2 ImageJ

Until a few years ago, the image-processing community was a relatively small group of people who either had access to expensive commercial image-processing tools or, out of necessity, developed their own software packages. Usually such home-brew environments started out with small software components for loading and storing images from and to disk files. This was not always easy because often one had to deal with poorly documented or even proprietary file formats. An obvious (and frequent) solution was to simply design a *new* image file format from scratch, usually optimized for a particular field, application, or even a single project, which naturally led to a myriad of different file formats, many of which did not survive and are forgotten today [163, 168]. Nevertheless, writing software for *converting* between all these file formats in the 1980s and early 1990s was an important business that occupied many people. Displaying images on computer screens was similarly difficult, because there was only marginal support from operating systems, APIs, and display hardware, and capturing images or videos into a computer was close to impossible on common hardware. It thus may have taken many weeks or even months before one could do just elementary things with images on a computer and finally do some serious image processing.

Fortunately, the situation is much different today. Only a few common image file formats have survived (see also Sec. 1.5), which are readily handled by many existing tools and software libraries. Most standard APIs for C/C++, Java, and other popular programming languages already come with at least some basic support for working with images and other types of media data. While there is still much development work going on at this level, it makes our job a lot easier and, in particular, allows us to focus on the more interesting aspects of digital imaging.

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9_2

2.1 Software for Digital Imaging

Traditionally, software for digital imaging has been targeted at either *manipulating* or *processing* images, either for practitioners and designers or software programmers, with quite different requirements.

Software packages for *manipulating* images, such as Adobe Photoshop, Corel Paint, and others, usually offer a convenient user interface and a large number of readily available functions and tools for working with images interactively. Sometimes it is possible to extend the standard functionality by writing scripts or adding self-programmed components. For example, Adobe provides a special API[1] for programming Photoshop "plugins" in C++, though this is a nontrivial task and certainly too complex for nonprogrammers.

In contrast to the aforementioned category of tools, digital image *processing* software primarily aims at the requirements of algorithm and software developers, scientists, and engineers working with images, where interactivity and ease of use are not the main concerns. Instead, these environments mostly offer comprehensive and well-documented software libraries that facilitate the implementation of new image-processing algorithms, prototypes, and working applications. Popular examples are Khoros/Accusoft,[2] MatLab,[3] ImageMagick,[4] among many others. In addition to the support for conventional programming (typically with C/C++), many of these systems provide dedicated scripting languages or visual programming aides that can be used to construct even highly complex processes in a convenient and safe fashion.

In practice, image manipulation and image processing are of course closely related. Although Photoshop, for example, is aimed at image manipulation by nonprogrammers, the software itself implements many traditional image-processing algorithms. The same is true for many Web applications using server-side image processing, such as those based on ImageMagick. Thus image processing is really at the base of any image manipulation software and certainly not an entirely different category.

2.2 ImageJ Overview

ImageJ, the software that is used for this book, is a combination of both worlds discussed in the previous section. It offers a set of ready-made tools for viewing and interactive manipulation of images but can also be extended easily by writing new software components in a "real" programming language. ImageJ is implemented entirely in Java and is thus largely platform-independent, running without modification under Windows, MacOS, or Linux. Java's dynamic execution model allows new modules ("plugins") to be written as independent pieces of Java code that can be compiled, loaded, and executed "on the fly" in the running system without the need to

[1] www.adobe.com/products/photoshop/.
[2] www.accusoft.com.
[3] www.mathworks.com.
[4] www.imagemagick.org.

even restart ImageJ. This quick turnaround makes ImageJ an ideal platform for developing and testing new image-processing techniques and algorithms. Since Java has become extremely popular as a first programming language in many engineering curricula, it is usually quite easy for students to get started in ImageJ without having to spend much time learning another programming language. Also, ImageJ is freely available, so students, instructors, and practitioners can install and use the software legally and without license charges on any computer. ImageJ is thus an ideal platform for education and self-training in digital image processing but is also in regular use for serious research and application development at many laboratories around the world, particularly in biological and medical imaging.

ImageJ was (and still *is*) developed by Wayne Rasband [193] at the U.S. National Institutes of Health (NIH), originally as a substitute for its predecessor, NIH-Image, which was only available for the Apple Macintosh platform. The current version of ImageJ, updates, documentation, the complete source code, test images, and a continuously growing collection of third-party plugins can be downloaded from the ImageJ website.[5] Installation is simple, with detailed instructions available online, in Werner Bailer's programming tutorial [12], and in the authors' *ImageJ Short Reference* [40].

Wayne Rasband (right) at the 1st ImageJ Conference 2006 (picture courtesy of Marc Seil, CRP Henri Tudor, Luxembourg).

In addition to ImageJ itself there are several popular software projects that build on or extend ImageJ. This includes in particular *Fiji*[6] ("Fiji Is Just ImageJ") which offers a consistent collection of numerous plugins, simple installation on various platforms and excellent documentation. All programming examples (plugins) shown in this book should also execute in Fiji without any modifications. Another important development is *ImgLib2*, which is a generic Java API for representing and processing n-dimensional images in a consistent fashion. ImgLib2 also provides the underlying data model for *ImageJ2*,[7] which is a complete reimplementation of ImageJ.

2.2.1 Key Features

As a pure Java application, ImageJ should run on any computer for which a current Java runtime environment (JRE) exists. ImageJ comes with its own Java runtime, so Java need not be installed separately on the computer. Under the usual restrictions, ImageJ can be run as a Java "applet" within a Web browser, though it is mostly used as a stand-alone application. It is sometimes also used on the server side in the context of Java-based Web applications (see [12] for details). In summary, the key features of ImageJ are:

- A set of ready-to-use, interactive tools for creating, visualizing, editing, processing, analyzing, loading, and storing images, with support for several common file formats. ImageJ also provides "deep" 16-bit integer images, 32-bit floating-point images, and image sequences ("stacks").

[5] http://rsb.info.nih.gov/ij/.

[6] http://fiji.sc.

[7] http://imagej.net/ImageJ2. To avoid confusion, the "classic" ImageJ platform is sometimes referred to as "ImageJ1" or simply "IJ1".

- A simple plugin mechanism for extending the core functionality of ImageJ by writing (usually small) pieces of Java code. All coding examples shown in this book are based on such plugins.
- A macro language and the corresponding interpreter, which make it easy to implement larger processing blocks by combining existing functions without any knowledge of Java. Macros are not discussed in this book, but details can be found in ImageJ's online documentation.[8]

2.2.2 Interactive Tools

When ImageJ starts up, it first opens its main window (Fig. 2.1), which includes the following menu entries:

- File: for opening, saving, and creating new images.
- Edit: for editing and drawing in images.
- Image: for modifying and converting images, geometric operations.
- Process: for image processing, including point operations, filters, and arithmetic operations between multiple images.
- Analyze: for statistical measurements on image data, histograms, and special display formats.
- Plugin: for editing, compiling, executing, and managing user-defined plugins.

The current version of ImageJ can open images in several common formats, including TIFF (uncompressed only), JPEG, GIF, PNG, and BMP, as well as the formats DICOM[9] and FITS,[10] which are popular in medical and astronomical image processing, respectively. As is common in most image-editing programs, all interactive operations are applied to the currently *active* image, i.e., the image most recently selected by the user. ImageJ provides a simple (single-step) "undo" mechanism for most operations, which can also revert modifications effected by user-defined plugins.

2.2.3 ImageJ Plugins

Plugins are small Java modules for extending the functionality of ImageJ by using a simple standardized interface (Fig. 2.2). Plugins can be created, edited, compiled, invoked, and organized through the Plugin menu in ImageJ's main window (Fig. 2.1). Plugins can be grouped to improve modularity, and plugin commands can be arbitrarily placed inside the main menu structure. Also, many of ImageJ's built-in functions are actually implemented as plugins themselves.

Program structure

Technically speaking, plugins are Java classes that implement a particular interface specification defined by ImageJ. There are two main types of plugins:

[8] http://rsb.info.nih.gov/ij/developer/macro/macros.html.
[9] Digital Imaging and Communications in Medicine.
[10] Flexible Image Transport System.

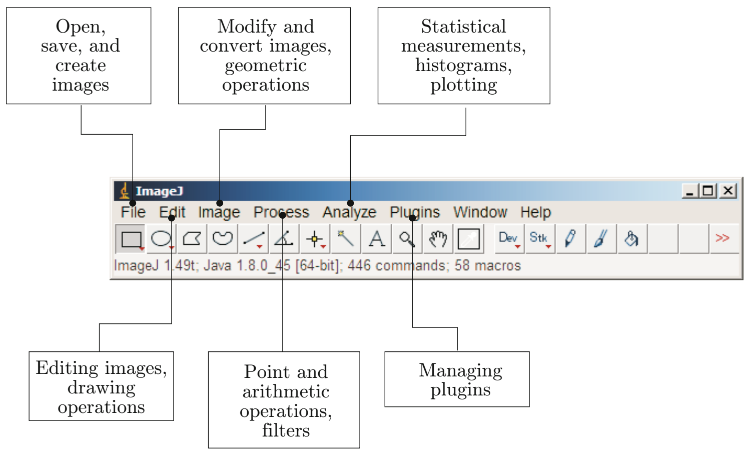
Open, save, and create images
Modify and convert images, geometric operations
Statistical measurements, histograms, plotting
ImageJ
File Edit Image Process Analyze Plugins Window Help
ImageJ 1.49t; Java 1.8.0_45 [64-bit]; 446 commands; 58 macros
Editing images, drawing operations
Point and arithmetic operations, filters
Managing plugins

Fig. 2.1
ImageJ main window (under Windows).

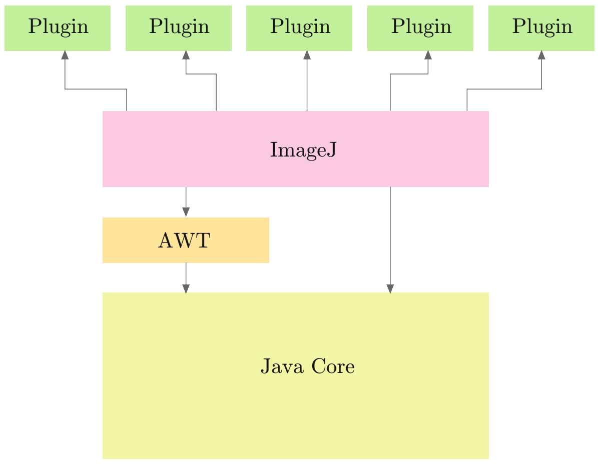

Fig. 2.2
ImageJ software structure (simplified). ImageJ is based on the Java core system and depends in particular upon Java's Advanced Windowing Toolkit (AWT) for the implementation of the user interface and the presentation of image data. Plugins are small Java classes that extend the functionality of the basic ImageJ system.

- `PlugIn`: requires no image to be open to start a plugin.
- `PlugInFilter`: the currently active image is passed to the plugin when started.

Throughout the examples in this book, we almost exclusively use plugins of the second type (i.e., `PlugInFilter`) for implementing image-processing operations. The interface specification requires that any plugin of type `PlugInFilter` must at least implement two methods, `setup()` and `run()`, with the following signatures:

`int setup (String` *`args`*`, ImagePlus` *`im`*`)`
When the plugin is started, ImageJ calls this method first to verify that the capabilities of this plugin match the target image. `setup()` returns a vector of binary flags (packaged as a 32-bit `int` value) that describes the plugin's properties.

`void run (ImageProcessor` *`ip`*`)`
This method does the actual work for this plugin. It is passed a single argument *ip*, an object of type `ImageProcessor`, which contains the image to be processed and all relevant information

about it. The `run()` method returns no result value (`void`) but may modify the passed image and create new images.

2.2.4 A First Example: Inverting an Image

Let us look at a real example to quickly illustrate this mechanism. The task of our first plugin is to invert any 8-bit grayscale image to turn a positive image into a negative. As we shall see later, inverting the intensity of an image is a typical *point operation*, which is discussed in detail in Chapter 4. In ImageJ, 8-bit grayscale images have pixel values ranging from 0 (black) to 255 (white), and we assume that the width and height of the image are M and N, respectively. The operation is very simple: the value of each image pixel $I(u, v)$ is replaced by its inverted value,

$$I(u,v) \quad \leftarrow \quad 255 - I(u,v),$$

for all image coordinates (u, v), with $u = 0, \ldots, M-1$ and $v = 0, \ldots, N-1$.

2.2.5 Plugin `My_Inverter_A` (using `PlugInFilter`)

We decide to name our first plugin "`My_Inverter_A`", which is both the name of the Java class and the name of the source file[11] that contains it (see Prog. 2.1). The underscore characters ("`_`") in the name cause ImageJ to recognize this class as a plugin and to insert it automatically into the menu list at startup. The Java source code in file `My_Inverter.java` contains a few `import` statements, followed by the definition of the class `My_Inverter`, which implements the `PlugInFilter` interface (because it will be applied to an existing image).

The `setup()` method

When a plugin of type `PlugInFilter` is executed, ImageJ first invokes its `setup()` method to obtain information about the plugin itself. In this example, `setup()` only returns the value `DOES_8G` (a static `int` constant specified by the `PlugInFilter` interface), indicating that this plugin can handle 8-bit grayscale images. The parameters `arg` and `im` of the `setup()` method are not used in this example (see also Exercise 2.7).

The `run()` method

As mentioned already, the `run()` method of a `PlugInFilter` plugin receives an object (`ip`) of type `ImageProcessor`, which contains the image to be processed and all relevant information about it. First, we use the `ImageProcessor` methods `getWidth()` and `getHeight()` to query the size of the image referenced by `ip`. Then we use two nested `for` loops (with loop variables `u`, `v` for the horizontal and vertical coordinates, respectively) to iterate over all image pixels. For reading and writing the pixel values, we use two additional methods of the class `ImageProcessor`:

[11] File `My_Inverter_A.java`.

```java
import ij.ImagePlus;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;

public class My_Inverter_A implements PlugInFilter {

  public int setup(String args, ImagePlus im) {
    return DOES_8G; // this plugin accepts 8-bit grayscale images
  }

  public void run(ImageProcessor ip) {
    int M = ip.getWidth();
    int N = ip.getHeight();

    // iterate over all image coordinates (u,v)
    for (int u = 0; u < M; u++) {
      for (int v = 0; v < N; v++) {
        int p = ip.getPixel(u, v);
        ip.putPixel(u, v, 255 - p);
      }
    }
  }

}
```

Prog. 2.1
ImageJ plugin for inverting 8-bit grayscale images. This plugin implements the interface `PlugInFilter` and defines the required methods `setup()` and `run()`. The target image is received by the `run()` method as an instance of type `ImageProcessor`. ImageJ assumes that the plugin modifies the supplied image and automatically redisplays it after the plugin is executed. Program 2.2 shows an alternative implementation that is based on the `PlugIn` interface.

`int getPixel (int` u`, int` v`)`
    Returns the pixel value at the given position or zero if (u, v) is outside the image bounds.

`void putPixel (int` u`, int` v`, int` a`)`
    Sets the pixel value at position (u, v) to the new value a. Does nothing if (u, v) is outside the image bounds.

Both methods check the supplied image coordinates and pixel values to avoid unwanted errors. While this makes them more or less fail-safe it also makes them slow. If we are sure that no coordinates outside the image bounds are ever accessed (as in `My_Inverter` in Prog. 2.1) and the inserted pixel values are guaranteed not to exceed the image processor's range, we can use the significantly faster methods `get()` and `set()` in place of `getPixel()` and `putPixel()`, respectively. The most efficient way to process the image is to avoid read/write methods altogether and directly access the elements of the associated (1D) pixel array. Details on these and other methods can be found in the ImageJ API documentation.[12]

2.2.6 Plugin `My_Inverter_B` (using `PlugIn`)

Program 2.2 shows an alternative implementation of the inverter plugin based on ImageJ's `PlugIn` interface, which requires a `run()` method only. In this case the reference to the current image is not supplied directly but is obtained by invoking the (static) method

[12] http://rsbweb.nih.gov/ij/developer/api/index.html.

Prog. 2.2
Alternative implementation of the inverter plugin, based on ImageJ's `PlugIn` interface. In contrast to Prog. 2.1 this plugin has no `setUp()` method but defines a `run()` method only. The current image (`im`) is obtained as an instance of class `ImagePlus` by invoking the `IJ.getImage()` method. After checking for the proper image type the associated `ImageProcessor` (`ip`) is retrieved from `im`. The parameter string (`args`) is not used in this example. The remaining parts of the plugin are identical to Prog. 2.1, except that the (slightly faster) pixel access methods `get()` and `set()` are used. Also note that the modified image is not re-displayed automatically but by an explicit call to `updateAndDraw()`.

```java
import ij.IJ;
import ij.ImagePlus;
import ij.plugin.PlugIn;
import ij.process.ImageProcessor;

public class My_Inverter_B implements PlugIn {

  public void run(String args) {
    ImagePlus im = IJ.getImage();

    if (im.getType() != ImagePlus.GRAY8) {
      IJ.error("8-bit grayscale image required");
      return;
    }

    ImageProcessor ip = im.getProcessor();
    int M = ip.getWidth();
    int N = ip.getHeight();

    // iterate over all image coordinates (u,v)
    for (int u = 0; u < M; u++) {
      for (int v = 0; v < N; v++) {
        int p = ip.get(u, v);
        ip.set(u, v, 255 - p);
      }
    }

    im.updateAndDraw();    // redraw the modified image
  }
}
```

`IJ.getImage()`. If no image is currently open, `getImage()` automatically displays an error message and aborts the plugin. However, the subsequent test for the correct image type (`GRAY8`) and the corresponding error handling must be performed explicitly. The `run()` method accepts a single string argument that can be used to pass arbitrary information for controlling the plugin.

2.2.7 When to use `PlugIn` or `PlugInFilter`?

The choice of `PlugIn` or `PlugInFilter` is mostly a matter of taste, since both versions have their advantages and disadvantages. As a rule of thumb, we use the `PlugIn` type for tasks that do not require any image to be open but for tasks that create, load, or record images or perform operations without any images. Otherwise, if one or more open images should be processed, `PlugInFilter` is the preferred choice and thus almost all plugins in this book are of type `PlugInFilter`.

Editing, compiling, and executing the plugin

The Java source file for our plugin should be stored in directory `<ij>/plugins/`[13] or an immediate subdirectory. New plugin files

[13] `<ij>` denotes ImageJ's installation directory.

can be created with ImageJ's Plugins ▷ New... menu. ImageJ even provides a built-in Java editor for writing plugins, which is available through the Plugins ▷ Edit... menu but unfortunately is of little use for serious programming. A better alternative is to use a modern editor or a professional Java programming environment, such as Eclipse,[14] NetBeans,[15] or JBuilder,[16] all of which are freely available.

For compiling plugins (to Java bytecode), ImageJ comes with its own Java compiler as part of its runtime environment. To compile and execute the new plugin, simply use the menu

Plugins ▷ Compile and Run...

Compilation errors are displayed in a separate log window. Once the plugin is compiled, the corresponding `.class` file is automatically loaded and the plugin is applied to the currently active image. An error message is displayed if no images are open or if the current image cannot be handled by that plugin.

At startup, ImageJ automatically loads all correctly named plugins found in the `<ij>/plugins/` directory (or any immediate subdirectory) and installs them in its Plugins menu. These plugins can be executed immediately without any recompilation. References to plugins can also be placed manually with the

Plugins ▷ Shortcuts ▷ Install Plugin...

command at any other position in the ImageJ menu tree. Sequences of plugin calls and other ImageJ commands may be recorded as macro programs with Plugins ▷ Macros ▷ Record.

Displaying and "undoing" results

Our first plugins in Prog. 2.1–2.2 did not create a new image but "destructively" modified the target image. This is not always the case, but plugins can also create additional images or compute only statistics, without modifying the original image at all. It may be surprising, though, that our plugin contains no commands for displaying the modified image. This is done automatically by ImageJ whenever it can be assumed that the image passed to a plugin was modified.[17] In addition, ImageJ automatically makes a copy ("snapshot") of the image before passing it to the `run()` method of a `PlugInFilter`-type plugin. This feature makes it possible to restore the original image (with the Edit ▷ Undo menu) after the plugin has finished without any explicit precautions in the plugin code.

Logging and debugging

The usual console output from Java via `System.out` is not available in ImageJ by default. Instead, a separate logging window can be used which facilitates simple text output by the method

```
IJ.log(String s).
```

[14] www.eclipse.org.
[15] www.netbeans.org.
[16] www.borland.com.
[17] No automatic redisplay occurs if the `NO_CHANGES` flag is set in the return value of the plugin's `setup()` method.

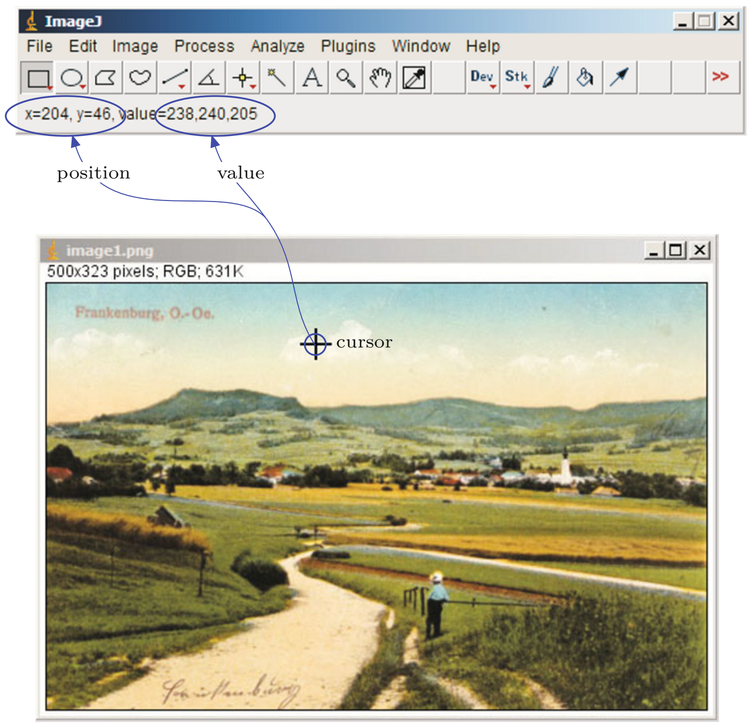
ImageJ
File Edit Image Process Analyze Plugins Window Help
x=204, y=46, value=238,240,205
position
value
image1.png
500x323 pixels; RGB; 631K
cursor
Frankenburg, O.-Oe.

Fig. 2.3
Information displayed in ImageJ's main window is extremely helpful for debugging image-processing operations. The current cursor position is displayed in pixel coordinates unless the associated image is spatially calibrated. The way pixel *values* are displayed depends on the image type; in the case of a color image (as shown here) integer RGB component values are shown.

Such calls may be placed at any position in the plugin code for quick and simple debugging at runtime. However, because of the typically large amounts of data involved, they should be used with caution in real image-processing operations. Particularly, when placed in the body of inner processing loops that could execute millions of times, text output may produce an enormous overhead compared to the time used for the actual calculations.

ImageJ itself does not offer much support for "real" debugging, i.e., for setting breakpoints, inspecting local variables etc. However, it is possible to launch ImageJ from within a programming environment (IDE) such as Eclipse or *Netbeans* and then use all debugging options that the given environment provides.[18] According to experience, this is only needed in rare and exceptionally difficult situations. In most cases, inspection of pixel values displayed in ImageJ's main window (see Fig. 2.3) is much simpler and more effective. In general, many errors (in particular those related to image coordinates) can be easily avoided by careful planning in advance.

2.2.8 Executing ImageJ "Commands"

If possible, it is wise in most cases to re-use existing (and extensively tested) functionality instead of re-implementing it oneself. In particuar, the Java library that comes with ImageJ covers many standard image-processing operations, many of which are used throughout this

[18] For details see the "HowTo" section at http://imagejdocu.tudor.lu.

```java
import ij.IJ;
import ij.ImagePlus;
import ij.plugin.PlugIn;

public class Run_Command_From_PlugIn implements PlugIn {

    public void run(String args) {
      ImagePlus im = IJ.getImage();
      IJ.run(im, "Invert", ""); // run the "Invert" command on im
      // ... continue with this plugin
    }
}
```

Prog. 2.3
Executing the ImageJ command "Invert" within a Java plugin of type `PlugIn`.

```java
public class Run_Command_From_PlugInFilter implements
    PlugInFilter {
  ImagePlus im;

  public int setup(String args, ImagePlus im) {
    this.im = im;
    return DOES_ALL;
  }

  public void run(ImageProcessor ip) {
    im.unlock();                // unlock im to run other commands
    IJ.run(im, "Invert", "");     // run "Invert" command on im
    im.lock();                  // lock im again (to be safe)
    // ... continue with this plugin
  }
}
```

Prog. 2.4
Executing the ImageJ command "Invert" within a Java plugin of type `PlugInFilter`. In this case the current image is automatically locked during plugin execution, such that no other operation may be applied to it. However, the image can be temporarily unlocked by calling `unlock()` and `lock()`, respectively, to run the external command.

book. Additional classes and methods for specific operations are contained in the associated (`imagingbook`) library.

In the context of ImageJ, the term "command" refers to any composite operation implemented as a (Java) plugin, a macro command or as a script.[19] ImageJ itself includes numerous commands which can be listed with the menu Plugins ▷ Utilities ▷ Find Commands.... They are usually referenced "by name", i.e., by a unique string. For example, the standard operation for inverting an image (Edit ▷ Invert) is implemented by the Java class `ij.plugin.filter.Filters` (with the argument `"invert"`).

An existing command can also be executed from within a Java plugin with the method `IJ.run()`, as demonstrated for the "Invert" command in Prog. 2.3. Some caution is required with plugins of type `PlugInFilter`, since these lock the current image during execution, such that no other operation can be applied to it. The example in Prog. 2.4 shows how this can be resolved by a pair of calls to `unlock()` and `lock()`, respectively, to temporarily release the current image.

A convenient tool for putting together complex commands is ImageJ's built-in *Macro Recorder*. Started with Plugins ▷ Macros ▷

[19] Scripting languages for ImageJ currently include *JavaScript*, *BeanShell*, and *Python*.

Record..., it logs all subsequent commands in a text file for later use. It can be set up to record commands in various modes, including *Java*, *JavaScript*, *BeanShell*, or ImageJ macro code. Of course it does record the application of self-defined plugins as well.

2.3 Additional Information on ImageJ and Java

In the following chapters, we mostly use concrete plugins and Java code to describe algorithms and data structures. This not only makes these examples immediately applicable, but they should also help in acquiring additional skills for using ImageJ in a step-by-step fashion. To keep the text compact, we often describe only the `run()` method of a particular plugin and additional class and method definitions if they are relevant in the given context. The complete source code for these examples can of course be downloaded from the book's supporting website.[20]

2.3.1 Resources for ImageJ

The complete and most current API reference, including source code, tutorials, and many example plugins, can be found on the official ImageJ website. Another great source for any serious plugin programming is the tutorial by Werner Bailer [12].

2.3.2 Programming with Java

While this book does not require extensive Java skills from its readers, some elementary knowledge is essential for understanding or extending the given examples. There is a huge and still-growing number of introductory textbooks on Java, such as [8, 29, 66, 70, 208] and many others. For readers with programming experience who have not worked with Java before, we particularly recommend some of the tutorials on Oracle's Java website.[21] Also, in Appendix F of this book, readers will find a small compilation of specific Java topics that cause frequent problems or programming errors.

2.4 Exercises

Exercise 2.1. Install the current version of ImageJ on your computer and make yourself familiar with the built-in commands (open, convert, edit, and save images).

Exercise 2.2. Write a new ImageJ plugin that reflects a grayscale image horizontally (or vertically) using `My_Inverter.java` (Prog. 2.1) as a template. Test your new plugin with appropriate images of different sizes (odd, even, extremely small) and inspect the results carefully.

[20] www.imagingbook.com.

[21] http://docs.oracle.com/javase/.

Exercise 2.3. The `run()` method of plugin `Inverter_Plugin_A` (see Prog. 2.1) iterates over all pixels of the given image. Find out in which order the pixels are visited: along the (horizontal) lines or along the (vertical) columns? Make a drawing to illustrate this process.

Exercise 2.4. Create an ImageJ plugin for 8-bit grayscale images of arbitrary size that paints a white frame (with pixel value 255) 10 pixels wide *into* the image (without increasing its size). Make sure this plugin also works for very small images.

Exercise 2.5. Create a plugin for 8-bit grayscale images that calculates and prints the result (with `IJ.log()`). Use a variable of type `int` or `long` for accumulating the pixel values. What is the maximum image size for which we can be certain that the result of summing with an `int` variable is correct?

Exercise 2.6. Create a plugin for 8-bit grayscale images that calculates and prints the minimum and maximum pixel values in the current image (with `IJ.log()`). Compare your output to the results obtained with Analyze ▷ Measure.

Exercise 2.7. Write a new ImageJ plugin that shifts an 8-bit grayscale image horizontally and circularly until the original state is reached again. To display the modified image after each shift, a reference to the corresponding `ImagePlus` object is required (`ImageProcessor` has no display methods). The `ImagePlus` object is only accessible to the plugin's `setup()` method, which is automatically called before the `run()` method. Modify the definition in Prog. 2.1 to keep a reference and to redraw the `ImagePlus` object as follows:

```java
public class XY_Plugin implements PlugInFilter {
  ImagePlus im;      // new variable!

  public int setup(String args, ImagePlus im) {
    this.im = im;    // reference to the associated ImagePlus object
    return DOES_8G;
  }

  public void run(ImageProcessor ip) {
    // ... modify ip
    im.updateAndDraw(); // redraw the associated ImagePlus object
    // ...
  }
}
```