Java Advanced Imaging API: A Tutorial

Rafael Santos[1]

Abstract: This tutorial shows how the Java language and its Java Advanced Imaging (JAI) Application Program Interface (API) can be used to create applications for image representation, processing and visualization. The Java language advantages are its low cost, licensing independence and inter-platform portability. The JAI API advantages are its flexibility and variety of image processing operators.
The purpose of this tutorial is to present the basic concepts of the JAI API, including several complete an verified code samples which implements simple image processing and visualization operations. At the end of the tutorial the reader should be able to implement his/her own algorithms using the Java language and the JAI API.

1 Introduction

In spite of the existence of several image processing softwares with many image processing functions, tailored for several different uses, there is often the need for implementation of specific algorithms which are not available on those softwares – for example, a user may want to implement his/her own image classification or filtering algorithm or tweak some already implemented algorithm parameters. Some of those softwares allow the development of user-defined modules, often using the same API developed for the software itself. The developer may be able to use those APIs to develop his/hew own routines, but often there is an additional cost or licensing restrictions.

A royalty-free, portable, flexible alternative for the implementation of generic applications is the Java language [1]. For image processing and representation, the JAI (Java Advanced Imaging) API (Application Program Interface) [2] can be used. Although the API is not part of a full-featured image processing software, the existing functions and extension possibilities allied to the low cost and ease of implementation makes this combination an attractive option for image processing algorithms development.

This tutorial will present some concepts on the JAI API and give code samples and short code snippets for image input and output, application of basic operators, image visualization and image data manipulation. The tutorial will not present some details like installation and configuration issues or advanced operations such as network imaging. It is assumed that the reader already have a good knowledge of Java or other modern languages (C++, Delphi) and basic image processing knowledge.

[1]Divisão de Sensoriamento Remoto – Instituto de Estudos Avançados – Centro Técnico Aeroespacial
santos@ieee.org

Instructions for installation the JAI libraries and running applications which use the JAI classes can be found in [2, 3]. This tutorial assumes that the user will have access to a complete JDK (Java Development Kit) installation (version 1.4 or later) with the JAI API installed (version 1.1.2 or later).

2 Image data representation

Image processing algorithms usually require the manipulation of the image data (pixels). In this section the model used by JAI for image data storage and manipulation will be presented, with the corresponding Java/JAI classes.

Images in JAI may be multidimensional (i.e. with several values associated to a single pixel) and may have pixel with either integer or floating point values (altough there are restrictions on the types of images which can be stored in disk). Pixels may be packed in different ways or unpacked in the image data array. Different color models can be used. As one may expect, in order to be able to represent a variety of image data, one must deal with a variety of classes.

Before showing examples of those classes, the basic classes for iamge data representation will be shown. Some of those classes are abstract, concrete subclasses of those behave on more or less the same way:

`PlanarImage`: Basic class for image representation in JAI, allows the representation of images with more flexibility than the Java class `BufferedImage`. Both `BufferedImage` and `PlanarImage` uses several different classes for flexible image data representation: its pixels are stored in an instance of `Raster` which contains an instance of a concrete subclass of `DataBuffer`, packed accordingly to the rules descriped by an instance of a concrete subclass of `SampleModel`. An instance of `PlanarImage` also have a `ColorModel` associated to it, which contains an instance of `ColorSpace`, which determines how a pixel's value can be translated to color values. Figure 1 shows how those classes are used to compose an instance of `PlanarImage`.
A `PlanarImage` is read-only, i.e. it may be created and its pixels values may be read in several different ways, but there are no methods that allow the modification of pixels values. `PlanarImage`s may have the origin of the image in a position different from the coordinate $(0, 0)$, or even pixel coordinates with negative values.

`TiledImage`: A subclass of `PlanarImage`, which can be used for reading **and** writing image data.

`RenderedOp`: Another subclass of `PlanarImage`, which represents a node in a *rendered imaging chain*. A rendered imaging chain is a powerful and interesting concept of JAI

which allows the processing of an image to be specified as a series of steps (operators and parameters) which are applied to one or more images.

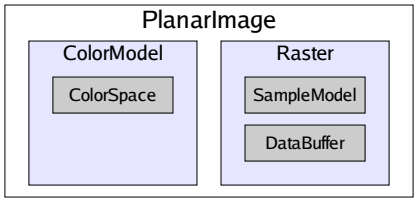

Figure 1. `PlanarImage` structure (after [3])

Another interesting concept used in JAI are *tiled images*. Tiles can be considered as subsets of the images which may be processed independently. Large images thus can be processed in Java/JAI with reasonable performance, even through rendered imaging chains, since there is no need to load the whole image data in memory at once. If the image is tiled, all its tiles must have the same width and height. JAI allows different origins for the pixels and for the tiles on an image, although there are few, if any, practical applications for this.

Figure 2 shows a simple tiled image, where the origin of the tiles coincides with the origin of the image but with the tiles extended past the image edges (as it is often the case). When a tile extends past the image edges, its contents are undefined. More information on tiled images may be found in [4].

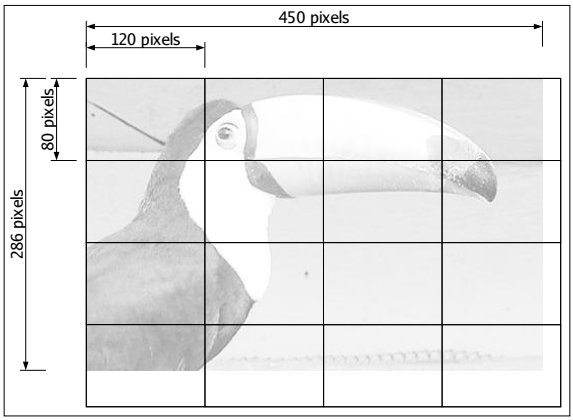

Figure 2. A tiled image.

With the knowledge of which classes are used for image data representation, it is relatively simple to create an image in-memory for storage or further processing.

Two different examples of creation of images will be presented, the first one will be the creation of a grayscale image with a floating-point pixel data, and the second will be the creation of a RGB image with integer pixel data. Both examples will use the following simple steps:

1. Create the image data in an array in memory. This array must be an unidimensional array, although for simplicity a multidimensional array can be created and flattened later.
2. Create an instance of a concrete subclass of `DataBuffer`, using one of its constructors and the image data array.
3. Create an instance of `SampleModel` with the same data type of the `DataBuffer` and desired dimensions. A factory method of the class `RasterFactory` may be used for this.
4. Create an instance of `ColorModel` compatible with the sample model being used. The static method `PlanarImage.createColorModel` may be used for this, using the sample model as an argument.
5. Create an instance of `WritableRaster` using the sample model and the image data array. The method `RasterFactory.createWritableRaster` can be used for this.
6. Create a writable image (instance of `TiledImage`) using the sample model, color model and dimensions.
7. Associate the instance of `Raster` with the image using the method `setData` of the class `TiledImage`.
8. Do something with the instance of `TiledImage`, like saving it to disk, displaying or processing it.

The code for those steps (as a complete Java application) that will create, as a result, a floating-point one-banded (grayscale) image is shown in listing 1. Due to space constraints, only the essential working code and comments will be shown in this tutorial. The reader may find more complete code for this and other examples, with comments, on [5].

Listing 1: Class `CreateGrayImage`.

```
1  package sibgrapi.tutorial;
2
3  import java.awt.Point;
4  import java.awt.image.*;
5  import javax.media.jai.*;
6
7  public class CreateGrayImage
8    {
9    public static void main(String[] args)
10     {
11     int width = 1024; int height = 1024; // Dimensions of the image.
12     float[] imageData = new float[width*height]; // Image data array.
```

```
    int count = 0; // Auxiliary counter.
    for(int w=0;w<width;w++) // Fill the array with a degradé pattern.
      for(int h=0;h<height;h++)
        imageData[count++] = (float)(Math.sqrt(w+h));
    // Create a DataBuffer from the values on the image array.
    javax.media.jai.DataBufferFloat dbuffer =
      new javax.media.jai.DataBufferFloat(imageData,width*height);
    // Create a float data sample model.
    SampleModel sampleModel =
       RasterFactory.createBandedSampleModel(DataBuffer.TYPE_FLOAT,
                                             width,height,1);
    // Create a compatible ColorModel.
    ColorModel colorModel = PlanarImage.createColorModel(sampleModel);
    // Create a WritableRaster.
    Raster raster = RasterFactory.createWritableRaster(sampleModel,dbuffer,
                                                       new Point(0,0));
    // Create a TiledImage using the float SampleModel.
    TiledImage tiledImage = new TiledImage(0,0,width,height,0,0,
                                           sampleModel,colorModel);
    // Set the data of the tiled image to be the raster.
    tiledImage.setData(raster);
    // Save the image on a file.
    JAI.create("filestore",tiledImage,"floatpattern.tif","TIFF");
    }
  }
```

Similar code for creation of a RGB image with a simple red and blue pattern is shown in listing 2. Again, the same basic steps are used, although instances of different concrete classes that inherit from `DataBuffer` and `SampleModel` are used.

Listing 2: Class `CreateRGBImage`.

```
package sibgrapi.tutorial;

import java.awt.*;
import java.awt.image.*;
import javax.media.jai.*;

public class CreateRGBImage
  {
  public static void main(String[] args)
    {
    int width = 121; int height = 121; // Dimensions of the image
    byte[] data = new byte[width*height*3]; // Image data array.
    int count = 0; // Temporary counter.
    for(int w=0;w<width;w++) // Fill the array with a pattern.
      for(int h=0;h<height;h++)
        {
        data[count+0] = (count % 2 == 0) ? (byte)255: (byte)  0;
        data[count+1] = 0;
        data[count+2] = (count % 2 == 0) ? (byte)  0: (byte)255;
        count += 3;
        }
```

```
22      // Create a Data Buffer from the values on the single image array.
23      DataBufferByte dbuffer = new DataBufferByte(data,width*height*3);
24      // Create an pixel interleaved data sample model.
25      SampleModel sampleModel =
26         RasterFactory.
27           createPixelInterleavedSampleModel(DataBuffer.TYPE_BYTE,
28                                             width,height,3);
29      // Create a compatible ColorModel.
30      ColorModel colorModel = PlanarImage.createColorModel(sampleModel);
31      // Create a WritableRaster.
32      Raster raster = RasterFactory.createWritableRaster(sampleModel,dbuffer,
33                                                         new Point(0,0));
34      // Create a TiledImage using the SampleModel.
35      TiledImage tiledImage = new TiledImage(0,0,width,height,0,0,
36                                             sampleModel,colorModel);
37      // Set the data of the tiled image to be the raster.
38      tiledImage.setData(raster);
39      // Save the image on a file.
40      JAI.create("filestore",tiledImage,"rgbpattern.tif","TIFF");
41      }
42    }
```

In order to get information about an existing image, several `get` methods from the classes `PlanarImage`, `SampleModel` and `ColorModel` can be used. Several of those methods are demonstrated in the code on the listing 3, which is a complete Java application which must get, as a command-line parameter, the file name of an existing image.

Listing 3: Class `ImageInfo`.

```
1   package sibgrapi.tutorial;
2
3   import java.awt.Transparency;
4   import java.awt.image.*;
5   import java.io.File;
6   import javax.media.jai.*;
7
8   public class ImageInfo
9     {
10    public static void main(String[] args)
11      {
12      // Open the image (using the name passed as a command line parameter)
13      PlanarImage pi = JAI.create("fileload", args[0]);
14      // Get the image file size (non-JAI related).
15      File image = new File(args[0]);
16      System.out.println("Image file size: "+image.length()+" bytes.");
17      // Show the image dimensions and coordinates.
18      System.out.print("Dimensions: ");
19      System.out.print(pi.getWidth()+"x"+pi.getHeight()+" pixels");
20      // Remember getMaxX and getMaxY return the coordinate of the next point!
21      System.out.println(" (from "+pi.getMinX()+","+pi.getMinY()+" to " +
22                         (pi.getMaxX()-1)+","+(pi.getMaxY()-1)+")");
23      if ((pi.getNumXTiles() != 1)||(pi.getNumYTiles() != 1)) // Is it tiled?
```

```
24        {
25        // Tiles number, dimensions and coordinates.
26        System.out.print("Tiles: ");
27        System.out.print(pi.getTileWidth()+"x"+pi.getTileHeight()+" pixels"+
28                         " ("+pi.getNumXTiles()+"x"+pi.getNumYTiles()+" tiles)");
29        System.out.print(" (from "+pi.getMinTileX()+","+pi.getMinTileY()+
30                           " to "+pi.getMaxTileX()+","+pi.getMaxTileY()+")");
31        System.out.println(" offset: "+pi.getTileGridXOffset()+","+
32                                       pi.getTileGridXOffset());
33        }
34      // Display info about the SampleModel of the image.
35      SampleModel sm = pi.getSampleModel();
36      System.out.println("Number of bands: "+sm.getNumBands());
37      System.out.print("Data type: ");
38      switch(sm.getDataType())
39        {
40        case DataBuffer.TYPE_BYTE:     System.out.println("byte"); break;
41        case DataBuffer.TYPE_SHORT:    System.out.println("short"); break;
42        case DataBuffer.TYPE_USHORT:   System.out.println("ushort"); break;
43        case DataBuffer.TYPE_INT:      System.out.println("int"); break;
44        case DataBuffer.TYPE_FLOAT:    System.out.println("float"); break;
45        case DataBuffer.TYPE_DOUBLE:   System.out.println("double"); break;
46        case DataBuffer.TYPE_UNDEFINED:System.out.println("undefined"); break;
47        }
48      // Display info about the ColorModel of the image.
49      ColorModel cm = pi.getColorModel();
50      if (cm != null)
51        {
52        System.out.println("Number of color components: "+
53                           cm.getNumComponents());
54        System.out.println("Bits per pixel: "+cm.getPixelSize());
55        System.out.print("Transparency: ");
56        switch(cm.getTransparency())
57          {
58          case Transparency.OPAQUE:      System.out.println("opaque"); break;
59          case Transparency.BITMASK:     System.out.println("bitmask"); break;
60          case Transparency.TRANSLUCENT:
61            System.out.println("translucent"); break;
62          }
63        }
64      else System.out.println("No color model.");
65      }
66    }
```

3 Simple JAI operators

The JAI API contains several image operators which can be applied with minimum programming. Those operators follow the concept of a rendered imaging chain, where the steps for the image processing are defined but will be carried only when needed (*deferred execution*).

Those operations are specified in a simple way: first an instance of `ParameterBlock`

is created, which is basically a vector of data that will be used for the operation, then the static method `create` of the class `JAI` is executed. This method gets as an argument a name for the operation and the instance of `ParameterBlock` and returns an instance of `RenderedOp` which can be manipulated as a `PlanarImage`. Alternatively one can add the original image in the instance of `ParameterBlock` as a parameter to its `addSource` method. Other parameters are added to the `ParameterBlock` with its `add` method. Other forms of the method does not require a `ParameterBlock` and accept other arguments.

One example of a JAI operator is the "filestore" operator, used in the code in the listings 1 and 2 to store an instance of `PlanarImage` (or of a subclass of it) in a file. The call for the `JAI.create` method used as arguments the name of the operator, the instance of `PlanarImage`, a file name and a string containing the desired image file name ("TIFF", "JPEG", "PNG", etc.).

Another example of operator, which does not use the instance of `ParameterBlock` was already shown in listing 3: a call to `JAI.create("fileload",imageName);` will load and return an image which file name is contained on the string `imageName`. Other operators and code snippets that illustrate its usage will be shown in this section. A list of all operators can be found on the JAI API documentation [6], on the documentation for the package `javax.media.jai.operator`.

The "invert" operator requires a simple `PlanarImage` as input, and can be executed as shown in the code in listing 4, which shows how to read and invert an image.

Listing 4: Code for image inversion.

```
1 // Read the image. Assume args[0] points to its filename.
2 PlanarImage input = JAI.create("fileload", args[0]);
3 // Invert the image.
4 PlanarImage output = JAI.create("invert", input);
```

The "scale" operator scales one image giving a scaled version as a result. It optionally may also translate the image. To use this operator, one need to create a `ParameterBlock` and add the original image, two floating point values corresponding to the X and Y scale and another two floating point values corresponding to the translation in X and Y of the images' pixels. When scaling an image, interpolation of the pixels must be performed, therefore one need also to add to the parameter block an instance of a concrete subclass of `javax.media.jai.Interpolation`. The code in listing 5 shows one example of usage of this operator.

Listing 5: Code for image scaling.

```
1 float scale = 2.0f;
2 ParameterBlock pb = new ParameterBlock();
```

```
pb.addSource(image);
pb.add(scale);
pb.add(scale);
pb.add(0.0f);
pb.add(0.0f);
pb.add(new InterpolationNearest());
PlanarImage scaledImage = JAI.create("scale", pb);
```

The "rotate" operator rotates one image using an angle in radians. Similarly to the "scale" operator, it also needs an interpolation method. In order to use this operator, one must create a `ParameterBlock`, add an image source to it, and add (in this order) the rotation angle, the two coordinates for the center of the rotation and an instance of a concrete subclass of `Interpolation`. The code in listing 6 shows one example of usage of the "rotate" operator, which rotates an image 45 degrees around its center.

Listing 6: Code for image rotation.

```
float angle = (float)Math.toRadians(45);
float centerX = image.getWidth()/2f;
float centerY = image.getHeight()/2f;
ParameterBlock pb = new ParameterBlock();
pb.addSource(image);
pb.add(centerX);
pb.add(centerY);
pb.add(angle);
pb.add(new InterpolationBilinear());
PlanarImage scaledImage = JAI.create("rotate", pb);
```

Convolution can be easily done with JAI. The "convolve" operator performs convolution of an image with a kernel, which can be created as an instance of the class `KernelJAI`. This instance is created with an array which represents the kernel values, then the instance of `KernelJAI` may be used even without a `ParameterBlock`. The code in listing 7 shows how one can create a 15×15 smoothing kernel and apply it to an input image, giving as a result an output image. The kernel values must be normalized, i.e. they must sum up to one.

Listing 7: Code for image smoothing.

```
int kernelSize = 15;
float[] kernelMatrix = new float[kernelSize*kernelSize];
for(int k=0;k<kernelMatrix.length;k++)
   kernelMatrix[k] = 1.0f/(kernelSize*kernelSize);
KernelJAI kernel = new KernelJAI(kernelSize,kernelSize,kernelMatrix);
PlanarImage output = JAI.create("convolve", input, kernel);
```

As another example, the code in listing 8 shows how one can create and apply a horizontal Sobel operator to an input image.

Listing 8: Code for Sobel edge detection in an image.

```
float[] kernelMatrix = { -1, -2, -1,
                          0,  0,  0,
                          1,  2,  1 };
KernelJAI kernel = new KernelJAI(3,3,kernelMatrix);
PlanarImage output = JAI.create("convolve", input, kernel);
```

It is possible to use some operators to manipulate the whole bands in an image. For example, one can select some bands of a multiband image to create another image. The operator "bandselect" uses an input image and an array of integer band indexes to select bands from that image and add them in the specified order in the output image. The code in listing 9 shows how one can invert a RGB image by selecting the reverse order (BGR) of its bands. Usage of a `ParameterBlock` is not needed in this case.

Listing 9: Code for inverting a RGB image through band selection.

```
PlanarImage output = JAI.create("bandselect",input,new int[] {2,1,0});
```

Another band manipulation operator is the "bandcombine" operator, which uses several image bands to combine them into a single multiband image. This method could be used to create a RGB image from three separate red, green and blue images, for example. The code snippet in listing 10 assumes that there is an array of image file names, read those images into an array of instances of `PlanarImage`s using the "fileload" operator, then add those images to an instance of `ParameterBlock` (in the same order they were read). Finally, the "bandcombine" operator combine all images in the `ParameterBlock` and stores the result in a TIFF image through the "filestore" operator.

Listing 10: Code for creating a multiband image from several separated bands.

```
PlanarImage[] inputs = new PlanarImage[args.length];
for(int im=0;im<args.length;im++)
  inputs[im] = JAI.create("fileload", args[im]);
ParameterBlock pb = new ParameterBlock();
for(int im=0;im<args.length;im++)
  pb.setSource(inputs[im], im);
PlanarImage result = JAI.create("bandmerge",pb,null);
JAI.create("filestore",result,"multiband.tiff","TIFF");
```

Some other simple operators are "add", "subtract", "multiply" and "divide", which performs basic arithmetic operations on two images, giving a third as result. The code snipped shown in listing 11 shows how two images (which are presumably already created or read from files) can be added, subtracted, multiplied or divided depending on which button on an user interface was clicked.

Listing 11: Code for performing arithmetic operations on two images.

```
ParameterBlock pb = new ParameterBlock();
pb.addSource(input1);
pb.addSource(input2);
if (e.getSource() == add)
  output = JAI.create("add", pb);
else if (e.getSource() == subtract)
  output = JAI.create("subtract", pb);
else if (e.getSource() == multiply)
  output = JAI.create("multiply", pb);
else if (e.getSource() == divide)
  output = JAI.create("divide", pb);
```

4 Image data access

Often there is the need to access individual pixel values from an image in order to perform some operation on that image – classification algorithms, for example, often require access to all pixels on an image for evaluation and classification.

One simple method that can be used to access pixels from an image is through the use of *iterators*. Iterators allow the access to the pixels on an image in a specific order. For example, an instance of `RectIter` scans the image column by column, from the top line to the bottom line, updating automatically the scanned pixels' coordinates and allowing the access (reading only) for all pixels in a band or for a pixel in a particular band at the present scan coordinate. Another iterator, `RandomIter` allows the direct access to a pixel using user-specified X and Y coordinates.

Both iterators are actually interfaces, and opaque instances of classes that implement those interfaces can be obtained through factory methods in classes `RectIterFactory` and `RandomIterFactory`, respectively. The factory methods require two arguments: one instance of `PlanarImage` (which will be the image being processed) and one instance of `Rectangle`, which will determine which rectangular subregion of the image will be considered for processing. If `null` is used instead of an instance of `Rectangle`, the whole image will be considered for processing.

The code snippet in listing 12 shows how one can dump all pixels' data values of an image to the console. The code gets the image dimensions, creates an array suitable for being filled with a pixel of the image (which may be a multiband image) and, iterating over all the pixels on the image, get its values and print them. Note that the method `nextPixel` must be called to increase the coordinates for the pixels.

Listing 12: Accessing all pixels in an image (using `RectIter`).

```
int width = pi.getWidth();
int height = pi.getHeight();
```

```
SampleModel sm = pi.getSampleModel();
int nbands = sm.getNumBands();
int[] pixel = new int[nbands];
RectIter iterator = RectIterFactory.create(pi, null);
for(int h=0;h<height;h++)
  for(int w=0;w<width;w++)
    {
    iterator.getPixel(pixel);
    System.out.print("at ("+w+","+h+"): ");
    for(int band=0;band<nbands;band++)
      System.out.print(pixel[band]+" ");
    System.out.println();
    iterator.nextPixel();
    }
```

The code in listing 13 is similar to the one in listing 12, except that an instance of `RandomIter` is created and used, so when the method `getPixel` is called, one must provide X and Y coordinates to it.

Listing 13: Accessing all pixels in an image (using `RandomIter`).

```
int width = pi.getWidth();
int height = pi.getHeight();
SampleModel sm = pi.getSampleModel();
int nbands = sm.getNumBands();
int[] pixel = new int[nbands];
RandomIter iterator = RandomIterFactory.create(pi, null);
for(int h=0;h<height;h++)
  for(int w=0;w<width;w++)
    {
    iterator.getPixel(w,h,pixel);
    System.out.print("at ("+w+","+h+"): ");
    for(int band=0;band<nbands;band++)
      System.out.print(pixel[band]+" ");
    System.out.println();
    }
```

Although pixel data acessing with iterators is quite simple and straightforward, it causes some overhead on the performance of the applications, since, for each pixel, there must be some method calls (with image boundary verification). A faster pixel data acessing method is through the image raster.

As seen on section 2, the image pixels are stored in a `Raster`, which encapsulates both a `DataBuffer` and a `SampleModel`. The developer does not need to concern how the pixels are packed inside the `Raster`, its `getPixel` method and variants will get the pixels as a data array, while its `getSample` method and variants will get a single data point (band of a pixel) from the image data. By getting a raster from the image and a data region from it, there will be fewer method calls and less overhead, so the application may perform

better. On the other hand, since processing will be done by image chunks, more memory may be required, depending on the size of the region used for processing.

The code snipped in listing 14 shows how one can access all pixels in an image through the image's `Raster`. The code is similar to the shown in listings 12 and 13, except that an instance of `Raster` is created by calling the method `getData` on class `PlanarImage`, then the method `getPixels` of the instance of `Raster` is called to get all the pixels of the image in a suitable structure, which must have the required dimensions. The method `getPixels` get as parameters the coordinates of the upper left pixel location and its width and height, and a reference to the array which will get the data. In the example, the whole image was used as data source. It must be pointed that since the array which will get the data must be unidimensional, proper tracking of the pixels and band coordinates must be done.

Listing 14: Accessing all pixels in an image (using `Raster.getPixels`).

```
1  int width = pi.getWidth();
2  int height = pi.getHeight();
3  SampleModel sm = pi.getSampleModel();
4  int nbands = sm.getNumBands();
5  Raster inputRaster = pi.getData();
6  int[] pixels = new int[nbands*width*height];
7  inputRaster.getPixels(0,0,width,height,pixels);
8  int offset;
9  for(int h=0;h<height;h++)
10   for(int w=0;w<width;w++)
11     {
12     offset = h*width*nbands+w*nbands;
13     System.out.print("at ("+w+","+h+"): ");
14     for(int band=0;band<nbands;band++)
15       System.out.print(pixels[offset+band]+" ");
16     System.out.println();
17     }
```

`PlanarImage`s and `Raster`s are read-only, but it is easy to create an application that process the images' pixels and store them for further use. From the instance of `Raster` one can create an instance of `WritableRaster` with the same structure (but without the pixels' values) calling the method `Raster.createCompatibleWritableRaster`. The pixels' values can be obtained as shown in listing 14. After processing the pixels' values through the data array, the array can be stored again on the `WritableRaster` through its `setPixels` method, which arguments are the same as the used in `Raster.getPixels`.

A `Raster` or `WritableRaster` cannot be inserted again on a `PlanarImage`, but it is easy to create a `TiledImage` calling one of its constructors, which uses as parameters an instance of an already existing `PlanarImage` and the desired tiles width and height. The `TiledImage` will have the same dimensions and other features as the original

`PlanarImage` and its `setData` method, with gets as an argument an instance of `Raster` or `WritableRaster` can be used to set its data. This `TiledImage` then can be further processed or stored.

Listing 15 shows the whole process. That listing shows a simple application where all pixels with values equal to zero are changed to 255. The input to the code is a `PlanarImage`, and its output is a `TiledImage` with the original values changed.

Listing 15: Accessing all pixels in an image (for reading and writing).

```
int width = pi.getWidth();
int height = pi.getHeight();
SampleModel sm = pi.getSampleModel();
int nbands = sm.getNumBands();
Raster inputRaster = pi.getData();
WritableRaster outputRaster = inputRaster.createCompatibleWritableRaster();
int[] pixels = new int[nbands*width*height];
inputRaster.getPixels(0,0,width,height,pixels);
int offset;
for(int h=0;h<height;h++)
  for(int w=0;w<width;w++)
    {
    offset = h*width*nbands+w*nbands;
    for(int band=0;band<nbands;band++)
      if (pixels[offset+band] == 0) pixels[offset+band] = 255;
    }
outputRaster.setPixels(0,0,width,height,pixels);
TiledImage ti = new TiledImage(pi,1,1);
ti.setData(outputRaster);
```

It is also possible to use writable iterators – for example, an instance of `WritableRandomIter` can be created through the method `RandomIterFactory.createWritable` and passing to this method an instance of `TiledImage` and an instance of `Rectangle` to set the bounds for the iterator or `null` to use the whole image. A writable iterator can be used in a similar way as a read-only iterator. The writable iterator will set the data directly on the output image, through its `setPixel` or `setSample` methods.

5 Simple visualization

Visualization is an important step on an image processing application. Although it is possible to do read an image, process it and store the results on disk and use external applications to view those results, there are certain types of images which can be processed and stored but not easily viewed with generic applications – floating-point images and multiband images, for example. It may also more interesting to do the processing and visualization on a single Java application instead of relying on external applications.

The JAI API provides a simple but extensible component for image display, implemented by the class `DisplayJAI`. This component inherits from `JPanel` and may be used as any other Java graphical component. This component can be used as-is or extended for different purposes.

One simple example is shown in listing 16. This complete Java application displays the image which file name is passed as a command line argument. An instance of `DisplayJAI` is created, using as argument for its constructor an instance of `PlanarImage` (the image on a `DisplayJAI` can be changed later through its `set` method). The instance of `DisplayJAI` is associated with a `JScrollPane` so images larger than the screen can be viewed through scrolling.

Listing 16: Simple usage of the `DisplayJAI` component.

```
1  package sibgrapi.tutorial;
2
3  import java.awt.*;
4  import javax.media.jai.*;
5  import javax.swing.*;
6  import com.sun.media.jai.widget.DisplayJAI;
7
8  public class DisplayJAIExample
9    {
10   public static void main(String[] args)
11     {
12     // Load the image which file name was passed as the first argument to
13     // the application.
14     PlanarImage image = JAI.create("fileload", args[0]);
15     // Get some information about the image
16     String imageInfo =
17       "Dimensions: "+image.getWidth()+"x"+image.getHeight()+
18       " Bands:"+image.getNumBands();
19     // Create a frame for display.
20     JFrame frame = new JFrame();
21     frame.setTitle("DisplayJAI: "+args[0]);
22     // Get the JFrame's ContentPane.
23     Container contentPane = frame.getContentPane();
24     contentPane.setLayout(new BorderLayout());
25     // Create an instance of DisplayJAI.
26     DisplayJAI dj = new DisplayJAI(image);
27     // Add to the JFrame's ContentPane an instance of JScrollPane
28     // containing the DisplayJAI instance.
29     contentPane.add(new JScrollPane(dj),BorderLayout.CENTER);
30     // Add a text label with the image information.
31     contentPane.add(new JLabel(imageInfo),BorderLayout.SOUTH);
32     // Set the closing operation so the application is finished.
33     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
34     frame.setSize(400,200); // adjust the frame size.
35     frame.setVisible(true); // show the frame.
36     }
37   }
```

A screenshot of the application in listing 16 is shown in figure 3. The application assumes that the image can be displayed without problems, but will yield an exception if images with more than three bands are used.

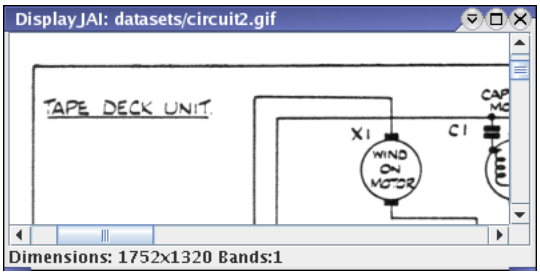

Figure 3. Screenshot of the `DisplayJAI` usage example application (listing 16).

5.1 Visualization with a surrogate image

The `DisplayJAI` component is able to display images which data type is not integer (e.g. floating point images) but its results are undefined – there is no explicit or controlable conversion of the image data. In this section, an example of extension of the `DisplayJAI` component will be shown. This example has two interesting points: it uses a *surrogate* image for display, which will be created from the original image data; and it also allows some basic user interaction, so the user can see the original value of the image pixel under the mouse cursor.

The component will be tailored for the displaying of digital elevation model (DEM) images, which are one-banded floating-point images, where the pixels are not a measure of a visible feature of the image but the elevation over the ocean level. In order to create a surrogate image which will visually represent the DEM, one must create a normalized and reformatted (casted) version of the original floating point image. The surrogate image pixels' values will be on the range $[0, 255]$, normalized considering the minimum and maximum values of the DEM – in other words, all pixels on the surrogate image will be calculated as the value of the corresponding DEM pixel multiplied by $255/(max - min)$ and added to min, where max is the maximum DEM value and min the minimum DEM value. The surrogate image data type will also be set to byte.

In order to create the surrogate image with these rules, three JAI operators will be used. Those operators were not shown in section 3, therefore their description and usage will be presented now. The first operator is the "extrema" operator, which does not use any other parameter except for an input image. After this operator is applied, the user can call the method `getProperty` of the resulting `RenderedOp` using "maximum" or "minimum" as

arguments to get arrays of double values corresponding to the maximum and minimum pixel values per band. On this example, the DEM image is considered to have only one band.

The second operator that will be used in this example is the "rescale" operator, which uses as parameters (using a `ParameterBlock`) an input image, an array of double values for multiplication of the input image pixels and another array of double values for addition to the image pixels. If the dimension of those arrays is the same as the number of bands on the image, the multiplication and addition will be done on a per band basis, otherwise only the first value on the arrays will be used. The resulting image pixels are calculated as $output = input \times m + a$ where m and a are the arrays for multiplication and addition, respectively.

The third operator used to create the surrogate image is the "format" operator, which get as parameters the input image and one of the constants `TYPE_BYTE`, `TYPE_SHORT`, `TYPE_USHORT`, `TYPE_INT`, `TYPE_FLOAT` or `TYPE_DOUBLE`, which are defined in the class `DataBuffer`. The resulting image data will be casted to the type corresponding to the `DataBuffer` constant.

Listing 17 shows the code for the modified component. This component (`DisplayDEM`) creates the surrogate image on its constructor, using the original image and the described steps for normalization and reformatting, also creating a `RandomIter` to obtain the original image pixels values. Part of the code on listing 17 allows the component to store the image data under the current mouse position, and to export those values as a `StringBuffer`.

Listing 17: Code for the `DisplayDEM` component.

```
1  package sibgrapi.tutorial;
2
3  import java.awt.event.*;
4  import java.awt.image.*;
5  import java.awt.image.renderable.*;
6  import javax.media.jai.*;
7  import javax.media.jai.iterator.*;
8  import com.sun.media.jai.widget.DisplayJAI;
9
10 public class DisplayDEM extends DisplayJAI implements MouseMotionListener
11   {
12   protected StringBuffer pixelInfo; // Pixel information (formatted as a
13                                     // StringBuffer).
14   protected double[] dpixel; // Pixel information as an array of doubles.
15   protected RandomIter readIterator; // a RandomIter that allow us to get
16                                      // the data of a single pixel.
17   protected PlanarImage surrogateImage; // The surrogate byte image.
18   protected int width,height; // Dimensions of the image
19   protected double minValue,maxValue; // Range of the image values.
20
```

```
21  /**
22   * The constructor of the class, which creates the data structures and
23   * surrogate image.
24   */
25   public DisplayDEM(RenderedImage image)
26     {
27     readIterator = RandomIterFactory.create(image, null);
28     // Get some facts about the image
29     width = image.getWidth();
30     height = image.getHeight();
31     dpixel = new double[image.getSampleModel().getNumBands()];
32     // We need to know the extrema of the image to create the surrogate
33     // image. Let's use the extrema operator to get them.
34     ParameterBlock pbMaxMin = new ParameterBlock();
35     pbMaxMin.addSource(image);
36     RenderedOp extrema = JAI.create("extrema", pbMaxMin);
37     double[] allMins = (double[])extrema.getProperty("minimum");
38     double[] allMaxs = (double[])extrema.getProperty("maximum");
39     minValue = allMins[0]; // Assume that the image is one-banded.
40     maxValue = allMaxs[0];
41     // Rescale the image with the parameters
42     double[] multiplyByThis = new double[1];
43     multiplyByThis[0] = 255./(maxValue-minValue);
44     double[] addThis = new double[1];
45     addThis[0] = minValue;
46     // Now we can rescale the pixels gray levels:
47     ParameterBlock pbRescale = new ParameterBlock();
48     pbRescale.add(multiplyByThis);
49     pbRescale.add(addThis);
50     pbRescale.addSource(image);
51     surrogateImage = (PlanarImage)JAI.create("rescale", pbRescale);
52     // Let's convert the data type for displaying.
53     ParameterBlock pbConvert = new ParameterBlock();
54     pbConvert.addSource(surrogateImage);
55     pbConvert.add(DataBuffer.TYPE_BYTE);
56     surrogateImage = JAI.create("format", pbConvert);
57     set(surrogateImage);
58     // Create the StringBuffer instance for the pixel information.
59     pixelInfo = new StringBuffer(50);
60     addMouseMotionListener(this); // Registers the mouse motion listener.
61     }
62
63   // This method is here just to satisfy the MouseMotionListener interface.
64   public void mouseDragged(MouseEvent e) { }
65
66   // This method will be called when the mouse is moved over the image.
67   public void mouseMoved(MouseEvent me)
68     {
69     pixelInfo.setLength(0); // Clear the StringBuffer
70     int x = me.getX(); // Get the mouse coordinates.
71     int y = me.getY();
72     if ((x >= width) || (y >= height)) // Avoid exceptions, consider only
73       {                                // pixels within image bounds.
74       pixelInfo.append("No data!");
75       return;
76       }
```

```
77      pixelInfo.append("(DEM data) "+x+","+y+": ");
78      readIterator.getPixel(x,y,dpixel); // Read the original pixel value.
79      pixelInfo.append(dpixel[0]); // Append to the StringBuffer.
80      } // end of method mouseMoved
81
82    // Allows other classes to access the pixel info string.
83    public String getPixelInfo()
84      {
85      return pixelInfo.toString();
86      }
87    }
```

The section of code which uses the “extrema” operator is between lines 34 and 40 of the code in listing 17. Usage of the “rescale” operator is shown between lines 42 and 51, and usage of the “format” operator is between lines 53 and 56.

The `DisplayDEM` component can be used in any Java application with a graphical user interface. This application may or not use the original image pixel information that can be obtained through the `DisplayDEM` component. One example of application is shown in listing 18 – it is a simple application, which uses the component with a `JLabel` to show the original image value for the pixel under the mouse cursor.

Listing 18: Application which uses the `DisplayDEM` component.

```
1  package sibgrapi.tutorial;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.media.jai.*;
6  import javax.swing.*;
7
8  public class DisplayDEMApp extends JFrame implements MouseMotionListener
9    {
10   private DisplayDEM dd; // An instance of the DisplayDEM component.
11   private JLabel label; // Label to display information about the image.
12
13   public DisplayDEMApp(PlanarImage image)
14     {
15     setTitle("Move the mouse over the image !");
16     getContentPane().setLayout(new BorderLayout());
17     dd = new DisplayDEM(image); // Create the component.
18     getContentPane().add(new JScrollPane(dd),BorderLayout.CENTER);
19     label = new JLabel("---"); // Create the label.
20     getContentPane().add(label,BorderLayout.SOUTH);
21     dd.addMouseMotionListener(this); // Register mouse events.
22     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23     setSize(400,200);
24     setVisible(true);
25     }
26
```

```
27    // This method is here just to satisfy the MouseMotionListener interface.
28    public void mouseDragged(MouseEvent e) { }
29
30    // This method will be executed when the mouse is moved over the
31    // application.
32    public void mouseMoved(MouseEvent e)
33      {
34      label.setText(dd.getPixelInfo()); // Update the label with the
35                                        // DisplayDEM instance info.
36      }
37
38    public static void main(String[] args)
39      {
40      PlanarImage image = JAI.create("fileload", args[0]);
41      new DisplayDEMApp(image);
42      }
43    }
```

The figure 4 shows a screenshot of the `DisplayDEMApp` application (listing 18). The the bottom part of the application shows the image coordinates and original DEM value under the mouse cursor.

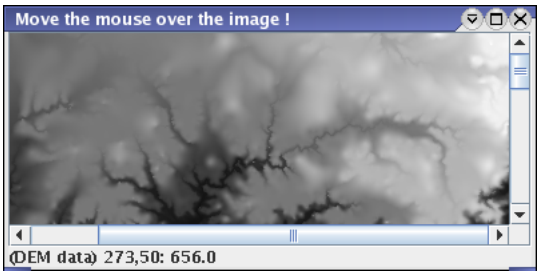

Figure 4. Screenshot of the `DisplayDEMApp` application (listing 18).

5.2 Visualization of images with annotations

Another frequent task in image processing applications is the display of an image with some kind of annotations over it – markers on the image, text, delimiters for regions of interest, etc. In this section a more complete set of classes will be described that allow the non-interactive creation of generic annotations and the display over images, using another extension of the `DisplayJAI` class.

In order to give a more complete and extensible example, an abstract class which encapsulates a drawable annotation is first devised. The class `DrawableAnnotation` is shown in listing 19, and simply declares an abstract `paint` method and a `Color` field, with a `set` and a `get` method for this field. Concrete classes that inherit from the `Drawable-`

`Annotation` class must implement the `paint` method, which will draw the intended annotation using an instance of `Graphics2D` as the drawing context.

Listing 19: Abstract class that encapsulates a drawable annotation.

```
1  package sibgrapi.tutorial;
2
3  import java.awt.*;
4
5  public abstract class DrawableAnnotation
6    {
7    private Color color;
8
9    public abstract void paint(Graphics2D g2d);
10
11   public void setColor(Color color)
12     {
13     this.color = color;
14     }
15
16   public Color getColor()
17     {
18     return color;
19     }
20   }
```

A concrete implementation of the drawable annotation class is shown in listing 20. That class allows the drawing of a diamond-shaped annotation, using as parameters for its constructor a central point for the annotation and the diamond-shaped width and height in pixels and a pen width (to allow the creation of annotations which will be drawn with different pen widths).

Listing 20: Class that encapsulates a diamond-shaped annotation.

```
1  package sibgrapi.tutorial;
2
3  import java.awt.*;
4  import java.awt.geom.*;
5
6  public class DiamondAnnotation extends DrawableAnnotation
7    {
8    private Point2D center; // Annotation center point.
9    private double width; // Width of diamond annotation.
10   private double height; // Height of diamond annotation.
11   private BasicStroke stroke; // "Pen" used for drawing.
12
13   // Constructor for the class.
14   public DiamondAnnotation(Point2D c,double w,double h,float pw)
15     {
```

```
    center = c;
    width = w;
    height = h;
    stroke = new BasicStroke(pw);
    }

  // Concrete implementation of the paint method.
  public void paint(Graphics2D g2d)
    {
    int x = (int)center.getX();
    int y = (int)center.getY();
    int xmin = (int)(x-width/2);
    int xmax = (int)(x+width/2);
    int ymin = (int)(y-height/2);
    int ymax = (int)(y+height/2);
    g2d.setStroke(stroke);
    g2d.setColor(getColor());
    g2d.drawLine(x,ymin,xmin,y);
    g2d.drawLine(xmin,y,x,ymax);
    g2d.drawLine(x,ymax,xmax,y);
    g2d.drawLine(xmax,y,x,ymin);
    }
  }
```

The main class in this section is the class that inherits from `DisplayJAI` and can display an image and draw annotations (instances of classes that inherit from `DrawableAnnotation`) over it. Annotations are stored as a list, and the class provides a method for adding annotations to the list.

This class overrides the `paint` method of the `DisplayJAI` class so after the image is painted (through a call to `super.paint`) all instances of annotations on the list have their `paint` method executed, using the same graphic context used to draw the image.

The code for the class that inherits from `DisplayJAI` (`DisplayJAIWithAnnotations`) is shown in listing 21.

Listing 21: Extension of the `DisplayJAI` class that draws annotations over the image.

```
package sibgrapi.tutorial;

import java.awt.*;
import java.awt.image.RenderedImage;
import java.util.ArrayList;
import com.sun.media.jai.widget.DisplayJAI;

public class DisplayJAIWithAnnotations extends DisplayJAI
  {
  protected ArrayList annotations; // List of annotations that will be
                                   // (non-interactively) drawn.

```

```
13    // Constructor for the class.
14    public DisplayJAIWithAnnotations(RenderedImage image)
15      {
16      super(image); // calls the constructor for DisplayJAI
17      annotations = new ArrayList(); // List that will held the drawings.
18      }
19
20    // This method paints the component and all its annotations.
21    public void paint(Graphics g)
22      {
23      super.paint(g);
24      Graphics2D g2d = (Graphics2D)g;
25      for (int a=0;a<annotations.size();a++) // For each annotation.
26        {
27        DrawableAnnotation element = (DrawableAnnotation)annotations.get(a);
28        element.paint(g2d);
29        }
30      }
31
32    // Add an annotation (instance of any class that inherits from
33    // DrawableAnnotation to the list of annotations which will be drawn.
34    public void addAnnotation(DrawableAnnotation a)
35      {
36      annotations.add(a);
37      }
38    }
```

Finally, a Java application which uses the `DisplayJAIWithAnnotations` class is shown in listing 22. That application creates three instances of `DiamondAnnotation` and adds them to an instance of `DisplayJAIWithAnnotations`, which will be painted inside a `JFrame`.

Listing 22: Application which uses the `DisplayJAIWithAnnotations` component.

```
1  package sibgrapi.tutorial;
2
3  import java.awt.Color;
4  import java.awt.geom.Point2D;
5  import javax.media.jai.*;
6  import javax.swing.*;
7
8  public class DisplayJAIWithAnnotationsApp
9    {
10   public static void main(String[] args)
11     {
12     PlanarImage image = JAI.create("fileload","datasets/bloodimg02.jpg");
13     DisplayJAIWithAnnotations display =
14       new DisplayJAIWithAnnotations(image);
15     // Create three diamond-shaped annotations.
16     DiamondAnnotation d1 =
17       new DiamondAnnotation(new Point2D.Double(229,55),20,20,2);
18     d1.setColor(Color.BLACK);
```

```
19      DiamondAnnotation d2 =
20        new DiamondAnnotation(new Point2D.Double(249,84),20,20,3);
21      d2.setColor(Color.BLACK);
22      DiamondAnnotation d3 =
23        new DiamondAnnotation(new Point2D.Double(303,33),35,35,5);
24      d3.setColor(Color.GRAY);
25      // Add the annotations to the instance of DisplayJAIWithAnnotations.
26      display.addAnnotation(d1);
27      display.addAnnotation(d2);
28      display.addAnnotation(d3);
29      // Create a new Frame and set the DisplayJAIWithAnnotations.
30      JFrame frame = new JFrame();
31      frame.setTitle("Annotations over an image");
32      frame.getContentPane().add(new JScrollPane(display));
33      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
34      frame.setSize(500,200); // Set the frame size.
35      frame.setVisible(true);
36      }
37    }
```

A screenshot of the `DisplayJAIWithAnnotationsApp` application (listing 22) is shown in figure 5.

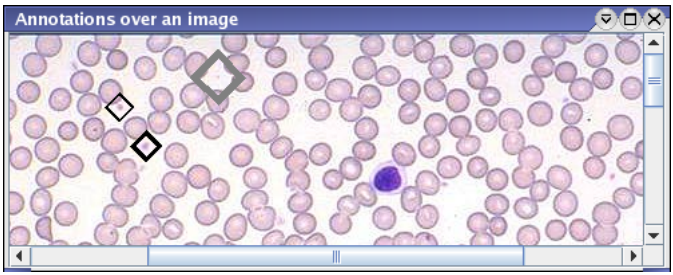

Figure 5. Screenshot of the `DisplayJAIWithAnnotationsApp` application (listing 22).

6 Complete Example: A Fuzzy C-Means Implementation

As a final example on this tutorial, let's see a complete implementation of the Fuzzy C-Means (FCM) clustering algorithm [7]. This algorithm iteractively cluster an image using fuzzy membership values instead of assigning each pixel to one and only one cluster.

The implementation is divided in two classes, one class which encapsulates the algorithm and which can perform the FCM in an image with any number of bands and another class which is an application that will call the methods in the first class.

The first class is shown in listing 23. This class has a constructor to get information about the image and set the several data structure required for the execution of the FCM algorithm, a `run` method which executes the algorithm itself (calling several private helper methods) and a `getRankedImage` method which get a ranked clustering result. The use of the rank concept allows the user to get the best clustering result, i.e. in which each pixel is assigned to the cluster in which it had the largest membership function (using $rank = 0$), or the second best, i.e. in which each pixel is assigned to the cluster in which it had the second largest membership function (using $rank = 1$) and so on – this allows the exploration of clustering alternatives for cluster analysis.

Listing 23: The `FuzzyCMeansClusteringTask` class.

```
1  package sibgrapi.tutorial;
2
3  import java.awt.*;
4  import java.awt.image.*;
5  import javax.media.jai.*;
6
7  public class FuzzyCMeansClusteringTask
8    {
9    private PlanarImage pInput; // A copy of the input image.
10   private int width,height,numBands; // The input image dimensions.
11   private int maxIterations,numClusters; // Some clustering parameters.
12   // FCM additional parameters and membership function values.
13   private float fuzziness; // "m"
14   private float[][][] membership; // Membership for pixels and clusters.
15   private int iteration; // Iteration counter (global).
16   private double j = Float.MAX_VALUE; // A metric of clustering "quality".
17   private double epsilon; // The minimum change between iterations.
18   private float[][] clusterCenters; // Cluster centers.
19   private int[] inputData; // All the input data (pixels).
20   private float[] aPixel; // A single pixel.
21   private short[][] outputData; // Output data (cluster indexes).
22
23   // Constructor for the class, which sets some algorithm parameters.
24   public FuzzyCMeansClusteringTask(PlanarImage pInput,
25                                    int numClusters,int maxIterations,
26                                    float fuzziness,double epsilon)
27     {
28     this.pInput = pInput;
29     width = pInput.getWidth(); // Get the image dimensions.
30     height = pInput.getHeight();
31     numBands = pInput.getSampleModel().getNumBands();
32     this.numClusters = numClusters; // Set some clustering parameters.
33     this.maxIterations = maxIterations;
34     this.fuzziness = fuzziness;
35     this.epsilon = epsilon;
36     iteration = 0;
37     // Allocate memory for the data arrays.
38     clusterCenters = new float[numClusters][numBands]; // Cluster centers.
39     membership = new float[width][height][numClusters]; // Memberships.
```

```
    aPixel = new float[numBands]; // A single pixel.
    outputData = new short[width][height]; // Cluster indexes.
    // Gets the raster and all pixel values for the input image.
    Raster raster = pInput.getData();
    inputData = new int[width*height*numBands];
    raster.getPixels(0,0,width,height,inputData);
    // Fill the membership function (MF) table with random values. The sum
    // of memberships for a pixel will be 1, and no MF values will be zero.
    for(int h=0;h<height;h++)
      for(int w=0;w<width;w++)
        {
        float sum = 0f;
        for(int c=0;c<numClusters;c++)
          {
          membership[w][h][c] = 0.01f+(float)Math.random();
          sum += membership[w][h][c];
          }
        for(int c=0;c<numClusters;c++) membership[w][h][c] /= sum;
        }
    }

  // This method performs the bulk of the processing. It runs the classic
  // Fuzzy C-Means clustering algorithm:
  //   1 - Calculate the cluster centers.
  //   2 - Update the membership function.
  //   3 - Calculate statistics and repeat from 1 if needed.
  public void run()
    {
    double lastJ; // The last "j" value (objective function).
    lastJ = calculateObjectiveFunction(); // Calculate objective function.
    // Do all required iterations (until the clustering converges)
    for(iteration=0;iteration<maxIterations;iteration++)
      {
      calculateClusterCentersFromMFs(); // Calculate cluster centers.
      calculateMFsFromClusterCenters(); // Calculate MFs.
      j = calculateObjectiveFunction(); // Recalculate J.
      if (Math.abs(lastJ-j) < epsilon) break; // Is it small enough?
      lastJ = j;
      }
    }

  // Calculates the cluster centers from the membership functions.
  private void calculateClusterCentersFromMFs()
    {
    float top,bottom; // Parts of the equation.
    // For each band and cluster...
    for(int b=0;b<numBands;b++)
      for(int c=0;c<numClusters;c++)
        {
        // For all data points calculate top and bottom parts of equation.
        top = bottom = 0;
        for(int h=0;h<height;h++)
          for(int w=0;w<width;w++)
            {
            int index = (h*width+w)*numBands;
            top += Math.pow(membership[w][h][c],fuzziness)*
```

```
                    inputData[index+b];
              bottom += Math.pow(membership[w][h][c],fuzziness);
              }
          clusterCenters[c][b] = top/bottom; // Calculate the cluster center.
          }
      }

    // Calculates the membership functions from the cluster centers.
    private void calculateMFsFromClusterCenters()
      {
      float sumTerms;
      // For each cluster and data point...
      for(int c=0;c<numClusters;c++)
        for(int h=0;h<height;h++)
          for(int w=0;w<width;w++)
            {
            // Get a pixel (as a single array).
            int index = (h*width+w)*numBands;
            for(int b=0;b<numBands;b++) aPixel[b] = inputData[index+b];
            // Distance of this data point to the cluster being read.
            float top = calcDistance(aPixel,clusterCenters[c]);
            // Sum of distances from this data point to all clusters.
            sumTerms = 0f;
            for(int ck=0;ck<numClusters;ck++)
              {
              float thisDistance = calcDistance(aPixel,clusterCenters[ck]);
              sumTerms += Math.pow(top/thisDistance,(2f/(fuzziness-1f)));
              }
            // Then the MF can be calculated as...
            membership[w][h][c] = (float)(1f/sumTerms);
            }
      }

    // Calculates the objective function ("j") (quality of the clustering).
    private double calculateObjectiveFunction()
      {
      double j = 0;
      // For all data values and clusters...
      for(int h=0;h<height;h++)
        for(int w=0;w<width;w++)
          for(int c=0;c<numClusters;c++)
            {
            // Get the current pixel data.
            int index = (h*width+w)*numBands;
            for(int b=0;b<numBands;b++) aPixel[b] = inputData[index+b];
            // Calculate the distance between a pixel and a cluster center.
            float distancePixelToCluster =
              calcDistance(aPixel,clusterCenters[c]);
            j += distancePixelToCluster*
                Math.pow(membership[w][h][c],fuzziness);
            }
      return j;
      }

    // Calculates the Euclidean distance between two N-dimensional vectors.
    private float calcDistance(float[] a1,float[] a2)
```

```
    {
    float distance = 0f;
    for(int e=0;e<a1.length;e++) distance += (a1[e]-a2[e])*(a1[e]-a2[e]);
    return (float)Math.sqrt(distance);
    }

  // This method will return a rank image, i.e. an image which pixels are
  // the cluster centers of the Nth best choice for the classification.
  public TiledImage getRankedImage(int rank)
    {
    // Create a SampleModel with the same dimensions as the input image.
    SampleModel sampleModel =
      RasterFactory.createBandedSampleModel(DataBuffer.TYPE_INT,
                                            width,height,numBands);
    // Create a WritableRaster using that sample model.
    WritableRaster raster =
      RasterFactory.createWritableRaster(sampleModel,new Point(0,0));
    // A pixel array will contain all bands for a specific x,y.
    int[] pixelArray = new int[numBands];
    // For all pixels in the image...
    for(int h=0;h<height;h++)
      for(int w=0;w<width;w++)
        {
        // Get the class (cluster center) with the specified rank.
        int aCluster = getRankedIndex(membership[w][h],rank);
        // Fill the array with that cluster center.
        for(int band=0;band<numBands;band++)
          pixelArray[band] = (int)clusterCenters[aCluster][band];
        raster.setPixel(w,h,pixelArray); // Put it on the raster.
        }
    TiledImage pOutput = new TiledImage(pInput,1,1); // Create an image.
    pOutput.setData(raster); // Set the raster on the output image.
    return pOutput;
    }

  // This method returns the ranked index of a cluster from an array
  // containing the membership functions.
  private int getRankedIndex(float[] data,int rank)
    {
    int[] indexes = new int[data.length]; // Temporary arrays for the
    float[] tempData = new float[data.length]; // indexes and data.
    for(int i=0;i<indexes.length;i++) // Fill those arrays.
      {
      indexes[i] = i; tempData[i] = data[i];
      }
    // Sort both arrays together, using data as the sorting key.
    for(int i=0;i<indexes.length-1;i++)
      for(int j=i;j<indexes.length;j++)
        {
        if (tempData[i] < tempData[j])
          {
          int tempI= indexes[i];
          indexes[i] = indexes[j];
          indexes[j] = tempI;
          float tempD = tempData[i];
          tempData[i] = tempData[j];
```

```
208             tempData[j] = tempD;
209             }
210           }
211       return indexes[rank]; // Return the cluster index for the desired rank.
212       }
213     }
```

Listing 24 shows the `FuzzyCMeansClusteringTaskApp` class, an application which shows how the clustering methods on class `FuzzyCMeansClusteringTask` can be used to cluster an image which file name is passed as an argument to the application.

Listing 24: The `FuzzyCMeansClusteringTaskApp` class.

```
1  package sibgrapi.tutorial;
2
3  import javax.media.jai.*;
4
5  public class SimpleFuzzyCMeansClusteringTaskApp
6    {
7    public static void main(String[] args)
8      {
9      // Check command line arguments.
10     if (args.length != 6)
11       {
12       System.err.println("Usage: java algorithms.FuzzyCMeans."+
13                          "SimpeFuzzyCMeansClusteringTaskApp "+
14                          "inputImage outputImage numberOfClusters "+
15                          "maxIterations fuzziness epsilon");
16       System.exit(0);
17       }
18     // Load the input image.
19     PlanarImage inputImage = JAI.create("fileload", args[0]);
20     // Create the task.
21     FuzzyCMeansClusteringTask task =
22       new FuzzyCMeansClusteringTask(inputImage,
23                                     Integer.parseInt(args[2]),
24                                     Integer.parseInt(args[3]),
25                                     Float.parseFloat(args[4]),
26                                     Float.parseFloat(args[5]));
27     // Run it.
28     task.run();
29     // Get the resulting image (best assignment result).
30     PlanarImage outputImage = task.getRankedImage(0);
31     // Save the image on a file.
32     JAI.create("filestore",outputImage,args[1],"TIFF");
33     }
34   }
```

7 Conclusions and further information

Several examples of the Java Advanced Imaging API were shown in this tutorial. The author expects that the example, while simple and short, were enough to give the readers an idea of the workings of the JAI API and serve as basis for the development of more complex classes and applications.

Several listings shown in this tutorial are simplified versions of code found in [5], with comments removed in order to save space.

References

[1] Sun Microsystems, Java home page, `http://java.sun.com` (last visited in July 2004).

[2] Sun Microsystems, JAI (Java Advanced Imaging) home page, `http://java.sun.com/products/java-media/jai/index.jsp` (last visited in July 2004).

[3] Rodrigues, L.H. *Building Imaging Applications with Java Technology*, Addison-Wesley, 2001.

[4] Sun Microsystems, JAI (Java Advanced Imaging) Frequently Asked Questions, `http://java.sun.com/products/java-media/jai/forDevelopers/jaifaq.html` (last visited in July 2004).

[5] Santos, R. *JAI Stuff (Tutorial):* `https://jaistuff.dev.java.net` (last visited in July 2004).

[6] Sun Microsystems, JAI (Java Advanced Imaging) API (Application Programming Interface) document home page, `http://java.sun.com/products/java-media/jai/forDevelopers/jai-apidocs/index.html` (last visited in July 2004).

[7] Bezdek, J.C. *Pattern Recognition with Fuzzy Objective Function Algorithms*, Plenum Press, 1981.