

Wilhelm Burger · Mark J. Burge

Digital Image Processing

An algorithmic introduction using Java

With 271 figures and 17 tables

2007

Springer

Berlin Heidelberg New York

Hongkong London

Mailand Paris Tokio

Contents

Preface	V
1 Crunching Pixels	1
1.1 Programming with Images	2
1.2 Image Analysis and Computer Vision	3
2 Digital Images	5
2.1 Types of Digital Images	5
2.2 Image Acquisition	5
2.2.1 The Pinhole Camera Model	7
2.2.2 The “Thin” Lens	8
2.2.3 Going Digital	8
2.2.4 Image Size and Resolution	10
2.2.5 Image Coordinate System	11
2.2.6 Pixel Values	11
2.3 Image File Formats	13
2.3.1 Raster versus Vector Data	14
2.3.2 Tagged Image File Format (TIFF)	15
2.3.3 Graphics Interchange Format (GIF)	15
2.3.4 Portable Network Graphics (PNG)	16
2.3.5 JPEG	17
2.3.6 Windows Bitmap (BMP)	20
2.3.7 Portable Bitmap Format (PBM)	21
2.3.8 Additional File Formats	21
2.3.9 Bits and Bytes	22
2.4 Exercises	24
3 ImageJ	27
3.1 Image Manipulation and Processing	28
3.2 ImageJ Overview	28
3.2.1 Key Features	29
3.2.2 Interactive Tools	30
3.2.3 ImageJ Plugins	31
3.2.4 A First Example: Inverting an Image	32
3.3 Additional Information on ImageJ and Java	35
3.3.1 Resources for ImageJ	35
3.3.2 Programming with Java	35

3.4	Exercises	36
4	Histograms	37
4.1	What Is a Histogram?	38
4.2	Interpreting Histograms	39
4.2.1	Image Acquisition	40
4.2.2	Image Defects	42
4.3	Computing Histograms	44
4.4	Histograms of Images with More than 8 Bits	46
4.4.1	Binning	46
4.4.2	Example	46
4.4.3	Implementation	47
4.5	Color Image Histograms	47
4.5.1	Intensity Histograms	48
4.5.2	Individual Color Channel Histograms	48
4.5.3	Combined Color Histogram	49
4.6	Cumulative Histogram	50
4.7	Exercises	51
5	Point Operations	53
5.1	Modifying Image Intensity	54
5.1.1	Contrast and Brightness	54
5.1.2	Limiting the Results by Clamping	54
5.1.3	Inverting Images	55
5.1.4	Threshold Operation	55
5.2	Point Operations and Histograms	55
5.3	Automatic Contrast Adjustment	57
5.4	Modified Auto-Contrast	58
5.5	Histogram Equalization	59
5.6	Histogram Specification	62
5.6.1	Frequencies and Probabilities	63
5.6.2	Principle of Histogram Specification	65
5.6.3	Adjusting to a Piecewise Linear Distribution	65
5.6.4	Adjusting to a Given Histogram (Histogram Matching)	67
5.6.5	Examples	68
5.7	Gamma Correction	72
5.7.1	Why Gamma?	73
5.7.2	The Gamma Function	74
5.7.3	Real Gamma Values	74
5.7.4	Applications of Gamma Correction	75
5.7.5	Implementation	76
5.7.6	Modified Gamma Correction	76
5.8	Point Operations in ImageJ	80
5.8.1	Point Operations with Lookup Tables	80
5.8.2	Arithmetic Operations	81
5.8.3	Point Operations Involving Multiple Images	81

5.8.4	Methods for Point Operations on Two Images	82
5.8.5	ImageJ Plugins for Multiple Images	82
5.9	Exercises	83
6	Filters	87
6.1	What Is a Filter?	87
6.2	Linear Filters	89
6.2.1	The Filter Matrix	89
6.2.2	Applying the Filter	90
6.2.3	Computing the Filter Operation	91
6.2.4	Filter Plugin Examples	92
6.2.5	Integer Coefficients	93
6.2.6	Filters of Arbitrary Size	94
6.2.7	Types of Linear Filters	95
6.3	Formal Properties of Linear Filters	98
6.3.1	Linear Convolution	99
6.3.2	Properties of Linear Convolution	100
6.3.3	Separability of Linear Filters	101
6.3.4	Impulse Response of a Filter	103
6.4	Nonlinear Filters	104
6.4.1	Minimum and Maximum Filters	105
6.4.2	Median Filter	106
6.4.3	Weighted Median Filter	107
6.4.4	Other Nonlinear Filters	110
6.5	Implementing Filters	111
6.5.1	Efficiency of Filter Programs	111
6.5.2	Handling Image Borders	111
6.5.3	Debugging Filter Programs	112
6.6	Filter Operations in ImageJ	113
6.6.1	Linear Filters	113
6.6.2	Gaussian Filters	114
6.6.3	Nonlinear Filters	115
6.7	Exercises	115
7	Edges and Contours	117
7.1	What Makes an Edge?	117
7.2	Gradient-Based Edge Detection	118
7.2.1	Partial Derivatives and the Gradient	119
7.2.2	Derivative Filters	119
7.3	Edge Operators	120
7.3.1	Prewitt and Sobel Operators	120
7.3.2	Roberts Operator	123
7.3.3	Compass Operators	123
7.3.4	Edge Operators in ImageJ	125
7.4	Other Edge Operators	125
7.4.1	Edge Detection Based on Second Derivatives	126
7.4.2	Edges at Different Scales	126

7.4.3	Canny Operator	127
7.5	From Edges to Contours	127
7.5.1	Contour Following	127
7.5.2	Edge Maps	129
7.6	Edge Sharpening	130
7.6.1	Edge Sharpening with the Laplace Filter	130
7.6.2	Unsharp Masking	133
7.7	Exercises	137
8	Corner Detection	139
8.1	Points of Interest	139
8.2	Harris Corner Detector	140
8.2.1	Local Structure Matrix	140
8.2.2	Corner Response Function (CRF)	141
8.2.3	Determining Corner Points	142
8.2.4	Example	142
8.3	Implementation	142
8.3.1	Step 1: Computing the Corner Response Function	144
8.3.2	Step 2: Selecting “Good” Corner Points	148
8.3.3	Displaying the Corner Points	152
8.3.4	Summary	152
8.4	Exercises	153
9	Detecting Simple Curves	155
9.1	Salient Structures	155
9.2	Hough Transform	156
9.2.1	Parameter Space	157
9.2.2	Accumulator Array	159
9.2.3	A Better Line Representation	159
9.3	Implementing the Hough Transform	160
9.3.1	Filling the Accumulator Array	161
9.3.2	Analyzing the Accumulator Array	163
9.3.3	Hough Transform Extensions	165
9.4	Hough Transform for Circles and Ellipses	167
9.4.1	Circles and Arcs	167
9.4.2	Ellipses	170
9.5	Exercises	170
10	Morphological Filters	173
10.1	Shrink and Let Grow	174
10.1.1	Neighborhood of Pixels	175
10.2	Basic Morphological Operations	175
10.2.1	The Structuring Element	175
10.2.2	Point Sets	176
10.2.3	Dilation	177
10.2.4	Erosion	178
10.2.5	Properties of Dilation and Erosion	178

10.2.6	Designing Morphological Filters	180
10.2.7	Application Example: Outline	181
10.3	Composite Operations	183
10.3.1	Opening	185
10.3.2	Closing	185
10.3.3	Properties of Opening and Closing	186
10.4	Grayscale Morphology	187
10.4.1	Structuring Elements	187
10.4.2	Dilation and Erosion	187
10.4.3	Grayscale Opening and Closing	188
10.5	Implementing Morphological Filters	189
10.5.1	Binary Images in ImageJ	189
10.5.2	Dilation and Erosion	191
10.5.3	Opening and Closing	193
10.5.4	Outline	194
10.5.5	Morphological Operations in ImageJ	194
10.6	Exercises	196
11	Regions in Binary Images	199
11.1	Finding Image Regions	200
11.1.1	Region Labeling with Flood Filling	200
11.1.2	Sequential Region Labeling	204
11.1.3	Region Labeling—Summary	209
11.2	Region Contours	209
11.2.1	External and Internal Contours	209
11.2.2	Combining Region Labeling and Contour Finding	212
11.2.3	Implementation	213
11.2.4	Example	216
11.3	Representing Image Regions	216
11.3.1	Matrix Representation	216
11.3.2	Run Length Encoding	218
11.3.3	Chain Codes	219
11.4	Properties of Binary Regions	222
11.4.1	Shape Features	222
11.4.2	Geometric Features	223
11.4.3	Statistical Shape Properties	226
11.4.4	Moment-Based Geometrical Properties	228
11.4.5	Projections	233
11.4.6	Topological Properties	234
11.5	Exercises	235
12	Color Images	239
12.1	RGB Color Images	239
12.1.1	Organization of Color Images	241
12.1.2	Color Images in ImageJ	244
12.2	Color Spaces and Color Conversion	253
12.2.1	Conversion to Grayscale	256

12.2.2	Desaturating Color Images	257
12.2.3	HSV/HSB and HLS Color Space	258
12.2.4	TV Color Spaces—YUV, YIQ, and YCbCr	267
12.2.5	Color Spaces for Printing—CMY and CMYK	271
12.3	Colorimetric Color Spaces	275
12.3.1	CIE Color Spaces	276
12.3.2	CIE L*a*b*	281
12.3.3	sRGB	283
12.3.4	Adobe RGB	288
12.3.5	Chromatic Adaptation	288
12.3.6	Colorimetric Support in Java	292
12.4	Statistics of Color Images	299
12.4.1	How Many Colors Are in an Image?	299
12.4.2	Color Histograms	299
12.5	Color Quantization	301
12.5.1	Scalar Color Quantization	303
12.5.2	Vector Quantization	305
12.6	Exercises	311
13	Introduction to Spectral Techniques	313
13.1	The Fourier Transform	314
13.1.1	Sine and Cosine Functions	314
13.1.2	Fourier Series of Periodic Functions	317
13.1.3	Fourier Integral	318
13.1.4	Fourier Spectrum and Transformation	319
13.1.5	Fourier Transform Pairs	320
13.1.6	Important Properties of the Fourier Transform	321
13.2	Working with Discrete Signals	325
13.2.1	Sampling	325
13.2.2	Discrete and Periodic Functions	330
13.3	The Discrete Fourier Transform (DFT)	332
13.3.1	Definition of the DFT	332
13.3.2	Discrete Basis Functions	334
13.3.3	Aliasing Again!	334
13.3.4	Units in Signal and Frequency Space	338
13.3.5	Power Spectrum	339
13.4	Implementing the DFT	340
13.4.1	Direct Implementation	340
13.4.2	Fast Fourier Transform (FFT)	342
13.5	Exercises	342
14	The Discrete Fourier Transform in 2D	343
14.1	Definition of the 2D DFT	343
14.1.1	2D Basis Functions	344
14.1.2	Implementing the Two-Dimensional DFT	344
14.2	Visualizing the 2D Fourier Transform	345
14.2.1	Range of Spectral Values	348

14.2.2	Centered Representation	348
14.3	Frequencies and Orientation in 2D	348
14.3.1	Effective Frequency	349
14.3.2	Frequency Limits and Aliasing in 2D	350
14.3.3	Orientation	350
14.3.4	Correcting the Geometry of a 2D Spectrum	351
14.3.5	Effects of Periodicity	352
14.3.6	Windowing	352
14.3.7	Windowing Functions	354
14.4	2D Fourier Transform Examples	359
14.5	Applications of the DFT	359
14.5.1	Linear Filter Operations in Frequency Space	363
14.5.2	Linear Convolution versus Correlation	364
14.5.3	Inverse Filters	364
14.6	Exercises	366
15	The Discrete Cosine Transform (DCT)	367
15.1	One-Dimensional DCT	367
15.1.1	DCT Basis Functions	368
15.1.2	Implementing the One-Dimensional DCT	368
15.2	Two-Dimensional DCT	370
15.2.1	Separability	371
15.2.2	Examples	371
15.3	Other Spectral Transforms	371
15.4	Exercises	373
16	Geometric Operations	375
16.1	2D Mapping Function	376
16.1.1	Simple Mappings	377
16.1.2	Homogeneous Coordinates	377
16.1.3	Affine (Three-Point) Mapping	378
16.1.4	Projective (Four-Point) Mapping	380
16.1.5	Bilinear Mapping	385
16.1.6	Other Nonlinear Image Transformations	386
16.1.7	Local Image Transformations	389
16.2	Resampling the Image	390
16.2.1	Source-to-Target Mapping	390
16.2.2	Target-to-Source Mapping	391
16.3	Interpolation	392
16.3.1	Simple Interpolation Methods	392
16.3.2	Ideal Interpolation	393
16.3.3	Interpolation by Convolution	397
16.3.4	Cubic Interpolation	397
16.3.5	Spline Interpolation	399
16.3.6	Lanczos Interpolation	402
16.3.7	Interpolation in 2D	404
16.3.8	Aliasing	410

16.4	Java Implementation	413
16.4.1	Geometric Transformations	413
16.4.2	Pixel Interpolation	423
16.4.3	Sample Applications	426
16.5	Exercises	427
17	Comparing Images	429
17.1	Template Matching in Intensity Images	430
17.1.1	Distance between Image Patterns	431
17.1.2	Implementation	438
17.1.3	Matching under Rotation and Scaling	439
17.2	Matching Binary Images	441
17.2.1	Direct Comparison	441
17.2.2	The Distance Transform	442
17.2.3	Chamfer Matching	446
17.3	Exercises	447
A	Mathematical Notation	451
A.1	Symbols	451
A.2	Set Operators	453
A.3	Complex Numbers	453
A.4	Algorithmic Complexity and \mathcal{O} Notation	454
B	Java Notes	457
B.1	Arithmetic	457
B.1.1	Integer Division	457
B.1.2	Modulus Operator	459
B.1.3	Unsigned Bytes	459
B.1.4	Mathematical Functions (Class <code>Math</code>)	460
B.1.5	Rounding	461
B.1.6	Inverse Tangent Function	462
B.1.7	<code>Float</code> and <code>Double</code> (Classes)	462
B.2	Arrays and Collections	462
B.2.1	Creating Arrays	462
B.2.2	Array Size	463
B.2.3	Accessing Array Elements	463
B.2.4	Two-Dimensional Arrays	464
B.2.5	Cloning Arrays	465
B.2.6	Arrays of Objects, Sorting	466
B.2.7	Collections	467
C	ImageJ Short Reference	469
C.1	Installation and Setup	469
C.2	ImageJ API	471
C.2.1	Images and Processors	471
C.2.2	Images (Package <code>ij</code>)	471
C.2.3	Image Processors (Package <code>ij.process</code>)	472

C.2.4	Plugins (Packages <code>ij.plugin</code> , <code>ij.plugin.filter</code>)	473
C.2.5	GUI Classes (Package <code>ij.gui</code>)	474
C.2.6	Window Management (Package <code>ij</code>)	475
C.2.7	Utility Classes (Package <code>ij</code>)	475
C.2.8	Input-Output (Package <code>ij.io</code>)	475
C.3	Creating Images and Image Stacks	476
C.3.1	<code>ImagePlus</code> (Class)	476
C.3.2	<code>ImageStack</code> (Class)	476
C.3.3	<code>IJ</code> (Class)	477
C.3.4	<code>NewImage</code> (Class)	477
C.3.5	<code>ImageProcessor</code> (Class)	478
C.4	Creating Image Processors	478
C.4.1	<code>ImagePlus</code> (Class)	478
C.4.2	<code>ImageProcessor</code> (Class)	478
C.4.3	<code>ByteProcessor</code> (Class)	479
C.4.4	<code>ColorProcessor</code> (Class)	479
C.4.5	<code>FloatProcessor</code> (Class)	479
C.4.6	<code>ShortProcessor</code> (Class)	480
C.5	Loading and Storing Images	480
C.5.1	<code>IJ</code> (Class)	480
C.5.2	<code>Opener</code> (Class)	481
C.5.3	<code>FileSaver</code> (Class)	482
C.5.4	<code>FileOpener</code> (Class)	484
C.6	Image Parameters	485
C.6.1	<code>ImageProcessor</code> (Class)	485
C.6.2	<code>ColorProcessor</code> (Class)	485
C.7	Accessing Pixels	485
C.7.1	Accessing Pixels by 2D Image Coordinates	485
C.7.2	Accessing Pixels by 1D Indices	487
C.7.3	Accessing Multiple Pixels	488
C.7.4	Accessing All Pixels at Once	489
C.7.5	Specific Access Methods for Color Images	490
C.7.6	Direct Access to Pixel Arrays	490
C.8	Converting Images	492
C.8.1	<code>ImageProcessor</code> (Class)	492
C.8.2	<code>ImagePlus</code> , <code>ImageConverter</code> (Classes)	492
C.9	Histograms and Image Statistics	494
C.9.1	<code>ImageProcessor</code> (Class)	494
C.10	Point Operations	494
C.10.1	<code>ImageProcessor</code> (Class)	495
C.11	Filters	497
C.11.1	<code>ImageProcessor</code> (Class)	497
C.12	Geometric Operations	497
C.12.1	<code>ImageProcessor</code> (Class)	497
C.13	Graphic Operations	499
C.13.1	<code>ImageProcessor</code> (Class)	499
C.14	Displaying Images and Image Stacks	500

C.14.1	ImagePlus (Class)	500
C.14.2	ImageProcessor (Class)	502
C.15	Operations on Image Stacks	504
C.15.1	ImagePlus (Class)	504
C.15.2	ImageStack (Class)	505
C.15.3	Stack Example	506
C.16	Regions of Interest	506
C.16.1	ImagePlus (Class)	509
C.16.2	Roi, Line, OvalRoi, PointRoi, PolygonRoi (Classes)	510
C.16.3	ImageProcessor (Class)	511
C.16.4	ImageStack (Class)	512
C.16.5	IJ (Class)	512
C.17	Image Properties	513
C.17.1	ImagePlus (Class)	514
C.18	User Interaction	515
C.18.1	IJ (Class)	515
C.18.2	GenericDialog (Class)	517
C.19	Plugins	518
C.19.1	PlugIn (Interface)	518
C.19.2	PlugInFilter (Interface)	519
C.19.3	Executing Plugins: IJ (Class)	520
C.20	Window Management	521
C.20.1	WindowManager (Class)	521
C.21	Additional Functions	522
C.21.1	ImagePlus (Class)	522
C.21.2	IJ (Class)	523
D	Source Code	525
D.1	Harris Corner Detector	525
D.1.1	Harris_Corner_Plugin (Class)	525
D.1.2	File Corner (Class)	527
D.1.3	File HarrisCornerDetector (Class)	527
D.2	Combined Region Labeling and Contour Tracing	532
D.2.1	Contour_Tracing_Plugin (Class)	532
D.2.2	Contour (Class)	533
D.2.3	BinaryRegion (Class)	535
D.2.4	ContourTracer (Class)	536
D.2.5	ContourOverlay (Class)	541
	References	543
	Index	549

Appendix C

ImageJ Short Reference

C.1 Installation and Setup

The most up-to-date information about downloading and installation is found on the ImageJ Website

<http://rsb.info.nih.gov/ij/>.

Currently this site contains complete installation packages for Linux (x86), Macintosh (OS 9, OS X), and Windows. The following information mainly refers to the Windows installation but is quite similar for the other platforms.

ImageJ can be installed in any file directory (which we refer to as <ij>) and can be used without installing any additional software (including the Java runtime). Figure C.1 (a) shows the contents of the installation directory (under Windows) with the following main contents:

<ij>/jre

A complete Java runtime environment, the “Java Virtual Machine” (JVM). This is required for actually executing Java programs.

<ij>/macros

Directory containing ImageJ *macros*, short programs written in ImageJ’s macro language (not covered here).

<ij>/plugins

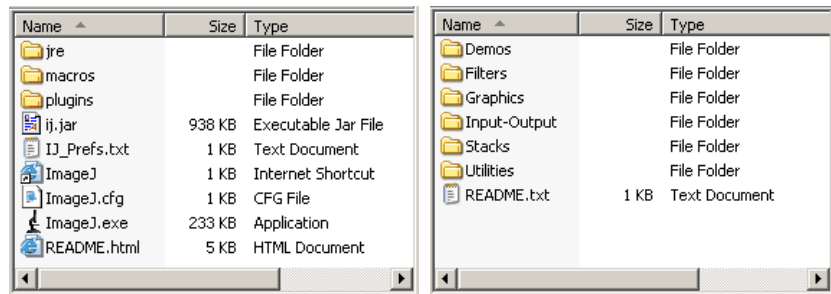
This directory contains all ImageJ plugins written by the user. It comes with some simple example plugins stored in subdirectories (Fig. C.1 (b)). Notice that user-defined plugins may not be located deeper than one level below the **plugins** directory. Otherwise the plugins are not recognized by ImageJ.

<ij>/ij.jar

A Java archive file that contains the entire core functionality of ImageJ. Only this file needs to be replaced when ImageJ is updated to

Fig. C.1

ImageJ installation under Windows. Contents of the installation directory <ij> (a) and its subdirectory <ij>/plugins (b).



(a)

(b)

a newer version. JAR files are ZIP-compressed archives containing collections of binary Java (.class) files.

<ij>/IJ_Prefs.txt

A text file used to define various settings and user options for ImageJ.

<ij>/ImageJ.cfg

Specifies the path to launch Java and startup parameters for the Java runtime. Under Windows, this is typically by the lines

```
.
jre\bin\javaw.exe
-Xmx340m -cp ij.jar ij.ImageJ
```

The option `-Xmx340m` in this case specifies that 340 MB of storage are allocated for the Java process. This may be too small for some applications and can be increased (up to about 1.7 GB on a 32-bit system) by editing this file or through the **Edit**→**Options**→**Memory** menu in ImageJ.

<ij>/ImageJ.exe

A small launch program that invokes Java and ImageJ and can be used like any native Windows program.

For writing new plugin programs, we also need a text *editor* for editing the Java source files and a Java *compiler*. The Java runtime environment (JRE) included with ImageJ contains both, even a compiler,¹ such that no additional software is required to get started. However, this basic programming environment is insufficient in practice even for small projects. Instead, it is recommended to embark on one of the freely available integrated Java programming environments, such as *Eclipse*,² *NetBeans*,³ or *Borland JBuilder*.⁴ These products also give superior sup-

¹ Unfortunately, the built-in compiler (contained in `jre/lib/ext/tools.jar`) does *not* support the language features introduced with Java 1.5 or higher and is thus incompatible with many examples in this book.

² www.eclipse.org.

³ www.netbeans.org.

⁴ www.borland.com/jbuilder.

port for managing larger plugin projects and provide context-dependent editing capabilities and advanced syntax analysis, which help to avoid many programming errors that may otherwise cause fatal execution errors.

C.2 ImageJ API

The complete documentation and source code for the ImageJ API⁵ is available online at

<http://rsb.info.nih.gov/ij/developer/>.

Both are extremely helpful resources for developing new ImageJ plugins, as is the ImageJ programming tutorial written by Werner Bailer [4]. In addition, the standard Java API documentation (available online at Sun Microsystems⁶) should always be at hand for any serious Java programming. In the following, we give a brief description of the most important packages and classes in the ImageJ API.⁷

C.2.1 Images and Processors

While the ImageJ API makes it easy to work with images on the programming level, their internal representation is fairly complex and incorporates several objects of different classes. Some of these classes are unique to ImageJ, while others are standard Java (AWT) classes or derived from standard classes. Figure C.2 contains a simplified diagram that shows the relationships between the key image objects.

The actual image data (pixels) are stored in either an `ImageProcessor` or `ImageStack` object, depending on whether it is a single image or a sequence (stack) of images, respectively. `ImageProcessor` or `ImageStack` objects can be used to process images but have no screen representation. Visible images are based on an `ImagePlus` object, which links to an AWT `Image` and `ImageWindow` (a subclass of `java.awt.Frame`) to map the image's pixel data onto the screen.

C.2.2 Images (Package `ij`)

`ImagePlus` (class)

This is an extended variant of the standard Java class `java.awt.Image` for representing images (Fig. C.3). An `ImagePlus` object represents an image (or image sequence) that can be displayed on the screen. It contains an instance of the class `ImageProcessor` (see below) that is not visible but provides the functionality for processing the corresponding image.

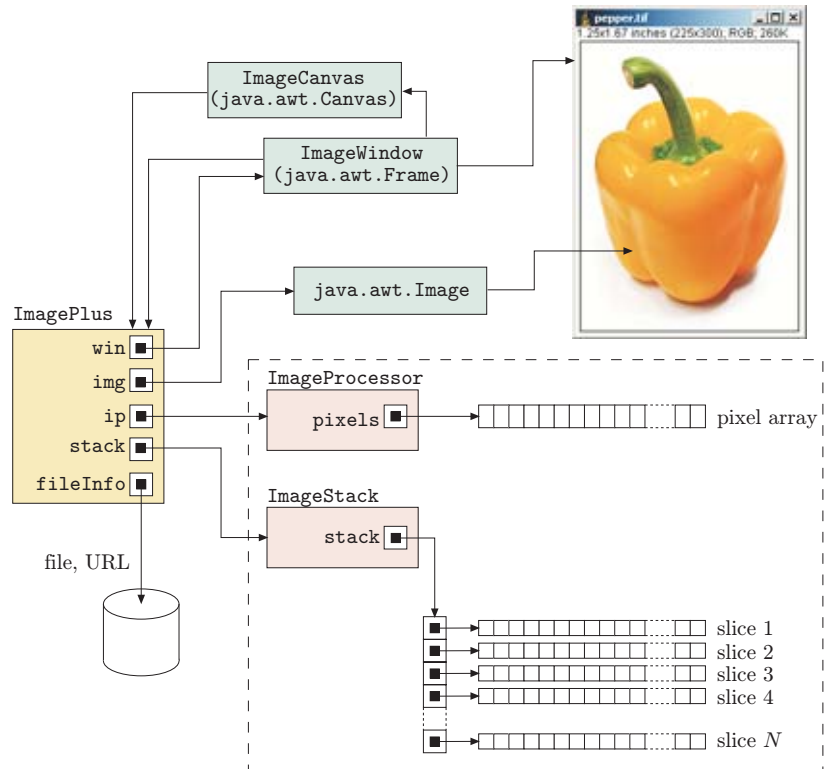
⁵ Application programming interface.

⁶ <http://java.sun.com/reference/api/>.

⁷ The UML diagrams in Figs. C.3–C.6 are taken from the ImageJ Website.

Fig. C.2

Internal representation of images and image stacks in ImageJ (simplified). **ImageProcessor** and **ImageStack** objects contain the actual pixel data of images and image sequences (stacks), respectively. A single image is stored in memory as a one-dimensional array of numerical pixel values. Image stacks are stored as a one-dimensional array of pixel arrays. **ImageProcessor** and **ImageStack** objects can be used to process and convert images but are not necessarily visible on screen. Opening, storing, and displaying an image or image stack requires an **ImagePlus** object, which uses standard AWT mechanisms for mapping to the screen (classes **Image**, **ImageWindow**, **ImageCanvas**).



ImageStack (class)

An extensible sequence (“stack”) of images that is usually attached to an **ImagePlus** object (see Fig. C.2).

C.2.3 Image Processors (Package `ij.process`)

ImageProcessor (class)

This is the (abstract) superclass for the four image processor classes available in ImageJ: **ByteProcessor**, **ShortProcessor**, **FloatProcessor**, **ColorProcessor** (Fig. C.4). Processing images is mainly accomplished with objects of class **ImageProcessor** or one of its subclasses, while **ImagePlus** objects (see above) are mostly used for displaying and interacting with images.

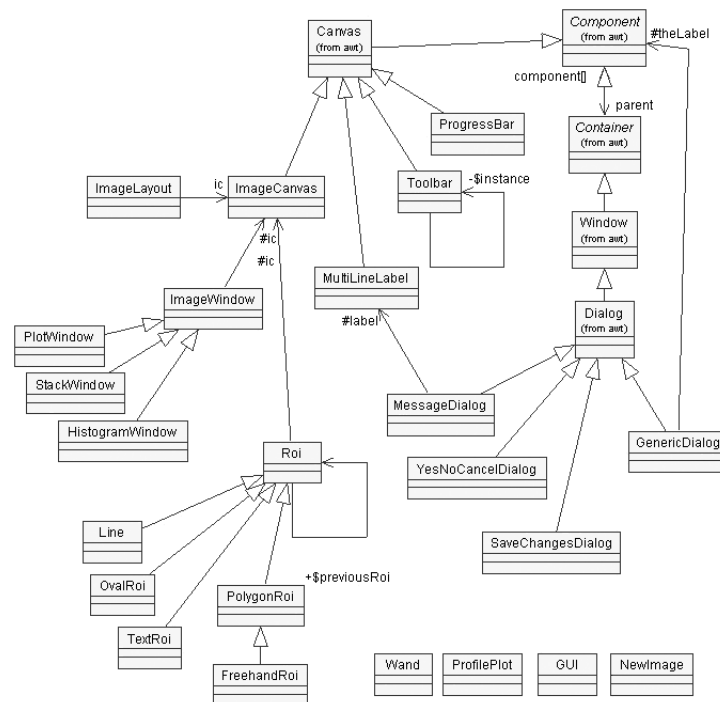
ByteProcessor (class)

Image processor for 8-bit (byte) grayscale and indexed color images. The derived subclass **BinaryProcessor** implements binary images that may only contain pixel values 0 and 255.

ShortProcessor (class)

Image processor for 16-bit grayscale images.

Fig. C.5
Class diagram for the ImageJ package `ij.gui`.



C.2.5 GUI Classes (Package `ij.gui`)

ImageJ's GUI⁸ classes provide the basic functionality for displaying and interacting with images (Fig. C.5):

ColorChooser (class)

Displays a dialog window for interactive color selection.

NewImage (class)

Provides the functionality for creating new images interactively and through static methods (see Sec. C.3.4).

GenericDialog (class)

Provides configurable dialog windows with a set of standard interaction fields.

ImageCanvas (class)

This subclass of the standard Java class `java.awt.Canvas` describes the mapping (source rectangle, zoom factor) for displaying the image in a window. It also handles the mouse and keyboard events sent to that window.

ImageWindow (class)

This subclass of the standard Java class `java.awt.Frame` represents a screen window for displaying images of type `ImagePlus`. An object of class `ImageWindow` contains an instance of class

⁸ Graphical user interface.

ImageCanvas (see above) for the actual presentation of the image.

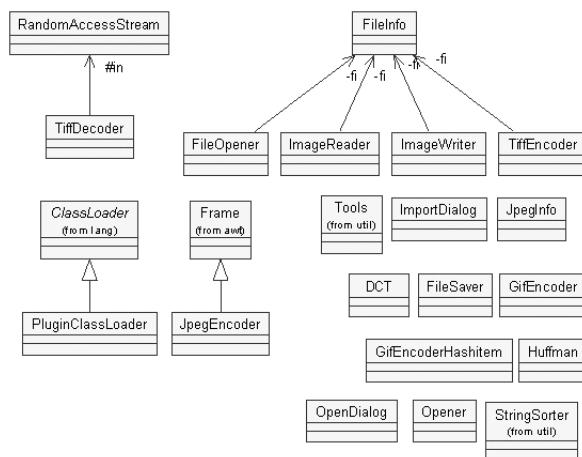
Roi (class)

C.2.6 Window Management (Package ij)

Provides a set of static methods to manage ImageJ's screen windows (Fig. C.3).

IJ (class)

C.2.8 Input-Output (Package `ij.io`)



Class diagram for ImageJ package
ij.io.

C.3 Creating Images and Image Stacks

In ImageJ, images, image stacks, and image processors can be created in a variety of different ways, either from existing images or from scratch.

C.3.1 ImagePlus (Class)

The class `ImagePlus` implements the following constructor methods for creating images:

`ImagePlus ()`

Constructor method: creates a new `ImagePlus` object without initialization.

`ImagePlus (String pathOrURL)`

Constructor method: opens the image *file* (TIFF, BMP, DICOM, FITS, PGM, GIF, or JPEG format) or *URL* (TIFF, DICOM, GIF, or JPEG format) specified by the location *pathOrURL* in a new `ImagePlus` object.

`ImagePlus (String title, Image img)`

Constructor method: creates a new `ImagePlus` object with the name *title* from a given image *img* of the standard Java type `java.awt.Image`.

`ImagePlus (String title, ImageProcessor ip)`

Constructor method: creates a new `ImagePlus` image with the name *title* from a given `ImageProcessor` object *ip*.

`ImagePlus (String title, ImageStack stack)`

Constructor method: creates a new `ImagePlus` object with the name *title* from a given image stack.

Other methods:

`ImageStack createEmptyStack ()`

Creates a new, *empty* stack with the same width, height, and color table as the given `ImagePlus` object to which this method is applied.

`ImageStack getStack ()`

Returns the image stack associated with the `ImagePlus` object to which this method is applied. If no stack exists, a new single-slice stack is created with the contents of that image (by calling `createEmptyStack()`).

C.3.2 ImageStack (Class)

The class `ImageStack` (in package `ij`) provides the following constructor methods for creating image stacks (usually contained inside an `ImagePlus` object):

`ImageStack(int width, int height)`

Constructor method: creates a new, empty image stack of size $width \times height$.

`ImageStack(int width, int height, ColorModel cm)`

Constructor method: creates a new, empty image stack of size $width \times height$ with the color model *cm* (of type `java.awt.image.ColorModel`).

C.3.3 IJ (Class)

`static ImagePlus createImage(String title, String type,
int width, int height, int slices)`

Creates a new `ImagePlus` object. *type* should contain the string "8", "16", "32", or "RGB" for creating 8-bit grayscale, 16-bit grayscale, float, or RGB images, respectively. In addition, *type* can be used to specify a fill option by attaching the string "white", "black", or "ramp" (the default is "white"). For example, the *type* string "16ramp" would specify a 16-bit grayscale image initially filled with a black-to-white ramp. *width* and *height* specify the size of the image, and *slices* specifies the number of stack slices (use 1 for a single image). The new image is returned but not automatically displayed (use `show()`).

`static void newImage(String title, String type,
int width, int height)`

Creates a new image and displays it. The meaning of the parameters is the same as above. No reference to the new image is returned (`IJ.getImage()` may be used to obtain the active image).

C.3.4 NewImage (Class)

The class `NewImage` (in package `ij.gui`) implements several static methods for creating single images of type `ImagePlus` and image stacks:

`static ImagePlus createByteImage(String title,
int width, int height, int slices, int fill)`

Creates a single 8-bit grayscale image or stack (if *slices* > 1) of size $width \times height$ with the name *title*. Admissible values for the *fill* argument are the constants `NewImage.FILL_BLACK`, `NewImage.FILL_WHITE`, and `NewImage.FILL_RAMP`.

`static ImagePlus createShortImage(String title,
int width, int height, int slices, int fill)`

Creates a single 16-bit grayscale image or stack.

`static ImagePlus createFloatImage(String title,
int width, int height, int slices, int fill)`

Creates a single 32-bit float image or stack.

```
static ImagePlus createRGBImage (String title,  
                                int width, int height, int slices, int fill)  
    Creates a single 32-bit RGB image or stack.  
static ImagePlus createImage (String title, int width,  
                              int height, int slices, int bitDepth, int fill)  
    Generic method that creates and returns an 8-bit grayscale, 16-  
    bit grayscale, float, or RGB image depending upon the value of  
    bitDepth, which can be 8, 16, 32, or 24, respectively. The other  
    parameters have the same meanings as above.
```

C.3.5 ImageProcessor (Class)

```
java.awt.Image createImage ()  
    Creates a copy of the ImageProcessor object and returns it as a  
    standard Java AWT image.
```

C.4 Creating Image Processors

In ImageJ, `ImageProcessor` objects represent images that can be created, processed, and destroyed but are not generally visible on the screen (see Sec. 3.14 on how to display images).

C.4.1 ImagePlus (Class)

```
ImageProcessor getProcessor ()  
    Returns a reference to the image's ImageProcessor object. If  
    there is no ImageProcessor, a new one is created. Returns null  
    if this image contains no ImageProcessor and no AWT image.  
void setProcessor (String title, ImageProcessor ip)  
    Replaces the image's current ImageProcessor, if any, by ip. If  
    title is null, the image title remains unchanged.
```

C.4.2 ImageProcessor (Class)

```
ImageProcessor createProcessor (int width, int height)  
    Returns a new, blank ImageProcessor object of the specified size  
    and the same type as the processor to which this method is ap-  
    plied. This is an abstract method that is implemented by every  
    subclass of ImageProcessor.  
ImageProcessor duplicate ()  
    Returns a copy of the image processor to which this method is  
    applied. This is an abstract method that is implemented by every  
    subclass of ImageProcessor.
```

C.4.3 ByteProcessor (Class)

`ByteProcessor (Image img)`

Constructor method: creates a new `ByteProcessor` object from an 8-bit image *img* of type `java.awt.Image`.

`ByteProcessor (int width, int height)`

Constructor method: creates a blank `ByteProcessor` object of size *width* × *height*.

`ByteProcessor (int width, int height, byte[] pixels,
ColorModel cm)`

Constructor method: creates a new `ByteProcessor` object of the specified size and the color model *cm* (of type `java.awt.image.ColorModel`), with the pixel values taken from the one-dimensional byte array *pixels*.

C.4.4 ColorProcessor (Class)

`ColorProcessor (Image img)`

Constructor method: creates a new `ColorProcessor` object from the RGB image *img* of type `java.awt.Image`.

`ColorProcessor (int width, int height)`

Constructor method: creates a blank `ColorProcessor` object of size *width* × *height*.

`ColorProcessor (int width, int height, int[] pixels)`

Constructor method: creates a new `ColorProcessor` object of the specified size with the pixel values taken from the one-dimensional `int` array *pixels*.

C.4.5 FloatProcessor (Class)

`FloatProcessor float[] [] pixels`

Constructor method: creates a new `FloatProcessor` object from the two-dimensional `float` array *pixels*, which is assumed to store the image data as *pixels*[*u*][*v*] (i.e., in column-first order).

`FloatProcessor int[] [] pixels`

Constructor method: creates a new `FloatProcessor` object from the two-dimensional `int` array *pixels*; otherwise the same as above.

`FloatProcessor (int width, int height)`

Constructor method: creates a blank `FloatProcessor` object of size *width* × *height*.

`FloatProcessor (int width, int height, double[] pixels)`

Constructor method: creates a new `FloatProcessor` object of the specified size with the pixel values taken from the one-dimensional double array *pixels*. The resulting image uses the default grayscale color model.

`FloatProcessor (int width, int height, int[] pixels)`

Same as above with *pixels* being an int array.

`FloatProcessor (int width, int height, float[] pixels,
ColorModel cm)`

Constructor method: creates a new `FloatProcessor` object of the specified size with the pixel values taken from the one-dimensional float array *pixels*. The resulting image uses the color model *cm* (of type `java.awt.image.ColorModel`), or the default grayscale model if *cm* is null.

C.4.6 ShortProcessor (Class)

`ShortProcessor (int width, int height)`

Constructor method: creates a new `ShortProcessor` object of the specified size. The resulting image uses the default grayscale color model, which maps zero to black.

`ShortProcessor (int width, int height, short[] pixels,
ColorModel cm)`

Constructor method: creates a new `ShortProcessor` object of the specified size with the pixel values taken from the one-dimensional short array *pixels*. The resulting image uses the color model *cm* (of type `java.awt.image.ColorModel`), or the default grayscale model if *cm* is null.

C.5 Loading and Storing Images

C.5.1 IJ (Class)

The class `IJ` provides the static method `void run()` for executing commands that apply to the currently active image. I/O commands include:

`IJ.run("Open...")`

Displays a file open dialog and then opens the image file selected by the user. Displays an error message if the selected file is not in one of the supported formats or if it is not found. The opened image becomes the active image.

`IJ.run("Revert")`

Reverts the active image to the original file version.

`IJ.run("Save")`

Saves the currently active image.

Class `IJ` also defines the following static methods for image I/O:

`static void open ()`

Displays a file open dialog and then opens the image file (TIFF, DICOM, FITS, PGM, JPEG, BMP, GIF, LUT, ROI, or text format) selected by the user. Displays an error message if the selected file is not in one of the supported formats or if it is not found. No

reference to the opened image is returned (use `IJ.getImage()` to obtain the active image).

static void open (String *path*)

Opens and displays an image file specified by *path*; otherwise the same as `open()` above. Displays an error message if the specified file is not in one of the supported formats or if it is not found.

static ImagePlus openImage (String *path*)

Tries to open the image file specified by *path* and returns a `ClassImagePlus` object (which is not automatically displayed) if successful. Otherwise `null` is returned and no error is raised.

static void save (String *path*)

Saves the currently active image, lookup table, selection, or text window to the specified file *path*, whose extension encodes the file type. *path* must therefore end in ".tif", ".jpg", ".gif", ".zip", ".raw", ".avi", ".bmp", ".lut", ".roi", or ".txt".

static void saveAs (String *format*, String *path*)

Saves the currently active image, lookup table, selection (region of interest), measurement results, XY coordinates, or text window to the specified file *path*. The *format* argument must be "tif", "jpeg", "gif", "zip", "raw", "avi", "bmp", "text image", "lut", "selection", "measurements", "xy", or "text"

C.5.2 Opener (Class)

Opener is used to open TIFF (and TIFF stacks), DICOM, FITS, PGM, JPEG, BMP, or GIF images, and lookup tables, using a file open dialog or a path.

Opener ()

Constructor method: creates a new **Opener** object.

void open ()

Displays a file open dialog box and then opens the file selected by the user. Displays an error message if the selected file is not in one of the supported formats. No reference to the opened image is returned (use `IJ.getImage()` to obtain the active image).

void open (String *path*)

Opens and displays a TIFF, DICOM, FITS, PGM, JPEG, BMP, GIF, LUT, ROI, or text file. Displays an error message if the file specified by *path* is not in one of the supported formats. No reference to the opened image is returned (use `IJ.getImage()` to obtain the active image).

ImagePlus openImage (String *path*)

Attempts to open the specified file as a TIF, BMP, DICOM, FITS, PGM, GIF or JPEG image. Returns a new **ImagePlus** object if successful, otherwise `null`. Activates the plugin `HandleExtraFileTypes` if the file type is not recognized.

ImagePlus openImage (String *directory*, String *name*)
Same as above, with *path* split into *directory* and *name*.

void openMultiple ()
Displays a standard file chooser and then opens the files selected by the user. Displays error messages if one or more of the selected files is not in one of the supported formats. No reference to the opened images is returned (methods in class **WindowManager** can be used to access these images; see Sec. C.20.1).

ImagePlus openTiff (String *directory*, String *name*)
Attempts to open the specified file as a TIFF image or image stack. Returns an **ImagePlus** object if successful, **null** otherwise.

ImagePlus openURL (String *url*)
Attempts to open the specified URL as a TIFF, ZIP-compressed TIFF, DICOM, GIF, or JPEG image. Returns an **ImagePlus** object if successful, **null** otherwise.

void setSilentMode (boolean *mode*)
Turns silent mode on or off. The “Opening: path” status message is not displayed in silent mode.

C.5.3 FileSaver (Class)

Saves images in TIFF, GIF, JPEG, RAW, ZIP, and text formats.

FileSaver (ImagePlus *im*)
Constructor method: creates a new **FileSaver** for a given **ImagePlus** object.

boolean save ()
Tries to save the image associated with this **FileSaver** as a TIFF file. Returns **true** if successful or **false** if the user cancels the file save dialog.

boolean saveAsBmp ()
Saves the image associated with this **FileSaver** in BMP format using a save file dialog.

boolean saveAsBmp (String *path*)
Saves the image associated with this **FileSaver** in BMP format at the specified path.

boolean saveAsGif ()
Saves the image associated with this **FileSaver** in GIF format using a save file dialog.

boolean saveAsGif (String *path*)
Saves the image associated with this **FileSaver** in GIF format at the specified path.

boolean saveAsJpeg ()
Saves the image associated with this **FileSaver** in JPEG format using a save file dialog.

boolean saveAsJpeg (String *path*)
Saves the image associated with this **FileSaver** in JPEG format at the specified path.

boolean saveAsLut ()
Saves the lookup table (LUT) of the image associated with this **FileSaver** using a save file dialog.

boolean saveAsLut (String *path*)
Saves the lookup table (LUT) of the image associated with this **FileSaver** at the specified path.

boolean saveAsPng ()
Saves the image associated with this **FileSaver** in PNG format using a save file dialog.

boolean saveAsPng (String *path*)
Saves the image associated with this **FileSaver** in PNG format at the specified path.

boolean saveAsRaw ()
Saves the image associated with this **FileSaver** in raw format using a save file dialog.

boolean saveAsRaw (String *path*)
Saves the image associated with this **FileSaver** in raw format at the specified path.

boolean saveAsRawStack (String *path*)
Saves the stack associated with this **FileSaver** in raw format at the specified path.

boolean saveAsRaw ()
Saves the image associated with this **FileSaver** in raw format using a save file dialog.

boolean saveAsRaw (String *path*)
Saves the image associated with this **FileSaver** in raw format at the specified path.

boolean saveAsText ()
Saves the image associated with this **FileSaver** as tab-delimited text using a save file dialog.

boolean saveAsText (String *path*)
Saves the image associated with this **FileSaver** as tab-delimited text at the specified path.

boolean saveAsTiff ()
Saves the image associated with this **FileSaver** in TIFF format using a save file dialog.

boolean saveAsTiff (String *path*)
Saves the image associated with this **FileSaver** in TIFF format at the specified path.

`boolean saveAsTiffStack (String path)`

Saves the stack associated with this `FileSaver` as a multiimage TIFF at the specified path.

`boolean saveAsZip ()`

Saves the image associated with this `FileSaver` as a TIFF in a ZIP archive using a save file dialog.

`boolean saveAsZip (String path)`

Saves the image associated with this `FileSaver` as a TIFF in a ZIP archive at the specified path.

C.5.4 FileOpener (Class)

`FileOpener (FileInfo fi)`

Constructor method: creates a new `FileOpener` from a given `FileInfo` object *fi*. Use `im.getFileInfo()` or `im.getOriginalFileInfo()` to retrieve the `FileInfo` from a given `ImagePlus` object *im*.

`void open ()`

Opens the image from the location specified by this `FileOpener` and displays it. No reference to the opened image is returned (use `IJ.getImage()` to obtain the active image).

`ImagePlus open (boolean show)`

Opens the image from the location specified by this `FileOpener`. The image is displayed if *show* is `true`. Returns an `ImagePlus` object if successful, otherwise `null`.

`void revertToSaved (ImagePlus im)`

Restores *im* to its original disk or network version.

Here is a simple example that uses the classes `Opener`, `FileInfo`, and `FileOpener` for opening and subsequently reverting an image to its original:

```
Opener op = new Opener();
op.open();
ImagePlus im = IJ.getImage();
ImageProcessor ip = im.getProcessor();
ip.invert();
im.updateAndDraw();
// .... more modifications
// revert to original:
FileInfo fi = im.getOriginalFileInfo();
FileOpener fo = new FileOpener(fi);
fo.revertToSaved(im);
```

C.6 Image Parameters

C.6.1 ImageProcessor (Class)

int getHeight ()
Returns this image processor's height (number of lines).

int getWidth ()
Returns this image processor's width (number of columns).

boolean getInterpolate ()
Returns **true** if bilinear interpolation is turned on for this processor.

void setInterpolate (boolean *interpolate*)
Turns pixel interpolation for this processor on or off. If turned *on*, the processor uses bilinear interpolation for **getLine()** and geometric operations such as **scale()**, **resize()**, and **rotate()**.

C.6.2 ColorProcessor (Class)

static double[] getWeightingFactors ()
Returns the weights used for the red, green, and blue component (as a 3-element **double**-array) for converting RGB colors to grayscale or intensity (see Sec. 12.2.1). These weights are used, for example, by the methods **getPixelValue()**, **getHistogram()**, and **convertToByte()** to perform color conversions. The weights can be set with the static method **setWeightingFactors()** described below.

static void setWeightingFactors
 (**double** *w_r*, **double** *w_g*, **double** *w_b*)
Sets the weights used for the red, green, and blue components for color-to-gray conversion (see Sec. 12.2.1). The default weights in ImageJ are $w_R = w_G = w_B = \frac{1}{3}$. Alternatively, if the "Weighted RGB Conversions" option is selected in the **Edit**→**Options**→**Conversions** dialog, the standard ITU-BT.709 [55] weights ($w_R = 0.299$, $w_G = 0.587$, $w_B = 0.114$) are used.

C.7 Accessing Pixels

The ImageJ class **ImageProcessor** provides a large variety of methods for accessing image pixels. All methods described in this section are defined for objects of class **ImageProcessor**.

C.7.1 Accessing Pixels by 2D Image Coordinates

Methods performing coordinate checking

The following methods are tolerant against passing out-of-bounds coordinate values. Reading pixel values from positions outside the image

canvas usually returns a zero value, while writing to such positions has no effect.

`int getPixel (int u, int v)`

Returns the pixel value at the image coordinate (*u*, *v*). Zero is returned for all positions outside the image boundaries (no error). Applied to a `ByteProcessor` or `ShortProcessor`, the returned `int` value is identical to the numerical pixel value. For images of type `ColorProcessor`, the α RGB bytes are arranged inside the `int` value in the standard way (see Fig. 12.6). For a `FloatProcessor`, the returned 32-bit `int` value contains the *bit-pattern* of the corresponding `float` pixel value, and *not* a converted numerical value! A bit pattern *p* may be converted to a numeric `float`-value using the method `Float.intBitsToFloat(p)` (in package `java.lang.Number`).

`void putPixel (int u, int v, int value)`

Sets the pixel at image coordinate (*u*, *v*) to *value*. Coordinates outside the image boundaries are ignored (no error). For images of types `ByteProcessor` and `ShortProcessor`, *value* is clamped to the admissible range. For a `ColorProcessor`, the 8-bit α RGB values are packed inside *value* in the standard arrangement. For a `FloatProcessor`, *value* is assumed to contain the 32-bit pattern of a `float` value, which can be obtained using the `Float.floatToIntBits()` method.

`int[] getPixel (int u, int v, int[] iArray)`

Returns the pixel value at the image coordinate (*u*, *v*) as an `int` array containing *one* element or, for a `ColorProcessor`, *three* elements (RGB component values with *iArray*[0] = *R*, *iArray*[1] = *G*, *iArray*[2] = *B*). If the argument passed to *iArray* is a suitable array (i.e., of proper size and not `null`), that array is filled with the pixel value(s) and returned; otherwise a new array is returned.

`void putPixel (int u, int v, int[] iArray)`

Sets the pixel at position (*u*, *v*) to the value specified by the contents of *iArray*, which contains either *one* element or, for a `ColorProcessor`, *three* elements (RGB component values, with *iArray*[0] = *R*, *iArray*[1] = *G*, *iArray*[2] = *B*).

`float getPixelValue (int u, int v)`

Returns the pixel value at the image coordinate (*u*, *v*) as a `float` value. For images of types `ByteProcessor` and `ShortProcessor`, a calibrated value is returned that is determined by the processor's (optional) calibration table. Invoked on a `FloatProcessor`, the method returns the actual (numeric) pixel value. In the case of a `ColorProcessor`, the gray value of the corresponding RGB pixel is returned (computed as a weighted sum of the RGB compo-

nents). The RGB component weights can be set using the method `setWeightingFactors()` (see p. 485).

```
void putPixelValue (int u, int v, double value)
```

Sets the pixel at position (u, v) to *value* (after clamping to the appropriate range and rounding). On a `ColorProcessor`, *value* is clamped to $[0 \dots 255]$ and assigned to all three color components, thus creating a gray color with the luminance equivalent to *value*.

Methods without coordinate checking

The following methods are faster at the cost of not checking the validity of the supplied coordinates, i. e., passing out-of-bounds coordinate values will result in a runtime exception.

```
int get (int u, v)
```

This is a faster version of `getPixel()` that does not do bounds checking on the coordinates.

```
void set (int u, int v, int value)
```

This is a faster version of `putPixel()` that does not clamp out-of-range values and does not do bounds checking on the coordinates.

```
float getf (int u, v)
```

Returns the pixel value as a `float`; otherwise the same as `get()`.

```
void setf (int u, int v, float value)
```

Sets the pixel at (u, v) to the `float` value *value*; otherwise the same as `set()`.

C.7.2 Accessing Pixels by 1D Indices

These methods are useful for processing images if the individual pixel coordinates are not relevant, e. g., for performing point operations on all image pixels.

```
int get (int i)
```

Returns the content of the `ImageProcessor`'s pixel array at position *i* as an `int` value, with $0 \leq i < w \cdot h$ (*w*, *h* are the width and height of the image, respectively). The method `getPixelCount()` (see below) retrieves the size of the pixel array.

```
void set (int i, int value)
```

Inserts *value* at the image's pixel array at position *i*.

```
float getf (int i)
```

Returns the content of the image processor's pixel array at position *i* as a `float` value.

```
void setf (int i, float value)
```

Inserts *value* at the image's pixel array at position *i*.

```
int getPixelCount ()
```

Returns the number of pixels in this image, i. e., the length of the pixel array.

Pixel access is faster with the above methods because the supplied index *i* directly addresses the one-dimensional pixel array (see p. 490 for a directly accessing the pixel array without using method calls, which is still faster). Note that these methods are efficient at the cost of not checking the validity of their arguments, i.e., passing an illegal index will result in a runtime exception. The typical use of these methods is demonstrated by the following example:

```
...
int M = ip.getPixelCount();
for (int i = 0; i < M; i++) {
    int a = ip.get(i);
    int b = ... ; // compute the new pixel value
    ip.set(i, b);
}
...
```

Note that we explicitly define the range variable *M* instead of writing

```
for (int i = 0; i < ip.getPixelCount(); i++) ...
```

directly, because in this case the method `getPixelCount()` would be (unnecessarily) invoked in every single iteration of the `for`-loop.

C.7.3 Accessing Multiple Pixels

Object `getPixels()`

Returns a reference (not a copy!) to the processor's one-dimensional pixel array, and thus any changes to the returned pixel array immediately affect the contents of the corresponding image. The array's element type depends on the type of processor:

```
ByteProcessor → byte[]
ShortProcessor → short[]
FloatProcessor → float[]
ColorProcessor → int[]
```

Since the type (`Object`) of the returned object is generic, type casting is required to actually use the returned array; e.g.,

```
ByteProcessor ip = new ByteProcessor(200, 300);
byte[] pixels = (byte[]) ip.getPixels();
```

Note that this typecast is potentially dangerous. To avoid a runtime exception one should assure that processor and array types match; e.g., using Java's `instanceof` operator:

```
if (ip instanceof ByteProcessor) ... or
if (ip.getPixels() instanceof byte[]) ...
```

void `setPixels(Object pixels)`

Replaces the processor's pixel array by *pixels*. The type and size of this one-dimensional array must match the specifications of the target processor (see `getpixels()`). The processor's snapshot array is reset.

Object `getPixelsCopy()`

Returns a reference to the image's snapshot (undo) array. If the snapshot array is `null`, a *copy* of the processor's pixel data is returned. Otherwise the use of the result is the same as with `getPixels()`.

`void getColumn(int u, int v, int[] data, int n)`

Returns n contiguous pixel values along the vertical column u , starting at position (u, v) . The result is stored in the array *data* (which must not be `null` and at least of size n).

`void putColumn(int u, int v, int[] data, int n)`

Inserts the first n pixels contained in *data* into the vertical column u , starting at position (u, v) . *data* must not be `null` and at least of size n .

`void getRow(int u, int v, int[] data, int m)`

Returns m contiguous pixel values along the horizontal line v , starting at position (u, v) . The result is stored in the array *data* (which must not be `null` and at least of size m).

`void putRow(int u, int v, int[] data, int m)`

Inserts the first m pixels contained in *data* into the horizontal row v , starting at position (u, v) . *data* must not be `null` and at least of size m .

`double[] getLine(double x1, double y1, double x2, double y2)`

Returns a one-dimensional array containing the pixel values along the straight line starting at position $(x1, y1)$ and ending at $(x2, y2)$. The length of the returned array corresponds to the rounded integer distance between the start and endpoint, any of which may be outside the image bounds. Interpolated pixel values are used if the processor's interpolation setting is on (see `setInterpolate()`).

`void insert(ImageProcessor ip, int u, int v)`

Inserts (pastes) the image contained in *ip* into this image at position (u, v) .

C.7.4 Accessing All Pixels at Once**`int[][] getIntArray()`**

Returns the contents of the image as a new two-dimensional `int` array, by storing the pixels as *array*[*u*][*v*] (i.e., in column-first order).

`void setIntArray(int[][] pixels)`

Replaces the image pixels with the contents of the two-dimensional `int` array *pixels*, which must be of exactly the same size as the target image. Pixels are assumed to be arranged in column-first order (as above).

`float[] [] getFloatArray ()`

Returns the contents of the image as a new two-dimensional `int` array, by storing the pixels as *array* [u] [v] (i. e., in column-first order).

`void setFloatArray (float[] [] pixels)`

Replaces the image pixels with the contents of the two-dimensional `float` array *pixels*, which must be of exactly the same size as the target image. Pixels are assumed to be arranged in column-first order (as above).

C.7.5 Specific Access Methods for Color Images

The following methods are only defined for objects of type `ColorProcessor`.

`void getRGB (byte[] R, byte[] G, byte[] B)`

Stores the red, green, and blue color planes into three separate `byte` arrays *R*, *G*, *B*, whose size must be at least equal to the number of pixels in this image.

`void setRGB (byte[] R, byte[] G, byte[] B)`

Fills the pixel array of this color image from the contents of the `byte` arrays *R*, *G*, *B*, whose size must be at least equal to the number of pixels in this image.

`void getHSB (byte[] H, byte[] S, byte[] B)`

Stores the *hue*, *saturation*, and *brightness* values into three separate `byte` arrays *H*, *S*, *B*, whose size must be at least equal to the number of pixels in this image.

`void setHSB (byte[] H, byte[] S, byte[] B)`

Fills the pixel array of this color image from the contents of the `byte` arrays *H* (*hue*), *S* (*saturation*), and *B* (*brightness*), whose size must be at least equal to the number of pixels in this image.

`FloatProcessor getBrightness ()`

Returns the *brightness* values (as defined by the HSV color model) of this color image as a new `FloatProcessor` of the same size.

`void setBrightness (FloatProcessor fp)`

Replaces the *brightness* values (as defined by the HSV color model) of this color image by the values of the corresponding pixels in the specified `FloatProcessor` *fp*, which must be of the same size as this image.

C.7.6 Direct Access to Pixel Arrays

The use of pixel access methods (such as `getPixel()` and `putPixel()`) is relatively time-consuming because these methods perform careful bounds checking on the given pixel coordinates. The alternative methods `set()` and `get()` do *no* bounds checking and are thus somewhat faster but still carry the overhead of method invocation.

```

1  public void run (ImageProcessor ip) {
2      // check if ip is really a valid ByteProcessor:
3      if (!(ip instanceof ByteProcessor)) return;
4      if (!(ip.getPixels() instanceof byte[])) return;
5      // get pixel array:
6      byte[] pixels = (byte[]) ip.getPixels();
7      int w = ip.getWidth();
8      int h = ip.getHeight();
9      // process pixels:
10     for (int v = 0; v < h; v++) {
11         for (int u = 0; u < w; u++) {
12             int p = 0xFF & pixels[v * w + u];
13             p = p + 1;
14             pixels[v * w + u] = (byte) (0xFF & p);
15         }
16     }
17 }

```

Program C.1

Direct pixel access for images of type `ByteProcessor`. Notice the use of the `instanceof` operator (lines 3–4) to verify the correct type of the processor. In this case (since the pixel coordinates (u , v) are not used in the computation), a single loop over the one-dimensional pixel array could be used instead.

If many pixels must be processed, *direct* access to the elements of the processor's pixel array may be considerably more efficient. For this we have to consider that the pixel arrays of Java and ImageJ images are one-dimensional and arranged in row-first order (also see Sec. B.2.3).

A reference to a processor's one-dimensional pixel array *pixels* is obtained by the method `getPixels()`. To retrieve a particular pixel at position (u, v) we must first compute its one-dimensional index i , where the width w (i. e., the length of each line) of the image must be known:

$$I(u, v) \equiv \text{pixels}[i] = \text{pixels}[v \cdot w + u] .$$

Program C.1 shows an example for direct pixel access inside the `run` method of a ImageJ plugin for an image of type `ByteProcessor`. The bit operations `0xFF & pixels[]` and `0xFF & p` (lines 12 and 14, respectively) are needed to use unsigned `byte` data in the range $[0 \dots 255]$ (as described in Sec. B.1.3). Analogously, the bit-mask `0xFFFF` and a typecast to `(short)` would be required when processing unsigned 16-bit images of type `ShortProcessor`.

If, as in Prog. C.1, the pixels' coordinates (u, v) are not used in the computation and the order of pixels being accessed is irrelevant, a single loop can be used to iterate over all elements of the one-dimensional pixel array (of length $w \cdot h$). This approach is used, for example, to process all pixels of a color image in Prog. 12.1 (p. 246). Also, one should consider the 1D access methods in Sec. C.7.2 as a simple and similarly efficient alternative to the direct access scheme described above.

C.8 Converting Images

C.8.1 ImageProcessor (Class)

The class `ImageProcessor` implements the following basic methods for converting between different types of images. Each method returns a new `ImageProcessor` object unless the original image is of the desired type already. If this is the case, only a reference to the source image is returned, i.e., *no* duplication occurs.

`ImageProcessor convertToByte (boolean doScaling)`

Copies the contents of the source image to a new object of type `ByteProcessor`. If *doScaling* is `true`, the pixel values are automatically scaled to the range of the target image; otherwise the values are clamped without scaling. If applied to an image of type `colorProcessor`, the intensity values are computed as the weighted sum of the RGB component values. The RGB weights can be set using the method `setWeightingFactors()` (see p. 485).

`ImageProcessor convertToShort (boolean doScaling)`

Copies the contents of the source image to a new object of type `ShortProcessor`. If *doScaling* is `true`, the pixel values are automatically scaled to the range of the target image; otherwise the values are clamped without scaling.

`ImageProcessor convertToFloat ()`

Copies the contents of the source image to a new object of type `FloatProcessor`.

`ImageProcessor convertToRGB ()`

Copies the contents of the source image to a new object of type `ColorProcessor`.

C.8.2 ImagePlus, ImageConverter (Classes)

Images of type `ImagePlus` can be converted by instances of the class `ImageConverter` (package `ij.process`). To convert a given `ImagePlus` object `imp`, we first create an instance of the class `ImageConverter` for that image and then invoke a conversion method; for example,

```
ImageConverter iConv = new ImageConverter(imp);  
iConv.convertToGray8();
```

This *destructively* modifies the image `imp` to an 8-bit grayscale image by replacing the attached `ImageProcessor` (among other things). No conversion takes place if the original image is of the target type already. The complete ImageJ plugin in Prog. C.2 illustrates how `ImageConverter` could be used to convert any image to an 8-bit grayscale image before processing.

In summary, the following methods are applicable to `ImageConverter` objects:

```

1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageConverter;
4 import ij.process.ImageProcessor;
5
6 public class Convert_ImagePlus_To_Gray8
7     implements PlugInFilter {
8
9     ImagePlus imp = null;
10
11     public int setup(String arg, ImagePlus imp) {
12         if (imp == null) {
13             IJ.noImage();
14             return DONE;
15         }
16         this.imp = imp;
17         return DOES_ALL; // this plugin accepts any type of image
18     }
19
20     public void run(ImageProcessor ip) {
21         ImageConverter iConv = new ImageConverter(imp);
22         iConv.convertToGray8();
23         ip = imp.getProcessor(); // ip is now of type ByteProcessor
24         // process grayscale image ...
25     }
26
27 } // end of class Convert_ImagePlus_To_Gray8

```

Program C.2

ImageJ sample plugin for converting any type of `ImagePlus` image to 8-bit grayscale. The actual conversion takes place on line 22. The updated image processor is retrieved (line 23) and can subsequently be used to process the converted image. Notice that the original `ImagePlus` object `imp` is not passed to the plugin's `run()` method but only to the `setup()` method, which is called first (by ImageJ's plugin mechanism) and keeps a reference in the instance variable `imp` (line 16) for later use.

`void convertToGray8 ()`

Converts the source image to an 8-bit (byte) grayscale image.

`void convertToGray16 ()`

Converts the source image to a 16-bit (short) grayscale image.

`void convertToGray32 ()`

Converts the source image to a 32-bit (float) grayscale image.

`void convertToRGB ()`

Converts the source image to a 32-bit (int) RGB color image.

`void convertToHSB ()`

Converts a given RGB image to an HSB⁹ (hue, saturation, brightness) stack, a stack of three independent grayscale images. May not be applied to another type of image.

`void convertHSBToRGB ()`

Converts an HSB image stack to a single RGB image.

`void convertRGBStackToRGB ()`

Converts an RGB image stack to a single RGB image.

⁹ HSB is identical to the HSV color space (see Sec. 12.2.3).

void convertToRGBStack ()
Converts an RGB image to a three-slice RGB image stack.

void convertRGBtoIndexedColor (int *nColors*)
Converts an RGB image to an indexed color image with *nColors* colors.

void setDoScaling (boolean *doScaling*)
Enables or disables the scaling of pixel values. If *doScaling* is **true**, pixel values are scaled to [0...255] when converted to 8-bit images and to [0...65,535] for 16-bit images. Otherwise no scaling is applied.

void getDoScaling ()
Returns **true** if scaling is enabled for that **ImageConverter** object.

C.9 Histograms and Image Statistics

C.9.1 ImageProcessor (Class)

int[] getHistogram ()
Returns the histogram of the image or the region of interest (ROI), if selected. For images of type **ColorProcessor**, the intensity histogram is returned, where intensities are computed as weighted sums of the RGB components. The RGB weights can be set using the method **setWeightingFactors()** (see p. 485).

double getHistogramMax ()
Returns the maximum pixel value used for computing histograms of float images.

double getHistogramMin ()
Returns the minimum pixel value used for computing histograms of float images.

int getHistogramSize ()
Returns the number of bins used for computing histograms of float images.

void setHistogramRange (double *histMin*, double *histMax*)
Specifies the range of pixel values used for computing histograms of float images.

int setHistogramSize (int *size*)
Specifies the number of bins used for computing histograms of float images.

Additional statistics can be obtained through the class **ImageStatistics** and its subclasses **ByteStatistics**, **ShortStatistics**, **FloatStatistics**, **ColorStatistics**, and **StackStatistics**.

C.10 Point Operations

Single-image operations

The following methods for objects of type `ImageProcessor` perform arithmetic or logic operations with a constant scalar value as the second operand. All operations are applied either to the whole image or to the pixels within the region of interest, if selected.

```
void abs ()
    Replaces every pixel by its absolute value.

void add (int value)
    Increments every pixel by value.

void add (double value)
    Increments every pixel by value.

void and (int value)
    Bitwise AND operation between the pixel and value.

void applyTable (int[] lut)
    Applies the mapping specified by the lookup table lut to each pixel.

void autoThreshold ()
    Converts the image to binary using a threshold determined automatically from the original histogram.

void gamma (double g)
    Applies a gamma correction with the gamma value g.

void log ()
    Replaces every pixel a by  $\log_{10}(a)$ .

void max (double value)
    Maximum operation: pixel values greater than value are set to value.

void min (double value)
    Minimum operation: pixel values smaller than value are set to value.

void multiply (double value)
    All pixels are multiplied by value.

void noise (double r)
    Increments every pixel by a random value with normal distribution in the range  $\pm r$ .

void or (int value)
    Bitwise OR operation between the pixel and value.

void sqr ()
    Replaces every pixel a by  $a^2$ .

void sqrt ()
    Replaces every pixel a by  $\sqrt{a}$ .
```

`void threshold (int th)`

Threshold operation: sets every pixel a with $a \leq th$ to 0 and all other pixels to 255.

`void xor (int value)`

Bitwise exclusive-OR (XOR) operation between the pixel and *value*.

Multi-image operations

The class `ImageProcessor` defines a single method for combining two images:

`void copyBits (ImageProcessor B, int u, int v, int mode)`

Copies the image *B* into the target image at position (*u*, *v*) using the transfer mode *mode*. The target image is destructively modified, and *B* remains unchanged.

Admissible *mode* values are defined as constants by the `Blitter` interface (see below); for example,

`ipA.copyBits(ipB, 0, 0, Blitter.COPY);`

for copying (pasting) the contents of image *ipB* into *ipA*. Another example for the use of `copyBits()` can be found in Sec. 5.8.3 (page 81).

In summary, `ij.process.Blitter` defines the following *mode* values for the `copyBits()` method (*A* refers to the target image, *B* to the source image):

ADD

$$A(u, v) \leftarrow A(u, v) + B(u, v)$$

AND

$$A(u, v) \leftarrow A(u, v) \wedge B(u, v)$$

Bitwise AND operation.

AVERAGE

$$A(u, v) \leftarrow (A(u, v) + B(u, v))/2$$

COPY

$$A(u, v) \leftarrow B(u, v)$$

COPY_INVERTED

$$A(u, v) \leftarrow 255 - B(u, v)$$

Only applicable to 8-bit grayscale and RGB images.

DIFFERENCE

$$A(u, v) \leftarrow |A(u, v) - B(u, v)|$$

DIVIDE

$$A(u, v) \leftarrow A(u, v)/B(u, v)$$

MAX

$$A(u, v) \leftarrow \max(A(u, v), B(u, v))$$

MIN

$$A(u, v) \leftarrow \min(A(u, v), B(u, v))$$

MULTIPLY

$$A(u, v) \leftarrow A(u, v) \cdot B(u, v)$$

OR

$$A(u, v) \leftarrow A(u, v) \vee B(u, v)$$

Bitwise OR operation.

SUBTRACT

$$A(u, v) \leftarrow A(u, v) - B(u, v)$$

XOR

$$A(u, v) \leftarrow A(u, v) \text{ xor } B(u, v)$$

Bitwise exclusive OR (XOR) operation.

C.11 Filters

C.11.1 ImageProcessor (Class)

void convolve (float[] *kernel*, int *w*, int *h*)

Performs a linear convolution of the image with the filter matrix *kernel* (of size $w \times h$), specified as a one-dimensional float array.

void convolve3x3 (int[] *kernel*)

Performs a linear convolution of the image with the filter matrix *kernel* (of size 3×3), specified as a one-dimensional int array.

void dilate ()

Dilation using a 3×3 minimum filter.

void erode ()

Erosion using a 3×3 maximum filter.

void findEdges ()

Applies a 3×3 edge filter (Sobel operator).

void medianFilter ()

Applies a 3×3 median filter.

void smooth ()

Applies a simple 3×3 average filter (box filter).

void sharpen ()

Sharpens the image using a 3×3 Laplacian-like filter kernel.

C.12 Geometric Operations

C.12.1 ImageProcessor (Class)

ImageProcessor crop ()

Creates a new ImageProcessor object with the contents of the current region of interest.

void flipHorizontal ()

Destructively mirrors the contents of the image (or region of interest) horizontally.

void flipVertical ()
Destructively mirrors the contents of the image (or region of interest) vertically.

ImageProcessor resize (int *width*, int *height*)
Creates a new **ImageProcessor** object containing a scaled copy of this image (or region of interest) of size *width* × *height*.

void rotate (double *angle*)
Destructively rotates the image (or region of interest) *angle* degrees clockwise.

ImageProcessor rotateLeft ()
Rotates the entire image 90° counterclockwise and returns a new **ImageProcessor** object that contains the rotated image.

ImageProcessor rotateRight ()
Rotates the entire image 90° clockwise and returns a new **ImageProcessor** object that contains the rotated image.

void scale (double *xScale*, double *yScale*)
Destructively scales (zooms) the image (or region of interest) in *x* and *y* by the factors *xScale* and *yScale*, respectively. The size of the image does not change.

void setBackgroundValue (double *value*)
Sets the background fill value used by the **rotate()** and **scale()** methods.

boolean getInterpolate ()
Returns **true** if (bilinear) interpolation is turned on for this image processor.

void setInterpolate (boolean *interpolate*)
Activates bilinear interpolation for geometric operations (otherwise nearest-neighbor interpolation is used).

double getInterpolatedPixel (double *x*, double *y*)
Returns the interpolated pixel value for the continuous coordinates (*x*, *y*) using bilinear interpolation. In case of a **ColorProcessor**, the gray value resulting from nearest-neighbor interpolation is returned (use **getInterpolatedRGBPixel()** to obtain interpolated color values).

double getInterpolatedRGBPixel (double *x*, double *y*)
Returns the interpolated RGB pixel value for the continuous coordinates (*x*, *y*) using bilinear interpolation. This method is defined for **ColorProcessor** only.

C.13.1 ImageProcessor (Class)

void drawDot (int *u*, int *v*)
Draws a dot centered at position (*u*, *v*) using the current line width and fill/draw value.

void drawLine (int *u1*, int *v1*, int *u2*, int *v2*)
Draws a line from position (*u1*, *v1*) to position (*u2*, *v2*).

void drawOval (int *u*, int *v*, int *w*, int *h*)
Draws an axis-parallel ellipse with a bounding rectangle of size *w* × height *h*, positioned at (*u*, *v*). See also `fillOval()`.

void drawPixel (int *u*, int *v*)
Sets the pixel at position (*u*, *v*) to the current fill/draw value.

void drawPolygon (java.awt.Polygon *p*)
Draws the polygon *p* using the current fill/draw value. See also `fillPolygon()`.

void drawRect (int *u*, int *v*, int *width*, int *height*)
Draws a rectangle of size *width* × *height* and parallel to the coordinate axes.

void drawString (String *s*)
Draws the string *s* at the *current drawing position* (set with `moveTo()` or `lineTo()`) using the current fill/draw value and font. Use `setAntialiasedText()` to control anti-aliasing for text rendering.

void drawString (String *s*, int *u*, int *v*)
Draws the string *s* at position (*u*, *v*) using the current fill/draw value and font.

void fill ()
Fills the image or region of interest (if selected) with the current fill/draw value.

void fill (ImageProcessor *mask*)
Fills the pixels that are inside both the region of interest and the mask image *mask*, which must be of the same size as the image or region of interest. A position is considered *inside* the mask if the corresponding mask pixel has a nonzero value.

void fillOval (int *u*, int *v*, int *w*, int *h*)
Draws and fills an axis-parallel ellipse with a bounding rectangle of size *w* × height *h*, positioned at (*u*, *v*). See also `drawOval()`.

void fillPolygon (java.awt.Polygon *p*)
Draws and fills the polygon *p* using the current fill/draw value. See also `drawPolygon()`.

void getStringWidth (String *s*)
Returns the width (in pixels) of the string *s* using the current font.

`void insert (ImageProcessor src, int u, int v)`
Inserts the image contained in *src* at position (*u*, *v*).

`void lineTo (int u, int v)`
Draws a line from the *current drawing position* to (*u*, *v*). Updates the current drawing position to (*u*, *v*).

`void moveTo (int u, int v)`
Sets the *current drawing position* to (*u*, *v*).

`void setAntialiasedText (boolean antialiasedText)`
Specifies whether or not text is rendered using anti-aliasing.

`void setClipRect (Rectangle clipRect)`
Sets the clipping rectangle used by the methods `lineTo()`, `drawLine()`, `drawDot()`, and `drawPixel()`.

`void setColor (java.awt.Color color)`
Sets the default fill/draw value for subsequent drawing operations to the pixel value closest to the specified color.

`void setFont (java.awt.Font font)`
Sets the font to be used by `drawString()`.

`void setJustification (int justification)`
Sets the justification used by `drawString()`. Admissible values for *justification* are the constants `CENTER_JUSTIFY`, `RIGHT_JUSTIFY`, and `LEFT_JUSTIFY` (defined in class `ImageProcessor`).

`void setLineWidth (int width)`
Sets the line width used by `lineTo()` and `drawDot()`.

`void setValue (double value)`
Sets the fill/draw value for subsequent drawing operations. Notice that the `double` parameter *value* is interpreted differently depending on the type of image. For `ByteProcessor`, `ShortProcessor`, and `FloatProcessor`, the numerical value of *value* is simply converted by typecasting to the corresponding pixel type. For images of type `ColorProcessor`, *value* is first typecast to `int` and the result is interpreted as a packed α RGB value.

C.14 Displaying Images and Image Stacks

Only images of type `ImagePlus` (which include stacks of images) may be displayed on the screen using the methods below. In contrast, objects of type `ImageProcessor` are not visible themselves but can only be displayed through an associated `ImagePlus` object, as described in Sec. C.14.2.

C.14.1 ImagePlus (Class)

`void draw ()`
Draws the image and the outline of the region of interest (if se-

lected). Does nothing if there is no window associated with this image (i. e., `show()` has not been called).

void draw (int *u*, int *v*, int *width*, int *height*)
Draws the image and the outline of the region of interest (as above) using the clipping rectangle specified by the four parameters.

int getCurrentSlice ()
Returns the index of the currently displayed stack slice or 1 if this `ImagePlus` is a single image. Use `setSlice()` to display a particular slice.

int getID ()
Returns this image's unique ID number. This ID can be used with the `WindowManager`'s method `getImage()` to reference a particular image.

String getShortTitle ()
Returns a shortened version of the image's name.

String getTitle ()
Returns the image's full name.

ImageWindow getWindow ()
Returns the window (of type `ij.gui.ImageWindow`, a subclass of `java.awt.Frame`) that is being used to display this `ImagePlus` image.

void hide ()
Closes any window currently displaying this image.

boolean isInvertedLut ()
Returns `true` if this image's `ImageProcessor` uses an inverting lookuptable (LUT) for displaying zero pixel values as white and 255 as black. The LUT can be inverted by calling `invertLut()` on the corresponding `ImageProcessor` (which is obtained with `getProcessor()`).¹⁰

void repaintWindow ()
Calls `draw()` to draw the image and also repaints the image window to update the header information (dimension, type, size).

void setSlice (int *index*)
Displays the specified slice of a stack. The parameter *index* must be $1 \leq \textit{index} \leq N$, where N is the number of slices in the stack. Redisplays the (single) image if this `ImagePlus` does not contain a stack.

void setTitle (String *title*)
Sets the image name to *title*.

void show ()
Opens a window to display this image and clears the status bar in the main `ImageJ` window.

¹⁰ The class `ImagePlus` also defines a method `invertLookupTable()`, but this method is not public.

void show (String *statusMsg*)
Opens a window to display this image and displays the text *statusMsg* in the status bar.

void updateAndDraw ()
Updates this image from the pixel data in its associated **ImageProcessor** object and then displays it (by calling **draw()**).

void updateAndRepaintWindow ()
Calls **updateAndDraw()** to repaint the current pixel data and also updates the header information (dimension, type, size).

C.14.2 ImageProcessor (Class)

As mentioned above, objects of type **ImageProcessor** are not visible automatically but require an associated **ImagePlus** object to be seen on the screen.

The **ImageProcessor** object passed to a typical ImageJ plugin (of class **PlugInFilter**) belongs to a visible image and thus already has an associated **ImagePlus**, which is passed to the **setup()** method of that plugin. This **ImagePlus** object can be used to redisplay the image at any time during plugin execution, as exemplified in Prog. C.3.

To display the contents of a new **ImageProcessor**, a corresponding **ImagePlus** object must first be created for it using the constructor methods described in Sec. C.3.1; e. g.,

```
ByteProcessor ip = new ByteProcessor(200,300);
ImagePlus im = new ImagePlus ("A New Image", ip);
im.show();           // show image on screen
ip.smooth();         // modify this image
im.updateAndDraw();  // redisplay modified image
```

Notice that there is no simple way to access the **ImagePlus** object associated with a given **ImageProcessor** or determine if one exists at all. In reverse, the image processor of a given **ImagePlus** can be obtained directly with the **getProcessor()** method (see Sec. C.21.1). Analogously, an image *stack* associated with a given **ImagePlus** object is retrieved by the method **getStack()** (see Sec. C.15.1).

The following methods for the class **ImageProcessor** control the mapping between the original pixel values and display intensities:

ColorModel getColorModel ()
Returns this image processor's color model (of type **java.awt.image.ColorModel**): **IndexColorModel** for grayscale and indexed color images, and **DirectColorModel** for RGB color images. For processors other than **ColorProcessor**, this is the base lookup table, not the one that may have been modified by **setMinAndMax()** or **setThreshold()**. An **ImageProcessor**'s color model can be changed with the method **setColorModel()**.

Program C.3

Animation example (redisplaying the image passed to an ImageJ plugin). The ImagePlus object associated with the ImageProcessor passed to the plugin is initially received by the `setup()` method, where a reference is stored in variable `im` (line 14). Inside the `run()` method, the ImageProcessor is repeatedly modified (lines 21–22) and subsequently redisplayed by invoking `updateAndDraw()` on the associated ImagePlus object `im` (line 24).

```
1 import ij.IJ;
2 import ij.ImagePlus;
3 import ij.plugin.filter.PlugInFilter;
4 import ij.process.ImageProcessor;
5
6 public class Display_Demo implements PlugInFilter {
7     ImagePlus im = null;
8
9     public int setup(String arg, ImagePlus im) {
10         if (im == null) {
11             IJ.noImage();
12             return DONE;
13         }
14         this.im = im; // keep reference to associated ImagePlus
15         return DOES_ALL;
16     }
17
18     public void run(ImageProcessor ip) {
19         for (int i = 0; i < 10; i++) {
20             // modify this image:
21             ip.smooth();
22             ip.rotate(30);
23             // redisplay this image:
24             im.updateAndDraw();
25             // sleep 100 ms so user can watch:
26             IJ.wait(100);
27         }
28     }
29
30 } // end of class Display_Demo
```

ColorModel getCurrentColorModel ()

Returns the current color model, which may have been modified by `setMinAndMax()` or `setThreshold()`.

double getMax ()

Returns the largest displayed pixel value a_{\max} (pixels $I(u, v) > a_{\max}$ are mapped to 255). a_{\max} can be modified with the method `setMinMax()`.

double getMin ()

Returns the smallest displayed pixel value a_{\min} (pixels $I(u, v) < a_{\min}$ are mapped to 0). a_{\min} can be modified with the method `setMinMax()`.

void invertLut ()

Inverts the values in this ImageProcessor's lookuptable for displaying zero pixel values as white and 255 as black. Does nothing if this is a ColorProcessor.

boolean isInvertedLut ()

Returns **true** if this **ImageProcessor** uses an inverting lookup-table for displaying zero pixel values as white and 255 as black.

void resetMinAndMax ()

For **ShortProcessor** and **FloatProcessor** images, the a_{\min} and a_{\max} values are recalculated to correctly display the image. For **ByteProcessor** and **ColorProcessor**, the lookuptables are reset to default values.

void setMinAndMax (double *a_{min}*, double *a_{max}*)

Sets the parameters a_{\min} and a_{\max} to the specified values. The image is displayed by mapping the pixel values in the range $[a_{\min} \dots a_{\max}]$ to screen values in the range $[0 \dots 255]$.

void setColorModel (java.awt.image.ColorModel *cm*)

Sets the color model. Except for **ColorProcessor**, *cm* must be of type **IndexColorModel**.

C.15 Operations on Image Stacks

C.15.1 ImagePlus (Class)

For creating ready-to-use multislice stack images, see the methods for class **NewImage** (Sec. C.3.4).

ImageStack createEmptyStack ()

Returns a new, *empty* stack with the same width, height, and color table as the given **ImagePlus** object to which this method is applied. Notice that the new stack is *not* automatically attached to this **ImagePlus** by this method (use **setStack()** for this purpose).

ImageStack getImageStack ()

Returns the image stack associated with the **ImagePlus** object to which this method is applied. Calls **getStack()** if the image has no stack yet.

ImageStack getStack ()

Returns the image stack associated with the **ImagePlus** object to which this method is applied. If no stack exists, a new single-slice stack is created with the contents of that image (by calling **createEmptyStack()**). After adding or removing slices to/from the returned **ImageStack** object, **setStack()** should be called to update the image and the window that is displaying it.

int getStackSize ()

If this **ImagePlus** contains a stack, the number of slices is returned; 1 is returned if this is a single image.

void setStack (String *title*, ImageStack *stack*)

Replaces the current stack of this **ImagePlus**, if any, with *stack* and assigns the name *title*.

C.15.2 ImageStack (Class)

For creating new `ImageStack` objects, see the constructor methods in Sec. C.3.2.

```
void addSlice (String label, ImageProcessor ip)
    Adds the image specified by ip to the end of the stack, assigning
    the title label to the new slice. No pixel data are duplicated.

void addSlice (String label, ImageProcessor ip, int n)
    Adds the image specified by ip to the stack following slice n,
    assigning the title label to the new slice. The slice is added
    to the beginning of the stack if n is zero. No pixel data are
    duplicated.

void addSlice (String label, Object pixels)
    Adds the image specified by pixels (which must be a suitable
    pixel array) to the end of the stack.

void deleteLastSlice ()
    Deletes the last slice in the stack.

void deleteSlice (int n)
    Deletes the nth slice from the stack, where  $1 \leq n \leq \text{getsize}()$ .

int getHeight ()
    Returns the height of the images in this stack.

Object[] getImageArray ()
    Returns the whole stack as an array of one-dimensional pixel ar-
    rays. Note that the size of the returned array may be greater than
    the number of slices currently in the stack, with unused elements
    set to null. No pixel data are duplicated.

Object getPixels (int n)
    Returns the one-dimensional pixel array for the nth slice of the
    stack, where  $1 \leq n \leq \text{getsize}()$ . No pixel data are duplicated.

ImageProcessor getProcessor (int n)
    Creates and returns an ImageProcessor for the nth slice of the
    stack, where  $1 \leq n \leq \text{getsize}()$ . No pixel data are duplicated.
    The method returns null if the stack is empty.

int getSize ()
    Returns the number of slices in this stack.

String getSliceLabel (int n)
    Returns the label of the nth slice, where  $1 \leq n \leq \text{getsize}()$ .
    Returns null if the slice has no label.

String[] getSliceLabels ()
    Returns the labels of all slices as an array of strings. Note that
    the size of the returned array may be greater than the number of
    slices currently in the stack. Returns null if the stack is empty
    or the label of the first slice is null.

String getShortSliceLabel (int n)
    Returns a shortened version (up to the first 60 characters or first
```


newline character and suffix removed) of the n th slice's label, where $1 \leq n \leq \text{getsize}()$. Returns `null` if the slice has no label.

`int getWidth()`

Returns the height of the images in this stack.

`void setPixels (Object pixels, int n)`

Assigns the pixel array *pixels* to the n th slice, where $1 \leq n \leq \text{getsize}()$. No pixel data are duplicated.

`void setSliceLabel (String label, int n)`

Assigns the title *label* to the n th slice, where $1 \leq n \leq \text{getsize}()$.

C.15.3 Stack Example

Programs C.4 and C.5 shows a working example for the use of image stacks that blends one image into another by a simple technique called “alpha blending” by producing a sequence of intermediate images stored in a stack. This is an extension of Progs. 5.5 and 5.6 (see p. 85), which produce only a single blended image.

The background image (`bgIp`) is the current image (i. e., the image to which the plugin is applied) that is passed to the plugin's `run()` method. The foreground image (`fgIp`) is selected through a dialog window (created with `GenericDialog`), as well as the number of slices in the stack to be created.

In the plugin's `run()` method (Prog. C.5, line 64), a stack with the required number of slices is created first using the static method `NewImage.createByteImage()`. In the following loop, a varying transparency value α (see Eqn. (5.42)) is computed for each frame, and the corresponding stack image (slice) is replaced by the weighted sum of the two original images. Note that the slices of a stack of size N are numbered $1 \dots N$ (`getProcessor()` in line 71), in contrast to the usual numbering scheme. A sample result and the corresponding dialog window are shown in Fig. C.7.

C.16 Regions of Interest

A region of interest (ROI) is used to select a particular image region for subsequent processing and is usually specified interactively by the user. ImageJ supports several types of ROI, including:

- rectangular (class `Roi`)
- elliptical (class `OvalRoi`)
- straight line (class `Line`)
- polygon/polyline (classes `PolygonRoi`, `FreehandRoi`)
- point set (class `PointRoi`)

```

1 import ij.IJ;
2 import ij.ImagePlus;
3 import ij.ImageStack;
4 import ij.WindowManager;
5 import ij.gui.*;
6 import ij.plugin.filter.PlugInFilter;
7 import ij.process.*;
8
9 public class Alpha_Blending_Stack implements PlugInFilter {
10     static int nFrames = 10;
11     ImagePlus fgIm; // foreground image (chosen interactively)
12
13     public int setup(String arg, ImagePlus imp) {
14         return DOES_8G;}
15
16     boolean runDialog() {
17         // get list of open images
18         int[] windowList = WindowManager.getIDList();
19         if(windowList==null) {
20             IJ.noImage();
21             return false;
22         }
23         String[] windowTitles = new String[windowList.length];
24         for (int i = 0; i < windowList.length; i++) {
25             ImagePlus imp = WindowManager.getImage(windowList[i]);
26             if (imp != null)
27                 windowTitles[i] = imp.getShortTitle();
28             else
29                 windowTitles[i] = "untitled";
30         }
31         GenericDialog gd = new GenericDialog("Alpha Blending");
32         gd.addChoice("Foreground image:",
33                     windowTitles, windowTitles[0]);
34         gd.addNumericField("Frames:", nFrames, 0);
35         gd.showDialog();
36         if (gd.wasCanceled())
37             return false;
38         else {
39             int img2Index = gd.getNextChoiceIndex();
40             fgIm = WindowManager.getImage(windowList[img2Index]);
41             nFrames = (int) gd.getNextNumber();
42             if (nFrames < 2)
43                 nFrames = 2;
44             return true;
45         }
46     } // continued...

```

C.16 REGIONS OF INTEREST

Program C.4

Stack example—alpha blending (*part 1 of 2*). This is an extended version of the alpha blending example described in the main text (Progs. 5.5 and 5.6). It blends two given images with incrementally changing alpha weights and stores the results in a new stack of images.

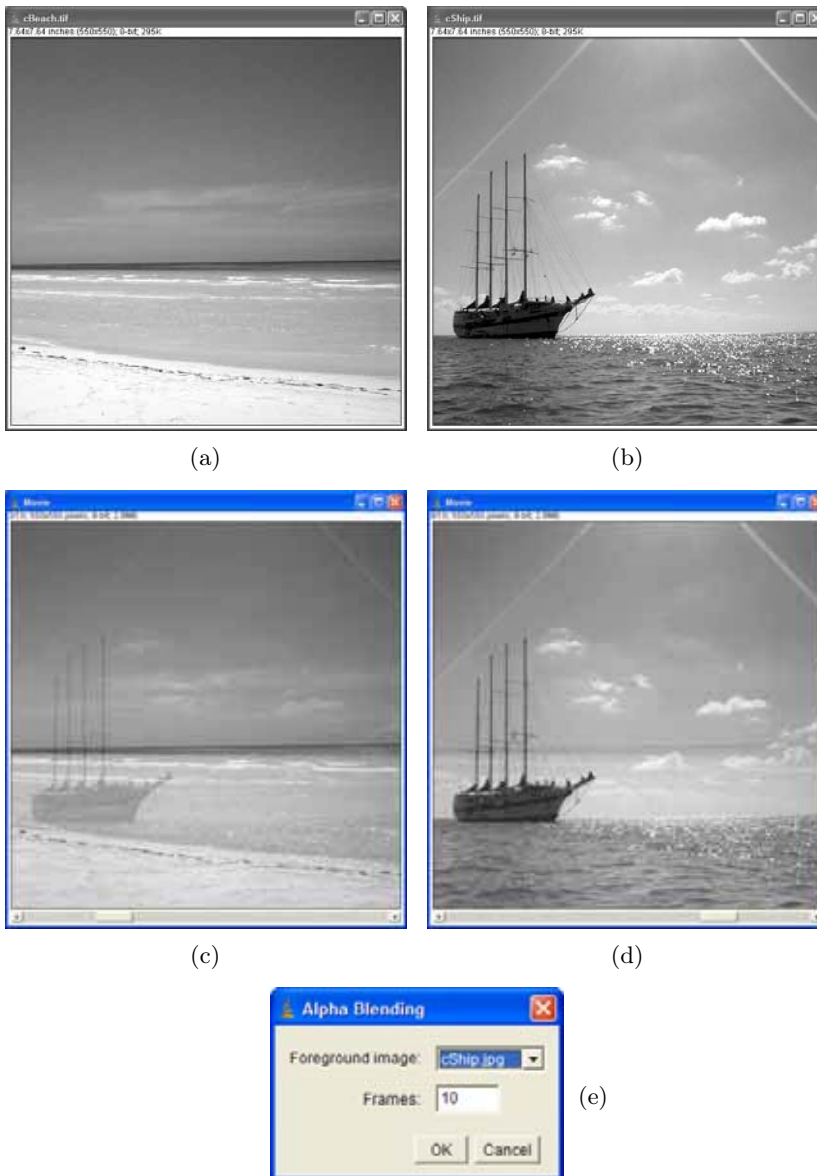
Program C.5
Stack example—alpha
blending (part 2 of 2).

```
48 // class Alpha_Blending_Stack (continued)
49
50 public void run(ImageProcessor bgIp) {
51     // bgIp = background image
52
53     if(runDialog()) { //open dialog box (returns false if cancelled)
54         int w = bgIp.getWidth();
55         int h = bgIp.getHeight();
56
57         // prepare foreground image
58         ImageProcessor fgIp =
59             fgIm.getProcessor().convertToByte(false);
60         ImageProcessor fgTmpIp = bgIp.duplicate();
61
62         // create image stack
63         ImagePlus movie =
64             NewImage.createByteImage("Movie",w,h,nFrames,0);
65         ImageStack stack = movie.getStack();
66
67         // loop over stack frames
68         for (int i=0; i<nFrames; i++) {
69             // transparency of foreground image
70             double iAlpha = 1.0 - (double)i/(nFrames-1);
71             ImageProcessor iFrame = stack.getProcessor(i+1);
72
73             // copy background image to frame i
74             iFrame.insert(bgIp,0,0);
75             iFrame.multiply(iAlpha);
76
77             // copy foreground image and make transparent
78             fgTmpIp.insert(fgIp,0,0);
79             fgTmpIp.multiply(1-iAlpha);
80
81             // add foreground image frame i
82             ByteBlitter blitter =
83                 new ByteBlitter((ByteProcessor)iFrame);
84             blitter.copyBits(fgTmpIp,0,0,Blitter.ADD);
85         }
86
87         // display movie (image stack)
88         movie.show();
89     }
90 }
91
92 } // end of class Alpha_Blending_Stack
```

C.16 REGIONS OF INTEREST

Fig. C.7

Alpha blending example using an image stack (results of Progs. C.4 and C.5). Original images: foreground image (a) and background image (b); frames 3 and 6 of the created image stack (c,d). Note the horizontal “slider” button at the window’s bottom for navigating through the stack. Dialog window for selecting the foreground image and the stack size (e).



The corresponding classes are defined in the `ij.gui` package. ROI objects are usually associated with objects of type `ImagePlus`, as described below.

C.16.1 ImagePlus (Class)

`Roi getRoi ()`

Returns the current ROI object (of type `Roi` or one of its sub-

classes `Line`, `OvalRoi`, `PolygonRoi`, `TextRoi`) of this image. Returns `null` if the image has no ROI.

`void killRoi ()`

Deletes the image's current region of interest.

`void setRoi (int u, int v, int w, int h)`

Assigns a rectangular ROI (of size $w \times h$ and upper left corner positioned at (u, v)) to this image and displays it.

`void setRoi (java.awt.Rectangle rect)`

Assigns the specified rectangular ROI to this image and displays it.

`void setRoi (Roi roi)`

Assigns the specified ROI (of type `Roi` or any of its subclasses) to this image and displays it. Any existing ROI is deleted if *roi* is `null` or its width or height is zero.

`ImageProcessor getMask ()`

For images with nonrectangular ROIs, this method returns a mask image (of type `ByteProcessor`); otherwise it returns `null`. This method calls the `getMask()` method on the image's `ImageProcessor` object and returns the result (see Sec. C.16.3 for details).

C.16.2 Roi, Line, OvalRoi, PointRoi, PolygonRoi (Classes)

`Roi (int u, int v, int width, int height)`

Constructor method: creates a rectangular ROI from the specified parameters.

`Roi (java.awt.Rectangle rect)`

Constructor method: creates a rectangular ROI from a given AWT `Rectangle` object *rect*.

`Roi (int u, int v, ImagePlus imp)`

Constructor method: starts the process of creating a user-defined rectangular ROI from starting point (u, v) in the image *imp*. The user determines the size of the region interactively using rubber banding.

`Line (int u1, int v1, int u2, int v2)`

Constructor method: creates a straight-line ROI between points $(u1, v1)$ and $(u2, v2)$.

`Line (int u, int v, ImagePlus imp)`

Constructor method: starts the process of creating a user-defined straight-line ROI from starting point (u, v) in the image *imp*. The user determines the end of the line interactively using rubber banding.

`OvalRoi (int u, int v, int width, int height)`

Constructor method: creates an elliptic ROI whose bounding box is determined by the given parameters.

`OvalRoi (int u, int v, ImagePlus imp)`
 Constructor method: starts the process of creating a user-defined oval ROI from starting point (*u*, *v*) in the image *imp*.

`PolygonRoi (int[] xPnts, int[] yPnts, int n, int type)`
 Constructor method: creates a new polygon or polyline ROI from the coordinate arrays *xPnts* and *yPnts*, where *n* is the number of polygon points. Admissible values for *type* are `Roi.POLYGON`, `Roi.FREEROI`, `Roi.TRACED_ROI`, `Roi.POLYLINE`, `Roi.FREELINE`, or `Roi.ANGLE`.

`PolygonRoi (java.awt.Polygon p, int type)`
 Creates a new polygon or polyline ROI from a given AWT Polygon object. *type* is used as above.

`PolygonRoi (int u, int v, ImagePlus imp)`
 Constructor method: starts the process of creating a user-defined polygon or polyline ROI from starting point (*u*, *v*) in the image *imp*.

`PointRoi (int[] xPnts, int[] yPnts, int n)`
 Constructor method: creates a new point-set ROI from the coordinate arrays *xPnts* and *yPnts*, where *n* is the number of polygon points.

`PointRoi (int u, int v)`
 Constructor method: creates a new single-point `PointRoi` at position (*u*, *v*).

`PointRoi (int u, int v, ImagePlus imp)`
 Creates a new `PointRoi` for the image *imp* using the screen coordinates (*u*, *v*).

`boolean contains (int u, int v)`
 Returns `true` if the point (*u*, *v*) is within this region of interest and `false` otherwise.

C.16.3 ImageProcessor (Class)

An `ImageProcessor` object may also have an associated region of interest. The mechanism is similar to but nevertheless different from the one used for `ImagePlus` objects. In particular, a nonrectangular ROI is represented by the bounding rectangle in combination with a mask of the same size specified by a one-dimensional `int` array.

`ImageProcessor getMask ()`
 For images with nonrectangular ROIs, this method returns a mask image (of type `ByteProcessor`); otherwise it returns `null`. Pixels “inside” the region of interest have nonzero mask values. This mask image is used for efficiently testing whether a particular (*u*, *v*) coordinate is inside or outside the ROI. Note that the origin of the mask image is *not* the same as for the original image but is anchored at the upper left corner of the ROI’s bounding box (see Prog. C.6 and Fig. C.8 for an example).

byte[] getMaskArray ()

Returns the mask's **byte** array, or **null** if this image has no mask. Note that the origin and the dimensions of the underlying mask image are *not* the same as for the original image. The origin of the mask is anchored at the upper left corner of the ROI's bounding box, and its size is identical to the size of the bounding box.

Rectangle getRoi ()

Returns a rectangle (of type **java.awt.Rectangle**) that represents the current region of interest.

void resetRoi ()

Sets the region of interest to include the entire image.

void setMask (ImageProcessor mask)

Defines a byte mask that limits processing to an irregular ROI. The size of *mask* must be the same as the current region of interest. Pixels "inside" the region of interest have nonzero mask values.

void setRoi (int u, int v, int width, int height)

Defines a rectangular region of interest and deletes the associated mask if *rect* is not the same size as the previous ROI.

void setRoi (java.awt.Rectangle rect)

Defines a rectangular region of interest and deletes the associated mask if *rect* is not the same size as the previous ROI. If *rect* is **null**, the ROI is reset (by calling **resetRoi()**).

void setRoi (Roi roi)

Defines a rectangular or nonrectangular region of interest that consists of a rectangular ROI and a mask.

void setRoi (java.awt.Polygon poly)

Defines a polygon-shaped region of interest that consists of a rectangular ROI and a mask.

C.16.4 ImageStack (Class)

Only rectangular ROIs are applicable to image stacks:

java.awt.Rectangle getRoi ()

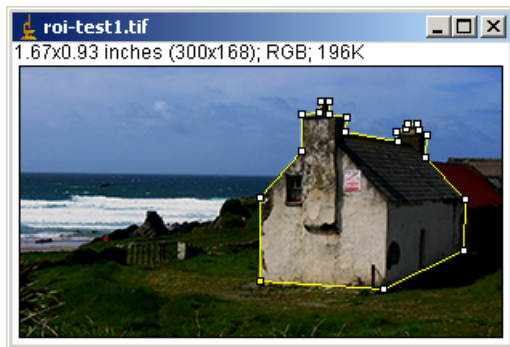
Returns an AWT **Rectangle** object that represents the current region of interest for this image stack.

void setRoi (java.awt.Rectangle rect)

Specifies a rectangular region of interest for this entire image stack.

C.16.5 IJ (Class)

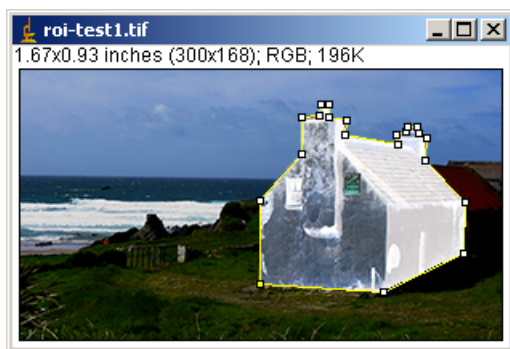
The following static ROI methods in class **IJ** apply to the currently active (user-selected) image:



(a)



(b)



(c)

C.17 IMAGE PROPERTIES

Fig. C.8

Nonrectangular ROI example. Original image with polygon-shaped selection (a). Binary mask image returned by the `ImageProcessor`'s `getMaskArray()` method (b). Note that the origin of the mask image is positioned at the upper left corner of the ROI's bounding box. Result with pixels inside the ROI being modified (c). See Prog. C.6 for implementation details.

```
static void makeLine (int u1, int v1, int u2, int v2)
    Creates a straight-line selection (region of interest) on the cur-
    rently active image (i.e., the image selected by the user).

static void makeOval (int u, int v, int w, int h)
    Creates an elliptical region of interest of size ( $w \times h$ ) on the cur-
    rently active image.

static void makeRectangle (int u, int v, int w, int h)
    Creates a rectangular region of interest of size ( $w \times h$ ) on the
    currently active image.
```

C.17 Image Properties

Sometimes it is necessary to pass results from one plugin to another, but the `run()` method itself does not provide a return value. One solution is to deposit the results of a plugin as a *property* in the corresponding `ImagePlus` object. A property consists of a *key/value* pair, where *key* is a string and *value* may be any Java object. In `ImageJ`, this mechanism is implemented as a hash table and supported by the following methods.

Program C.6

Working with a nonrectangular region of interest (ROI). This example shows the use of a mask image for processing nonrectangular ROIs. Objects of class `ImageProcessor` always return a valid bounding box (`Rectangle`), whether an ROI is selected or not (line 15). If no ROI is selected, the resulting rectangle covers the full image. `ImageProcessor` only returns a mask image (line 16) if the specified ROI is nonrectangular (e.g., `OvalRoi`, `PolygonRoi`); otherwise `null` is returned. The processing loop (lines 29–30) only scans over the bounding rectangle of the ROI. Inside this loop (line 31), the mask image is used to determine if pixels are inside the ROI or not. Only the pixels inside the ROI are modified (see Fig. C.8 for example images).

```
1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4 import java.awt.Rectangle;
5
6 public class Roi_Demo implements PlugInFilter {
7     boolean showMask = true;
8
9     public int setup(String arg, ImagePlus imp) {
10         return DOES_RGB;
11     }
12
13     public void run(ImageProcessor ip) {
14
15         Rectangle roi = ip.getRoi();
16         ImageProcessor mask = ip.getMask();
17         boolean hasMask = (mask != null);
18         if (hasMask && showMask) {
19             (new ImagePlus("The Mask", mask)).show();
20         }
21
22         // ROI corner coordinates:
23         int rLeft = roi.x;
24         int rTop = roi.y;
25         int rRight = rLeft + roi.width;
26         int rBottom = rTop + roi.height;
27
28         // process all pixels inside the ROI
29         for (int v = rTop; v < rBottom; v++) {
30             for (int u = rLeft; u < rRight; u++) {
31                 if (!hasMask || mask.getPixel(u-rLeft, v-rTop) > 0) {
32                     int p = ip.getPixel(u, v);
33                     ip.putPixel(u, v, ~p); // invert pixel values
34                 }
35             }
36         }
37     }
38
39 } // end of class Roi_Demo
```

C.17.1 ImagePlus (Class)

`java.util.Properties getProperties ()`

Returns the `Properties` object (a hash table) with all property entries of this image, or `null` if the image has no properties.

`Object getProperty (String key)`

Returns the property value associated with *key*, or `null` if no such property exists.

```
void setProperty (String key, Object value)
```

Adds a property with name *key* and content *value* to this image's properties. If a property with the same key already exists for this image, it is replaced by the new value. If *value* is null, the corresponding property is deleted.

Example

Program C.7 shows a simple example for the use of properties involving two ImageJ plugins. The first plugin (**Plugin_1**) computes the histogram of the image and inserts the result as a property with the key **HISTOGRAM** (line 17). The second plugin (**Plugin_2**) uses the same key to retrieve the histogram from the image's properties (line 36) for further processing. In this example, the common key is made available through the static variable **HistKey**, defined in class **Plugin_1** (line 35).

C.18 User Interaction

C.18.1 IJ (Class)

Text output, logging

```
static void error (String msg)
```

Displays the message *msg* in a dialog box titled "Image".

```
static void error (String title, String msg)
```

Displays the message *msg* in a dialog box with the specified title.

```
static void log (String msg)
```

Displays a line of text (*msg*) in ImageJ's "Log" window.

```
static void write (String msg)
```

Writes a line of text (*msg*) in ImageJ's "Results" window.

Dialog boxes

```
static double getNumber (String prompt, double defVal)
```

Allows the user to enter a number in a dialog box.

```
static String getString (String prompt, double defStr)
```

Allows the user to enter a string in a dialog box.

```
static void noImage ()
```

Displays a "no images are open" dialog box.

```
static void showMessage (String msg)
```

Displays a message in a dialog box titled "Message".

```
static void showMessage (String title, String msg)
```

Displays a message in a dialog box with the specified title.

```
static boolean showMessageWithCancel (String title,  
String msg)
```

Displays a message in a dialog box with the specified title. Returns false if the user pressed "Cancel".

Program C.7

Use of image properties (example). Properties can be used to pass results from one plugin to another. Here, the first plugin (`Plugin_1`) computes the histogram of the image in its `run()` method and attaches the result as a property to the `ImagePlus` object `im` (line 17). The second plugin retrieves the histogram from the image (line 36) for further processing. Note that the typecast to `int[]` in line 36 is potentially dangerous and should not be used without additional measures.

`Plugin_1.java`:

```
1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4
5 public class Plugin_1 implements PlugInFilter {
6     ImagePlus im;
7     public static final String HistKey = "HISTOGRAM";
8
9     public int setup(String arg, ImagePlus im) {
10         this.im = im;
11         return DOES_ALL + NO_CHANGES;
12     }
13
14     public void run(ImageProcessor ip) {
15         int[] hist = ip.getHistogram();
16         // add histogram to image properties:
17         im.setProperty(HistKey, hist);
18     }
19
20 } // end of class Plugin_1
```

`Plugin_2.java`:

```
21 import ij.IJ;
22 import ij.ImagePlus;
23 import ij.plugin.filter.PlugInFilter;
24 import ij.process.ImageProcessor;
25
26 public class Plugin_2 implements PlugInFilter {
27     ImagePlus im;
28
29     public int setup(String arg, ImagePlus im) {
30         this.im = im;
31         return DOES_ALL;
32     }
33
34     public void run(ImageProcessor ip) {
35         String key = Plugin1_.HistKey;
36         int[] hist = (int[]) im.getProperty(key);
37         if (hist == null){
38             IJ.error("This image has no histogram");
39         }
40         else {
41             // process histogram ...
42         }
43     }
44
45 } // end of class Plugin_2
```

Progress and status bar

static void showProgress (double *progress*)

Updates the progress bar in ImageJ's main window, where $0 \leq \textit{progress} < 1$. The length of the displayed bar is *progress* times its maximum length. The progress bar is not displayed if the time between the first and second calls to this method is less than 30 milliseconds. The bar is erased if *progress* ≥ 1 .

static void showProgress (int *i*, int *n*)

Updates the progress bar in ImageJ's main window, where $0 \leq i < n$ is the current index and *n* is the maximum index. The length of the displayed bar is (*i/n*) times its maximum length. The bar is erased if *i* $\geq n$.

static void showStatus (String *msg*)

Displays a message in the ImageJ status bar.

Keyboard queries

static boolean altKeyDown ()

Returns true if the alt key is down.

static boolean escapePressed ()

Returns true if the **esc** key was pressed since the last ImageJ command started to execute or since **resetEscape()** was called.

static void resetEscape ()

This method sets the **esc** key to the “up” position.

static boolean shiftKeyDown ()

Returns true if the shift key is down.

static boolean spaceBarDown ()

Returns true if the space bar is down.

Miscellaneous

static void beep ()

Emits a beep signal.

static ImagePlus getImage ()

Returns a reference to the *active* image, i. e., the ImagePlus object currently selected by the user.

static void wait (int *msecs*)

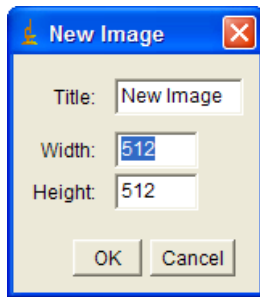
Waits (suspends processing) for *msecs* milliseconds.

C.18.2 GenericDialog (Class)

The class **GenericDialog** offers a simple mechanism for creating dialog windows containing multiple fields of various types. The layout of these dialog windows is created automatically. A small example and the corresponding results are shown in Prog. C.8. Other examples can be found in Secs. 5.8.5 and C.15.3. For additional details, see the ImageJ online documentation and the tutorial in [4].

Program C.8

Use of the `GenericDialog` class (example). This ImageJ plugin creates a new image with the title and size specified interactively by the user. The corresponding dialog window is shown below.



```
1 import ij.ImagePlus;
2 import ij.gui.GenericDialog;
3 import ij.gui.NewImage;
4 import ij.plugin.PlugIn;
5
6 public class Generic_Dialog_Example implements PlugIn {
7     static String title = "New Image";
8     static int width = 512;
9     static int height = 512;
10
11     public void run(String arg) {
12         GenericDialog gd = new GenericDialog("New Image");
13         gd.addStringField("Title:", title);
14         gd.addNumericField("Width:", width, 0);
15         gd.addNumericField("Height:", height, 0);
16         gd.showDialog();
17         if (gd.wasCanceled())
18             return;
19         title = gd.getNextString();
20         width = (int) gd.getNextNumber();
21         height = (int) gd.getNextNumber();
22
23         ImagePlus imp = NewImage.createByteImage(
24             title, width, height, 1, NewImage.FILL_WHITE);
25         imp.show();
26     }
27
28 } // end of class Generic_Dialog_Example
```

C.19 Plugins

ImageJ plugins come in two different variants, both of which are implemented as Java “interfaces”:

- **PlugIn**: can be applied without any image and is used to acquire images, display windows, etc.
- **PlugInFilter**: is applied to process an existing image.

C.19.1 PlugIn (Interface)

The **PlugIn** interface only specifies the implementation of a single method:

```
void run (String arg)
    Starts this plugin, where arg is used to specify options (may be an empty string).
```

The `PlugInFilter` interface requires the implementation of the following two methods:

```
void run (ImageProcessor ip)
```

Starts this plugin. The parameter *ip* specifies the image (`ImageProcessor` object) to which this plugin is applied.

```
int setup (String arg, ImagePlus im)
```

When the plugin is applied, the `setup()` method is called before the `run()` method. The parameter *im* refers to the target image (`ImagePlus` object) and *not* its `ImageProcessor`! If access to *im* is required within the `run()` method, it is usually assigned to a suitable object variable of this plugin by the `setup()` method (see, e.g., Prog. C.3). Note that the plugin's `setup()` method is called even when no images are open—in this case `null` is passed instead of the currently active image! The return value of the `setup()` method is a 32-bit (`int`) pattern, where each bit is a flag that corresponds to a certain feature of that plugin. Different flags can be easily combined by summing predefined constants, as listed below. The `run()` method of the plugin is not invoked if `setup()` returns `DONE`.

The following *return flags* for the `setup()` method are defined as `int` constants in the class `PlugInFilter`:

```
DOES_8G
```

This plugin accepts (unsigned) 8-bit grayscale images.

```
DOES_8C
```

This plugin accepts 8-bit indexed color images.

```
DOES_16
```

This plugin accepts (unsigned) 16-bit grayscale images.

```
DOES_32
```

This plugin accepts 32-bit `float` images.

```
DOES_RGB
```

This plugin accepts 3×8 bit RGB color images.

```
DOES_ALL
```

This plugin accepts any type of image (`DOES_ALL = DOES_8G + DOES_8C + DOES_16 + DOES_32 + DOES_RGB`).

```
DOES_STACKS
```

The plugin's `run` method shall be applied to all slices of a stack.

```
DONE
```

The plugin's `run` method shall not be invoked.

```
NO_CHANGES
```

The plugin does not modify the original image.

```
NO_IMAGE_REQUIRED
```

This plugin does not require that an image be open. In this case, `null` is passed to the `run()` method as the argument for *ip*.

NO_UNDO

This plugin does not require undo.

ROI_REQUIRED

This plugin requires a region of interest (ROI) to be explicitly specified.

STACK_REQUIRED

This plugin requires a stack of images.

SUPPORTS_MASKING

For nonrectangular ROIs, this plugin wants ImageJ to automatically restore that part of the image that is inside the bounding rectangle but outside of the ROI. This greatly simplifies the use of nonrectangular ROIs.

The flags above are integer values, each with only a single bit set (1) and the remaining bits being zero. Flags can be combined either by a bitwise OR operation (e. g., `DOES_8G | DOES_16`) or by simple arithmetic addition. For example, the `setup()` method for a `PlugInFilter` that can handle 8- *and* 16-bit grayscale images *and* does not modify the original image could be defined as follows:

```
public int setup (String arg, ImagePlus im) {  
    return DOES_8G + DOES_16G + NO_CHANGES;  
}
```

C.19.3 Executing Plugins: `IJ` (Class)

`static Object runPlugIn (String className, String arg)`

Creates a new plugin object of class *className* and executes its `run()` method, passing the string argument *arg*. If the plugin is of type `PlugInFilter`, the new instance is applied to the currently active image by first invoking the `setup()` method. The `runPlugIn()` method returns a reference to the new plugin object.

C.20 Window Management

C.20.1 WindowManager (Class)

The class `ij.WindowManager` defines a set of static methods for manipulating the screen windows in ImageJ:

```
static boolean closeAllWindows ()
    Closes all windows and returns true if successful. Stops and re-
    turns false if the “save changes” dialog is canceled for any unsaved
    image.

static ImagePlus getCurrentImage ()
    Returns the currently active image of type ImagePlus.

static ImageWindow getCurrentWindow ()
    Returns the currently active window of type ImageWindow.

static int[] getIDList ()
    Returns an array containing the IDs of all open images, or null
    if no image is open. The image IDs are negative integer values.

static ImagePlus getImage (int imageID)
    For imageID less than zero, this method returns the ImagePlus
    object with the specified imageID. It returns null if either
    imageID is zero, no open image has a matching ID, or no images
    are open at all. For imageID greater than zero, it returns the
    image at the corresponding position in the image array delivered
    by getIDList().

static ImagePlus getImage (String title)
    Returns the first image that has the specified title or null if no
    such image is found.

static int getImageCount ()
    Returns the number of open images.

static ImagePlus getTempCurrentImage ()
    Returns the image temporarily made current (by setTempCur-
    rentImage()), which may be null.

static int getWindowCount ()
    Returns the number of open image windows.

static void putBehind ()
    Moves the current active image to the back and activates the next
    image in a cyclic fashion.

static void repaintImageWindows ()
    Repaints all open image windows.

static void setCurrentWindow (ImageWindow win)
    Makes the specified image active.

static void setTempCurrentImage (ImagePlus im)
    Makes im temporarily the active image and thus allows processing
    of images that are currently not displayed in a window. Another
    call with the argument null reverts to the previously active image.
```


C.21 Additional Functions

C.21.1 ImagePlus (Class)

Locking and unlocking images

ImageJ plugins may execute simultaneously as different Java threads in the same runtime environment. Locking may be required to avoid mutual interferences between plugins that operate on the same image.

boolean lock ()

Locks this image so other threads can test to see if it is in use. Returns **true** if the image was successfully locked. Beeps, displays a message in the status bar, and returns **false** if the image is already locked.

boolean lockSilently ()

Similar to **lock** but does not beep. Displays an error message if the attempt to lock the image fails.

void unlock ()

Unlocks this image.

Internal clipboard

ImageJ maintains a single internal clipboard image (as an **ImagePlus** object) that can be manipulated interactively with the **Edit** menu or accessed through the following methods:

void copy (boolean cut)

Copies the contents of the current selection (region of interest) to the internal clipboard. The entire image is copied if there is no selection. The selected part of the image is *cleared* (i.e., filled with the current background value or color) if **cut** is **true**.

void paste ()

Inserts the contents of the internal clipboard into this (**ImagePlus**) image. If the target image has a selection the same size as the image on the clipboard, the clipboard content is inserted into that selection, otherwise the clipboard content is inserted into the center of the image.

static ImagePlus getClipboard ()

Returns the internal clipboard (as an **ImagePlus** object) or **null** if the internal clipboard is empty. Note that this is a *static* method and is thus called in the form **ImagePlus.getClipboard()**.

File information

FileInfo getFileInfo ()

Returns a **FileInfo** object containing information, including the pixel array, needed to *save* this image. Use **getOriginalFileInfo()** to get a copy of the **FileInfo** object used to *open* the image.

`FileInfo getOriginalFileInfo ()`

Returns the `FileInfo` object that was used to *open* this image. This includes fields such as `fileName (String)`, `directory (String)`, and `description (String)`. Returns `null` for images created internally or using the `File→New` command.

C.21.2 IJ (Class)

Directory information

`static String getDirectory (String target)`

Returns the path to ImageJ's *home*, *startup*, *plugins*, *macros*, *temp*, or *image* directory, depending on the value of *target* ("home", "startup", "plugins", "macros", "temp", or "image"). If *target* (which may not be `null`) is none of the above, the method displays a dialog and returns the path to the directory selected by the user. `null` is returned if the specified directory is not found or the user cancels the dialog.

Memory management

`static long currentMemory ()`

Returns the amount of memory (in bytes) currently being used by ImageJ.

`static String freeMemory ()`

Runs the garbage collector and returns a string showing how much of the available memory is in use.

`static long maxMemory ()`

Returns the maximum amount of memory available to ImageJ or zero if ImageJ is unable to determine this limit.

System information

`static String getVersion ()`

Returns ImageJ's version number as a string.

`static boolean isJava2 ()`

Returns `true` if ImageJ is running on Java 2.

`static boolean isJava14 ()`

Returns `true` if ImageJ is running on a Java 1.4 or greater JVM.

`static boolean isMacintosh ()`

Returns `true` if the current platform is a Macintosh computer.

`static boolean isMacOSX ()`

Returns `true` if the current platform is a Macintosh computer running OS X.

`static boolean isWindows ()`

Returns `true` if this machine is running Windows.

`static boolean versionLessThan (String version)`

Displays an error message and returns `false` if the current version of ImageJ is less than the one specified.

Appendix D

Source Code

D.1 Harris Corner Detector

The following Java source code represents a complete implementation of the Harris corner detector, as described in Ch. 8. It consists of the following classes (files):

- **Harris_Corner_Plugin**: a sample ImageJ plugin that demonstrates the use of the corner detector.
- **Corner** (p. 527): a class representing an individual corner object.
- **HarrisCornerDetector** (p. 527): the actual corner detector. This class is instantiated to create a corner detector for a given image.

D.1.1 Harris_Corner_Plugin (Class)

```
1 import harris.HarrisCornerDetector;
2 import ij.IJ;
3 import ij.ImagePlus;
4 import ij.gui.GenericDialog;
5 import ij.plugin.filter.PlugInFilter;
6 import ij.process.ImageProcessor;
7
8 public class Harris_Corner_Plugin implements PlugInFilter {
9     ImagePlus im;
10     static float alpha = HarrisCornerDetector.DEFAULT_ALPHA;
11     static int threshold = HarrisCornerDetector.
        DEFAULT_THRESHOLD;
12     static int nmax = 0; //points to show
13
14     public int setup(String arg, ImagePlus im) {
15         this.im = im;
```

Appendix D
SOURCE CODE

```
16     if (arg.equals("about")) {
17         showAbout();
18         return DONE;
19     }
20     return DOES_8G + NO_CHANGES;
21 }
22
23 public void run(ImageProcessor ip) {
24     if (!showDialog()) return; //dialog canceled or error
25     HarrisCornerDetector hcd =
26         new HarrisCornerDetector(ip,alpha,threshold);
27     hcd.findCorners();
28     ImageProcessor result = hcd.showCornerPoints(ip);
29     ImagePlus win =
30         new ImagePlus("Corners from " + im.getTitle(),
31             result);
32     win.show();
33 }
34
35 void showAbout() {
36     String cn = getClass().getName();
37     IJ.showMessage("About "+cn+" ...",
38         "Harris Corner Detector");
39 }
40
41 private boolean showDialog() {
42     // display dialog, and return false if canceled or in error.
43     GenericDialog dlg = new GenericDialog("Harris Corner
44         Detector", IJ.getInstance());
45     float def_alpha = HarrisCornerDetector.DEFAULT_ALPHA;
46     dlg.addNumericField("Alpha (default: "+def_alpha+")",
47         alpha, 3);
48     int def_threshold = HarrisCornerDetector.
49         DEFAULT_THRESHOLD;
50     dlg.addNumericField("Threshold (default: "+def_threshold+
51         ")", threshold, 0);
52     dlg.addNumericField("Max. points (0 = show all)", nmax,
53         0);
54     dlg.showDialog();
55     if(dlg.wasCanceled())
56         return false;
57     if(dlg.invalidNumber()) {
58         IJ.showMessage("Error", "Invalid input number");
59         return false;
60     }
61     alpha = (float) dlg.getNextNumber();
62     threshold = (int) dlg.getNextNumber();
63     nmax = (int) dlg.getNextNumber();
64     return true;
65 }
66 } // end of class Harris_Corner_Plugin
```

```
1 package harris;
2 import ij.process.ImageProcessor;
3
4 class Corner implements Comparable {
5     int u;
6     int v;
7     float q;
8
9     Corner (int u, int v, float q) {
10         this.u = u;
11         this.v = v;
12         this.q = q;
13     }
14
15     public int compareTo (Object obj) {
16         // used for sorting corners by corner strength q
17         Corner c2 = (Corner) obj;
18         if (this.q > c2.q) return -1;
19         if (this.q < c2.q) return 1;
20         else return 0;
21     }
22
23     double dist2 (Corner c2) {
24         // returns the squared distance between this corner and corner c2
25         int dx = this.u - c2.u;
26         int dy = this.v - c2.v;
27         return (dx*dx)+(dy*dy);
28     }
29
30     void draw(ImageProcessor ip) {
31         // draw this corner as a black cross in ip
32         int paintvalue = 0; // black
33         int size = 2;
34         ip.setValue(paintvalue);
35         ip.drawLine(u-size,v,u+size,v);
36         ip.drawLine(u,v-size,u,v+size);
37     }
38
39 } // end of class Corner
```

D.1.3 File HarrisCornerDetector (Class)

```
1 package harris;
2 import ij.IJ;
3 import ij.ImagePlus;
4 import ij.plugin.filter.Convolver;
5 import ij.process.Blitter;
6 import ij.process.ByteProcessor;
```

Appendix D

SOURCE CODE

```
7 import ij.process.FloatProcessor;
8 import ij.process.ImageProcessor;
9 import java.util.Arrays;
10 import java.util.Collections;
11 import java.util.List;
12 import java.util.Vector;
13
14 public class HarrisCornerDetector {
15
16     public static final float DEFAULT_ALPHA = 0.050f;
17     public static final int DEFAULT_THRESHOLD = 20000;
18     float alpha = DEFAULT_ALPHA;
19     int threshold = DEFAULT_THRESHOLD;
20     double dmin = 10;
21     final int border = 20;
22
23     // filter kernels (1D part of separable 2D filters)
24     final float[] pfilt = {0.223755f, 0.552490f, 0.223755f};
25     final float[] dfilt = {0.453014f, 0.0f, -0.453014f};
26     final float[] bfilt = {0.01563f, 0.09375f, 0.234375f, 0.3125f,
27         , 0.234375f, 0.09375f, 0.01563f};
28     // = [1, 6, 15, 20, 15, 6, 1]/64
29     ImageProcessor ipOrig;
30     FloatProcessor A;
31     FloatProcessor B;
32     FloatProcessor C;
33     FloatProcessor Q;
34     List<Corner> corners;
35
36     HarrisCornerDetector(ImageProcessor ip) {
37         this.ipOrig = ip;
38     }
39
40     public HarrisCornerDetector(ImageProcessor ip,
41         float alpha, int threshold)
42     {
43         this.ipOrig = ip;
44         this.alpha = alpha;
45         this.threshold = threshold;
46     }
47
48     public void findCorners() {
49         makeDerivatives();
50         makeCrf(); //corner response function (CRF)
51         corners = collectCorners(border);
52         corners = cleanupCorners(corners);
53     }
54
55     void makeDerivatives() {
56         FloatProcessor Ix =
57             (FloatProcessor) ipOrig.convertToFloat();
```

```

57     FloatProcessor Iy =
58         (FloatProcessor) ipOrig.convertToFloat();
59
60     Ix = convolve1h(convolve1h(Ix,pfilt),dfilt);
61     Iy = convolve1v(convolve1v(Iy,pfilt),dfilt);
62
63     A = sqr((FloatProcessor) Ix.duplicate());
64     A = convolve2(A,bfilt);
65
66     B = sqr((FloatProcessor) Iy.duplicate());
67     B = convolve2(B,bfilt);
68
69     C = mult((FloatProcessor)Ix.duplicate(),Iy);
70     C = convolve2(C,bfilt);
71 }
72
73 void makeCrf() { // corner response function (CRF)
74     int w = ipOrig.getWidth();
75     int h = ipOrig.getHeight();
76     Q = new FloatProcessor(w,h);
77     float[] Apix = (float[]) A.getPixels();
78     float[] Bpix = (float[]) B.getPixels();
79     float[] Cpix = (float[]) C.getPixels();
80     float[] Qpix = (float[]) Q.getPixels();
81     for (int v=0; v<h; v++) {
82         for (int u=0; u<w; u++) {
83             int i = v*w+u;
84             float a = Apix[i], b = Bpix[i], c = Cpix[i];
85             float det = a*b-c*c;
86             float trace = a+b;
87             Qpix[i] = det - alpha * (trace * trace);
88         }
89     }
90 }
91
92 List<Corner> collectCorners(int border) {
93     List<Corner> cornerList = new Vector<Corner>(1000);
94     int w = Q.getWidth();
95     int h = Q.getHeight();
96     float[] Qpix = (float[]) Q.getPixels();
97     for (int v=border; v<h-border; v++){
98         for (int u=border; u<w-border; u++) {
99             float q = Qpix[v*w+u];
100             if (q>threshold && isLocalMax(Q,u,v)) {
101                 Corner c = new Corner(u,v,q);
102                 cornerList.add(c);
103             }
104         }
105     }
106     Collections.sort(cornerList);
107     return cornerList;

```

Appendix D
SOURCE CODE

```
108 }
109
110 List<Corner> cleanupCorners(List<Corner> corners) {
111     double dmin2 = dmin*dmin;
112     Corner[] cornerArray = new Corner[corners.size()];
113     cornerArray = corners.toArray(cornerArray);
114     List<Corner> goodCorners =
115         new Vector<Corner>(corners.size());
116     for (int i=0; i<cornerArray.length; i++){
117         if (cornerArray[i] != null){
118             Corner c1 = cornerArray[i];
119             goodCorners.add(c1);
120             // delete all remaining corners close to c
121             for (int j=i+1; j<cornerArray.length; j++){
122                 if (cornerArray[j] != null){
123                     Corner c2 = cornerArray[j];
124                     if (c1.dist2(c2)<dmin2)
125                         cornerArray[j] = null; //delete corner
126                 }
127             }
128         }
129     }
130     return goodCorners;
131 }
132
133 void printCornerPoints(List<Corner> crf) {
134     int i = 0;
135     for (Corner ipt: crf){
136         IJ.write((i++) + ": " + (int)ipt.q + " " + ipt.u + " " +
137             ipt.v);
138     }
139 }
140
141 public ImageProcessor showCornerPoints(ImageProcessor ip) {
142     ByteProcessor ipResult = (ByteProcessor)ip.duplicate();
143     // change background image contrast and brightness
144     int[] lookupTable = new int[256];
145     for (int i=0; i<256; i++){
146         lookupTable[i] = 128 + (i/2);
147     }
148     ipResult.applyTable(lookupTable);
149     // draw corners:
150     for (Corner c: corners) {
151         c.draw(ipResult);
152     }
153     return ipResult;
154 }
155
156 void showProcessor(ImageProcessor ip, String title) {
157     ImagePlus win = new ImagePlus(title,ip);
158     win.show();
159 }
```

```

158 }
159
160 // utility methods for float processors —
161
162 static FloatProcessor convolve1h
163     (FloatProcessor p, float[] h) {
164     Convolver conv = new Convolver();
165     conv.setNormalize(false);
166     conv.convolve(p, h, 1, h.length);
167     return p;
168 }
169
170 static FloatProcessor convolve1v
171     (FloatProcessor p, float[] h) {
172     Convolver conv = new Convolver();
173     conv.setNormalize(false);
174     conv.convolve(p, h, h.length, 1);
175     return p;
176 }
177
178 static FloatProcessor convolve2
179     (FloatProcessor p, float[] h) {
180     convolve1h(p,h);
181     convolve1v(p,h);
182     return p;
183 }
184
185 static FloatProcessor sqr (FloatProcessor fp1) {
186     fp1.sqr();
187     return fp1;
188 }
189
190 static FloatProcessor mult (FloatProcessor fp1,
191     FloatProcessor fp2) {
192     int mode = Blitter.MULTIPLY;
193     fp1.copyBits(fp2, 0, 0, mode);
194     return fp1;
195 }
196
197 static boolean isLocalMax (FloatProcessor fp,int u,int v) {
198     int w = fp.getWidth();
199     int h = fp.getHeight();
200     if (u<=0 || u>=w-1 || v<=0 || v>=h-1)
201         return false;
202     else {
203         float[] pix = (float[]) fp.getPixels();
204         int i0 = (v-1)*w+u, i1 = v*w+u, i2 = (v+1)*w+u;
205         float cp = pix[i1];
206         return
207             cp > pix[i0-1] && cp > pix[i0] && cp > pix[i0+1] &&
208             cp > pix[i1-1] && cp > pix[i1+1] &&

```

```
208         cp > pix[i2-1] && cp > pix[i2] && cp > pix[i2+1] ;
209     }
210 }
211
212 } // end of class HarrisCornerDetector
```

D.2 Combined Region Labeling and Contour Tracing

The following Java source code represents a complete implementation of the combined region labeling and contour tracing algorithm described in Sec. 11.2. It consists of the following classes (files):

- **Contour_Tracing_Plugin**: a sample ImageJ plugin that demonstrates the use of this region labeling implementation.
- **Contour** (p. 533): a class representing a contour object.
- **BinaryRegion** (p. 535): a class representing a binary region object.
- **ContourTracer** (p. 536): the actual region labeler and contour tracer. This class is instantiated to create a region labeler for a given image.
- **ContourOverlay** (p. 541): a class for displaying contours as vector graphics on top of images.

D.2.1 Contour_Tracing_Plugin (Class)

```
1 import java.util.List;
2 import regions.BinaryRegion;
3 import regions.RegionLabeling;
4 import contours.Contour;
5 import contours.ContourOverlay;
6 import contours.ContourTracer;
7
8 import ij.IJ;
9 import ij.ImagePlus;
10 import ij.gui.ImageWindow;
11 import ij.plugin.filter.PlugInFilter;
12 import ij.process.ImageProcessor;
13
14 // This plugin implements the combined contour tracing and
15 // component labeling algorithm as described in [22].
16 // It uses the ContourTracer class to create lists of points
17 // representing the internal and external contours of each region in
18 // the binary image. Instead of drawing directly into the image,
19 // we make use of ImageJ's ImageCanvas to draw the contours
20 // in a separate layer on top of the image. It illustrates how to use
21 // the Java2D API to draw the polygons and scale and transform
22 // them to match ImageJ's zooming.
23
24
```

```
25 public class Contour_Tracing_Plugin implements PlugInFilter
26 {
27     ImagePlus origImage = null;
28     String origTitle = null;
29     static boolean verbose = true;
30
31     public int setup(String arg, ImagePlus im) {
32         origImage = im;
33         origTitle = im.getTitle();
34         RegionLabeling.setVerbose(verbose);
35         return DOES_8G + NO_CHANGES;
36     }
37
38     public void run(ImageProcessor ip) {
39         ImageProcessor ip2 = ip.duplicate();
40
41         // label regions and trace contours
42         ContourTracer tracer = new ContourTracer(ip2);
43
44         // extract contours and regions
45         List<Contour> outerContours = tracer.getOuterContours();
46         List<Contour> innerContours = tracer.getInnerContours();
47         List<BinaryRegion> regions = tracer.getRegions();
48         if (verbose) printRegions(regions);
49
50         // change lookup table to show gray regions
51         ip2.setMinAndMax(0,512);
52         // create an image with overlay to show the contours
53         ImagePlus im2 = new ImagePlus("Contours of " + origTitle,
54             ip2);
55         ContourOverlay cc = new ContourOverlay(im2, outerContours,
56             innerContours);
57         new ImageWindow(im2, cc);
58     }
59
60     void printRegions(List<BinaryRegion> regions) {
61         for (BinaryRegion r: regions) {
62             IJ.write("" + r);
63         }
64     }
65 } // end of class Contour_Tracing_Plugin
```

D.2.2 Contour (Class)

```
1 package contours;
2 import ij.IJ;
3 import java.awt.Point;
4 import java.awt.Polygon;
5 import java.awt.Shape;
```

```
6 import java.awt.geom.Ellipse2D;
7 import java.util.ArrayList;
8 import java.util.Iterator;
9 import java.util.List;
10
11 public class Contour {
12     static int INITIAL_SIZE = 50;
13     int label;
14     List<Point> points;
15
16     Contour (int label, int size) {
17         this.label = label;
18         points = new ArrayList<Point>(size);
19     }
20
21     Contour (int label) {
22         this.label = label;
23         points = new ArrayList<Point>(INITIAL_SIZE);
24     }
25
26     void addPoint (Point n) {
27         points.add(n);
28     }
29
30     Shape makePolygon() {
31         int m = points.size();
32         if (m>1) {
33             int[] xPoints = new int[m];
34             int[] yPoints = new int[m];
35             int k = 0;
36             Iterator<Point> itr = points.iterator();
37             while (itr.hasNext() && k < m) {
38                 Point cpt = itr.next();
39                 xPoints[k] = cpt.x;
40                 yPoints[k] = cpt.y;
41                 k = k + 1;
42             }
43             return new Polygon(xPoints, yPoints, m);
44         }
45         else { // use circles for isolated pixels
46             Point cpt = points.get(0);
47             return new Ellipse2D.Double
48                 (cpt.x-0.1, cpt.y-0.1, 0.2, 0.2);
49         }
50     }
51
52     static Shape[] makePolygons(List<Contour> contours) {
53         if (contours == null)
54             return null;
55         else {
56             Shape[] pa = new Shape[contours.size()];
```

```
57     int i = 0;
58     for (Contour c: contours) {
59         pa[i] = c.makePolygon();
60         i = i + 1;
61     }
62     return pa;
63 }
64 }
65
66 void moveBy (int dx, int dy) {
67     for (Point pt: points) {
68         pt.translate(dx,dy);
69     }
70 }
71
72 static void moveContoursBy
73     (List<Contour> contours, int dx, int dy) {
74     for (Contour c: contours) {
75         c.moveBy(dx, dy);
76     }
77 }
78
79 } // end of class Contour
```

D.2.3 BinaryRegion (Class)

```
1 package regions;
2 import java.awt.Rectangle;
3 import java.awt.geom.Point2D;
4
5 public class BinaryRegion {
6     int label;
7     int numberOfPixels = 0;
8     double xc = Double.NaN;
9     double yc = Double.NaN;
10    int left = Integer.MAX_VALUE;
11    int right = -1;
12    int top = Integer.MAX_VALUE;
13    int bottom = -1;
14
15    int x_sum = 0;
16    int y_sum = 0;
17    int x2_sum = 0;
18    int y2_sum = 0;
19
20    public BinaryRegion(int id){
21        this.label = id;
22    }
23
24    public int getSize() {
```

```
25     return this.numberOfPixels;
26 }
27
28 public Rectangle getBoundingBox() {
29     if (left == Integer.MAX_VALUE)
30         return null;
31     else
32         return new Rectangle
33             (left, top, right-left+1, bottom-top+1);
34 }
35
36 public Point2D.Double getCenter(){
37     if (Double.isNaN(xc))
38         return null;
39     else
40         return new Point2D.Double(xc, yc);
41 }
42
43 public void addPixel(int x, int y){
44     numberOfPixels = numberOfPixels + 1;
45     x_sum = x_sum + x;
46     y_sum = y_sum + y;
47     x2_sum = x2_sum + x*x;
48     y2_sum = y2_sum + y*y;
49     if (x<left) left = x;
50     if (y<top) top = y;
51     if (x>right) right = x;
52     if (y>bottom) bottom = y;
53 }
54
55 public void update(){
56     if (numberOfPixels > 0){
57         xc = x_sum / numberOfPixels;
58         yc = y_sum / numberOfPixels;
59     }
60 }
61
62 } // end of class BinaryRegion
```

D.2.4 ContourTracer (Class)

```
1 package contours;
2 import java.awt.Point;
3 import java.util.ArrayList;
4 import java.util.LinkedList;
5 import java.util.List;
6 import regions.BinaryRegion;
7 import ij.IJ;
8 import ij.process.ImageProcessor;
9
```

```
10 public class ContourTracer {
11     static final byte FOREGROUND = 1;
12     static final byte BACKGROUND = 0;
13     static boolean beVerbose = true;
14
15     List<Contour> outerContours = null;
16     List<Contour> innerContours = null;
17     List<BinaryRegion> allRegions = null;
18     int regionId = 0;
19
20     ImageProcessor ip = null;
21     int width;
22     int height;
23     byte[][] pixelArray;
24     int[][] labelArray;
25
26     // label values in labelArray can be:
27     // 0 ... unlabeled
28     // -1 ... previously visited background pixel
29     // > 0 ... a valid label
30
31     // constructor method
32     public ContourTracer (ImageProcessor ip) {
33         this.ip = ip;
34         this.width = ip.getWidth();
35         this.height = ip.getHeight();
36         makeAuxArrays();
37         findAllContours();
38         collectRegions();
39     }
40
41     public static void setVerbose(boolean verbose) {
42         beVerbose = verbose;
43     }
44
45     public List<Contour> getOuterContours() {
46         return outerContours;
47     }
48
49     public List<Contour> getInnerContours() {
50         return innerContours;
51     }
52
53     public List<BinaryRegion> getRegions() {
54         return allRegions;
55     }
56
57     // nonpublic methods
58
59     void makeAuxArrays() {
60         int h = ip.getHeight();
```

Appendix D
SOURCE CODE

```
61  int w = ip.getWidth();
62  pixelArray = new byte[h+2][w+2];
63  labelArray = new int[h+2][w+2];
64  // initialize auxiliary arrays
65  for (int v = 0; v < h+2; v++) {
66      for (int u = 0; u < w+2; u++) {
67          if (ip.get(u-1,v-1) == 0)
68              pixelArray[v][u] = BACKGROUND;
69          else
70              pixelArray[v][u] = FOREGROUND;
71      }
72  }
73 }
74
75 Contour traceOuterContour (int cx, int cy, int label) {
76     Contour cont = new Contour(label);
77     traceContour(cx, cy, label, 0, cont);
78     return cont;
79 }
80
81 Contour traceInnerContour(int cx, int cy, int label) {
82     Contour cont = new Contour(label);
83     traceContour(cx, cy, label, 1, cont);
84     return cont;
85 }
86
87 // trace one contour starting at (xS,yS) in direction dS
88 Contour traceContour (int xS, int yS, int label, int dS,
89     Contour cont) {
89     int xT, yT; // T = successor of starting point (xS,yS)
90     int xP, yP; // P = previous contour point
91     int xC, yC; // C = current contour point
92     Point pt = new Point(xS, yS);
93     int dNext = findNextPoint(pt, dS);
94     cont.addPoint(pt);
95     xP = xS; yP = yS;
96     xC = xT = pt.x;
97     yC = yT = pt.y;
98
99     boolean done = (xS==xT && yS==yT); // true if isolated pixel
100
101     while (!done) {
102         labelArray[yC][xC] = label;
103         pt = new Point(xC, yC);
104         int dSearch = (dNext + 6) % 8;
105         dNext = findNextPoint(pt, dSearch);
106         xP = xC; yP = yC;
107         xC = pt.x; yC = pt.y;
108         // are we back at the starting position?
109         done = (xP==xS && yP==yS && xC==xT && yC==yT);
110         if (!done) {
```

```
111     cont.addPoint(pt);
112     }
113 }
114 return cont;
115 }
116
117 int findNextPoint (Point pt, int dir) {
118     // starts at Point pt in direction dir, returns the
119     // final tracing direction, and modifies pt
120     final int[][] delta = {
121         { 1,0}, { 1, 1}, {0, 1}, {-1, 1},
122         {-1,0}, {-1,-1}, {0,-1}, { 1,-1}};
123     for (int i = 0; i < 7; i++) {
124         int x = pt.x + delta[dir][0];
125         int y = pt.y + delta[dir][1];
126         if (pixelArray[y][x] == BACKGROUND) {
127             // mark surrounding background pixels
128             labelArray[y][x] = -1;
129             dir = (dir + 1) % 8;
130         }
131         else { // found a nonbackground pixel
132             pt.x = x; pt.y = y;
133             break;
134         }
135     }
136     return dir;
137 }
138
139 void findAllContours() {
140     outerContours = new ArrayList<Contour>(50);
141     innerContours = new ArrayList<Contour>(50);
142     int label = 0; // current label
143
144     // scan top to bottom, left to right
145     for (int v = 1; v < pixelArray.length-1; v++) {
146         label = 0; // no label
147         for (int u = 1; u < pixelArray[v].length-1; u++) {
148
149             if (pixelArray[v][u] == FOREGROUND) {
150                 if (label != 0) { // keep using the same label
151                     labelArray[v][u] = label;
152                 }
153                 else {
154                     label = labelArray[v][u];
155                     if (label == 0) {
156                         // unlabeled—new outer contour
157                         regionId = regionId + 1;
158                         label = regionId;
159                         Contour oc = traceOuterContour(u, v, label);
160                         outerContours.add(oc);
161                         labelArray[v][u] = label;
```

```
162     }
163     }
164 }
165     else { // background pixel
166         if (label != 0) {
167             if (labelArray[v][u] == 0) {
168                 // unlabeled—new inner contour
169                 Contour ic = traceInnerContour(u-1, v, label);
170                 innerContours.add(ic);
171             }
172             label = 0;
173         }
174     }
175 }
176 }
177 // shift back to original coordinates
178 Contour.moveContoursBy (outerContours, -1, -1);
179 Contour.moveContoursBy (innerContours, -1, -1);
180 }
181
182
183 // creates a container of BinaryRegion objects
184 // collects the region pixels from the label image
185 // and computes the statistics for each region
186 void collectRegions() {
187     int maxLabel = this.regionId;
188     int startLabel = 1;
189     BinaryRegion[] regionArray =
190         new BinaryRegion[maxLabel + 1];
191     for (int i = startLabel; i <= maxLabel; i++) {
192         regionArray[i] = new BinaryRegion(i);
193     }
194     for (int v = 0; v < height; v++) {
195         for (int u = 0; u < width; u++) {
196             int lb = labelArray[v][u];
197             if (lb >= startLabel && lb <= maxLabel
198                 && regionArray[lb] != null) {
199                 regionArray[lb].addPixel(u, v);
200             }
201         }
202     }
203
204     // create a list of regions to return, collect nonempty regions
205     List<BinaryRegion> regionList =
206         new LinkedList<BinaryRegion>();
207     for (BinaryRegion r: regionArray) {
208         if (r != null && r.getSize() > 0) {
209             r.update(); // compute the statistics for this region
210             regionList.add(r);
211         }
212     }
}
```

```

213     allRegions = regionList;
214 }
215
216 } // end of class ContourTracer

```

D.2.5 ContourOverlay (Class)

```

1 package contours;
2 import ij.ImagePlus;
3 import ij.gui.ImageCanvas;
4 import java.awt.BasicStroke;
5 import java.awt.Color;
6 import java.awt.Graphics;
7 import java.awt.Graphics2D;
8 import java.awt.Polygon;
9 import java.awt.RenderingHints;
10 import java.awt.Shape;
11 import java.awt.Stroke;
12 import java.util.List;
13
14 public class ContourOverlay extends ImageCanvas {
15     private static final long serialVersionUID = 1L;
16     static float strokeWidth = 0.5f;
17     static int capsstyle = BasicStroke.CAP_ROUND;
18     static int joinstyle = BasicStroke.JOIN_ROUND;
19     static Color outerColor = Color.black;
20     static Color innerColor = Color.white;
21     static float[] outerDashing = {strokeWidth * 2.0f,
22         strokeWidth * 2.5f};
23     static float[] innerDashing = {strokeWidth * 0.5f,
24         strokeWidth * 2.5f};
25     static boolean DRAW_CONTOURS = true;
26
27     Shape[] outerContourShapes = null;
28     Shape[] innerContourShapes = null;
29
30     public ContourOverlay(ImagePlus im,
31         List<Contour> outerCs, List<Contour> innerCs)
32     {
33         super(im);
34         if (outerCs != null)
35             outerContourShapes = Contour.makePolygons(outerCs);
36         if (innerCs != null)
37             innerContourShapes = Contour.makePolygons(innerCs);
38     }
39
40     public void paint(Graphics g) {
41         super.paint(g);
42         drawContours(g);
43     }

```

```
42
43 // nonpublic methods
44
45 private void drawContours(Graphics g) {
46     Graphics2D g2d = (Graphics2D) g;
47     g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
48         RenderingHints.VALUE_ANTIALIAS_ON);
49
50     // scale and move overlay to the pixel centers
51     double mag = this.getMagnification();
52     g2d.scale(mag, mag);
53     g2d.translate(0.5-this.srcRect.x, 0.5-this.srcRect.y);
54
55     if (DRAW_CONTOURS) {
56         Stroke solidStroke = new BasicStroke
57             (strokeWidth, capsstyle, jointstyle);
58         Stroke dashedStrokeOuter = new BasicStroke
59             (strokeWidth, capsstyle, jointstyle, 1.0f,
60             outerDashing, 0.0f);
61         Stroke dashedStrokeInner = new BasicStroke
62             (strokeWidth, capsstyle, jointstyle, 1.0f,
63             innerDashing, 0.0f);
64
65         if (outerContourShapes != null)
66             drawShapes(outerContourShapes, g2d, solidStroke,
67                 dashedStrokeOuter, outerColor);
68         if (innerContourShapes != null)
69             drawShapes(innerContourShapes, g2d, solidStroke,
70                 dashedStrokeInner, innerColor);
71     }
72 }
73
74 void drawShapes(Shape[] shapes, Graphics2D g2d,
75     Stroke solidStrk, Stroke dashedStrk, Color col) {
76     g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
77         RenderingHints.VALUE_ANTIALIAS_ON);
78     g2d.setColor(col);
79     for (int i = 0; i < shapes.length; i++) {
80         Shape s = shapes[i];
81         if (s instanceof Polygon)
82             g2d.setStroke(dashedStrk);
83         else
84             g2d.setStroke(solidStrk);
85         g2d.draw(s);
86     }
87 }
88
89 } // end of class ContourOverlay
```

References

1. Adobe Systems. “Adobe RGB (1998) Color Space Specification” (2005). <http://www.adobe.com/digitalimag/pdfs/AdobeRGB1998.pdf>.
2. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN. “The Design and Analysis of Computer Algorithms”. Addison-Wesley, Reading, MA (1974).
3. K. ARNOLD, J. GOSLING, AND D. HOLMES. “The Java Programming Language”. Addison-Wesley, Reading, MA, fourth ed. (2005).
4. W. BAILER. “Writing ImageJ Plugins—A Tutorial” (2003). <http://www.fh-hagenberg.at/mtd/depot/imaging/imagej/>.
5. D. H. BALLARD AND C. M. BROWN. “Computer Vision”. Prentice Hall, Englewood Cliffs, NJ (1982).
6. C. B. BARBER, D. P. DOBKIN, AND H. HUHDANPAA. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software* **22**(4), 469–483 (1996).
7. H. G. BARROW, J. M. TENENBAUM, R. C. BOLLES, AND H. C. WOLF. Parametric correspondence and chamfer matching: two new techniques for image matching. In R. REDDY, editor, “Proceedings of the 5th International Joint Conference on Artificial Intelligence”, pp. 659–663, Cambridge, MA (1977). William Kaufmann, Los Altos, CA.
8. R. E. BLAHUT. “Fast Algorithms for Digital Signal Processing”. Addison-Wesley, Reading, MA (1985).
9. J. BLOCH. “Effective Java Programming Language Guide”. Addison-Wesley, Reading, MA (2001).
10. C. BOOR. “A Practical Guide to Splines”. Springer-Verlag, New York (2001).
11. G. BORGEFORS. Distance transformations in digital images. *Computer Vision, Graphics and Image Processing* **34**, 344–371 (1986).
12. G. BORGEFORS. Hierarchical chamfer matching: a parametric edge matching algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **10**(6), 849–865 (1988).
13. J. E. BRESENHAM. A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM* **20**(2), 100–106 (1977).
14. E. O. BRIGHAM. “The Fast Fourier Transform and Its Applications”. Prentice Hall, Englewood Cliffs, NJ (1988).
15. I. N. BRONSHTEN AND K. A. SEMENDYAYEV. “Handbook of Mathematics”. Springer-Verlag, Berlin, third ed. (2007).
16. H. BUNKE AND P. S. P. WANG, editors. “Handbook of Character Recognition and Document Image Analysis”. World Scientific, Singapore (2000).
17. W. BURGER AND M. J. BURGE. “Digitale Bildverarbeitung—Eine Einführung mit Java und ImageJ”. Springer-Verlag, Berlin, second ed. (2006).

18. P. J. BURT AND E. H. ADELSON. The Laplacian pyramid as a compact image code. *IEEE Transactions on Communications* **31**(4), 532–540 (1983).
19. J. F. CANNY. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **8**(6), 679–698 (1986).
20. K. R. CASTLEMAN. “Digital Image Processing”. Prentice Hall, Upper Saddle River, NJ (1995).
21. E. CATMULL AND R. ROM. A class of local interpolating splines. In R. E. BARNHILL AND R. F. RIESENFELD, editors, “Computer Aided Geometric Design”, pp. 317–326. Academic Press, New York (1974).
22. F. CHANG AND C. J. CHEN. A component-labeling algorithm using contour tracing technique. In “Proceedings of the Seventh International Conference on Document Analysis and Recognition ICDAR2003”, pp. 741–745, Edinburgh (2003). IEEE Computer Society, Los Alamitos, CA.
23. F. CHANG, C. J. CHEN, AND C. J. LU. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision, Graphics, and Image Processing: Image Understanding* **93**(2), 206–220 (February 2004).
24. P. R. COHEN AND E. A. FEIGENBAUM. “The Handbook of Artificial Intelligence”. William Kaufmann, Los Altos, CA (1982).
25. T. H. CORMAN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN. “Introduction to Algorithms”. MIT Press, Cambridge, MA, second ed. (2001).
26. L. S. DAVIS. A survey of edge detection techniques. *Computer Graphics and Image Processing* **4**, 248–270 (1975).
27. R. O. DUDA, P. E. HART, AND D. G. STORK. “Pattern Classification”. Wiley, New York (2001).
28. B. ECKEL. “Thinking in Java”. Prentice Hall, Englewood Cliffs, NJ, fourth ed. (2006). Earlier versions available online.
29. N. EFFORD. “Digital Image Processing—A Practical Introduction Using Java”. Pearson Education, Upper Saddle River, NJ (2000).
30. D. FLANAGAN. “Java in a Nutshell”. O’Reilly, Sebastopol, CA, fifth ed. (2005).
31. J. D. FOLEY, A. VAN DAM, S. K. FEINER, AND J. F. HUGHES. “Computer Graphics: Principles and Practice”. Addison-Wesley, Reading, MA, second ed. (1996).
32. A. FORD AND A. ROBERTS. “Colour Space Conversions” (1998). <http://www.poynton.com/PDFs/coloureq.pdf>.
33. W. FÖRSTNER AND E. GÜLCH. A fast operator for detection and precise location of distinct points, corners and centres of circular features. In A. GRÜN AND H. BEYER, editors, “Proceedings, International Society for Photogrammetry and Remote Sensing Intercommission Conference on the Fast Processing of Photogrammetric Data”, pp. 281–305, Interlaken (June 1987).
34. D. A. FORSYTH AND J. PONCE. “Computer Vision—A Modern Approach”. Prentice Hall, Englewood Cliffs, NJ (2003).
35. H. FREEMAN. Computer processing of line drawing images. *ACM Computing Surveys* **6**(1), 57–97 (March 1974).
36. M. GERVAUTZ AND W. PURGATHOFER. A simple method for color quantization: octree quantization. In A. GLASSNER, editor, “Graphics Gems I”, pp. 287–293. Academic Press, New York (1990).

37. A. S. GLASSNER. "Principles of Digital Image Synthesis". Morgan Kaufmann Publishers, San Francisco (1995).
38. R. C. GONZALEZ AND R. E. WOODS. "Digital Image Processing". Addison-Wesley, Reading, MA (1992).
39. R. L. GRAHAM, D. E. KNUTH, AND O. PATASHNIK. "Concrete Mathematics: A Foundation for Computer Science". Addison-Wesley, Reading, MA, second ed. (1994).
40. P. GREEN. Colorimetry and colour differences. In P. GREEN AND L. MACDONALD, editors, "Colour Engineering", ch. 3, pp. 40–77. Wiley, New York (2002).
41. E. L. HALL. "Computer Image Processing and Recognition". Academic Press, New York (1979).
42. C. G. HARRIS AND M. STEPHENS. A combined corner and edge detector. In C. J. TAYLOR, editor, "4th Alvey Vision Conference", pp. 147–151, Manchester (1988).
43. P. S. HECKBERT. Color image quantization for frame buffer display. *Computer Graphics* **16**(3), 297–307 (1982).
44. P. S. HECKBERT. Fundamentals of texture mapping and image warping. Master's thesis, University of California, Berkeley, Dept. of Electrical Engineering and Computer Science (1989). <http://www.cs.cmu.edu/~ph/#papers>.
45. J. HOLM, I. TASTL, L. HANLON, AND P. HUBEL. Color processing for digital photography. In P. GREEN AND L. MACDONALD, editors, "Colour Engineering", ch. 9, pp. 179–220. Wiley, New York (2002).
46. B. K. P. HORN. "Robot Vision". MIT-Press, Cambridge, MA (1982).
47. P. V. C. HOUGH. Method and means for recognizing complex patterns. US Patent 3,069,654 (1962).
48. M. K. HU. Visual pattern recognition by moment invariants. *IEEE Transactions on Information Theory* **8**, 179–187 (1962).
49. R. W. G. HUNT. "The Reproduction of Colour". Wiley, New York, sixth ed. (2004).
50. J. ILLINGWORTH AND J. KITTLER. A survey of the Hough transform. *Computer Vision, Graphics and Image Processing* **44**, 87–116 (1988).
51. International Color Consortium. "Specification ICC.1:2004-10 (Profile Version 4.2.0.0): Image Technology Colour Management—Architecture, Profile Format, and Data Structure" (2004). http://www.color.org/documents/ICC1v42_2006-05.pdf.
52. International Electrotechnical Commission, IEC, Geneva. "IEC 61966-2-1: Multimedia Systems and Equipment—Colour Measurement and Management, Part 2-1: Colour Management—Default RGB Colour Space—sRGB" (1999). <http://www.iec.ch>.
53. International Organization for Standardization, ISO, Geneva. "ISO 13655:1996, Graphic Technology—Spectral Measurement and Colorimetric Computation for Graphic Arts Images" (1996).
54. International Organization for Standardization, ISO, Geneva. "ISO 15076-1:2005, Image Technology Colour Management—Architecture, Profile Format, and Data Structure: Part 1" (2005). Based on ICC.1:2004-10.
55. International Telecommunications Union, ITU, Geneva. "ITU-R Recommendation BT.709-3: Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange" (1998).

56. International Telecommunications Union, ITU, Geneva. "ITU-R Recommendation BT.601-5: Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios" (1999).
57. K. JACK. "Video Demystified—A Handbook for the Digital Engineer". LLH Publishing, Eagle Rock, VA, third ed. (2001).
58. B. JÄHNE. "Practical Handbook on Image Processing for Scientific Applications". CRC Press, Boca Raton, FL (1997).
59. B. JÄHNE. "Digitale Bildverarbeitung". Springer-Verlag, Berlin, fifth ed. (2002).
60. A. K. JAIN. "Fundamentals of Digital Image Processing". Prentice Hall, Englewood Cliffs, NJ (1989).
61. X. Y. JIANG AND H. BUNKE. Simple and fast computation of moments. *Pattern Recognition* **24**(8), 801–806 (1991).
62. J. KING. Engineering color at Adobe. In P. GREEN AND L. MACDONALD, editors, "Colour Engineering", ch. 15, pp. 341–369. Wiley, New York (2002).
63. R. A. KIRSCH. Computer determination of the constituent structure of biological images. *Computers in Biomedical Research* **4**, 315–328 (1971).
64. L. KITCHEN AND A. ROSENFELD. Gray-level corner detection. *Pattern Recognition Letters* **1**, 95–102 (1982).
65. T. LINDBERG. Feature detection with automatic scale selection. *International Journal of Computer Vision* **30**(2), 77–116 (1998).
66. D. G. LOWE. Object recognition from local scale-invariant features. In "Proceedings of the 7th IEEE International Conference on Computer Vision ICCV'99", vol. 2, pp. 1150–1157, Kerkyra, Corfu, Greece (1999). IEEE Computer Society, Los Alamitos, CA.
67. B. LUCAS AND T. KANADE. An iterative image registration technique with an application to stereo vision. In P. J. HAYES, editor, "Proceedings of the 7th International Joint Conference on Artificial Intelligence IJCAI'81", pp. 674–679, Vancouver, BC (1981). William Kaufmann, Los Altos, CA.
68. S. MALLAT. "A Wavelet Tour of Signal Processing". Academic Press, New York (1999).
69. D. MARR AND E. HILDRETH. Theory of edge detection. *Proceedings of the Royal Society of London, Series B* **207**, 187–217 (1980).
70. E. H. W. MEIJERING, W. J. NIESSEN, AND M. A. VIERGEVER. Quantitative evaluation of convolution-based methods for medical image interpolation. *Medical Image Analysis* **5**(2), 111–126 (2001). <http://imagescience.bigr.nl/meijering/software/transformj/>.
71. J. MIANO. "Compressed Image File Formats". ACM Press, Addison-Wesley, Reading, MA (1999).
72. D. P. MITCHELL AND A. N. NETRAVALI. Reconstruction filters in computer-graphics. In R. J. BEACH, editor, "Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'88", pp. 221–228, Atlanta, GA (1988). ACM Press, New York.
73. P. A. MLSNA AND J. J. RODRIGUEZ. Gradient and laplacian-type edge detection. In A. BOVIK, editor, "Handbook of Image and Video Processing", pp. 415–431. Academic Press, New York (2000).
74. J. D. MURRAY AND W. VANRYPER. "Encyclopedia of Graphics File Formats". O'Reilly, Sebastopol, CA, second ed. (1996).

75. M. NADLER AND E. P. SMITH. "Pattern Recognition Engineering". Wiley, New York (1993).
76. A. V. OPPENHEIM, R. W. SHAFER, AND J. R. BUCK. "Discrete-Time Signal Processing". Prentice Hall, Englewood Cliffs, NJ, second ed. (1999).
77. T. PAVLIDIS. "Algorithms for Graphics and Image Processing". Computer Science Press / Springer-Verlag, New York (1982).
78. C. A. POYNTON. "Digital Video and HDTV Algorithms and Interfaces". Morgan Kaufmann Publishers, San Francisco (2003).
79. W. S. RASBAND. "ImageJ". U.S. National Institutes of Health, MD (1997–2007). <http://rsb.info.nih.gov/ij/>.
80. C. E. REID AND T. B. PASSIN. "Signal Processing in C". Wiley, New York (1992).
81. D. RICH. Instruments and methods for colour measurement. In P. GREEN AND L. MACDONALD, editors, "Colour Engineering", ch. 2, pp. 19–48. Wiley, New York (2002).
82. I. E. G. RICHARDSON. "H.264 and MPEG-4 Video Compression". Wiley, New York (2003).
83. L. G. ROBERTS. Machine perception of three-dimensional solids. In J. T. TIPPET, editor, "Optical and Electro-Optical Information Processing", pp. 159–197. MIT Press, Cambridge, MA (1965).
84. A. ROSENFELD AND P. PFALTZ. Sequential operations in digital picture processing. *Journal of the ACM* **12**, 471–494 (1966).
85. J. C. RUSS. "The Image Processing Handbook". CRC Press, Boca Raton, FL, third ed. (1998).
86. C. SCHMID AND R. MOHR. Local grayvalue invariants for image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **19**(5), 530–535 (May 1997).
87. C. SCHMID, R. MOHR, AND C. BAUCKHAGE. Evaluation of interest point detectors. *International Journal of Computer Vision* **37**(2), 151–172 (2000).
88. Y. SCHWARZER, editor. "Die Farbenlehre Goethes". Westerweide Verlag, Witten (2004).
89. M. SEUL, L. O'GORMAN, AND M. J. SAMMON. "Practical Algorithms for Image Analysis". Cambridge University Press, Cambridge (2000).
90. L. G. SHAPIRO AND G. C. STOCKMAN. "Computer Vision". Prentice Hall, Englewood Cliffs, NJ (2001).
91. G. SHARMA AND H. J. TRUSSELL. Digital color imaging. *IEEE Transactions on Image Processing* **6**(7), 901–932 (1997).
92. N. SILVESTRINI AND E. P. FISCHER. "Farbsysteme in Kunst und Wissenschaft". DuMont, Cologne (1998).
93. Y. SIRISATHITKUL, S. AUWATANAMONGKOL, AND B. UYYANONVARA. Color image quantization using distances between adjacent colors along the color axis with highest color variance. *Pattern Recognition Letters* **25**, 1025–1043 (2004).
94. S. M. SMITH AND J. M. BRADY. SUSAN—a new approach to low level image processing. *International Journal of Computer Vision* **23**(1), 45–78 (1997).
95. M. SONKA, V. HLAVAC, AND R. BOYLE. "Image Processing, Analysis and Machine Vision". PWS Publishing, Pacific Grove, CA, second ed. (1999).

REFERENCES

96. M. STOKES AND M. ANDERSON. "A Standard Default Color Space for the Internet—sRGB". Hewlett-Packard, Microsoft, www.w3.org/Graphics/Color/sRGB.html (1996).
97. S. SÜSSTRUNK. Managing color in digital image libraries. In P. GREEN AND L. MACDONALD, editors, "Colour Engineering", ch. 17, pp. 385–419. Wiley, New York (2002).
98. S. THEODORIDIS AND K. KOUTROUMBAS. "Pattern Recognition". Academic Press, New York (1999).
99. E. TRUCCO AND A. VERRI. "Introductory Techniques for 3-D Computer Vision". Prentice Hall, Englewood Cliffs, NJ (1998).
100. K. TURKOWSKI. Filters for common resampling tasks. In A. GLASSNER, editor, "Graphics Gems I", pp. 147–165. Academic Press, New York (1990).
101. T. TUYTELAARS AND L. J. VAN GOOL. Matching widely separated views based on affine invariant regions. *International Journal of Computer Vision* **59**(1), 61–85 (August 2004).
102. D. WALLNER. Color management and transformation through ICC profiles. In P. GREEN AND L. MACDONALD, editors, "Colour Engineering", ch. 11, pp. 247–261. Wiley, New York (2002).
103. A. WATT. "3D Computer Graphics". Addison-Wesley, Reading, MA, third ed. (1999).
104. A. WATT AND F. POLICARPO. "The Computer Image". Addison-Wesley, Reading, MA (1999).
105. G. WOLBERG. "Digital Image Warping". IEEE Computer Society Press, Los Alamitos, CA (1990).
106. G. WYSZECKI AND W. S. STILES. "Color Science: Concepts and Methods, Quantitative Data and Formulae". Wiley-Interscience, New York, second ed. (2000).
107. T. Y. ZHANG AND C. Y. SUEN. A fast parallel algorithm for thinning digital patterns. *Communications of the ACM* **27**(3), 236–239 (1984).
108. S. ZOKAI AND G. WOLBERG. Image registration using log-polar mappings for recovery of large-scale similarity and projective transformations. *IEEE Transactions on Image Processing* **14**(10), 1422–1434 (October 2005).

Index

Symbols

* (convolution operator), 99, 452
⊗ (correlation operator), 432, 452
⊕ (dilation operator), 177, 452
⊖ (erosion operator), 178, 452
? (operator), 267
[,] , 47, 61, 452
∂, 119, 140, 452
∇, 119, 130, 452
& (operator), 244, 304, 460, 491
&& (operator), 150, 162, 203, 514
| (operator), 245, 304, 520
|| (operator), 514
~ (operator), 514
» (operator), 245, 304
« (operator), 245
% (operator), 459

A

abs (method), 81, 461, 495
absolute value, 495
accessing pixels, 485
accumulator array, 159
achromatic, 260
acos (method), 461
ADD (constant), 82, 85, 496
add (method), 81, 495, 529, 530, 534, 540
addChoice (method), 86, 507
addNumericField (method), 86, 507, 518, 526
addSlice (method), 505
addStringField (method), 518
adjoint matrix, 382
Adobe
 Illustrator, 15
 Photoshop, 59, 94, 115, 134
 RGB, 288
affine mapping, 378, 386
AffineMapping (class), 417, 418
AffineTransform (class), 413
aliasing, 329, 334, 337, 339, 350, 410
alpha
 blending, 83, 85, 506
 channel, 17, 244
 value, 83, 244
altKeyDown (method), 517
ambient lighting, 280
amplitude, 314–316
AND (constant), 82, 496
and (method), 495
angular frequency, 314, 315, 334, 338, 344
anti-aliasing, 500
apply (method), 297
applyTable (method), 68, 77, 80, 152, 495, 530
applyTo (method), 415, 416, 421, 423, 426, 427
approximation, 401, 402
ArcTan function, 122, 350, 388, 452, 462
area
 polygon, 224
 region, 224
arithmetic operation, 81, 82, 495
array, 462–467
 accessing elements, 463
 creation, 462
 duplication, 465
 size, 463
 sorting, 466
 two-dimensional, 464
ArrayList (class), 149, 468, 534, 539
Arrays (class), 300, 466
Arrays.sort (method), 466
asin (method), 461
associativity, 101, 178
atan (method), 461
atan2 (method), 452, 461, 462
auto-contrast, 57
 modified, 58

autoThreshold (method), 495
AVERAGE (constant), 82, 496
AWT, 244, 294

B

background, 173
 bandwidth, 330
 Bartlett window, 355, 357, 358
BasicStroke (class), 541, 542
 basis function, 333–337, 344, 349, 367, 368, 373
beep (method), 517
 bias problem, 165
 bicubic interpolation, 407
BicubicInterpolator (class), 425, 426
 big endian, 22–24
 bilinear
 interpolation, 405, 485
 mapping, 385, 386
BilinearInterpolator (class), 424, 426
BilinearMapping (class), 421
 binarization, 55
 binary
 image, 13, 129, 173, 189, 199, 472
 morphology, 173–186
BinaryProcessor (class), 55, 195, 472
binnedHistogram (method), 48
 binning, 46–48, 51
 bit
 mask, 244
 operation, 246
 bitmap image, 13, 218
 bitwise AND operator, 460
black (constant), 541
 black box, 99
 black-generation function, 273
Blitter (interface), 82, 85
BMP, 20, 23, 246
 bounding box, 225
 box filter, 92, 102, 121, 497
 Bradford model, 289, 292
BradfordAdaptation (class), 297
 breadth-first, 202
 Bresenham algorithm, 168
 brightness, 54, 490
 byte, 22
byte (type), 459, 472
ByteBlitter (class), 508

ByteProcessor (class), 82, 252, 253, 472

ByteStatistics (class), 494

C

C2-continuous, 400
 camera obscura, 7
 Canny edge operator, 127, 129
Canvas (class), 474
 card, 38, 452, 453
 cardinal spline, 398, 400
 cardinality, 452, 453
 Catmull-Rom interpolation, 400
 CCITT, 15
Cdf (method), 70
 cdf, *see* cumulative distribution function
ceil (method), 461
CENTER_JUSTIFY (constant), 500
centralMoment (method), 229
 centroid, 226
 CGM format, 15
 chain code, 219, 224
 chamfer
 algorithm, 443
 matching, 446
 chroma, 270
 chromatic adaptation, 288
 Bradford model, 289, 292
 XYZ scaling, 289
ChromaticAdaptation (class), 297
 CIE, 276
 chromaticity diagram, 277, 280
 L*a*b*, 275, 281, 282
 standard illuminant, 279
 XYZ, 276, 282, 285, 286, 295, 298
 circle, 167, 380
 circularity, 224
 circumference, 223
 city block distance, 443
 clamping, 54, 92
 clipboard, 522
clone (method), 299, 300, 416, 465
Cloneable (interface), 465
 cloning arrays, 465
close (method), 193, 194
closeAllWindows (method), 521
 closing, 185, 188, 193
 clutter, 447
 CMYK, 271–275
collectCorners (method), 149

Collections (class), 150
 collision, 207
Color (class), 261–263, 294, 541, 542
 color
 count, 299
 difference, 283
 image, 13, 239–312, 473
 keying, 266
 management, 296
 pixel, 242, 244, 245
 saturation, 257
 table, 243, 248, 250, 311
 temperature, 279
 color quantization, 43, 244, 250, 254, 301–310
 3:3:2, 303
 median-cut, 305
 octree, 306
 populosity, 305
 color space, 253–299
 CMYK, 271
 colorimetric, 275–299
 HLS, 258
 HSB, 258, 295
 HSV, 258, 295
 in Java, 292–299
 L*a*b*, 281
 RGB, 240
 sRGB, 283
 XYZ, 276
 YCbCr, 270
 YIQ, 269
 YUV, 268
 color system
 additive, 239
 subtractive, 272
COLOR_RGB (constant), 248
ColorChooser (class), 474
ColorModel (class), 250, 294, 502, 503
ColorProcessor (class), 194, 245, 247, 251, 253, 257, 300, 473, 485
ColorSpace (class), 293–295, 297
ColorStatistics (class), 494
 comb function, 327
 commutativity, 100, 178, 179
 compactness, 224
Comparable (interface), 467
compareTo (method), 150, 467, 527
 comparing images, 429–450
 complementary set, 176
Complex (class), 341
 complex number, 316, 453
 complexity, 454
 component
 histogram, 48
 ordering, 242
computeMatch (method), 438, 441
 computer
 graphics, 2
 vision, 3
concat (method), 417
concolve (method), 137
Concolver (class), 137
 conic section, 380
 connected components problem, 207
 container, 149
contains (method), 511
 contour, 127, 209–216
ContourOverlay (class), 216
 contrast, 40, 54
 automatic adjustment, 57
convertHSBToRGB (method), 254, 493
convertRGBStackToRGB (method), 493
convertRGBtoIndexedColor (method), 254, 494
convertToByte (method), 85, 137, 196, 253, 257, 485, 492, 508
convertToFloat (method), 137, 253, 492, 528
convertToGray16 (method), 254, 493
convertToGray32 (method), 254, 493
convertToGray8 (method), 254, 493
convertToHSB (method), 254, 493
convertToRGB (method), 253, 254, 492, 493
convertToRGBStack (method), 494
convertToShort (method), 253, 492
 convex hull, 225, 236
 convexity, 225, 234
 convolution, 99, 364, 433, 455
 property, 324, 363
convolve (method), 114, 148, 497
convolve3x3 (method), 497
Convolver (class), 114, 148, 531
 coordinate
 Cartesian, 378
 homogeneous, 377, 416
COPY (constant), 496
copy (method), 522

- `COPY_INVERTED` (constant), 496
 - `copyBits` (method), 82, 85, 137, 193, 194, 496, 508, 531
 - `Corner` (class), 148, 150
 - corner, 139
 - detection, 139–153
 - point, 153
 - response function, 141, 145
 - strength, 141
 - `CorrCoeffMatcher` (class), 438, 440
 - correlation, 99, 364, 432
 - coefficient, 434
 - `cos` (method), 461
 - cosine function, 322
 - one-dimensional, 314
 - two-dimensional, 346, 347
 - cosine transform, 18, 367
 - cosine² window, 357, 358
 - `countColors` (method), 300
 - counting colors, 299
 - `createByteImage` (method), 477, 508, 518
 - `createEmptyStack` (method), 476, 504
 - `createFloatImage` (method), 477
 - `createImage` (method), 477, 478
 - `createProcessor` (method), 192, 478
 - `createRGBImage` (method), 478
 - `createShortImage` (method), 477
 - creating
 - image processors, 478–480
 - new images, 52, 476–478
 - `crop` (method), 497
 - cross correlation, 432–435
 - CRT, 240
 - `CS_CIEXYZ` (constant), 295
 - `CS_GRAY` (constant), 295
 - `CS_LINEAR_RGB` (constant), 295
 - `CS_PYCC` (constant), 295
 - `CS_sRGB` (constant), 295
 - `CS_sRGBt` (constant), 297
 - cubic
 - B-spline interpolation, 401
 - interpolation, 397
 - spline, 400
 - cumulative
 - distribution function, 64
 - histogram, 50, 58, 62, 64
 - `currentMemory` (method), 523
 - cycle length, 314
- ## D
- D50, 279, 280, 295
 - D65, 280, 282, 284
 - DCT, 367–373
 - one-dimensional, 367, 368
 - two-dimensional, 370
 - DCT (method), 370
 - debugging, 112
 - deconvolution, 365
 - `deleteLastSlice` (method), 505
 - `deleteSlice` (method), 505
 - delta function, 325
 - depth-first, 202
 - derivative
 - estimation, 119
 - first, 118, 144
 - partial, 119
 - second, 126, 130
 - desaturation, 257
 - determinant, 382
 - DFT, 332–366, 452
 - one-dimensional, 332–341
 - two-dimensional, 343–366
 - DFT (method), 341
 - diameter, 226
 - DICOM, 31
 - `DIFFERENCE` (constant), 82, 194, 496
 - difference filter, 98
 - digital images, 8
 - `dilate` (method), 191, 193, 194, 497
 - dilation, 177, 187, 191, 497
 - Dirac function, 103, 179, 320, 325
 - `DirectColorModel` (class), 502
 - directory information, 523
 - discrete
 - cosine transform, 367–373
 - Fourier transform, 332–366, 452
 - sine transform, 367
 - displaying images, 500, 503
 - distance, 151, 431
 - city block, 443
 - Euclidean, 432, 443
 - Manhattan, 443
 - mask, 443
 - maximum difference, 431
 - sum of differences, 431
 - sum of squared differences, 432
 - transform, 442
 - `DIVIDE` (constant), 82, 496
 - `DOES_16` (constant), 519
 - `DOES_32` (constant), 519

DOES_8C (constant), 248, 249, 251, 519
DOES_8G (constant), 32, 45, 519
DOES_ALL (constant), 493, 519
DOES_RGB (constant), 246, 247, 519
DOES_STACKS (constant), 519
DONE (constant), 493, 503, 519
dots per inch (dpi), 11, 339
Double (class), 462
double (type), 92, 458
dpi, 339
draw (method), 152, 500, 501, 542
drawDot (method), 499, 500
drawLine (method), 152, 499, 527
drawOval (method), 499
drawPixel (method), 499
drawPolygon (method), 499
drawRect (method), 499
drawString (method), 499, 500
DST, 367
duplicate (method), 85, 93, 94, 109, 137, 152, 194, 416, 478, 508, 529, 533
duplicateArray (method), 466
DXF format, 15
dynamic range, 40

E

E (constant), 461
eccentricity, 231, 236
Eclipse, 34, 470
edge
 filter, 497
 map, 129, 155
 sharpening, 130–137
 strength, 141
edge operator, 120–127
 Canny, 127, 129
 compass, 123
 in ImageJ, 125
 Kirsch, 123
 LoG, 126, 129
 Prewitt, 120, 129
 Roberts, 123, 129
 Sobel, 120, 125, 129
effective gamma value, 79
eigenvalue, 141, 231
eigenvector, 141
ellipse, 170, 232, 380, 499
Ellipse2D (class), 534
elliptical window, 356

elongatedness, 231
EMF format, 15
Encapsulated PostScript (EPS), 15
erode (method), 193, 194, 497
erosion, 178, 187, 191, 497
error (method), 515
escapePressed (method), 517
Euclidean distance, 151, 438, 443
Euler number, 235
Euler's notation, 316
executing plugins, 520
EXIF, 19, 284
exp (method), 461
exposure, 40

F

fast Fourier transform, 342, 345, 359, 364, 455
fax encoding, 219
feature, 222
FFT, *see* fast Fourier transform, *see* fast Fourier transform
file format, 24
 BMP, 20
 EXIF, 19
 GIF, 15
 JFIF, 19
 JPEG-2000, 19
 magic number, 23
 PBM, 21
 Photoshop, 23
 PNG, 16
 RAS, 22
 RGB, 22
 TGA, 22
 TIFF, 15
 XBM/XPM, 22
file information, 522
FileInfo (class), 484, 522, 523
FileOpener (class), 484
FileSaver (class), 482
FileSaver (method), 482
fill (method), 52, 499
fillOval (method), 499
fillPolygon (method), 499
filter, 87–116, 497
 average, 497
 border handling, 91, 111
 box, 92, 97, 102, 121, 497
 color image, 136
 computation, 91

- debugging, 112
- derivative, 119
- difference, 98
- edge, 120–125, 497
- efficiency, 111
- Gaussian, 97, 102, 114, 133, 140, 144
- ImageJ, 113–115
- impulse response, 103
- in frequency space, 363
- indexed image, 248
- inverse, 364
- kernel, 99
- Laplace, 98, 131, 136
- Laplacian, 116
- linear, 89–104, 113, 497
- low-pass, 97
- mask, 89
- matrix, 89
- maximum, 105, 115, 197, 497
- median, 106, 115, 173, 497
- minimum, 105, 115, 197, 497
- morphological, 173–197
- nonlinear, 104–111, 115
- normalized, 93
- separable, 101, 102, 131
- smoothing, 92, 94, 96, 134, 497
- unsharp masking, 133
- weighted median, 107
- Find_Corners** (plugin), 153
- findCorners** (method), 152
- findEdges** (method), 125, 497
- FITS, 31
- flipHorizontal** (method), 497
- flipVertical** (method), 498
- Float** (class), 462
- float** (type), 473
- floating-point image, 13, 473
- FloatProcessor** (class), 253, 438, 473
- FloatStatistics** (class), 494
- floatToIntBits** (method), 486
- flood filling, 200–204
- floor** (method), 461
- floor function, 453
- font, 500
- for-loop, 488
- foreground, 173
- four-point mapping, 380
- Fourier, 317
 - analysis, 318
 - coefficients, 318
 - descriptor, 221
 - integral, 318
 - series, 317
 - spectrum, 222, 319, 330
 - transform, 314–452
 - transform pair, 320, 322, 323
- Frame** (class), 474, 501
- FreehandRoi** (class), 475, 506
- freeMemory** (method), 523
- frequency, 314, 338
 - angular, 314, 315, 334, 344
 - common, 315
 - directional, 348
 - distribution, 63
 - effective, 348, 349
 - fundamental, 317, 318, 339
 - maximum, 329, 350
 - space, 320, 338, 363
 - two-dimensional, 348
- fromCIEXYZ** (method), 292–294, 297
- function
 - basis, 333–337, 344
 - cosine, 314
 - delta, 325
 - Dirac, 320, 325
 - impulse, 320, 325
 - periodic, 314
 - sine, 314
- fundamental
 - frequency, 317, 318, 339
 - period, 338
- G**
- gamma** (method), 81, 495
- gamma correction, 72–80, 256, 292, 295, 298, 495
 - applications, 75
 - inverse, 79
 - modified, 76–80, 285
- gamut, 273, 280, 283, 288
 - Adobe RGB, 288
 - sRGB, 288
- garbage, 463
- Gaussian
 - area formula, 224
 - distribution, 51
 - filter, 97, 102, 114, 133, 140, 144
 - filter size, 102
 - function, 321, 323
 - separable, 102

window, 355, 356, 358
 GaussKernel1d (class), 137
 GenericDialog (class), 84, 86, 474,
 507, 517, 518, 526
 geometric operation, 375–428, 497
 get (method), 34, 54, 63, 111, 259,
 487, 490, 534, 538
 get2dHistogram (method), 301
 getBitDepth (method), 248
 getBlues (method), 249, 251
 getBrightness (method), 490
 getClipboard (method), 522
 getColorModel (method), 249–251,
 502
 getColumn (method), 489
 getComponents (method), 295
 getCurrentColorModel (method),
 503
 getCurrentImage (method), 249, 521
 getCurrentSlice (method), 501
 getCurrentWindow (method), 521
 getDirectory (method), 523
 getDoScaling (method), 494
 getf (method), 440, 441, 487
 getFileInfo (method), 522
 getFloatArray (method), 490
 getGreens (method), 249, 251
 getHeight (method), 33, 93, 485,
 491, 505
 getHistogram (method), 45, 52, 63,
 68, 299, 485, 494
 getHistogramMax (method), 494
 getHistogramMin (method), 494
 getHistogramSize (method), 494
 getHSB (method), 490
 getID (method), 501
 getIDList (method), 86, 507, 521
 getImage (method), 86, 484, 501,
 507, 517, 521
 getImageArray (method), 505
 getImageCount (method), 521
 getImageStack (method), 504
 getIntArray (method), 489
 getInterpolate (method), 485, 498
 getInterpolatedPixel (method),
 424, 425, 498
 getInterpolatedRGBPixel
 (method), 498
 getInverse (method), 415
 getLine (method), 485, 489
 getMagnification (method), 542

getMapSize (method), 249–251
 getMask (method), 510, 511, 514
 getMaskArray (method), 512, 513
 getMatchValue (method), 441
 getMax (method), 503
 getMin (method), 503
 getNextChoiceIndex (method), 86,
 507
 getNextNumber (method), 86, 507,
 518, 526
 getNextString (method), 518
 getNumber (method), 515
 getOriginalFileInfo (method),
 484, 523
 getPixel (method), 33, 93, 109, 111,
 245, 304, 460, 486, 490
 getPixelCount (method), 487
 getPixels (method), 463, 488, 491,
 505, 529
 getPixelsCopy (method), 489
 getPixelSize (method), 249
 getPixelValue (method), 485, 486
 getProcessor (method), 85, 478,
 484, 493, 501, 502, 505, 508
 getProperties (method), 514
 getProperty (method), 514, 516
 getReds (method), 249, 251
 getRGB (method), 490
 getRoi (method), 509, 512, 514
 getRow (method), 489
 getShortSliceLabel (method), 505
 getShortTitle (method), 86, 501,
 507
 getSize (method), 505
 getSliceLabel (method), 505
 getSliceLabels (method), 505
 getStack (method), 476, 502, 504,
 508
 getStackSize (method), 504
 getString (method), 515
 getStringWidth (method), 499
 getTempCurrentImage (method), 521
 getTitle (method), 501, 526
 getType (method), 248
 getVersion (method), 523
 getWeightingFactors (method),
 257, 485
 getWidth (method), 33, 93, 485, 491,
 506
 getWindow (method), 501
 getWindowCount (method), 521

GIF, 15, 23, 31, 43, 219, 243, 248
 gradient, 118, 119, 140, 144
 graph, 207
Graphics (class), 541
 graphics overlay, 216
Graphics2D (class), 542
 grayscale
 conversion, 256, 287
 image, 12, 17, 472
 morphology, 187–188

H

Hadamard transform, 372
HandleExtraFileTypes (class), 481
 Hanning window, 354, 355, 357, 358
 Harris corner detector, 140
HarrisCornerDetector (class), 148, 153
HashSet (class), 468
hasNext (method), 534
 HDTV, 271
 Hertz, 315, 339
 Hessian normal form, 159, 166
 hexadecimal, 245, 460
hide (method), 501
 hierarchical techniques, 127
 histogram, 37–51, 299–301, 452, 494
 binning, 46
 channel, 48
 color image, 47
 component, 48
 computing, 44
 cumulative, 50, 58, 64
 equalization, 59
 matching, 67
 normalized, 63
 specification, 62–72
 HLS, 258, 264–266, 270
HLStoRGB (method), 268
 homogeneous
 coordinate, 377, 416
 point operation, 53, 60, 63
 hot spot, 89, 176
 Hough transform, 130, 156–171
 bias problem, 165
 edge strength, 167
 for circles, 167–169
 for ellipses, 170
 for lines, 156–167
 generalized, 170
 hierarchical, 167

HSB, *see* HSV
HSBtoRGB (method), 263, 264, 295
 HSV, 254, 255, 258, 261, 266, 270, 295, 490
 hue, 490
 Huffman code, 18

I

i (imaginary unit), 316, 452, 453
 ICC, 292
 profile, 296
ICC_ColorSpace (class), 295, 296
ICC_Profile (class), 296
iDCT (method), 370
 idempotent, 186
IJ (class), 475, 477, 480, 523
 ij (package), 471, 475
 ij.gui (package), 474, 509
 ij.io (package), 475
 ij.plugin (package), 473
 ij.plugin.filter (package), 473
 ij.process (package), 472
Illuminant (class), 297
 illuminant, 279
Image (class), 476, 478
 image
 acquisition, 5
 analysis, 3
 binary, 13, 199, 472
 bitmap, 13
 color, 13, 473
 compression and histogram, 43
 coordinates, 11, 452
 creating new, 52
 defects, 42
 depth, 12
 digital, 8
 display, 52
 file, 480
 file format, 5, 13
 floating-point, 13, 473
 grayscale, 12, 17, 472
 height, 485
 indexed color, 13, 17, 472
 intensity, 12
 loading, 480
 locking, 522
 palette, 13
 parameter, 485
 plane, 7
 properties, 513–515

raster, 14
 RGB, 473
 size, 10
 space, 100, 363
 special, 13
 statistics, 494
 storing, 480
 true color, 17
 warping, 387
 width, 485
ImageCanvas (class), 474, 541
ImageConverter (class), 253, 254, 492
ImageJ, 27–36
 accessing pixels, 485
 animation, 503
 API, 471–475
 directory, 523
 displaying images, 500
 filter, 113–115, 497
 geometric operation, 413, 497
 graphic operation, 499
 GUI, 474
 histogram, 494
 image conversion, 492
 installation, 469
 loading images, 480
 locking, 522
 macro, 30, 34
 main window, 30
 memory, 523
 open, 480
 plugin, 31–35, 518
 point operation, 80–86, 494
 region of interest, 506
 revert, 480
 save, 480
 snapshot, 35
 stack, 30, 504, 505
 storing images, 480
 system information, 523
 tutorial, 35
 undo, 31, 35
 Website, 35
ImagePlus (class), 153, 247, 251, 252, 471, 481, 509, 517, 522, 541
ImageProcessor (class), 32, 194, 246, 247, 249–253, 259, 463, 472, 485
ImageStack (class), 472, 505, 508, 512
ImageStatistics (class), 494
ImageWindow (class), 474, 533
 impulse
 function, 103, 320, 325
 response, 103, 183
 in place, 345
IndexColorModel (class), 249, 251, 252, 502
 indexed color image, 13, 17, 243, 248, 254, 472
insert (method), 137, 489, 500, 508
instanceof (operator), 488, 491
intBitsToFloat (method), 486
Integer.MAX_VALUE (constant), 535
 intensity
 histogram, 48
 image, 12
 interest point, 139
 interpolation, 392–410, 423–425, 485
 B-spline, 400, 401
 bicubic, 407, 409, 425
 bilinear, 405, 409, 424, 498
 by convolution, 397
 Catmull-Rom, 399, 400, 425, 426
 color, 498
 cubic, 397
 ideal, 393
 kernel, 397
 Lanczos, 402, 408, 428
 linear, 397
 Mitchell-Netravali, 400, 401, 428
 nearest-neighbor, 397, 405, 409, 412, 424, 498
 spline, 399
 two-dimensional, 404–410
invalidNumber (method), 526
 invariance, 224, 227, 228, 233, 234, 430
 inverse
 filter, 364
 gamma function, 74
 tangent function, 462
 inversion, 55
invert (method), 55, 81, 193, 415, 417
invertLookupTable (class), 501
invertLut (method), 190, 501, 503
isInvertedLut (method), 501, 504
isJava14 (method), 523

isJava2 (method), 523
isLocalMax (method), 150
isMacintosh (method), 523
isMacOSX (method), 523
isNaN (method), 536
isotropic, 89, 119, 131, 133, 140, 153, 181
isWindows (method), 523
Iterator (class), 534
iterator (method), 534
ITU601, 271
ITU709, 75, 79, 256, 271, 284, 485

J

Jama (package), 381, 421, 422
Java
 applet, 29
 arithmetic, 457
 array, 462–467
 AWT, 31
 class file, 34
 collection, 462
 compiler, 34
 editor, 470
 integer division, 63, 457
 JVM, 23, 469, 523
 mathematical functions, 460
 rounding, 461
 runtime environment, 29, 469, 470
 virtual machine, 23, 469
JBuilder, 34, 470
JFIF, 19, 21, 23
JPEG, 15, 17–19, 21, 23, 31, 44, 219, 243, 284, 287, 312, 369
JPEG-2000, 19
JVM, 523

K

kernel, 99
killRoi (method), 510
Kirsch operator, 123

L

$L^*a^*b^*$, 281
Lab_ColorSpace (class), 293, 297, 298
label, 200
Lanczos interpolation, 402, 408, 428
Laplace
 filter, 98, 131, 132, 136
 operator, 130

Laplacian of Gaussian (LoG), 116
LEFT_JUSTIFY (constant), 500
lens, 8
Line (class), 475, 506, 510
line, 499, 500
 endpoints, 166
 equation, 156, 159
 Hessian normal form, 159
 intercept/slope form, 156
 intersection, 166
linear
 convolution, 99, 497
 correlation, 99
 interpolation, 397
 transformation, 382
linearity, 100, 321
LinearMapping (class), 416, 419
lines per inch (lpi), 11
lineTo (method), 500
LinkedList (class), 202, 540
List (interface), 149, 150, 468, 528, 534, 537, 540, 541
list, 451
little endian, 22–24
loading images, 480
local mapping, 389
lock (method), 522
locking images, 522
lockSilently (method), 522
LoG
 filter, 116
 operator, 129
log (method), 81, 461, 495, 515
logic operation, 495
lookup table, 80, 152, 190, 530
LSB, 22
luminance, 256, 270
LZW, 15

M

magic number, 23
major axis, 228
makeGaussKernel1d (method), 103, 137
makeIndexColorImage (method), 252
makeInverseMapping (method), 423
makeLine (method), 513
makeMapping (method), 418, 420
makeOval (method), 513
makeRectangle (method), 513
managing windows, 521

Manhattan distance, 443
Mapping (class), 414
mapping
 affine, 378, 386
 bilinear, 385, 386
 four-point, 380
 function, 376
 linear, 382
 local, 389
 nonlinear, 386
 perspective, 380
 projective, 380–386
 ripple, 388
 spherical, 388
 three-point, 378
 twirl, 387
mask, 134, 218
 image, 511, 514
matchHistograms (method), 68
Math (class), 460, 461
Matrix (class), 421
MAX (constant), 82, 115, 496
max (method), 81, 461, 495
maximum
 filter, 105, 197
 frequency, 329, 350
maxMemory (method), 523
media-oriented color, 287
MEDIAN (constant), 115
median filter, 106, 115, 173, 497
 cross-shaped, 110
 weighted, 107
median-cut algorithm, 305
medianFilter (method), 497
memory management, 523
mesh partitioning, 389
MIN (constant), 82, 115, 496
min (method), 81, 461, 495
minimum filter, 105, 197
Mitchell-Netravali interpolation,
 401, 428
mod operator, 340, 395, 453
modified auto-contrast, 58
modulus, *see* mod operator
moment, 219, 226–233
 central, 227
 Hu's, 233, 236
 invariant, 233
 least inertia, 228
moment (method), 229
morphing, 390

morphological filter, 173–197
 binary, 173–186
 closing, 185, 188, 193
 color, 187
 dilation, 177, 187, 191
 erosion, 178, 187, 191
 grayscale, 187–188
 opening, 185, 188, 193
 outline, 181, 194
moveTo (method), 500
MSB, 22
multi-resolution techniques, 127
MULTIPLY (constant), 82, 497, 531
multiply (method), 81, 85, 137, 495,
 508
My_Inverter (plugin), 33

N
NaN (constant), 462, 535
nearest-neighbor interpolation, 397
NearestNeighborInterpolator
 (class), 424
NEGATIVE_INFINITY (constant), 462
neighborhood, 175, 200, 223
NetBeans, 34, 470
neutral
 point, 279
neutral element, 179
NewImage (class), 474, 477, 508, 518
newImage (method), 477
next (method), 534
nextGaussian (method), 51
nextInt (method), 51
NIH-Image, 29
NO_CHANGES (constant), 35, 45, 251,
 519
NO_IMAGE_REQUIRED (constant), 519
NO_UNDO (constant), 520
Node (class), 202
noImage (method), 36, 86, 493, 503,
 507, 515
noise (method), 495
nominal gamma value, 79
nonhomogeneous operation, 53
nonmaximum suppression, 164
normal distribution, 51
normalCentralMoment (method), 229
normalization, 93
normalized histogram, 63
NTSC, 75, 267, 269
null (constant), 463

Nyquist, 330, 350

O \mathcal{O} notation, 454

object, 451

OCR, 222, 235

octree algorithm, 306

open (method), 193, 194, 480, 481, 484**Opener** (class), 481**openImage** (method), 481, 482

opening, 185, 188, 193

opening images, 480–482, 484

openMultiple (method), 482**openTiff** (method), 482**openURL** (method), 482

optical axis, 7

OR (constant), 82, 497**or** (method), 495

orientation, 228, 348, 350

orthogonal, 373

oscillation, 314, 315

outer product, 102

outline, 181, 194

outline (method), 194, 195**OvalRoi** (class), 475, 506, 510

overlay, 533

P

packed ordering, 242–244

PAL, 75, 267

palette, 243, 248, 250

image, *see* indexed color image

parameter space, 157

partial derivative, 119

Parzen window, 354, 355, 357, 358

paste (method), 522

pattern recognition, 3, 222

PDF, 15

pdf, *see* probability density function

perimeter, 223

period, 314

periodicity, 314, 344, 349, 352

perspective

image, 170

mapping, 380

transformation, 7

phase, 315, 339

angle, 316

Photoshop, 23

PI (constant), 461

PICT format, 15

piecewise linear function, 65

pinhole camera, 7

pixel, 5

value, 11

PixelInterpolator (class), 423

planar ordering, 242

Plessey detector, 140

PlugIn (interface), 31, 473, 518**PlugInFilter** (interface), 31, 246, 473, 519

PNG, 16, 23, 31, 247, 248, 284

Pnt2d (class), 414**Point** (class), 534, 535, 538

point operation, 53–86, 494

arithmetic, 80

effects on histogram, 55

gamma correction, 72

histogram equalization, 59

homogeneous, 80

in ImageJ, 80–86

inversion, 55

thresholding, 55

point set, 176

point spread function, 104

Point2D.Double (class), 536**PointRoi** (class), 506**Polygon** (class), 511, 534, 542

polygon, 499

area, 224

PolygonRoi (class), 475, 506, 510**pop** (method), 203

populosity algorithm, 305

POSITIVE_INFINITY (constant), 462

PostScript, 15

pow (method), 77, 461

power spectrum, 339, 348

Prewitt operator, 120, 129

primary color, 240

print pattern, 363

probability, 64

density function, 63

distribution, 63

profile connection space, 292, 295

projection, 233, 237, 301

projective mapping, 380–386

ProjectiveMapping (class), 419, 426

pseudo-perspective mapping, 380

pseudocolor, 311

push (method), 203**putBehind** (method), 521

putColumn(method), 489
putPixel(method), 33, 93, 94, 109, 111, 245, 460, 486, 490
putPixelValue(method), 487
putRow(method), 489
 pyramid techniques, 127

Q

quadrilateral, 380
 quantization, 10, 55, 301–310
 linear, 303
 scalar, 303
 vector, 305

R

Random(package), 51
 random
 process, 63
 variable, 64
random(method), 51, 461
 random image, 51
rank(method), 115
RankFilters(class), 115
 RAS format, 22
 raster image, 14
 RAW format, 247
 reading and writing pixels, 485
Rectangle(class), 536
 rectangular pulse, 321, 323
 window, 356
reflect(method), 193
 reflection, 177, 179, 180
 refraction index, 388
 region, 199–237
 area, 224, 228, 236
 centroid, 226, 236
 convex hull, 225
 diameter, 226
 eccentricity, 231
 labeling, 200–209
 major axis, 228
 matrix representation, 216
 moment, 226
 orientation, 228
 perimeter, 223
 projection, 233
 run length encoding, 218
 topology, 234
 region of interest, 475, 481, 494, 495, 497, 501, 506–513, 520, 522
 relative colorimetry, 289

remainder operator, 459
RenderingHints(class), 542
repaintImageWindows(method), 521
repaintWindow(method), 501
 resampling, 390–392
resetEscape(method), 517
resetMinAndMax(method), 504
resetRoi(method), 512
resize(method), 485, 498
 resolution, 10
 reverting, 480, 484
revertToSaved(method), 484
 RGB
 color image, 239–253
 color space, 240, 270
 format, 22
 image, 473
RGBtoHLS(method), 267
RGBtoHSB(method), 261–263, 295
RIGHT_JUSTIFY(constant), 500
rint(method), 461
 ripple mapping, 388
 Roberts operator, 123, 129
 ROI, *see* region of interest
Roi(class), 475, 506
ROI_REQUIRED(constant), 520
rotate(method), 485, 498
rotateLeft(method), 498
rotateRight(method), 498
Rotation(class), 418, 426
 rotation, 233, 361, 375, 377, 426
round(method), 77, 93, 94, 461
 round function, 81, 453
 rounding, 54, 82, 458, 461
 roundness, 224
 rubber banding, 510
run(method), 32, 480, 518, 519
 run length encoding, 218
runPlugIn(method), 520

S

sampling, 325–330
 frequency, 350
 interval, 327, 328
 spatial, 9
 theorem, 329, 330, 337, 338, 350, 394
 time, 9
 saturation, 42, 257, 490
save(method), 481, 482
saveAs(method), 481

saveAsBmp (method), 482
saveAsGif (method), 482
saveAsJpeg (method), 482, 483
saveAsLut (method), 483
saveAsPng (method), 483
saveAsRaw (method), 483
saveAsRawStack (method), 483
saveAsText (method), 483
saveAsTiff (method), 483
saveAsTiffStack (method), 484
saveAsZip (method), 484
saving images, 480–484, 521
scale (method), 485, 498, 542
Scaling (class), 418
scaling, 233, 375, 377
separability, 101, 115, 181, 371
separable filter, 131
sequence, 451
Set (interface), 468
set, 176, 451
set (method), 34, 54, 63, 111, 259, 487, 490
setAntialiasedText (method), 500
setBackgroundValue (method), 498
setBrightness (method), 490
setClipRect (method), 500
setColor (method), 500, 542
setColorModel (method), 249, 250, 252, 504
setCurrentWindow (method), 521
setDoScaling (method), 494
setf (method), 441, 487
setFloatArray (method), 490
setFont (method), 500
setHistogramRange (method), 494
setHistogramSize (method), 494
setHSB (method), 490
setIntArray (method), 489
setInterpolate (method), 485, 489, 498
setJustification (method), 500
setLineWidth (method), 500
setMask (method), 512
setMinAndMax (method), 502–504, 533
setNormalize (method), 114, 137, 148, 531
setPixels (method), 488, 506
setProcessor (method), 478
setProperty (method), 515, 516
setRenderingHint (method), 542
setRGB (method), 490
setRoi (method), 510, 512
setSilentMode (method), 482
setSlice (method), 501
setSliceLabel (method), 506
setStack (method), 504
setStroke (method), 542
setTempCurrentImage (method), 521
setThreshold (method), 502, 503
setTitle (method), 501
setup (method), 32, 35, 85, 246, 248, 493, 519, 520, 525, 533
setValue (method), 52, 152, 500, 527
setWeightingFactors (method), 257, 485
Shah function, 327
Shannon, 330
Shape (class), 534, 541, 542
shape
 feature, 222
 number, 220, 221, 236
sharpen (method), 497
Shear (class), 418
shearing, 377
shift property, 324
shiftKeyDown (method), 517
short (type), 491
ShortProcessor (class), 253, 472, 491
ShortStatistics (class), 494
show (method), 52, 153, 247, 501, 502, 508, 514, 518, 530
showDialog (method), 86, 507, 518, 526
showMessage (method), 515, 526
showMessageWithCancel (method), 515
showProcessor (method), 530
showProgress (method), 517
showStatus (method), 517
signal space, 100, 320, 338
similarity, 324
sin (method), 461
Sinc function, 321, 394, 404
sine function, 322
 one-dimensional, 314
sine transform, 367
size (method), 534
skeletonization, 195
skeletonize (method), 195

slice, 506
smooth (method), 497
 smoothing filter, 89, 92
 snapshot array, 489
 Sobel operator, 120, 129, 497
 software, 28
solve (method), 422
sort (method), 109, 150, 300, 466, 529
 sorting arrays, 466
 source-to-target mapping, 390
spaceBarDown (method), 517
 spatial sampling, 9
 special image, 13
 spectrum, 313–373
 spherical mapping, 388
 spline
 cardinal, 398, 400
 Catmull-Rom, 399, 400, 402
 cubic, 400, 401
 cubic B-, 400–402, 428
 interpolation, 399
sqr (method), 81, 495, 531
sqrt (method), 81, 461, 495
 square window, 358
 sRGB, 79, 256, 257, 283, 285, 286, 288, 292
 ambient lighting, 280
 grayscale conversion, 287
 white point, 280
Stack (class), 202, 203
 stack, 200, 246, 504, 505, 508, 512, 520
STACK_REQUIRED (constant), 520
StackStatistics (class), 494
 standard deviation, 51
 standard illuminant, 279, 288
 storing images, 480
Stroke (class), 542
 structure, 451
 structuring element, 175, 176, 180, 181, 187, 191
SUBTRACT (constant), 82, 497
super (method), 541
 super-Gaussian window, 355, 356
SUPPORTS_MASKING (constant), 520

T

tan (method), 461
 tangent function, 462

target-to-source mapping, 387, 391, 415
 template matching, 429–431, 440
 temporal sampling, 9
TextRoi (class), 475, 510
 TGA format, 22
 thin lens, 8
 thinning, 195
 thread, 522
 three-point mapping, 378
 threshold, 55, 129, 163
threshold (method), 55, 496
 TIFF, 15, 19, 21, 23, 31, 219, 246, 248
 time unit, 315
toArray (method), 151, 530
toCIEXYZ (method), 292–295, 297
toDegrees (method), 461
 topological property, 234
toRadians (method), 461
 tracking, 139
 transform pair, 320
TransformJ (package), 413
translate (method), 535, 542
Translation (class), 418
 translation, 233, 377
 transparency, 83, 244, 252
 tree, 202
 true color image, 13, 17, 241, 243, 244
 truncate function, 453, 459
 truncation, 82
 tuple, 451
 twirl mapping, 387
TwirlMapping (class), 422
 type cast, 54, 458, 488
TYPE_Lab (constant), 297
TypeConverter (class), 252

U

undercolor-removal function, 273
 undo array, 489
 uniform distribution, 51
 unit square, 386
unlock (method), 522
 unsharp masking, 133–137
UnsharpMask (class), 136
unsharpMask (method), 137
unsigned byte (type), 459
updateAndDraw (method), 36, 52, 249, 484, 502

-
- `updateAndRepaintWindow` (method), 502
 - user interaction, 515–517
 - V**
 - variance, 434
 - `Vector` (class), 149, 463, 468, 529
 - vector, 451
 - graphic, 14
 - graphics, 216
 - `versionLessThan` (method), 523
 - viewing angle, 280
 - W**
 - `wait` (method), 517
 - Walsh transform, 372
 - warping, 387
 - `wasCanceled` (method), 86, 507, 518, 526
 - wave number, 334, 344, 349, 368
 - wavelet, 373
 - Website for this book, 35
 - `white` (constant), 541
 - white point, 258, 279, 282
 - D50, 279, 292
 - D65, 280, 284
 - window management, 521
 - windowed matching, 439
 - windowing, 352
 - windowing function, 354–357
 - Bartlett, 355, 357, 358
 - cosine², 357, 358
 - elliptical, 355, 356
 - Gaussian, 355, 356, 358
 - Hanning, 355, 357, 358
 - Parzen, 355, 357, 358
 - rectangular pulse, 356
 - super-Gaussian, 355, 356
 - `WindowManager` (class), 86, 249, 475, 507, 521
 - WMF format, 15
 - `write` (method), 515
 - X**
 - XBM/XPM format, 22
 - `XOR` (constant), 497
 - `xor` (method), 496
 - XYZ scaling, 289
 - Y**
 - YCbCr, 272
 - YCbCr, 270
 - YIQ, 269, 272
 - YUV, 255, 268, 270, 272
 - Z**
 - ZIP, 15
 - zoom, 498