

Image and Video Processing Libraries in Ptolemy II

by James Yeh
Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, in partial satisfaction of the requirements
for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Edward Lee
Research Advisor

* * * * *

Avideh Zahkor
Second Reader

Published as:
Memorandum No. UCB/ERL M03/52
16 December 2003
Electronics Research Laboratory
College of Engineering
University of California, Berkeley 94720

Contents

| | | |
|----------|---------------------------------------|-----------|
| 1 | Introduction | 11 |
| 1.1 | Ptolemy II | 11 |
| 1.2 | JAI and JMF Integration | 12 |
| 2 | Image Processing Platform | 14 |
| 2.1 | Images in JAI | 14 |
| 2.1.1 | Origin | 14 |
| 2.1.2 | Data Types | 14 |
| 2.2 | JAIImageToken | 15 |
| 2.3 | Arithmetic Functions | 15 |
| 2.3.1 | Addition and Subtraction | 15 |
| 2.3.2 | Multiplication and Division | 16 |
| 2.4 | Non JAI actors | 16 |
| 3 | JAI Actor Library | 18 |
| 3.1 | File I/O | 18 |
| 3.1.1 | JAIImageReader | 18 |
| 3.1.2 | JAIConstant | 23 |
| 3.1.3 | File Writing | 23 |
| 3.2 | Image Manipulation | 26 |
| 3.2.1 | JAIAffineTransform | 26 |
| 3.2.2 | JAIBorder | 28 |
| 3.2.3 | JAICrop | 30 |
| 3.2.4 | JAIInvert | 30 |
| 3.2.5 | JAIPeriodicShift | 30 |
| 3.2.6 | JAIRotate | 31 |
| 3.2.7 | JAIScale | 31 |
| 3.2.8 | JAITranslate | 33 |
| 3.2.9 | JAITranspose | 33 |
| 3.3 | Band Manipulation | 33 |
| 3.3.1 | JAIBandSelect | 34 |
| 3.3.2 | JAIBandCombine | 34 |
| 3.4 | Filtering | 34 |
| 3.4.1 | JAIConvolve | 36 |

| | | |
|----------|---|-----------|
| 3.4.2 | JAIBoxFilter | 36 |
| 3.4.3 | JAIMedianFilter | 37 |
| 3.4.4 | JAIEdgeDetection | 38 |
| 3.5 | Frequency Domain Processing | 38 |
| 3.5.1 | JAIDFT | 39 |
| 3.5.2 | JAIDFT | 40 |
| 3.5.3 | JAIMagnitude, JAIPhase | 41 |
| 3.5.4 | JAIPolarToComplex | 41 |
| 3.5.5 | JAIDCT | 41 |
| 3.5.6 | JAIDCT | 41 |
| 3.6 | Data Type Manipulation | 42 |
| 3.7 | Dynamic Range Modification | 42 |
| 3.8 | Extension to Ptolemy II | 42 |
| 3.8.1 | ImageToJAI | 42 |
| 3.8.2 | JAIToDoubleMatrix | 42 |
| 3.8.3 | DoubleMatrixToJAI | 43 |
| 4 | Image Processing Examples | 44 |
| 4.1 | A More Complex Example: Image Reconstruction from Phase Information | 45 |
| 4.2 | Custom Actor Writing | 45 |
| 4.2.1 | Salt and Pepper Noise Modeling | 45 |
| 4.2.2 | Adaptive Median Filtering | 45 |
| 5 | Video Processing Platform | 55 |
| 5.1 | Buffer | 55 |
| 5.2 | JMFImageToken | 56 |
| 6 | JMF Actor Library | 57 |
| 6.1 | Buffer Based Actors | 57 |
| 6.1.1 | MovieReader | 57 |
| 6.1.2 | MovieWriter | 58 |
| 6.1.3 | VideoCamera | 58 |
| 6.1.4 | ColorFinder | 59 |
| 6.2 | Streaming Actors | 59 |
| 6.2.1 | StreamLoader | 59 |
| 6.2.2 | AudioPlayer | 59 |
| 6.2.3 | VideoPlayer | 60 |
| 7 | Video Processing Examples | 61 |
| 7.1 | Playing a Movie | 61 |
| 7.2 | Item Tracking | 61 |
| 7.3 | Putting it all Together, JMF and JAI | 62 |
| 8 | Future Work | 63 |

List of Figures

| | | |
|------|---|----|
| 1.1 | Screenshot of Vergil, the user interface for Ptolemy II | 12 |
| 2.1 | Non JAI Image Processing actors. | 17 |
| 3.1 | List of JAIActors | 19 |
| 3.2 | A sample PBM file | 21 |
| 3.3 | A sample PGM file. | 22 |
| 3.4 | A sample PPM file. | 22 |
| 3.5 | An image about to undergo an affine transformation. | 27 |
| 3.6 | An affine transformation using an affine matrix of [0.707107, 0.707107, 0.0; -0.707107, 0.707107, 0.0]. Using this matrix results in a 45 degree counterclockwise rotation. | 27 |
| 3.7 | BandSelect-BandRecombine example. | 35 |
| 3.8 | Median Filtering Mask Shapes | 37 |
| 3.9 | Frei and Chen Horizontal and Vertical Masks | 38 |
| 3.10 | Prewitt Horizontal and Vertical Masks | 38 |
| 3.11 | Roberts Cross Edge Masks | 39 |
| 3.12 | Sobel Horizontal and Vertical Masks | 39 |
| 3.13 | Diagonal and Backdiagonal Masks | 40 |
| 4.1 | A simple model to load and display an image. | 44 |
| 4.2 | A simple model showing the need for the JAIDataConvert actor. | 45 |
| 4.3 | A model to reconstruct an image from its phase. | 46 |
| 4.4 | The image we will take our phase information from. | 46 |
| 4.5 | The initial guess. | 47 |
| 4.6 | The image after one iteration. | 47 |
| 4.7 | The image after five iterations. | 48 |
| 4.8 | The image after ten iterations. | 48 |
| 4.9 | The image after fifteen iterations. | 49 |
| 4.10 | Sample code implementing Salt and Pepper noise. | 49 |
| 4.11 | An image about to be corrupted with noise. | 50 |
| 4.12 | The image after being corrupted with salt and pepper noise with $p = 0.1$ | 51 |
| 4.13 | A model that adds salt and pepper noise to an image, performs adaptive median filtering, and saves to a PNG file. | 52 |

| | |
|--|----|
| 4.14 Median Filtering on a Salt and Pepper Noised Image. | 53 |
| 4.15 Adaptive Median Filtering on a Salt and Pepper Noised Image . | 54 |

Acknowledgements

The following project report is the culmination of 6 years at U.C. Berkeley, 4 years as an undergraduate, and 2 years as a graduate student. So kudos to me!

This report would not be possible without Professor Edward Lee and the rest of the Ptolemy Group. I cannot imagine a better research group to work for. They have put up with me through the tough times, and for that, I am eternally grateful.

4 years ago, I joined my first research group as an undergraduate, and I owe a great thanks to Professor John Wawrzynek here at U.C. Berkeley, and Professor André DeHon at the California Institute of Technology. Without them, I would not be writing the report.

Thanks goes out to James “Madness Gears” Byun, Eylon Caspi, Chris Chang, Michael Chu, Ken “Uncanny X-” Chen, Elaine Cheong, Allie Fletcher, Professor Vivek Goyal at MIT (hehehe), Ben Green, Tom “Monkey Magic” Greene, Randy Huang, Jörn Janneck, Yury Markovskiy, Eugene “The Fast And The Fourier” Kim, Kelvin “Merton Hanks” Lwin, Trevor “Coded” Meyerowitz, Gabe Moy, Doug “RAW” Patterson, and Nick Weaver.

To Rory, Todd, and the rest of the folks at Comic Relief on University Avenue, you have given me a paradise away from my hectic academic life.

A “can’t leave it out cause I’d look like a horrible son” thank you to my family.

And a special thanks to Yolanda Hong. You are the light that guided me through my darkest times.

James Yeh uses Fender, Godin, and Seagull Guitars, D’addario Strings, Johnson and Crate Amplifiers, Boss effects pedals, Jim Dunlop Tortex Fin Picks of 1 mm in thickness.

“All this machinery, making modern music, can still be open-hearted. Not so fully-charted it’s really just a question of your honesty.” - Rush, The Spirit of Radio

“Berkeley is so much a part of who I am that I find it almost impossible to reflect rationally on what it “meant” to me, what I “got out of it,” what it “did” for me.

... What Berkeley had offered me was for all practical purposes infinite: why had my ability to accept it been so finite? Why did I still have so many questions? Why did I have so few answers? Would I not be a more finished person

had I been provided a chart, a map, a design for living?

I believe so.

I also believe that the world I know, given such a chart, would have been narrowed, constricted, diminished: a more ordered and less risky world but not the world I wanted, not free, not Berkeley, not me.” - Joan Didion

Chapter 1

Introduction

1.1 Ptolemy II

Ptolemy II [3] is open-source software, currently under development at U.C. Berkeley, used to model systems, in particular, embedded systems. These complex systems mix a variety of different operations, including feedback control, sequential decision making, and signal processing. Ptolemy II is written in Java, and runs on a variety of platforms.

Ptolemy II specifically studies how different components interact with each other. Each component could represent a myriad of things, from threads to hardware. Components are connected by relations, and information is passed through tokens and into ports. How these components and relations interact with each other is determined by a director which explicitly states how these components (called actors in Ptolemy II) should be scheduled for execution.

The signal processing done in Ptolemy II has mainly been focused on one-dimensional signal processing. The goal of this project was to be able to introduce both two-dimensional signal processing (in the form of images), and three dimensional signal processing (in the form of video) components into Ptolemy II.

A decision had to be made on whether or not to use packages that are outside of the Java Development Kit (JDK). The advantage of not using outside packages is simply to save the user from having to install extra libraries. However, given that there was precedent in other projects, for instance, Chamberlain Fong's [4] use of Java 3D for the GR Director, the decision was made to use external libraries (A huge advantage of using these libraries is the ability to develop a more complete library of actors).

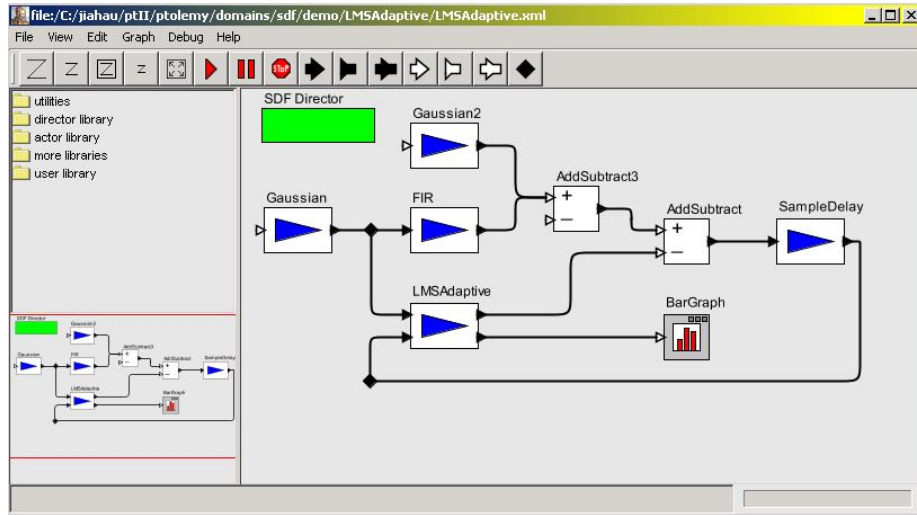


Figure 1.1: A screen shot of Vergil, the user interface for Ptolemy II.

Java Advanced Imaging (JAI)

The Java Advanced Imaging (JAI) API [2] was created by Sun Microsystems to work with their Java Programming Language. It is available for Windows, Solaris, and Linux. Sun's goal for this API is to provide a framework that allows many image processing algorithms to be implemented with little or no custom code writing.

Java Media Framework (JMF)

Similarly, the Java Media Framework (JMF) [1] was developed by Sun Microsystems and runs on Windows, Solaris, and Linux. It was created to allow programmers to more easily work with time-based media (more specifically, audio and video) in Java.

1.2 JAI and JMF Integration

Ptolemy II has many different token types to be used for transport of data. This includes an `ObjectToken` used to transport any `Object` in Java.

Originally, `ObjectTokens` were used to encapsulate the data structures used in both JAI and JMF. However, when it came to displaying images or movies, a separate display actor had to be written for `ObjectTokens` containing different types of data structures.

We decided that there was a need for an ImageToken class. The goal of the ImageToken was to be able to have a Token type that would be able to work not only with the previously written image processing actors (based on the Advanced Windowing Toolkit in the standard JDK), but also to work with the newly written actors in both the JAI and JMF actor libraries. Luckily, both API's provide conversions to and from the Image class located in the Advanced Windowing Toolkit (AWT). This allows us to use actors based on JAI on images originated from a JMF-based actor.

The ImageToken class is an abstract class that any token containing an image data structure should implement; the only requirement is implementing the asAWTImage() method.

Chapter 2

Image Processing Platform

2.1 Images in JAI

The `RenderedOp` is the data structure from JAI that is used in this platform. Images inside this data structure are not rendered until they need to be (for instance if the width or height of the image is required, or if the images themselves need to be rendered to display).

Images are a collection of points called pixels, and typically are thought of as having a band for each salient feature. For instance in an RGB (Red, Green, Blue) image, all of the red pixels form a band, all of the green pixels form a band, and all of the blue ones form a band.

However, bands do not necessarily contain intensity data. When a Discrete Fourier Transform (see section 3.5.1 on page 39) is performed on an image, the image will be represented as frequency data. This data, mathematically, is typically complex, so a color in the spatial domain will be represented as two bands in the frequency domain (one for each part of a complex number).

2.1.1 Origin

The origin of an image is typically the top left corner. Almost all the actors preserve the location of the origin. The origin can be moved however, by using the `JAITranslate` (see section 3.2.8 on page 33) actor. This does not change how the image is displayed or saved, but it will effect the output of some actors that involve two or more images (for instance, when two images are added together, only the intersection of the two images are used).

2.1.2 Data Types

When stored or loaded, image data is in one of the many non-floating point data types. However what goes on within a model may change the internal data type

of the bands. The values in an image can take on the following types: byte, short, unsigned short (ushort for short), int, float, and double.

Certain JAI based actors (such as the JAIDFT, see section 3.5.1 on pg. 39) produce an image with double as its internal data type. These images contain data that use the full range of doubles, which does not correspond with traditional double representation of images (MATLAB uses the range 0.0 to 1.0).

The platform allows for conversion to double matrices. The user can choose among the two different representations (see section 3.8.2 on pg. 42).

2.2 JAIIImageToken

The JAIIImageToken encapsulates a RenderedOp in a Token. When they are passed from actor to actor, the RenderedOps contained within are not rendered. They only become rendered when information about the image is needed (whether it is the width, height, or the image itself).

Because it is a subclass of the abstract ImageToken class, it contains the method asAWTImage(). When asAWTImage() is called, getRendering() is called on the RenderedOp. This returns a PlanarImage. After that, getAsBufferedImage() is called on this PlanarImage, and a BufferedImage is returned, which is a subclass of AWTImage.

2.3 Arithmetic Functions

Arithmetic functions in Ptolemy II are executed with the Add/Subtract and Multiply/Divide actors.

Currently JAIIImageTokens must be operated on with other JAIIImageTokens in these two actors. Conversions exist to convert arrays of values (JAICoordinate, pg. 23), matrices of values (DoubleMatrixToJAI, pg. 43), and ImageTokens (ImageToJAI, pg. 42) to JAIIImageTokens.

Images do not need to have the same height or width to undergo these operations, nor do they need to contain the same number of bands. The number of bands after one of the four operations is executed is the smallest number of bands in either of the two sources. The height and width of the image is determined by the intersection of the two images.

2.3.1 Addition and Subtraction

In addition and subtraction, the resulting data type is the smallest data type with sufficient range to support the ranges of the input data types (which isn't necessarily the range of the sum).

2.3.2 Multiplication and Division

In multiplication and division, the resulting data type is the smallest data type with sufficient range to support the ranges of the input data types. Note that multiplication and division are done bandwise, therefore results may not be as expected if the images contain complex data. To circumvent this, the magnitude and phases of the images can be taken, manipulated, and reassembled using the JAIPolarToComplex actor.

2.4 Non JAI actors

The following actors were previously implemented by various members of the Ptolemy Group. Before, they used ObjectTokens containing AWTImages. They have since been modified to use AWTImageToken, a subclass of ImageToken.

| | |
|---------------|--|
| MonitorImage | Displays images on the workspace by making the image at the input its icon. |
| ImageDisplay | Used to display images in a separate window. This actor and the MonitorImage actor are the only actors that can currently display ImageTokens. |
| ImageReader | Reads in a file or URL and outputs an AWTImageToken. |
| ImageRotate | Rotates an AWTImageToken by a certain amount of degrees. |
| ImageToString | Takes an AWTImageToken at the input and outputs information about the image as a StringToken. |
| URLToImage | Reads a URL and outputs an AWTImageToken. This actor has since been made obsolete by the ImageReader actor. |

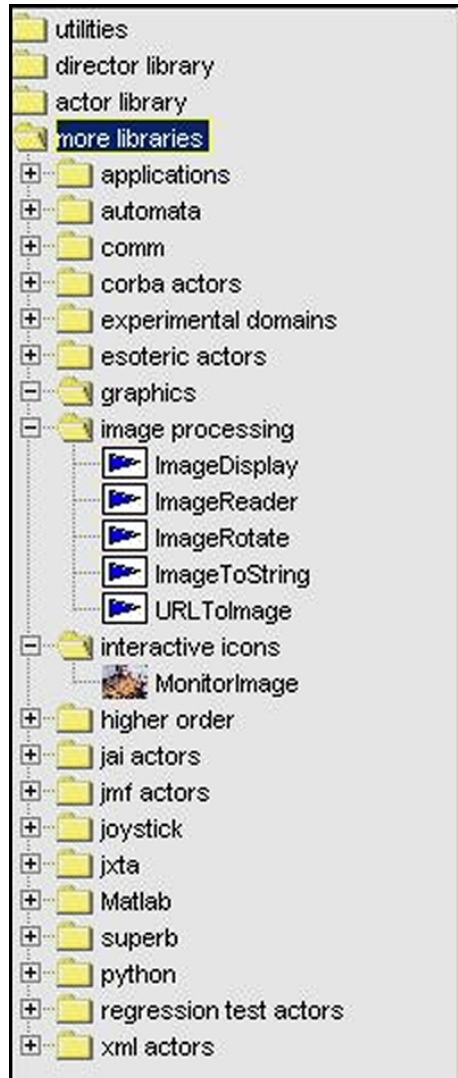


Figure 2.1: Non JAI Image Processing actors.

Chapter 3

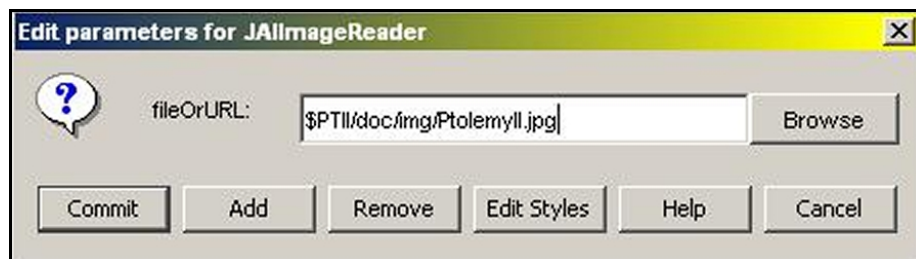
JAI Actor Library

The following actors are all dependent on the JAI API, which is why the actor names are all prefixed.

3.1 File I/O

One of the main reasons JAI was chosen to implement the image processing platform was the ease that images could be loaded and saved in a variety of different file types.

3.1.1 JAIImageReader



All file loading, regardless of file type, in the platform is done in the JAIImageReader actor. It has one parameter, FileOrURL, that points to an appropriate image file. The following file types are supported.

Bitmap (BMP)

The Bitmap file format was developed by Microsoft. It is commonly used on their operating systems (for example, the desktop background). Version 2 and Version 3 bitmaps can be read with no problems. Version 4 bitmaps can be specified with an alpha value (which controls transparency) and a gamma value

which allows intensity correction for different types of display devices; however they are not handled by JAI.

FlashPix (FPX)

The FlashPix file format was developed by Eastman Kodak. It separates images into 64 by 64 blocks known as tiles. These tiles can be uncompressed, JPEG compressed, or single color compressed.

Graphics Interchange Format (GIF)

GIF was developed by Compuserve. It is supported by virtually every platform. It is limited 256 colors, and of recent note, many patent issues have arisen with this file format. Because of this, GIFs can be read in this platform, but cannot be written.

Joint Photographic Experts Group (JPEG)

The JPEG standard was developed by the Joint Photographic Experts Group. JPEG images can be either grayscale or color, and they come in varying resolutions and sizes.

Portable Network Graphics (PNG)

PNG was developed as a patent-free alternative to GIF. It is supported by UNIX, Macintosh, and Windows systems, and provides a lossless and compressed storage of images. PNG is different than the other file formats, in that it can include an alpha channel, used to control transparency. This alpha channel is traditionally the last band in an image (in a grayscale image, it will be the second band, in a RGB image, it will be the fourth band). PNG also includes a gamma value, used to adjust an image for different display devices.

Portable Bitmap, Graymap, Pixmap (PBM, PGM, PPM)

PBM (black and white images), PGM (grayscale images), and PPM (color images), are all types of the Portable Anymap format. They were designed to be simple file formats that would be easily read by different machines. Figures 3.2 to 3.4 show examples of these file types.

Tag Image File Format (TIFF)

TIFF is a common file format compatible with UNIX, Macintosh, and Windows systems. It is frequently the file format of choice when acquiring images from devices like scanners and video frame capture devices due to the fact that it was specifically designed for large arrays of raster data.

```
P1
#comments
10 10
1001001110
1001000100
1001000100
1001000100
1111000100
1001000100
1001000100
1001000100
1001000100
1001001110
```

Figure 3.2: A sample PBM file. The P1, a.k.a. the magic number, specifies that it is a PBM ASCII file. To specify the file with RAW data, the magic number would be P4. Characters following a pound sign are ignored. The width of the image is then specified with ASCII characters as a decimal number. This is followed by a whitespace (blanks, tabs, carriage returns, line feeds). Then the height is specified with ASCII characters as a decimal number. Then width times height bits should follow, with 1 being black, and 0 being white. No line should be longer than 70 characters. Whitespaces are ignored when specifying the bits.

```

P2
#comments
10 10
15
5 0 0 5 0 0 10 10 10 0
5 0 0 5 0 0 0 10 0 0
5 0 0 5 0 0 0 10 0 0
5 0 0 5 0 0 0 10 0 0
5 5 5 5 0 0 0 10 0 0
5 0 0 5 0 0 0 10 0 0
5 0 0 5 0 0 0 10 0 0
5 0 0 5 0 0 0 10 0 0
5 0 0 5 0 0 0 10 0 0
5 0 0 5 0 0 0 10 0 0
5 0 0 5 0 0 10 10 10 0

```

Figure 3.3: A sample PGM file. The P2 specifies that it is a PGM ASCII file. To specify the file with RAW data, the magic number would be P5. Characters following a pound sign are ignored. The width of the image is then specified with ASCII characters as a decimal number. This is followed by a whitespace. Then the height is specified with ASCII characters as a decimal number. This is followed by another whitespace. Then the maximum gray value is specified, again as a ASCII decimal. Then width times height gray values are specified by ASCII decimals separated by whitespaces. No line should be longer then 70 characters.

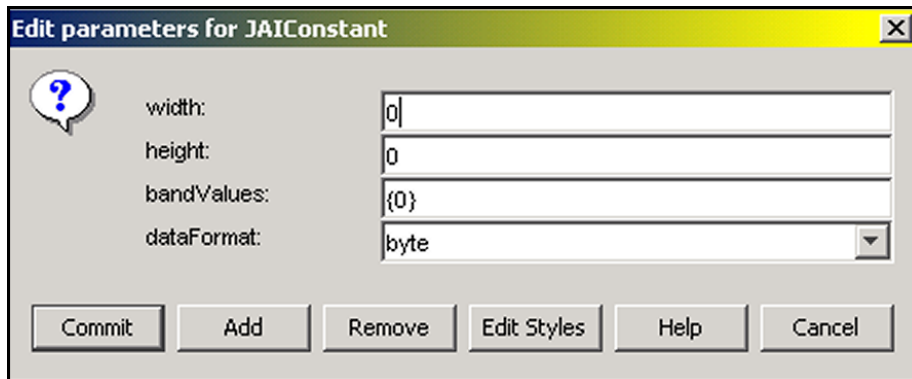
```

P3
4 4
15
0 0 0 0 0 0 0 0 0 15 0 15
0 0 0 0 15 7 0 0 0 0 0 0
0 0 0 0 0 0 0 15 7 0 0 0
15 0 15 0 0 0 0 0 0 0 0 0

```

Figure 3.4: A sample PPM file. The P3 specifies that it is a PPM ASCII file. To specify the file with RAW data, the magic number would be P6. Character following a pound sign are ignored. The width and height of the image are specified like a PGM file, however the value following these integers is the maximum color value. The three values for each pixel represent red, green, and blue respectively.

3.1.2 JAIConstant



The dialog box titled "Edit parameters for JAIConstant" features a yellow title bar with a close button. On the left, there is a blue question mark icon. The main area contains four labeled text fields: "width:" with the value "0", "height:" with the value "0", "bandValues:" with the value "{0}", and "dataFormat:" with a dropdown menu set to "byte". At the bottom, there are six buttons: "Commit", "Add", "Remove", "Edit Styles", "Help", and "Cancel".

The JAIConstant actor can be used to create an image with constant intensity. The number of bands is the length of the array, and the value of each band is the value at the index of the array.

The following data types are supported by this actor: byte, double, float, int, and short. The range of the band values corresponds to the range of the data type.

3.1.3 File Writing

There are separate file writers for each file type because of the different nuances that image file types have. All file writers share two common parameters. The first is a FileOrURL parameter that specifies the location and name of the file to be written.

The second is a confirmOverwrite parameter. If this is marked true, then Ptolemy II will issue a window if the file already exists and will prompt the user whether to overwrite the file or not. If the file doesn't exist, then the file is written without any prompts.

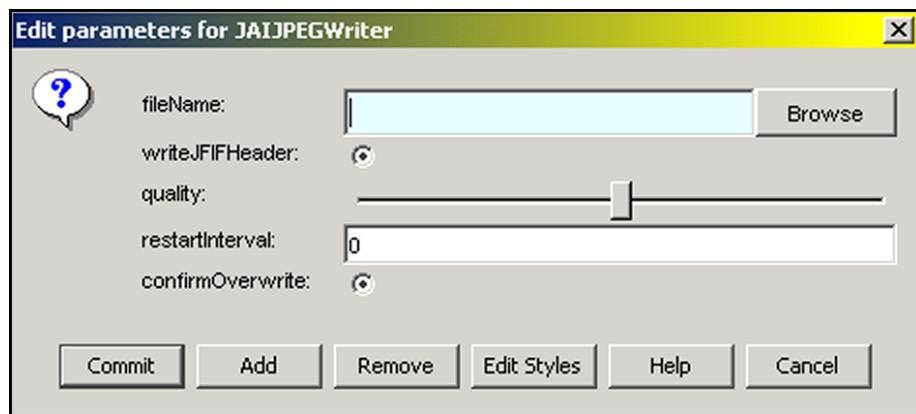
JAIBMPWriter



The dialog box titled "Edit parameters for JAIBMPWriter" has a yellow title bar with a close button. It includes a blue question mark icon on the left. The parameters are: "fileName:" with an empty text field and a "Browse" button; "storeTopDown:" with an unselected radio button; and "confirmOverwrite:" with a selected radio button. The bottom row contains six buttons: "Commit", "Add", "Remove", "Edit Styles", "Help", and "Cancel".

The JAIBMPWriter actor allows Version 3 BMP's to be written. It includes a storeTopDown parameter, which by default is false. If marked true, the file will be stored in reverse order, and the image will be seen flipped upside down using any standard file viewer.

JAIJPEGWriter



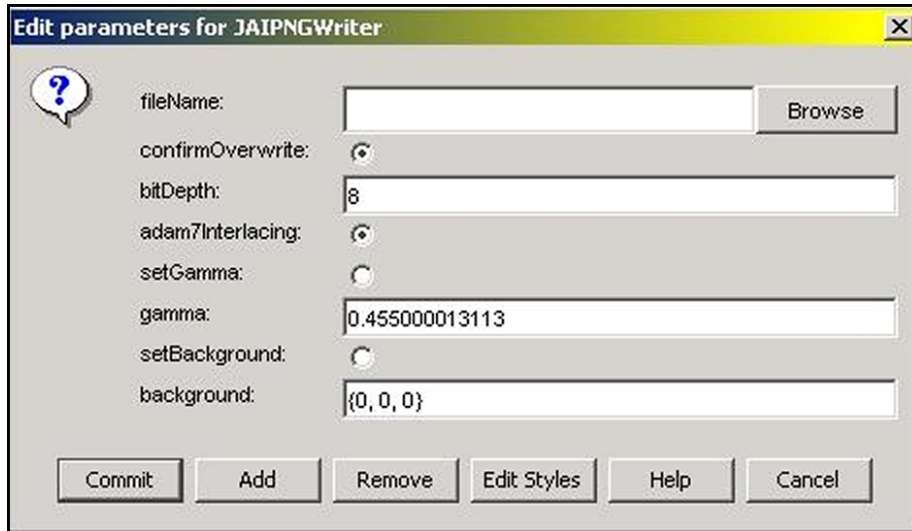
The JAIJPEGWriter writes JPEG files. It is the only file writer where the user can control the quality of the image.

The writeJFIFHeader parameter allows the user to choose whether or not to write the JFIF Header in their JPEG file. By default, this is true. The header includes data such as the version number, horizontal and vertical pixel density, and pixel aspect ratio.

The quality slider bar parameter allows the user to choose the amount of compression to use. Going from left to right increases the quality of the image and decreases the amount of compression used.

The restartInterval parameter is the number of JPEG Minimum Coded Units (MCU) per interval. JPEG images can use these restart markers to periodically delineate image segments to limit the effect of bitstream errors to a single interval. The default value is zero (no restart interval markers).

JAIPNGWriter



The JAIPNGWriter differs from all the other file writers in that it can write more than just single banded grayscale images and three banded RGB images.

If the JAIPNGWriter reads in a two banded image, it will assume that the first band is grayscale data, and the second is an alpha channel. If it reads in a four banded image, it will assume the first 3 bands are RGB, and the fourth to be an alpha channel. The alpha channel controls translucency. A value of 0 denotes complete translucency, whereas a value of $2^{\text{bitdepth}} - 1$, where *bitdepth* is the number of bits used per sample, denotes complete opacity.

The *bitDepth* allows the user to choose how many bits are used for each sample. While the default value is 8, grayscale images can have bitdepths of 1, 2, 4, 8, and 16. RGB images can have bitdepths of 8 and 16 (per color, which means 24 and 48 bits per pixel).

If the Adam7 Interlacing option is turned off (the default), pixels are stored left to right and from top to bottom. If it is turned on, seven distinct passes are made over the image, each transmitting a subset of the pixels.

The *gamma* value changes the intensity of how the image is displayed according to the following equation.

$$\text{sample} = \text{lightout}^{\text{gamma}}$$

The default value of *gamma* is 1/2.2.

If the image uses an alpha channel, then the background color of the image is pertinent, and can be changed. For a grayscale image, only the first value

in the array is used. The value can range from 0 (for black), to $2^{\text{bitdepth}} - 1$ (for white). For RGB images, all three values are used, with each value ranging from 0 to $2^{\text{bitdepth}} - 1$ as well.

JAITIFFWriter



The JAITIFFWriter has one extra parameter, writeTiled. By default, this is marked false. If marked true, the image data will be written in tiles instead of strips. For larger images, this makes data access more efficient.

3.2 Image Manipulation

The following actors manipulate images in the spatial domain.

3.2.1 JAIAffineTransform



An affine transformation preserves the straightness of straight lines, and the parallelism of parallel lines, But the distance between parallel lines may not be preserved.

The mapping between the destination pixel (x, y) and the source position (x', y') is given by:



Figure 3.5: An image about to undergo an affine transformation.



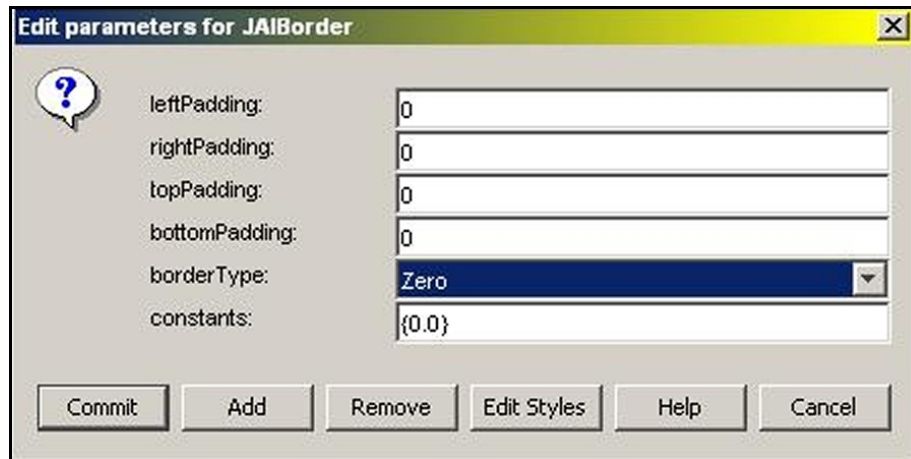
Figure 3.6: An affine transformation using an affine matrix of $[0.707107, 0.707107, 0.0; -0.707107, 0.707107, 0.0]$. Using this matrix results in a 45 degree counterclockwise rotation.

$$\begin{aligned} x' &= m_{00} * x + m_{01} * y + m_{02} \\ y' &= m_{10} * x + m_{11} * y + m_{12} \end{aligned}$$

The source pixel may not exist in the original image, and interpolation must be done. Interpolation is explained in the JAIScale actor (section 3.2.7 on page 31).

The default value for the affineMatrix parameter corresponds to an identity operation; the output is the same as the input. See figures 3.5 and 3.6 for more examples.

3.2.2 JAIBorder



Operations occasionally will shrink an image (for example, median filtering). This is sometimes undesirable. To prevent shrinkage, an image can be bordered before undergoing such an operation.

The JAIBorder Actor offers several different border types:

Constant

| | | | | | | |
|---|---|----------|----------|----------|---|---|
| c | c | c | c | c | c | c |
| c | c | c | c | c | c | c |
| c | c | 1 | 2 | 3 | c | c |
| c | c | 4 | 5 | 6 | c | c |
| c | c | 7 | 8 | 9 | c | c |
| c | c | c | c | c | c | c |
| c | c | c | c | c | c | c |

The border is filled with a supplied constant. If there are not enough constants for each band, then the first constant will be used to fill the border of each band.

Copy

| | | | | | | |
|---|---|----------|----------|----------|---|---|
| 1 | 1 | 1 | 2 | 3 | 3 | 3 |
| 1 | 1 | 1 | 2 | 3 | 3 | 3 |
| 1 | 1 | 1 | 2 | 3 | 3 | 3 |
| 4 | 4 | 4 | 5 | 6 | 6 | 6 |
| 7 | 7 | 7 | 8 | 9 | 9 | 9 |
| 7 | 7 | 7 | 8 | 9 | 9 | 9 |
| 7 | 7 | 7 | 8 | 9 | 9 | 9 |

The border pixels are copies of the edge pixels.

Reflect

| | | | | | | |
|---|---|----------|----------|----------|---|---|
| 5 | 4 | 4 | 5 | 6 | 6 | 5 |
| 2 | 1 | 1 | 2 | 3 | 3 | 2 |
| 2 | 1 | 1 | 2 | 3 | 3 | 2 |
| 5 | 4 | 4 | 5 | 6 | 6 | 5 |
| 8 | 7 | 7 | 8 | 9 | 9 | 8 |
| 8 | 7 | 7 | 8 | 9 | 9 | 8 |
| 5 | 4 | 4 | 5 | 6 | 6 | 5 |

The border pixels are filled with mirrored copies of an image. This border is useful when you want to avoid a discontinuity between the image and the border.

Wrap

| | | | | | | |
|---|---|----------|----------|----------|---|---|
| 5 | 6 | 4 | 5 | 6 | 4 | 5 |
| 8 | 9 | 7 | 8 | 9 | 7 | 8 |
| 2 | 3 | 1 | 2 | 3 | 1 | 2 |
| 5 | 6 | 4 | 5 | 6 | 4 | 5 |
| 8 | 9 | 7 | 8 | 9 | 7 | 8 |
| 2 | 3 | 1 | 2 | 3 | 1 | 2 |
| 5 | 6 | 4 | 5 | 6 | 4 | 5 |

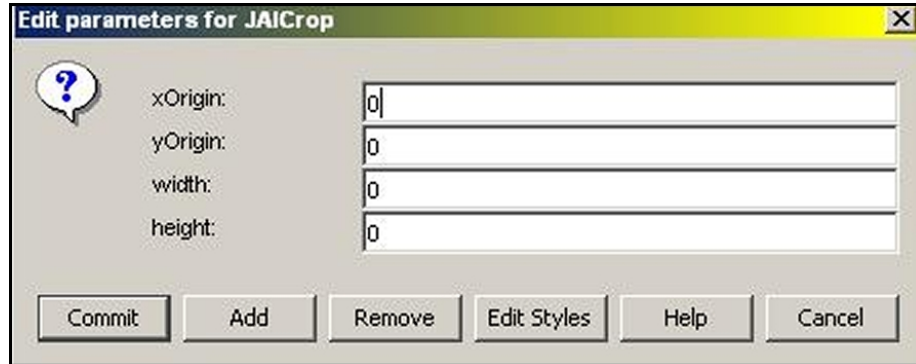
The border pixels are filled with copies of the image. This border is particularly useful for images that are periodic in nature, e.g. the Fourier Transform of an image.

Zero

| | | | | | | |
|---|---|----------|----------|----------|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 2 | 3 | 0 | 0 |
| 0 | 0 | 4 | 5 | 6 | 0 | 0 |
| 0 | 0 | 7 | 8 | 9 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The border is filled with zeroes in every band.

3.2.3 JAICrop

A dialog box titled "Edit parameters for JAICrop" with a yellow header bar and a close button (X) in the top right corner. On the left, there is a blue question mark icon in a speech bubble. To its right are four labels: "xOrigin:", "yOrigin:", "width:", and "height:". Each label is followed by a text input field, all of which contain the number "0". At the bottom of the dialog, there are six buttons: "Commit", "Add", "Remove", "Edit Styles", "Help", and "Cancel".

Operations will occasionally enlarge the dimensions of an image. When this is undesirable, the JAICrop actor can be used to select a rectangular region of interest that is useful. Other times, only a portion of the image is interesting in which case, this actor may be used to restrict it to a smaller region.

3.2.4 JAIInvert

The JAIInvert actor inverts every band (subtracts each band from the maximum value allowed by the internal data type) on the input image. This is often useful for inverting images that are very dark and would waste toner if printed as is.

3.2.5 JAIPeriodicShift

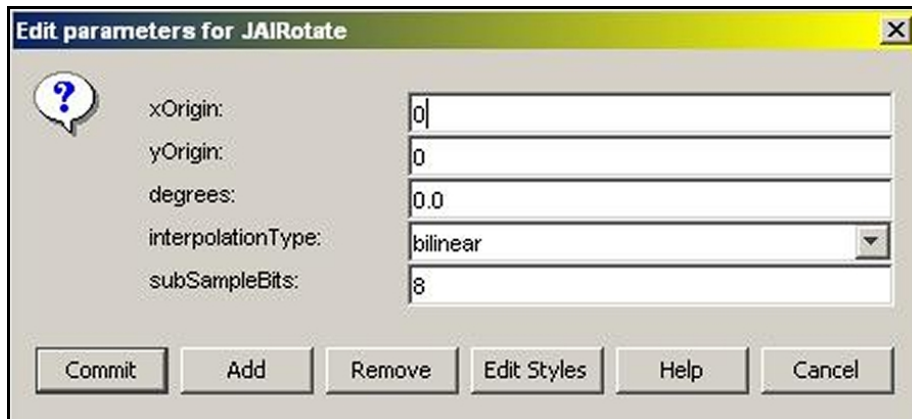
A dialog box titled "Edit parameters for JAIPeriodicShift" with a yellow header bar and a close button (X) in the top right corner. On the left, there is a blue question mark icon in a speech bubble. To its right are two labels: "xShift:" and "yShift:". Each label is followed by a text input field, both of which contain the number "0". At the bottom of the dialog, there are six buttons: "Commit", "Add", "Remove", "Edit Styles", "Help", and "Cancel".

The JAIPeriodicShift actor takes an image and extends it periodically in all four directions (with the horizontal and vertical periods being the width and height respectively). It then shifts the image. The xShift parameter indicates the amount to shift in the horizontal direction. A positive integer corresponds to a shift to the right. A negative integer corresponds to a shift to the left. A value of zero corresponds to no shift at all in the horizontal direction.

The yShift parameter indicates the amount to shift in the vertical direction. A positive integer corresponds to a shift downwards. A negative integer corre-

sponds to a shift upwards. A value of zero corresponds to no shift at all in the vertical direction. The image is then clipped to the bound of the input.

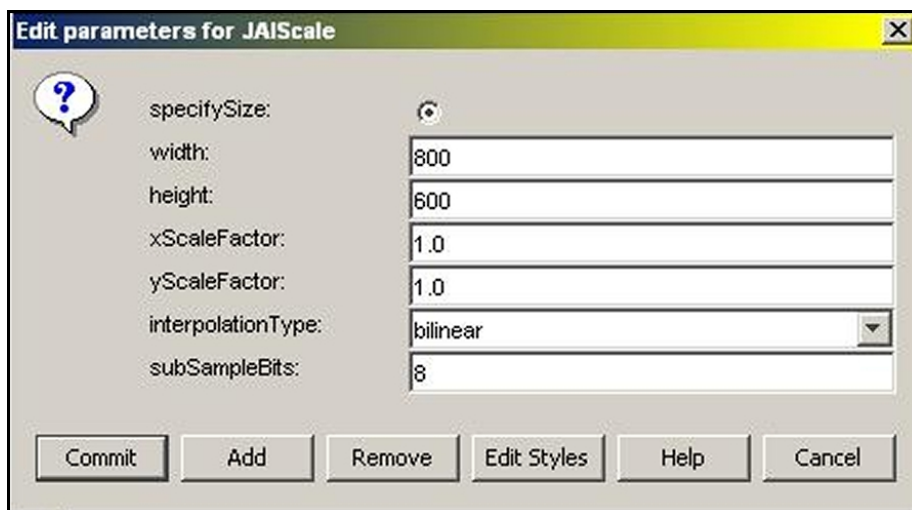
3.2.6 JAIRotate



The dialog box titled "Edit parameters for JAIRotate" features a yellow title bar with a close button. On the left, there is a help icon (a question mark in a speech bubble). The main area contains five labeled text input fields: "xOrigin:" with the value "0", "yOrigin:" with the value "0", "degrees:" with the value "0.0", "interpolationType:" with a dropdown menu showing "bilinear", and "subSampleBits:" with the value "8". At the bottom, there are six buttons: "Commit", "Add", "Remove", "Edit Styles", "Help", and "Cancel".

The JAIRotate actor rotates an image in the clockwise direction. The amount it is rotated is specified by the degrees parameter. Unless the rotation is by a multiple of 90 degrees (in which case, the JAITranspose actor should be used), interpolation must be done to figure out points not specified in the original image. The interpolation types are explained in the JAIScale actor (see next section).

3.2.7 JAIScale



The dialog box titled "Edit parameters for JAIScale" features a yellow title bar with a close button. On the left, there is a help icon (a question mark in a speech bubble). The main area contains six labeled controls: "specifySize:" with a radio button icon, "width:" with the value "800", "height:" with the value "600", "xScaleFactor:" with the value "1.0", "yScaleFactor:" with the value "1.0", "interpolationType:" with a dropdown menu showing "bilinear", and "subSampleBits:" with the value "8". At the bottom, there are six buttons: "Commit", "Add", "Remove", "Edit Styles", "Help", and "Cancel".

The JAIScale actor can be used to shrink and enlarge images. The main problem in doing this is how to determine the values of the points in the new image.

For example, an image that is scaled by 1.7 in the x direction, and 1.4 in the y direction, will have values at (0,0), (1.4,0), (0,1.7), (1.4,1.7), (2.8,1.7), (1.4,3.4), (2.8,3.4). However, we need points at (0,0), (1,0), (0,1), (1,1), etc.

The JAIScale actor provides four different interpolation methods to figure out the values of the points. All but nearest neighbor interpolation makes use of the SubSampleBits Parameter which indicates how many bits to use to estimate points.

Nearest Neighbor

Nearest Neighbor interpolation simply takes the value of the point that is nearest. For example, the value of (1,0) in the new image corresponds to a point (5/7, 0) in the old image, and thus would correspond to (1, 0) in the old image. While this method is computationally inexpensive, it tends to leave artifacts.

Bilinear

Bilinear interpolation takes the four closest points in the source image and computes the value of the destination pixel. If the subsample value is given by (u, v) , and the 4 points (going left to right, top to bottom) are $p_{00}, p_{01}, p_{10}, p_{11}$, the output is defined to be:

$$(1 - v) * [(1 - u) * p_{00} + u * p_{01}] + v * [(1 - u) * p_{10} + u * p_{11}]$$

While the results are usually better than nearest neighbor interpolation, artifacts may still remain.

Bicubic

Bicubic interpolation takes the 16 closes points and uses a bicubic waveforms instead of the linear waveforms used in bilinear interpolation. It uses the following function to interpolate points.

$$\begin{aligned} r(x) &= \frac{3}{2}|x|^3 - \frac{5}{2}|x|^2 + 1, & 0 \leq x < 1 \\ r(x) &= \frac{-1}{2}|x|^3 + \frac{5}{2}|x|^2 - 4|x| + 2, & 1 \leq x < 2 \\ r(x) &= 0, & otherwise \end{aligned}$$

Bicubic2

Bicubic2 interpolation uses a different polynomial.

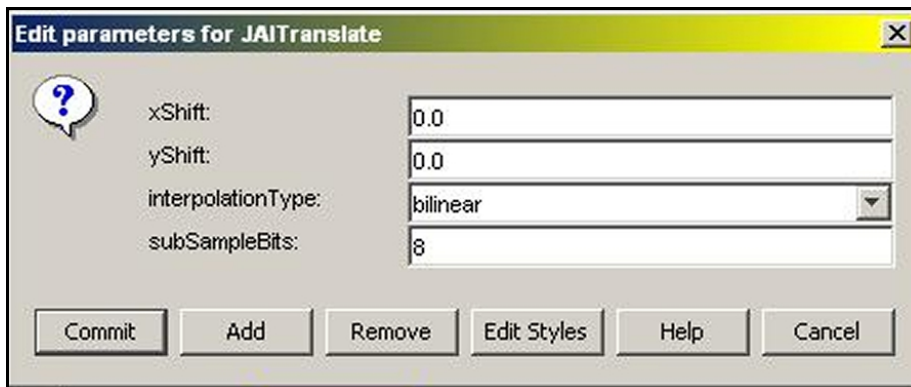
$$\begin{aligned} r(x) &= |x|^3 - 2|x|^2 + 1, & 0 \leq x < 1 \\ r(x) &= -|x|^3 + 5|x|^2 - 8|x| + 4, & 1 \leq x < 2 \\ r(x) &= 0, & otherwise \end{aligned}$$

The Bicubic and Bicubic2 interpolations tend to perform better than the bilinear interpolation at the expense of extra computation time. In fact, most of the

images in this report were scaled up using this option.

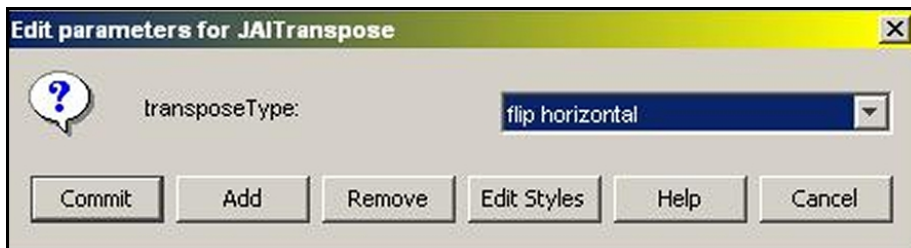
This actor also allows the user to specify the exact size of the image. To do this, the `specifySize` parameter must be checked. If it is checked, the actor will scale the image horizontally and vertically to the pixel lengths in the width and height parameters respectively.

3.2.8 JAIShift



The JAIShift actor changes the origin of an image. This does not change how an image is displayed or saved. It does effect operations that involve 2 or more images (for instance the addition operation).

3.2.9 JAIShuffle



The JAIShuffle actor provides seven different transpositions, three different rotations (90, 180, and 270 degrees), and flipping over the (horizontal, vertical, diagonal, anti-diagonal) axes.

3.3 Band Manipulation

Often it is necessary to be able to operate on separate bands independently. The following two actors can be used to select bands, as well as combine bands

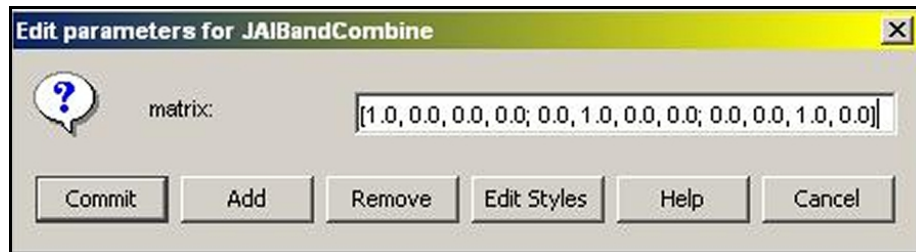
into one image.

3.3.1 JAIBandSelect



The JAIBandSelect actor takes one array of integers as a parameter. The number of bands at the output is equal to the length of the specified array. Each integer in the array specifies the band to select at the input. This actor is particularly useful in selecting a single band, or isolating image data from its alpha channel.

3.3.2 JAIBandCombine



The JAIBandCombine actor takes a m by $n + 1$ matrix of floats as an input, where m is the number of bands at the output, and n is the number of bands at the input. The first n values of each row specify the scalar amount to multiply each band by. The last value is a DC offset. In conjunction with the Add/Subtract actor, this actor can be used to combine several images into one.

3.4 Filtering

Filtering allows us to deal with how the real world interferes with signals. Among the many operations we can perform on signals (in this case images) are blur, sharpen, and denoise.

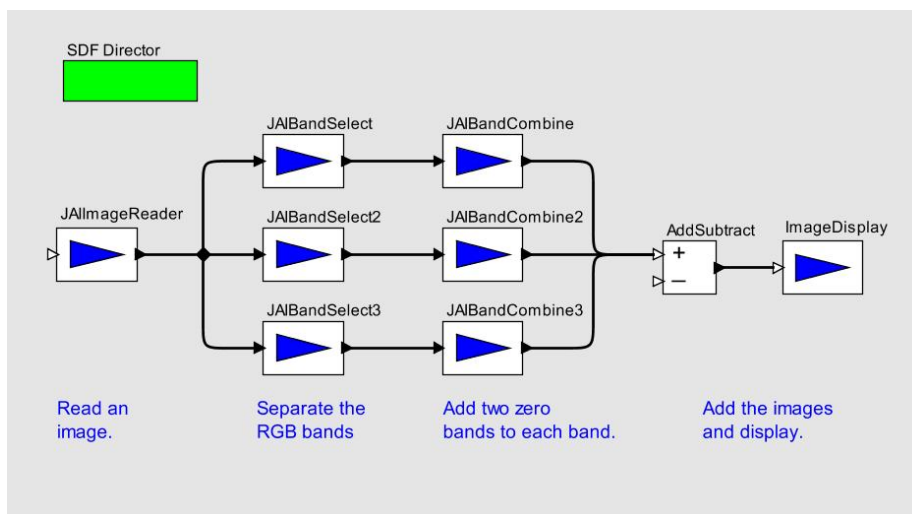
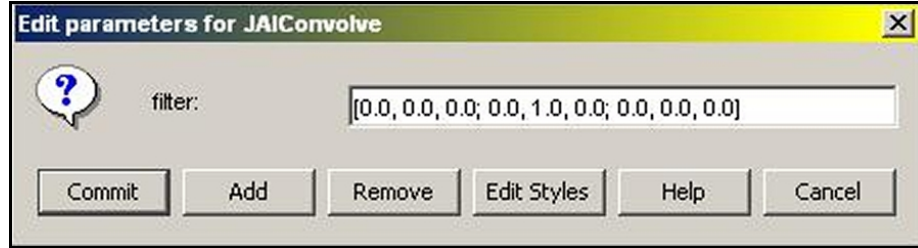


Figure 3.7: An example of how to separate and recombine bands. The JAIBand-Select actors are selecting one single band of the RGB image. The JAIBand-Combine actors are adding bands back onto these single bands. Two bands of zeros are being appended to the red band, a band of zeros are being attached to each side of the green band, and two bands of zeros are being put in front of the blue band. The Add/Subtract actor adds these three and outputs the original image.

3.4.1 JAIconvolve

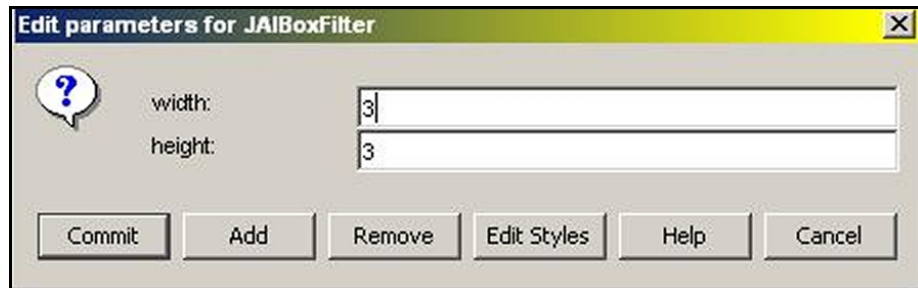


Linear Shift Invariant (LSI) systems can be characterized by an impulse response $h(n_1, n_2)$ [5]. For a two dimensional discrete input $x(n_1, n_2)$, the output $y(n_1, n_2)$ is:

$$\begin{aligned} y(n_1, n_2) &= x(n_1, n_2) * h(n_1, n_2) \\ &= \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} x(k_1, k_2) h(n_1 - k_1, n_2 - k_2) \end{aligned}$$

The filter matrix is $h(n_1, n_2)$. The top left entry of the matrix is $h(0, 0)$, the bottom right entry is $h(m, n)$ where m is the number of rows and n is the number of columns. Note that while this actor takes a matrix of doubles as a parameter, the data type of the image is preserved from input to output.

3.4.2 JAIBoxFilter



The JAIBoxFilter actor is a convenience actor that implements box filtering. A box filter is defined to be a m by n rectangular filter with the value at each tap defined to be $1/mn$. This filter is convolved with an input image to produce a blurred output.

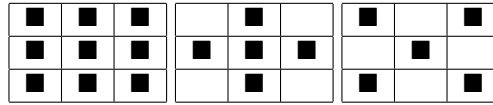


Figure 3.8: Mask shapes for median filtering. From left to right: Square, Plus, and Cross. The black boxes indicate the neighborhood.

3.4.3 JAIFilter



Various versions of median filtering are used to get rid of noise spikes (be they of low or high intensity).

Traditional median filtering is a neighborhood-based ranking filter that orders the pixels in the neighborhood according to their values. The median value of the group is taken to be the output.

Separable median filtering is slightly different. The median of each row in the neighborhood is calculated, and the output is the median of these medians.

The JAIFilter actor supports these two median filtering methods. There are three masks available for traditional median filtering, a square mask, a plus-shaped mask, and a X-shaped mask (shown in Figure 3.8). The size of these masks must be n by n where n is an odd number. The separable median filtering algorithm can be chosen by selecting “Separable Square” from the shape pull-down menu, however only 3x3 and 5x5 masks are valid.

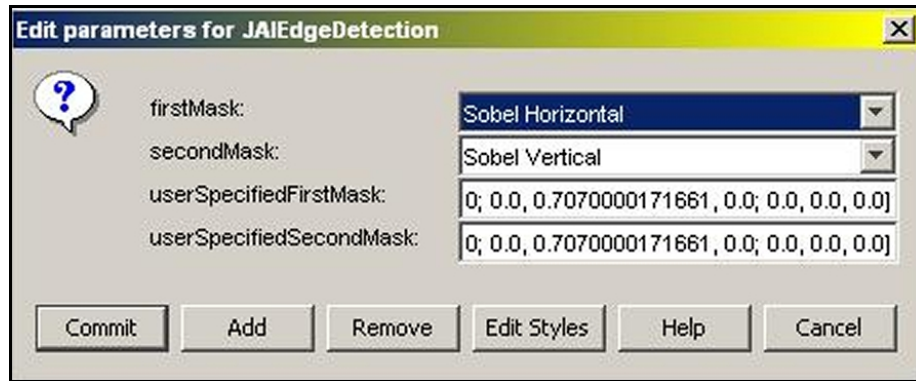
| | | | | | |
|------------|---|-------------|----|-------------|----|
| 1 | 0 | -1 | -1 | $-\sqrt{2}$ | -1 |
| $\sqrt{2}$ | 0 | $-\sqrt{2}$ | 0 | 0 | 0 |
| 1 | 0 | -1 | 1 | $\sqrt{2}$ | 1 |

Figure 3.9: Frei and Chen Horizontal and Vertical Masks

| | | | | | |
|---|---|----|----|----|----|
| 1 | 0 | -1 | -1 | -1 | -1 |
| 1 | 0 | -1 | 0 | 0 | 0 |
| 1 | 0 | -1 | 1 | 1 | 1 |

Figure 3.10: Prewitt Horizontal and Vertical Masks

3.4.4 JAIEdgeDetection



It is often useful to be able to detect edges in images. A common use of edge detection is to be able to segment an image into different objects.

The JAIEdgeDetection uses a gradient based method. The input image is filtered with two different masks. The two intermediate images are squared, added to each other, and then square rooted.

There are several built in masks, shown in figures 3.9 to 3.13. The user can choose to input a custom mask by choosing “User Specified” from either or both of the pull down menus, and inputting the appropriate values in the field(s).

3.5 Frequency Domain Processing

Traditionally we look at images as a plot of intensity over space. However we can transform these images to plots of magnitude and phase over frequency.

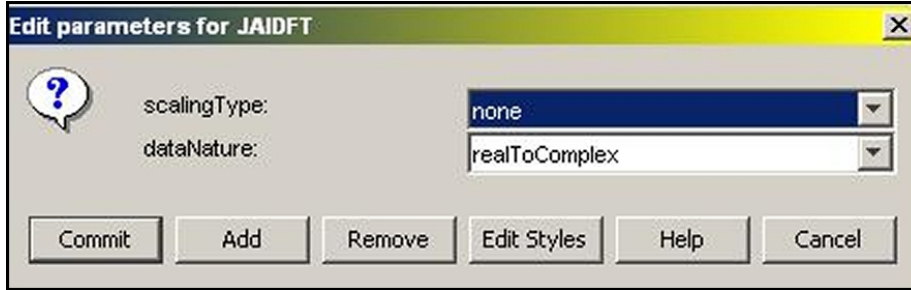
| | | | | | |
|---|---|----|----|---|---|
| 0 | 0 | -1 | -1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |

Figure 3.11: Roberts Cross Edge Masks

| | | | | | |
|---|---|----|----|----|----|
| 1 | 0 | -1 | -1 | -2 | -1 |
| 2 | 0 | -2 | 0 | 0 | 0 |
| 1 | 0 | -1 | 1 | 2 | 1 |

Figure 3.12: Sobel Horizontal and Vertical Masks

3.5.1 JAIDFT



The Discrete Fourier Transform (DFT) [5] transforms an image from the spatial domain to the frequency domain. Generally, the images contain real data. However, when it is transformed into the frequency domain, each data point is complex (unless the image is symmetric). The number of bands at the output is double the number of bands at the input, alternating between real and imaginary¹. In the case that the data is complex at the input, the `dataNature` parameter should be changed to `ComplexToComplex`. For a single banded image, the Discrete Fourier Transform is defined to be:

$$X(k_1, k_2) = \begin{cases} \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) e^{-j(2\pi/N_1)k_1 n_1} e^{-j(2\pi/N_2)k_2 n_2}, & \text{for } 0 \leq n_1 \leq N_1 - 1, \\ & 0 \leq n_2 \leq N_2 - 1 \\ 0, & \text{otherwise} \end{cases}$$

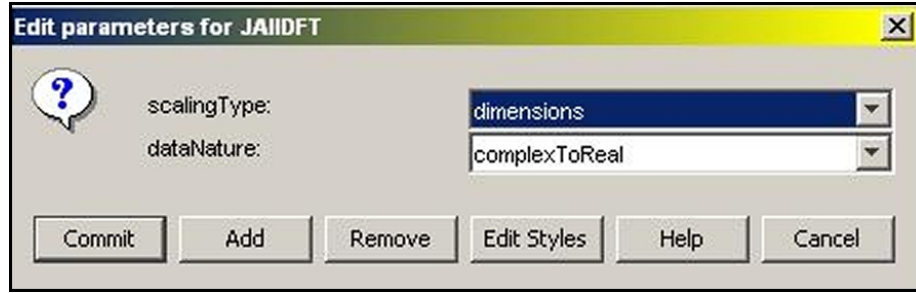
The JAIDFT actor uses a 2 dimensional Fast Fourier Transform (FFT). This corresponds to an increase in image size to the next power of two in each dimension unless that dimension is a power of two itself.

¹Note that while the number of bands could have stayed the same and the data type of the image could be a new data structure, the internal data types supported are all subclasses of the Number class. The Number class currently has no subclass that handles complex numbers.

| | | | | | |
|---|----|----|----|----|---|
| 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | -1 | -1 | 0 | 1 |
| 0 | -1 | -1 | -1 | -1 | 0 |

Figure 3.13: Diagonal and Backdiagonal Masks

3.5.2 JAIDFT



The Inverse Discrete Fourier Transform [5] transforms an image from the frequency domain to the spatial domain. Generally, images defined in the frequency domain contain complex data. The default setting is to attempt to transform this data into a real image. The JAIDFT actor can also transform the data into complex data by switching the dataNature parameter to ComplexToComplex. For a single-banded image, the Inverse Discrete Fourier Transform is defined to be:

$$x(n_1, n_2) = \begin{cases} \frac{1}{N_1 N_2} \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} X(k_1, k_2) e^{j(2\pi/N_1)k_1 n_1} e^{j(2\pi/N_2)k_2 n_2}, & \text{for } 0 \leq n_1 \leq N_1 - 1, \\ 0, & \text{otherwise} \end{cases}$$

The JAIDFT actor uses a 2 dimensional Inverse Fast Fourier Transform (IFFT). This means that each dimension is padded with zeros up to the next power of two, if it is not a power of two itself. Note that the JAIDFT actor at the output of the JAIDFT encapsulates a RenderedOp that contains an image with a data type of double. The JAIDFT preserves the data type of the input, therefore, at the end of a DFT-IDFT chain, the JAIDFT actor at the output encapsulates an image with data type double.

Currently, the only two actors to display images are the ImageDisplay and MonitorImage actors. When asAWTImage() is called on a JAIDFT actor, if the data type of the encapsulated is anything else but byte, the image will not render properly.

The JAIDFT actor can be used to convert the internal data type to byte.

3.5.3 JAIMagnitude, JAIPhase

The JAIMagnitude and JAIPhase actors take the magnitude and phase of a complex image, respectively. The input image is assumed to have an even number of bands, alternating between real and imaginary coefficients.

3.5.4 JAIPolarToComplex

The JAIPolarToComplex actor takes 2 inputs, a magnitude input, and a phase input. It converts the polar notated inputs into a complex image at the output. The two inputs must have the same number of bands. The output image will have twice as many bands as each of the inputs.

3.5.5 JAIDCT

The Discrete Cosine Transform (DCT) is the frequency transformation used most often in image coding due to its improvement in energy compaction over the DFT [5]. While the data at the output of a DFT is complex, the output of a DCT is real. The DCT is defined to be:

$$C_x(k_1, k_2) = \begin{cases} \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} 4x(n_1, n_2) \cos \frac{\pi}{2N_1} k_1(2n_1 + 1) \cos \frac{\pi}{2N_2} k_2(2n_2 + 1), \\ \text{for } 0 \leq k_1 \leq N_1 - 1, 0 \leq k_2 \leq N_2 - 1 \\ 0, \end{cases} \quad \text{otherwise}$$

Like the JAIDFT actor, the data type of the image encapsulated by the JAI-ImageToken at the output is double.

3.5.6 JAIIDCT

The JAIIDCT actor implements the Inverse Discrete Cosine Transform (IDCT) [5].

$$x(n_1, n_2) = \begin{cases} \frac{1}{N_1 N_2} \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} w_1(k_1) w_2(k_2) C_x(k_1, k_2) \cos \frac{\pi}{2N_1} k_1(2n_1 + 1) \cos \frac{\pi}{2N_2} k_2(2n_2 + 1), \\ \text{for } 0 \leq n_1 \leq N_1 - 1, 0 \leq n_2 \leq N_2 - 1 \\ 0, \text{ otherwise} \end{cases}$$

where

$$w_1(k_1) = \begin{cases} 1/2, & k_1 = 0 \\ 1, & 1 \leq k_1 \leq N_1 - 1 \end{cases}$$

$$w_2(k_2) = \begin{cases} 1/2, & k_2 = 0 \\ 1, & 1 \leq k_2 \leq N_2 - 1 \end{cases}$$

3.6 Data Type Manipulation

Certain actors, like the JAIDFT and JAIDCT, change the internal data type of the image to double. Since most actors preserve the internal data type of the image, often at the end of a graph, the data type of the image will still be of type double. The JAIDataConvert actor can be used to cast the internal data type to the following data types: byte, double, float, int, short, and ushort (unsigned short).

3.7 Dynamic Range Modification

Occasionally, an image will occur with a large dynamic range. For instance, the magnitude of a Discrete Fourier Transform may have too large a range suitable for displaying or saving. The JAILog actor takes the logarithm base 10 of each pixel value in each band.

3.8 Extension to Ptolemy II

3.8.1 ImageToJAI

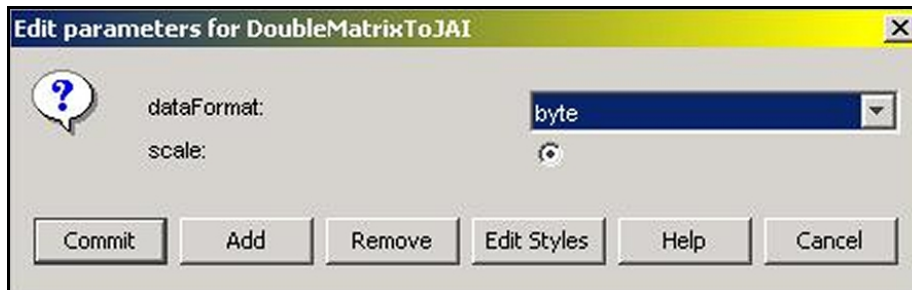
The ImageToJAI actor takes an ImageToken at the input and outputs a JAI-ImageToken at the output. This allows current and future libraries involving imaging to be able to interact with the actors that are dependant on JAI as long as a conversion to AWT imaging exists.

3.8.2 JAIDoubleMatrix



The JAIDoubleMatrix actor allows a conversion from a JAIImageToken to a DoubleMatrixToken which lets the user manipulate data at its most distilled level. It has one parameter, normalize, which allows the normalizing of data types to the range of 0.0 to 1.0 in doubles. The notion of origin is lost in this conversion, and when converted back, the origin will be set at the top left corner.

3.8.3 DoubleMatrixToJAI



The DoubleMatrixToJAI actor converts DoubleMatrixTokens back into a JAI-ImageToken. It has one parameter, scale. This option should be checked if the normalize option in the JAIDoubleMatrix actor is checked.

While the actor is a natural pairing with the previous actor to use after a custom image processing function, the reverse chain can also be used. For example, if a discrete Fourier Transform of a matrix is needed, the user can convert the matrix into an image (albeit, it might not be aesthetically pleasing), take the DFT using the JAIDFT actor, and convert back.

Chapter 4

Image Processing Examples

Hello Ptolemaeus Figure 4.1 shows a “Hello World” example. Note that the JAIImageReader (section 3.1.1 on page 18) loads a picture of Claudius Ptolemaeus by default.

DFT-IDFT Chain Figure 4.2 illustrates the need for the JAIDataConvert actor (section 3.6 on page 42). The JAIDFT actor changes the data type of the encapsulated image to double. The JAIIDFT actor preserves the data type of the input. The data type must be changed to byte so that the image can be displayed. While there was an effort made to abstract away the need to control the internal data type of the image, in some cases, such as this DFT-IDFT chain, it is necessary for the user to change it.

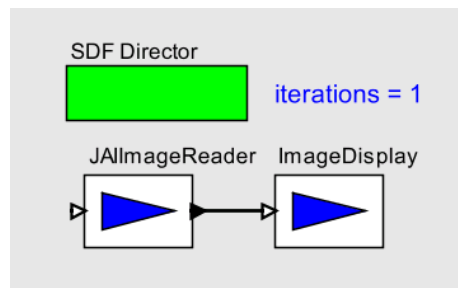


Figure 4.1: A simple model to load and display an image.

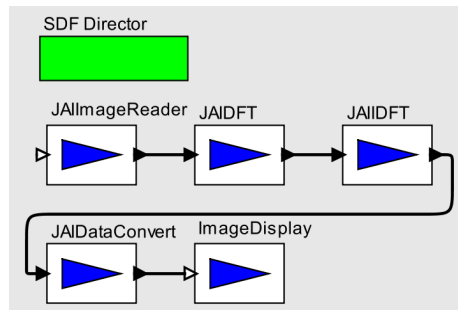


Figure 4.2: A simple model showing the need for the JAIDataConvert actor.

4.1 A More Complex Example: Image Reconstruction from Phase Information

It can be shown that an image can be perfectly reconstructed from the phase of its Fourier Transform [5]. Given finite precision computing however, perfect reconstruction is usually impossible. However an iterative technique exists that gets very close to the original image within a scaling factor (if two images differ by a scaling factor, the phase of their Fourier Transforms will be the same). See figures 4.3 to 4.9

4.2 Custom Actor Writing

While the current image processing library can handle a wide range of tasks, there is no doubt that custom actors will sometimes need to be written. While custom actors can be written using the JAI framework, it is likely the user will not want to make a committed effort into learning a brand new API. Instead, using the JAIDoubleMatrix actor (section 3.8.2 on 42), custom image processing actors can be written using double matrices as the data structure of choice.

4.2.1 Salt and Pepper Noise Modeling

Salt and Pepper Noise acts upon grayscale images. Each pixel is turned either black or white with a probability p . See figures 4.10 to 4.12.

4.2.2 Adaptive Median Filtering

Traditional median filtering can be used to get rid of salt and pepper noise, however it will leave the image blurred. An adaptive algorithm does much better. See figures 4.14 and 4.15 for images and 4.13 for the model. Code for this actor is attached as an Appendix.

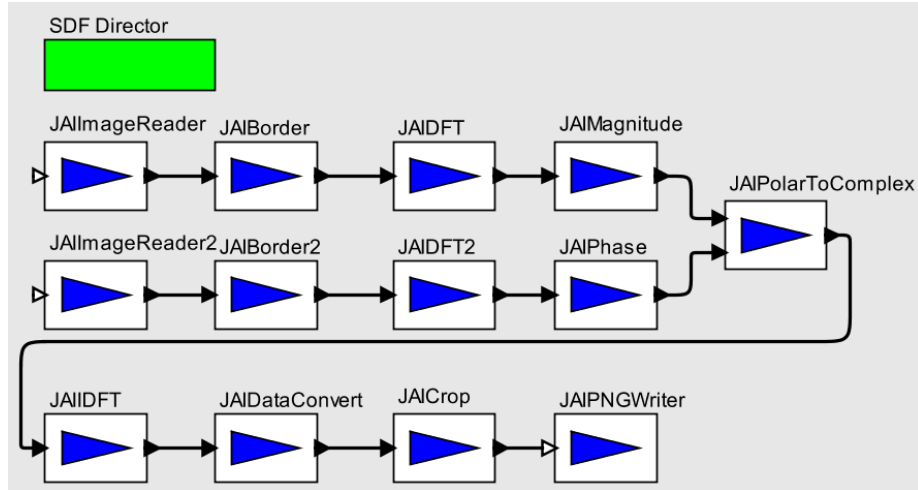


Figure 4.3: A model to reconstruct an image from its phase. The top `JAImageReader` loads our guess, while the bottom one loads the image that we would like to converge to. The `JAIBorder` actors add a zero border to each image so that a region of interest can be determined. The DFT of each image is taken, and we take the magnitude of the guess, and the phase of the image we would like to converge to. We combine these two to form one image, take the IDFT, convert the internal data to byte, and crop it to its original size. We then save our guess (which will be loaded by the the top `JAImageReader`).



Figure 4.4: The image we will take our phase information from.



Figure 4.5: The initial guess.



Figure 4.6: The image after one iteration.



Figure 4.7: The image after five iterations.



Figure 4.8: The image after ten iterations.



Figure 4.9: The image after fifteen iterations.

```

public void fire() throws IllegalArgumentException {
    super.fire();
    DoubleMatrixToken doubleMatrixToken = (DoubleMatrixToken) input.get(0);
    double data[][] = doubleMatrixToken.doubleMatrix();
    int width = doubleMatrixToken.getRowCount();
    int height = doubleMatrixToken.getColumnCount();
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            double value = Math.random();
            if (value < _probability) {
                if (value < _probability/2) {
                    data[i][j] = 0.0F;
                } else {
                    data[i][j] = 1.0F;
                }
            }
        }
    }
    output.send(0, new DoubleMatrixToken(data));
}

```

Figure 4.10: Sample code implementing Salt and Pepper noise.



Figure 4.11: An image about to be corrupted with noise.



Figure 4.12: The image after being corrupted with salt and pepper noise with $p = 0.1$.

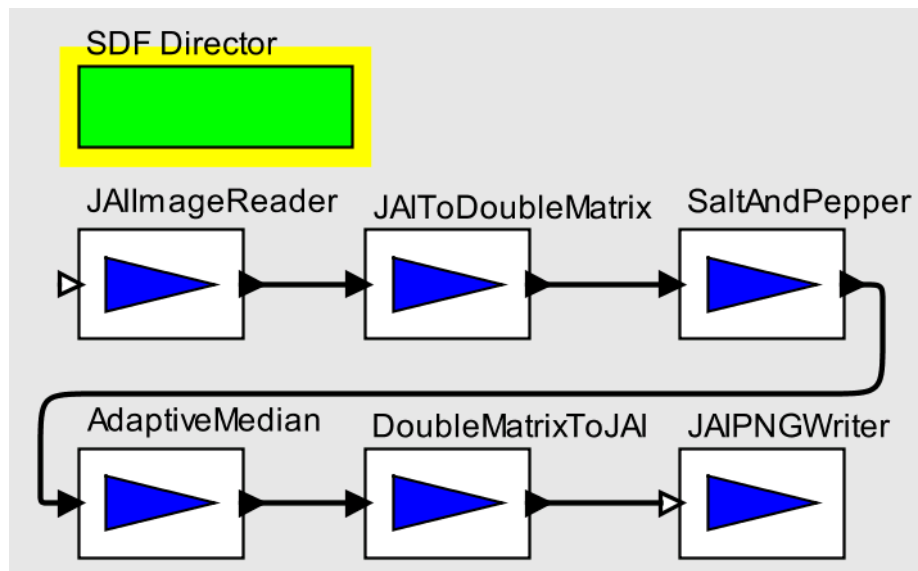


Figure 4.13: A model that adds salt and pepper noise to an image, performs adaptive median filtering, and saves to a PNG file.



Figure 4.14: Median Filtering on a Salt and Pepper Noised Image.



Figure 4.15: The noised image after using an adaptive median filtering algorithm. Note that finer details show up in this image that didn't in the previous ones, in particular, the shingles on the rooftops.

Chapter 5

Video Processing Platform

The Video Processing platform is built on top of the Java Media Framework (JMF). JMF provides two different data structures for transporting video, the Buffer and the DataSource.

A Buffer carries one frame of data (along with the format of the data), whereas a DataSource passes a reference to the entire video file.

The goal of this platform was to be able to use actors defined in this platform along with ones defined in the Image Processing platform. For example, videos with smaller resolutions generally take up less bandwidth. However, the user may want a larger video but doesn't have the money to pay for a broadband connection. A JMF actor outputting series of images could be linked to a JAI actor that scales the image up, if a conversion between Buffers and RenderedOps exists.

Luckily JMF provides a way to convert Buffers into AWT images (which then can be converted into a JAIImageToken using the ImageToJAI actor). Actors using ObjectTokens containing DataSources are discussed at the end of the next chapter.

5.1 Buffer

The Buffer is the data carrying structure in JMF. The Buffer data structure contains one frame of data, as well as the format of the data. This allows us to operate on single frames.

It supports the following formats, Cinepak, MPEG-1, H.261, H.263, JPEG, Indeo, RGB, and YUV.

5.2 JMFImageToken

The JMFImageToken encapsulates a Buffer in a Token. Because it is a subclass of the abstract ImageToken class, it contains the method `asAWTImage()`. It implements this method using the JMF class, `BufferedImage`.

Chapter 6

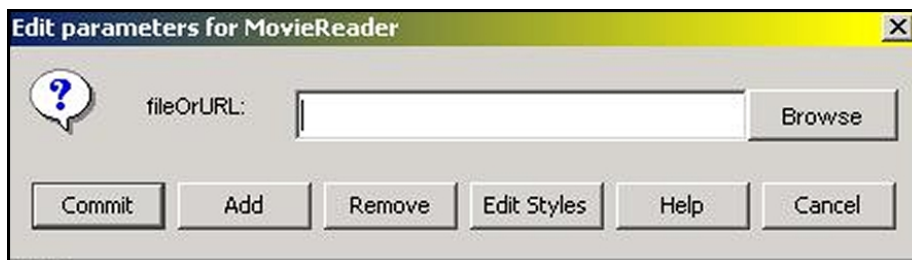
JMF Actor Library

The following actors are all dependent on JMF. There are two categories of actors, those that use Buffers, and those that use DataSources. Since Buffers are convertible to and from AWT images, they work seamlessly with the image processing framework. The DataSource actors are currently a curiosity that will be looked at in the future.

6.1 Buffer Based Actors

Buffer based actors process images individually, using JMFImageTokens as their means of transporting data.

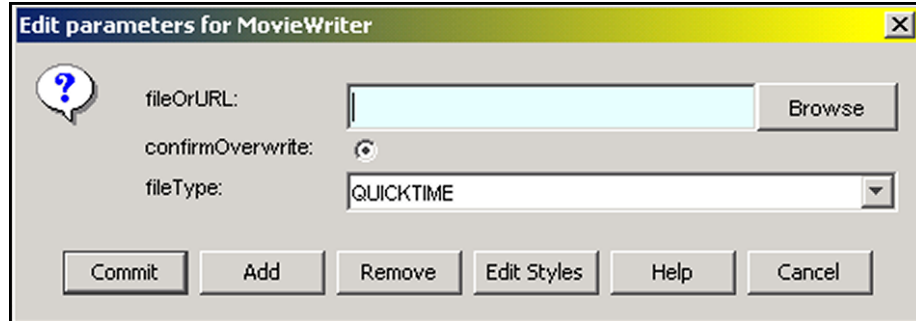
6.1.1 MovieReader



The MovieReader actor loads a video file from a file or URL. The frames are fired off in chronological order. It supports AVI (Windows Video File Format), Quicktime (Apple Computer's video standard), and MPEG (defined by the Motion Picture Experts Group).

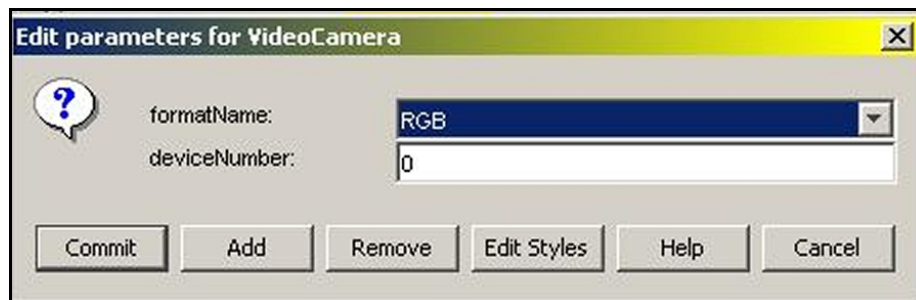
Not all AVI, Quicktime, and MPEG files can be read by this actor. All three may contain video encoded with codecs that JMF is not configured to use, and since the MovieReader outputs every frame, JMF must be able to access each frame individually.

6.1.2 MovieWriter



The MovieWriter actor accepts JMFImageTokens and writes them to the specified file (either in AVI or Quicktime format). Like the image file writers (see section 3.1.3 on pg. 23.), this actor has the parameter, confirmOverwrite.

6.1.3 VideoCamera



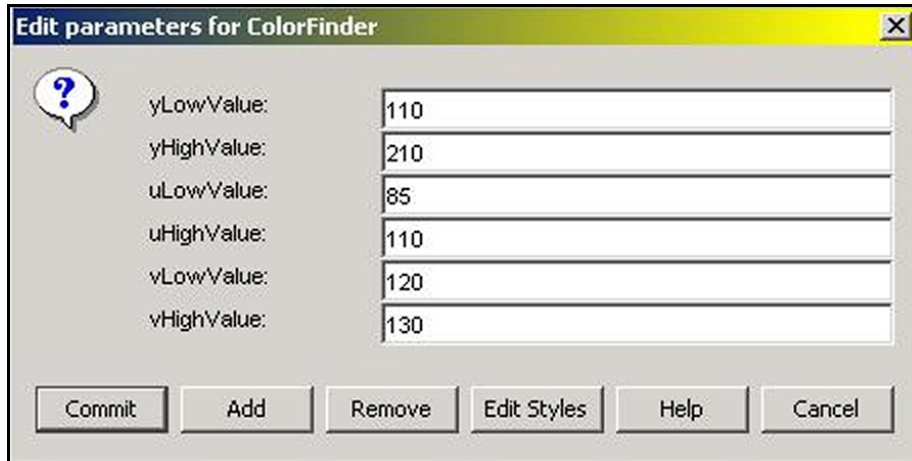
The VideoCamera actor¹ allows a video camera (more specifically “webcams”), to be used as a source for video. It allows the user to choose between either using the RGB or YUV formats. While the images might look the same, accessing the data that each frame contains is different.

For systems with more than one video camera attached to their system, the deviceNumber parameter allows you to choose which camera to use. To determine which camera corresponds to which device number, the user should run the JMF Registry Editor which comes with JMF. Multiple cameras cannot be used because each one requires a separate Java Virtual Machine.

Unlike the MovieReader actor, the VideoCamera actor will output frames at best effort, and likely will not output every single frame that the camera reads in.

¹based on code originally written by Paul Yang and David Lee

6.1.4 ColorFinder



The ColorFinder actor² tracks the “center of mass” for a certain range of colors. This range is specified by a range of Y, U, and V values to look for. Each parameter must be an integer between 0 to 255.

The output is the x and y coordinate of the center (note that the notion of origin is different because in JAI, the origin is in the top left corner, but in an x-y plot in Ptolemy II, the origin is in the bottom left corner).

6.2 Streaming Actors

The following actors exist outside of the framework, and cannot be used with any other actors. In JMF, media files can be encapsulated inside data structures called DataSources. DataSources are operating on a whole instead of image by image with the previous actors. A conversion to AWT imaging doesn’t exist, however, these actors are here for the basis of future work.

When using these actors, it is important to use a SDF director with the iterations set to 1. This is because a audio or video file is encapsulated all at once, so only one iteration of a model is needed.

6.2.1 StreamLoader

The StreamLoader actor encapsulates a media file from a file or URL.

6.2.2 AudioPlayer

The AudioPlayer actor plays MIDI, MP3, and CD audio files. All playback control is done through the control panel, which pops up after the model containing

²based on code originally written by Paul Yang and David Lee

this actor is run.

6.2.3 VideoPlayer

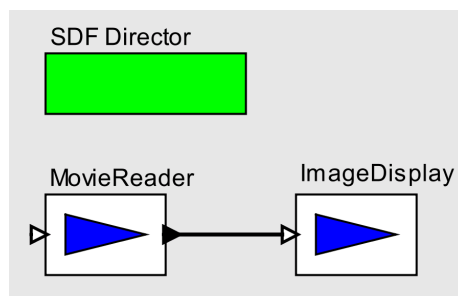
The VideoPlayer actor plays AVI, Quicktime, and MPEG files, just like the MovieReader actor. It is less restrictive in which types of these files it can play because the MovieReader actor requires the file to be able to not only be able to be accessible for each frame, but it also requires control of movement in the movie file.

Chapter 7

Video Processing Examples

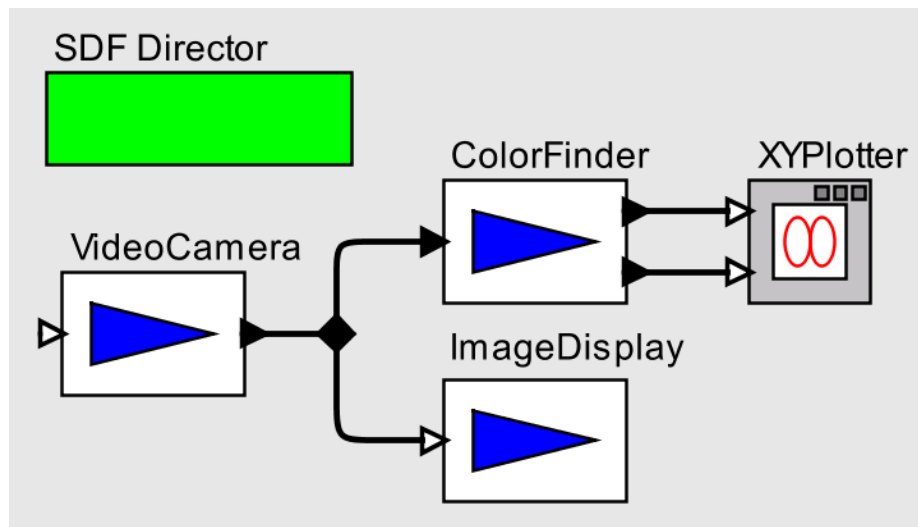
7.1 Playing a Movie

This simple model loads and displays a movie.



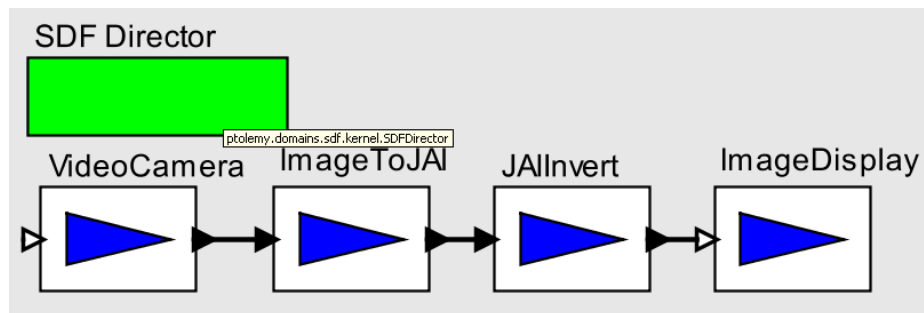
7.2 Item Tracking

This model takes a YUV output from a webcam and uses the ColorFinder actor to attempt to look for a certain color (by default, green), in an image. If the user waves a piece of green construction paper in front of the camera, this actor will be able to detect its “center of mass” and output an x-y coordinate.



7.3 Putting it all Together, JMF and JAI

The ImageToJAI actor allows us to use JAI actors on any token that an AW-ImageToken can be extracted from. This model simply inverts the webcam output and displays it.



Chapter 8

Future Work

Sun has recently released Image I/O[6], a replacement for JAI's file loading/writing schemes. It adds JPEG 2000 reading and writing capabilities, something that is most certainly lacking in the current implementation.

In theory, this makes writing a file reader for a custom file type much easier due to Image I/O's plug-in architecture. In JAI's old file loading scheme, all file loading nuances were abstracted away from the user, and while this made it easy to use, it made it impossible to customize.

There is also a lot of potential work to be done in the video processing platform. The DataSource paradigm can be explored in further detail, and perhaps there is a natural link between it and the Buffer paradigm.

Bibliography

- [1] -----, “Java Media Framework API Guide”, Sun Microsystems. Mountain View, CA. November 19, 1999.
- [2] -----, “Programming in Java Advanced Imaging”, Sun Microsystems. Palo Alto, CA. November, 1999.
- [3] J. Davis, C. Hylands, J. Janneck, E.A. Lee, J. Liu, X. Liu, S. Neuendorffer, S. Sachs, M. Stewart, K. Vissers, P. Whitaker, Y. Xiong. “Overview of the Ptolemy Project”. Technical Memorandum UCB/ERL M01/11, EECS, University of California, Berkeley, March 6, 2001.
- [4] C. Fong, “Discrete-Time Dataflow Models for Visual Simulation in Ptolemy II”, MS Report, Dept. of EECS, University of California, Berkeley, CA 94720. December 2000.
- [5] J.S. Lim, “Two-Dimensional Signal and Image Processing”, Prentice-Hall Inc. Englewood Cliffs, NJ. 1990.
- [6] P. Race, D. Rice, R. Vera, “Java Image I/O API Guide”, Sun Microsystems. Palo Alto, CA. April 2001.

Chapter 9

Appendix

The following is Java code that implements an adaptive median filtering problem.

```
/* An actor that performs adaptive median filtering on a double
matrix.
```

```
@Copyright (c) 2002-2003 The Regents of the University of
California. All rights reserved.
```

```
Permission is hereby granted, without written agreement and
without license or royalty fees, to use, copy, modify, and
distribute this software and its documentation for any purpose,
provided that the above copyright notice and the following two
paragraphs appear in all copies of this software.
```

```
IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY
PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS
DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN
ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE
SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE
UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE,
SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
```

```
PT_COPYRIGHT_VERSION 2
COPYRIGHTENDKEY
```

```

*/

package ptolemy.actor.lib.jai;

import ptolemy.actor.lib.Transformer;
import ptolemy.data.DoubleMatrixToken;
import ptolemy.data.IntToken;
import ptolemy.data.expr.Parameter;
import ptolemy.data.type.BaseType;
import ptolemy.kernel.CompositeEntity;
import ptolemy.kernel.util.Attribute;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.NameDuplicationException;

////////////////////////////////////
//// AdaptiveMedian

/**
    This actor performs adaptive median filtering on an image. The
    algorithm is as follows. For each pixel in the image, a square region
    of interest is formed with the pixel at the center. If said region
    can not be formed (because it is at or near the edge).

    If the median of this region of interest is strictly less than the
    maximum value in the region of interest and strictly greater than
    the minimum value in the region of interest, then we keep the pixel
    if it too is strictly less than the maximum value in the region of
    interest and strictly greater than the minimum value in the region
    of interest. If it is not, then use the median of the region of
    interest instead of the pixel.

    If the pixel is not strictly less than the maximum value and strictly
    greater than the minimum value, an attempt is made at using larger
    region of interest. If successful, then this process is repeated until
    a value can be determine, or we hit the the maximum window size. If this
    happens, then the pixel is kept. This process is repeated for each pixel.

    @author James Yeh
    @version $Id: AdaptiveMedian.java,v 1.1 2003/11/20 22:29:28 jiahau Exp $
    @since Ptolemy II 3.0
*/

public class AdaptiveMedian extends Transformer {

    /** Construct an actor with the given container and name.
     *  @param container The container.

```

```

* @param name The name of this actor.
* @exception IllegalArgumentException If the actor cannot be contained
*   by the proposed container.
* @exception NameDuplicationException If the container already has an
*   actor with this name.
*/
public AdaptiveMedian(CompositeEntity container, String name)
    throws IllegalArgumentException, NameDuplicationException {
    super(container, name);
    input.setTypeEquals(BaseType.DOUBLE_MATRIX);
    output.setTypeEquals(BaseType.DOUBLE_MATRIX);

    maxWindowSize = new Parameter(this, "maxWindowSize", new IntToken(7));
}

////////////////////////////////////
////                                ////
/** The largest window size to use. This number must be an odd
 * integer. The default value is 7.
 */
public Parameter maxWindowSize;

////////////////////////////////////
////                                ////

/** Override the base class and set the largest window size.
 * @param attribute The attribute that changed.
 * @exception IllegalArgumentException If the largest window size is
 *   not an odd integer.
 */
public void attributeChanged(Attribute attribute)
    throws IllegalArgumentException {
    if (attribute == maxWindowSize) {
        _maxWindowSize = ((IntToken)maxWindowSize.getToken()).intValue();
        if (_maxWindowSize%2 == 0) {
            throw new IllegalArgumentException(this,
                "Window Size must be odd!!");
        }
    } else {
        super.attributeChanged(attribute);
    }
}

/** Fire this actor.
 * Perform the adaptive median filtering.

```

```

    * @exception IllegalArgumentException If a contained method throws it.
    */
    public void fire() throws IllegalArgumentException {
        super.fire();
        DoubleMatrixToken doubleMatrixToken = (DoubleMatrixToken) input.get(0);
        double data[][] = doubleMatrixToken.doubleMatrix();
        int width = doubleMatrixToken.getRowCount();
        int height = doubleMatrixToken.getColumnCount();
        double outputData[][] = new double[width][height];
        int windowSize = 3;
        //Iterate over each pixel.
        for (int i = 0; i < width; i++) {
            for (int j = 0; j < height; j++) {
                while (true) {
                    int dist = (windowSize - 1)/2;
                    //check if we can create a region of interest or not.
                    //If we can't (i.e. we are at or near the edge of an image)
                    //then just keep the data.
                    if ((i - dist < 0) || (j - dist < 0)
                        || (i + dist >= width) || (j + dist >= height)) {
                        outputData[i][j] = data[i][j];
                        windowSize = 3;
                        break;
                    }
                }
                else {
                    double temp[][] = new double[windowSize][windowSize];
                    //create a local region of interest around the pixel.
                    for (int k = (i - dist); k <= (i + dist); k++) {
                        for (int l = (j - dist); l <= (j + dist); l++) {
                            temp[k - (i - dist)][l - (j - dist)] = data[k][l];
                        }
                    }
                    double median = _getMedian(temp, windowSize);
                    double max = _getMax(temp, windowSize);
                    double min = _getMin(temp, windowSize);
                    //If the median of the region of interest is
                    //strictly less than the maximum value, and strictly
                    //greater than the minimum value, we then have two
                    //routes. If the data in the center is strictly
                    //greater than the minimum, and stricly less than
                    //the maximum, then just keep the data point.
                    //If it is either the minimum or the maximum, then
                    //output the medium because there is a very good chance
                    //that the pixel was noised. After this, the window size
                    //is reset.
                    if ((median > min) && (median < max)) {

```

```

        if((data[i][j] > min) && (data[i][j] < max)) {
            outputData[i][j] = data[i][j];
            windowSize = 3;
            break;
        }
        else {
            outputData[i][j] = median;
            windowSize = 3;
            break;
        }
    } else if (windowSize < _maxWindowSize) {
        //If this statement is reached, this means that
        //the median was equal to either the minimum or the
        //maximum (or quite possibly both if the region of
        //interest had constant intensity. Increase the window
        //size, if it is less than the maximum window size.
        windowSize = windowSize + 2;
    } else {
        //If this statement is reached, we've already hit
        //the maximum window size, in which case, just output
        //the data and reset the window size.
        outputData[i][j] = data[i][j];
        windowSize = 3;
        break;
    }
}
}
}
}
}
output.send(0, new DoubleMatrixToken(outputData));
}

////////////////////////////////////
////                                private methods                                ////

/** Find the largest value in a region of interest.
 */
private double _getMax(double[][] input, int size) {
    double temp = input[0][0];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (input[i][j] > temp) {
                temp = input[i][j];
            }
        }
    }
}
}

```

```

        return temp;
    }

    /** Find the median value in a region of interest.
     */
    private double _getMedian(double[][] input, int size) {
        double[] temp = new double[size*size];
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                temp[i*size + j] = input[i][j];
            }
        }
        for (int i = 0; i < size*size; i++) {
            for (int j = 0; j < size*size - 1; j++) {
                if(temp[j] > temp[j+1]) {
                    double tempval = temp[j];
                    temp[j] = temp[j+1];
                    temp[j+1] = tempval;
                }
            }
        }
        return temp[(size*size-1)/2];
    }

    /** Find the minimum value in a region of interest.
     */
    private double _getMin(double[][] input, int size) {
        double temp = input[0][0];
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                if (input[i][j] < temp) {
                    temp = input[i][j];
                }
            }
        }
        return temp;
    }

    //////////////////////////////////////
    ///                                private variables                                ///
    //////////////////////////////////////

    // The largest window size.
    private int _maxWindowSize;
}

```