Comstat2 - a modern 3D image analysis environment for biofilms

Martin Vorregaard
s053247

Kongens Lyngby
31^{th} January 2008

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

This project concerns the development of Comstat2, a Java-based computerprogram for the analysis and treatment of biofilm images in 3D. Various algorithms for gathering knowledge on biofilm structure, etc., vil be examined. Emphasis is put on creating a future-proof program structure and to ensure compatibility with an earlier version of the program. An, in this area, new method for the evaluation of biofilm thickness in 3D is introduced.

Resumé

Denne opgave omhandler udviklingen af Comstat2, et Java-baseret computerprogram til analyse og behandling af 3D billeder af biofilm. Der vil blive gennemgået forskellige algoritmer til at opsamle viden om biofilms struktur, mv. Vægten er lagt på at skabe en fremtidssikret programstruktur samt at sikre bagudkompatibilitet med en tidligere version af programmet. En, på området, så vidt vides, ny metode til bedømmelse af tykkelsen af biofilm i 3D vil blive introduceret.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark.

Supervisor:
Bjarne Kjær Ersbøll (be@imm.dtu.dk)
Additional Supervisors:
Claus Sternberg (cst@biocentrum.dtu.dk)
Thomas Martini Jørgensen (thomas.martini@risoe.dk)

Lyngby, January 2008

Martin Vorregaard

Contents

Chapter 1

Introduction

1.1 Reader's guide

The idea of this report is to give the reader a good overview of the construction of the image analysis tool Comstat2. The user will get both an in-depth look at the structure of the program but also a run-through of the algorithms used.

Different user may not want to read the entire report, so it is divided into chapters according to contents:

1. **Analysis** elaborates on the preliminary thoughts made in the process of planning Comstat2. Also discusses the limitations set in terms of the environment.

2. **Design & Implementation** focuses on the actual structure of the programme.

3. **Functions and Algorithms** contains descriptions of how the functionalities of the original Comstat were converted to Java. Also contains some considerations on memory usage, etc.

4. **Test** is a summary of tests performed, including a mass-test in the field

5. **Summary** of the project, including what went good/bad, concluding comments, etc.

For the reader with little knowledge of biology or Comstat, the introduction is the right place to start.

1.2 What is biofilm?

When talking of bacteria the common perception is that they are single-celled organisms floating around in a medium (air, water, etc.) much like plancton in water. However, another state exists in which the bacteria collaborate in stationary groups to gain advantages. This state is called biofilm. Many types of bacteria are capable of entering the biofilm-state under varying conditions but not all show the same prevalence.

One of the main advantages of being in a biofilm is that the resitance against alien threats is lowered. If biofilm is generated as part of an infection in a chronic wound the resistance to antibiotics typically increases to a level where the only viable cure is amputation of the infected limb. A similar pattern can be found in cystic fibrosis patients where traditional antibiotic treatments can be used to keep the chronic lung infections at a very low level up to the point where a bacteria strain that creates biofilms is introduced (or triggered). From then, the vitality of the patient unfortunately only goes down.

Even though the phenomenon is well-known, there is not much knowledge on how bacteria in the biofilm-state behave, what triggers the contruction, how communication is done, etc. Only by investigating these can science defeat the abovementioned (and several other) deceases.

1.3 The Motivation for Comstat

The first Comstat was developed to be a quantitative instrument for analyzing image stacks containing biofilms obtained using confocal microscopy. The original version was made by Arne Heydorn and written as an extensive collection of scripts for Matlab. Comstat has since its introduction been cited numerous times in scientific articles.

Although the program does well, some points to be improved has surfaced along

the way:

- Matlab is required, is very expensive
- user interface is commandline based, poor overview
- only image files that Matlab knows can be treated
- program structure difficult to outline
- no control with the source code - people can easily change and redistribute

Many of the points are with reference to Matlab. Matlab script was chosen as platform for Comstat because it offered a vast mathematical api and i/o-functionality straight out of the box. The alternatives at the time would have been to either implement Comstat in C++ which would require the programming of a lot of math-functionality and no platform independence or in Java which at that time was not very popular and would still require a lot of math-programming.

So the outline for creating a new version was to remove the bond to Matlab and to optimize the ease of use in order to widen the group of possible users.

1.4 The Basis for Comstat2

Before the project to make Comstat2 was started it was decided to base the new version on a free image manipulation programme called ImageJ, if possible.

ImageJ is written in Java and is completely Open Source. It supports the addition of third-party plugins that have full access to ImageJ's api - and can even interact with other plugins. A great many plugins exist to do everything from i/o-operations to advanced image analysis so some of these would hopefully be time savers.

The main task in the development process was a trial-on-error approach to see whether an implementation was at all possible. If so, the functionality of the original version was to be reimplemented and new functions added as well.

Chapter 2

Analysis

2.1 Method

Already before starting the project of developing Comstat2 it was clear that a full-scale development scheme like OOA&D would be overkill because the application is meant to be a single user, single computer tool for doing mathematical analysis on images rather than a multi-role, multi-user system.

However, a certain level of control and overview is gained by giving some thoughts to the domain of the program, so the following has been done:

- Identification of classes and events in the problem area
- A simple, structural overview
- Sketches of class behaviour
- Identification of actors
- User interface sketch

The motivation will be given in the subsequent sections.

2.2 Classification

After a thorough evaluation, the following classes from the problem area were identified:

- Image
- Data
- Algorithm
- User

Finding the objects of the problem area gives an idea of what the main concepts of the program are. The following events were then found:

- load
- save
- print
- run

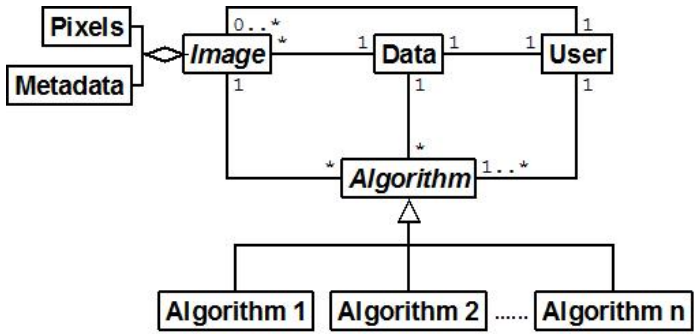
Connecting the above in an event table looks like:

event/class	User	Image	Data	Algorithm
load	*	+		
save	*	*		
print			*	*
run	*	*		*

Notice that the multiplicity of the events have been included (+ for once, $*$ for many. The only event that can only happen once for an object type is the loading of an image.

2.3 Structure

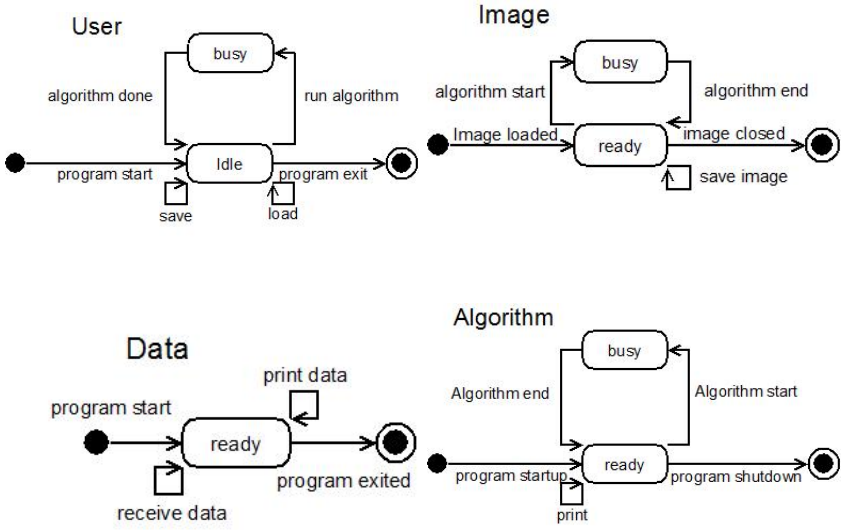
The first class diagram was then made:



A couple of extra classes have been added to signify that an image is made up of two entities in the current context.

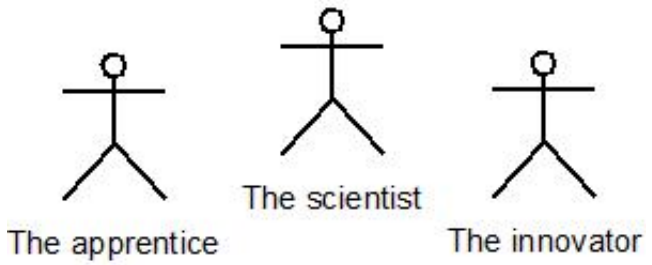
2.4 Class Behaviour

Following identification of the classes comes a description of the individual classes' lifespan behaviour.



2.5 Actors

The following three actors were found:



2.5.1 The Apprentice

The apprentice is being teached about biofilms and will use Comstat to confirm important points in connection with labwork. Typically, a limited amount of tests are made.

He has a basic knowledge of both biofilms and what the output of the program tells but not necessarily any in-depth knowledge of computers or image analysis.

2.5.1.1 Typical usage pattern

The apprentice starts up Comstat2 after having recorded a few image stacks with a microscope. He follows a step-by-step guide and must answer some questions on what conclusions he can make with the given data. The process of obtaining results is in focus. In the end, data is treated using Microsoft Excel in order to create diagrams, etc. for a class presentation.

2.5.2 The Scientist

The scientist has a profound knowledge of biofilms and a user-level knowledge of image analysis and computers in general. She will be using the program to find statistical evidence in order to prove or disprove presumptions.

2.5.2.1 Typical usage pattern

The scientist opens Comstat2 at the beginning of the day and selects a vast number of image stacks for analysis. She makes the program run the desired functions and leaves the machine to do the work. At the end of the day she returns and gathers the obtained data. The data is further processed using some statistical tools, perhaps for use in an article.

2.5.3 The Innovator

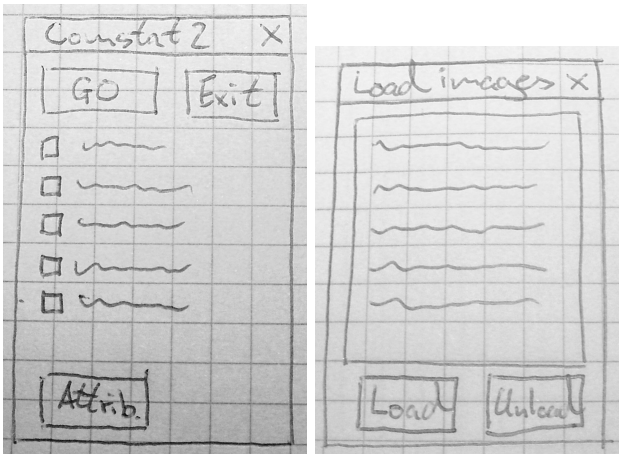
The innovator resembles the scientist but has a somewhat higher level of computer knowledge.

2.5.3.1 Typical usage pattern

The innovator comes up with a brilliant new idea for a function to be implemented in Comstat2. He finds relevant litterature on the algorithm and implements it in an afternoon using Comstat2's api. He then distributes the improved application for his peers to review.

2.6 Interfaces

The interface from Comstat2 to the system is given in the sense that we are to use ImageJ's plugin-api. Since few details on the api was known at this time, weigth was put on the (graphical) user interface and its relation to the defined actors. The following preliminary drawings were made:



The two windows are the main toolbox and a separate window for managing open files. The latter is required in order to satisfy the scientists need of overviewing a lot of open images. The choice of using small toolbox-like windows is already made as this is the general concept of ImageJ's gui.

Chapter 3

Design & Implementation

3.1 Implementing the new Comstat

This chapter is two-sided: On one hand it contains some of the considerations and specific solutions made during the implementation phase while on the other it contains documentation of the project in the form of classdiagrams, etc.

The chapter is divided as follows:

- General structure
- Specific solutions
- Review of classes
- Review of the GUI

3.2 The general structure

The final structure of the program can be seen in the main classdiagram, that - due to space constraints - has been separated in two: Fig. 3.1 and 3.2.

In total, the program ended up having 8 core-program classes and 9 function classes - not counting the parent of functions, ComstatFunction. Some additional classes are hidden inside i.e. the LocalThickness3D-class. These are put there in order to make the program resistent to future changes and of course make users independent of downloading a thirdparty plugin. The thresholding algorithms of ComstatMultiMode are also adoptions of the ones delivered with Comstat, but they are not separated into inner classes.

The program consists of over 4500 lines lines of code. Again, not all lines are original and some regrettable redundancy exists. See 3.2.1 for an example.

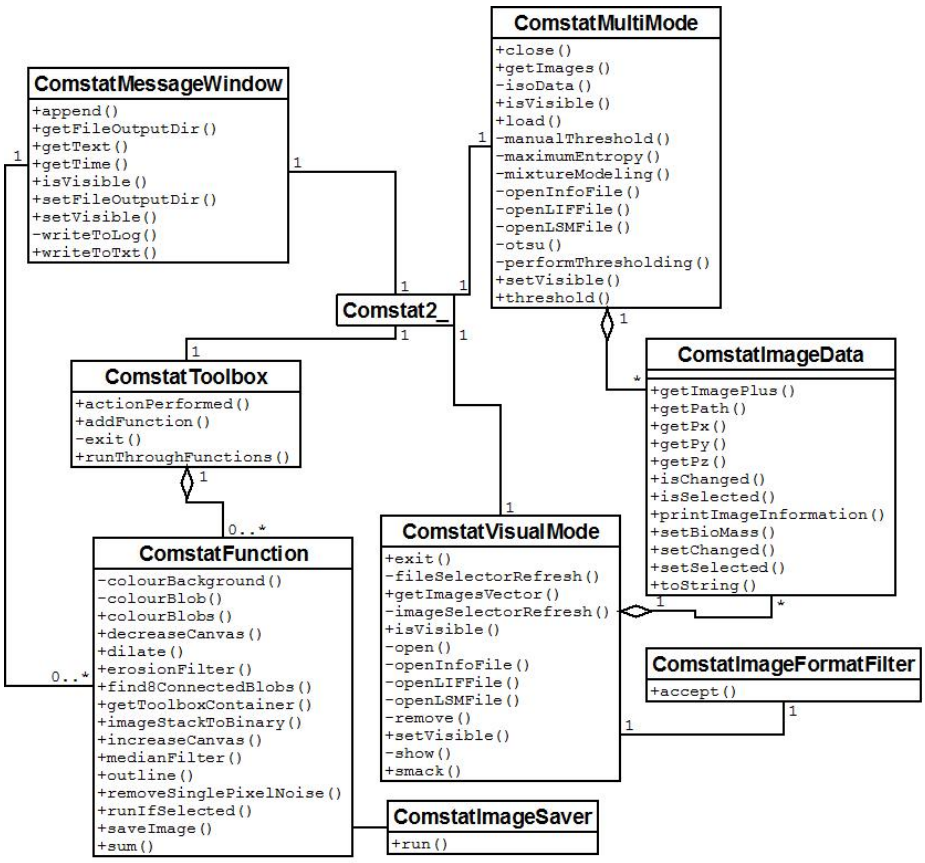


Figure 3.1: *This is the main classdiagram with methods*

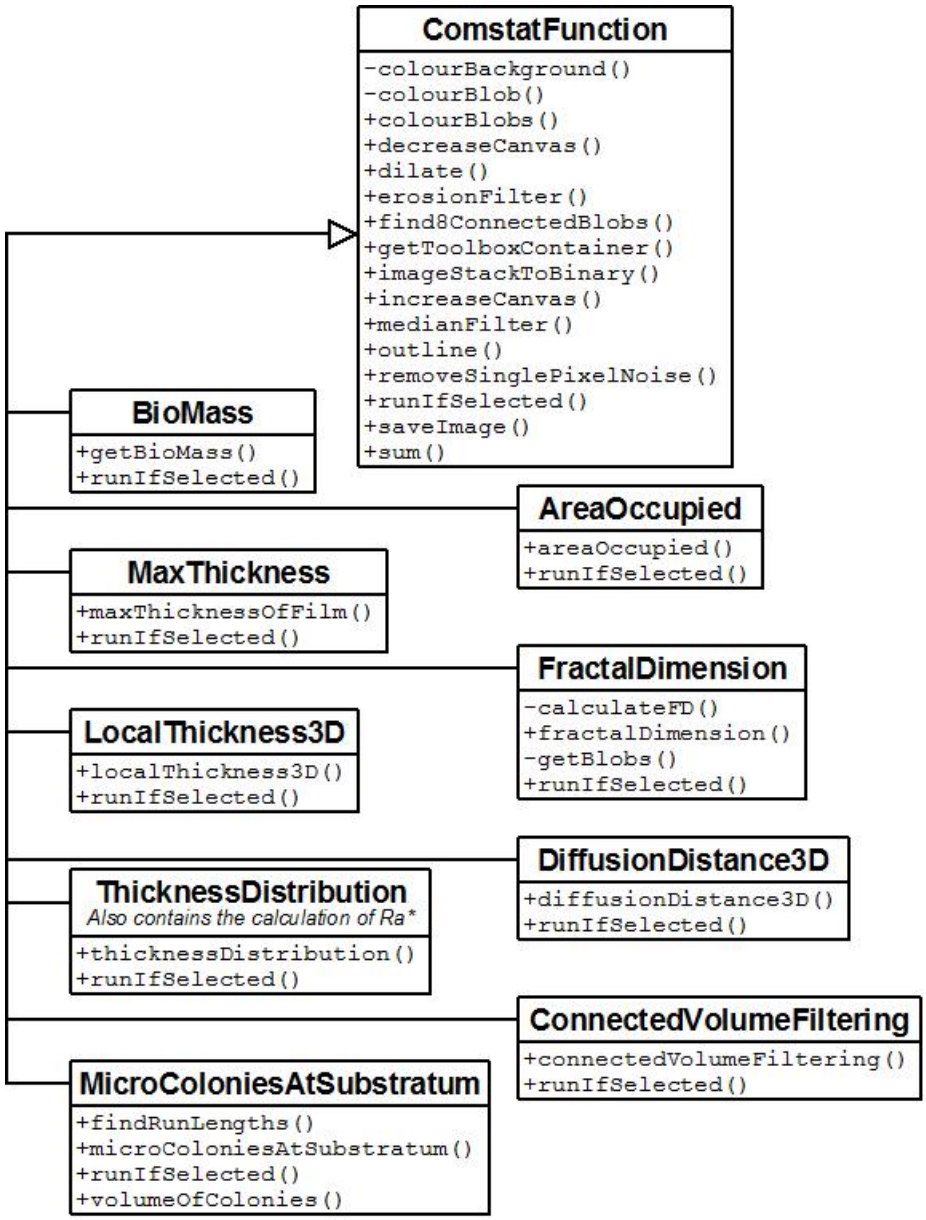


Figure 3.2: *The second part of the classdiagram with the most important classes: The functions.*

3.2.1 Use of inheritance

Exactly as in the first sketch-up of the problem area, the algortihms for working with the imagestacks are all derived from a single class, ComstatFunction (See sec. 3.2. A developer can then extend it and make his own function but still have access to all the general purpose methods that have been put into ComstatFunction.

Unfortunately, an example of bad planning does also exist within the project: The two modes. If these had been derived from a single class, a lot of redundant code could have been avoided. In a future release, it would be adviseable to refactor according to fig. 3.3

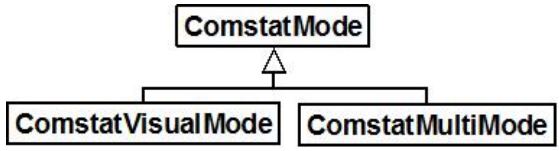


Figure 3.3: *If the project is refactored, code redundancy can be lowered and the addition of future modes would also be simpler.*

3.3 Specific solutions

This is a review of some of the specific software solutions made with relations to programs general function. Description of algorithms used in functions can be found in chapter. 4.

3.3.1 Thresholding

The algorithms used for thresholding are the ones implemented in ImageJ. In the future it would be nice to implement better algorithms like the one suggested in [6]. Aside from that, the manual, eyebased method is still the best.

Even though the method for finding thresholds in ImageJ are used in ComstatMultiMode, the built-in way of setting it is not. Instead, a for-loop is made to run over all pixels in the image, backgrounding any with a value less than the found threshold in the performThresholding-method of ComstatMultiMode.

When the MultiThresholder or the manual thresholding in ComstatMultiMode is used, one must take notice that thresholding here means "less than" the threshold and not "Less than or equal to" as in the original Comstat.

3.3.2 Dynamic addition of functions

The use of the word "dynamic" can actually be taken to mean "at runtime" - this is actually possible - but is to mean that addition (or removal) of functions at loadtime can happen easily and that the userinterface will dynamically adjust to this when the program is run.

The way to accomplish this is to keep a list of functions within the ComstatToolobox and let ComstatFunctions be added to this through a dedicated method called addFunction. ComstatFunction prescribes that a Java AWT Container named toolboxContainer be defined. The toolboxContainer is designed to be the onscreen space in which the developer of the given function can put GUI-components for interaction. When addFunction is run, it simply adds the content of the new function's toolboxContainer to the local toolbox-pane. Allthough the developer has free hands to decide on how a functions GUI looks, it is strongly recommended to at least put a CheckBox with the name of the funtion in there. All the components added to the toolboxContainer can at all times still be accessed by the ComstatFunction that it belongs to and thus adjustments made by a user retrieved.

3.3.3 Running of functions

Running functions on imagestacks is what Comstat2 is all about. Imagestacks are kept in memory by either ComstatVisualMode or ComstatMultiMode. The two has access to the local ComstatToolbox, that has a method named runThroughFunctions and takes a Vector of ComstatImageDatas as parameter. As the name implies, the method runs through the list of functions, calling their runIfSelected-method with the ComstatImageDatas from the Vector as parameter. If the function is selected in the GUI (preferably indicated by a CheckBox), the called runIfSelected will then take care of running the functionality offered by the particular ComstatFunction on the given ComstatImageData.

All installed functions are run one image at a time until the entire Vector has been emptied.

3.3.4 Loading of files

When loading files in Comstat2, two API's are used:

- LOCI BioFormats for reading:
  - Leica Image File Format (.liff) files
  - Tiff (.tif) files
- LSM Reader for reading:
  - Zeiss LSM files

Both are ImageJ-plugins. The BioFormats importer is used for TIFF files even though ImageJ natively loads them, as ImageJ cannot read LZW-compressed ones.

The reason for using external API's for loading images are clear from a software-developers point of view: If someone has already made a code snippet that implements some needed functionality it must be used. Especially if the code has a large userbase - that per implication ensures it is relieable. It obviously also saves some work - and with a little luck, filesupport will increase with time. There is, however, also drawbacks. If, for instance, something central to the API is changed, the program might stop working entirely until someone has the time to counter the changes.

With relations to the task in focus here, the two microscope formats contain a huge amount of metadata about everything from instrument serialnumber to pixelsizes (and interslice distance). Implementing a filereader for such an advanced format would be a project of it's own.

In Comstat2, two classes uses the BioFormats and LSMReader to open image-files:

- ComstatMultimode
- ComstatVisualMode

The way the two uses them are very similar, but the way images are kept in memory is a bit different. See the class reviews in sec. 3.5 for a description

3.3.5 Writing of files

In order to store results made by functions performing calculations on imagestacks, Comstat wrote output to simple textfiles in which values where stored as tab-separated. These could then be imported using Excel in order to make nice graphs and otherwise treat the data.

A similar approach is taken in Comstat2. Here, the individual functions are given the responsability of writing their output on a certain form, to allow easy printing to log and disk.

3.3.5.1 Example: The Biomass

Let's imagine that the biomass of an imagestack has been calculated and stored in `biomass`. The name of the treated image is stored in `name`. A string `out` is then generated:

```
out = BIOMASS\n
      Image name\t  Biomass (um^3/um^2)\n
      name\t        biomass\n
```

Now, the local ComstatMessageWindow, `msgOut`, has it's setFileOutputDir called with the path to the current imagestacks dir. Then the append method is called with `out` and the ComstatImageData containing the imagestack. append immediately prints `out` to the onscreen- and ondisk log (placed in the users homedirectory) and then calls the method appendToTxt:

```
Cut first line of out, store in "type"
if type = BIOMASS
  set output filename outName to BioMass.txt
else if type = THICKNESS
  ...
end if
if file outName exists in current directory
  append the rest of out
else
  create file outName, append a title and date
  append out
end if
```

The principle is, that if a new function is added to Comstat2, a new "else if" must be inserted in ComstatMessageWindow to check for a new type-name (like "BIOMASS"). Of course, a new filename for the output must also be chosen.

3.4 The Graphical User Interface

One of the main reasons for doing a new version of Comstat was that the user-interface of the original was commandline based and thus tedious and strictly sequential in use. During the analysis phase, simple drawings where made of how the userinterface could look.

3.4.1 Why use multiple windows?

When you create GUI's for editing images there are commonly two ways of setting up a GUI:

- the Adobe Photoshop way (one main window, childwindows inside)
- the GIMP way (one main toolbar, many childwindows spread over the screen)

The Photoshop way offers the advantage of cohesion - all open images, toolboxes, etc. are collected as a single unit whereas the GIMP fashion offers more space for doing work (at least visually).

The reason for walking down the GIMP-road was that ImageJ already sported this and - most importantly - the first part of the development of Comstat2 was performed on an Apple computer running Mac OS X. Tiger, as it was nicknamed, has a set of GUI-methods named Exposé. One of these methods (typically activated by F10) lets the user see an overview over all open windows of the program currently in focus and choose which one to zoom in on. With this function, the GUI of Comstat2 shines.

On Windows (XP or Vista) the userinterface gets a little more cluttered, especially when many windows are open. However, all windows will still be grouped together on the taskbar and in Vista when Aero is running a small image of the window pointed at will be shown when holding the mouse over the programmes button, so no critical problems arise. Thus creating a separate GUI for Windows would be a waste of time.

3.4.2 The "Comstat2"-window

The layout of the what appears to the user as the main window is straightforward: In the top, all the generic, program specific controls are placed. In the menus, it is possible to change the current program mode, switch off the saving of produced images, hide file-/directoryselector and finally exit the program. The Go-button simply starts processing the desired images with the functions added.

The added functions graphical parts are all placed in the lower part of the window. The programmer has completely free hands to add the necessary graphical components of her function, but in the development of the basefunctions a strict scheme with a checkbox for selection and possibly a line with a label and a textbox has been used. The final layout actually resembles the imagined one of the analysis quite well - see sec. 2.6.

One could argue that the looks are relatively dull, but the simplicity means that adding extra functions dynamically works great - and that the interface actually works across platforms (Windows XP/Vista/Vista x64/Mac OS X has been verified). The last should be a no-worry, considering the platform-independence of Java, but empirical evidence shows otherwise.

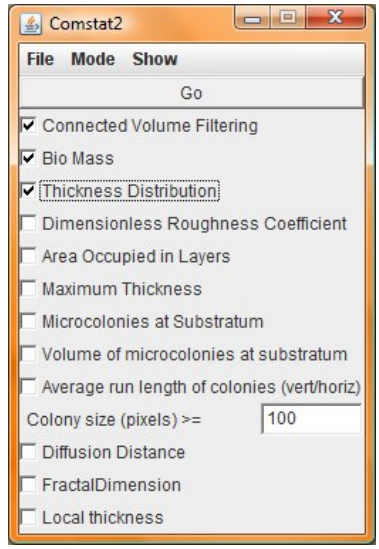


Figure 3.4: *The Comstat2-window that is in fact part of the ComstatToolbox class*

3.4.3 The "File Selector"-window

The File Selector is the visualization of the class ComstatVisualMode.

If you look at fig. 3.5, the upper combobox shows the list of open imagefiles, the lower one the images in the selected file. There is also a checkbox to indicate whether the selected image is to be used when running functions. The next two buttons are used for calling up the thresholding or reshowing the selected image if it has been hidden (closed) and finally, an add and a remove button is placed at the bottom for adding or removing files from the file combobox.

Basically, it would be simpler to have only one combobox for images - like the design made during analysis (See sec. 2.6), but this would clutter the perception of files as entities possibly containing multiple imagestacks.

Figure 3.5: *The visual-mode file selecetor.*

3.4.4 The "Directory Selector" window

This window is the visual representative of the ComstatMultiMode class that again is an expression of the "Scientist". Thresholding is adjusted to reflect the wishes of the user on the right. The textbox at the manual thresholding is in fact a little too wide; this is a result of the BoxLayout used resizing the component to the full width of the container - not much can be done about this without excessive coding.

The list on the left contains names of directories added.

In general, the interface is kept rather simple - some would argue that it is too simple in that it does not handle individual files (and perhaps even image-stacks) within directories. However, the user has this functionality through the ComstatVisualMode and she can also copy the files for a large batchrun into a new directory - that way the resulting data will also be saved to one (for each function) file.

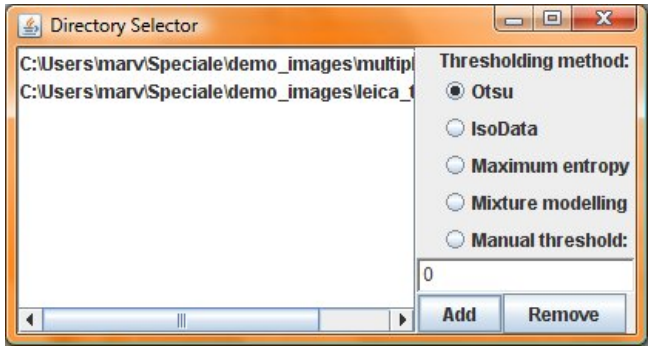


Figure 3.6: The directory selector with choice of thresholding

3.4.5 The "Log" window

The Log window is visible in both modes of operation. Since its only purpose is to show output and status messages, it is kept very simple. The need for a text log was not thought of in the analysis phase of the project, so no original sketch was made. However, in its first version, the Log window (which under the hood is contained in the ComstatMessageWindow) did indeed contain two buttons for saving and clearing the log. This just was not smart - in a crash, the contained text would be lost if not written to disk immediately and the clear

button was dangerous because it migth clear the current log before it has been written.

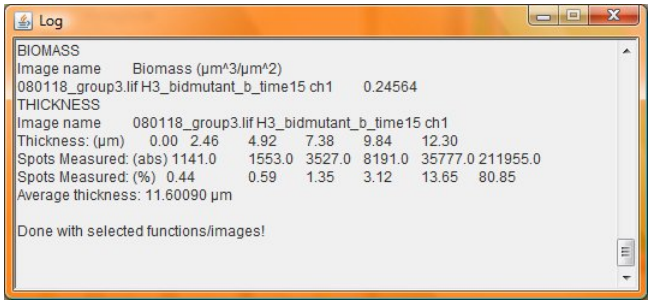


Figure 3.7: *Simplicity. The log window to which output from functions and status data are printed.*

3.4.6 The ImageJ window

As a final remark, the ImageJ window's status area is also used for printing status messages while functions are run. This makes it possible for the user to see whether the program is still running - especially when no printing to the log has been performed for a while.

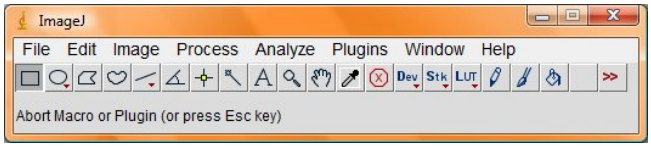


Figure 3.8: *The main ImageJ window. The statusarea is the gray area at the bottom saying "Abort macro or plugin...".*

3.5 Review of classes and their functions

The following is a runthrough of the implemented classes and their raison d'etre. The complete class diagram can be found in fig. 3.1 and 3.2. Some of the methods will be mentioned in the text, but the complete overview is only shown in the classdiagrams or in the enclosed javadoc.

3.5.1 Comstat2_

This is the main class of Comstat2. Notice the '_' in the name. It is simply a prerequisite for ImageJ to accept the class as a plugin. Basically, the class is responsible for creating the log-window, the file-loading windows the toolbox and adding functions to it.

3.5.2 ComstatFunction

This class is the main skeleton for the further implementation of the functions wanted. The class specifies that a Java SWING container called 'toolbox-Container' be created. The idea of this Container is that all required GUI-components for the function can be added to it and with ease be handled for both showing and if updates made to the components must be retrieved. The method runIfSelected is designed to be used to check whether the components in the container are changed in such a way that the function of the class containing it is to be run. The method must be overloaded in all childclasses (although it in a code related sense is not required).

Through the development of the program, various imagemanipulation methods have migrated from their originating classes (children) to ComstatFunction to increase their scope to all classes that inherits it. This saves a lot of redundancy and lowers the maintenance time required. The methods find8ConnectedBlobs (there are two - distinct in the datatypes on which they operate) are examples of such.

The class also contains a method for saving out imagestacks. One could with some reason argue that this ought to be placed in either of the fileloading classes (ComstatMultiMode or ComstatVisualMode), but due to its common use and lack of direct link to the Mode-classes the placement seems correct.

3.5.3 ComstatImageData

The class is the designed to unite images and often-used related metadata. The type of the images kept is simply ImageJ's built-in, ImagePlus. This type is very flexible w.r.t. pixeldata format and has a great viewer. It actually also contains some of the metadata that has been put into ComstatImageData, but it can be rather tedious to retrieve - several method-calls must be made in serial to get to them.

At load time the images voxelsizes (the sides of pixels, x and y) in the images of the imagestack and the distance between slices (z) are stored along the path to the orignal imagefile. All can later be retrieved using dedicated get's.

The class also contains a possibility to register changes to the contained ImagePlus. Although ImagePlus has its own "changes"-field, the control with it is completely up to the method that handles the image to change it or not. This is not always preferable - maybe changes are made to a copy of the image and only the imagestack is copied back to the original ImagePlus and info on changes are lost - thus, as a definition, the methods that do invasive procedures on an ImagePlus from a ComstatImageData must call the setChanged-method with a true-parameter. Alternatively, this method allows us to ignore the "save image"-dialog when closing ImageWindows because it only appears when the windows ImagePlus has its changes set to true.

3.5.4 ComstatMessageWindow

The ComstatMessageWindow has two distinct functions:

- printing status information/data to a log
- writing results from functions to files

The two are related in the fact that most of the information logged is results from functions. The class has a method called append that writes the string it is called with to an onscreen log-window, the on-disk version of the log (it must be written immediately - if the program crashes in the middle of processing a large stack of images, valuable data that could possibly explain the crash would else be lost) and a file in the processed image's directory, bearing the name of the function run. Based on the text in the string append was called with, the writeToTxt-method decides the filename under which the data is to be stored, either creates or reopens it and writes the necessary parts of the string. Prior to calling append, however, the setFileOutputDir should be called with the processed image's path.

3.5.5 ComstatMultiMode

This class is the imprint of the "Scientist"-actor found in the analysis process. The ComstatMultimode does three things:

- keeps a list of directories to be examined for CLSM-images
- loads images on request
- thresholds images according to user demands

The directory way of loading images makes it easy to process vast amounts of images and in combination with the many thresholding options makes long unsupervised batch runs available.

Considering the thresholding algorithms, the class implements 5:

- iso data
- maximum entropy
- mixture modeling
- Otsu's method
- manual threshold

The last one actually is not a "method" but a possibility to set one threshold for all images. Its inclusion is on behalf of Arne Heydorn, the author of the original Comstat, and the reasoning is that being able to set a manual threshold for all, variance in output data can be made lower.

The choosing between the thresholdings is done via a window where the user can choose between them. Once a command is received to start loading images, the load-method is called and starts to look through the added directories in search of microscope-files. The images found are put into memory using the appropriate method (openInfoFile, openLIFFile, openLSMFile). The desired thresholding, i.e. Otsu, is called with the loaded image as parameter (through the threshold-method) , a suiting value will be found and finally the performThresholding is called. The latter physically alters the data in the image to reflect the given threshold, contrary to ImageJ's built-in thresholding that just stores the value and updates.

Note that the thresholding methods are direct adaptions of the ones that come bundled with ImageJ. The reason for putting them directly in the code of Comstat2 is simply control. Including it guarantees that no incompatibilities turn up at a later time.

3.5.6 ComstatToolbox

This is probably the class with the most saying name. Aside from the generic userinterface components, objects of this class holds a list of installed functions, a placeholder for the installed functions GUI-components and the method addFunction for the addition of further functionality.

ComstatToolbox also has control with the two usermodes. These can be changed via a dropdown menu in the class window in runtime. The toolbox makes available a method (runThroughFunctions)for running all active functions on a given set of images delivered in a Vector.

3.5.7 ComstatVisualMode

ComstatVisualMode is the embodiment of the apprentice actor found in the analysis phase. The interface allows him to add files containing many imagestacks and to treat them individually. Normally, images are rendered to the screen to underline changes and allow the user to manually change the colourspace used (the look-up-table) and use all other imagemanipulation tools and plugins of ImageJ on the images, but it is also possible to hide (and reshow, using the show method) loaded imagestacks.

Where the ComstatMultiMode used its own implementations of the thresholding algorithms, the ComstatVisualMode uses a 3rd party plugin called MultiThresholder. The plugin contains all the same thresholding methods as ComstatMultimode but can visually show the parts that will be removed by the use of a special lookup table. The use of the plugin is, however, a solution made due to time constraints.

The class contains its own implementation of fileloaders. When a user presses the "add" button to add a file, a generic OpenDialog appears and allows the user to select a file. Depending on the filetype, one of the implemented methods for opening is called with the path. Then the filename is added to the filelist and the names of the contained imagestacks are added to the images list. Internally the imagestacks are stored in a Hashtable.

Chapter 4

Functions & Algorithms

This is a runthrough of the implemented functions. For more info on the ones installed in the original Comstat, see [3]

4.1 Connected Volume Filtering

4.1.1 Purpose and Definition

The purpose of Connected Volume Filtering (CVF) is to remove elements that are not part of a biofilm from an image stack. If an element in the image stack is not in some way connected to the substratum it is not considered part of a biofilm but rather a piece of floating debris. The debris can be of any kind, but it is likely pieces of torn off biomaterial or even part of a biofilm connected to the substratum outside the scope of the image stack under observation.

4.1.2 Implementation in Comstat

In Comstat, the function takes as input a 2D matrix, `startlayer` containing the substratum and a 3D matrix `inmatrix` containing the complete image stack.

The first layer in `inmatrix` is the same as `startlayer`. Thus, `startlayer` is strictly speaking not necessary and neither is the element-by-element multiplication made between the two in order to obtain the substratum of the resulting `filt_images`. All elements found in the substratum are naturally touching the substratum.

Next step is to compare the current layer pixels to those of the next. If a pixel is set in both, its position is stored. Once the check is done for the entire layer, the program moves on to the next layer. Here, the program uses the build-in Matlab-function *bwselect* to expand from the common pixel positions in an 8-connected fashion. Once this is done, the current layer is multiplied element-by-element by the corresponding layer from `inmatrix` and the result is stored as a layer in `filt_images`. The current filtered layer is now used for finding common pixels with the above layer etc. until the entire stack in `inmatrix` has been processed.

4.1.3 Requirements for reimplementation

Aside from a lot of mathematical notation made possible by the use of Matlab, the only function that takes special attention is *bwlabel*. Looking at Matlab's help entry for bwlabel one can quickly observe that it is a very far-reaching function that has a lot of unnecessary functionality (in this case, anyway) and relies on several other build-in Matlab functions. Thus, the most fair way to reimplement the functionality of *bwlabel* would be to make a custom implementation that suits the purpose.

The purpose is to take some pixels as input and expand as much as possible in an 8-connected fashion.

4.1.4 Reimplementation of Connected Volume Filtering

The general algorithm goes like this:

1. Make an outputstack the same size as the inputstack

2. Copy the substratum from input to output. The current layer is now the one just above the substratum.

3. Copy the previous layer to a worklayer.

4. Find common pixels and expand from them in an 8-connected fashion in the worklayer.

5. Copy the worklayer to output.

6. The previous layer is now the old worklayer. The currentlayer is now the next layer in input. Repeat from 4 until the entire inputstack has been processed.

In this runthrough, the 8-connected expansion was very gently ignored. However, this is the difficult part of the algorithm.

4.1.5 8-connected Expansion

As mentioned earlier this is what *bwlabel* does in Comstat. Basically, there are two ways to implement this function: An iterative and a recursive approach. Although the recursive approach is often the most beautiful and simple, an iterative solution was chosen because of the more transparent memory-usage pattern. Using recursion could possibly create stack overflows.

The algorithm in Comstat2 is based on a list of not-yet visited, set pixels which is updated until empty. Notice, that the algorithm is based on 1D-arrays as this is the standard in ImageJ - and in this case it saves us from creating a new datatype. The complete algorithm goes:

```
Create list "cmn" with indexes of common pixels
"last" is made to point at the last taken index in cmn
while cmn isn't empty
   look at pixel px with index defined by cmn[last]
   set px as read by setting it to infinity
   decrease last
   examine px's 8 neighbours
   add neighbours index to cmn if set and not infinity
   update last according to number of added neighbours
end while
```

The memoryconsumption of the algorithm is linear with respect to the number of pixels in the image (and the number of layers in the inputstack). By taking a look at the code, one will see that several versions of the function called *find8ConnectedBlobs* (See fig. 3.2) exist. this is because the function is reused in other algorithms that are to work on other datatypes than the raw byte pixel type used for normal images.

4.2 Calculation of biomass

4.2.1 Purpose and Definition

Biomass can be many different things. By looking in a traditional encyclopedia, the biomass is the amount of biologic material present in a given area or sample. In this context, however, it is defined to be:

$$\frac{V}{a} = \frac{[\mu m^3]}{[\mu m^2]}$$

In words, the biomass volume pr. unit area. This is an expression of how much of the imagestack is covered by bacteria.

4.2.2 Implementation in Comstat

This is the simplest function of Comstat. The area of one pixel, the volume of one voxel (the area of one pixel times the distance between layers) and the stack itself is provided to the function.

After counting all occupied pixels in the stack, the biomass is calculated as:

```
y=count*voxel/(xyarea*size(bb,1)*size(bb,2));
```

`bb` contains the 3D-stack.

4.2.3 Reimplementation

Basically, the reimplementation is as simple as the original:

```
create a variable "count"
for all layers in the imagestack
   for all set pixels in the layer
    increase count
   end for
end for
calculate the biomass as:
biomass = count*voxel/(xyarea*stack_height*stack_width);
```

The biomass is then printed through the local ComstatMessageWindow. As one of the only functions, biomass has a specific variable associated in the metadata-file, ComstatImageData. The value is at this point stored. However, if it is to be used later a recalculation is preferable as changes to the stack may also have influenced the biomass.

4.3 Thickness Distribution

4.3.1 Purpose and definition

The thickness distribution is - as its name implies - a list containing the thicknesses found and their prevalence. Additional info such as the avg. thickness and the projected coverage in both pixels and percent is also delivered, the former also in a separate file.

The function is relatively primitive and can thus be tricked by holes in biofilms or floating debris if connected volume filtering has not been performed. See fig. 4.1

4.3.2 Implementation in Comstat

As the one and only function in the original version of Comstat, the thickness distribution runs from the top of the imagestack and down to the substratum. Conceptually, this is the most obvious as we find the uppermost elements of the biofilms first and not some that are later superseeded by higher occurrences. The implementation uses little specific *Matlab*-notation and makes no calls to other *Matlab*-functions so to avoid repetition, the algorithm is examined in the reeimplementation.

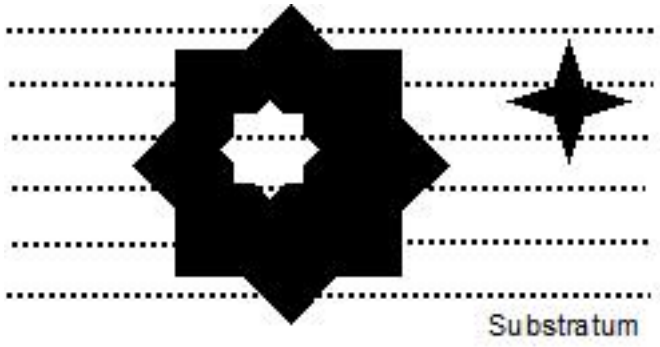


Figure 4.1: *For the large object, the thickness (measured from the top down) would be the same whether or not there was a hole as indicated. The height in the part of the imagestack where the little object floats around would indicate that the volume it projects unto the substratum was filled.*

4.3.3 Reimplementation of Thickness Distribution

The original implementation took as input an imagestack and the distance between the slices in the stack. Luckily, the distance is delivered alongside images so we only need an image. The desired output was mentioned above.

The following is the first part of the algorithm used:

```
ThickDist(imgStck)
create "nOfSlices" = number of slices in imgStck
create "pixelSizeZ" = pixel size in z-dir between slices in imgStck
create "binImgStck" = imgStckToBin(imgStck) //binary conversion
create array "hM" same dimensions as images of imgStck //hM = heightMatrix
hMNotEmpty = 0;
for all slices sl of binImgStck{
   for all indexes i of sl
      hMNotEmpty = (hM[i]>0)?1:0 //condition?true:false
      superseeded = sl[i]-hMNotEmpty*sl[i]
      hM[i] = (hM[i]+superseeded*pixelSizeZ*(indexOf(sl)-1));
   end for
end for
//To be continued...
```

In words, the algorithm starts looking at the top of the imagestack (farthest from the substratum), going down. In the beginning all elements of `hM` will be empty

and thus `hMNotEmpty` will be false (0). This makes the variable `superseeded` take the value at the current index (pixel) in the binary imagearray. The next line will then store the thickness in `hM` as the distance between slices times the height of the previous slice. If a higher placed pixel was previously registered, `hM` will not be zero at that particular index which results in `hMNotEmpty` becoming true (1) and subsequently `superseeded` becoming false (0). Multiplying `superseeded` unto the calculation of the height of the slice, returns 0, so the value already stored in the current index of `hM` will be stored back in the same place.

Now the heights of all pixelcolumns have been found so the transform of the data into a printable form is made. First the average thickness:

```
//...continued:
create "avgThickness" = 0
for all indexes i in hM
   avgThickness = hM[i]+averageThickness
end for
avgThickness = avgThickness/(height(imageStack)*width(imageStack))
//To be continued...
```

The average thickness is calculated as a total for all pixels in the 2D image. Alternatively, this could have been done as $t_{avg} = \frac{\sum value(px)}{\sum px \neq 0}$, that is: as the average of set pixels. One could argue that it would be more correct to use only the relevant pixels but as it is now, the average tells us about the general shape of the biofilm, especially in relation to other, similar images.

Now, the distribution itself is made from `hM`:

```
//...continued
make array "thickness" with nOfSlices elements
make array "spots" with nOfSlices elements
make array "spotsPercent" with nOfSlices elements

for i=1 to nOfSlices
   sl = imgStck[i]
   for all pixels px in hM
      if value(px) = pixelSizeZ *(i-1)
         spots[i-1] = spots[i-1]+1
      end if
   end for
   thickness[i-1] = pixelSizeZ *(i-1);
```

```
   spotsPercent[i-1] = 100*(spots[i-1]]
      /(height(imageStack)*width(imageStack)))
end for
//The End!
```

The contents of `thickness`, `spots` and `spotsPercent` are now the possible thicknesses, how many pixelcolumns have that thickness and the same number given as a percentage. The values (including the average thickness) is printed using the local `ComstatMessageWindow`.

Please note that the algorithm of the old Comstat, although identical in function, worked a little different in some points, i.e. when finding indices with the same value in the height matrix. Here the old version makes all the occurences of the sought value zero and then the function `find` is used to find all non-zero indices, that are then counted. This has a much higher complexity, so it is no surprise that the new implementation runs faster, albeit improvements are possible.

Finally, the algorithm for converting the imagestack to binary form:

```
imgStckToBin(imgstack):
make new image stack "output" same size as imgstack
for all slices sl in imagestack
   for all pixels px in sl
      if value(px)>0
         pixel at px's place in output, px_out = 1
      else
         pixel at px's place in output, px_out = 0
      end if
   end for
 end for
 return output
```

"Binary" is not related to the datatype used but rather that 1 and 0 are the only values used a la the normal use in C++.

4.4 Dimensionless Roughness Coefficient

4.4.1 Definition and purpose

The definition of the roughness coefficient stems from the original definition of the average roughness coefficient, R_a, given by

$$R_a = \frac{1}{L} \int_{x=0}^{x=L} |y| \, dx$$

in 1D. Fig. 4.2 explains the terms.

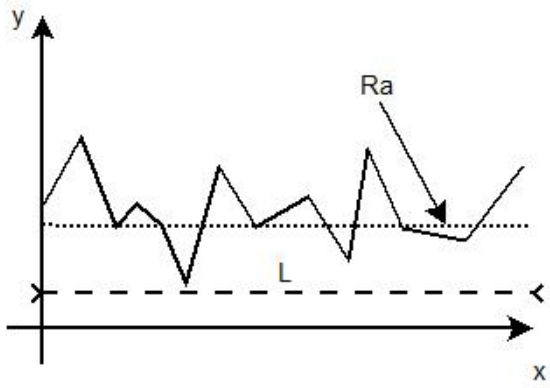


Figure 4.2: *The term R_a simply covers the average height of a function*

However, it has been extended to

$$R_a^* = \frac{1}{N \cdot t_{avg}} \sum_{i=1}^{slices} spots[i] \cdot (thickness[i] - t_{avg})$$

where N is the total number of spots found (See 4.3), $spots[i]$ contains the spots of slice i and $thickness[i]$ contains the thickness defined by slice i. t_{avg} is the average thickness as found in 4.3.

In words, R_a^* is the average deviation from the average thickness pr. spot, normalised and dimensionless.

The origin of this definition is an article printed in Biotechnology and bioengineering that has not been recovered.

4.4.2 Implementation in Comstat

In the original program, the thickness distribution was first found and then R_a^* calculated as:

```
N=sum(height_distri(:,2)); % Total number of thickness points
ialt=0;
for ii=1:length(height_distri(:,1))
   ialt=ialt+abs(height_distri(ii,2)*(height_distri(ii,1)
      -average_height));
end
if average_height~=0
   Ra_star=1/N/average_height*ialt;
else
   Ra_star=0;
end
```

4.4.3 Reimplementation in Comstat2

The relatively simple algorithm uses no advanced Matlab functions, so reimplementing was straight-forward:

```
Run ThicknessDistribution
for i=0 to slices
   N = N + spots[i]
   ialt = ialt + |spots[i]*(thickness[i]-averageThickness)|
end for
if(averageThickness!=0){
   RaStar = ((1/N)/(averageThickness))*ialt
else
   RaStar = 0
end if
```

`RaStar` is then just printed the traditional way.

4.5 Area Occupied in Layers

4.5.1 Definition and purpose

The functionality of this function is very clear given its name: For all layers in an imagestack, the number of set pixels must be found, the area found using the known pixel size and the covered area as a percentage of the total slice size

The area occupied is another expression of how much biomaterial is in the imagestack, however, with a possibility of seeing the distribution over the layers.

4.5.2 Implementation in Comstat

The original version does not print the three numbers mentioned above, but only the coverage as a percentage. The calculation itself is:

```
imagestack //the CVF-filtered imagestack
create array bactCover with as many elements as
   slices in imagestack
for all slices sl
   bactCover[numberOf(sl)] = number of non-0 elements in sl
      /(width(imagestack)*height(imagestack))
end for
```

The result in `bactCover` was then multiplied by 100 and printed as a percentage.

4.5.3 Reimplementation

All the numbers discussed in sec. 4.5.1 are available through the calculation of the coverage percentage and thus they might as well be printed. Although the percentage does give a good relative estimate of how much biomaterial is in the layers, small differences will not appear from a percentage so there is good reason to print the absolutes as well. One should, however, notice that there is a higher risk of the thresholding influencing on the absolute number of set pixels than on the percentages.

The implementation is made in a more or less similar fashion:

```
AreaOccupiedInLayers(imagestack)
create array bactCover with as many elements as
   slices in imagestack
for all slices sl
   bactCover[numberOf(sl)] = number of non-0 elements in sl
   bactAreal[numberOf(sl)] = bactCover[numberOf(sl)]
      *pixelSizeX*pixelSizeY;
   bactPercent[numberOf(sl)] = 100 * bactCover[numberOf(sl)]
      /(width(imagestack)*height(imagestack))
end for
```

Again, the values for each slice are printed to file and log using the local `ComstatMessageWindow`.

4.6 Maximum Thickness

4.6.1 Purpose and definition

When comparing names, maximum thickness resembles the thickness distribution a lot. And one might ask oneself why one does not just look at the greatest thickness found doing the thickness distribution. The two functions are, though they have similar names, distinct in their way of defining the thickness. Whereas the method used in finding the thickness distribution disregards holes and other cavities and just registers the highest set pixel in each pixelcolumn as seen from above, the maximum thickness finds the compacted thickness as illustrated by fig. 4.3. Thus no information on the general shape of biofilms is found in the maximum thickness.

4.6.2 Implementation in Comstat

The original implementation in Comstat was based on the same method as the thickness distribution, that is: It does not regard holes in biofilms. It simply goes like:

```
imagestack //prefiltered using CVF
if more than 1 slice in imagestack
   for all slices sl in imagestack, starting from substratum+1
```

Figure 4.3: *Due to the nature of the Maximum Thickness function, the two thicknesses T_1 and T_2 are equal. Only slices with contents are observed.*

```
      sum all values in sl, save in "tot_px"
      if tot_px>0
         "max_thick" = pixelSizeZ * (numberOf(sl)-1)
         //-1 is because the thickness starts below current sl
      end if
   end for
else
   max_thick = 1;
end if
```

Strictly speaking, the algorithm is inefficient, as it would save a lot of time if the algorithm started from the top of the imagestack instead of the bottom. The thickness value in `max_thick` could then be printed.

4.6.3 Reimplementation

Since all elements in each layer of the imagestacks was observed under all circumstances in the old algorithm, it is nearly costless to find the thickness in each pixelcolumn as seen from above and store it in a new array. Then, when all slices have been observed, the maximal thickness can be found as the largest index. The algorithm:

```
imagestack
create "max_thick"=1
create array "height" the size of a slice of imagestack
if more than 1 slice in imagestack
```

```
   for all slices sl in imagestack, starting from substratum+1
      for all elements i of sl
         if sl[i]!=0
            height[i] = height[i]+1
         end if
      end for
   end for
   for all elements i of height[]
      if height[i]>max_thick
         max_thick = height[i]
      end if
   end for
end if
max_thick = max_thick * pixelSizeZ
   //max_thick-1 depending on indexing method
```

The thickness found is the compacted thickness as described in the definition.

However, since Comstat2 must return the same result as the original, the definition used in the original is the one implemented. That one simply runs through the layers from the bottom and finds the highest one that has set pixels. The maximum thickness is then:

```
max_thick = (nOf(sl)-1) * pixelSizeZ
```

4.7 Microcolonies at Substratum

4.7.1 Definition

As the name implies this function finds the microcolonies present at the substratum of an image stack and identifies them uniquely. Unlike all other functions in Comstat2 Microcolonies at Substratum (MaS for short) only works on the first image in the stack (alternatively, the two first if smacking is used).

4.7.2 Implementation in Comstat

The original implementation of MaS takes as input an image containing the substratum and an integer describing the minimum number of pixels required

in a connected group in order to define a colony.

In the other end, a list of the size-distribution of the contained colonies, an image of the resulting substratum (with all colonies smaller than the defined value and single pixel noise removed) and an image in which all colonies are identified using unique colours are returned.

Under the hood, input images first have single-pixel (salt'n'pepper) noise removed and are then median- and erosionfiltered using the Matlab-function *ordfilt2*. The latter filters are also applied to remove noise; the median filter will lower the immediate changes in areas with great contrasts and the erosion filter vil remove thin, long filaments along the edges of colonies. The importance of these filterings surfaces in the next step: Pixels that are 8-connected are found and coloured similarly using the function *bwlabel* - and all groups (of pixels) with less than the specified number of members are made background. Had the images not been filtered, colonies would perhaps be in-/excluded or erroneously seen as a single colony based on noise.

4.7.3 Requirements for reimplementation

In order to implement the MaS in Comstat2 using Java, the following was obviously needed:

- a single pixel filter
- a filter-function similar to *ordfilt2*
- a function for finding connected components

At the time of the implementation 2 distinct solutions for the filters were possible: Either a complete implementation of *ordfilt2* (and the remaining code copied from the original Comstat) or specific implementations of the filters should be made. As *ordfilt2* consists of several hundred lines of code and relies heavily on the matlab api the decision to make specific implementations seems most logic.

4.7.4 The Single Pixel Noise Filter

The original implementation of this filter actually also uses *ordfilt2*. This is, however, unnecessary as the operation of the filter is straightforward. It goes as

follows:

```
for all pixels:
   if any surrounding pixel is set
      leave the current
   else
      make the current pixel background
   end if
end for
```

Once the algorithm has been run, it must be run again with the meaning of fore- and background reversed. It will thus remove single pixel noise within colonies.

4.7.5 The Median Filtering

The essence of median filtering is to observe a pixel px and its neighbourhood (3×3, 5×5, etc.) and find the median of the set made up of the pixel values. Consider the following example:

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 2 & 2 \\ 1 & 2 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & 0 & 0 \\ 0 & px & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Where $px = 0$. We now apply a 3×3 median filter to input:

$$medfilt\left[\begin{pmatrix} 1 & 2 & 1 \\ 2 & 2 & 2 \\ 1 & 2 & 1 \end{pmatrix}\right] \rightarrow$$
$$px = median\,(1,1,1,1,2,2,2,2,2) = 2$$

and the output becomes:

$$B = \begin{pmatrix} 0 & 0 & 0 \\ 0 & px & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The above example is overly simplified as we are only able to find the median of the center pixel - px is in the general case not the center pixel. For all other pixels in A we will touch the edge. The solution chosen is to always make sure that the input has a frame of 0's added before filtering and then letting the

algorithm start with an offset of 1 into it in both directions. This solution will often lower the median at the edges but the alternative is not to have a value at all.

The filtering is performed on all pixels in succession, but cannot be done in place. The space requirements of the algorithm is $2n$ with n being the number of pixels. The pseudocode algorithm goes as follows:

```
make an empty array  "out" the same size as the input
for all pixels px in the input
   put the eight neighbour pixels and px in an array
   sort the array in a- or descending order
   store the 5th element of the array in px's place in out
end for
```

The 5th element (no pun intended) is chosen because it is the middle element in an array with 9 elements and thus will contain the median if the array is sorted in either fashion.

4.7.6 Erosion Filtering

As with a median filter, the erosionfilter is based on observing a pixel px and its neighbourhood. However, a structuring element is involved here. Consider the following example:

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad struct = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

$$B_{before} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad B_{after} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

In A, 0'es are already padded along the edges so the filter may be applied directly with an offset of 1 in both directions. The application itself is done by centering $struct$ on all pixels px in A and checking whether all non-zero compnents of the structuring element ($struct$) are covered by non-zero elements in A. Only if this is true the pixel in px's place in B will become set. In the example only the

centerpixel of B gets set. This filter can, as implied in the necessity of B, not be applied in place so again $2n$ is the space requirement, n being the number of pixels.

The algorithm implemented goes as follows:

```
create empty array "out"
for all pixels px
   put px's 3x3 neighbourhood in an array (not px)
   sort the array in ascending order
   put the first element of the array in pxs place in out
end for
```

The algorithm only works because we make sure that the image is in a binary state before execution and because we are using

$$struct = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

as structuring element. Thus if we take the eight elements around px in the input, sort them and find 0 as the smallest element we know that one or more of the neighbours are 0.

4.7.7 Connected Components Analysis

Connected components analysis (CCA) has already been implemented in connected volume filtering (CVF). In CVF the algorithm works in 3D to determine if pixels are somehow connected to the substratum. Here, CCA is to work only in 2D and with a slightly different purpose.

The function *bwlabel* from Matlab takes as input a 2D array (i.e. an image), finds the connected components (either 4- or 8-connected) and labels all elements according to grouping. The following examples show how *bwlabel* would work on a matrix A using 8-connection:

$$A = \begin{pmatrix} 1 & 3 & 4 & 0 & 3 \\ 2 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad A_{labeled} = \begin{pmatrix} 1 & 1 & 1 & 0 & 2 \\ 1 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The CCA implemented in CVF is based on an iterative algorithm that will only find all connected elements and keep them in the image but not label them. The CVF's CCA is working great so it is reused and a labeling algorithm is implemented.

4.7.7.1 The Recursive Blob Colouring Algorithm

The algorithm for colouring blobs can be seperated in a part for locating the blobs in the image and a part for colouring. Before using this algorithm on an input matrix (image) A, it is necessary to ensure that all elements of A are in a binary form so that all background pixels are 0 and foreground ones are 1. This way, the locating algorithm will have to look only for ones and need not take the value of previous colourings into account. The locating algorithm is the one called when blob colouring is to be performed and hence, it has been named `ColourAllBlobs`:

```
ColourAllBlobs(A):
set initial colour = 2
for all pixels px
   if value(px)==1
      call ColourBlob with px and colour
   end if
   colour++
end for
```

The part of the algorithm doing the actual colouring:

```
ColourBlob(pixel,colour):
set value(pixel) = colour
if any neighbour  npx in 8-connected neighbourhood is !=1
   call ColourBlob with npx and current colour
end if
```

In line 3 it is very important only to search for 1 in the neighbouring pixels. If the check was changed to $\neq 0$ or > 0, pixels previously coloured would be called with ColourBlob again and the program end up in a possibly-neverending state.

The recursiveness of `ColourBlob` ensures that the algorithm does not go "bottoms up" before all pixels in a blob have been coloured and since `ColourAllBlobs` observes all pixels in the input image, all blobs will be found and coloured in a unique way.

4.7.8 Preparation of data

After labelling the colonies, data must be harvested. First the sizes of the colonies are found and all that are smaller than the preset minimum size removed. One could argue that this should be done prior to the colouring, but at that time no unique identification of the blobs were available and so the removal would have to be done in a fashion resembling the colouring and be pretty expensive.

After the removal of lesser than wanted colonies, there may be vacant indexes in terms of colour value in the colony image, so compacting must be performed. The algorithm is quite simple:

```
find largest colour value max_c
while max_c has not been reached
   find smallest colour value > 1 and store in min_c
   subtract 1 from all colour values greater than min_c
   update max_c
end while
```

- yet rather uneffective.

4.7.8.1 Data required

In accordance with the original Comstat, MaS must print:

- Number of microcolonies
- Areadistribution of microcolonies
- Average colony size
- Average pixel intensity of colony pixels

The number of found microcolonies is $n_{col} = max_c - 1$ where max_c is the maximum colour index after compacting.

The area distribution is found by going through the labeled image counting the number of pixels with a given index. Since this is to be done for all indexes in the image it can be done like this:

```
create an array dist with n_col elements
for all px in labeled image
   dist[value(px)]++
end for
```

The distribution can now be printed; the colony number (colour) is the offset into `dist` and the value at that place is the size in pixels. Since the area of a pixel is known ahead, the actual physical-world size can be determined as $a_{real} = n_{px} \cdot a_{px}$ where n_{px} is the number of pixels, a_{px} is the area of one pixel.

The average colony size is just the total amount of set pixels divided by n_{col}. The average pixel intensity $\overline{px_i}$ is not really linked to the labeled image but is:

$$\overline{px_i} = \frac{\Sigma(value(px))}{n_{tot}}$$

where px are the pixels of the original input substratum with the lesser colonies removed and n_{tot} is the total number of pixels in the image.

All output is written using the asociated `ComstatMessageWindow` (See xxx).

4.8 Volume of Microcolonies at Substratum

4.8.1 Purpose and definition

This function is related to Micro Colonies at Substratum in the sense that the latter is used to find microcolonies and the first expands them in the imagestack to find the volumes they fill.

Since only 2D images arranged with a predefined distance between them are available, we must defined how we measure volumes. The chosen solution is:
For all all layers starting from the substratum, if a pixel is set, the voxel defined by the perpendicular projection of the pixel onto the next layer is part of the total volume of the current colony.
For the top layer, the projection just goes d into the void, where d is the distance between layers. For an illustration, see fig. 4.4

Techically, this means that the function "just" has to find all pixels belonging to a colony and multiply with the x, y and z sizes of voxels to find the desired volume. The trouble comes in as we only have the substratum with separated colonies.

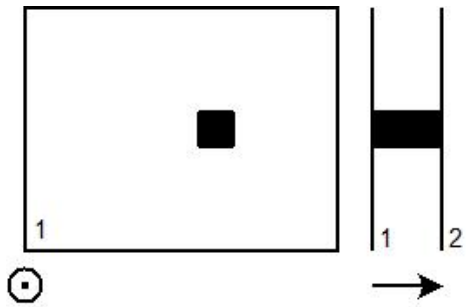


Figure 4.4: ***Left:*** *The substratum (layer 1) seen from above. 1 pixel is set.* ***Right:*** *The same substratum with the set pixel projected onto the second layer.*

4.8.2 Implementation in Comstat

The original version of this function uses *bwlabel* to find colonies in a binary version of the first image of an imagestack. *bwlabel* enumerates the colonies so everyone has its own index. Further, the algorithm does:

```
inmatrix //imagestack
startImg //substratum
if startImg contains colonies
   lbld_start = bwlabel(startImg)
   max = max(lbld_start)//1 = substratum
   for all colonies in substratum
      area_in_layer = set all other colonies 0, current to 1
      sum_voxs = sum pixels in area_in_layer
      for all slices sl //substratum not included
         com_px = common between prev. and cur. slice
         coords = coordinates of set pixels in com_px
         area_in_layer = bwselect(sl,coords,8)//8 as 8-conn.
         sum_voxs = sum_voxs + sum of pixels in area_in_layer
      end for
      col_vol_dist[valueOf(slice)] = sum_vox
   end for
   sort(col_vol_dist)
else
   col_vol_dist = 0
```

Finally, the values in `col_vol_dist` are multiplied by the voxelsize and printed one by one.

4.8.3 Reimplementation

The old implementation for finding the volumes of microcolonies relied heavily on the *bwlabel* and *bwselect* that was previously delt with in section 4.7. The idea is to find all 8-connected elements given some starting point(-s) - exactly the same problem that was solved by `find8ConnectedBlobs`.

As the two implementations are practically only different in the way connected pixels are found and the, only the inner for-loop is shown in text here:

```
startImg = MicroColoniesAtSubstratum(imagestack)
\\...
for all slices sl //substratum not included
   com_px = common between prev. and cur. slice
   area_in_layer = find8ConnectedBlobs(com_px,sl)
   sum_voxs = sum_voxs + sum of pixels in area_in_layer
end for
\\...
```

`startImg` contains the colonies of the substratum. Technically, the algortihm resembles the Connected Volume Filtering a lot. Voxelsize is found because the pixelsize in the x, y, and z direction are delivered alongside an imagestack.

4.9 Average Run Length of Colonies

This function is all new. The idea is to find all lengths of colonies in the substratum. The length is measured as consecutive set horizontal or vertical pixels as shown in fig. 4.5. This function first identifies all colonies at the substratum and then the length is found for all vertical and horizontal lines in the colony and the average is found. The inclusion of the runlength is inspired by [1].

4.9.1 Implementation in Comstat2

There is already a function implemented for finding the individual colonies, so this is of course used. Then we do as follows:

```
subst = colonies in substratum of imagestack
```

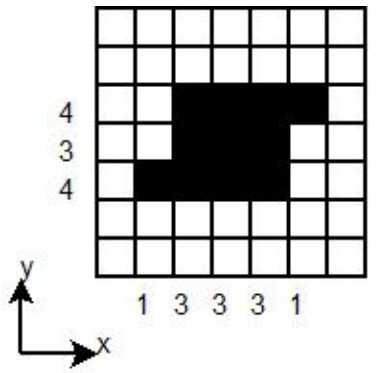


Figure 4.5: A colony and its horizontal and vertical runlengths.

```
for all colonies col in subst
   disregard all other colonies in subst
   px_tot = number of pixels in col
   columns = columns in col
   rows = rows in col
   avgRunLengthH[col] = px_tot/columns
   avgRunLengthV[col] = px_tot/rows
end for
```

Finally, `avgRunLengthH` and `avgRunLengthV` can be printed.

4.10 The Fractal Dimension

4.10.1 Purpose and definition

The subject of Fractal Dimension has no fixed definition. The definition used here is based on work done by Mandelbrot. He showed in his paper how the length of the British coastline depended upon the length of the measure used. In his own words: "Given a yardstick of length ϵ, walk the yardstick along the coastline. The number of steps multiplied by ϵ is an approximate length $L(\epsilon)$ of the coastline. For a coastline, when ϵ becomes smaller, the observed length $L(\epsilon)$ increases without limit". In his paper, he further used work by Richardson who defined the fractal dimension from the equation

$$L(\epsilon) = M\epsilon^{(1-D)} \tag{4.1}$$

as D where $D > 1$ and M is some positive constant. The term fractal dimension was, however, not used in the paper.

In Allen et al. several measures for obtaining D are found. Especially the method where a stick $2\epsilon_1$ long is held perpendicular (ϵ to each side) to the coast as one walks along it was interesting. When the traversal is done, the area covered a_{coast} can be calculated and the process repeated using another stick $2\epsilon_2$ long. Subsequently, the length can be found as

$$L(\epsilon) = \frac{a_{coast}}{2\epsilon}$$

for both ϵ_1 and ϵ_2. Now, this does not give us the fractal dimension but we can plot the results as in fig. 4.6 and see the slope $s = (1 - D)$. s can also be calculated as

$$s = \frac{log(L(\epsilon_1)) - log(L(\epsilon_2))}{log(\epsilon_1) - log(\epsilon_2)} \rightarrow D = 1 - s$$

And thus the fractal dimension is found from it.

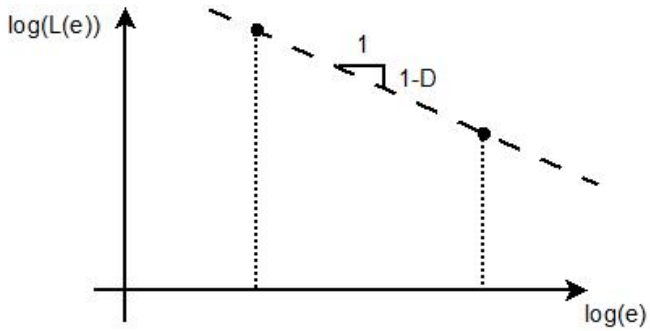


Figure 4.6: *Plotting $log(L(\epsilon))$ vs. $log(\epsilon)$, we can find the slope to be (1-D).*

4.10.2 Implementation in Comstat

The fractal dimension was in fact implemented in the original Comstat in a different form. However, it often returned erroneous results, so it was disabled. The function implemented was based on the so-called Minkowski-sausage logic, but as it did not work, there is only little point to going in detail with the implementation.

4.10.3 New implementation in Comstat2

The function fractalDimension will look at the substratum of an imagestack and find fractal dimensions for all colonies found.

The main algorithm is:

```
find all blobs in image input using getBlobs
for all blobs b in input
   disregard all other blobs in input
   find a random background pixel px outside the blob
   call colourBackground starting at px
   All background left must be a sea in the current blob
   colour background the same colour as the current blob
   call outline on image
   call calculateFD
   add FD to sum of FD's
   print FD
end for
divide sum of FD's by number of blobs and print
```

The algorithm to find blobs:

```
getBlobs:
make the substratum binary
remove single pixel noise
do erosion filtering
do median filtering
find 8 connected blobs
colour all blobs
remove all blobs less than 10 pixels
return substratum
```

All subfunctions of *getBlobs* are described elsewhere (filters, find8ConnectedBlobs and colourAllBlobs can be found in the section on Micro Colonies at Substratum)

The algorithm that outlines blobs:

```
outline(inarray):
```

```
create "outarray" the same size as inarray
for all px in inarray
   if px is set and either 8-connected neighbour is not set
      set pixel at same position as px in outarray
   end if
end for
```

Most of the above algorithms all delt with the preparation for finding the fractal dimension. The algorithm itself is based on the procedure desribed in the definition and a paper by Flook on using dilation logic to find the fractal dimension:

```
CalculateFD(outlined_image):
dilate(outlined_image, small) and save number of set pixels in "a1"
L1 =a1/3
epsilon_1 = 3/2 //because struc is 2*epsilon in diameter.
dilate(outlined_image, big) and save number of set pixels in "a2"
L2 =a2/7
epsilon_2 = 7/2
"s"=log(L1/L2) / log(epsilon_1/epsilon_2) //slope
"FD" = abs(1-s)
return FD
```

The dilation is implemented as a specific implementation. This can be argued, but it would be rather simple to expand to using a structuring element after own choice instead of two predefined ones. The current procedure is:

```
dilate(in,struc_size):
make new array "out" same size as in
if struc_size=big
   use circular struc w/diam. 7
else
   use circular struc w/diam. 3
end if
for all pixels px in image
   if px is set
      insert struc centered on px in out
   end if
end for
return out
```

Instead of the variable that decides whether to use the large structuring element, one could imagine using an array to supply ones own. A little change would

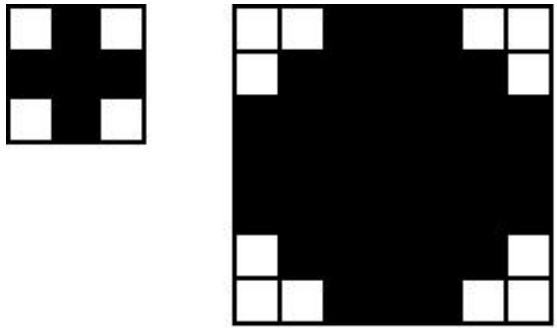


Figure 4.7: The two structuring elements used for dilation

have to be made to the part that writes in out (due to the spread out over the neighbourhood)

4.11 Diffusion Distance in 3D

4.11.1 Purpose and definition

The diffusion distance is the minimum distance from a living cell to the void. Void could potentially contain nutrients, oxygen, etc., so the diffusion distance plays a serious role in the development of biofilms.

Luckily, the problem of finding the minimum distance is well-known from other areas such as mathematics and digital image analysis. Here it is known as the Euclidean distance. The term "Euclidean" is used to signify that we are capable of moving in any direction when measuring distance from point to point in space. Other distances include the Manhattan- and Chebyshev distance.

The generic method for calculating the Euclidean Distance Map for all voxels in an imagestack is very well described in [4]:

Let a binary 3D image be defined by

$$F = f_{ijk}, \{i \in [1; L], j \in [1; M], k \in [1; N]\}$$

The image F is the altered by

$$g_{ijk} = min_x\{(i - x)^2; f_{xjk} = 0, x \in [1, L]\}$$

That is to say that the minimum distance in the i-direction to a background pixel is found for all voxels f_{ijk}. On the resultant image, G, a similar operation is performed in the j-direction:

$$h_{ijk} = min_y\{g_{iyk} + \alpha(j-y)^2, y \in [1, M]\} \tag{4.2}$$

...to find the minimum distance on all ij-planes. α is the relationship between the length of voxels in the j- and the i-direction. The output 3D matrix, H, has it's voxels treated by

$$s_{ijk} = min_z\{h_{ijz} + \beta(k-z)^2, z \in [1, N]\}$$

so that what is now stored in S is the squared Euclidean Distance Map of F. In order to obtain D, the Euclidean distance map, we must than calculate the square root of all elements in S. β is the relationship between the k- and the i-direction w.r.t. voxels. The proof of the algorithm can be found in [5] and [4].

4.11.2 Implementation in Comstat

In the original Comstat, the diffusion distance was implemented through two functions: diffudist_func and edm_3d_biofilm_func. The first uses the second to obtain the EDM and then prints the distribution of distances, etc. Focus is here put on the calculation of the EDM.

The algorithm used here is - of course - compatible with the definition of the EDM made previously but in logic it is slightly different.

First, the maximum length possible within a given imagestack is found - it is given by the distance from one corner to the opposite - and then a quantity of 1 is added in order to ensure that no equality problems occur. Note that the imagestack is always cuboid.

Then, a 3 dimensional structuring element is defined. The elements in the struct are valued according to their distance to the kernel of it:

```
m = ceil(struc_side/2)
for k = 1:struc_side
  for j = 1:struc_side
    for i = 1:struc_side
      struct[i,j,k] = -sqrt((m-k)^2+(m-j)^2+(m-i)^2))
    end for
  end for
end for
```

In order to allow cuboid voxels, it is possible to multiply the distances being squared with the voxel size in the i, j and k-directions.

Now, the imagestack is copied to a new variable, D. D is padded with ones and all non-zero elements are set to the maximum length. The top layer of D is set to zero.

Hereafter, the structuring element is repeatedly run over all elements in D, adding it to the surroundings, corresponding to the size of the struct (The struct is always centered at the current element) and the minimum value found is stored as current element. The process is repeated until no further changes occur in D at which point the EDM has been found.

The distributions of the EDM-values in D can then be found and printed. This is done in diffudist_func.

4.11.3 Reimplementation in Comstat2

Although the method used in the old version of Comstat is understandable when examined, it does not run very fast. Thus, a search for a better algorithm was performed - and a very good one was found in [5].

The algorithm here is taken directly from [5] and altered in the way that step 3 is explicitly shown and the α and β values in case of cuboid voxels are added.

```
for(k=1:N) {
  //forward scan
  for(j= 1:M) {
    df= L;
    for (i = 1:L) {
      if (f_ijk!=0)
        df =df + 1;
      else
        df = 0;
      end if
      f_ijk = df^2
    }
  }
}
for(k=1:N) {
  //backward scan
```

```
  for(j= 1:M){
  db = L
  for (i=L:1){
    if(f_ijk!=0)
      db=db+l;
    else
      db = 0;
      f_ijk = min(f_ijk,db^2);
    end if
  }
}
//(Step 2)
for(k=l:N){
  for (i=1:L){
    for(j=1:M){
      buff(j) = f_ijk
    }
    for(j=1:M){
      d = buff(j)
      if(d!=0)
        rMax = int(sqrt(d))+1
        rStart = min(rMax, (j-1))
rEnd = min(rMax, (M- j))
for(n =-rStart:rEnd) {
          w = buff(j+n)+alfa*n^2
if(w<d)
  d = w
end if
}
  end if
  f_ijk = d
    }
  }
}
//(Step 3)
for(k=l:N){
  for (i=1:L){
    for(j=1:M){
      buff(j) = f_ijk
    }
    for(k=1:N){
      d = buff(k)
      if(d!=0)
        rMax = int(sqrt(d))+1
```

```
        rStart = min(rMax, (k-1))
rEnd = min(rMax, (N-k))
for(n =-rStart:rEnd) {
          w = buff(k+n)+beta*n^2
if(w<d)
  d = w
end if
}
  end if
  f_ijk = d
    }
  }
}
```

After the algorithm has been run, the distribution of values in the resultant image can be found simply by running through it and registering occurences of values. The values can be put in bins according to the maximum value in the image and the desired amount of intervals.

4.12 Local Thickness

4.12.1 Purpose and definition

The motivation for including the Local Thickness in Comstat2 even though two other thickness functions are implemented, was that none of the included functions were particularly clever - they are both easily tricked by cavities. The local thickness is not. It is most simply defined by:

> *"The diameter of the largest sphere that fits inside the object and contains the point"*

The "object" mentioned could very well be a biofilm in an imagestack and the point a voxel in it.

4.12.2 A basic look at the algorithm

The algorithm runs in three steps:

- Perform an Euclidean distance transform on the input imagestack
- Find distance ridges in the EDM
- Obtain the local thickness using the distance ridges

The first step of the algorithm in the original plugin, the distance transform, has been changed to our implementation because the provided one was somewhat slower - presumably because of multithreading overhead. The second step is kept; it is about removing as many points from the EDM while preserving the so-called distance ridges. Centered on the ridges are all non-redundant points. By redundancy is meant points, whose maximum-sphere can be covered completely by that of another point. Finally, the local thickness can be found as a function of the nearest distance ridge point. For a closer review of the algorithm, see [2].

Chapter 5

Test & Conclusion

5.1 Test of function compliance

One of the main points of the project has been to create a new version that could find the same values as the original. In this chapter, the functions of the old Comstat and those of the new, has been put to work on 10 imagestacks of the oldfashioned info-type. The results of the two are then compared using graphs (generated with Microsoft Excel). Some of the functions cannot be compared directly, because the two versions differ in implementation so alternative tests have been made.

5.1.1 The biomass

As can be seen from fig. 5.1, the function is acceptable - the two versions return the same result for all imagestacks.

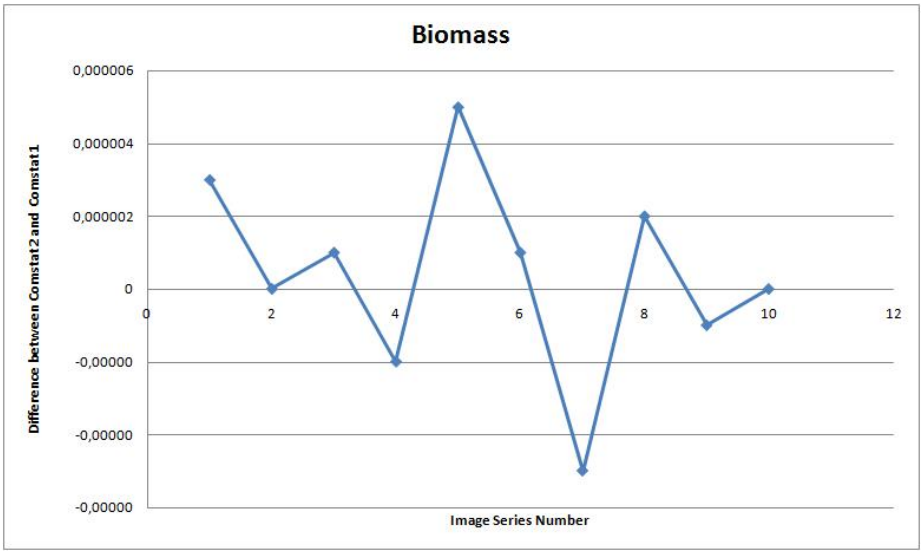


Figure 5.1: *The differences in the biomass are neglectable and related to rounding*

5.1.2 The thickness distribution

Of fig. 5.2 it is evident that the average thicknesses of the two version are the same. The likeliness of the thickness distribution being wrong is thus very little.

5.1.3 The dimensionless roughness coefficient R_a^*

The new version of the function delivers no measureable difference from the predecessor. See fig.5.3.

5.1.4 The area occupied in layers

The area in layers functions agree on the values calculated. See fig. 5.4.

5.1.5 Maximum thickness

Agreement between versions. See fig.5.5.

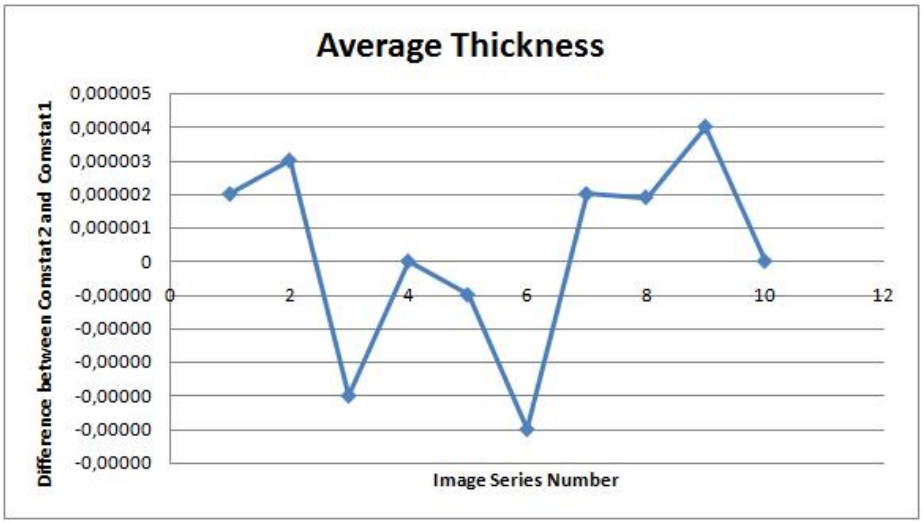


Figure 5.2: *The average thickness is used as comparison. Again, rounding is the source of error*

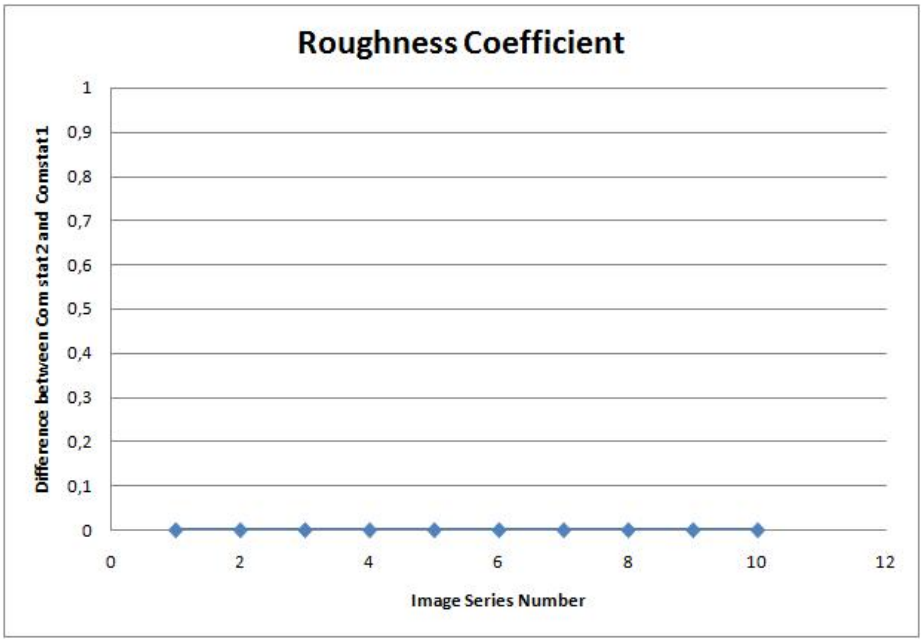


Figure 5.3: R_a^*. *For once, rounding is not a source error. The rounding done on the coefficient has eliminated all noise.*

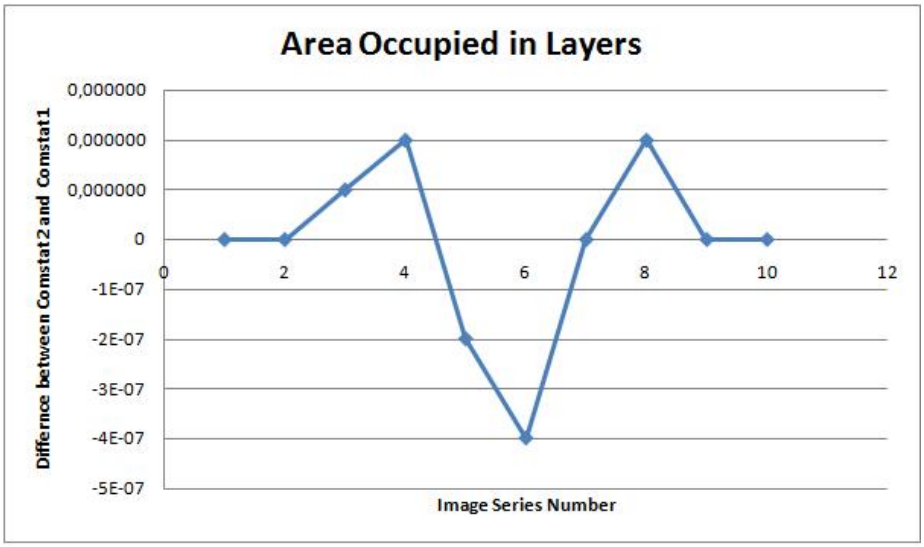


Figure 5.4: *Again, rounding is the errorsource*

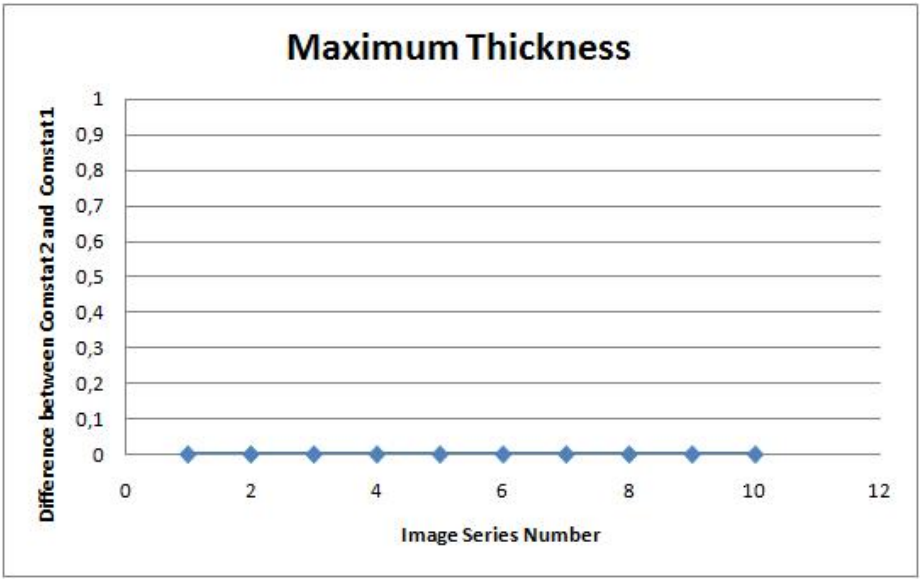


Figure 5.5: *A low precision also makes deviations very low*

5.1.6 Microcolonies at substratum

The number of microcolonies are unanimous through all imagestacks. See fig.5.6

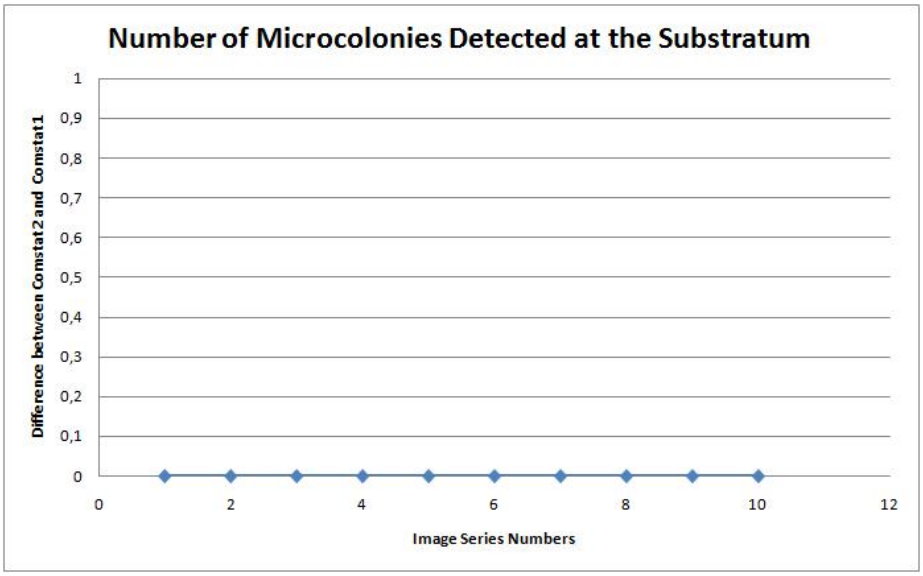


Figure 5.6: *The number of colonies found using both the old and new version of microcolonies at substratum are the same.*

5.1.7 Volume of microcolonies at substratum

In this test, the first obstacle is hit. The volume found in one of the ten tests deviated a little from version to version. However, the difference is not necessarily an error as the deviation is possibly very small compared with the volume measured - and could thus be a roundingerror again.

5.2 Test of additional functions

Some of the functions implemented cannot be tested against old version for one or another reason. Different testmethods have been applied to see if they work.

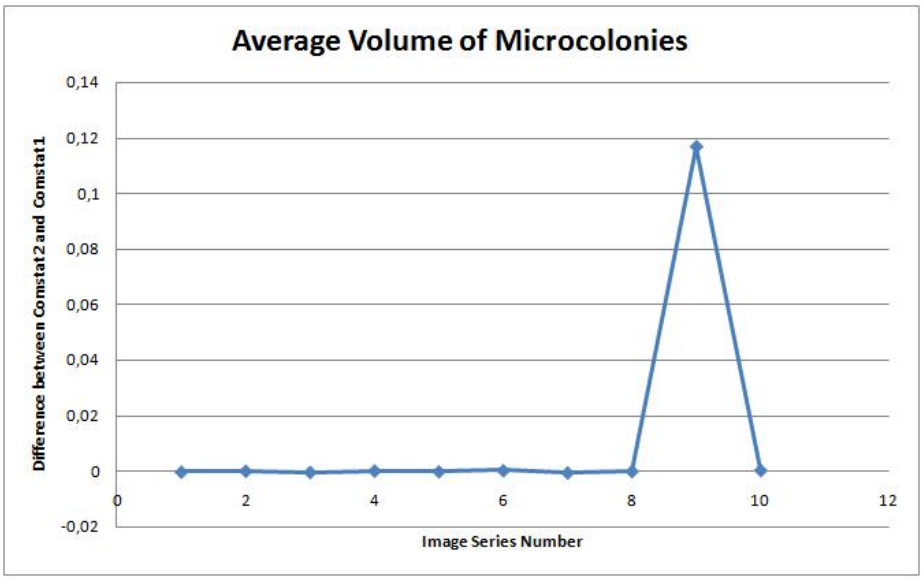


Figure 5.7: *Volume of found microcolonies. Notice the peak at test 9*

5.2.1 The fractal dimension

In our definition, the fractal dimension will be 1 for a perfect circle, so three circles with varying size where drawn in an info-stack (See fig.5.8) and the function was called upon.

The results of running was:

```
Colony FD
1 1.0309
2 1.0302
3 1.0307
4 1.0293
5 1.0305

Average fractal dimension 1.0303
```

The expected result was a value of 1 for all complete circles. Even though the values are slightly higher, the result is satisfactory because the numbers shows that a small difference is present. This is due to the circles being different sizes and discrete - they are not perfect circles and thus are affected by scaling.

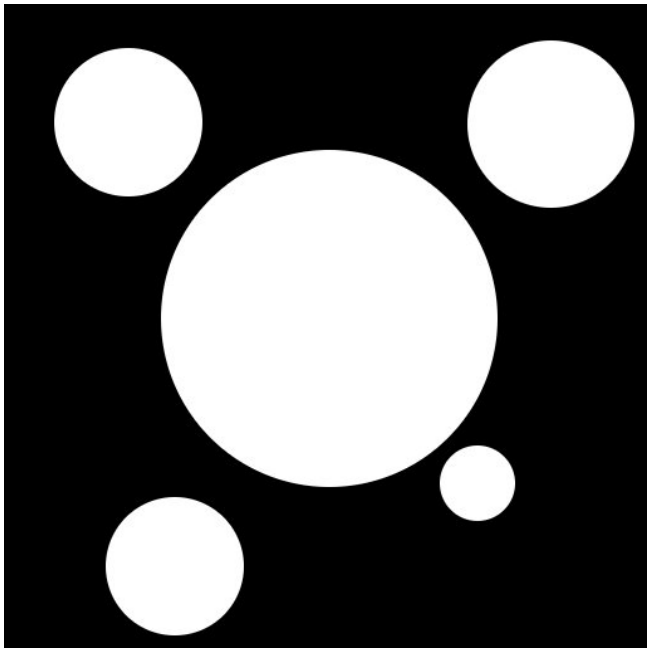


Figure 5.8: *The substratum used to test the fractal dimension*

5.2.2 The diffusion distance

As expected, this function returns a completely different result than the old one. This is due to the very different methods. The new one has - aside from the fact that it is a direct implementation of the algorithm described in [5] - been compared to the multithreaded implementation from the original LocalThickness-plugin. Both algorithms come to the same result. So the output made is correct. To demonstrate the algorithm, it has been run on the image of fig.5.8 to produce the result in fig.5.9

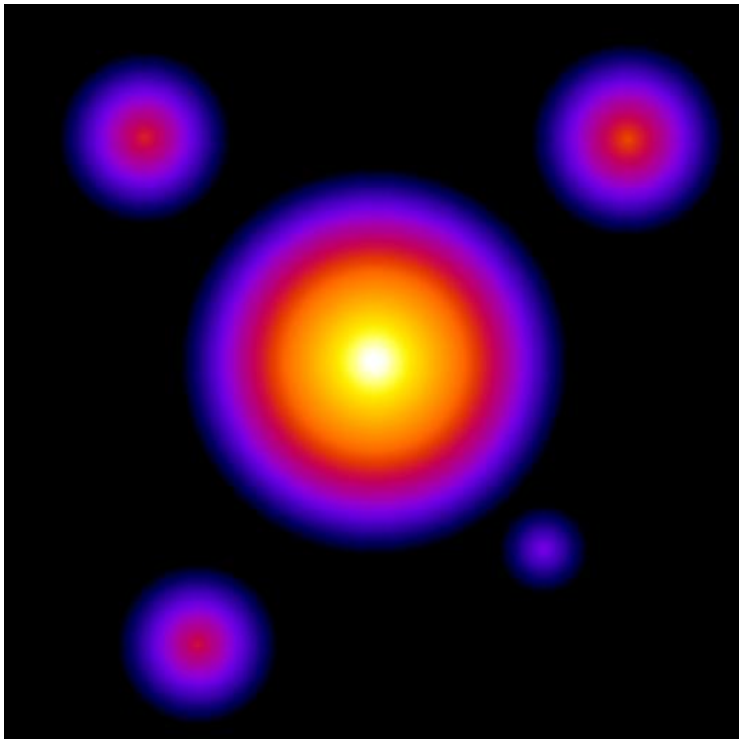


Figure 5.9: *The substratum of the EDM of perfect circles*

5.3 Conclusion

What has been accomplished during this project:

- To implement most functions with a satisfactory result
- Gained a lot of knowledge on using Java for manipulating images
- A reasonably easy-to-extend framework
- A very small, very easy to distribute program
- A very stable program
- A huge speed jump compared to Comstat
- A fairly good GUI

What should have been better:

- One of the original functions was forgotton and has thus not been implemented yet (surface to volume ratio, 10 in Comstat)
- Better version tracking should have been made
- Continuous checking of function compliance. Several working functions decayed in the run of the project and had to be reestablished at the end.
- The interface of the multimode image choosing
- Focus on the use of inheritance (w.r.t. the prevention of code redundancy)
- Test of average runlength missing because of trouble with the algorithm not running properly when very few colonies are present.

Appendix A

Appendix

A.1 Howto...

Here's a short guide to the enclosed CD

A.1.1 ...run Comstat2 off the CD

Go to /ImageJ/ and run Comstat2.bat. For non-Windows users: Go to /ImageJ/ run the command

```
javaw -Xms64m -Xmx768m -Xss32m -jar ij.jar
```

A.1.2 ...use Comstat2

Go to /Comstat2_man/ and read Comstat2.doc.

A.1.3 ...setup the workspace with Eclipse

The workspace used for developing Comstat2 can be found at: /kode/workspace/.
A guide for setting up Eclipse can be found at: http://www.cs.dartmouth.edu/~kimo/blog/computer/im

A.1.4 ...see the code without installing Eclipse

Go to /kode/pure_source/ where all the original java-files can be found. Don't try to compile, though.

A.1.5 ...see the files used for testing

Point Comstat2 (or another image manipulation program) at /test/ where both imagefiles and outputfiles can be found.

A.1.6 ...find the original Comstat for comparison

Look in /comstat-v1.01/ - it can be run from Matlab by calling `comstat` in the directory containing the images to load. Add the comstat folder to the path first.

A.1.7 ...se the javadoc

Go take a look at /javadoc/index.html

Bibliography

[1] Haluk Beyenal, Gary Harkin, and Zbigniew Lewandowski. Quantifying biofilm structure: Facts and fiction. *Biofouling*, 20:1–23, 2004.

[2] Robert P. Dougherty and Karl-Heinz Kunzelmann. Computing local thickness of 3d structures with imagej. http://www.optinav.com/LocalThicknessEd.pdf, 2007.

[3] Arne Heydorn, Alex Toftgaard Nielsen, Morten Hentzer, Claus Sternberg, Michael Givskov, Bjarne Kjær Ersbøll, and Søren Molin. Quantifying biofilm structure using image analysis. *Microbiology*, 146:2395–2407, 2000.

[4] Toyofumi Saito and Jun-Ichiro Toriwaki. Fast algorithms for n-dimensional euclidean distance transformation. *Proceedings of the 8th Scandinavian Conference on Image Analysis*, 2:747–754, 1993.

[5] Toyofumi Saito and Jun-Ichiro Toriwaki. New algorithms for euclidean distance transformation of an n-dimensional digitized picture with applications. *Pattern recognition*, 27:1551–1565, 1994.

[6] Xinmin Yang, Haluk Beyenal, Gary Harkin, and Zbigniew Lewandowski. Evaluation of biofilm image thresholding methods. *Wat. Res*, 35:1149–1158, 2001.