



P V - W A V E 7 . 5[®]

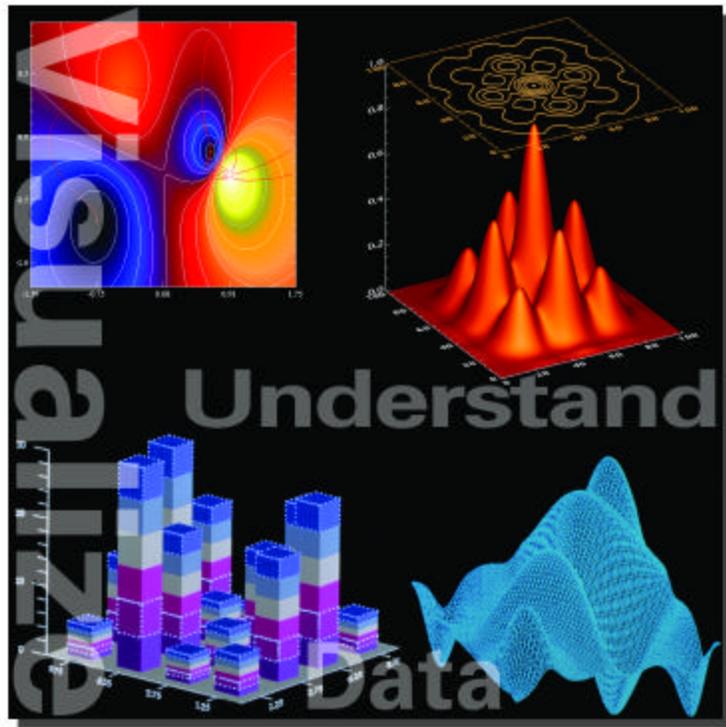


Image Processing Toolkit User's Guide

HELPING CUSTOMERS **SOLVE** COMPLEX PROBLEMS

Visual Numerics, Inc.

Visual Numerics, Inc.
2500 Wilcrest Drive
Suite 200
Houston, Texas 77042-2579
United States of America
713-784-3131
800-222-4675
(FAX) 713-781-9260
<http://www.vni.com>
e-mail: info@boulder.vni.com

Visual Numerics, Inc.
7/F, #510, Sect. 5
Chung Hsiao E. Rd.
Taipei, Taiwan 110 ROC
+886-2-727-2255
(FAX) +886-2-727-6798
e-mail: info@vni.com.tw

Visual Numerics S.A. de C.V.
Cerrada de Berna 3, Tercer Piso
Col. Juarez
Mexico, D.F. C.P. 06600
Mexico

Visual Numerics, Inc. (France) S.A.R.L.
Tour Europe
33 place des Corolles
Cedex 07
92049 PARIS LA DEFENSE
FRANCE
+33-1-46-93-94-20
(FAX) +33-1-46-93-94-39
e-mail: info@vni-paris.fr

Visual Numerics International GmbH
Zettachring 10
D-70567 Stuttgart
GERMANY
+49-711-13287-0
(FAX) +49-711-13287-99
e-mail: info@visual-numerics.de

Visual Numerics, Inc., Korea
Rm. 801, Hanshin Bldg.
136-1, Mapo-dong, Mapo-gu
Seoul 121-050
Korea

Visual Numerics International, Ltd.
Suite 1
Centennial Court
East Hampstead Road
Bracknell, Berkshire
RG 12 1 YQ
UNITED KINGDOM
+01-344-458-700
(FAX) +01-344-458-748
e-mail: info@vniuk.co.uk

Visual Numerics Japan, Inc.
Gobancho Hikari Building, 4th Floor
14 Gobancho
Chiyoda-Ku, Tokyo, 102
JAPAN
+81-3-5211-7760
(FAX) +81-3-5211-7769
e-mail: vda-spirt@vnij.co.jp

© 1990-2001 by Visual Numerics, Inc. An unpublished work. All rights reserved. Printed in the USA. 2001

Information contained in this documentation is subject to change without notice.

IMSL, PV-WAVE, Visual Numerics and PV-WAVE Advantage are either trademarks or registered trademarks of Visual Numerics, Inc. in the United States and other countries.

The following are trademarks or registered trademarks of their respective owners: Microsoft, Windows, Windows 95, Windows NT, Fortran PowerStation, Excel, Microsoft Access, FoxPro, Visual C, Visual C++ — Microsoft Corporation; Motif — The Open Systems Foundation, Inc.; PostScript — Adobe Systems, Inc.; UNIX — X/Open Company, Limited; X Window System, X11 — Massachusetts Institute of Technology; RISC System/6000 and IBM — International Business Machines Corporation; Java, Sun — Sun Microsystems, Inc.; HPGL and PCL — Hewlett Packard Corporation; DEC, VAX, VMS, OpenVMS — Compaq Computer Corporation; Tektronix 4510 Rasterizer — Tektronix, Inc.; IRIX, TIFF — Silicon Graphics, Inc.; ORACLE — Oracle Corporation; SPARCstation — SPARC International, licensed exclusively to Sun Microsystems, Inc.; SYBASE — Sybase, Inc.; HyperHelp — Bristol Technology, Inc.; dBase — Borland International, Inc.; MIFF — E.I. du Pont de Nemours and Company; JPEG — Independent JPEG Group; PNG — Aladdin Enterprises; XWD — X Consortium. Other product names and companies mentioned herein may be the trademarks of their respective owners.

IMPORTANT NOTICE: Use of this document is subject to the terms and conditions of a Visual Numerics Software License Agreement, including, without limitation, the Limited Warranty and Limitation of Liability. If you do not accept the terms of the license agreement, you may not use this documentation and should promptly return the product for a full refund. Do not make illegal copies of this documentation. No part of this documentation may be stored in a retrieval system, reproduced or transmitted in any form or by any means without the express written consent of Visual Numerics, unless expressly permitted by applicable law.

Table of Contents

Preface [v](#)

Intended Audience [v](#)

Typographical Conventions [vi](#)

Customer Support [vi](#)

Chapter 1: Getting Started [1](#)

Starting PV-WAVE [1](#)

Initializing the Image Processing Toolkit [3](#)

Starting the Image Processing Navigator [4](#)

Accessing Online Help for the Image Processing Toolkit [5](#)

Stopping the Image Processing Toolkit [6](#)

Image Processing: A Brief Overview [6](#)

Chapter 2: Reference [17](#)

BLEND Function [17](#)

CANNY Function [19](#)

CENTROID Function [20](#)

DCT Function [22](#)

DENSITY_SLICE Function [24](#)

DIST_MAP Function [26](#)

ENTROPY Function [31](#)

FILT_DWMTM Function [32](#)

FILT_FREQ Function [35](#)

FILT_MMSE Function [39](#)

FILT_NONLIN Function [42](#)

FILT_NOTCH Function [51](#)
FILT_SMOOTH Function [56](#)
FILT_WIENER Function [58](#)
GAUSS_KERNEL Function [62](#)
GLCM Function [63](#)
GLCM_STATS Function [65](#)
GLRL Function [67](#)
GLRL_STATS Function [69](#)
HAAR Function [71](#)
HIST_STATS Function [74](#)
HIT_MISS Function [76](#)
HOUGH Function [78](#)
IPALOG Function [83](#)
IPCLASSIFY Function [84](#)
IPCLUSTER Function [88](#)
IPCOLOR_24_8 Function [92](#)
IPCONVOL Function [95](#)
IPCORRELATE Function [99](#)
IPCREATE_FILTER Function [102](#)
IPCT Function [103](#)
IPHISTOGRAM Function [105](#)
IPLINEAR_GRAY Function [107](#)
IPMATH Function [109](#)
IPQMFDESIGN Function [113](#)
IPREAD_FILTER Function [115](#)
IPSCALE Function [117](#)
IPSPECTRUM Function [120](#)

IPSTATS Function	124
IPWAVELET Function	126
IPWIN Function	129
IPWRITE_FILTER Function	133
IS_GRAY_CMAP Function	135
KURTOSIS Function	137
MAJOR_AXIS Function	138
MODE Function	139
MOMENT2D Function	140
MORPH_CLOSE Function	142
MORPH_OPEN Function	144
MORPH_OUTLINE Function	146
NOISE_GEN Function	149
NOISE_IMPULSE Function	156
NOISE_PERIODIC Function	158
NOISE_RAYLEIGH Function	159
PAD_IMAGE Function	161
PCT Function	163
PERIMETER Function	165
POLAR_FFT Function	166
RADON Function	169
RANGE Function	171
REGION_COUNT Function	172
REGION_FIND Function	173
REGION_GROW Function	175
REGION_MERGE Function	177
REGION_SPLIT Function	180

REGION_STATS Function 182

SHIFT_EDGE Function 185

SKELETONIZE Function 188

SKEWNESS Function 190

SLANT Function 191

THRESH_ADAP Function 193

THRESHOLD Function 196

TOP_HAT Function 198

UNIFORMITY Function 200

***Appendix A: Bibliography* A-1**

***Image Processing Index* 1**

Preface

This manual explains how to use PV-WAVE:Image Processing Toolkit; it contains the following parts:

- **Preface** — Describes the contents of this manual, lists the typographical conventions used, and explains how to obtain customer support.
- **Chapter 1, *Getting Started*** — Introduces the PV-WAVE:Image Processing Toolkit and explains the techniques and methods used to process images.
- **Chapter 2, *Reference*** — Detailed descriptions of the PV-WAVE:Image Processing functions.
- **Appendix A, *Bibliography*** — A complete bibliography of technical literature cited in this manual.
- ***Image Processing Index***

Intended Audience

This guide is for the experienced PV-WAVE user. Only minimal information on using PV-WAVE is covered here. Some knowledge of image processing techniques is helpful.

Typographical Conventions

The following typographical conventions are used in this guide:

- `Courier` font is used for program code examples, the names of system files, and system messages.
- Names of routines are all uppercase letters; keywords are shown in initial caps and italic font; and parameters are shown in all lowercase and italic font.

Customer Support

If you have problems installing, unlocking, or running your software, contact Visual Numerics Technical Support by calling:

Office Location	Phone Number
Corporate Headquarters Houston, Texas	713-784-3131
Boulder, Colorado	303-939-8920
France	+33-1-46-93-94-20
Germany	+49-711-13287-0
Japan	+81-3-5211-7760
Korea	+82-2-3273-2633
Mexico	+52-5-514-9730
Taiwan	+886-2-727-2255
United Kingdom	+44-1-344-458-700

Users outside the U.S., France, Germany, Japan, Korea, Mexico, Taiwan, and the U.K. can contact their local agents.

Please be prepared to provide the following information when you call for consultation during Visual Numerics business hours:

- Your license number, a six-digit number that can be found on the packing slip accompanying this order. (If you are evaluating the software, just mention that you are from an evaluation site.)

- The name and version number of the product. For example, PV-WAVE 7.0.
- The type of system on which the software is being run. For example, SPARC-station, IBM RS/6000, HP 9000 Series 700.
- The operating system and version number. For example, HP-UX 10.2 or IRIX 6.5.
- A detailed description of the problem.

FAX and E-mail Inquiries

Contact Visual Numerics Technical Support staff by sending a FAX to:

Office Location	FAX Number
Corporate Headquarters	713-781-9260
Boulder, Colorado	303-245-5301
France	+33-1-46-93-94-39
Germany	+49-711-13287-99
Japan	+81-3-5211-7769
Korea	+82-2-3273-2634
Mexico	+52-5-514-4873
Taiwan	+886-2-727-6798
United Kingdom	+44-1-344-458-748

or by sending E-mail to:

Office Location	E-mail Address
Boulder, Colorado	support@boulder.vni.com
France	support@vni-paris.fr
Germany	support@visual-numerics.de
Japan	vda-sprt@vnij.co.jp
Korea	support@vni.co.kr
Taiwan	support@vni.com.tw
United Kingdom	support@vniuk.co.uk

Electronic Services

Service	Address
General e-mail	<code>info@boulder.vni.com</code>
Support e-mail	<code>support@boulder.vni.com</code>
World Wide Web	<code>http://www.vni.com</code>
Anonymous FTP	<code>ftp.boulder.vni.com</code>
FTP Using URL	<code>ftp://ftp.boulder.vni.com/VNI/</code>
PV-WAVE	
Mailing List:	<code>Majordomo@boulder.vni.com</code>
To subscribe include:	<code>subscribe pv-wave YourEmailAddress</code>
To post messages	<code>pv-wave@boulder.vni.com</code>

Getting Started

Image processing is widely used in engineering and scientific research and development for representing, transforming, and manipulating images and the information they contain. The functions in the Image Processing Toolkit are designed for easy use, while providing many options for solving difficult problems.

Purpose of this Chapter

The purpose of this chapter is to establish terminology and provide a brief overview of the functionality of the PV-WAVE:Image Processing Toolkit. It is assumed that you have a basic working knowledge of image processing and filtering.

Starting PV-WAVE

If PV-WAVE isn't already installed on your system, install it first. You also need the following options installed in order to run the PV-WAVE:Image Processing Toolkit: PV-WAVE IMSL Mathematics, PV-WAVE IMSL Statistics, and of course, the PV-WAVE:Image Processing Toolkit. Once PV-WAVE and the options are installed, you're ready to begin.

Starting PV-WAVE Under Windows NT

Step 1 Double-click the PV-WAVE Home Window icon in the PV-WAVE Program Group.

— OR —

Double-click the PV-WAVE Console icon in the PV-WAVE Program Group.

After a brief pause, the corresponding PV-WAVE window appears displaying the prompt:

```
WAVE>
```

When you see this prompt, PV-WAVE is ready for you to enter commands or initialize the Image Processing Toolkit.

Step 2 Go to the section on *Initializing the Image Processing Toolkit*, and perform the initialization.

Starting PV-WAVE Under Windows 95

Step 1 Using the **Start** button, select **Start=>Programs=>PV-WAVE 6.10=>PV-WAVE Home Window**.

— OR —

Select **Start=>Programs=>PV-WAVE 6.10=>PV-WAVE Console**.

After a brief pause, the corresponding PV-WAVE window appears displaying the prompt:

```
WAVE>
```

When you see this prompt, PV-WAVE is ready for you to enter commands or initialize the Image Processing Toolkit.

Step 2 Go to the section on *Initializing the Image Processing Toolkit*, and perform the initialization.

Starting PV-WAVE Under UNIX

Step 1 Start PV-WAVE, by typing the following command at your UNIX system prompt:

```
wave
```

The command line prompt, WAVE> appears in the terminal.

When you see this prompt, PV-WAVE is ready for you to enter commands or initialize the Image Processing Toolkit.

Step 2 Go to the section on *Initializing the Image Processing Toolkit*, and perform the initialization.

Initializing the Image Processing Toolkit

Now that you have PV-WAVE and the required options installed and started, the remaining commands for initializing the toolkit and starting the Image Processing Navigator are the same no matter what platform you're using.

Step 1 At the WAVE> prompt, enter the following command to load and initialize the PV-WAVE:Image Processing Toolkit:

```
WAVE> @ip_startup
```

Once you see the following series of messages, you are ready to use the PV-WAVE:Image Processing Toolkit.

```
PV-WAVE:Image Processing is initialized.
```

```
Enter "IPNAVIGATOR" at the WAVE> prompt to start the PV-WAVE:Image  
Processing Navigator.
```

Step 2 You are now ready to use the PV-WAVE:Image Processing Toolkit to begin processing your images.

Starting the Image Processing Navigator

The Image Processing Navigator provides easy access to image file I/O, as well as these Image Processing visual data analysis (VDA) tools:

WzIPImage — The Image Tool is used to display a variable containing image data. The variable can be a 2D array containing 8-bit image data or a 3D variable containing 24-bit RGB image data.

WzIPPlot — The Plot Tool tool is used to plot one or more variables.

WzIPHistogram — The Histogram Tool is used to visualize quantitative trends in large amounts of 1D, 2D, or 3D data.

WzIPSurface — The Surface Tool is used to display one 2D variable containing surface data.

WzIPContour — The Contour Tool is used to display one 2D variable containing contour data. The Contour Tool also lets you display an image of your data underneath the contour plot.

WzIPColorEdit — The Color Tool is used to select and set the image and plot colors for displaying data.

To start the Image Processing Navigator, enter the following command at the **PV-WAVE** prompt:

```
WAVE> ipnavigator
```

Once the Image Processing Navigator is running, you can use the Image Processing VDA tools exclusively to perform your image processing, or you can use a combination of Image Processing VDA tools and the **PV-WAVE** command line to accomplish your goals.

Accessing Online Help for the Image Processing Toolkit

Online help is provided on all platforms supported for image processing. The online help comes in two forms, both of which are provided: as manuals online (this User's Guide), and context sensitive help for the Image Processing VDA Tools.

The Manuals Online System

This User's Guide is accessible online from the command line by typing `HELP` at the `WAVE>` prompt. This initiates the main help files for `PV-WAVE`. When the contents page is active, select the "Optional `PV-WAVE` Products" link, then select the "`PV-WAVE:Image Processing Toolkit`" link.

In addition to the online manual documentation, there is a similar context-sensitive online document provided. This context-sensitive help is accessible from the Navigator, or any of the Image Processing VDA Tool windows by doing the following:

- Select **Help=>On PV-WAVE...** from the Image Processing Navigator menu bar.

This brings up the online help contents page, from which you can select the `PV-WAVE:Image Processing Toolkit`.

Online Help for the Image Processing Navigator VDA Tools

To access the context-sensitive online help for the Image Processing VDA Tools and the Image Processing Navigator, do the following:

- Select **Help=>On Window...** from the Image Processing Navigator or VDA Tool window menu bar.

This initiates the contents page for the window from which Help was called. For another way to access context-sensitive Help do the following:

- Select the **Help** button in any active dialog to activate the context-sensitive help.

Stopping the Image Processing Toolkit

If the PV-WAVE:Image Processing Toolkit is loaded and you want to exit it, perform the following procedure:

- At the WAVE> prompt, enter the following command to unload the PV-WAVE:Image Processing Toolkit.

```
WAVE> @ip_unload
```

Unloading returns your system to the state it was in before using the Image Processing Toolkit by doing the following three things:

- It unloads the PV-WAVE:Image Processing Toolkit functions from memory.
- It returns the Image Processing Toolkit license to the license manager, freeing the license up for others to use.
- And it deletes all common variables in IP_COMMON.

Image Processing: A Brief Overview

Many techniques are typically used by scientists and engineers to alter or process digital images, including: point operations, filtering, and image transforms. The fields of computer vision and pattern recognition often overlap with digital image processing (IP) and they use the following processing techniques: segmentation, classification, and difference analysis. All of these operations are built upon a basic set of IP routines, which are discussed in this chapter.

Point Operations

Point operations are a method of image processing in which each pixel in the output image is only dependent upon the corresponding pixel in the input image. In general, point operations are mathematical and/or logical operations performed on a single image, or between two images of equal size on a point-by-point basis.

Algebraic and Logical Operations

Algebraic, or mathematical operations used in image processing include addition, subtraction, multiplication, and, sometimes, a ratio of two images. Logical operations such as AND, OR and exclusive-OR are also used to process images. These mathematical and logical operations are performed between two images of equal size, or between an image and a scalar value.

Differences and similarities between two images can be enhanced and examined using algebraic operations. For instance, multiplying every pixel value in an image by two can increase the overall contrast; image subtraction, on the other hand, is a simple way to reveal image differences.

Algebraic and logical operations such as these can be performed using the `IPMATH` function in the Image Processing Toolkit.

Algebraic operations are also used for dynamic range scaling or shifting; however, you must keep the pixel-value range in mind when performing mathematical image operations because of the possibility of negative values appearing in the result. For example, when one image is subtracted from another, negative pixel values may result. Since the display color of a negative value is typically undefined, the negative image values in the result must be either clipped or scaled. Two images multiplied together can also result in values for which no corresponding color exists in the colormap.

Thresholding

Thresholding is another point operation method, and binary thresholding is a specific type of thresholding. To perform binary thresholding, you first select a range of pixel values and a logical operator to form the conditional equation. Any pixel values that evaluate to “true” within the logical operation are set to white (or black) and all other pixel values are set to black (or white).

Multilevel thresholding is similar to binary thresholding except that many conditional equations and output levels are defined for a single input image. For example, you may wish to set all pixels between 10 and 20 to black, all pixels between 50 and 100 to medium gray, all pixels between 120 and 220 to white, and leave all other pixels unchanged.

The routines provided in the Image Processing Toolkit to perform thresholding operations are the `DENSITY_SLICE`, `THRESH_ADAP`, and `THRESHOLD` functions.

Histogram Operations

Another point operation used frequently in image processing is the histogram operation. A histogram is a plot of the number of pixels versus pixel values in an image. It's possible to improve the overall appearance of an image by modifying its histogram. Two commonly used techniques to accomplish this are: histogram equalization, and histogram stretching. These operations both serve to modify the overall contrast and brightness of an image.

Two functions are used to perform these operations in the Image Processing Toolkit: the HIST_STATS, and IPHISTOGRAM functions; in addition to the HIST_EQUAL function found in PV-WAVE.

Filtering

Besides point operations, filtering is another commonly used image processing operation. Basically, filtering is used either to remove unwanted information in an image, or to enhance the information already present. There are several categories of filters, such as linear filters, nonlinear filters, and adaptive filters.

The Image Processing Toolkit routines that perform filtering operations are listed in the following table along with the filter category to which each belongs. For your convenience, some associated PV-WAVE filter routines are also listed.

Filter Category	Image Processing Toolkit Routines	PV-WAVE Routines
Linear Filter - Spatial Domain	CANNY, FILT_SMOOTH, GAUSS_KERNEL, IPCONVOL, IPCREATE_FILTER, IPREAD_FILTER, IPWRITE_FILTER	ROBERTS, SOBEL
Linear Filter - Frequency Domain	FILT_FREQ, FILT_NOTCH, FILT_WIENER	
Nonlinear Filter	FILT_NONLIN	
Adaptive Filter	FILT_DWMTM, FILT_MMSE	

Linear Filters

Linear filters are defined by a filter kernel, which is itself just a small image. Filter kernels, also called windows, are usually 3-by-3, 5-by-5, or 7-by-7 pixels, which contain values that mathematically define the characteristics of the linear transform. Two broad categories of linear filtering operators for digital IP are the spatial and frequency domain operators. The spatial domain refers to the original image plane itself, whereas the frequency domain representation is a fast Fourier transform (FFT) of that image.

The linear filters category can be divided into four subcategories: lowpass, highpass, bandpass, and bandstop filters. Lowpass filters are typically used to smooth or blur images. Highpass filters, on the other hand, enhance image edges by eliminating low-frequency components. Bandstop filters are typically employed to remove periodic noise; and bandpass filters are useful in image enhancement.

Spatial Domain

Linear filtering in the spatial domain is performed by convolution between the image and a filter kernel. Convolution involves passing the filter kernel over the entire input image. Pixel values in the output image are defined at the corresponding location in the input image under the center pixel in the filter kernel. Output values for the edges of the image can be ambiguous when part of the kernel hangs off the image edge. Typical methods for dealing with undefined output pixels are to simply copy the edge pixels from the input image directly to the output image or to extend the boundary of the input image by the size of the filter kernel before convolution is performed.

The PV-WAVE:Image Processing Toolkit comes with many spatial filter kernels which are located in the following directory:

(UNIX) `ip-1_0/data/kernel/*.ker`

(Windows) `ip-1_0\data\kernel*.ker`

Typical spatial filters include the Gaussian filter, gradient masks, highpass spatial filters, Laplacian, lowpass spatial filters, and the Roberts and Sobel filters. See the table for the list of Image Processing Toolkit functions and associated PV-WAVE routines for linear filtering in the spatial domain.

Frequency Domain

Spatial frequency filters are most often used for image restoration and enhancement. Restoration algorithms remove degradation or noise that has corrupted the image. The Wiener filter and any circularly symmetric filters are good examples of this filtering technique.

Filtering in the frequency domain is performed by multiplying the frequency-domain image with a frequency-domain filter. The product of the image and the filter is then transformed back into the spatial domain by performing an inverse FFT. This technique of using the spatial domain is often used for filters with large kernels.

See the table for the list of Image Processing Toolkit functions and associated PV-WAVE routines for linear filtering in the spatial domain.

Nonlinear Filters

Nonlinear filters are used for removal of so-called salt-and-pepper noise and Gaussian noise, as well as edge detection in an image.

Nonlinear filters operate by passing a small window over an image and computing an output image pixel based on a given nonlinear function of the input image pixels under that window. Typical window sizes are 3-by-3, 5-by-5 or 7-by-7 pixels-squared.

The FILT_NONLIN function in the Image Processing Toolkit offers the following nine nonlinear filters via the use of function keywords:

FILT_NONLIN Function

Nonlinear Filter	Function Keyword	Typical Image Processing Usage
Alpha-Trimmed Mean	<i>Atmeanf</i>	Removing salt-and-pepper, Gaussian noise
Contra-Harmonic Mean	<i>Chmeanf</i>	Removing Gaussian noise while preserving edge features
Geometric Mean	<i>Gmeanf</i>	Removing Gaussian noise
Maximum	<i>Maxf</i>	Removing outlying low or negative values
Minimum	<i>Minf</i>	Removing outlying high values
Mode	<i>Modf</i>	Removing noise
Range	<i>Rangef</i>	Edge-detection
Rank	<i>Rankf</i>	Removing salt-and-pepper noise
Yp Mean	<i>Ypmeanf</i>	Removing Gaussian noise while preserving edge features

The characteristics of the image information can influence the relative success of nonlinear filters which are not adaptive. Because adaptive filters alter their filtering characteristics based on local image content, they often perform better than their non-adaptive counterparts.

Adaptive Filters

Adaptive filters are particularly useful for noise reduction. By contrast, non-adaptive filters require *a priori* knowledge of the image noise characteristics to achieve similar optimal results.

The FILT_DWMTM and FILT_MMSE functions provide the adaptive filtering operations in the PV-WAVE:Image Processing Toolkit.

Morphological Image Processing

Image preprocessing for pattern recognition and image analysis applications involve morphological operations. Morphological image processing routines alter or in some way act upon shapes within an image. Erosion and dilation are both fundamental morphological operators that function to erode or dilate, respectively, objects in an image.

The morphological opening operation is erosion followed by dilation, whereas the morphological closing operation is just the opposite, dilation followed by erosion. Opening is a useful processing tool for smoothing contours, eliminating narrow extensions, and breaking thin links. Closing, on the other hand, is used to smooth contours, to link narrow regions, and to fill small gaps or holes.

In addition to the opening and closing morphological operations, the hit-or-miss transform is another morphological operation used primarily for shape definition. This transform is particularly useful in pattern recognition.

The PV-WAVE:Image Processing Toolkit includes seven morphological routines to perform morphological image processing, which complements the ERODE and DILATE routines already provided in PV-WAVE. The seven Image Processing Toolkit routines are: DIST_MAP, HIT_MISS, MORPH_CLOSE, MORPH_OPEN, MORPH_OUTLINE, SKELETONIZE, TOP_HAT.

Mensuration

For digital images, mensuration refers to the quantification, or measurement of object features within an image. Mensuration is useful in classification and object recognition, and is the image processing technique widely employed in the field of medicine for tumor identification and surgical planning. Common measures

include computing the area, average graylevel, standard deviation, centroid, circularity, and perimeter of a single object.

Another measure is the principal axes of a region. An object's principle axes are computed from the eigenvectors of the covariance matrix. This is known as the Hotelling transform. Because the principal axis representation of an object is insensitive to rotation, it is often used for target recognition and tracking.

Functions in the Image Processing Toolkit providing measurement operations are CENTROID, IPSTATS, MAJOR_AXIS, MODE, MOMENT2D, PERIMETER, and RANGE.

Representation and Description

Image representation and description operations are used to preprocess images for pattern recognition and classification.

Texture

Texture analysis can be extremely important in describing regions in an image. Quantitative texture descriptions, such as smoothness, coarseness, and regularity are often used in image classification and pattern recognition. Texture values are typically based on regional statistical properties such as the moments of the regional histogram. Routines providing these types of statistical texture measurements are the GLCM, GLCM_STATS, GLRL, GLRL_STATS, and the HIST_STATS function.

Spectral analysis can also be used to describe texture. In a textural scale ranging from rough to smooth, for instance, a region with high spatial frequencies may be defined as rough, while a low spatial frequency area would be smooth. The POLAR_FFT function is used to implement spectral texture analysis in the Image Processing Toolkit.

Correlation

The correlation between two images or between an image and a template is often used in template or prototype matching for pattern recognition. Correlation can also be used to facilitate automatic registration between images. The IPCORRELATE function performs correlation between an image and a template.

Image Transforms

There are numerous transforms that can be applied to any image. The two most common transforms are the fast Fourier transform (FFT) and its inverse. The FFT converts an image from the spatial domain to the spatial frequency domain. Many other transforms exist, however, and are useful for various applications.

The discrete cosine transform (DCT) is used in image compression. The Hough transform is useful in contour linking and identification of geometric shapes and lines in an image. It maps data from a Cartesian coordinate space into a polar parameter space. The Slant transform uses sawtooth waveforms as a basis set and reveals connectivity.

The principal components transform (PCT), also referred to as the Hotelling transform or the Karhunen-Loève transform, is useful for image compression and de-correlation and is widely used in remote sensing applications. The PCT is applied to the covariance matrix of the different spectral bands of a remote sensing image. Its output is an image that has a minimum amount of correlation. This maximum variance image combines most of the information present in the total spectral bands of the original image.

Image transforms provided in the Image Processing Toolkit include the DCT, HAAR, HOUGH, IPCT, IPWAVELET, PCT, RADON, and SLANT functions; in addition to the FFT function in PV-WAVE, and the FFTCOMP function in PV-WAVE IMSL Mathematics.

Geometric Transforms

Geometric transforms such as image rotation, scaling, and warping are also important in many applications. In particular multi-modal data can be registered to a common coordinate system through the use of geometric transforms. Geometric transforms modify the spatial relationships between image pixels. These functions are sometimes referred to as “rubber-sheet transformations” because of their likeness to stretching an image that has been transferred onto a sheet of rubber.

When a geometric transform is applied to an image, pixels in the input image don't always map directly to a position in the output image. For this reason, some form of graylevel interpolation is necessary to determine the value at each pixel in the output image. Several methods of interpolation exist, among them are the bilinear and nearest neighbor methods.

In the nearest neighbor method, values in the output image are taken to be the value of the closest matching pixel in the input image. The choice of interpolation method is dependent on the application for which the image is being processed. For

example, nearest neighbor interpolation is often used if the image is to be classified.

In the Image Processing Toolkit, the IPSCALE function performs geometric operations. There is also interactive image warping capability in the Image Tool (WzIPIImage), as well as the following geometric operation routines found in PV-WAVE: CONGRID, REBIN, ROT, and ROTATE.

Color Image Processing

There are two general categories of color image processing: full color and pseudo-color processing. Full color images are acquired with a sensor that detects the full visible range, such as a television camera. Pseudo-color images are artificially formed from images that are representative of spectral information outside the visible spectrum.

Color Models

The most common color models are the red, green, blue (RGB) model; and the hue, saturation, and value (HSV) model; whereas color printers use the cyan, magenta, yellow (CMY) model. The choice of a color model obviously depends on the origin of the image information, the sensor used to obtain the information, and the desired results of the image processing application. Additional information about color models can be found in the *PV-WAVE User's Guide*.

Density Slicing

Intensity or density slicing is an example of pseudo-color image processing. In density slicing, planes parallel to the zero amplitude plane of the image are used to slice the pixel amplitude levels into a discrete set of ranges, smaller than the original dynamic range of the pixel values. The plane locations are chosen by the user and between each plane, the pixel values are mapped into a different color. In the Image Processing Toolkit, the DENSITY_SLICE function performs density slicing on an image.

Classification and Segmentation

Segmentation and classification are closely related, and sometimes confused. Segmentation is used to identify regions of common characteristics in an image. For example, a medical image can be segmented into areas of soft tissue, bone, and air; or a satellite image can be segmented into regions of vegetation, ground clutter, and water. Thresholding techniques, edge detection, and region growing are also commonly used for segmentation analysis.

Classification is a step beyond segmentation in which particular substances or objects are identified within an image and then segmented. Classification usually involves determining the number of separate classes contained within the image. This can be performed by the user, in which case it is termed supervised classification; or it can be performed automatically, also known as unsupervised classification.

The Image Processing Toolkit routines for classification and segmentation operations include IPCLASSIFY, IPCLUSTER, REGION_GROW, REGION_MERGE, REGION_SPLIT, THRESH_ADAP, and THRESHOLD.

Reference

This chapter describes each of the procedures and functions of the PV-WAVE:Image Processing Toolkit. These descriptions are arranged in alphabetical order by routine name.

BLEND Function

Blends two images together.

Usage

result = BLEND(*img1*, *a*, *img2*[, *b*])

Input Parameters

img1 — A 1D, 2D, or 3D array containing a signal; point or signal-interleaved signals; an image; image, row or pixel-interleaved images; or a volume.

a — An integer or array specifying a scaling factor for *img1*. When *a* is an array, it specifies the scaling factors for an array of multi-layered images.

img2 — A 1D, 2D, or 3D array containing the second image. This parameter must be an array of exactly the same size and dimensions as *img1*.

b — (optional) An integer or array specifying a scaling factor for *img2*. When *b* is an array, it specifies the scaling factors for an array of multi-layered images.
(Default: $1.0 - a$)

Returned Value

result — An array, containing the blended image, that is the same dimensions and size as *img1* and *img2*. The *result* array equals *img1* multiplied by the scale factor *a* plus *img2* multiplied by its scale factor *b*.

$$result = a[img1] + b[img2]$$

Keywords

Intleave — A scalar string indicating the type of interleaving of 2D input signals and 3D image arrays. Valid strings and the corresponding interleaving methods are:

'point' — The 2D input array arrangement is (*p*, *x*) for *p* point-interleaved signals of length *x*.

'signal' — The 2D input *image* array arrangement is (*x*, *p*) for *p* signal-interleaved signals of length *x*.

'pixel' — The input array arrangement is (*p*, *x*, *y*) for *p* pixel-interleaved images of *x*-by-*y*.

'row' — The 3D *image* array arrangement is (*x*, *p*, *y*) for *p* row-interleaved images of *x*-by-*y*.

'image' — The 3D *image* array arrangement is (*x*, *y*, *p*) for *p* image-interleaved images of *x*-by-*y*.

'volume' — The input image array is treated as a single entity.

No_Clip — If set, the *result* data type is larger than the input image data type.

TIP The *No_Clip* keyword prevents underflow or overflow conditions from occurring.

Zero_Negatives — If set, all negative values in the result are set to zero.

Discussion

The two images, *img1* and *img2* are blended using the *a* and *b* scale factors to produce an image that is the linear combination of *img1* and *img2*. Blending is sometimes used to produce a double exposure effect (e.g. fading one image out and another image in), or to combine segmented images into a single image for further analysis.

Example

Blend two images together to produce a double exposure effect.

```
img1 = IMAGE_READ(!IP_Data + 'mandril.tif')
img2 = IMAGE_READ(!IP_Data + 'aerial.tif')
img3 = BLEND(img1('pixels'), 0.5, img2('pixels'))
```

See Also

[IPMATH](#)

CANNY Function

Applies the Canny edge-detection method to an image.

Usage

result = CANNY(*image*, *ncoeffs*, *sigma*)

Input Parameters

image — A 2D or 3D array of any type except string or complex.

ncoeffs — An integer value greater than 0 that specifies the number of coefficients in the Canny filter.

sigma — A floating point value greater than 0.0 that specifies the standard deviation used in the Canny operator.

Returned Value

result — A double array containing the filtered image.

Keywords

Intleave — A scalar string indicating the type of interleaving of 2D input signals and 3D image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-inter-

leaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

Discussion

The Canny edge operator finds the local maxima in the gradient of the Gaussian-smoothed image. This is accomplished by applying a separable Gaussian gradient filter to *image*, and then determining local maxima from each 3-by-3 neighborhood of the gradient magnitude image. These maxima are returned in the output image, *result*, with all non-maximal pixels set to 0.

Example

```
test_image = IMAGE_READ(!IP_Data + 'objects.tif')
; Read an image.
edge_image = CANNY(test_image('pixels'), 40, 2.0)
; Apply the Canny edge detection method.
TVSCL, edge_image
; View the edges of the image.
```

See Also

[FILT_NONLIN](#), [SHIFT_EDGE](#)

In the *PV-WAVE Reference*: ROBERTS, SOBEL

CENTROID Function

Calculates the centroid of a binary region in an image.

Usage

```
result = CENTROID(image, pixels)
```

Input Parameters

image — A 2D array containing a region.

pixels — A long array containing the element numbers of the pixels that compose the region pixels in *image*. These pixels compose the region for which the centroid is calculated.

Returned Value

result — A long scalar value that is the element number in *image* of the centroid of the region.

Keywords

None.

Discussion

The centroid of a binary region, also known as the center of gravity, is a measure of the region's center. The centroid is expressed in pixel-coordinate values and is calculated as the ratio of the first-order to the zeroth-order spatial moments:

$$centroidx = M(1, 0)/M(0, 0)$$

$$centroidy = M(0, 1)/M(0, 0)$$

The returned value, *result* is the element number (*n*, *m*) in the *image* array, such that:

$$centroid = centroidx + centroidy*n$$

Example

Get the pixels for a region in *image*, where the region has an amplitude of 100.

```
region_pixels = WHERE(image EQ 100)
result = CENTROID(image, region_pixels)
```

See Also

[MAJOR_AXIS](#), [MOMENT2D](#), [PERIMETER](#)

DCT Function

Performs the discrete cosine transform on a 2D image or a 3D array of images.

Usage

```
result = DCT(array [, direction])
```

Input Parameters

array — A 2D or 3D array containing an image; or image, row or pixel-interleaved images.

direction — (optional) A scalar value indicating the direction of the transform:

- 1 Forward transform (default)
- 1 Backward transform

Returned Value

result — A float array of the same size and dimensions as *array*.

Keywords

Intleave — A scalar string indicating the type of interleaving of 3D input image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

Discussion

The discrete cosine transform (DCT) is often used in image compression and coding. A simple compression algorithm truncates a percentage of the coefficients in the DCT result, and then performs an inverse DCT of the truncated array. The quality of the compressed image is related to the percentage of truncated coefficients.

The equation for the DCT is similar to that of the FFT. In two dimensions, the forward transform is defined by the following equation:

$$DCT(n,m) = k(n)k(m) \sum_{y=0}^{M-1} \sum_{x=0}^{N-1} \text{image}(x,y) \cos\left(\frac{\pi n(2x+1)}{2N}\right) \cos\left(\frac{\pi m(2y+1)}{2M}\right)$$

where n and m are defined from 1 to $N-1$ and $M-1$, respectively and

$$k(n) = \begin{cases} \sqrt{\frac{1}{N}} & \text{for } n = 0 \\ \sqrt{\frac{2}{N}} & \text{otherwise} \end{cases}$$

$$k(m) = \begin{cases} \sqrt{\frac{1}{M}} & \text{for } m = 0 \\ \sqrt{\frac{2}{M}} & \text{otherwise} \end{cases}$$

The inverse transform is defined as:

$$IDCT(x,y) = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} k(n)k(m) DCT(n,m) \cos\left(\frac{\pi n(2x-1)}{2N}\right) \cos\left(\frac{\pi m(2y+1)}{2M}\right)$$

Example

Take the forward DCT of an image.

```
image = IMAGE_READ(!IP_Data + 'face.tif')
; Read in an image.

image_dct = DCT(image('pixels'), -1)
; Take the direct cosine transform of the image.

TVSCL, IPALOG(image_dct)
; Display the result.

image_idct = DCT(image_dct, 1)
; Take the inverse DCT.

TVSCL, image_idct
```

See Also

[HAAR](#), [IPWAVELET](#), [SLANT](#)

In the *PV-WAVE Reference*: FFT

DENSITY_SLICE Function

Performs color density slicing on individual 2D images.

Usage

result = DENSITY_SLICE(*image*, *nbands*)

Input Parameters

image — A 2D image array.

nbands — A scalar integer indicating the number of color bands with which to slice image. (*nbands* ≥ 1)

Returned Value

result — An array of the same size, type, and dimensions as *image*.

Keywords

Bands — A scalar value or an array with the number of elements (up to *nbands*) identifying the amplitude ranges for slicing the input *image*. If *Bands* is not used to define the individual bands, the value of the *Band_Step* keyword is used as the step size.

Band_Step — A scalar value specifying the step-size between bands, when the number of bands defined is less than *nbands*. If *Band_Step* isn't defined, the difference between the first two bands is used as the step size.

Levels — A scalar value or an array with the number of elements (up to *nbands*) identifying the color values for the *result* image bands. If the keyword *Levels* is not defined, then *Levels* = *Bands*. If a level number is not defined for each band, then *Level_Step* is used as the step size.

Level_Step — A scalar value specifying the step-size between levels, when the number of levels defined is less than *nbands*. If *Level_Step* is not defined, the difference between the first two levels is used as the step size. Otherwise, if only one level is defined, the step size is the *image* (max - min + 1)/*nbands*.

Discussion

NOTE Density slicing is defined for 2D images only.

Density slicing is a form of pseudo-color image processing in which the input image is treated as a 2D intensity function. The output image is produced by “slicing” the 2D intensity function using slicing planes that are parallel to the image coordinate plane.

A different level, or color, is assigned to values in the input image that fall between each slicing plane. Values on the face of the upper plane are assigned the color level corresponding to the area between the plane and the one below it.

Example

```
image = IMAGE_READ(!IP_Data + 'xray.tif')
        ; Read in an 8-bit color image.

density_image = DENSITY_SLICE(image('pixels'), $
        5, bands=[20, 60, 120, 160, 200])
        ; Density-slice the image into 5 bands.

DEVICE, pseudo = 8
        ; Set the device to 8 bit.

LOADCT, 6
        ; Load the prism color table.

TV, density_image
        ; Display the density image.
```

See Also

[THRESH_ADAP](#), [THRESHOLD](#)

***DIST_MAP* Function**

Computes the Euclidean distance map (EDM) for an image.

Usage

```
result = DIST_MAP(image[, threshold])
```

Input Parameters

image — A nonzero 2D or 3D byte array of any data type except string or complex. This array contains an image; or image, row or pixel-interleaved images.

NOTE The *image* parameter must be a binary image containing only two gray-scale levels, if *threshold* is not given.

threshold — (optional) Specifies the threshold value for the object in *image*; the *image* object is assumed to be composed of pixels with values greater than or equal to *threshold*.

Returned Value

result — A double array of the same size and dimensions as *image*, containing the Euclidean distance map of *image*.

Keywords

Intleave — A scalar string indicating the type of interleaving of 3D input *image* arrays. Valid strings and the corresponding interleaving methods are:

'*pixel*' — The input array arrangement is (*p*, *x*, *y*) for *p* pixel-interleaved images of *x*-by-*y*.

'*row*' — The 3D *image* array arrangement is (*x*, *p*, *y*) for *p* row-interleaved images of *x*-by-*y*.

'*image*' — The 3D *image* array arrangement is (*x*, *y*, *p*) for *p* image-interleaved images of *x*-by-*y*.

Value — Specifies a scalar value greater than 0 for the binary object in *image*.

Discussion

The Euclidean distance map is used to identify object boundaries. It is computed by finding the shortest distance from each object-pixel in *image* to a background pixel. The Euclidean distance is defined as follows:

$$\text{Euclidean distance} = \sqrt{x^2 + y^2} ,$$

where $x = (x_{object} - x_{background})$ and $y = (y_{object} - y_{background})$.

Example 1

```
test_image = IMAGE_READ(!IP_Data + 'objects.tif')
; Read a binary image.

edm_image = DIST_MAP(test_image('pixels'), 100)
; Find the Euclidean distance map (EDM) of the image.

thresh_image = THRESHOLD(edm_image, 0, 2, $
/A_Neq, /Binary)
; Threshold the EDM to find the object outlines.

TVSCL, thresh_image
; View the object outlines.
```

Example 2

```
test_image = IMAGE_READ(!IP_Data + 'blobs.tif')
; Read a binary image of different blob shapes.

image = THRESHOLD(test_image('pixels'), 5, 250, /Binary)
region_seeds = REGION_FIND(image, 255, 6)
; Find and fill the regions.

region_image = REGION_GROW(image, $
region_seeds, 2.0)

TVSCL, region_image
```



Figure 2-1 The test blobs with different grayscale colors.

```
edm_image = DIST_MAP(region_image, value = 2)
    ; Find the EDM of a single region.
TVSCL, edm_image
```

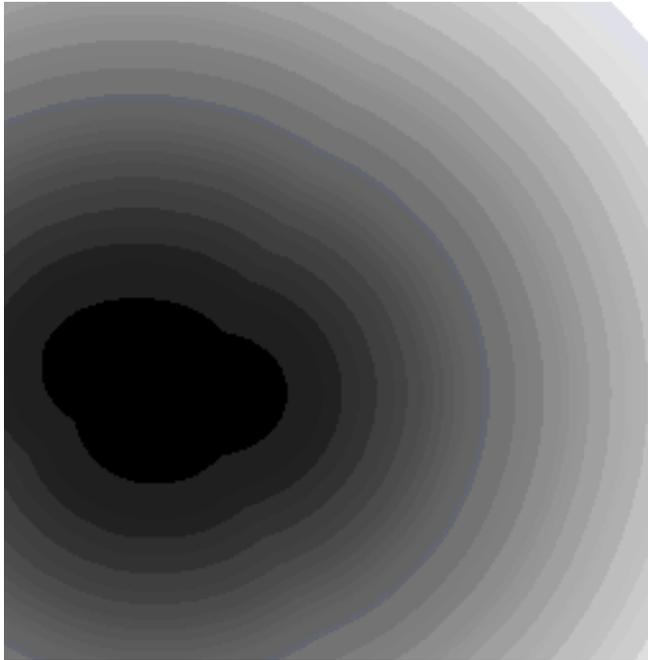


Figure 2-2 The distance map of a single blob.

```
thresh_image = THRESHOLD(edm_image, 0, 2, /A_Neq, /Binary)
; Threshold the EDM to find region outline.

TVSCL, thresh_image
; View the object outlines.
```

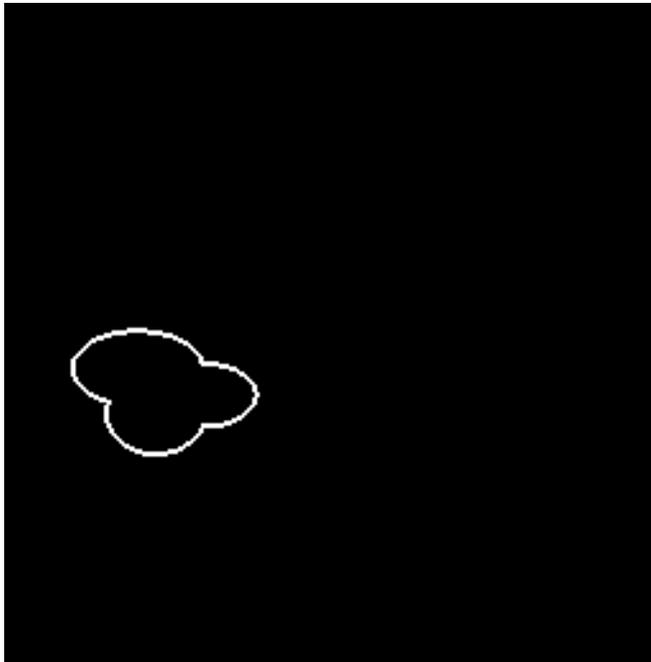


Figure 2-3 The isolated blob outline, or boundary.

See Also

[HIT_MISS](#), [MORPH_CLOSE](#), [MORPH_OPEN](#),
[MORPH_OUTLINE](#), [SKELETONIZE](#), [TOP_HAT](#)

In the *PV-WAVE Reference*: [DILATE](#), [ERODE](#)

ENTROPY Function

Computes the entropy of an array.

Usage

result = ENTROPY(*array*)

Input Parameters

array — A 2D array.

Returned Value

result — A floating-point scalar value with the array entropy.

Keywords

None.

Discussion

Computes the entropy of *array(k, l)* as:

$$H = \sum_{k=0}^{N-1} \sum_{l=0}^{M-1} (\text{array}(k, l) \cdot \ln(\text{array}(k, l))) ,$$

where *array(k, l)* > 0.

Values in *array* that are less than or equal to 0 don't contribute to the entropy calculation because the natural log function is undefined for these values. The entropy of an array that is all zeros and/or negative values is undefined.

Example

```
image = IMAGE_READ(!IP_Data + 'noise_test.tif')
; Read an image.
ent = ENTROPY(image('pixels'))
; Compute the entropy of the image.
PRINT, 'Entropy = ', ent
```

See Also

[KURTOSIS](#), [MODE](#), [RANGE](#), [SKEWNESS](#)

***FILT_DWMTM* Function**

Performs a 1D, 2D, or 3D adaptive double-window-modified trimmed mean filter.

Usage

```
result = FILT_DWMTM(image, noise_std, thresh_factor,  
medxdim[, medydim[, medzdim]])
```

Input Parameters

image — A 1D, 2D or 3D array containing a signal; point or signal-interleaved signals; an image; image, row or pixel interleaved images; or a volume.

noise_std — The standard deviation of the noise in *image*.

thresh_factor — The factor used to compute the threshold range for points included in the mean calculation. If *thresh_factor* = 0, then *FILT_DWMTM* is the same as a median filter.

medxdim — The width of the median filtering window.

medydim — (optional) The height of the median filtering window. (Used for images and volumes only.)

medzdim — (optional) The depth of the 3D median filtering window. (Used for volumes only.)

Returned Value

result — An array of the same size and dimensions as *image*, unless otherwise affected by using the *Edge* keyword.

Keywords

Edge — A scalar string indicating how edge effects are handled. (Default: 'zero') Valid strings are:

'pad' — The input image is padded before the filtering operation with the value specified using the *Pad_Value* keyword.

'zero' — Sets the border of the output image to zero. (Default)

'copy' — Copies the border of the input image to the output image.

'reduce' — Returns a reduced-size image that is smaller than the input image by the kernel dimensions.

Intleave — A scalar string indicating the type of interleaving of 2D input signals and 3D image arrays. Valid strings and the corresponding interleaving methods are:

'point' — The 2D input array arrangement is (p, x) for p point-interleaved signals of length x .

'signal' — The 2D input *image* array arrangement is (x, p) for p signal-interleaved signals of length x .

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

Mxdim — The width of the mean filtering window. The value of *Mxdim* must be $\geq medxdim$. (Default: $medxdim + 2$)

Mydim — The height of the mean filtering window. The value of *Mydim* must be $\geq medydim$. (Used for images and volumes only.) (Default: $medydim + 2$)

Mzdim — The depth of the 3D mean filtering window. The value of *Mzdim* must be $\geq medzdim$. (Used for volumes only.) (Default: $medzdim + 2$)

NOTE *Mzdim* is only used for 3D volumes.

Pad_Value — The value to use for the image padding. (Default: 0)

NOTE The *Pad_Value* keyword is valid only when *Edge* = 'pad'.

Spot — This keyword specifies explicit positioning of the element number of the filter window “sweet spot.” The *Spot* keyword default depends on the setting of the *Edge* keyword as shown in the following table:

<i>Edge</i> Keyword	<i>Spot</i> Keyword	<i>Spot</i> Keyword Default Setting
not used (Default: 'zero')	As specified, or the default.	kernel center
'pad'	Ignored.	kernel center
'zero'	As specified, or the default.	kernel center
'copy'	As specified, or the default.	kernel center
'reduce'	Ignored.	<i>Spot</i> = 0

Zero_Negatives — If set, all negative values in the result image are set to zero.

Discussion

The adaptive double-window-modified trimmed mean filter (DWMTM) is superior to the mean filter for removing Gaussian noise in the presence of impulse noise. The filter operation begins by computing the median value to use for the median filter. An output pixel is then computed from the mean of the mean filtering window. Values in the mean filtering window lying outside the median plus or minus a constant c (where, $c = thresh_factor * noise_std$) don't contribute to the mean calculation.

In this way, high and low outliers, such as impulse noise are eliminated from the mean calculation. Typical values for *thresh_factor* are in the 1.5 to 2.5 range. If *thresh_factor* = 0, then `FILT_DWMTM` is the same as a median filter.

Example

```
image = IMAGE_READ(!IP_Data + 'face.tif')
; Read an image.

noise = NOISE_GEN(image('width'), $
    image('height'), /Normal, High = 128.0, $
    Low = 50.0)
; Corrupt the image with Gaussian noise.

noise_image = IPMATH(image('pixels'), '+', noise)

status = NOISE_IMPULSE(0.15, noise_image)
; Corrupt the noise image with impulse noise.
```

```
dwmtm_image = FILT_DWMTM(noise_image, 1.0, 2.0, 3, 3)
    ; Try to remove the noise using the DWMTM filter.
TVSCL, dwmtm_image
    ; Display the image.
```

See Also

[FILT_MMSE](#), [FILT_NONLIN](#)

FILT_FREQ Function

Generates a 2D Butterworth or ideal lowpass, bandpass, bandstop, or highpass spatial frequency domain filter.

Usage

```
filter = FILT_FREQ(cutoff [, ucutoff ] [, order])
```

Input Parameters

cutoff — A scalar float value for the cutoff frequency of the filter.

ucutoff — (optional) A scalar float value for the upper cutoff frequency of the filter

NOTE The *ucutoff* parameter is only valid for the bandpass and bandstop filters.

order — (optional) A scalar float value for the filter order.

NOTE The *order* parameter is only valid for the Butterworth filter.

Returned Value

result — A filter object containing an “unshifted” spectral filter. A spectral filter object is an associative array with the following keys:

‘kernel’ — A 2D floating-point array of the filter values.

‘cutoff’ — The filter cutoff frequency or frequencies. For lowpass and highpass filters, a scalar value. For bandpass and bandstop filters, a two-element array containing the lower cutoff frequency and the upper cutoff frequency.

'pass' — A string indicating one of the following kinds of filter:

low — A lowpass filter.

high — A highpass filter.

band — A bandpass filter.

stop — A bandstop filter.

notch — A notch filter.

'dc_offset' — A floating point scalar value containing the DC offset of the filter.

'maximum' — A floating point scalar value which is the maximum value of the filter.

'type' — One of the following strings indicating the type of filter.

ideal — An ideal filter.

butterworth — A Butterworth filter.

'domain' — One of the following strings indicating the filter domain.

spectral — The filter is in the spectral (spatial frequency) domain.

spatial — The filter is in the spatial domain.

NOTE `FILT_FREQ` produces only *spectral* domain filters. For information about *spatial* domain filters, see the [IPCREATE_FILTER](#).

'xloc' — A scalar value specifying the *x*-location of the filter center. Valid for Notch filters only.

'yloc' — A scalar value specifying the *y*-location of the filter center. Valid for Notch filters only.

'center' — If set to 1, the filter center (DC value) is shifted to the array ('kernel') center; otherwise, the filter is unshifted.

'order' — A scalar value specifying the filter order. Valid for Butterworth filters only.

Keywords

Band — If set, generates a bandpass filter.

Butterworth — If set, generates a Butterworth filter instead of an ideal filter.

Center — If set, shifts the output so that the center of the filter, the DC component, is the same as the center of the array.

CO_Frac — A scalar float that is the fraction of the maximum value of the filter at the cutoff frequency.

$$\left(\text{Default: } \frac{1}{\sqrt{2.0}} \right)$$

DC_Offset — A scalar float that is the DC offset of the filter.
(Default: 0.0)

High — If set, generates a highpass filter.

Low — If set, generates a lowpass filter. (Default: set)

Maximum — A scalar float that is the maximum value of the filter. (Default: $DC_Offset + 1.0$)

Stop — If set, generates a bandstop filter.

Xdim — The x -dimension of the filter. (Default: $ydim$, if $Ydim$ is specified; 256 otherwise)

Ydim — The y -dimension of the filter. (Default: $xdim$, if $Xdim$ is specified; 256 otherwise)

Discussion

FILT_FREQ produces circularly symmetric spatial frequency domain filters which can be applied to frequency domain images. Ideal filters are simplistic, but the filtered spatial domain image displays ringing after an ideal filter is applied. The Butterworth filter provides a smooth transition from the passband to the cutoff band, reducing the ringing effect.

The following equations are used for a Butterworth filter:

Lowpass:

$$H(u,v) = \frac{1}{1 + \left(\frac{D_0}{\sqrt{u^2 + v^2}} \right)^{2n}}$$

where n is the filter order and D_0 is the cutoff frequency.

Highpass:

$$H(u,v) = \frac{1}{1 + \left(\frac{D_0}{\sqrt{u^2 + v^2}} \right)^{-2n}}$$

where n is the filter order and D_0 is the cutoff frequency.

Bandstop:

$$H(u,v) = \frac{1}{1 + \left(\frac{D_L}{\sqrt{u^2 + v^2}} \right)^{2n}} + \frac{1}{1 + \left(\frac{D_U}{\sqrt{u^2 + v^2}} \right)^{-2n}}$$

where n is the filter order, and D_L and D_U are the lower and upper cutoff frequencies, respectively.

Bandpass:

$$H(u,v) = \frac{1}{1 + \left(\frac{D_L}{\sqrt{u^2 + v^2}} \right)^{2n}} \times \frac{1}{1 + \left(\frac{D_U}{\sqrt{u^2 + v^2}} \right)^{-2n}}$$

where n is the filter order, and D_L and D_U are the lower and upper cutoff frequencies, respectively.

Example

```
image = IMAGE_READ(!IP_Data + 'noise_test.tif')
; Read a noisy image.

image_fft = FFT(image('pixels'), -1)
; Take the fast Fourier transform (FFT) of the image.

lpf = FILT_FREQ(30, 1.0, /Butterworth, /Low, $
  Xdim = image('width'), Ydim = image('height'))
; Create a lowpass Butterworth filter.

image_filt = IPMATH(lpf('kernel'), '*', image_fft)
; Apply the filter to the FFT of the image.

image_ifft = ABS(FFT(image_filt, 1))
; Take the inverse FFT of the filtered image.

TVSCL, image_ifft
; Display the filtered image.
```

See Also

[FILT_NOTCH](#), [IPMATH](#), [IPWIN](#)

In the *PV-WAVE Reference*: FFT

***FILT_MMSE* Function**

Performs 1D, 2D or 3D adaptive minimum mean-squared error filtering.

Usage

```
result = FILT_MMSE(image, noise_var, wxdim[, wydim[, wzdim]])
```

Input Parameters

image — A 1D, 2D, or 3D array containing a signal; point or signal-interleaved signals; an image; image, row or pixel interleaved images; or a volume.

noise_var — A scalar float data type that is the variance of the noise in *image*.

wxdim — The width of the filtering window.

wydim — (optional) The height of the filtering window. (Used for images and volumes only.)

wzdim — (optional) The depth of the 3D median filtering window. (Used for volumes only.)

Returned Value

result — An array of the same size and dimensions as *image*, unless otherwise affected by using the *Edge* keyword.

Keywords

Edge — A scalar string indicating how edge effects are handled. (Default: 'zero') Valid strings are:

'pad' — The input image is padded before the filtering operation with the value specified using the *Pad_Value* keyword.

'zero' — Sets the border of the output image to zero. (Default)

'copy' — Copies the border of the input image to the output image.

'reduce' — Returns a reduced-size image that is smaller than the input image by the kernel dimensions.

Intleave — A scalar string indicating the type of interleaving of 2D input signals and 3D image arrays. Valid strings and the corresponding interleaving methods are:

'point' — The 2D input array arrangement is (p, x) for p point-interleaved signals of length x .

'signal' — The 2D input *image* array arrangement is (x, p) for p signal-interleaved signals of length x .

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

Pad_Value — The value to use for the image padding. (Default: 0)

NOTE The *Pad_Value* keyword is valid only when *Edge* = 'pad'.

Spot — This keyword specifies explicit positioning of the element number of the filter window “sweet spot.” The *Spot* keyword default depends on the setting of the *Edge* keyword as shown in the following table:

Edge Keyword	Spot Keyword	Spot Keyword Default Setting
not used (Default: 'zero')	As specified, or the default.	kernel center
'pad'	Ignored.	kernel center
'zero'	As specified, or the default.	kernel center
'copy'	As specified, or the default.	kernel center
'reduce'	Ignored.	<i>Spot</i> = 0

Zero_Negatives — If set, all negative values in the result image are set to zero.

Discussion

The adaptive minimum mean-squared error (MMSE) filter is useful for removing Gaussian or Rayleigh noise. Each pixel in the filtered result is computed from the input image pixels in the filter window as follows:

$$result(i, j) = \left(1 - \frac{noise_var}{local_var}\right) image(i, j) + \frac{noise_var}{local_var} \cdot local_mean ,$$

where *local_var* and *local_mean* are the variance and mean, respectively, of the pixels in the filter window.

Example

```
image = IMAGE_READ(!IP_Data + 'face.tif')
    ; Read an image.
noise_var = 15.0
    ; Corrupt the image with Rayleigh noise that has a
    ; variance of 15.0.
noise = NOISE_RAYLEIGH(noise_var, $
    image('width'), image('height'))
noise = noise + 128
noise_image = IPMATH(image('pixels'), '+', $
    noise, /No_Clip)
mmse_image = FILT_MMSE(noise_image, noise_var, $
    3, 3)
    ; Remove the noise using the MMSE filter.
TVSCL, mmse_image
    ; Display the image.
```

See Also

[FILT_DWMTM](#), [FILT_NONLIN](#)

***FILT_NONLIN* Function**

Performs nonlinear filtering operations on 1D, 2D, or 3D image arrays.

Usage

```
result = FILT_NONLIN(image, wxdim[, wydim[, wzdim]])
```

Input Parameters

image — A 1D, 2D or 3D array containing a signal; point or signal-interleaved signals; an image; image, row or pixel-interleaved images; or a volume.

wxdim — The width of the filtering window.

wydim — (optional) The height of the filtering window. (Used for images and volumes.)

wzdim — (optional) The depth of the filtering window. (Used for volumes.)

Returned Value

result — An array containing the filtered data that is of the same size and dimensions as *image*, unless otherwise affected using the *Edge* keyword.

Keywords

Atmeanf — If set, applies an alpha-trimmed mean filter.

Chmeanf — If set, applies a contra-harmonic mean filter.

Edge — A scalar string indicating how edge effects are handled. (Default: 'zero') Valid strings are:

'pad' — The input image is padded before the filtering operation with the value specified using the *Pad_Value* keyword.

'zero' — Sets the border of the output image to zero. (Default)

'copy' — Copies the border of the input image to the output image.

'reduce' — Returns a reduced-size image that is smaller than the input image by the kernel dimensions.

Filt_Type — A scalar byte indicating which nonlinear filter (listed in the following table) to apply. This keyword may be used in place of the corresponding filter keywords (*Maxf*, *Minf*, ..., *Ypmeanf*). (Default: 0)

Filt_Type (scalar)	Nonlinear Filter Type	Corresponding Keyword
0 (default)	Maximum Filter (default)	Maxf
1	Minimum Filter	Minf
2	Range Filter	Rangef
3	Geometric Mean Filter	Gmeanf
4	Mode Filter	Modef
5	Rank Filter	Rankf
6	Alpha-Trimmed Mean Filter	Atmeanf
7	Contra-Harmonic Mean Filter	Chmeanf
8	Yp Mean Filter	Ypmeanf

F_Order — A scalar value that specifies the order of the filter.

NOTE The *F_Order* keyword is only valid for the Y_p mean and contra-harmonic mean filters.

Gmeanf — If set, applies a geometric mean filter.

Intleave — A scalar string indicating the type of interleaving of 2D input signals and 3D image arrays. Valid strings and the corresponding interleaving methods are:

'point' — The 2D input array arrangement is (p, x) for p point-interleaved signals of length x .

'signal' — The 2D input *image* array arrangement is (x, p) for p signal-interleaved signals of length x .

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

Maxf — If set, applies a maximum filter.

Minf — If set, applies a minimum filter.

Modf — If set, applies a mode filter.

Pad_Value — A scalar value to use for the image padding. (Default: 0)

NOTE The *Pad_Value* keyword is valid only when *Edge* = ' pad ' .

Rangef — If set, applies a range filter.

Rankf — If set, applies a rank filter.

Rank_Num — The rank number of the rank filter. The value must be in the following range: $0 \leq Rank_Num < \text{the number of elements in the filter window}$.

NOTE The *Rank_Num* keyword is required when the rank filter is specified.

Spot — This keyword specifies explicit positioning of the element number of the filter window “sweet spot.” The *Spot* keyword default depends on the setting of the *Edge* keyword as shown in the following table:

<i>Edge</i> Keyword	<i>Spot</i> Keyword	<i>Spot</i> Keyword Default Setting
not used (Default: ' zero ')	As specified, or the default.	kernel center
' pad '	Ignored.	kernel center
' zero '	As specified, or the default.	kernel center
' copy '	As specified, or the default.	kernel center
' reduce '	Ignored.	<i>Spot</i> = 0

Trim — A scalar integer or long value specifying the number of array values to trim from the mean calculation. The value must be in the following range:

$$0 \leq Trim < 1/2 (N - 1) ,$$

where *N* is the number of elements in the filter window.

NOTE The *Trim* keyword is required when the alpha-trimmed mean filter is specified.

Ypmeanf — If set, a Y_p mean filter is applied

Zero_Negatives — If set, all negative values in the result are set to zero.

Discussion

The filter type can be specified using the *Filt_Type* keyword, or any of the specific filter keywords. No more than one filter type can be defined at one time. The default filter type for the function is the *Filt_Type* keyword default of 0, meaning the maximum filter is applied.

Geometric Mean Filter

The geometric-mean filter is a nonlinear filter sometimes used for removing Gaussian distributed noise. Each pixel in the result is computed as the product of the N pixels to the N^{-1} power within the filter window of N elements defined by *wxdim*, *wydim*, and/or *wzdim*.

Maximum Filter

The maximum filter is a nonlinear filter that can be used for removing outlying low or negative values from an image. Each pixel in *result* is computed as the maximum of the N pixels within the filter window of N elements defined by *wxdim*, *wydim*, and/or *wzdim*.

Minimum Filter

The minimum filter is a nonlinear filter that can be used for removing outlying high values from an image. Each pixel in *result* is computed as the minimum of the N pixels within the filter window of N elements defined by *wxdim*, *wydim*, and/or *wzdim*.

Mode Filter

The mode filter is a nonlinear filter that can be used for noise removal. Each pixel in *result* is computed as the mode (the most frequent pixel value) of the N pixels within the filter window of N elements defined by *wxdim*, *wydim*, and/or *wzdim*.

Range Filter

The range filter is a nonlinear filter that can be used for edge detection. Each pixel in *result* is computed as the range (the maximum minus the minimum) of the filter window of N elements defined by *wxdim*, *wydim*, and/or *wzdim*.

Rank Filter

The rank filter is a nonlinear filter that can be used for removal of impulse noise. Each pixel in *result* is computed as the rank of the filter window of N elements defined by $wxdim$, $wydim$, and/or $wzdim$. The rank is defined as the value of the pixel in the *Rank_Num* position when all pixels in the filtering window are arranged in ascending order. The rank filter degenerates to the MEDIAN filter when

$$Rank_Num = \frac{N-1}{2} .$$

Alpha-Trimmed Mean Filter

The alpha-trimmed mean filter is used to remove noise from images corrupted with both Gaussian and impulse noise. The output of the alpha-trimmed mean filter is the mean of the pixels in the filter window, with a number of values, defined by *Trim*, excluded. First, the pixels in the filter window are arranged in ascending order according to grayscale value. The output is then calculated from:

$$AT\ Mean = \left(\frac{1}{N-2(Trim)} \right) \sum_{i=Trim}^{N-Trim-1} A_i$$

where the filter window has N elements and A_i are the grayscale values in the window $A_1 \leq A_2 \leq \dots \leq A_N$.

Contra-Harmonic Mean Filter

The contra-harmonic mean filter is useful for removing Gaussian noise without destroying edge features. The output is calculated from:

$$CH\ Mean = \frac{\sum_{i=0}^{F_Order+1} A_i^{(F_Order+1)}}{\sum_{i=0}^{F_Order} A_i^{(F_Order)}}$$

for each pixel, A_i in the filter window of size N .

Yp Mean Filter

The Yp mean filter is useful for removing Gaussian noise, while preserving edge features. For negative values of F_Order , the Yp mean filter removes positive outliers. For positive values of F_Order , the Yp mean filter removes negative outliers. The output of the Yp mean filter is:

$$Y_p \text{ Mean} = \left(\frac{\sum_{i=0}^{N-1} A_i^{(F_Order)}}{N} \right)^{(1/F_Order)}$$

Example 1

Use the Yp mean filter for removing Gaussian noise while preserving image edges.

```
image = IMAGE_READ(!IP_Data + 'face.tif')
    ; Read an image.

noise = NOISE_GEN(image('width'), image('height'), $
    /Normal, High = 128, Low = 50)

noise_image = IPMATH(image('pixels'), '+', noise)
    ; Corrupt the image with Gaussian noise.

yp_image = FILT_NONLIN(noise_image, 3, 3, $
    F_Order = 2.0, /Ypmeanf)
    ; Try to remove the noise using the Yp mean filter.

TVSCL, yp_image
    ; Display the image.
```

Example 2

The rank filter is the same as the median filter when the rank number is set to the filter window center.

```
image = IMAGE_READ(!IP_Data + 'airplane.tif')
    ; Read an image.

IMAGE_DISPLAY, image
    ; Display the original image.
```



Figure 2-4 The original image of the airplane.

```
pixels = image('pixels')
status = NOISE_IMPULSE(0.15, pixels)
        ; Corrupt the image with speckle noise.
image('pixels') = pixels
IMAGE_DISPLAY, image
```



Figure 2-5 The image corrupted with speckle noise.

```
rank_image = FILT_NONLIN(image('pixels'), 3, 3, $  
    Rank_Num = 4, /Rankf)  
    ; Remove the noise using a rank filter.  
TVSCL, rank_image  
    ; Display the image.
```



Figure 2-6 The filtered image.

See Also

[FILT_DWMTM](#), [FILT_MMSE](#)

In the *PV-WAVE Reference*: [MEDIAN](#)

FILT_NOTCH Function

Generates a 2D ideal notch spatial frequency domain filter.

Usage

result = `FILT_NOTCH(radius, xloc, yloc)`

Input Parameters

radius — The filter cutoff region radius, in pixels.

xloc — The *x*-location of the center of the filter cutoff region.

yloc — The *y*-location of the center of the filter cutoff region.

Returned Value

result — A filter object containing a notch filter in the filter object format. A filter object is an associative array with the following keys:

'kernel' — A 2D floating-point array of the filter values.

'cutoff' — The filter cutoff frequency or frequencies. For lowpass and highpass filters, a scalar value. For bandpass and bandstop filters, a two-element array containing the lower cutoff frequency and the upper cutoff frequency.

'pass' — A string indicating one of the following kinds of filter:

low — A lowpass filter.

high — A highpass filter.

band — A bandpass filter.

stop — A bandstop filter

notch — A notch filter.

'dc_offset' — A floating point scalar value containing the DC offset of the filter.

'maximum' — A floating point scalar value which is the maximum value of the filter.

'type' — One of the following strings indicating the type of filter.

ideal — An ideal filter.

butterworth — A Butterworth filter.

'domain' — One of the following strings indicating the filter domain.

spectral — The filter is in the spectral (spatial frequency) domain.

spatial — The filter is in the spatial domain.

NOTE `FILT_NOTCH` produces only *spectral* domain filters. For information about *spatial* domain filters, see the [IPCREATE_FILTER](#).

'xloc' — A scalar value specifying the *x*-location of the filter center. Valid for Notch filters only.

'yloc' — A scalar value specifying the *y*-location of the filter center. Valid for Notch filters only.

'center' — If set to 1, the filter center (DC value) is shifted to the array ('kernel') center; otherwise, the filter is unshifted.

'order' — A scalar value specifying the filter order. Valid for Butterworth filters only.

Keywords

Center — If set, shifts the output so that the center of the filter, the DC component, is the same as the center of the array.

DC_Offset — A scalar float containing the DC offset of the filter. (Default: 0.0)

Maximum — A scalar float that is the maximum value of the filter. (Default: $DC_Offset + 1.0$)

Xdim — The *x*-dimension (width) of the filter. (Default: the value of *Ydim*, if specified; 256 otherwise)

Ydim — The *y*-dimension (height) of the filter. (Default: the value of *Xdim*, if specified; 256 otherwise)

Discussion

`FILT_NOTCH` produces a circularly symmetric, spatial frequency domain, ideal notch filter which can be applied to a frequency domain image. Notch filters are

useful for removing narrow frequency ranges from images. Notch filtering is commonly used to remove periodic (coherent) noise.

Example

```
image = IMAGE_READ(!IP_Data + 'face.tif')  
      ; Read an image.  
IMAGE_DISPLAY, image
```



Figure 2-7 The original image.

```
noise = NOISE_PERIODIC(image('width'), $  
      image('height'), F1 = 1.0, F2 = 3.5, $  
      Amp = 10.0, DC_Offset = 127.0)  
noisy_image = image('pixels') + noise  
            ; Corrupt the image with periodic noise.  
TVSCL, noisy_image
```



Figure 2-8 The image corrupted with periodic noise.

```
notch_filter = FILT_NOTCH(6.0, 42, 12, $  
    Xdim = image('width'), Ydim = image('height'))  
clean_image = FFT(noisy_image, -1) * notch_filter('kernel')  
clean_image = ABS(FFT(clean_image, 1))  
    ; Remove the noise from the image.  
TVSCL, clean_image  
    ; Display the clean image.
```



Figure 2-9 The cleaned up image.

See Also

[FILT_FREQ](#)

***FILT_SMOOTH* Function**

Performs smoothing on 1D, 2D or 3D image arrays.

Usage

```
result = FILT_SMOOTH(image, wxdim [, wydim [, wzdim]])
```

Input Parameters

image — A 1D, 2D, or 3D array containing a signal; point or signal-interleaved signals; an image; image, row or pixel-interleaved images; or a volume.

wxdim — The width of the filtering window.

wydim — (optional) The height of the filtering window. (Required for images and volumes.)

wzdim — (optional) The depth of the filtering window. (Required for volumes.)

Returned Value

result — An array containing the filtered data that is of the same size and dimensions as *image*, unless otherwise affected using the *Edge* keyword.

Keywords

Edge — A scalar string indicating how edge effects are handled. (Default: 'zero') Valid strings are:

'pad' — The input image is padded before the filtering operation with the value specified using the *Pad_Value* keyword.

'zero' — Sets the border of the output image to zero. (Default)

'copy' — Copies the border of the input image to the output image.

'reduce' — Returns a reduced-size image that is smaller than the input image by the kernel dimensions.

Intleave — A scalar string indicating the type of interleaving of 2D input signals and 3D image arrays. Valid strings and the corresponding interleaving methods are:

'point' — The 2D input array arrangement is (*p*, *x*) for *p* point-interleaved signals of length *x*.

'signal' — The 2D input *image* array arrangement is (x, p) for p signal-interleaved signals of length x .

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

Pad_Value — A scalar value to use to pad the image. (Default: 0)

Spot — This keyword specifies explicit positioning of the element number of the filter window “sweet spot.” The *Spot* keyword default depends on the setting of the *Edge* keyword as shown in the following table:

Edge Keyword	Spot Keyword	Spot Keyword Default Setting
not used (Default: 'zero')	As specified, or the default.	kernel center
'pad'	Ignored.	kernel center
'zero'	As specified, or the default.	kernel center
'copy'	As specified, or the default.	kernel center
'reduce'	Ignored.	<i>Spot</i> = 0

Zero_Negatives — If set, all negative values in the result are set to zero.

Discussion

The smoothing filter is used to clear up image blurring, perform lowpass filtering, and to remove noise. `FILT_SMOOTH` uses the same algorithm as the `PV-WAVE SMOOTH` function, except that `FILT_SMOOTH` offers greater control over filter window dimensions, edge effects, and interleaving.

Example

```
image = IMAGE_READ(!IP_Data + 'noise_test.tif')  
; Read an image.
```

```
IMAGE_DISPLAY, image
    ; Look at the noisy image.

smooth_image = FILT_SMOOTH(image('pixels'), 3,5)
    ; Use smoothing to remove noise.

TVSCL, smooth_image
    ; Display the smoothed image.
```

See Also

[FILT_NONLIN](#)

In the *PV-WAVE Reference*: [MEDIAN](#), [SMOOTH](#)

FILT_WIENER Function

Computes and applies a parametric Wiener filter to an image that is either in the spatial or the spatial frequency domain.

Usage

```
result = FILT_WIENER(gamma, image, degrad, noise [, original])
```

Input Parameters

gamma — A scalar float that controls the least-squared error constraint of the Wiener filter.

image — The input corrupted image to be filtered as a 2D array containing a single image, or a 3D array containing interleaved images in ‘image’ ‘row’ or ‘pixel’ form (see *intleave* keyword below). This and the following three parameters must all be in either the spatial or in the frequency domain.

degrad — The degradation function as a scalar, or as a 2D array of a single image, or as a 3D array of interleaved images in ‘image’, ‘row’, or ‘pixel’ form.

noise — An estimate of the noise contained in the corrupted input image, as: a constant scalar, or as a 2D array of a single image, or a 3D array of interleaved images in ‘image’, ‘row’, or ‘pixel’ form.

original — Optional. An estimate of the original uncorrupted image as: a scalar estimate of its magnitude, or as a 2D array of a single image, or as a 3D array of interleaved images in ‘image’, ‘row’, or ‘pixel’ form.

NOTE The input parameters: *image*, *degrad*, *noise*, and *original* must all be in either the spatial or in the frequency domain. The input parameters: *degrad*, *noise*, and *original* must all three be of the same size and dimensions. For example, if *degrad* is a scalar, *noise* and *original* must also be scalars.

Returned Value

result — A complex array of the same size and dimensions as the corrupted *image*.

Keywords

Intleave — A scalar string indicating the type of interleaving of 3D input image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

Imag — If set, returns only the imaginary portion of the result.

Mag — If set, the magnitude of the computed result is returned.

Real — If set, only the real portion of the computed result is returned.

Spatial — If set, parameters *image*, *degrad*, *noise*, and *original* are all assumed to be in the spatial domain. Otherwise, they are all assumed to be in the spatial frequency domain.

Wiener — Specifies a variable to receive the computed Wiener filter. The filter is in the frequency domain and is a complex array.

Discussion

The Wiener filter, also known as the least mean square (LMS) filter, is useful in restoring an image when *a priori* knowledge of the degradation process is known. The equation for the parametric Wiener filter, in the frequency domain, is

$$\hat{F}(u,v) = \left(\frac{H^*(u,v)}{|H(u,v)|^2 + \gamma(S_n(u,v)/S_f(u,v))} \right) G(u,v) \quad ,$$

where (u, v) are the frequency coordinates, $H(u, v)$ is the degradation function, $G(u, v)$ is the degraded image, $S_f(u, v)$ is the spectrum of the original image estimate, and $S_n(u, v)$ is the noise spectrum.

Example

In this example, we work in the spatial domain. We create the corrupted image by first blurring the original with a smoothing function and then adding noise to it in the frequency domain. We then transform the result in the spatial domain.

```
IMAGE_DATA = getenv('VNI_DIR')+'/image-1_0/data/'
; Define the directory where the image resides.

test_image = IMAGE_READ( IMAGE_DATA + 'teluride24.jpg')
; Read an image. For example, teluride24.jpg is a 3D array in "image" form (p=3).

wd = test_image('width')
ht = test_image('height')
original = test_image('pixels')
WINDOW, 0, xsiz=wd, ysiz=ht, xpos=0, ypos=5, title='ORIGINAL'
TVSCL, original, true=3
; Display the original image.

degrad = FLTARR(wd, ht, 3)
FOR i=0,2 DO degrad(wd/2,ht/2,i) = 1.0
d=5
; each pixel value will be replaced by the box-car average of a 5x5 window.

degrad = SMOOTH(degrad,d,intleave='image')
; Provide an estimate of the degradation function h(x,y) in the spatial domain.
; For example, degrad = h(x,y) may be a smoothing function (such as a blurred point
; source).

freq_blurred = wd*ht*fft(degrad,-1,intleave='image')*$
fft(original,-1,intleave='image')
; Blur the original in the frequency domain:
```

```

high = 25.0
low = 0.00001
gauss_noise = low + (high-low) * RANDOMU(Seed,wd,ht,3)
freq_corrupted = freq_blurred + fft(gauss_noise, -1,
    intleave='image')
    ; Generate the noise to add to the blurred image.
corrupted = fft(freq_corrupted, 1, intleave='image')
    ; Transform into spatial domain.
WINDOW, 1, xsiz=wd, ysiz=ht, xpos=0.5*wd, ypos=5, title='CORRUPTED'
TVSCL, shift(corrupted,wd/2,ht/2,0), true=3 ;
    ; Display the corrupted image.
gamma = 1.0
    ; In the case that an estimate of the original is not provided gamma greater 1.
    ; For example for the teluride24.jpg gamma =300.
restored_image = FILT_WIENER(gamma, corrupted, degrad, $
    gauss_noise, original, /Spatial)
    ; Now use Wiener filtering to restore the image.
WINDOW, 2, xsiz=wd, ysiz=ht, xpos=wd, ypos=5, title='RESTORED'
TVSCL, restored_image, true=3
    ; Display the result.

```

See Also

[FILT_FREQ](#)

GAUSS_KERNEL Function

Computes a 1D or 2D spatial Gaussian filter kernel.

Usage

```
result = GAUSS_KERNEL(xdim[, ydim])
```

Input Parameters

xdim — The width of the Gaussian filter kernel.

ydim — (optional) The height of the Gaussian filter kernel.

Returned Value

result — A Gaussian filter in the filter object format. (See the [IPREAD_FILTER](#) function for more information on the filter object format.)

Keywords

Scale — If set, the values in *result* are normalized such that their sum is equal to 1.0.

Std — The standard deviation of the Gaussian distribution.

(Default: $\frac{xdim}{2\sqrt{2}}$)

Discussion

Gaussian filters are used to remove high frequency noise and to blur images.

NOTE Use the [IPCONVOL](#) function to apply the filter after generating the Gaussian kernel.

Example

In this example, a 7-by-7 Gaussian filter is generated and [IPCONVOL](#) is used to apply the filter.

```
g = GAUSS_KERNEL(7, 7, /Scale)
; Generate the Gaussian filter.
```

```
blur_image = IPCONVOL(image, g)
; Apply the filter to an image to blur the image.
SURFACE, g('kernel')
```

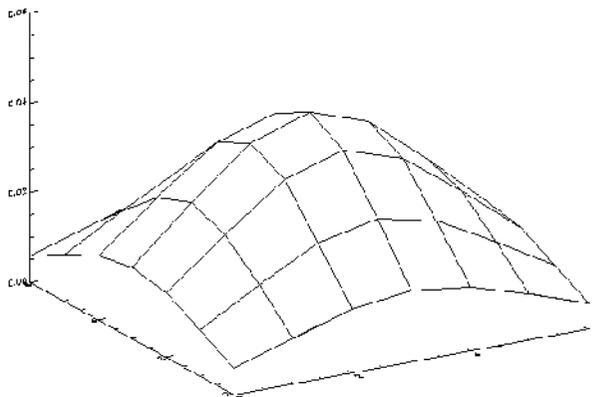


Figure 2-10 Surface plot of the 2D Gaussian filter.

See Also

[FILT_NONLIN](#), [FILT_SMOOTH](#), [IPCONVOL](#),
[IPCREATE_FILTER](#), [IPWRITE_FILTER](#)

GLCM Function

Computes the graylevel co-occurrence probability matrix (GLCM) for a graylevel image.

Usage

```
result = GLCM(image)
```

Input Parameters

image — A 2D array containing a graylevel image.

Returned Value

result — A floating-point array of the graylevel co-occurrence probabilities for *image*.

Keywords

Maxgray — The maximum graylevel in the image to consider. (Default: MAX(*image*))

Mingray — The minimum graylevel in the image to consider. (Default: MIN(*image*))

Sums — If set, returns a long array containing the graylevel co-occurrence sums matrix instead of the probabilities.

Xoffset — The pixel position offset in the *x*-direction. (Default: 1)

NOTE Either *Xoffset* or *Yoffset* must be nonzero.

Yoffset — The pixel position offset in the *y*-direction. (Default: 1)

Discussion

Each element (*a*, *b*) of the graylevel co-occurrence matrix is the joint probability that graylevel *b* is at a distance specified by *Xoffset*, and *Yoffset* from graylevel *a*. Performing statistics on the graylevel co-occurrence probability matrix (GLCM) provides quantitative information about image texture.

Example

```
image = IMAGE_READ(!IP_Data + 'texture.tif')
    ; Read a grayscale image.

glcm_array = GLCM(image('pixels'))
    ; Compute the GLCM.

TVSCL, glcm_array
    ; Display the glcm_array.

glcm_texture = GLCM_STATS(glcm_array)
    ; Compute the GLCM texture statistics for this image.
```

See Also

[GLCM_STATS](#), [GLRL](#), [GLRL_STATS](#), [HIST_STATS](#), [POLAR_FFT](#)

GLCM_STATS Function

Calculates five statistics on the graylevel co-occurrence matrix (the result of a call to the GLCM function).

Usage

result = GLCM_STATS(*glcm_matrix*[, *k*])

Input Parameters

glcm_matrix — The graylevel co-occurrence matrix computed by GLCM.

k — (optional) A scalar float that is the order of the element difference moment and the inverse element difference moment statistics. (Default: 1.0)

Returned Value

result — A five-element double array containing the following statistics: the maximum, the element difference moment of order *k*, the inverse element difference moment of order *k*, the entropy, and the uniformity.

Keywords

None.

Discussion

The first statistic in the returned array is the maximum value of the graylevel co-occurrence matrix (GLCM). The maximum is the greatest response to the *Xoffset* and *Yoffset* used in the GLCM computation.

The second statistic returned is the element difference moment of order *k*. This statistic is lowest when large values of the GLCM are clustered near the matrix diagonal. The element difference moment of order *k* is given by

$$\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} (i-j)^k \text{glcm_matrix}(i, j) ,$$

where *glcm_matrix* is an *N*-by-*M* array.

The third statistic returned is the inverse element difference moment of order *k*. The inverse element difference moment of order *k* is lowest when large GLCM values are located away from the matrix diagonal. The inverse element difference moment of order *k* is given by

$$\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \frac{\text{glcm_matrix}(i, j)}{(i-j)^k}, \text{ where } i \neq j$$

and *glcm_matrix* is an *N*-by-*M* array.

The GLCM entropy, the fourth element of the returned array indicates the “randomness” of the GLCM. The entropy is inversely proportional to the randomness; that is, the entropy is smallest for the greatest amount of randomness in the matrix, and is at a maximum when the elements of the GLCM are equal. (See the ENTROPY function for the applicable equation.)

The fifth statistic returned is the GLCM uniformity. The uniformity is minimum when all values of the GLCM are equal. (See the UNIFORMITY function for the applicable equation.)

Example

```
image = IMAGE_READ(!IP_Data + 'texture.tif')
; Read a grayscale image.

glcm_array = GLCM(image('pixels'))
; Compute the graylevel co-occurrence matrix (GLCM).

TVSCL, glcm_array
; Display the glcm_array.

glcm_texture = GLCM_STATS(glcm_array)
; Compute the GLCM texture statistics for this image.

PRINT, glcm_texture
; Print the texture statistics.
```

See Also

[ENTROPY](#), [GLCM](#), [GLRL](#), [GLRL_STATS](#), [HIST_STATS](#),
[POLAR_FFT](#), [UNIFORMITY](#)

GLRL Function

Computes the graylevel run length (GLRL) matrix used for textural analysis of an image.

Usage

result = GLRL(*image*[, *theta*])

Input Parameters

image — A 2D image of data type byte, integer, or long.

theta — (optional) A scalar integer specifying the angle (in degrees) at which to measure run lengths. The following values are valid for the *theta* parameter: 0, 45, 90, and 135 degrees. (Default: 0)

Returned Value

result — A 2D long array, *result*(*m*, *n*), where *m* is the number of graylevels considered and *n* is the run length, ranging from 0 to *N* - 1 for an *x*-by-*y* image.

For *theta* = 45, or 135 degrees, $N = (x^2 + y^2)^{1/2}$

For *theta* = 0, $N = x$

For *theta* = 90, $N = y$

The value at *result*(*n*, *m*) is the number of *m*-length runs at graylevel *n* + *Mingray*.

Keywords

Maxgray — The maximum gray value in the image to consider. (Default: MAX(*image*))

Mingray — The minimum gray value in the image to consider. (Default: MIN(*image*))

Omax — Specifies a variable to hold the maximum gray value used.

Omin — Specifies a variable to hold the minimum gray value used.

Discussion

Graylevel run lengths indicate texture directionality and coarseness in an image. Short graylevel runs indicate finer textures, whereas coarse textures result in long graylevel runs. A diagonal texture produces long graylevel runs at $\theta = 45$, and 135 degrees. Vertically and horizontally shifted textures produce long graylevel runs when $\theta = 0$ and 90 degrees.

Example

```
image = IMAGE_READ(!IP_Data + 'texture.tif')
    ; Read a grayscale image.

glrl_array = GLRL(image('pixels'))
    ; Compute the graylevel run length (GLRL).

SURFACE, glrl_array
    ; Display the glrl_array.

glrl_texture = GLRL_STATS(glrl_array, $
    image('width'), image('height'))
    ; Compute the GLRL texture statistics for this image.
```

See Also

[GLCM](#), [GLCM_STATS](#), [GLRL_STATS](#), [HIST_STATS](#),
[POLAR_FFT](#)

GLRL_STATS Function

Performs five statistical calculations on the graylevel run length matrix obtained by a call to the GLRL function.

Usage

result = GLRL_STATS(*glrl_matrix*, *xdim*, *ydim*)

Input Parameters

glrl_matrix — The graylevel run length matrix computed by GLRL.

xdim — The *x*-dimension, the number of columns in the image from the call to GLRL.

ydim — The *y*-dimension, the number of rows in the image from the call to GLRL.

Returned Value

result — A five-element double array containing the following GLRL statistics: the short run emphasis, the long run emphasis, the graylevel distribution, the run-length distribution, and the run percentages.

Keywords

None.

Discussion

The first statistic in the returned array is the short run emphasis. This is maximum for fine textured graylevel runs. The short run emphasis is given by

$$\frac{1}{T} \sum_{i=1}^N \sum_{j=0}^{M-1} \frac{1}{(i^2)} \text{glrl_matrix}(i-1, j) \quad , \quad \text{where } T = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \text{glrl_matrix}(i, j)$$

and *glrl_matrix* is an *N*-by-*M* array.

The second statistic returned is the long run emphasis. The long run emphasis is maximum for coarse textured graylevel runs. The long run emphasis is given by

$$- \sum_{i=1}^N \sum_{j=0}^{M-1} i^2 \text{glrl_matrix}(i-1, j), \text{ where } T = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \text{glrl_matrix}(i, j)$$

and *glrl_matrix* is an *N*-by-*M* array.

The third statistic returned is the graylevel distribution. The graylevel distribution provides an indication of the number of runs relative to the number of graylevels in the GLRL matrix. The graylevel distribution is given by

$$\frac{1}{T} \sum_{i=0}^{N-1} \left(\sum_{j=0}^{M-1} \text{glrl_matrix}(i, j) \right)^2, \text{ where } T = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \text{glrl_matrix}(i, j)$$

and *glrl_matrix* is an *N*-by-*M* array.

The fourth statistic returned is the run-length distribution. The run-length distribution indicates the occurrence of run-lengths relative to the number of graylevels in the GLRL matrix. The run-length distribution is given by

$$\frac{1}{T} \sum_{j=0}^{M-1} \left(\sum_{i=0}^{N-1} \text{glrl_matrix}(i, j) \right)^2, \text{ where } T = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \text{glrl_matrix}(i, j)$$

and *glrl_matrix* is an *N*-by-*M* array.

The fifth statistic returned is the GLRL run percentages. The run percentages are given by

$$\frac{1}{xdim \cdot ydim} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \text{glrl_matrix}(i, j)$$

and *glrl_matrix* is an *N*-by-*M* array.

Example

```
image = IMAGE_READ(!IP_Data + 'texture.tif')
; Read a grayscale image.

glrl_array = GLRL(image('pixels'))
; Compute the graylevel run length (GLRL).

TVSCL, glrl_array
; Display the glrl_array.
```

```
glrl_texture = GLRL_STATS(glrl_array, $
    image('width'), image('height'))
    ; Compute the GLRL texture statistics for this image.
PRINT, glrl_texture
    ; Print the texture statistics.
```

See Also

[GLCM](#), [GLCM_STATS](#), [GLRL](#), [HIST_STATS](#), [POLAR_FFT](#)

HAAR Function

Performs a Haar transform on a square image. If the input image is not square, it is padded with zeros to make it square before the transform is performed. Images whose dimensions are not a power of two are padded to have dimensions that are the nearest power of two.

Usage

```
result = HAAR(image[, direction])
```

Input Parameters

image — A 2D or 3D array containing an image; or image, row, or pixel-interleaved images.

direction — (optional) Specifies the direction of the transform.

- 1 Forward transform (default)
- 1 Backward transform

Returned Value

result — A 2D or 3D square floating-point matrix, whose dimensions are N -by- N where N is the largest dimension of the input parameter *image* rounded to the nearest larger power of two.

Keywords

Intleave — A scalar string indicating the type of interleaving of 3D input image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

Discussion

The Haar transform uses the Haar basis functions, $h_k(z)$, which are defined in the interval $z \in [0, 1]$, and for $k = 0, 1, 2, \dots, N - 1$, where N is a power of 2. The following equations define the Haar functions:

$$k = 2^p + q - 1$$

where p is in the range $0 \leq p \leq n - 1$; and $q = 0$ or 1 for $p = 0$; and $1 \leq q \leq 2^p$ when $p \neq 0$.

$$h_0(z) = h_{00}(z) = \frac{1}{\sqrt{N}}, \text{ and}$$

$$h_k(z) = h_{pq}(z) = \frac{2^{p/2}}{\sqrt{N}} \quad \text{for} \quad \left(\frac{q-1}{2^p} \leq z < \frac{q-1/2}{2^p} \right)$$

$$h_k(z) = h_{pq}(z) = \frac{-2^{p/2}}{\sqrt{N}} \quad \text{for} \quad \left(\frac{q-1/2}{2^p} \leq z < \frac{q}{2^p} \right)$$

$$h_k(z) = h_{pq}(z) = 0 \quad \text{otherwise,}$$

for $z \in [0, 1]$.

A Haar matrix is formed from the elements of $h(z)$, with the j -th row of the matrix formed from elements of $h_j(z)$ for $z = 0/N, 1/N, 2/N, \dots, (N-1)/N$.

For example, the 2-by-2 Haar matrix is:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

The Haar transform is then applied as:

$T = H * image * H$, where H is the Haar matrix and T is the transform result.

Example

```
image = IMAGE_READ(!IP_Data + 'airplane.tif')
        ; Read an image.
haar_image = HAAR(image('pixels'))
        ; Compute the Haar transform.
TVSCL, IPALOG(haar_image)
        ; Display the transformed image.
```

See Also

[DCT](#), [SLANT](#)

HIST_STATS Function

Computes six different statistical operations on image histograms.

Usage

result = HIST_STATS(*hist_data*)

Input Parameters

hist_data — A 1D or 2D long array containing one or more histograms.

Returned Value

result — A double array with six statistical results (mean, variance, skewness, kurtosis, energy, and entropy, respectively) for each histogram in the *hist_data* array.

Keywords

Binsize — The bin size used to compute the histogram. (Default: 1.0)

Maxgray — Specifies the maximum graylevel that was considered in computing the histogram values (*hist_data*).
(Default: N_ELEMENTS(*hist_data*))

Mingray — Specifies the minimum graylevel that was considered in computing the histogram values (*hist_data*). (Default: 0)

Discussion

Histogram statistics are useful for quantitative image feature descriptions. The statistics returned by HIST_STATS are computed for the normalized image histogram, P , where P is the histogram divided by its total.

The statistics returned by HIST_STATS are defined as follows:

Mean:

$$M = \sum_{i=0}^{L-1} iP(i)$$

Standard Deviation:

$$D = \left[\sum_{i=0}^{L-1} (i-M)^2 \right]^{1/2}$$

Skewness:

$$S = \frac{1}{\sigma^3} \sum_{i=0}^{L-1} (i-M)^3 P(i)$$

Kurtosis:

$$K = \frac{1}{\sigma^4} \sum_{i=0}^{L-1} (i-M)^4 P(i) - 3$$

Energy:

$$N = \sum_{i=0}^{L-1} [P(i)]^2$$

Entropy:

$$E = - \sum_{i=0}^{L-1} P(i) \log_2[P(i)]$$

Example

```
image = IMAGE_READ(!IP_Data + 'airplane.tif')
; Read an image.

image_hist = IPHISTOGRAM(image('pixels'))
stats = HIST_STATS(image_hist)

PRINT, stats
; Compute the histogram statistics and print them
; to the screen.
```

See Also

[IPHISTOGRAM](#)

HIT_MISS Function

Performs the morphologic hit-or-miss transform for shape processing.

Usage

```
result = HIT_MISS(image, hit_structure, miss_structure  
[, xhit, yhit][, xmiss, ymiss])
```

Input Parameters

image — A 2D array.

hit_structure — A 1D or 2D array containing the structuring element to apply to the original (un-complemented) *image*. The structuring elements are considered to be binary values (either 0 or nonzero), unless the *Gray* keyword is specified.

miss_structure — A 1D or 2D array containing the structuring element to apply to the complemented *image*. The structuring elements are considered to be binary values (either 0 or nonzero), unless the *Gray* keyword is specified.

xhit — (optional) The *x*-coordinate of the origin of *hit_structure*.

yhit — (optional) The *y*-coordinate of the origin of *hit_structure*.

xmiss — (optional) The *x*-coordinate of the origin of *miss_structure*.

ymiss — (optional) The *y*-coordinate of the origin of *miss_structure*.

Returned Value

result — The hit-or-miss transformed image, a byte array of the same size and dimensions as *image*.

Keywords

Gray — If set, grayscale erosion is used rather than binary erosion.

Hit_Values — An array of the same dimensions and number of elements as *hit_structure*, containing the hit-structuring element values.

Miss_Values — An array of the same dimensions and number of elements as *miss_structure*, containing the miss-structuring element values.

Discussion

Morphological operations are defined for grayscale byte images. If *image* is not originally of type byte, PV-WAVE makes a temporary copy of *image* that is of type byte before using it for the morphological processing.

The morphological hit-or-miss transform is useful for object and character recognition, because it can be used to identify features of a binary object. The hit-or-miss operator is defined as the intersection between the eroded image (eroded using the *hit_structure* parameter) with the eroded image complement (using the *miss_structure* parameter). Optimum results are obtained when *hit_structure* and *miss_structure* are disjoint. In other words, nonzero values of *hit_structure* are zero in *miss_structure*, and vice-versa.

Example

```
hit_struct = BYTARR(5, 5)
hit_struct(*) = 1B
hit_struct(1:3, 1:3) = 0B
miss_struct = NOT(hit_struct) - 254B
    ; Make a structuring element pair.
test_image = IMAGE_READ(!IP_Data + 'squares.tif')
    ; Read an image.
hitmiss_image = HIT_MISS(test_image('pixels'), $
    hit_struct, miss_struct)
    ; Find the squares matching the structuring elements.
```

See Also

[MORPH_CLOSE](#), [MORPH_OPEN](#), [MORPH_OUTLINE](#),
[SKELETONIZE](#), [TOP_HAT](#)

In the *PV-WAVE Reference*: [DILATE](#), [ERODE](#)

HOUGH Function

Computes the line or circle Hough transform of an image.

Usage

```
result = HOUGH(image[, radius][, thresh])
```

Input Parameters

image — A 2D or 3D array containing an image; or image, row or pixel-interleaved images.

radius — (optional) The radius of the circle for the Hough circle transform.

thresh — (optional) The threshold value for *image*, if *image* is not already a binary image. Values in *image* that are greater than or equal to *thresh* are used for the Hough transform calculation.

Returned Value

result — The Hough accumulator array.

For a 2D *image* input parameter, *result* is a 2D long array whose dimensions are the diagonal of the original image by the value of the *N_Angles* keyword.

For a 3D *image* array (an array of 2D images), *result* is a 3D long array whose dimensions are the diagonal of the original image by the value of the *N_Angles* keyword by the number of images in the array.

Keywords

Intleave — A scalar string indicating the type of interleaving of 3D input image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

N_Angles — Specifies the number of angles ($0 \leq N_Angles \leq 360$), to quantize for the Hough accumulator array. (Default: 360)

Discussion

The line Hough transform is based on the parametric description of a line in the polar coordinate system, as in the following equation:

$$\rho = x \cos \theta + y \sin \theta ,$$

where ρ is the distance of the line from the origin, θ is the angle of the line with respect to the x -axis, and x and y are the Cartesian coordinates of a pixel in *image*. The transformation consists of filling an accumulator array, $H(\rho, \theta)$, where ρ is calculated for each nonzero xy -pixel in *image* for $0 \leq \theta < 360$.

The circle Hough transform is based on the description of a circle, as in the following equation:

$$\rho^2 = (x - a)^2 + (y - b)^2 ,$$

where ρ is the radius of the circle, a and b are the center of the circle; and x and y are the Cartesian coordinates of a pixel in *image*.

This equation can be broken down into the parametric equations:

$$a = \rho \cos \theta + x$$

$$b = \rho \sin \theta + y$$

The circle Hough transform consists of an accumulator array $H(a, b)$ where a and b are computed for each nonzero xy -pixel in *image* for $0 \leq \theta < 360$ that is filled.

Edge linking is one useful chore that the line Hough transform can perform. Local maxima in the Hough accumulator array correspond to straight lines in the transformed image. By locating maxima in the Hough array, the edge-enhanced image is placed on top of the corresponding straight lines, thus linking any unclosed edges.

The circle Hough transform, on the other hand, allows you to identify points in an image which lie on a circle of a given radius. Again, these points correspond to local maxima in the Hough accumulator array.

Example 1

The following example illustrates the use of the Hough line transform.

```
image = IMAGE_READ(!IP_Data + 'airplane.tif')
    ; Read an image.
hough_line = HOUGH(image('pixels'), 100)
    ; Compute the line Hough transform.
TVSCL, hough_line
    ; Display the Hough transform.
max_hough = THRESHOLD(hough_line, 0.95 * $
    MAX(hough_line), /Binary)
    ; Find the maximum values in the Hough line transform,
    ; corresponds to straight lines.
TVSCL, max_hough
    ; Display only the maximums.
```

Example 2

This example illustrates the use of the Hough circle transform.

```
image = IMAGE_READ(!IP_Data + 'cells.tif')
    ; Read an image.
IMAGE_DISPLAY, image
```

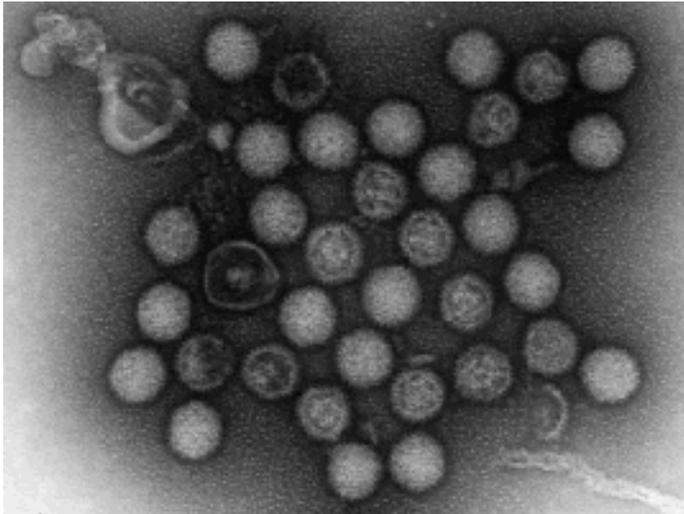


Figure 2-11 The image of cells.

```

hough_circle = HOUGH(image('pixels'), 11.0, 75, /Circle)
    ; Compute the circle Hough transform at a radius of 20 pixels.
hough_circle = NOT(BYTSC(hough_circle))
TVSCL, hough_circle
    ; Display the Hough transform.

```

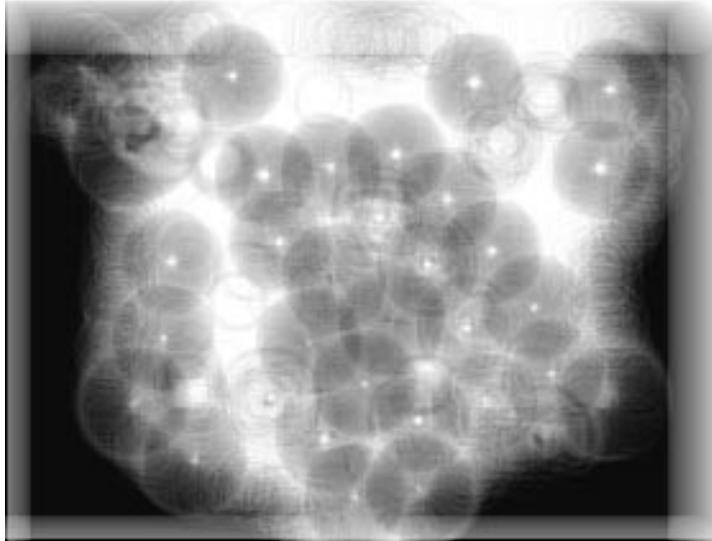


Figure 2-12 The circle Hough transform with a 20 pixel radius.

```

max_hough = THRESHOLD(hough_circle, 0.90 * $
    MAX(hough_circle), /Binary)
    ; Find the maximum values in the Hough circle transform,
    ; corresponds to circles in the image of radius 20.0.
TVSCL, max_hough
    ; Display only the maximums.

```

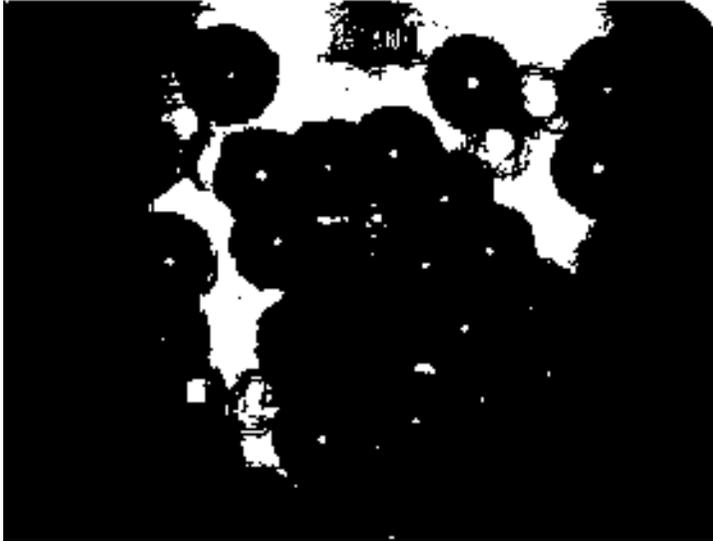


Figure 2-13 The Hough transform maxima.

See Also

[HAAR](#), [RADON](#), [SLANT](#)

IPALOG Function

Computes the natural logarithm of an image, excluding zero values.

Usage

$result = IPALOG(image)$

Input Parameters

image — An array of any data type except string.

Returned Value

result — The logarithm to the base e (the natural log) of *image*, excluding zero values. For double-precision floating-point and complex values *result* is returned with the same data type. All other valid data types are converted to single-precision floating-point and *result* is returned as a floating-point data type.

Keywords

None.

Discussion

The IPALOG function is useful for displaying images with wide dynamic range and is defined as follows:

$$y = \log_e x$$

Values in *image* which are zero are left as zero.

The IPALOG function handles complex numbers in the following way:

$$IPALOG(image) \equiv Complex(\log_e(|image|), \arctan(image))$$

TIP When error messages regarding the handling of zero values in the input are not wanted, you should use the IPALOG function instead of the PV-WAVE ALOG function.

Example

```
x = DIST(256)
    ; Generate a test image.

f = FFT(x, -1)
    ; Take the fast Fourier transform of x.

TVSCL, IPALOG(ABS(f))
    ; Display the log magnitude of the image FFT,
    ; with zero values retained as zeros.
```

See Also

In the *PV-WAVE Reference*: ALOG

IPCLASSIFY Function

Performs supervised classification using the maximum likelihood classifier.

Usage

```
result = IPCLASSIFY(image, training_pixels[, threshold,
probability])
```

```
result = IPCLASSIFY(image, class_params)
```

Input Parameters

image — A 2D or 3D array of any type except string or complex that contains an image; or image, row, or pixel-interleaved images.

training_pixels — Training information for the desired classification. This parameter is an associative array with the following keys:

'pixels' — A list of pixel-element number arrays, each defining a separate training region.

'class_numbers' — An integer array of class numbers greater than 0 corresponding to the training regions listed in **'pixels'**.

threshold — (optional) A floating-point array containing the threshold for each class.

probability — (optional) A floating-point array containing the probability for each class.

class_params — The statistical parameters for each class. This parameter is an associative array with the following keys:

'class' — An integer array containing the class number (greater than 0) for each class.

'covariance' — A list of floating-point arrays (for 3D input images) or an array of floating-point values (for 2D input images) that are the covariances for each class.

'mean' — A list of floating-point arrays (for 3D input images) or an array of floating-point values (for 2D input images) that are the mean vectors for each class.

'probability' — (optional) A floating-point array containing the probability for each class.

'threshold' — (optional) A floating-point array containing the threshold for each class.

Returned Value

result — A 2D array of byte or integer data type containing the class number of each pixel in *image*.

Keywords

Class_Stats — Specifies a variable to hold the classification statistics determined for the image. The *Class_Stats* keyword is an associative array with the following keys:

'class' — An integer array containing the class number (greater than 0) for each class.

'covariance' — A list of floating-point arrays (for 3D input images) or an array of floating-point values (for 2D input images) that are the covariances for each class.

'mean' — A list of floating-point arrays (for 3D input images) or an array of floating-point values (for 2D input images) that are the mean vectors for each class.

'probability' — A floating-point array containing the probability for each class.

'threshold' — A floating-point array containing the threshold for each class.

Intleave — A scalar string indicating the type of interleaving of 3D input image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

Discussion

IPCLASSIFY uses the maximum likelihood decision rule to classify each pixel in an image. The decision-rule is based on discriminant functions, $g_i(\mathbf{x})$ for each pixel position \mathbf{x} and each class ω_i :

$$g_i(\mathbf{x}) = \ln(p(\mathbf{x}|\omega_i)) + \ln(p(\omega_i))$$

Classification is performed according to:

$$\mathbf{x} \in \omega_i \text{ if } p(\omega_i|\mathbf{x}) > p(\omega_j|\mathbf{x}) \text{ for all } j \neq i$$

and

$$g_i(\mathbf{x}) > T_i .$$

Assuming normal statistics, the discriminant function for maximum likelihood classification is represented as follows:

$$g_i(x) = \ln(p(\omega_i)) - \frac{1}{2} \ln|\Sigma_i| - \frac{1}{2} (x - m_i)^t \Sigma_i^{-1} (x - m_i) \quad ,$$

where Σ_i is the covariance matrix, m_i is the mean vector, $p(\omega_i)$ is the probability, and T_i is the threshold for class i . The default case is for equal prior probabilities. In other words, for M total classes,

$$p(\omega_i) = 1.0/M \text{ for all } i$$

The default threshold values are chosen such that 95% of all pixels in a class will be classified, based on a Chi-squared distribution:

$$T_i = -4.744 - \frac{1}{2} \ln |\Sigma_i| + \ln(p(\omega_i))$$

Example

```
image = IMAGE_READ(!IP_Data + 'boulder_image.tif')
; Read an image.

training_pixels = ASARR('pixels', LIST(region1, $
    region2, region3), 'class_numbers', [1, 2, 3])
; Make an associative array of training information.
; Region1, region2, and region3 are long arrays
; that contain pixel numbers. Equal prior probabilities
; will be used as well as default class thresholds.

class_image = IPCLASSIFY(image('pixels'), $
    training_pixels, class_stats = class_stats)
; Classify the image using a maximum likelihood classifier.

TVSCL, class_image
; Display the classified image.

land_image = IMAGE_READ(!IP_Data + 'landsat.tif')
class_image2 = IPCLASSIFY(land_image('pixels', $
    Class_Stats)
; New images can now be classified using the
; Class_Stats keyword.

TVSCL, class_image2
; Display the newly classified image.
```

See Also

[IPCLUSTER](#), [REGION_GROW](#), [REGION_MERGE](#),
[REGION_SPLIT](#), [THRESH_ADAP](#), [THRESHOLD](#)

IPCLUSTER Function

Performs image segmentation using K-means clustering based on regional statistical measures of the mean, mode, minimum, maximum, and/or range of the image pixels.

Usage

result = IPCLUSTER(*image*, *cluster_seeds*)

Input Parameters

image — A 2D or 3D array of any data type, except string or complex, containing an image; or image, row, or pixel-interleaved images.

cluster_seeds — A long array containing the pixel element numbers in *image*. These are used as the seed points with which individual clusters are identified. ($2 \leq \text{cluster_seeds} \leq \text{total pixels in image}$)

Returned Value

result — A 2D or 3D array of byte or integer data type containing the cluster number of each pixel in image.

Keywords

Fill_Mean — If set, clusters are filled with the mean value of the image pixels in that cluster.

Fill_Values — An array of values greater than 0 and less than or equal to the number of cluster seeds which are used to fill the clusters.

Intleave — A scalar string indicating the type of interleaving of 3D input image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

Maximum — If set, the maximum of each window (as defined using the *Wxdim* and *Wydim* keywords) in *image* is used as a statistic for the clustering algorithm.

Max_iter — The maximum number of iterations to use in the clustering algorithm. (Default: 30)

Mean — If set, the mean of each window (as defined using the *Wxdim* and *Wydim* keywords) in *image* is used as a statistic for the clustering algorithm.

Minimum — If set, the minimum of each window (as defined using the *Wxdim* and *Wydim* keywords) in *image* is used as a statistic for the clustering algorithm.

Mode — If set, the mode of each window (as defined using the *Wxdim* and *Wydim* keywords) in *image* is used as a statistic for the clustering algorithm.

Range — If set, the range of each window (as defined using the *Wxdim* and *Wydim* keywords) in *image* is used as a statistic for the clustering algorithm.

Value — If set, the pixel value is used as a statistic for the clustering algorithm.

Wxdim — The window width used for computing the *image* statistics. (Default: 3)

Wydim — The window height used for computing the *image* statistics. (Default: 3)

Discussion

The IPCLUSTER function computes a measurement vector for each individual pixel in *image*. The following keywords control the statistical measures used for the clustering algorithm: *Mean*, *Mode*, *Minimum*, *Maximum*, *Range*, and *Value*. If no statistical keywords are specified, the *Mean* and *Mode* are used as the default.

The IPCLUSTER function is a wrapper for the K_MEANS function. The K_MEANS function is a *PV-WAVE:IMSL Statistics* routine used to identify clusters in *image* based on similar statistical features. The K_MEANS function computes Euclidean metric clusters for the measurement vectors. This begins with initial estimates of the mean values of the clusters determined from the *cluster_seed* points.

NOTE The K_MEANS function requires that each cluster seed have a unique statistical property. The following informational message may appear when using the IPCLUSTER function:

```
% Number of clusters reduced due to  
identical statistical measures.
```

This indicates that some of the statistical measures for the cluster seeds are identical and have been eliminated from the clustering process.

Example

```
image = IMAGE_READ(!IP_Data + 'xray.tif')  
    ; Read an image.  
IMAGE_DISPLAY, image
```



Figure 2-14 A chest X-ray image.

```

cluster_seeds = [20L, 15L + image('width') * 30L, $
  137L + image('width') * 200L, $
  12L + 190 * image('width')]
; Pick some random points for cluster seeds.

seg_image = IPCLUSTER(image('pixels'), $
  cluster_seeds, /Mean, /Mode, /Maximum, $
  /Fill_Mean)
; Segment the image using K_MEANS clustering.

TVSCL, seg_image
; Display the segmented image.

```

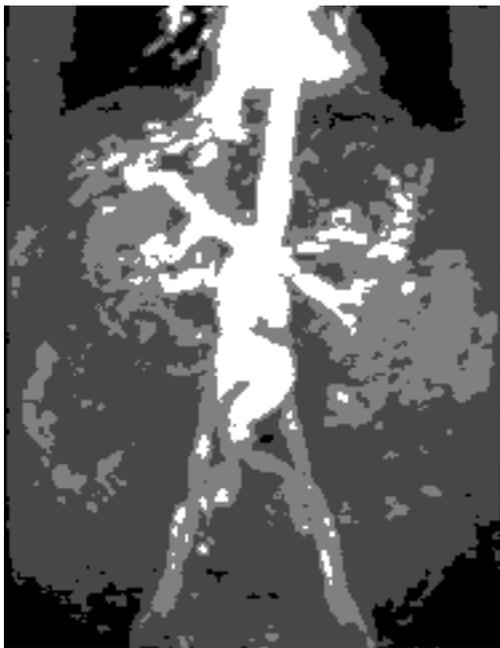


Figure 2-15 The segmented X-ray image.

See Also

[REGION_GROW](#), [REGION_MERGE](#), [REGION_SPLIT](#),
[THRESH_ADAP](#), [THRESHOLD](#)

In the *PV-WAVE: IMSL Statistics Reference*: [K_MEANS](#)

IPCOLOR_24_8 Function

Converts a 24-bit color image to an 8-bit color image; or, converts an 8-bit color image to a 24-bit color image.

Usage

```
result = IPCOLOR_24_8(image[, in_cmap])
```

Input Parameters

image — A 2D or 3D array containing an image; or image, row or pixel-interleaved images.

in_cmap — (Used for 8 to 24-bit conversion only.) A 3-by-*n_colors* array containing the colormap for *image*, where *n_colors* ≤ 256.

Returned Value

result — A 2D or 3D byte array containing the converted image.

If *image* is 3D, it is assumed to contain a 24-bit color image; therefore, the conversion is from 24-bit color to 8-bit color, and *result* is an 8-bit color image (a 2D byte array of the same *x* and *y*-dimensions as *image*).

If *image* is 2D, it is assumed to contain an 8-bit color image; therefore, the conversion is from 8-bit color to 24-bit color, and *result* is an 24-bit color image (a 3D byte array of the same *x* and *y*-dimensions as *image*).

Keywords

Valid For Both Types of Conversion:

Bmap — Specifies a variable to hold the blue colormap created for the converted image.

Gmap — Specifies a variable to hold the green colormap created for the converted image.

Intleave — (For 3D *image*: 24-bit to 8-bit conversion) A scalar string indicating the type of interleaving of the 3D input array.

(For 2D *image*: 8-bit to 24-bit conversion) A scalar string specifying the interleav-

ing of the 24-bit *result* image. Valid strings and the corresponding interleaving methods are:

'*pixel*' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'*row*' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'*image*' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

Noloadct — If set, the colormap created during conversion is not automatically loaded.

Rmap — Specifies a variable to hold the red colormap created for the converted image.

For 24 to 8-Bit Conversion Only:

N_Colors — The number of colors used in the 8-bit *result* image. (Default: !D.Table_Size)

One of the following keywords should be set; the *MedCut* keyword is the default, if none is selected.

Floyd — If set, Floyd-Steinberg dithering is used to quantize a 24-bit image to 8 bits.

MedCut — Is set, the median cut algorithm is used to quantize a 24-bit image to 8 bits. (Default: set)

Pctrans — If set, the principle components transform is used to quantize a 24-bit image to 8 bits.

For 8 to 24-Bit Conversion Only:

Linear — If set, specifies linear ramps from 0 to 255 for the red, green, and blue color tables; otherwise, the current PV-WAVE color table is used. Valid only if the *in_cmap* parameter is not specified.

Discussion

Converting an image from 24-bit to 8-bit is called color quantization and is useful for displaying 24-bit images on 8-bit displays. It's also convenient to compress the image information from three planes into a single plane to reduce the computation time needed for image processing operations.

NOTE Information is lost during the conversion from 24-bit to 8-bit, which means the conversion is not fully reversible.

Conversion from 8-bit to 24-bit is sometimes a useful trick for processing the image as three separate planes. It is also useful for loading a mixture of 8-bit and 24-bit images into the Image Tool on the Image Processing Navigator by converting the 8-bit images to 24-bit before starting the Image Tool.

Example 1: 8-Bit to 24-Bit Conversion

Converting an image from 8-bit to 24-bit is sometimes convenient as a “trick” for processing the image as separate layers. For example, you can conveniently apply three different threshold ranges to a 24-bit image as follows:

```
image_8bit = IMAGE_READ(!IP_Data + 'photo.tif')
    ; Read in an 8-bit image.

image_24bit = IPCOLOR_24_8(image_8bit('pixels'),$
    image_8bit('colormap'))
    ; Convert the 8-bit image to 24 bit.

thresh_planes = THRESHOLD(image_24bit, $
    [20, 30, 25], [50, 45, 100], Intleave = 'image')
    ; Threshold each plane separately as follows:
    ; Plane 1: 20 <= x <= 50
    ; Plane 2: 30 <= x <= 45
    ; Plane 3: 25 <= x <= 100.

WINDOW, 0
TVSCL, thresh_planes(*, *, 0)
WINDOW, 1
TVSCL, thresh_planes(*, *, 1)
WINDOW, 2
TVSCL, thresh_planes(*, *, 2)
    ; Now, display each plane separately.
```

Example 2: 24-Bit to 8-Bit Conversion

```
image_24bit = IMAGE_READ(!IP_Data + $
    'boulder_image.tif')
    ; Read in a 24 bit image

image_8bit = IPCOLOR_24_8(image_24bit('pixels'),$
    /Floyd, Rmap = r, Gmap = g, Bmap = b)
```

```
    ; Convert the image to 8 bit for display, using the
    ; median cut algorithm and Floyd-Steinberg dithering.
TVLCT, r, g, b
    ; Set the device to 8-bit pseudocolor.
TV, image_8bit
    ; Display the image.
```

See Also

[PCT](#)

In the *PV-WAVE Reference*: `IMAGE_COLOR_QUANT`

IPCONVOL Function

Performs 1D, 2D, or 3D convolution on signals, images, and volumes.

Usage

```
result = IPCONVOL(image, kernel[, scale_factor])
```

Input Parameters

image — A 1D, 2D, or 3D array of any data type except string or complex that contains a signal; point or signal-interleaved signals; an image; image, row or pixel interleaved images; or a volume.

kernel — The array used to convolve each value in *image*. This parameter can be of any data type except string. The *kernel* parameter can also be a spatial filter object. (See the *Returned Value* description for the [IPREAD_FILTER](#) function for information about the filter spatial object.) The dimensions of *kernel* must be less than the dimensions of *image*.

scale_factor — (optional) A scale factor used to reduce the size of the output *result* by scaling the weighting factors in the filter kernel. This parameter is ignored if *kernel* is a filter object.

Returned Value

result — An array of the same dimensions and size as *image*; however, the size of the array may be affected by using the *Edge* keyword, or the *scale_factor* parameter.

Keywords

Edge — A scalar string indicating how edge effects are handled. (Default: 'zero') Valid strings are:

'pad' — The input image is padded before the filtering operation with the value specified using the *Pad_Value* keyword.

'zero' — Sets the border of the output image to zero. (Default)

'copy' — Copies the border of the input image to the output image.

'reduce' — Returns a reduced-size image that is smaller than the input image by the kernel dimensions.

Intleave — A scalar string indicating the type of interleaving of 2D input signals and 3D image arrays. Valid strings and the corresponding interleaving methods are:

'point' — The 2D input array arrangement is (p, x) for p point-interleaved signals of length x .

'signal' — The 2D input *image* array arrangement is (x, p) for p signal-interleaved signals of length x .

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

No_Clip — If set, clipping is avoided by setting the output image type large enough to contain the maximum of the two combined images.

TIP The *No_Clip* keyword prevents underflow or overflow conditions from occurring.

No_Mirror — If set, the kernel is not mirrored for the convolution operation. (Default: kernel mirrored before convolution)

Pad_Value — The value to use for the image padding. (Default: 0)

NOTE The *Pad_Value* keyword is only valid when *Edge* = 'pad'.

Spot — This keyword specifies explicit positioning of the element number of the filter window “sweet spot.” The *Spot* keyword default depends on the setting of the *Edge* keyword as shown in the following table:

Edge Keyword	Spot Keyword	Spot Keyword Default Setting
not used (Default: 'zero')	As specified, or the default.	kernel center
'pad'	Ignored.	kernel center
'zero'	As specified, or the default.	kernel center
'copy'	As specified, or the default.	kernel center
'reduce'	Ignored.	<i>Spot</i> = 0

Zero_Negatives — If set, all negative values in the result image are set to zero.

Discussion

Convolution is a general purpose method used for smoothing and blurring an image, as well as edge detection, shifting, and other filtering functions. Convolution is often performed in conjunction with other image processing functions such as the `FILT_NONLIN` and `FILT_SMOOTH` functions.

NOTE Equations describing the convolution process can be found in the `CONVOL` function description in the *PV-WAVE Reference*.

The *kernel* parameter can be either a filter object (see the *Returned Value* description for the `IPREAD_FILTER` function for information about the filter object) or an array. The dimensions of the filter kernel describe the size of the neighborhood surrounding each value in *image* that is analyzed. The filter kernel also includes weighting values for each point in the array. These weighting values are used to determine the average value returned in the *result* array.

When *kernel* is a filter object, the *scale_factor* parameter is not used; however, there is a *scale_factor* key in the *kernel* associative array. A typical use of the *scale_factor* parameter is to set it equal to the sum of the filter kernel values. In this way, the amplitude gain of the filter is normalized to 1.0.

In many image processing applications, the kernel “sweet spot,” that is, the positioning of the filter kernel with respect to the underlying image pixels, is important. The *Spot* keyword allows exact positioning of the filter kernel as it is convolved with the image pixels, and is typically located at the center of the filter kernel. Other commonly used kernel sweet spots are the first element or the last element of the kernel array.

Controlling edge effects is important in many image processing applications as well, and this can be accomplished by specifying a control method with the *Edge* keyword. Edge effects occur where the filter kernel array “overhangs” the image. Two methods most commonly used to deal with edge effect behavior are copying the input *image* edges to the *result* image, or zeroing the edges of the *result* image: accomplished with the ‘*copy*’ and ‘*zero*’ strings, respectively, for the *Edge* keyword. Padding the input *image* before convolution and reducing the size of the *result* image are other methods for edge effects. These methods are accomplished with the ‘*pad*’ and ‘*reduce*’ strings, respectively, for the *Edge* keyword.

The filter kernel array is always mirrored before the convolution operation. To disable this behavior, use the *No_Mirror* keyword.

Example

```
image = IMAGE_READ(!IP_Data + 'dollars.tif')
    ; Read in an image.

kernel = IPREAD_FILTER(!IP_Data + 'kernel/kirsch_sw3.ker')
    ; Read in a filter kernel for edge detection.

edge_image = IPCONVOL(image('pixels'), kernel, $
    Spot = 0, Edge = 'reduce')
    ; Apply the kernel, with a sweet spot in the upper left
    ; corner. Reduce the output image, getting rid of edge
    ; effects.

TVSCL, edge_image
    ; Display the edge-enhanced image.
```

See Also

[FILT_NONLIN](#), [FILT_SMOOTH](#), [IPCORRELATE](#),
[IPCREATE_FILTER](#), [IPREAD_FILTER](#)

IPCORRELATE Function

Performs direct (spatial domain) or indirect (spatial frequency domain) correlation between an array and a template.

Usage

result = IPCORRELATE(*image*, *template*)

Input Parameters

image — A 1D, 2D, or 3D array of any data type except string or complex that contains a signal; point or signal-interleaved signals; an image; image, row or pixel interleaved images; or a volume.

template — A 1D, 2D, or 3D array of any data type except string containing the template used for correlation. The *template* parameter can also be a filter object. (See the [IPREAD_FILTER](#) function for information about the filter objects.)

Returned Value

result — An array of the same dimensions and size as *image*, unless otherwise affected by use of the *Edge* keyword.

Keywords

Direct — If set, direct method correlation is performed in the spatial domain using convolution. For the direct method, the *template* parameter may be either an array or a spatial filter object (see the [IPREAD_FILTER](#) function for more information on the spatial filter object associative array format).

Edge — A scalar string indicating how edge effects are handled. (Default: 'zero')

NOTE The *Edge* keyword is only valid when using the direct method.

Valid strings are:

'pad' — The input image is padded before the filtering operation with the value specified using the *Pad_Value* keyword.

'zero' — Sets the border of the output image to zero. (Default)

'copy' — Copies the border of the input image to the output image.

'reduce' — Returns a reduced-size image that is smaller than the input image by the kernel dimensions.

Intleave — A scalar string indicating the type of interleaving of 2D input signals and 3D image arrays. Valid strings and the corresponding interleaving methods are:

'point' — The 2D input array arrangement is (p, x) for p point-interleaved signals of length x .

'signal' — The 2D input *image* array arrangement is (x, p) for p signal-interleaved signals of length x .

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

No_Clip — If set, clipping is avoided by setting the output image type large enough to contain the maximum of the two combined images.

TIP The *No_Clip* keyword prevents underflow or overflow conditions from occurring.

NOTE The *No_Clip* keyword is only valid when using the direct method.

Pad_Value — The value to use for the image padding. (Default: 0)

NOTE The *Pad_Value* keyword is only valid when *Edge* = 'pad', and when using the direct method.

Spot — This keyword specifies explicit positioning of the element number of the filter window “sweet spot.” The *Spot* keyword default depends on the setting of the *Edge* keyword as shown in the following table:

<i>Edge</i> Keyword	<i>Spot</i> Keyword	<i>Spot</i> Keyword Default Setting
not used (Default: 'zero')	As specified, or the default.	kernel center
'pad'	Ignored.	kernel center
'zero'	As specified, or the default.	kernel center
'copy'	As specified, or the default.	kernel center
'reduce'	Ignored.	<i>Spot</i> = 0

NOTE The *Spot* keyword is only valid when using the direct method.

Zero_Negatives — If set, all negative values in the result image are set to zero.

Discussion

Correlation is often used for template matching in conjunction with other Image Processing Toolkit functions such as THRESHOLD to locate maxima in the correlation output. Correlation is computed in either the spatial or the spatial frequency domain. The spatial frequency domain computation is the default method for IPCORRELATE. This method is used most often, and performs correlation as the multiplication of the FFT of the image with the conjugate of the FFT of the correlation template. The spatial domain computation, also called the direct method, performs correlation as the convolution operation without mirroring the filter kernel. The spatial domain computation is accomplished by using the *Direct* keyword in the calling sequence.

NOTE For equations describing the direct method correlation process, see CONVOL in the *PV-WAVE Reference*.

Example

```
image = IMAGE_READ(!IP_Data + 'vnitext.tif')
; Read in an image.
```

```

template = IMAGE_READ(!IP_Data + 'letter_n.tif')
    ; Create a template for object recognition.

corr_image = IPCORRELATE(image('pixels') / $
    MAX(image('pixels'), $
    template('pixels') / MAX(image('pixels'))))
    ; Correlate the image with the template.

TVSCL, corr_image
    ; Display the correlation image.

max_value = MAX(corr_image, object_loc)
    ; The maximum value in the correlation image corresponds
    ; to the location of the template in the image.

PRINT, object_loc

```

See Also

[FILT_NONLIN](#), [FILT_SMOOTH](#), [IPCONVOL](#),
[IPCREATE_FILTER](#), [IPREAD_FILTER](#)

In the *PV-WAVE Reference*: [CONVOL](#)

IPCREATE_FILTER Function

Creates a spatial filter object, given a kernel and other appropriate fields.

Usage

```
result = IPCREATE_FILTER(kernel [, scale_factor])
```

Input Parameters

kernel — A 2D array containing the filter kernel.

scale_factor — (optional) A scalar float containing the scaling factor of the filter.

Returned Value

result — An associative array containing a spatial filter object. (See the *Returned Value* description for the [IPREAD_FILTER](#) function for information about the spatial filter object.)

Keywords

None.

Discussion

The `IPCREATE_FILTER` function takes an input array and puts it into the spatial filter object format. Spatial filter objects are used with the `IPCONVOL` and `IPCORRELATE` functions, in addition to the `IPREAD_FILTER` and `IPWRITE_FILTER` functions.

Example

```
kernel = BYTARR(3,3)
kernel(*) = 1B
    ; Create a 3-by-3 low pass filter.
scale_factor = FLOAT(TOTAL(kernel))
    ; Use a scaling factor to normalize the kernel
    ; during convolution.
filter = IPCREATE_FILTER(kernel, scale_factor)
    ; Create a filter object.
status = IPWRITE_FILTER(filter, 'my_lpf.ker')
    ; Save the filter to a file.
```

See Also

[IPREAD_FILTER](#), [IPWRITE_FILTER](#)

IPCT Function

Performs the inverse principle components transform on a multi-layered image.

Usage

```
result = IPCT(pct_images, pct_trans)
```

Input Parameters

pct_images — A 3D real or complex array containing image, row or pixel-interleaved images.

pct_trans — An n -by- n real or complex array where n is the number of images in the *pct_images* array. This is the principle components transform matrix for *pct_images*, usually resulting from a previous call to PCT.

Returned Value

result — A 3D complex array containing the transformed images.

Keywords

Imaginary — If set, returns only the imaginary portion of the *result*.

Intleave — A scalar string indicating the type of interleaving of 3D input image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

Mx — The mean of the sample array formed from the images. This keyword is a 1D, n -element array.

Offset — If set, *result* is offset by its minimum value so that the returned *result* has a minimum of zero.

Real — If set, returns only the real portion of the *result*.

Discussion

The inverse principle components transform (IPCT) differs from most inverse transforms in that the transformation matrix is not generic. The inverse transform can't be performed without the transformation matrix, in this case the *pct_trans* input parameter. The *pct_trans* input parameter is generated by calling the PCT function with the *Pct_trans* keyword. The returned matrix is used as the *pct_trans* input parameter in the IPCT function.

Example

The image in this example is image-interleaved, which is the default for the *Intleave* keyword.

```

rgb_i = IMAGE_READ(!IP_Data + 'boulder_image.tif')
rgb_i = rgb_i('pixels')
      ; Read a 24-bit image.

pct_i = PCT(rgb_i, Pct_trans = itrans, Mx = imx)
      ; Compute the principle components transform (PCT).

FOR I = 0, 2 DO BEGIN & $
TVSCL, pct_i(*, *, I) & $
HAK, /Mesg

ipct_i = IPCT(pct_i, itrans, Mx = imx)
      ; Compute the inverse transform.

FOR I = 0, 2 DO BEGIN & $
TVSCL, ipct_i(*, *, I) & $
HAK, /Mesg

```

See Also

[PCT](#)

IPHISTOGRAM Function

Computes the density function or the cumulative density function for an image.

Usage

result = IPHISTOGRAM(*image*)

Input Parameters

image — A 2D or 3D array containing an image; image, row, or pixel-interleaved images; or a volume.

Returned Value

result — A 1D or 2D long array.

Keywords

Cumulative — If set, the cumulative density function is returned.

Intleave — A scalar string indicating the type of interleaving of 3D image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

Max — The maximum value to consider. This keyword can be a scalar when *image* is 2D or 3D; or an array when *image* is a 3D image, row or pixel-interleaved array. (Default: MAX(*image*))

Min — The minimum value to consider. This keyword can be a scalar when *image* is 2D or 3D; or an array when *image* is a 3D image, row or pixel-interleaved array. (Default: MIN(*image*))

Omax — Specifies a variable to hold the maximum value or values considered.

Omin — Specifies a variable to hold the minimum value or values considered.

Discussion

When the *Cumulative* keyword is set, the histograms of *image* are first computed. The cumulative histogram or histograms are then calculated by summing successive values in the histogram array.

NOTE For more information on the density function calculation, see the HISTOGRAM function *Discussion* section in the *PV-WAVE Reference*.

Example

```
image = IMAGE_READ(!IP_Data + 'boulder_image.tif')
; Read a 24-bit image.

hists = IPHISTOGRAM(image('pixels'), $
    Intleave = 'image', Min = 0, Max = 255, $
    /Cumulative)
; Compute the cumulative histogram of each
; plane in the image.
```

```
TEK_COLOR
PLOT, hist(*,0), wocolorconvert = 1
OPLOT, hist(*,1), wocolorconvert = 2
OPLOT, hist(*,2), wocolorconvert = 3
      ; Plot the histograms.
```

See Also

[HIST_STATS](#)

IPLINEAR_GRAY Function

Transforms an image with a nonlinear grayscale or pseudocolor colormap to a linear grayscale image.

Usage

```
result = IPLINEAR_GRAY(image, in_cmap)
```

Input Parameters

image — A 2D array containing an image.

in_cmap — A 3-by-*n_colors* array containing the colormap for *image*, where $n_colors \leq 256$.

Returned Value

result — A 2D byte array containing the image transformed to a linear grayscale colormap.

Keywords

Colormap — Specifies a variable to receive the output linear grayscale colormap.

The following three keywords control the IPLINEAR_GRAY function behavior for *image* values falling outside the colormap range. The *Clip* keyword is the default if none is specified.

Clip — If set, clips values outside the colormap range to the range {0, ..., to $n_colors - 1$ }. (Default)

Scale — If set, scales values lying outside the colormap range into the range $\{0, \dots, n_colors - 1\}$.

Wrap — If set, uses the MOD operator to wrap values lying outside the colormap range into the range $\{0, \dots, n_colors - 1\}$.

Discussion

The *Scale*, *Clip*, and *Wrap* keywords are particularly useful when *image* is not a byte array, because they specify how values outside the colormap range are translated into the colormap.

The transformation process for IPLINEAR_GRAY works differently for grayscale than for pseudocolor input images. If the input colormap is already a grayscale colormap, the input colormap is used as a translation table to form the output image from the input image. If the input colormap is not a grayscale colormap, it is first transformed to the HSV (Hue Saturation Value) color model. The value component, which represents brightness, is then used as the translation table to form the output image from the input image.

Example

This example illustrates how to convert an 8-bit pseudocolor image to linear grayscale. First, read an image using the appropriate path name for your operating system as shown.

```
(UNIX)      image = IMAGE_READ(!Data_Dir + ' ../..' + $
            '/image-1_0/data/multi_dogs.gif')

(Windows)   image = IMAGE_READ(!Data_Dir + ' ..\..' + $
            '\image-1_0\data\multi_dogs.gif')
            ; Read an image.

gray_image = IPLINEAR_GRAY(image('pixels'), $
            image('colormap'), /Clip, Colormap = cmap)
            ; Transform the image to grayscale.

TVLCT, REFORM(cmap(0,*)), REFORM(cmap(1,*)), $
            REFORM(cmap(2,*))
            ; Load the new colormap.

TV, gray_image
            ; Display the image.
```

See Also

[IPCOLOR_24_8](#), [IS_GRAY_CMAP](#)

IPMATH Function

Performs mathematical and logical operations on a single image or between two images.

Usage

result = IPMATH(*image*, *operation*[, *operand*])

Input Parameters

image — A 3D array containing image, row, or pixel-interleaved images. This parameter is the first operand for the mathematical operation.

operation — A scalar string specifying the mathematical or logical operation to perform (see *Discussion* section).

operand — (optional) The second operand for the mathematical operation. This parameter may be a scalar or an array exactly the same size as *image*.

Returned Value

result — An array or scalar value which is the result of the mathematical or logical operation performed.

Keywords

Intleave — A scalar string indicating the type of interleaving of 3D input image arrays. When both *image* and *operand* are used, *Intleave* specifies the interleaving method for both parameters. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (*p*, *x*, *y*) for *p* pixel-interleaved images of *x*-by-*y*.

'row' — The 3D *image* array arrangement is (*x*, *p*, *y*) for *p* row-interleaved images of *x*-by-*y*.

'image' — The 3D *image* array arrangement is (*x*, *y*, *p*) for *p* image-interleaved images of *x*-by-*y*.

No_Clip — If set, the *result* data type is larger than the input image data type.

TIP The *No_Clip* keyword prevents underflow or overflow conditions from occurring.

Plane_Num — Specifies a single plane number for the function operation, where $0 \leq \text{Plane_Num} < \text{the number of planes in either } image \text{ or } operand \text{ arrays}$. This keyword is valid when *image* and/or *operand* are 3D arrays.

Zero_Negatives — If set, all negative values in *result* are set to zero.

Discussion

The *operation* parameter specifies the logical or mathematical operation to apply to one or two operands. The following two tables contain the lists of valid scalar strings for the *operation* parameter: the first table contains the list of strings which apply existing PV-WAVE operators; the second table contains a list of the existing PV-WAVE routines which can be applied to the operands.

Operation	Operands Required	Operation Performed
'+'	2	<i>image</i> + <i>operand</i> , (addition)
'-'	2	<i>image</i> – <i>operand</i> , (subtraction)
'*'	2	<i>image</i> x <i>operand</i> , (multiplication)
'/'	2	<i>image</i> ÷ <i>operand</i> , (division)
'#'	2	[<i>image</i>] x [<i>operand</i>], (matrix multiplication)
'^'	2	(<i>image</i>) ^{<i>operand</i>} , (exponentiation)
'<'	2	comparison of <i>image</i> and <i>operand</i> to find minimum
'>'	2	comparison of <i>image</i> and <i>operand</i> to find maximum
'AND'	2	<i>image</i> AND <i>operand</i> , (Boolean AND)
'MOD'	2	<i>image</i> MOD <i>operand</i> , (modulo operator)
'NOT'	1	compliment of <i>image</i> , (Boolean compliment)
'OR'	2	<i>image</i> OR <i>operand</i> , (Boolean inclusive OR)
'XOR'	2	<i>image</i> XOR <i>operand</i> , (Boolean exclusive OR)

<i>operation</i>	Operands Required	Operation Performed
'ABS'	1	<i>image</i> , (absolute value)
'ALOG'	1	ln <i>image</i> , (natural logarithm)
'ALOG10'	1	log ₁₀ <i>image</i> , (logarithm to base 10)
'COS'	1	cos(<i>image</i>), (cosine)
'EXP'	1	e^{image} , (natural exponential function)
'MAX'	1	maximum value of <i>image</i>
'MEDIAN'	1	statistical median of the <i>image</i> values
'MIN'	1	minimum value of <i>image</i>
'MODE'	1	statistical mode of <i>image</i>
'RANGE'	1	range of values in <i>image</i>
'SIN'	1	sin(<i>image</i>), (sine)
'SQRT'	1	(<i>image</i>) ^{1/2} , (square root)
'STDEV'	1	statistical standard deviation of <i>image</i>
'TAN'	1	tan(<i>image</i>), (tangent)
'VARIANCE'	1	statistical variance of <i>image</i>

Example 1

In this example, a portion of an existing image is masked using the Boolean AND operator. To do this, the mask image is first created, and then specified in the IPMATH calling sequence.

```
image = BYTARR(512,512)
OPENR, unit, !Data_Dir + 'mandril.img', $
      /Stream, /Get_Lun
READU, unit, image
CLOSE, 1
FREE_LUN, unit
mask = BYTARR(512, 512)
mask(100:300, 250:300) = 255
```

```
result = IPMATH(image, 'AND', mask)
TVSCL, result
```

Example 2

In this example, the IPMATH calling sequence is used to improve the brightness of an image by adding a scalar value.

```
result = IPMATH(image, '+', 50, /No_Clip)
TVSCL, result
```

Example 3

In this example, a common background is subtracted from several images to reveal a moving object. Any negative values in the result are set to 0.

```
background = IMAGE_READ(!IP_Data + 'frame.tif')
result1 = IPMATH(image1, '-', $
    background('pixels'), /Zero_Negatives)
result2 = IPMATH(image2, '-', $
    background('pixels'), /Zero_Negatives)
result3 = IPMATH(image3, '-', $
    background('pixels'), /Zero_Negatives)
TVSCL, result1
TVSCL, result2
TVSCL, result3
    ; Display each image to see the object without
    ; the background.
```

Example 4

In this example, a 24-bit color image is “tinted” red.

```
image = IMAGE_READ(!IP_Data + 'boulder_image.tif')
red_image = IPMATH(image('pixels'), '+', $
    10, Plane_Num = 0, Intleave = 'image')
TV, red_image, True = 3
    ; Display the “tinted” image.
```

See Also

Operators — *PV-WAVE Programmer's Guide*.

Routines — *PV-WAVE Reference*.

IPQMFDESIGN Function

Generates one of several quadrature mirror filters for use with the IPWAVELET function. A four-coefficient Daubechies wavelet filter is returned by default.

Usage

result = IPQMFDESIGN()

Input Parameters

None.

Returned Value

result — A filter object containing a 1D QMF filter as the kernel for use with the IPWAVELET function. (See [IPREAD_FILTER](#) for more information on the filter object format.)

Keywords

Only one of the following keywords may be set at a time:

Biorthogonal — A one or two-digit number: the first number specifies the number of vanishing moments in the reconstruction scaling function, and the second number (if present) specifies the number of vanishing moments in the decomposition scaling function for a one moment symmetric/antisymmetric, two moment symmetric/symmetric or three moment symmetric/antisymmetric biorthogonal wavelet filter. Valid values are:

Symmetric/Antisymmetric, one moment: 1, 13, 15

Symmetric/Symmetric, two moments: 2, 22, 24, 26, 28

Symmetric/Antisymmetric, three moments: 3, 31, 33, 35, 37, 39

Coifman — Specifies the number of coefficients in the Coifman wavelet filter. Valid numbers are 6, 12, 18, 24, 30.

Daubechies — Specifies the number of coefficients in the Daubechies wavelet filter. Valid numbers are: 2, 4, 6, 8, 10, 12, 14, 16, 18, and 20. (Default: 4)

Discussion

A quadrature mirror filter $H(z)$ is defined by the following equation:

$$|H(z)|^2 + |H(-z)|^2 = 1 .$$

IPQMFDESIGN generates three different types of wavelet filters as specified by using one of the following keywords: *Daubechies*, *Coifman*, and *Biorthogonal*. If none of these keywords are specified, the default is *Daubechies* = 4.

NOTE The *Daubechies*, *Coifman*, and *Biorthogonal* keywords can't be used together in the same calling sequence.

Example

```
image = DIST(256)
daub6 = IPQMFDESIGN(Daubechies = 6)
        ; Generate a 6-coefficient Daubechies wavelet filter.
coif12 = IPQMFDESIGN(Coifman = 12)
        ; Generate a 12-coefficient Coifman wavelet filter.
bior13 = IPQMFDESIGN(Biorthogonal = 13)
        ; Generate a 1, 3 Biorthogonal wavelet filter.
wvlt_daub6 = IPWAVELET(image, daub6, -1)
wvlt_coif12 = IPWAVELET(image, coif12, -1)
wvlt_bior13 = IPWAVELET(image, bior13, -1)
        ; Apply the wavelet transform to an image using each of
        ; the different filters, separately.

Window, 0
TVSCL, wvlt_daub6
Window, 1
TVSCL, wvlt_coif12
Window, 2
TVSCL, wvlt_boir13
        ; Display the results.
```

See Also

[IPWAVELET](#)

IPREAD_FILTER Function

Reads an ASCII text or XDR file depending on whether the file contains a spatial or spectral filter (respectively).

Usage

result = IPREAD_FILTER(*filename*)

Input Parameters

filename — A scalar string containing the name of the filter file.

Returned Value

result — An associative array containing the filter object. The filter object associative array is described as follows:

For a spatial filter, the filter object has three strings containing the key names of elements of the associative array:

'kernel' — A 2D array of the filter spatial values.

'domain' — A string set to 'SPATIAL'.

'scale' — The scale factor.

For a spectral filter, the filter object has ten strings containing the key names of elements of the associative array:

'kernel' — A 2D array of the filter spectral values.

'domain' — A string set to 'SPECTRAL'.

'cutoff' — A one-element array (for highpass and lowpass filters), or a two-element array (for bandpass and bandstop filters) containing the filter cutoff frequency, or frequencies.

'pass' — A string indicating the filter type: 'low', 'high', 'stop', 'band', or 'notch'.

'dc_offset' — A float value containing the filter DC offset.

'co_frac' — A float value containing the fraction of the maximum filter value at the cutoff frequency.

'maximum' — A float value containing the maximum filter amplitude.

'type' — A string indicating the filter type: 'ideal', or 'Butterworth'.

'xloc' — (For notch filters only.) The x -location of the filter center.

'yloc' — (For notch filters only.) The y -location of the filter center.

'center' — If set, the filter center is at the center of the array.

'order' — (For Butterworth filters only.) A floating point value indicating the filter order.

Keywords

None.

Discussion

There are many filter files provided with the Image Processing Toolkit. The files are located in the following directories:

(UNIX) ip-1_0/data/kernel/*.ker

(UNIX) ip-1_0/data/filter/*.flt

(Windows) ip-1_0\data\kernel*.ker

(Windows) ip-1_0\data\filter*.flt

The file format for the spatial filter files provided and for user-written spatial filter files must conform to the following conventions:

- Comments begin with a semicolon.
- The first un-commented line in the file is the data type of the filter weights. This line is one of the following valid strings: 'byte', 'int', 'long', 'float', or 'double'.
- The next un-commented line contains the dimensions of the kernel, with the x -dimension listed first, followed by the y -dimension.
- Next is the list of space and/or line feed-separated kernel values.
- Finally, the scale factor, if present, appears. The scale factor is a floating point value.

For spectral filters, the file format is simply the associative array saved in XDR format.

Example

In this example, the `IPREAD_FILTER` function is used to read in the Frei-Chen column edge gradient kernel from the directory of files provided with the Image Processing Toolkit.

```
freichen = IPREAD_FILTER(!IP_Data + $
    '/kernel/' + 'freichen_c3.ker')
result = IPCONVOL(image, freichen)
; Apply the Frei-Chen column edge gradient filter to an image.
```

See Also

[FILT_FREQ](#), [IPCONVOL](#), [IPWRITE_FILTER](#)

IPSCALE Function

Scales an image by shrinking or expanding it.

Usage

```
result = IPSCALE(image, xscale, yscale[, zscale])
```

Input Parameters

image — A 2D or 3D array of any data type except complex or string containing an image; image, row or pixel-interleaved images; or a volume.

xscale — A scalar value specifying the scaling factor for the *image* width.

yscale — A scalar value specifying the scaling factor for the *image* height.

zscale — (optional) A scalar value specifying the scaling factor for the *image* depth. (Valid only for volumes when using the pixel replication or depletion scaling method.)

Returned Value

result — An array of the same data type as *image*.

Keywords

Bilinear — If set, the bilinear interpolation scaling algorithm is used. (Valid for images only; not valid for volumes.)

Intleave — A scalar string indicating the type of interleaving of 3D image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

Neighbor — If set, the nearest neighbor interpolation scaling algorithm is used. This is the default scaling algorithm for the IPSCALE function. (Valid for images only; not valid for volumes.)

Pixel — If set, the pixel replication or depletion scaling algorithm is used. In this method, the *xscale*, *yscale*, and *zscale* input parameters are converted to their nearest integer values.

Shrink — If set, shrinks the image by the *xscale*, *yscale*, and *zscale* scaling factors.

Discussion

The IPSCALE function shrinks or expands the number of elements in image. If the *Pixel* keyword is used for selecting pixel replication/depletion, *image* can only be scaled by integer factors. New values in the expanded image are formed from replicated pixels in the original input image. This method provides expansion of the pixel size and has the advantage of exactly replicating existing information in the image instead of forming new values through interpolation. When expanding the image, the result will always appear as individual blocks.

If nearest neighbor or bilinear interpolation are selected using the *Neighbor* or *Bilinear* keywords, respectively, the image can be scaled by floating-point factors. New values in the expanded image are interpolated from original values at intervals, and appear where there might not have been values before.

The nearest-neighbor interpolation method is the default interpolation method. This is not a linear method, because new values that are needed are merely set equal

to the nearest existing value of image. Therefore, when increasing the image size, the result may appear as individual blocks.

Example 1

```
image = IMAGE_READ(!IP_Data + 'airplane.tif')
      ; Read an image.
IMAGE_DISPLAY, image
      ; Display the image.
image_scale = IPSCALE(image('pixels'), 3, $
                    3, /Pixel)
      ; Expand the image using pixel replication.
TV, image_scale
      ; Display the expanded image.
image_scale = IPSCALE(image('pixels'), 3, $
                    3, /Pixel, /Shrink)
      ; Shrink the original image using pixel depletion.
TV, image_scale
      ; Display the smaller image.
```

Example 2

```
image = IMAGE_READ(!IP_Data + 'photo.tif')
      ; Read an image.
IMAGE_DISPLAY, image
      ; Display the image.
image_scale = IPSCALE(image('pixels'), 1.5, 1.5)
      ; Expand the image.
TV, image_scale
      ; Display the expanded image.
```

See Also

In the *PV-WAVE Reference*: CONGRID

IPSPECTRUM Function

Estimates the power spectrum (power spectral density) of an image.

Usage

```
result = IPSPECTRUM(image[, width, height, overlap])
```

Input Parameters

image — A 2D array of any data type except string containing an image.

width — (optional) The width of the subsections of *image*. (Default: $x_{dim}/5$, where x_{dim} is the width of *image*)

height — (optional) The height of the subsections of *image*. (Default: $y_{dim}/5$, where y_{dim} is the height of *image*)

overlap — (optional) The fraction of the window size that successive subsections of *image* overlap. (Default: 0.5)

Returned Value

result — A 2D array containing the power spectrum of *image*.

Keywords

Squared — If set, computes the power spectral density as the magnitude squared of the FFT of *image*.

NOTE The *Squared* keyword cannot be used with any other keyword, or with the optional parameters *width*, *height*, and *overlap*.

Window_Param — A scalar float used only when computing either the Kaiser or Chebyshev windows.

Window_Type — A scalar value (see the following table) used to specify a window type to use when computing the spectrum of subsections of *image*.

Window_Type	Window
1	Rectangular
2	Triangular
3	Hanning
4	Hamming
5	Kaiser
6	Blackman
7	Chebyshev

NOTE For descriptions and equations for the seven window types, see the *Discussion* section for the [IPWIN](#) function.

Discussion

IPSPECTRUM computes one of several different power spectrum estimates $P(f)$ depending on the input parameters. Specific estimates available include the periodogram, the modified periodogram, Bartlett's method, and Welch's method. In all cases uniform samples of the power spectrum are returned. The equations shown below are for the case of a 1D signal, x , with a length, L . (The derivation of the 2D equations is not shown here, but is found in most Image Processing references.)

The frequency sample values are

$$f_k = k/L, \quad k = 0, 1, \dots, M \quad \text{for real data.}$$

where $M = [(L + 2)/2]$ for L even and $M = [(L + 1)/2]$ for L odd for real data.

For complex data, $M = L$.

The periodogram is defined as

$$p(f) = \frac{1}{L} \left| \sum_{l=0}^{L-1} x(l) e^{-j\pi f l} \right|^2,$$

where the frequency variable f is normalized to the Nyquist frequency of 1.0. To obtain uniform samples of the periodogram using `IPSPECTRUM`, set *length* equal to the length of the data x , and set *Window_Type* to rectangular.

The modified periodogram is defined as

$$m_p(f) = \frac{1}{L} \left| \sum_{l=0}^{L-1} w(l)x(l)e^{-j\pi fl} \right|^2,$$

where $w(l)$ is a data window sequence. To obtain uniform samples of the modified periodogram using `IPSPECTRUM`, set *length* equal to the length of the data x and set *Window_Type* to any of the seven window types.

NOTE Window types are discussed in the `IPWIN` function.

Bartlett's method breaks the data into non-overlapping data segments represented as

$$x_i(n) = x(n + iL) \quad n = 0, 1, \dots, L-1 \quad i = 0, 1, \dots, I-1.$$

A periodogram

$$P_i(f) = \left| \sum_{n=0}^{L-1} x_i(n)e^{-j\pi fn} \right|^2$$

is computed for each data segment and averaged to obtain the Bartlett estimate

$$P_B(f) = \frac{1}{I} \sum_{i=1}^{I-1} P_i(f).$$

To obtain uniform samples of Bartlett's estimate using `IPSPECTRUM`, set *length* to be less than the data length, set *overlap* to 0 and set *Window_Type* to rectangular.

The Welch method breaks the data into length L overlapping data segments represented as $(n + iL)$.

$$x_i(n) = x(n + iQ) \quad n = 0, 1, \dots, L-1 \quad i = 0, 1, \dots, I-1$$

where $Q = (L - \text{overlap})$.

A modified periodogram is then computed for each data segment given by

$$P_i(f) = \frac{1}{Lu} \left| \sum_{n=0}^{L-1} x_i(n)w(n)e^{-j\pi fn} \right|^2$$

where

$$u = \frac{1}{L} \sum_{n=0}^{L-1} w^2(n).$$

The Welch power spectrum estimate is the average of the modified periodogram of each data segment, given by

$$P_w(f) = \frac{1}{I} \sum_{i=1}^{I-1} P_i(f).$$

To obtain uniform samples of the Welch estimate using `IPSPECTRUM`, set *length* less than the data length, set *overlap* to a nonzero value and set *Window_Type* to the desired window type.

NOTE In estimating the power spectrum it is assumed that the input signal is stationary (i.e., the frequency content does not change with time). If the signal is non-stationary, the `IPWAVELET` function can often provide better results than `IPSPECTRUM`.

Example

```
image = IMAGE_READ(!IP_Data + 'airplane.tif')
        ; Read an image.

IMAGE_DISPLAY, image
        ; Display the image.

image_spectrum = IPSPECTRUM(image('pixels'), $
        /Squared)
        ; Compute the power spectrum.

TVSCL, image_spectrum
        ; Display the power spectrum.
```

See Also

[IPWAVELET](#), [IPWIN](#)

IPSTATS Function

Computes up to eight different statistical operations on an array, including the mean, variance, standard deviation, skewness, kurtosis, minimum, maximum, and range.

Usage

result = IPSTATS(*array*)

Input Parameters

array — An array.

Returned Value

result — Returns either a scalar statistical value or an array of the requested statistical values.

Keywords

All — If set, computes all eight statistical calculations.

Kurtosis — If set, computes the kurtosis of *array*.

Maximum — If set, finds the maximum value in *array*.

Mean — If set, computes the mean of *array*.

Minimum — If set, finds the minimum value in *array*.

Range — If set, determines the range of values in *array*.

Skewness — If set, computes the skewness of *array*.

Std — If set, computes the standard deviation of *array*.

Var — If set, computes the variance of *array*.

Discussion

If no keywords are specified, the mean of the array is computed and returned as a scalar value. If more than one keyword is specified, or if *All* is specified, a floating point array containing the requested statistics is returned.

The values in the *result* array will be in the following order: mean, variance, standard deviation, skewness, kurtosis, minimum, maximum, range.

Example 1

Compute statistics on an image.

```
texture = IMAGE_READ(!IP_Data + 'texture.tif')
; Read an image.
stats = IPSTATS(texture('pixels'), /All)
; Compute statistics for the image.
PRINT, stats
; Print the statistics.
```

Example 2

Compute statistics on a series of images.

```
image1 = IMAGE_READ(!IP_Data + 'frame1.tif')
image2 = IMAGE_READ(!IP_Data + 'frame2.tif')
image3 = IMAGE_READ(!IP_Data + 'frame3.tif')
image4 = IMAGE_READ(!IP_Data + 'frame4.tif')
image5 = IMAGE_READ(!IP_Data + 'frame5.tif')
; Read in some images.
stats = FLTARR(5, 8)
stats(0,*) = IPSTATS(image1('pixels'), /All)
stats(1,*) = IPSTATS(image2('pixels'), /All)
stats(2,*) = IPSTATS(image3('pixels'), /All)
stats(3,*) = IPSTATS(image4('pixels'), /All)
stats(4,*) = IPSTATS(image5('pixels'), /All)
; Compute statistics on the frames.
PLOT, stats(*, 3)
; Plot the skewness of the frames.
```

See Also

[KURTOSIS](#), [RANGE](#), [SKEWNESS](#)

IPWAVELET Function

Computes the separable wavelet transform for an image.

Usage

```
result = IPWAVELET(image, qmfilt, n_stages [, direction])
```

Input Parameters

image — A 2D or 3D array containing an image; or image, row, or pixel-interleaved images.

qmfilt — A 1D quadrature mirror filter designed using the IPQMFDESIGN function.

n_stages — The number (greater than or equal to 1) of wavelet transform stages to perform.

direction — (optional) A scalar indicating the direction of the transform as follows:

- 1 Forward transform (default)
- 1 Backward transform

Returned Value

result — A 2D or 3D double array containing the wavelet transform.

Keywords

Intleave — A scalar string indicating the type of interleaving of 3D input image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

Discussion

Computing the wavelet transform of an image using a compactly supported orthonormal wavelet is equivalent to applying the quadrature mirror filter-bank structure to the image as shown in the following figure.

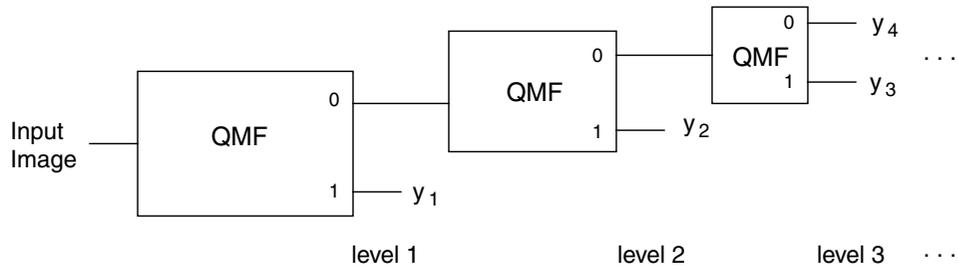


Figure 2-16 Filter structure for computing the forward wavelet transform.

The details of how and why the structure shown in the figure is connected to a compactly supported orthonormal wavelet are found in Akansu and Haddad, 1992; Daubechies, 1988, and 1992; Rioul and Vetterli, 1991; and Vaidyanathan, 1993.

The input and output sequences of each block in the figure represent the inputs and output of the forward part of the quadrature mirror filtering technique. The specific input-output relation between each block and the quadrature mirror filter is shown in the following figure.

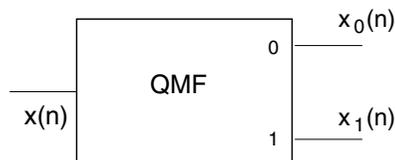


Figure 2-17 Basic computational cell in the forward wavelet transform structure.

IPWAVELET requires a quadrature mirror filter to be supplied. Such a filter is obtained by using the IPQMFDESIGN function.

The input parameter n_stages specifies the number of levels of the wavelet transform structure to compute.

The backwards wavelet transform is computed using the filter bank structure shown in the following figure.

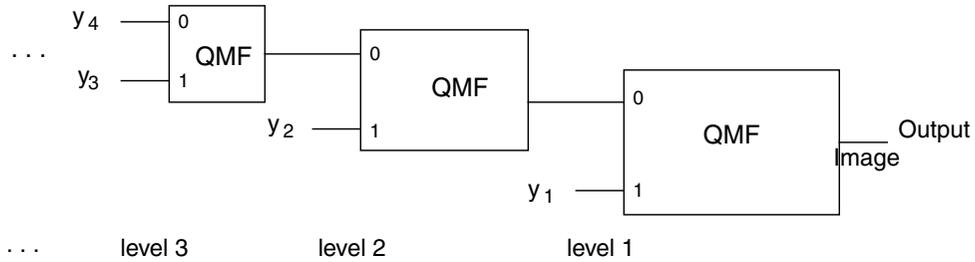


Figure 2-18 Filter structure for computing the backward wavelet transform.

Example

This example illustrates how to compute the wavelet transform of an image.

```
image = IMAGE_READ(!IP_Data + 'blobs.tif')
    ; Read an image.

qmfilt = IPQMFDESIGN(Daubechies = 4)

wvlt = IPWAVELET(image('pixels'), qmfilt, 2, -1)
    ; Compute the 2-stage forward wavelet transform with
    ; the Daubechies 4 wavelet.

TVSCL, wvlt
    ; Display the wavelet transform.

iwvlt = IPWAVELET(wvlt, qmfilt, 2, 1)
    ; Compute the inverse transform.

wvlt_error = iwvlt - image('pixels')
    ; There will be some distortion due to machine noise.

PRINT, MAX(wvlt_error)
```

See Also

[DCT](#), [HAAR](#)

IPWIN Function

Computes one of several different data windows: Blackman, Chebyshev, Hamming, Hanning, Kaiser, rectangular, or triangular.

Usage

result = IPWIN(*n*[, *m*] [, *a*])

Input Parameters

n — The first dimension of the window sequence.

m — (optional) The second dimension of the window sequence. If *m* is not given, a one-dimensional window of length *n* is returned.

a — (optional) A scalar float parameter used only when computing either the Kaiser or Chebyshev windows.

Returned Value

result — A 1D or 2D array containing the requested window.

Keywords

Blackman — If set, a Blackman window is returned.

Center — If set, the window is shifted such that its center is at the center of the *result* array.

Chebyshev — If set, a Chebyshev window is returned. If *Chebyshev* is set, input parameter *a* (*a* is θ_0 in the Chebyshev window equation) is also required.

Hamming — If set, a Hamming window is returned.

Hanning — If set, a Hanning window is returned.

Kaiser — If set, a Kaiser window is returned. If *Kaiser* is set, input parameter *a* (*a* is α in the Kaiser window equation) is also required.

Rectangular — If set, a rectangular window is returned. This is also known as a boxcar window.

Triangular — If set, a triangular window is returned.

Window_Type — A scalar value (see the following table) used to specify a window type. This keyword may be used in place of the particular window keywords.

Window_Type	Window
1	Rectangular
2	Triangular
3	Hanning
4	Hamming
5	Kaiser
6	Blackman
7	Chebyshev

Discussion

A listing of the applicable time domain equations is given below, where $w(n)$, $0 < n < N - 1$ for each of the windows.

Blackman window:

$$w(n) = 0.42 - 0.5 \cos \frac{2\pi(n - \frac{N-1}{2})}{N-1} + 0.08 \cos \frac{4\pi(n - \frac{N-1}{2})}{N-1}$$

Chebyshev window:

for $N = 0$,

$$w_0(k) = \begin{cases} 1, & \text{for } k = 0 \\ 0, & \text{otherwise} \end{cases}$$

for $N = 1$,

$$w_1(k) = \begin{cases} x_0 - 1, & \text{for } k = 0 \\ \frac{x_0}{2}, & \text{for } |k| = 1 \\ 0, & \text{otherwise} \end{cases}$$

for $N > 1$,

$$w_N(k) = 2(x_0^2 - 1)w_{N-1}(k) \\ + (x_0^2)[w_{N-1}(k-1) + w_{N-1}(k+1)] - w_{N-2}(k)$$

where

$$x_0 = \frac{1}{\cos(\theta_0/2)}$$

and θ_0 is the IPWIN input parameter a .

Hamming window:

$$w(n) = 0.54 - 0.46 \cos \frac{2\pi(n - \frac{N-1}{2})}{N-1}$$

Hanning window:

$$w(n) = 0.5 - 0.5 \cos \frac{2\pi(n - \frac{N-1}{2})}{N-1}$$

Kaiser window:

$$w(n) = \frac{I_0\left(\alpha \sqrt{1 - \left[2\frac{(n - \frac{N-1}{2})^2}{N-1}\right]}\right)}{I_0(\alpha)},$$

where $I_0(x)$ is the zeroth order Bessel function of the first kind, and α is the IPWIN input parameter a .

Rectangular (or boxcar) window:

$$w(n) = 1, \text{ for all } n.$$

Triangular window:

$$w(n) = 1 - \frac{2\left|n - \frac{N-1}{2}\right|}{N-1}, \text{ for all } n.$$

Example

This example illustrates how to generate each of the different window types for a 128-by-128 window.

```
n = 128
m = 128
rect = IPWIN(n, m, /Rectangular, /Center)
TVSCL, rect
    ; Generate and view a rectangular window.
tri = IPWIN(n, m, /Triangular, /Center)
TVSCL, tri
    ; Generate and view a triangular window.
hann = IPWIN(n, m, /Hanning, /Center)
TVSCL, hann
    ; Generate and view a Hanning window.
hamm = IPWIN(n, m, /Hamming, /Center)
TVSCL, hamm
    ; Generate and view a Hamming window.
kaiser = IPWIN(n, m, 0.5, /Kaiser, /Center)
TVSCL, kaiser
    ; Generate and view a Kaiser window.
black = IPWIN(n, m, /Blackman, /Center)
TVSCL, black
    ; Generate and view a Blackman window.
cheby = IPWIN(n + 1, m + 1, 0.15, /Chebyshev, $
    /Center)
TVSCL, cheby
    ; Generate and view a Chebyshev window.
```

See Also

In the *PV-WAVE Reference*: HANNING

IPWRITE_FILTER Function

Saves a 2D convolution kernel to an ASCII text file.

Usage

status = IPWRITE_FILTER(*kernel*, *filename*)

Input Parameters

kernel — A 2D array containing the convolution kernel to be saved.

filename — A scalar string containing the name of the kernel file.

Returned Value

status — A scalar value indicating the success of the write operation. Expected values are:

- 1 Indicates a successful write.
- 0 Indicates an error such as an invalid filename.

Keywords

Comment — A scalar string or an array of comment strings to be printed at the top of the kernel file.

Scale — The scale factor to apply to the filter kernel. (Default: 1.0)

Discussion

There are many filter files provided with the Image Processing Toolkit. The files are located in the following directories:

(UNIX) ip-1_0/data/kernel/*.ker

(UNIX) ip-1_0/data/filter/*.flt

(Windows) ip-1_0\data\kernel*.ker

(Windows) ip-1_0\data\filter*.flt

The file format for the spatial filter files provided and for user-written spatial filter files must conform to the following conventions:

- Comments begin with a semicolon.
- The first un-commented line in the file is data type of the filter weight. This line is one of the following valid strings: 'byte', 'int', 'long', 'float', or 'double'.
- The next un-commented line contains the dimensions of the kernel, with the *x*-dimension listed first, followed by the *y*-dimension.
- Next is the list of space and/or line feed-separated kernel values.
- Finally, the scale factor, if present, appears. The scale factor is a floating point value.

The format for spectral filters is simply the associative array passed to the routine and saved in XDR format.

Example 1

```
gauss = GAUSS_KERNEL(5, 5)
      ; Create a spatial Gaussian filter.
status = IPWRITE_FILTER(gauss, 'gauss_5.ker')
      ; Save the filter in a file.
```

Example 2

```
lpf = FILT_FREQ(30, /Low, Xdim = 300, $
              Ydim = 300)
      ; Create an ideal lowpass filter.
status = IPWRITE_FILTER(lpf, 'lpf_300.flt')
      ; Save the filter to a file.
```

See Also

[IPCREATE_FILTER](#), [IPREAD_FILTER](#)

IS_GRAY_CMAP Function

Determines if a colormap is grayscale.

Usage

result = IS_GRAY_CMAP(*colormap*)

Input Parameters

colormap — A 3-by-*n_colors* array containing the colormap, where *n_colors* is the number of colors in the colormap.

Returned Value

result — A value indicating if the colormap is grayscale as follows:

- 1 Grayscale colormap
- 0 Colormap not grayscale

Keywords

Linear — If set, determines if the colormap is monotonically increasing.

Discussion

In a grayscale colormap, the red, green, and blue components are equal to each other for each element in the colormap. Therefore, to test for a grayscale colormap, the following equation is true:

$$\text{red}(i) \text{ EQ } \text{blue}(i) \text{ EQ } \text{green}(i), \text{ for } 0 \leq i < n_colors$$

where *n_colors* is the number of colors in the colormap.

A grayscale colormap is said to be linear if the grayscale value is monotonically increasing. To determine linearity for a grayscale colormap, it is sufficient to test only the red component as follows:

$$\text{red}(i - 1) < \text{red}(i), \text{ for } 1 \leq i < n_colors$$

where *n_colors* is the number of colors in the colormap.

Example

```
TVLCT, r, g, b, /Get
    ; Get the current colormap.

cmap = BYTARR(3, N_ELEMENTS(r))
cmap(0, *) = r
cmap(1, *) = g
cmap(2, *) = b
    ; Put it into a colormap array.

result = IS_GRAY_CMAP(cmap, /Linear)
IF result EQ 1 THEN PRINT, "Colormap is " + $
    "linear grayscale." $ &
ELSE PRINT, "Colormap is not linear grayscale."
    ; Check to see if it is linear grayscale.
```

See Also

[IPLINEAR_GRAY](#)

KURTOSIS Function

Computes the kurtosis of an array.

Usage

result = KURTOSIS(*array*)

Input Parameters

array — An array of any data type except string.

Returned Value

result — A double array containing the kurtosis of *array*.

Keywords

None.

Discussion

Kurtosis is a useful measure for statistical texture analysis. The KURTOSIS function computes the kurtosis of *array(k, l)* as:

$$\text{kurtosis} = \sum_{k=0}^{K-1} \sum_{l=0}^{L-1} \left(\frac{\text{array}(k, l) - \text{mean}}{\text{std}} \right)^4 - 3.0$$

where *mean* = AVG(*array*) and *std* = STDEV(*array*).

Example

```
image = IMAGE_READ(!IP_Data + 'noise_test.tif')
      ; Read an image.

kurt = KURTOSIS(image('pixels'))

PRINT, 'Kurtosis = ', kurt
      ; Compute the kurtosis of the image.
```

See Also

[ENTROPY](#), [MODE](#), [RANGE](#), [SKEWNESS](#), [UNIFORMITY](#)

MAJOR_AXIS Function

Computes the major axis of a region in an image.

Usage

result = MAJOR_AXIS(*image*, *pixels*)

Input Parameters

image — A 2D array containing a region.

pixels — A long array containing the element numbers of the pixels in *image* that compose the region.

Returned Value

result — A double scalar value that is the angle defined by the horizontal axis and the major axis through the region.

Keywords

Eigval — Specifies a variable to receive the eigenvalues of the region.

Eigvect — Specifies a variable to receive the eigenvectors of the region.

Discussion

The major axis of a region describes the direction of maximal dispersion. Specifically, the major axis of a region is determined from the eigenvectors of the covariance matrix of the pixel location array, where the pixel location array is composed of the row and column values of the region pixel locations. The eigenvector corresponding to the maximum eigenvalue of the covariance matrix is the major axis angle.

Example

```
image = IMAGE_READ(!IP_Data + 'blobs.tif')
      ; Read an image.

one_blob = THRESHOLD(image('pixels'), 20, $
      25, /Binary)
      ; Segment the image using thresholding.

blob_pixels = WHERE(one_blob NE 0)

major = MAJOR_AXIS(one_blob, blob_pixels)
```

```
        ; Find the major axis of the remaining blob.  
PRINT, 'Major axis angle = ', major  
        ; Print the major axis.
```

See Also

[CENTROID](#), [PERIMETER](#)

MODE Function

Determines the mode of an array.

Usage

```
result = MODE(array)
```

Input Parameters

array — An array of any data type except string.

Returned Value

result — An array of the same data type as *array* containing the mode of *array*.

Keywords

None.

Discussion

The mode of an array is the most frequently occurring value in the array.

Example

```
image = IMAGE_READ(!IP_Data + 'objects.tif')  
        ; Read an image.  
  
image_mode = MODE(image('pixels'))  
        ; Compute the mode of the image.  
  
PRINT, image_mode  
        ; Print the mode. This is the value of the largest region.  
  
thresh_image = THRESHOLD(image('pixels'), $  
        image_mode, /Binary)  
        ; Threshold the image for the largest region.
```

```
TVSCL, thresh_image  
; Display the image.
```

See Also

[ENTROPY](#), [KURTOSIS](#), [RANGE](#), [SKEWNESS](#), [UNIFORMITY](#)

MOMENT2D Function

Computes the 2D moments of an array.

Usage

```
result = MOMENT2D(image[, p, q])
```

Input Parameters

image — A 2D or 3D array containing an image; or image, row or pixel-interleaved images.

p — (optional) The *p*-order of the moment $M(p, q)$.

q — (optional) The *q*-order of the moment $M(p, q)$.

Returned Value

result — For 2D *image* arrays, *result* is a scalar double containing the computed moment.

For 3D *image* arrays *result* is a double array containing the computed moments.

Keywords

CCentroid — If set, computes the column centroid $M(0, 1)/M(0, 0)$.

Central — If set, computes the central moment $M_C(p, q)$.

CIertia — If set, computes the column inertia $M_{CS}(0, 2)$.

Column — If set, computes the column moment $M(0, 1)$.

Intleave — A scalar string indicating the type of interleaving of 3D input image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for *p* pixel-

interleaved images of x -by- y .

' row ' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

' image ' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

RCentroid — If set, computes the row centroid $M(1, 0)/M(0, 0)$.

RCInertia — If set, computes the row-column cross inertia $M_{CS}(1, 1)$.

RIInertia — If set, computes the row inertia $M_{CS}(2, 0)$.

Row — If set, computes the row moment $M(1, 0)$.

Scaled — If set, computes the central moment $M_S(p, q)$.

Surface — If set, computes the surface moment $M(0, 0)$.

Discussion

Spatial moments are used in shape analysis. Moments describe the distribution of grayscale values in an image. The equation for a two-dimensional moment is as follows:

$$M_{pq} = \sum_{i=0}^{I-1} \sum_{j=0}^{J-1} i^p j^q \text{image}(i, j)$$

The calculation of a central moment, M_C , involves the subtraction of the image centroid from each pixel as shown in these equations:

$$\bar{x} = \frac{M_{10}}{M_{00}}, \quad \bar{y} = \frac{M_{01}}{M_{00}}$$

$$\mu_{pq} = \sum_{i=0}^{I-1} \sum_{j=0}^{J-1} (i - \bar{x})^p (j - \bar{y})^q \text{image}(i, j)$$

Scaled moments, M_S , are divided by $I^p * J^q$ for moment M_{pq} computed for an I -by- J image.

Example

```
image = IMAGE_READ(!IP_Data + 'blobs.tif')
    ; Read an image.

thresh_image = THRESHOLD(image('pixels'), 20, $
    30, /Binary)
    ; Threshold the image for a single blob.

moment = MOMENT2D(thresh_image, /CCentroid)
    ; Compute the column centroid.

PRINT, moment
    ; Print the centroid.
```

See Also

[CENTROID](#), [MAJOR_AXIS](#)

MORPH_CLOSE Function

Performs the morphologic-close operation for shape processing. The close operation is defined as dilation followed by erosion

Usage

```
result = MORPH_CLOSE(image, structure[, x0, y0])
```

Input Parameters

image — The array to be eroded and dilated.

structure — A 1D or 2D array of structuring elements. The elements are interpreted as binary elements (values are either zero or nonzero), unless the *Gray* keyword is used.

x0 — (optional) The *x*-coordinate of the structure's origin.

y0 — (optional) The *y*-coordinate of the structure's origin.

Returned Value

result — The closed image that is of the same size and dimensions as *image*.

Keywords

Gray — If set, uses grayscale dilation and erosion. (Default: binary dilation and erosion)

Values — An array of values of the structuring element. The *Values* array must have the same dimensions and number of elements as the *structure* parameter.

Discussion

Morphological operations are defined for grayscale byte images. If *image* is not originally of type byte, PV-WAVE makes a temporary copy of *image* that is of type byte before using it for the morphological processing.

The morphological closing operation is defined as dilation followed by erosion. These operations are defined for byte data type images only. Closing is typically used to fill in small artifacts in images such as small gaps or holes. Larger gaps and holes can be filled by applying the morphological closing operation multiple times in succession.

Example

```
morph_struct = BYTARR(3, 3)
morph_struct(*) = 1B
morph_struct(1, 1) = 0B
    ; Make a square structuring element.
test_image = IMAGE_READ(!IP_Data + 'vnitext.tif')
    ; Read an image.
close_image = MORPH_CLOSE(test_image('pixels'), $
    morph_struct)
    ; Remove the thin parts of the letters.
TVSCL, close_image
    ; Display the image.
```

See Also

[HIT_MISS](#), [MORPH_OPEN](#), [MORPH_OUTLINE](#),
[SKELETONIZE](#), [TOP_HAT](#)

In the *PV-WAVE Reference*: [DILATE](#), [ERODE](#)

MORPH_OPEN Function

Performs the morphologic-open operation for shape processing. The open operation is defined as erosion followed by dilation.

Usage

result = MORPH_OPEN(*image*, *structure*[, *x0*, *y0*])

Input Parameters

image — The array to be opened.

structure — A 1D or 2D array of structuring elements. The elements are interpreted as binary elements (values are either zero or nonzero), unless the *Gray* keyword is used.

x0 — (optional) The *x*-coordinate of the structure's origin.

y0 — (optional) The *y*-coordinate of the structure's origin.

Returned Value

result — The opened image that is of the same size and dimensions as *image*.

Keywords

Gray — If set, uses grayscale erosion and dilation. (Default: binary erosion and dilation)

Values — An array of values of the structuring element. The *Values* array must have the same dimensions and number of elements as the *structure* parameter.

Discussion

Morphological operations are defined for grayscale byte images. If *image* is not originally of type byte, PV-WAVE makes a temporary copy of *image* that is of type byte before using it for the morphological processing.

The morphological opening operation is defined as erosion followed by dilation. These operations are defined for byte data type images only. Opening is typically used to smooth the boundaries of objects in images while removing noise spikes. Opening can be applied multiple times in succession to achieve greater degrees of smoothing or noise reduction.

Example

```
morph_struct = BYTARR(3, 3)
morph_struct(*) = 1B
morph_struct(1, 1) = 0B
    ; Make a square structuring element.
test_image = IMAGE_READ(!IP_Data + 'vnitext.tif')
    ; Read an image.
open_image = MORPH_OPEN(test_image('pixels'), morph_struct)
    ; Smooth the letters.
TVSCL, open_image
    ; Display the image.
```

See Also

[HIT_MISS](#), [MORPH_CLOSE](#), [MORPH_OUTLINE](#),
[SKELETONIZE](#), [TOP_HAT](#)

In the *PV-WAVE Reference*: [DILATE](#), [ERODE](#)

MORPH_OUTLINE Function

Performs morphologic outlining for shape processing.

Usage

result = MORPH_OUTLINE(*image*, *structure* [, *x0*, *y0*])

Input Parameters

image — The array to be morphologically outlined.

structure — A 1D or 2D array containing the structuring element. The array elements are interpreted as binary values (either zero or nonzero), unless the *Gray* keyword is used.

x0 — (optional) The *x*-coordinate of structure's origin.

y0 — (optional) The *y*-coordinate of structure's origin.

Returned Value

result — An array of the same size and dimensions as *image*.

Keywords

Dilation — If set, performs outlining by subtracting the original image from the dilated image; otherwise, the eroded image is subtracted from the original image.

Gray — If set, uses grayscale, rather than binary closing.

Values — An array of the same dimensions and number of elements as *structure*, containing the values of the structuring element.

Discussion

Morphological operations are defined for grayscale byte images. If *image* is not originally of type byte, PV-WAVE makes a temporary copy of *image* that is of type byte before using it for the morphological processing.

The morphological outline operation is defined as the difference between either the dilated or eroded image and itself. These operations are defined for byte data type images only. Outlining produces an output image in which all pixels are the background gray-level value except those pixels that lie on an object's boundary. The thickness of the boundary is determined by the dimensions of the structuring element.

Example

```
morph_struct = BYTARR(3, 3)
morph_struct(*) = 1B
morph_struct(1, 1) = 0B
    ; Make a square structuring element.
test_image = IMAGE_READ(!IP_Data + 'blobs.tif')
    ; Read an image.
IMAGE_DISPLAY, test_image
```



Figure 2-19 A collection of differently shaped blobs.

```
outline_image = MORPH_OUTLINE(test_image('pixels'), $
    morph_struct, values = morph_struct)
    ; Find the blob outlines.
LOADCT, 0
TVSCL, outline_image
```

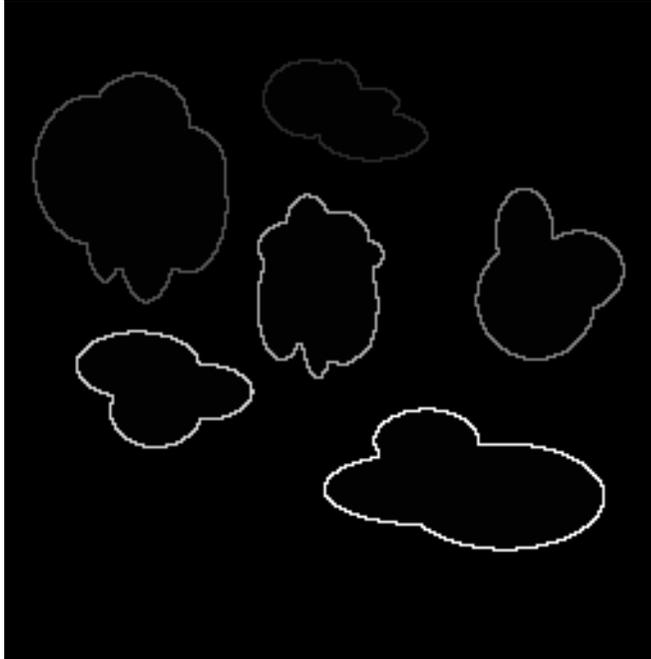


Figure 2-20 The outlines of the blobs.

See Also

[HIT_MISS](#), [MORPH_CLOSE](#), [MORPH_OPEN](#),
[SKELETONIZE](#), [TOP_HAT](#)

In the *PV-WAVE Reference*: [DILATE](#), [ERODE](#)

NOISE_GEN Function

Returns an m -dimensional array of the desired noise distribution.

Usage

result = NOISE_GEN(*d1*, *d2*, ..., *d8*)

Input Parameters

d1, ..., *d8* — The dimensions of the *result* output array.

Returned Value

result — An m -dimensional array of the desired noise distribution, where m is from 1 to 8.

Keywords

A — A positive scalar float that is the shape parameter for a gamma distribution.

Background — The desired background value of the *result* array.

NOTE The *Background* keyword is only valid for impulse noise.

Beta — If present and nonzero, the beta distribution is used.

Binary — If present and nonzero, returns a binary distribution.

NOTE The *Binary* keyword is only valid for impulse distribution.

Covariances — A 2D, square variance-covariance matrix defined for the *Mvar_Normal* distribution.

The 2D result is of size n -by- $N_ELEMENTS(Covariances(*, 0))$.

NOTE Both the *Covariances* and *Mvar_Normal* keywords must be specified to return numbers from a multivariate normal distribution.

Double — If present and nonzero, double precision is used.

Exponential — If present and nonzero, uses the exponential distribution.

F1, ..., F8 — (Used with the *Periodic* keyword.) The frequency for dimension *i* (*i* = 1, ..., 8) used for the periodic noise distribution.

Gamma — If present and nonzero, uses the gamma distribution.

High — The desired maximum of the returned values.

Impulse — If present and nonzero, uses the impulse distribution.

Low — The desired minimum of the returned values.

Mvar_Normal — If present and nonzero, uses the multivariate normal distribution.

NOTE Both the *Mvar_Normal* and *Covariances* keywords must be specified to return numbers from a multivariate normal distribution.

Normal — If present and nonzero, uses the normal distribution.

Periodic — If present and nonzero, uses the periodic distribution.

Pin — The first parameter of the beta distribution, a positive value.

Poisson — If present and nonzero, uses the Poisson distribution.

Prob — The noise probability for the impulse distribution.

Qin — The second parameter of the beta distribution, a positive value.

Rayleigh — If present and nonzero, uses the Rayleigh distribution.

S_Amp — A seed for the random number generator.

NOTE The *S_Amp* keyword is only valid for use with the *Impulse*, *Rayleigh*, and *Periodic* keywords.

S_Index — A seed for the index random number generator.

NOTE The *S_Index* keyword is valid for use with the impulse distribution only.

Sine_Amp — The amplitude of sinusoidal (periodic) noise.

Theta — The mean value of the Poisson distribution, a positive value.

Type — A string indicating the desired data type of *result*. Valid strings include: 'byte', 'fix', 'float', or 'double'.

Uniform — If present and nonzero, uses the uniform distribution.

Var — The noise variance for the Rayleigh distribution.

Discussion

NOTE The PV-WAVE:Image Processing Toolkit NOISE_GEN function is a wrapper for the noise generators which are already available in PV-WAVE Advantage. Specifically, these include the RANDOM and RANDOMOPT routines in PV-WAVE IMSL Statistics.

NOISE_GEN returns an array of noise with the probability distribution specified by the *Normal*, *Poisson*, *Uniform*, *Impulse*, *Rayleigh*, *Exponential*, *Mvar_Normal*, *Beta*, *Gamma*, and *Periodic* keywords. If NOISE_GEN is called without any keywords, then the returned array contains random numbers from a uniform (0, 1) distribution.

Uniform (0,1) Distribution

The default action of NOISE_GEN generates pseudo-random numbers from a uniform (0, 1) distribution using a multiplicative, congruent method. The form of the generator follows:

$$x_i \equiv cx_{i-1} \bmod(2^{31}-1)$$

Each x_i is then scaled into the unit interval (0, 1). The possible values for c in the generators are 16807, 397204094, and 950706376. The selection is made by using the RANDOMOPT procedure with the *Gen_Option* keyword. The choice of 16807 results in the fastest execution time. If no selection is made explicitly, the functions use the multiplier 16807.

The RANDOMOPT procedure called with the *Set* keyword is used to initialize the seed of the random-number generator.

You can select a shuffled version of these generators. In this scheme, a table is filled with the first 128 uniform (0,1) numbers resulting from the simple multiplicative congruent generator. Then, for each x_i from the simple generator, the low-order bits of x_i are used to select a random integer, j , from 1 to 128. The j -th entry in the table is then delivered as the random number, and x_i , after being scaled into the unit interval, is inserted into the j -th position in the table.

The values returned are positive and less than 1.0. Some values returned may be smaller than the smallest relative spacing; however, it may be the case that some value, for example $r(i)$, is such that $1.0 - r(i) = 1.0$.

Deviates from the distribution with uniform density over the interval (a , b) can be obtained by scaling the output.

Normal Distribution

Calling RANDOM with keyword *Normal* generates pseudorandom numbers from a standard normal (Gaussian) distribution using an inverse CDF technique. In this method, a uniform (0, 1) random deviate is generated. Then, the inverse of the normal distribution function is evaluated at that point using the PV-WAVE function NORMALCDF with keyword *Inverse*.

Deviate from the normal distribution with mean specific mean and standard deviation can be obtained by scaling the output from RANDOM.

Exponential Distribution

Calling RANDOM with keyword *Exponential* generates pseudorandom numbers from a standard exponential distribution. The probability density function is $f(x) = e^{-x}$, for $x > 0$. RANDOM uses an antithetic inverse CDF technique. In other words, a uniform random deviate U is generated, and the inverse of the exponential cumulative distribution function is evaluated at $1.0 - U$ to yield the exponential deviate.

Poisson Distribution

Calling RANDOM with keywords *Poisson* and *Theta* generates pseudorandom numbers from a Poisson distribution with positive mean *Theta*. The probability function (with $\theta = \textit{Theta}$) follows:

$$f(x) = (e^{-\theta}\theta^x)/x! , \quad \text{for } x = 0, 1, 2, \dots$$

If *Theta* is less than 15, RANDOM uses an inverse CDF method; otherwise, the PTPE method of Schmeiser and Kachitvichyanukul (1981) is used. (See also Schmeiser, 1983.) The PTPE method uses a composition of four regions, a triangle, a parallelogram, and two negative exponentials. In each region except the triangle, acceptance/rejection is used. The execution time of the method is essentially insensitive to the mean of the Poisson.

Gamma Distribution

Calling RANDOM with keywords *Gamma* and *A* generates pseudorandom numbers from a Gamma distribution with shape parameter $a = A$ and unit scale parameter. The probability density function follows:

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x} \quad \text{for } x \geq 0$$

Various computational algorithms are used depending on the value of the shape parameter a . For the special case of $a = 0.5$, squared and halved normal deviates are used; for the special case of $a = 1.0$, exponential deviates are generated. Otherwise, if a is less than 1.0, an acceptance-rejection method due to Ahrens, described in Ahrens and Dieter (1974), is used. If a is greater than 1.0, a 10-region rejection procedure developed by Schmeiser and Lal (1980) is used.

The Erlang distribution is a standard Gamma distribution with the shape parameter having a value equal to a positive integer; hence, RANDOM generates pseudorandom deviates from an Erlang distribution with no modifications required.

Beta Distribution

Calling RANDOM with keywords *Beta*, *Pin*, and *Qin* generates pseudorandom numbers from a beta distribution with parameters *Pin* and *Qin*, both of which must be positive. With $p = Pin$ and $q = Qin$, the probability density function is

$$f(x) = \frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)} x^{p-1} (1-x)^{q-1}$$

where $\Gamma(\cdot)$ is the Gamma function.

The algorithm used depends on the values of p and q . Except for the trivial cases of $p = 1$ or $q = 1$, in which the inverse CDF method is used, all the methods use acceptance/rejection. If p and q are both less than 1, the method of Jöhnk (1964) is used. If either p or q is less than 1 and the other is greater than 1, the method of Atkinson (1979) is used. If both p and q are greater than 1, algorithm BB of Cheng (1978), which requires very little setup time, is used if x is less than 4, and algorithm B4PE of Schmeiser and Babu (1980) is used if x is greater than or equal to 4.

NOTE Note that for p and q both greater than 1, calling RANDOM to generate random numbers from a beta distribution a loop getting less than four variates on each call yields the same set of deviates as executing one call and getting all the deviates at once.

The values returned are less than 1.0 and greater than ϵ , where ϵ is the smallest positive number such that $1.0 - \epsilon < 1.0$.

Multivariate Normal Distribution

Calling RANDOM with keywords *Mvar_Normal* and *Covariances* generates pseudorandom numbers from a multivariate normal distribution with mean array

consisting of all zeros and variance-covariance matrix defined using keyword *Covariances*. First, the Cholesky factor of the variance-covariance matrix is computed. Then, independent random normal deviates with mean zero and variance 1 are generated, and the matrix containing these deviates is post-multiplied by the Cholesky factor. Because the Cholesky factorization is performed in each invocation, it is best to generate as many random arrays as needed at once.

Deviates from a multivariate normal distribution with means other than zero can be generated by using *RANDOM* with keywords *Mvar_Normal* and *Covariances*, then adding the arrays of means to each row of the result.

Rayleigh Distribution

Calling *NOISE_GEN* with keywords *Rayleigh* and *Var* generates pseudorandom numbers from a Rayleigh distribution. With $\alpha = Var$, the probability density function for Rayleigh noise is:

$$f(x) = \frac{G(x)}{\alpha^2} e^{(-G(x))^2/2\alpha^2} \quad \text{for } 0 \leq G(x) < \infty .$$

Impulse Distribution

Calling *NOISE_GEN* with the *Impulse* and *Prob* keywords generates an array of impulse, or salt-and-pepper noise. Impulse noise appears as bright and dark spots within an image. The *High* and *Low* keywords control the graylevel values for the salt (bright) and pepper (dark) noise, respectively. Salt and pepper noise are generated with equal probabilities, 0.5 each. In other words, one half of the noise is expected to be pepper noise and one half is expected to be salt noise. The *Prob* keyword indicates the probability that a given pixel in the resulting image will be corrupted by noise. Typical values for *Prob* are from 0.05 to 0.3.

Periodic Distribution

Calling *NOISE_GEN* with the *Periodic* and F_1, \dots, F_8 keywords generates an array of periodic noise. Periodic, or coherent, noise is composed of 2D sinusoidal functions. This is often seen in the form of electrical noise at 60 (or 50) Hz. In the spatial frequency domain, periodic noise corruption is easily seen as bright spot in the spectrum. A notch filter is generally useful in eliminated periodic noise from an image.

The formula for periodic noise is as follows:

$$n(i, j) = A \sin(f_0 i + f_1 j) .$$

Example

```
rayleigh = NOISE_GEN(256, 256, /Rayleigh, Var = 30.0)
normal = NOISE_GEN(256, 256, /Normal)
gamma = NOISE_GEN(256, 256, /Gamma, a = 0.7)
TVSCL, rayleigh
TVSCL, normal
TVSCL, gamma
    ; Generate several noise images and view them.
image = IMAGE_READ(!IP_Data + 'face.tif')
image_rayleigh = IPMATH(image('pixels'), $
    '+', rayleigh + 128.0, /No_Clip)
image_normal = IPMATH(image('pixels'), $
    '+', normal + 128.0, /No_Clip)
image_gamma = IPMATH(image('pixels'), $
    '+', gamma + 128.0, /No_Clip)
    ; Corrupt an image with the noise and view the noisy images.
```

See Also

[NOISE_IMPULSE](#), [NOISE_PERIODIC](#), [NOISE_RAYLEIGH](#)

In the *PV-WAVE: IMSL Statistics Reference*: [RANDOM](#),
[RANDOMOPT](#)

NOISE_IMPULSE Function

Generates an array of impulse noise, also known as salt-and-pepper noise, in a blank array with a specified background; or applies the impulse noise to an existing image array.

Usage

result = NOISE_IMPULSE(*probability*, *d1* [, *d2*, ... , *d8*])

status = NOISE_IMPULSE(*probability*, *image*)

Input Parameters

probability — A floating point scalar, which is the probability of noise in the array. The *probability* parameter values are between 0.0 and 1.0.

d1*, ..., *d8 — The dimensions of the noise array.

NOTE The *d1*, ..., *d8* input parameters are ignored for the *status* usage with the *image* parameter.

image — An array to corrupt by replacing its pixels with noise values.

Returned Value

result — An array corrupted with salt-and-pepper impulse noise in a solid background. The value of background is set using the *Background* keyword. The array data type is specified with the *Type* keyword. (Default: byte).

status — A value indicating the status of the noise corruption of *image*.

- 1 Indicates successful noise corruption.
- 0 Indicates a failed attempt.

Keywords

Amp_Seed — A scalar long value used as the seed for generating random-impulse amplitudes.

Background — A scalar setting for the desired background value of the *result* array. (Default = 127)

NOTE The *Background* keyword is ignored for the *status* usage with the *image* parameter. 2

Binary — If set and nonzero, the generated noise is binary, containing only the high and low values.

High — A scalar value setting for the high, “salt” noise. (Default: 255)

Index_Seed — A scalar long value used as the seed for generating random-impulse noise locations.

Low — A scalar value setting for the low, “pepper” noise. (Default: 0)

Type — A string indicating the desired data type of *result*. Valid strings include: 'byte', 'fix', 'float', or 'double'.

NOTE The *Type* keyword is ignored for the *status* usage with the *image* parameter.

Discussion

Impulse noise is often referred to as salt-and-pepper noise, and usually appears as bright and dark spots within an image. The *High* and *Low* keywords control the gray-level values for the salt (bright) and pepper (dark) noise, respectively. Salt and pepper noise are generated with equal probabilities, 0.5 each. In other words, one half of the noise is expected to be pepper noise and one half is expected to be salt noise. The probability parameter indicates the probability that a given pixel in the resulting image will be corrupted by noise. Typical values for probability are from 0.05 to 0.3.

Example

```
image = IMAGE_READ(!IP_Data + 'airplane.tif')
    ; Read an image.

pixels = image('pixels')
status = NOISE_IMPULSE(0.15, pixels)
    ; Corrupt the image with impulse noise.

image('pixels') = pixels
IMAGE_DISPLAY, image
    ; Display the corrupted image.

filt_image = FILT_NONLIN(image('pixels'), 3, $
    3, /Rankf, Rank_Num = 4)
    ; Attempt to remove the noise with a rank filter.

TVSCL, filt_image
    ; Display the filtered image.
```

See Also

[NOISE_GEN](#), [NOISE_PERIODIC](#), [NOISE_RAYLEIGH](#)

NOISE_PERIODIC Function

Generates an array of periodic (coherent) noise.

Usage

result = NOISE_PERIODIC(*d1*[, *d2*, ... , *d8*])

Input Parameters

d1, ..., *d8* — The dimensions of the noise array.

Returned Value

result — An array containing periodic noise. The array data type is specified using the *Type* keyword.

Keywords

Amp — A scalar float that is the maximum amplitude of the periodic noise. (Default: 1.0)

DC_Offset — A scalar float containing the DC offset (also known as the zero frequency) of the periodic noise. (Default: 0.0)

F1, ..., *F8* — The spatial frequency of the noise in dimensions *d1*, ..., *d8*, respectively. (Default: 0.0)

Type — A scalar string specifying the data type of the returned noise array. Valid strings are: 'byte', 'fix', 'long', 'float', and 'double'. (Default: 'float')

Discussion

Periodic, or coherent, noise is composed of 2D sinusoidal functions. This is often seen in the form of electrical noise at 60 (or 50) Hz. In the spatial frequency domain, periodic noise corruption is easily seen as bright spot in the spectrum. A notch filter is generally useful in eliminated periodic noise from an image.

The formula for periodic noise is as follows:

$$n(i, j) = A \sin(f_0i + f_1j)$$

Example

```
noise = NOISE_PERIODIC(128, 128, F1 = 30.0, $  
    F2 = 100.0, Amp = 15.0, DC_Offset = 128.0)
```

First, get an array of periodic noise.

```
corrupt_image = IPMATH(image, '+', noise, $  
    /No_Clip)
```

Now, corrupt an image with the generated periodic noise.

See Also

[FILT_NOTCH](#), [NOISE_GEN](#), [NOISE_IMPULSE](#),
[NOISE_RAYLEIGH](#)

NOISE_RAYLEIGH Function

Generates an array of Rayleigh distribution noise.

Usage

```
result = NOISE_RAYLEIGH(variance, d1 [, d2, ... , d8])
```

Input Parameters

variance — The desired noise variance.

d1, ..., *d8* — The dimensions of the *result* array.

Returned Value

result — An array of Rayleigh-distributed random numbers. The *result* array data type is set using the *Type* keyword.

Keywords

Seed — A scalar long value specifying the seed used for the random number generator.

Type — A string indicating the desired data type of *result*. Valid strings include: 'byte', 'fix', 'long', 'float', or 'double'. (Default: 'byte')

Discussion

Rayleigh noise is derived from uniform noise and is commonly seen in radar and velocity images. The probability density function for Rayleigh noise is:

$$f(x) = \frac{G(x)}{\alpha^2} e^{(-G(x))^2/2\alpha^2} \quad \text{for } 0 \leq G(x) < \infty$$

NOISE_RAYLEIGH computes a Rayleigh distributed noise array from a uniform noise array as follows:

$$\text{rayleigh} = \sqrt{-2.0 \cdot 2.3299 \cdot \sigma^2 \cdot \ln(1.0 - \text{uniform})}$$

Example

```
image = IMAGE_READ(!IP_Data + 'objects.tif')
; Read an image.

noise = NOISE_RAYLEIGH(15.0, image('width'), $
    image('height'))
; Generate some Rayleigh noise with a variance of 15.0.

noise_image = IPMATH(noise + 128.0, '+', $
    image('pixels'), /No_Clip)
; Add the noise to the image.

filt_image = FILT_MMSE(noise_image, 15.0, $
    3, 3, Edge = 'copy')
; Test the MMSE filter for removing the noise.

TVSCL, filt_image
; Display the filtered image.
```

See Also

[NOISE_GEN](#), [NOISE_IMPULSE](#), [NOISE_PERIODIC](#)

***PAD_IMAGE* Function**

Places a constant border around a volume, image, or signal.

Usage

```
result = PAD_IMAGE(image, xdim[, ydim[, zdim]])
```

Input Parameters

image — A 1D, 2D, or 3D array containing a signal; point or signal-interleaved signals; an image; image, row or pixel interleaved images; or a volume.

xdim — The width setting for the padded volume, image or signal.

ydim — (optional) The height setting for the padded volume or image. (Required for images and volumes.)

zdim — (optional) The depth setting for the padded volume. (Required for volumes.)

Returned Value

result — An array of the same data type as *image*, whose dimensions are set using the *xdim* and, optionally, the *ydim*, and *zdim* parameters.

Keywords

Center — If set, the original image is centered in the padding.

Intleave — A scalar string indicating the type of interleaving of 2D input signals and 3D image arrays. Valid strings and the corresponding interleaving methods are:

'point' — The 2D input array arrangement is (p, x) for p point-interleaved signals of length x .

'signal' — The 2D input *image* array arrangement is (x, p) for p signal-interleaved signals of length x .

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

Pad_Value — A scalar constant used for padding the image.
(Default: 0)

Discussion

It is often useful to increase the dimensions of an image before applying image processing operations to the image. The `PAD_IMAGE` function surrounds an image with a constant valued border. The image may be either centered or placed in the upper left hand corner of the border, by using the *Center* keyword.

Example

```
image = IMAGE_READ(!IP_Data + 'photo.tif')
      ; Read in an image.

square_image = PAD_IMAGE(image('pixels'), $
      512, 512, /Center)
      ; Zero-pad the image to a square size that is a
      ; power of 2 before computing the FFT.

image_fft = FFT(square_image, -1)
      ; Compute the image FFT.
```

See Also

[IPCONVOL](#)

PCT Function

Performs the principle components transform on a multi-layered image.

Usage

result = PCT(*images*)

Input Parameters

images — A 3D array of byte, integer or float data type that contains image, row or pixel-interleaved images.

Returned Value

result — A 3D complex array.

Keywords

Covar — Specifies a variable which holds the covariance matrix of the sample array formed from the images.

Eigvals — Specifies a variable which holds the Eigenvalues of the covariance matrix, a complex array.

Eigvects — Specifies a variable which holds the Eigenvectors of the covariance matrix, a complex array.

Imaginary — If set, returns only the imaginary portion of *result*.

Intleave — A scalar string indicating the type of interleaving of 3D input image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

Mx — Specifies a variable which holds the mean of the sample array formed from the images.

Offset — If set, *result* is offset by its minimum value so that the returned *result* has a minimum of zero.

Pct_trans — Specifies a variable which holds the principle components transformation matrix, a complex array.

Real — If set, returns only the real portion of *result*.

Discussion

The principle components transform (also known as the Hotelling Transform, or the Karhunen-Loève Transform) is commonly used in remote sensing applications. It de-correlates the input images, and the *result* contains the de-correlated images arranged according to the sample variance; that is, the image with the largest sample variance is first, and so on to the image with the smallest sample variance.

The principle components transform is computed by building a sample matrix, *X*, from the input image array. Each sample of *X* is a vector comprised of the corresponding pixels in each image of the array. The transformation matrix specified using the *Pct_trans* keyword is then computed from the eigenvectors of the correlation matrix of *X*.

Example

This example illustrates a principle components transform applied to a pixel-interleaved image.

```
rgb_i = IMAGE_READ(!IP_Data + 'boulder_image.tif')
rgb_i = rgb_i('pixels')
      ; Read in a 24 bit image.

pct_i = PCT(rgb_i, Pct_trans = itrans, Mx = imx, $
           Intleave = 'image')
      ; Compute the principle components transform (PCT).

FOR I = 0,2 DO BEGIN & $
TVSCL, pct_i(*, *, I)) & $
HAK, /Mesg
      ; The first image contains most of the information.
```

See Also

[IPCT](#)

PERIMETER Function

Computes the perimeter of a region in an image.

Usage

result = PERIMETER(*image*, *region*)

Input Parameters

image — A 2D array.

region — A long array containing the pixel element numbers in *image* that compose the region.

Returned Value

result — A long scalar value, in units of pixels, that is the perimeter of the region.

Keywords

None.

Discussion

The perimeter of a region is defined as the number of pixels in its boundary. The boundary of a specified region in *image* is computed by considering the 8-neighbors of every pixel in *image*. Any pixel having an 8-neighbor that is not a member of the region is marked as a boundary pixel. The total number of boundary pixels in the *image* is the region perimeter.

Example

```
image = IMAGE_READ(!IP_Data + 'blobs.tif')
      ; Read an image.
thresh_image = THRESHOLD(image('pixels'), 20, 30, /Binary)
      ; Threshold the image for a single blob.
region = WHERE(thresh_image EQ 255)
      ; Find the pixel number of the object.
perim = PERIMETER(thresh_image, region)
      ; Calculate the perimeter and print it to the screen.
PRINT, perim
```

See Also

[CENTROID](#), [MAJOR_AXIS](#), [MOMENT2D](#)

POLAR_FFT Function

Transforms the FFT of an image or images from a rectangular-coordinate space into a polar-coordinate space, and then sums the polar FFT along ρ and θ .

Usage

result = POLAR_FFT(*image*)

Input Parameters

image — A 2D or 3D array containing an image; or image, row, or pixel-interleaved images.

Returned Value

result — A complex array of the same dimensions and interleaving as the *image* array, which contains the FFT in a polar-coordinate system.

Keywords

Intleave — A scalar string indicating the type of interleaving of 3D input *image* arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

Rho — Specifies a variable, ρ to hold the radial FFT magnitude summation.

Theta — Specifies a variable, θ to hold the angular FFT magnitude summation.

Discussion

The POLAR_FFT function is a spectral approach to texture description. Periodic patterns within an image typically appear as peaks in the Fourier spectrum of the image. These peaks provide information about the periodic patterns, or texture, of the image, such as the direction and the spatial period. Determining pattern direc-

tion and spatial period, however, is simplified by expressing the spectrum in a polar coordinate system. The summation of the polar spectrum along either the radius or the angle component, (ρ and θ , respectively) provides global textural information in a 1D signal. The polar spectrum can also be evaluated for a constant ρ or θ , which also produces a 1D signal. Statistical properties, such as the maximum, mean, and variance, of the 1D signals are useful textural descriptors.

Example

```
image = IMAGE_READ(!IP_Data + 'texture.tif')
        ; Read an image.
IMAGE_DISPLAY, image
```

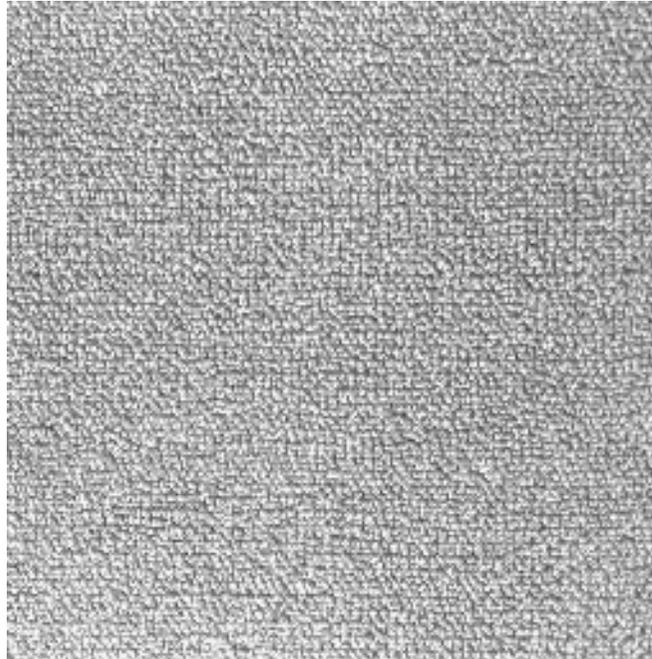


Figure 2-21 The texture image.

```
image_fft = POLAR_FFT(image('pixels'), $
    Theta = theta_sum, Rho = rho_sum)
        ; Compute the polar FFT of the image. Get the
        ; summation along  $\theta$  and  $\rho$  as output variables.
PLOT, rho_sum
```

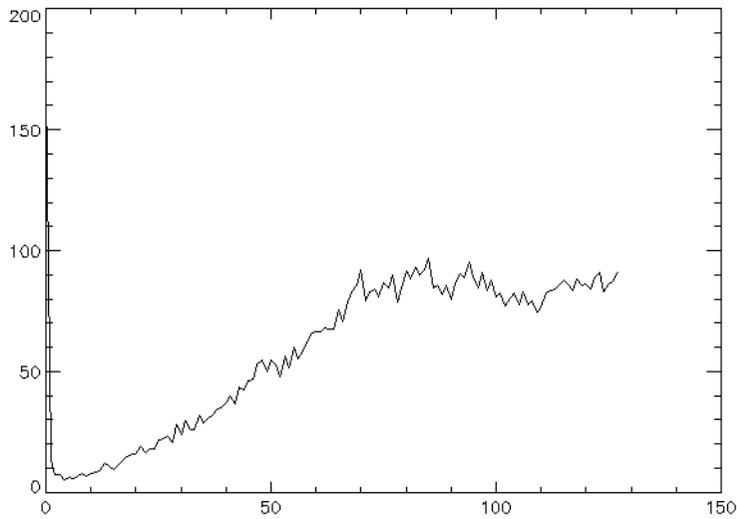


Figure 2-22 The plot of the summation of the polar rho variable.

```
PLOT, theta_sum
; Plot the 1D signals. Peaks in the signals indicate
; texture periodicity and direction.
```

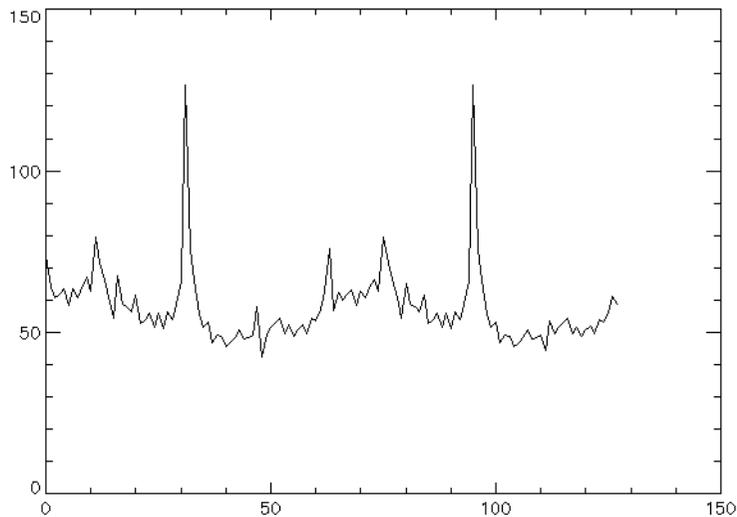


Figure 2-23 The plot of the summation of the polar theta variable.

```
texture_stats = FLTARR(3)
texture_stats(0) = MAX(rho_sum)
texture_stats(1) = AVG(rho_sum)
texture_stats(2) = STDEV(rho_sum, /Variance)
    ; Compute statistics for this textural feature.
PRINT, texture_stats
```

See Also

[GLCM](#), [GLRL](#), [HIST_STATS](#)

RADON Function

Computes the forward radon transform of an image.

Usage

result = RADON(*image*)

Input Parameters

image — A 2D or 3D array containing an image; or image, row or pixel-interleaved images.

Returned Value

result — A 2D or 3D array containing the radon transformation. If *image* is a byte or integer data type, *result* is a long; otherwise, *result* is a double array.

For a 2D *image* input parameter, *result* is a 2D long array whose dimensions are the diagonal of the original image by the value of the *N_Angles* keyword.

For a 3D *image* array (an array of 2D images), *result* is a 3D long array whose dimensions are the diagonal of the original image by the value of the *N_Angles* keyword by the number of images in the array, *p*.

Keywords

Intleave — A scalar string indicating the type of interleaving of 3D input image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

N_Angles — Specifies the number of angles, between 0 and 360, to quantize in the transform calculation. (Default: 360)

Discussion

The Radon transform is composed of the combined projections of an image over all possible angles. Specifically, the image is rotated about its center by a rotation angle. The columns of the rotated image are then summed to form the radon transform result for that particular rotation angle. The rotation angle begins at zero degrees and is increased on each iteration by the angle step size, where the step size is the number of requested angles (*N_Angles*) divided by 360. The rotation and summing technique is continued until the rotation angle reaches 360 degrees. The Radon transform is commonly used in image reconstruction and machine vision.

Example

```
image = IMAGE_READ(!IP_Data + 'blobs.tif')
; Read an image.

image('pixels') = THRESHOLD(image('pixels'), $
    30, 40, /Binary)
; Threshold the image.

radon_image = RADON(image('pixels'), $
    N_Angles = 180)
; Compute the Radon transform over 180 projections.

TVSCL, radon_image
; Display the radon transform.
```

See Also

[HAAR](#), [HOUGH](#), [SLANT](#)

RANGE Function

Computes the range of values in an array.

Usage

result = RANGE(*array*)

Input Parameters

array — An array of any data type except string.

Returned Value

result — The range of values in *array*.

Keywords

None.

Discussion

The range of an array is the maximum value in the array minus the minimum value of the array.

Example

```
image = IMAGE_READ(!IP_Data + 'objects.tif')
      ; Read an image.
image_range = RANGE(image('pixels'))
      ; Find the range of the image.
PRINT, image_range
```

See Also

[ENTROPY](#), [KURTOSIS](#), [MODE](#), [SKEWNESS](#)

REGION_COUNT Function

Computes the number of regions in an image.

Usage

```
result = REGION_COUNT(image)
```

Input Parameters

image — A 2D array containing binary regions. Each binary region must be filled with a different value.

NOTE If *image* does not contain binary regions, the *Search_Values* keyword must be specified.

Returned Value

result — The number of separate regions in *image*.

Keywords

Search_Values — A scalar value greater than zero, or an array containing values greater than zero to search for in *image*. These values are then used as seed points for growing regions. The number of regions grown are then counted.

Seeds — A 1D long array of *n_region* points specifying a variable to receive the element-number of a point in each region.

Discussion

When the *Search_Values* or *Seeds* keywords are defined, regions in the array are grown based on the amplitude values specified with the *Search_Values* keyword; or, regions are grown based on the seed values in *Seed*. The number of regions grown from the seeds are then counted; otherwise, the separate regions in *image* are counted.

Example

```
image = IMAGE_READ(!IP_Data + 'blobs.tif')  
; Read an image.
```

```

values = INDGEN(250) + 1
    ; Look for a maximum of 250 regions.
n_regions = REGION_COUNT(image('pixels'), $
    search_values = values, seeds = region_seeds)
    ; Grow and count regions in the image based on the
    ; amplitude values in the values array.
PRINT, n_regions
    ; Print the actual number of regions found.
PRINT, region_seeds
    ; Print the seed pixel for each region.

```

See Also

[REGION_FIND](#), [REGION_GROW](#), [REGION_STATS](#)

REGION_FIND Function

Locates a possible seed point for a region in an image.

Usage

result = REGION_FIND(*image*, *values*, *n_regions*)

Input Parameters

image — A 2D array containing a homogeneous region.

values — An array containing the amplitude values used to search for regions in *image*.

n_regions — The maximum number of regions to search for in *image*. (0 < *n_regions* < 255)

Returned Value

result — A long array of seed points for regions found in *image*. Each seed point corresponds to an element number in *image*.

Keywords

Fill_Values — A scalar byte value or byte array of amplitudes with which to fill regions in *image*. (0 < *Fill_Values* < 255)

Region_Image — Specifies a variable to hold an array in which the regions found in *image* are identified and filled with the values *Fill_Values*.

Discussion

It is not always convenient to interactively indicate region seeds in an image. The `REGION_FIND` function provides an automated method for locating seed points and growing the regions at those locations. The region finding process begins by searching the image for pixels with amplitudes equal to an element of the *values* array. When a pixel with the specified value is located, it is grown, using the `REGION_GROW` function. In this way, the pixels belonging to the region for which the located pixel is a seed are eliminated from the next search. The image is continuously searched in this way until either no more region seeds can be located or the number of regions, indicated by the *n_regions* parameter, is reached.

Example

Find connected regions in a thresholded image.

```
image = IMAGE_READ(!IP_Data + 'blobs.tif')
      ; Read an image.

thresh_image = THRESHOLD(image('pixels'), 10, $
      200, /Binary)
      ; Look for at least 20 regions of value 255
      ; in thresh_image.

fill_values = BINDGEN(20) + 1B
      ; Fill regions with 1, ..., 20.

seeds = REGION_FIND(thresh_image, 255, $
      20, Region_Image = region_image, $
      Fill_Values = fill_values)

TVSCL, region_image
      ; Display the filled regions.

PRINT, seeds
      ; Print the region seed locations.
```

See Also

[REGION_COUNT](#), [REGION_GROW](#), [REGION_STATS](#)

REGION_GROW Function

Grows homogeneous regions in an image, where the homogeneity is based on the region average.

Usage

```
result = REGION_GROW(image, region_seeds[, threshold])
```

Input Parameters

image — A 2D or 3D array with homogenous regions that contains an image; image, row or pixel-interleaved images; or a volume of any data type except string or complex.

region_seeds — A long array containing the element numbers of pixels in *image* to be used as seed points for growing individual regions. The number of elements in this parameter, *n_regions* must be < 255 elements.

threshold — (optional) A floating point scalar value that is the threshold around the region average used in determining whether or not a pixel belongs to a given region. (Default: 0.1)

Returned Value

result — A 2D or 3D byte array containing filled regions.

Keywords

Fill_Mean — If set, each region is filled with the mean of the image pixels in the region.

Fill_Values — A byte array of values used to fill the regions which were grown, where the values are $0 < \text{Fill_Values}_i < 255$. The number of values in *Fill_Values* is $\{1, \dots, n_regions\}$, where *n_regions* is the number of elements in the *region_seeds* input parameter.

Intleave — A scalar string indicating the type of interleaving of 3D image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for *p* pixel-interleaved images of *x*-by-*y*.

'row' — The 3D *image* array arrangement is (x, p, y) for *p* row-interleaved images of *x*-by-*y*.

'image' — The 3D *image* array arrangement is (x, y, p) for *p* image-interleaved images of *x*-by-*y*.

'volume' — The input image array is treated as a single entity.

Max_Iter — Specifies the maximum number of iterations used to grow a region. A region is grown until it converges (i.e., no more pixels can be added to the region) or *Max_Iter* is reached. (Default: 1000)

Discussion

Region growing is a segmentation method which uses the local pixel amplitude as the segmentation criteria. The REGION_GROW function performs image segmentation by pixel aggregation. Each region begins with a “seed” point as the initial region point. Neighboring pixels within a 3-by-3 area of each region point are then successively added to the region if the pixel value does not alter the region average by more than the value specified using the *threshold* parameter. When a new pixel is added to the region, the region average is updated to reflect all region members. The region growing process ends when no additional neighbors of the region pixels can be added to the region.

NOTE The REGION_MERGE function should be used in place of REGION_GROW when the region seed points are unknown.

Example

```
seeds = LONARR(3)
TVSCL, image
imgdims = SIZE(image, /Dimensions)
CURSOR, x, y, /Device
seeds(0) = LONG(x) + LONG(y)*imgdims(1)
CURSOR, x, y, /Device
seeds(1) = LONG(x) + LONG(y)*imgdims(1)
CURSOR, x, y, /Device
seeds(2) = LONG(x) + LONG(y)*imgdims(1)
region_image = REGION_GROW(image, seeds, 1.0, $
    Fill_Values = [10B, 20B, 30B])
    ; Fill homogenous regions in the image.
```

See Also

[IPCLASSIFY](#), [IPCLUSTER](#), [REGION_COUNT](#),
[REGION_FIND](#), [REGION_STATS](#)

REGION_MERGE Function

Merges homogeneous regions in an image, where the homogeneity is based on the region average.

Usage

```
result = REGION_MERGE(image, max_n_regions[, threshold])
```

Input Parameters

image — A 2D or 3D array of any data type except string or complex that has homogeneous regions containing an image; image, row or pixel-interleaved images; or a volume.

max_n_regions — An integer greater than 0 and less than 255, specifying the maximum number of regions for subdividing *image*.

threshold — (optional) A floating-point scalar threshold value used to determine whether or not a pixel belongs to a given region.
(Default: 0.1)

Returned Value

result — A 2D or 3D byte array containing filled regions.

Keywords

Fill_Mean — If set, each region is filled with the mean of the image pixels in the region.

Fill_Values — A byte array of values used to fill the regions which were merged, where the values are $0 < \textit{Fill_Values}_i < 255$. The number of values in *Fill_Values* is $\{1, \dots, \textit{max_n_regions}\}$.

Intleave — A scalar string indicating the type of interleaving of 3D image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

Discussion

The algorithm for REGION_MERGE analyzes each pixel in the image and attempts to add it to a region. The pixel may be added to a region, if its value does not change region average by more than the threshold.

The REGION_MERGE function is useful for image segmentation and object identification and counting. This function can be used in place of, or in conjunction with, the REGION_GROW function. When the region seed points needed for the REGION_GROW function are unknown, the REGION_MERGE function should be used instead.

When using REGION_MERGE in conjunction with REGION_GROW, use the REGION_GROW function first, making sure that the *Fill_Mean* keyword is set. Then, to reduce the number of regions grown, apply REGION_MERGE with an appropriate threshold value.

Example 1

```
image = IMAGE_READ(!IP_Data + 'sagknee.tif')
      ; Read an image.

region_image = REGION_MERGE(image('pixels'), 254, 2.0)
      ; Find separate regions using merging technique.

n_regions = REGION_COUNT(region_image)
      ; Count the number of objects.

TEK_COLOR

TVSCL, region_image
      ; Display the region image.
```

Example 2

```
image = IMAGE_READ(!IP_Data + 'sagknee.tif')
      ; Read an image.

region_image = REGION_GROW(image('pixels'), region_seeds, 0.5)
      ; Grow separate regions.

n_regions = REGION_COUNT(region_image)
      ; Count the number of individual regions.
```

```
region_image = REGION_GROW(image('pixels'), $
    region_seeds, 0.5, /Fill_Mean)
    ; Fill the regions with their means.

merge_image = REGION_MERGE(region_image, n_regions, 2.0)
    ; See if any regions can be merged.

TEK_COLOR
TVSCL, merge_image
    ; Display the final image.
```

See Also

[IPCLUSTER](#), [REGION_GROW](#), [REGION_SPLIT](#),
[THRESH_ADAP](#), [THRESHOLD](#)

REGION_SPLIT Function

Splits homogeneous regions in an image, where the homogeneity is based on the region range.

Usage

```
result = REGION_SPLIT(image[, threshold])
```

Input Parameters

image — A 2D or 3D array of any data type except string or complex having homogeneous regions containing an image; image, row or pixel-interleaved images; or a volume.

threshold — (optional) A floating-point scalar threshold value used to determine whether or not a pixel belongs to a given region.
(Default: 0.1).

Returned Value

result — A 2D or 3D byte array containing filled regions.

Keywords

Intleave — A scalar string indicating the type of interleaving of 3D image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

Max_Iter — Specifies the maximum number of iterations used to split a region.
(Default: 1000)

Discussion

REGION_SPLIT works best on images with separate square or rectangularly shaped regions. It is a recursive algorithm that subdivides the image into smaller and smaller units until either a homogeneous region is left or the maximum number of iterations has been reached.

NOTE Because the algorithm is recursive, *Max_Iter* should be kept reasonably small to avoid stack overflow problems.

Example

```
image = IMAGE_READ(!IP_Data + 'objects.tif')
        ; Read an image.
region_image = REGION_SPLIT(image('pixels'), 1.0)
        ; Find separate regions using splitting technique.
TEK_COLOR
TVSCL, region_image
        ; Display the region image.
```

See Also

[IPCLUSTER](#), [REGION_GROW](#), [REGION_MERGE](#), [THRESH_ADAP](#), [THRESHOLD](#)

REGION_STATS Function

Performs several statistical calculations on specified regions in an image.

Usage

```
result = REGION_STATS(image[, region_image])
```

Input Parameters

image — An image with binary regions. This parameter must be a byte array data type, if *region_image* is not given.

region_image — (optional) A 2D byte array with binary regions. This parameter is used as a mask to identify the pixels that compose each region in *image*.

Returned Value

result — A 2D double array of statistics with dimensions 11-by-*n_regions*, where *n_regions* is the number of regions for which the statistics are computed. The 11 statistics of the regions in *image* are as follows:

area — The number of pixels in the region.

mean — The average of pixels in the region.

min — The minimum pixel value in the region.

max — The maximum pixel value in the region.

stdev — The standard deviation of the pixel values in the region.

mode — The most frequent pixel value in the region.

range — The difference between the minimum and the maximum value in the region.

perimeter — The distance around the border of the region in pixels.

compactness — A measure of the compactness of the region. Defined as the perimeter squared over the area.

major axis angle — The angle of the major axis through the region. The major axis is directed in the area of maximal region dispersion.

centroid — The centroid of the region, given as the element number in the image.

Keywords

Region_Values — A 1D byte array specifying the regions in *image* or *region_image* for which to compute statistics. (Default: all regions in *image* or *region_image*)

Discussion

Statistical measures on regions within an image provide valuable quantitative descriptions of segmented objects. The statistical measures performed by the REGION_STATS function for each region value, *i*, are defined as follows:

$Area_i = N$, where *N* is the number of elements in *region_image* equal to *i*.

$Mean_i = TOTAL(x)/N$, where *N* is the number of elements in *region_image* equal to *i*, and *x* is the subset of pixels in *image* where *region_image* is equal to *i*.

$Min_i = MIN(x)$, where *x* is the subset of pixels in *image* where *region_image* is equal to *i*.

$Max_i = MAX(x)$, where *x* is the subset of pixels in *image* where *region_image* is equal to *i*.

$Stdev_i = STDEV(x)$, where *x* is the subset of pixels in *image* where *region_image* is equal to *i*.

$Mode_i = MODE(x)$, where *x* is the subset of pixels in *image* where *region_image* is equal to *i*.

$Range_i = RANGE(x)$, where *x* is the subset of pixels in *image* where *region_image* is equal to *i*.

$Perimeter_i = PERIMETER(x, e)$, where *x* is the subset of pixels in *image* where *region_image* is equal to *i*, and *e* is an array of the element numbers in *region_image* that are equal to *i*.

$Compactness_i = [PERIMETER(x, e)*PERIMETER(x, e)]/Area_i$, where *x* is the subset of pixels in *image* where *region_image* is equal to *i*, and *e* is an array of the element numbers in *region_image* that are equal to *i*.

$Major Axis_i = MAJOR_AXIS(x, e)$, where *x* is the subset of pixels in *image*

where *region_image* is equal to *i*, and *e* is an array of the element numbers in *region_image* that are equal to *i*.

$Centroid_i = \text{CENTROID}(x, e)$, where *x* is the subset of pixels in *image* where *region_image* is equal to *i*, and *e* is an array of the element numbers in *region_image* that are equal to *i*.

Example

```
knee = IMAGE_READ(!IP_Data + 'sagknee.tif')
; Read an image.

cluster_seeds = [20, 30, 500, 1000, 2000]

region_image = IPCLUSTER(knee('pixels'), $
    cluster_seeds, Fill_Values = [1, 2, 3, 4, 5])
; Find the regions in the image using K-mean
; segmentation. Choose random points for the
; cluster seeds.

result = REGION_STATS(knee('pixels'), region_image)
; Compute the statistics of the original image pixels
; underlying the identified regions.

PRINT, result
; View the statistics for each region.
```

See Also

[CENTROID](#), [IPCLUSTER](#), [MAJOR_AXIS](#), [MODE](#),
[PERIMETER](#), [RANGE](#), [REGION_COUNT](#),
[REGION_FIND](#), [REGION_GROW](#)

SHIFT_EDGE Function

Performs edge enhancement on an image using either the shift and subtract, or the shift and XOR technique.

Usage

```
result = SHIFT_EDGE(image[, subt_or_xor])
```

Input Parameters

image — A 2D or 3D array containing an image; image, row or pixel-interleaved images; or a volume.

subt_or_xor — (optional) A scalar specifying the enhancement technique to use:

- 0 Exclusive OR technique
- 1 Subtraction technique (default)

When the exclusive OR shift technique is used (*subt_or_xor* = 0), the input parameter *image* must be a byte, integer, or long data type; no other data types are valid for the exclusive OR technique. (Default: 1, the subtraction technique)

Returned Value

result — An array containing the enhanced data.

Keywords

Intleave — A scalar string indicating the type of interleaving of 3D image arrays. Valid strings and the corresponding interleaving methods are:

'pixel' — The input array arrangement is (*p*, *x*, *y*) for *p* pixel-interleaved images of *x*-by-*y*.

'row' — The 3D *image* array arrangement is (*x*, *p*, *y*) for *p* row-interleaved images of *x*-by-*y*.

'image' — The 3D *image* array arrangement is (*x*, *y*, *p*) for *p* image-interleaved images of *x*-by-*y*.

'volume' — The input image array is treated as a single entity.

ShiftArr — An array of shift parameters. For an *image* parameter of *m* dimensions, *ShiftArr* contains *m* elements specifying the shift parameter for each dimension. The *ShiftArr* keyword can be used in place of the *Xshift* and *Yshift* keywords.

Xshift — The distance, in pixels, to shift *image* in the *x*-direction. The *Xshift* keyword can also be used as an array defining the shift in the *x*-direction for each image when *image* is a 2D array or 3D image-interleaved array. (Default: 1 pixel)

Yshift — The distance, in pixels, to shift *image* in the *y*-direction. The *Yshift* keyword can also be specified as an array defining the shift in the *y*-direction for each image when *image* is a 2D array or 3D image-interleaved array. (Default: 1 pixel)

Discussion

When the subtraction operation is used, the `SHIFT_EDGE` function creates an embossing effect on grayscale images.

The direction and amount of the shift reveals specific details in an otherwise jumbled image. For example, a horizontal shift can highlight a specific pattern in an image; likewise, for a vertical shift.

Typically, a single-element (one pixel) shift produces more pronounced edges, while shifts of more than ten elements begin to blur features in the image.

Examples

These examples are a collection of partial code designed to illustrate the many uses of the `SHIFT_EDGE` function and its keywords.

In this example, the `SHIFT_EDGE` function is used to highlight the edges to the left of each feature in the mandrill image.

```
mandril_edge = SHIFT_EDGE(mandrill, Yshift = 0)
```

Here, `SHIFT_EDGE` is used to highlight the edges below and to the left of each feature:

```
mandril_edge = SHIFT_EDGE(mandrill)
```

To perform edge enhancement on a group of images in one function call, do the following:

```
edge_set = SHIFT_EDGE(images, $
    Intleave = 'images')
edge_image = SHIFT_EDGE(image, 0, $
    Xshift = 2, Yshift = 2)
```

Another way to do this is:

```
edge_image = SHIFT_EDGE(image, 0, $  
    ShiftArr = [2, 2])
```

Here is an example using a 5D array:

```
arr5D = INDGEN(20, 5, 7, 3, 2)  
edge_5D = SHIFT_EDGE(arr5D, ShiftArr = $  
    [1, 2, 1, 1, 2])
```

See Also

[CANNY](#), [IPCONVOL](#)

In the *PV-WAVE Reference*: [ROBERTS](#), [SOBEL](#)

***SKELETONIZE* Function**

Performs the morphologic skeletonizing operation for shape processing.

Usage

```
result = SKELETONIZE(image, structure[, x0, y0])
```

Input Parameters

image — The array to be skeletonized.

structure — A 1D or 2D array containing the structuring elements. The structure elements are interpreted as binary elements with values of either zero or nonzero, unless the *Gray* keyword is used.

x0 — (optional) The *x*-coordinate of the structure's origin.

y0 — (optional) The *y*-coordinate of the structure's origin.

Returned Value

result — A skeletonized image of the same size and dimensions as *image*.

Keywords

Gray — If set, indicates that gray-scale erosion and dilation is to be used. (Default: binary erosion and dilation)

Max_Iter — Specifies the maximum number of iterations to perform in the skeletonization process. (Default: 100)

Values — An array of values of the structuring element. The *Values* array must have the same dimensions and number of elements as the *structure* parameter.

Discussion

Morphological operations are defined for grayscale byte images. If *image* is not originally of type byte, PV-WAVE makes a temporary copy of *image* that is of type byte before using it for the morphological processing.

The optional parameters *x0* and *y0* specify the row and column coordinates of the structuring element's origin. If these parameters aren't used, the structuring ele-

ment origin is set to the center, $(Nx/2, Ny/2)$, where Nx and Ny are the dimensions of the structuring element.

The skeletonization of an object describes its structure. Skeletonization is also referred to as the medial axis transform. The result of a call to `SKELETONIZE` is an image consisting of the set of points that are equidistant from the boundary of an object. These points are the “medial axis” of the object.

The skeletonization process is performed using loops in which *image* is logically ORed with the difference of the erosion and morphological opening operations. The looping process ends when either the maximum iteration is reached, or the next pass through the loop would result in a completely eroded (null) image. In other words, skeletonization is the union of the difference between the *i*-th eroded image and the opening of the *i*-th eroded image. Where *i* is the minimum of *Max_Iter* or the iteration which produces the null image. The image returned by the skeletonization process is then the union of all *i* difference images.

Example

```
blobs = IMAGE_READ(!IP_Data + 'blobs.tif')
      ; Read an image.

blobs = THRESHOLD(blobs('pixels'), 20, /Binary)
      ; Threshold the objects to form a binary image.

str_ele = BYTARR(3, 3)
str_ele(*) = 1B
str_ele(1, 1) = 0B
str_ele(0, 0) = 0B
str_ele(0, 2) = 0B
str_ele(2, 0) = 0B
str_ele(2, 2) = 0B
      ; Create a structuring element.

skel_image = SKELETONIZE(blobs, str_ele)
      ; Skeletonize the objects using the circular structuring element.

TVSCL, skel_image
      ; Display the skeletonized image.
```

See Also

[HIT_MISS](#), [MORPH_CLOSE](#), [MORPH_OPEN](#), [TOP_HAT](#)

In the *PV-WAVE Reference*: [ERODE](#), [DILATE](#)

SKEWNESS Function

Computes the skewness of an array.

Usage

result = SKEWNESS(*array*)

Input Parameters

array — An array of any data type except string.

Returned Value

result — A scalar value containing the skewness of the array.

Keywords

None.

Discussion

Computes the skewness of *array(k, l)* as:

$$skewness = \frac{1}{(K \cdot L)} \sum_{k=0}^{K-1} \sum_{l=0}^{L-1} \left(\frac{array(k, l) - mean}{std} \right)^3,$$

where *mean* = AVG(*array*) and *std* = STDEV(*array*).

The skewness is a useful measure for statistical texture analysis.

Example

```
image = IMAGE_READ(!IP_Data + 'noise_test.tif')
; Read an image.

skew = SKEWNESS(image('pixels'))
PRINT, 'Skewness = ', skew
; Compute the skewness of the image.
```

See Also

[ENTROPY](#), [KURTOSIS](#), [MODE](#), [RANGE](#), [UNIFORMITY](#)

SLANT Function

Performs a Slant transform on a 2D square image. Images which are not already square, are zero-padded to have square dimensions before the transform is applied. Images whose dimensions are not a power of two are padded to have dimensions that are the nearest power of two.

Usage

result = SLANT(*image*[, *direction*])

Input Parameters

image — A 2D array.

direction — (optional) A parameter specifying the direction of the transform. (Default: -1)

- 1 The forward Slant transform is applied (default).
- 1 The reverse Slant transform is applied.

Returned Value

result — A 2D floating-point matrix with dimensions N -by- N , where N is the largest dimension of the input *image* rounded to the nearest larger power of two.

Keywords

None.

Discussion

The Slant transform uses sawtooth shaped basis functions, $s_k(z)$, which are defined recursively from the 2-by-2 basis. The equations for the Slant transform matrix generation are found in Gonzalez and Woods, p. 147.

The Slant Transform is then applied as:

$T = S * image * S$, where S is the Slant transform matrix and T is the transform result.

Example

```
image = IMAGE_READ(!IP_Data + 'airplane.tif')
; Read an image.

slant_image = SLANT(image('pixels'))
; Compute the Slant transform.

TVSCL, IPALOG(slant_image)
; Display the transformed image.
```

See Also

[DCT](#), [HAAR](#)

THRESH_ADAP Function

Performs adaptive thresholding on an array.

Usage

```
result = THRESH_ADAP(image, wxdim[, wydim[, wzdim]])
```

Input Parameters

image — A 1D, 2D, or 3D array containing a signal; point or signal-interleaved signals; an image; image, row or pixel interleaved images; or a volume.

wxdim — The width of the threshold window. This value is $wxdim \leq$ the width of *image*.

wydim — (optional) The height of the threshold window. This value is $wydim \leq$ the height of *image*. (Required for images and volumes.)

wzdim — (optional) The depth of the threshold window. This value is $wzdim \leq$ the depth of *image*. (Required when *image* is a volume.)

Returned Value

result — An array of data type byte, if binary thresholding is performed; otherwise, the array returned is the same data type as *image*.

Keywords

Binary — If set, binary thresholding is performed.

Intleave — A scalar string indicating the type of interleaving of 2D input signals and 3D image arrays. Valid strings and the corresponding interleaving methods are:

'point' — The 2D input array arrangement is (p, x) for p point-interleaved signals of length x .

'signal' — The 2D input *image* array arrangement is (x, p) for p signal-interleaved signals of length x .

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-inter-

leaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

Lower_Tol — The lower tolerance used for the threshold comparison. Either *Tolerance*, or both *Upper_Tol* and *Lower_Tol* must be specified. (Default: *Tolerance*)

Set_False — The value that the result is set to when the comparison is false. (Default: 0)

NOTE The *Set_False* keyword is ignored if *Binary* = 0.

Set_True — The value that the result is set to when the comparison is true. (Default: 255)

Stat_Type — A scalar string indicating the statistical function to compute within the threshold window. Valid strings are: 'avg', 'min', 'max', 'median', 'stdev', or 'range'. (Default: 'avg')

Tolerance — The tolerance around the statistical calculation used for the threshold comparison. Either *Tolerance*, or both *Upper_Tol* and *Lower_Tol* must be specified.

Upper_Tol — The upper tolerance used for the threshold comparison. Either *Tolerance*, or both *Upper_Tol* and *Lower_Tol* must be specified. (Default: *Tolerance*)

Discussion

The purpose of image segmentation is to separate the image into regions possessing similar characteristics. General thresholding uses only the pixel amplitude as the segmentation characteristic. Some difficulties arise with using only the pixel amplitude for thresholding, such as: choosing an absolute pixel amplitude at which to threshold can be tough, and sometimes noise or other undesired artifacts are incorrectly grouped with the desired objects. Using the THRESH_ADAP function is one way around such difficulties.

Adaptive thresholding provides a method for rejecting local pixel values based on a tolerance around a statistical measure within a local image window. Adaptive thresholding also does not require an absolute pixel amplitude threshold value. Instead, the threshold value is constantly changed based on a local image statistical measure.

For all statistical measures (using the keyword *Stat_Type*) except the standard deviation, a pixel is set to *Set_True* if it is within \pm the upper and lower tolerances, respectively, of the local statistical measure. For the standard deviation, a pixel is set to *Set_True* if it is within \pm the upper and lower tolerances of the local mean \pm the local standard deviation.

Example 1

```
image = IMAGE_READ(!IP_Data + 'xray.tif')
      ; Read an image.

seg_image = THRESH_ADAP(image('pixels'), $
      5, 5, /Binary, Stat_Type = 'avg', $
      Tolerance = 1.5)
      ; Segment the image using adaptive thresholding
      ; based on the average value in a 5-by-5 window.
      ; The tolerance around the window average will
      ; be  $\pm 1.5$ .

TVSCL, seg_image
      ; Display the binary segmented image.
```

Example 2

```
image = IMAGE_READ(!IP_Data + 'objects.tif')
      ; Read an image.

seg_image = THRESH_ADAP(image('pixels'), $
      3, 3, /Binary, Stat_Type = 'stdev', $
      Tolerance = 2.0)
      ; Segment the objects using the standard deviation.
      ; The tolerance will be  $\pm 2.0$  of the window
      ; average  $\pm$  the standard deviation.

TVSCL, seg_image
      ; Display the binary segmented image.
```

See Also

[THRESHOLD](#)

THRESHOLD Function

Performs either binary or grayscale global thresholding on an image.

Usage

result = THRESHOLD(*image*, *a*[, *b*])

Input Parameters

image — A 1D, 2D or 3D array containing a signal; point or signal-interleaved signals; an image; image, row, or pixel-interleaved images; or a volume.

a — The value on the left-hand side of the threshold equation. This parameter is a scalar for a single signal, image, or volume; or an array for an interleaved arrangement of signals or images.

b — (optional) The value on the right-hand side of the threshold equation. This parameter is a scalar for a single signal, image, or volume; or an array for an interleaved arrangement of signals or images.

Returned Value

result — An array of data type byte (for binary thresholding), or of the same data type as *image* (grayscale thresholding). The data is thresholded in the following manner:

For two input parameters, *image* and *a*, all values in *image* which are greater than or equal to *a* are set to the value of the *Set_True* keyword.

For three input parameters, *image*, *a*, and *b*, all values in *image* which are greater than or equal to *a* and are less than or equal to *b* are set to the value of the *Set_True* keyword.

Keywords

A_Neq — If set, the comparison on the left-side of the threshold equation does not include the value of *a*.

Binary — If set, binary thresholding is performed; otherwise, grayscale thresholding is performed. (Default: 0)

B_Neq — If set, the comparison on the right-side of the threshold equation does not include the value of *b*.

Intleave — A scalar string indicating the type of interleaving of 2D input signals and 3D image arrays. Valid strings and the corresponding interleaving methods are:

'point' — The 2D input array arrangement is (p, x) for p point-interleaved signals of length x .

'signal' — The 2D input *image* array arrangement is (x, p) for p signal-interleaved signals of length x .

'pixel' — The input array arrangement is (p, x, y) for p pixel-interleaved images of x -by- y .

'row' — The 3D *image* array arrangement is (x, p, y) for p row-interleaved images of x -by- y .

'image' — The 3D *image* array arrangement is (x, y, p) for p image-interleaved images of x -by- y .

'volume' — The input image array is treated as a single entity.

Inverse — If set, the comparison relationships of *image*, and a , and (optionally) b are reversed. This creates disjoint regions for thresholding as follows:

For the two input parameters, *image* and a , all values in *image* which are less than or equal to a are set to the value of the *Set_True* keyword.

For the three input parameters, *image*, a , and b , all values in *image* which are less than or equal to a , or are greater than or equal to b are set to the value of the *Set_True* keyword.

Set_False — The amplitude of *result* where the comparison is false. (Default: 0)

NOTE The *Set_False* keyword is used only for binary thresholding.

Set_True — The amplitude of *result* where the comparison is true. (Default: 255)

Discussion

Thresholding is used as a simple segmentation method for identifying objects or regions within an image that have consistent graylevels.

Example

To threshold an image, x , such that values between, but not including, 30 and 55 are set to 190, and all other values in x are 10, do the following:

```
result = THRESHOLD(x, 30, 55, Set_True = 190, $
    Set_False = 10, /Binary, /A_Neq, /B_Neq)
```

This is conceptually the same as the following logic statement:

If $30 < x(i, j) < 55$, then $x(i, j) = 190$, else $x(i, j) = 10$, where $0 \leq i < N$ and $0 \leq j < M$ for all $x(N, M)$.

See Also

[DENSITY_SLICE](#), [THRESH_ADAP](#)

TOP_HAT Function

Performs the morphologic top-hat transform for shape processing.

Usage

```
result = TOP_HAT(image, structure[, x0, y0])
```

Input Parameters

image — A 2D array.

structure — A 1D or 2D array containing the structuring element. The elements are interpreted as binary values (either zero or nonzero), unless the *Gray* keyword is used.

x0 — (optional) The *x*-coordinate of the structuring element's origin.

y0 — (optional) The *y*-coordinate of the structuring element's origin.

Returned Value

result — An array of the same size and dimensions as *image* containing the top-hat transformed image.

Keywords

Gray — If set, grayscale, rather than binary erosion is used.

Valley — If set, the valley detector top-hat transform is performed; otherwise, the peak detector transform is used.

Values — An array of the same dimensions and number of elements as *structure* containing the values of the structuring element.

Discussion

Morphological operations are defined for grayscale byte images. If *image* is not originally of type byte, PV-WAVE makes a temporary copy of *image* that is of type byte before using it for the morphological processing.

The morphological top-hat transform is used to identify small pixel clusters and edges. The peak detection top-hat operator is defined as the original image minus the opened image. The valley detection top-hat operator is defined as the closed image minus the original image.

Example

```
morph_struct = BYTARR(3, 3)
morph_struct(*) = 1B
morph_struct(1, 1) = 0B
    ; Make a square structuring element.
test_image = IMAGE_READ(!IP_Data + 'objects.tif')
    ; Read an image.
tophat_image = TOP_HAT(test_image('pixels'), morph_struct)
    ; Perform the top-hat transform to find regional peaks.
TVSCL, tophat_image
```

See Also

[HIT_MISS](#), [MORPH_CLOSE](#), [MORPH_OPEN](#),
[MORPH_OUTLINE](#), [SKELETONIZE](#)

In the *PV-WAVE Reference*: [DILATE](#), [ERODE](#)

UNIFORMITY Function

Computes the uniformity of an array.

Usage

result = UNIFORMITY(*array*)

Input Parameters

array — An array of any data type except string.

Returned Value

result — A floating point scalar value that is the array uniformity.

Keywords

None.

Discussion

The uniformity of an array is defined as:

$$U = \sum_{k=0}^{N-1} \sum_{l=0}^{M-1} (array(k, l))^2,$$

for an array, *array*(*N*, *M*).

Uniformity is a useful statistical measure for textural analysis.

Example

```
image = IMAGE_READ(!IP_Data + 'noise_test.tif')
; Read an image.

uniform = UNIFORMITY(image('pixels'))

PRINT, 'Uniformity = ', uniform
; Compute the uniformity of the image.
```

See Also

[ENTROPY](#), [KURTOSIS](#), [MODE](#), [RANGE](#), [SKEWNESS](#)

Bibliography

- Ahrens, J. H., and Dieter, U. (1974), Computer methods for sampling from gamma, beta, Poisson, and binomial distributions, *Computing*, **12**, 223–246.
- Akansu, A.N., and Haddad, R. A. (1992), *Multiresolution Signal Decomposition: Transforms, Subbands, and Wavelets*, Academic, Boston, MA.
- Atkinson, A.C. (1979), A family of switching algorithms for the computer generation of beta random variates, *Biometrika*, **66**, 141–145.
- Cheng, R.C.H. (1978), Generating beta variates with nonintegral shape parameters, *Communications of the ACM*, **21**, 317–322.
- Chui, C. K., (1992) *An Introduction to Wavelets*, Academic Press, New York, NY.
- Cooley, J.W., and J.W. Tukey (1965), An algorithm for the machine computation of complex Fourier series, *Mathematics of Computation*, **19**, 297–301.
- Daubechies, I. (1988), Orthonormal Bases of Compactly Supported Wavelets, *Communications on Pure Applied Mathematics*, **41**, 909–996.
- Daubechies, I. (1992), *Ten Lectures on Wavelets*, Society for Industrial and Applied Mathematics, Philadelphia, PA.
- Harris, F. J. (1978), On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform, *Proceedings of the IEEE*, **66**, **1**, 51–83
- Jackson, L. B. (1991), *Signals, Systems, and Transforms*, Addison-Wesley, Reading, MA.

- Jöhnk, M.D. (1964), Erzeugung von Betaverteilten und Gammaverteilten Zufallszahlen, *Metrika*, **8**, 5–15.
- Kay, S. M. (1987), *Modern Spectral Estimation: Theory and Application*, Prentice-Hall, Englewood Cliffs, NJ.
- Marple, S. L. (1987), *Digital Spectral Analysis with Applications*, Prentice-Hall, Englewood Cliffs, NJ.
- Rioul, O, and Vetterli, M., (1991) Wavelets and Signal Processing, *IEEE Signal Processing Magazine*, October, 14–38.
- Schmeiser, Bruce (1983), Recent advances in generating observations from discrete random variates, *Computer Science and Statistics: Proceedings of the Fifteenth Symposium on the Interface*, (edited by James E. Gentle), North-Holland Publishing Company, Amsterdam, 154–160.
- Schmeiser, Bruce W., and A.J.G. Babu (1980), Beta variate generation via exponential majorizing functions, *Operations Research*, **28**, 917–926.
- Schmeiser, Bruce, and Voratas Kachitvichyanukul (1981), *Poisson Random Variate Generation*, Research Memorandum 81-4, School of Industrial Engineering, Purdue University, West Lafayette, IN.
- Vaidyanathan, P. P. (1993), *Multirate Systems and Filter Banks*, Prentice-Hall, Englewood Cliffs, NJ.
- Welch, P. D. (1967), The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms, *IEEE Transactions on Audio Electroacoustics*, AU-15, **2**, 70–73.

Image Processing Index

A

- adaptive filters 11, 32, 39
- algebraic operations 7, 110
- alpha-trimmed mean filter 42
- AT mean filter
 - See alpha-trimmed mean filter

B

- bandpass filter 9, 35
- bandstop filter 9, 35
- Bartlett's method
 - estimate for rectangular window 122
 - non-overlapping data segments 122
 - power spectrum estimate 121
- beta distribution 153
- biorthogonal wavelet filter 114
- Blackman window 129, 130
- BLEND function 17
- boxcar window
 - See rectangular window
- Butterworth filter 35

C

- CANNY function 19
- centroid computation 140
- CENTROID function 20
- CH mean filter
 - See contra-harmonic mean filter
- Chebyshev window 129, 130
- Cholesky factor 154
- classification
 - maximum likelihood 84
 - overview 15
- CMY color model 14

- coherent noise 158
- Coifman wavelet filter 113
- color conversion 92, 107, 135
- color image processing techniques 14
- color models 14
- Color Tool 4
- common variables
 - See IP_COMMON
- Contour Tool 4
- contra-harmonic mean filter 42
- correlation 99

D

- Daubechies wavelet filter 113
- DCT function 22
- density function 105
- density slicing 14
- DENSITY_SLICE function 24
- dilation 142, 144, 146
- discrete cosine transform 13
- DIST_MAP function 26

E

- edge detection 14, 19, 27, 42, 146, 185, 199
- ENTROPY function 31
- Erlang distribution 153
- erosion 142, 144, 146
- Euclidian distance map 26
- exiting the Toolkit 6
- expanding an image 117
- exponential
 - deviate 152
 - distribution 152

F

- fast Fourier transform 13, 166
- FFT 13, 166
- FILT_DWMTM function 32
- filter
 - alpha-trimmed mean 42
 - Canny 19
 - circularly symmetric 37, 52
 - contra-harmonic mean 42
 - geometric mean 43
 - least mean square 60
 - linear 9, 19, 56, 58, 102
 - maximum 44
 - minimum 44
 - mode 44
 - nonlinear 10, 42
 - objects 115
 - overview 8
 - quadrature mirror 114
 - range 44
 - rank 44
 - reading from a file 115
 - saving in a file 133
 - spatial 62, 115
 - spectral 115
 - Wiener 58
 - Yp mean 45
- filter method
 - adaptive 11, 32, 39
 - edge detection 10, 19
 - frequency domain 9, 35, 51, 58
 - spatial domain 9, 19, 42, 56, 95, 102, 115
 - spectral domain 35, 51, 58, 115
- FILT_FREQ function 35
- FILT_MMSE function 39
- FILT_NONLIN function 42
- FILT_NOTCH function 51
- FILT_SMOOTH function 56
- FILT_WEINER function 58
- forward radon transform 169
- frequency domain filtering 9, 35, 51

G

- gamma distribution 152
- gamma function 153
- Gaussian distribution 152

- GAUSS_KERNEL function 62
- geometric mean filter 43
- geometric transforms 13
- GLCM function 63
- GLCM_STATS function 65
- GLRL function 67
- GLRL_STATS function 69
- graylevel co-occurrence matrix 63
- graylevel run length matrix 67

H

- HAAR function 71
- Hamming window 129, 131
- Hanning window 129, 131
- help
 - context-sensitive online 5
 - manuals online 5
- highpass filter 9
- histogram operations 8, 74, 106
- Histogram Tool 4
- HIST_STATS function 74
- HIT_MISS function 76
- Hotelling transform 12, 13, 164
- HOUGH function 78
- HSV color model 14

I

- Image Processing Navigator help 5
- Image Processing Toolkit
 - Navigator 4
 - online documentation 5
 - online help 5
 - starting (loading) 3
 - stopping (unloading) 6
- Image Tool 4
- initializing the Toolkit 3
- intensity slicing 24
- inverse CDF method
 - beta distribution 153
 - Poisson distribution 152
- IPALOG function 83
- IPCLASSIFY function 84
- IPCLUSTER function 88
- IPCOLOR_24_8 function 92
- IP_COMMON, common variables 6
- IPCONVOL function 95
- IPCORRELATE function 99

IPCREATE_FILTER function 102
 IPCT function 103
 IPHISTOGRAM function 105
 IPLINEAR_GRAY function 107
 IPMATH function 109
 ipnavigator 4
 IPQMFDESIGN function 113
 IPREAD_FILTER function 115
 IPSCALE function 117
 IPSPECTRUM function 120
 ip_startup 3
 IPSTATS function 124
 ip_unload 6
 IPWAVELET function 126
 IPWIN function 129
 IPWRITE_FILTER function 133
 IS_GRAY_CMAP function 135

K

Kaiser window 129, 131
 Karhunen-Loève transform 13, 164
 K-means clustering 88
 KURTOSIS function 137

L

least mean square filter 60
 linear filters 9
 logical operations 7
 lowpass filter 9, 35, 57

M

MAJOR_AXIS function 138
 maximum filter 44
 mean filter

- alpha-trimmed 42
- contra-harmonic 42
- geometric 43
- Yp 45

 medial axis transform 189
 mensuration 11
 minimum filter 44
 mode filter 44
 MODE function 139
 MOMENT2D function 140
 moments 20, 140
 MORPH_CLOSE function 142

morphological operation

- closing a shape 142
- hit-or-miss transform 76
- opening a shape 144
- outlining 146
- overview 11
- shape definition 76
- skeletonizing 188
- top-hat transform 198

 MORPH_OPEN function 144
 MORPH_OUTLINE function 146
 multivariate normal distribution 153

N

Navigator

- online help 5
- starting 4
- VDA tools 4

 noise

- generating 149, 158
- impulse 156
- periodic 158
- Rayleigh distribution 159
- removing 11, 32, 39, 42, 53, 57

 NOISE_GEN function 149
 NOISE_IMPULSE function 156
 NOISE_PERIODIC function 158
 NOISE_RAYLEIGH function 159
 nonlinear filters 10, 42
 non-stationary input signals 123
 normal distribution 152
 notch filter 51
 Nyquist normalized frequency 122

O

online documentation system 5
 online help 5

P

PAD_IMAGE function 161
 PCT function 163
 PERIMETER function 165
 periodogram

- Bartlett's method 122
- equation 121, 122
- power spectrum estimate 121

- Welch's method 122
- Plot Tool 4
- point operation
 - blending 17
 - math and logic 109
 - overview 6
- Poisson distribution 152
- polar-coordinate space 166
- POLAR_FFT function 166
- power spectrum estimate
 - Bartlett's method 121
 - modified periodogram 121
 - periodogram 121
 - Welch's method 121
- principle components transform 13, 103, 163
- pseudo-color image processing 14
- PTPE method 152

Q

- quadrature mirror filter 114

R

- RADON function 169
- Radon transform 170
- random numbers
 - beta distribution 153
 - exponential distribution 152
 - Gamma distribution 152
 - generator seed 151
 - multivariate normal distribution 153
 - normal distribution 152
 - Poisson distribution 152
- range filter 44
- RANGE function 171
- rank filter 44
- Rayleigh distributed noise 160
- rectangular window 122, 129, 131
- rectangular-coordinate space 166
- REGION_COUNT function 172
- REGION_FIND function 173
- REGION_GROW function 175
- REGION_MERGE function 177
- regions
 - counting 172
 - growing 175
 - merging 177

- merging versus growing 178
- overview 14
- seed points 173
- splitting 180
- statistical calculations 182
- texture analysis 12
- thresholding graylevels 197
- REGION_SPLIT function 180
- REGION_STATS function 182
- RGB color model 14
- rubber-sheet transformations 13

S

- salt-and-pepper noise 10, 156
- scaling 117
- segmentation
 - growing regions 175
 - intensity slicing 24
 - K-means clustering 88
 - locating seed points 173
 - merging homogeneous regions 177
 - overview 14
 - region counting 172
 - region statistics 182
 - splitting regions 180
 - thresholding 24, 193, 196
- shape analysis 76, 141, 142, 144, 146, 188, 198
- SHIFT_EDGE function 185
- shrinking an image 117
- skeletonization 189
- SKELETONIZE function 188
- SKEWNESS function 190
- SLANT function 191
- Slant transform 13, 191
- spatial filtering
 - convolution 95
 - creating a filter object 102
 - noise removal 42, 56
 - overview 9
- spatial moments 20, 141
- spectral filtering
 - bandpass filter 35
 - bandstop filter 35
 - Butterworth filter 35
 - design 37
 - lowpass filter 35

- notch filter 52
- starting the Navigator 4
- starting the Toolkit 3
- stationary input signals 123
- statistical measures 31, 65, 69, 74, 124, 137, 138, 139, 140, 165, 171, 172, 173, 175, 177, 180, 182, 190, 200
- stopping the Toolkit 6
- Surface Tool 4

T

- template matching 99
- texture
 - overview 12
 - spectral 166
 - statistical 63, 65, 67, 69, 74, 190, 200
- THRESH_ADAP function 193
- THRESHOLD function 196
- thresholding
 - adaptive 193
 - global binary or grayscale 196
 - intensity slicing 24
 - overview 7, 14
 - segmentation 24
- TOP_HAT function 198
- top-hat transform 198
- transforms
 - discrete cosine 13, 22
 - fast Fourier 13
 - forward radon 169
 - geometric 13
 - Haar 71
 - Hotelling 13, 164
 - Hough 78
 - inverse principle components 103
 - Karhunen-Loève 13, 164
 - medial axis 189
 - overview 13
 - polar FFT 166
 - power spectrum 120
 - principle components 13, 163
 - Slant 13, 191
 - top-hat 198
 - wavelet 126
- triangular window 129, 132

U

- UNIFORMITY function 200

V

- variance-covariance matrix 154
- VDA tools
 - Color Tool 4
 - Contour Tool 4
 - Histogram Tool 4
 - Image Tool 4
 - online help 5
 - overview 4
 - Plot Tool 4
 - Surface Tool 4
- visual data analysis tools 4

W

- wavelet filters
 - biorthogonal 114
 - Coifman 113
 - Daubechies 113
- wavelet transform 126
- Welch's method
 - modified periodogram 122
 - overlapping data segments 122
 - power spectrum estimate 121
- Wiener filter 58
- windows
 - Blackman 129, 130
 - Chebyshev 129, 130
 - Hamming 129, 131
 - Hanning 129, 131
 - Kaiser 129, 131
 - rectangular 129, 131
 - sequence 129
 - triangular 129, 132
- WzIPColorEdit 4
- WzIPContour 4
- WzIPHistogram 4
- WzIPImage 4
- WzIPPlot 4
- WzIPSurface 4

Y

- Yp mean filter 45

