The Generic Java Image Processing Library

Stephan Preibisch, Stephan Saalfeld, Johannes Schindelin, Pavel Tomancak

The purpose of the *"Generic Java Image Processing Library"* is to provide an abstract system enabling Java developers to design and implement data processing algorithms without having to consider the type of data (e.g. byte, float, complex float), its dimensionality or storage type (e.g. array, cubes, paged cubes).

Types

The developer therefore uses different kinds of abstract *cursors* `(e.g. mpi.imglib.cursor.LocalizableCursor)` that an *image* `(mpi.imglib.image.Image)` can create based on its own type, dimensionality and storage strategy. To ensure type safety *cursors* as well as *images* are typed using Java Generics `(e.g. LocalizableCursor<T>, Image<T>)`. Types supported so far comprise:

- ByteType `(mpi.imglib.type.ByteType)`, 8 bit signed integer
- ShortType `(mpi.imglib.type.ShortType)` 16 bit signed integer
- IntType `(mpi.imglib.type.IntType)` 32 bit signed integer
- LongType `(mpi.imglib.type.LongType)` 64 bit signed integer
- FloatType `(mpi.imglib.type.FloatType)` 32 bit signed floating point
- DoubleType `(mpi.imglib.type.DoubleType)` 64 bit signed floating point
- RGBALegacyType `(mpi.imglib.type.imagej.RGBALegacyType)` 4 channel, 8bit signed stored as 32 bit signed integer

All of the types support basic math functions, which can be called independent of their actual type using Java generics `(see mpi.imglib.type.Type)`:

```
Image<T> img;
...
Type<T> type = img.createType();
type.inc();
type.mul( Math.PI );
```

Images

To instantiate an *image*, the developer 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 the developer has to define the storage strategy `(mpi.imglib.container.ContainerFactory)` that defines in which way memory is allocated. Currently the following storage strategies are supported:

- ArrayContainer `(mpi.imglib.container.array.ArrayContainerFactory)`, stores all values in one single array, fast but limited to 2 billion pixels (2^{31} pixels)
- CubeContainer `(mpi.imglib.container.cube.CubeContainerFactory)`, stores the values in cubes, i.e. the data will be stored in many small arrays of fixed dimensions (e.g. 64x64x64 for 3d). This strategy is slower but not limited in size

- ImagePlusContainer `(mpi.imglib.container.imageplus.ImagePlusContainerFactory)`, supported to edit ImageJ images directly, limited to 3 dimensions

Images can be instantiated in the following ways:

- Loading an image
    - `Image<?> img = LOCI.openLOCI(“D:/Temp/img.tif”, new ArrayContainerFactory());`
    - Here, the LOCI reader decides which *Type* the *Image* will be loaded as. This image, however, can be used to call any generic method (see section Generic Classes and Methods)

    - The developer can also try to load an image as a certain Type, this is very useful if the algorithm for example is written for a certain type only

    -
      ```
      Image<FloatType> img = LOCI.openLOCIFloatType(“D:/Temp/img.tif”,
                                                    new ArrayContainerFactory());
      ```
    - Not all Types are supported yet

- Creating a new image
    - When creating an *Image*, the developer has to decide which Type to use

    -
      ```
      ImageFactory<FloatType> imageFactory =
          new ImageFactory<FloatType>(new FloatType(), new CubeContainerFactory());
      Image<FloatType> img = factory.createImage(new int[]{100,50,10,2});
      ```
      This creates a 4-dimensional image of float using cubes for storage

- Creating a new image from an existing one
    - If an image instance is given already, a new image of the same type and storage strategy can be instantiated
    - `Image<T> img;`
    - `Image<T> img2 = img.createNewImage();`
      This creates an image of the same size
    - `Image<T> img2 = img.createNewImage(new int[]{100,100});`
      This creates a 2d image of the same type and storage strategy

Cursors

Once images are loaded or created the developer performs computation using the already mentioned cursors. Each cursor has a Type<T> available through `cursor.getType()`. This Type<T> contains the value of the pixel pointed to by the cursor. To this end, five types of cursors are available:

- Cursor<T> `(mpi.imglib.cursor.Cursor<T>)`, iterates over all pixels of the image with no predefined order. The order depends on the storage strategy but will be the same for image with same dimensions and exactly the same storage strategy `(Image<T> img2 = img.createNewImage())`. This is the fastest cursor but also the most limited one; it is useful to compute the average of an image or subtract two images, for example.
  To instantiate call: `Cursor<T> cursor = img.createCursor();`

- LocalizableCursor<T> `(mpi.imglib.cursor.LocalizableCursor<T>)`, behaves exactly as the simple Cursor<T>, but knows its position inside the image which makes it slightly slower. This cursor can be used to compute algorithms such as the center of mass or to link positions between two images of different storage strategies.
  To instantiate call: `LocalizableCursor<T> cursor = img.createLocalizableCursor();`
- LocalizableCursorByDim<T> `(mpi.imglib.cursor.LocalizableCursor<T>)`, can iterate over the image as well but is further able to move around freely inside or even outside of the image. This is the most flexible cursor that can basically do anything but is therefore also the slowest one.
  To instantiate call:
  ```
  LocalizableCursorByDim<T> cursor = img.createLocalizbleCursorByDim();
  ```
  or
  ```
  LocalizableCursorByDim<T> cursor
                = img.createLocalizbleCursorByDim( new OutsideStrategyFactory() );
  ```
  The OutsideStrategyFactory defines the behavior of the cursor when moving outside of the image, it can currently return a constant value or mirror the image content. If called without any OutsideStrategyFactory it will perform no check whether the coordinate is inside the image (crashes or gives wrong results if not!). This version is more optimized but should only be used if the developer is sure that the cursor will not move out of the image.

Interpolation

The image can also create interpolators which are needed to iterate over the image off the pixel grid.

To this end, Nearest Neighbor

```
InterpolatorFactory<T> interpolatorFactory =
new NearestNeighborInterpolatorFactory<T>( outsideStrategyFactory );
```

and Linear Interpolation

```
InterpolatorFactory<T> interpolatorFactory = new
LinearInterpolatorFactory<T>( outsideStrategyFactory );
```

are supported. The final interpolator is then created as follows:

```
Interpolator<T> interpolator = img.createInterpolator( interpolatorFactory );
```

Developing algorithms

The introduction showed how computation in general works using the library. Here is example code of a simple algorithm:

Open image and add an increasing value (which will visualize the way the cursor moves over the image):

```
Image<?> img = LOCI.openLOCI(“D:/Temp/img.tif”, new ArrayContainerFactory());
addValues( img );
```

Or

```
ImageFactory<FloatType> imageFactory =
       new ImageFactory<FloatType>(new FloatType(), new CubeContainerFactory());
Image<FloatType> img2 = factory.createImage(new int[]{100,50,10,2});
addValues( img2 );
```

Implementation of the algorithm as generic method

```
public <T extends Type<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();

              // set cursor to the value of type
              c.getType().add( type );

              // increase type
              type.inc();
       }

       // close the cursor
       c.close();
}
```

Writing algorithms for one specific Type (e.g. only FloatType) will look like this:

```
Image<FloatType> img=LOCI.openLOCIFloatType(“D:/Temp/img.tif”,new ArrayContainerFactory());
computeAverage( img );

public FloatType computeAverage( Image<FloatType> img )
{
        // create cursor
        final Cursor<FloatType> c = image.createCursor();

        // create variable of same type and set to one
        final float sum = 0;

        // the number of pixels in the image
        final int numPixels = img.getNumPixels();

        // iterate over image
        while ( c.hasNext() )
        {
                // move iterator forward
                c.fwd();

                // set cursor to the value of type
                // division not outside loop to prevent overflow
                // correctly one would have to use BigInteger here
                sum += c.getType().get() / numPixels;

        }

        // close the cursor
        c.close();

        return new FloatType( sum );
}
```

To display images there are currently two methods, displaying them as ImageJ Virtual Stacks (preferred) which does not use any more memory or copying them to ImageJ ImagePlus instances which can then be displayed. In any case, the dimensionality might be reduced as only 3 dimensions can be displayed.

```
Image<?> img = LOCI.openLOCI(“D:/Temp/img.tif”, new ArrayContainerFactory());
img.getDisplay().setMinMax();

ImageJFunctions.displayAsVirtualStack( img ).show();
ImageJFunctions.copyToImagePlus( img ).show();
```

There are more details to the whole library which will become more obvious once one starts using it. One major issue is not solved yet, which is how to deal generically with Multi-Channel images. The current design is, however, almost finished; it will be implemented on top of the Image<T> class and will integrate seamlessly into the current processing structure. Multichannel images will simply create different cursors.

More example code can be found in the Test.java class located here: `mpi.imglib.Test`.

For developing we suggest using the 32-bit version of Eclipse IDE for Java Developers (JDT) (http://www.eclipse.org/downloads/) which can also run 64-bit Java Virtual Machines. Necessary libraries are:

- LOCI Bioformats library standalone for import of images (bio-formats.jar, http://www.loci.wisc.edu/ome/formats-download.html)
- Fiji/ImageJ (ij.jar, http://rsbweb.nih.gov/ij/download/zips/ij142.zip)

The source of this library should be added as a new project which is then added as necessary project for the development of the own source code. All JAR imports as well as necessary projects are configured by right clicking on Referenced Libraries -> Build Path -> Configure Build path.

Please consider that this library is still alpha stage, it has been tested but there will be still some errors. The most stable container is the ArrayContainer, use it whenever possible but also the ImagePlusContainer should work properly. The algorithms in there (Gaussian Convolution, Affine Transformation (needs the mpicbg packa;ge available through Fiji, http://pacific.mpi-cbg.de/), Linear Interpolation, Nearest Neighbor Interpolation) are all written completely generically. Therefore their performance is not optimal. We added some implementations for optimized speed, currently for 3d and ArrayStorage. This optimization is still not available for the Linear Interpolator which will be added soon.

There is a compatibility layer to ImageJ using the ImagePlusContainer. To work on an existing ImagePlus it has to be wrapped into an Image<T> of the library; this works without copying the data.

```
ImagePlus imp = new Opener().openImage("D:/Temp/img.tif");

// converts it into an image of unknown type
Image<?> img = ImagePlusAdapter.wrap( imp );

// converts it into an image of FloatType
Image<FloatType> img = ImagePlusAdapter.wrapFloat( imp );
```

Generic Classes and Methods

Generic classes or methods can be called with any instance of Image<?>. It ensures that inside that class or method all occurrences of <T> are of the same Type, allowing generic programming. A generic class will look like this:

```
public class GaussianConvolution< T extends Type<T> >
{
	Cursor<T> cursor;

	public GaussianConvolution( Image<T> img )
	{
	...
	}
}
```

All <T> in this class will be of the same type, and any Image<?>, Image<T> or specialized version e.g. Image<FloatType> can be given as parameter. To instantiate a typed class, a typed method is, however, necessary, see the example below.

```
public class UnTypedClass
{
	pulic void untypedMethod()
	{
		Image<?> img = LOCI.openLOCI(“D:/Temp/img.tif”, new ArrayContainerFactory());

		// here we cannot instantiate the typed class, but a typed method which can
		typedMethod( img );
	}


	public <T extends Type<T>> void typedMethod( Image<T> img )
	{
		// inside here, all <T> are the same, that’s why it works
		GaussianConvolution<T> convol = new GaussianConvolution( img );
	}
}
```