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

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:

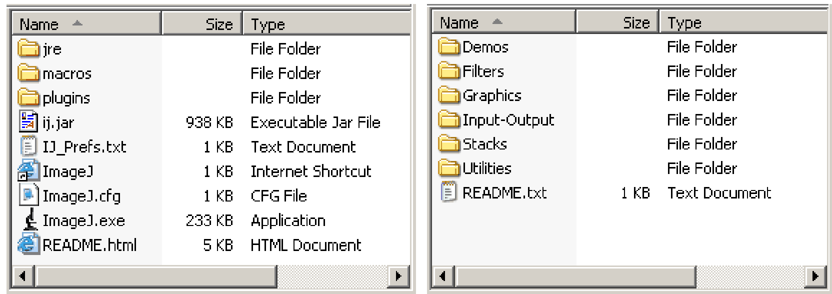
`<ij>/jre`
: A complete Java runtime environment, the "Java Virtual Machine" (JVM). This is required for actually executing Java programs.

`<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 API[5] 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 Microsystems[6]) 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.

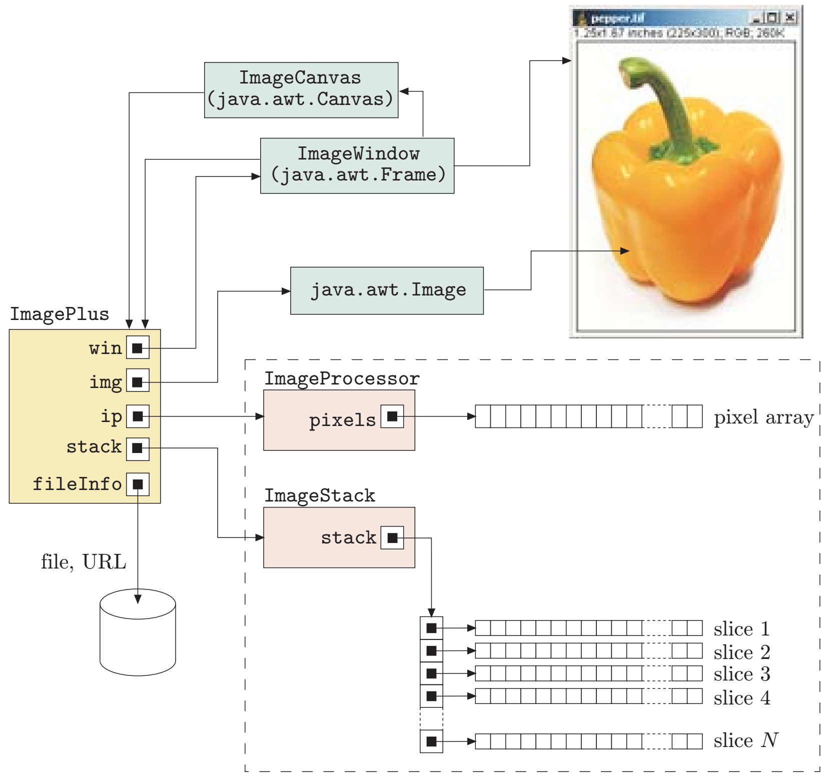


Fig. C.2
Internal representation of images and image stacks in ImageJ (simplified). `ImageProcessor` and `ImageStack` objects contain the actual pixel data of images and image sequences (stacks), respectively. A single image is stored in memory as a one-dimensional array of numerical pixel values. Image 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.

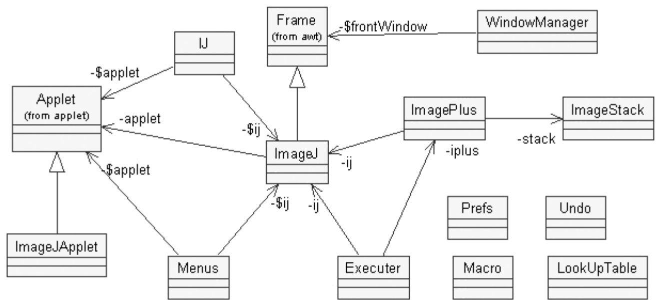


Fig. C.3
Class diagram for ImageJ package `ij`.

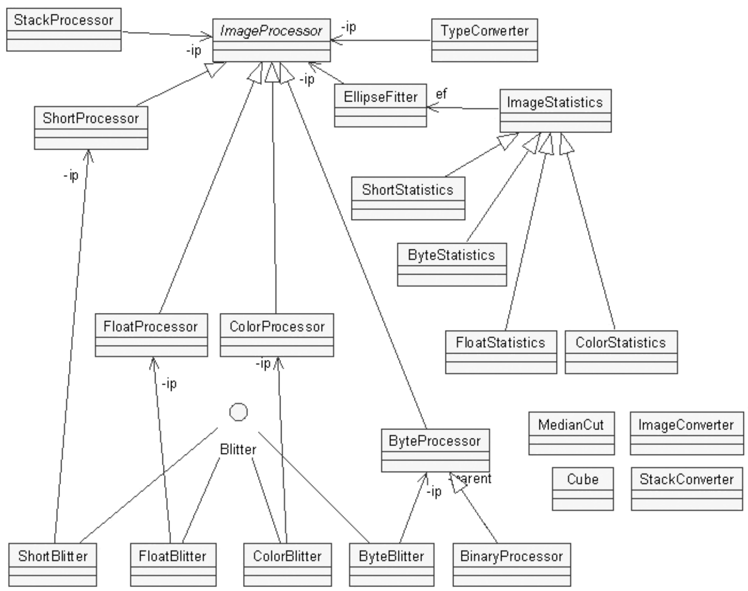


Fig. C.4
Class diagram for the ImageJ package `ij.process`.

`FloatProcessor` (class)
: Image processor for 32-bit floating-point (`float`) images.

`ColorProcessor` (class)
: Image processor for 32-bit color (3×8 bits RGB plus 8-bit α-channel) images.

C.2.4 Plugins (Packages `ij.plugin`, `ij.plugin.filter`)

`PlugIn` (interface)
: Interface for generic ImageJ plugins that import or display images or plugins that do not use any images.

`PlugInFilter` (interface)
: Interface for ImageJ plugins that process (and usually modify) images.

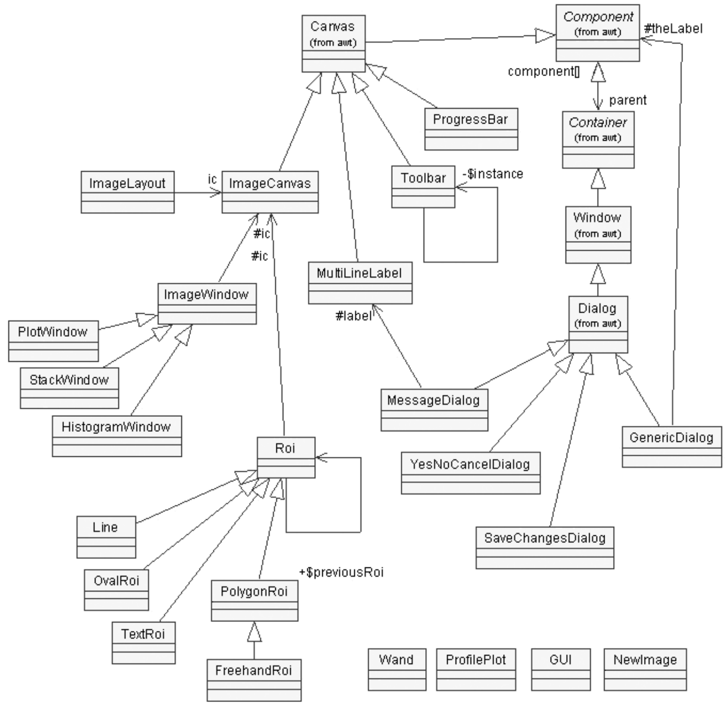


Fig. C.5
Class diagram for the ImageJ package `ij.gui`.

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

ImageJ's GUI[8] 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)
: Defines a rectangular "region of interest" (ROI) and is the superclass of other ROI classes: `Line`, `OvalRoi`, `PolygonRoi` (with subclass `FreehandRoi`), and `TextRoi`.

C.2.6 Window Management (Package `ij`)

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

C.2.7 Utility Classes (Package `ij`)

`IJ` (class)
: Provides a set of static utility methods, including methods for selecting, creating, opening, and saving images and obtaining information about the operating environment (Sec. C.21.2).

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

The `ij.io` package contains classes for reading (loading) and writing images from and to files in various image formats and encodings (Fig. C.6).

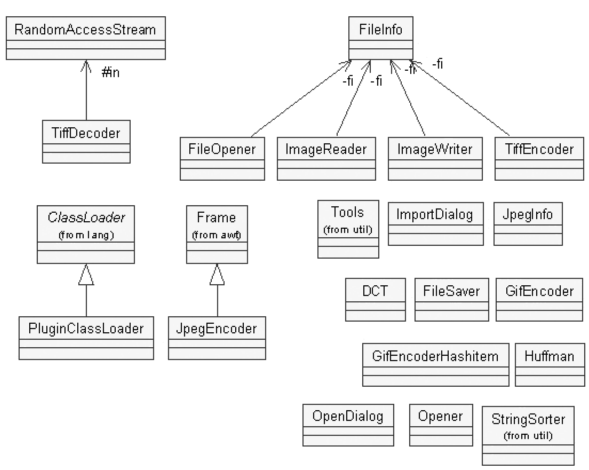


Fig. C.6
Class diagram for ImageJ package `ij.io`.

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* × *height*.

ImageStack (int *width*, int *height*, ColorModel *cm*)
: Constructor method: creates a new, empty image stack of size *width* × *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* × *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 applied. 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` *`wr`*`, double` *`wg`*`, double` *`wb`*`)`
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 αRGB 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*`,` *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 αRGB 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 $[0 \ldots 255]$ 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:

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

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 $[0 \ldots 255]$ (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:

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.

```
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
```

`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 HSB[9] (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 $[0 \ldots 255]$ when converted to 8-bit images and to $[0 \ldots 65{,}535]$ 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

C.10.1 `ImageProcessor` (Class)

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 Graphic Operations

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* $\times$ 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* $\times$ *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* $\times$ 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 αRGB 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 $1 \leq index \leq N$, 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()`.

```
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
```

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).

`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 lookuptable 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 amin, double amax)`
Sets the parameters $a_{\min}$ and $a_{\max}$ to the specified values. The image is displayed by mapping the pixel values in the range $[a_{\min} \ldots a_{\max}]$ to screen values in the range $[0 \ldots 255]$.

`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 *n*th slice from the stack, where $1 \leq n \leq$ `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 arrays. 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 *n*th slice of the stack, where $1 \leq n \leq$ `getsize()`. No pixel data are duplicated.

`ImageProcessor getProcessor (int n)`
: Creates and returns an `ImageProcessor` for the *n*th slice of the stack, where $1 \leq n \leq$ `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 *n*th slice, where $1 \leq n \leq$ `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 $1 \leq n \leq$ `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 $1 \leq n \leq$ `getsize()`. No pixel data are duplicated.

`void setSliceLabel (String label, int n)`
Assigns the title *label* to the *n*th slice, where $1 \leq n \leq$ `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 $1 \ldots N$ (`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`)

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.

```
import ij.IJ;
import ij.ImagePlus;
import ij.ImageStack;
import ij.WindowManager;
import ij.gui.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;

public class Alpha_Blending_Stack implements PlugInFilter {
  static int nFrames = 10;
  ImagePlus fgIm;  // foreground image (chosen interactively)

  public int setup(String arg, ImagePlus imp) {
    return DOES_8G;}

  boolean runDialog() {
    // get list of open images
    int[] windowList = WindowManager.getIDList();
    if(windowList==null) {
      IJ.noImage();
      return false;
    }
    String[] windowTitles = new String[windowList.length];
    for (int i = 0; i < windowList.length; i++) {
      ImagePlus imp = WindowManager.getImage(windowList[i]);
      if (imp != null)
        windowTitles[i] = imp.getShortTitle();
      else
        windowTitles[i] = "untitled";
    }
    GenericDialog gd = new GenericDialog("Alpha Blending");
    gd.addChoice("Foreground image:",
        windowTitles, windowTitles[0]);
    gd.addNumericField("Frames:", nFrames, 0);
    gd.showDialog();
    if (gd.wasCanceled())
      return false;
    else {
      int img2Index = gd.getNextChoiceIndex();
      fgIm = WindowManager.getImage(windowList[img2Index]);
      nFrames = (int) gd.getNextNumber();
      if (nFrames < 2)
        nFrames = 2;
      return true;
    }
  } // continued...
```

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

```
  // class Alpha_Blending_Stack (continued)

  public void run(ImageProcessor bgIp) {
    // bgIp = background image

    if(runDialog()) { //open dialog box (returns false if cancelled)
      int w = bgIp.getWidth();
      int h = bgIp.getHeight();

      // prepare foreground image
      ImageProcessor fgIp =
          fgIm.getProcessor().convertToByte(false);
      ImageProcessor fgTmpIp = bgIp.duplicate();

      // create image stack
      ImagePlus movie =
        NewImage.createByteImage("Movie",w,h,nFrames,0);
      ImageStack stack = movie.getStack();

      // loop over stack frames
      for (int i=0; i<nFrames; i++) {
        // transparency of foreground image
        double iAlpha = 1.0 - (double)i/(nFrames-1);
        ImageProcessor iFrame = stack.getProcessor(i+1);

        // copy background image to frame i
        iFrame.insert(bgIp,0,0);
        iFrame.multiply(iAlpha);

        // copy foreground image and make transparent
        fgTmpIp.insert(fgIp,0,0);
        fgTmpIp.multiply(1-iAlpha);

        // add foreground image frame i
        ByteBlitter blitter =
            new ByteBlitter((ByteProcessor)iFrame);
        blitter.copyBits(fgTmpIp,0,0,Blitter.ADD);
      }

      // display movie (image stack)
      movie.show();
    }
  }

} // end of class Alpha_Blending_Stack
```

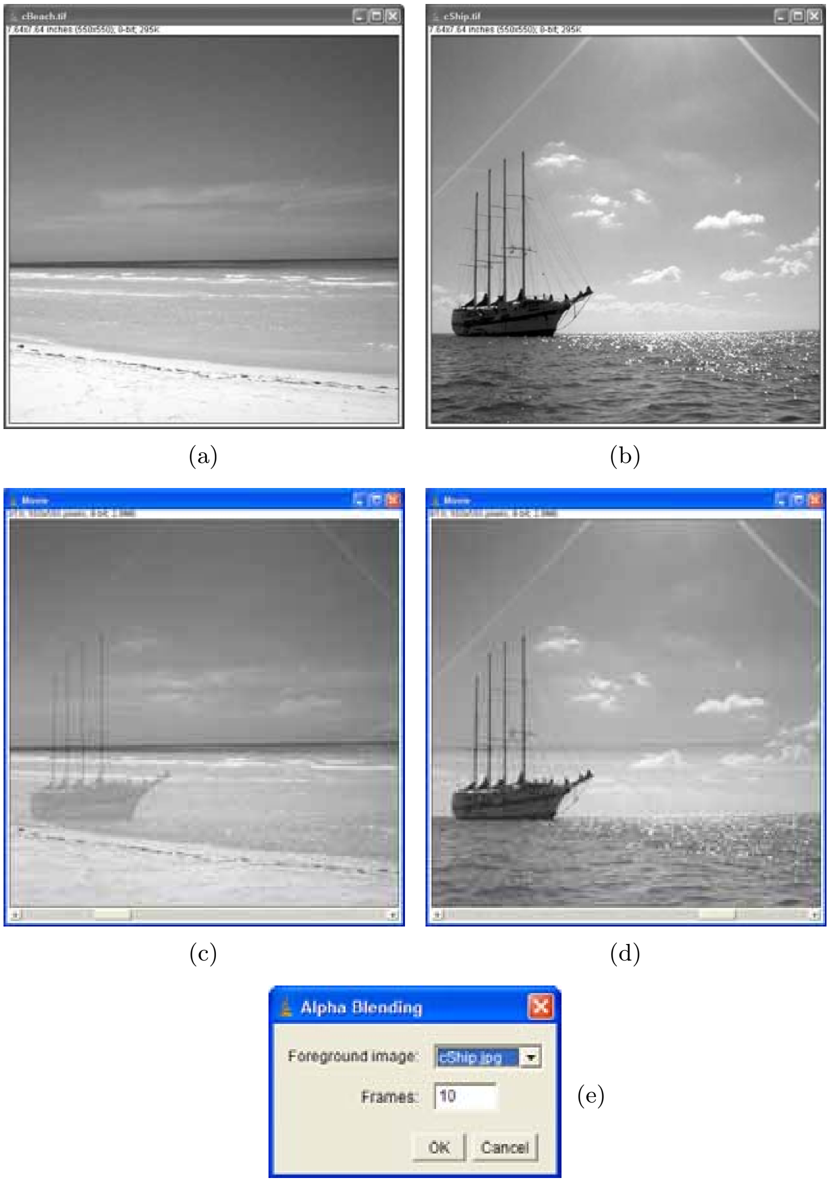


Fig. C.7
Alpha blending example using an image stack (results of Progs. C.4 and C.5). Original images: foreground image (a) and background image (b); frames 3 and 6 of the created image stack (c, d). Note the horizontal "slider" button at the window's bottom for navigating through the stack. Dialog window 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)

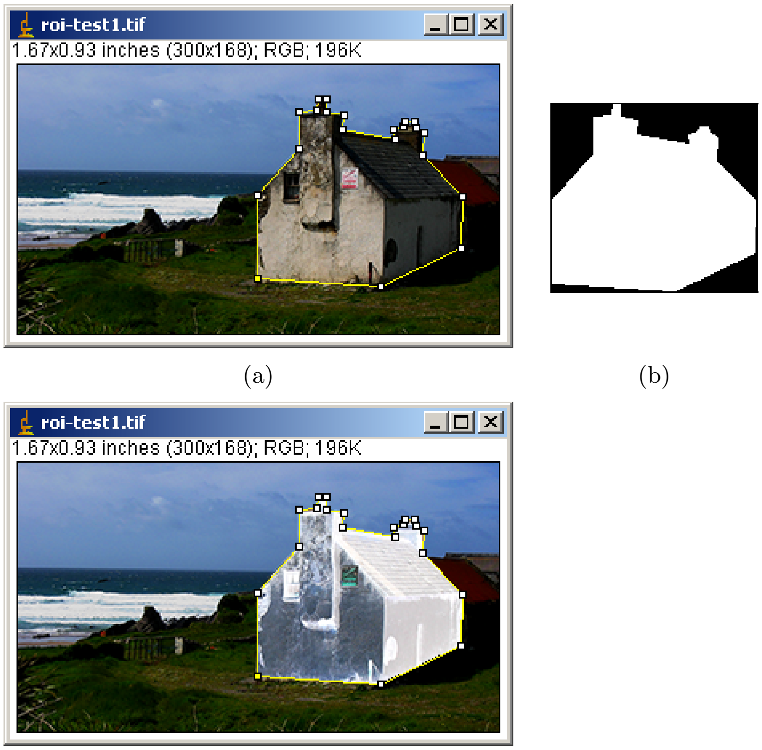
Only rectangular ROIs are applicable to image stacks:

`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)

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 currently 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 currently 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 `imp` (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 $0 \leq$ *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.

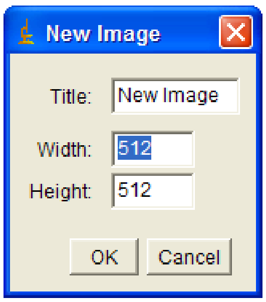
`static ImagePlus getImage ()`
: Returns a reference to the *active* image, i. e., the `ImagePlus` object currently selected by the user.

`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.




```java
import ij.ImagePlus;
import ij.gui.GenericDialog;
import ij.gui.NewImage;
import ij.plugin.PlugIn;

public class Generic_Dialog_Example implements PlugIn {
  static String title = "New Image";
  static int width = 512;
  static int height = 512;

  public void run(String arg) {
    GenericDialog gd = new GenericDialog("New Image");
    gd.addStringField("Title:", title);
    gd.addNumericField("Width:", width, 0);
    gd.addNumericField("Height:", height, 0);
    gd.showDialog();
    if (gd.wasCanceled())
      return;
    title = gd.getNextString();
    width = (int) gd.getNextNumber();
    height = (int) gd.getNextNumber();

    ImagePlus imp = NewImage.createByteImage(
      title, width, height, 1, NewImage.FILL_WHITE);
    imp.show();
  }

} // 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).

C.19.2 `PlugInFilter` (Interface)

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 returns `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 `setTempCurrentImage()`), 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)

```
import harris.HarrisCornerDetector;
import ij.IJ;
import ij.ImagePlus;
import ij.gui.GenericDialog;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;

public class Harris_Corner_Plugin implements PlugInFilter {
   ImagePlus im;
   static float alpha = HarrisCornerDetector.DEFAULT_ALPHA;
   static int threshold = HarrisCornerDetector.
        DEFAULT_THRESHOLD;
   static int nmax = 0; //points to show

   public int setup(String arg, ImagePlus im) {
     this.im = im;
```

```
        if (arg.equals("about")) {
            showAbout();
            return DONE;
        }
        return DOES_8G + NO_CHANGES;
    }

    public void run(ImageProcessor ip) {
        if (!showDialog()) return; //dialog canceled or error
        HarrisCornerDetector hcd =
            new HarrisCornerDetector(ip,alpha,threshold);
        hcd.findCorners();
        ImageProcessor result = hcd.showCornerPoints(ip);
        ImagePlus win =
            new ImagePlus("Corners from " + im.getTitle(),
                result);
        win.show();
    }

    void showAbout() {
        String cn = getClass().getName();
        IJ.showMessage("About "+cn+" ...",
            "Harris Corner Detector");
    }

    private boolean showDialog() {
      // display dialog, and return false if canceled or in error.
      GenericDialog dlg = new GenericDialog("Harris Corner
          Detector", IJ.getInstance());
      float def_alpha = HarrisCornerDetector.DEFAULT_ALPHA;
      dlg.addNumericField("Alpha (default: "+def_alpha+")",
          alpha, 3);
      int def_threshold = HarrisCornerDetector.
          DEFAULT_THRESHOLD;
      dlg.addNumericField("Threshold (default: "+def_threshold+
          ")", threshold, 0);
      dlg.addNumericField("Max. points (0 = show all)", nmax,
          0);
      dlg.showDialog();
      if(dlg.wasCanceled())
        return false;
      if(dlg.invalidNumber()) {
        IJ.showMessage("Error", "Invalid input number");
        return false;
      }
      alpha = (float) dlg.getNextNumber();
      threshold = (int) dlg.getNextNumber();
      nmax = (int) dlg.getNextNumber();
      return true;
    }
} // end of class Harris_Corner_Plugin
```

D.1.2 File Corner (Class)

```java
package harris;
import ij.process.ImageProcessor;

class Corner implements Comparable {
  int u;
  int v;
  float q;

  Corner (int u, int v, float q) {
    this.u = u;
    this.v = v;
    this.q = q;
  }

  public int compareTo (Object obj) {
    // used for sorting corners by corner strength q
    Corner c2 = (Corner) obj;
    if (this.q > c2.q) return -1;
    if (this.q < c2.q) return 1;
    else return 0;
  }

  double dist2 (Corner c2) {
    // returns the squared distance between this corner and corner c2
    int dx = this.u - c2.u;
    int dy = this.v - c2.v;
    return (dx*dx)+(dy*dy);
  }

  void draw(ImageProcessor ip) {
    // draw this corner as a black cross in ip
    int paintvalue = 0; // black
    int size = 2;
    ip.setValue(paintvalue);
    ip.drawLine(u-size,v,u+size,v);
    ip.drawLine(u,v-size,u,v+size);
  }

} // end of class Corner
```

D.1.3 File HarrisCornerDetector (Class)

```java
package harris;
import ij.IJ;
import ij.ImagePlus;
import ij.plugin.filter.Convolver;
import ij.process.Blitter;
import ij.process.ByteProcessor;
```

```java
import ij.process.FloatProcessor;
import ij.process.ImageProcessor;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Vector;

public class HarrisCornerDetector {

  public static final float DEFAULT_ALPHA = 0.050f;
  public static final int DEFAULT_THRESHOLD = 20000;
  float alpha = DEFAULT_ALPHA;
  int threshold = DEFAULT_THRESHOLD;
  double dmin = 10;
  final int border = 20;

  // filter kernels (1D part of separable 2D filters)
  final float[] pfilt = {0.223755f,0.552490f,0.223755f};
  final float[] dfilt = {0.453014f,0.0f,-0.453014f};
  final float[] bfilt = {0.01563f,0.09375f,0.234375f,0.3125f
       ,0.234375f,0.09375f,0.01563f};
              // = [1,6,15,20,15,6,1]/64
  ImageProcessor ipOrig;
  FloatProcessor A;
  FloatProcessor B;
  FloatProcessor C;
  FloatProcessor Q;
  List<Corner> corners;

  HarrisCornerDetector(ImageProcessor ip) {
    this.ipOrig = ip;
  }

  public HarrisCornerDetector(ImageProcessor ip,
          float alpha, int threshold)
  {
    this.ipOrig = ip;
    this.alpha = alpha;
    this.threshold = threshold;
  }

  public void findCorners() {
    makeDerivatives();
    makeCrf(); //corner response function (CRF)
    corners = collectCorners(border);
    corners = cleanupCorners(corners);
  }

  void makeDerivatives() {
    FloatProcessor Ix =
          (FloatProcessor) ipOrig.convertToFloat();
```

```
  FloatProcessor Iy =
          (FloatProcessor) ipOrig.convertToFloat();

  Ix = convolve1h(convolve1h(Ix,pfilt),dfilt);
  Iy = convolve1v(convolve1v(Iy,pfilt),dfilt);

  A = sqr((FloatProcessor) Ix.duplicate());
  A = convolve2(A,bfilt);

  B = sqr((FloatProcessor) Iy.duplicate());
  B = convolve2(B,bfilt);

  C = mult((FloatProcessor)Ix.duplicate(),Iy);
  C = convolve2(C,bfilt);
}

void makeCrf() { // corner response function (CRF)
  int w = ipOrig.getWidth();
  int h = ipOrig.getHeight();
  Q = new FloatProcessor(w,h);
  float[] Apix = (float[]) A.getPixels();
  float[] Bpix = (float[]) B.getPixels();
  float[] Cpix = (float[]) C.getPixels();
  float[] Qpix = (float[]) Q.getPixels();
  for (int v=0; v<h; v++) {
    for (int u=0; u<w; u++) {
      int i = v*w+u;
      float a = Apix[i], b = Bpix[i], c = Cpix[i];
      float det = a*b-c*c;
      float trace = a+b;
      Qpix[i] = det - alpha * (trace * trace);
    }
  }
}

List<Corner> collectCorners(int border) {
  List<Corner> cornerList = new Vector<Corner>(1000);
  int w = Q.getWidth();
  int h = Q.getHeight();
  float[] Qpix = (float[]) Q.getPixels();
  for (int v=border; v<h-border; v++){
    for (int u=border; u<w-border; u++) {
      float q = Qpix[v*w+u];
      if (q>threshold && isLocalMax(Q,u,v)) {
        Corner c = new Corner(u,v,q);
        cornerList.add(c);
      }
    }
  }
  Collections.sort(cornerList);
  return cornerList;
```

```
}

List<Corner> cleanupCorners(List<Corner> corners) {
  double dmin2 = dmin*dmin;
  Corner[] cornerArray = new Corner[corners.size()];
  cornerArray = corners.toArray(cornerArray);
  List<Corner> goodCorners =
      new Vector<Corner>(corners.size());
  for (int i=0; i<cornerArray.length; i++){
    if (cornerArray[i] != null){
      Corner c1 = cornerArray[i];
      goodCorners.add(c1);
      // delete all remaining corners close to c
      for (int j=i+1; j<cornerArray.length; j++){
        if (cornerArray[j] != null){
          Corner c2 = cornerArray[j];
          if (c1.dist2(c2)<dmin2)
            cornerArray[j] = null; //delete corner
        }
      }
    }
  }
  return goodCorners;
}

void printCornerPoints(List<Corner> crf) {
  int i = 0;
  for (Corner ipt: crf){
    IJ.write((i++) + ": " + (int)ipt.q + " " + ipt.u + " " +
         ipt.v);
  }
}

public ImageProcessor showCornerPoints(ImageProcessor ip) {
  ByteProcessor ipResult = (ByteProcessor)ip.duplicate();
  // change background image contrast and brightness
  int[] lookupTable = new int[256];
  for (int i=0; i<256; i++){
    lookupTable[i] = 128 + (i/2);
  }
  ipResult.applyTable(lookupTable);
  // draw corners:
  for (Corner c: corners) {
    c.draw(ipResult);
  }
  return ipResult;
}

void showProcessor(ImageProcessor ip, String title) {
  ImagePlus win = new ImagePlus(title,ip);
  win.show();
```

```
}

// utility methods for float processors —

static FloatProcessor convolve1h
              (FloatProcessor p, float[] h) {
  Convolver conv = new Convolver();
  conv.setNormalize(false);
  conv.convolve(p, h, 1, h.length);
  return p;
}

static FloatProcessor convolve1v
              (FloatProcessor p, float[] h) {
  Convolver conv = new Convolver();
  conv.setNormalize(false);
  conv.convolve(p, h, h.length, 1);
  return p;
}

static FloatProcessor convolve2
              (FloatProcessor p, float[] h) {
  convolve1h(p,h);
  convolve1v(p,h);
  return p;
}

static FloatProcessor sqr (FloatProcessor fp1) {
  fp1.sqr();
  return fp1;
}

static FloatProcessor mult (FloatProcessor fp1,
    FloatProcessor fp2) {
  int mode = Blitter.MULTIPLY;
  fp1.copyBits(fp2, 0, 0, mode);
  return fp1;
}

static boolean isLocalMax (FloatProcessor fp,int u,int v) {
  int w = fp.getWidth();
  int h = fp.getHeight();
  if (u<=0 || u>=w-1 || v<=0 || v>=h-1)
    return false;
  else {
    float[] pix = (float[]) fp.getPixels();
    int i0 = (v-1)*w+u, i1 = v*w+u, i2 = (v+1)*w+u;
    float cp = pix[i1];
    return
      cp > pix[i0-1] && cp > pix[i0] && cp > pix[i0+1] &&
      cp > pix[i1-1] &&             cp > pix[i1+1] &&
```

```
        cp > pix[i2-1] && cp > pix[i2] && cp > pix[i2+1] ;
    }
  }

} // 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)

```
import java.util.List;
import regions.BinaryRegion;
import regions.RegionLabeling;
import contours.Contour;
import contours.ContourOverlay;
import contours.ContourTracer;

import ij.IJ;
import ij.ImagePlus;
import ij.gui.ImageWindow;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;

// This plugin implements the combined contour tracing and
// component labeling algorithm as described in [22].
// It uses the ContourTracer class to create lists of points
// representing the internal and external contours of each region in
// the binary image. Instead of drawing directly into the image,
// we make use of ImageJ's ImageCanvas to draw the contours
// in a separate layer on top of the image. It illustrates how to use
// the Java2D API to draw the polygons and scale and transform
// them to match ImageJ's zooming.


```

```
public class Contour_Tracing_Plugin implements PlugInFilter
{
  ImagePlus origImage = null;
  String origTitle = null;
  static boolean verbose = true;

  public int setup(String arg, ImagePlus im) {
    origImage = im;
    origTitle = im.getTitle();
    RegionLabeling.setVerbose(verbose);
    return DOES_8G + NO_CHANGES;
  }

  public void run(ImageProcessor ip) {
    ImageProcessor ip2 = ip.duplicate();

    // label regions and trace contours
    ContourTracer tracer = new ContourTracer(ip2);

    // extract contours and regions
    List<Contour> outerContours = tracer.getOuterContours();
    List<Contour> innerContours = tracer.getInnerContours();
    List<BinaryRegion> regions = tracer.getRegions();
    if (verbose) printRegions(regions);

    // change lookup table to show gray regions
    ip2.setMinAndMax(0,512);
    // create an image with overlay to show the contours
    ImagePlus im2 = new ImagePlus("Contours of " + origTitle,
        ip2);
    ContourOverlay cc = new ContourOverlay(im2, outerContours,
        innerContours);
    new ImageWindow(im2, cc);
  }

  void printRegions(List<BinaryRegion> regions) {
    for (BinaryRegion r: regions) {
      IJ.write("" + r);
    }
  }

} // end of class Contour_Tracing_Plugin
```

D.2.2 `Contour` (Class)

```
package contours;
import ij.IJ;
import java.awt.Point;
import java.awt.Polygon;
import java.awt.Shape;
```

```java
import java.awt.geom.Ellipse2D;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Contour {
  static int INITIAL_SIZE = 50;
  int label;
  List<Point> points;

  Contour (int label, int size) {
    this.label = label;
    points = new ArrayList<Point>(size);
  }

  Contour (int label) {
    this.label = label;
    points = new ArrayList<Point>(INITIAL_SIZE);
  }

  void addPoint (Point n) {
    points.add(n);
  }

  Shape makePolygon() {
    int m = points.size();
    if (m>1) {
      int[] xPoints = new int[m];
      int[] yPoints = new int[m];
      int k = 0;
      Iterator<Point> itr = points.iterator();
      while (itr.hasNext() && k < m) {
        Point cpt = itr.next();
        xPoints[k] = cpt.x;
        yPoints[k] = cpt.y;
        k = k + 1;
      }
      return new Polygon(xPoints, yPoints, m);
    }
    else { // use circles for isolated pixels
      Point cpt = points.get(0);
      return new Ellipse2D.Double
          (cpt.x-0.1, cpt.y-0.1, 0.2, 0.2);
    }
  }

  static Shape[] makePolygons(List<Contour> contours) {
    if (contours == null)
      return null;
    else {
      Shape[] pa = new Shape[contours.size()];
```

```
    int i = 0;
    for (Contour c: contours) {
      pa[i] = c.makePolygon();
      i = i + 1;
    }
    return pa;
  }
}

void moveBy (int dx, int dy) {
  for (Point pt: points) {
    pt.translate(dx,dy);
  }
}

static void moveContoursBy
        (List<Contour> contours, int dx, int dy) {
  for (Contour c: contours) {
    c.moveBy(dx, dy);
  }
}

} // end of class Contour
```

D.2.3 BinaryRegion (Class)

```
package regions;
import java.awt.Rectangle;
import java.awt.geom.Point2D;

public class BinaryRegion {
  int label;
  int numberOfPixels = 0;
  double xc = Double.NaN;
  double yc = Double.NaN;
  int left = Integer.MAX_VALUE;
  int right = -1;
  int top = Integer.MAX_VALUE;
  int bottom = -1;

  int x_sum = 0;
  int y_sum = 0;
  int x2_sum = 0;
  int y2_sum = 0;

  public BinaryRegion(int id){
    this.label = id;
  }

  public int getSize() {
```

```
    return this.numberOfPixels;
  }

  public Rectangle getBoundingBox() {
    if (left == Integer.MAX_VALUE)
      return null;
    else
      return new Rectangle
        (left, top, right-left+1, bottom-top+1);
  }

  public Point2D.Double getCenter(){
    if (Double.isNaN(xc))
      return null;
    else
      return new Point2D.Double(xc, yc);
  }

  public void addPixel(int x, int y){
    numberOfPixels = numberOfPixels + 1;
    x_sum = x_sum + x;
    y_sum = y_sum + y;
    x2_sum = x2_sum + x*x;
    y2_sum = y2_sum + y*y;
    if (x<left) left = x;
    if (y<top) top = y;
    if (x>right) right = x;
    if (y>bottom) bottom = y;
  }

  public void update(){
    if (numberOfPixels > 0){
      xc = x_sum / numberOfPixels;
      yc = y_sum / numberOfPixels;
    }
  }

} // end of class BinaryRegion
```

D.2.4 ContourTracer (Class)

```
package contours;
import java.awt.Point;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import regions.BinaryRegion;
import ij.IJ;
import ij.process.ImageProcessor;

```

```
public class ContourTracer {
  static final byte FOREGROUND = 1;
  static final byte BACKGROUND = 0;
  static boolean beVerbose = true;

  List<Contour> outerContours = null;
  List<Contour> innerContours = null;
  List<BinaryRegion> allRegions = null;
  int regionId = 0;

  ImageProcessor ip = null;
  int width;
  int height;
  byte[][] pixelArray;
  int[][] labelArray;

  // label values in labelArray can be:
  //   0 ... unlabeled
  // −1 ... previously visited background pixel
  // > 0 ... a valid label

  // constructor method
  public ContourTracer (ImageProcessor ip) {
    this.ip = ip;
    this.width = ip.getWidth();
    this.height = ip.getHeight();
    makeAuxArrays();
    findAllContours();
    collectRegions();
  }

  public static void setVerbose(boolean verbose) {
    beVerbose = verbose;
  }

  public List<Contour> getOuterContours() {
    return outerContours;
  }

  public List<Contour> getInnerContours() {
    return innerContours;
  }

  public List<BinaryRegion> getRegions() {
    return allRegions;
  }

  // nonpublic methods

  void makeAuxArrays() {
    int h = ip.getHeight();
```

```
  int w = ip.getWidth();
  pixelArray = new byte[h+2][w+2];
  labelArray = new int[h+2][w+2];
  // initialize auxiliary arrays
  for (int v = 0; v < h+2; v++) {
    for (int u = 0; u < w+2; u++) {
      if (ip.get(u-1,v-1) == 0)
        pixelArray[v][u] = BACKGROUND;
      else
        pixelArray[v][u] = FOREGROUND;
    }
  }
}

Contour traceOuterContour (int cx, int cy, int label) {
  Contour cont = new Contour(label);
  traceContour(cx, cy, label, 0, cont);
  return cont;
}

Contour traceInnerContour(int cx, int cy, int label) {
  Contour cont = new Contour(label);
  traceContour(cx, cy, label, 1, cont);
  return cont;
}

// trace one contour starting at (xS,yS) in direction dS
Contour traceContour (int xS, int yS, int label, int dS,
    Contour cont) {
  int xT, yT; // T = successor of starting point (xS,yS)
  int xP, yP; // P = previous contour point
  int xC, yC; // C = current contour point
  Point pt = new Point(xS, yS);
  int dNext = findNextPoint(pt, dS);
  cont.addPoint(pt);
  xP = xS; yP = yS;
  xC = xT = pt.x;
  yC = yT = pt.y;

  boolean done = (xS==xT && yS==yT); // true if isolated pixel

  while (!done) {
    labelArray[yC][xC] = label;
    pt = new Point(xC, yC);
    int dSearch = (dNext + 6) % 8;
    dNext = findNextPoint(pt, dSearch);
    xP = xC; yP = yC;
    xC = pt.x; yC = pt.y;
    // are we back at the starting position?
    done = (xP==xS && yP==yS && xC==xT && yC==yT);
    if (!done) {
```

```
      cont.addPoint(pt);
    }
  }
  return cont;
}

int findNextPoint (Point pt, int dir) {
  // starts at Point pt in direction dir, returns the
  // final tracing direction, and modifies pt
  final int[][] delta = {
    { 1,0}, { 1, 1}, {0, 1}, {-1, 1},
    {-1,0}, {-1,-1}, {0,-1}, { 1,-1}};
  for (int i = 0; i < 7; i++) {
    int x = pt.x + delta[dir][0];
    int y = pt.y + delta[dir][1];
    if (pixelArray[y][x] == BACKGROUND) {
      // mark surrounding background pixels
      labelArray[y][x] = -1;
      dir = (dir + 1) % 8;
    }
    else { // found a nonbackground pixel
      pt.x = x; pt.y = y;
      break;
    }
  }
  return dir;
}

void findAllContours() {
  outerContours = new ArrayList<Contour>(50);
  innerContours = new ArrayList<Contour>(50);
  int label = 0;  // current label

  // scan top to bottom, left to right
  for (int v = 1; v < pixelArray.length-1; v++) {
    label = 0; // no label
    for (int u = 1; u < pixelArray[v].length-1; u++) {

      if (pixelArray[v][u] == FOREGROUND) {
        if (label != 0) { // keep using the same label
          labelArray[v][u] = label;
        }
        else {
          label = labelArray[v][u];
          if (label == 0) {
            // unlabeled—new outer contour
            regionId = regionId + 1;
            label = regionId;
            Contour oc = traceOuterContour(u, v, label);
            outerContours.add(oc);
            labelArray[v][u] = label;
```

```
          }
        }
      }
      else { // background pixel
        if (label != 0) {
          if (labelArray[v][u] == 0) {
            // unlabeled—new inner contour
            Contour ic = traceInnerContour(u-1, v, label);
            innerContours.add(ic);
          }
          label = 0;
        }
      }
    }
  }
  // shift back to original coordinates
  Contour.moveContoursBy (outerContours, -1, -1);
  Contour.moveContoursBy (innerContours, -1, -1);
}


// creates a container of BinaryRegion objects
// collects the region pixels from the label image
// and computes the statistics for each region
void collectRegions() {
  int maxLabel = this.regionId;
  int startLabel = 1;
  BinaryRegion[] regionArray =
        new BinaryRegion[maxLabel + 1];
  for (int i = startLabel; i <= maxLabel; i++) {
    regionArray[i] = new BinaryRegion(i);
  }
  for (int v = 0; v < height; v++) {
    for (int u = 0; u < width; u++) {
      int lb = labelArray[v][u];
      if (lb >= startLabel && lb <= maxLabel
          && regionArray[lb]!=null) {
        regionArray[lb].addPixel(u, v);
      }
    }
  }

  // create a list of regions to return, collect nonempty regions
  List<BinaryRegion> regionList =
            new LinkedList<BinaryRegion>();
    for (BinaryRegion r: regionArray) {
      if (r != null && r.getSize()>0) {
        r.update(); // compute the statistics for this region
        regionList.add(r);
      }
    }
```

```
      allRegions = regionList;
    }

} // end of class ContourTracer
```

D.2.5 `ContourOverlay` (Class)

```
package contours;
import ij.ImagePlus;
import ij.gui.ImageCanvas;
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Polygon;
import java.awt.RenderingHints;
import java.awt.Shape;
import java.awt.Stroke;
import java.util.List;

public class ContourOverlay extends ImageCanvas {
  private static final long serialVersionUID = 1L;
  static float strokeWidth = 0.5f;
  static int capsstyle = BasicStroke.CAP_ROUND;
  static int joinstyle = BasicStroke.JOIN_ROUND;
  static Color outerColor = Color.black;
  static Color innerColor = Color.white;
  static float[] outerDashing = {strokeWidth * 2.0f,
      strokeWidth * 2.5f};
  static float[] innerDashing = {strokeWidth * 0.5f,
      strokeWidth * 2.5f};
  static boolean DRAW_CONTOURS = true;

  Shape[] outerContourShapes = null;
  Shape[] innerContourShapes = null;

  public ContourOverlay(ImagePlus im,
        List<Contour> outerCs, List<Contour> innerCs)
  {
    super(im);
    if (outerCs != null)
      outerContourShapes = Contour.makePolygons(outerCs);
    if (innerCs != null)
      innerContourShapes = Contour.makePolygons(innerCs);
  }

  public void paint(Graphics g) {
    super.paint(g);
    drawContours(g);
  }
```

```java

  // nonpublic methods

  private void drawContours(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
         RenderingHints.VALUE_ANTIALIAS_ON);

    // scale and move overlay to the pixel centers
    double mag = this.getMagnification();
    g2d.scale(mag, mag);
    g2d.translate(0.5-this.srcRect.x, 0.5-this.srcRect.y);

    if (DRAW_CONTOURS) {
      Stroke solidStroke = new BasicStroke
          (strokeWidth, capsstyle, joinstyle);
      Stroke dashedStrokeOuter = new BasicStroke
          (strokeWidth, capsstyle, joinstyle, 1.0f,
               outerDashing, 0.0f);
      Stroke dashedStrokeInner = new BasicStroke
          (strokeWidth, capsstyle, joinstyle, 1.0f,
               innerDashing, 0.0f);

      if (outerContourShapes != null)
        drawShapes(outerContourShapes, g2d, solidStroke,
             dashedStrokeOuter, outerColor);
      if (innerContourShapes != null)
        drawShapes(innerContourShapes, g2d, solidStroke,
             dashedStrokeInner, innerColor);
    }
  }

  void drawShapes(Shape[] shapes, Graphics2D g2d,
        Stroke solidStrk, Stroke dashedStrk, Color col) {
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
         RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setColor(col);
    for (int i = 0; i < shapes.length; i++) {
      Shape s = shapes[i];
      if (s instanceof Polygon)
        g2d.setStroke(dashedStrk);
      else
        g2d.setStroke(solidStrk);
      g2d.draw(s);
    }
  }

} // 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. Bronshtein 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 Trans. 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. Lindeberg. 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).

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

A

B

C

E

F

G

N

T

U