UNIVERSITY OF JOENSUU

DEPARTMENT OF COMPUTER SCIENCE

Report Series A

Java Tools for Morphological Image Analysis and Research

Alexey Podlasov and Eugene Ageenko

Report A–2004–2

ACM I.4.0, I.4.3

ISBN 952-458-591-X

ISSN 0789-7316

Java Tools for Morphological Image Analysis and Research

Alexey Podlasov and Eugene Ageenko

Department of Computer Science, University of Joensuu, Box 111, FIN-80101 Joensuu, Finland

Abstract

As a standard, research work in the field of image processing and compression includes lots of experimentation. New ideas shall be first implemented in some programming languages or environment, and then undergo extensive empirical evaluation. Analysis of the weak sides will lead to a further research and development work. The implementation process (programming of the developed method) is usually the most time-consuming part of any research process. Therefore the flexibility of the implementation, such as code readability, fast modification feasibility, friendly programming environment, etc. do often prevail over the implementation efficiency, memory load and computation time. The efficient implementation could be often too expensive to conclude, especially in the research process, when frequent algorithm modifications are required. Still, the implementation efficiency should not be overlooked in the algorithm design process. The experiments should possible be conducted in a realistic time-frame and using reasonable hardware resources. The current work examines the usability of NIH ImageJ Java-based framework in the image processing research in the case of implementing morphological operations. We demonstrate how the environment can be easily understood, and the new operations implemented and modified by researchers familiar with the basics of Java language. The set of morphological operators is implemented as a test case and shows good usability in terms of ease programming and computation efficiency.

1 Introduction

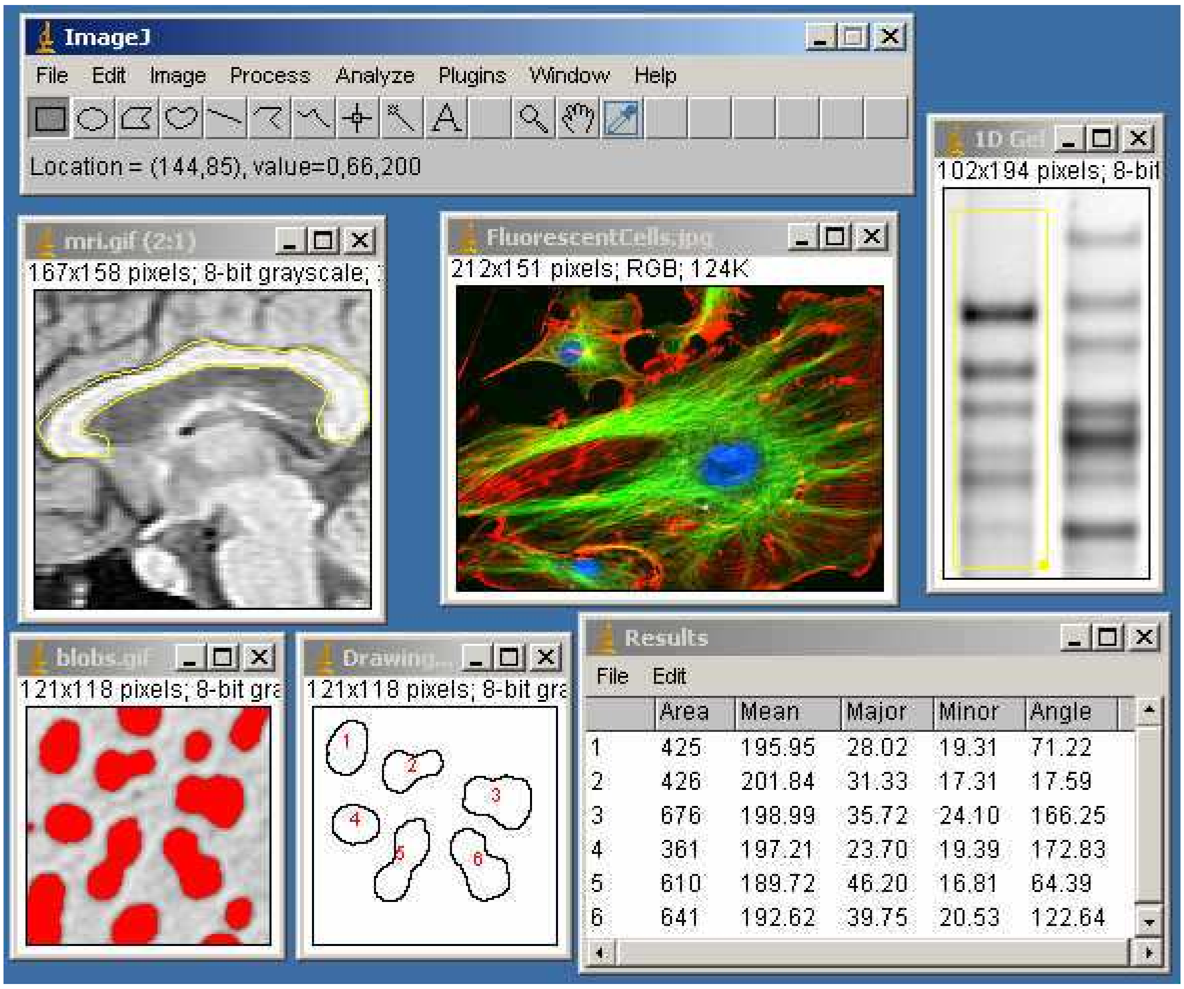
The *ImageJ* is a public domain Java image processing program inspired by the "NIH Image" for the Macintosh [1]. The ImageJ developed at the National Institutes of Health (USA) by Wayne Rasband (Research Services Branch) in the course of his official duties. Pursuant to Title 17, Section 105 of the United States Code, this software is not subject to copyright protection and is in the public domain. It runs, either as an online applet or as a downloadable application, on any computer with a Java 1.1 or later virtual machine. The source code is freely available [2].

ImageJ provides simple and usable concept of "plugins" – user-made program modules integrated into the main framework. Framework provides methods implementing most of routine operations e.g. file manipulations, image conversion, arithmetic operations, common image processing operators, etc. when researcher concentrates on the task under research substituting technical operations with ImageJ interfaces or previously developed plugins. This concept makes ImageJ used in defferent areas of Image Processing. Nowadays more than 100 plugins performing in such areas as image acquisition, color processing, various image filtering, quantization and clustering, and a lot of others are implemented and freely available on the Web [3]. A big variety of works where ImageJ is used could be found on the web. Scientific works and even entire books [4] that use ImageJ as an experimentation tool start to appear in the literature in recent time.

In this work we demonstrate how the environment can be easily understood, and the new operations implemented and modified by researchers familiar with the basics of Java language [5, 6]. As a test case we implement a set of morphological operators for binary image, which we ourselves are using in our research and development work [7]. Every operator supports 10 different structuring elements. Also we show how new operators could be created using ImageJ's built-in macro language (e.g. create Alternative-Sequential filters using Open and Close operators). Our goal is to examine the framework itself and its usability for image processing research. Under "usability in research" we mean that the framework has to:

- Provide flexible interface and a set of standard tools for viewing, observing and operating with images.
- Support a big variety of image formats and provide a unified format-independent access to the image content for a programmer when implementing new algorithms.
- Allow "friendly" programming, i.e. code have to be readable and easy-to-understand as much as possible. Ideally, algorithm has to be implementable closely to its mathematical notation.
- Provide simple mechanism for using of already developed algorithms as parts of more complicated techniques.

In our experience, the ImageJ package has shown a good usability. A lot of important features are included and most of popular image formats are supported. Further, ImageJ allows using of already developed plugins in the implementation of new ones. When developing our set of morphological operators, we were trying to use simple already developed plugins in the course of development of the complicated ones. For example the basis of most plugins is a Rank operator. Then Dilation and Erosion use Rank plugin, Open and Close use Dilation and Erosion plugins and so on. Another important feature is that ImageJ is open-source software, not demanding any kind of license or registering. And finally, it is platform independent and can be executed on virtually any computer environment where Java Virtual Machine is installed.


ImageJ
File Edit Image Process Analyze Plugins Window Help
Location = (144,85), value=0,66,200
1D Gel
102x194 pixels; 8-bit
mri.gif (2:1)
167x158 pixels; 8-bit grayscale;
FluorescentCells.jpg
212x151 pixels; RGB; 124K
blobs.gif
121x118 pixels; 8-bit gra
Drawing...
121x118 pixels; 8-bit gra
1
2
3
4
5
6
Results
File Edit

	Area	Mean	Major	Minor	Angle
1	425	195.95	28.02	19.31	71.22
2	426	201.84	31.33	17.31	17.59
3	676	198.99	35.72	24.10	166.25
4	361	197.21	23.70	19.39	172.83
5	610	189.72	46.20	16.81	64.39
6	641	192.62	39.75	20.53	122.64

Figure 1: ImageJ visual appearance

2 ImageJ in brief

ImageJ is a public domain Java image processing program inspired by "NIH Image" for the Macintosh. It runs, either as an online applet or as a downloadable application, on any computer with a Java 1.1 or later virtual machine.

It can display, zoom (in and out), scroll, edit, analyze, process, save and print 8-bit, 16-bit and 32-bit images in TIFF, GIF, JPEG, BMP, DICOM, FITS or "raw" formats. It supports "stacks", a series of images that share a single window. It is multithreaded, so time-consuming operations such as image file reading can be performed in parallel with other operations.

ImageJ is the world's fastest pure Java image processing program. It can filter a 2048x2048 image in 0.1 seconds (using Process/Smooth command for 8-bit image on 3GHz Windows PC computer running Microsoft Internet Explorer 6.0 with Microsoft Java VM 1.1.4) gaining speed of about 40 million pixels per second [2].

It contains a vast of tools for image analysis inlcuding such tools as regions of interest, area, length and angle measurements, mean, standard deviation, calibration using density standards, it can generate histograms and profile plots. All analysis and processing functions work at any magnification factor.

ImageJ is designed with an open architecture that provides extensibility via Java plugins. It can be used as a standalone application or an image processing toolkit to develop applets, servelts or applications. Further, custom acquisition, analysis and processing plugins can be developed using ImageJ's built in editor and Java compiler. User-written plugins make it possible to solve almost any image processing or analysis problem.

2.1 Installation and update guide

For running ImageJ one need ImageJ class and configuration files, a Java Runtime Environment (*JRE*) and a Java compiler with the required libraries, as for example included in the Java 2 SDK Standard Edition (*J2SE*) from Sun Microsystems.

ImageJ installation

The latest distribution of ImageJ can be downloaded from http://rsb.info.nih.gov/ij/download.html. If one already has a JRE (and a Java compiler) installed on the computer and is familiar with Java, one need to:

- Download and unpack installation archive to a dedicated folder (e.g. c:\ImageJ)
- Run ImageJ using one of provided shortcuts or via command line:

```
javaw.exe -cp ij.jar;jimi.jar; ij.ImageJ
```

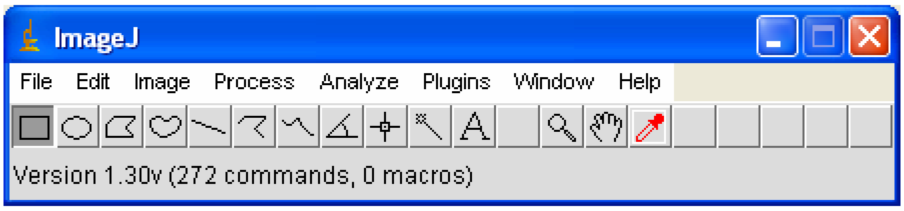
One can update ImageJ to the latest version by replacing the ImageJ JAR file (`ij.jar`). The latest version is available at http://rsb.info.nih.gov/ij/upgrade/index.html. One need just replace the existing `ij.jar` file with the new one to perform the upgrade process.

2.2 Using ImageJ

2.2.1 Basic concepts

The "ImageJ" window contains a menu bar, tool bar, status bar, and a progress bar. Images, histograms, line profile, etc. are displayed in additional windows. Measurement results are displayed in the "Results" window. Histograms and plots are ordinary image windows that can be copied (to the internal clipboard), edited, printed and saved.

Menu



Version 1.30v (272 commands, 0 macros)

Figure 2: ImageJ main window

All operations with the images are accessible from the menu. These operations include operations for loading and saving images, as well as to access various image processing operations and plugins.

Tool Bar

The tool bar contains tools for making selections, for zooming and scrolling images, and for changing the drawing color. Click on a tool and a description of that tool is displayed in the status bar.

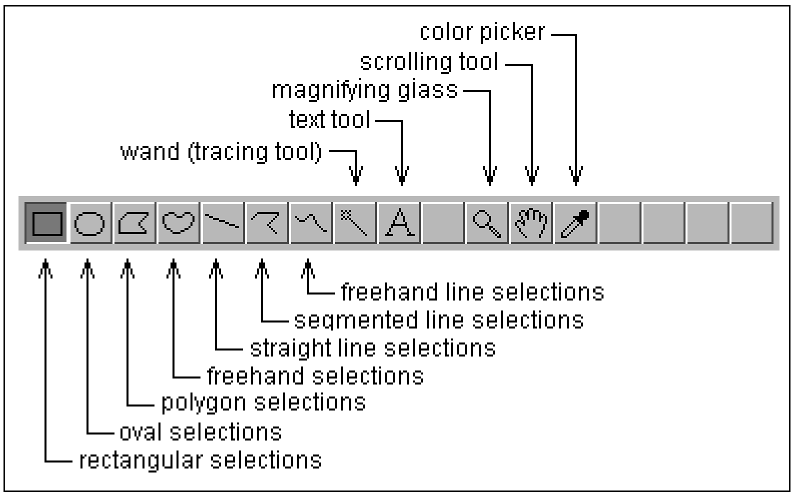
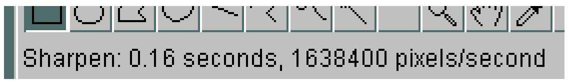


Figure 3: Tool bar

Status Bar



Sharpen: 0.16 seconds, 1638400 pixels/second

Figure 4: Status bar

The status bar, when the cursor is over an image, displays pixel coordinates and values. After running a filter, it displays the elapsed time and processing rate in pixels/second.

Progress Bar

Figure 5: Progress bar

The progress bar, located to the right of the status bar, shows the progress of time-consuming operations. It will not appear if the operation requires less then approximately one second.

Images

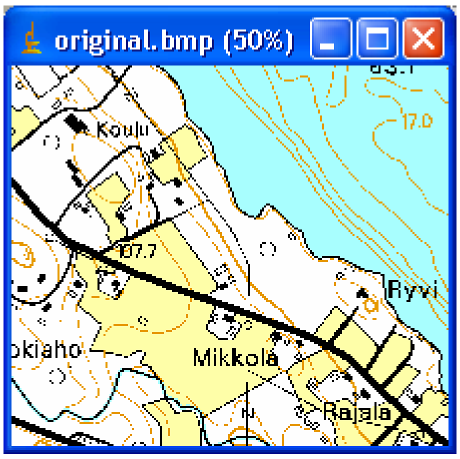


Figure 6: Image window

ImageJ allows multiple images to be displayed on the screen at one time. The active window has its title bar highlighted. All operations will be performed on the active image. ImageJ supports 8-bit, 16-bit and 32-bit (real) grayscale images and 8-bit and 32-bit color images.

Stacks

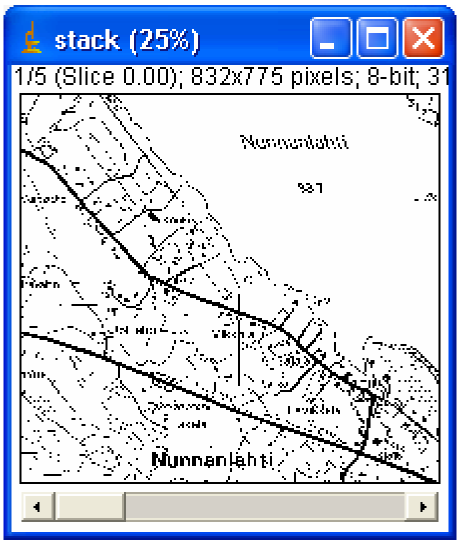


Figure 7: Stack window

ImageJ can display multiple spatially or temporally related images in a single window. These image sets are called stacks. The images that make up a stack are called slices. All the slices in a stack must be the same size and bit depth. A scroll bar provides the ability to move through the slices.

Selections

Selections are user defined areas or lines within an image. Only one selection can be active at a time. Area selections are created using the rectangular, oval, polygonal and freehand selection tools.

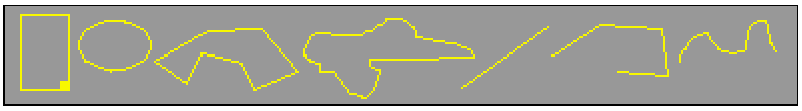


Figure 8: Types of selections

File Formats

The *File/Open* command opens TIFF (uncompressed), GIF, JPEG, DICOM, BMP and FITS images. It also opens lookup tables and selections (*ROIs*).

2.2.2 Plugins

Plugins are loadable code modules that extend the capabilities of ImageJ. Using plugins one can implement any needed algorithm concentrating on implementation of how actually image has to be processed. All background work when reading, displaying, magnifying, applying ROI, etc. will be handled by ImageJ. Plugins can be developed using any Java IDE but the easiest way to create a plugin is to open the command recorder, record a series of commands, and then click *Create Plugin*. One can use such plugin as a basis for further development for example using *Plugins/Edit* menu command.

Using plugins

Let's take a look at the sequence of operations necessary to execute Rank operator:

1. Install Mathematical Morphology plugins
2. Open an image using File/Open menu.
3. Select Rank plugin in Plugins/Morphology menu

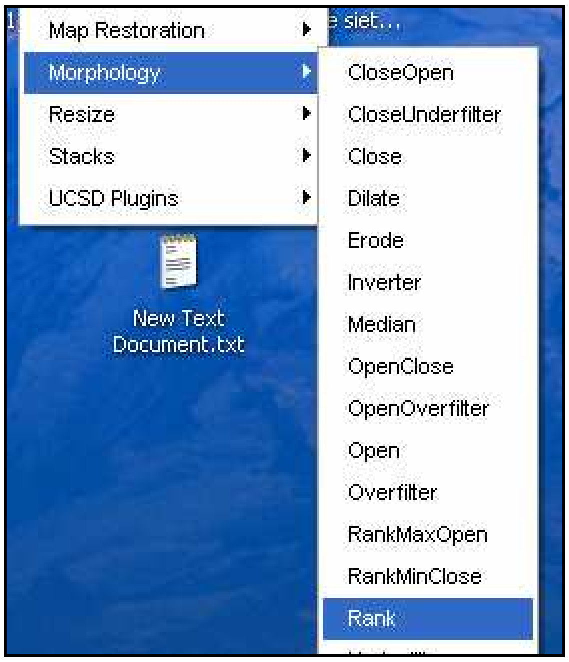


Figure 9: "Morphology" submenu of "Plugins" menu

4. If selected image is not gray-scale, then ImageJ automatically shows error dialog box and the plugin will not apply

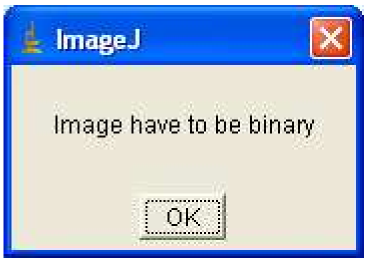
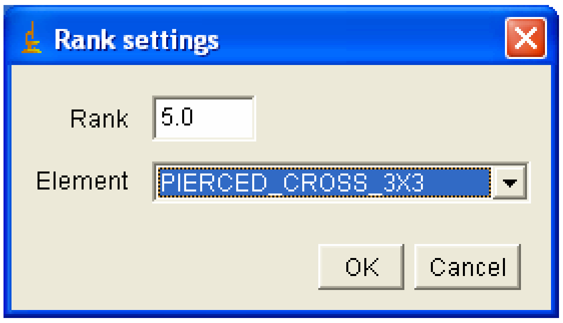

ImageJ

Image have to be binary

OK

Figure 10: Not supported image type message box

5. If source image was chosen properly, then parameters dialog box will appear. Parameters chosen in that dialog box will be handled by ImageJ and transferred into plugin.


Rank settings

Rank 5.0

Element PIERCED_CROSS_3X3

OK Cancel

Figure 11: Plugin parameters

6. If “Ok” is selected, the plugin starts execution of the algorithm, or exits in case of “Cancel”.

When the plugin is executed from within another ImageJ plugin or from a macro, one can specify parameters in a parameter string. In that case the parameters will be handled and transferred into the plugin without dialog box appearance.

3 Development with ImageJ

3.1.1 ImageJ class structure

This is an overview of the class structure in ImageJ. Only basic and most important for getting started classes are listed and briefly described. For complete description reader should refer to [3, 8].

Class structure

The ImageJ framework has the following class hierarchy:

`i) ij (package)`

- **`i.i) Executer`**

  - runs menu commands in a separate thread. It can be used for invoking already existing plugins inside the new one.

- **`i.ii) IJ`**

  - a Class containing a lot of useful utility methods.

- **`i.iii) ImagePlus`**

  - one of the most important classes in ImageJ representing an image itself. It is based on `ImageProcessor`.

- **`i.iv) ImageStack`**

  - represents the stack of images (see 2.2.1).

`ii) ij.gui (package)`

- contains classes for implementing GUI elements and different types of ROI. Classes of our interest are:

- **`ii.i) GenericDialog`**

  - this class implements customizable modal dialog box which also used for transferring parameters from macros into plugins.

- **`ii.ii) NewImage`**

  - new image dialog box plus several useful static utility methods for creating new images.

`iii) ij.plugin (package)`

- most of ImageJ commands are implemented as plugins and could be found in that package or one of its subpackages.

`iii.i)   PlugIn`

- this interface has to be implemented by plugins that require more complicated input than just an input image.

`iii.ii)  ij.plugin.filter (package)`

`iii.ii.i)    PlugInFilter`

- this interface has to be implemented by plugins that require just an input image.

`iii.iii) ij.plugin.frame (package)`

`iii.iii.i)   PlugInFrame`

- this is a closable window that plugins can extend.

`iv) ij.process (package)`

- this package contains image processors classes which are the main tool to operate with images when developing with ImageJ.

`iv.i) ImageProcessor`

- one of the most important classes for developing with ImageJ. Contains a big variety of standard operations. All custom operations with image should be implemented using this class.

`iv.ii)StackProcessor`

- this class process stacks.

3.1.2 Plugin concept in ImageJ

Functions provided by ImageJ could be extended by user plugins. In fact, the most of ImageJ standard functions are implemented as plugins. User plugin is a Java class implementing necessary interface and placed into certain folder. Plugins could be written in ImageJ Java editor or in any existing Java environment, recorded with ImageJ macro recorder or written on ImageJ internal macro language. Plugins found by ImageJ are placed into *Plugins* menu or submenus of it.

Types of plugins

There are two basic types of plugins in ImageJ. They are ones which implement *PlugIn* interface and ones which implement *PlugInFilter* interface. The difference is that *PlugIn* interface receives only a string with parameters as its input and programmer has to implement access to desired image by himself. On the other hand plugins implementing PlugInFilter interface do not care about that and receive access to currently active image directly as parameter.

`PlugIn` ***interface:***

Interface has only one method:

```
void run(java.lang.String arg)
```

This method executes (runs) the plugin. `arg` is a string argument passed to the plugin.

`PlugInFilter` *interface:*

When implementing `PlugInFilter` `setup` method is called first

```
int setup(java.lang.String arg, ImagePlus imp)
```

This method sets up the plugin for use. This method returns a flag word that represents plugins capabilities. The combination of the flags that could be returned is shown in the Table 1. Actual functionality of the plugin is implemented in the method

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

This method runs the plugin and what you implement here is what the plugin will actually do. It has currently active image as its argument. `ImageProcessor` provides a lot of already developed operations over the image. It also provides direct access to image pixels to implement any custom functionality.

Table 1: Flags returned by setup

`static int`	`DOES_16` Set this flag if the filter handles 16 bit grayscale images.
`static int`	`DOES_32` Set this flag if the filter handles 32 bit floating point grayscale images.
`static int`	`DOES_8C` Set this flag if the filter handles 8-bit color images.
`static int`	`DOES_8G` Set this flag if the filter handles 8-bit grayscale images.
`static int`	`DOES_ALL` Set this flag if the filter handles all types of images.
`static int`	`DOES_RGB` Set this flag if the filter handles RGB images.
`static int`	`DOES_STACKS` Set this flag if the filter wants its run() method to be called for all the slices in a stack.
`static int`	`DONE` Set this flag and run method will not be called.
`static int`	`NO_CHANGES` Set this flag if the filter makes no changes to the pixel data.
`static int`	`NO_IMAGE_REQUIRED` Set this flag if the filter does not require that an image to be open.
`static int`	`NO_UNDO` Set this flag if the filter does not require undo.
`static int`	`ROI_REQUIRED` Set this flag if the filter requires an ROI.
`static int`	`STACK_REQUIRED` Set this flag if the filter requires a stack.
`static int`	`SUPPORTS_MASKING` Set this flag if the filter wants ImageJ, for non-rectangular ROIs, to restore that part of the image that's inside the bounding rectangle but outside of the ROI.

Plugin installation

One can install a plugin in a following way:

1. Put plugin's .class file into a "plugins" folder which is a subfolder of ImageJ base folder. Plugins with underscore in its name (i.e. "`Inverter_.class`") will appear in Plugins menu automatically when ImageJ is restarted.
2. Other plugins have to be added to menu using "Plugins/Shortcut/Install plugin..." menu command.
3. If plugin files are located in on of the subfolder of "plugins" folder they will be located in submenus with corresponding names.

Recording plugins

If your plugin shall just execute a sequence of ImageJ menu commands, you do not have to write your plugin, you can simply record it. "Plugins/Record..." opens a window and records your actions.

Compiling and running plugins

There are basically two ways to execute the plugin:

1. Using the menu "Plugins/Compile and run..." which opens a file dialog which lets user to select a .java file which will be compiled into a class file and executed as plugin.
2. Using "File/Compile and run..." in the built-in plugin editor which will compile and run the code in the editor window.

If the plugin requires other libraries than standard ones, they have to be added into the "Target:" field of the of the "Shortcut" properties of the ImageJ shortcut as follow:

```
javaw.exe mx80m -classpath ij.jar;tools11.jar;mylib.jar ij.ImageJ
```

3.1.3 Images in ImageJ

In ImageJ images are represented by `ImagePlus` and `ImageProcessor` objects. In the following section we'll take a closer look at the way images are handled.

Types of images

ImageJ supports following types of images:

- **8 bit grayscale image**, a pixel is represented by a `byte` variable.
- **8 bit color image**, can display 256 colors that are specified in a lookup table (LUT), a pixel is represented by a `byte` variable.

- **16 bit grayscale image**, a pixel is represented by a `short` variable.
- **RGB color image**, a pixel is represented by an `int` variable.
- **32 bit image floating point grayscale image**, a pixel is represented by a `float` variable.

`ImagePlus` class

An `ImagePlus` is an object that represents an image. It is based on an `ImageProcessor`, a class that holds the pixel array and does the actual work on the image. The type of the `ImageProcessor` used depends on the type of the image. The image types are represented by constants declared in `ImagePlus`:

• `ImagePlus.COLOR_256`	8 bit color image with a look-up table.
• `ImagePlus.COLOR_RGB`	RGB color image.
• `ImagePlus.GRAY16`	16 bit grayscale image.
• `ImagePlus.GRAY32`	A 32 bit floating point grayscale image.
• `ImagePlus.GRAY8`	A 8 bit grayscale image.

ImageJ displays images using a class called `ImageWindow`. It handles repainting, zooming, changing masks etc.

There are different ways to construct an `ImagePlus` object:

- Using one of the available constructors:

  ```
  ImagePlus image = new ImagePlus();
  ```

  Default constructor, creates a new empty `ImagePlus`

  ```
  ImageProcessor ip = new ImageProcessor(width, height);
  ImagePlus image = new ImagePlus(title, ip);
  ```

  Constructs a new `ImagePlus` based on given `ImageProcessor ip`.

  ```
  ImageStack stack = new ImageStack(width, height)
  ImagePlus image = new ImagePlus(title, stack)
  ```

  Constructs a new `ImagePlus` based on given `ImageStack stack`.

- Using `ImagePlus.createImagePlus()`

  ```
  ImagePlus givenImage = ….
  ImagePlus image = givenImage.createImagePlus();
  ```

  To create a new image with this image properties (e.g. dimension and color model)

- The class `NewImage` offers some useful static methods for creating new images of a certain type. For these methods the title, width, height, number of slices and filling are specified as parameters.

  `NewImage.createByteImage` - creates a new 8 bit grayscale or color;

  `NewImage.createFloatImage` - creates a new 32 bit floating point image.

  `NewImage.createRGBImage` – Creates a new RGB image.

  `NewImage.createShortImage` - a new 16 bit grayscale

The type of an ImagePlus can be retrieved using

```
int getType()
```

Similar methods exist for getting such image properties like dimension, title, the AWT image that represents the `ImagePlus` and the file information:

```
int getHeight()
int getWidth()
java.lang.String getTitle()
java.awt.Image getImage()
ij.io.FileInfo getFileInfo()
```

`ImageProcessor` class

Each image is based on the image processor. The type of the processor depends on the type of the image. One can get and set the image processor using these two methods of the `ImagePlus` class:

```
ImageProcessor getProcessor()
```

- returns a reference to the image's `ImageProcessor`.

```
void setProcessor(java.lang.String title, ImageProcessor ip)
```

- sets the image processor to the one specified.

When working with plugin filters one does not have to care about retrieving the processor from the `ImagePlus`, because it is passed as argument to the `run` method.

`ImageProcessor` is an abstract class. Depending on the type of the image we use one of its subclasses. There are:

- `ByteProcessor` – used for 8 bit grayscale and color images.
- `ShortProcessor` – used for 16 bit grayscale images.
- `ColorProcessor` – used for 32 bit RGB images.
- `FloatProcessor` – used for 32 bit floating point images.

Accessing pixel values

Retrieving the pixel values can be done by using an `ImageProcessor`'s

```
java.lang.Object getPixels()
```

method. It returns a reference to this image's pixel array. As the type of this array depends on the image type we need to cast this array to the appropriate type when we get it:

```
int[] pixels = (int[]) myProcessor.getPixels()
```

This example would work for an RGB image. Notice that we receive a one-dimensional array. It contains the image pixel values scanline by scanline. To convert a position in this array to a (x,y) coordinate in an image, we need to know image's dimensions. The width and height of an `ImageProcessor` can be retrieved using:

```
int getHeight()
int getWidth()
```

Reading pixels from `ByteProcessor`, `ShortProcessor` and from `ColorProcessor` need more explanation.

Java's `byte` data type is *signed* and has values ranging from 128 to 127, while we would expect a 8 bit grayscale image to have values from 0 to 255. If we cast a `byte` variable to another type we have to make sure, that the sign bit is eliminated. This can be done using a binary AND operation (&):

```
int pix = 0xff & pixels[i];
...
pixels[i] = (byte) pix;
```

It's the same with Java's `short` data type, which is also signed and has values ranging from -32768 to 32767, while we would expect a 16 bit grayscale image to have values from 0 to 65535. Again, we can be sure that sign bit is eliminated by using binary AND operation:

```
int pix = pixels[i] & 0xffff;
...
pixels[i] = (short) pix;
```

`ColorProcessors` return the pixel array as an `int[]`. The values of the three color components are stored in one `int` variable. They can be accessed as follows:

```
int red = (int)(pixels[i] & 0xff0000)>>16;
int green = (int)(pixels[i] & 0x00ff00)>>8;
int blue = (int)(pixels[i] & 0x0000ff);
...
pixels[i]=((red & 0xff)<<16)+((green & 0xff)<<8) + (blue & 0xff);
```

The pixel array we work on is actually a reference to the `ImageProcessor`'s pixel array. Therefore any modifications affect the `ImageProcessor` immediately. However, one can set any appropriate array to be `ImageProcessor`'s pixel array using

```
void setPixels(java.lang.Object pixels)
```

One does not always have to retrieve or set the whole pixel array. `ImageProcessor` offers alternative methods for accessing pixel values:

```
int getPixel(int x, int y)
```

Returns the value of the specified pixel.

```
void putPixel(int x, int y, int value)
```

Sets the pixel at (x, y) to the specified value.

```
float getPixelValue(int x, int y)
```

Returns the value of the specified pixel.

```
void getColumn(int x, int y, int[] data, int length)
```

Returns the pixels down the column starting at (x, y) in data.

```
void putColumn(int x, int y, int[] data, int length)
```

Inserts the pixels contained in data into a column starting at (x, y).

```
void getRow(int x, int y, int[] data, int length)
```

Returns the pixels along the horizontal line starting at (x,y) in data.

```
void putRow(int x, int y, int[] data, int length)
```

Inserts the pixels contained in data into a horizontal line starting at (x,y).

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

Returns the pixels along the line with start point (x1,y1) and end point (x2,y2).

The method

```
int[] getPixel(int x, int y)
```

of ImagePlus returns the pixel value at (x,y) as a four element array.

NOTE: developer should avoid methods described above when modifying large parts of the image because of its slow speed. It is much faster to work with the pixel array.

Regions of interest (ROI)

A plugin filter does not always have to work on the whole image. ImageJ supports regions of interest (or selections). The bounding rectangle of the current ROI can be retrieved from the `ImageProcessor` using

```
java.awt.Rectangle getRoi()
```

This makes possible just to handle the pixels that are inside this rectangle. It is also possible to set a processor's ROI with

```
void setRoi(int x, int y, int rwidth, int rheight)
```

More methods for working with ROIs can be found in `ImagePlus`:

```
void ImagePlus.setRoi(int x, int y, int width, int height)
```

Creates a rectangular selection starting at (x,y) with specified width and height.

```
void ImagePlus.setRoi(java.awt.Rectangle r)
```

Creates a rectangular selection.

```
void ImagePlus.setRoi(Roi roi)
```

Creates a selection based on the specified ROI object.

```
Roi ImagePlus.getRoi()
```

Returns a ROI object representing the current selection.

The classes representing the different types of ROIs can be found in `ij.gui`. These classes are:

```
FreehandROI
OvalROI
PolygonROI
ROI
TextROI
```

Displaying images

ImageJ uses a class called `ImageWindow` to display `ImagePlus` images. `ImagePlus` contains everything that is necessary for updating or showing newly created images. Typically one need just to show a newly created image using `void show()` or update the modified one using `void updateAndDraw()`.

3.1.4 Sample plugin

With the knowledge of previous sections we can write our own plugin. Let's modify standard `Inverter_` plugin to demonstrate already known features: *to get the image, accessing pixel values, using ROI and create and showing the resulting image.*

```
import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import java.awt.*;
```

Our plugin must have an underscore in its name. It needs an image as its input, so it has to implement `PlugInFilter`.

```
public class Inverter_ implements PlugInFilter {
```

First we need to set up a plugin using `setup` method. The return value tells the ImageJ that our plugin will handle 8 bit grayscale images and stacks with masking. If one would try to apply that plugin to another image, ImageJ will handle it and responds about error automatically. If string argument is "about" then plugin shows about box and do nothing by returning `DONE`.

```
    public int setup(String arg, ImagePlus imp) {
        if (arg.equals("about"))
            {showAbout(); return DONE;}
        return DOES_8G+DOES_STACKS+SUPPORTS_MASKING;
    }
```

`The run` method implements the actual function of the plugin. We get the processor as input parameter and retrieve image's dimensions, pixel values and ROI.

```
    public void run(ImageProcessor ip) {
        byte[] pixels = (byte[])ip.getPixels();
        int width = ip.getWidth();
        int height = ip.getHeight();
        Rectangle r = ip.getRoi();
```

Then we create destination image processor to represent resulting image. By copying bit we construct that part of the image that will not be modified, then we get pixel array to modify area which is in ROI.

```
        ByteProcessor dip = new ByteProcessor(width,height);
        dip.copyBits(ip,0,0,Blitter.COPY);
        byte[] dstpixels = (byte[])dip.getPixels();
```

To process pixels inside ROI we implement 2 nested loops to walk through rectangle row by row and assign modified values to destination image processor.

```
        int offset, i;
        for (int y=r.y; y<(r.y+r.height); y++) {
            offset = y*width;
            for (int x=r.x; x<(r.x+r.width); x++) {
                i = offset + x;
                dstpixels[i] = (byte)(255-pixels[i]);
            }
        }
```

Finally, we create and show a window containing resulting image. Notice that ImagePlus object is created on the basis of destination image processor.

```
        new ImagePlus("Inverted",dip).show();
    }

    void showAbout() {
        IJ.showMessage("About Inverter_...",
            "Long live my genius plugin!!!"
        );
    }
}
```

3.1.5 Stacks

ImageJ supports expandable arrays of images called image stacks, which consist of images (slices) of the same size. In a plugin filter one can access the currently open stack by retrieving it from the current ImagePlus using

```
ImageStack getStack()
```

The number of slices of a stack can be retrieved using

```
int getSize()
```

Pixels of each slice in a stack could be accessed using

```
java.lang.Object getPixels(int n)
void setPixels(java.lang.Object pixels, int n)
```

or via `ImageProcessor` class

```
ImageProcessor getProcessor(int n)
```

To determine and change currently displayed slice use

```
int getCurrentSlice()
void setSlice(int index)
```

One can create new stack to use it as image's stack. Create stack using ImagePlus method

```
ImageStack createEmptyStack()
```

or one of ImageStack constructors

```
ImageStack(int width, int height)
ImageStack(int width, int height, java.awt.image.ColorModel cm)
```

To set the newly created stack as the stack of an image use

```
void ImagePlus.setStack(java.lang.String title, ImageStack stack)
```

There are also useful methods when working with stacks:

```
void addSlice(java.lang.String sliceLabel, ImageProcessor ip)
void addSlice(java.lang.String sliceLabel, java.lang.Object pixels)
void deleteLastSlice()
void deleteSlice(int n)
int getHeight()
int getWidth()
```

3.1.6 Utility methods

The ImageJ API contains a class called IJ that contains some very useful static methods [9].

Simple messages

It is often necessary that a plugin displays a message - be it an error message or any other information. In the first case one will use

```
static void error(java.lang.String msg)
```

in the second case

```
static void showMessage(java.lang.String msg)
```

One can also specify the title of the message box using

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

More flexible method is

```
static boolean showMessageWithCancel(java.lang.String title,
java.lang.String msg)
```

This method returns false if the user clicks cancel and true otherwise.

There are also some predefined messages:

```
static void noImage()
static void outOfMemory(java.lang.String name)
static boolean versionLessThan(java.lang.String version)
```

Using main window, status and progress bars.

To display a line of text in the results window use

```
static void write(java.lang.String s)
```

One will often want to displays numbers, which one can format for output using

```
static java.lang.String d2s(double n)
static java.lang.String d2s(double n, int precision)
```

Converts a number to a formatted string using specified precision.

Text can also be displayed in the status bar at the bottom of the main window using the method

```
static void showStatus(java.lang.String s)
```

The progress of the current operation can be visualized using ImageJ's progress bar.

```
static void showProgress(double progress)
```

updates the position of the progress bar to the specified value (in the range from 0.0 to 1.0).

Calling menu commands

One can access all menu commands from a plugin. There are two different methods:

```
static void doCommand(java.lang.String command)
```

Starts executing a menu command in a separate thread and returns immediately.

```
static void run(java.lang.String command)
```

Runs a menu command in the current thread, the program is will continue after the command has finished.

Calling other plugins

Like menu commands one can also run other plugins.

```
static java.lang.Object runPlugIn(java.lang.String
className,java.lang.String arg)
```

Runs the plugin specified by its class name and initializes it with the specified argument.

3.1.7 Building GUI

Sometimes one need to use a more complicated user input than just an input image. In that case one has to build GUI for the plugin to interact with the user. Typically one will need to create a modal dialog box to ask for additional parameters. Use `PlugInFrame` and `GenericDialog` classes for building GUI. These classes provide simple way for retrieving parameters from the user. Moreover, using those classes it is possible to set parameters in a parameter string when running plugin from a command line or macro. In that case parameters will be transferred automatically to the plugin by ImageJ and modal dialog will not appear.

PlugInFrame

A `PlugInFrame` is a subclass of an AWT frame that implements the `PlugIn` interface. The plugin will be implemented as a subclass of `PlugInFrame`. There is one constructor for a `PlugInFrame`. It receives the title of the window as argument:

```
PlugInFrame(java.lang.String title)
```

As this class is a plugin, the method

```
void run(java.lang.String arg)
```

declared in the `PlugIn` interface is implemented and should be overwritten by the plugin's `run` method.

Of course all methods declared in `java.awt.Frame` and its superclasses can be overwritten. For details consult the Java AWT API documentation.

GenericDialog

The `GenericDialog` class is used to build a modal AWT dialog. It can be built on the fly and one doesn't have to care about event handling. There are two constructors:

```
GenericDialog(java.lang.String title)
```

Creates a new GenericDialog with the specified title.

```
GenericDialog(java.lang.String title, java.awt.Frame parent)
```

Creates a new `GenericDialog` using the specified title and parent frame (e. g. your plugin class, which is derived from `PlugInFrame`). The ImageJ frame can be retrieved using `IJ.getInstance()`.

The dialog can be displayed using

```
void showDialog()
```

Adding controls

GenericDialog offers several methods for adding standard controls to the dialog:

```
void addCheckbox(java.lang.String label, boolean defaultValue)
```

Adds a checkbox with the specified label and default value.

```
void addChoice(java.lang.String label, java.lang.String[] items,
java.lang.String defaultItem)
```

Adds a drop down list (popup menu) with the specified label, items and default value.

```
void addMessage(java.lang.String text)
```

Adds a message consisting of one or more lines of text.

```
void addNumericField(java.lang.String label,
double defaultValue, int digits)
```

Adds a numeric field with the specified label, default value and number of digits.

```
void addStringField(java.lang.String label,
java.lang.String defaultText)
```

Adds a 8 column text field with the specified label and default value.

```
void addTextAreas(java.lang.String text1,
java.lang.String text2, int rows, int columns)
```

Adds one or two text areas (side by side) with the specified initial contents and number of rows and columns. If text2 is null, the second text area will not be displayed.

Getting values from controls

After the user has closed the dialog window, one can access the values of the controls with the methods listed here. There is one method for each type of control. If the dialog contains more than one control of the same type, each call of the method will return the value of the next control of this type in the order in which they were added to the dialog. If plugin was started from the command line, and one or all parameters were specified in a parameter string, then modal dialog will not appear and values of the parameters will be transferred automatically to retrieving methods (methods below).

```
boolean getNextBoolean()
```

Returns the state of the next checkbox.

```
java.lang.String getNextChoice()
```

Returns the selected item in the next drop down list (popup menu).

```
int getNextChoiceIndex()
```

Returns the index of the selected item in the next drop down list (popup menu).

```
double getNextNumber()
```

Returns the contents of the next numeric field.

```
java.lang.String getNextString()
```

Returns the contents of the next text field.

```
java.lang.String getNextText()
```

Returns the contents of the next text area.

The method

```
boolean wasCanceled()
```

returns true, if the user closed the dialog using the "Cancel" button, and false, if the user clicked the "OK" button.

4 Developing morphological operators

4.1 Mathematical Morphology

Morphological image processing has become a standard part of the imaging scientist's toolbox and today is applied daily to a wide range of industrial applications, including (and certainly not limited to) inspection, biomedical imaging, document processing, pattern recognition, metallurgy, microscopy, and robot vision. Because the morphological operations can serve as a universal language for image processing, their application is only limited by the ability to design effective algorithms and efficient computational implementation [12].

Mathematical morphology refers to a branch of nonlinear image processing and analysis developed initially by Georges Matheron [14] and Jean Serra [15] that concentrates on the geometric structure within an image. That structure may be of a macro nature, where the goal is the analysis of shapes such as a tools or printed characters, or may be of a micro nature, where one might be interested in particle distributions or textures generated by small primitives.

In mathematical morphology the binary image space E is defined as $E = \mathbb{Z}^2$ (the space of all possible image pixel locations), and the binary image X – as a set $X \subseteq E$ of foreground pixel locations: $X = \left\{ h \mid X(h) = foreground, \quad h = (i, j) \in \mathbb{Z}^2 \right\}$

The main principle of mathematical morphology is to analyze the geometrical and topological structure of the image X by "probing" that image with another small set $A \subseteq E$ called *structuring element*. By the "probing" we understand applying the structuring element A at every pixel location $h \in E$ and determining the cardinal value $card(X \cap A_h)$ that is the number of elements of X coinciding with elements of A translated along h, i.e. $A_h = \{a + h \mid a \in A\}$. In this way structuring element A defines the positions of analyzed pixels relatively to a given pixel location h. Coinciding of pixels with the translated structuring element is illustrated on the Figure 12.

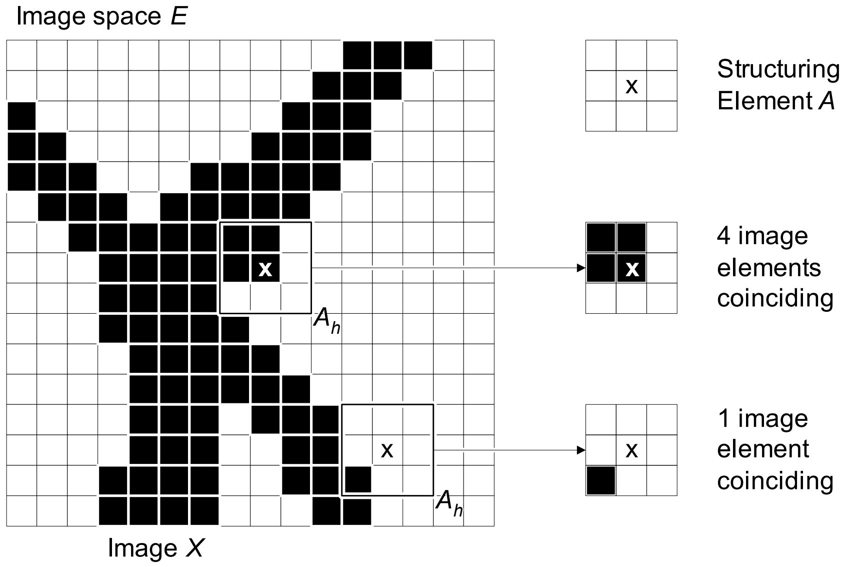


Figure 12. Probing the image with the structuring element.

The *morphological transformation* $\psi_A(X)$ is defined as some transformation of the image using the information retrieved with "probing" of X with structuring element A. The choice of the appropriate structuring element strongly depends on the particular application at hand. This however should not be viewed as a limitation, since it usually leads to additional flexibility in algorithm design.

Classical mathematical morphology is based on set theory with transformations exhibiting all-or-nothing precision of pure logic and therefore such transformations could be sensitive to image noise. Sometimes this logic-based paradigm is referred as *standard* or *crisp morphology*. *Soft morphology* is an alternative approach, in which morphological operations are based on more general weighted order statistics. The definitions of the soft morphological operators are similar to crisp operators but incorporate a factor, r, of how well the structuring element fits within the image. This property makes soft morphological filters to be less sensitive to noise that is very practical feature in many image processing applications [10-13].

Today mathematical morphology is an established discipline in the areas of image and signal processing, filtering, segmentation and image and signal coding, among others. It has strong links with more recent theories such as scale-space and level set methods. On the practical level most introductory books in image analysis and most available commercial imaging software include at least few morphological operators.

4.2 Data representations

4.2.1 Binary images in ImageJ

For binary images we will use the following default photometric assumption that white pixel color corresponds to the image background, when black to the foreground. Using that we assume that empty image is all white by default.

Because ImageJ does not support directly binary images, we use a grayscale image representation to hold binary image. We will assume that pixel value 0 corresponds to a black (foreground) pixel, and pixel value 255 (0xFF) corresponds to a white (background) pixel of binary image. In further when we refer to a binary image we will assume its grayscale representation.

4.2.2 Structuring elements

Structuring element defines a local neighborhood of a pixel where morphological operators are applied. We implemented structuring element concept in class `Element`.

Class:

```
Element
```

Fields:

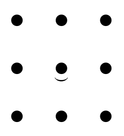
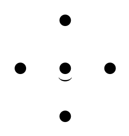
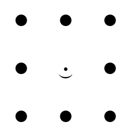
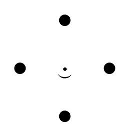
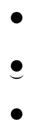
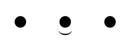
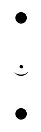
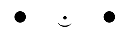
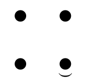
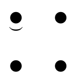
```
int length
int[] xbody
```

```
    int[] ybody
```

Class `Element` represents the structuring element. Structuring element is represented by 2 integer arrays containing x and y coordinates of its pixels. Coordinates are defined relatively to (0,0) which assumed to be in a center of structuring element. For example element CROSS_3X3 (see table below) is defined by two arrays `xbody = {0,-1,0,1,0}` and `ybody = {-1,0,0,0,1}`. `length` is a length of coordinate arrays.

All operators described below use name of structuring element as one of its input parameters. Class Element contains 10 statically predefined structuring elements (see Table 2). Plugins access structuring element via "element" parameter which must contain element's name. For example to use element CROSS_3X3 call plugin with "element='CROSS_3X3'".

Table 2: Basic structuring elements. The origin is marked with "curved underline"

 **BLOCK_3X3**	 **CROSS_3X3**	 **PIERCED_BLOCK_3X3**	 **PIERCED_CROSS_3X3**
 **VERTICAL_BAR_1X3**	 **HORIZONTAL_BAR_3X1**	 **PIERCED_V_BAR_1X3**	 PIERCED_H_BAR_3X1
	 **SMALL_BLOCK_2X2**	 **SMALL_TRANS_BLOCK_ 2X2**	

Here assumed that current pixel is situated in the center of the 3x3 block (bar or cross etc.). Plugins developed below assume that element BLOCK_3X3 applied by default.

4.3 Working principles

All operators are implemented as "plugins". When running, plugins use currently selected image as a source image. Plugin apply its algorithm directly to the source image, so the changes will appear in the same window. This allows to create sequences of plugins to perform operations on the image sequentially, avoiding additional image selection and deleting of temporary results.

Typically plugins need more parameters than just source image. That parameters should be specified in a dialog box or in the command line, if plugin will be executed from the ImageJ's command line or using `IJ.run()` operator. In last case dialog box `GenericDialog` will not appear and parameters will be transferred from the command line into plugin automatically.

4.4 Plugin descriptions

The plugins implemented within this project are summarized in Table 3.

Table 3: The summary of developed morphological plugins

Plugin name	**Plugin description**
Rank	Plugin implements rank operator $\rho_{A,s}(X)$
Median	Plugin implements median operator μ which is equal to $\rho_{A,card(A)/2}$
Erode	Implements morphological erosion $\varepsilon_A(X)$, also denoted as $X \ominus A$
Dilate	Implements morphological dilation $\delta_A(X)$, also denoted as $X \oplus A$
Open	Implements morphological opening, which is $\delta_A(\varepsilon_A(X))$
Close	Implements morphological closing, which is $\varepsilon_A(\delta_A(X))$
OpenClose	Implements sequential opening and closing
CloseOpen	Implements sequential closing and opening
RankMaxOpen	Implements soft analogue of opening operator $\alpha_{A,k} = id \wedge \delta_A \rho_{A,k}$
RankMinClose	Implements soft analogue of closing operator $\beta_{A,k} = id \vee \varepsilon_{\hat{A}} \rho_{A,n+1-k}$
Overfilter	Implements Overfilter operator considered in [12]
Underfilter	Implements Underfilter operator considered in [12]
OpenOverfilter	Implements OpenOverfilter which is sequential Over- and Under- filtering [12]
CloseUnderfilter	Implements CloseUnderfilter which is sequential Under- and Over- filtering [12]

4.4.1 Rank operator

Plugin Name:

Rank

Input:

binary image

"rank" – integer rank parameter:

"element" – string name of structuring element

Output:

binary image

Usage:

```
IJ.run("Rank ", "rank=2 element='BLOCK_3X3'");
```

The rank operator **Rank**(*Element, Rank*) denoted by $\rho_{A,s}(X)$, where *Element* or A is a structuring element defining the neighborhood and *Rank* or *s* is rank parameter, sets pixel to be black if the local neighborhood of the current pixel location contains at least *Rank* black pixels.

Algorithm:

```
Procedure Rank(A : structuring element, s : integer)
For each y do
  For each x do
      Count pixels in intersection of A
      and source image at current pixel location (x, y)
      If that number is bigger than s then
          Put black pixel to a destination
      Else
          Put white pixel to a destination
  End for
End for
End
```

Rank plugin is implemented using 2 nested functions: `run()` - interface function which takes input image and parameters and then calls `doFilter()` function, which takes source and destination `ImageProcessor` classes, structuring element `Element` and integer rank parameter and do the actual ranking. If rank parameter is greater than `Element.length` or less than 0 then it is assumed to be equal to `Element.length`.

4.4.2 RankBasedOperator

Rank plugin is very important, because it could be used to implement a lot of other morphological operators. For example dilation is a ranking with rank parameter set to 1. On the other hand we can not use it directly because rank needs two parameters, element and ranking, when other operators typically needs only element and the value of ranking parameter depends on the nature of the operator or do not necessary at all. This leads us to implement `RankBasedOperator` class which handles structuring element from ImageJ and transfers it to function `doRank` to do actual ranking. `doRank` is an empty function which should be overloaded by a plugin extending RankBasedOperator.

Class:

`RankBasedOperator extends PlugInFrame`

Methods:

`run` – implements parameter handling

`doRank` – empty function to be overloaded by a child plugin

Following functions – Median, Dilation, Erosion, Opening, Closing, OpenClose and CloseOpen are implemented as extensions of `RankBasedOperator`.

4.4.3 Median

Plugin name:

Median

Input:

binary image
"element" – string name of structuring element

Output:

binary image

Usage:

```
IJ.run("Median ", "element='BLOCK_3X3'");
```

Median operator denoted by μ_A is a rank operator with ranking parameter equal to a number of pixels in structuring element divided by two or mathematically $\mu = \rho_{A,card(A)/2}$. Therefore we can calculate that number for given structuring element and give it to already implemented rank operator.

Algorithm:

```
Procedure Median(A : structuring element)
Switch(A)
  n = number of pixels in A divided by 2
  Rank(A, n)
End
```

Median operator extends `RankBasedOperator` class by overloading `doRank` function. Here is the part of actual Java code implementing the plugin (`str` is a string containing the name of a structuring element).

```
IJ.run("Rank","rank="+IJ.d2s((int)(element.length/2))+"element="+str);
IJ.run("Rename...","title='Median with "+str+"'");
```

4.4.4 Erode

Plugin name:

Erode

Input:

binary image
"element" – string name of structuring element

Output:

binary image

Usage:

```
IJ.run("Erode ", "element='BLOCK_3X3'");
```

Erosion operator is denoted by $\varepsilon_A(X)$ or $X \ominus A$. As it has been shown in [13], erosion could be implemented via rank operator: $\varepsilon_A = \rho_{A,\|A\|}$, where $\|A\|$ is the number of pixels in the structuring element A.

Algorithm:

```
Procedure Erode(A : structuring element)
  n = number of pixels in A
  Rank(A, n)
End
```

Erode operator extends `RankBasedOperator` class by overloading `doRank` function. The main part of the plugins Java code is (`str` is a string containing the name of a structuring element):

```
IJ.run("Rank ","rank=-1 element="+str);
IJ.run("Rename...","title='Erosion with "+str+"'");
```

4.4.5 Dilate

Plugin name:

Dilate

Input:

binary image

“element” – string name of structuring element

Output:

binary image

Usage:

```
IJ.run("Dilate ", "element='BLOCK_3X3'");
```

Dilation operator is denoted by $\delta_A(X)$ or $X \oplus A$. Implementation of dilation is much the same as implementation of erosion. As it has been shown in [12], dilation could be implemented via rank operator $\delta_A = \rho_{A,1}$. We don’t even need to calculate number of pixels in A.

Algorithm:

```
Procedure Dilate(A : structuring element)
  Rank(A, 1)
End
```

Dilate operator extends `RankBasedOperator` class by overloading `doRank` function. The main part of the plugins Java code is (`str` is a string containing the name of a structuring element):

```
IJ.run("Rank ","rank=1 element="+str);
IJ.run("Rename...","title='Dilation with "+str+"'");
```

4.4.6 Open

Plugin name:

Open

Input:

binary image
"element" – string name of structuring element

Output:

binary image

Usage:

```
IJ.run("Open ", "element='BLOCK_3X3'");
```

As defined, opening operator is a result of applying erosion and dilation sequentially. Therefore it is natural to use plugins described above to implement it as follows.

Algorithm:

```
Procedure Open(A : structuring element)
  Erode(A)
  Dilate(A)
End
```

Open operator extends `RankBasedOperator` class by overloading `doRank` function. The main part of the plugins Java code is (`str` is a string containing the name of a structuring element):

```
IJ.run("Erode ","element='"+ str +"'");
IJ.run("Dilate ","element='"+ str +"'");
IJ.run("Rename...","title='Opening with "+ str +"'");
```

4.4.7 Close

Plugin name:

Close

Input:

binary image
"element" – string name of structuring element

Output:

binary image

Usage:

```
IJ.run("Close  ", "element='BLOCK_3X3'");
```

In a similar to previous section we implement closing operator in a same manner. Closing operator is a result of sequential application of dilation and erosion.

Algorithm:

```
Procedure Close(A : structuring element)
  Dilate(A)
  Erode(A)
End
```

Close operator extends `RankBasedOperator` class by overloading `doRank` function. The main part of the plugins Java code is (`str` is a string containing the name of a structuring element):

```
IJ.run("Dilate ","element='"+str+"'");
IJ.run("Erode ","element='"+str+"'");
IJ.run("Rename...","title='Closing with "+str+"'");
```

4.4.8 OpenClose

Plugin name:

OpenClose

Input:

binary image

"element" – string name of structuring element

Output:

binary image

Usage:

```
IJ.run("OpenClose  ", "element='BLOCK_3X3'");
```

This plugin is also implemented using already developed plugins by sequential applying of open and close operators. Therefore its implementation does not demand any detailed observation. OpenClose operator extends `RankBasedOperator` class by overloading `doRank` function. The main part of the plugins Java code is (`str` is a string containing the name of a structuring element):

```
IJ.run("Open ","element='"+str+"'");
IJ.run("Close ","element='"+str+"'");
IJ.run("Rename...","title='OpenClosing with "+str+"'");
```

4.4.9 CloseOpen

Plugin name:

CloseOpen

Input:

binary image
"element" – string name of structuring element

Output:

binary image

Usage:

```
IJ.run("CloseOpen  ", "element='BLOCK_3X3'");
```

This plugin is also implemented using already developed plugins by sequential applying of close and open operators. Therefore its implementation does not demand any detailed observation. `CloseOpen` operator extends `RankBasedOperator` class by overloading `doRank` function. The main part of the plugins Java code is (`str` is a string containing the name of a structuring element):

```
IJ.run("Close ","element='"+str+"'");
IJ.run("Open ","element='"+str+"'");
IJ.run("Rename...","title='CloseOpening with "+str+"'");
```

4.4.10 Rank-max opening

Plugin name:

RankMaxOpen

Input:

binary image
"rank" – integer rank parameter:
"element" – string name of structuring element

Output:

binary image

Usage:

```
IJ.run("RankMaxOpen  ","rank=5 element='BLOCK_3X3'");
```

Rank-max opening operator $\alpha_{A,k} = id \wedge \delta_A \rho_{A,k}$ also could be implemented using plugins developed above. But here we can not use `RankBasedOperator` because rank-max needs two parameters. Algorithm is as follows:

Algorithm:

```
Procedure RankMaxOpening (A : structuring element, k: integer)
  Image T = Create copy of input image.
  Do Rank(A,k) with input image
  Do Dilate(A) with input image
  Image R = input image AND T
  Delete T
  Return R
End
```

4.4.11 Rank-min closing

Plugin name:

RankMinClose

Input:

binary image

“rank” – integer rank parameter:

“element” – string name of structuring element

Output:

binary image

Usage:

```
IJ.run("RankMinClose ","rank=5 element='BLOCK_3X3'");
```

Rank-min closing operator $\beta_{A,k} = id \vee \varepsilon_{\hat{A}}\rho_{A,n+1-k}$, where $\hat{A}$ is the *reflection* of A and n is the number of pixels in A. We can avoid reflecting of structuring element by using the fact that $\varepsilon_{\hat{A}}(X) = \delta_A(X^c)^c$, where X^c is the *complement* of X. This allows us to use Dilate plugin to implement the following algorithm.

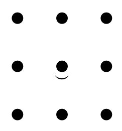
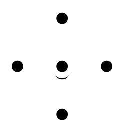
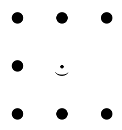
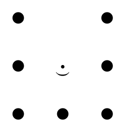
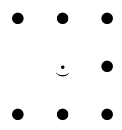
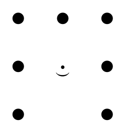
Algorithm:

```
Procedure RankMinClosing (A : structuring element, k: integer)
  Image T = Create copy of input image.
  Do Rank(A,n+1-k) with input image
  Do Invert with input image
  Do Dilate(A) with input image
  Do Invert with input image
  Image R = input image OR T
  Delete T
  Return R
End
```

4.4.12 InfOverBasedOperator

Inf- and over- operators typically use their own predefined structuring elements. As in [13], we use following structuring elements, when implementing inf- and over- operators.

Table 4: Structuring elements for inf- and over- operators

 **BLOCK_3X3**	 **CROSS_3X3**
 **A1**	 **A2**
 **B1**	 **B2**

All structuring elements are statically defined in class `Element`.

There are 2 types of inf- and over- operators supported. They are ones referred as TYPE1 and TYPE2. Plugins discussed below use the type of the filter as its input parameter. This leads us to implement a base class `InfOverBasedOperator` to be a parent of all following plugins. It implements `run` function to handle parameters and call `doFilter` function to be overloaded and implements actual filtering.

Class:

`InfOverBasedOperator extends PlugInFrame`

Methods:

`run` – implements parameter handling

`doFilter` – empty function to be overloaded by a child plugin

4.4.13 Overfilter

Plugin name:

Overfilter

Input:

binary image

"type" – string name of filter type (TYPE1 or TYPE2)

Output:

binary image

Usage:

```
IJ.run("Overfilter ","type='TYPE1'");
```

Overfilter plugin implements TYPE1 and TYPE2 overfilters.

TYPE1 filter is $\phi(X) = (X \ominus PIERCED\_BLOCK\_3X3) \oplus BLOCK\_3X3$

TYPE2 filter is $\phi(X) = ((X \ominus A1) \cup (X \ominus A2)) \oplus BLOCK\_3X3$

Plugin is implemented as a child of `InfOverBasedOperator class` using plugins developed above and its implementation is very close to mathematical notation (see source code).

4.4.14 Underfilter

Plugin name:

Underfilter

Input:

binary image
"type" – string name of filter type (TYPE1 or TYPE2)

Output:

binary image

Usage:

```
IJ.run("Underfilter ","type='TYPE1'");
```

Underfilter plugin implements TYPE1 and TYPE2 overfilters:

TYPE1 filter is $\psi(X) = (X \oplus PIERCED\_BLOCK\_3X3) \ominus BLOCK\_3X3$

TYPE2 filter is $\psi(X) = ((X \oplus B1) \cap (X \oplus B2)) \ominus BLOCK\_3X3$

Plugin is implemented as a child of `InfOverBasedOperator class` using plugins developed above and its implementation is very close to mathematical notation (see source code).

4.4.15 Open Overfilter

Plugin name:

OpenOverfilter

Input:

binary image
“type” – string name of filter type (TYPE1 or TYPE2)

Output:

binary image

Usage:

```
IJ.run("OpenOverfilter ","type='TYPE1'");
```

OpenOverfilter is an operator $(\phi\psi)(X)$. It is implemented as sequential applying of Overfilter and Underfilter.

4.4.16 Close Underfilter

Plugin name:

CloseUnderfilter

Input:

binary image
“type” – string name of filter type (TYPE1 or TYPE2)

Output:

binary image

Usage:

```
IJ.run("CloseUnderfilter ","type='TYPE1'");
```

CloseUnderfilter is an operator $(\psi\phi)(X)$. It is implemented as sequential applying of Underfilter and Overfilter.

4.5 Macro-operations

With ImageJ one can easily create macros to implement new filters using existing ones from within ImageJ's environment. Here we present two simple macros implementing Alternative-Sequential filters. Alternative sequential filters are the compositions of openings and closings with increasing structuring element [16]. The result performance of the filter depends on the order (open-close or close-open), the shape of structuring element and the speed of its increasing.

4.5.1 AS-filter macro

First macro implements AS-filter Open-Close using Erosion and Dilation plugin.

Macro:

```
for(i=1; i<4; i++) {
  for(k=0; k<i; k++) {
    run("Erode ","element=BLOCK_3X3");
  }
  for(k=0; k<i; k++) {
    run("Dilate ","element=BLOCK_3X3");
  } \\ open operation
  for(k=0; k<i; k++) {
    run("Dilate ","element=BLOCK_3X3");
  }
  for(k=0; k<i; k++) {
    run("Erode ","element=BLOCK_3X3");
  } \\ close operation
}
```

Notice, that we can easily modify that macro to implement Close-Open AS-filter. One just need to change Dilate and Erode respectively.

4.5.2 Rank AS-filter macro

Second macro implements Ranked AS-filter using Rank-max Opening and Rank-min Closing plugins.

Macro:

```
for(i=0; i<9; i++) {
  run("RankMinClose ","rank="+i+" element=BLOCK_3X3");
  run("RankMaxOpen ","rank="+i+" element=BLOCK_3X3");
}
```

This macro implements Close-Open AS-filter, but it is easy to modify it to implement Open-Close AS-filter.

4.6 Changes in the system

Adding custom plugin

To add your own plugin to our package you just need to install your plugin in the same folder, which has to be a subfolder of ImageJ's "Plugins" folder. Also you plugin have to have at least one underscore "_" in its filename to be automatically integrated into ImageJ's menu. For further details see Section 3.1.2.

Adding custom structuring element

To add your own structuring element you must follow these guidelines:

Add definition of your structuring element into `Element.java` file. New element have to be static field of class `Element`. You have to specify its length and X and Y coordinate arrays. Then recompile `Element.java`. Now your element became accessible as a static field of `Element` class.

Also you can dynamically define your custom structuring element anywhere.

Within all developed plugins only the `Rank` uses `Element` directly. All other plugins use `Rank` and do not deal with class `Element` itself.

To add your structuring element into dialog box parameter list you have to add it into dialog's choise list control (as described in Section 3.1.7).

See `RankBasedOperator` source code to have an example of handling structuring elements.

5 Experiments and evaluation

5.1 Plugin illustrations

Here we present a set of images illustrating applying of implemented morphological operators over the test image. All operators were applied over the following 256x256 binary image (House affected by salt-and-pepper noise). This image was used in works of H. Heijmans [12].

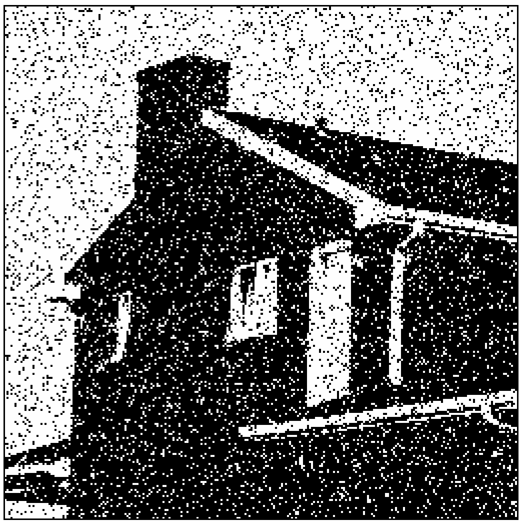


Figure 13: Original image

As most of implemented filters can use a lot of different structural elements and some of them have also rank parameter (e.g. Rank-max Opening use 10 structuring elements and a rank parameter 1…9) we present below just a one example for each plugin. But in those examples we will try to use different structuring elements and parameters to give an illustrative observation of whole functionality.

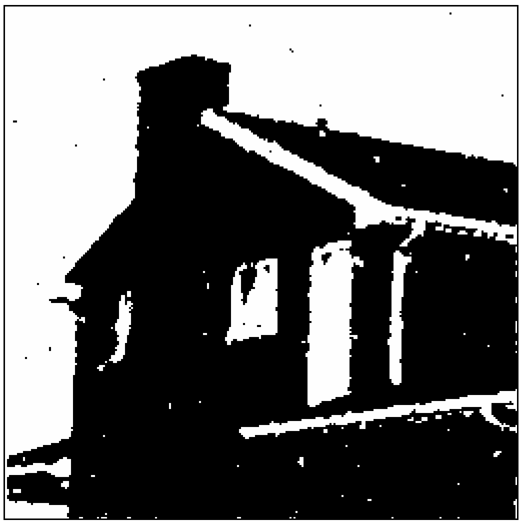


Figure 14: Rank 5 with block 3x3

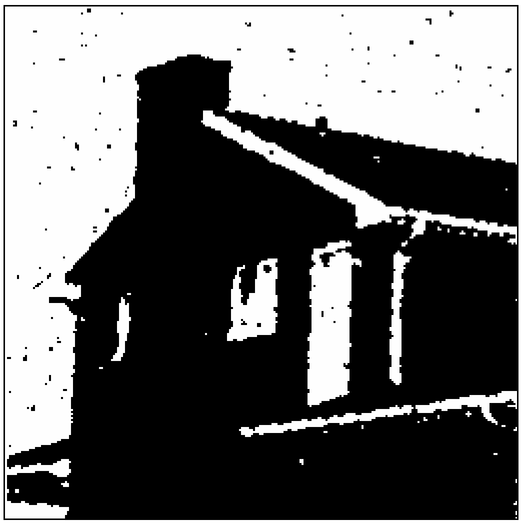


Figure 15: Median with block 3x3

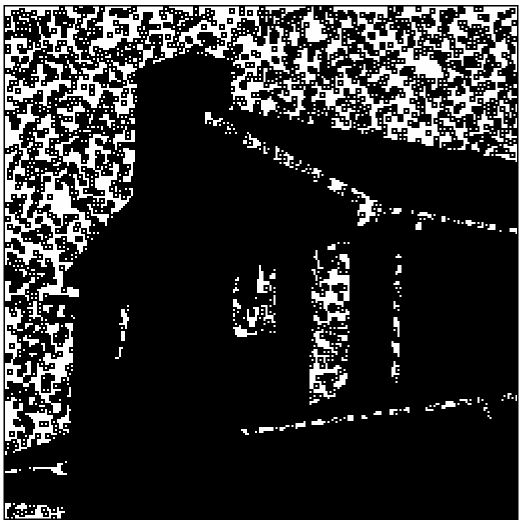


Figure 16: Dilation with pierced block 3x3

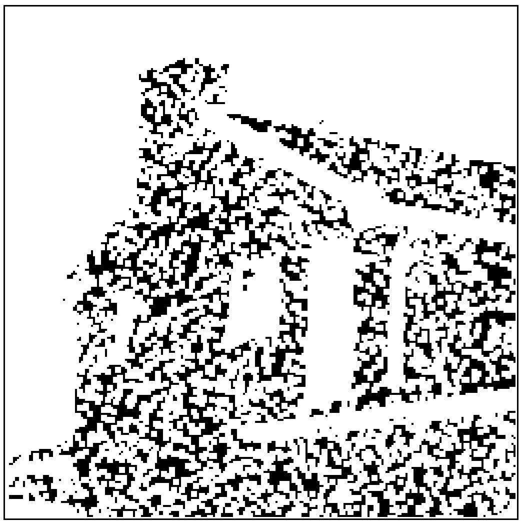


Figure 17: Erosion with block 3x3

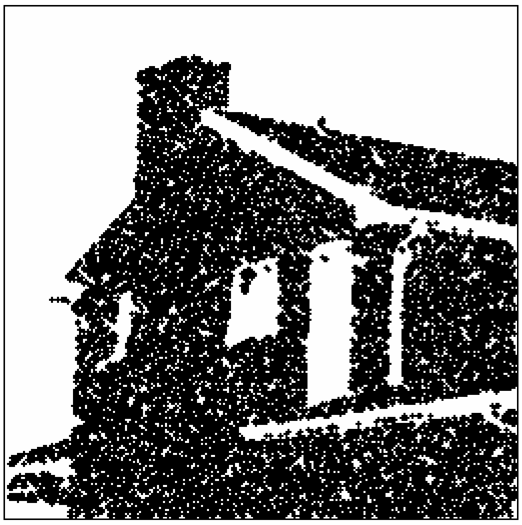


Figure 18: Opening with cross 3x3

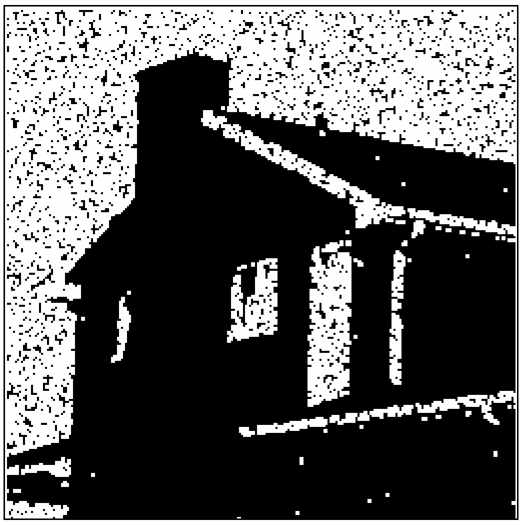


Figure 19: Closing with small block 2x2

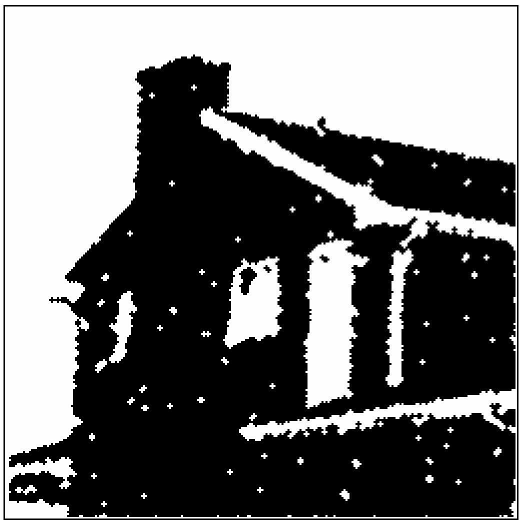


Figure 20: OpenClosing with block 3x3

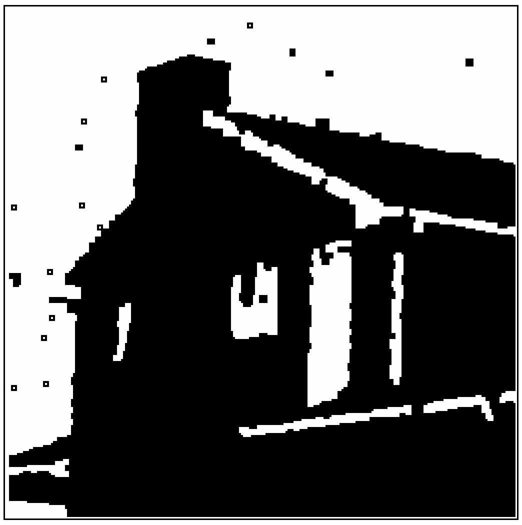


Figure 21: CloseOpening with pierced block 3x3

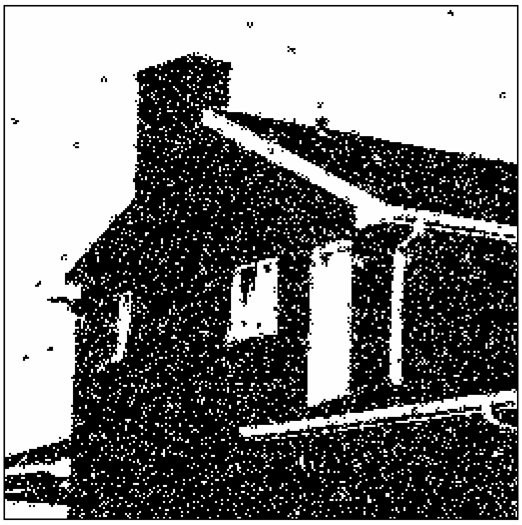


Figure 22: Rank-max opening with block 3x3

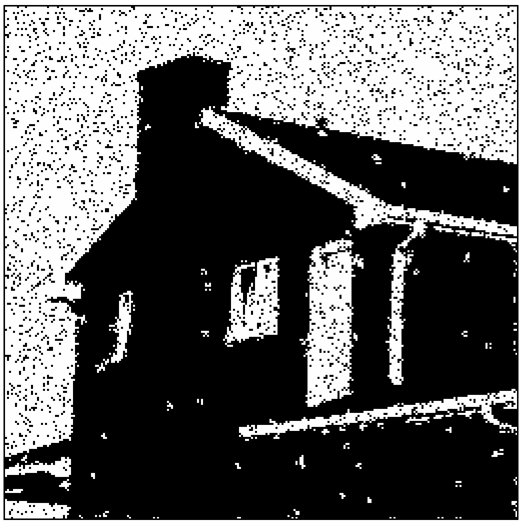


Figure 23: Rank-min closing with block 3x3

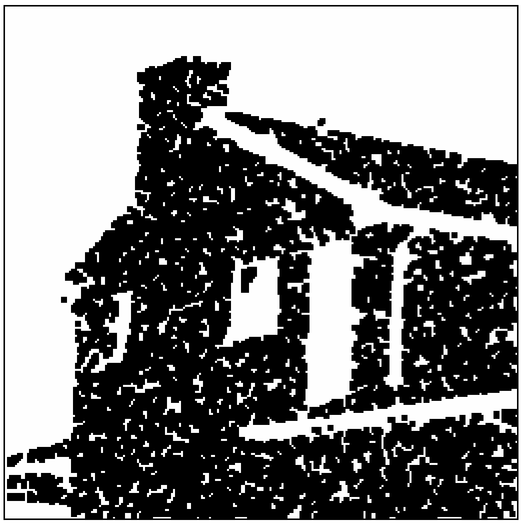


Figure 24: Overfilter TYPE1

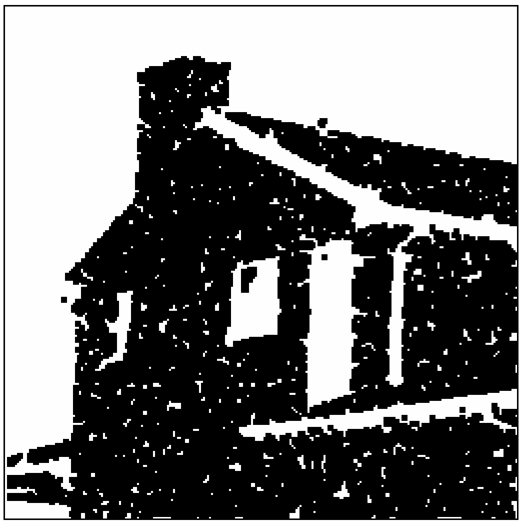


Figure 25: Overfilter TYPE2

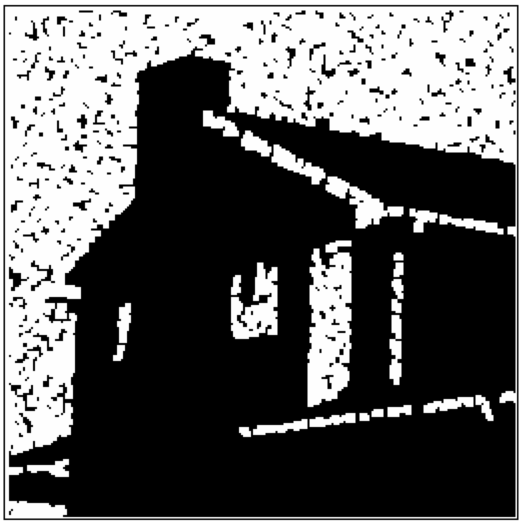


Figure 26: Underfilter TYPE1

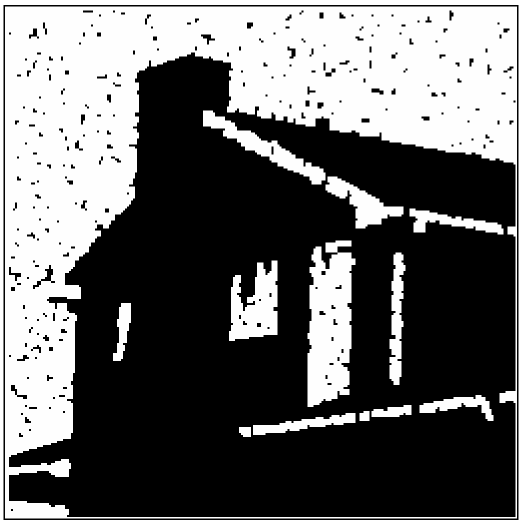


Figure 27: Underfilter TYPE2

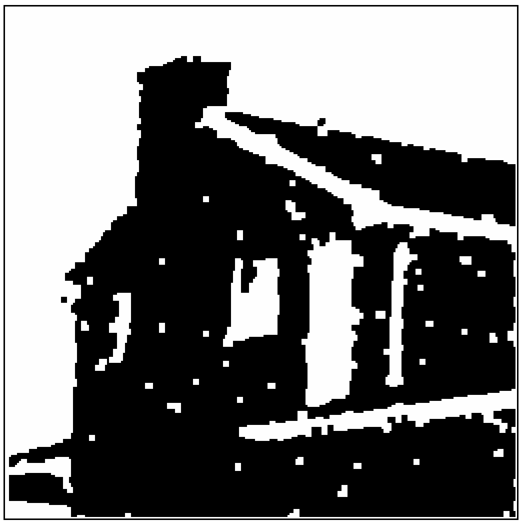


Figure 28: OpenOverfilter TYPE1

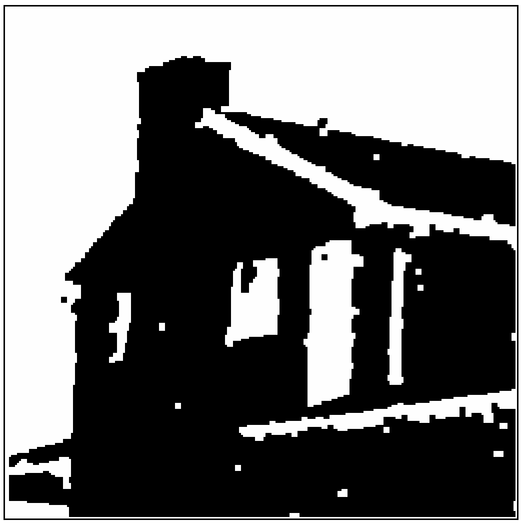


Figure 29: OpenOverfilter TYPE2

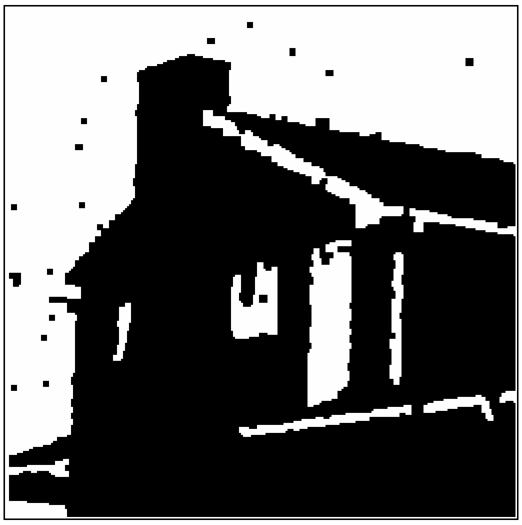


Figure 30: CloseUnderfilter TYPE1

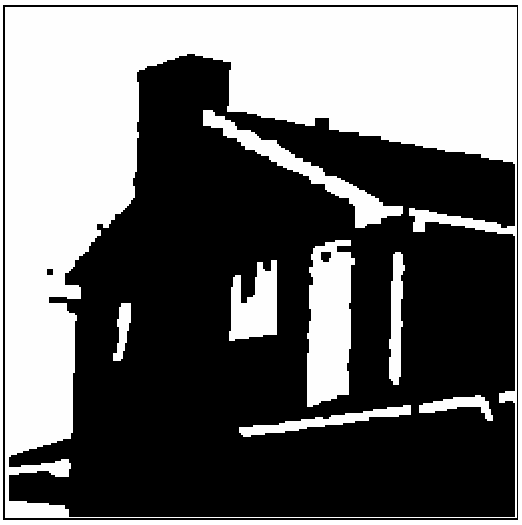


Figure 31: CloseUnderfilter TYPE2

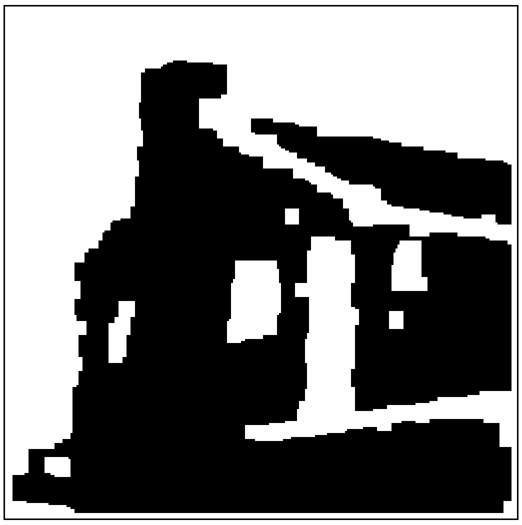


Figure 32: AS-filter Open-Close with block 3x3

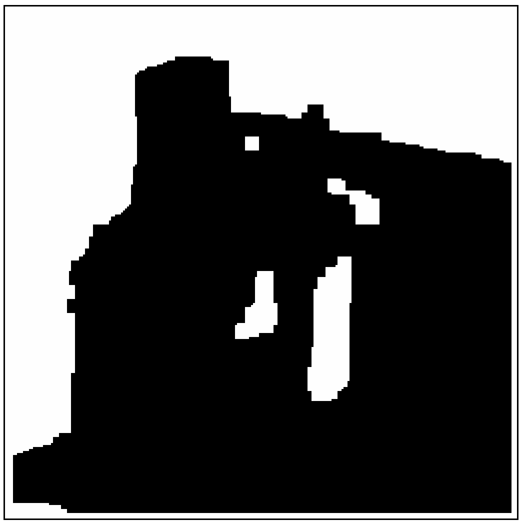


Figure 33: AS-filter Close-Open with block 3x3

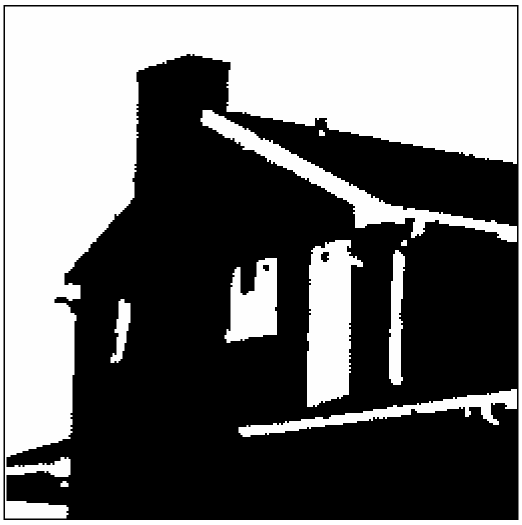


Figure 34: Rank AS Close-Open with block 3x3

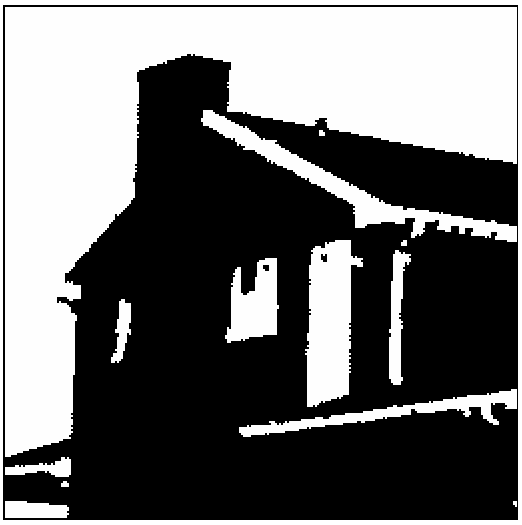


Figure 35: Rank AS Open-Close with block 3x3

5.2 Execution time

Here we evaluate execution times for several plugins. All plugins were applied for the large 5000×5000 pixels test image. The Pentium 4, 1.80GHz, 256MB RAM based computer running Microsoft Windows XP Professional has been used for execution time evaluation. The Table 5. represents average execution times for implemented morphological plugins. We have used ImageJ version 1.3.2j running on Java Virtual Machine version 1.3.1_03 by Sun Microsystems. For comparison we give running times for the morphological filtering algorithms implemented by us in C-language and compiled to native Windows (WinApi) code using Microsoft Visual Studio 6 compiler suite. The algorithms were executed several hundreds times and average running time has been computed. As one can see the speed of about 3 Mpixels per second has been obtained.

Table 5. Average execution times for morphological plugins using large 25 Mpixels image.

Plugin	Rank	Median	Dilation	Erosion	Open	Close	OpenClose	CloseOpen
ImageJ execution time, **s**	8,2	7,4	7,4	7,3	14,6	14,9	34,4	32,2
Winapi execution time, **s**	8,1	7,7	9,0	6,5	12,8	12,9	26,06	25,85

For reference we have also applied developed plugins for a small 256×256 reference image discussed in the previous section and have obtained the results shown in the Table 6. Even though the plugins written in Java are working slower for small images, the running times are still quite small for the delay to be noticed. On large images the plugins works almost as fats as the algorithms written in native windows environment, and in some cases even faster (probably due to Windows file caching and similar factors.

Table 6. Average execution times for morphological plugins using small 65Kpixels image.

Plugin	Rank	Median	Dilation	Erosion	Open	Close	OpenClose	CloseOpen
ImageJ execution time, **ms**	45.94	58.12	52.82	50.3	111.56	98.74	265.3	234.36
Winapi execution time, **ms**	7.34	7.9	7.77	8.12	10.97	10.93	20.01	19.99

6 Conclusions

In this work we have shown how the algorithms in image processing can be rapidly developed using ImageJ Java-based framework. As a test case we have implemented morphological operations such as Rank, Median, Erosion, Dilation, Open, Close, OpenClose, CloseOpen, Rank-Max Opening, Rank-Min Closing, Overfilter, Underfilter, Open Overfilter and Close Underfilter. Every operator supports 10 different structuring elements. Also we have shown how the new operators can be created using ImageJ's built-in macro language by implementing AS-filters using Open and Close operators.

In our experience, the ImageJ package has shown a good usability. A lot of important features are included and most of popular image formats are supported. Further, ImageJ allows rapid development and using of already developed plugins in the implementation of new ones. For instance, we have used our already developed basic plugins in the course of developing new more sophisticated ones. Another important feature is that ImageJ framework is an open-source project, not demanding any kind of licensing or registration fees. Because it is Java-based, it can be executed on virtually any computer environment where Java Virtual Machine is installed. And finally execution times of the algorithms developed in ImageJ can be compared with the running times of the algorithms implemented in low-level languages and compiled to the system native machine code.

7 References

[1] **David S. Bright**, Digital Image Processing with NIH Image (Mac) / Scion Image (PC) / ImageJ, Surface and Microanalysis Science Division, National Institute of Standards and Technology, Gaithersburg, MD 20899-8371, Report, 2004, http://www.nist.gov/lispix/imlab/labs.html

[2] **W. Rasband,** *ImageJ*, Research Services Branch, National Institutes of Health, USA, http://rsb.info.nih.gov/ij/

[3] *Documentation on ImageJ:* Web-resource: http://rsb.info.nih.gov/ij/docs/index.html

[4] **W. Burger, M. J. Burge,** *Digitale Bilderverarbeitung*, (in German), 330 Seiten - Springer, Berlin, 330 pages, November 2004.

[5] **D. Flanagan**, *Java in a Nutshell: A Desktop Quick Reference* (Java Series), 648 pages, O'Reilly & Associates, 3rd edition (November 1999).

[6] *The Sun Java Series*, Detailed information can be found at http://java.sun.com/docs/books/

[7] **A. Podlasov, E. Ageenko,** "Morphological Reconstruction of Semantic Layers in Map Images", University of Joensuu, Department of Computer Science, Technical Report Series, A-2004-1, Joensuu, Finland, 2004.

[8] **W. Bailer,** Writing ImageJ Plugins – A Tutorial, Upper Austria University of Applied Sciences, Dept. of Media Technology and Design, Hagenberg, Austria, Research Report 1.6, June 26, 2003

[9] *Image J Application Programmer Interface*, Web-resource, http://rsb.info.nih.gov/ij/developer/api/index.html

[10] **Doughberty E., Lotufo R.,** *Hands-on morphological image processing*. SPIE Optical Engineering Press, 2003.

[11] **Gonzalez R.C., Woods R.E.,** *Digital image processing*, 2nd edition, Addison-Wesley, 2002.

[12] **Heijmans H.J.A.M.,** *Morphological image operators*. Boston: Academic Press, 1994.

[13] **Heijmans H.J.A.M.**, "A new class of alternating sequential filters" *1995 IEEE Workshop on Nonlinear Signal and Image Processing*, Neos Marmaras, Halkidiki, Greece, June 20-22, 1995.

[14] **Matheron G.** *Random Sets and Integral Geometry*, J. Wiley & Sons, New York, 1975.

[15] **Serra J.,** *Image Analysis and Mathematical morphology*. London: Academic Press, 1982.

[16] **J. Goutsias and H.J.A.M. Heijmans** *Fundamenta Morphologicae Mathematicae* Fundamenta Informaticae 41 (1-2), p.1-31, 2000