Fast Region Competition in ImageJ

Bachelor Thesis
Stephan Semmler
September 14, 2012

Advisors: Prof. Dr. Ivo Sbalzarini, Janick Cardinale
Department of Computer Science, ETH Zürich
Abstract

In this work, a porting of a region competition based image segmentation algorithm from C++ to Java is presented. The algorithm is implemented as a plugin for the open source image processing software ImageJ. The design of the implementation of the plugin aims for usability and expansibility. The segmentation algorithm depends on concepts of digital topology, and by the lack of an existing solution on this field for Java, a framework for digital topology that works in a generic way is implemented.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Project Description</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Image Segmentation</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Algorithm</td>
<td>1</td>
</tr>
<tr>
<td>1.4 ImageJ</td>
<td>2</td>
</tr>
<tr>
<td>1.4.1 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.4.2 The program</td>
<td>2</td>
</tr>
<tr>
<td><strong>Basic Data Structures</strong></td>
<td>4</td>
</tr>
<tr>
<td>2.1 Point</td>
<td>4</td>
</tr>
<tr>
<td>2.2 IntensityImage</td>
<td>5</td>
</tr>
<tr>
<td>2.3 LabellImage</td>
<td>5</td>
</tr>
<tr>
<td>2.4 IndexIterator</td>
<td>5</td>
</tr>
<tr>
<td>2.5 Remarks</td>
<td>6</td>
</tr>
<tr>
<td><strong>Digital Topology</strong></td>
<td>7</td>
</tr>
<tr>
<td>3.1 Theoretical part</td>
<td>7</td>
</tr>
<tr>
<td>3.1.1 Connectivity</td>
<td>7</td>
</tr>
<tr>
<td>3.1.2 Topological number</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Implementation</td>
<td>9</td>
</tr>
<tr>
<td>3.2.1 Connectivity</td>
<td>9</td>
</tr>
<tr>
<td>3.2.2 UnitCubeCCCounter</td>
<td>10</td>
</tr>
<tr>
<td>3.2.3 TopologicalNumberImageFunction</td>
<td>10</td>
</tr>
<tr>
<td>3.2.4 Result</td>
<td>11</td>
</tr>
<tr>
<td>3.3 Flood Fill</td>
<td>12</td>
</tr>
<tr>
<td>3.3.1 Introduction</td>
<td>12</td>
</tr>
<tr>
<td>3.3.2 Multiple Threshold function</td>
<td>13</td>
</tr>
<tr>
<td>3.3.3 Application</td>
<td>13</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Project Description

The topic of this bachelor thesis is to port a given image segmentation algorithm, written in C++ for the Insight Segmentation and Registration Toolkit [3] to Java as a plugin for the image processing software ImageJ [2]. The purpose of this is to make the algorithm available to non-expert users. The plugin should provide visualization of segmentation and the progress of it. The algorithm shall work in a generic way for 2D and 3D images. And at last, the algorithm should run reasonable fast, that is, comparable to the performance of the C++ version.

1.2 Image Segmentation

Image segmentation is the process of partitioning a digital image into multiple segments. A segmented image may be easier to analyze or more meaningful for a human, or a secondary algorithm. An application of such an algorithm could consist of some measurements or object recognition, for example. There are several methods to achieve image segmentation, from simple thresholding, over clustering methods, “snakes”, level set methods and graph partitioning methods. The present algorithm is a particular implementation of the “region-competition” scheme proposed by [11].

1.3 Algorithm

In this section, a short review of the underlying algorithm is given, so in later sections of the text, more detailed explanations can be understood easily and put into relation with the concepts.

As written above, the algorithm is based on region competition. A region is defined to be a connected component within the image. The regions have
active contours which enable the region to grow and shrink and compete with each other. If two regions are in touch, they can merge to a new region. On the other hand, it is possible that a single region splits into two or more new regions. All this is driven by minimizing an energy functional which is a model to describe the image. In each iteration of the algorithm, the difference of the energy functional for the possible moves of a contour particle is calculated, and the most improving move is taken. While each individual move is guaranteed to decrease the individual energy, this may not be true for simultaneously performed moves. This can cause contour and energy oscillations. To handle this, an oscillation detection step is added after the optimization step.

Later in this work, the important components needed to implement this algorithm are described. For more detailed information about the algorithm, see [6].

1.4 ImageJ

1.4.1 Motivation

The algorithm is intended to be used in the field of biomedical image segmentation. It should be usable to non-expert users. So the software should be easy and intuitively to work with. Since ImageJ is already widely-used in the community, and offers easy installation and usage, it was chosen to implement the algorithm for.

1.4.2 The program

ImageJ is an image processing program. It is written in Java and thus platform independent and runs on Windows, Linux and MacOS. ImageJ and its source code are freely available. ImageJ can display, edit, analyze, process, save and print 8-bit, 16-bit and 32-bit images. It can read many image formats including TIFF, GIF, JPEG, BMP, DICOM, FITS and ‘raw’. It supports ‘stacks’ (and hyperstacks), a series of images that share a single window which is comfortable to display slices of a 3D image or a series of images evolving over time. ImageJ has a graphical user interface (GUI), so all the functionality it offers can be performed in an intuitive and familiar way. ImageJ’s functionality can be expanded through the use of plugins written in Java. Plugins can add support for new file formats or they can filter or analyze images. ImageJ has a built in macro function. A macro is a simple program that automates a series of input commands. One way to create a macro in ImageJ is to use the command recorder. If running, the recorder records all the ImageJ-commands performed by the user, which then can be executed again. So, repeating tasks don’t have to be executed manually over and over again. The macros are recorded in the form of either a dedicated
macro language, in JavaScript or in Java. Macros of the two former kinds can be executed immediately within ImageJ, while these recorded in Java are used to generate a plugin performing the desired actions. The macros can also be edited manually or completely written from scratch. If a plugin in ImageJ implements the appropriate interface, the command recorder can record the parameters which the user set up for the plugin. So, the plugin can be easily used in a batch mode.

Figure 1.1: A screenshot of ImageJ
Chapter 2

Basic Data Structures

In this section, the most basic data structures the algorithm uses are presented. These are the structures to hold the image data, the segmentation progress and helper classes to address the pixels within these data structures and to iterate over them.

2.1 Point

The first and most basic data structure to be presented is the Point. The image segmentation algorithm deals with images and the pixels of images need to be addressed somehow. In the present algorithm, the pixels are addressed in two different ways: The first one is by a linear integer index. The whole image is then treated as a single linear array. The advantage of this method is that iterating over the image is very fast, since one can simply increment the index to access the next pixel that will lie next to the actual pixel in the memory, too. Furthermore, the index is a single native integer, consuming only 32bit of memory. The second method is by accessing the pixel by its coordinates within the image. On a 2D image, the pixel has an x and a y coordinate: for 3D, there is an additional z coordinate. These coordinates are stored in an integer array with the according dimension. The array again is a public field in a class called Point. The class provides methods to do vector arithmetic. The advantages of this class are that neighbors in all directions can easily be accessed, checking if a Point lies within a region is easier, and coordinates are more intuitive to interpret for humans than linear indices. The drawback is that on creating a new Point index, a new object has to be constructed, which makes it slower than an integer index. The class Point offers some basic vector arithmetic that are used for this framework, and specific overrides of hashCode(), clone(), equals() and toString(). The hashCode() is implemented by concatenating the 10-bit representations of the coordinates of the Point. So it will return distinct values for Points
within a cube of side-length $2^{10}$.

In the following, the term ‘index’ stands for the integer index, whereas ‘Point’ or ‘Point index’ is used to denote a Point index, depending on the context.

### 2.2 IntensityImage

The input image, that is, the image being segmented, is stored within the class IntensityImage. The image data itself, normalized to values between 0 and 1 is saved as a float array. The IntensityImage can be initialized by an ImagePlus. The class offers the functionality to normalize an ImagePlus and can access pixels by a Point or an integer index.

### 2.3 LabelImage

The label image stores the mapping of the pixels of the input image to the label currently given to that pixel by the segmentation algorithm. Or: it represents the segmentation of the input image. The labels of the regions are integer values. There is a designated value (0) for the background region, and a value (Integer.MAX_VALUE) for the “forbidden region”, that is, regions that the algorithm should not process. Pixels of regular regions touching pixels of another region are boundary points and are represented in the label image by the negative value of their label.

Since the labels are integer values, the label image is stored as an integer array of the same size as the input image. The data is encapsulated into the class LabelImage. The LabelImage offers methods to get and set the label (defined to be non-negative) or the value (which is negative for contour particles) at integer and Point indices. There are methods to classify the type of a pixel (inner point, boundary point) and the type of a label value (contour label, inner label).

### 2.4 IndexIterator

To deal with the integer and Point indexes, the class IndexIterator was written. It offers iterators to iterate over the images mentioned above, based on their dimensions. The return types of the iterators are either Integers (the wrapper class) or Points. Furthermore, it offers methods to convert between integer and Point indices within the given image. And there is a method to check if a Point lies within the given image. Though IndexIterator is considered to be nice since it implements the Java interfaces for Iterator and Iterable, it counts as non-performant and should not be used for time-critical applications. More on this in the chapter Masked Iterators.
2.5 Remarks

It would have been nice to let the two Images inherit from a common to offer a uniform access to the getter and setter methods. But since these methods are accessed very often and thus are time-critical, and the two basic data types are different (int vs. float), it seemed the better choice not to couple them.

Another consideration concerning the label image was, instead of integer values, to use short values. The algorithm would have benefitted from memory saving and a speedup. However, the maximal value of shorts is 32767, which can, depending on the input image, be too small, especially when during the segmentation the labels are continually incremented and not reused. This gave the motivation to write a LabelDispenser (see later) which collects unused labels and allows reusing them. Since refactoring of native data types in java is nearly impossible, the idea of short labels was abandoned and the more robust solution with integer labels was chosen.

Yet another consideration was not to use linear arrays store the data but to use ImageJ’s image format, that is, ImagePlus. A big benefit would have been if the same data had been usable for the segmentation and for the visualization. Unfortunately, the 32-bit integer image type of ImageJ 1.x is considered to be a RGB image, and ImageJ rejects to use a LUT (lookup table) for this kind of image. Since LUTs are great for the visualization of the regions, so the advantage of sharing the data fell away became negligible, the choice fell on the linear array because of fast linear access and easier programming and the drawback of the ability to use ImageJ plugins to process the data.
Chapter 3

Digital Topology

A central point of this work is the digital topology framework. Digital topology (DT) deals with properties and features of digital images that correspond to topological properties (e.g., connectedness) or topological features (e.g., boundaries) of objects [1]. Applications of digital topology for the present segmentation algorithm include determining the connected components within an image, detecting the boundaries of these components, or finding neighbors of a point. Since there was no framework available for Java to deal with the issues mentioned above in a generic way (w.r.t dimension and type of connectivity), such a framework was implemented in order to be used with the algorithm. In the following sections, the theoretical background is set up at first, then the implementation is described, and finally, the framework is compared to an existing version written in C++ for ITK.

3.1 Theoretical part

3.1.1 Connectivity

Connectedness is the property that pixels have if they are connected to each other. There are different ways how pixels can be connected; hence, there are also different types of connectedness. A particular type of connectedness is often called connectivity. For a particular dimension, different connectivities can be defined. They differ in the number of neighbors a pixel can have in this connectivity. The neighbors of a pixel are all the pixels that are connected to the former pixel. In 2D, there are two reasonable cases for different connectivities: The first one is the 4-connectivity: The 4 pixels that share a common edge with the central pixel are considered to be in the neighborhood. The second one is the 8-connectedness: The neighborhood is then defined by all the pixels which share at least a corner with the central pixel.
In 3D, the connectivities can be deduced accordingly. A 6-connectivity is defined by the 6 voxels that share a face with the central voxel. In the case of 16-connectivity the neighboring voxels share one or more edges with the central voxel. And in 28-connectivity, at least a corner is shared between them.

This leads to another possibility to discern different connectivities. Instead of giving just the number of neighbors to define the connectivity, one can define the connectivity by a pair of numbers. The first one is the dimension of the space. And the other one is the dimension of the highest dimensional primitive that has to be shared between two pixels to be considered to be connected. The primitives are a point for dimension 0 (sharing a corner), a line for dimension 1 (sharing an edge) and a plane for dimension 3 (sharing a face). For example, the 4-connectivity would be denoted by (2,1), since it is a 2 dimensional connectivity, and the primitive shared between neighbors are of type edge (dimension 1).

Having only considered single pixels thus far, now the issue of objects consisting of multiple pixels needs to be addressed. Two objects on a digital image are connected if at least one pixel of the first object is connected to a pixel of the second object, with respect to the defined connectivity. Considering the objects as the foreground and the rest of the image as the background, it emerges that different connectivities must be used for the background and the foreground [10]. This is due to Jordan theorem [5]. The Jordan theorem states that a simple (i.e. non-overlapping) closed surface divides the space in two parts, namely an interior and an exterior one. In a discrete space, depending on how the connectivities are chosen for the foreground and the background, this theorem can be invalidated when using an invalid connectivity pair for the FG and BG.

Therefore, compatible connectivities must be chosen for the foreground and the background. It has been proven [9] that either for the foreground or for the background, the connectivity of type (d, d-1) must be used. The other connectivity is usually chosen to be (d, 0) (i.e. the “full” neighborhood).
3.1.2 Topological number

During the iterations of the segmentation algorithm, points are removed from and added to regions in order to minimize the energy. Not every point can be removed or added without changing the topology of the object. For example, if a point is the only one connecting two parts of the object, then removing it would split the object into two. A point that can be added or removed without changing the topology of neither the foreground nor the background is called a simple point. An efficient way to determine if a point is a simple point is to determine the topological numbers of that point. The topological number of a point is defined by the number of connected components within a certain geodesic neighborhood. The geodesic neighborhood of a point is recursively defined by ((formula)) A point is simple if the topological numbers for both the foreground and the background equal 1.

3.2 Implementation

In this section, the technical details and the implementation of the digital topology framework built for this thesis are presented and how the components are related to the segmentation algorithm.

3.2.1 Connectivity

The class Connectivity is used to define the connectivities used by the segmentation algorithm. It provides functionality to get the neighbors to a particular point, which is used for instance to let the active contour grow. It is also the basic component for determining the connected components within an image and detecting the contour of a region.

The class Connectivity offers a constructor which takes the type for the connectivity as input. It initializes two arrays which contain the neighbors for the corresponding connectivity type. One array is of an integer type and represents the neighbors by the number they would obtain if one enumerated the complete neighborhood in a canonical way. The other array saves the neighbors as a Point-offset, that is, representing the neighbors by the distance vector they have to the central point. This duality was chosen due to efficiency reasons, such that neighborhood queries for both Point offsets and integer offsets can be answered without conversions of the arguments.

((image arrays als index und als point))

The class offers a method to get a compatible background connectivity to the actual one. It follows the rule defined in the theoretical part above.
Another method gets the neighborhood connectivity for this connectivity. This is used for the computation of the topological numbers.

Furthermore, there are methods to check if an offset is in the neighborhood, and if two Points are connected within this connectivity. For this, the distance vector of the two Points is calculated, which is then an offset that can be looked up in the arrays.

The class offers Iterators, such that one can iterate over the offsets, either in Point offset or in integer offset form. An additional Iterator takes a point as input and iterates over all neighbors of this point by adding the offsets to it.

```java
Connectivity fg = new Connectivity(3, 2);
Connectivity bg = fg.getComplementaryConnectivity();
for (Point p : fg.iterateNeighbors(center)) {
    // iterating over neighbors
}
```

### 3.2.2 UnitCubeCCCounter

For the calculation of the topological numbers of a point, one has to count the number of the connected components in the corresponding geodesic neighborhood. The class UnitCubeCCCounter serves this purpose. The constructor of the class takes the desired connectivity as input and then identifies the neighborhood connectivity which corresponds to the geodesic neighborhood. To specify the point that is of interest, a subimage of the input image is passed which consists of the point and its full neighborhood. In this subimage, the number of connected components is counted with a flood fill algorithm. For this, a first seed is taken that is in the neighborhood of the central point and belongs to the foreground/object. All foreground pixels that are connected within the neighborhood connectivity are then marked as visited. These points (if there are any) are the first connected component. To find additional connected components, the process is repeated until there are no new, non-visited seeds. Finally, the number of connected components is returned. During this procedure, the neighborhood relation between two offsets has to be tested in each iteration. Thus, for all possible combinations of offsets this relation is precalculated and stored in an array for fast access.

```java
UnitCubeCCCounter c = new UnitCubeCCCounter(connFG);
c.setImage(subImage);
int n = c.connectedComponents();
```

### 3.2.3 TopologicalNumberImageFunction

One remaining problem is that the UnitCubeCCCounter, according to theory (digital topology is strictly restricted to binary images, only processes
binary images consisting of foreground and background. However, the label image of the present segmentation algorithm is a multi-labeled one, thus non-binary.

In order to eliminate this shortcoming, the class TopologicalNumberImageFunction is introduced. The TopologicalNumberImageFunction is initialized with the foreground and the background connectivity and the label image. To compute the topological numbers for a particular point, the function reads out the complete neighborhood of that point into a subimage. For each different label present in the neighborhood, it converts the subimage into a binary representation by changing the current label to a foreground value and all other labels to a background value. Now the UnitCubeCCCounter can compute the topological number for the foreground. To compute topological number for the background, the subimage is inverted [not the middle; implicitly out due to strict neighborhood] and a UnitCubeCCCounter initialized with the background connectivity is called. The results are saved into a list together with the corresponding label value. The list with the foreground and background topological numbers of all labels present in the neighborhood of the point is then returned. The point is simple, if the foreground and background numbers of all labels are 1.

3.2.4 Result

Before going to the flood fill algorithm in the next section, let’s first compare the framework written for this work to an existing solution.

An existing implementation of the digital topology framework by [8] is written in C++ and makes heavily use of template metaprogramming, so many calculation takes place at compile time. Since there is no comparable metaprogramming in Java, the connectivities are computed at runtime during the construction of the object. The C++ version uses singletons of the connectivities. Since the different connectivities in the framework are “passed” by template parameters, that makes sense. In the present Java version, the connectivities are passed as arguments anyway, so there is no need for a singleton pattern, since typically one constructs the connectivity only once. Differently, the methods to get the complementary (background, foreground) and the neighborhood connectivity behaves. These methods may be called more than once, so they implement a pseudo-singleton, by returning a stored instance of the corresponding connectivities. Just for the case, a pure singleton pattern was implemented. The class Connectivity has a static field of a List or Arrays of Connectivities. The indexes in the list stand for the dimension the connectivity has, and the length of the corresponding array is the number of maximal possible connectivities in this dimension. So it is possible to fast retrieve singletons for arbitrary number dimensions and different connectivities.
3.3 Flood Fill

3.3.1 Introduction

In several places of the algorithm, it is necessary to relabel regions. This is the case in topological changes, when a region splits into two or more regions or when regions fuse. In either case, the label image needs to be updated. Since the label image has the form of an image, relabeling can be achieved by flood fill. Flood fill is an algorithm that determines the area connected to a given node in a multi-dimensional array. The algorithm takes three arguments: a start node, the value to be replaced and the replacement value. The algorithm searches for all nodes in the array which are connected to the start node by a path of the target value, and switches them to the replacement value.

ImageJ has a built in flood fill function, but there are reasons why it is not possible to make use of it. Firstly, the ImageJ implementation only supports 4 and 8 connected filling in two dimensions. But since the framework is required to be generic with respect to the dimensions, that is, should work in three or higher dimensions, and defines its own connectivities, the built-in flood fill is not an option. Secondly, the contour pixels in the label image have a value that is different from the label of the region which they belong to in order to identify them as a contour pixel. The flood fill algorithm needs to be aware of this circumstance.

For the latter reason, the flood fill function for this framework allows for specifying multiple values to be replaced. By specifying the value for a label and the value of the corresponding contour label, the search for neighboring pixels does no longer discriminate between contour and label and thus the whole region is found.

In the next sections, the flood fill algorithm and its components are described in detail.
3.3.2 Multiple Threshold function

The ability to specify multiple values to be replaced during the flood fill is provided by the class MultipleThresholdFunction. It allows adding an arbitrary number of values, which are later compared to the value of a pixel. If one of the added values match the pixel value, the function returns true. Inspired by the corresponding function in ITK, the MultipleThresholdFunction not only allows adding discrete values to be checked, but also intervals of values. An interval is added by specifying a lower and an upper bound of the interval. The comparison to the pixel value is then accomplished by checking if the value is in at least one of the defined intervals.

```java
MultipleThresholdImageFunction foo; Foo.AddThresholdBetween(lower, upper); boolean b = Foo.EvaluateForValue(pixelValue);
```

For the present framework, only integer labels are being relabeled so that a limitation to integer thresholds would suffice. But to guarantee the reusability of the components, the threshold function is implemented to work with (native) doubles. Though experiments to implement the class parameterized to work with any kind of comparable data types succeeded, the design seemed to be too complicated and the gain in simplicity, readability, and even in performance clearly outweighed the gain in generality.

The class MultipleThresholdFunction is extended to the abstract class MultipleThresholdImageFunction. It offers the abstract methods to return the result of the evaluation method by not giving the value but the index of an image source. Realizing the latter class allows to specify an arbitrary type of image source to read the value from. This comes into play when the source of the value to be compared differs from the destination to write to in the flood fill algorithm.

3.3.3 Application

The flood fill algorithm is initialized with a MultipleThresholdImageFunction, in this particular case, with a MultipleThresholdLabelImageFunction, a connectivity, and a seed point. Due to the fact that the border pixels of the label image are set to the forbidden label, no explicit boundary checks have to be performed. Flood filling is then executed and all points found are saved in a set. Accessing the points found is done after the initialization by an iterator.

```java
FloodFill ff =
new FloodFill(connectivity, thresholdFunction, seed);
for(Point p : ff) {
// do sth with the points found 
}
```
3.3.4 Improvements

Depending on the image to be segmented, the flood filling process may take a considerably long time of one iteration (in the order of 30%). This typically happens if the image consists of many little objects and the label image is initialized by a bounding box. When the bounding box shrinks, the little objects will get cut off, inducing a relabeling of both the region of the little object and the (potentially big) remaining bounding box. This may happen multiple times for each cut off region during a single iteration of the segmentation algorithm. One solution would be to speed up the flood fill algorithm, for example by using a scanline fill algorithm [4]. Another possibility would be to implement a heuristic to choose, in case of a split, not to relabel a (assumingly bigger) side of a split and to process only the other (smaller) side(s) of the split. In case of merges, only the label with the smaller number of pixels could be adjusted to the label of the region with the higher number of pixels. Difficulties may arise when adapting the energy statistics.
Chapter 4

Masked Iterators

4.1 Introduction

For the present work, different iterators had to be implemented, since their functionality was not provided by ImageJ. An example for an iterator needed for this work, able to iterate over spherical regions, is as follows:

To deal with noise in the input image, a kind of regularization has to be applied to the algorithm. This is achieved by adding a regularization term to the energy. One possibility for the regularization is to choose contour length regularization. Exploiting the fact that curvature-regularization is equivalent to contour-length regularization, one can use an approximation of the local curvature. [7] proposed a method that calculates the regularization flow within a (hyper-) sphere around a point. So, for this method, one needs to be able to iterate over a spherical region. Another use of the spherical iterator lies in the energy functional for the piecewise smooth image model. For this, local statistics within a sphere around a point have to be calculated.

Dynamically checking, if a point lies within a sphere around another point can be expansive. Therefore, for this work, masked iterators are used. Masked iterators precalculate the calculations and store it in a binary array, called mask. In this way, the geometrical calculations are reduced to a memory look-up.

In the following sections, the masked iterator, able to work in arbitrary dimensions, and the components needed for it are described.
4.2 Technical part

4.2.1 Rectangular Region

A rectangular region is defined by its dimensions (width, height, ...) and the offset from the input image. The offset is measured from the upper left corner of the input image to the upper left corner of the region. If a point close to the image boundary is chosen, the region to iterate over may not lie completely within the boundaries of the input image. Accesses to data in such a region may lead to wrong lookups in the input image or an Exception. Therefore, the region has to be cropped in such a way that the region doesn’t exceed the boundaries anymore. If the region lies beyond the lower boundary, i.e., the offset of the region is negative, the offset of the region is set to zero and the size reduced accordingly. If a point of the region exceeds only the upper boundary, the offset of the region stays the same and only its size gets adjusted. After cropping it is safe to iterate over the region.

To iterate over the region, one could iterate a linear index over the size of the cropped region, convert the index to a Point (linear indices can’t be added due to the different coordinate systems) and add it to the offset of the region, to get the corresponding index in the input image. But in that way, many unnecessary conversions are done, that slows down iterating.

Using the fact that the iterator is usually most of the time just incrementing in the lowest dimension, and these increments can be added directly in the coordinates of the region and the input image, one can optimize the iterating process. Instead of a complete coordinate transformation of a point for each iteration step, just an increment and a boundary check in the lowest dimension has to be done. Only when the index exceeds the boundary of the region in one dimension, a coordinate transformation has to be triggered, and only for the next higher dimension.

In a first attempt, this iterator was implemented by using the Iterator interface of Java, allowing uniform access to the iteration process. Since this interface requires using autoboxed Integers, it slowed down the iteration hugely. After removing the interface (though using the same method names for the access) and using native integers, a speedup by about 20% was achieved.

```java
RegionIterator it =
    new RegionIterator(input, region, offset);
while(it.hasNext()) {
    int idx = it.next();
}
```
4.3. Discussion

4.2.2 Masked iterators

For the use of masked iterators, one need to be able to iterate over a rectangular region, which is fulfilled by the region iterator described above. An additional iterator has to iterate over the cropped region, but in the coordinates of the uncropped region. This can be achieved by cropping the region, and feeding in the uncropped and the cropped region into a normal region iterator.

If for the region, the dimensions of a mask are used, one can simultaneously iterate over the both region iterators, where the first iterator returns the index for the input image, and the second iterator returns the index for the mask, corresponding to the index in the input image. Checking if a point in the input image is masked by the mask is then straight-forward. To allow an efficient access to the masked iterator in the usual way with hasNext() and next(), one has to attend to some details. In contrast to rectangular regions, it is not a priori clear, if there is a next value. In the worst case, one has to check the whole mask. To not render this calculation useless, the value is stored for the next invocation of the next() method. And sequent calls to hasNext(), without calling next() in between, has to be considered.

RegionIterator it;
Mask mask = new SphereMask(rad, dim);
it = new RegionIteratorSphere(mask, inputSize);
while (it.hasNext()) {
    int idx = it.next();
}

4.3 Discussion

Iterators for iterating over rectangular and arbitrary masked regions have been implemented. They are highly optimized, though the java Iterator interface had to be abandoned if favor of performance. However, uniform access is granted by using Java naming conventions.
Chapter 5

Software Architecture

The main topic of the present work was to translate the original code to Java, and to design it in such a way, that it uses Java specific features and conveniences and renders more reusable, expandable and maintainable.

The translation of the core mechanics of the algorithm is done in such a way, that it is as close as possible to the original algorithm. This is done for several reasons. A major reason was that during the time of this work, the algorithm was still under development. Parts of it were removed or replaced, new features were added, and bugs were fixed. The adaptions of these changes were expected to be much easier to make if the codes resemble each other. Another intention was, that after this bachelor thesis being finished, for the author of the original algorithm, understanding and maintaining the code would be easier.

In the gross, the technical details of the algorithm are the same as in the original work. Some minor adaptions or improvements were made, see section Technical changes. For detailed description of the algorithm, see [6]. Though within this work, no own chapter is dedicated to the implementation and the mechanics of the algorithm, still it was put a lot of effort into getting the algorithm run in the correct way.

The big differences to the original algorithm lie in structure of the program, in a software architectural way. At the start of this bachelor thesis, the code of the original algorithm was in a pre-refactored stage. As it happens to be, the big part of the produced source code was located in one big source file, in a single class. Some methods tended to be long and sometimes cluttered.

So, for the current work, where it was possible and meaningful, components of the algorithms were laid out into separate methods and classes. The overall design now mirrors the way of utilization of the algorithm and let them being expanded in the black-box manner that is one of the benefits of this algorithm.
A detailed description of the design can be read in the following section.

5.1 Structure

Since the algorithm can be fed with many different kinds of energy functionals, the calculations of the energies were laid out into Energy classes. Each Energy class has a method to calculate the energy difference when changing a contour particle (or a background pixel) to a new value. For these energies, where it is advantageous, the result of a merging criterion is returned, too. Besides the data dependent energies, the regularization and merging criterions are laid out into separate classes. Since the oscillation detected implemented originally is deprecated meanwhile, the oscillation detection is laid out too, to make it later on easy to put the new one into the algorithm. Additionally, depending on the application or the used energy or parameters, different oscillation detectors may be favored. (Additional oscillation detection was written for this work, too).

Further design specific changes can be seen in Figure ?? on page ??

5.2 Technical changes

Technical changes to the algorithm were done as the need arises. To speed up the algorithm if the label image is initialized with a bounding box, a parameter shrinkFirst was introduced. If shrinkFirst is set to true, it causes the algorithm to only shrink in a first stage. Energy calculations for growing into the background or to compete with other regions can be omitted. Depending on the other settings of the algorithm, it is about twice as fast for the first few iterations. After the convergence of the shrinking stage, the algorithm continues normally.

One problem arose when initializing label images with bubbles, and the balloon force was allowed. The balloon force introduces a hysteresis into the energy functional, because growing into the background is boosted, while shrinking the same way back is calculated conservatively. Considering a particular grow step of a contour particle, which would not be favored by the energy minimizer without the balloon force, it will likely undo this step in the next iteration. Consider now two opposite contour particles that grow to each other in that first step. The growing shall be only allowed, because it is boosted by the balloon force. If the newly added contour pixels then touch, a next growing will not be boosted by the balloon force, rendering a competing step unfavorable, and they have no chance to compete and check, if a merging would be beneficial (since they will instead undo their previous step). To overcome this, a field newlyCreated has been introduced to the contour particles and it unset after one iteration. Meanwhile, if a
5.3 Initializers

A crucial point to image segmentation with region competition is the initialization. If the initialization is chosen inappropriately, the segmentation can take significantly more time than it would with an appropriate one, or it can even – subjectively – fail. Under this consideration, there are separate classes which are in charge of initializing the label image as an entry point to the segmentation. The classes all inherit from an abstract super class called Initializer. Two types of initializers are distinguished: The ones inheriting directly from Initializer and these inheriting from DataDrivenInitializer. While the first type initializes the label image without knowing the input image, the data driven initializers initialize the label image depending on the input image. Examples for the first type are bounding boxes or randomly placed bubbles, whereas the latter one could consist of thresholds of the input image.

For the current word, the following initializers were implemented: ZeroInitializer which fills the whole label image with the background label. BoxInitializer which fills the label image with a box, the side lengths in a specified ratio to the image dimensions. BubbleInitializer which fills the label image with spheres, with a given radius and padding between the bubbles. MaximumFinderInitializer which first creates a smoothed version of the input image, searches local maxima and either places bubbles in the maxima or flood fills the maxima within a specified range. ThresholdWithHolesInitializer which thresholds the input image and places holes at the local minima of the input image.

For the BoxInitializer, the RegionIterator of this work is used. For the BubbleInitializer, a BubbleDrawer class was written, which uses the sphere masked region iterator. For the second version, the FloodFill class was used. For MaximumFinderInitializer, the MaximumFinder (local maxima) plugin of ImageJ had to be adapted to work with 3D data. ThresholdWithHolesInitializer works with the ImageJ threshold facilities and the MaximumFinder mentioned before.
6.1 Input Dialog

6.1.1 Introduction
In order to get the best results out of the algorithm, some parameters should be adjusted before starting it. Apart from the choice of the energy functional and the choice of the initialization, there may be more specific parameters one wants to set. To make this as intuitive as possible, a graphical input dialog is the entry point for the user to work with the algorithm.

6.1.2 Motivation
ImageJ provides a class GenericDialog that allows generating input dialogs in an easy way. It provides methods to put together an input dialog consisting of basic input elements such as numeric and string fields, checkboxes and drop-down lists. If the values are read out by the methods provided by GenericDialog, the macro recorder of ImageJ records the values as input parameters for the plugin. If the macro is then run, the input dialog is not actually displayed to the user, but the input parameters are directly available by the read-out methods mentioned above. So the developer can use the same input dialog for both user input values and macro values.

6.1.3 Implementation
Due to these benefits, the input dialog for the present plugin is based on GenericDialog. It encapsulates an instance of GenericDialog, and provides methods to build up the dialog, to display it, and to read out all the input values into a settings structure.

Due to the simplicity of GenericDialog, there are some shortcomings to build more complex input dialogs, which the present input dialog tries to over-
come. For instance, if there are many different input elements, the dialog can get quite large and disorganized since there is no option to somehow hide additional parameters or restructure the dialog dynamically. Since the LayoutManager of the GenericDialog is known to be a GridBagLayout and the add-methods of the Container class are available, it is possible to add additional elements to the dialog – albeit only to a certain extent. For the present dialog, additional buttons were placed right to dropdown lists. These buttons open further GenericDialogs that specify parameters concerning the item selected in the dropdown list. These parameters then do not have to be shown in the main dialog, but are encapsulated in their own input dialog.

Other improvements are drag-and-drop ability for text areas, buttons to open a FileDialog and a MouseWheelListener for numeric fields, which allows changing the values when scrolling the mouse wheel over the fields.

6.1.4 Results

The present input dialog looks as follows: First there are drop-down lists to choose the energy functional, the regularization, and the type of initialization. Each list has an option to open a further dialog to set additional parameters for the concerning choice. Then options follow for topological constraints, and general parameters that are common to all choices. Next, there are two text fields, where the path to the input file, and to the initialization file, respectively, can be entered. The fields support drag-and-drop from the file system. Placed below are buttons to open a file opener dialog for the two files. Then, as a last input method, images already opened in ImageJ can be chosen. On the last line, there are options concerning the output and display of the segmentation.

6.1.5 Conclusion

Though GenericDialog is very convenient to quickly build up small input dialogs, it quickly reaches its limits as soon as the dialogs become more complex. While the automatic macro record ability is a benefit of GenericDialog, it forces the programmer to read out the values in the same order that the input elements were placed in the dialog. This is somewhat cumbersome when rearranging the elements and can lead to bugs that are hard to find (since two numeric values may be switched). Another drawback is that the enhancements made to the actual input dialog depend on non-high-level interfaces and thus on the implementation of GenericDialog (e.g. the type of elements and the layout manager). So it is not guaranteed that an adapted input dialog will work smoothly with every update of IJ. A solution to this is to simply make a duplicate of the GenericDialog and to rely on this copy rather than on the ImageJ GenericDialog itself. In consideration
of these points, it may be beneficial to build the input dialog independently of GenericDialog and implement the macro record ability by hand.

6.2 Visualization

The plugin is able to visualize the current progress of the segmentation. The user has the choice to only see the last iteration result, or if it would like to see all previous results, showed in a stack. After every iteration, a new slice is added to the stack. In the case of 3D images, the result is displayed in a hyperstack and the user can browse through the time and the z-axis. The label image is visualized by 8-bit (to save memory) image with a 3-3-2 RGB look up table. So, the different regions can easily be distinguished by the different colors assigned. The final segmentation (and the history) for 3D images can be showed (and animated) with the ImageJ plugin 3D Viewer.
6.2. Visualization

Figure 6.2: A screenshot of the stack, visualizing the progress of the segmentation
Chapter 7

Results

In this section, the results of image segmentation with the new algorithm are presented. The images taken are mainly the same as in [6], to facilitate the comparison. Not always the same initializations as in the paper could be used, so the results cannot be compared 1:1.

Though for the comparison in this chapter, exactly the same initiations and parameters were chosen for both algorithms. Time was measured by console outputs.

The algorithm in ImageJ was run on Intel Core i7-2600K @ 3.40 GHz with 8 GB of RAM on a Windows 7 Professional 64-bit with Java build 1.6.0_25. The ITK version for 2D images was compiled under the same system with Visual Studio 10, the version for 3D ran on the same machine but under Ubuntu 12.04 (64-bit).

The first image is an artificial image of 5 overlapping disks with different, but constant intensities and added noise. It is initialized by 6x6 bubbles. The energy chosen to segment is the piecewise constant, the regularization radius is set to 4. Other parameters can be seen in Table 7.1 on page 26.

The algorithm in Java converges after 71 iterations, the C++ after 69. The result shows that the segmentation is identical for both algorithms. Though, during the iteration processes, some minor differences emergent. While the Java version takes 0.3 to finish, the C++ needs 1.2 seconds. The big difference in runtime seems to be due to the compilation with Visual Studio. The same image on Ubuntu in the virtual machine took only 0.5 seconds.

All other results are identical, too, or there are some minor differences of some pixels. These can come from numerical issues, or especially for PS energy, different oscillation detection.

The big difference in the number of iteration in the image Icecream PS 2-D, 130x130 comes from the new oscillation detection. While the original
7.1. Outlook

Some further work that could be done in this subject area: Of course, following the software design, implementing more energies can be done. The C++ version already supports computation on the GPU. This would be possible for ImageJ too, but personally I think, such a low level development doesn't match the Java philosophy. But even if no GPU would be used, a parallelization using threads would be perfectly possible and generate a speed-up. ImageJ version 2 is about to finish and may be released at the end of the year. Version 2 has its own versions of iterators, which may be faster than the actual implementation, so changing to them could be beneficial. Additionally, there are new classes for image data that could be included more seamlessly into the algorithm.

Table 7.1: Results of the two algorithm versions

<table>
<thead>
<tr>
<th>Image</th>
<th>Initialization</th>
<th>Energy</th>
<th>Parameters</th>
<th>Regularization</th>
<th>Iterations</th>
<th>CPU time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Icecream PC 2-D, 130 x 130</td>
<td>6 x 6 bubbles</td>
<td>PC</td>
<td>t=0.04</td>
<td>t=0.2, R=4</td>
<td>71</td>
<td>69</td>
</tr>
<tr>
<td>Icecream PC 2-D, 410 x 410</td>
<td>8 x 8 bubbles</td>
<td>PC</td>
<td>t=0.04</td>
<td>t=0.2, R=8</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>Zebrafish em: bryo nuclei 3-D, 512 x 512 x 39</td>
<td>Local maxima</td>
<td>PC</td>
<td>t=0.04</td>
<td>t=0, R=2</td>
<td>57</td>
<td>57</td>
</tr>
<tr>
<td>Bird, 481 x 32</td>
<td>18 x 12 bubbles</td>
<td>PC</td>
<td>t=0.2</td>
<td>t=0.3, R=8</td>
<td>87</td>
<td>87</td>
</tr>
<tr>
<td>Icecream PS 2-D, 130 x 130</td>
<td>5 x 5 bubbles</td>
<td>PS</td>
<td>t=0.04, b=0.05, r=8</td>
<td>t=0.2, R=4</td>
<td>48</td>
<td>108</td>
</tr>
<tr>
<td>Zebrafish em: bryo germ cells 3-D, 188 x 165 x 30</td>
<td>Bounding box</td>
<td>PS</td>
<td>t=0.08, b=0.005, r=4</td>
<td>R=4</td>
<td>150</td>
<td>-</td>
</tr>
<tr>
<td>Cloud 2-D, 481 x 32</td>
<td>18 x 12 bubbles</td>
<td>PS</td>
<td>t=0.2, b=0.1, r=30</td>
<td>t=0.2, R=8</td>
<td>180</td>
<td>-</td>
</tr>
</tbody>
</table>

oscillation detection long fails to detect the oscillation, the new one succeeds much earlier. The final result is the same except for a single pixel, which come from oscillation.

Figure 7.1: Original and segmented image
Chapter 8

Conclusion

The given image segmentation algorithm was successfully ported to Java. By its graphical user interface, the plugin can be handled in an intuitive way. The results produced by the ported algorithm are mostly the same or at least very similar to these of the original version. Some minor improvements were made to the algorithm that allowed better results in some cases. The performance is comparable to the C++ version. The components implemented for the algorithm are well structured and reusable, especially the digital topology framework, that is novel to Java.


