100% of the first quantization level to neighborhoods containing 100% of the second quantization level.

Limited color display: When fewer colors are able to be displayed (*e.g.* 256) than are present in the input image (*e.g.* 24 bit color), then patterns of adjacent pixels are used to simulate the appearance of the unrepresented colors.

A.9 Edge Detectors

Edges are places in the image with strong intensity contrast. Since edges often occur at image locations representing object boundaries, edge detection is extensively used in image segmentation when we want to divide the image into areas corresponding to different objects. Representing an image by its edges has the further advantage that the amount of data is reduced significantly while retaining most of the image information.

Since edges consist of mainly high frequencies, we can, in theory, detect edges by applying a highpass frequency filter (p.167) in the Fourier domain or by convolving (p.227) the image with an appropriate kernel (p.233) in the spatial domain. In practice, edge detection is performed in the spatial domain, because it is computationally less expensive and often yields better results.

Since edges correspond to strong illumination gradients, we can highlight them by calculating the derivatives of the image. This is illustrated for the one-dimensional case in Figure A.3.

We can see that the position of the edge can be estimated with the maximum of the 1st derivative or with the zero-crossing of the 2nd derivative. Therefore we want to find a technique to calculate the derivative of a two-dimensional image. For a discrete one-dimensional function $f(i)$, the first derivative can be approximated by

$$\frac{df(i)}{d(i)} = f(i+1) - f(i)$$

Calculating this formula is equivalent to convolving the function with $[-1 \ 1]$. Similarly the 2nd derivative can be estimated by convolving $f(i)$ with $[1 \ -2 \ 1]$.

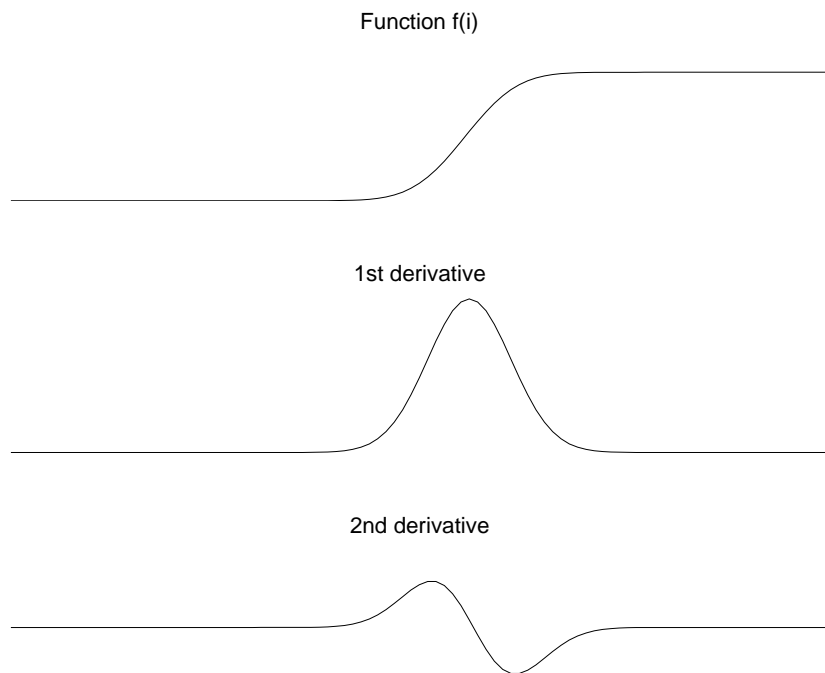


Figure A.3: 1st and 2nd derivative of an edge illustrated in one dimension.

Different edge detection kernels which are based on the above formula enable us to calculate either the 1st or the 2nd derivative of a two-dimensional image. There are two common approaches to estimate the 1st derivative in a two-dimensional image, Prewitt compass edge detection (p.195) and *gradient edge detection*.

Prewitt compass edge detection involves convolving the image with a set of (usually 8) kernels, each of which is sensitive to a different edge orientation. The kernel producing the maximum response at a pixel location determines the edge magnitude and orientation. Different sets of kernels might be used: examples include Prewitt, Sobel, Kirsch and Robinson kernels.

Gradient edge detection is the second and more widely used technique. Here, the image is convolved with only two kernels, one estimating the gradient in the x -direction, G_x , the other the gradient in the y -direction, G_y . The absolute gradient magnitude is then given by

$$|G| = \sqrt{G_x^2 + G_y^2}$$

and is often approximated with

$$|G| = |G_x| + |G_y|$$

In many implementations, the gradient magnitude is the only output of a gradient edge detector, however the edge orientation might be calculated with

$$\theta = \arctan(G_y/G_x)$$

The most common kernels used for the gradient edge detector are the Sobel (p.188), Roberts Cross (p.184) and Prewitt (p.188) operators.

After having calculated the magnitude of the 1st derivative, we now have to identify those pixels corresponding to an edge. The easiest way is to threshold (p.69) the gradient image, assuming that all pixels having a local gradient above the threshold must represent an edge. An alternative technique is to look for local maxima in the gradient image, thus producing one pixel wide edges. A more sophisticated technique is used by the Canny edge detector (p.192). It first applies a gradient edge detector to the image and then finds the edge pixels using *non-maximal suppression* and *hysteresis tracking*.

An operator based on the 2nd derivative of an image is the Marr edge detector (p.199), also known as *zero crossing detector*. Here, the 2nd derivative is calculated using a Laplacian of Gaussian (LoG) (p.173) filter. The Laplacian has the advantage that it is an isotropic (p.233) measure of the 2nd derivative of an image, *i.e.* the edge magnitude is obtained independently from the edge orientation by convolving the image with only one kernel. The edge positions are then given by the zero-crossings in the LoG image. The scale of the edges which are to be detected can be controlled by changing the variance of the Gaussian.

A general problem for edge detection is its sensitivity to noise (p.221), the reason being that calculating the derivative in the spatial domain (p.240) corresponds to accentuating high frequencies and hence magnifying noise. This problem is addressed in the Canny and Marr operators by convolving the image with a smoothing operator (Gaussian) before calculating the derivative.

A.10 Frequency Domain

For simplicity, assume that the image I being considered is formed by projection from scene S (which might be a two- or three-dimensional scene, *etc.*).

The *frequency domain* is a space in which each image value at image position F represents the amount that the intensity values in image I vary over a specific distance related to F . In the frequency domain, changes in image position correspond to changes in the spatial frequency (p.240), (or the rate at which image intensity values) are changing in the spatial domain image I .

For example, suppose that there is the value 20 at the point that represents the frequency 0.1 (or 1 period every 10 pixels). This means that in the corresponding spatial domain image I the intensity values vary from dark to light and back to dark over a distance of 10 pixels, and that the contrast between the lightest and darkest is 40 gray levels (2 times 20).

The spatial frequency domain is interesting because: 1) it may make explicit periodic relationships in the spatial domain (p.240), and 2) some image processing operators are more efficient or indeed only practical when applied in the frequency domain.

In most cases, the Fourier Transform (p.209) is used to convert images from the spatial domain into the frequency domain and vice-versa.

A related term used in this context is *spatial frequency*, which refers to the (inverse of the) periodicity with which the image intensity values change. Image features with high spatial frequency (such as edges) are those that change greatly in intensity over short image distances.

A.11 Grayscale Images

A grayscale (or graylevel) image is simply one in which the only colors are shades of gray. The reason for differentiating such images from any other sort of color image is that less information needs to be provided for each pixel. In fact a 'gray' color is one in which the red, green and blue components all have equal intensity in RGB space (p.240), and so it is only necessary to specify a single intensity value for each pixel, as opposed to the three intensities needed to specify each pixel in a full color image (p.225).

Often, the grayscale intensity is stored as an 8-bit integer giving 256 possible different shades of gray from black to white. If the levels are evenly spaced then the difference between successive graylevels is significantly better than the graylevel resolving power of the human eye.

Grayscale images are very common, in part because much of today's display and image capture hardware can only support 8-bit images. In addition, grayscale images are entirely sufficient for many tasks and so there is no need to use more complicated and harder-to-process color images.

A.12 Image Editing Software

There is a huge variety of software for manipulating images in various ways. Much of this software can be grouped under the heading *image processing software*, and the bulk of this reference is concerned with that group.

Another very important category is what we call *image editing software*. This group includes painting programs, graphic art packages and so on. They are often useful in conjunction with image processing software packages, in situations where direct immediate interaction with an image is the easiest way of achieving something. For instance, if a region of an image is to be masked out (p.235) for subsequent image processing, it may be easiest to create the mask using an art package by directly drawing on top of the original image. The mask used in the description of the AND operator (p.55) was created this way for instance. Art packages also often allow the user to move sections of the images around and brighten or darken selected regions interactively. Few dedicated image processing packages offer the same flexibility and ease of use in this respect.

A.13 Idempotence

Some operators have the special property that applying them more than once to the same image produces no further change after the first application. Such operators are said to be *idempotent*. Examples include the morphological operators opening (p.127) and closing (p.130).

A.14 Isotropic Operators

An isotropic operator in an image processing context is one which applies equally well in all directions in an image, with no particular sensitivity or bias towards one particular set of directions (*e.g.* compass directions). A typical example is the zero crossing edge detector (p.199) which responds equally well to edges in any orientation. Another example is Gaussian smoothing (p.156). It should be borne in mind that although an operator might be isotropic in theory, the actual implementation of it for use on a discrete pixel grid may not be perfectly isotropic. An example of this is a Gaussian smoothing filter with very small standard deviation on a square grid.

A.15 Kernel

A kernel is a (usually) smallish matrix of numbers that is used in image convolutions (p.227). Differently sized kernels containing different patterns of numbers give rise to different results under convolution. For instance, Figure A.4 shows a 3×3 kernel that implements a mean filter.

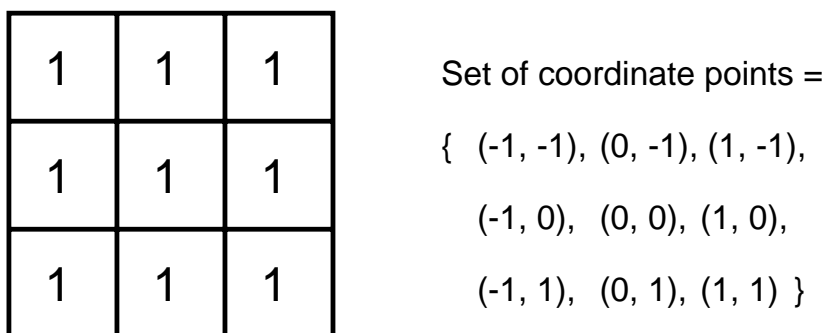


Figure A.4: Convolution kernel for a mean filter with 3×3 neighborhood.

The word ‘kernel’ is also commonly used as a synonym for ‘structuring element’ (p.241), which is a similar object used in mathematical morphology (p.236). A structuring element differs from a kernel in that it also has a specified *origin*. This sense of the word ‘kernel’ is not used in HIPR.

A.16 Logical Operators

Logical operators are generally derived from *Boolean algebra*, which is a mathematical way of manipulating the *truth values* of concepts in an abstract way without bothering about what the concepts actually *mean*. The truth value of a concept in Boolean value can have just one of two possible values: true or false. Boolean algebra allows you to represent things like:

The block is both red and large

by something like:

A AND B

where A represents ‘The block is red’, and B represents ‘The block is large’. Now each of these sub-phrases has its own truth value in any given situation: each sub-phrase is either true or false. Moreover, the entire composite phrase also has a truth value: it is true if both of the sub-phrases are true, and false in any other case. We can write this AND (p.55) combination rule (and its dual operation NAND (p.55)) using a *truth-table* as shown in Figure A.5, in which we conventionally represent true by 1, and false by zero.

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

NAND

Figure A.5: Truth-tables for AND and NAND

The left hand table shows each of the possible combinations of truth values of A and B , and the resulting truth value of A AND B . Similar truth-tables can be set up for the other logical operators: NAND (p.55), OR (p.58), NOR (p.58), XOR (p.60), XNOR (p.60) and NOT (p.63).

Turning now to an image processing context, the pixel values in a binary image (p.225), which are either 0 or 1, can be interpreted as truth values as above. Using this convention we can carry out logical operations on images simply by applying the truth-table combination rules to the pixel values from a pair of input images (or a single input image in the case of NOT). Normally, corresponding pixels from each of two identically sized binary input images are compared to produce the output image, which is another binary image of the same size. As with other image arithmetic operations, it is also possible to logically combine a single input image with a constant logical value, in which case each pixel in the input image is compared to the same constant in order to produce the corresponding output pixel. See the individual logical operator descriptions for examples of these operations.

Logical operations can also be carried out on images with integer pixel values. In this extension the logical operations are normally carried out in *bitwise* fashion on binary representations of those integers, comparing corresponding bits with corresponding bits to produce the output pixel value. For instance, suppose that we wish to XOR (p.60) the integers 47 and 255 together using 8-bit integers. 47 is 00101111 in binary and 255 is 11111111. XORing these together in bitwise fashion, we have 11010000 in binary or 208 in decimal.

Note that not all implementations of logical operators work in such bitwise fashion. For instance some will treat zero as false and any non-zero value as true and will then apply the conventional 1-bit logical functions to derive the output image. The output may be a simple binary image itself, or it may be a graylevel image formed perhaps by multiplying what would be the binary output image (containing 0's and 1's) with one of the input images.

A.17 Look-up Tables and Colormaps

Look-Up Tables or *LUTs* are fundamental to many aspects of image processing. An LUT is simply a table of cross-references linking *index numbers* to *output values*. The most common use is to determine the colors and intensity values with which a particular image will be displayed, and in this context the LUT is often called simply a *colormap*.

The idea behind the colormap is that instead of storing a definite color for each pixel in an image, for instance in 24-bit RGB format (p.226), each pixel's value is instead treated as an index number into the colormap. When the image is to be displayed or otherwise processed, the colormap is used to look up the actual colors corresponding to each index number. Typically, the output values stored in the LUT would be RGB color values (p.240).

There are two main advantages to doing things this way. Firstly, the index number can be made to use fewer bits than the output value in order to save storage space. For instance an 8-bit index number can be used to look up a 24-bit RGB color value in the LUT. Since only the 8-bit index number needs to be stored for each pixel, such 8-bit color images (p.226) take up less space than a full 24-bit image of the same size. Of course the image can only contain 256 different colors (the number of entries in an 8-bit LUT), but this is sufficient for many applications and usually the observable image degradation is small.

Secondly the use of a color table allows the user to experiment easily with different color labeling schemes for an image.

One disadvantage of using a colormap is that it introduces additional complexity into an image format. It is usually necessary for each image to carry around its own colormap, and this LUT must be continually consulted whenever the image is displayed or processed.

Another problem is that in order to convert from a full color image to (say) an 8-bit color image using a color image, it is usually necessary to throw away many of the original colors, a process known as color quantization (p.227). This process is lossy, and hence the image quality is degraded during the quantization process. Additionally, when performing further image processing on such images, it is frequently necessary to generate a new colormap for the new images, which involves further color quantization, and hence further image degradation.

As well as their use in colormaps, LUTs are often used to remap the pixel values within an image. This is the basis of many common image processing point operations (p.68) such as thresholding, gamma correction and contrast stretching. The process is often referred to as *anamorphosis*.

A.18 Masking

A mask is a binary image (p.225) consisting of zero- and non-zero values. If a mask is applied to another binary or to a grayscale (p.232) image of the same size, all pixels which are zero in the mask are set to zero in the output image. All others remain unchanged.

Masking can be implemented either using pixel multiplication (p.48) or logical AND (p.55), the latter in general being faster.

Masking is often used to restrict a point (p.68) or arithmetic operator (p.42) to an area defined by the mask. We can, for example, accomplish this by first masking the desired area in the input image and processing it with the operator, then masking the original input image with the inverted (p.63) mask to obtain the unprocessed area of the image and finally recombining the two partial images using image addition (p.43). An example can be seen in the worksheet on the logical AND (p.55) operator. In some image processing packages, a mask can directly be defined as an optional input to a point operator, so that automatically the operator is only applied to the pixels defined by the mask .

A.19 Mathematical Morphology

The field of mathematical morphology contributes a wide range of operators to image processing, all based around a few simple mathematical concepts from set theory. The operators are particularly useful for the analysis of binary images (p.225) and common usages include edge detection, noise removal, image enhancement and image segmentation.

The two most basic operations in mathematical morphology are erosion (p.123) and dilation (p.118). Both of these operators take two pieces of data as input: an image to be eroded or dilated, and a structuring element (p.241) (also known as a *kernel*). The two pieces of input data are each treated as representing sets of coordinates in a way that is slightly different for binary and grayscale images.

For a binary image, white pixels are normally taken to represent foreground regions, while black pixels denote background. (Note that in some implementations this convention is reversed, and so it is very important to set up input images with the correct polarity (p.225) for the implementation being used). Then the set of coordinates corresponding to that image is simply the set of two-dimensional Euclidean coordinates of all the foreground pixels in the image, with an origin normally taken in one of the corners so that all coordinates have positive elements.

For a grayscale image, the intensity value (p.239) is taken to represent height above a base plane, so that the grayscale image represents a surface in three-dimensional Euclidean space. Figure A.6 shows such a surface. Then the set of coordinates associated with this image surface is simply the set of three-dimensional Euclidean coordinates of all the points within this surface *and also all points below the surface, down to the base plane*. Note that even when we are only considering points with integer coordinates, this is a lot of points, so usually algorithms are employed that do not need to consider all the points.

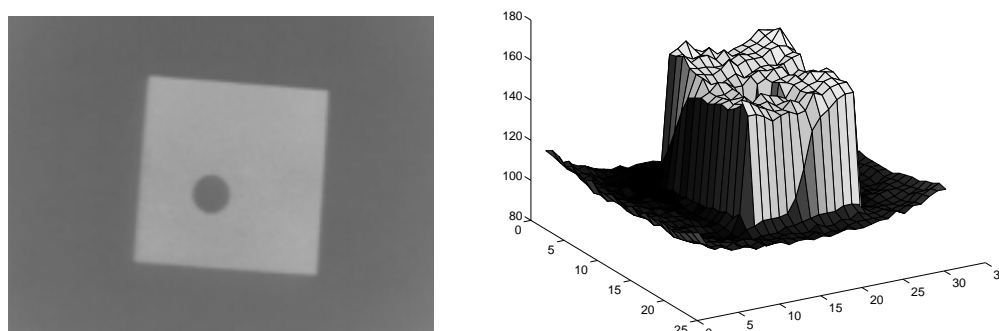


Figure A.6: Simple graylevel image and the corresponding surface in image space

The structuring element (p.241) is already just a set of point coordinates (although it is often represented as a binary image). It differs from the input image coordinate set in that it is normally

much smaller, and its coordinate origin is often not in a corner, so that some coordinate elements will have negative values. Note that in many implementations of morphological operators, the structuring element is assumed to be a particular shape (*e.g.* a 3×3 square) and so is hardwired into the algorithm.

Binary morphology can be seen as a special case of graylevel morphology in which the input image has only two graylevels at values 0 and 1.

Erosion and dilation work (at least conceptually) by translating the structuring element to various points in the input image, and examining the intersection between the translated kernel coordinates and the input image coordinates. For instance, in the case of erosion, the output coordinate set consists of just those points to which the origin of the structuring element can be translated, while the element still remains entirely 'within' the input image.

Virtually all other mathematical morphology operators can be defined in terms of combinations of erosion and dilation along with set operators such as intersection and union. Some of the more important are opening (p.127), closing (p.130) and skeletonization (p.145).

A.20 Multi-spectral Images

A multi-spectral image is a collection of several monochrome images of the same scene, each of them taken with a different sensor. Each image is referred to as a *band*. A well known multi-spectral (or multi-band image) is a RGB color (p.225) image, consisting of a red, a green and a blue image, each of them taken with a sensor sensitive to a different wavelength. In image processing, multi-spectral images are most commonly used for Remote Sensing applications. Satellites usually take several images from frequency bands in the visual and non-visual range. *Landsat 5*, for example, produces 7 band images with the wavelength of the bands being between *450* and *1250 nm*.

All the standard single-band image processing operators can also be applied to multi-spectral images by processing each band separately. For example, a multi-spectral image can be edge detected (p.230) by finding the edges in each band and then ORing (p.58) the three edge images together. However, we would obtain more reliable edges, if we associate a pixel with an edge based on its properties in all three bands and not only in one.

To fully exploit the additional information which is contained in the multiple bands, we should consider the images as one multi-spectral image rather than as a set of monochrome graylevel images. For an image with k bands, we can then describe the brightness of each pixel as a point in a k -dimensional space represented by a vector of length k .

Special techniques exist to process multi-spectral images. For example, to classify (p.107) a pixel as belonging to one particular region, its intensities in the different bands are said to form a *feature vector* describing its location in the k -dimensional feature space. The simplest way to define a class is to choose a *upper* and *lower* threshold (p.69) for each band, thus producing a k -dimensional 'hyper-cube' in the feature space. Only if the feature vector of a pixel points to a location within this cube, is the pixel classified as belonging to this class. A more sophisticated classification (p.107) method is described in the corresponding worksheet.

The disadvantage of multi-spectral images is that, since we have to process additional data, the required computation time and memory increase significantly. However, since the speed of the hardware will increase and the costs for memory will decrease in the future, it can be expected that multi-spectral images will become more important in many fields of computer vision.

A.21 Non-linear Filtering

Suppose that an image processing operator F acting on two input images A and B produces output images C and D respectively. If the operator F is *linear*, then

$$F(a \times A + b \times B) = a \times C + b \times D$$

where a and b are constants. In practice, this means that each pixel in the output of a *linear* operator is the weighted sum of a set of pixels in the input image.

By contrast, *non-linear* operators are all the other operators. For example, the threshold (p.69) operator is non-linear, because individually, corresponding pixels in the two images A and B may be below the threshold, whereas the pixel obtained by adding A and B may be above threshold. Similarly, an absolute value operation is non-linear:

$$|-1 + 1| \neq |-1| + |1|$$

as is the exponential operator:

$$\exp(1 + 1) \neq \exp(1) + \exp(1)$$

A.22 Pixels

In order for any digital computer processing to be carried out on an image, it must first be stored within the computer in a suitable form that can be manipulated by a computer program. The most practical way of doing this is to divide the image up into a collection of discrete (and usually small) cells, which are known as *pixels*. Most commonly, the image is divided up into a rectangular grid of pixels, so that each pixel is itself a small rectangle. Once this has been done, each pixel is given a pixel value (p.239) that represents the color of that pixel. It is assumed that the whole pixel is the same color, and so any color variation that did exist within the area of the pixel before the image was discretized is lost. However, if the area of each pixel is very small, then the discrete nature of the image is often not visible to the human eye.

Other pixel shapes and formations can be used, most notably the hexagonal grid, in which each pixel is a small hexagon. This has some advantages in image processing, including the fact that pixel connectivity (p.238) is less ambiguously defined than with a square grid, but hexagonal grids are not widely used. Part of the reason is that many image capture systems (*e.g.* most CCD cameras and scanners) intrinsically discretize the captured image into a rectangular grid in the first instance.

A.23 Pixel Connectivity

The notation of pixel connectivity describes a relation between two or more pixels. For two pixels to be connected they have to fulfill certain conditions on the pixel brightness and spatial adjacency.

First, in order for two pixels to be considered connected, their pixel values must both be from the same set of values V . For a grayscale image, V might be any range of graylevels, *e.g.* $V = \{22, 23, \dots, 40\}$, for a binary image we simply have $V = \{1\}$.

To formulate the adjacency criterion for connectivity, we first introduce the notation of *neighborhood*. For a pixel p with the coordinates (x, y) the set of pixels given by:

$$N_4(p) = \{(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)\}$$

is called its *4-neighbors*. Its *8-neighbors* are defined as

$$N_8(p) = N_4 \cup \{(x + 1, y + 1), (x + 1, y - 1), (x - 1, y + 1), (x - 1, y - 1)\}$$

From this we can infer the definition for *4-* and *8-connectivity*:

Two pixels p and q , both having values from a set V are 4 -connected if q is from the set $N_4(p)$ and 8 -connected if q is from $N_8(p)$.

General connectivity can either be based on 4 - or 8 -connectivity; for the following discussion we use 4 -connectivity.

A pixel p is connected to a pixel q if p is 4 -connected to q or if p is 4 -connected to a third pixel which itself is connected to q . Or, in other words, two pixels q and p are connected if there is a path from p and q on which each pixel is 4 -connected to the next one.

A set of pixels in an image which are all *connected* to each other is called a *connected component*. Finding all connected components in an image and marking each of them with a distinctive label is called connected component labeling (p.114).

An example of a binary image with two connected components which are based on 4 -connectivity can be seen in Figure A.7. If the connectivity were based on 8 -neighbors, the two connected components would merge into one.

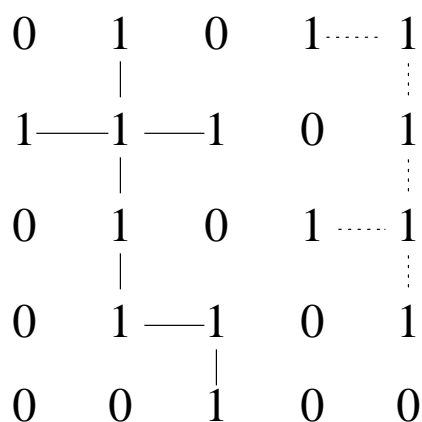


Figure A.7: Two connected components based on 4 -connectivity.

A.24 Pixel Values

Each of the pixels (p.238) that represents an image stored inside a computer has a *pixel value* which describes how bright that pixel is, and/or what color it should be. In the simplest case of binary images (p.225), the pixel value is a 1-bit number indicating either foreground or background. For a grayscale images (p.232), the pixel value is a single number that represents the brightness of the pixel. The most common *pixel format* is the *byte image*, where this number is stored as an 8-bit integer giving a range of possible values from 0 to 255. Typically zero is taken to be black, and 255 is taken to be white. Values in between make up the different shades of gray.

To represent color images (p.225), separate red, green and blue components must be specified for each pixel (assuming an RGB colorspace (p.240)), and so the pixel ‘value’ is actually a vector of three numbers. Often the three different components are stored as three separate ‘grayscale’ images known as *color planes* (one for each of red, green and blue), which have to be recombined when displaying or processing.

Multi-spectral images (p.237) can contain even more than three components for each pixel, and by extension these are stored in the same kind of way, as a vector pixel value, or as separate color planes.

The actual grayscale or color component intensities for each pixel may not actually be stored explicitly. Often, all that is stored for each pixel is an index into a colormap (p.235) in which the actual intensity or colors can be looked up.

Although simple 8-bit integers or vectors of 8-bit integers are the most common sorts of pixel values used, some image formats support different types of value, for instance 32-bit signed integers or floating point values. Such values are extremely useful in image processing as they allow processing to be carried out on the image where the resulting pixel values are not necessarily 8-bit integers. If this approach is used then it is usually necessary to set up a colormap which relates particular ranges of pixel values to particular displayed colors.

A.25 Primary Colors

It is a useful fact that the huge variety of colors that can be perceived by humans can all be produced simply by adding together appropriate amounts of red, blue and green colors. These colors are known as the primary colors. Thus in most image processing applications, colors are represented by specifying separate intensity values for red, green and blue components. This representation is commonly referred to as RGB (p.240).

The primary color phenomenon results from the fact that humans have three different sorts of color receptors in their retinas which are each most sensitive to different visible light wavelengths.

The primary colors used in painting (red, yellow and blue) are different. When paints are mixed, the 'addition' of a new color paint actually *subtracts* wavelengths from the reflected visible light.

A.26 RGB and Colorspaces

A color perceived by the human eye can be defined by a linear combination of the three primary colors (p.240) red, green and blue. These three colors form the basis for the RGB-colorspace (p.240). Hence, each perceivable color can be defined by a vector in the three-dimensional colorspace. The intensity is given by the length of the vector, and the actual color by the two angles describing the orientation of the vector in the colorspace.

The RGB-space can also be transformed into other coordinate systems, which might be more useful for some applications. One common basis for the color space is *IHS*. In this coordinate system, a color is described by its intensity, hue (average wavelength) and saturation (the amount of white in the color). This color space makes it easier to directly derive the intensity and color of perceived light and is therefore more likely to be used by human beings.

A.27 Spatial Domain

For simplicity, assume that the image I being considered is formed by projection from scene S (which might be a two- or three-dimensional scene, *etc.*).

The *spatial domain* is the normal image space, in which a change in position in I directly projects to a change in position in S . Distances in I (in pixels) correspond to real distances (*e.g.* in meters) in S .

This concept is used most often when discussing the frequency with which image values change, that is, over how many pixels does a cycle of periodically repeating intensity variations occur. One would refer to the number of pixels over which a pattern repeats (its periodicity) in the spatial domain.

In most cases, the Fourier Transform (p.209) will be used to convert images from the spatial domain into the frequency domain (p.232).

A related term used in this context is *spatial frequency*, which refers to the (inverse of the) periodicity with which the image intensity values change. Image features with high spatial frequency (such as edges) are those that change greatly in intensity over short image distances.

Another term used in this context is *spatial derivative*, which refers to how much the image intensity values change per change in image position.

A.28 Structuring Elements

The field of mathematical morphology (p.236) provides a number of important image processing operations, including erosion (p.123), dilation (p.118), opening (p.127) and closing (p.130). All these morphological operators take two pieces of data as input. One is the input image, which may be either binary or grayscale for most of the operators. The other is the *structuring element*. It is this that determines the precise details of the effect of the operator on the image.

The structuring element is sometimes called the *kernel*, but we reserve that term for the similar objects used in convolutions (p.227).

The structuring element consists of a pattern specified as the coordinates of a number of discrete points relative to some origin. Normally cartesian coordinates are used and so a convenient way of representing the element is as a small image on a rectangular grid. Figure A.8 shows a number of different structuring elements of various sizes. In each case the origin is marked by a ring around that point. The origin does not have to be in the center of the structuring element, but often it is. As suggested by the figure, structuring elements that fit into a 3×3 grid with its origin at the center are the most commonly seen type.

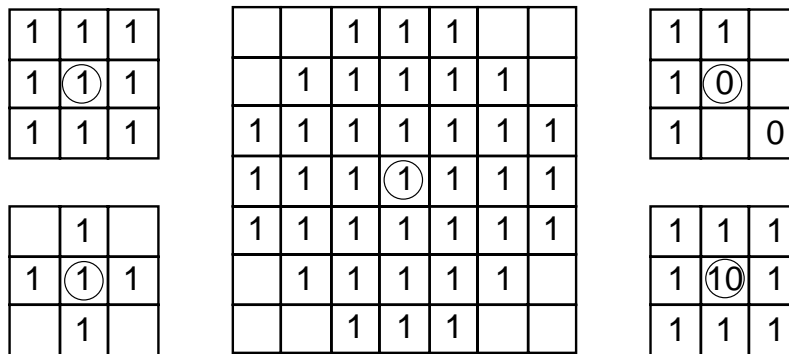


Figure A.8: Some example structuring elements.

Note that each point in the structuring element may have a value. In the simplest structuring elements used with binary images for operations such as erosion, the elements only have one value, conveniently represented as a one. More complicated elements, such as those used with thinning (p.137) or grayscale morphological operations, may have other pixel values.

An important point to note is that although a rectangular grid is used to represent the structuring element, not every point in that grid is part of the structuring element in general. Hence the elements shown in Figure A.8 contain some blanks. In many texts, these blanks are represented as zeros, but this can be confusing and so we avoid it here.

When a morphological operation is carried out, the origin of the structuring element is typically translated to each pixel position in the image in turn, and then the points within the translated structuring element are compared with the underlying image pixel values. The details of this comparison, and the effect of the outcome depend on which morphological operator is being used.

A.29 Wrapping and Saturation

If an image is represented in a *byte* or *integer* pixel format (p.239), the maximum pixel value is limited by the number of bits used for the representation, *e.g.* the pixel values of a 8-bit image are

limited to 255.

However, many image processing operations produce output values which are likely to exceed the given maximum value. In such cases, we have to decide how to handle this *pixel overflow*.

One possibility is to wrap around the overflowing pixel values. This means that if a value is greater than the possible maximum, we subtract the *pixel value range* so that the value starts again from the possible minimum value. Figure A.9 shows the mapping function for wrapping the output values of some operation into an 8-bit format.

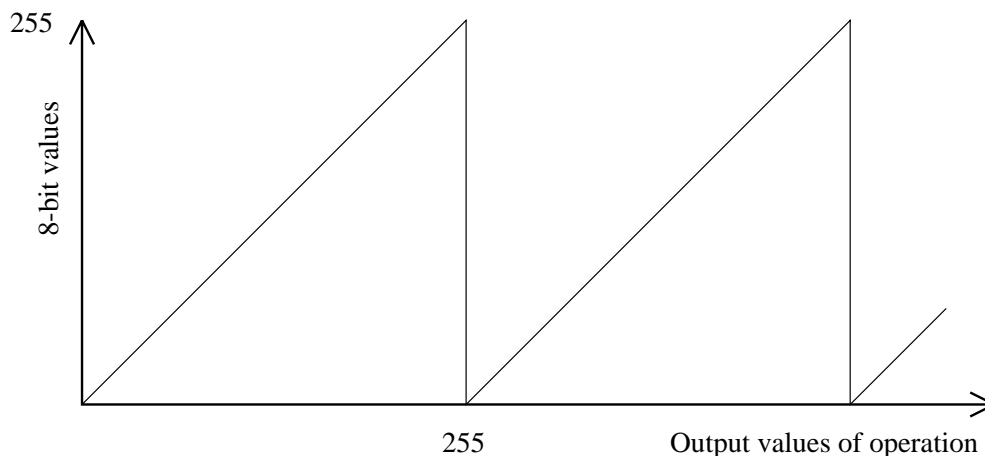


Figure A.9: Mapping function for wrapping the pixel values of an 8-bit image.

Another possibility is to set all overflowing pixels to the maximum possible values — an effect known as saturation. The corresponding mapping function for an 8-bit image can be seen in Figure A.10.

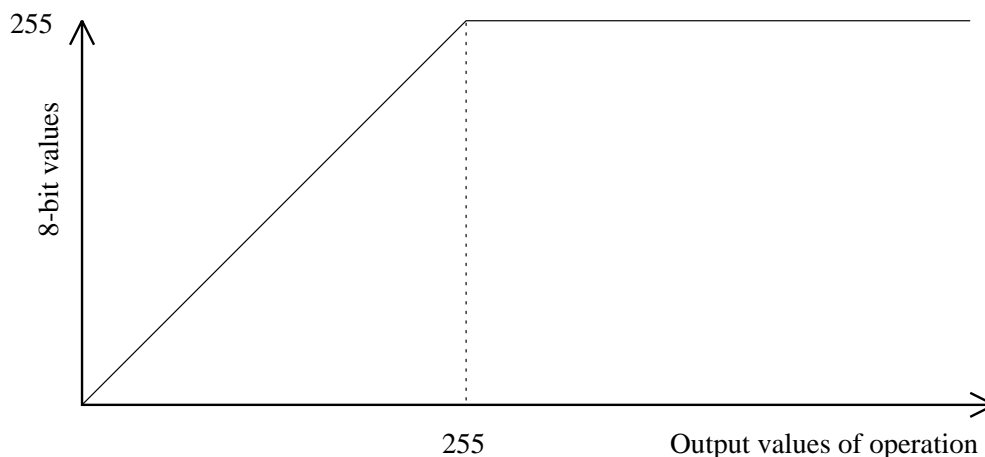


Figure A.10: Mapping function for saturating an 8-bit image.

If only a few pixels in the image exceed the maximum value it is often better to apply the latter technique, especially if we use the image for display purposes. However, by setting all overflowing pixels to the same value we lose an essential amount of information. In the worst case, when all pixels exceed the maximum value, this would lead to an image of constant pixel values. Wrapping around overflowing pixel retains the differences between values. On the other hand, it might cause the problem that pixel values passing the maximum ‘jump’ from the maximum to the minimum

value. Examples for both techniques can be seen in the worksheets of various point operators (p.68). If possible, it is easiest to change the image format, for example to *float* format, so that all pixel values can be represented. However, we should keep in mind that this implies an increase in processing time and memory.

Appendix B

Common Software Implementations

It is often useful to know whether a particular operator is implemented within a given package, and what it is called. The tables below aim to help in this task a little by listing some common software implementations of standard operators. Due to space and time constraints we cannot present anything like a full survey of all of the operators in all the many image processing packages in existence, so we have instead concentrated instead on four of the more common packages that we use here at Edinburgh.

The four packages we have chosen are: Visilog, Khoros, the Matlab Image Processing Toolbox and HIPS. Information about these packages, and advice as to where they can be obtained, is given below.

If your image processing software is not mentioned here then you will have to consult the documentation that came with it for help on operator equivalents.

Note that while we have done our best to describe the contents of these packages accurately, it is possible that we have made some omissions, or that the implementation/version that you are using is different from ours. Where a package has several operators that do similar things to an operator documented in HIPR, we have mentioned only the one we think is closest.

B.1 Visilog

Visilog is a GUI (Graphical User Interface) based image processing package produced by Noesis (Noesis, Immeuble Nungesser, 13 Avenue Morane Saulnier, 78140 Velizy, France; Tel: (33-1)34-65-08-95; Fax: (33-1)34-65-99-14). It is available commercially for MS-Windows and UNIX with X-Windows. The version described here is 4.1.4 for UNIX and X-Windows.

For academic users in the UK, Visilog is available at reduced cost under the CHEST (Combined Higher Education Software Team) agreement.

All image processing functions in Visilog are accessed via pull-down menus. The *NOTES* column in the tables below indicates where the functions are located within those menus, *e.g.* to use the addition operator, first open the 'Process' menu, then open the 'Point Ops' sub-menu, then the 'Arithmetic' sub-sub-menu and from that menu select the 'add' option. Where the specified menu path ends in an ellipsis (...), this indicates that at this stage a dialog box opens which the user can use to select the details of the operation to be performed.

Image Arithmetic

OPERATOR	INC	NOTES
Addition	Yes	Process/Point Ops/Arithmetic/add
Subtraction	Yes	Process/Point Ops/Arithmetic/subtract
Multiplication	Yes	Process/Point Ops/Arithmetic/multiply
Division	Yes	Process/Point Ops/Arithmetic/divide
Blending	Yes	Process/Point Ops/Arithmetic/blend
Logical AND/NAND	Yes	Process/Point Ops/Logical/and/nand
Logical OR/NOR	Yes	Process/Point Ops/Logical/or/nor
Logical XOR/XNOR	Yes	Process/Point Ops/Logical/xor/nxor
Invert/Logical NOT	Yes	Process/Point Ops/Logical/not
Bitshift Operators	Yes	Process/Point Ops/Logical/shift

Point Operations

OPERATOR	INC	NOTES
Thresholding	Yes	Process/Point Ops/Segmentation/threshold
Adaptive Threshold	No	
Contrast Stretching	Yes	Process/Point Ops/Anamorphosis/normalize
Hist. Equalization	Yes	Process/Point Ops/Anamorphosis/hequalize
Logarithm Operator	No	Process/Point Ops/Anamorphosis/anamorphosis
Raise to Power	No	Process/Point Ops/Anamorphosis/anamorphosis

Geometric Operations

OPERATOR	INC	NOTES
Scale	Yes	(Only reduces) Process/Geometry/Basic Op/sample
Rotate	Yes	Process/Geometry/Basic Op/rotation
Reflect	Yes	Process/Geometry/Basic Op/symmetry
Translate	Yes	Process/Geometry/Basic Op/slide
Affine Transform	Yes	(Polynomial warp) Process/Geometry/Warping/applywarp

Image Analysis

OPERATOR	INC	NOTES
Intensity Histogram	Yes	Analyze/Statistics/histogram
Classification	Yes	Analyze/Individual...
Labeling	Yes	Process/Point Ops/Segmentation/label

Morphology

OPERATOR	INC	NOTES
Dilation	Yes	Process/Morphology/Basic Op/dilate
Erosion	Yes	Process/Morphology/Basic Op/erode
Opening	Yes	Process/Morphology/Basic Op/opening
Closing	Yes	Process/Morphology/Basic Op/closing
Hit/Miss Transform	Yes	Process/Morphology/Hit or Miss...
Thinning	Yes	Process/Morphology/Thin/Thick...
Thickening	Yes	Process/Morphology/Thin/Thick...
Skeletonization	Yes	Process/Morphology/Thin/Thick/skeleton

Digital Filters

OPERATOR	INC	NOTES
Mean Filter	Yes	Process/Filters/Smoothing/lowpass...
Median Filter	Yes	Process/Filters/Smoothing/median
Gaussian Smoothing	Yes	Process/Filters/Smoothing/lowpass...
Conservative Smooth	No	
Crimmins	No	
Frequency Filters	Yes	Use Fourier Transform plus multiplication
Laplacian Filter	Yes	Process/Edge Detection/Laplacian/laplacian...
Unsharp Filter	Yes	Process/Filters/Sharpening/highpass...

Feature Detectors

OPERATOR	INC	NOTES
Roberts Cross	Yes	Process/Edge Detection/gradient3x3...
Sobel	Yes	Process/Edge Detection/gradient3x3...
Canny	Yes	Process/Edge Detection/Gradient/rgradient + Process/Edge Detection/Gradient/gradient_mag + Process/Edge Detection/Gradient/lmaxima (No hysteresis tracking)
Compass	Yes	Process/Edge Detection/compass3x3...
Zero Crossing	Yes	Process/Edge Detection/Laplacian/zero_crossings
Line Detector	No	

Image Transforms

OPERATOR	INC	NOTES
Distance Transform	Yes	Extensions/Morpho+/distance
Fourier Transform	Yes	Process/Frequency/2 Dimensions/fft2d
Hough Transform	No	

Image Synthesis

OPERATOR	INC	NOTES
Noise Generation	No	(Image editing is possible however)

B.2 Khoros

Khoros is a sophisticated visual programming and scientific software development environment, that also includes extensive image processing facilities. It is a very large package because it does a lot more than just image processing. However, it is available free via FTP from several sites around the world. Khoros was developed at the University of New Mexico and the developers have recently formed a company to continue the work on Khoros. The company is called Khoral Research (Khoral Research Inc, 6001 Indian School Rd, Ne, Suite 200, Albuquerque, NM 87110, USA; Tel:(505)837-6500; Fax:(505)881-3842). You can find out more about Khoros by pointing a World Wide Web browser such as Netscape to

<http://www.khoros.unm.edu/>

Apart from including a wide range of image processing tools, Khoros also allows you to link operators together in a graphical way to form image processing pipelines.

In addition to the standard operators you see here, there are also user-contributed toolboxes for Khoros available free via FTP which provide virtually any additional operator you can think of. See the Khoros World Wide Web home page for details.

Khoros is only available for UNIX systems. The version described here is version 1.0. The latest version is Khoros 2.0 which is quite different in appearance to Khoros 1.0. Strangely, Khoros 2.0 contains *fewer* image processing functions than Khoros 1.0 — mainly because the developers are trying to emphasize that it is more than just an image processing package. However, the toolboxes mentioned above can be added to Khoros 2.0 to give it virtually any operators you require.

When using *cantata*, the Khoros visual programming environment, image processing operations are accessed using pull-down menus. In the tables below, the locations of each operator within the menu structure is shown in the *NOTES* column, *e.g.* to use the addition operator, first open the 'Arithmetic' menu, then open the 'Binary Arithmetic' sub-menu and from that menu select the 'Add' option. Where the specified menu path ends in an ellipsis (...), this indicates that at this stage a dialog box opens which the user can use to select the details of the operation to be performed.

Image Arithmetic

OPERATOR	INC	NOTES
Addition	Yes	Arithmetic/Binary Arithmetic/Add
Subtraction	Yes	Arithmetic/Binary Arithmetic/Subtract
Multiplication	Yes	Arithmetic/Binary Arithmetic/Multiply
Division	Yes	Arithmetic/Binary Arithmetic/Divide
Blending	Yes	Arithmetic/Binary Arithmetic/Blend
Logical AND/NAND	Yes	Arithmetic/Logical Operations/AND
Logical OR/NOR	Yes	Arithmetic/Logical Operations/OR
Logical XOR/XNOR	Yes	Arithmetic/Logical Operations/XOR
Invert/Logical NOT	Yes	Arithmetic/Unary Arithmetic/NOT/Invert
Bitshift Operators	Yes	Arithmetic/Logical Operations/Left/Right Shift

Point Operations

OPERATOR	INC	NOTES
Thresholding	Yes	Image Analysis/Segmentation/Threshold
Adaptive Threshold	Yes	Image Analysis/Segmentation/Dynamic Threshold
Contrast Stretching	Yes	Image Processing/Histograms/Stretch
Hist. Equalization	Yes	Image Processing/Histograms/Equalize
Logarithm Operator	Yes	Arithmetic/Unary Operators/Logarithm
Raise to Power	Yes	Arithmetic/Unary Operators/Exponential

Geometric Operations

OPERATOR	INC	NOTES
Scale	Yes	Image Processing/Geometric Manip./Resize
Rotate	Yes	Image Processing/Geometric Manip./Rotate
Reflect	Yes	Image Processing/Geometric Manip./Flip
Translate	Yes	Image Processing/Geometric Manip./Translate
Affine Transform	Yes	Remote & GIS/Warping...

Image Analysis

OPERATOR	INC	NOTES
Intensity Histogram	Yes	Image Processing/Histograms/Histogram
Classification	Yes	Image Analysis/Classification...
Labeling	Yes	Image Analysis/Classification/Labeling

Morphology

OPERATOR	INC	NOTES
Dilation	Yes	Image Processing/Morphology/Dilation
Erosion	Yes	Image Processing/Morphology/Erosion
Opening	Yes	Image Processing/Morphology/Opening
Closing	Yes	Image Processing/Morphology/Closing
Hit/Miss Transform	No	
Thinning	No	
Thickening	No	
Skeletonization	Yes	Image Processing/Morphology/Skeletonization

Digital Filters

OPERATOR	INC	NOTES
Mean Filter	Yes	Image Processing/Spatial Filters/2D Conv + Input Sources/Input Data File/Kernels (avgNxN)
Median Filter	Yes	Image Processing/Spatial Filters/Median
Gaussian Smoothing	Yes	Image Processing/Spatial Filters/2D Conv + Input Sources/Create 2D Image/Gauss Image
Conservative Smooth	No	
Crimmins	Yes	Image Processing/Spatial Filters/Speckle Removal
Frequency Filters	Yes	Image Processing/Frequency Filters...
Laplacian Filter	Yes	Image Processing/Spatial Filters/2D Conv + Input Sources/Create 2D Image/Marr Filter
Unsharp Filter	No	

Feature Detectors

OPERATOR	INC	NOTES
Roberts Cross	Yes	Image Processing/Spatial Filters/Gradient
Sobel	Yes	Image Processing/Spatial Filters/Sobel
Canny	Yes	Image Processing/Spatial Filters/DRF Edge Extract (Actually Difference Recursive Filter which is similar)
Compass	No	
Zero Crossing	No	(Not directly anyway, but easy to do)
Line Detector	No	

Image Transforms

OPERATOR	INC	NOTES
Distance Transform	No	
Fourier Transform	Yes	Image Processing/Transforms/FFT
Hough Transform	No	

Image Synthesis

OPERATOR	INC	NOTES
Noise Generation	Yes	Input Sources/Create 2D Image/Gauss/Shot Noise

B.3 Matlab Image Processing Toolbox

Matlab is a popular and flexible mathematics package that has particular strengths in manipulating matrices and visualizing data. The Matlab Image Processing Toolkit is one of a range of extensions that can be added onto Matlab in order to provide extra functionality for specialized applications.

Matlab is marketed commercially by The MathWorks (The MathWorks, Inc, 24 Prime Park Way, Natick, MA 01760-1500; Tel:(508)653-1415; Fax:(508)653-2997; E-mail:info@mathworks.com), and is available for a wide range of machine architectures.

Matlab image processing functions are implemented as *M-files* which are called by name from the Matlab command line. The *NOTES* column in the tables below gives the name of these M-files. Matlab provides a powerful and easy to use scripting language that enables you to write additional image processing functions yourself relatively easily.

Image Arithmetic

OPERATOR	INC	NOTES
Addition	Yes	+
Subtraction	Yes	-
Multiplication	Yes	*
Division	Yes	/
Blending	Yes	Trivial to perform. e.g. 'p*X + (1-p)*Y'
Logical AND/NAND	Yes	&
Logical OR/NOR	Yes	
Logical XOR/XNOR	Yes	xor
Invert/Logical NOT	Yes	~
Bitshift Operators	No	

Point Operations

OPERATOR	INC	NOTES
Thresholding	Yes	im2bw
Adaptive Threshold	No	
Contrast Stretching	Yes	imadjust
Hist. Equalization	Yes	histeq
Logarithm Operator	Yes	log
Raise to Power	Yes	^

Geometric Operations

OPERATOR	INC	NOTES
Scale	Yes	imresize
Rotate	Yes	imrotate
Reflect	Yes	fliplr/fliped
Translate	Yes	imgcrop
Affine Transform	No	

Image Analysis

OPERATOR	INC	NOTES
Intensity Histogram	Yes	imhist
Classification	No	
Labeling	No	

Morphology

OPERATOR	INC	NOTES
Dilation	Yes	dilate
Erosion	Yes	erode
Opening	Yes	bwmorph
Closing	Yes	bwmorph
Hit/Miss Transform	Yes	bwmorph
Thinning	Yes	bwmorph
Thickening	Yes	bwmorph
Skeletonization	Yes	bwmorph

Digital Filters

OPERATOR	INC	NOTES
Mean Filter	Yes	fspecial
Median Filter	Yes	medfilt2
Gaussian Smoothing	Yes	fspecial
Conservative Smooth	No	
Crimmins	No	
Frequency Filters	Yes	Many functions...
Laplacian Filter	Yes	fspecial
Unsharp Filter	Yes	fspecial

Feature Detectors

OPERATOR	INC	NOTES
Roberts Cross	Yes	edge
Sobel	Yes	edge
Canny	No	
Compass	No	
Zero Crossing	Yes	edge
Line Detector	No	

Image Transforms

OPERATOR	INC	NOTES
Distance Transform	No	
Fourier Transform	Yes	fft2
Hough Transform	No	

Image Synthesis

OPERATOR	INC	NOTES
Noise Generation	Yes	imnoise

B.4 HIPS

HIPS is a software package for image processing that runs under the UNIX operating system. HIPS is modular and flexible, it provides automatic documentation of its actions, and is almost entirely independent of special equipment. It handles sequences of images (movies) in precisely the same manner as single frames. Programs have been developed for simple image transformations, filtering, convolution, Fourier and other transform processing, edge detection and line drawing manipulation, digital image compression and transmission methods, noise generation and image statistics computation. Over 150 such image transformation programs have been developed. As a result, almost any image processing task can be performed quickly and conveniently.

HIPS is marketed commercially by SharpImage Software, P.O. Box 373, Prince Street Station, New York, NY 10012-0007; Tel: (212) 998-7857; Email: landy@nyu.edu. It is relatively inexpensive, and is highly discounted to academic and nonprofit institutions. The software runs under virtually any UNIX environment. HIPS is supplied with source code (written in C), on-line documentation, and a library of convolution masks.

HIPS functions are generally used as UNIX shell commands, although almost all of the functionality of HIPS is available as subroutines at various levels of abstraction. HIPS has its own image header format, and comes with programs which convert to and from a wide variety of other image data formats.

Image Arithmetic

OPERATOR	INC	NOTES
Addition	Yes	addseq
Subtraction	Yes	diffseq
Multiplication	Yes	mulseq
Division	Yes	divseq
Blending	Yes	Combine scale and addseq
Logical AND/NAND	Yes	andseq
Logical OR/NOR	Yes	orseq
Logical XOR/XNOR	Yes	xorseq
Invert/Logical NOT	Yes	neg
Bitshift Operators	Yes	shiftpix

Point Operations

OPERATOR	INC	NOTES
Thresholding	Yes	thresh
Adaptive Threshold	No	
Contrast Stretching	Yes	scale, histostretch
Hist. Equalization	Yes	histoeq
Logarithm Operator	Yes	logimg
Raise to Power	Yes	powerpix

Geometric Operations

OPERATOR	INC	NOTES
Scale	Yes	imresize
Rotate	Yes	rotate90, rotate180, hfant
Reflect	Yes	pictranspose, reflect
Translate	Yes	drift, wrapping
Affine Transform	Yes	affine

Image Analysis

OPERATOR	INC	NOTES
Intensity Histogram	Yes	histo, disphist, seehist
Classification	Yes	(Various tools in Allegory Extensions)
Labeling	Yes	cobjects, label

Morphology

OPERATOR	INC	NOTES
Dilation	Yes	morphdilate
Erosion	Yes	morpherode
Opening	Yes	mopen
Closing	Yes	mclose
Hit/Miss Transform	No	
Thinning	Yes	thin
Thickening	Yes	thicken
Skeletonization	No	

Digital Filters

OPERATOR	INC	NOTES
Mean Filter	Yes	meanie
Median Filter	Yes	median
Gaussian Smoothing	Yes	dog, mask, gauss
Conservative Smooth	No	
Crimmins	No	
Frequency Filters	Yes	Many functions...
Laplacian Filter	Yes	imgtopyr, mask
Unsharp Filter	Yes	mask

Feature Detectors

OPERATOR	INC	NOTES
Roberts Cross	Yes	mask
Sobel	Yes	mask
Canny	Yes	canny, deriche
Compass	No	
Zero Crossing	Yes	zc
Line Detector	No	

Image Transforms

OPERATOR	INC	NOTES
Distance Transform	No	
Fourier Transform	Yes	fourtr, inv.fourtr (also: dct, walsh)
Hough Transform	No	

Image Synthesis

OPERATOR	INC	NOTES
Noise Generation	Yes	bnoise, gnoise, noise

Appendix C

HIPRscript Reference Manual

Note that this section is really only relevant to the person responsible for installing and maintaining HIPR on your system.

What is HIPRscript?

See the introductory section on *Making Changes with HIPRscript* (p.34) for an introduction to the role of HIPRscript in generating HIPR.

How does HIPRscript Work?

HIPRscript is a specially designed language for use with HIPR, similar in respects to both HTML and \LaTeX . It is designed to allow the HIPRscript author to express the information needed by both hypermedia and hardcopy versions of HIPR in a relatively high-level way and it is readily translatable into both HTML and \LaTeX using the supplied translation programs.

Almost 300 HIPRscript source files go together to make HIPR. In general, each HIPRscript source file gives rise to one HTML file and one \LaTeX file. Each HTML file corresponds to a ‘scrollable page’ of hypertext, while the \LaTeX files are merged together to generate the hardcopy version of HIPR. There are a few exceptions to this rule — for instance the `.loc` source files used for entering local information are *included* into other files.

Another important exception to this rule is the top-level file for the HTML and \LaTeX versions. The HTML top-level file is called `hipr_top.htm` and lives in the `html` sub-directory. The \LaTeX file lives in the `tex` sub-directory and is called `hipr_top.tex`. These are very different from one another, and so are not generated from a common HIPRscript source file (*i.e.* there is no `hipr_top.hpr` file). You should be careful not to delete these files since they cannot be regenerated, unlike most of the other HTML and \LaTeX files.

To convert HIPRscript into HTML and \LaTeX , a *Perl* program called `hiprgen.pl` is used. This program can be found in the `progs` sub-directory. The effect of the program when run on a HIPRscript source file is to generate corresponding HTML and \LaTeX files in the appropriate directories.

To run `hiprgen.pl` you need to have at the very least a recent version of *Perl* installed on your system. In addition, if you wish to have the program automatically generate equations, figures and thumbnails as described below, then you will have to install additional utilities. The *Installation Guide* (p.32) has all the details.

Apart from the `hpr`, `html` and `tex` sub-directories, four other sub-directories are also important to the running of HIPRscript.

The `eqns` sub-directory contains inline images representing equations for use in HTML documents.

These image files are generated automatically by `hiprgen.pl` from information in the HIPRscript files and are incorporated into the displayed document by the HTML browser.

The `figs` sub-directory contains the images used for figures in two different formats: GIF for HTML pages, and encapsulated PostScript for inclusion into L^AT_EX output. As a HIPRscript author you must create the PostScript version of the figure yourself and put it in this directory. `hiprgen.pl` will then create a matching GIF file automatically if one does not already exist.

The `thumbs` sub-directory contains ‘thumbnails’ (miniature versions of images that are used for imagelinks). These are normally generated automatically by `hiprgen.pl` from corresponding full-size images.

The `index` sub-directory contains information used in generating the HIPR main index. The files in this directory are created automatically by `hiprgen.pl` and should not be edited by hand.

Running the Translation Program

The exact method of running `hiprgen.pl` depends upon the computer system you are using. Since it is a Perl program you need to have a Perl interpreter installed somewhere. You must then use this interpreter to run `hiprgen.pl`. Make sure that the ‘current’ or ‘working’ directory is the `src` sub-directory when you run it. You must also pass the program a command line argument which is simply the name of the source file to be translated, without the `.hpr` extension.

For instance, on my UNIX system, in order to translate `dilate.hpr` into HTML and L^AT_EX, I type (from within the `src` sub-directory):

```
perl ../progs/hiprgen.pl dilate
```

The program will run and the appropriate HTML and L^AT_EX files will be generated. If there are any errors, the translator will stop with an error message and no output files are generated. Error messages are detailed in a later section.

An Introduction to HIPRscript Syntax

It is useful at this point to take a look at the contents of a typical HIPRscript file. For instance the file `dilate.hpr` starts something like:

```
\links{}{erode}{morops}
\index{Dilation}{\section{Dilation}}
\title{Morphology - Dilation}

\strong{Common Names:} Dilate, Grow, Expand

\subsection{Brief Description}

Dilation is one of the two basic operators in the area of
\ref{matmorph}{mathematical morphology}, the other being
\ref{erode}{erosion}. It is typically applied to
...
```

Have a look at the dilation worksheet (p.118) to see what this becomes (note that if you are using Netscape or Mosaic then you might want to click on the link with the middle mouse button which will display the worksheet in a separate window).

As with both HTML and L^AT_EX, a HIPRscript source file is an ASCII file containing a mixture of raw text and *tags* which define how that raw text is to be displayed. Tags in HIPRscript have the following syntax:

- Each tag starts with a backslash: \
- This is then followed by the *tagname*. Most tagnames contain only alphanumeric characters, but there are also a few tagnames consisting of a single non-alphanumeric character. Note that there is no space between the backslash and the tagname.
- Finally there comes the list of arguments associated with the tag. Each separate argument is enclosed in its own pair of curly braces: { . . . }. Note that there must not be any whitespace between arguments, or between the arguments and the tagname.
- A tag with no arguments is terminated by any non-alphanumeric character except an opening brace ({}). A tag with arguments is terminated by any character at all except an opening brace. A caret (^) can be used to terminate any tag but has a special meaning described below.

The simplest tags take no arguments. For instance the tag: `\times` produces a multiplication symbol like this: \times .

Slightly more complicated are tags which take a single argument. An example of this is the `\em` tag which produces *emphasized italic text*. For instance, the phrase *emphasized italic text* in the last sentence was produced by the following HIPRscript fragment: `\em{emphasized italic text}`.

Finally, some tags take multiple arguments. A common example of this is the `\ref` tag which is used for cross references to other HIPR pages. To create a reference to the worksheet on dilation for example, we could write `\ref{dilate}{This is a link to the dilation worksheet}` which produces the following: This is a link to the dilation worksheet (p.118).

Many tags can be nested inside one another. If we modify the example in the last paragraph to `\ref{dilate}{This is a link to the \em{dilation} worksheet}`, then we get: This is a link to the *dilation* worksheet (p.118).

Arguments can usually contain newlines without any problems, but you should make sure that there are no newlines *between* arguments belonging to the same tag. For instance:

```
\ref{dilate}{This is a link
to the dilation worksheet}
```

is fine, whereas:

```
\ref{dilate}
{This is a link to the dilation worksheet}
```

is not.

One very important point is that like \LaTeX and HTML, HIPRscript pays very little attention to the amount of whitespace (spaces, tabs and newlines) that you put in your document. One space between words is treated exactly the same as one hundred spaces — both will cause just a single space to be displayed between the words.

Single newlines in sentences are also largely ignored. In general HIPRscript will decide for itself where it wants to break lines in order to make them fit onto the page. Note: In actual fact this tricky decision is taken care of by the magic of \LaTeX and HTML browsers.

Two or more newlines in succession *are* treated as significant. They indicate that you want a *paragraph break* at that point. This will normally cause a vertical space to be inserted between the preceding and following lines on the page.

Normally line-breaks are inserted wherever \LaTeX or your HTML browser feels appropriate. This is usually when the current line has as much text as will fit on it already. Sometimes though, we would like to ensure that a line-break does *not* occur between two words. For instance the phrase ‘Section 2’ would look odd if a line-break appeared between ‘Section’ and ‘2’. Therefore HIPRscript

allows you to specify a ‘non-breaking space’ which will never be broken over two lines. The symbol for this is just ‘~’. Using this feature, the phrase above would be entered as: `Section~2`.

As mentioned earlier, tags that don’t take any arguments are terminated by any non-alphanumeric character. This can cause a problem if you *do* want to immediately follow such a tag with an alphanumeric character without putting a space in between. For instance if you want to display ‘2×2’, you cannot write `2\times2` since `\times2` is not a recognized tag and will cause an error. And if you write `2\times 2` then what you get is ‘2× 2’, *i.e.* with an extra unwanted space. The solution is to terminate the tag with a caret. This special character terminates a tag without being printed itself. So `2\times^2` will produce the display you want. If you do want to follow a tag immediately with a caret for some reason, then simply use two of them. The other time you might want to use a caret to terminate a tag is if you want to put a backslash at the end of an argument. Without the caret, the final backslash would escape the closing brace and hence cause H_{IP}Rscript to think that you haven’t terminated the argument properly. So you would write: `...\\^`.

Since the backslash and braces are special characters, there are tags for displaying them normally. `\\`, `\{` and `\}` will display `\`, `{` and `}` respectively.

It is often useful to be able to write into H_{IP}Rscript files chunks of text that will be completely ignored by the interpreter. Such a piece of text is known as a *comment*. They can be used to explain to anyone reading the source file why you chose to say certain things or why you chose to express things in certain ways. This can be useful if someone else wishes to work on the file after you have finished with it, or if you wish to be reminded what you were doing last when you come back to it. Comments can also be used to force the translator to ignore large chunks of your source file. This is particularly useful for tracking down errors in your source file. There are two forms of comments in H_{IP}Rscript:

Line Comments Anything on a line following a hash sign `#` is ignored, including the hash sign itself. If you want a hash sign then use the `\#` tag.

Block Comments The special tag `\comment{...}` simply causes everything in its single argument to be ignored. This is good for commenting out large blocks of text in a single swoop.

Comments can be nested inside one another with no problems.

The Elements of H_{IP}Rscript

We now present a complete listing of all the tags that can be used in H_{IP}Rscript, together with explanations and hints for their use.

`#` Line comment; the remainder of the current line is ignored.

`~` Non-breaking space; a space that will never be substituted by a line-break.

`\\` Backslash: `\`

`\#` Hash sign: `#`

`\blob` or `\blob{COLOR}` Produces a small colored blob in the HTML output *only*. The optional `COLOR` argument can take the values `yellow` or `red`. The default value is `yellow`.

`\br` Forces a line break. Note that in a few situations this will cause a L^AT_EX error since L^AT_EX doesn’t like being told where to break lines. If this happens use the `\html` tag to ignore the line break in the L^AT_EX output.

`\chapter{HEADING}` Starts a new chapter with a title given by `HEADING`. This is the second largest document division in H_{IP}R.

`\comment{TEXT}` Block comment — simply ignores everything within its argument and produces no output.

`\deg` Degree symbol: °

`\dd{TEXT}` See the `\description` tag.

`\description{DESCRIPTIONLIST}` Description list. One of three HIPRscript list types — this one is used for lists where each list item consists of a brief label (known as the *topic*) followed by a block of text (known as the *discussion*). The argument of this tag must consist solely of alternate `\dt` and `\dd` tags. The `\dt` tag comes first and its argument gives the *Description Topic*. The `\dd` tag then follows with the *Description Discussion* as its argument. Any number of pairs of `\dt` and `\dd` tags may be inserted.

Example — the following fragment of HIPRscript:

```
\description{
  \dt{HIPR}
  \dd{The Hypermedia Image Processing Reference.}
  \dt{HIPRscript}
  \dd{The language used to create HIPR.}
}
```

produces the following effect:

```
HIPR The Hypermedia Image Processing Reference.
HIPRscript The language used to create HIPR.
```

Note that the `\dd` tag's argument can contain any other tags, including additional lists. The `\dt` tag on the other hand should only contain raw text and appearance changing tags such as `\em`.

`\dt{TEXT}` See the `\description` tag.

`\eg` Inserts an italicized 'e.g.': *e.g.*

`\em{TEXT}` Causes TEXT to be displayed in an *italic* font.

`\enumerate{ENUMERATELIST}` Enumerated list. One of three HIPRscript list types — this one is used for ordered lists where each item is to be numbered in consecutive order. HIPRscript will take care of the numbering automatically. The argument of this tag consists of a mixture of text and `\item` tags. Each `\item` tag encountered tells HIPRscript that a new list item is starting, complete with a new number.

Example — the following HIPRscript fragment:

```
\enumerate{
  \item Capture an image.
  \item Choose a threshold value.
  \item Apply thresholding to produce a binary image.
}
```

produces the following effect:

1. Capture an image.
2. Choose a threshold value.
3. Apply thresholding to produce a binary image.

`\enumerate` lists can contain most other sorts of tag, including other list tags. Usually a different numbering scheme (*e.g.* roman numerals or letters) is used for nested `\enumerate` lists.

- `\eqnd{EQUATION}{GIF_FILE}` or `\eqnd{EQUATION}{GIF_FILE}{SIZE}` Displayed equation. Used for mathematical equations which are to be set apart from the text that describes them. The EQUATION argument describes the equation using *LaTeX format*. *i.e.* the code that goes here is exactly what you would type to produce the equation in *LaTeX*. The GIF_FILE argument is the name of a file (without the .gif suffix) in the eqns sub-directory where an image representing that equation will be stored. It is possible to get HIPRscript to generate the GIF file automatically from the EQUATION argument, but note that this involves many extra complications which are described below in the subsection on *Using Figure, Equation and Imageref Tags* (p.265). The optional SIZE argument can take values of `small`, `large` or `huge` and determines the scaling of the displayed equation. The default is `large`.
- `\eqnl{EQUATION}{GIF_FILE}` Similar to the `\eqnd` tag except that it produces an in-line equation and takes no optional SIZE argument.
- `\etc` Inserts an italicized ‘etc.’: *etc.*
- `\fig{FIG_FILE}{CAPTION}` or `\fig{FIG_FILE}{CAPTION}{SCALE}` Include a figure at this point in the text. The FIG_FILE argument refers to two similarly named files in the figs sub-directory that contain the image to be included in two different formats, GIF and encapsulated PostScript. The FIG_FILE argument should specify just the stem-name of the files. To this HIPRscript will add .eps for the PostScript file and .gif for the GIF file. It is possible to get HIPRscript to generate the GIF file automatically from the PostScript file, but note that this involves many extra complications which are described below in the subsection on *Using Figure, Equation and Imageref Tags* (p.265). The CAPTION argument gives the text that will go with the figure, and may contain appearance changing tags such as `\em` and also cross-reference tags such as `\ref`. The optional SCALE argument gives the scaling of the figure. A size of 1 (the default) means that the image will be included at its ‘natural’ size. A number less than this will reduce the size, larger numbers will increase it. Note that HIPRscript will assign a figure number to your figure automatically.
- `\figref{FIG_FILE}` Used to reference a particular figure in the text. The FIG_FILE argument should match up with the FIG_FILE argument of a `\fig` tag somewhere in the same HIPRscript file. That will be the figure to which the `\figref` refers. The visible effect of the tag is to insert the text: ‘Figure *N*’ where *N* is the number of the figure concerned.
- `\hb` Places a solid colored horizontal dividing bar across the screen in the HTML output *only*.
- `\hr` Places a thin horizontal dividing rule across the screen in the HTML output *only*.
- `\html{TEXT}` Process the TEXT argument as if it were normal HIPRscript, but only produce HTML output. Nothing appears in the *LaTeX* output for this tag.
- `\ie` Inserts an italicized ‘i.e.’: *i.e.*
- `\imageref{IMAGE_FILE}` or `\imageref{IMAGE_FILE}{MODE}` Refer to an image in the images sub-directory. The IMAGE_FILE argument should give the name of the image file concerned, minus any file suffix such as .gif. The visible effect of this tag depends upon whether you are looking at the HTML or the *LaTeX* output. In the HTML output, by default it creates a hyperlink to the named image, in the form of a small thumbnail image. The thumbnail image is in fact just a reduced size version of the full image and is found in the thumbs sub-directory. It is possible to get HIPRscript to generate the thumbnail automatically from the full size image, but note that this involves many extra complications which are described below in the subsection on *Using Figure, Equation and Imageref Tags* (p.265). In the *LaTeX* output, this tag simply prints the IMAGE_FILE argument in `typewriter` font. The optional argument MODE can be used to alter the appearance of the HTML output slightly (it doesn’t affect the *LaTeX* though). Setting MODE to `text` means that the imagelink will simply appear as the name of the image file rather than as a thumbnail. Setting MODE to `both` causes the link to appear both as text *and* as a thumbnail. Setting MODE to `thumbnail` produces the default behavior.

`\inc{FILE}{TEXT}` Includes the HIPRscript file named by the `FILE` argument plus a `.hpr` extension, into the current file. In the HTML output, this inclusion appears as a hyperlink to the named file. The text of the hyperlink is given by the `TEXT` argument. In the \LaTeX output there is no such link — the named file is included as if its entire contents had been typed into the current file at that point. The `TEXT` argument is ignored in the \LaTeX output.

`\inc2{FILE}{TEXT}` Includes the HIPRscript file named by the `FILE` argument plus a `.hpr` extension, into the current file. This is rather like the `\inc` tag except that `TEXT` is printed in both the HTML *and* the \LaTeX output. In the HTML output, `TEXT` acts as a hyperlink to the included file. In the \LaTeX output, `TEXT` is merely printed immediately before the named file is included.

`\inc3{FILE}{TEXT}` Acts identically to the `\inc` tag, except that `FILE` is treated as a full filename of the file to be linked in, including any path information.

`\index{ENTRY1}{ENTRY2}...{TEXT}` Creates an entry in the HIPR index. `ENTRY1`, `ENTRY2` and so on give the names of topic entries in the index. Each index entry may have up to three levels of nesting in order to specify the entry precisely. An entry with more than one level of nesting is indicated by a `ENTRY` argument containing one or two `|` symbols. The `|` symbols are used as separators between the different levels in the entry.

For instance if we wanted an index entry to appear under the general category of ‘Edge detectors’ and then within that category, under ‘Canny’, then we would have an entry argument that looked like: `{Edge detectors|Canny}`.

Note that every `ENTRY` argument in the *whole* of HIPR must be unique. We can only have one place in HIPR which has the index entry `{Edge detectors|Canny}` for instance. However, we *could* have another entry for `{Edge detectors|Roberts}` for instance.

The `TEXT` argument indicates which chunk of HIPRscript is to be pointed to by the index entry and can contain almost anything. It should not however contain any `\target` or `\title` tags.

A particular place in HIPR can have more than one index entry associated with it. Simply use as many `ENTRY` arguments as you need.

`\input{FILE}` Like the `\inc` tag, this tag includes a named file into the current one. This time however, the effect for both HTML and \LaTeX is as if the contents of the named file had been inserted directly into the current file in the tag’s place. No links are created.

`\item` Marks the beginning of a list item in `\enumerate` and `\itemize` lists. It gives an error if used anywhere else.

`\itemize{ITEMIZE_LIST}` Itemized list. One of three HIPRscript list types — this one is used for unordered lists where each item is to be marked with a ‘bullet’, but not numbered. The argument of this tag consists of a mixture of text and `\item` tags. Each `\item` tag encountered tells HIPRscript that a new list item is starting, and causes a bullet to be printed.

Example — the following HIPRscript fragment:

```
\itemize{
\item Available on-line.
\item Extensive cross-referencing.
\item Uses Netscape browser.
}
```

produces the following effect:

- Available on-line.
- Extensive cross-referencing.
- Uses Netscape browser.

`\itemize` lists can contain most other sorts of tag, including other list tags. Usually different bullet styles are used for nested `\itemize` lists.

`\latex{TEXT}` Process the TEXT argument as if it were normal HIPRscript, but only produce L^AT_EX output. Nothing appears in the HTML output for this tag.

`\LaTeX` A special tag that produces a nicely formatted version of the L^AT_EX tradename. In the hardcopy version it prints as: L^AT_EX . In the hypermedia version it prints simply as: ‘LaTeX’.

`\links{LEFT}{RIGHT}{UP}` Causes navigation buttons to appear in the HTML version of HIPR. It has no effect on the L^AT_EX output. LEFT, RIGHT and UP should be the names of other HIPRscript files within HIPR minus the .hpr suffix. The navigation buttons appear at the top of HTML page, and, if the page is more than about a screenful, are duplicated at the bottom. If there is no appropriate link for any of the arguments, then simply use a pair of empty braces for that argument, and no navigation button will be generated for that direction.

For instance the tag:

```
\links{}{erode}{morops}
```

creates a ‘right’ navigation button leading to the `erode.htm` worksheet, and an ‘up’ button leading to the `morops.htm` section.

See the section on *Navigation Buttons* (p.17) for more information about navigation buttons.

`\minus` Minus sign. This is slightly different from a hyphen and should be used instead of a simple ‘-’ when a minus sign is intended. *e.g.* `\minus^7` produces ‘-7’. It will probably only cause a noticeable difference in the hardcopy output.

`\newpage` Causes a pagebreak. Only affects the L^AT_EX output.

`\part{HEADING}` Starts a new ‘part’ with a title given by the HEADING argument. This is the largest document division in HIPR.

`\pm` Plus-or-minus symbol: ±

`\quote{TEXT}` Indents the HIPRscript contained in the TEXT argument from the left margin slightly.

`\rawhtml{HTML}` Passes the HTML argument directly into the HTML output with no processing. It differs from the `\html` tag in that the HTML argument is *not* treated as HIPRscript to be further processed. Has no effect on the L^AT_EX output.

`\rawlatex{LATEX}` Passes the LATEX argument directly into the L^AT_EX output with no processing. It differs from the `\latex` tag in that the LATEX argument is *not* treated as HIPRscript to be further processed. Has no effect on the HTML output.

`\ref{FILE}{TEXT}` In the HTML output, this creates a hyperlink using TEXT pointing at the HIPRscript file specified by FILE (minus the .hpr file extension as usual).

`\section{HEADING}` Starts a new section with a title given by the HEADING argument. This is the third largest document division in HIPR.

`\sqr` A squared symbol: ²

`\strong{TEXT}` Causes the HIPRscript within TEXT to be displayed in a **bold font**.

`\subsection{HEADING}` Starts a new subsection with a title given by the HEADING argument. This is the fourth largest document division in HIPR.

`\subsubsection{HEADING}` Starts a new subsubsection with a title given by the HEADING argument. This is the fifth largest document division in HIPR, and the smallest.

`\tab{COL1}{COL2}...` Used for specifying the data to go into a table created with the `\table` tag. The text in COL1 goes in the first column, the text in COL2 goes in the second column and so on. There should be the same number of arguments as there are data columns in the table, and the number of characters in each argument should be less than the width of the table columns. See the `\table` tag for details.

`\table{COL1}{COL2}...{DATA}` Creates a table at the current place in the document. The last argument contains the actual data that will go into the table. The previous arguments define the column layout of the table. If a COL argument is a number, then that indicates a data column. The number gives the width in characters of that column. Data appears left justified within the column. If the COL argument is a | character, this indicates an internal vertical dividing line. If you use two such arguments in a row, then you will have a double vertical dividing line.

The DATA argument contains the body of the table. It must consist solely of `\tab` tags (specifying the data to go in each column), and `\tabline` tags (specifying horizontal lines in the table).

Note that HIPRscript will automatically put lines around the outside of a table and so these do not need to be specified.

As an example, the following fragment of HIPRscript:

```
\table{6}{|}{|}{8}{8}{8}{
\tab{Type}{Test A}{Test B}{Test C}
\tabline
\tab{1}{Yes}{Yes}{No}
\tab{2}{No}{Yes}{No}
\tab{3}{Yes}{Yes}{Yes}
}
```

produces the following:

Type	Test A	Test B	Test C
1	Yes	Yes	No
2	No	Yes	No
3	Yes	Yes	Yes

Note that at the time of writing not all browsers support proper tables, and so tables are implemented rather crudely in HTML output. Significantly better looking tables appear in the L^AT_EX output.

`\tabline` Used between `\tab` tags within a table to produce a horizontal line across the table. Note that the `\table` tag itself produces horizontal lines at the top and bottom of the table.

`\target{LABEL}{TEXT}` Associates the chunk of HIPRscript in TEXT with LABEL for reference to by a `\textref` tag. LABEL must be a single word and can only contain alphanumeric characters. TEXT can be any bit of HIPRscript.

`\textref{FILE}{LABEL}{TEXT}` Like the `\ref` tag, this creates a hyperlink around TEXT to the HIPRscript file named in FILE. However, whereas following a `\ref` tag automatically takes a user to the top of that file, the `\textref` tag allows you to jump into the middle of a file. The point to be jumped to must be marked with a `\target` tag and that tag's LABEL argument must match the LABEL argument used here.

`\tilde` Tilde: ~

`\times` A multiplication symbol: ×

`\title{TEXT}` Most HTML documents are associated with a *document title* which does not appear on the page itself, but which is often shown separately by the HTML browser. This tag allows you to specify the HTML document title. It has no visible effect on the L^AT_EX output, although it does create an essential internal L^AT_EX cross-reference. This tag **must** be positioned near the top of the HIPRscript file, just after the first document sectioning command in the file. (*i.e.* after the first `\chapter`, `\section` *etc.* tag in the file). Therefore the only tags that normally appear before a `\title` tag are a sectioning tag and a `\links` tag. The `\title` tag should only be used once in any given file.

`\tt{TEXT}` Causes the HIPRscript contained in TEXT to be displayed in a **typewriter font**.

`\verbatim{TEXT}` TEXT is displayed in both HTML and L^AT_EX output exactly as it appears in the HIPRscript file. Unlike normal HIPRscript text, all whitespace is preserved as is. The text is also normally displayed in a fixed space typewriter font.

Making an Index with HIPRscript

One of the more useful features of HIPR is its extensive index. As mentioned above, index entries are produced using the `\index` tag within HIPRscript pages. However, whereas most sections of HIPR are produced by single HIPRscript files, the index is contributed to by almost all the HIPRscript files. Hence a slightly different procedure is used to generate the index.

What actually happens is that every time a HIPRscript file is processed, all the index entry information in that file is written into a corresponding *IDX file* in the `index` sub-directory. To generate the index section, what we have to do is scan this sub-directory, collate all the information in all the IDX files there, and then use this information to produce the index pages. In fact, the index section is itself written to the `src` sub-directory as a HIPRscript file. The scanning and analysis of the IDX files are performed by another Perl program called `hiprindx.pl`, also found in the `progs` sub-directory.

Finally the HIPRscript file for the index is processed as normal to produce the required HTML and L^AT_EX index pages.

Summarizing, the sequence is as follows:

1. Run `hiprgen.pl` on each HIPRscript file in the `src` sub-directory to both generate the corresponding HTML and L^AT_EX files, and also to write the relevant index information to IDX files in the `index` sub-directory.
2. Run `hiprindx.pl` in order to analyze the IDX files and generate `index.hpr` in the `src` sub-directory.
3. Finally, run `hiprgen.pl` on `index.hpr` in order to produce the HTML and L^AT_EX index sections.

Note that this procedure can be somewhat simplified through the use of a *makefile* as described later (p.264).

HIPRscript Error Messages

Error messages are intended to be fairly self-explanatory. A typical one runs something like:

```
hiprgen - Error [chngchip(2)]: End of file in argument of \index
```

which means that an error was found on line 2 of the file `chngchip.hpr`. In this case the translator is saying that it encountered the end of the file before it had finished reading the argument of an `\index` tag. The most likely cause of this is a missing end-brace (`}`).

We present here a list of all the error messages you are likely to encounter when using HIPRscript, together with brief explanations of the likely cause.

`'\'` character at end of block It is not legal to have a single backslash at the end of a HIPRscript file or at the end of an argument to a tag that is to be processed as HIPRscript. If you do want a backslash, then you must use `\\^`. The final `^` is necessary to prevent the closing brace of the argument being escaped by the `\`.

Couldn't open *filename.hpr* Couldn't find the HIPRscript file specified on the command line to `hiprgen.pl` as *filename*. Make sure that the filename is spelled correctly and that the file

does exist and is not read protected. Also make sure that when you run `hiprgen.pl` the current *working directory* is `src`. Finally note that when specifying *filename* on the command line, you should not add the suffix `.hpr`.

Couldn't open `../html/filename` Couldn't create the HTML output file corresponding to the current HIPRscript file. Make sure that the `html` sub-directory exists and is writeable. Check also that if the HTML file to be produced already exists, then it should also be writeable, so that it can be overwritten with the new version.

Couldn't open `../index/filename` Couldn't create the IDX output file corresponding to the current HIPRscript file. Make sure that the `index` sub-directory exists and is writeable. Check also that if the IDX file to be produced already exists, then it should also be writeable, so that it can be overwritten with the new version.

Couldn't open `../tex/filename` Couldn't create the L^AT_EX output file corresponding to the current HIPRscript file. Make sure that the `tex` sub-directory exists and is writeable. Check also that if the L^AT_EX file to be produced already exists, then it should also be writeable, so that it can be overwritten with the new version.

Could not open *filename* for `\input` statement The file specified in an `\input` tag could not be found. Check that the file exists in the `src` sub-directory and is readable. Note also that *filename* should be specified without the `hpr` suffix.

Couldn't open temporary file `../html/tmp_hipr.html` Couldn't create the temporary file that HIPRscript uses while building up the HTML output. Check that the `tex` sub-directory is writeable to and that the temporary file does not already exist.

Couldn't open temporary file `../index/tmp_hipr.idx` Couldn't create the temporary file that HIPRscript uses while building up the index output. Check that the `tex` sub-directory is writeable to and that the temporary file does not already exist.

Couldn't open temporary file `../tex/tmp_hipr.tex` Couldn't create the temporary file that HIPRscript uses while building up the L^AT_EX output. Check that the `tex` sub-directory is writeable to and that the temporary file does not already exist.

End of file in argument of `\tag` No terminating brace was found for the named tag before the end of the file was reached. Check to see that each opening brace at the start of an argument has a matching brace at the end of the argument. Also make sure that braces that do not delimit arguments are escaped with a backslash, *e.g.* `\{`

`\fig` scale parameter must be a number The `\fig` tag's scale parameter must be a number. Note that it can be a floating point number, *e.g.*: `0.8`.

Found `\dd` special outside of `\description` environment The `\dd` tag can only be used inside a `\description` list.

Found `\dt` special outside of `\description` environment The `\dt` tag can only be used inside a `\description` list.

Found `\item` special outside of `\enumerate` or `\itemize` environment The `\item` tag should only be used within a `\enumerate` or `\itemize` list.

Invalid `\blob` color An invalid argument was given to the `\blob` tag. The only valid arguments are `red` or `yellow`. Note that the argument is in fact optional.

Invalid size argument to `\eqnd`: *text* The string *text* was passed as an optional size argument to a `\eqnd` tag. The only allowed values are `small`, `large` and `huge`.

Must have at least two arguments to `\index` tag An `\index` tag must have at least two arguments — one for the index entry and one being the bit of HIPRscript to be pointed to by the index entry. There may be more than one index entry argument, however.

-
- Non alphanumeric character in figure label: *text* The figure name passed as the first argument to a `\fig` tag must only contain alphanumeric characters.
- Non alphanumeric character in referenced anchor: *text* The second argument to a `\textref` tag must match a corresponding `\target` tag's first argument, and hence must only contain alphanumeric characters.
- Non alphanumeric character in referenced filename The first argument to a `\ref` or `\textref` tag must refer to a HIPRscript file and must only contain alphanumeric characters.
- Non alphanumeric character in target anchor: *text* The label specified as the first argument of a `\target` tag must only contain alphanumeric characters.
- Optional second argument to `\imageref` must be "text", "thumbnail" or "both" An invalid argument was passed to a `\imageref` tag.
- Table entry: *text* too wide All columns in a table created with a `\table` tag are of the same width, which is specified in one of the arguments to that tag. Subsequent `\tab` arguments which give the data to go into each column must not contain data that *equal* or exceed this width.
- Usage - `hiprgen.pl source_filename` When started, `hiprgen.pl` must be passed a single argument specifying the HIPRscript file to be processed.
- Unreferenced figure label: *text* A figure created by a `\figure` must have been *previously* referenced in the text by a corresponding `\figref` tag.

Using Makefiles with HIPRscript

In order to keep everything in order, whenever you make a change to a HIPRscript file, you should regenerate the corresponding L^AT_EX and HTML files as well. However, running `hiprgen.pl` on each such file, and keeping track of which files you've changed and which you haven't is a tedious and error-prone job. Fortunately there is a better way using a *make* utility.

Make utilities solve exactly the problem described above of keeping track of large collections of files which depend upon one another, and of regenerating just those files that need to be regenerated in a simple way. They are available for almost all computer systems, and can generally be obtained free from the Internet at the standard software archives for your machine if you don't have one already.

A make utility is used in conjunction with a special text file known as a *makefile*. This file is conventionally just called `makefile` and in HIPR you will find a ready prepared one in the `src` sub-directory suitable for use with UNIX make utilities (and with many MS-DOS implementations as well that automatically convert `/` to `\` in pathnames). If you are not using one of these systems and wish to use the makefile then you will need to edit it slightly in order to set the pathnames to the important sub-directories correctly. Look at the top of the supplied makefile for guidance and consult the documentation on the make utility for your system for details.

You do not need to understand what this file does in order to use it — just make sure that the current working directory is the `src` sub-directory, and then run the make program, normally by just typing `make`. If you have installed everything else correctly then this is all you will need to do. The make program will check to see which HIPRscript files have been modified since the corresponding HTML and L^AT_EX files were created, and will run `hiprgen.pl` on just those files.

The makefile will also allow you to generate the index relatively painlessly. Simply run `make` with the argument 'index', *e.g.* by typing `make index`.

If you cannot use a make utility then it is of course still possible to regenerate HIPRscript files by hand and this is entirely appropriate for relatively small changes.

Using Figure, Equation and Imageref Tags

The tags `\fig`, `\eqn1`, `\eqnd` and `\imageref` all involve special files in addition to the standard HTML and \LaTeX files that are created from a HIPRscript file. These additional files sometimes need to be supplied by the writer of the HIPRscript file, and in other cases, they can be generated automatically by `hiprgen.pl`. The following sections describe these additional files and where they come from.

Note that the facilities that enable `hiprgen.pl` to generate these additional files automatically are amongst the least portable aspects of HIPR. They were used extensively during the development of HIPR and are included as a bonus here for those experienced enough to make use of them. However, we cannot provide any support or advice other than that supplied here. Note that you should not even begin to think about using this feature unless you have a UNIX system. You will also need to obtain several other programs and utilities such as the PBMplus image manipulation utilities and GhostScript. Details are given in the relevant section of the *Installation Guide* (p.34).

Since the automatic generation of additional files for use with figures, equations and `\imageref` tags is so non-portable, this feature is disabled by default.

`\fig` Tags

With both HTML and \LaTeX , the picture that goes into a figure is included into the HTML or \LaTeX from an external file in the `figs` sub-directory. In the case of HTML, this picture must be a GIF image file, whereas in the case of \LaTeX , the picture must be stored in encapsulated PostScript (EPS) format. Therefore both GIF and EPS versions of each figure must be available in the `figs` sub-directory. The two versions are distinguished by their different file extensions: `.gif` and `.eps`, but otherwise have identical stemnames (which must match up with the first argument of a corresponding `\fig` tag).

It is possible to get `hiprgen.pl` to generate the GIF file automatically from the corresponding EPS file. This requires that you first install the special HIPR utility `hfig2gif`, as well as the PBMplus library, GhostScript and `pstoppm.ps`. Then you must edit the `hiprgen.pl` file, and near the top of the file you should change the line:

```
$AUTO_FIGURE = 0;
```

to:

```
$AUTO_FIGURE = 1;
```

From this point on, if, during the course of processing a HIPRscript file, a `\fig` tag is encountered for which there is a corresponding EPS file, but no GIF file, then a corresponding GIF file will be generated from the EPS file. If you subsequently change the EPS file and you want to force `hiprgen.pl` to regenerate a new matching GIF file, then simply delete the old GIF file.

`\eqn1` and `\eqnd` Tags

At the time of writing, HTML provides no support for mathematical equations, unlike \LaTeX which does. This situation may be remedied in the future, but for now we have had to use a workaround. Therefore, equations in HTML are included as small in-line images stored as GIF files in the `eqns` sub-directory. \LaTeX does provide support for equations, and so no external files are required for the \LaTeX versions of equations.

While these image-equations for use with HTML could be generated by hand, it is possible to get `hiprgen.pl` to generate the GIF file automatically from the \LaTeX equation code contained in the equation tag. This requires that you first install the special HIPR utility `heqn2gif`, as well as the PBMplus library, GhostScript, `pstoppm.ps` and \LaTeX . Then you must edit the `hiprgen.pl` file, and near the top of the file you should change the line:

```
$AUTO_EQUATION = 0;
```

to:

```
$AUTO_EQUATION = 1;
```

From this point on, if, during the course of processing a HIPRscript file, a `\eqn1` or `\eqnd` tag is encountered for which there is no corresponding GIF file then one will be generated from the L^AT_EX description of the equation. If you subsequently change this description and you want to force `hiprgen.pl` to regenerate a new matching GIF file, then simply delete the old GIF file.

`\imageref` Tags

In the HTML output files, `\imageref` tags cause a hyperlink to an image in the `images` sub-directory to be created. By default, this *imagelink* takes the form of a miniature version of the full-size image, known as a *thumbnail*. This thumbnail must be a GIF file and is stored in the `thumbs` sub-directory. The files have the slightly unconventional suffix `.GIF` (*i.e.* capitalized) in order to avoid confusion with the full-size images. Conventionally, thumbnails are a factor of four smaller than their parent image.

Various image manipulation utilities could be used to generate thumbnails by hand, but it is possible to get `hiprgen.pl` to generate the thumbnail automatically from the full-size image. This requires that you first install the special HIPR utility `himg2thm` and the PBMplus library. Then you must edit the `hiprgen.pl` file, and near the top of the file, you should change the line:

```
$AUTO_THUMBNAIL = 0;
```

to:

```
$AUTO_THUMBNAIL = 1;
```

From this point on, if, during the course of processing a HIPRscript file, a `\imageref` tag is encountered for which there is no corresponding thumbnail file, then one will be generated automatically. If you subsequently change the full-size image and you want to force `hiprgen.pl` to regenerate a new matching thumbnail, then simply delete the old thumbnail file and then re-run `hiprgen.pl` on any source file containing a reference to that image.

When it comes to producing smaller versions of images, not all images are equal. Some images contain fine detail which the standard reduction process, which uses the PBMplus utility `pnmscale`, destroys. So, for these images, an alternative reduction utility `ppmsample` has been provided. Whenever `hiprgen.pl` has to produce a thumbnail, it looks in a file called `FINE.txt` which contains a list of images that require special treatment. If the image to be reduced is in that file, then the HIPR utility `himg2thm` is called with an additional flag (`-f`) which tells it to use `ppmsample` if it is available. Details on how to compile and install `ppmsample` are provided in the `himg2thm` script.

Appendix D

Bibliography

General References

- I. Aleksander (ed.)** *Artificial Vision for Robots*, Chapman and Hall, 1983.
- N. Ahuja and B. Schachter** *Pattern Models*, John Wiley & Sons, 1983.
- T. Avery and G. Berlin** *Fundamentals of Remote Sensing and Airphoto Interpretation*, Maxwell Macmillan International, 1985.
- D. Ballard and C. Brown** *Computer Vision*, Prentice-Hall, 1982.
- B. Batchelor, D. Hill and D. Hodgson** *Automated Visual Inspection*, IFS (Publications) Ltd, 1985.
- R. Bates and M. McDonnell** *Image Restoration and Reconstruction*, Oxford University Press, 1986.
- R. Bellman** *Introduction to Matrix Analysis, 2nd edition*, McGraw-Hill, 1970.
- R. Blahut** *Fast Algorithms for Digital Signal Processing*, Addison-Wesley Publishing Company, 1985.
- R. Bolles, H. Baker and M. Hannah** *The JISCT Stereo Evaluation*, ARPA Image Understanding Workshop Proceedings, 1993.
- R. Boyle and R. Thomas** *Computer Vision: A First Course*, Blackwell Scientific Publications, 1988.
- M. Brady (ed.)** *Computer Vision*, North-Holland, 1981.
- M. Brady and R. Paul (eds)** *Robotics Research: The First International Symposium*, MIT Press, 1984.
- D. Braggins and J. Hollingham** *The Machine Vision Sourcebook*, IFS (Publications) Ltd, 1986.
- A. Browne and L. Norton-Wayne** *Vision and Information Processing for Automation*, Plenum Press, 1986.
- V. Bruce and P. Green** *Visual Perception: Physiology, Psychology and Ecology, 2nd edn.*, Lawrence Erlbaum Associates, 1990.
- J. Canny** *A Computational Approach to Edge Detection*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 8, No. 6, Nov. 1986.
- K. Castleman** *Digital Image Processing*, Prentice-Hall, 1979.
- S. Chang** *Principles of Pictorial Information Design*, Prentice-Hall, 1986.
- E. Charniak and D. McDermott** *Introduction to Artificial Intelligence*, Addison-Wesley Publishing Company, 1985.

-
- R. Clark** *Transform Coding of Images*, Academic Press, 1985.
- F. Cohen** *The Handbook of Artificial Intelligence, Vol. 2*, Pitman, 1982.
- T. Crimmins** *The Geometric filter for Speckle Reduction*, Applied Optics, Vol. 24, No. 10, 15 May 1985.
- E. Davies** *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990.
- G. Dodd and L. Rossol (eds)** *Computer Vision and Sensor-Based Robotics*, Plenum Press, 1979.
- E. Dougherty** *An Introduction to Morphological Image Processing*, SPIE Press, 1992.
- R. Duda and P. Hart** *Pattern Classification and Scene Analysis*, John Wiley & Sons, 1978.
- M. Ekstrom (ed.)** *Digital Image Processing Techniques*, Academic Press, 1984.
- D. Elliott and K. Rao** *Fast Transforms: Algorithms and Applications*, Academic Press, 1983.
- O. Faugeras** *Fundamentals In Computer Vision - An Advanced Course*, Cambridge University Press, 1983.
- K. Fu, R. Gonzalez and C. Lee** *Robotics: Control, Seeing, Vision and Intelligence*, McGraw-Hill, 1987.
- J. Frisby** *Seeing*, Oxford University Press, 1979.
- R. Gonzalez and P. Wintz** *Digital Image Processing, 2nd edition*, Addison-Wesley Publishing Company, 1987.
- R. Gonzalez and R. Woods** *Digital Image Processing*, Addison-Wesley Publishing Company, 1992.
- W. Green** *Digital Image Processing - A Systems Approach*, Van Nostrand Reinhold Co., 1983.
- R. Hamming** *Digital Filters*, Prentice-Hall, 1983.
- R. Haralick and L. Shapiro** *Computer and Robot Vision*, Addison-Wesley Publishing Company, 1992.
- E. Hildreth** *The Measurement of Visual Motion*, MIT Press, 1983.
- B. Horn** *Robot Vision*, MIT Press, 1986.
- B. Horn and M. Brooks (eds)** *Shape from Shading*, MIT Press, 1989.
- T. Huang** *Image Sequence Analysis*, Springer-Verlag, 1981.
- T. Huang (ed.)** *Advances in Computer Vision and Image Processing, Vol. 2*, JAI Press, 1986.
- IEEE Trans. Circuits and Syst. Special Issue on Digital Filtering and Image Processing, Vol. CAS-2, 1975.**
- IEEE Trans. Inform. Theory, Special Issue on Quantization, Vol. IT-28, 1982.**
- IEEE Trans. Pattern Analysis and Machine Intelligence Special Issue on Industrial Machine Vision and Computer Vision Technology, Vol. 10, 1988.**
- A. Jain** *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986.
- L. Kanal and A. Rosenfeld (eds)** *Progress in Pattern Recognition, Vol. 2*, North Holland, 1985.
- J. Kittler and M. Duff (eds)** *Image Processing System Architectures*, Research Studies Press Ltd, 1985.
- M. Levine** *Vision in Man and Machine*, McGraw-Hill, 1985.
- A. Marion** *An Introduction to Image Processing*, Chapman and Hall, 1991.
- D. Marr** *Vision*, Freeman, 1982.

-
- L. McClelland and D. Rumelhart**, *Parallel Distributed Processing: Explorations in the Microstructures of Cognition, Vol. 2*, MIT Press, 1986.
- R. Nevatia** *Machine Perception*, Prentice-Hall, 1982.
- W. Niblack** *An Introduction to Image Processing*, Strandberg, 1985.
- H. Norton** *Sensor and Analyzer Handbook*, Prentice-Hall, 1982.
- Y. Pao** *Adaptive Pattern Recognition and Neural Networks*, Addison-Wesley Publishing Company, 1989.
- A. Pugh (ed.)** *Robot Vision*, IFS (Publications) Ltd, Springer-Verlag, 1983.
- W. Pratt** *Digital Image Processing, 2nd edition*, John Wiley & Sons, 1991.
- Proc. IEEE Special Issue on Digital Picture Processing**, Vol. 60, 1972.
- L. Roberts** *Machine Perception of Three-dimensional Solids*, Optical and Electro-optical Information Processing, MIT Press 1965.
- A. Rosenfeld and A. Kak** *Digital Picture Processing, Vols 1 and 2*, Academic Press, 1982.
- A. Rosenfeld and J.L. Pfaltz** *Distance Functions in Digital Pictures*, Pattern Recognition, Vol. 1, 1968.
- D. Rumelhart and J. McClelland** *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, MIT Press, 1986.
- J. Serra** *Images Analysis and Mathematical Morphology*, Academic Press, 1982.
- R. Schalkoff** *Digital Image Processing and Computer Vision*, John Wiley & Sons, 1989.
- H. Stark** *Image Recovery: Theory and Application*, Academic Press, 1987.
- J. Stoffel (ed.)** *Graphical and Binary Image Processing and Applications*, Artech House, 1982.
- R. Tsai** *A Versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-Shelf TV Cameras and Lenses*, IEEE Journal of Robotics and Automation, 1987.
- S. Ullman and W. Richards (eds)** *Image Understanding 1984*, Ablex Publishing Co, 1984.
- D. Vernon** *Machine Vision*, Prentice-Hall, 1991.
- B. Widrow and S. Stearns** *Adaptive Signal Processing*, Prentice-Hall, 1985.
- A. Zisserman** *Notes on Geometric and Invariance in Vision*, British Machine Vision Association and Society for Pattern Recognition, 1992, Chap. 2.

Local References

Additional useful references may be added here by the person who installed HIPR on your system.

Appendix E

Acknowledgements

The creation of HIPR was a team effort which would not have been possible without the advice, contributions and criticisms of many people.

In particular, we would like to thank the following for advice on what should be included in HIPR: Dr. G. Briarty (Nottingham University), Dr. C. Duncan (Edinburgh University) and Dr. A. Etemadi (Imperial College).

We would like to thank the staff and students of Edinburgh's Meteorology (Dr. C. Duncan), Physics (Dr. W. Hossack) and Artificial Intelligence for being guinea pigs with HIPR as it was being developed.

Prof. M. Landy of New York University provided the information about the HIPS operators.

Many of the input images were acquired specifically for HIPR, however, we would also like to thank those individuals and organizations who, without endorsing the material in the HIPR package, donated images from their own research or private photo collections. All images included in HIPR can be freely used for educational or research purposes, but further distribution is prohibited. Any publication of a contributed image must include the acknowledgement enclosed near its position in the image subject catalog.

The image contributors include: Ms. R. Aguilar-Chongtay (wom3, cel4), Dr. N. Wyatt and Dr. J. B. Lloyd (ugr1), Mr. S. Beard (egg1), Calibrated Imaging Laboratory at Carnegie Mellon University (st00 - st11, sponsored, in part, by DARPA, NSF and NASA), Dr. A. Dil (puf1, crs1, cot1, goo1, dov1, pdc1, bri1, cas1, lao1, win1), Ms. S. Flemming (hse2, hse3, hse4), Dr. M. Foster (mri1, mri2), Georgia Institute of Technology on-line image database (cam1, wom2, trk1, brg1, cln1, urb1, wom1), Mr. D. Howie (leg1), Mr. X. Huang (bab4, wom5, wom6, txt3), Dr. N. Karssemeijer and University Hospital Nijmegen. (stl1, stl2, slt1, slt2), Dr. B. Lotto (cel1, cel2, cla3, clb3, clc3, cel5, cel6, cel7, axn1, axn2), Mr. A. MacLean (pcb1, pcb2), EUMETSAT (air1, avs1, bir1, bvs1, eir1, evs1, sir1, sv1, uir1, uvs1), Mr. A. Miller (AudioVisual Services for loan of the tst2 testcard), NASA & (specifically) Dryden Research Aircraft Photo Archive (lun1, shu1, shu2, shu3, fei1, arp1, arp2, ctr1, mof1), Pilot European Image Processing Archive (PEIPA) (rot1), Dr. L. Quam and SRI International (yos2-16), Royal Greenwich Observatory, the Starlink Project and Rutherford Appleton Laboratory (str1, str2, str3), Ms. V. Temperton (rck1, rck2, rck3), Dr. P. Thanisch and A. Hume (mam1, mam2), Mr. D. Walker (sff1, sfn1, sfc1), Mr. M. Westhead (rob1).

A number of the images were copied (with many thanks) from C. A. Glasbey and G. W. Horgan, Image Analysis for the Biological Sciences (John Wiley, 1995). We also thank the original contributors of those images: Dr. J. Darbyshire, Macaulay Land Use Research Institute, Aberdeen, Scotland (soi1), Dr. M. A. Foster, Biomedical Physics & Bioengineering, University of Aberdeen (mri1, mri2), Dr. C. Maltin, Rowett Research Institute, Aberdeen, Scotland (mus1), Dr. N. Martin, Scottish Agricultural Co, Ayr, Scotland (alg1), Dr. K. Ritz, Scottish Crop Research Institute, Dundee, Scotland (fun1), Dr. G. Simm, Genetics and Behavioural Sciences Dept, Scottish Agricultural College, Edinburgh, Scotland (usd1, xra1), Dr. N. Strachan, Torry Research Station,

Aberdeen, Scotland (fsh2), Dr. F. Wright, Biomathematics and Statistics Scotland, University of Edinburgh, Edinburgh, Scotland (dnal).

Thanks go to many helpful people in the Department of Artificial Intelligence for technical assistance with the hypertext and image productions, including N. Brown, T. Colles, A. Fitzgibbon and M. Westhead.

Funding for the development of this project was provided by the United Kingdom's Joint Information Systems Committee, under their New Technology Initiative, Project 194, and by the University of Edinburgh. We greatly appreciate their support and advice through Dr. T. Franklin (Manchester University).

Appendix F

The HIPR Copyright

HYPERMEDIA IMAGE PROCESSING REFERENCE

Copyright ©1996 Robert Fisher, Simon Perkins, Ashley Walker and Erik Wolfart

Published by:

John Wiley & Sons Ltd
Baffins Lane, Chichester
West Sussex, PO19 1UD, England
National 01243 779777
International (+44) 1234 779777

Visit the Wiley Home Page at:

<http://www.wiley.com>

or

<http://www.wiley.co.uk>

Ordering and site licence information:

US orders: kball@wiley.com

UK and rest of the world: lglen@wiley.co.uk

Any other queries: hipr@wiley.co.uk

All rights reserved.

User License. (p.274)

All material contained within Hypermedia Image Processing Reference (HIPR) is protected by copyright, whether or not a copyright notice appears on the particular screen where the material is displayed. No part of the material may be reproduced or transmitted in any form or by any means, or stored in a computer for retrieval purposes or otherwise, without written permission from Wiley, unless this is expressly permitted in a copyright notice or usage statement accompanying the materials or in the user license (p.274). Requests for permission to reproduce or reuse material for any purpose should be addressed to the Copyright and Licensing Department, John Wiley & Sons Ltd, Baffins Lane, Chichester, West Sussex, PO19 1UD, UK; Telephone +44 1243 770429; Email info-assets@wiley.co.uk

Neither the authors nor John Wiley & Sons Ltd accept any responsibility or liability for loss or damage occasioned to any person or property through using the materials, instructions, methods

or ideas contained herein, or acting or refraining from acting as a result of such use. The authors and Publisher expressly disclaim all implied warranties, including merchantability or fitness for any particular purpose. There will be no duty on the authors or Publisher to correct any errors or defects in the software.

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons are aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

Other Wiley Editorial Offices:

- John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, USA
- Jacaranda Wiley Ltd, 33 Park Road, Milton, Queensland 4064, Australia
- John Wiley & Sons (Canada) Ltd, 22 Worcester Road Rexdale, Ontario, M9W 1L1, Canada
- John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop Jin Xing Distripark, Singapore 0512

Appendix G

User License

Although HIPR can be run directly from the installation CD, the package is intended for local area network use. If you wish to use HIPR in a networked configuration, a site license must be purchased (p.314) from the Publisher (p.272). The license restricts the range of access to a specified site. It is prohibited to make HIPR available over the WWW outside the licensed site.

A site is defined as a single academic department of a university, college or other school of learning, where all taught courses fall within a single subject discipline. A license for networking the product by multiple academic departments or by a commercial organization is available on application. There is no limit to the number of users at a licensed site or to the term of the site license.

What the license entitles you to do:

Once you have paid your license fee and signed the license agreement, you are entitled to make the product available electronically to all the people you have paid for. They can use it freely, including making multiple printouts of the complete text, to a maximum of one copy per student for internal classroom use, or downloading extracts temporarily to hard disk for research or study. They can use all images for research or educational purposes but will need to make specific acknowledgements; refer to the online subject image library for details of acknowledgements required. They are entitled to create customized versions of the product for use within the licensed site. One copy of the original can be made for backup purposes.

What you can't do:

You must not allow electronic access to the product or to customized versions of the product, or send printed copies, to anyone other than those for whom it is licensed. You may not resell any part of the product, or use extracts to create a further product for sale, without written agreement from Wiley. It is up to you to make sure that the product is used properly.

The following are the full terms and conditions which apply when HIPR is used as a networked product.

SITE LICENSE TERMS AND CONDITIONS

Definitions

- 1 (a) 'The Site' means and refers to, inclusively, all offices and facilities of the Licensee at which the Product is authorized for use by Users, the addresses of which are listed in the Order Form.
- 1 (b) 'The Product' means and refers to the electronic publication or service set forth in the Order Form.
- 1 (c) 'User' means and refers to all professors, teaching staff and enrolled students on the course(s)/within the academic departments listed in the Order Form OR all paid employees at the Site.

-
- 1 (d) 'Fee' means and refers to the amount to be paid by the Licensee for the license granted by this agreement, as set forth in the Order Form. This Agreement shall not be binding upon Wiley unless Wiley has received timely payment of the Fee.

Commencement and License

- 2 (a) This Agreement commences upon receipt by Wiley of the signed and unaltered Agreement ('the Commencement Date').
- 2 (b) This is a license agreement ('the Agreement') for the use of the product by the Licensee, and not an agreement for sale.
- 2 (c) Wiley licenses the Licensee on a non-exclusive and non-transferable basis to use the Product on condition that the Licensee complies with the terms of the Agreement. The Licensee acknowledges that it is only permitted to use the Product in accordance with the terms of the Agreement.
- 2 (d) The person signing this agreement for and on behalf of the Licensee represents that he or she is authorized to do so, and to bind the Licensee thereby.

Installation and Use

- 3 (a) Wiley shall supply to the Licensee the number of copies of the Product specified in the Order Form.
- 3 (b) Delivery shall be to the address specified on the Order Form.
- 3 (c) The Licensee shall be responsible for installing the Product and for the effectiveness of such installation..
- 3 (d) The Licensee may store the Product electronically and may arrange for Users to have access to the Product at computer terminals at the Site only.

Payment

- 4 (a) The Licensee agrees to pay Wiley the amount set out in the Order Form('the Fee').
- 4 (b) Fees are exclusive of VAT or other taxes, which are the responsibility of the Licensee.
- 4 (c) Wiley shall have the right as a non-exclusive remedy to withhold delivery of the Product until all overdue amounts payable by the Licensee under this Agreement have been paid.

Permitted Activities

- 5 (a) The Licensee shall be entitled:
- (i) to use the Product at the Site
 - (ii) to use the Product for its own internal purposes
 - (iii) to make a maximum of one paper copy per student for internal classroom use
 - (iv) to make one copy of the Product for backup/archival/disaster recovery purposes
 - (v) to modify and customize the Product and to make one copy of the modified product for backup/archival/disaster recovery purposes, provided that such copy is clearly identified and distinguished from the original product.
- 5 (b) Licensee may transmit the Product electronically to Users at any computer terminal within the Site.

-
- 5 (c) Each User shall have the right to access, search and view the Product at any computer terminal at the Site.
- 5 (d) The Licensee acknowledges that its rights to use the Product are strictly as set out in this Agreement, and all other uses (whether expressly mentioned in Clause 6 below or not) are prohibited.

Prohibited Activities

- 6 (a) The Licensee may not transmit the Product or extracts therefrom electronically, in print or by any other means, or allow copies of the Product to be otherwise stored or accessed by any person or at any computer terminal, whether or not owned and operated by the User, anywhere beyond the premises of the authorized Site.
- 6 (b) The Licensee may not connect the Product to a system network which permits use anywhere other than at the addresses designated as the Site in the Order Form.
- 6 (c) The Licensee may not exploit any part of the Product commercially, including commercial training courses or seminars.
- 6 (d) The Licensee may not sell, rent, loan (free or for payment), hire out or sublicense the Product or any derivative Work to any third party, without the express written consent of Wiley.
- 6 (e) The Licensee may not undertake any activity which may impede Wiley's ability or opportunity to market the Product.
- 6 (f) The Licensee may not provide services to third parties using the Product, whether by way of trade or otherwise.
- 6 (g) The Licensee may not use the Product to make any derivative work, product or service save as expressly provided for in this Agreement.
- 6 (h) The Licensee may not alter, amend, modify or change the Product in any way, whether for purposes of error-correction or otherwise, other than as explicitly permitted by the nature of the Product.
- 6 (i) The Licensee may not make the contents of the Product available on a computer bulletin board without the express written permission of Wiley.

General Responsibilities of the Licensee

- 7 (a) The Licensee will take all reasonable steps to ensure that the Product is used in accordance with the terms and conditions of this Agreement, and shall advise all Users of the permitted use, restrictions and provisions set out herein.
- 7 (b) The Licensee acknowledges that damages may not be a sufficient remedy for Wiley in the event of breach of this Agreement by the Licensee, and that an injunction may be appropriate.
- 7 (c) The Licensee agrees to indemnify Wiley against any and all other claims, damages, losses and expenses (including reasonable legal fees) arising from
- (i) any misuse of the Product by the Licensee or Users
 - (ii) any misuse by any third party,
- where such misuse occurs, in either (i) or (ii), as a result of breach by the Licensee of this Agreement.
- (iii) any breach by Licensee or Users of any provisions of this Agreement
 - (iv) any of Licensee's or Users' activities relating to this agreement.

-
- 7 (d) The Licensee undertakes to keep the Product safe, and to use its best endeavors to ensure that the Product does not fall into the hands of third parties, whether as a result of theft or otherwise.
- 7 (e) Wiley may, at its option, institute or defend any action arising out of the aforesaid clauses with a legal advisor of its own choice.

Warranties and Indemnities

- 8 (a) Wiley warrants that it has the authority to enter into this Agreement, and that it has secured all necessary rights and permissions necessary to enable the Licensee to use the Product in accordance with this Agreement.
- 8 (b) Wiley warrants that the CD-rom/Diskettes as supplied on the Commencement shall be free of defects in materials and workmanship, and undertakes to replace any defective CD-rom/diskette upon notice of such defect, or within thirty (30) days of such notice being received, provided such notice is received within ninety (90) days of supply. As an alternative to replacement/remedy, Wiley agrees to refund the Fee if the Licensee so requests, provided that the Licensee returns the CD-rom/Diskettes to Wiley. The provisions of this clause do not apply where the defect results from an accident or from misuse by the Licensee.
- 8 (c) Clause 8 (b) sets out the sole and exclusive remedy of the Licensee in relation to defects in the Product.
- 8 (d) Wiley and the Licensee acknowledge that Wiley supplies the Product on an 'as is' basis. Wiley gives no warranties
- (i) that the Product satisfies the individual requirements of the Licensee
 - (ii) that the Product is otherwise fit for the Licensee's purpose
 - (iii) that the Product is accurate or complete or free of errors or omissions or
 - (iv) that the Product is compatible with the Licensee's hardware equipment and software operating environment.
- 8 (e) Wiley hereby disclaims all other warranties and conditions, express or implied, which are not stated above.
- 8 (f) Nothing in this clause limits Wiley's liability to the Licensee in the event of death or personal injury resulting from Wiley's negligence.
- 8 (g) Subject to subclause 8 (f) above, Wiley's liability under this Agreement shall be limited to the Fee.
- 8 (h) The above warranties and indemnities shall survive the termination of this Agreement.

Intellectual Property Rights

- 9 (a) Nothing in this Agreement affects the ownership of copyright or other intellectual property rights in the Product.
- 9 (b) The Licensee agrees to display the authors' copyright notice as it appears in the Product, and not to remove such copyright notice from any part of the Product.
- 9 (c) Other than as provided in Clause 9 (b) above, the Licensee shall not use (including, without limitation, reprint or reproduce in any way) the Wiley logo or any trademark in connection with any permitted use of the Product.

Termination

-
- 10 (a)** Wiley shall have the right to terminate this Agreement, at its own discretion, if:
- (i) the Licensee is in material breach of this Agreement and fails to remedy such breach (where capable of remedy) within fourteen (14) days of a written notice from Wiley requiring it to do so
 - (ii) the Licensee ceases to operate in business/education
 - (iii) upon thirty (30) days' notice to Licensee if Wiley discontinues marketing the Product. In such case, Wiley may make a pro rata refund if the product is less than five (5) years old.
 - (iv) automatically, if this license is not returned to Wiley signed within sixty (60) days of receipt by Licensee.
- 10 (b)** The Licensee shall have the right to terminate this Agreement for any reason upon ninety (90) days written notice. The Licensee shall not be entitled to any refund for payments made under this Agreement prior to termination under this sub-clause.
- 10 (c)** This Agreement shall be terminated automatically, on return of the product and any print copies made of it in undamaged condition, within thirty (30) days of receipt by the Licensee. In such case, the Fee will be refunded in full.
- 10 (d)** Termination by either of the parties is without prejudice to any other rights or remedies under the general law to which they may be entitled, or which survive such termination.
- 10 (e)** Upon Termination of this Agreement the Licensee must:
- (i) deinstall the Product from all computers and sign a written undertaking to that effect
 - (ii) destroy all back-up copies of the Product and any derivative Work in electronic or print format
 - (iii) return the CD-rom/Diskettes to Wiley.
- 10 (f)** No Fee shall be refunded on Termination, other than under the provisions of Clause 10 (a) (iii) and 10 (c) above. To qualify for a refund of the Fee, the Licensee must deinstall the Product from all computers and provide a written undertaking to that effect.

Notice

- 11 (a)** Any Notice required or given under this Agreement by any party hereto shall be provided in writing to the other party at the address set out at the end of this Agreement or at other such address as such parties shall provide.
- 11 (b)** All Notices to Wiley or enquiries concerning this Agreement shall be sent to John Wiley & Sons Ltd, Baffins Lane, Chichester, West Sussex, PO19 1UD, marked for the attention of the Director of Copyright and Licensing.
- 11 (c)** Any Notices to the Licensee or enquiries concerning this Agreement shall be sent to the Licensee at the address and contact name identified in the Order Form.

General

- 12 (a)** This Agreement embodies the entire agreement between Wiley and the Licensee concerning the Product, and supersedes any and all prior understanding and agreements, oral or written, relating to the subject matter hereof.
- 12 (b)** Wiley may assign this Agreement to its successors, related companies or assignees. This Agreement may not be assigned by the Licensee without Wiley's written consent.
- 12 (c)** Any amendments hereto must be in writing and signed by both parties.

-
- 12 (d)** Wiley hereby agrees to comply fully with all relevant export laws and regulations of the United Kingdom to ensure that the Product is not exported, directly or indirectly, in violation of English law.
- 12 (e)** The parties accept no responsibility for breaches of this Agreement which occur as a result of circumstances beyond their control.
- 12 (f)** Any failure or delay by either party to exercise or enforce any right conferred by this Agreement shall not be deemed to be a waiver of such right.
- 12 (g)** If any provision of this Agreement is found to be invalid or unenforceable by a court of law of competent jurisdiction, such a finding shall not affect the other provisions of this Agreement and all provisions of this Agreement unaffected by such a finding shall remain in full force and effect.
- 12 (h)** This Agreement is construed and interpreted under UK law and the parties hereby agree that any dispute arising under this Agreement shall be subject to the jurisdiction of the English courts.

Appendix H

About the Authors

HIPR was created in the Department of Artificial Intelligence at the University of Edinburgh by the following people:

Dr. Robert Fisher has been a tenured lecturer in the Department of Artificial Intelligence since 1984 and a Senior Lecturer since 1992. He has been researching 3D scene understanding for over 10 years, and has worked on model based object recognition using range data, feature extraction in range data, range data sensors, scanner data and parallel vision algorithms. Recently, he has been investigating model-matching algorithms, automatic construction of geometric models from shown examples and surface inspection using range data. He has been the principal investigator of over 1 million pounds of externally funded research projects, and currently has an EPSRC/ACME range data interpretation project, an EPSRC robotics grasping project, a JISC educational materials development grant and two EC/HCM visiting fellow research grants. He has published two books and over 70 conference and journal papers.

Mr. Simon Perkins is currently a PhD student in the Department of Artificial Intelligence, looking at ways of integrating engineering-based and evolutionary-based methods of robot design to produce mobile robots capable of performing complex visual behaviors. Prior to this he worked full-time on the HIPR project.

Ms. Ashley Walker is working on a PhD thesis in Robotics at the University of Edinburgh. Her work involves modeling of SONAR sensory phenomena and aims to develop signal processing algorithms for use in the construction of acoustic maps for mobile robots.

Mr. Erik Wolfart: After completing his first degree in Electrical Engineering in Germany, he went to Edinburgh in 1993 to do the MSc course in the Department of Artificial Intelligence. He is now based at UK Robotics Ltd and is developing vision systems for environmental modeling and robot control.

Appendix I

The Image Library

I.1 Introduction

The HIPR package contains a large number of images which can be used as a general purpose image library for image processing experiments. This section catalogues the images in the library and describes each image briefly. The image file naming convention is described in the section on filename conventions (p.25).

The images may be divided into two main categories. *Raw images* are images that have had almost no processing done to them since they were captured. They are the kinds of input image that standard image sensors — video cameras, scanners, *etc.* — produce. For the HIPR image library, such images were obtained by a variety of methods. Some were scanned in from photographs, some were captured using video cameras, and some were obtained from standard image libraries on the Internet. The second category of image consists of *processed images*. These are simply raw images to which at least one stage of ‘image processing’ has been applied.

These images are catalogued in two different ways, firstly by subject and type, and secondly as a straight alphabetical index of image filenames. Note that the alphabetical listing is **ONLY** available in the hypermedia version of HIPR.

The subject/type catalogue is useful if you know the sort of raw image you are looking for and want to see if HIPR has anything similar in its library. The top level of this catalogue is a list of image types and categories. Underneath these are listed the different raw images that belong to each category. Finally, underneath each of the raw image headings is listed the various processed images that have been produced, starting with that raw image. A brief description of the processing required to produce each image is also included.

The alphabetical catalogue is more simple, and consists of a straight alphabetical list of the filenames of all the images in the HIPR library. Each filename is in fact a hyperlink to the relevant explanatory section in the subject/type catalogue (this is why it is only available in the hypermedia version). It is useful if you know the filename of a particular image, and want to find out where it comes from and how it was produced.

Additional Sources of Images

If you have an Internet connection, then it is possible to obtain further images from public domain databases on the Internet. A good starting point for looking for images is the *Computer Vision Home Page* which at the time of writing can be found at:

<http://www.cs.cmu.edu/~cil/vision.html>

Alternatively, collections of images on CD-ROM can be purchased for a variety of purposes from appropriate suppliers. Computer magazines are generally a good source of manufacturers' addresses.

I.2 Image Catalogue by Subject/Type

This is the Image Catalogue by Subject/Type. Here the large (*i.e.* over 700 items) list of Image Library items has been divided into a collection of indexed smaller item sections based on the subject matter depicted within each image. Each of the categories listed provides a link to a *second* layer of the catalogue, which contains a description of every *raw* image of that subject type. (Note that where images fall into more than one of the categories below, they will appear in each.) For every raw image in the second layer which has been processed as part of a worksheet example, a link to the *third* (and final) catalogue layer exists. This third layer displays the *derived* images with links to the worksheets which describe the operation(s) performed on them in the image processing examples.

Image Subject/Type

Architecture

The *Architecture* Image Catalogue section contains images of buildings (*e.g.* office buildings, homes, castles, churches, *etc.*) and other structures (*e.g.* bridges, statues, stained glass windows, *etc.*) of architectural interest.

b1d1 412×663 grayscale of Scott Monument, Edinburgh, Scotland.

b1d1pin1 Result of scaling (p.90) **b1d1** using pixel interpolation.

b1d1psh1 Result of scaling (p.90) **b1d1** using pixel replacement.

b1d1hst1 Intensity histogram (p.105) of **b1d1**.

b1d1cuh1 Cumulative intensity histogram (p.105) of **b1d1**.

b1d1heq1 Result of histogram equalizing (p.78) **b1d1**.

b1d1crp1 Result of cropping **b1d1**.

b1d1cuh2 Cumulative intensity histogram (p.105) of **b1d1crp1**.

b1d1heq2 Result of histogram equalizing (p.78) **b1d1crp1**.

b1d1or1 Result of ORing (p.58) **b1d1crp1** with its histogram (p.105).

b1d1xor1 Result of XORing (p.60) **b1d1crp1** with its histogram (p.105).

brg1 256×256 grayscale of wooden bridge over a small river.

brg1blu1 Non-ideal optical transfer function (OTF) image simulated by multiplying (p.48) the Fourier Transform (p.209) of **brg1** with the Fourier Transform of a Gaussian image with standard deviation = 5.

brg1dec1 Result obtained by dividing the Fourier Transform (p.209) of **brg1blu1** by the Fourier Transform of the Gaussian blurring kernel.

brg1blu2 Result of adding 0.1% salt and pepper noise (p.221) to **brg1blu1**.

brg1dec2 Result obtained by dividing the Fourier Transform (p.209) of **brg1blu2** by the Fourier Transform of the Gaussian blurring kernel.

brg1dec3 Result obtained by dividing the Fourier Transform (p.209) of **brg1blu2** by the Fourier Transform of the Gaussian blurring kernel and ignoring values of the OTC which fall below a threshold of 3.

brg1blu3 Non-ideal optical transfer function (OTF) image simulated by convolving **brg1** with a spatial domain Gaussian image with standard deviation = 5.

brg1dec4 Result obtained by dividing the Fourier Transform (p.209) of **brg1blu3** by the Fourier Transform of the Gaussian blurring kernel.

brg1dec5 Result obtained by dividing the Fourier Transform (p.209) of **brg1blu3** by the Fourier Transform of the Gaussian blurring kernel and ignoring values of the OTC which fall below a threshold of 5.

Any use of this image must include the acknowledgement: 'We thank Georgia Institute of Technology for the use of these images acquired from their on-line image database.'

brg2 583×350 8-bit color of Forth Road Bridge, Edinburgh, Scotland.

brg3 Graylevel equivalent of **brg2**.

brg3can1 Result of applying the Canny edge detector (p.192) to **brg3**.

brg3lda1 Result of applying line detection (p.202) to **brg3**.

brg3add2 Result of smoothing before applying line detection (p.202) to **brg3**.

bri1 695×488 8-bit color of Brighton Pavilion, Brighton, England.

bri2 Grayscale equivalent of **bri1**.

Any use of this image must include the acknowledgement: 'We thank Dr. A. Dil of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

cam1 256×256 grayscale of man with camera.

cam1pin1 Result of zooming (p.90) **cam1** using pixel interpolation.

cam1exp1 Result of zooming (p.90) **cam1** using pixel replication.

Any use of this image must include the acknowledgement: 'We thank Georgia Institute of Technology for the use of these images acquired from their on-line image database.'

cas1 346×552 8-bit color of Edinburgh Castle, Scotland. Any use of this image must include the acknowledgement: 'We thank Dr. A. Dil of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

crs1 388×675 8-bit color of Celtic cross at Iona Abbey, Scotland. Any use of this image must include the acknowledgement: 'We thank Dr. A. Dil of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

lao1 395×583 8-bit color of Laon Cathedral, France. Any use of this image must include the acknowledgement: 'We thank Dr. A. Dil of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

lib1 293×386 8-bit color of Statue of Liberty, Ellis Island, U.S.A.

pdcl 439×391 8-bit color of Palais de Chaillot, France. Any use of this image must include the acknowledgement: 'We thank Dr. A. Dil of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

sfc1 599×360 8-bit color close-up of city center financial district. Any use of this image must include the acknowledgement: 'We thank Mr. D. Walker for the use of this image.'

sff1 587×359 8-bit color of San Francisco, Calif, U.S.A. on a foggy day.

sff1sca1 Cropped grayscale version of the original.

sff1can1 Result of edge detecting **sff1sca1** using the Canny operator (p.192).

sff1hou1 Hough transform (p.214) space of **sff1can1**.

sff1hou2 Reconstructed edges with peak value bigger than 70% of maximum peak from the Hough Transform in **sff1hou1** superimposed on **sff1can1**.

sff1hou3 Reconstructed edges with peak value bigger than 50% of maximum peak from the Hough Transform in **sff1hou1** superimposed on **sff1can1**.

Any use of this image must include the acknowledgement: 'We thank Mr. D. Walker for the use of this image.'

sfn1 589×392 8-bit color of San Francisco, Calif, U.S.A. on a clear night. Any use of this image must include the acknowledgement: 'We thank Mr. D. Walker for the use of this image.'

sta1 409×598 8-bit color of small statue.

sta2 Graylevel equivalent of **sta1**.

sta2mea1 Result of mean filtering (p.150) **sta1** using a 3×3 kernel.

sta2mea2 Result of mean filtering (p.150) **sta1** using a 7×7 kernel.

sta2mea3 Result of applying 3 passes of a 3×3 mean filter (p.150) to **sta1**.

sta2noi1 Result of adding 5% salt and pepper noise (p.221) to **sta2**.

sta2csm1 Result of applying conservative smoothing (p.161) to **sta2noi1**.

sta2med1 Result of median filtering (p.153) **sta2noi1** with a 3×3 kernel.

sta2med2 Result of median filtering (p.153) **sta2noi1** with a 7×7 kernel.

sta2med3 Result of applying 3 passes of a 3×3 median filtering (p.153) to **sta2noi1**.

sta2crm1 Result of applying 10 iterations of Crimmins Speckle Removal Algorithm (p.164) to **sta2noi1**.

sta2noi2 Result of adding 1% salt and pepper noise (p.221) to **sta2**.

sta2gsm1 Result of Gaussian smoothing (p.156) **sta2noi2** with a standard deviation = 1 filter.

sta2gsm2 Result of Gaussian smoothing (p.156) **sta2noi2** with a standard deviation = 2 filter.

wal1 620×406 grayscale of period office building.

wal2 256×256 grayscale of modern office building.

win1 334×659 8-bit color of stained glass window in St. Margaret's Chapel, Edinburgh, Scotland. Any use of this image must include the acknowledgement: 'We thank Dr. A. Dil of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

Artificial

The *Artificial* Image Catalogue contains a collection of computer generated binary images. Most images contain a single (of very few) instance(s) of a geometric shape (constructed using, *e.g.* a paint program), however, there are some more sophisticated computer graphics (*e.g.* generated by modeling software).

art1 222×217 binary of circles and lines.

art1rot1 Result of rotating (p.93) **art1** by 180 degrees about a point ($x = 150, y = 150$).

art1rot2 Result of rotating (p.93) **art1** by 45 degrees about the image center.

art2 205×245 binary of criss-cross lines.

art2trn1 Result of translating (p.97) **art2** by 110 pixels in the x-direction and 120 pixels in the y-direction, and wrapping the result.

-
- art2sub1** Result of translating (p.97) **art2** by 1 pixel in the x-direction and y-direction, and then subtracting this from the original.
- art2trn2** Result of rotating (p.93) **art2** by 180 degrees about its center, displacing it by 1 pixel in the x and y-directions and then subtracting this result from the original.
- art2opn1** Result of opening (p.127) **art2** with a 3×9 structuring element (p.241).
- art2opn2** Result of opening (p.127) **art2** with a 9×3 structuring element (p.241).
- art2ldh1** Normalized result of horizontal line detecting (p.202) **art2**.
- art2crp1** Cropped and zoomed (p.90) version of **art2**.
- art2crp2** Cropped and zoomed (p.90) version of **art2ldh1**.
- art2sk11** Skeletonized (p.145) version of **art2**.
- art2ldh2** Normalized result of horizontal line detecting (p.202) **art2sk11**.
- art2ldh3** Result of thresholding (p.69) **art2ldh2**.
- art2ldh4** Result of applying the same thresh (p.69) to **art2ldh1**.
- art2crp3** Cropped and zoomed (p.90) version of **art2ldh1**.
- art2crp4** Cropped and zoomed (p.90) version of **art2ldh2**.
- art2ldv2** Normalized result of vertical line detecting (p.202) **art2sk11**.
- art2ldv1** Result of thresholding (p.69) **art2ldv2**.
- art2ldp2** Normalized result of line detecting (p.202) the oblique 45 degree lines in **art2sk11**.
- art2ldp1** Result of thresholding (p.69) **art2ldp2**.
- art2ldn2** Normalized result of line detecting (p.202) the oblique 135 degree lines in **art2sk11**.
- art2ldn1** Result of thresholding (p.69) **art2ldn2**.
- art2add1** Result of pixel adding (p.43) **art2ldh3**, **art2ldv1**, **art2ldp1** and **art2ldn1**.
- art3** 222×217 binary inversion of **art1**.
- art3opn1** Result of opening (p.127) **art3** with a disk of diameter 11.
- art3trn1** Result of translating (p.97) **art3** by 150 pixels in the x and y-directions.
- art4** 300×300 binary of circle filled with holes.
- art4neg1** Negative (p.63) of **art4**
- art4clo1** Result of closing (p.130) **art4** with a disk of diameter 22.
- art4ctr1** Result of translating (p.97) **art4** in order to center the image subject.
- art4trn1** Result of translating (p.97) **art4ctr1** by 150 pixels in the x and y directions.
- art4trn2** Result of translating (p.97) **art4ctr1** by 150 pixels in the x and y directions, and wrapping the result.
- art4trn3** Result of translating (p.97) **art4ctr1** by 35 pixels in the x direction and 37 pixels in the y direction.
- art7add1** Result of pixel adding (p.43) **art7trn1** to **art4trn3**.
- art4rot1** Result of rotating (p.93) **art4ctr1** by 180 degrees about the image center.
- art4ref1** Result of reflecting (p.95) **art4ctr1** about a vertical axis through the image center.
- art4dil1** Result of dilating (p.118) **art4**.
- art4dil2** Result of dilating (p.118) **art4neg1**.
- art5** 256×256 binary of rectangle.
- art5dst1** Result of Euclidean distance transforming (p.206) **art5** (and then pixel multiplying (p.48) by 5).

-
- art5sk11** Result of skeletonizing (p.145) **art5**.
art5mat1 The medial axis transform (p.145) of **art5**.
art5cha1 Result of image editing (p.233) **art5**.
art5ske2 Result of skeletonizing (p.145) **art5cha1** (using an algorithm which does not guarantee a connected skeleton).
art5ske3 Result of skeletonizing (p.145) **art5cha1**.
art5cha2 Result of image editing (p.233) **art5**.
art5dst3 The brightened (p.48) (by a factor of 6) distance transform (p.206) of **art5cha2**.
art5noi1 Result of adding pepper noise (p.221) to **art5**.
art5dst2 The brightened (p.48) (by a factor of 15) distance transform (p.206) of **art5noi1**.
art5ske5 Result of skeletonizing (p.145) **art5noi1**.
art5low1 Result of frequency filtering (p.167) **art5** with an ideal lowpass filter with a cut-off of $\frac{2}{3}$.
art5low3 Result of histogram equalizing (p.78) **art5low1**.
art5low2 Result of frequency filtering (p.167) **art5** with an ideal lowpass filter with a cut-off of $\frac{1}{3}$.
art5low4 Result of frequency filtering (p.167) **art5** with a Butterworth filter with a cut-off of $\frac{2}{3}$.
- art6** 256×256 binary of rounded, elongated object.
- art6inv1** The inverse (p.63) of **art6**.
art6dst1 Result of Euclidean distance transforming (p.206) **art6** (and then pixel multiplying (p.48) by 3).
art6sk11 Result of skeletonizing (p.145) **art6**.
art6mat1 The medial axis transform (p.145) of **art6**.
- art7** 256×256 binary of skewed doughnut shaped object.
- art7ref1** Result of reflecting (p.95) **art7** about a horizontal axis passing through the image center.
art7trn1 Result of subsampling (p.90) **art7** and then translating (p.97) the image subject to the upper left corner of the image.
art7add1 Result of pixel adding (p.43) **art7trn1** to **art4trn3**.
art7add2 Result of translating (p.97) **art7** and then pixel adding (p.43) back onto the original.
art7dst1 Result of distance transforming (p.206) **art7** (and then pixel multiplying (p.48) by 4).
art7sk11 Result of skeletonizing (p.145) **art7**.
art7mat1 The medial axis transform (p.145) of **art7**.
art7ham1 Result of hit-and-miss transforming (p.133) **art7sk11**, followed by dilation (p.118) with a cross-shaped element and, finally, ORing (p.58) with the original skeleton.
art7ham2 Result of hit-and-miss transforming (p.133) **art7sk11**, followed by dilation (p.118) with a square-shaped element and, finally, ORing (p.58) with the original skeleton.
- art8** 256×256 binary of multiple crosses.
- art8thk1** Result of thickening (p.142) **art8** to produce convex hull.
art8thk2 Result of thickening (p.142) **art8** to produce skeleton of background.

-
- art8thk3** Result of thickening (p.142) **art8** to produce a pruned version of **art8thk2**, *i.e.* the SKIZ.
- art8lab2** The labeled 8-bit color result of connected components labeling (p.114) **art8**.
- art8lab1** The grayscale equivalent of **art8lab2**.
- art8noi1** The result of adding salt noise (p.221) to **art8**.
- art8thk4** The SKIZ (p.142) of **art8noi1**.
- art9** 323×300 binary of propeller shaped object.
- art9ref1** Result of reflecting (p.95) **art9** about an oblique axis, *e.g.* one passing through 45 degrees.
- che1** 256×256 grayscale of chess board.
- che1noi1** Result of adding 2% salt and pepper noise (p.221) to **che1**.
- che1noi2** Result of adding Gaussian noise (p.221) (standard deviation = 13) to **che1**.
- cir1** 171×150 binary of doughnut shaped object.
- cir2** 512×512 binary circle centered in the right of the image.
- cir2neg1** Result of inverting (p.63) **cir2**.
- cir2or1** Result of ORing (p.58) **cir2** with **cir3**.
- cir2or2** Result of ORing (p.58) **cir2neg1** with **cir3neg1**.
- cir3** 512×512 binary circle centered in the left of the image.
- cir3neg1** Photographic negative of **cir3**.
- reg1** 160×160 binary of artificial boundary.
- reg1neg1** The inversion (p.63) of **reg1**.
- reg1fst1** Artificial binary image showing one pixel which lies inside the region defined in **reg1**.
- reg1dil1** Result of dilating (p.118) **reg1fst1** using custom structuring element.
- reg1and1** Result of ANDing (p.55) **reg1dil1** and **reg1neg1**.
- reg1dil2** Result of dilating (p.118) **reg1and1**.
- reg1and2** Result of ANDing (p.55) **reg1dil2** and **reg1neg1**.
- reg1and3** Result of dilating (p.118) **reg1and2** and then ANDing (p.55) it with **reg1neg1**.
- reg1and4** Result of dilating (p.118) **reg1and3** and then ANDing (p.55) it with **reg1neg1**.
- reg1and5** Result of dilating (p.118) **reg1and4** and then ANDing (p.55) it with **reg1neg1**.
- reg1and6** Result of dilating (p.118) **reg1and5** and then ANDing (p.55) it with **reg1neg1**.
- reg1and7** Result of dilating (p.118) **reg1and6** and then ANDing (p.55) it with **reg1neg1**.
- reg1fil1** Result of ORing (p.58) **reg1and7** with **reg1**.
- rlf1** 256×256 binary of triangle, hexagon and squares.
- rlf1aff1** Result of applying an affine transformation (p.100) (*i.e.* translation) to **rlf1**.
- rlf1aff2** Result of applying an affine transformation (p.100) (*i.e.* rotation) to **rlf1**.
- rlf1aff3** Result of applying an affine transformation (p.100) (*i.e.* reflection) to **rlf1**.
- rlf1aff4** Result of applying an affine transformation (p.100) (*i.e.* general affine warping) to **rlf1**.

rlf1aff5 Result of applying an affine transformation (p.100) (*i.e.* translation and geometric scaling) to **rlf1**.

sqr1 256×256 binary of occluding rectangles.

sqr1can1 Result of applying the Canny edge detector (p.192) to **sqr1**.

sqr1can2 Result of adding 1% salt and pepper noise (p.221) to **sqr1can1**.

sqr1can3 Result of image editing (p.233) **sqr1can1**.

sqr1hou1 Hough transform (p.214) of **sqr1can1**.

sqr1hou2 Histogram equalized (p.78) version of **sqr1hou1**.

sqr1hou3 Lines found from analysis of Hough transform (p.214) of **sqr1can1**, overlaid on top of original image **sqr1**.

sqr1hou4 Hough transform (p.214) of **sqr1can2** (after histogram equalization (p.78)).

sqr1hou5 Lines found from analysis of Hough transform (p.214) of **sqr1can2**, overlaid on top of original image **sqr1**.

sqr1hou6 Hough transform (p.214) of **sqr1can3** (after histogram equalization (p.78)).

sqr1hou7 Lines found from analysis of Hough transform (p.214) of **sqr1can3**, overlaid on top of original image **sqr1**.

stp1 256×256 binary of diagonal stripes.

stp1fur1 Result of applying a Fourier Transform (p.209) to **stp1**.

stp1fur2 Result of applying a logarithmic transform (p.82) to the Fourier Transform (p.209) of **stp1**.

stp1fur3 Result of thresholding (p.69) **stp1fur1** at a value of 13.

stp1fur4 Result of thresholding (p.69) the adding (p.43) of the Fourier Transform (p.209) of **stp1** and **stp2**.

stp1fil2 Inverse Fourier Transform (p.209) of **stp1fur4**.

stp1fur5 Logarithmic transform (p.82) of the product (p.48) of **stp1fur1** and an image of a circle (radius = 32).

stp1fil1 Inverse Fourier Transform (p.209) of **stp1fur5**.

stp1fur6 Result of scaling down (p.48) the Fourier Transform (p.209) of **stp1** with *0.0001* before applying logarithmic transform (p.82).

stp2 256×256 binary of vertical stripes (2 pixels wide).

stp2fur1 Result of applying a Fourier Transform (p.209) to **stp2**.

stp1fur4 Result of thresholding (p.69) the adding (p.43) of the Fourier Transform (p.209) of **stp1** and **stp2**.

stp1fil2 Inverse Fourier Transform (p.209) of **stp1fur4**.

stp3 256×256 binary of vertical and diagonal stripes

stp3fur1 Result of applying a Fourier transform (p.209) to **stp3**.

(*i.e.* the pixel addition (p.43) of **stp1** and **stp2**).

Astronomical

The *Astronomical Image Catalogue* contains a collection of images depicting lunar surfaces and galactic star clusters.

lun1 640×400 8-bit color of lunar vehicle. Any use of this image must include the acknowledgement: ‘We thank the NASA Dryden Research Aircraft Photo Archive use of this image.’

mof1 320×200 8-bit color of astronaut on the moon. Any use of this image must include the acknowledgement: ‘We thank the NASA Dryden Research Aircraft Photo Archive use of this image.’

moo1 512×512 grayscale of lunar surface.

moo1bld1 Result of blending (p.53) **moo1** and **fce6**.

Any use of this image must include the acknowledgement: ‘We thank the NASA Dryden Research Aircraft Photo Archive use of this image.’

moo2 512×512 grayscale of lunar surface (low contrast).

moo2hst2 Intensity histogram (p.105) of **moo2**.

moo2str1 Result of contrast stretching (p.75) **moo2**.

moo2heq1 Result of histogram equalizing (p.78) **moo2**.

moo2hst1 Intensity histogram (p.105) of **moo2heq1**.

moo2lab1 The labeled 8-bit color image resulting from applying connected components labeling (p.114) to a binary version of **moo2str1**.

moo2lab2 The grayscale equivalent of **moo2str1**.

Any use of this image must include the acknowledgement: ‘We thank the NASA Dryden Research Aircraft Photo Archive use of this image.’

str1 256×256 grayscale of Horsehead Nebula. Any use of this image must include the acknowledgement: ‘We thank Royal Greenwich Observatory, the Starlink Project and Rutherford Appleton Laboratory for the use of this image.’

str2 256×256 grayscale of NGC1365 spiral galaxy. Any use of this image must include the acknowledgement: ‘We thank Royal Greenwich Observatory, the Starlink Project and Rutherford Appleton Laboratory for the use of this image.’

str3 256×256 grayscale of NGC1097 spiral galaxy (interesting because of a jet coming from the galactic center. Any use of this image must include the acknowledgement: ‘We thank Royal Greenwich Observatory, the Starlink Project and Rutherford Appleton Laboratory for the use of this image.’

Faces

The *Faces Image Catalogue* contains a collection of images depicting animal (mostly human) faces. Each image listed below contains a different subject unless otherwise noted.

ape1 512×512 grayscale of mandrill.

ape1and1 Result of ANDing (p.55) **ape1** with 00000001 binary.

ape1and4 Result of ANDing (p.55) **ape1** with 00001000 binary.

ape1and6 Result of ANDing (p.55) **ape1** with 00100000 binary.

ape1and7 Result of ANDing (p.55) **ape1** with 01000000 binary.

- ape1and8** Result of ANDing (p.55) **ape1** with 10000000 binary.
- ape1opn1** Result of graylevel opening (p.127) **ape1** with a 5×5 square structuring element.
- ape1clo1** Result of graylevel closing (p.130) **ape1** with a 3×3 square structuring element.
- ape1ref1** Result of reflecting (p.95) the left half of **ape1** about a vertical axis through the middle of the image.
- ape1ref2** Result of reflecting (p.95) the right half of **ape1** about a vertical axis through the middle of the image.
- bab4** 393×334 8-bit color of baby. Any use of this image must include the acknowledgement: 'We thank Dr. X. Huang of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'
- cln1** 256×256 grayscale of clown.
- cln1can1** Result of applying the Canny edge detector (p.192) to **cln1** (threshold: (p.69) 255, 1; Gaussian σ : 1.0).
- cln1can2** Result of applying the Canny edge detector (p.192) to **cln1** (threshold: (p.69) 255, 220; Gaussian σ : 1.0).
- cln1can3** Result of applying the Canny edge detector (p.192) to **cln1** (threshold: (p.69) 128, 1; Gaussian σ : 1.0).
- cln1can4** Result of applying the Canny edge detector (p.192) to **cln1** (threshold: (p.69) 128, 1; Gaussian σ : 2.0).
- cln1fur1** The magnitude image resulting from Fourier Transforming (p.209) **cln1**.
- cln1fur2** The logarithm of the magnitude image resulting from Fourier Transforming (p.209) **cln1**.
- cln1fur3** The phase image resulting from Fourier Transforming (p.209) **cln1**.
- cln1fil1** The inverse Fourier Transform (p.209) of **cln1fur1**.
- cln1noi1** Result of adding Gaussian noise (p.221) to **cln1**, mean = 0, standard deviation = 8.
- cln1low1** Result of lowpass filtering (p.167) **cln1noi1** with an ideal filter and a cut-off frequency of $\frac{1}{3}$.
- cln1low2** Result of lowpass filtering (p.167) **cln1noi1** with an ideal filter and a cut-off frequency of 0.5.
- cln1low3** Result of lowpass filtering (p.167) **cln1noi1** with a Butterworth filter and a cut-off frequency of $\frac{1}{3}$.
- cln1low4** Result of lowpass filtering (p.167) **cln1noi1** with a Butterworth filter and a cut-off frequency of 0.5.
- cln1hig1** Result of highpass filtering (p.167) **cln1** with a Butterworth filter and a cut-off frequency of 0.5.
- cln1hig2** The absolute value of the filtered spatial domain image **cln1hig1**.
- cln1log1** Result of applying the Laplacian of Gaussian filter (p.173), mean = 0, $\sigma = 1.0$, to **cln1**.
- cln1log2** Result of applying the Laplacian of Gaussian filter (p.173), mean = 0, $\sigma = 2.0$, to **cln1**.
- cln1log3** Result of applying the Laplacian of Gaussian filter (p.173), mean = 0, $\sigma = 3.0$, to **cln1**.
- cln1zer1** Result of applying zero crossing detection (p.199) to **cln1log1**.
- cln1zer2** Result of applying zero crossing detection (p.199) to **cln1log1**, where shallow crossings are ignored.
- cln1zer3** Result of applying zero crossing detection (p.199) to **cln1log2**.

cln1zer4 Result of applying zero crossing detection (p.199) to **cln1log3**.
cln1rob1 Result of applying the Roberts Cross edge detector (p.184) to **cln1**.
cln1rob2 Result of thresholding (p.69) **cln1rob1** at a pixel value of 80.
cln1sob1 Result of applying the Sobel edge detector (p.188) to **cln1**.
cln1sob2 Result of histogram equalizing (p.78) **cln1sob1**.
cln1sob3 Result of thresholding (p.69) **cln1sob2** at a pixel value of 200.
cln1thn1 Result of thinning (p.137) **cln1sob1**.
cln1cmp1 Result of applying the Prewitt compass edge detector (p.195) to **cln1**.
cln1cmp2 The grayscale orientation images of **cln1cmp1** after contrast stretching (p.75).
cln1cmp3 The labeled 8-bit color orientation images of **cln1cmp1** after contrast stretching (p.75).
cln1cmp4 Result of histogram equalizing (p.78) **cln1cmp1**.
cln1trn1 Result of translating (p.97) **cln1** by 1 pixel in the x and y directions and then subtracting from the original.
cln1trn2 Result of translating (p.97) **cln1** by 6 pixels in the x and y directions and then subtracting from the original.

Any use of this image must include the acknowledgement: 'We thank Georgia Institute of Technology for the use of these images acquired from their on-line image database.'

fac1 128×128 range image of face.

fce1 399×291 grayscale of woman.

fce2 299×361 grayscale of man.

fce2usp1 Result of unsharp masking (p.178) **fce2**.

fce2log1 Result of taking the Laplacian of Gaussian (p.173) of **fce2** (standard deviation 1.0).

fce2log2 Result of taking the Laplacian of Gaussian (p.173) of **fce2** (standard deviation 1.0) and then subtracting (p.45) from the original.

fce2lap2 Result of taking the Laplacian (p.173) of **fce2**.

fce2lap1 Result of taking the Laplacian (p.173) of **fce2** and then subtracting (p.45) from the original.

fce3 302×357 grayscale of man.

fce4 283×367 grayscale of man.

fce4exp1 Result of exponential transform (p.85) with basis *1.005*

fce4exp2 Result of exponential transform (p.85) with basis *1.01*

fce4pow1 Result of raising each pixel to the power (p.85) of *1.5*.

fce4pow2 Result of raising each pixel to the power (p.85) of *2.5*.

fce5 306×341 grayscale of man.

fce5noi1 Result of adding salt noise (p.221) of 0.5% to **fce5**.

fce5ero1 Result of eroding (p.123) **fce5noi1** with a 3×3 structuring element (p.241).

fce5opn1 Result of opening (p.127) **fce5noi1** with a 3×3 structuring element (p.241).

fce5clo2 Result of closing (p.130) **fce5noi1** with a 3×3 structuring element (p.241).

fce5noi2 Result of adding pepper noise (p.221) of 0.5% to **fce5**.

fce5opn2 Result of opening (p.127) **fce5noi2** with a 3×3 structuring element (p.241).

-
- fce5clo1* Result of closing (p.130) *fce5noi2* with a 3×3 structuring element (p.241).
- fce5di11* Result of dilating (p.118) *fce5noi2* with a 3×3 structuring element (p.241).
- fce5noi3* Result of adding salt and pepper noise (p.221) of 0.5% (each) to *fce5*.
- fce5csm1* Result of conservative smoothing (p.161) *fce5noi3*.
- fce5mea1* Result of mean smoothing (p.150) *fce5noi3* with a 3×3 kernel.
- fce5mea2* Result of mean smoothing (p.150) *fce5noi3* with a 5×5 kernel.
- fce5med1* Result of median smoothing (p.153) *fce5noi3* with a 3×3 kernel.
- fce5crm1* Result of filtering *fce5noi3* with 13 iterations of Crimmins Speckle Removal Algorithm (p.164).
- fce5noi4* Result of adding Gaussian noise (p.221) of mean = 0, standard deviation = 8 to *fce5*.
- fce5mea3* Result of mean smoothing (p.150) *fce5noi4* with a 3×3 kernel.
- fce5mea6* Result of mean smoothing (p.150) *fce5noi4* with a 5×5 kernel.
- fce5med2* Result of median smoothing (p.153) *fce5noi4* with a 3×3 kernel.
- fce5gsm1* Result of Gaussian smoothing (p.156) *fce5noi4* with a 5×5 kernel.
- fce5crm4* Result of filtering *fce5noi4* with 2 iterations of Crimmins Speckle Removal Algorithm (p.164).
- fce5crm5* Result of filtering *fce5noi4* with 1 iteration of Crimmins Speckle Removal Algorithm (p.164).
- fce5usp1* Result of unsharp masking (p.178) *fce5noi4*.
- fce5noi5* Result of adding Gaussian noise (p.221) of mean = 0, standard deviation = 13 to *fce5*.
- fce5mea4* Result of mean smoothing (p.150) *fce5noi5* with a 3×3 kernel.
- fce5med3* Result of median smoothing (p.153) *fce5noi5* with a 3×3 kernel.
- fce5csm2* Result of conservative smoothing (p.161) *fce5noi5* with a 3×3 kernel.
- fce5noi6* Result of adding Gaussian noise (p.221) of mean = 0, standard deviation = 20 to *fce5*.
- fce5noi7* Result of adding 3% salt and pepper noise (p.221) to *fce5*.
- fce5noi8* Result of adding 5% salt noise (p.221) to *fce5*.
- fce5crm3* Result of filtering *fce5noi8* with 8 iterations of Crimmins Speckle Removal Algorithm (p.164).
- fce6* 274×303 grayscale of woman.
- moo1bld1* Result of blending (p.53) *fce6* with *moo1*.
- goo1* 546×420 8-bit color of goose face.
- grd1* 381×369 8-bit color of 2 graduates.
- man1* 311×437 8-bit color of man carrying dog.
- man8* 256×256 grayscale of man.
- man8log1* Logarithmic transform (p.82) and scaling (p.48) of *man8*.
- man8exp1* Exponential transform (p.85) of *man8log1*
- man8exp2* Exponential transform (p.85) of unscaled version of *man8log1* which was stored in floating point format (p.239)
- wom1* 256×256 grayscale of woman.

- wom1hst1 Intensity histogram (p.105) of wom1.
- wom1heq1 Result of histogram equalizing (p.78) wom1.
- wom1str1 Result of contrast stretching (p.75) wom1 using linear interpolation between $c = 79$ and $d = 136$.
- wom1hst2 Intensity histogram (p.105) of wom1str1.
- wom1str2 Result of contrast stretching (p.75) wom1 using a *cutoff fraction*, $c = 0.03$.
- wom1hst3 Intensity histogram (p.105) of wom1str2.
- wom1str3 Result of contrast stretching (p.75) wom1 using $c = 0.8$.
- wom1hst4 Intensity histogram (p.105) of wom1str3.
- wom1noi1 Result of adding 0.1% salt and pepper noise (p.221) to wom1str2.
- wom1crp1 Result of cropping and zooming (p.90) wom1noi1.
- wom1crm1 Result of filtering wom1noi1 using 1 iteration of Crimmins Speckle Removal Algorithm (p.164).
- wom1crp2 Result of cropping and zooming (p.90) wom1crm1.
- wom1crm2 Result of filtering wom1noi1 using 4 iterations of Crimmins Speckle Removal Algorithm (p.164).
- wom1crp3 Result of cropping and zooming (p.90) wom1crm2.
- wom1crm3 Result of filtering wom1noi1 using 8 iterations of Crimmins Speckle Removal Algorithm (p.164).
- wom1crm4 Result of adding 3% salt and pepper noise (p.221) to wom1str2 and then applying 11 iterations of Crimmins Speckle Removal Algorithm (p.164).

Any use of this image must include the acknowledgement: 'We thank Georgia Institute of Technology for the use of these images acquired from their on-line image database.'

wom2 256×256 grayscale of woman.

- wom2exp1 Result of applying exponential operator (p.85) to wom2.
- wom2heq1 Result of histogram equalizing (p.78) wom2.
- wom2hst1 Histogram (p.105) of wom2
- wom2hst2 Histogram (p.105) of wom2heq

Any use of this image must include the acknowledgement: 'We thank Georgia Institute of Technology for the use of these images acquired from their on-line image database.'

wom3 409×572 8-bit color of woman. Any use of this image must include the acknowledgement: 'We thank R. Aguilar-Chongtay of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

wom4 386×537 8-bit color of woman.

wom5 427×346 8-bit color of woman. Any use of this image must include the acknowledgement: 'We thank Dr. X. Huang of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

wom6 386×544 8-bit color of woman. Any use of this image must include the acknowledgement: 'We thank Dr. X. Huang of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

Food

The *Food* Image Catalogue contains a collection of images depicting food in various stages of processing. Most commonly, fruits and vegetables are depicted in a still-life setting with a variety of different illumination scenarios.

fsh2 500×200 grayscale of haddock. Any use of this image must include the acknowledgement: 'We thank Dr. N. J. C. Strachan, Torry Research Station and Drs. C. Glasbey and G. W. Horgan of Biomathematics and Statistics Scotland, University of Edinburgh for the use of this image.'

fru1 256×256 grayscale of apples, bananas and grapes.

fru2 427×352 8-bit color of orange.

fru3 536×321 8-bit color of oranges and bananas.

fru4 575×319 8-bit color of oranges and bananas (strong illumination gradient).

fru5 533×283 8-bit color of red apples.

gar1 391×162 grayscale of garlic.

kiw1 515×393 grayscale of kiwis.

lek1 367×278 grayscale of leeks.

orn1 583×383 grayscale of oranges in barrel (out of focus).

orn1crp1 Result of cropping **orn1**.

orn1dec1 Result of estimating the optical transfer function (OTF) for **orn1crp1** with a Gaussian image (standard deviation = 3) and then applying inverse filtering with a minimum OTF threshold of 10. (See frequency filtering (p.167) worksheet.)

orn1dec2 Result of estimating the optical transfer function (OTF) for **orn1crp1** with a Gaussian image (standard deviation = 10) and then applying inverse filtering with a minimum OTF threshold of 10. (See frequency filtering (p.167) worksheet.)

orn2 541×391 grayscale of oranges in barrel.

pmk1 685×237 carved Halloween pumpkins.

pot1 353×541 8-bit color of potatoes.

sap1 233×238 grayscale of salt and pepper shakers.

tom1 546×392 grayscale of tomatos (out of focus).

tom2 543×392 grayscale of tomatos.

Line

The *Line* Image Catalogue contains a collection of line-based imagery including text (English, unless otherwise specified), blue prints, circuit diagrams, testcards, cartoons, *etc.* In some images, strong illumination gradients exist.

brd1 327×211 8-bit color of circuit board (bad focus).

brd2 498×325 8-bit color of circuit board.

hnd1 504×507 grayscale of hand-addressed envelope face.

hnd2 500×600 8-bit color of Guest House register.

hse1 256×256 binary line drawing of a house.

hse1rob1 Result of applying the Roberts Cross edge detector (p.184) to **hse1**.

hse1fou1 Fourier Transform (p.209) of **hse1**. Displayed after histogram equalization (p.78).

hse1msk3 Lowpass frequency filtering kernel.

hse1fou2 Result of multiplying (p.48) the Fourier Transform (p.209) of **hse1** with **hse1msk3**.

hse1fil1 Inverse Fourier Transform (p.209) of **hse1fou2**.

hse1msk1 Vertical stripe smoothing frequency mask for **hse1**.

hse1fil2 Result of multiplying (p.48) the frequency domain representation of **hse1** and **hse1msk1**, and then inverse Fourier Transforming (p.209) (and normalizing) the product.

hse1msk2 Vertical stripe preserving frequency mask for **hse1**.

hse1fil3 Result of multiplying (p.48) the frequency domain representation of **hse1** and **hse1msk2**, and then inverse Fourier Transforming (p.209) (and normalizing) the product.

hse1fil4 Result of thresholding (p.69) **hse1fil3**.

hse2 860×583 grayscale architectural drawing of a domestic dwellings. Any use of this image must include the acknowledgement: 'We thank Ms. S. Flemming for the use of this image.'

hse3 642×809 grayscale architectural drawing of building interior. Any use of this image must include the acknowledgement: 'We thank Ms. S. Flemming for the use of this image.'

hse4 203×464 grayscale architectural sketch (faint and broken lines). Any use of this image must include the acknowledgement: 'We thank Ms. S. Flemming for the use of this image.'

pcb1 264×268 grayscale of underside of single-sided printed circuit board. Any use of this image must include the acknowledgement: 'We thank Mr. A. MacLean of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

pcb2 827×546 grayscale of double-sided printed circuit board. Any use of this image must include the acknowledgement: 'We thank Mr. A. MacLean of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

rob1 580×481 8-bit color of robot cartoon.

rob1sca1 Grayscale result of subsampling (p.90) **rob1** by a factor of 2.

rob1ldh1 Normalized result of line detecting (p.202) **rob1**.

rob1ldh2 Result of thresholding (p.69) **rob1ldh1**.

Any use of this image must include the acknowledgement: 'We thank Mr. M. Westhead of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

shu2 308×396 binary line drawing of a NASA shuttle. Any use of this image must include the acknowledgement: 'We thank the NASA Dryden Research Aircraft Photo Archive use of this image.'

son1 384×500 grayscale of sonnet text with illumination gradient.

son1thr1 Result of thresholding (p.69) **son1** at a pixel value of 128.

son1sub1 Result of pixel subtracting (p.45) **son2** from **son1** and adding (p.43) an offset of 100.

son1sub2 Result of pixel subtracting (p.45) **son2** from **son1** using wrapping (p.241).

son1sub3 Result of pixel subtracting (p.45) **son2** from **son1** using absolute difference.

son1sub4 Result of applying gamma correction (p.85) to **son1sub3**.

son1thr3 Result of thresholding (p.69) **son1sub1** at a pixel value of 80.

- son1div1** Result of pixel dividing (p.50) **son1** by **son2**.
- son1thr2** Result of thresholding (p.69) **son1div1** at a pixel value of 160.
- son1adp1** Result of adaptive thresholding (p.72) **son1** using the *mean* of a 7×7 neighborhood.
- son1adp2** Result of adaptive thresholding (p.72) **son1** using the *mean*- C as a local threshold, where $C = 7$ and the neighborhood size is 7×7 .
- son1adp3** Result of adaptive thresholding (p.72) **son1** using the *mean*- C as a local threshold, where $C = 10$ and the neighborhood size is 75×75 .
- son1adp4** Result of adaptive thresholding (p.72) **son1** using the *median*- C as a local threshold, where $C = 4$ and the neighborhood size is 7×7 .
- son2** 384×500 grayscale of sonnet text lightfield.
- son3** 512×512 grayscale of sonnet text.
- son3fur2** The logarithm of the magnitude of the Fourier Transform (p.209) image corresponding to **son3**.
- son3fur4** The thresholded magnitude of the Fourier Transform (p.209) image corresponding to **son3**.
- son3rot1** Result of rotating (p.93) **son3** by 45 degrees clockwise about its center.
- son3fur1** The logarithm of the magnitude of the Fourier Transform (p.209) image corresponding to **son3rot1**.
- son3fur3** The thresholded magnitude of the Fourier Transform (p.209) image corresponding to **son3rot1**.
- tst1** 360×360 8-bit color of testcard 1.
- tst2** 760×1003 grayscale of testcard 2.
- tst2sca1** Result of reducing the resolution of (*i.e.* subsampling) **tst2** by a factor of 2.
- tst2sca2** Result of reducing the resolution of (*i.e.* subsampling) **tst2** by a factor of 4.
- tst2sca3** Result of reducing the resolution of (*i.e.* subsampling) **tst2** by a factor of 8.
- tst2sca4** Result of reducing the resolution of (*i.e.* subsampling) **tst2** by a factor of 16.
- txt1** 500×480 grayscale of text (poor quality reproduction).
- txt2** 256×256 binary of alpha-numeric text.
- txt2msk1** Mask for **txt2** containing only the letter X.
- txt2fur1** Result of scaling (p.48) the Fourier transform (p.209) of **txt2msk1** and thresholding (p.69) it at a value of 10. (See frequency filter (p.167) worksheet.)
- txt2fil1** Result of Fourier Transforming (p.209) **txt2** and **txt2msk1**, multiplying (p.48) the images, applying an inverse Fourier Transform to the multiplied image, and, finally, scaling (p.48) the spatial domain image.
- txt2fil2** Result of thresholding (p.69) **txt2fil1** at the value 255.
- txt2fil3** Result of multiplying (p.48) **txt2fur1** and the Fourier Transform (p.209) of **txt2**, and then applying an inverse Fourier transform to the multiplied image.
- txt2fil4** Result of thresholding (p.69) **txt2fil3**.
- txt3** 722×857 grayscale of handwritten Japanese text.
- txt3crp1** Result of cropping **txt3** and then zooming (p.90) by a factor of 4.
- txt3thr1** Result of thresholding (p.69) **txt3** at a value of 180.
- txt3thn1** Result of thinning (p.137) **txt3thr1** with custom structuring element.

- txt3dil1** Result of dilating (p.118) **txt3thr1** twice with a 3×3 square structuring element.
- txt3thn2** Result of thinning (p.137) **txt3dil1** with custom structuring element.
- txt3prn1** Result of pruning **txt3thn2** with 2 iterations of each orientation of the custom structuring element.
- txt3dil2** Result of dilating (p.118) **txt3thr1** three times with a 3×3 square structuring element.
- txt3thn3** Result of thinning (p.137) **txt3dil2** with custom structuring element.
- txt3prn2** Result of pruning **txt3thn3** with 4 iterations of each orientation of the custom structuring element.
- txt3end1** Result of applying hit-and-miss operator (p.133) to **txt3thr1** using custom structuring element.
- txt3end2** Result of applying 5 iterations of *conditional dilation* - *i.e.* dilating (p.118) **txt3end1** using a 3×3 structuring element and then ANDing (p.55) it with the thinned version.
- txt3mor1** Result of ORing (p.58) **txt3end2** with pruned version.

Any use of this image must include the acknowledgement: 'We thank Dr. X. Huang of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

- ugr1** 522×733 grayscale of Ugaritic text on tablet. Any use of this image must include the acknowledgement: 'We thank Dr. N. Wyatt and Dr. J. B. Lloyd of the Ras Shamra project at University of Edinburgh for the use of this image.'

Medical

The *Medical* Image Catalogue contains a variety of medical imagery. Subjects range from photomicroscopy of murine neurological structures to x-ray and magnetic resonance imagery of human organs.

- alg1** 512×512 grayscale of algal cells (differential interference microscope). This image was collected as part of a research program to manage algal ponds for waste treatment. The objective is to identify, count and measure the cells. Any use of this image must include the acknowledgement: 'We thank Dr. N. J. Martin, Scottish Agricultural College, and Drs. C. Glasbey and G. W. Horgan of Biomathematics and Statistics Scotland, University of Edinburgh for the use of this image.'
- axn1** 400×300 grayscale of axonal growth (microscope). This image and the next were used to investigate the directionality of growth of axons. In both scenarios, the target which the axons sought to innervate was located off to the right of the image. The initial/start position of the growing axons varies in each image. Any use of this image must include the acknowledgement: 'We thank Dr. B. Lotto of the Department of Physiology, University of Edinburgh for the use of this image.'
- axn2** 400×300 grayscale of axonal growth (microscope). Any use of this image must include the acknowledgement: 'We thank Dr. B. Lotto of the Department of Physiology, University of Edinburgh for the use of this image.'
- ce11** 597×403 8-bit color of embryonic mouse neurones (confocal microscope). This image (and **ce12**) were collected as part of a study investigating neuronal growth factors influencing cell longevity. The small white cells are dead. Any use of this image must include the acknowledgement: 'We thank Dr. B. Lotto of the Department of Physiology, University of Edinburgh for the use of this image.'
- ce12** 577×355 8-bit color of embryonic mouse neurones (confocal microscope). Any use of this image must include the acknowledgement: 'We thank Dr. B. Lotto of the Department of Physiology, University of Edinburgh for the use of this image.'

ce14 450×450 grayscale of human nerve cells (large dark) and glia cells (small black).

ce14thr1 Result of thresholding (p.69) **ce14** so as to retain (as foreground) pixel values originally in the intensity band: 0 - 150.

ce14lab1 Result of connected components labeling (p.114) **ce14thr1**.

ce14thr2 Result of thresholding (p.69) **ce14** so as to retain (as foreground) pixel values originally in the intensity band: 130 - 150.

ce14thr3 Result of thresholding (p.69) **ce14** at a value of 210.

ce14opn1 Result of opening (p.127) **ce14thr3** using an 11 pixel circular structuring element.

ce14opn2 Result of opening (p.127) **ce14thr3** using a 7 pixel circular structuring element.

This image was collected as part of a research program investigating the effects of the AIDS virus on the quantity of nerve cells. Any use of this image must include the acknowledgement: 'We thank R. Aguilar-Chongtay of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

ce15 551×686 grayscale of embryonic mouse neurones with endothelial cell (upper left) (electron-micrograph). This image and the following image were used in a study comparing attributes of embryonic vs postnatal cells. (The younger cells are less round, smaller and darker.) Any use of this image must include the acknowledgement: 'We thank Dr. B. Lotto of the Department of Physiology, University of Edinburgh for the use of this image.'

ce16 546×672 grayscale of postnatal mouse neurones. Any use of this image must include the acknowledgement: 'We thank Dr. B. Lotto of the Department of Physiology, University of Edinburgh for the use of this image.'

ce17 546×680 grayscale of embryonic neurones and a blood cell enclosed within an endothelial cell.

ce17neg1 Photographic negative of **ce17**.

Any use of this image must include the acknowledgement: 'We thank Dr. B. Lotto of the Department of Physiology, University of Edinburgh for the use of this image.'

cla3 200×260 grayscale of mouse nervous cells extracted on embryonic day 15 (transilluminated microscope).

cla3hst1 Histogram (p.105) of **cla3**

cla3str1 Result of contrast stretching (p.75) **cla3**

cla3hst2 Histogram (p.105) of **cla3str1**

cla3hst3 Result of stretching the y-axis of **cla3hst2**

This image (and **clb3** and **clc3**) were collected as part of a study investigating neuronal growth factors influencing cell longevity. The dark cells are dead. Any use of this image must include the acknowledgement: 'We thank Dr. B. Lotto of the Department of Physiology, University of Edinburgh for the use of this image.'

clb3 200×260 grayscale of embryonic day 17 mouse nervous cells. Any use of this image must include the acknowledgement: 'We thank Dr. B. Lotto of the Department of Physiology, University of Edinburgh for the use of this image.'

clc3 200×260 grayscale of embryonic day 19 mouse nervous cells.

clc3thr1 Result of thresholding (p.69) **clc3** at a value of 150

clc3lab1 Result of applying connected component labeling (p.114) to **clc3thr1**.

clc3lab2 Result of assigning 1 out of 8 distinctive colors to each class in **clc3lab1**.

clc3lab3 Result of assigning a different color to each class in **clc3lab1**.

Any use of this image must include the acknowledgement: 'We thank Dr. B. Lotto of the Department of Physiology, University of Edinburgh for the use of this image.'

dna1 300×512 grayscale of one stage in the DNA-sequencing of gene fragments (autoradiograph). In this study, approximately fifty mixtures were positioned as distinct spots along the top of the gel (as it is currently displayed). Each mixture then migrated and DNA fragments produced separate, approximately horizontal bands. Any use of this image must include the acknowledgement: 'We thank Dr. F. G. Wright, Dr. C. Glasbey and Dr. G. W. Horgan Biomathematics and Statistics Scotland, University of Edinburgh for the use of this image.'

egg1 512×512 grayscale of sheep parasite eggs (microscope).

egg1add1 Result of adding (p.43) a constant value of 50 to **egg1**

egg1add2 Result of adding (p.43) a constant value of 100 to **egg1** (using wrapping (p.241))

egg1add3 Result of scaling (p.48) **egg1** with a factor of 0.8 and adding (p.43) a constant value of 100 to the result (using wrapping (p.241))

egg1add4 Result of adding (p.43) a constant value of 100 to **egg1** (using a hard limit (p.241))

egg1sca1 Result of scaling (p.48) **egg1** with a factor of 1.3.

Any use of this image must include the acknowledgement: 'We thank Mr. S. Beard of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

fun1 500×500 grayscale of part of a fungal mycelium *Trichoderma viride*, which is a network of hyphae from a single fungal organism (photograph of fungal hyphae on cellophane-coated nutrient agar). Image analysis was required here to understand the spatial structure of the hyphae in relation to their environment. Any use of this image must include the acknowledgement: 'We thank Dr. K. Ritz of Scottish Crop Research Institute, and Drs. C. Glasbey and G. W. Horgan of Biomathematics and Statistics Scotland, University of Edinburgh for the use of this image.'

mam1 2048×2048 8-bit color mammography image of cancerous breast. (There are 3 cancers at coordinates (x,y,radius): (1374,1348,36), (1502,1178,20) and (1444,778,14).) Any use of this image must include the acknowledgement: 'We thank Dr. P. Thanisch and Mr. A. Hume of the Department of Computer Science, University of Edinburgh for the use of this image.'

mam2 2048×2048 8-bit color mammography image of cancerous breast. (There are 5 cancers at coordinates (x,y,radius): (1278,126,80), (1248,908,50), (1178,567,50), (1669,409,192) and (612,1060,150).) Any use of this image must include the acknowledgement: 'We thank Dr. P. Thanisch and Mr. A. Hume of the Department of Computer Science, University of Edinburgh for the use of this image.'

mri1 128×128 grayscale of woman's chest - where the subject has a cubic test object between her breasts (magnetic resonance imaging). This image and the next were obtained as part of an investigation into the changes in breast volume during the menstrual cycle. The aim of the study was to segment the images into different tissues. Any use of this image must include the acknowledgement: 'We thank Dr. M. A. Foster, Biomedical Physics & Bioengineering, University of Aberdeen, and Drs. C. Glasbey and G. W. Horgan of Biomathematics and Statistics Scotland, University of Edinburgh for the use of this image.'

mri2 128×128 grayscale of woman's chest, where the subject has a cubic test object between her breasts (magnetic resonance imaging). Any use of this image must include the acknowledgement: 'We thank Dr. M. A. Foster, Biomedical Physics & Bioengineering, University of Aberdeen, and Drs. C. Glasbey and G. W. Horgan of Biomathematics and Statistics Scotland, University of Edinburgh for the use of this image.'

mus1 512×512 grayscale of a section through a rat's soleus muscle. The transverse section has been stained to demonstrate the activity of Ca²⁺ (activated myofibrillar ATPase) and allows one to classify three types of fiber: fast-twitch oxidative glycolytic (dark), slow-twitch oxidative (light) and fast-twitch glycolytic (mid-gray). The quantity and size of the fibers are used

for research into clenbuterol, a drug which enhances muscle development. Any use of this image must include the acknowledgement: 'We thank Dr. C. A. Maltin, Rowett Research Institute, and Drs. C. Glasbey and G. W. Horgan of Biomathematics and Statistics Scotland, University of Edinburgh for the use of this image.'

- slt1** 1024×1024 binary showing locations of stellate lesions in image **stl1**. Any use of this image must include the acknowledgement: 'Data for this research was provided by Dr. N. Karssemeijer and the University Hospital Nijmegen.'
- slt2** 1024×1024 binary showing location of stellate lesions in image **stl2**. Any use of this image must include the acknowledgement: 'Data for this research was provided by Dr. N. Karssemeijer and the University Hospital Nijmegen.'
- soi1** 512×512 grayscale montage of a soil aggregate embedded in acrylic resin (backscattered electron scanning micrographs). The black areas are soil pores and the lighter areas are the inorganic and organic soil matrix. The image was used in a study of porosity and pore-size distribution within a sample of soil aggregates which related these characteristics to microbial activity in and outside the aggregates. Any use of this image must include the acknowledgement: 'We thank Dr. J. F. Darbyshire, Macaulay Land Use Research Institute, and Drs. C. Glasbey and G. W. Horgan of Biomathematics and Statistics Scotland, University of Edinburgh for the use of this image.'
- stl1** 1024×1024 8-bit color of stellate lesions. Any use of this image must include the acknowledgement: 'Data for this research was provided by Dr. N. Karssemeijer and the University Hospital Nijmegen.'
- stl2** 1024×1024 8-bit color of stellate lesions. Any use of this image must include the acknowledgement: 'Data for this research was provided by Dr. N. Karssemeijer and the University Hospital Nijmegen.'
- usd1** 360×300 graylevel of a cross-section through a sheep's back (ultrasound). Images were collected in order to estimate sheep body composition. The top, approximately horizontal, white line is the transducer-skin boundary, below which are the skin-fat and fat-muscle boundaries. The backbone is on the bottom left, from which a rib can be seen sloping slightly upwards. Any use of this image must include the acknowledgement: 'We thank Dr. G. Simm of Genetics and Behavioural Sciences, Scottish Agricultural College, and Drs. C. Glasbey and G. W. Horgan of Biomathematics and Statistics Scotland, University of Edinburgh for the use of this image.'
- usd2** 512×480 grayscale of heart (ultrasound).
- wrm1** 512×512 grayscale of nematodes (microscope).
- xra1** 256×256 grayscale of a cross-section through the thorax of a live sheep (x-ray computed tomography). This image was used in a study to estimate the quantity of fat and lean tissue. The lightest image areas are the backbone and the parts of the ribs which intersect the imaging plane. The muscles and internal organs appear slightly lighter than the fat tissue because they are slightly more opaque to x-rays. (The U-shaped plastic cradle in which the sheep was lying can also be seen.) x-ray attenuation is measured in Hounsfield units, which range between -1000 and about 1000. (The data have been reduced to the range -250 to 260, with all values < -250 assigned a pixel value of 0 and all values > 260 set to 255.) Any use of this image must include the acknowledgement: 'We thank Dr. G. Simm of Genetics and Behavioural Sciences, Scottish Agricultural College, and Drs. C. Glasbey and G. W. Horgan of Biomathematics and Statistics Scotland, University of Edinburgh for the use of this image.'

Motion Sequence

The images here comprise a computer generated motion sequence of part of a flight through the Yosemite Valley produced by Lynn Quam at SRI. For reference, the clouds are moving at a rate

of about 2 pixels per frame to the right, while the rest of the image flow is divergent, with speed of about 5 pixels per frame in the lower left corner.

Any use of these images must include the acknowledgement: 'Data for this research was provided by Dr. L. Quam and SRI.'

- yos2 316×252 grayscale of Yosemite fly-through. Frame 1.
- yos3 316×252 grayscale of Yosemite fly-through. Frame 2.
- yos4 316×252 grayscale of Yosemite fly-through. Frame 3.
- yos5 316×252 grayscale of Yosemite fly-through. Frame 4.
- yos6 316×252 grayscale of Yosemite fly-through. Frame 5.
- yos7 316×252 grayscale of Yosemite fly-through. Frame 6.
- yos8 316×252 grayscale of Yosemite fly-through. Frame 7.
- yos9 316×252 grayscale of Yosemite fly-through. Frame 8.
- yos10 316×252 grayscale of Yosemite fly-through. Frame 9.
- yos11 316×252 grayscale of Yosemite fly-through. Frame 10.
- yos12 316×252 grayscale of Yosemite fly-through. Frame 11.
- yos13 316×252 grayscale of Yosemite fly-through. Frame 12.
- yos14 316×252 grayscale of Yosemite fly-through. Frame 13.
- yos15 316×252 grayscale of Yosemite fly-through. Frame 14.
- yos16 316×252 grayscale of Yosemite fly-through. Frame 15.

Nature Scenes

The *Nature Scenes* Image Catalogue contains scenes depicting a variety of natural imagery, ranging from wildlife (*e.g.* wild birds, forests, beaches) to more domestic scenes of landscaped flower beds, park benches, pets and zoo animals.

- ape1 512×512 grayscale of mandrill.
 - ape1and1 Result of ANDing (p.55) ape1 with 00000001 binary.
 - ape1and4 Result of ANDing (p.55) ape1 with 00001000 binary.
 - ape1and6 Result of ANDing (p.55) ape1 with 00100000 binary.
 - ape1and7 Result of ANDing (p.55) ape1 with 01000000 binary.
 - ape1and8 Result of ANDing (p.55) ape1 with 10000000 binary.
 - ape1opn1 Result of graylevel opening (p.127) ape1 with a 5×5 square structuring element.
 - ape1clo1 Result of graylevel closing (p.130) ape1 with a 3×3 square structuring element.
 - ape1ref1 Result of reflecting (p.95) the left half of ape1 about a vertical axis through the middle of the image.
 - ape1ref2 Result of reflecting (p.95) the right half of ape1 about a vertical axis through the middle of the image.
- cow1 549×320 8-bit color of five brown cows.

- dov1 688×492 8-bit color of dove on grassy hill. Any use of this image must include the acknowledgement: 'We thank Dr. A. Dil of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'
- flr1 651×373 8-bit color of flower bed.
- flr3 498×696 8-bit color of flowers with bee.
- flr4 586×383 8-bit color of tulip flower bed.
- flr5 575×392 grayscale of flowers.
- goo1 546×420 8-bit color of goose face. Any use of this image must include the acknowledgement: 'We thank Dr. A. Dil of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'
- hic1 575×238 8-bit color of highland cows.
- hic2 386×253 8-bit color of highland cow.
- pea1 589×396 8-bit color of peacock.
- pen1 311×212 grayscale of penguins on parade.
- prk1 595×259 8-bit color of park bench.
- puf1 692×488 8-bit color of puffins on a rock. Any use of this image must include the acknowledgement: 'We thank Dr. A. Dil of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'
- rck1 688×480 8-bit color of many sized rocks. Any use of this image must include the acknowledgement: 'We thank Ms. V. Temperton for the use of this image.'
- rck2 489×694 8-bit color of large rock with moss. Any use of this image must include the acknowledgement: 'We thank Ms. V. Temperton for the use of this image.'
- rck3 697×486 8-bit color of many small rocks (shallow focus). Any use of this image must include the acknowledgement: 'We thank Ms. V. Temperton for the use of this image.'
- swa1 432×389 8-bit color of swans on a lake.
- tur1 463×313 8-bit color four black turkeys.
- tur1gry1 Grayscale equivalent to tur1
- tur1thr1 Result of thresholding (p.69) tur1gry1
- tur1lab2 Result of color labeling (p.114) tur1thr1
- tur1lab1 Result of grayscale labeling (p.114) tur1thr1

Office and Laboratory Scenes

The *Office and Laboratory Scenes* Image Catalogue contains images depicting typical indoor working environments. (Most images contain subject matter which deals in some way with computer technology.) The illumination varies greatly from image to image.

- ben1 400×606 grayscale of a mobile robot.
- ben2 Result of scaling (p.90) ben1.
- ben2gau1 Result of Gaussian smoothing (p.156) ben2 with standard deviation 1.0 (5×5 kernel).
- ben2gau2 Result of Gaussian smoothing (p.156) ben2 with standard deviation 2.0 (9×9 kernel).

- ben2gau3 Result of Gaussian smoothing (p.156) ben2 with standard deviation 4.0 (15×15 kernel).
- ben2usp1 Result of unsharp masking (p.178) ben2.
- bok1 357×567 grayscale of bookshelf.
- bok1cmp1 Magnitude image resulting from Prewitt compass edge detecting (p.195) bok1.
- bok1cmp2 Orientation image resulting from Prewitt compass edge detecting (p.195) bok1.
- bok1noi1 Result of adding Gaussian noise (p.221) (with standard deviation = 15) to bok1.
- bok1cmp3 Magnitude image resulting from Prewitt compass edge detecting (p.195) bok1noi1.
- bok1cmp4 Orientation image resulting from Prewitt compass edge detecting (p.195) bok1noi1.
- brd1 327×211 8-bit color of circuit board (bad focus).
- brd2 498×325 8-bit color of circuit board.
- cmp1 611×378 grayscale of PC computer.
- cmp2 523×393 grayscale of PC computers.
- ctr1 564×414 8-bit color of aeronautical test range mission control room at NASA Dryden Flight Research Center. Any use of this image must include the acknowledgement: 'We thank the NASA Dryden Research Aircraft Photo Archive use of this image.'
- leg1 700×500 grayscale of Lego vehicle. Any use of this image must include the acknowledgement: 'We thank Mr. D. Howie of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'
- rot1 288×360 grayscale of laboratory scene.
- rot1str1 Result of contrast stretching (p.75) rot1 using a *cutoff fraction*, $c = 0.9$.
- rot1aff1 The affine transformation (p.100) of rot1str1.
- Any use of this image must include the acknowledgement: 'We thank the Pilot European Image Processing Archive (PEIPA) use of this image.'
- stc1 346×581 grayscale descending spiral staircase.
- stc1cmp1 Edge magnitude image resulting from the application of the Prewitt compass edge detector (p.195).
- stc1cmp2 Edge orientation image resulting from the application of the Prewitt compass edge detector (p.195).
- wom4 386×537 8-bit color of woman at workstation.
- wrk1 507×384 grayscale of lab workbench (natural illumination).

Remote Sensing

The *Remote Sensing* Image Catalogue contains both low altitude (*e.g.* collected from aircraft) and high altitude (*i.e.* satellite) imagery. In the case of the latter, there are two spectral band images (*i.e.* visible and infra-red) available for each subject.

- aer1 696×474 8-bit color aerial view of suburban region.
- aer2 Gray-scale equivalent of the original image.
- air1 625×625 grayscale satellite (infra-red spectrum) image taken over Africa.

- air2** A scaled (p.90) version of **air1**.
- avi2cls1** The labeled image resulting from classifying (p.107) the multi-spectral image composed of **air2** and **avs2** into two distinct spectral classes.
- avi2lab1** The result of applying connected components labeling (p.114) to **avi2cls1**.
- avi2lab2** A grayscale version of **cls1lab1**.
- avi2cls2** The labeled image resulting from classifying (p.107) the multi-spectral image composed of **air2** and **avs2** into four distinct spectral classes.
- avi2cls4** A grayscale version of **avi2cls2**.
- avi2lab3** The result of applying connected components labeling (p.114) to **avi2cls2**.
- avi2lab4** A grayscale version of **cls2lab2**.
- avi2cls3** The labeled image resulting from classifying (p.107) the multi-spectral image composed of **air2** and **avs2** into six distinct spectral classes.
- avi2cls5** A grayscale version of **avi2cls3**.

Any educational or research users may freely use the high altitude images with the acknowledgement: 'We thank EUMETSAT for the use of this image.'. Any commercial users of the images must contact EUMETSAT Contracts and Legal Affairs, Germany +49-(6151)-807-819.

- arp1** 564×416 8-bit color aerial view of NASA Dryden Flight Research Center - first perspective. Any use of this image must include the acknowledgement: 'We thank the NASA Dryden Research Aircraft Photo Archive use of this image.'
- arp2** 564×412 8-bit color aerial view of NASA Dryden Flight Research Center: second perspective. Any use of this image must include the acknowledgement: 'We thank the NASA Dryden Research Aircraft Photo Archive use of this image.'
- avs1** 625×625 grayscale satellite (visible spectrum) image taken over Africa.

- avs2** A scaled (p.90) version of **avs1**.
- avi2cls1** The labeled image resulting from classifying (p.107) the multi-spectral image composed of **air2** and **avs2** into two distinct spectral classes.
- avi2lab1** The result of applying connected components labeling (p.114) to **avi2cls1**.
- avi2lab2** A grayscale version of **cls1lab1**.
- avi2cls2** The labeled image resulting from classifying (p.107) the multi-spectral image composed of **air2** and **avs2** into four distinct spectral classes.
- avi2cls4** A grayscale version of **avi2cls2**.
- avi2lab3** The result of applying connected components labeling (p.114) to **avi2cls2**.
- avi2lab4** A grayscale version of **cls2lab2**.
- avi2cls3** The labeled image resulting from classifying (p.107) the multi-spectral image composed of **air2** and **avs2** into six distinct spectral classes.
- avi2cls5** A grayscale version of **avi2cls3**.

Any educational or research users may freely use the high altitude images with the acknowledgement: 'We thank EUMETSAT for the use of this image.'. Any commercial users of the images must contact EUMETSAT Contracts and Legal Affairs, Germany +49-(6151)-807-819.

- bir1** 625×625 grayscale satellite (infra-red spectrum) image taken over the Americas.
- bir1hst1** The intensity histogram (p.105) of **bir1**.
- bvi1tdh1** The 2-D intensity histogram (p.105) of **bir1** and **bvs1**.

Any educational or research users may freely use the high altitude images with the acknowledgement: 'We thank EUMETSAT for the use of this image.'. Any commercial users of the images must contact EUMETSAT Contracts and Legal Affairs, Germany +49-(6151)-807-819.

bvs1 625×625 grayscale satellite (visible spectrum) image taken over the Americas.

bvs1hst1 The intensity histogram (p.105) of **bvs1**.

bvi1tdh1 The 2-D intensity histogram (p.105) of **bir1** and **bvs1**.

Any educational or research users may freely use the high altitude images with the acknowledgement: 'We thank EUMETSAT for the use of this image.'. Any commercial users of the images must contact EUMETSAT Contracts and Legal Affairs, Germany +49-(6151)-807-819.

eir1 625×313 grayscale satellite (infra-red spectrum) image taken over Europe. Any educational or research users may freely use the high altitude images with the acknowledgement: 'We thank EUMETSAT for the use of this image.'. Any commercial users of the images must contact EUMETSAT Contracts and Legal Affairs, Germany +49-(6151)-807-819.

evs1 625×313 grayscale satellite (visible spectrum) image taken over Europe. Any educational or research users may freely use the high altitude images with the acknowledgement: 'We thank EUMETSAT for the use of this image.'. Any commercial users of the images must contact EUMETSAT Contracts and Legal Affairs, Germany +49-(6151)-807-819.

pdcl 439×391 8-bit color aerial view of Palais de Chaillot, France.

sir1 640×480 grayscale satellite (infra-red spectrum) image taken over Scandinavia. Any educational or research users may freely use the high altitude images with the acknowledgement: 'We thank EUMETSAT for the use of this image.'. Any commercial users of the images must contact EUMETSAT Contracts and Legal Affairs, Germany +49-(6151)-807-819.

svs1 640×480 grayscale satellite (visible spectrum) image taken over Scandinavia.

svs1log1 Logarithmic transform (p.82) of **svs1**

Any educational or research users may freely use the high altitude images with the acknowledgement: 'We thank EUMETSAT for the use of this image.'. Any commercial users of the images must contact EUMETSAT Contracts and Legal Affairs, Germany +49-(6151)-807-819.

trn1 489×300 grayscale aerial view of train station.

uir1 512×512 grayscale satellite (infra-red spectrum) image taken over the United Kingdom. Any educational or research users may freely use the high altitude images with the acknowledgement: 'We thank EUMETSAT for the use of this image.'. Any commercial users of the images must contact EUMETSAT Contracts and Legal Affairs, Germany +49-(6151)-807-819.

uvs1 512×512 grayscale satellite (visible spectrum) image taken over the United Kingdom. Any educational or research users may freely use the high altitude images with the acknowledgement: 'We thank EUMETSAT for the use of this image.'. Any commercial users of the images must contact EUMETSAT Contracts and Legal Affairs, Germany +49-(6151)-807-819.

urb1 256×256 grayscale aerial view of urban area. Any use of this image must include the acknowledgement: 'We thank Georgia Institute of Technology for the use of these images acquired from their on-line image database.'

Stereo

The *Stereo* Image Catalogue contains a single dataset of static scene images (*i.e.* 11 images in total) with accurate descriptions of 3-D object locations. The images were collected at the Calibrated Imaging Laboratory at Carnegie Mellon University and are freely available through public ftp. (These data are provided under Contract No. F49620-92-C-0073, ARPA Order No. 8875.)

Any publication which includes reproductions of these images or data, or parts thereof, must be accompanied by the following annotation: 'Data for this research was partly provided by the Calibrated Imaging Laboratory at Carnegie Mellon University, sponsored by DARPA, NSF, and NASA.'

Details about data acquisition and data representation are available in:

HIPRDIR/images/stereo/general

(where HIPRDIR is the directory where HIPR is installed). Also, individual camera calibration parameters, data points and ground truth coordinates for each scene/image may be obtained from HIPRDIR/images/stereo/calib and HIPRDIR/images/stereo/points.

st00 576×384 grayscale of center image with the ground truth points highlighted.

st01 576×384 grayscale of first stereo image.

st02 576×384 grayscale of second stereo image.

st03 576×384 grayscale of third stereo image.

st04 576×384 grayscale of fourth stereo image.

st05 576×384 grayscale of fifth stereo image.

st06 576×384 grayscale of sixth stereo image.

st07 576×384 grayscale of seventh stereo image.

st08 576×384 grayscale of eighth stereo image.

st09 576×384 grayscale of ninth stereo image.

st10 576×384 grayscale of tenth stereo image.

st11 576×384 grayscale of eleventh stereo image.

Simple 2-D Objects

The *Simple 2-D Objects* Image Catalogue contains images depicting a primitive object (or objects) lying flat (and, in many cases silhouetted) against a simple background. Some images contain an illumination gradient.

mon1 337×238 grayscale of silhouetted touching coins.

mon1thr1 Result of thresholding (p.69) mon1 at a pixel value of 90.

mon1ero1 Result of eroding (p.123) mon1thr1 twice with a disk of diameter 11.

mon1ero2 Result of eroding (p.123) mon1thr1 twice with a square of diameter 11.

pap1 512×512 grayscale of pen and paper frame 1.

pap1and1 Result of ANDing (p.55) pap1 and pap2.

pap1and2 Result of ANDing (p.55) pap1 and pap3.

pap1xor1 Result of XORing (p.60) pap1 and pap3.

-
- pap1xor2** Result of XORing (p.60) **pap1** and **pap2**.
- pap2** 512×512 grayscale of pen and paper frame 2.
- pap1and1** Result of ANDing (p.55) **pap2** and **pap1**.
- pap1xor2** Result of XORing (p.60) **pap2** and **pap1**.
- pap3** 512×512 grayscale of pen and paper frame 3.
- pap1and2** Result of ANDing (p.55) **pap3** and **pap1**.
- pap1xor1** Result of XORing (p.60) **pap3** and **pap1**.
- scr1** 507×384 grayscale of screwdriver jumble frame 1.
- scr1sub1** The absolute difference of **scr1** and **scr2** obtained by pixel subtraction (p.45).
- scr1div1** Result of dividing (p.50) **scr1** by **scr2** and contrast stretching (p.75) the output.
- scr1div2** Result of dividing (p.50) **scr1** by **scr2** and histogram equalizing (p.78) the output.
- scr1div3** Like **scr1div1**, but calculation performed in pixel integer format (p.239).
- scr2** 507×384 grayscale of screwdriver jumble frame.
- scr1sub1** The absolute difference of **scr1** and **scr2** obtained by pixel subtraction (p.45).
- scr1div1** Result of dividing (p.50) **scr1** by **scr2** and contrast stretching (p.75) the output.
- scr1div2** Result of dividing (p.50) **scr1** by **scr2** and histogram equalizing (p.78) the output.
- scr1div3** Like **scr1div1**, but calculation performed in pixel integer format (p.239).
- scr3** 512×512 grayscale of screw and pen frame 1.
- scr3and1** Result of ANDing (p.55) graylevels of **scr3** and **scr4**.
- scr3and2** Result of ANDing (p.55) negatives (p.63) of **scr3** and **scr4**.
- scr3thr1** Result of thresholding (p.69) **scr3**
- scr3and3** Result of ANDing (p.55) negatives (p.63) of **scr3thr1** and **scr4thr1**.
- scr3or1** Result of ORing (p.58) negatives (p.63) of **scr3thr1** and **scr4thr1**.
- scr3or2** Result of ORing (p.58) **scr3thr1** and **scr4thr1**.
- scr3xor1** Result of XORing (p.60) **scr3thr1** and **scr4thr1**.
- scr4** 512×512 grayscale of screw and pen frame 2.
- scr3and1** Result of ANDing (p.55) graylevels of **scr4** and **scr3**.
- scr3and2** Result of ANDing (p.55) negatives (p.63) of **scr4** and **scr3**.
- scr4thr1** Result of thresholding (p.69) **scr4**
- scr3and3** Result of ANDing (p.55) negatives (p.63) of **scr4thr1** and **scr3thr1**.
- scr3or1** Result of ORing (p.58) negatives (p.63) of **scr4thr1** and **scr3thr1**.
- scr3or2** Result of ORing (p.58) **scr4thr1** and **scr3thr1**.
- scr3xor1** Result of XORing (p.60) **scr4thr1** and **scr3thr1**.
- t1s1** 337×238 grayscale of silhouetted tools.
- tol1** 256×256 grayscale of tool and blocks.
- tol1thr1** Result of thresholding (p.69) **tol1** at a value of 110.
- tol1crp1** Result of cropping **tol1**.
- tol1sk11** Skeleton (p.145) of **tol1**.

- `tol1shi1` Result of bitshifting (p.65) `tol1` to the right by one pixel.
- `tol1sk12` Result of bitshifting (p.65) `tol1sk11` to the right by one pixel.
- `tol1add1` Result of adding (p.43) `tol1shi1` and `tol1sk12`.
- `bld1lab1` Result of applying connected components labeling (p.114) and a distance (p.206) operator to `tol1crp1`.
- `wat1` 512×512 grayscale of digital watch.
- `wat1str1` Result of contrast stretching (p.75) `wat1`.
- `wat1ref1` Result of reflecting (p.95) `wat1str1` about a point in the center of the image.
- `wat1psh1` Result of scaling (p.90) `wat1str1` using pixel replacement to perform the sub-sampling.
- `wat1pin1` Result of scaling (p.90) `wat1str1` using pixel interpolation to perform the sub-sampling.
- `wat1exp1` Result of scaling (p.90) `wat1str1` using pixel replication to perform the zooming.
- `wat1pin2` Result of scaling (p.90) `wat1str1` using pixel interpolation to perform the zooming.
- `wdg1` 506×384 grayscale of a uniformly illuminated T-shaped part.
- `wdg1usp1` Result of mean filtering (p.150) `wdg1` with a 3×3 square kernel, and then pixel subtracting (p.45) that away from the original. See unsharp masking (p.178).
- `wdg1usp2` Result of mean filtering (p.150) `wdg1` with a 3×3 square kernel, translating (p.97) (for re-registration) and then pixel subtracting (p.45) that away from the original. See unsharp masking (p.178).
- `wdg1usp3` Result of mean filtering (p.150) `wdg1` with a 3×3 square kernel, translating (p.97) (for re-registration), dimming this result by pixel multiplication (p.48) and then pixel subtracting (p.45) away from the original. See unsharp masking (p.178).
- `wdg1usp4` Result of cropping and zooming (p.90) a section of `wdg1usp2`.
- `wdg2` 507×384 grayscale of square part with hole.
- `wdg2hst1` Intensity histogram (p.105) of `wdg2`.
- `wdg2hst2` An edited (*i.e.* enlarged to the size of `wdg2`) version of the intensity histogram (p.105) of `wdg2`.
- `wdg2xor1` Result of XORing (p.60) `wdg2` and `wdg2hst2` in a bitwise fashion.
- `wdg2or1` Result of ORing (p.58) `wdg2` with `wdg2hst2`.
- `wdg2hst3` Intensity histogram (p.105) of `wdg2` with the y-axis expanded.
- `wdg2thr2` Result of thresholding (p.69) `wdg2` at a pixel value of 120.
- `wdg2thr3` Result of thresholding (p.69) `wdg2` at a pixel value of 120. Inverse of `wdg2thr2`.
- `wdg2ded1` Result of applying dilation edge detection (p.118) with a 3×3 kernel to `wdg2thr2`.
- `wdg2dil1` Result of applying dilation (p.118) twice by a disk of diameter 11 to `wdg2thr2`.
- `wdg2ero1` Result of applying erosion (p.123) four times by a disk of diameter 11 to `wdg2thr2`.
- `wdg2sk11` Result of skeletonizing (p.145) `wdg2thr2`.
- `wdg2mat1` Medial axis transform (p.145) of `wdg2thr2`.
- `wdg2sob1` Result of applying the Sobel (p.188) operator to `wdg2`.
- `wdg2sob2` Result of thresholding (p.69) `wdg2sob1` at a pixel value of 60.
- `wdg2thn1` Result of thinning (p.137) `wdg2sob2`.
- `wdg2thn2` Result of pruning `wdg2sob2` using thinning (p.137) for five iterations.
- `wdg2can1` Result of applying the Canny edge detector (p.192) to `wdg2`.

-
- `wdg2add2` Result of adding (p.43) `wdg2can1` to `wdg2`.
- `wdg2edt1` Result of breaking up the smoothly tracked edges of `wdg2can1` using a paint program.
- `wdg2thr1` Result of thresholding (p.69) `wdg2can1` at a pixel value of 128.
- `wdg2and1` Result of inverting (p.63) `wdg2thr1` and then logical ANDing (p.55) with the original.
- `wdg2add1` Result of pixel adding (p.43) `wdg2and1` to `wdg2can1`.
- `wdg2bld1` Result of blending (p.53) `wdg2` and `wdg2can1` with the blending ratio set to $X=0.5$.
- `wdg2bld2` Result of blending (p.53) `wdg2` and `wdg2can1` with the blending ratio set to $X=0.7$.
- `wdg2str1` Result of contrast stretching (p.75) `wdg2` and `wdg2can1` and then blending (p.53) with a blending ratio of $X=0.5$.
- `wdg2bld3` Result of blending (p.53) `wdg2can1` and `wdg2` with a blending mask made from `can1thr1`.
- `wdg2hou1` Hough transform (p.214) of `wdg2can1`.
- `wdg2deh1` Hough transform (p.214) of `wdg2can1` mapped back into cartesian image coordinate space.
- `wdg2deo1` Result of pixel adding (p.43) `wdg2deh1` and `wdg2`.
- `wdg2hou3` Hough transform (p.214) of `wdg2sob2`.
- `wdg2deh2` Hough transform (p.214) of `wdg2sob2` mapped back into cartesian image coordinate space.
- `wdg2hou2` Hough transform (p.214) of `wdg2edt1`.
- `wdg2deh3` Hough transform (p.214) of `wdg2edt1` mapped back into cartesian image coordinate space.
- `wdg2cmp2` The normalized graylevel orientation image resulting from applying the Prewitt compass Operator (p.195) to `wdg2`.
- `wdg2cmp3` The color label orientation image resulting from applying the Prewitt compass Operator (p.195) to `wdg2`.
- `wdg3` 507×384 grayscale of obliquely illuminated T-shaped part.
- `wdg3hst1` Intensity histogram (p.105) of `wdg3`.
- `wdg3thr1` Result of thresholding (p.69) `wdg3` at a pixel value of 80.
- `wdg3thr2` Result of thresholding (p.69) `wdg3` at a pixel value of 120.
- `wdg3adp1` Result of adaptive thresholding (p.72) `wdg3` using the *mean-C* as a local threshold, where $C = 4$ and the neighborhood size is 7×7 .
- `wdg3adp2` Result of adaptive thresholding (p.72) `wdg3` using the *mean-C* as a local threshold, where $C = 8$ and the neighborhood size is 140×140 .
- `wdg4` 506×384 grayscale of first L-shaped part.
- `wdg4log1` Result of filtering `wdg4` with a Laplacian of Gaussian (p.173) (standard deviation 1.0).
- `wdg5` 506×384 grayscale of second L-shaped part.

Simple 3-D Objects

The *Simple 3-D Objects* Image Catalogue contains images depicting a small number (*i.e.* usually only one) of primitive three-dimensional objects against simple background surface (*i.e.* one of consistent texture and color). In some images, an illumination gradient or a specular reflection(s) exist.

bae1 256×256 range image of industrial part 1.

ba11 510×384 grayscale of shiny ball.

blb1 507×383 grayscale of shiny bulb occluding shiny cube.

blb1di11 Result of two grayscale dilations (p.118) of **blb1** using a 3×3 square structuring element.

blb1di12 Result of five grayscale dilations (p.118) of **blb1** using a 3×3 square structuring element.

blb1ero1 Result of two grayscale erosions (p.123) of **blb1** using a 3×3 square structuring element.

blb1ero2 Result of five grayscale erosions (p.123) of **blb1** using a 3×3 square structuring element.

blk1 507×383 grayscale of shiny cube.

phn1 510×384 grayscale of telephone receiver.

phn1thr1 Result of thresholding (p.69) **phn1** at a value of 100.

phn1ske1 Result of skeletonizing (p.145) **phn1thr1**.

phn1clo1 Result of closing (p.130) **phn1thr1** with a circular structuring element of size 20.

phn1ske2 Result of skeletonizing (p.145) **phn1clo1**.

phn1dst1 Result of scaling (p.48) **phn1thr1** by a factor of 6 and then distance transforming (p.206) the brightened image.

phn2 256×157 range image of telephone receiver.

ply1 510×326 grayscale of polyhedral object.

prt1 542×406 grayscale of camshaft on table.

prt2 368×493 grayscale of shiny machine part A.

prt2can1 Result of applying the Canny edge detector (p.192) (using Gaussian filter standard deviation = 1.0, upper threshold = 255, lower threshold = 1) **prt2**.

prt2can2 Result of applying the Canny edge detector (p.192) (using Gaussian filter standard deviation = 1.0, upper threshold = 150, lower threshold = 1) **prt2**.

prt2can3 Result of applying the Canny edge detector (p.192) (using Gaussian filter standard deviation = 1.0, upper threshold = 100, lower threshold = 1) **prt2**.

prt3 500×378 grayscale of machine part B perspective 1.

prt4 502×373 grayscale of machine part B perspective 2.

prt5 552×417 grayscale of machine part B perspective 3.

prt6 512×384 grayscale of machine part B perspective 4, specular reflections.

prt7 453×309 grayscale of machine part C, specular reflections.

pum1 510×384 grayscale of Puma robot model.

-
- pum1dim1 Dimmer version of pum1.
- pum1mul1 Result of pixel multiplying (p.48) pum1dim1 by a factor of 3.
- pum1mul2 Result of pixel multiplying (p.48) pum1dim1 by a factor of 5.
- pum1shi1 Result of bitshifting (p.65) pum1dim1 to the left by 1 bit.
- pum1shi2 Result of bitshifting (p.65) pum1dim1 to the left by 2 bits.
- ren1 256×256 range image of Renault industrial part.
- ren1sob1 Result of applying the Sobel edge detector (p.188) to ren1.
- ren1sob2 The normalized result of applying the Sobel edge detector (p.188) to a scaled (p.48) version (*i.e.* intensity reduction by a factor of 4) of ren1.
- ren1can1 Result of applying the Canny edge detector (p.192) (Gaussian smoothing standard deviation = 1, upper threshold = 255, lower threshold = 1) to ren1.
- ren1can2 Result of applying the Canny edge detector (p.192) (Gaussian smoothing standard deviation = 1.8, upper threshold = 255, lower threshold = 1) to ren1.
- ren1can3 Result of applying the Canny edge detector (p.192) (Gaussian smoothing standard deviation = 1.8, upper threshold = 200, lower threshold = 1) to a scaled (p.48) version (*i.e.* intensity reduction by a factor of 4) of ren1.
- ren2 128×132 range image of Renault truck part.
- ufo1 300×512 range image of machine part C perspective 1.
- ufo1rob1 Result of applying Roberts Cross Operator (p.184) to ufo1.
- ufo1rob2 Result of applying Roberts Cross Operator (p.184) to ufo1 and then thresholding (p.69) at a value of 8.
- ufo2 300×440 range image of machine part C perspective 2.
- ufo2rob1 Result of applying the Roberts Cross Operator (p.184) to ufo2.
- ufo2rob2 Result of applying the Roberts Cross Operator (p.184) to ufo2 and then thresholding (p.69) the result at a value of 20.
- ufo2sob1 Result of applying the Sobel Operator (p.188) to ufo2.
- ufo2sob2 Result of applying the Sobel Operator (p.188) to ufo2 and then thresholding (p.69) the result at a value of 150.
- ufo2noi1 Result of applying Gaussian noise (p.221) with a standard deviation of 8 to ufo2.
- ufo2rob3 Result of applying the Roberts Cross Operator (p.184) to ufo2noi1.
- ufo2rob4 Result of applying the Roberts Cross Operator (p.184) to ufo2noi1 and then thresholding (p.69) the result at a value of 20.
- ufo2sob3 Result of applying the Sobel Operator (p.188) to ufo2noi1.
- ufo2sob4 Result of applying the Sobel Operator (p.188) to ufo2noi1 and then thresholding (p.69) the result at a value of 150.
- ufo2noi2 Result of applying Gaussian noise (p.221) with a standard deviation of 15 to ufo2.
- ufo2can1 Result of applying the Canny Operator (p.192) (standard deviation 1) to ufo2noi2.
- ufo2sob5 Result of applying the Sobel Operator (p.188) to ufo2noi2.
- ufo2sob6 Result of applying the Sobel Operator (p.188) to ufo2noi2 and then thresholding (p.69) the result at a value of 150.
- ufo3 300×512 range image of machine part C perspective 3.

Texture

The *Texture* Image Catalogue contains images depicting a variety of texture patterns. Some images were chosen because they illustrate a strong single texture theme, others because they contain several competing texture patterns.

- cot1** 698×489 8-bit color of a pair of crofter's cottages. Any use of this image must include the acknowledgement: 'We thank Dr. A. Dil of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'
- crs1** 388×675 8-bit color of celtic cross at Iona Abbey, Scotland. Any use of this image must include the acknowledgement: 'We thank Dr. A. Dil of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'
- flr5** 575×392 grayscale of flowers.
- mat1** 227×392 grayscale of matchsticks.
- mat2** 217×391 grayscale of matchsticks (out of focus).
- rck1** 688×480 8-bit color of many sized rocks. Any use of this image must include the acknowledgement: 'We thank Ms. V. Temperton for the use of this image.'
- rck2** 489×694 8-bit color of large rock with moss. Any use of this image must include the acknowledgement: 'We thank Ms. V. Temperton for the use of this image.'
- rck3** 697×486 8-bit color of many small rocks (shallow focus). Any use of this image must include the acknowledgement: 'We thank Ms. V. Temperton for the use of this image.'
- wan1** 256×256 8-bit color of net lying across wood.
- wod1** 580×398 grayscale of wood grain with illumination gradient.
- wod2** 336×291 grayscale of wood grain.

Vehicles

The *Vehicles* Image Catalogue contains images depicting a variety of land, sea, air and space crafts.

- boa1** 582×364 grayscale of old boat.
- boa1aff1** Result of applying an affine transformation (p.100) to **boa1**.
- bus1** 578×389 8-bit color of buses at busy intersection.
- car1** 396×621 grayscale of street scene with cars.
- car1msk1** Selected area of **car1** set to zero using paint program.
- car1thr1** Result of thresholding (p.69) **car1msk1** in order to select only the 0 valued pixels.
- car1and1** Result of logical ANDing (p.55) **car1** and **car1thr1**.
- car1add1** Result of brightening **car1and1** and then pixel adding (p.43) it to a dimmed version of **car1msk1**.
- fei1** 640×480 8-bit color of F18. Any use of this image must include the acknowledgement: 'We thank the NASA Dryden Research Aircraft Photo Archive use of this image.'
- gli1** 700×472 8-bit color of person hangliding.
- leg1** 700×500 grayscale of Lego vehicle. Any use of this image must include the acknowledgement: 'We thank Mr. D. Howie of the Department of Artificial Intelligence, University of Edinburgh for the use of this image.'

- lun1** 640×400 8-bit color of lunar vehicle. Any use of this image must include the acknowledgement: 'We thank the NASA Dryden Research Aircraft Photo Archive use of this image.'
- shu1** 663×502 8-bit color of NASA shuttle. Any use of this image must include the acknowledgement: 'We thank the NASA Dryden Research Aircraft Photo Archive use of this image.'
- shu3** 552×401 8-bit color of NASA shuttle with drag chute.
- tra1** 589×386 grayscale of tractor.
- trk1** 512×512 grayscale of truck in rural setting. Any use of this image must include the acknowledgement: 'We thank Georgia Institute of Technology for the use of these images acquired from their on-line image database.'

Appendix J

Order form

The order form does not appear in the hardcopy.

Index

- 24-bit color images, 226,
- 8-bit color images, 226,
- AND, 55,
- Acknowledgements, 270,
- Addition, 43,
- Advanced topics, 20,
- Affine transformations, 100,
- Aliasing, 91,
- Anamorphosis, 68,
- Antilogarithm, 85,
- Bi-modal intensity distribution, 105,
- Bibliography, 267,
- Binary images, 225,
- Bit-plane, 56,
- Bit-reversal mask, 61,
- Bit-slicing, 56,
- Bitshift operators, 65,
- Blending, 53,
- Canny edge detector, 192,
- Change detection,
 - Using XOR operator, 60,
 - Using image division, 50,
 - Using image subtraction, 46,
- Circle detection,
 - Using the Hough transform, 214,
- Classification, 107,
 - Bayes', 110,
 - K-means, 110,
 - Minimum distance, 108,
 - Split and merge, 111,
- Closing, 130,
- Color images, 225,
- Color quantization, 227,
- Colormaps, 235,
- Comments in HIPRscript, 256,
- Common software implementations, 244,
- Connected components labeling, 114,
- Conservative smoothing, 161,
- Contrast stretching, 75,
- Convex hull, 142,
- Convolution, 227,
 - Use in Canny edge detector, 192,
 - Use in Compass edge detector, 195,
 - Use in Gaussian smoothing, 156,
 - Use in Laplacian of Gaussian, 173,
 - Use in Line detection, 202,
 - Use in Mean filter, 150,
 - Use in Unsharp filter, 178,
- Corner detection, 133,
- Crimmins speckle removal, 164,
- Density slicing, 70,
- Detector noise, 221,
- Difference of Gaussians filter, 176,
- Difference of boxes filter, 176,
- Dilation, 118,
 - Conditional, Example 1, 120,
 - Conditional, Example 2, 140,
- Directory structure of HIPR, 20,
- Discrete cosine transform (DCT), 212,
 - And image compression, 212,
- Distance metrics, 229,
- Distance transform, 206,
 - And the medial axis transform, 207,
 - Erosion method, 206,
 - Recursive morphological method, 206,
 - Sensitivity to noise, 207,
- Dithering, 230,
- Division, 50,
- DoG filter, 176,
- Dual lattice, 200,
- Edge Detectors, 230,
- Edge detectors,
 - As a feature detector, 183,
 - Canny, 192,
 - Compass, 195,
 - Laplacian of Gaussian, 173,
 - Marr, 199,
 - Roberts Cross, 184,
 - Shape of kernel, 186,
 - Sobel, 188,
 - Sub-pixel precision, 200,
 - Zero crossing, 199,
- Editing HTML and L^AT_EX directly, 38,
- Enhancement,
 - Unsharp filter, 178,
 - Using LoG filter, 175,
 - Using contrast stretching, 75,
 - Using histogram equalization, 78,
- Erosion, 123,
- Exponential operator, 85,
- External image viewers and Netscape, 30,
- Fast fourier transform (FFT), 210,
- Feature vector, 107,
- Filename conventions, 23,
- Fourier transform, 209,
 - And frequency filtering, 211,

- And the frequency domain, 209,
- Filter response, 158,
- Used for frequency filtering, 167,
- Frequency domain, 232,
 - And the Fourier transform, 209,
- Frequency filters, 167,
 - Band pass, 167,
 - Deconvolution, 170,
 - Gaussian as a lowpass filter, 158,
 - High pass, 167,
 - Low pass, 167,
 - Mean filter, 151,
 - Ringings, 167,
 - Use in image reconstruction, 170,
 - Use in pattern matching, 171,
- Gamma correction, 87,
- Gaussian filter,
 - Frequency response, 158,
- Gaussian noise, 222,
- Gaussian smoothing, 156,
- Geodesic distance, 115,
- Grayscale images, 232,
- Guide to contents, 8,
- HIPRscript syntax, 254,
- HIPRscript, 253,
- HIPS, 251,
- Hardcopy version of HIPR, 26,
- Histogram equalization, 78,
- Histogram modeling, 78,
- Histogram specification, 80,
- Histogram, intensity, 105,
- Hit-and-miss transform, 133,
- Hough transform, 214,
- How to use HIPR, 11,
- Idempotence, 233,
- Image editors, 233,
- Image enhancement, 68,
- Image formats, 22,
 - Changing the default, 36,
- Image library, 281,
- Image segmentation, 69,
- Image sharpening, 178,
- Installation guide, 28,
- Inverse fourier transform, 209,
- Invert, 63,
- Isotropic filters,
 - Laplacian of Gaussian, 176,
- Isotropic operators, 233,
- Kernel, 233,
- Khoros, 246,
- LUT-transformation, 68,
- LaTeX,
 - In practice, 26,
- Laplacian filter, 173,
 - Digital filtering, 148,
- Laplacian of Gaussian filter, 173,
- Lightfield,
 - Used with image division, 50,
 - Used with image subtraction, 45,
- Line Detection, 202,
- Line detection,
 - As a feature detector, 183,
 - Using the Hough transform, 214,
- Local enhancement, 80,
- Local information, 40,
 - Customization, 37,
- Logarithm, 82,
- Logical operators, 234,
- Look-up tables, 235,
- Making changes to HIPR, 34,
- Mapping function, 68,
- Marr edge detector, 199,
- Marr filter, 173,
- Masking, 235,
- Mathematical morphology, 236,
- Matlab, 249,
- Maximum entropy filtering, 170,
- Mean filter, 150,
 - Digital filtering, 148,
- Medial axis transform, 145,
- Median filter, 153,
- Median,
 - Digital filtering, 149,
- Morphological filters, 132,
- Multi-spectral images, 237,
- Multiplication, 48,
- NAND, 55,
- NOR, 58,
- NOT, 63,
- Noise generation, 221,
- Noise reduction,
 - Conservative smoothing, 161,
 - Crimmins speckle removal, 164,
 - Gaussian filtering, 156,
 - Mean filtering, 150,
 - Median filtering, 153,
- Non-linear filtering, 237,
- Non-maximal suppression,
 - Use in Canny operator, 192,
- Normalization, 75,
- Nyquist frequency, 91,
- OR, 58,
- Opening, 127,
- Optical flow, 46,
- Order form, 314,
- Pixel connectivity, 238,
- Pixel format, 239,
- Pixel values, 239,
- Pixels, 238,
- Plane transformation: Euclidean, Similarity,
 - Affine, Projective, 101,
- Polarity, 225,
- Prewitt edge detectors,
 - Compass operator, 195,

- Gradient operator, 190,
- Primary colors, 240,
- Pruning, 137,
- RGB and colorspaces, 240,
- Raise to power, 85,
- Range images,
 - Depth discontinuities, 185,
 - Processing using Roberts Cross, 185,
 - Processing using Sobel, 189,
- Ratioing, 50,
- Reflect, 95,
- Resolution, 91,
- Roberts Cross, 184,
- Rotate, 93,
- Salt and pepper noise, 223,
- Sampling theory, 91,
- Saturation, 241,
- Scale, geometric, 90,
- Scaling, graylevel, 48,
- Skeleton by zone of influence (SKIZ), 143,
- Skeletonization, 145,
- Smoothing, 150,
- Sobel edge detector, 188,
- Spatial domain, 240,
- Speckle, 221,
- Structuring elements, 241,
- Subsampling, 90,
- Subtraction, 45,
- Support, 19,
- Symmetry analysis, 96,
- Thickening, 142,
- Thinning, 137,
- Threshold averaging, 151,
- Thresholding, 69,
 - Adaptive, 72,
 - Chow and Kanenko, 72,
 - Local, 72,
- Tracking,
 - Edge tracking in line detection, 202,
 - Edge tracking in the Canny operator, 192,
- Transfer function, 79,
- Translate, 97,
- Truth-table,
 - AND/NAND, 55,
 - NOT, 63,
 - OR/NOR, 58,
 - XOR/XNOR, 60,
- Two's complement, 65,
- Union of images, 58,
- Unsharp filter, 178,
- Visilog, 244,
- Voronoi diagram, 143,
- Welcome to HIPR!, 6,
- What is HIPR?, 7,
- Wiener filtering, 170,
- Wrapping, 241,
- XNOR, 60,
- XOR, 60,
- Zero crossing detector, 199,
 - Effect of Gaussian smoothing, 158,
- hiprgen.pl, 253,