Effective Quantification of the Paper Surface 3D Structure

Preface

This document describes the results of Svein Fidjestøl's diploma work and in-depth study of image processing techniques in relation to analysis of the three-dimensional surface structure of paper. The diploma work was conducted during the Spring semester 2005.

The title of the thesis is "Effective Quantification of the Paper Surface 3D Structure". The project was given by Gary Chinga, research scientist at PFI[1] specializing in the field of surface structure. The project description as given from Gary Chinga is cited in its entirety on the front cover of this report.

I would like to thank Gary Chinga as well as my academic advisor, Richard E. Blake (IDI[2], NTNU [3]). They have been supportive with their advice and provision of comments throughout the semester.

Trondheim, June 16, 2005

Svein Fidjestøl

[1]PFI: Paper and Fibre Research Institute
[2]IDI: Department of Computer and Information Science
[3]NTNU: Norwegian University of Science and Technology

Summary

This thesis covers the topic of image processing in relation to the segmentation and analysis of pores protruding the surface in the three dimensional surface structure of paper. The successful analysis of pores is related to a greater goal of relating such an analysis to the perceived quality of the surface of a paper sample.

The first part of the thesis gives an introduction to the context of image processing in relation to paper research. Also, an overview of the image processing framework used for image processing plugin development, ImageJ, is provided, together with the current status of ImageJ plugins for surface characterization.

The second part of the thesis gives an overview of an envisioned future paper quality assessment system. The quality assesment system described consists of six phases, three of which are treated in this thesis. These are the Image Processing phase, the Modeling phase, and the Measurement phase. The Image Processing phase is further divided into three subphases. These are the Error Correction subphase, the Pore Extraction subphase, and the Segmentation phase. Together with the description of the phases of the system, techniques are presented that are relevant to the phase currently being described.

The third part of the thesis covers the development of new plugins for surface characterization within the ImageJ framework[4]. Examples are given and evaluated to show the usage and results of each plugin, and each plugin is related to a specific part of the quality assesment system. Also, a tutorial covering use of several plugins in sequence is presented. The parts of the system receiving the most attention in relation to plugin development are segmentation and modeling.

[4]A CD-ROM is included with this thesis, containing ImageJ version 1.34n, the developed plugins including source code, and example images

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This project concerns three-dimensional surface analysis in the context of paper research. The main topic is pores occuring in the paper surface, and segmentation and analysis of these. These pores are also discussed in relation to their consequences in the printing process. More specifically, the problem of surface areas with large pores or clusters of smaller pores that still have spots without ink after printing is discussed. For simplification purposes, this is here called the *missing dot* problem.

First an overview of the current status of pore analysis is given, both in terms of theoretical background and the status of current software related to this topic. A framework with the purpose of describing a system for estimating missing dots is presented, and also the image processing, modeling and measurement parts of this system are presented.

With this background in mind, software implementations for these parts of the system are presented together with their corresponding results. The example images used have been acquired with LLP (Lehrman Laser Profilometry), which is a relatively convenient technique for surface image acquisition, and presumably accurate enough for the present purposes.

1.1 Underlying Fiber Structure

At the core of surface analysis in paper research is examination of the relationships between the underlying fiber structures at the microscopic level and the high-level qualities of the produced material. Automated and semi-automated analysis of these relationships with the help of image processing software would be very useful in this context. Gradually, during the last few decades, the declining price and exponentially increasing power of workstation computers capable of powerful image processing arrived at the point where the use of image processing techniques from the field of computer science has become helpful for the pulp and paper research community. As of today, graphic workstations are often more than powerful enough for processing of the (sometimes quite volumous) amounts of data samples acquired in pulp and paper research contexts.

1.2 Missing Dot Problem

Several types of analysis are available to aid in the analysis of missing dots. Data sets of the type used in this report have been prepared and used for other reports at PFI, among others Chinga's paper [7]. These data sets have been obtained through the use of the Lehrman Laser Profilometry (LLP) method. In the context of analysis of transferred vs. missing dots, LLP has apparently traditionally not been the method of choice, though it has other advantages over the usual methods, for instance speed of image acquisition.

There are many other techniques in common use. The use of Scanning Electron Microscope (SEM) has been convenient when analyzing printed surfaces in terms of transferred and missing dots. An example image acquired with SEM may be useful in visualizing the missing dot problem. Such an example image is shown in Figure 1-1.

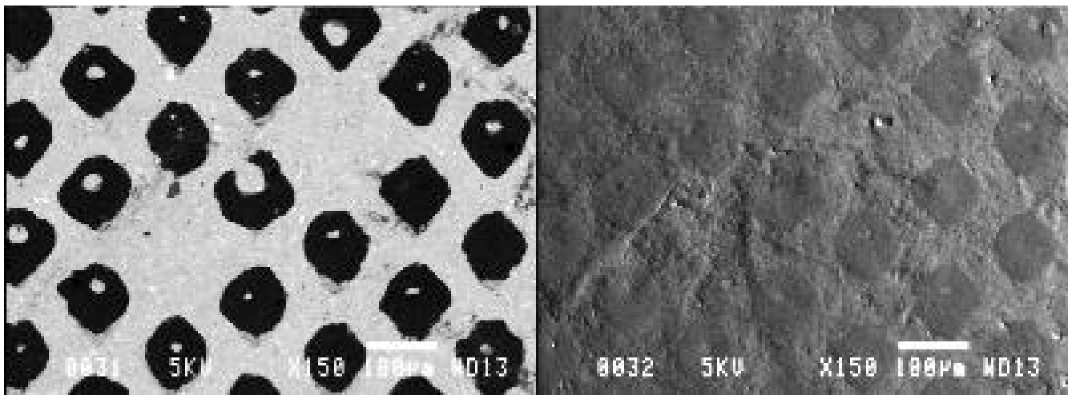


Figure 1-1: Missing Dot Example, SEM Backscatter Images [8]

1.3 Focus

The focus of the present diploma work will be on image processing methods which can be useful in assessing missing dots that are due to pores protruding the paper surface. A goal is to enable easier quantification of missing dots prior to actual printing as this is a useful measure for the quality of the paper. Ideally, it should be possible to process an LLP image to such a degree that the percentage and layout of missing dots in the sample, after printing, can be predetermined. The processing should be as automated as possible, though operator intervention is desirable if it is shown to considerably increase the accuracy of the estimated percentage and layout of missing dots.

The benefits of a system offering semi- or fully-automated missing dot estimates directly from an LLP sample should be obvious: As the percentage and layout of missing dots is a large component in determining final print quality of a paper sample, an automated system which can perform this analysis with minimal operator intervention would be of highly useful. However, it is important that the estimates generated by the analysis are close approximations to real-world measurements after printing. The quality of the estimates would be assessed by comparing them with empirical measurements.

A goal in a *broader* context is, of course, to apply such results to the paper production process itself. The benefits that can ultimately be gained from a more precise automated image analysis technique for paper quality evaluation are many. For instance, one can imagine the production of paper with ever improved properties, or even production of paper with the same properties as today, but with lower production cost. According to Holmstad [14], the crucial elements with regards to product improvement will be *"to have a quantitative measure of the detailed structure characteristics. Without quantitative measures there are no means for process control or objective assessment of the process improvements."* The present diploma work is, as already presented, a work in the chain of research that aims at improving the possibility of obtaining a quantitative measure, more specifically surface quality characterization of paper due to pores protruding the surface.

1.4 Computer Science and Image Processing in Pulp and Paper Research

It is the purpose of this thesis to bring forward the collaborative efforts that are present between PFI and IDI, NTNU. These collaborative efforts attempt to combine new imaging techniques used by PFI researchers with new developments in image processing. A partial goal of the collaborative efforts is to develop tools helpful in analyzing newly acquired sets of images at PFI in new and interesting ways.

An observation that should be made is that while the author (as well as the other IDI students doing their thesis work in collaboration with PFI) has knowledge of software development and image processing, he (and the other IDI students) has little knowledge of e.g. microscopy and other paper analysis techniques used at PFI. It could be that some aspects of the present thesis seem somewhat out of place in the context of paper research, however, it is the view of the author that new approaches to paper surface analysis could become apparent by viewing the acquired images from new angles. Therefore it is the goal that new techniques will be presented and implemented throughout this thesis that can lead to better results in surface analysis (paper surface as well as possible other types of surfaces).

1.4.1 X-Ray Microtomography and Surface Analysis

In recent years there has been a move to include more techniques from the field of Image Processing to the pulp and paper research community. Following are some of the present focus areas for image processing techniques at PFI.

As mentioned in Chapter 1.4, there is an ongoing collaboration between PFI and IDI, NTNU. Most of the collaborative projects have involved analysis of 3D X-Ray microtomography images obtained through the use of a synchrotron at ESRF (European Synchrotron Radiation Facility). It should be noted that the data sets obtained through the synchrotron are of extremely high detail and are therefore large and computationally expensive to analyze even with workstation computers. One of the goals of the X-ray Microtomography project is that of simplifying the data sets while still retaining the important bits of data. It should be noted that the data volumes of images used in this thesis are much smaller than for those including microtomography images.

In the future it might be possible to perform more accurate pore analysis by utilizing data from X-ray microtomography images. However, at the present stage it is unknown if this will be neccessary in the context of pore segmentation and analysis, and its relation to the missing dot problem.

Chapter 2

Background

2.1 History

Following in the steps of Holmstad's work (a doctoral thesis in collaboration with PFI), new methods of measurement lead to a wealth of possibilities for the microscopy and image analysis of normal and also cross-sectional images for characterization of the paper structure. Holmstad [14] divides the possibilities into three distinct categories:

1. Observation of paper structure features and quality problem detection by visual inspection
2. Image analysis for benchmarking and product development
3. Image analysis for explanation of physical behavior

It is clear that the present work will mostly be concerned with the first and third category.

2.2 Applications

An interesting aspect of the topic at hand is that image processing of the type described in this report has a wide range of applications. From the current consideration of image processing in the domain of pulp and paper, there are also other domains which can find great interest in the topics covered in this report. In principle, much of the work on surface characterization in the context of paper surface analysis could be of use to completely different research areas. One application is to the domain of road surface analysis. In his paper, Payne [18] presents an integrated system for road surface analysis which uses similar image processing techniques to analyze a road profile. Another possible application where the SurfCharJ plugin itself has actually been applied, is in the domain of soil analysis in the event of volcano eruptions.

2.3 Missing Dots and Printing

2.3.1 Causes

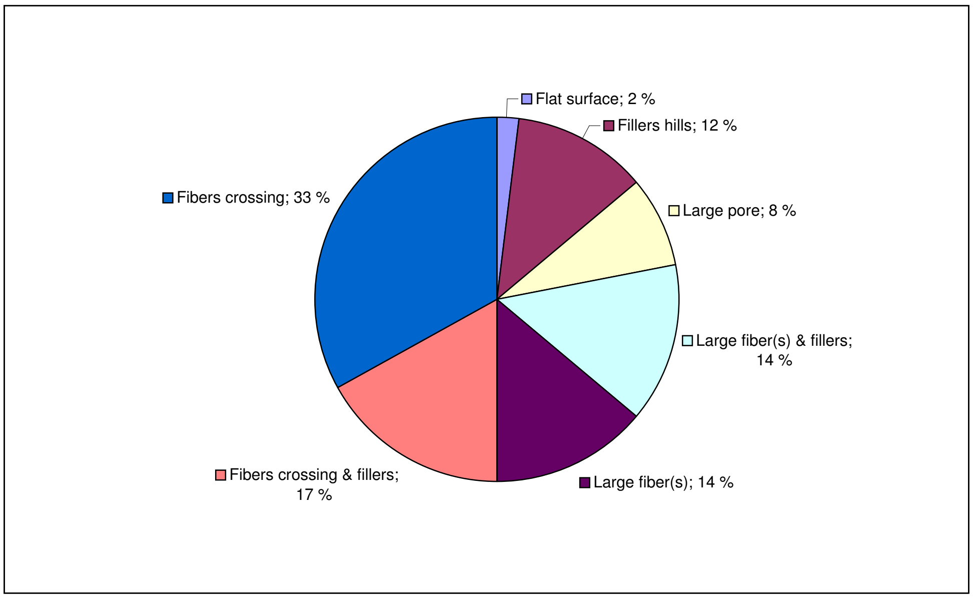


Figure 2-1: Classification of Missing Dots by Underlying Structure [1]

Pores are not the only cause of missing dots. As can be seen in Figure 2-1, which describes measurements made by Antoine [1], pores are actually not the even *main* cause of missing dots after printing. Nevertheless the present work has pore detection, pore segmentation and the relationship between pores in the surface to missing dots after printing as its main topic. The main reason for this is that this is an easily isolated subgroup where much progress is still needed in order to obtain good results.

2.3.2 Physical Consequences During Printing

As explained in Chinga's thesis and corresponding presentation [5, 6], there are several components to the problem of missing dots after printing. It can be influenced by three main interactions:

- paper and the printing ink
- paper and the printing press

- light and print

As the physical structure of the surface is the main focus of this thesis, it means that the interaction between the paper and the printing press is the most interaction of most interest here. A simplified discussion of this is presented next.

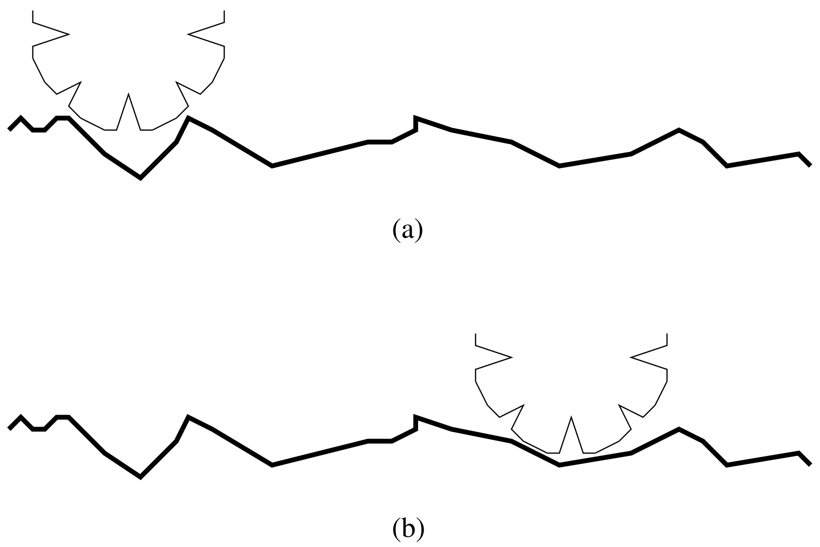


Figure 2-2: Missing Dots After Printing. (a) With, (b) Without a Pore Protruding the Surface

Figure 2-2 attempts to show how pores can cause problems during the printing process. The round shape with indentations is the printing head and the drawn profile represents the paper surface at microscopic scale. Each indentation of the printing head contains some ink before it hits the paper surface, whereupon it circles around and acquires new ink before hitting the paper surface again. The missing dot problem can occur when an indentation in the printing head does not come close enough to the paper surface. Then the ink may remain inside the indentation instead of transfering to the paper surface. This can happen when there is a pore present in the surface at that particular area. There are also many other causes, which are not part of the focus of this thesis. These are briefly mentioned in Chapter 2.3.1.

Again referring to Figure 2-2, at the point in time shown in (a) the surface contains a pore that causes ink not to transfer to the paper surface. At the point in time shown in (b) the surface does not contain a pore. Therefore the possibility of ink *not* transferring to the paper is thought to be greater in (a) than in (b).

2.4 ImageJ

ImageJ is a complete image processing application written fully in the Java programming language. It is easily extendible to new contexts due to its open archtecture and active plugin support. In addition, it has been placed completely in the public domain by the U.S. National Institute of Health, which makes it easily obtainable from the NIH home page [17]. ImageJ has been used with success in academic circles. Sage and Unser [19] describe an approach where

ImageJ was used as the base toolkit for a complete introductory course in image processing because of its ease of use and open nature.

However, some comments about ImageJ are in place. Sage and Unser [19] point to some important peculiarities of the ImageJ package, some of which were also noticed during plugin development in the present work. As noted above, ImageJ includes an easy-to-use plugin architecture. However, the approach taken by the internal (non-plugin) operators is often different from the preferred approach one would like to use when writing plugins. Therefore, in many cases it is actually preferable to write plugins together with associated routines from scratch even if the same routines are already present in the ImageJ base package. For some routines the coupling between abstract routines and the ImageJ program itself is too high to be useful in alternative contexts. Therefore, in the present work most routines rely little on calculation routines that come with the ImageJ base package, instead implementing most from scratch.

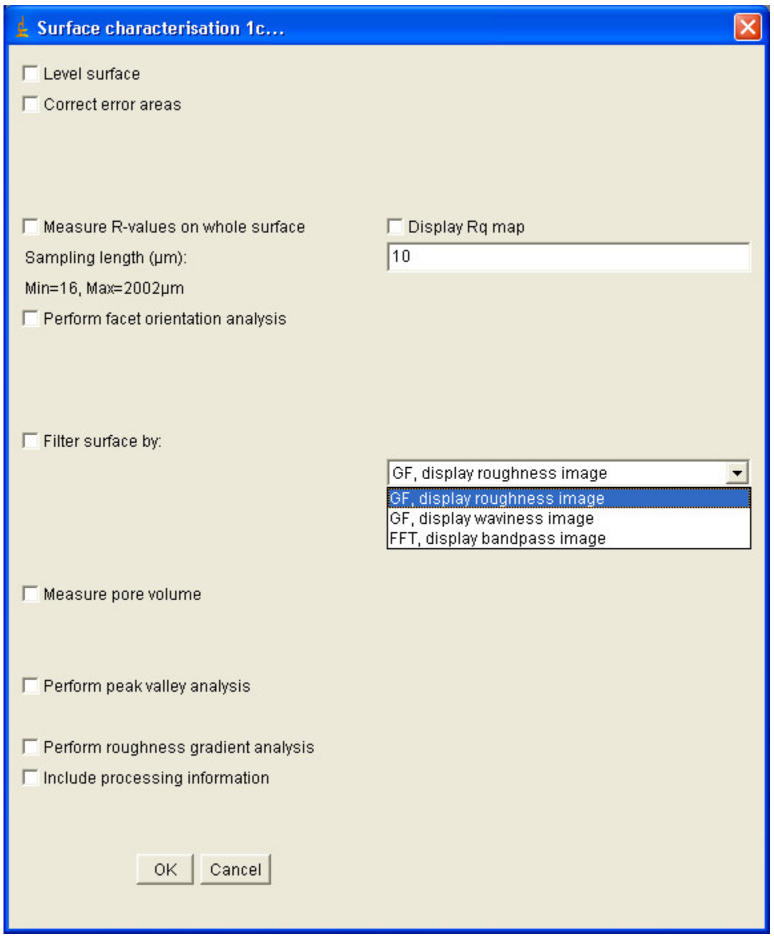
2.5 SurfCharJ Surface Characterization Plugin for ImageJ

SurfCharJ is a comprehensive plugin for ImageJ developed mostly by Gary Chinga. Two versions currently exist: a free version, and a version with additions proprietary to PFI. It should be noted that the free version is available for download from Chinga's private home page, with source code included [9].

SurfCharJ is, as mentioned, a comprehensive plugin, useful for surface characterization and analysis. It could be viewed as a general tool for semi-automated surface characterization and analysis work. As it is developed partly for PFI purposes, it has been tested and used mostly for paper surface microscopic images at the present point. However, since many people in other fields of research also could be interested in semi-automated surface characterization it may well be used for completely different types of surfaces in the future.

Few of the options provided by SurfCharJ are used for the present diploma work, as SurfCharJ covers an array of options, many which are not relevant here. However, two preprocessing options have been used extensively throughout the present work. These are the "Level surface" and the "Correct error areas" options, as they are quite generic and useful for nearly all image processing tasks involving LLP surface images.

Two screenshots presenting the available options and filters in SurfCharJ are shown next, with minimal options visible in Figure 2-3, and with all options visible in Figure 2-4. In the SurfCharJ interface an option becomes visible as one selects its "parent" option, that is, options that it depends on. Instructions for use and explanations of the various options and filters not used in the present work are available at the SurfCharJ home page [9].



Surface characterisation 1c...

☐ Level surface

☐ Correct error areas

☐ Measure R-values on whole surface ☐ Display Rq map

Sampling length (µm): 10

Min=16, Max=2002µm

☐ Perform facet orientation analysis

☐ Filter surface by: GF, display roughness image

- GF, display roughness image
- GF, display waviness image
- FFT, display bandpass image

☐ Measure pore volume

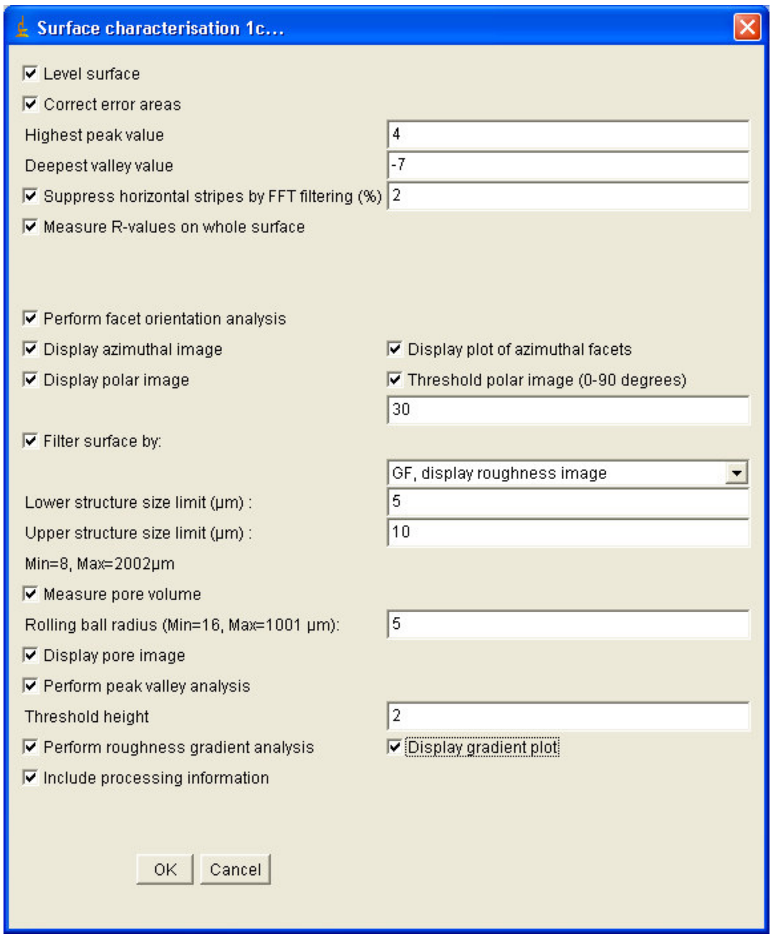
☐ Perform peak valley analysis

☐ Perform roughness gradient analysis

☐ Include processing information

OK Cancel

Figure 2-3: SurfCharJ, no Options Enabled


Surface characterisation 1c...

☑ Level surface
☑ Correct error areas

Highest peak value	4
Deepest valley value	-7
☑ Suppress horizontal stripes by FFT filtering (%)	2

☑ Measure R-values on whole surface

☑ Perform facet orientation analysis

☑ Display azimuthal image	☑ Display plot of azimuthal facets
☑ Display polar image	☑ Threshold polar image (0-90 degrees)
	30
☑ Filter surface by:	GF, display roughness image
Lower structure size limit (µm) :	5
Upper structure size limit (µm) :	10

Min=8, Max=2002µm

☑ Measure pore volume

Rolling ball radius (Min=16, Max=1001 µm):	5

☑ Display pore image
☑ Perform peak valley analysis

Threshold height	2
☑ Perform roughness gradient analysis	☑ Display gradient plot

☑ Include processing information

Figure 2-4: SurfCharJ, all Options Enabled

Chapter 3

Techniques

3.1 Overview

This chapter follows the outline of a future envisioned paper quality assesment system as presented in Figure 3-1. In the current outline of the system, six general, distinct phases are identified:

1. Sample Preparation
2. Image Acquisition
3. Image Processing (3 subphases)
4. Modeling
5. Measurement
6. Quality Assesment

In the current chapter, each of the general phases will be presented in turn and related to the present work. It should be mentioned that the present work has a focus on phases 3, 4, and 5, as the author has his background from computer science and image processing and has no background in microscopy and image acquisition from microscopy. Therefore the sections on phases 1 and 2 as well as 6 (which is closely connected to empirical research) will be brief.

Of special note is the final phase, Quality Assessment, which ends with a missing dot estimation. As outlined in Chapter 1.2, an important focus of the present work is approaching an assesment solution for the problem of missing dots in printing. A good assesment of the percentage of missing dots in the finished, printed paper would be of great use to the paper research community. This thesis does not arrive at this final stage but will hopefully be of use when designing the remaining stages.

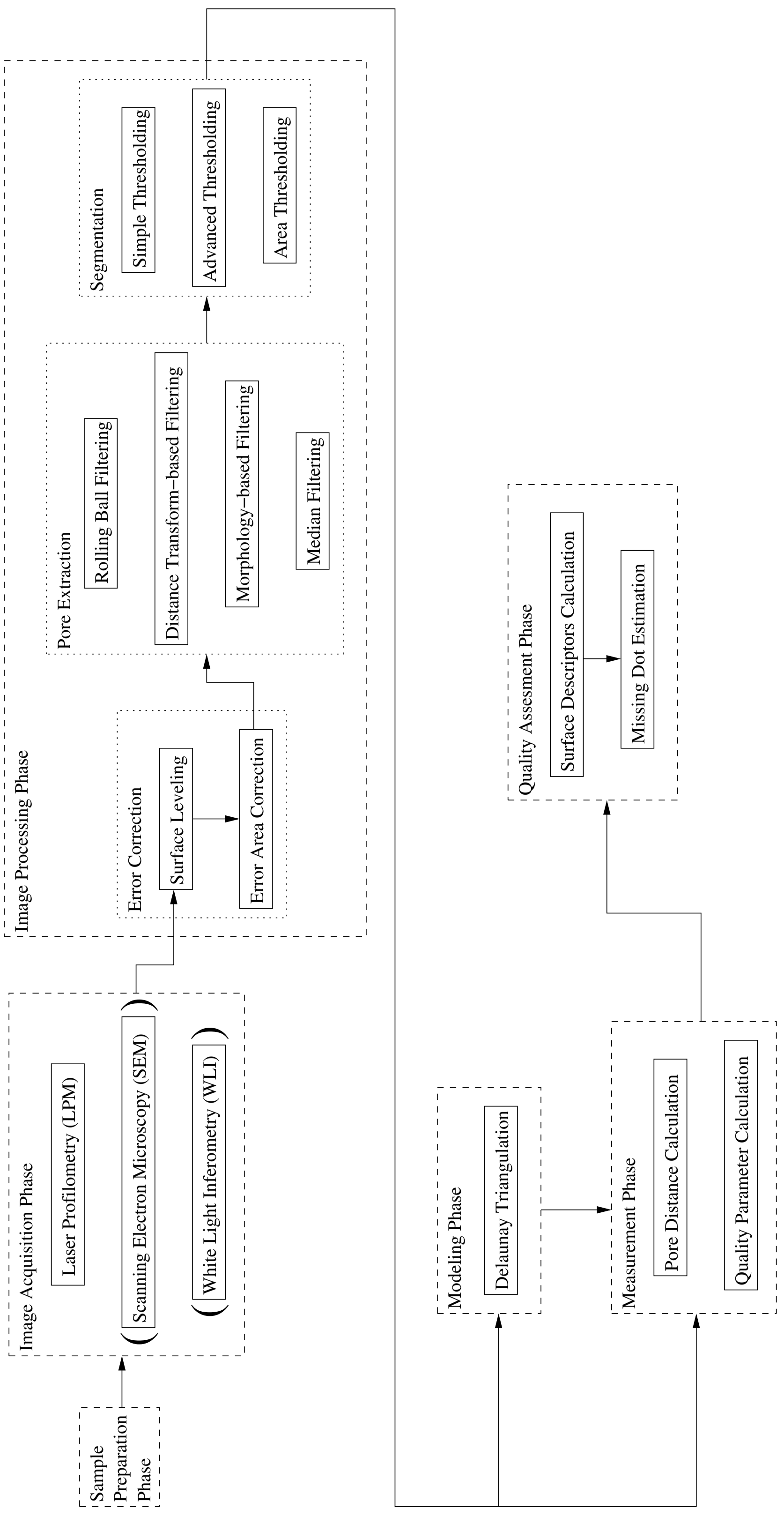


Figure 3-1: Component Overview

3.2 Sample Preparation Phase

Preparation of samples is not part of the present work. The paper samples used here are the same as described in Chinga [7].

3.3 Image Acquisition Phase

Lehrman Laser Profilometry was used in the present work for image acquisition, which yields images with an accuracy of about 1/100 of a micron for the pixel values, and a grid resolution of about 1 to 4 microns in the images. Of course, many other techniques are available for image acquisition. A short discussion on this can be found in Chinga [7]. As mentioned earlier this phase in Chapter 3.1, this phase is only mentioned briefly in the present work as the author has no background in microscopy and image acquisition from microscopy.

3.4 Image Processing Phase

This is the first phase covered in detail in the present work. The author has chosen to have a *single* phase encompassing all image processing in the system. The reason for this decision is the thinking that we then have a single phase that

- has *raw* surface data as input, and
- has *all* types of relevant images for further processing as output.

One important aspect during parts of the Image Processing phase, is to make sure that noise is handled with some degree of physical correctness. According to Holmstad [14] there are some considerations that should be made when attempting to remove noise from an acquired image. Most importantly, *"filtering techniques must be applied with precaution to find an optimum trade-off between removal of disturbing noise and physically present elements"* [14]. The conclusion is that one should, during image analysis routine development, visually compare the filtered image to the non-filtered image and assess the degree to which the filtered image preserves the structure topology. Preferrably, the optimal filtering should be chosen as the filter of choice, that is, the filtering routine which best preserves the structure topology.

As the measurement phase (Chapter 3.9) is set to produce concrete measurements from adequately preprocessed data, it is natural to apply all image (pre)processing techniques in one phase as is done here. However, it is also natural to divide this large phase into subphases. These are covered in the subsequent sections of this chapter.

3.5 Error Correction Subphase

The Error Correction subphase is based in whole on the implementation provided by the SurfCharJ plugin (Chapter 2.5) for ImageJ. The filtering used in this subphase was introduced fairly recently by Chinga [7].

3.5.1 Surface Leveling

The first part of the Error Correction subphase is the surface leveling filter provided by SurfCharJ. This filter provides a fully automatic way of normalizing the image data before any further processing.

The filter, called "Level surface", provides a way of removing disambiguation related to the orientation of the paper sample. In the present context, the orientation of the paper sample is simply refered to as the orientation of the image plane. It is clear that imaging through the use of Laser Profilometry will not yield uniformly oriented image planes when comparing samples. Minute differences in the image plane orientation will always occur with measurements at such microscopic levels.

The filter works by iterating over the entire image, calculating the least square estimates for the x and y dimensions and applying a regression model to all pixels in the image. The process of least squares estimation and regression model application [24] is repeated 4 times in order to gain a more balanced regression plane. It should be mentioned that, by its definition, a property of the resulting leveled image obtained through the calculation of a regression plane is that its average height after leveling is exactly zero.

3.5.2 Error Area Correction

The second part of the Error Correction subphase is the Error Area Correction filter provided by SurfCharJ. This filter provides a semi-automatic way of removing the pixel values in the LLP image that are clearly in error. The reason it is semi-automatic and not fully automatic is that the filter, called "Error area correction", takes as input two threshold values. It might be a possibility in the future to develop a fully automatic error correction filter which identifies reasonable threshold values through segmentation and automatic analysis of the segmented image, however it is not a goal of the present work. One of the reasons for this is that such an automatic analysis would preclude the use of external, empirical knowledge which plays a large part in the current semi-automatic process.

In the present work, knowledge of cross-sectional Scanning Electron Microscope (SEM) measurement values for Surface Calendered paper is used for determining the error correction thresholds. Chinga [7] estimated the optimal threshold values at -7.0 and +4.0 microns, since SEM measurements showed that *"the highest peak and deepest valley expected in SC paper can not exceed 4 and -7 microns respectively"* [7]. As such, these values are also used in the present work since surface calendered paper is used exclusively in the examples provided here.

The filter works by, first, creating a 3x3 mask locally on points exceeding the thresholds, second, calculating a mean value and storing this mean value at the point. If the calculated mean value is still outside the threshold, the mask is increased until its mean value is inside the threshold.

3.6 Pore Extraction Subphase

The Pore Extraction subphase was also, at first, based on the implementation provided by the SurfCharJ plugin (Chapter 2.5) for ImageJ. However, as part of new implementations in the present work, the Rolling Ball approach found in SurfCharJ was refined (Chapter 4.3.2) and included as one of several approaches considered here.

It is of importance to note that the following sections intend to describe several *independent* approches to solving the problem of pore extraction. Future studies could of combine several approaches but this will not be a focus in the present work.

3.6.1 Rolling Ball Filtering

The first approach is the use of a Rolling Ball filtering algorithm, as described by Chinga [7].

The Rolling Ball approach is a simple 2D spatial filtering approach. A detailed discussion of such 2D image filtering/enhancement in the spatial domain can be found in Gonzalez and Woods [13]. The Rolling Ball filter in particular is a smoothing filter, with a single parameter: the radius of the rolling ball. In essence, it preserves the pixel values in regions of the image with little change while smoothing the regions of the image with abrupt changes in the negative direction, assigning to the pixels in these regions values from the general regions around them. The hope is that the regions with abrupt changes will correspond closely to pores for the present purposes. The filtered image is then subtracted from the original image, yielding a new filtered image containing only pores (the rest of the image should in general be quite flat). The analogy to a physical rolling ball traversing the surface is clear, however it is important that the rolling ball has a large enough radius so it preserves the largest pores. According to crude estimates extracted from the images themselves, a diameter of 500 microns is possible for some very large pores, so the radius should ideally be at least 250 microns for good results. Later on, it could be of interest to filter with several radii and combine the results somehow to improve the accuracy.

At first, the SurfCharJ approach to the Rolling Ball filter was used ("Measure pore volume" option in SurfCharJ.) Later a separate independent plugin which overcome some problems with the SurfCharJ implementation was developed. One goal of the new approach was to be able to evaluate the quality of the implementation by providing the user with an overview of temporary images generated by the algorithm. These are now shown as separate result images in ImageJ after running the new plugin. Another goal of the new approach was to have a rotation invariant plugin. This was not the case with the SurfCharJ approach, as it did not seem this was a goal of that implementation (it did not treat the edge cases correctly). The new approach is perfectly rotation invariant for 90, 180, 270 degree rotations of the source image.

3.6.2 Distance Transform-based Filtering

One goal of the present work was to improve the performance of the Rolling Ball spatial filtering algorithm (Chapter 3.6.1). As such, it became natural to explore alternative approaches to the problem, since the problem lies in the O($n^2 \times m^2$) time complexity of the usual implementation of spatial filters.

A natural alternative approach was therefore a Distance Transform approach, successfully applied to image processing problems in pulp and paper research by a Swedish research community (Borgefors et al. [2, 3, 20, 21]). Distance Transform approaches for 3D environments take as input a 3D image in the form of a voxel[1] structure.

An example of how the simplest form of the Distance Transform works is shown in Figure 3-3. This figure shows a Distance Map in two dimensions on a binary image. As can be seen in the figure, the distance to the background is assigned to every pixel. Thus, pixels deep inside the segmented area are assigned larger values than pixels closer to the border. The version of the distance transform thought to be useful for the present purposes is extended in a few ways compared to the simplest version. The most useful way to describe it seems to be as follows. A binary image is still used. However, it is a 3-dimensional binary image as opposed to a 2-dimensional binary image. In an ImageJ context this would be implemented as a binary image consisting of several slices. In addition to having to calculate the distance transform in 3 dimensions instead of 2 there is the problem of scale. The LLP images used for the present purposes usually have at most a resolution of 1 micron per pixel in the pixel grid. However, measured values are on the order of 1/100 of a micron. This suggests generating 100 slices for each unit step for pixel values. However, this causes elongated voxels, which can be a pain to work with according to Sintorn and Borgefors [20]. If elongated voxels are to be avoided, the resolution of the pixel grid for each slice also has to be increased by a factor of at least 100 in both the x and y direction for the current LLP images through some method of interpolation. It is clear that the most feasible option might be some form of tradeoff between elongation and accuracy.

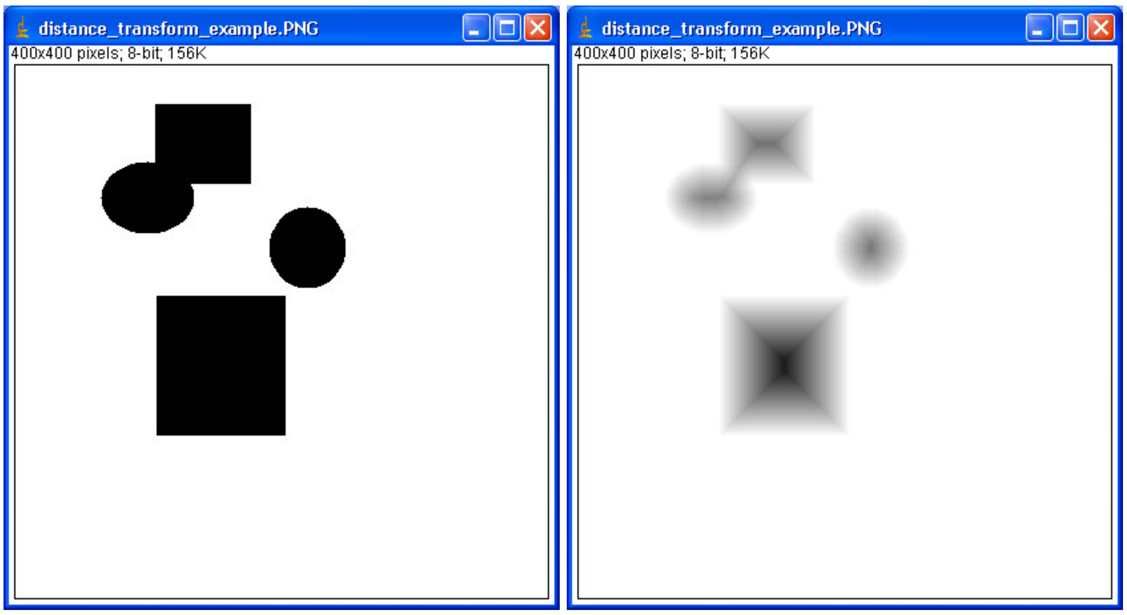
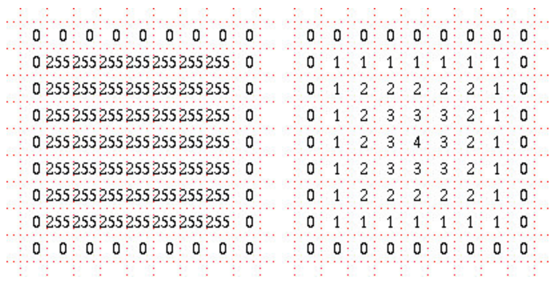

distance_transform_example.PNG
400x400 pixels; 8-bit; 156K
distance_transform_example.PNG
400x400 pixels; 8-bit; 156K

Figure 3-2: Example Image Before and After Distance Map Calculation

[1]3D equivalent of a pixel, usually in the form of a unit cube



0	0	0	0	0	0	0	0	0
0	255	255	255	255	255	255	255	0
0	255	255	255	255	255	255	255	0
0	255	255	255	255	255	255	255	0
0	255	255	255	255	255	255	255	0
0	255	255	255	255	255	255	255	0
0	255	255	255	255	255	255	255	0
0	255	255	255	255	255	255	255	0
0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	0
0	1	2	2	2	2	2	1	0
0	1	2	3	3	3	2	1	0
0	1	2	3	4	3	2	1	0
0	1	2	3	3	3	2	1	0
0	1	2	2	2	2	2	1	0
0	1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0

Figure 3-3: Example Array Before and After Distance Map Calculation

3.6.3 Morphology-based Filtering

Image processing through the use of morphological operation is quite often used on binary images as part of a segmentation process. Two examples of morphology-based filtering are shown in Figures 3-4 and 3-5. The first example (Figure 3-4) is from the domain of OCR (Optical Character Recognition). Here, morphological techniques have proven useful because of the fact that text consists of connected regions, each of which represents a unique letter. Since morphological techniques can connect nearby, but disconnected, regions easily it is clear why this is a good technique for OCR purposes. The second example (Figure 3-5) is an illustration of morphological processing in the context of pore segmentation (the image is copied from the later tutorial which Figure 4-12 is a part of).

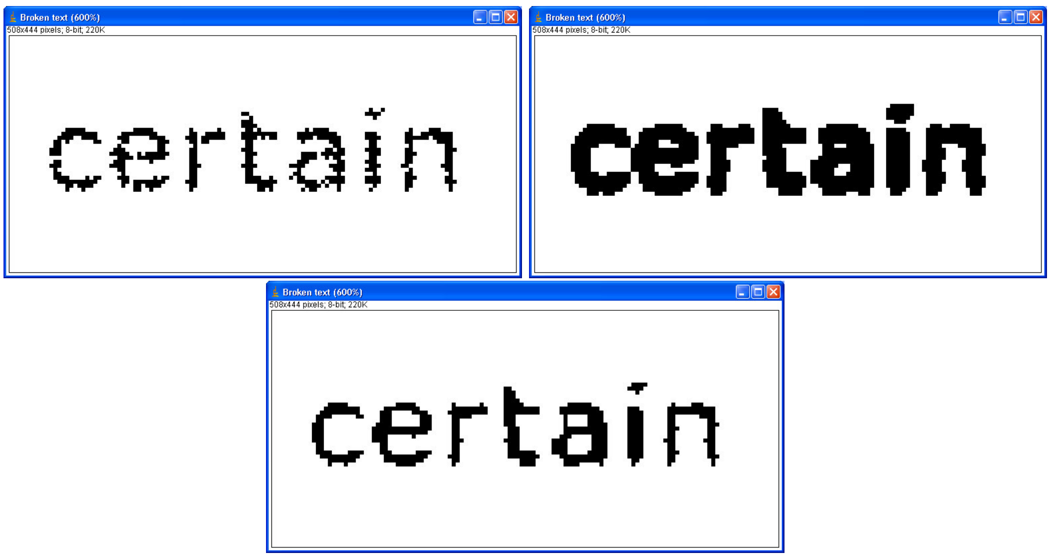

Broken text (600%)
508x444 pixels; 8-bit; 220K
certain
Broken text (600%)
508x444 pixels; 8-bit; 220K
certain
Broken text (600%)
508x444 pixels; 8-bit; 220K
certain

Figure 3-4: Broken Text Processed with Morphological Filtering (Dilation and Erosion) [13]

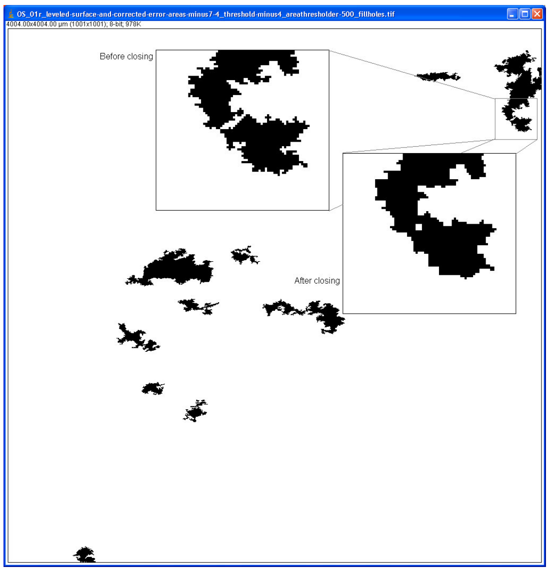

OS_01r_leveled-surface-and-corrected-error-areas-minus7-4_threshold-minus4_areathresholder-500_fillholes.tif
4004.00x4004.00 µm (1001x1001); 8-bit; 978K
Before closing
After closing

Figure 3-5: Morphological Filtering on a Disconnected Pore

As can be seen in Figure 3-4 morphological filtering can be quite effective at closing gaps at places where no gap should occur. It should be evident to the reader that the first image is a bad scan of the word "certain", with the letters "c", "e", and "r" actually consisting of several disconnected regions from an image processing viewpoint. Segmenting the image into 7 regions (8 if one counts the dot over the letter i) is easy for humans, as cognitive processes in the brain closes the evident gaps automatically and groups them into separate, understandable letters. For a computer, however, explicit processing is needed, and our cognitive process can be simulated by the morphological closing operation, consisting of one dilation operation followed by one erosion operation [13]. Most would agree that the resulting image is extremely close to the brain's actual impression of the first image. It should be mentioned that morphological filtering also can be used in the opposite situation, that is, introducing gaps where two separate regions are connected which should be disconnected.

In relating morphological filtering to the present work, Holmstad [14] clearly shows that pore interconnection is a serious challange in surface pore analysis: "Except for high density paper grades, most of the pores in paper are interconnected" [14]. In the current context gaps (or missing gaps) will arise if some sort of thresholding is used as a first step in pore segmentation. For instance, gaps may arise in those areas where the threshold value used is set too low, and gaps may be missing in those areas where the threshold value is set too high. To illustrate this, a small region has been zoomed into in Figure 3-5. This region is interesting because it seems to a user that it contains a single pore, when in fact it is treated as two separate pores because of the disconnection that arises as an artifact of thresholding. That they are in fact two separate pores is evident after Delaunay Triangulation, which is described in Chapter 4.6.3 and the related Figure 4-16. Applying the morphological closing operation to this region, as in the OCR example above, solves this problem elegantly, though it could be argued that the new white "hole" in the middle of the segmented pore should filled after filtering (analogous to Figure 4-15), for simplification purposes.

3.6.4 Median Filtering

Median filtering is an alternative which is actually quite closely related to area thresholding outlined below in Chapter 3.7.3. Median filter is most famous for removing so-called "salt and pepper" noise from images with minimal distorsion in the replaced areas if a reasonable mask size is used.

For the present work, median filtering was not used, instead corresponding results were obtained later in the process with the area thresholding technique.

3.7 Segmentation Subphase

The Segmentation subphase was not originally included in any options provided by the SurfCharJ plugin (Chapter 2.5) for ImageJ. In general, there has been little emphasis so far in developing options for segmentation. Therefore it is interesting to explore the available options in the present work.

It is of importance to note that the following sections are intended to describe *complementary* techniques. As is known from most aspects of automatic image processing, most problems in the image processing domain do not have one single "correct" solution. This is also for the most part correct for the pore segmentation aspect of surface analysis. However in the present context of pores in paper one can point to more guidelines for "correct" segmentation than for some other problems. That is, since a goal of this segmentation is to identify problems and properties at a macro level with micro level LLP images, it is of large importance to restrict the segmentation effort to those objects which cause differing properties at the macro level. A simple example of this is the measured surface area of a pore – regardless of the shape and form of the pore it will probably be insignificant in the printing process if its surface area is significantly less than the surface are covered by one ink dot at the given dot density for this printing process.

3.7.1 Simple Thresholding

The first option is to use simple thresholding. In this case simple thresholding consists simply of input of a threshold value in microns, outputting a binary image where every pixel with a pixel value below the threshold is set to zero and every pixel with a pixel value above the threshold is set to one.

This approach is simple and gives quick results but is not robust against noise. Therefore other techniques need to be combined with simple thresholding to yield usable results.

3.7.2 Advanced Thresholding

There are several options when it comes to more advanced forms of thresholding. The first refinement that comes to mind would be to use adaptive thresholding as described in Gonzalez and Woods [13]. However for several reasons this is not of much interest in the present context.

First, a short discussion. One goal of using adaptive thresholding as opposed to simple/global thresholding is that often, many image attributes are unknown and cause trouble when thresholding at a single level globally. These attributes are for instance lighting, orientation in the depth plane, tilt, and any other causes of local variations in the image.

The main reason why adaptive thresholding is not so interesting in the present context is the fact that a modular system is being described, where all the attributes mentioned above should already be normalized through use of the SurfCharJ plugin for ImageJ. More specifically, the two most important attributes needing normalization are tilt and measurement scale. Tilt is corrected by use of the "Level surface" option (Chapter 3.5.1) in SurfCharJ. Measurement scale is stored inside the 32-bit TIFF image itself, so that all image operation parameters can be specified in microns with no further considerations about the ratio of pixels/pixel values to microns.

One extension of adaptive thresholding which *could* be interesting, however, would be to combine morphological operations (as discussed in Chapter 3.6.3) with the thresholding process itself. An idea for future work would be to do this in form of a feedback algorithm. In such an algorithm, a local thresholding would first be performed. Second, the number of regions together with their surface areas and shape would be measured and assigned some weight defined by an appropriate weighting function. Third, one or more morphological closing operations (Figure 3-4) as well as one or more morphological opening operations would be applied, whereupon the surface areas are reevaluated. Here, shape data about pores obtained previously would be of interest for this reevaluation. If it is clear that two (or more) regions that have become connected after closing really only represent *one* pore, these regions remain connected. If it is clear that one region that has become disconnected after opening really represents *two* (or more) pores, this region remains disconnected. Regions not significantly affected by the morphological filtering remain untouched.

3.7.3 Area Thresholding

Area thresholding is an approach orthogonal to the thresholding methods outlined above. This type of thresholding is for the present purposes meant to be applied to binary images only. The binary image may be obtained through a thresholding as outlined above, or by other approaches that may be explored in future work.

Area thresholding is experimented with in the present work, with results discussed in Chapter 4.1, in relation to Figure 4-14.

3.8 Modeling Phase

In an ideal system, hopefully, the main image resulting from the complete Image Processing phase is a largely simplified image which is easy to process further. The format experienced most with in the present work is a binary (black-and-white) segmented image, with pore areas colored black and non-pore areas colored white. However, it should be noted that this is only one of many possible result images which could be used for future systems. It could also be useful to return several result images from the Image Processing phase, each with its own characteristics relevant for e.g. different types of measurements and other further processing in the system. For instance, a segmented binary image is useful when measuring objectively the simple 2D shape of pores, as well as distances and other types of relationships *between* two or more pores. However, segmented binary images contain little information on the 3D pore shape of *individual* pores. For analysis of 3D pore shape on can combine the original (leveled and error corrected) image with the binary segmented image (used as a mask). This could be a useful option, returning one pore image for each interesting region in the binary segmented image.

It should be mentioned that the Modeling phase is thought of as an optional preprocessing stage performed prior to the Measurement phase, returning some model built from the raw results from the Image Processing phase. The only modeling component included thus far is Delaunay Triangulation, but other model-building components could be useful in future systems.

3.8.1 Delaunay Triangulation

The theory behind, as well as applications of Delaunay triangulation and possible implementation considerations are discussed in Chapter 4.6.3.

3.9 Measurement Phase

As already mentioned in relation to the Modeling phase (Chapter 3.8), it might be useful to return multiple result images from the Image Processing phase. In addition to these result images, the measurement phase also has the option of including results derived from the Modeling phase, for instance Delaunay triangulation data.

3.9.1 Pore Distance Calculation

An example of pore distance calculation retrieved directly from the Delaunay triangulation is given in Figure 4-20. A problem with this approach is that too many edges are included by default. Methods of reducing the number of edges are explored and described in Chapter 4.6.4 which includes Convex hull and Minimum spanning tree calculation as two starting points.

3.9.2 Quality Parameter Calculation

It is thought that in future work it may be possible to combine data from the Image Processing phase, the Measurement phase as well as pore distribution data in order to arrive at suitable quality parameters. One example of such a parameter could be statistical properties of the pore distance distribution. Evaluating possible quality parameters is, however, not a goal of the present work and so is only touched on briefly.

3.10 Quality Assesment Phase

The goal of the Quality Assesment phase is to arrive at a suitable estimate of missing dots in the paper sample, both the quantity of missing dots and their layout. The quality assesment phase is peripheral to the present work and so is only touched on briefly. It is, however, thought that calculation of standard surface descriptors in any case would be useful in arriving at the final missing dot estimate.

3.10.1 Surface Descriptors Calculation

Here the usual surface descriptors used in surface analysis should be calculated. These surface descriptors are described in the ISO 4287/2000 standard as mathematical line profile expressions, and digital approximations of these are presented in Chinga, Gregersen, and Dougherty [10]. The digital approximations are implemented as routines in the SurfCharJ plugin for ImageJ (Chapter 2.5). In order, the surface descriptions are

- *Ra*: Arithmetical mean deviation,
- *Rq*: Root mean squeare deviation
- *Rsk*: Skewness,
- *Rku*: Kurtosis,
- *Rv*: Largest depth measurement,
- *Rp*: Largest height measurement,
- *Rt*: *Rv* + *Rp*, and
- *Rz*: <= *Rt* (same as *Rt* but within a restricted sampling length).

Further details as well as a graphical presentation of the descriptors is given in Chinga, Gregersen, and Dougherty [10]. As well as the standardized descriptors given here, further research could well come up with new descriptors relating more closely to the missing dot problem.

3.10.2 Missing Dot Estimation

It is thought that the ultimate stage of missing dot estimation would take into account both the results calculated in the Measurement phase and more general measurements as derived from the surface descriptors. Finding a suitable combination of the two that can give a reasonable estimate is left as a basis for further research.

Chapter 4

Implementation

4.1 Area Thresholder

The concept of area thresholding was first introduced in Chapter 3.7.3. A plugin for ImageJ was developed that attempts to perform area thresholding in a way useful to paper surface segmentation. The interface for the plugin is structured in the same way as SurfCharJ, with options becoming available when "parent" options are checked. The interface is shown in Figure 4-1.

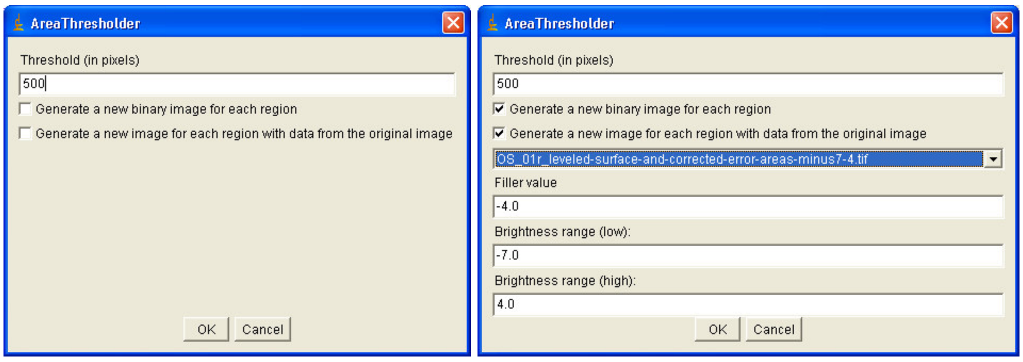
Using the interface, an area threshold value is first given. The value is given in pixels, so with the example value of 500 pixels all areas containing more than 500 (4-)connected pixels remain black in the output images, areas containing less than 500 are changed to the white background color. Using the plugin with only this option gives a binary image image of the type shown in Figure 4-14 when using the binary image shown in Figure 4-13 as input (the figures are part of a later example).

Two other main options are given in the interface. If the first checkbox is checked new images are generated for *every* region remaining in the image after area thresholding. These new images are generated by simply copying subregions of the binary input image. Though, only the *current* region is included in each output image; other regions that may be present in the subregion are not included in the new image generated from the subregion.

The second checkbox may be of more interest for pore analysis work. Before checking this option the original, 32-bit image should be opened in ImageJ. When this option is checked a dropdown menu appears containing the currently open images in ImageJ. From this dropdown menu the original image must be selected. If the plugin is now run, new images are generated for every region as before, however, 32-bit binary data backtranslated from the original image is now copied into the regions (instead of simply black and white binary data). In paper surface terms, this means each pore gets its own image containing only this pore (and void area around it).

The three remaining textfields are options for the second checkbox. "Filler value" chooses which value should be assigned to void areas. The two "Brightness range" fields are analogous to the options provided by the "Image->Adjust->Brightness/Contrast..." command in ImageJ.

They select which brightness range should be used for the new, generated images. These are only for practical display and comparison purposes; they do not change the generated images themselves. The reason for including this option, however, is that it would be a very tedious job to set the values manually for every generated image.


AreaThresholder

Threshold (in pixels)
500

☐ Generate a new binary image for each region
☐ Generate a new image for each region with data from the original image

AreaThresholder

Threshold (in pixels)
500

☑ Generate a new binary image for each region
☑ Generate a new image for each region with data from the original image

OS_01r_leveled-surface-and-corrected-error-areas-minus7-4.tif

Filler value
-4.0

Brightness range (low):
-7.0

Brightness range (high):
4.0

Figure 4-1: AreaThresholder Interface, with no Options Enabled and all Options Enabled

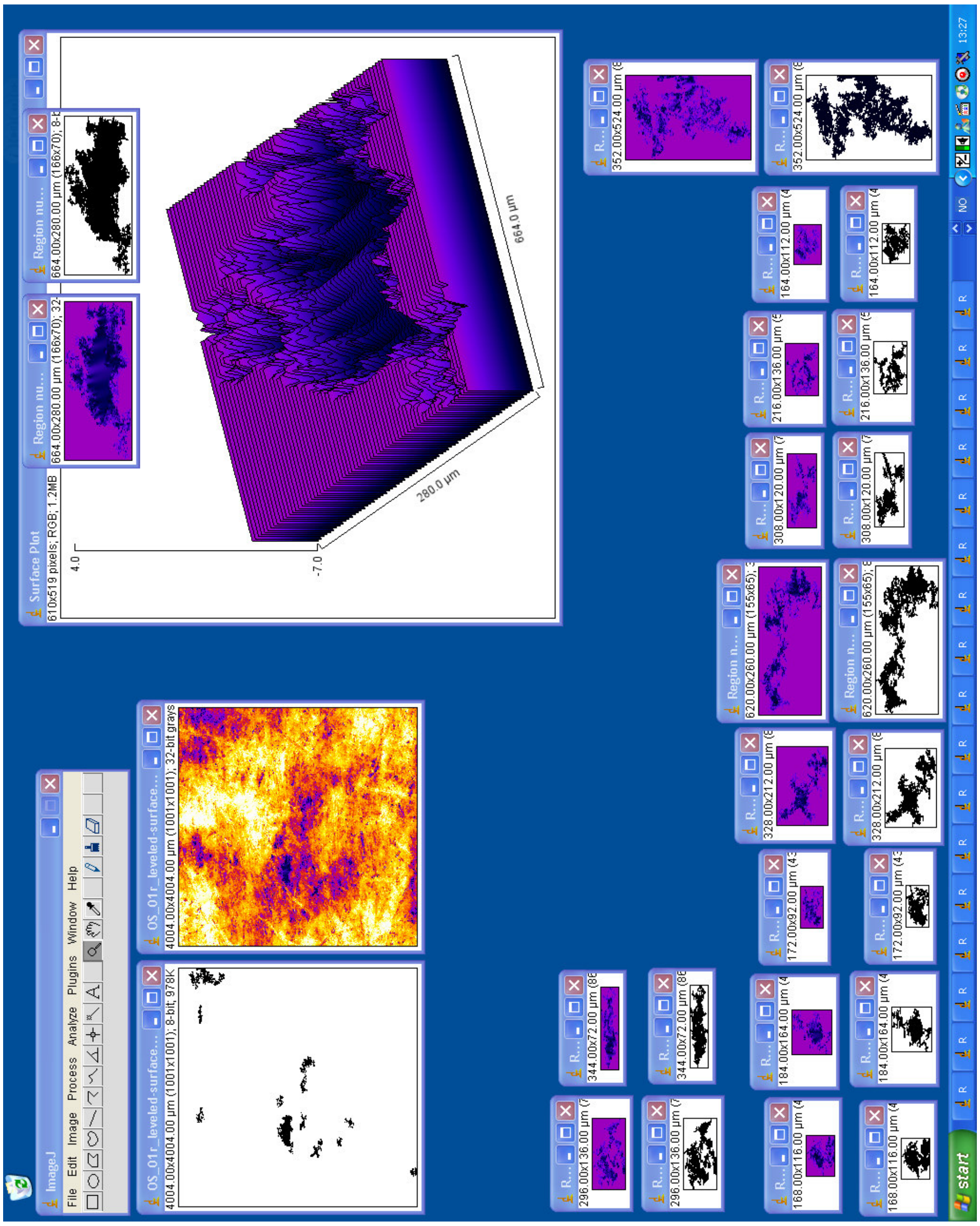

Surface Plot
610x519 pixels; RGB; 1.2MB
4.0
-7.0
280.0 µm
664.0 µm
ImageJ
File Edit Image Process Analyze Plugins Window Help
OS_01r_leveled-surface...
4004.00x4004.00 µm (1001x1001); 8-bit; 978K
4004.00x4004.00 µm (1001x1001); 32-bit grays
Region nu...
664.00x280.00 µm (166x70); 32-
664.00x280.00 µm (166x70); 8-
start
13:27

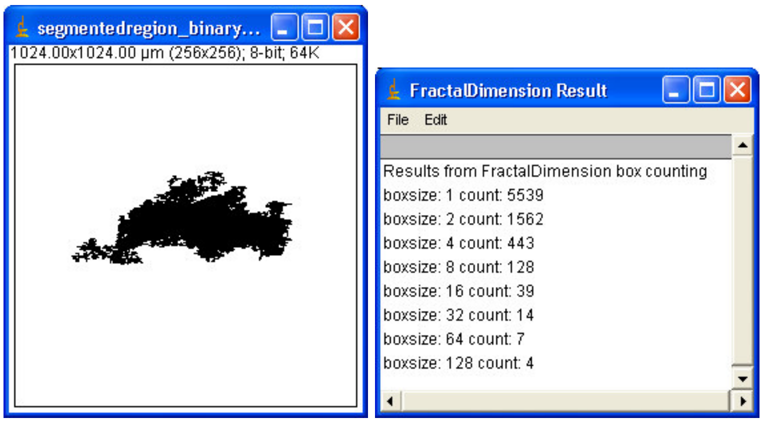
Figure 4-2: Screenshot After Filtering with AreaThresholder

An example of the region image generation functionality of the AreaThresholder plugin is given in Figure 4-2. Here the ImageJ toolbar, the input binary image, and the original 32-bit image are present in the upper left corner. Scattered throughout the screen are the generated region images, both binary versions and backtranslated 32-bit versions. In the upper right corner the largest region image has been plotted to a 3D surface with ImageJ's built-in Surface Plot command. A filler value of -4.0 micron for the void area was used, since this was the original height thresholding value used for the example image taken from Figure 4-14.

4.2 Fractal Dimension

Fractal dimension by box counting [25] was implemented as a plugin called FractalDimension. This plugin takes a binary image as input, performs box counting with increasing box size 1x1, 2x2, 4x4, 8x8 etc, and shows the result in a text window. If one takes the log on each column and calculates the regression line, the Fractal dimension of that image is found.

An example run of the FractalDimension plugin is shown in Figure 4-3. The binary region image which was surface plotted in Figure 4-2 is used as input, padded to 256x256 pixels since the FractalDimension plugin is designed for images with resolution an integer multiple of a power of two. The linear regression of log values needed for fractal dimension calculation was for this example calculated with a TI-83 pocket calculator, yielding a Fractal dimension of 1.53. A further analysis of this number is not the topic of the present work, but sufficient to say, a Fractal dimension of 1.53 might indicate an image with structure "slightly less complex than a 2D (2.0D) image of average complexity".


segmentedregion_binary...
1024.00x1024.00 µm (256x256); 8-bit; 64K
FractalDimension Result
File Edit

```
Results from FractalDimension box counting
boxsize: 1 count: 5539
boxsize: 2 count: 1562
boxsize: 4 count: 443
boxsize: 8 count: 128
boxsize: 16 count: 39
boxsize: 32 count: 14
boxsize: 64 count: 7
boxsize: 128 count: 4
```

Figure 4-3: Fractal Dimension by Box Counting

4.3 Rolling Ball

The use of a Rolling Ball algorithm at PFI in conjunction with paper surface structure characterization was first presented in Chinga's paper [7]. In the following sections the method and its particular properties, advantages, and disadvantages are presented and related to the present work.

4.3.1 Current Status

In the beginning of the project work the "Measure Pore Volume" feature of the SurfCharJ plugin for ImageJ (Chapter 2.5) was used as a starting point for further work on pore volume measurement. Some time was spent at the beginning of the project for studying the SurfCharJ plugin, both the parts related to the measurement of pore volume by the Rolling Ball algorithm and also the remaining parts. This was done partly to gain some understanding of surface characterization, and partly to learn about the underlying structure of the SurfCharJ plugin, how it is designed and how its modules are connected together. In addition it was of interest to learn more about general programming using the ImageJ framework, as the author had no previous experience with this particular image processing package.

It was found early on that the SurfCharJ is somewhat divided into separate parts implemented as separate classes:

- SurfCharJ_1d (main plugin class)
- FacetOrientation
- SurfaceFiltering
- SurfaceMath
- SurfacePeakValley
- SurfacePoreVolume
- SurfaceRoughness
- SurfaceRoughnessGradients

The class of most interest to the present work is the SurfacePoreVolume class, which is the enabling class for pore volume measurement in the SurfCharJ plugin.

Initial analysis was conducted using the "Measure Pore Volume", that is, the SurfacePoreVolume class, in order to gain an understanding of the results of surface analysis through the use of the Rolling Ball algorithm. To aid in this analysis the rolling ball filtering parts were separated from the SurfCharJ code into a "RollingBallOriginal" plugin which calls the API of SurfCharJ for simpler use during experimentation as well as accurate progress monitoring.

Some problems with the SurfacePoreVolume were observed, however. It was therefore decided to write a new Rolling Ball plugin from scratch in a new implementation separate from the SurfCharJ framework. One important aspect with the separate implementation would be the possibility of displaying intermediate results from the analysis. This would enable a thorough analysis of the inner workings of the algorithm and point out possible problem areas of using a Rolling Ball analysis for general surface characterization.

One of the problems was related to the accuracy of the analysis. It was found that the method for applying the Rolling Ball filter was not rotation independent. This was easy to verify, as a simple application of the "Measure Pore Volume" option in SurfCharJ to an image, and then to the same image rotated 90 degrees (rotating the resulting filtered image by -90 degrees, of course), yielded two different result images. Since there was no apparent reason for this inherent in the filtering algorithm, and since in general it is a good property of a filter to be as transformation independent as possible, the new implementation in the "RollingBallNew" plugin remedies this by giving bit-for-bit identical results irrespective of initial orientation.

4.3.2 Improved Rolling Ball

As mentioned in Chapter 4.3.1, it was decided to write a new plugin from scratch, independent of the existing SurfCharJ plugin which the author had not taken part in the development of.

The new plugin for Surface Characterization was simply implemented as a completely separate plugin using the general ImageJ framework

The results from running the new Rolling Ball plugin, "RollingBallNew", is shown in Figures 4-4 through 4-8. The plugin has been run with two different parameters. The first run was with radius 20 microns, the second run was with radius 252 microns. The first run completed in reasonable time for interactive use, that is, within a minute on a 2.0 GHz P4. The second run needed much more time due to the time complexity of the filtering. It finished in about 15 minutes. However, it is clear both from the theory described in Chapter 3.6.1 and from the result images themselves that a large rolling ball is neccessary for useful analysis. This will be discussed next.

The first image (Figure 4-4) shows what is basically a copy of the original LLP input image with the surface leveled and error areas corrected. The difference between this and the original, Figure 4-12, is only that some edge pixels have been chopped off. The copy is provided as output from the "RollingBallNew" plugin for easier pixel-by-pixel comparison with the complete, filtered image. The radius of the rolling ball defines how much is chopped off at each edge, and the chopping is neccessary because the rolling ball image relies on the use of neighboring pixels up to the distance equal to its radius. When the rolling ball is convolved close to the edge there will be no neighboring pixels at a distance this far away. Padding the image could circumvent this, but as that would yield only approximate, not completely accurate values it was chosen not to pad the image along the edges. It should also be noted that the rolling ball mask as shown in Figure 4-5 is not to scale with the other images as the image is zoomed in this figure.

The second image (Figure 4-6) shows what was at first thought of as an uninteresting temporary data structure internal to the algorithm, but later turned out to be quite interesting when analyzing the resulting filtered image. The image shows the rolling ball mask value used for each pixel in the filtered image. That is, lower (more blue) values in this image represents pixels where the resulting pixel stored at this location was taken from a close neighboring pixel compared to those pixels with higher (more white) values. It is therefore the case that those areas with deep (detected) pores contain more higher-valued pixels, since the rolling ball in those positions only touches positions far away from the pixel under consideration.

The third image (Figure 4-7) shows the resulting filtered image, and the fourth image (Figure 4-8) shows the output image. The output image is calculated by subtracting the filtered image from the original, shown in Figure 4-4. Ideally the output image should only contain detected pores. As is evident from the left (rolling ball radius 20 micron) image, few pores are detected with a low radius rolling ball. The right image, however, does quite well in comparison. Most of the uninteresting, non-pore areas are now quite smooth, and pores remain quite similar to their appearance in the original image (Figure 4-4).

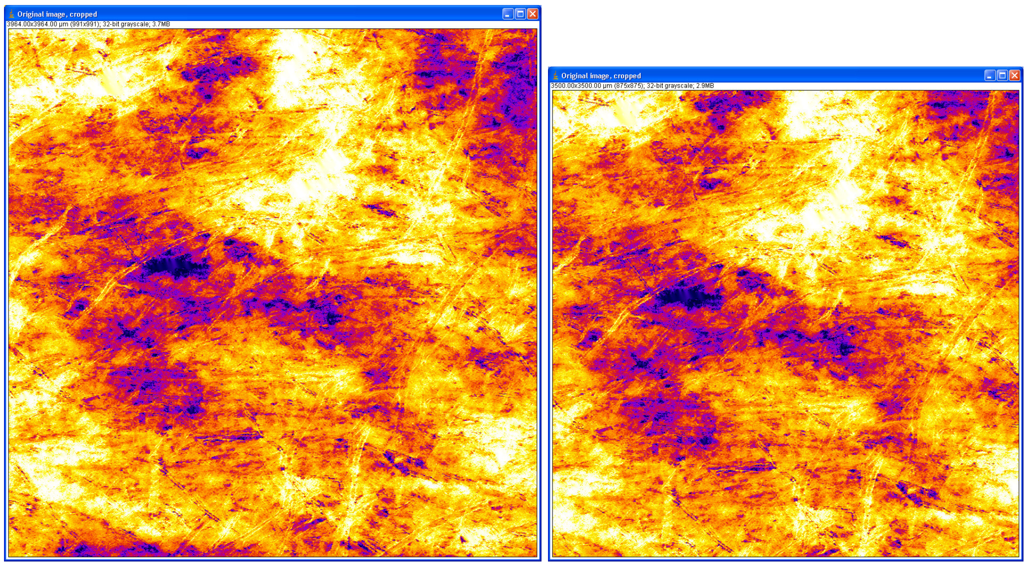

Original image, cropped
3964.00x3964.00 µm (991x991); 32-bit grayscale; 3.7MB
Original image, cropped
3500.00x3500.00 µm (875x875); 32-bit grayscale; 2.9MB

Figure 4-4: Original Image, Cropped to Equal Size as Rolling Ball Filtered Image (radius=20 and 252 micron)

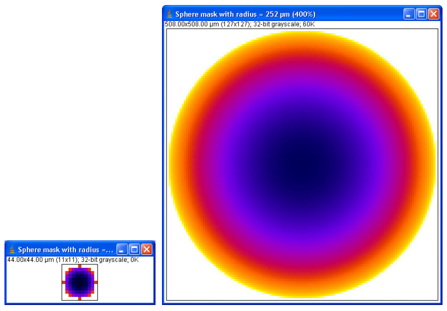

Sphere mask with radius =...
44.00x44.00 µm (11x11); 32-bit grayscale; 0K
Sphere mask with radius = 252 µm (400%)
508.00x508.00 µm (127x127); 32-bit grayscale; 60K

Figure 4-5: Rolling Ball Mask (radius=20 and 252 micron)

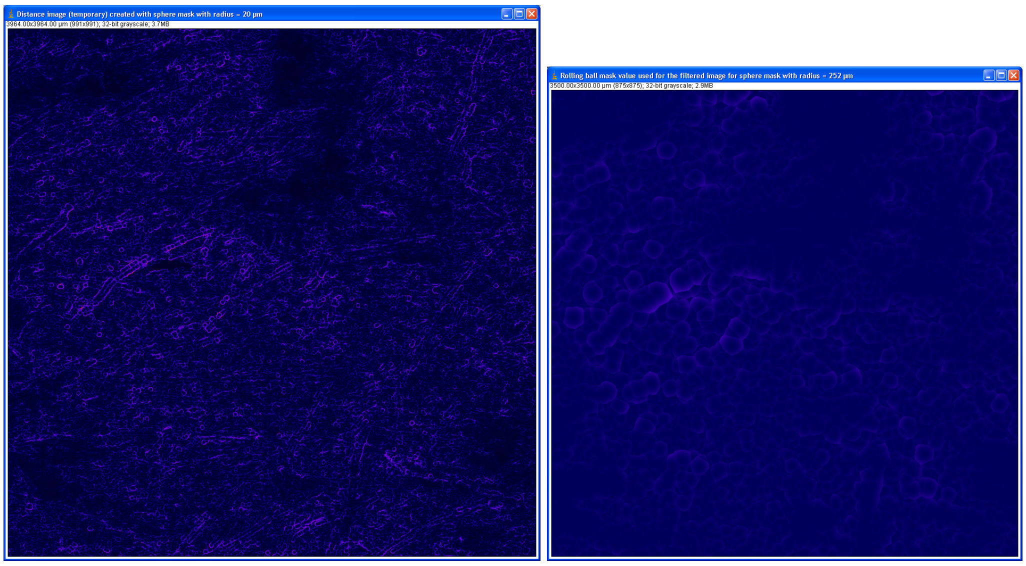

Distance image (temporary) created with sphere mask with radius = 20 µm
3964.00x3964.00 µm (991x991); 32-bit grayscale; 3.7MB
Rolling ball mask value used for the filtered image for sphere mask with radius = 252 µm
3500.00x3500.00 µm (875x875); 32-bit grayscale; 2.9MB

Figure 4-6: Rolling Ball Mask Value used for the Filtered Image (radius=20 and 252 micron)

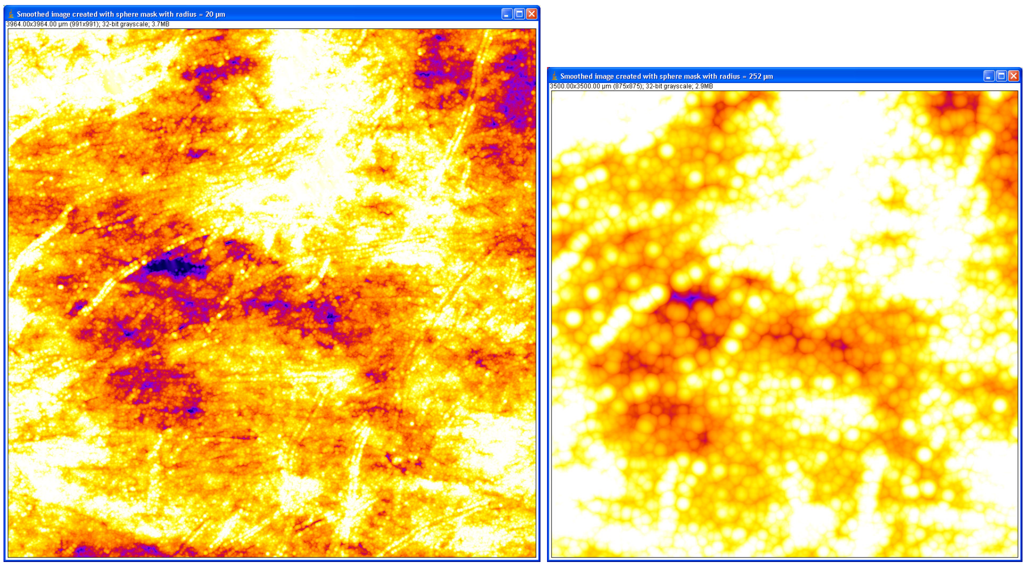

Smoothed image created with sphere mask with radius = 20 µm
3964.00x3964.00 µm (991x991); 32-bit grayscale; 3.7MB
Smoothed image created with sphere mask with radius = 252 µm
3500.00x3500.00 µm (875x875); 32-bit grayscale; 2.9MB

Figure 4-7: Rolling Ball Filtered (Smoothed) Image (radius=20 and 252 micron)

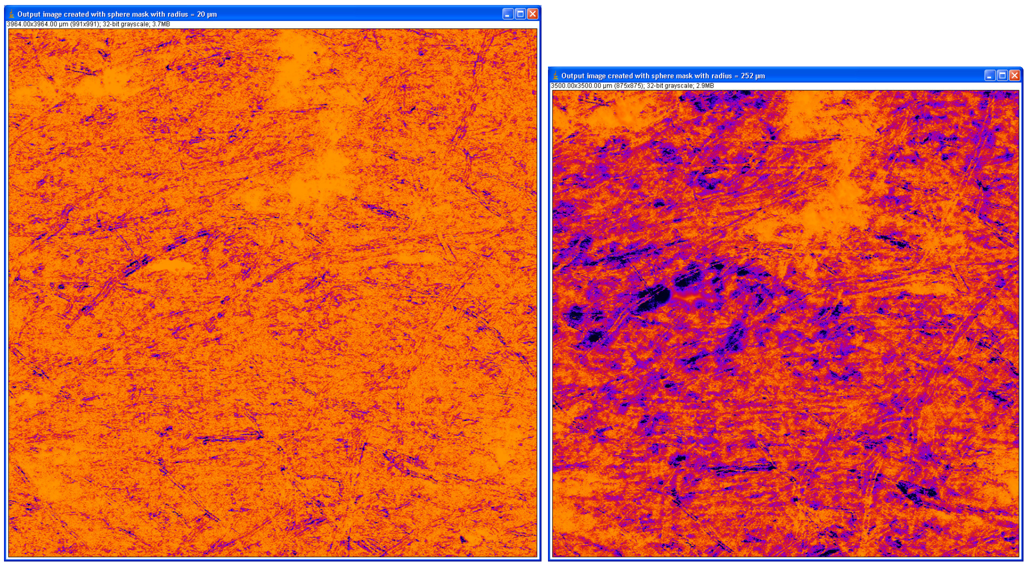

Output image created with sphere mask with radius = 20 µm
3964.00x3964.00 µm (991x991); 32-bit grayscale; 3.7MB
Output image created with sphere mask with radius = 252 µm
3500.00x3500.00 µm (875x875); 32-bit grayscale; 2.9MB

Figure 4-8: Output Image after Subtracting the Smoothed Image from the Original Image (radius=20 and 252 micron)

4.4 Distance Transform

One plausible alternative to the use of a rolling sphere algorithm (which is of large, $O(N^2 \times M^2)$ complexity) is the use of Distance transform. The concept of Distance transform was first introduced in Chapter 3.6.2.

One drawback connected to the use of Distance transform in the current context is that the use of Distance transform usually requires a discretized image. In the current context a Distance transform calculation can therefore only be performed *after* some preprocessing step has been applied to the original image / data set.

For the present diploma work, the most obvious solution was selected. That is, the image is disretized by converting the data set to a volumetric representation. One can look at this conversion as "slicing" the image into some number of aligned slices. Since the pixel value in the original data set is only related to height along the Z-axis, the resulting volumetric data representation contains several times as many voxels as there were pixels in the original image. The image "slices" themselves, however, are simply binary images (for convenience stored as 8-bit images containing only zero- and 255-valued pixels).

A first attempt at "slicing" the image in a meaningful way before running a distance transform algorithm was done with a new "DiscretizeZValues" plugin. It was clear that some integer approximation routine would be needed in order to yield data that could be feeded into a distance transform algorithm. This was concluded from the observations made by Borgefors [2]: "Using real-valued local distances is generally not computationally desirable". This first approach simply slices the image using as dimension the unit most applicable for the example image, microns. For some LLP images (those with 1 micron granularity in the x and y directions) this will give entirely cubic voxels, however for the present image (with 4 micron granularity in the x and y directions) the voxels will be slightly elongated in the x and y directions compared with the z direction.

The results of the first attempt is given in Figure 4-9, obtained by providing the image from Figure 4-12 as input to the "DiscretizeZValues" plugin. Shown first is the slice at pixel value -4.0 microns, with each image in sequence showing the slice at one micron further up in the image stack, ending at pixel value +7.0 microns. In these binary images, the white areas correspond to areas with material, and the black areas correspond to empty areas. Observing the images, it is clear that the areas first to gain black coverage in the sequence are the pores (as well as some smaller, noisy areas). It is therefore clear that these are the areas that will be assigned the largest value after running the distance transform, when looking at the top slice (Figure 4-10). The end result in Figure 4-10 was obtained by providing the sliced, binary image as input to the new 3D Distance transform plugin, "DistanceTransform3D".

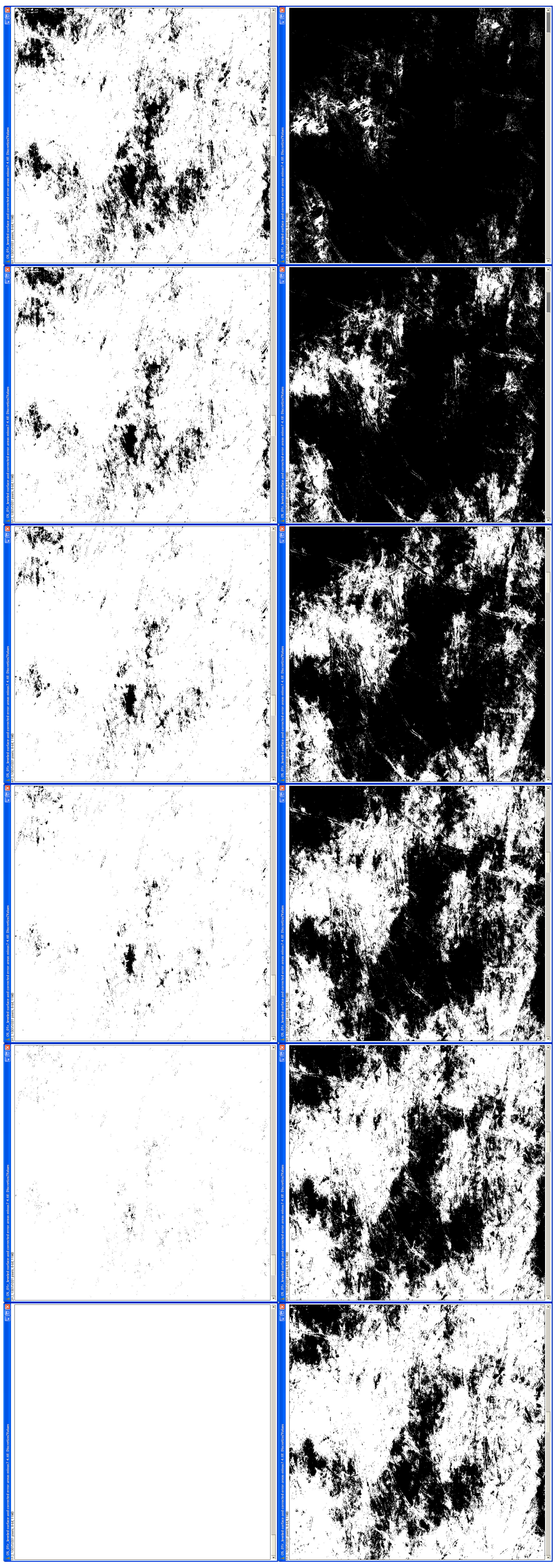


Figure 4-9: Image Sequence Showing all 12 Slices after running DiscretizeZValue

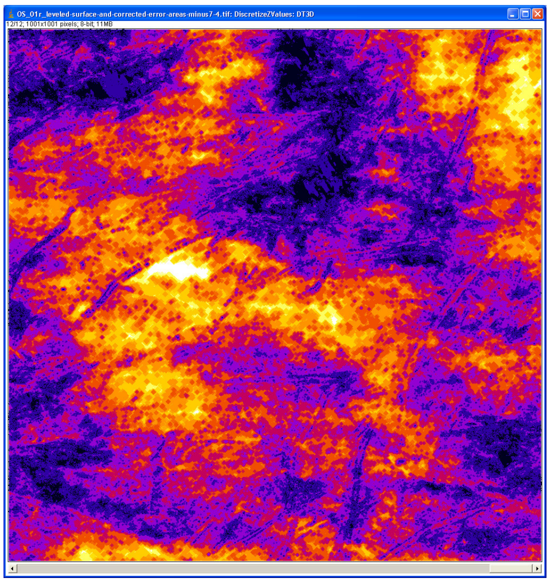

OS_01r_leveled-surface-and-corrected-error-areas-minus7-4.tif: DiscretizeZValues: DT3D
12/12; 1001x1001 pixels; 8-bit; 11MB

Figure 4-10: Distance Transform Results, Top Slice

4.5 Image Filtering Tutorial

In this section a practical application of the developed plugins is presented. Therefore, an explanation of how the images shown in Figures 4-11 through 4-16 was obtained is given next.

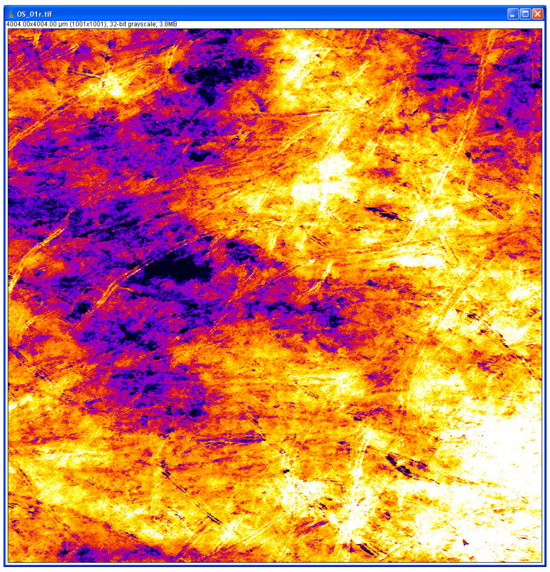


Figure 4-11: Image Acquired by Laser Profilometry

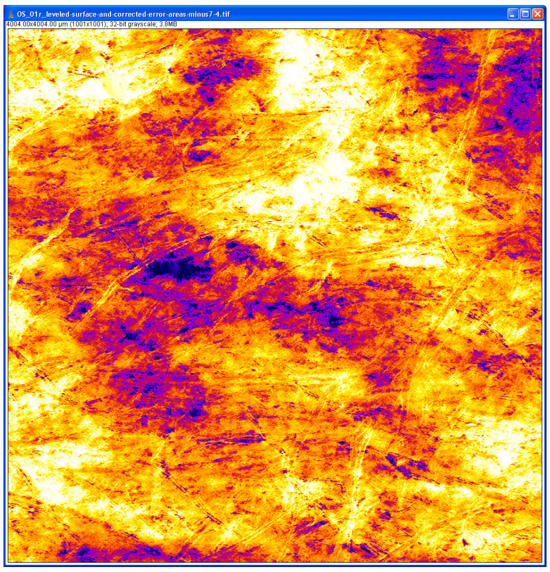

OS_01r_leveled-surface-and-corrected-error-areas-minus7-4.tif
4004.00x4004.00 µm (1001x1001); 32-bit grayscale; 3.8MB

Figure 4-12: Image Acquired by Laser Profilometry after Surface Leveling, Error Correction

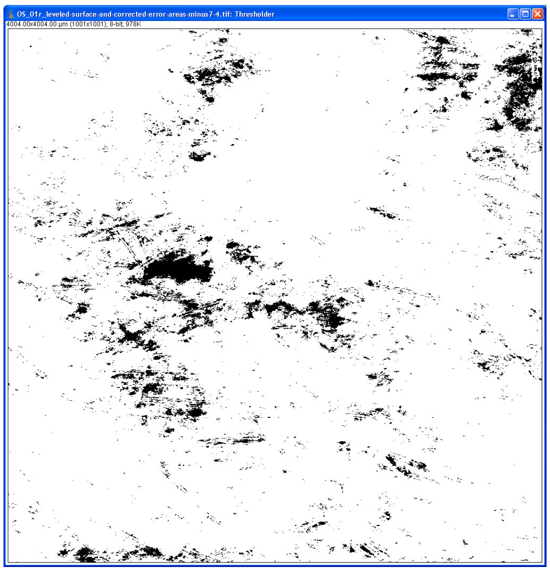

OS_01r_leveled-surface-and-corrected-error-areas-minus7-4.tif: Thresholder
4004.00x4004.00 µm (1001x1001); 8-bit; 978K

Figure 4-13: Image After Thresholding

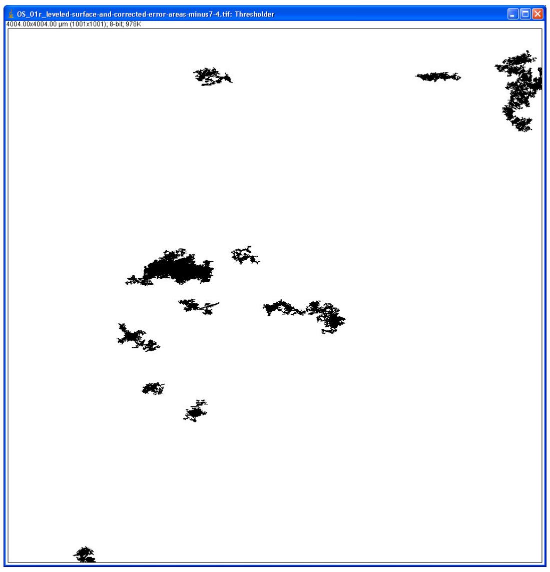

OS_01r_leveled-surface-and-corrected-error-areas-minus7-4.tif: Thresholder
4004.00x4004.00 µm (1001x1001); 8-bit; 978K

Figure 4-14: Image after Thresholding and Area Thresholding

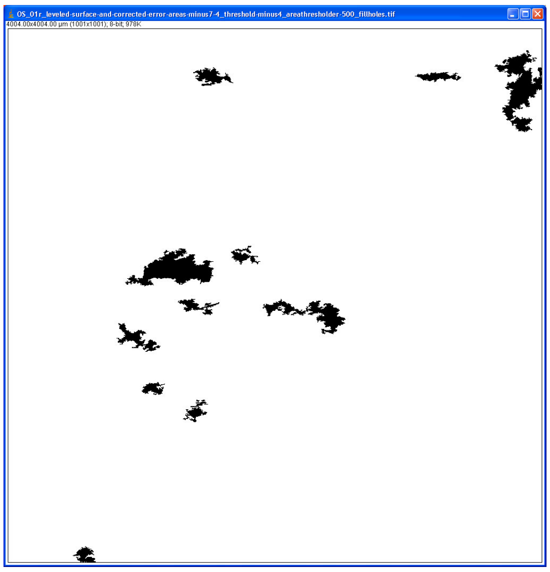

OS_01r_leveled-surface-and-corrected-error-areas-minus7-4_threshold-minus4_areathresholder-500_fillholes.tif
4004.00x4004.00 µm (1001x1001); 8-bit; 978K

Figure 4-15: Image after Thresholding, Area Thresholding, and Filling of Holes

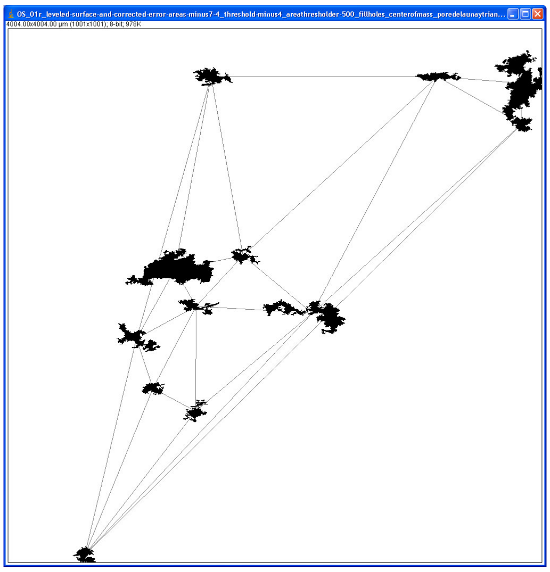

OS_01r_leveled-surface-and-corrected-error-areas-minus7-4_threshold-minus4_areathresholder-500_fillholes_centerofmass_poredelaunaytrian...
4004.00x4004.00 μm (1001x1001); 8-bit; 978K

Figure 4-16: Image after Delaunay Triangulation

First, Figure 4-11. This is the raw image retrieved by the use of Lehrman Laser Profilometry (LLP) [7]. The image is stored in the, quite uncommon, 32-bit floating point version of the TIFF format. Thankfully, ImageJ is one of the few image processing applications which include built-in support for this and other floating-point formats which is very useful for the present purposes. The use of a flointing-point format implies that a large range of values can be represented in this image format, and as such, the values at every (x, y) position in the image represents the measured LLP value directly. For instance, a value of -1.53 at position $(x, y) = (992, 372)$ indicates that the measured LLP value at $(992, 372)$ was in fact -1.53 microns, so the values in the images *are* in fact microns and can be used directly as physical values during processing in ImageJ. As mentioned in Chapter 3.3, LLP is able to register pixel values with an accuracy of about 1/100 of a micron, and as such, a floating point image representation is able to represent this sub-micron step size fairly well..

After importing the 32-bit TIFF image into ImageJ, the image was quite gray and uniform. In order to better visualize the LLP image, the "Image->Adjust->Brightness/Contrast..." command in ImageJ was used to manually set the min and max brightness values to -7.0 and +4.0 microns, respectively. The reason for choosing the values -7.0 and +4.0 were explained in Chapter 3.5.2. Then, in order to increase contrast between regions, the "Image/Lookup Tables" command was used to set the color map to "Fire" which is a color map with good contrast from low to high values (black through purple, red, orange, yellow to white).

Then, Figure 4-12. This is the same LLP image after some basic filtering with the SurfCharJ plugin (Chapter 2.5).

- "Level surface"
- "Correct error areas" (threshold for error correction: -7.0 and +4.0 microns)

The first filter, "Level surface", simply corrects any eventual tilting of the paper sample relative to the profilometer. The filter is further discussed in Chapter 3.5.1. In this particular example, it is evident in Figure 4-11 that the sample was tilted slightly downwards in the upper left corner, and correspondingly upwards in the lower right corner, since the resulting image is *generally* darker in the upper left corner and *generally* lighter in the lower right corner. After filtering, it is evident in Figure 4-12 that the surface has been leveled, and the upper left and lower right corners are no longer different in height compared with the remaining parts of the surface.

The second filter, "Correct error areas", takes care of some special cases in the image where LLP measurements are completely in error. The filter is further discussed in Chapter 3.5.2. Here, the threshold for error correction is set to -7.0 and +4.0 microns, which implies that any measurement below or above these thresholds should be replaced by a filtered, smoothed average of any neighboring measurements (pixel values in the image) which are inside the given threshold range.

A comprehensive discussion of the physical characteristics of LLP imaging that generates these error measurements is not in place here, as the topic of microscopy and image acquisition is not part of the present work. However, suffice to say, the error measurements are most common along fiber boundaries, and occur because of sudden change in angle between two measurement positions in the sample. A further and more detailed description of the physical properties of LLP imaging, as well as a more detailed discussion of the "Correct error areas" filter can be found in Chinga's paper on gloss assessment [7].

Then, the segmentation. This is shown in Figures 4-13 through 4-15. These figures show a first attempt at generating a segmentation of the image. The aim in this case is refering to generation of a reasonable *pore* segmentation. The segmentation itself is not too complicated, in order to quickly yield a sample image for further processing.

First, the image was thresholded through the use of a simple plugin written from scratch called "Thresholder". This plugin simply thresholds the 32-bit floating point image yielding a new, binary black-and-white image stored as an 8-bit image (for convenience) with only zero- and 255- valued pixels. A dialog box takes the threshold value as input. A tentative goal was in this case to generate a segmentation which included all pores as continous regions. The inclusion of all pore areas was the most important factor, as a lot of the noise could be removed later. In this case a reasonable value seemed to be about -4.0 microns, determined after some trial and error. This seemed to yield a good compromise between good segmentation and noise level. The segmentation with a threshold value of -4.0 microns is shown in Figure 4-13.

Second, a segmented image had now been generated. However, this segmented image had much noise which led to the need for more filtering operations. This was already mentioned above and was adressed partly by applying a plugin written from scratch called "AreaThresholder". This plugin takes as input a binary image (for convenience, an 8-bit image with only zero and 255 values), and thresholds *regions* based on their size. The end result is somewhat similar to median filtering, however the regions which end up being part of the result image are guaranteed to be unaltered by the filtering operation, which may *not* be guaranteed after the use of a median filter. The area thresholder relies on use of a flood fill algorithm in order to count the number of pixels in regions. There are two passes: One which decides which areas to discard, and another which actually discards those areas, setting them to 255 (white). The result after area thresholding with a value of 500 pixels (which equals 8000 microns^2 in the given image scale) is shown in Figure 4-14. A further discussion of the "AreaThresholder" plugin was presented in Chapter 4.1.

Third, now with the pores properly segmented, in order to give a clearer shape to pores while retining their outer boundary, the image is filtered with the "Process->Binary->Fill Holes" command in ImageJ, which removes holes from the already-segmented pores. It should be mentioned that most of these holes were actually much smaller in area than the parameter set for area thresholding (500 pixels = 8000 microns^2), however they were of the "opposite" type of the regions removed during area thresholding and were thus not removed during area thresholding. The result after filling of holes is shown in Figure 4-15.

Then, Figure 4-16. Here a Delaunay Triangulation (Chapter 3.8.1) of the pores has been overlaid on the image. Actually, it is not the pores themselves that have been triangulated, rather it is the point set consisting of the centers of mass of all 13 pores present in the image. This point set was generated by the use of the new "CenterOfMass" plugin. The point set may or may not be an accurate representation of the centerpoints of the pores, however it could be argued that it is accurate enough for the present purpose, which is to obtain a distance measure between neighboring pores. There are several disadvantages with using center of mass as the center definition for pores. For instance, some pores may be elongated, and so defining *one* single center point for the whole pore might be too crude an estimate. More importantly regarding center of mass, if a pore has a curved form the center of mass may well be at a point in the surface outside of the pore itself. This lies in the nature of the center of mass definition and so may or may not be acceptible for the purpose of pore center calculation depending on the task at hand.

4.6 Pore Neighborhood

This section discusses concepts related to the Modeling phase as presented in Chapter 3.8. It is mostly related to relationships *between* pores, and focuses on neighbors and neighborhoods of pores.

4.6.1 Neighbor

The first challenge is to define exactly *which* pores are neighbors. If uses the most pessimistic approach as a starting point, *all* other pores in an image are potentially neighboring pores. In the example image, shown in Figure 4-14, we observe that the successful segmentation of pores in the end yielded 12 segmented pores.

Note that in the upper right there seems to be a large single pore which has been segmented out, when in fact this pore has been segmented into 2 separate pores. In fact, the segmentation performed in that example does not include any provision for connecting nearby but disconnected segmented regions. It would from the example given here seem to be of great relevance if such a technique is included as part of the segmentation process, perhaps as an implementation of a morphological filtering.

4.6.2 Pore Representation

One problem that arises when trying to construct a relationship between pores (or more specifically, defining which pores are neighbors with which other pores), is the problem of *representing* segmented pores. One could use distance from the outer bound of a segmented pore, but in many cases this is not neccessary and unnecessarily complicates further processing.

The approach used in the present work is to reduce every region representing a segmented pore into a single point. The advantage of this approach is that the resulting image/data set is simplified as much as possible while still retaining the general structure of the distance relationships in the data set. Also, the analysis of this data set can now be done with known methods from computational geometry instead of ad hoc approaches. Therefore, in the current attempt at identifying potential neighbors for each pore, it seems as if a conversion from a segmented, binary image to a network graph representation is justified. An introduction to (weighted) network graphs is presented in the appendix of Cormen, Leiserson, and Rivest [11]. The first part of the conversion is done by the use of the new "CenterOfMass" plugin discussed briefly towards the end of Chapter 4.5, the second part of the conversion is handled by the Delaunay triangulation, described next.

4.6.3 Delaunay Triangulation

Treating the neighborhood analysis as a work in computational geometry leads to a wealth of possibilities for analysis.

The method chosen for generating a network graph representation in the present work is the method of *Delaunay triangulation*. According to the MATLAB documentation [15], Delaunay triangulation can be defined as follows:

"Given a set of data points, the Delaunay triangulation is a set of lines connecting each point to its natural neighbors. The Delaunay triangulation is related to the Voronoi diagram – the circle circumscribed about a Delaunay triangle has its center at the vertex of a Voronoi polygon."

Delaunay triangulation (and its dual, Voronoi diagram) is useful in areas as diverse as theoretical computational geometry, visualization problems, cartography, simulation of the growth of crystals, metallurgy, and in this case, assesment of neighborhood in surface topography analysis. Of course, surface topography analysis in itself is a varied topic – this was already mentioned briefly in Chapter 2.2.

The algorithm used for Delaunay triangulation in the present project is, as mentioned, suboptimal. However, it is still the algorithm of choice here as it is simple to understand and implement, while being more than fast enough for the present purposes.

```
DELAUNAY_TRIANGULATION:
for i=0 to pointlist_length
    for j=i+1 to pointlist_length
        for k=j+1 to pointlist_length
            for a=0 to pointlist_length
                if point a is not inside the circle passing
                through points i,j,k for any i,j,k not equal to a
                    add the triangle defined by points i,j,k
```

Also, an even more suboptimal but nevertheless simple algorithm was used for Voronoi diagram generation.

```
VORONOI_DIAGRAM:
for a=0 to pointlist_length
    pick color c at random
    for j=0 to image_height
        for i=0 to image_width
            b = pixel at (i,j)
            if b isn't already drawn OR a is closer than b's
                            currently assigned closest point
                assign a as b's closest point
                draw pixel b with color c
```

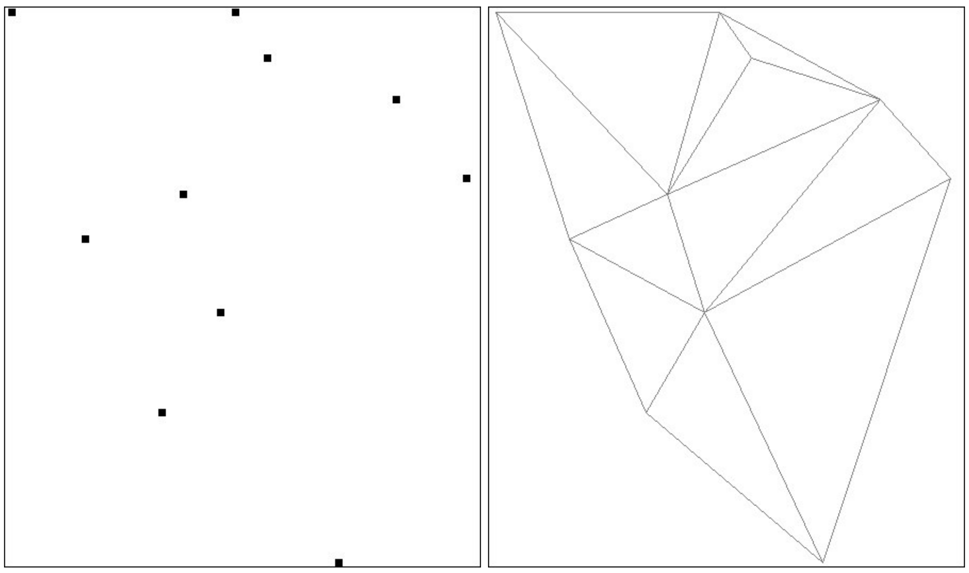


Figure 4-17: Set of Dots before and after Delaunay Triangulation

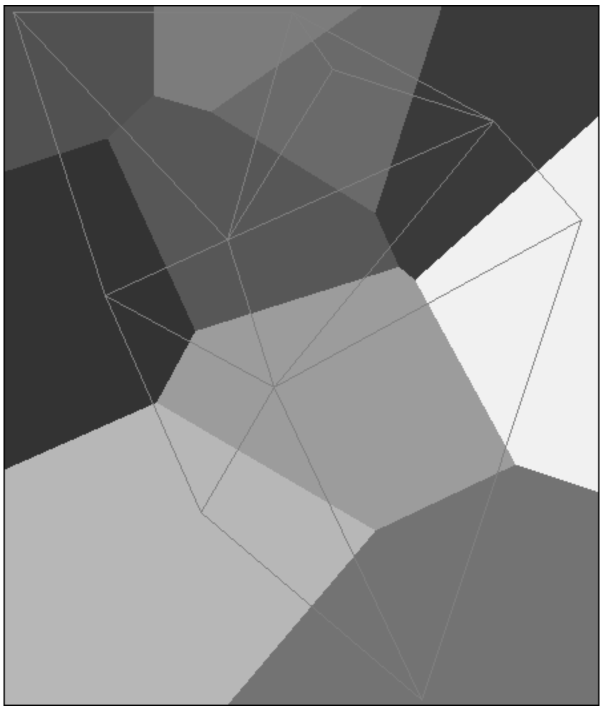


Figure 4-18: Set of Dots after Delaunay Triangulation (also with Voronoi Diagram)

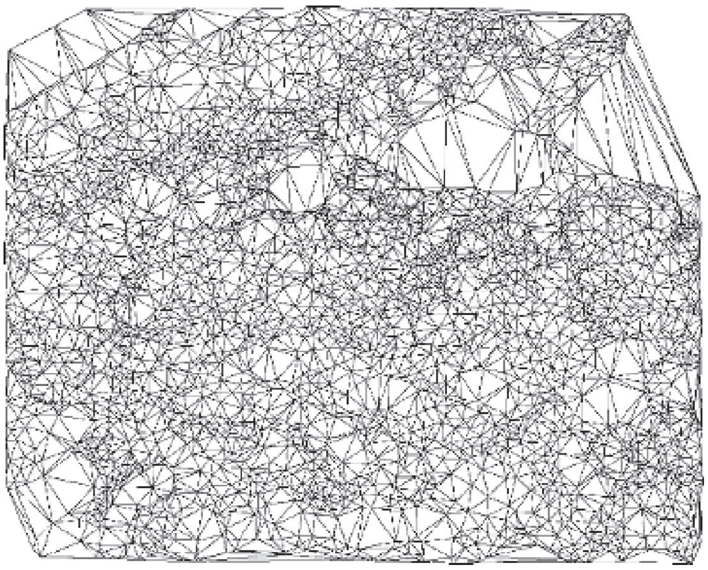


Figure 4-19: Triangulation at Macro Scale [4]

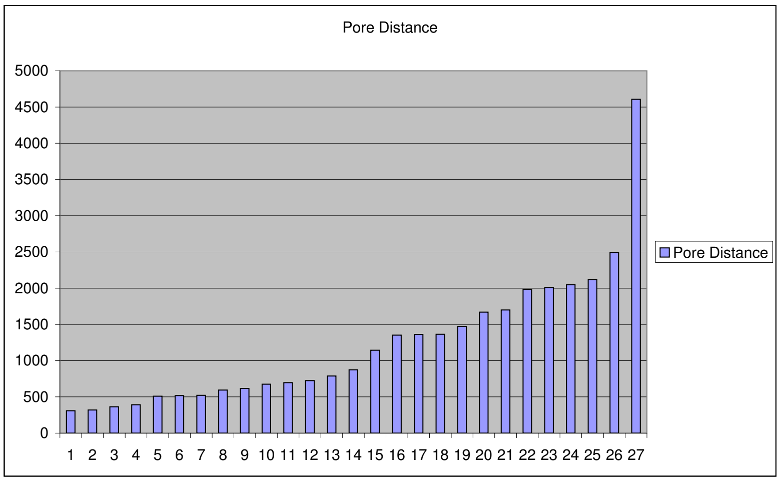


Figure 4-20: Pore Distances Given by the Delaunay Triangulation (in microns)

An example of a Delaunay triangulation and its corresponding Voronoi diagram of a simple data set is shown in Figures 4-17 and 4-18. If a large paper surface sample is Delaunay triangulated in the future, it is thought that the result might look similar to the macro scale triangulation data presented in Figure 4-19. It should be noted that Figure 4-19 is an example image from a different domain, *not* a real paper surface triangulation. The LLP images used in the present work did not lend themselves to macro scale triangulations since the surface area shown in each image is not very large. However, in the future other measurement methods could be used as a basis for macro scale triangulation. Alternatively, several overlapping LLP images could be combined before processing and triangulation, for instance by some mosaic/"stitching" technique as described in Forsyth and Ponce [12], and Szeliski and Shum [22, 23] and exemplified in Figure 4-21 (courtesy of The Applied Computer Science Group at Universität Bielefeld [16]).

Also, the measured pore neighbor distances in microns are shown in Figure 4-20. It is important to note that these distances simply reflect edge lengths in the Delaunay triangulation – some edges may be highly redundant. This will be discussed in the following sections.

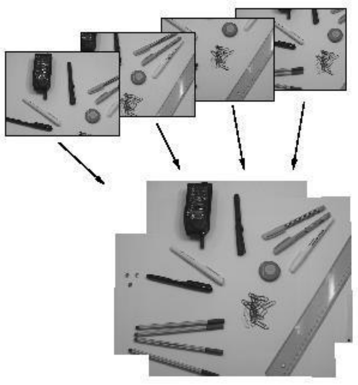
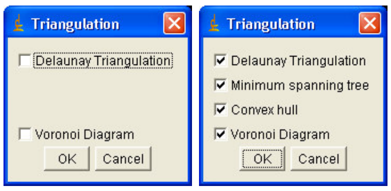


Figure 4-21: Mosaicing Example [16]

Lastly, the interface of the Triangulation plugin is shown in Figure 4-22. As can be seen in the window, the plugin also includes options for two other types of calculations in addition to the Delaunay triangulation and Voronoi diagram. These options are explained in the next sections.



Triangulation

☐ Delaunay Triangulation

☐ Voronoi Diagram

OK Cancel

Triangulation

☑ Delaunay Triangulation

☑ Minimum spanning tree

☑ Convex hull

☑ Voronoi Diagram

OK Cancel

Figure 4-22: Triangulation Interface, with no Options Enabled and all Options Enabled

4.6.4 Removing Edges after Delaunay Triangulation

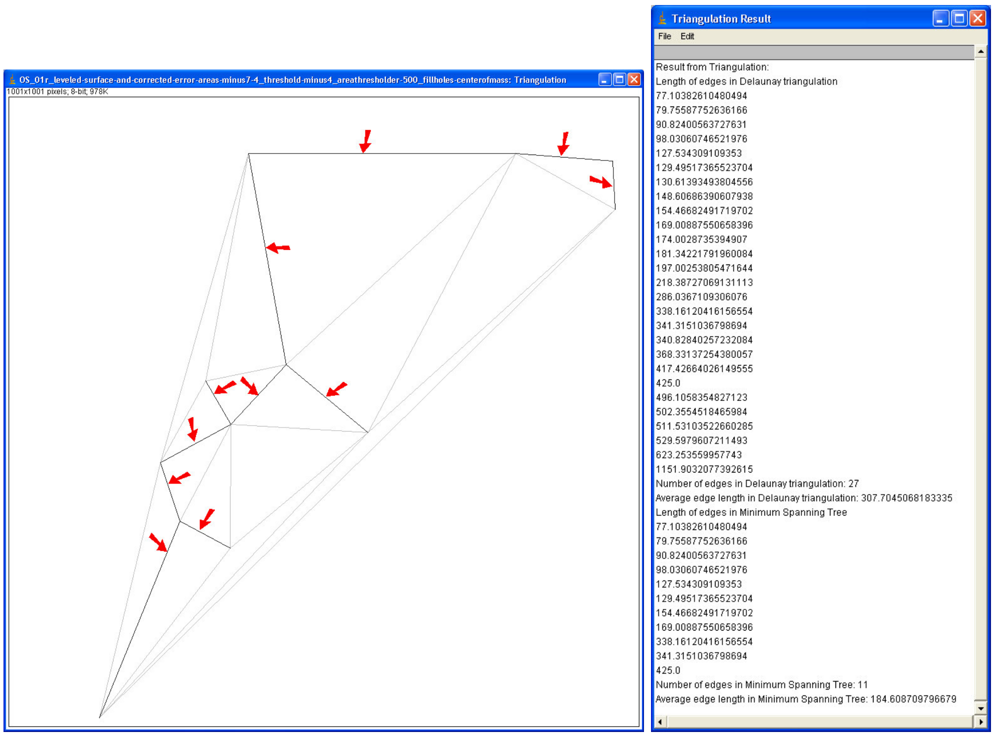
Even after Delaunay triangulation of the example LLP paper surface image it is evident from Figure 4-16 that further processing is needed in identifying relevant neighbors. Some of the edges do indeed represent relevant neighbors, however some do obviously not to a human observer.

The question is then: Which edges should be removed after triangulation? In order to answer this question it is neccessary to analyze properties of the different types of generated edges, and more specifically, determine which criteria should be met in order for the current edge under consideration to be removed safely.

Vertex Degree and Minimum Spanning Tree

One attribute of the Delaunay triangulation of particular note is the *degree* of the vertices (that is, pores) after triangulation. The degree of each vertex is simply the number of neighbors assigned to the corresponding pore. In a Delaunay triangulation each vertex has degree of at least two, since every vertex is a corner point of at least one triangle. This can cause problems if, for example, vertices/pores are close to colinear.

Some elementry graph theory can be helpful in finding a solution to the neighbor definition problem. One technique that can be useful is the minimum spanning tree as an ultimate form of reduction. An example minimum spanning tree superimposed on the current triangulation example (Figure 4-16) is shown in Figure 4-23. This spanning tree was generated by running the new "Triangulation" plugin with the minimum spanning tree calculation option checked. The algorithm used for calculating the minimum spanning tree is Prim's algorithm. Pseudocode for this algorithm is presented in Cormen, Leiserson, and Rivest [11].


OS_01r_leveled-surface-and-corrected-error-areas-minus7-4_threshold-minus4_areathresholder-500_fillholes-centerofmass: Triangulation
1001x1001 pixels; 8-bit; 978K
Triangulation Result
File
Edit

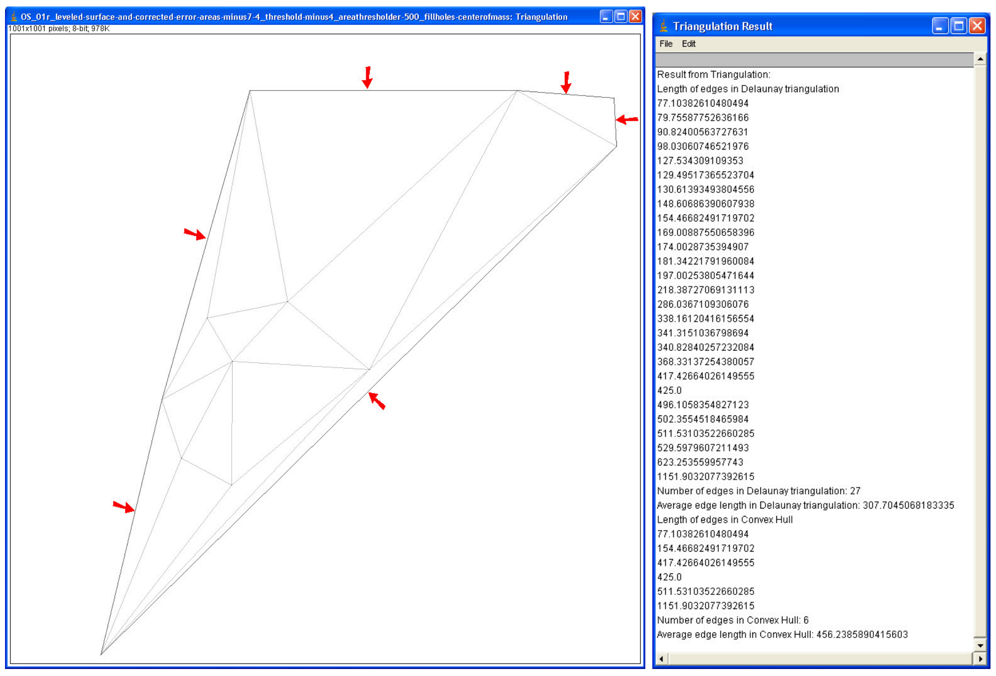
```
Result from Triangulation:
Length of edges in Delaunay triangulation
77.10382610480494
79.75587752636166
90.82400563727631
98.03060746521976
127.534309109353
129.49517365523704
130.61393493804556
148.60686390607938
154.46682491719702
169.00887550658396
174.0028735394907
181.34221791960084
197.00253805471644
218.38727069131113
286.0367109306076
338.16120416156554
341.3151036798694
340.82840257232084
368.33137254380057
417.42664026149555
425.0
496.1058354827123
502.3554518465984
511.53103522660285
529.5979607211493
623.253559957743
1151.9032077392615
Number of edges in Delaunay triangulation: 27
Average edge length in Delaunay triangulation: 307.7045068183335
Length of edges in Minimum Spanning Tree
77.10382610480494
79.75587752636166
90.82400563727631
98.03060746521976
127.534309109353
129.49517365523704
154.46682491719702
169.00887550658396
338.16120416156554
341.3151036798694
425.0
Number of edges in Minimum Spanning Tree: 11
Average edge length in Minimum Spanning Tree: 184.608709796679
```

Figure 4-23: Minimum Spanning Tree Example (Included Edges Marked with Arrows), with Output

Convex Hull

One of the natural places to start when considering *redundant* edges is to consider the convex hull of the pore network. As mentioned towards the end of Chapter 4.6.3 (in relation to Figure 4-19), it would be interesting to use triangulation at a macro scale in the future, however, this is not possible due to the small area coved by each of the present LLP images. The smallness of the LLP images also creates some problems when calculating a Delaunay triangulation on these small patches by themselves. Most importantly, the convex hull is always created when triangulating a patch (which may be part of some larger area). In the context of neighborhood analysis this will often look like an "artificial" boundary that does not represent any *real* neighbor relations. The reason why it is "artificial" is that this boundary would not have been present in a triangulation of a larger surface area covering more than just the patch area, since pores outside the patch area would then have been connected to the rest of the network. It is evident that the convex hull (boundary) is usually not important when evaluating pore distances and pore distance distributions. The convex hull has been marked out in Figure 4-24. Here, it is clear to a human observer that all edges that are part of the convex hull can be removed without removing any "real" neighbor relations.

The convex hull was generated by running the newly written "Triangulation" plugin with the convex hull option checked. The algorithm used for calculating the convex hull is Graham's Scan algorithm. Pseudocode for this algorithm is presented in Cormen, Leiserson, and Rivest [11].

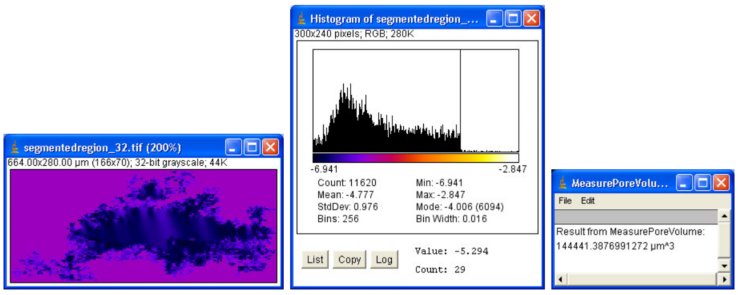

OS_01r_leveled-surface-and-corrected-error-areas-minus7-4_threshold-minus4_areathresholder-500_fillholes-centerofmass: Triangulation
1001x1001 pixels; 8-bit; 978K
Triangulation Result
File
Edit

```
Result from Triangulation:
Length of edges in Delaunay triangulation
77.10382610480494
79.75587752636166
90.82400563727631
98.03060746521976
127.534309109353
129.49517365523704
130.61393493804556
148.60686390607938
154.46682491719702
169.00887550658396
174.0028735394907
181.34221791960084
197.00253805471644
218.38727069131113
286.0367109306076
338.16120416156554
341.3151036798694
340.82840257232084
368.33137254380057
417.42664026149555
425.0
496.1058354827123
502.3554518465984
511.53103522660285
529.5979607211493
623.253559957743
1151.9032077392615
Number of edges in Delaunay triangulation: 27
Average edge length in Delaunay triangulation: 307.7045068183335
Length of edges in Convex Hull
77.10382610480494
154.46682491719702
417.42664026149555
425.0
511.53103522660285
1151.9032077392615
Number of edges in Convex Hull: 6
Average edge length in Convex Hull: 456.2385890415603
```

Figure 4-24: Convex Hull Example (Included Edges Marked with Arrows), with Output

4.7 Measure Pore Volume

The last plugin written, called "MeasurePoreVolume" is useful after performing a segmentation with e.g. "AreaThresholder". It takes as input a pore image, and asks for a threshold plane in microns – the threshold plane can e.g. be equal to the pixel value threshold used. The volume is calculated by first determining the voxel size in the 3D space of the surface, e.g. with a pixel delta width and height of 4.0 micron and a pixel delta depth of 1.0 micron, the voxel size is 16.0 micron^3. The pore image is then traversed and summed up by using the voxel size as the unit measurement, showing the result in a new window. An example is given in Figure 4-25, and complete measurement of all pores from the screenshot of Figure 4-2 is given in Table 4-1 (in order from left to right, then top to bottom, refering to the window layout in the screenshot). A threshold plane value of -4.0 micron was used for the measurements, as the pixel value threshold used for segmentation was -4.0 micron.


segmentedregion_32.tif (200%)
664.00x280.00 µm (166x70); 32-bit grayscale; 44K
Histogram of segmentedregion_...
300x240 pixels; RGB; 280K
-6.941
-2.847
List
Copy
Log
MeasurePoreVolu...
File
Edit

```
Count: 11620        Min: -6.941
Mean: -4.777        Max: -2.847
StdDev: 0.976       Mode: -4.006 (6094)
Bins: 256           Bin Width: 0.016

Value: -5.294
Count: 29
```

```
Result from MeasurePoreVolume:
144441.3876991272 µm^3
```

Figure 4-25: Segmented Region, its Histogram, and Pore Volume Measurement, Threshold Plane at -4.0 micron

Table 4-1: Pore Volume Measurements of Pores in Figure 4-2

Pore number	*Pore volume*
1	144441 micron^3
2	18612 micron^3
3	16390 micron^3
4	12798 micron^3
5	14241 micron^3
6	9654 micron^3
7	26781 micron^3
8	58406 micron^3
9	13896 micron^3
10	11727 micron^3
11	9239 micron^3
12	75240 micron^3

Chapter 5

Further Research

As is, the present work presents only a partially complete solution for paper quality and missing dot evaluation in the context of pore segmentation. Possible approaches have been presented as well as possible pointers as to how they can be best implemented and integrated in a complete system. Some of the most noteworthy possible approaches for future work are mentioned throughout both Chapter 3 and other places, and are summarized here:

- Development of a fully automatic error area correction routine
- Combination of several approaches for successful pore segmentation (instead of choosing only one)
- Adaptive thresholding combined with morphological operations in a feedback system
- Using several types of images as input to the modeling phase, and building of several new model-building components
- Completion of graph reduction of the Delaunay triangulation in order to yield a plausible neighborhood graph, moving beyond the minimum spanning tree and convex hull starting points
- Calculation and evaluation of distributions related to pore neighbor distances
- Further investigation of fractal dimension and similar shape measurements, also in three dimensions
- Developing components for the Measurement phase and the Quality Assesment phase

Chapter 6

Conclusion

Image analysis and image processing methods are becoming more and more useful to the pulp and paper research community. In this thesis a small subtopic within the huge research field of surface analysis was treated – the topic of pores in the paper surface, and segmentation and analysis of single pores as well as clusters of pores. So far it has not been entirely clear which knobs to turn in order to get a suitable segmentation, though simple methods have been available to obtain preliminary results that could later be replaced by more novel approaches.

In the first part of this thesis a general framework of a paper quality assesment system using automated image processing techniques was presented. The system would give a paper quality measure derived from an estimation of the number of *missing dots* that would occur when printing the paper. In line with the main topic of this thesis, only parts of the framework relating to missing dots due to pores and pore distance distribution are considered for now. Nevertheless, missing dots due to other factors such as fibers crossing could be included in future refinements of the system.

The area achieving the most progress in the course of this thesis was the study of relationships between neighboring pores. The approach presented here transforms raw LLP image data into a network structure that can later be reduced and turned into a weighted graph showing neighboring pores and their relationships. It is the hope of the author that further research will lead to models that give good neighbor graphs, and that these graphs later can be correlated to empirical measurements of missing dots after printing. Ultimately a model that gives good estimation of missing dots due to pore distance distribution is desired.

References

[1] C. Antoine, P. J. Mangin, J. L. Valade, M.-C. Belandand K. Chartier, and M. A. MacGregor. The influence of underlying paper surface structure on missing dots in gravure. In J. A. Bristow, editor, *Advances in Print Science and Technology*, volume 23, pages 401–414. IARIGAI, 1997.

[2] Gunilla Borgefors. On digital distance transforms in three dimensions. *Comput. Vis. Image Underst.*, 64(3):368–376, 1996.

[3] Gunilla Borgefors and Stina Svensson. Optimal Local Distances for Distance Transforms in 3D Using an Extended Neighbourhood. In *IWVF-4: Proceedings of the 4th International Workshop on Visual Form*, pages 113–122, London, UK, 2001. Springer-Verlag.

[4] Paul Bourke. Efficient triangulation algorithm suitable for terrain modelling, May 2005. `http://astronomy.swin.edu.au/~pbourke/terrain/triangulate/`.

[5] Gary Chinga. Printability of rotogravure papers as affected by their structural and chemical characteristics. Presentation, 2001.

[6] Gary Chinga. *Structural studies of LWC paper coating layers using SEM and image analysis techniques*. PhD thesis, NTNU, 2002.

[7] Gary Chinga. Detailed characterization of paper surface structure for gloss assessment. *Journal of pulp and paper science*, 30(11):222–227, August 2004.

[8] Gary Chinga. COST Action E32 home page, May 2005. `http://www.pfi.no/gary/COSTE32.htm`.

[9] Gary Chinga. SurfCharJ at Gary Chinga's private home page, May 2005. `http://home.online.no/~gary.c/IJ/SurfCharJ.htm`.

[10] Gary Chinga, Øyvind Gregersen, and Robert Dougherty. Paper surface characterisation by laser profilometry and image analysis. *Journal of microscopy and analysis*, 84:5–7, 2003.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001.

[12] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference, 2002.

[13] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[14] Rune Holmstad. *Methods for paper structure characterisation by means of image analysis*. PhD thesis, NTNU, 2004.

[15] MathWorks. MATLAB Function Reference, Delaunay Triangulation, May 2005. `http://www.mathworks.com/access/helpdesk/help/techdoc/ref/delaunay.html`.

[16] Birgit Möller. Mosaic Project web page, Applied Computer Science Group, Universität Bielefeld, June 2005. `http://www.techfak.uni-bielefeld.de/ags/ai/projects/mosaic/`.

[17] U.S. National Institute of Health. ImageJ, Image Processing and Analysis in Java, home page, March 2005. `http://rsb.info.nih.gov/ij/`.

[18] L. Donnell Payne. Automating road surface analysis. In *SAC '92: Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing*, pages 944–950, New York, NY, USA, 1992. ACM Press.

[19] Daniel Sage and Michael Unser. Easy Java programming for teaching image-processing. In *Proceedings of the 2001 IEEE International Conference on Image Processing*, volume 3, pages 298–301. IEEE, 2001.

[20] Ida-Maria Sintorn and Gunilla Borgefors. Weighted distance transforms for volume images digitized in elongated voxel grids. *Pattern Recogn. Lett.*, 25(5):571–580, 2004.

[21] Stina Svensson and Mattias Aronsson. Using distance transform based algorithms for extracting measures of the fiber network in volume images of paper. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 33(4):562–571, 2003.

[22] Richard Szeliski. Video mosaics for virtual environments. *IEEE CG&A*, pages 22–30, March 1996.

[23] Richard Szeliski and Heung-Yeung Shum. Creating full view panoramic image mosaics and environment maps. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 251–258, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[24] R. E. Walpole, R. H. Myers, and S. L. Myers. *Probability and Statistics for Engineers and Scientists*. Prentice-Hall, Inc., New Jersey, sixth edition, 1998.

[25] Dave Wilson. Fractal Dimension by Box Counting, June 2005. `http://www.ees.nmt.edu/ davew/P362/boxcnt.htm`.

Appendix A

Code Listing

A.1 AreaThresholder_.java

```
import ij.*;
import ij.gui.GenericDialog;
import ij.gui.NewImage;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;
import ij.process.ImageStatistics;

import java.awt.Checkbox;
import java.awt.Choice;
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.TextField;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

/**
 * @author Svein Fidjestøl
 *
 * Area Thresholder plugin for ImageJ. Takes as input a binary black-and-white
 * image, in 8-bit grayscale format, with white as the background
 * color and black as the foreground (region) color. Outputs a binary black-and-white
 * image with regions smaller than the threshold removed. Also the regions can
 * be backtranslated into the original, unsegmented 32-bit floating point image
 * if this is provided as input.
 *
 * The flood fill algorithm used borrows heavily from Floodfiller.java
 * and BinaryFiller.java in the ImageJ 1.34n code base (also used in
 * the CenterOfMass plugin)
 */
public class AreaThresholder_ implements PlugInFilter, ItemListener {

        // variables from Floodfiller.java
        int maxStackSize = 500; // will be increased as needed
        int[] stack = new int[maxStackSize];
        int stackSize;
        int max;

        /**
         * Original flood fill algorithm from Floodfiller.java. Takes
         * as input a seed point and fills scanline by scanline.
         *
         * @param x X-coordinate of seed point
```

```
     * @param y Y-coordinate of seed point
     * @param ip The image being processed
     * @param ip2 A second image to write to
     * @return True if everything went OK, false if seed point already filled
     */
    public boolean fill(int x, int y, ImageProcessor ip, ImageProcessor ip2) {
            int width = ip.getWidth();
            int height = ip.getHeight();
            int color = ip.getPixel(x, y);
            ip.drawLine(x, y, x, y);
            int newColor = ip.getPixel(x, y);
            ip.putPixel(x, y, color);
            if (color==newColor) return false;
            stackSize = 0;
            push(x, y);
            while(true) {
                    int coordinates = pop();
                    if (coordinates ==-1) return true;
                    x = coordinates&0xffff;
                    y = coordinates>>16;
                    int x1 = x; int x2 = x;
                    while (ip.getPixel(x1,y)==color && x1>=0) x1--; // find start of scan-line
                    x1++;
                    while (ip.getPixel(x2,y)==color && x2<width) x2++;  // find end of scan-line
                    x2--;
                    ip.drawLine(x1,y, x2,y); // fill scan-line
                    if(ip2 != null) {
                            ip2.drawLine(x1,y, x2, y); // fill scan-line
                    }
                    boolean inScanLine = false;
                    for (int i=x1; i<=x2; i++) { // find scan-lines above this one
                            if (!inScanLine && y>0 && ip.getPixel(i,y-1)==color)
                                    {push(i, y-1); inScanLine = true;}
                            else if (inScanLine && y>0 && ip.getPixel(i,y-1)!=color)
                                    inScanLine = false;
                    }
                    inScanLine = false;
                    for (int i=x1; i<=x2; i++) { // find scan-lines below this one
                            if (!inScanLine && y<height-1 && ip.getPixel(i,y+1)==color)
                                    {push(i, y+1); inScanLine = true;}
                            else if (inScanLine && y<height-1 && ip.getPixel(i,y+1)!=color)
                                    inScanLine = false;
                    }
            }
    }

    // variables that store boundary values for the current region
    private int currentLowX;
    private int currentHighX;
    private int currentLowY;
    private int currentHighY;

    /**
     * Same as fill method but also returns a pixel count reflecting
     * how many pixels were filled.
     *
     * @param x X-coordinate of seed point
     * @param y Y-coordinate of seed point
     * @param ip The image being processed
     * @return True if everything went OK, false if seed point already filled
     */
    public int fill2(int x, int y, ImageProcessor ip) {
            int count = 0;
            int width = ip.getWidth();
            int height = ip.getHeight();
            int color = ip.getPixel(x, y);
```

```
        ip.drawLine(x, y, x, y);
        int newColor = ip.getPixel(x, y);
        ip.putPixel(x, y, color);
        if (color==newColor) return -1; // returned false in original method
        stackSize = 0;
        push(x, y);
        while(true) {
            int coordinates = pop();
            if (coordinates ==-1) return count; // returned true in original method
            x = coordinates&0xffff;
            y = coordinates>>16;
            int x1 = x; int x2 = x;
            while (ip.getPixel(x1,y)==color && x1>=0) x1--; // find start of scan-line
            x1++;
            while (ip.getPixel(x2,y)==color && x2<width) x2++;  // find end of scan-line
            x2--;
            if(x1 < currentLowX) {
                currentLowX = x1;
            }
            if(x2 > currentHighX) {
                currentHighX = x2;
            }
            if(y < currentLowY) {
                currentLowY = y;
            }
            if(y > currentHighY) {
                currentHighY = y;
            }
            ip.drawLine(x1,y, x2,y); // fill scan-line
            count += x2-x1+1;
            boolean inScanLine = false;
            for (int i=x1; i<=x2; i++) { // find scan-lines above this one
                if (!inScanLine && y>0 && ip.getPixel(i,y-1)==color)
                    {push(i, y-1); inScanLine = true;}
                else if (inScanLine && y>0 && ip.getPixel(i,y-1)!=color)
                    inScanLine = false;
            }
            inScanLine = false;
            for (int i=x1; i<=x2; i++) { // find scan-lines below this one
                if (!inScanLine && y<height-1 && ip.getPixel(i,y+1)==color)
                    {push(i, y+1); inScanLine = true;}
                else if (inScanLine && y<height-1 && ip.getPixel(i,y+1)!=color)
                    inScanLine = false;
            }
        }
    }

    /**
     * Helper method for fill algorithm
     *
     * @param x X-coordinate of current seed point
     * @param y Y-coordinate of current seed point
     */
    final void push(int x, int y) {
        stackSize++;
        if (stackSize==maxStackSize) {
            int[] newStack = new int[maxStackSize*2];
            System.arraycopy(stack, 0, newStack, 0, maxStackSize);
            stack = newStack;
            maxStackSize *= 2;
        }
        stack[stackSize-1] = x + (y<<16);
    }

    /**
     * Helper method for fill algorithm
```

```
         *
         * @param x X-coordinate of current seed point
         * @param y Y-coordinate of current seed point
         */
        final int pop() {
                if (stackSize==0)
                        return -1;
                else {
                        int value = stack[stackSize-1];
                        stackSize--;
                        return value;
                }
        }

        /**
         * Stores whether the lookup table is inverted
         */
        protected boolean backgroundIsZero;

        /**
         * The input image
         */
        private ImagePlus imp;

        // Components
        private Checkbox cbBinaryRegion;
        private Checkbox cbOriginalRegion;
        private Choice chOriginalImage;
        private Label lblDisplayRange;
        private Label lblFillerValue;
        private TextField tfFillerValue;
        private Label lblLow;
        private TextField tfLow;
        private Label lblHigh;
        private TextField tfHigh;

        /**
         * Standard ImageJ plugin setup method
         *
         * @param arg Plugin arguments
         * @param imp The input image
         */
        public int setup(String arg, ImagePlus imp) {

                if (imp==null)
                {IJ.noImage(); return DONE;}
                this.imp = imp;
                ImageStatistics stats=imp.getStatistics();
                if (stats.histogram[0]+stats.histogram[255]!=stats.pixelCount){
                        IJ.error("8-bit binary image (0 and 255) required.");
                        return DONE;
                }
                backgroundIsZero = Prefs.blackBackground;
                if (imp.isInvertedLut())
                        backgroundIsZero = !backgroundIsZero;
                return IJ.setupDialog(imp, DOES_8G);
        }

        /**
         * Standard ImageJ plugin run method. Quite monolithic for simplicity purposes
         *
         * @param ip Selected region for processing
         */
        public void run(ImageProcessor ip) {
                // Calibration info is not used for parameters in this implementation,
                // however it is retained in the result image
```

```
// Get the classname without trailing underscore
String className = getClass().getName();
if(className.charAt(className.length()-1) == '_') {
        className = className.substring(0, className.length()-1);
}

// Get some info from the input image
byte[] old_pixels = (byte[]) ip.getPixels();
int width = ip.getWidth();
int height = ip.getHeight();

// Retrieve list of open images
int[] wList = WindowManager.getIDList();

// Get threshold
int threshold;
GenericDialog gd = new GenericDialog(className);
gd.setLayout(new GridLayout(12,1));

// Default value 500 (Should ideally be customized
// for Calibration (microns))
Label lblThreshold = new Label("Threshold (in pixels)");
TextField tfThreshold = new TextField("500", 0);

// Draw GUI
boolean binaryRegion = false;
boolean originalRegion = false;
cbBinaryRegion = new Checkbox("Generate a new binary image for each region",
                              binaryRegion);
cbOriginalRegion = new Checkbox("Generate a new image for each region with " +
                                "data from the original image", originalRegion);
chOriginalImage = new Choice();
for (int i=0; i<wList.length; i++) {
        ImagePlus imp = WindowManager.getImage(wList[i]);
        if (imp!=null && imp.getWidth() == width && imp.getHeight() == height)
                chOriginalImage.add(imp.getTitle());
        else
                chOriginalImage.add("");
}
lblFillerValue = new Label("Filler value");
tfFillerValue = new TextField("-4.0", 0);
lblLow = new Label("Brightness range (low):");
tfLow = new TextField("-7.0");
lblHigh = new Label("Brightness range (high):");
tfHigh = new TextField("4.0");

cbOriginalRegion.addItemListener(this);

gd.add(lblThreshold);
gd.add(tfThreshold);
gd.add(cbBinaryRegion);
gd.add(cbOriginalRegion);
gd.add(chOriginalImage);

gd.add(lblFillerValue);
gd.add(tfFillerValue);
gd.add(lblLow);
gd.add(tfLow);
gd.add(lblHigh);
gd.add(tfHigh);

lblThreshold.setVisible(true);
tfThreshold.setVisible(true);
cbBinaryRegion.setVisible(true);
cbOriginalRegion.setVisible(true);
```

```
chOriginalImage.setVisible(false);

lblFillerValue.setVisible(false);
tfFillerValue.setVisible(false);
lblLow.setVisible(false);
tfLow.setVisible(false);
lblHigh.setVisible(false);
tfHigh.setVisible(false);

gd.showDialog();
if (gd.wasCanceled()) {
        return;
}

// Retrieve parameters from dialog
threshold = Integer.parseInt(tfThreshold.getText());
binaryRegion = cbBinaryRegion.getState();
originalRegion = cbOriginalRegion.getState();
float fillerValue = Float.parseFloat(tfFillerValue.getText());
double low = Double.parseDouble(tfLow.getText());
double high = Double.parseDouble(tfHigh.getText());

//original converted to white particles (part of original Floodfiller.java algorithm)
if (!backgroundIsZero)
        ip.invert();

// Get some info from the original image (32-bit floating point format)
ImagePlus orig_imp;
float[] orig_pixels = null;
if(originalRegion) {
        orig_imp = WindowManager.getImage(wList[chOriginalImage.getSelectedIndex()]);
        orig_pixels = (float[]) orig_imp.getProcessor().getPixels();
}

// Generate a new, empty image
ImagePlus new_imp = NewImage.createByteImage("temp1", width, height, 1,
                                             NewImage.FILL_WHITE);
new_imp.setCalibration(imp.getCalibration());
ImageProcessor new_ip = new_imp.getProcessor();
// Copy the old image into the new one
byte[] pixels = (byte[]) new_imp.getProcessor().getPixels();
for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
                pixels[x + y * width] = old_pixels[x + y*width];
        }
}

// Generate a new, empty image once more
ImagePlus new_imp2 = NewImage.createByteImage("temp2", width, height, 1,
                                              NewImage.FILL_WHITE);
new_imp2.setCalibration(imp.getCalibration());
ImageProcessor new_ip2 = new_imp2.getProcessor();
// Copy the old image into the new one
byte[] pixels2 = (byte[]) new_imp2.getProcessor().getPixels();
for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
                pixels2[x + y * width] = old_pixels[x + y*width];
        }
}

// Main algorithm part
int regionCount = 0;
int offset;
for(int j = 0; j < height; j++) {
        offset = j*height;
        for(int i = 0; i < width; i++) {
```

```
if(pixels[offset+i] == -1) { // check for black
        // Measure size of region by filling, and store
        // low and high coordinates
        currentLowX = currentLowY = Integer.MAX_VALUE;
        currentHighX = currentHighY = Integer.MIN_VALUE;
        int count = fill2(i,j,new_ip);
        if(count < threshold) {
                fill(i,j,ip, null);
        } else if(binaryRegion || originalRegion) { // Generate
                                        // segmented output images
                regionCount++;
                // bounding box
                int bb_width = currentHighX - currentLowX;
                int bb_height = currentHighY - currentLowY;

                // Generate a new, empty image
                ImagePlus bb_imp = NewImage.createByteImage(
                        "Region number " + regionCount +
                        ", BoundingBoxInPixels([" + currentLowX +
                        "-" + currentHighX + "],[" + currentLowY +
                        "-" + currentHighY + "]), binary", width,
                        height, 1, NewImage.FILL_WHITE);
                bb_imp.setCalibration(imp.getCalibration());
                ImageProcessor bb_ip = bb_imp.getProcessor();

                // Fill both new_ip2 and at the same locations in bb_ip
                fill(i,j,new_ip2,bb_ip);
                bb_ip.setRoi(currentLowX,currentLowY,
                bb_width,bb_height);
                bb_imp.setProcessor(null, bb_ip.crop());
                byte[] bb_pixels =
                (byte[]) bb_imp.getProcessor().getPixels();
                if(binaryRegion) {
                        bb_imp.show();
                } else {
                        // prepare for garbage collection
                        bb_ip = null;
                        bb_imp = null;
                }
                if(originalRegion) {
                 // Backtranslate segmented regions to original image
                 ImagePlus bborig_imp =
                 NewImage.createFloatImage("Region number " +
                 regionCount + ", BoundingBoxInPixels([" +
                 currentLowX + "-" + currentHighX + "],[" +
                 currentLowY + "-" + currentHighY + "]), original",
                 bb_width, bb_height, 1, NewImage.FILL_WHITE);
                 bborig_imp.setCalibration(imp.getCalibration());
                 float[] bborig_pixels =
                        (float[]) bborig_imp.getProcessor().
                        getPixels();
                 for (int x = 0; x < bb_width; x++) {
                  for (int y = 0; y < bb_height; y++) {
                   if(bb_pixels[y*bb_width + x] == 0) {
                    bborig_pixels[x + y * bb_width] =
                        orig_pixels[(currentLowY+y)*width +
                        (currentLowX+x)];
                   } else {
                    // Which value to use at empty pixels?
                    //bborig_pixels[x + y * bb_width] =
                        //Float.POSITIVE_INFINITY;
                    //bborig_pixels[x + y * bb_width] =
                        //(float) 0.0;
                    //bborig_pixels[x + y * bb_width] =
                        //Float.NaN;
                    bborig_pixels[x + y * bb_width] =
                                                        fillerValue;
                                                      }
                                                     }
                                                    }
                                                    ImageProcessor bborig_ip = bborig_imp.getProcessor();
                                                    bborig_ip.setMinAndMax(low, high);
                                                    bborig_imp.show();
                                                    Executer e = new Executer("Fire", bborig_imp);
                                                    e.run();
                                                   }
                                            }

                                    }
                            }
                    }

                    //return to original state (part of original Floodfiller.java algorithm)
                    if (!backgroundIsZero)
                            ip.invert();

                    // prepare for garbage collection
                    new_imp = null;
                    new_ip = null;
            }

            /**
             * Callback method used for hiding and showing
             * components in the dialog
             *
             * @param e The ItemEvent
             */
            public void itemStateChanged(ItemEvent e) {
                    if(e.getItemSelectable() == cbOriginalRegion) {
                            if(e.getStateChange() == ItemEvent.SELECTED) {
                                    chOriginalImage.setVisible(true);
                                    lblFillerValue.setVisible(true);
                                    tfFillerValue.setVisible(true);
                                    lblLow.setVisible(true);
                                    tfLow.setVisible(true);
                                    lblHigh.setVisible(true);
                                    tfHigh.setVisible(true);
                            } else {
                                    chOriginalImage.setVisible(false);
                                    lblFillerValue.setVisible(false);
                                    tfFillerValue.setVisible(false);
                                    lblLow.setVisible(false);
                                    tfLow.setVisible(false);
                                    lblHigh.setVisible(false);
                                    tfHigh.setVisible(false);
                            }
                    }
            }

}
```

A.2 CalculateAverageSurfaceHeight_.java

```
import java.text.DecimalFormat;

import ij.ImagePlus;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;
import ij.text.TextWindow;

/**
 * @author Svein Fidjestøl
 *
 * CalculateAverageSurfaceHeight plugin for ImageJ. Meant to be used
 * for surface images. Takes as input a 32-bit float image and prints
 * out the average height value in a new window.
 *
 */
public class CalculateAverageSurfaceHeight_ implements PlugInFilter {

        /**
         * Standard ImageJ plugin setup method
         *
         * @param arg Plugin arguments
         * @param imp The input image
         */
        public int setup(String arg, ImagePlus imp) {
                return DOES_32;
        }

        /**
         * Standard ImageJ plugin run method. Quite monolithic for simplicity purposes
         *
         * @param ip Selected region for processing
         */
        public void run(ImageProcessor ip) {
                // Get the classname without trailing underscore
                String className = getClass().getName();
                if(className.charAt(className.length()-1) == '_') {
                        className = className.substring(0, className.length()-1);
                }

                // Get some info from the input image
                float[] pixels = (float[]) ip.getPixels();
                int width = ip.getWidth();
                int height = ip.getHeight();

                // Sum
                double cumsum = 0.0;
                double size = height*width;
                for(int j = 0; j < height; j++) {
                        for(int i = 0; i < width; i++) {
                                cumsum += pixels[j*width+i];
                        }
                }

                // Calculate the average
                TextWindow tw = new TextWindow(
                        "CalculateAverageSurfaceHeight Result",
                        "Average surface height: " +
                        new DecimalFormat(
                                "###.####").format(cumsum/((double)(height*width))),
                        400, 150);
        }
}
```

A.3 CenterOfMass_.java

```
import ij.*;
import ij.gui.NewImage;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;
import ij.process.ImageStatistics;

import java.awt.Point;
import java.util.ArrayList;
import java.util.Iterator;


/**
 * @author Svein Fidjestøl
 *
 * CenterOfMass plugin for ImageJ. Takes as input a binary black-and-white
 * image, in 8-bit grayscale format, with white as the background
 * color and black as the foreground (region) color. Outputs a binary black-and-white
 * image with every black region in the image substituted with a single
 * pixel at the position which is the region's center of mass point.
 *
 * The flood fill algorithm used borrows heavily from Floodfiller.java
 * and BinaryFiller.java in the ImageJ 1.34n code base (also used in
 * the AreaThresholder plugin)
 */
public class CenterOfMass_ implements PlugInFilter {

        // variables from Floodfiller.java
        int maxStackSize = 500; // will be increased as needed
        int[] stack = new int[maxStackSize];
        int stackSize;
        int max;

        /**
         * Original flood fill algorithm from Floodfiller.java. Takes
         * as input a seed point and fills scanline by scanline.
         *
         * @param x X-coordinate of seed point
         * @param y Y-coordinate of seed point
         * @param ip The image being processed
         * @param ip2 A second image to write to
         * @return True if everything went OK, false if seed point already filled
         */
        public boolean fill(int x, int y, ImageProcessor ip) {
                int width = ip.getWidth();
                int height = ip.getHeight();
                int color = ip.getPixel(x, y);
                ip.drawLine(x, y, x, y);
                int newColor = ip.getPixel(x, y);
                ip.putPixel(x, y, color);
                if (color==newColor) return false;
                stackSize = 0;
                push(x, y);
                while(true) {
                        int coordinates = pop();
                        if (coordinates ==-1) return true;
                        x = coordinates&0xffff;
                        y = coordinates>>16;
                        int x1 = x; int x2 = x;
                        while (ip.getPixel(x1,y)==color && x1>=0) x1--; // find start of scan-line
                        x1++;
                        while (ip.getPixel(x2,y)==color && x2<width) x2++;  // find end of scan-line
                        x2--;
                        ip.drawLine(x1,y, x2,y); // fill scan-line
```

```
                    boolean inScanLine = false;
                    for (int i=x1; i<=x2; i++) { // find scan-lines above this one
                            if (!inScanLine && y>0 && ip.getPixel(i,y-1)==color)
                                    {push(i, y-1); inScanLine = true;}
                            else if (inScanLine && y>0 && ip.getPixel(i,y-1)!=color)
                                    inScanLine = false;
                    }
                    inScanLine = false;
                    for (int i=x1; i<=x2; i++) { // find scan-lines below this one
                            if (!inScanLine && y<height-1 && ip.getPixel(i,y+1)==color)
                                    {push(i, y+1); inScanLine = true;}
                            else if (inScanLine && y<height-1 && ip.getPixel(i,y+1)!=color)
                                    inScanLine = false;
                    }
            }
    }

    /**
     * Same as fill method but also returns a pixel count reflecting
     * how many pixels were filled.
     *
     * @param x X-coordinate of seed point
     * @param y Y-coordinate of seed point
     * @param ip The image being processed
     * @return True if everything went OK, false if seed point already filled
     */
    public ArrayList fill2(int x, int y, ImageProcessor ip) {
            ArrayList pointList = new ArrayList();
            //int count = 0;
            int width = ip.getWidth();
            int height = ip.getHeight();
            int color = ip.getPixel(x, y);
            ip.drawLine(x, y, x, y);
            int newColor = ip.getPixel(x, y);
            ip.putPixel(x, y, color);
            if (color==newColor) return null; //false
            stackSize = 0;
            push(x, y);
            while(true) {
                    int coordinates = pop();
                    if (coordinates ==-1) return pointList; //true
                    x = coordinates&0xffff;
                    y = coordinates>>16;
                    int x1 = x; int x2 = x;
                    while (ip.getPixel(x1,y)==color && x1>=0) x1--; // find start of scan-line
                    x1++;
                    while (ip.getPixel(x2,y)==color && x2<width) x2++;  // find end of scan-line
                    x2--;
                    ip.drawLine(x1,y, x2,y); // fill scan-line
                    //count += x2-x1+1;
                    for(int i = x1; i <= x2; i++) {
                            pointList.add(new Point(i,y));
                    }
                    boolean inScanLine = false;
                    for (int i=x1; i<=x2; i++) { // find scan-lines above this one
                            if (!inScanLine && y>0 && ip.getPixel(i,y-1)==color)
                                    {push(i, y-1); inScanLine = true;}
                            else if (inScanLine && y>0 && ip.getPixel(i,y-1)!=color)
                                    inScanLine = false;
                    }
                    inScanLine = false;
                    for (int i=x1; i<=x2; i++) { // find scan-lines below this one
                            if (!inScanLine && y<height-1 && ip.getPixel(i,y+1)==color)
                                    {push(i, y+1); inScanLine = true;}
                            else if (inScanLine && y<height-1 && ip.getPixel(i,y+1)!=color)
                                    inScanLine = false;
```

```
                }
            }
        }

        /**
         * Helper method for fill algorithm
         *
         * @param x X-coordinate of current seed point
         * @param y Y-coordinate of current seed point
         */
        final void push(int x, int y) {
                stackSize++;
                if (stackSize==maxStackSize) {
                        int[] newStack = new int[maxStackSize*2];
                        System.arraycopy(stack, 0, newStack, 0, maxStackSize);
                        stack = newStack;
                        maxStackSize *= 2;
                }
                stack[stackSize-1] = x + (y<<16);
        }

        /**
         * Helper method for fill algorithm
         *
         * @param x X-coordinate of current seed point
         * @param y Y-coordinate of current seed point
         */
        final int pop() {
                if (stackSize==0)
                        return -1;
                else {
                        int value = stack[stackSize-1];
                        stackSize--;
                        return value;
                }
        }

        /**
         * Stores whether the lookup table is inverted
         */
        protected boolean backgroundIsZero;

        /**
         * The input image
         */
        private ImagePlus imp;

        /**
         * Standard ImageJ plugin setup method
         *
         * @param arg Plugin arguments
         * @param imp The input image
         */
        public int setup(String arg, ImagePlus imp) {

                if (imp==null)
                {IJ.noImage(); return DONE;}
                this.imp = imp;
                ImageStatistics stats=imp.getStatistics();
                if (stats.histogram[0]+stats.histogram[255]!=stats.pixelCount){
                        IJ.error("8-bit binary image (0 and 255) required.");
                        return DONE;
                }
                backgroundIsZero = Prefs.blackBackground;
                if (imp.isInvertedLut())
                        backgroundIsZero = !backgroundIsZero;
                return IJ.setupDialog(imp, DOES_8G);
        }

        /**
         * Standard ImageJ plugin run method.
         *
         * @param ip Selected region for processing
         */
        public void run(ImageProcessor ip) {
                // Get the classname without trailing underscore
                String className = getClass().getName();
                if(className.charAt(className.length()-1) == '_') {
                        className = className.substring(0, className.length()-1);
                }

                // original converted to white particles (part of original Floodfiller.java algorithm)
                if (!backgroundIsZero)
                        ip.invert();

                byte[] old_pixels = (byte[]) ip.getPixels();
                int width = ip.getWidth();
                int height = ip.getHeight();

                // Generate a new, empty image
                ImagePlus new_imp = NewImage.createByteImage(imp.getTitle() + ": " + className,
                width, height, 1, NewImage.FILL_WHITE);
                new_imp.setCalibration(imp.getCalibration());
                ImageProcessor new_ip = new_imp.getProcessor();
                // Copy the old image into the new one
                byte[] pixels = (byte[]) new_imp.getProcessor().getPixels();
                for (int x = 0; x < width; x++) {
                        for (int y = 0; y < height; y++) {
                                pixels[x + y * width] = old_pixels[x + y*width];
                        }
                }

                // Generate a new, empty image once more
                ImagePlus new_imp2 = NewImage.createByteImage(imp.getTitle() + ": " + className,
                width, height, 1, NewImage.FILL_WHITE);
                new_imp2.setCalibration(imp.getCalibration());
                ImageProcessor new_ip2 = new_imp2.getProcessor();

                // Main algorithm start
                int offset;
                for(int j = 0; j < height; j++) {
                        offset = j*height;
                        for(int i = 0; i < width; i++) {
                                if(pixels[offset+i] == -1) { // check for black
                                        ArrayList pointList = fill2(i,j,new_ip);

                                        // Sum
                                        Point p;
                                        long cumsum_x = 0;
                                        long cumsum_y = 0;
                                        int numPoints = 0;
                                        for(Iterator it = pointList.iterator(); it.hasNext();) {
                                                numPoints++;
                                                p = (Point) it.next();
                                                cumsum_x += p.x;
                                                cumsum_y += p.y;
                                        }

                                        // Calculate midpoint
                                        int geom_midpoint_x = (int) (cumsum_x / numPoints);
                                        int geom_midpoint_y = (int) (cumsum_y / numPoints);
                                        // Draw it
                                        new_ip2.drawPixel(geom_midpoint_x, geom_midpoint_y);
                                }
                        }
                }

                //return to original state (part of original Floodfiller.java algorithm)
                if (!backgroundIsZero)
                        ip.invert();

                new_imp2.show();

                // prepare for garbage collection
                new_imp = null;
                new_ip = null;
        }

}
```

A.4 DiscretizeZValues_.java

```java
import ij.IJ;
import ij.ImagePlus;
import ij.gui.NewImage;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;
import ij.process.ImageStatistics;


/**
 * @author Svein Fidjestøl
 *
 * DiscretizeZValues plugin for ImageJ. Takes as input a 32-bit
 * surface image and "slices" the image in unit increments, from the lowest
 * integer value to the highest rounded integer value detected in the image.
 * The output is an image stack of binary images where white pixels
 * correspond to filled areas and black pixels correspond to empty areas.
 *
 */
public class DiscretizeZValues_ implements PlugInFilter {

        /** The input image */
        private ImagePlus imp;

        /**
         * Standard ImageJ plugin setup method
         *
         * @param arg Plugin arguments
         * @param imp The input image
         */
        public int setup(String arg, ImagePlus imp) {
                this.imp = imp;
                return DOES_32;
        }

        /**
         * Standard ImageJ plugin run method. Simple plugin so
         * everything is done in here
         *
         * @param ip Selected region for processing
         */
        public void run(ImageProcessor ip) {
                // Get some colors
                IJ.run("Fire");

                // Get the classname without trailing underscore
                String className = getClass().getName();
                if(className.charAt(className.length()-1) == '_') {
                        className = className.substring(0, className.length()-1);
                }

                // Get width and height of input image
                int width = imp.getWidth();
                int height = imp.getHeight();
                float[] pixels = (float[]) ip.getPixels();

                // Get rounded integer min and max of input image
                ImageStatistics stats = imp.getStatistics();
                int min = (int) Math.round(stats.min);
                int max = (int) Math.round(stats.max);
                int slices = max - min;

                // Generate a new, empty image
                ImagePlus new_imp = NewImage.createByteImage(imp.getTitle() + ": " + className,
                width, height, slices+1, NewImage.FILL_BLACK);

                // Slice the image
                float pixel;
                byte discretized_pixel;
                for(int j = 0; j < height; j++) {
                        for(int i = 0; i < width; i++) {
                                pixel = pixels[j*width+i];

                                // Round as last step, else it doesn't work correctly for negative
                                // pixel values
                                discretized_pixel = (byte) Math.round(pixel - min);
                                for(int k = 0; k <= discretized_pixel; k++) {
                                        ((byte[]) new_imp.getStack().getPixels(k+1))[j*width+i] =
                                        (byte) 0xff;
                                }
                        }
                }

                // Show the new, sliced image
                new_imp.show();
        }

}
```

A.5 DistanceTransform3D_.java

```
// Original GNU copyright message follows    ——sveinfid June 13, 2005
/* Makes a 3D discrete distance transform. Uses ImageJ.

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.

Written by Maria Axelsson, ported to ImageJ/Java by
Jens Bache-Wiig <jensbw%at%gmail.com> and
Per Christian Henden <perchrh%at%pvv.org>

Further customized for use in paper surface analysis by Svein Fidjestøl

*/

import ij.IJ;
import ij.ImagePlus;
import ij.gui.NewImage;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;

/**
 * @author Svein Fidjestøl (original code: Jens Bache-Wiig, Per Christian Henden
 *         and Maria Axelsson)
 *
 * 3D Distance Transform plugin for ImageJ. Takes as input voxel data in the
 * form of a sliced binary black-and-white image (as e.g. generated by the
 * DiscretizeZValues plugin) where white pixels
 * correspond to filled areas and black pixels correspond to empty areas.
 * Calculates the 3D Distance Transform from this voxel data and outputs
 * a sliced 8-bit gray level image.
 */
public class DistanceTransform3D_ implements PlugInFilter{

        /** The input image */
        private ImagePlus imRef_old;

        /**
         * Standard ImageJ plugin setup method
         *
         * @param arg Plugin arguments
         * @param imp The input image
         */
        public int setup(String arg, ImagePlus imp) {
                if (arg.equals("about")){
                        showAbout();
                        return DONE;
                }
                imRef_old = imp;
                return DOES_8G;
        }

        /**
```

```
     * Standard ImageJ plugin run method. Quite monolithic for simplicity purposes
     *
     * @param ip Selected region for processing
     */
    public void run(ImageProcessor ip) {
            // Get the classname without trailing underscore
            String className = getClass().getName();
            if(className.charAt(className.length()-1) == '_') {
                    className = className.substring(0, className.length()-1);
            }

            // Get some info from the input image
            final int width =  ip.getWidth();
            final int height = ip.getHeight();
            final int depth = imRef_old.getStackSize()+2;

            //final int a = 3, b = 4, c = 3, d = 4, e = 5;  //alternate weights
            final int a = 3, b = 4, c = 5, d = 3, e = 7;

            final int[] wf = new int[] { e, d, e, d, c, d, e, d, e,b, a, b, a, 255, 255,
            255, 255, 255, };
            final int[] wb = new int[] { 255, 255, 255, 255, 255, a, b, a, b,e, d, e, d,
            c, d, e, d, e, };
            int[] slask = new int[2 * 3 * 3];

            // Generate a new, empty image
            ImagePlus imRef = NewImage.createByteImage(imRef_old.getTitle() + ": " +
            className, width, height, depth, NewImage.FILL_BLACK);
            imRef.setCalibration(imRef_old.getCalibration());
            // Copy the old image into the new one slice by slice and
            // generate padding slices at top and bottom of stack
            byte[] pixels;
            byte[] pixels_old;
            pixels = (byte[]) imRef.getStack().getPixels(1);
            for (int x = 0; x < width; x++) {
                    for (int y = 0; y < height; y++) {
                            pixels[x + y * width] = (byte) 0x0;
                    }
            }
            pixels = (byte[]) imRef.getStack().getPixels(depth);
            for (int x = 0; x < width; x++) {
                    for (int y = 0; y < height; y++) {
                            pixels[x + y * width] = (byte) 0xff;
                    }
            }
            for (int z = 0; z < depth-2; z++) {
                    pixels = (byte[]) imRef.getStack().getPixels(z+2);
                    pixels_old = (byte[]) imRef_old.getStack().getPixels(z+1);
                    for (int x = 0; x < width; x++) {
                            for (int y = 0; y < height; y++) {
                                    pixels[x + y * width] = (byte) ((byte) 0xff ^
                                    (byte) pixels_old[x + y * width]);
                            }
                    }
            }

            //Border pixels are ignored to simplify the convolutions below
            /*for (int z = 0; z < depth; z++) {
                    for (int x = 0; x < width; x++) {
                            for (int y = 0; y < height; y++) {

                                    if(z==0 || x==0 || y==0 || z==depth-1 ||
                                    y==height-1 || x==width-1) {
                                            ((byte[]) imRef.getStack().getPixels(z+1))[x +
                                            y * width]
                                            = (byte) 0x0;
```

```
                        }

                }
        }
}*/

//Forward iteration
for (int z = 1; z < depth-1; z++) {
        IJ.showProgress(z, 2*depth-2);
        for (int x = 1; x < width-1; x++) {
                for (int y = 1; y < height-1; y++) {

                        for (int k = -1; k < 1; k++) {
                                for (int j = -1; j < 2; j++) {
                                        for (int i = -1; i < 2; i++) {
                                                int slaskindex =
                                                (i+1) + (j+1) * 3 + (k+1) * 3 * 3;
                                                int pixel = 0xff &
                                                ((byte[]) imRef.getStack().
                                                getPixels(
                                                        z + k+1))[(x+i) +
                                                        (y+j) * width];
                                                slask[slaskindex]=
                                                pixel + wf[slaskindex];
                                        }
                                }
                        }
                        int minval = slask[0]; //the lowest value so far
                        for (int i = 1; i < slask.length; i++) {
                                if((slask[i])<minval)minval=(slask[i]);
                        }

                        int pixel = 0xff & ((byte[]) imRef.getStack().getPixels(
                                z+1))[(x) + (y) * width];
                        if(pixel>minval) ((byte[]) imRef.getStack().getPixels(
                                z+1))[x + y * width] = (byte) (minval&0xff);
                }
        }
}

//Backward iteration
for (int z = depth-2; z > 0; z--){
        IJ.showProgress(2*depth-z, 2*depth-2);
        for (int x = width-2; x > 0; x--){
                for (int y = height-2; y >0 ; y--){

                        for (int k = 0; k < 2; k++) {
                                for (int j = -1; j < 2; j++) {
                                        for (int i = -1; i < 2; i++) {
                                                int slaskindex = (i+1) + (j+1) * 3
                                                + (k) * 3 * 3;
                                                int pixel = 0xff &
                                                ((byte[]) imRef.getStack().
                                                getPixels(
                                                        z + k + 1))[(x+i) +
                                                        (y+j) * width];
                                                slask[slaskindex]= pixel
                                                + wb[slaskindex];
                                        }
                                }
                        }

                        int minval = slask[0]; //the lowest value so far
                        for (int i = 1; i < slask.length; i++) {
                                if((slask[i])<minval)minval=(slask[i]);
                        }
```

```
                                        int pixel = 0xff & ((byte[]) imRef.getStack().getPixels(
                                                z+1))[(x) + (y) * width];
                                        if(pixel>minval) ((byte[]) imRef.getStack().getPixels(
                                                z+1))[x + y * width] = (byte) (minval&0xff);
                                }
                        }
                }

                // Delete padding slices and show result
                // Generate a new, empty image
                ImagePlus imRef2 = NewImage.createByteImage(imRef_old.getTitle() +
                ": " + className, width, height, depth-2, NewImage.FILL_BLACK);
                imRef2.setCalibration(imRef_old.getCalibration());
                // Copy the old image into the new one
                byte[] pixels2;
                byte[] pixels1;
                for (int z = 0; z < depth-2; z++) {
                        pixels2 = (byte[]) imRef2.getStack().getPixels(z+1);
                        pixels1 = (byte[]) imRef.getStack().getPixels(z+2);
                        for (int x = 0; x < width; x++) {
                                for (int y = 0; y < height; y++) {
                                        pixels2[x + y * width] = pixels1[x + y * width];
                                }
                        }
                }
                imRef2.show();
        }

        void showAbout() {
                IJ.showMessage("About DT3D...",
                                "This plug-in filter calculates the 3D distances transform " +
                                "of a binary image with " +
                                "white (255) as background and black (0) as foreground.\n");
        }

} //class
```

A.6 FractalDimension_.java

```
import ij.ImagePlus;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;
import ij.text.TextWindow;

/**
 * @author Svein Fidjestøl
 *
 * FractalDimension plugin for ImageJ.  Takes as input a binary black-and-white
 * image, in 8-bit grayscale format, with white as the background
 * color and black pixels as regions. Prints out the result of
 * box counting which is neccessary for calculating the
 * fractal dimension of the binary image. Box counting is done by starting
 * with 1x1 regions, increasing to 2x2, 4x4, 8x8 etc up to the maximum size
 * allowed in the input image.
 *
 * Inspired by the tutorial of the fractal dimension by box counting
 * web page at
 * http://www.ees.nmt.edu/~davew/P362/boxcnt.htm
 *
 */
public class FractalDimension_ implements PlugInFilter {

        /**
         * Standard ImageJ plugin setup method
         *
         * @param arg Plugin arguments
         * @param imp The input image
         */
        public int setup(String arg, ImagePlus imp) {
                // TODO Auto-generated method stub
                return DOES_8G;
        }

        /**
         * Standard ImageJ plugin run method.
         *
         * @param ip Selected region for processing
         */
        public void run(ImageProcessor ip) {
                // Get some info from the input image
                byte[] pixels = (byte[]) ip.getPixels();
                int width = ip.getWidth();
                int height = ip.getHeight();

                // number of iterations as limited by the input image
                int iterations = (int) (Math.log(width)/Math.log(2));
                int numboxes[] = new int[iterations];
                int count[] = new int[iterations];
                int boxsize[] = new int[iterations];

                // Do the counting
                int currentboxsize, y_steps, x_steps;
                for(int it = 0; it < iterations; it++) {
                        currentboxsize = boxsize[it] = (int) Math.pow(2,it);
                        y_steps = height / currentboxsize;
                        x_steps = width / currentboxsize;
                        numboxes[it] = x_steps*y_steps;
                        for(int j = 0; j < y_steps; j++) {
                                for(int i = 0; i < x_steps; i++) {
                                        innerloops:
                                        for(int j2 = 0; j2 < currentboxsize; j2++) {
                                                for(int i2 = 0; i2 < currentboxsize; i2++) {
                                                    // Check for black pixel
                                                    if(pixels[((j*currentboxsize+j2)*width)+
                                                    (i*currentboxsize+i2)] == (byte) 0) {
                                                            count[it]++;
                                                            break innerloops;
                                                    }
                                            }
                                    }
                            }
                    }
            }

            // Generate result string
            String result = "";
            for(int i = 0; i < iterations; i++) {
                    result += "\nboxsize: " +       boxsize[i] + "\tcount: " + count[i];
            }

            // Show in a result window
            TextWindow tw = new TextWindow("FractalDimension Result",
            "Results from FractalDimension box counting" + result, 400, 150);
    }

}
```

A.7 MeasurePoreVolume_.java

```
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.TextField;

import ij.IJ;
import ij.ImagePlus;
import ij.gui.GenericDialog;
import ij.measure.Calibration;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;
import ij.text.TextWindow;

/**
 * @author Svein Fidjestøl
 *
 * MeasurePoreVolume plugin for ImageJ. Measures the volume
 * of a single pore in microns.
 */
public class MeasurePoreVolume_ implements PlugInFilter {

        /** The input image */
        private ImagePlus imp;

        /**
         * Standard ImageJ plugin setup method
         *
         * @param arg Plugin arguments
         * @param imp The input image
         */
        public int setup(String arg, ImagePlus imp) {
                if (arg.equals("about")){
                        showAbout();
                        return DONE;
                }
                this.imp = imp;
                return DOES_32;
        }

        /**
         * Standard ImageJ plugin run method.
         *
         * @param ip Selected region for processing
         */
        public void run(ImageProcessor ip) {
                // Get some info from the input image
                float[] pixels = (float[]) ip.getPixels();
                int width = ip.getWidth();
                int height = ip.getHeight();

                Calibration cal = imp.getCalibration();
                String unit = cal.getUnit();
                if(!unit.equals("um")) { // microns
                        IJ.showMessage("Measurement unit for image must be um"); // microns
                        return;
                }

                // Read in threshold plane value
                GenericDialog gd = new GenericDialog("MeasurePoreVolume");
                gd.setLayout(new GridLayout(2,2));
                Label lblThresholdPlaneValue = new Label("Threshold plane value (in um): ");
                // microns
                TextField tfThresholdPlaneValue = new TextField("−4.0",0);
                gd.add(lblThresholdPlaneValue);
                gd.add(tfThresholdPlaneValue);
                lblThresholdPlaneValue.setVisible(true);
                tfThresholdPlaneValue.setVisible(true);
                gd.showDialog();
                if (gd.wasCanceled()) {
                        return;
                }

                // Get the threshold value
                float t = Float.parseFloat(tfThresholdPlaneValue.getText());

                double pd = cal.pixelDepth;
                double pw = cal.pixelWidth;
                double ph = cal.pixelHeight;
                double voxelvolume = pd*pw*ph;

                double volume = 0.0;
                int offset;
                float pixel;
                for(int j = 0; j < height; j++) {
                        offset = j*width;
                        for(int i = 0; i < width; i++) {
                                pixel = pixels[offset+i];
                                volume += (t-pixel)*voxelvolume;
                        }
                }
                TextWindow tw = new TextWindow("MeasurePoreVolume Result",
                "Result from MeasurePoreVolume:\n" + volume + " " + unit + "^3", 400, 150);
        }

        /**
         * About window.
         */
        public void showAbout() {
                IJ.showMessage("About MeasurePoreVolume...",
                                "Measures the volume of a single pore in microns^3");
        }
}
```

A.8 RollingBallNew_.java

```
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.TextField;

import ij.IJ;
import ij.ImagePlus;
import ij.gui.GenericDialog;
import ij.measure.Calibration;
import ij.plugin.filter.PlugInFilter;
import ij.process.FloatProcessor;
import ij.process.ImageProcessor;

/**
 * @author Svein Fidjestøl
 *
 * RollingBallNew plugin for ImageJ. A new implementation of
 * the rolling ball filtering algorithm. Takes as input a 32-bit float
 * image and outputs
 * 1) The original input image, cropped to the same size as the finished,
 *    filtered image
 * 2) The sphere mask image
 * 3) Rolling ball mask value image
 * 4) Rolling ball filtered/smoothed image
 * 5) Output image: Original image with filtered image subtracted
 *    (image 1 minus image 4)
 * If a rolling ball of large enough radius is used, image 5 should contain
 * an image that contains important pores and creases in the image and flattens
 * out the remaining areas suitably (including smaller and unintereing
 * pores and creases).
 */
public class RollingBallNew_ implements PlugInFilter {

        /** The input image */
        private ImagePlus imp;

        /**
         * Standard ImageJ plugin setup method
         *
         * @param arg Plugin arguments
         * @param imp The input image
         */
        public int setup(String arg, ImagePlus imp) {
                this.imp = imp;
                return DOES_32;
        }

        /**
         * Standard ImageJ plugin run method.
         *
         * @param ip Selected region for processing
         */
        public void run(ImageProcessor ip) {
                // Checks for correct unit and stores scale
                Calibration cal = imp.getCalibration();
                String unit = cal.getUnit();
                if(!unit.equals("um")) { // microns
                        IJ.showMessage("Measurement unit for image must be um"); // microns
                        return;
                }
                int scale = (int) Math.round(cal.pixelWidth);

                // Read in radius of the rolling ball with a dialog box
                GenericDialog gd = new GenericDialog("RollingBallNew");
```

```
gd.setLayout(new GridLayout(2,2));
Label lblRadius = new Label("Rolling ball radius (in microns): ");
TextField tfRadius = new TextField("5",0);
Label lblRadiusComment = new Label("Radius in microns must be divisable by " +
scale);
gd.add(lblRadius);
gd.add(tfRadius);
gd.add(lblRadiusComment);
lblRadius.setVisible(true);
tfRadius.setVisible(true);
lblRadiusComment.setVisible(true);
gd.showDialog();
if (gd.wasCanceled()) {
        return;
}

// Get the rolling ball radius and diameter and adjust for scale
int r = Integer.parseInt(tfRadius.getText());
if(r % scale != 0) {
        GenericDialog gd2 = new GenericDialog("Error");
        gd2.setLayout(new GridLayout(1,1));
        Label lblErrorMessage = new Label("Radius in microns not divisable by " +
        scale);
        gd2.add(lblErrorMessage);
        lblErrorMessage.setVisible(true);
        gd2.showDialog();
        return;
}
r = r / scale;

int d = 2*r;
int mw = d+1; // width of mask is 1 pixel more than diameter so
              // mask width becomes odd-valued

float d_float = d;
float r_float = r;

// Create mask in the form of a sphere.
float maxValueInImage = (float) ip.getMax();
float[] sphereMask = new float[mw*mw];
int offset1, offset2;
float i_float, j_float;
float t1, t2;
float tmp;
float x_sq, y_sq;
for(int j = 0; j <= r; j++) {
        offset1 = mw*j;
        offset2 = mw*(mw-(j+1));
        j_float = j;
        for(int i = 0; i <= r; i++) {
                i_float = i;
                t1 = r_float-j_float; t2 = r_float-i_float;
                float dist = (float) Math.sqrt(t1*t1 + t2*t2);
                if(dist > r_float) {
                        // void area
                        sphereMask[offset1+i] = Float.POSITIVE_INFINITY;
                        sphereMask[offset1+(mw-(i+1))] = Float.POSITIVE_INFINITY;
                        sphereMask[offset2+(mw-(i+1))] = Float.POSITIVE_INFINITY;
                        sphereMask[offset2+i] = Float.POSITIVE_INFINITY;
                } else {
                        // place lowest point on rolling ball in line with
                        // highest pixel in image
                        tmp = r_float + maxValueInImage -
                        (float) Math.sqrt(r_float*r_float - dist*dist);
                        sphereMask[offset1+i] = tmp;
                        sphereMask[offset1+(mw-(i+1))] = tmp;
```

```
                                sphereMask[offset2+(mw-(i+1))] = tmp;
                                sphereMask[offset2+i] = tmp;
                        }
                }
        }

        // Shows sphere mask image
        FloatProcessor ipSphereMask = new FloatProcessor(mw, mw, sphereMask ,null);
        ImagePlus impSphere = new ImagePlus("Sphere mask with radius = " + r*scale +
        "um", ipSphereMask); // microns
        impSphere.setCalibration(imp.getCalibration());
        impSphere.show();
        IJ.run("Fire");


        // The next sections convolve the image with the mask and shows
        // result as a new filtered/smoothed image
        int w = imp.getWidth();
        int h = imp.getHeight();
        float[] impPixels = (float[]) ip.getPixels();
        int new_w = w-d;
        int new_h = h-d;
        float[] smoothed_pixels = new float[new_w*new_h];
        float tmpdist, mindist;
        int offset_imp, offset_imp_new, offset_mask, offset_imp_j, imp_i, idx_mask, idx_imp;
        int mindistidx_mask = 0, mindistidx_imp = 0;
        int midindex = mw*r+r;

        // Find the distance
        float[] dist_pixels = new float[new_w*new_h];
        for(int j = r; j < (h-r); j++) {
                offset_imp = w*j;
                offset_imp_new = new_w*(j-r);
                for(int i = r; i < (w-r); i++) {
                        mindist = Float.MAX_VALUE;
                        IJ.showProgress((j-r),(h-r));
                        // Loop through the mask to find min distance
                        for(int js = 0; js < mw; js++) {
                                offset_mask = mw*js;
                                offset_imp_j = offset_imp+w*(js-r);
                                for(int is = 0; is < mw; is++) {
                                        imp_i = i+(is-r);
                                        idx_mask = offset_mask+is;
                                        idx_imp = offset_imp_j+imp_i;
                                        tmpdist = sphereMask[idx_mask] - impPixels[idx_imp];
                                        if(tmpdist < mindist) {
                                                mindist = tmpdist;
                                                mindistidx_mask = idx_mask;
                                                mindistidx_imp = idx_imp;
                                        }
                                }
                        }
                        if(mindistidx_imp != midindex) {
                                // if the middle of the rolling ball did not provide the
                                //min distance,
                                // calculate new value from value found from rolling ball
                                //mask (mindist)
                                smoothed_pixels[offset_imp_new+(i-r)] =
                                maxValueInImage - mindist;
                                dist_pixels[offset_imp_new+(i-r)] =
                                sphereMask[mindistidx_mask];
                        } else {
                                // ...else no processing needed
                                smoothed_pixels[offset_imp_new+(i-r)] =
                                impPixels[offset_imp+i];
                                dist_pixels[offset_imp_new+(i-r)] = (float) 0.0;
```

```
                    }
                }
            }

            FloatProcessor ipDist = new FloatProcessor(new_w, new_h, dist_pixels, null);
            ImagePlus impDist = new ImagePlus(
                    "Rolling ball mask value used for the filtered image for sphere " +
                    "mask with radius = " + r*scale + " um", ipDist); // microns
            impDist.setCalibration(imp.getCalibration());
            impDist.show();

            // Shows filtered/smoothed image
            FloatProcessor ipSmoothed = new FloatProcessor(new_w, new_h,
            smoothed_pixels, null);
            ImagePlus impSmoothed = new ImagePlus("Smoothed image created with sphere " +
            "mask with radius = " + r*scale + " um", ipSmoothed); // microns
            impSmoothed.setCalibration(imp.getCalibration());
            impSmoothed.show();
            IJ.run("Fire");

            // Subtract the filtered image from the original, creating a new image
            // containing only the pores and show the resulting image
            float[] output_pixels = new float[new_w*new_h];
            for(int j = r; j < (h-r); j++) {
                offset_imp = w*j;
                offset_imp_new = new_w*(j-r);
                for(int i = r; i < (w-r); i++) {
                    output_pixels[offset_imp_new+(i-r)] = impPixels[offset_imp+i] -
                    smoothed_pixels[offset_imp_new+(i-r)];
                }
            }

            // Shows output image
            FloatProcessor ipOutput = new FloatProcessor(new_w, new_h, output_pixels, null);
            ImagePlus impOutput = new ImagePlus("Output image created with sphere " +
            "mask with radius = " + r*scale + " um", ipOutput); // microns
            impOutput.setCalibration(imp.getCalibration());
            impOutput.show();
            IJ.run("Fire");

            // Shows the original image, cropped (for comparison)
            float[] cropped_pixels = new float[new_w*new_h];
            for(int j = r; j < (h-r); j++) {
                offset_imp = w*j;
                offset_imp_new = new_w*(j-r);
                for(int i = r; i < (w-r); i++) {
                    cropped_pixels[offset_imp_new+(i-r)] = impPixels[offset_imp+i];
                }
            }
            FloatProcessor ipCropped = new FloatProcessor(new_w, new_h, cropped_pixels, null);
            ImagePlus impCropped = new ImagePlus("Original image, cropped", ipCropped);
            impCropped.setCalibration(imp.getCalibration());
            impCropped.show();
            IJ.run("Fire");
        }
}
```

A.9 RollingBallOriginal_.java

```
import java.awt.Checkbox;
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.TextField;

import ij.IJ;
import ij.ImagePlus;
import ij.ImageStack;
import ij.gui.GenericDialog;
import ij.measure.Calibration;
import ij.plugin.filter.PlugInFilter;
import ij.process.FloatProcessor;
import ij.process.ImageProcessor;

/*
 * Created on 06.apr.2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

/**
 * @author Svein Fidjestøl (SurfCharJ API: Gary Chinga)
 *
 * RollingBallOriginal plugin for ImageJ. Used for initial analysis in the thesis.
 * Separates out the rolling ball filtering routines from SurfCharJ
 * by calling the SurfCharJ API, and adds progress measurement.
 */
public class RollingBallOriginal_ implements PlugInFilter {

        /** The input image */
        private ImagePlus imp;

        /**
         * Standard ImageJ plugin setup method
         *
         * @param arg Plugin arguments
         * @param imp The input image
         */
        public int setup(String arg, ImagePlus imp) {
                this.imp = imp;
                return DOES_32;
        }

        /**
         * Standard ImageJ plugin run method.
         *
         * @param ip Selected region for processing
         */
        public void run(ImageProcessor ip) {

                boolean dPoreVol = true;

                // Get some info from the input image
        Calibration cal = imp.getCalibration();
        double pSize = cal.pixelWidth;
                int w = ip.getWidth();
        int h = ip.getHeight();
                int mMask;
        if (h<w) mMask=h; else mMask=w;            //Calculates the smallest dimension
                String units = imp.getCalibration().getUnits();

                // Draw GUI
        GenericDialog gd = new GenericDialog("RollingBallOriginal");
        gd.setLayout(new GridLayout(2,2));
        Label lblRadius = new Label("Rolling ball radius (Min="+(int)(4*pSize)+
        ", Max="+(int)(mMask*pSize/4)+" "+units+"): ");
        TextField tfRadius = new TextField("5",0);
        Checkbox cbDisplayVol = new Checkbox("Display pore image ",dPoreVol);
        gd.add(lblRadius);
        gd.add(tfRadius);
        gd.add(cbDisplayVol);
        lblRadius.setVisible(true);
        tfRadius.setVisible(true);
        cbDisplayVol.setVisible(true);
        gd.showDialog();

        // Retrieve parameters from dialog
        double bRadius = (double)(Double.parseDouble(tfRadius.getText()));
        try{
                bRadius = (double)(Double.parseDouble(tfRadius.getText()));
                if (bRadius<(int)(4*pSize)||bRadius>(int)(mMask*pSize/4)){
                        IJ.showMessage("Invalid pore radius input (min="+(int)(4*pSize)+
                        " and max="+(int)(mMask*pSize/4)+"), pore volume analysis " +
                        "will not be performed");
                }
        } catch(NumberFormatException e){
                IJ.showMessage("Invalid pore radius input, pore volume analysis will " +
                "not be performed");
        }

        // Do the analysis
        int bRad= (int)(bRadius/pSize);

        int nSlices = imp.getStackSize(); // assume whole stack (and there is only 1 slice)

        int parameters=7, PVPar = 1;
        float[][] roughnessValues= new float[nSlices][parameters];

        int nn = roughnessValues[0].length-PVPar;

        ImageStack imsPVolume = new ImageStack(w,h);

        SurfacePoreVolume2 spvol = new SurfacePoreVolume2();
        FloatProcessor ipPV = spvol.getPoreImage(ip, bRad); // get slice 1 only
        roughnessValues[0][nn] = spvol.getPoreVolume(ipPV, bRad);
        ipPV.setMinAndMax(0,0);
        imsPVolume.addSlice("Smoothed image created with sphere mask with radius = " +
        bRadius + " um",ipPV); // microns
        if (dPoreVol){
                createImagePlus(imsPVolume,"Output image created with sphere mask " +
                "with radius = " + bRadius + " um", cal); // microns
                IJ.run("Fire");
                ipPV.setMinAndMax(0,0);
        }
}

/**
 * Creates the volume output ImagePlus image
 *
 * @param imsTemp Volume output image as a stack
 * @param txt Name of ImagePlus image
 * @param c Calibration of the image
 */
void createImagePlus(ImageStack imsTemp, String txt, Calibration c){
        // Create new images using the new stacks.
        ImagePlus impTemp = new ImagePlus(txt,imsTemp);
        impTemp.setCalibration(c);
        impTemp.setStack(null,imsTemp);
```

```
                impTemp.show();
        }
}
```

A.10 SimulatePrinting_.java

```
import ij.IJ;
import ij.ImagePlus;
import ij.gui.NewImage;
import ij.gui.OvalRoi;
import ij.measure.Calibration;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;

/**
 * @author Svein Fidjestøl
 *
 * SimulatePrinting plugin for ImageJ. Simulates printing at
 * 300 dpi (dots per inch) == 12 dots per mm
 * which places the center of each dot 85 microns apart.
 * Takes as input a 32-bit float image and outputs a corresponding
 * "printed" image with the ink points raised to the max value
 * in the image. Intended to show some relationship between
 * dot sizes and pore sizes.
 */
public class SimulatePrinting_ implements PlugInFilter {

        /** The input image */
        private ImagePlus imp;

        /**
         * Standard ImageJ plugin setup method
         *
         * @param arg Plugin arguments
         * @param imp The input image
         */
        public int setup(String arg, ImagePlus imp) {
                this.imp = imp;
                return DOES_32;
        }

        /**
         * Standard ImageJ plugin run method.
         *
         * @param ip Selected region for processing
         */
        public void run(ImageProcessor ip) {
                Calibration cal = imp.getCalibration();
                String unit = cal.getUnit();
                if(!unit.equals("um")) { // microns
                        IJ.showMessage("Measurement unit for image must be um"); // microns
                        return;
                }
                IJ.run("Fire");

                // Get the classname without trailing underscore
                String className = getClass().getName();
                if(className.charAt(className.length()-1) == '_') {
                        className = className.substring(0, className.length()-1);
                }

                // Get some info from the input image
                float[] old_pixels = (float[]) ip.getPixels();
                int width = imp.getWidth();
                int height = imp.getHeight();

                // Generate a new, empty image
                ImagePlus new_imp = NewImage.createFloatImage(imp.getTitle() + ": " +
                className, width, height, 1, NewImage.FILL_WHITE);
                new_imp.setCalibration(cal);
                ImageProcessor new_ip = new_imp.getProcessor();
                // Copy the old image into the new one
                float[] pixels = (float[]) new_imp.getProcessor().getPixels();
                for (int x = 0; x < width; x++) {
                        for (int y = 0; y < height; y++) {
                                pixels[x + y * width] = old_pixels[x + y*width];
                        }
                }

                // Iterate through the dots and paint them
                int r;
                OvalRoi dot;
                double max = new_imp.getStatistics().max;
                new_ip.setValue(max);
                for(int j = (int) (cal.getRawValue(85.0) / 2.0); j < height -
                (cal.getRawValue(85.0) / 2.0); j += cal.getRawValue(85.0)) {
                        for(int i = (int) (cal.getRawValue(85.0) / 2.0); i < width -
                        (cal.getRawValue(85.0) / 2.0); i += cal.getRawValue(85.0)) {
                                r = (int) cal.getRawValue(30.0);
                                dot = new OvalRoi(i,j,2*r,2*r,new_imp);
                                new_ip.fillPolygon(dot.getPolygon());
                        }
                }
                // Set brightness values and show result
                new_ip.setMinAndMax(ip.getMin(),ip.getMax());
                new_imp.show();
                IJ.run("Fire");
        }

}
```

A.11 Triangulation_.java

```
import java.awt.Checkbox;
import java.awt.GridLayout;
import java.awt.Point;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Stack;

import ij.IJ;
import ij.ImagePlus;
import ij.gui.GenericDialog;
import ij.gui.Line;
import ij.gui.NewImage;
import ij.gui.PointRoi;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;
import ij.text.TextWindow;

/**
 * @author Svein Fidjestøl
 *
 * Triangulation plugin for ImageJ. Takes as input a binary black-and-white
 * image, in 8-bit grayscale format, with white as the background
 * color and black single pixels as vertices. Outputs either a Delaunay
 * triangulated image, its corresponding Voronoi diagram, or both. If a
 * Delaunay Triangulation is calculated, the plugin may also calculate
 * the minimum spanning tree and the convex hull.

 * The Minimum Spanning Tree uses Prim's algorithm
 * The Convex Hull uses Graham's Scan algorithm
 * The Delaunay Triangulation uses a simple but suboptimal algorithm
 * The Voronoi Diagram also uses a simple but highly suboptimal algorithm
 */
public class Triangulation_ implements PlugInFilter, ItemListener, Comparator {

        /** The input image */
        private ImagePlus imp;

        /**
         * Standard ImageJ plugin setup method
         *
         * @param arg Plugin arguments
         * @param imp The input image
         */
        public int setup(String arg, ImagePlus imp) {
                if (arg.equals("about")){
                        showAbout();
                        return DONE;
                }
                this.imp = imp;
                return DOES_8G;
        }

        // Components
        private Checkbox cbDelaunay;
        private Checkbox cbMST;
        private Checkbox cbConvexHull;
        private Checkbox cbVoronoi;
```

```
    /**
     * Standard ImageJ plugin run method. Quite monolithic for simplicity purposes
     *
     * @param ip Selected region for processing
     */
    public void run(ImageProcessor ip) {
        // Get the classname without trailing underscore
        String className = getClass().getName();
        if(className.charAt(className.length()-1) == '_') {
            className = className.substring(0, className.length()-1);
        }

        // Get options
        GenericDialog gd = new GenericDialog(className);
        gd.setLayout(new GridLayout(5,1));

        boolean delaunay = true;
        boolean mst = false;
        boolean convexhull = false;
        boolean voronoi = false;
        cbDelaunay = new Checkbox("Delaunay Triangulation", delaunay);
        cbMST = new Checkbox("Minimum spanning tree", mst);
        cbConvexHull = new Checkbox("Convex hull", convexhull);
        cbVoronoi = new Checkbox("Voronoi Diagram", voronoi);

        cbDelaunay.addItemListener(this);

        gd.add(cbDelaunay);
        gd.add(cbMST);
        gd.add(cbConvexHull);
        gd.add(cbVoronoi);

        cbDelaunay.setVisible(true);
        cbMST.setVisible(true);
        cbConvexHull.setVisible(true);
        cbVoronoi.setVisible(true);

        gd.showDialog();
        if (gd.wasCanceled()) {
            return;
        }

        delaunay = cbDelaunay.getState();
        mst = cbMST.getState();
        convexhull = cbConvexHull.getState();
        voronoi = cbVoronoi.getState();

        // Read in vertices/points
        ArrayList points = new ArrayList();
        byte[] pixels = (byte[]) ip.getPixels();
        int width = ip.getWidth();
        int height = ip.getHeight();
        int offset;
        for(int j = 0; j < height; j++) {
            offset = j*width;
            for(int i = 0; i < width; i++) {
                if(pixels[offset+i] == 0) {
                    points.add(new Point(i,j));
                }
            }
        }

        // Convert to simple array
        int len = points.size();
        Point[] points2 = new Point[len];
        for(int i = 0; i < len; i++) {
                points2[i] = (Point) points.get(i);
        }

        // Convert to RealPoint representation (for helper methods)
        RealPoint[] points3 = new RealPoint[len];
        Point tmpPoint;
        for(int i = 0; i < len; i++) {
                tmpPoint = (Point) points.get(i);
                points3[i] = new RealPoint(tmpPoint.x,tmpPoint.y);
        }

        // Generate a new, empty image
        ImagePlus new_imp = NewImage.createByteImage(imp.getTitle() + ": " +
        className, width, height, 1, NewImage.FILL_WHITE);
        new_imp.setCalibration(imp.getCalibration());
        ImageProcessor new_ip = new_imp.getProcessor();
        byte[] new_pixels = (byte[]) new_imp.getProcessor().getPixels();

        // Storage for triangulation data
HashSet lines = new HashSet();
HashSet point_tuple;

// Delaunay triangulation part
        if(delaunay) {
                /*
                 * The following code borrows heavily on example code from
                 * Delaunay.java which can be found at
                 * http://www.cs.princeton.edu/introcs/35inheritance/
                 *
                 * The performance of the following algorithm used is only n^4,
                 * but this probably doesn't matter when using small datasets
                 * (as in 1000x1000 pixel surface analysis)
                 */

                // Determine if i-j-k is a circle with no interior points.
                // If so, add the corresponding triangle to the triangulation
        for (int i = 0; i < len; i++) {
            IJ.showProgress(i, len*3-3);
                for (int j = i+1; j < len; j++) {
                for (int k = j+1; k < len; k++) {
                    boolean isTriangle = true;
                    for (int a = 0; a < len; a++) {
                        if (a == i || a == j || a == k) continue;
                        Circle c = new Circle();
                        c.circumCircle(points2[i], points2[j], points2[k]);
                        if (c.inside(points2[a])) {
                        isTriangle = false;
                           break;
                        }
                    }
                    if (isTriangle) {
                        point_tuple = new HashSet();
                        point_tuple.add(points2[i]); point_tuple.add(points2[j]);
                        lines.add(point_tuple);
                        point_tuple = new HashSet();
                        point_tuple.add(points2[i]); point_tuple.add(points2[k]);
                        lines.add(point_tuple);
                        point_tuple = new HashSet();
                        point_tuple.add(points2[j]); point_tuple.add(points2[k]);
                        lines.add(point_tuple);
                    }
                }
            }
        }
        }
        // Voronoi diagram generation and painting part
    if(voronoi) {
        RealPoint[][] nearest = new RealPoint[width][height];
        for(int pt = 0; pt < points3.length; pt++) {
            IJ.showProgress(points3.length-1 + pt, points3.length*3-3);
            // Draw regions with random value
            new_ip.setValue(Math.random() * 256.0);
            for(int j = 0; j < height; j++) {
                for(int i = 0; i < width; i++) {
                    RealPoint q = new RealPoint(i,j);
                    if(nearest[i][j] == null || q.distance(points3[pt]) <
                    q.distance(nearest[i][j])) {
                        nearest[i][j] = points3[pt];
                        PointRoi pr = new PointRoi(i,j,new_imp);
                        pr.drawPixels(new_ip);
                    }
                }
            }
        }
    }

    String result = "";

    // Delaunay triangulation painting part
    if(delaunay) {
        // Draw with light gray
        new_ip.setValue(192.0);
        ArrayList linesList = new ArrayList();
        Line line;
        Point p1,p2;
        Object[] lineArray = lines.toArray();
        for(int i = 0; i < lineArray.length; i++) {
            IJ.showProgress(lineArray.length*2-2 + i, lineArray.length*3-3);
            point_tuple = (HashSet) lineArray[i];
            if(point_tuple.size() < 2) // safety check
                continue;
            Iterator it2 = point_tuple.iterator();
            p1 = (Point) it2.next();
            p2 = (Point) it2.next();
            line = new Line(p1.x,p1.y,p2.x,p2.y, new_imp);
            linesList.add(line);
            line.drawPixels(new_ip);
         }

         // Print out some info on the triangulation to the console
         Collections.sort(linesList,new Comparator() {
             public int compare(Object arg0, Object arg1) {
                 return (int) (((Line)arg0).getLength() - ((Line)arg1).getLength());
             }
         });
         result += "Length of edges in Delaunay triangulation\n";
         for(int i = 0; i < linesList.size(); i++) {
             Line l = (Line) linesList.get(i);
             result += l.getLength() + "\n";
         }
         result += "Number of edges in Delaunay triangulation: " + lines.size() + "\n";
         double averageLengthTRI = calcAverageLength(linesList);
         result += "Average edge length in Delaunay triangulation: " +
         averageLengthTRI + "\n";

         if(convexhull) {
             ArrayList chLines = grahamscanConvexHull(linesList);
             // Print out some info on the convex hull to the console
             Collections.sort(chLines,new Comparator() {
                 public int compare(Object arg0, Object arg1) {
```

```
                                        return (int) (((Line)arg0).getLength() -
                                        ((Line)arg1).getLength());
                                }
                        });
                        result += "Length of edges in Convex Hull\n";
                        for(int i = 0; i < chLines.size(); i++) {
                                Line l = (Line) chLines.get(i);
                                result += l.getLength() + "\n";
                        }
                        result += "Number of edges in Convex Hull: " + chLines.size() + "\n";
                        double averageLengthCH = calcAverageLength(chLines);
                        result += "Average edge length in Convex Hull: " +
                        averageLengthCH + "\n";

                        // Draw the Convex Hull
                        new_ip.setValue(96.0);
                        Line l3, l4;
                        for(int i = 0; i < chLines.size(); i++) {
                                l3 = (Line) chLines.get(i);
                                l3.drawPixels(new_ip);
                        }
                }

                if(mst) {
                        ArrayList mstLines = primMST(linesList);
                        // Print out some info on the minimum spanning tree to the console
                        Collections.sort(mstLines,new Comparator() {
                             public int compare(Object arg0, Object arg1) {
                                        return (int) (((Line)arg0).getLength() -
                                        ((Line)arg1).getLength());
                                }
                        });
                        result += "Length of edges in Minimum Spanning Tree\n";
                        for(int i = 0; i < mstLines.size(); i++) {
                                Line l = (Line) mstLines.get(i);
                                result += l.getLength() + "\n";
                        }
                        result += "Number of edges in Minimum Spanning Tree: " +
                        mstLines.size() + "\n";
                        double averageLengthMST = calcAverageLength(mstLines);
                        result += "Average edge length in Minimum Spanning Tree: " +
                        averageLengthMST + "\n";

                        // Draw the Minimum Spanning Tree
                        new_ip.setValue(32.0);
                        Line l3, l4;
                        for(int i = 0; i < mstLines.size(); i++) {
                                l3 = (Line) mstLines.get(i);
                                l3.drawPixels(new_ip);
                        }
                }
        }

    // Lastly, draw the black dots from source image to dest image
     for (int y = 0; y < height; y++) {
                offset = y*width;
                for (int x = 0; x < width; x++) {
                        if(pixels[offset+x] == 0) {
                                new_pixels[offset+x] = 0;
                        }
                        }
                }
        new_imp.show();
        TextWindow tw = new TextWindow("Triangulation Result",
        "Result from Triangulation:\n" + result, 400, 150);
     }
    // Needed for access from the compare() function,
    // used for convex hull
    private Point p0;

    /**
     * Graham's Scan algorithm, calculates the convex hull
     * of a set of points.
     * NOTE: Would work with points only, line structure unneccessary
     *       but useful in the current context
     *
     * Details in Cormen, Leiserson, Rivest's
     * "Introduction To Algorithms" 2nd ed chapter 33
     *
     * @param linesList Lines in the graph in the ImageJ
     *                  Line format
     * @return The lines (in ImageJ Line format) constituting
     *         the convex hull of the graph
     */
    private ArrayList grahamscanConvexHull(ArrayList linesList) {
            // Retrieve points
            ArrayList pts = new ArrayList();
            for(int i = 0; i < linesList.size(); i++) {
                    Line curLine = (Line) linesList.get(i);
                    Point p1 = new Point(curLine.x1,curLine.y1);
                    Point p2 = new Point(curLine.x2,curLine.y2);
                    boolean foundP1 = false;
                    boolean foundP2 = false;
                    for(int j = 0; j < pts.size(); j++) {
                            Point p = (Point) pts.get(j);
                            if((p.x == p1.x) && (p.y == p1.y)) {
                                    foundP1 = true;
                            }
                            if((p.x == p2.x) && (p.y == p2.y)) {
                                    foundP2 = true;
                            }
                    }
                    if(!foundP1)
                            pts.add(p1);
                    if(!foundP2)
                            pts.add(p2);
            }

            // Retrieve point with lowest y value,
            // leftmost if two points with same y value...
            int minYSoFar = Integer.MAX_VALUE;
            Line minYSoFar_l = null;
            boolean isFirstPointOnLine = true;
            for(int i = 0; i < linesList.size(); i++) {
                    Line l = (Line) linesList.get(i);
                    if(l.y1 < minYSoFar) {
                            minYSoFar = l.y1;
                            minYSoFar_l = l;
                            isFirstPointOnLine = true;
                    } else if(l.y1 == minYSoFar) {
                            if((isFirstPointOnLine) && (l.x1 < minYSoFar_l.x1)) {
                                    minYSoFar = l.y1;
                                    minYSoFar_l = l;
                                    isFirstPointOnLine = true;
                            } else if((!isFirstPointOnLine) && (l.x1 < minYSoFar_l.x2)) {
                                    minYSoFar = l.y1;
                                    minYSoFar_l = l;
                                    isFirstPointOnLine = true;
                            }
                    }
                    if(l.y2 < minYSoFar) {
```

```
                                minYSoFar = l.y2;
                                minYSoFar_l = l;
                                isFirstPointOnLine = false;
                        } else if(l.y2 == minYSoFar) {
                                if((isFirstPointOnLine) && (l.x2 < minYSoFar_l.x1)) {
                                        minYSoFar = l.y2;
                                        minYSoFar_l = l;
                                        isFirstPointOnLine = false;
                                } else if((!isFirstPointOnLine) && (l.x2 < minYSoFar_l.x2)) {
                                        minYSoFar = l.y2;
                                        minYSoFar_l = l;
                                        isFirstPointOnLine = false;
                                }
                        }
                }
                // ...and retrieve it from the point list
                for(int i = 0; i < pts.size(); i++) {
                        Point p = (Point) pts.get(i);
                        if(isFirstPointOnLine) {
                                if((p.x == minYSoFar_l.x1) && (p.y == minYSoFar_l.y1)) {
                                        p0 = p;
                                        break;
                                }
                        } else if(!isFirstPointOnLine) {
                                if((p.x == minYSoFar_l.x2) && (p.y == minYSoFar_l.y2)) {
                                        p0 = p;
                                        break;
                                }
                        }
                }

                // Construct an array of the remaining points,
                // sorted after increasing polar angle in relation
                // with first point (p0)
                ArrayList pts_rem = new ArrayList();
                for(int i = 0; i < pts.size(); i++) {
                        Point p = (Point) pts.get(i);
                        if(!(p.equals(p0))){
                                pts_rem.add(p);
                        }
                }
                Collections.sort(pts_rem, this);

                // Stack containing first three points
                Stack S = new Stack();
                S.push(p0);
                S.push((Point)pts_rem.get(0));
                S.push((Point)pts_rem.get(1));

                // Main loop: Loop over remaining points
                // and add correct points to theconvex hull
                // by popping and pushing
                Line l_tmp = new Line(0,0,1,1,imp); // dummy line
                Point p_tmp = (Point)pts_rem.get(0);
                for(int i = 2; i < pts_rem.size(); i++) {
                        Point p_i = (Point) pts_rem.get(i);
                        while(true) {
                                // Retrieve the two top elements
                                Point topP = (Point) S.pop();
                                Point nextToTopP = (Point) S.peek();
                                S.push(topP); // put it back

                                // Calculate angle between point next to top of stack,
                                // point attop on stack, and current point
                                double a1 = l_tmp.getAngle(topP.x, topP.y, nextToTopP.x,
                                nextToTopP.y);
                        double a2 = l_tmp.getAngle(topP.x, topP.y, p_i.x, p_i.y);

                        // Adjust as neccessary
                        if(a1 <= 0.0) {
                                a1 = -1.0 * a1;
                        } else {
                                a1 = 360.0 - a1;
                        }
                        if(a2 <= 0.0) {
                                a2 = -1.0 * a2;
                        } else {
                                a2 = 360.0 - a2;
                        }
                        double delta_angle = a1-a2;
                        if(((delta_angle < 0.0) && (delta_angle > -180.0)) ||
                        (delta_angle > 180.0)) {
                                S.pop(); // point on top of stack not part of convex hull
                        } else {
                                break;
                        }
                }
                S.push(p_i);
        }

        // Construct Line objects (edges) from points
        ArrayList chLines = new ArrayList();
        Point p_init, p_prev, p = null;
        p_prev = p_init = (Point) S.pop();
        while(!S.isEmpty()) {
                p = (Point) S.pop();
                Line l = new Line(p_prev.x,p_prev.y,p.x,p.y,imp);
                chLines.add(l);
                p_prev = p;
        }
        // Add closing line (edge)
        Line l = new Line(p.x,p.y,p_init.x,p_init.y,imp);
        chLines.add(l);

        // Extract the original lines, so they can be equals()'ed
        // in later use
        ArrayList chLines2 = new ArrayList();
        for(int i = 0; i < linesList.size(); i++) {
                Line l1 = (Line) linesList.get(i);
                for(int j = 0; j < chLines.size(); j++) {
                        Line l2 = (Line) chLines.get(j);
                        if(((l1.x1 == l2.x1) && (l1.y1 == l2.y1) && (l1.x2 == l2.x2)
                        && (l2.y2 == l2.y2))
                                || ((l1.x1 == l2.x2) && (l1.y1 == l2.y2) &&
                                (l1.x2 == l2.x1) && (l1.y2 == l2.y1))) {
                                chLines2.add(l1);
                        }
                }
        }
        return chLines2;
}

/**
 * Prim's algorithm, calculates the minimum spanning tree
 * in a graph.
 *
 * Details in Cormen, Leiserson, Rivest's
 * "Introduction To Algorithms" 2nd ed chapter 23
 *
 * @param linesList Lines in the graph in the ImageJ
 *                  Line format
 * @return The lines (in ImageJ Line format) constituting
```

```
     *          the minimum spanning tree of the graph
     */
    private ArrayList primMST(ArrayList linesList) {
            // Retrieve points
            ArrayList pts = new ArrayList();
            for(int i = 0; i < linesList.size(); i++) {
                    Line curLine = (Line) linesList.get(i);
                    PrimPoint p1 = new PrimPoint(curLine.x1,curLine.y1);
                    PrimPoint p2 = new PrimPoint(curLine.x2,curLine.y2);
                    boolean foundP1 = false;
                    boolean foundP2 = false;
                    for(int j = 0; j < pts.size(); j++) {
                            PrimPoint p = (PrimPoint) pts.get(j);
                            if((p.x == p1.x) && (p.y == p1.y)) {
                                    foundP1 = true;
                            }
                            if((p.x == p2.x) && (p.y == p2.y)) {
                                    foundP2 = true;
                            }
                    }
                    if(!foundP1)
                            pts.add(p1);
                    if(!foundP2)
                            pts.add(p2);
            }

            // Fill adjacency list in all points
            for(int i = 0; i < pts.size(); i++) {
                    PrimPoint p = (PrimPoint) pts.get(i);
                    for(int j = 0; j < linesList.size(); j++) {
                            Line l = (Line) linesList.get(j);
                            if((l.x1 == p.x) && (l.y1 == p.y)) {
                                    for(int k = 0; k < pts.size(); k++) {
                                            PrimPoint p_tmp = (PrimPoint) pts.get(k);
                                            if((l.x2 == p_tmp.x) && (l.y2 == p_tmp.y)) {
                                                    p.adj.add(p_tmp);
                                            }
                                    }
                            } else if((l.x2 == p.x) && (l.y2 == p.y)) {
                                    for(int k = 0; k < pts.size(); k++) {
                                            PrimPoint p_tmp = (PrimPoint) pts.get(k);
                                            if((l.x1 == p_tmp.x) && (l.y1 == p_tmp.y)) {
                                                    p.adj.add(p_tmp);
                                            }
                                    }
                            }
                    }
            }

            // Initialize keys, setting key 0 to zero so this becomes
            // the starting point
            for(int i = 0; i < pts.size(); i++) {
                    PrimPoint p = (PrimPoint) pts.get(i);
                    if(i == 0)
                            p.key = 0.0;
                    else
                            p.key = Double.MAX_VALUE;
                    p.parent = -2; // for sanity checking
            }

            // Initialize priority queue
            ArrayList PQ = new ArrayList();
            for(int i = 0; i < pts.size(); i++) {
                    PrimPoint p = (PrimPoint) pts.get(i);
                    PQ.add(p);
            }
```

```
// Main loop
while(!PQ.isEmpty()) {
        // u <- Extract-min(PQ)
        double minKey = Double.MAX_VALUE;
        int minKeyIdx = -1;
        for(int i = 0; i < PQ.size(); i++) {
                PrimPoint p = (PrimPoint) PQ.get(i);
                if(p.key <= minKey) {
                        minKey = p.key;
                        minKeyIdx = i;
                }
        }
        PrimPoint u = (PrimPoint) PQ.remove(minKeyIdx);

        // Find index of u
        int u_idx = -1;
        for(int k = 0; k < pts.size(); k++) {
                PrimPoint p_tmp2 = (PrimPoint) pts.get(k);
                if((p_tmp2.x == u.x) && (p_tmp2.y == u.y)) {
                        u_idx = k;
                }
        }

        // Loop over v's that are both in PQ and adjacenct to u
        for(int i = 0; i < u.adj.size(); i++) {
                PrimPoint v = (PrimPoint) u.adj.get(i);
                if(PQ.contains(v)) {
                        for(int j = 0; j < linesList.size(); j++) {
                                Line l = (Line) linesList.get(j);
                                if(((u.x == l.x1) && (u.y == l.y1) &&
                                (v.x == l.x2) && (v.y == l.y2))
                                   || ((v.x == l.x1) && (v.y == l.y1) &&
                                   (u.x == l.x2) && (u.y == l.y2))) {
                                        // Find index of v
                                        int v_idx = -1;
                                        for(int k = 0; k < pts.size(); k++) {
                                                PrimPoint p_tmp3 =
                                                (PrimPoint) pts.get(k);
                                                if((p_tmp3.x == v.x) &&
                                                (p_tmp3.y == v.y)) {
                                                        v_idx = k;
                                                }
                                        }
                                        // Store new parent and keys
                                        double len = l.getLength();
                                        if(len < v.key) {
                                                v.parent = u_idx;
                                                v.key = len;
                                        }
                                }
                        }
                }
        }
}

// Read out minimum spanning tree into mstLines
// by connecting children and parents
ArrayList mstLines = new ArrayList();
Line firstLine = (Line) linesList.get(0);
for(int i = 1; i < pts.size(); i++) { // skip first point by starting at one
        PrimPoint p1 = (PrimPoint) pts.get(i);
        PrimPoint p2 = (PrimPoint) pts.get(p1.parent);
        for(int j = 0; j < linesList.size(); j++) {
                Line l = (Line) linesList.get(j);
                if(((p1.x == l.x1) && (p1.y == l.y1) &&
                                    (p2.x == l.x2) && (p2.y == l.y2))
                                        || ((p2.x == l.x1) && (p2.y == l.y1) &&
                                        (p1.x == l.x2) && (p1.y == l.y2))) {
                                            mstLines.add(l);
                                    }
                            }
                    }
                    return mstLines;
            }

            /**
             * Helper method to calculate average length of several lines
             *
             * @param lines A List object containing two or more Line objects
             * @return Average length of the lines
             */
            private double calcAverageLength(List lines) {
                    double cumsum = 0.0;
                    Line line;
                    for(Iterator it = lines.iterator(); it.hasNext();) {
                            line = (Line) it.next();
                            cumsum += line.getLength();
                    }
                    return (cumsum / lines.size());
            }

            /**
             * Compare method used in grahamscanConvexHull()
             *
             * @param arg0 First point
             * @param arg1 Second point
             * @return Point with the lowest polar angle to a horizontal line
             */
            public int compare(Object arg0, Object arg1) {
                    Point p1 = (Point) arg0;
                    Point p2 = (Point) arg1;
                    Line l = new Line(0,0,1,1,imp); // dummy line
                    double angleWithP0_1 = l.getAngle(p0.x, p0.y, p1.x, p1.y);
                    double angleWithP0_2 = l.getAngle(p0.x, p0.y, p2.x, p2.y);
                    if(angleWithP0_1 <= 0.0) {
                            angleWithP0_1 = -1.0 * angleWithP0_1;
                    } else {
                            angleWithP0_1 = 360.0 - angleWithP0_1;
                    }
                    if(angleWithP0_2 <= 0.0) {
                            angleWithP0_2 = -1.0 * angleWithP0_2;
                    } else {
                            angleWithP0_2 = 360.0 - angleWithP0_2;
                    }
                    return (int) (angleWithP0_1 - angleWithP0_2);
            }



            /**
             * Callback method used for hiding and showing
             * components in the dialog
             *
             * @param e The ItemEvent
             */
            public void itemStateChanged(ItemEvent e) {
                    if(e.getItemSelectable() == cbDelaunay) {
                            if(e.getStateChange() == ItemEvent.SELECTED) {
                                    cbMST.setVisible(true);
                                    cbConvexHull.setVisible(true);
                            } else {
                                    cbMST.setVisible(false);
                                    cbConvexHull.setVisible(false);
                            }
                    }
            }

            /**
             * About window.
             */
        public void showAbout() {
            IJ.showMessage("About Triangulation...",
                            "Triangulation plugin for ImageJ. Takes as input a binary black-and-white" +
                            "image, in 8-bit grayscale format, with white as the background" +
                            "color and black single pixels as vertices. Outputs either a Delaunay" +
                            "triangulated image, its corresponding Voronoi diagram, or both.");
        }

            /*
             * The remaining helper classes and methods borrow heavily on example applet
             * TriangulationApplet.java which can be found at
             * http://goanna.cs.rmit.edu.au/~gl/classes/TriangulationApplet.java
             * http://goanna.cs.rmit.edu.au/~gl/research/comp_geom/delaunay/delaunay.html
             */

            /**
             * Circle class. Circles are fundamental to computation of Delaunay
             * triangulations. In particular, an operation which computes a
             * circle defined by three points is required.
             */
            class Circle {
                RealPoint c;
                float r;

                Circle() { c = new RealPoint(); r = 0.0f; }
                Circle(RealPoint c, float r) { this.c = c; this.r = r; }
                public RealPoint center() { return c; }
                public float radius() { return r; }
                public void set(RealPoint c, float r) { this.c = c; this.r = r; }

                /**
                 * Tests if a RealPoint object lies inside the circle instance.
                 *
                 * @param p The RealPoint input object
                 * @return True if the RealPoint is inside, false otherwise
                 */
                public boolean inside(RealPoint p) {
                    if (c.distanceSq(p) < r * r)
                        return true;
                    else
                        return false;
                }

                /**
                 * Tests if a Point object lies inside the circle instance.
                 *
                 * @param p The Point input object
                 * @return True if the Point is inside, false otherwise
                 */
                public boolean inside(Point p) {
                    return inside(new RealPoint(p.x,p.y));
                }

                /**
                 * Compute the circle defined by three RealPoints (circumcircle).
                 *
                 * @param p1 First point defining the circle boundary
```

```
     * @param p2 Second point defining the circle boundary
     * @param p3 Third point defining the circle boundary
     */
    public void circumCircle(RealPoint p1, RealPoint p2, RealPoint p3) {
        float cp;

        cp = crossProduct(p1, p2, p3);
        if (cp != 0.0)
            {
                float p1Sq, p2Sq, p3Sq;
                float num, den;
                float cx, cy;

                p1Sq = p1.x() * p1.x() + p1.y() * p1.y();
                p2Sq = p2.x() * p2.x() + p2.y() * p2.y();
                p3Sq = p3.x() * p3.x() + p3.y() * p3.y();
                num = p1Sq*(p2.y() - p3.y()) + p2Sq*(p3.y() - p1.y()) +
                p3Sq*(p1.y() - p2.y());
                cx = num / (2.0f * cp);
                num = p1Sq*(p3.x() - p2.x()) + p2Sq*(p1.x() - p3.x()) +
                p3Sq*(p2.x() - p1.x());
                cy = num / (2.0f * cp);

                c.set(cx, cy);
            }

        // Radius
        r = c.distance(p1);
    }

    /**
     * Wrapper to use the circumCircle function on Point objects
     * as well as RealPoint objects
     *
     * @param p1 First point defining the circle boundary
     * @param p2 Second point defining the circle boundary
     * @param p3 Third point defining the circle boundary
     */
    public void circumCircle(Point p1, Point p2, Point p3) {
        circumCircle(new RealPoint(p1.x,p1.y),new RealPoint(p2.x,p2.y),
        new RealPoint(p3.x,p3.y));
    }
}

/**
 * Helper class representing a Point in float format, with corresponding
 * helper methods
 */
class RealPoint {
    float x, y;

    RealPoint() { x = y = 0.0f; }
    RealPoint(float x, float y) { this.x = x; this.y = y; }
    RealPoint(RealPoint p) { x = p.x; y = p.y; }
    public float x() { return this.x; }
    public float y() { return this.y; }
    public void set(float x, float y) { this.x = x; this.y = y; }

    public float distance(RealPoint p) {
        float dx, dy;

        dx = p.x - x;
        dy = p.y - y;
        return (float)Math.sqrt((double)(dx * dx + dy * dy));
    }
```

```
        public float distanceSq(RealPoint p) {
            float dx, dy;

            dx = p.x - x;
            dy = p.y - y;
            return (float)(dx * dx + dy * dy);
        }
    }

    /**
     * Vector class.  Includes a few elementary vector operations as helper methods
     */
    class Vector {
        float u, v;

        Vector() { u = v = 0.0f; }
        Vector(RealPoint p1, RealPoint p2) {
            u = p2.x() - p1.x();
            v = p2.y() - p1.y();
        }
        Vector(float u, float v) { this.u = u; this.v = v; }

        float dotProduct(Vector v) { return u * v.u + this.v * v.v; }



        float crossProduct(Vector v) { return u * v.v - this.v * v.u; }



        void setRealPoints(RealPoint p1, RealPoint p2) {
            u = p2.x() - p1.x();
            v = p2.y() - p1.y();
        }
    }

    /**
     * PrimPoint class.  Used for representing points
     * in primMST (Prim's minimum spanning tree algorithm)
     */
class PrimPoint extends Point {
            private static final long serialVersionUID = 1L;
    public double key;
    public ArrayList adj;
    public int parent;

    public PrimPoint(int x, int y) {
            super(x,y);
            adj = new ArrayList();
    }
}

    /**
     * Vector dot product on RealPoint objects
     *
     * @param p1 First point in dot product calculation
     * @param p2 Second point in dot product calculation
     * @param p3 Third point in dot product calculation
     * @return The vector dot product
     */
static float dotProduct(RealPoint p1, RealPoint p2, RealPoint p3) {
            float u1, v1, u2, v2;

            u1 =  p2.x() - p1.x();
            v1 =  p2.y() - p1.y();
            u2 =  p3.x() - p1.x();
```

```
                v2 =  p3.y() - p1.y();

                return u1 * u2 + v1 * v2;
    }


    /**
     * Vector cross product on RealPoint objects
     *
     * @param p1 First point in vector product calculation
     * @param p2 Second point in vector product calculation
     * @param p3 Third point in vector product calculation
     * @return The vector cross product
     */
    static float crossProduct(RealPoint p1, RealPoint p2, RealPoint p3) {
                float u1, v1, u2, v2;

                u1 =  p2.x() - p1.x();
                v1 =  p2.y() - p1.y();
                u2 =  p3.x() - p1.x();
                v2 =  p3.y() - p1.y();

                return u1 * v2 - v1 * u2;
        }
}
```

A.12 Thresholder_.java

```
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.TextField;

import ij.ImagePlus;
import ij.gui.GenericDialog;
import ij.gui.NewImage;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;

/**
 * @author Svein Fidjestøl
 *
 * Thresholder plugin for ImageJ. Takes as input a 32-bit float
 * image and outputs a corresponding (height-) thresholded
 * 8-bit binary black-and-white image.
 */
public class Thresholder_ implements PlugInFilter {

        /** The input image */
        private ImagePlus imp;

        /**
         * Standard ImageJ plugin setup method
         *
         * @param arg Plugin arguments
         * @param imp The input image
         */
        public int setup(String arg, ImagePlus imp) {
                // TODO Auto-generated method stub
                this.imp = imp;
                return DOES_32;
        }

        /**
         * Standard ImageJ plugin run method. Quite monolithic for simplicity purposes
         *
         * @param ip Selected region for processing
         */
        public void run(ImageProcessor ip) {
                // Get the classname without trailing underscore
                String className = getClass().getName();
                if(className.charAt(className.length()-1) == '_') {
                        className = className.substring(0, className.length()-1);
                }

                // Read in threshold value with a dialog box
                String currentSelection;
                float threshold;
                GenericDialog gd = new GenericDialog(className);
                gd.setLayout(new GridLayout(2,1));

                Label lblThreshold = new Label("Threshold");
                TextField tfThreshold = new TextField("0.0",0);

                gd.add(lblThreshold);
                gd.add(tfThreshold);

                lblThreshold.setVisible(true);
                tfThreshold.setVisible(true);

                gd.showDialog();
                threshold = Float.parseFloat(tfThreshold.getText());
                // Get some info from the input image
                float[] pixels = (float[]) ip.getPixels();
                int width = ip.getWidth();
                int height = ip.getHeight();

                // Generate a new, empty image
                ImagePlus new_imp = NewImage.createByteImage(imp.getTitle() + ": " +
                className, width, height, 1, NewImage.FILL_WHITE);
                new_imp.setCalibration(imp.getCalibration());
                byte[] new_pixels = (byte[]) new_imp.getProcessor().getPixels();

                // Threshold the image
                float pixel;
                for(int j = 0; j < height; j++) {
                        for(int i = 0; i < width; i++) {
                                pixel = pixels[j*width+i];
                                if(pixel < threshold) {
                                        new_pixels[j*width+i] = (byte) 0;
                                } else {
                                        new_pixels[j*width+i] = (byte) 255;
                                }
                        }
                }
                new_imp.show();
        }


}
```