Into ImgLib—Generic Image Processing in Java

Stephan Preibisch, Pavel Tomančák, and Stephan Saalfeld

Max Planck Institute of Molecular Cell Biology and Genetics, Pfotenhauerstrasse 108, Dresden, Germany

ABSTRACT

The purpose of ImgLib, a Generic Java Image Processing Library, is to provide an abstract framework enabling Java developers to design and implement data processing algorithms without having to consider dimensionality, type of data (e. g. byte, float, complex float), or strategies for data access (e. g. linear arrays, cells, paged cells). This kind of programming has significant advantages over the classical way. An algorithm written once for a certain class of Type will potentially run on any compatible Type, even if it does not exist yet. Same applies for data access strategies and the number of dimensions.

We achieve this abstraction by accessing data through Iterators and Type interfaces. Iterators guarantee efficient traversal through pixels depending on whether random coordinate access is required or just all pixels have to be visited once, whether real or integer coordinates are accessed, whether coordinates outside of image boundaries are accessed or not. Type interfaces define the supported operators on pixel values (like basic algebra) and hide the underlying basic type from algorithm implementation.

Keywords: imglib, generic programming, image processing, java, fiji

1. INTRODUCTION

Advanced image processing tasks require implementation of complex algorithms. At the same time, the rising amount and shear size of n-dimensional (2d, 3d, 4d, and more) image data requires intelligent strategies for storage (e. g. in memory, paged on disc, distributed over the net) and data access (e. g. iteration, random access). Moreover, diverse imaging modalities generate a multitude of pixel types (e. g. wavelength, frequency spectra, labels, orientation histograms) at various precisions (e. g. 1-bit boolean, 8-bit unsigned integer, 32-bit signed float). ImgLib aims at separating algorithm development from core design: storage, access, and pixel operators. Algorithm developers can focus on the core of the algorithm exclusively, i. e. he/she has to implement the algorithm once and it will run on any compatible pixel type, number of dimensions, and independently of how the data is stored. Another benefit is the more concise source code compared to classical implementations. It is, therefore, easier to understand the essence of the algorithms while low level access is optimized at the level of *Types*, *Cursors*, and *Containers*.

2. DESIGN

To achieve dimension, storage, and type independence, we employ four major abstraction levels. Figure 1 visualizes these basic levels of abstraction and, by that, outlines the major design principles. The complete source code can be accessed via gitweb `http://pacific.mpi-cbg.de/cgi-bin/gitweb.cgi?p=imglib.git`.

1. *Types* define the abstraction layer for supported data types, that is, the values single pixels can store. Inheritance hierarchies allow to implement algorithms for certain subgroups (e. g. *ComparableType*) or individual *Types* (e. g. *FloatType*). The leafs of this inheritance tree are final implementations that guarantee maximal performance for the supported operations and minimize the required storage space (e. g. 1-bit boolean, 12-bit unsigned integer). *Types* define to which and to how many bits of a Java basic type they

Further author information: (Send correspondence to S.P. or S.S.)
S.P. E-mail: preibisch@mpi-cbg.de Telephone: +49 351 210-2758
P.T. E-mail: tomancak@mpi-cbg.de Telephone: +49 351 210-2670
S.S. E-mail: saalfeld@mpi-cbg.de Telephone: +49 351 210-2753

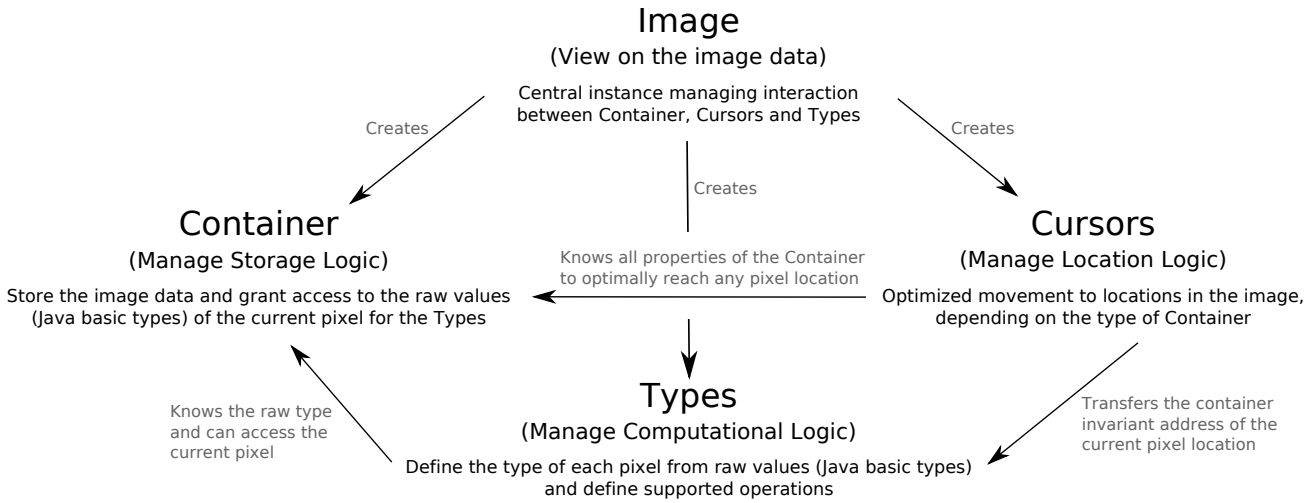

Figure 1. Visualization of the four major abstraction levels *Type*, *Cursor*, *Container*, and *Image*.

are mapped for storage. Currently, ImgLib provides a selection of commonly used numeric types covering the Java basic types, unsigned integers, complex numbers and booleans. Additional types for special purpose algebras are easy to implement (e. g. base pairs, labels, compounds).

2. *Cursors* are used to address certain pixel locations in the image independently of the *Type*. *Cursors* are implemented for each *Container* and, therefore, completely abstract the addressing of pixel locations from the actual implementation of algorithms. *Cursors* differ in terms of their flexibility to move, whereas higher flexibility results in lower performance. There are two main classes of *Cursors*: *Iterators* and *Random Access Cursors*. Specialized *Cursors* (e. g. *Iterators* for local neighborhoods) simplify standard operations in algorithm design.

3. *Containers* implement in which order the data fragments that represent one pixel (*Type*) are stored, how they are fetched, generated, and/or potentially cached. *Cursors* define how to reach a certain location and the order of iteration but they are not aware of the *Type*. For the *Type*, the *Container* provides access to the actual data at the current location whereas the *Type* itself is unaware of both the *Container* and the *Cursor*. Subsuming, *Cursors* define the location and *Types* the value of a pixel, *Containers* implement the logic that connects both. ImgLib currently provides linear arrays, a multi-cell container, a generative vector container that creates pixels from Java AWT shapes, and a legacy ImagePlus container that allows running an ImgLib algorithm on an existing ImagePlus instance.

4. *Images* are the central instances that link *Containers*, *Cursors*, and *Types* and provide a view to the data. An *Image* is instantiated by the developer providing *Factories* that define which *Container* and *Type* to use. *Cursors* are created by the *Image* upon demand propagating the request to the *Container*. That way, the *Container* implicitly defines which *Cursors* have to be used.

Higher level concepts build on top of these basic layers of abstraction. E. g. random access strategies at coordinates that might be outside of image boundaries are implemented through *Cursors* that map such coordinates to a defined location inside the *Image* (e. g. periodic, mirror, boundary extension) and/or perform operations on the *Type* level (e. g. constant values, fading). Random access at real coordinates is realized through *Interpolators* (e. g. nearest neighbor, linear) that perform *Type* operations in a defined integer neighborhood that, again, is iterated using *Cursors*. That way, both out of boundary access as well as interpolation serve as examples of basic algorithms whose implementation is independent of data storage, type, and dimensionality with the optional requirement that the *Type* implements a set of required operations.

3. USAGE

To instantiate an *Image*, one can load an *Image*, create an *Image* of a certain *Type* or create a new *Image* based on an existing instance which will have the same *Type*. To load or create an *Image*, one has to define the *Container* that defines in which way memory is allocated. Once an *Image* is instantiated, the algorithm accesses the pixels using appropriate *Cursors* and calculates using the operations provided by the *Type*. The following example code opens and image, iterates all pixels and adds an increasing number to each pixel.

```
// define T as some RealType in this method
public <T extends RealType<T>> void example() {
	// open the image and use an CellContainer with cell size 4 × 4 × ...4 to store it
	Image<T> image = LOCI.openLOCI("img.tif", new CellContainerFactory(4));
	// call the generic method to add some values to it
	addValues(image);
}
public <T extends RealType<T>> void addValues(Image<T> image) {
	// create cursor
	final Cursor<T> c = image.createCursor();
	// create variable of same type and set to one
	final T type = image.createType();
	type.setOne();
	// iterate over image
	while (c.hasNext()) {
		// move iterator forward
		c.fwd();
		// add the value of type to the current pixel value
		c.getType().add(type);
		// increase type
		type.inc();
	}
	// close the cursor
	c.close();
}
```

Note that the generic method `addValues(Image<T> image)` in this example works for any *Image* that contains pixels with real values, independent of the dimensionality of the *Image* or the *Container* that is employed. The next example creates a new *Image* whose pixels are of *FloatType* (32-bit signed floating point) and calls the same method to highlight the generic properties.

```
public void example() {
	// create an Image of FloatType with an ArrayContainer to store it
	ImageFactory<FloatType> factory =
			new ImageFactory<FloatType>(new FloatType(), new ArrayContainerFactory());
	Image<FloatType> image = factory.createImage(new int[]{10, 10, 10});
	// call the generic method to add some values to it
	addValues(image);
}
```

The implementation is independent of the *Container* whereas, in this example, the order of iteration has an effect on the result. That is to highlight that basic iteration is optimal in terms of the *Container* but does not guarantee any particular order. However, for many applications (e. g. estimating min and max, averaging), the order of iteration is irrelevant.

Images can be displayed through ImageJ,[1] either as ImageJ virtual stacks or by copying the content into a new ImageJ `ImagePlus` instance. Displaying *Images* as virtual stacks should be preferred as it requires no duplication of data. Both displays can show an arbitrary subset of image dimensions in arbitrary order (e. g. y, xy, xyz, zxy, txy) which offers a very elegant and fast way to display images in different orientations.

```
public <T extends RealType<T>> void example() {
      // open a 3-dimensional image using ImageJ
      ImagePlus imp = new Opener().openImage("image.tif");
      // create an Image on top of an existing ImagePlus instance
      Image<T> img = ImagePlusAdapter.wrap(imp);
      // reset min and max of the display
      img.getDisplay().setMinMax();
      // display as Virtual Stack
      ImageJFunctions.displayAsVirtualStack(img).show();
      // display by copying into a new ImagePlus instance, the stack will be displayed
      // in the order z,x,y as defined by the id's of the dimensions {2,0,1}
      ImageJFunctions.copyToImagePlus(img,new int[]{2,0,1}).show();
}
```

For algorithm development with ImgLib, we suggest using the Eclipse IDE for Java Developers (JDT) (http://www.eclipse.org/downloads/) and a Java 6 JDK (http://www.oracle.com/technetwork/java/javase/downloads/index.html). Required libraries are:

1. ImgLib (imglib.jar, included in Fiji http://pacific.mpi-cbg.de/)
2. LOCI Bioformats library standalone for import of images (bio-formats.jar, http://www.loci.wisc.edu/ome/formats-download.html)
3. Fiji/ImageJ (ij.jar, included in Fiji http://pacific.mpi-cbg.de/)
4. Transformation Package (mpicbg_.jar, included in Fiji http://pacific.mpi-cbg.de/)
5. FFT Package (edu_mines_jtk.jar, included in Fiji http://pacific.mpi-cbg.de/)

The source code of ImgLib can be downloaded from the public GIT repository ssh://contrib@pacific.mpi-cbg.de/srv/git/imglib.git. Example projects, code templates and other workshop material is available on-line at http://fly.mpi-cbg.de/imglib-workshop/.

4. CONCLUSIONS

We introduced ImgLib, a state-of-the-art generic image processing framework for the Java programming language that enables developers to implement algorithms independently of the number of dimensions, data type and the way the data are stored. For Java, this is the first framework that targets this level of generality, whereas basic design principles are similar to ITK[2] and Vigra[3] which are powerful generic image processing frameworks for C++.

ImgLib has been tested in complex applications. It serves as the image processing library for our software for reconstructing multi-angle long-term time-lapse acquisitions imaged with the selective plane illumination microscope.[4,5] ImgLib has proven to be capable of reliable and fast processing data sets of up to 0.5 terabytes. Together with the development team of LOCI Bioformats http://www.loci.wisc.edu/software/bio-formats, we are working to incorporate ImgLib as the basis for the new NIH-funded ImageJ2 http://imagejdev.org/, underlining the potential impact of the framework.

ACKNOWLEDGMENTS

Stephan Preibisch and Stephan Saalfeld were supported by a DIGS-BB PhD stipend. We thank Johannes Schindelin, the LOCI team, in particular Curtis Rueden and Grant Harris, and Larry Lindsey for their support and valuable discussion.

REFERENCES

[1] Rasband, W., "ImageJ: Image processing and analysis in Java [version 1.44e]."

[2] Ibanez, L., Schroeder, W., Ng, L., and Cates, J., *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-15-7, http://www.itk.org/ItkSoftwareGuide.pdf, second ed. (2005).

[3] Koethe, U., *Generische Programmierung für die Bildverarbeitung*, PhD thesis, Universität Hamburg (2000).

[4] Huisken, J., Swoger, J., Bene, F. D., Wittbrodt, J., and Stelzer, E. H. K., "Optical sectioning deep inside live embryos by selective plane illumination microscopy," *Science* **305**, 1007–1010 (2004).

[5] Preibisch, S., Saalfeld, S., Schindelin, J., and Tomančák, P., "Software for bead-based registration of selective plane illumination microscopy data," *Nature Methods* **7**, 418–419 (June 2010).