Universidade do Minho
Escola de Engenharia

Vítor Damião Baixinho da Silva

Image analysis of *Saccharomyces cerevisiae* cells: Development of a plugin for ImageJ

Dissertation of Master in Bioinformatics

Work supervised by:
Eugénio Manuel de Faria Campos Ferreira
and
Daniela Patrícia Bernardino Mesquita

October 2014

Acknowledgements/Agradecimentos

Todo este trabalho não teria sido possível de realizar sem as ações multifacetadas dos seguintes intervenientes, aos quais tenho o maior prazer de mencionar e prestar o meu agradecer:

O Professor Doutor Eugénio Manuel de Faria Campos Ferreira que se disponibilizou de imediato em orientar-me neste projeto, motivando-me com total empenho, a ele agradeço a oportunidade concedida em realizar este estudo, bem como todas as sugestões críticas e reuniões.

Muito devo à dedicação e empenho da Doutora Daniela Patrícia Bernardino Mesquita na supervisão do meu projeto, ficando-lhe imensamente agradecido pela sua motivação e incentivo no trabalho desenvolvido, no qual resulta esta dissertação.

Enalteço o Doutor Marek Mösche, pela oportunidade concedida em participar neste projecto. Um especial agradecimento deixo à colega Susanne Polap, por todas as imagens de levedura e ajudas fornecidas para o desenvolvimento do estudo, uma vez que a sua inexistência tornaria muito difícil a realização deste trabalho.

Para finalizar, e não menos importante, gostaria de agradecer a todos os elementos da minha família por todo o apoio fornecido, caso contrário tudo isto não seria tão facilmente conseguido.

Acima de tudo, gostaria de mencionar a Helena Ferreira, uma das principais razões para finalizar este mestrado, ao qual lhe agradeço todo o apoio e paciência para comigo, me incentivando e motivando nos momentos mais difíceis.

Isto tudo foi possível e realizado com o apoio financeiro concebido pela empresa Uniferm – Bakery Ingredients.

Abstract

The yeast *Saccharomyces cerevisiae* is one of the microorganisms with increased use at industrial, academic and scientific level. The easy growth in any culture medium, as well as the complete sequencing of its genome, are two of the main interesting factors, making this microorganism one of the most important worldwide.

The morphological analysis of yeast using optical microscopy is a research area of great interest. Over the last decades, significant advances have been made in digital image processing as well in the analysis of microscopic images of cells. In this context, the development of specific software such as ImageJ is of particular interest since it is free access and open source.

The main goal of this work was to develop a computer program, in the form of an extension module (plugin), in order to add certain features to ImageJ software. For this purpose, programming and processing methods were applied for a more reliable evaluation of cell and finally implemented as an ImageJ plugin. The plugin code was carried out using the Java programming language, since certain required functions were not present in the main program source code. Results presented as .xls file included the identification of cells, as well as counting and cataloguing after images processing and analysis. The features developed in this work allowed the user to process and analyse different microscopic images of *S. cerevisiae* cells.

Finally, the plugin code was tested using multiple images of *S. cerevisiae*. The final version has shown high efficiency in *S. cerevisiae* culture images with different exposure times. Nevertheless, the plugin code was able to detect almost all cells in the images and classify them as large, normal, small and bud.

Resumo

A levedura *Saccharomyces cerevisiae* é um dos microrganismos com maior utilização a nível industrial, académico e científico. A facilidade de crescimento em qualquer meio de cultura, assim como a completa sequenciação do seu genoma, são dois dos principais fatores que tornam este microrganismo dos mais importantes no mundo.

A análise morfológica de leveduras é uma das áreas. Nas últimas décadas, foram feitos avanços significativos no processamento de imagens digitais, bem como na análise de imagens microscópicas de células. Neste contexto, o desenvolvimento de *software* específico, tal como o *ImageJ* é de particular importância, uma vez que é de acesso livre e possui implementação livre e aberta (*open source*).

Este trabalho teve como principal objectivo o desenvolvimento de um programa de computador na forma de módulo de extensão (*plugin*) com a finalidade de adicionar determinadas funcionalidades ao *software ImageJ*. Para permitir ao utilizador do *ImageJ* uma avaliação fidedigna de imagens de células obtidas por microscopia, foram aplicados diversos métodos de processamento. O código do *plugin* foi desenvolvido utilizando a linguagem de programação Java, uma vez que certas funções exigidas não estavam presentes no código fonte do programa. Os resultados apresentados como tipo de arquivo .xls incluíram a identificação de células, bem como a contagem e catalogação após o processamento e análise de imagens. As funcionalidades desenvolvidas durante a execução deste trabalho permitem ao utilizador processar e analisar diferentes imagens de células de *S. cerevisiae* obtidas por microscopia.

O código do *plugin* foi testado utilizando várias imagens de *S. cerevisiae*. A versão final revelou uma elevada eficiência na caracterização de imagens de culturas de *S. cerevisiae* com diferentes tempos de exposição. Foi ainda possível detetar quase todas as células presentes nas imagens e classificá-las como grandes, normais, pequenas e germinadas.

Contents

List of figures

List of tables

1. Introduction

This chapter aims to contextualize the reader to the subject of this work. Also the overall objectives of the dissertation are presented, as well as an outline of how the chapters are organized.

1.1. Motivation

The yeast *Saccharomyces cerevisiae* is one of the most important microorganisms employed in industry. *S. cerevisiae* size and shape distributions are affected by growth rate, mutation, and environmental conditions (composition, temperature, pressure, presence of oxidant agents, *etc*). *S. cerevisiae* cells have a round morphology with a size that may vary between 4 and 10 micrometers in diameter. Also have a division process known as budding. Budding is a form of asexual reproduction in which the original cell divides into “mother” and “daughter” cells. This project intends to classify *S. cerevisiae* morphology using microscopy and image analysis.

Image analysis is the process of extracting useful data directly from images by a computer or other device. Applications of image analysis are found in several areas of science and industry. A specialized image processing program using ImageJ will be developed to count cells and determine size fractions of various groups (small, large, budding).

ImageJ is a public domain Java image processing program [1]. Wayne Rasband, the core author of the program, designed it with an open architecture that provides extensibility via add-on programs written in Java (plugins) or in ImageJ´s programming language (macros). An ImageJ user has the four essential freedoms defined by Richard Stallman in 1986:

- The freedom to run the program, for any purpose.
- The freedom to study how the program works, and change it to make it do what you wish.
- The freedom to redistribute copies so you can help your neighbour.
- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits.

1.2. Objectives

The main goal of this work is to program a Java plugin for ImageJ in such a way that:

- Various groups of cell sizes and budding cells are counted automatically;
- The true cell size shall be estimated in order to group the cells into the right size group;
- Budding cells have to be distinguished from overlapping cells and should not be separated into mother and bud;
- As yeast cells do not appear with a clear black contrast in the microscope images, probably a custom shape recognition algorithm has to be created;
- Cell morphological parameters, as well cell counting and cataloguing should be returned as results;
- Background noise should not cause artefacts;
- Error handling routine: Information to the user, if the image cannot be analysed. What is the reason? E.g.: Wrong magnification, dirty objective, too few/many cells, *etc*.

1.3. Dissertation structure

This dissertation is organized into six chapters. These are oriented to present the collected information during the investigation of the state of the art of the main topics underlying the dissertation and to present all steps taken for the development of an efficient plugin for processing and analysis of *S. Cerevisiae* images.

In Chapter 1, it is intended to guide the reader to the subject of this work, presenting the context and motivations to perform this work. The general objectives of the dissertation are also described.

Chapter 2 constitutes a literature review on the main themes of this work. A short description of *Saccharomyces cerevisiae* microorganism is performed, covering general aspects relating to its industrial applicability history, cell cycle and morphological aspects. Concepts of image processing and analysis are also given. Both sets of image analysis and processing techniques are distinguished with some examples of the most used and known methods. Moreover, the process of acquisition and sensing is also displayed in more detail as well as some work performed on the processing and analysis of *S. cerevisiae* microscopic images.

Next, the software for image processing – ImageJ – is described in Chapter 3 showing some processing techniques included in the program. It is also described in more detail the cell counting process program presented in the program. In addition, the program API classes and some examples of plugins coding are presented.

The methodologies and algorithms implemented in this work are presented in Chapter 4. It is explained the progress of the plugin code mainly the evaluation of the methods included. Also, the program body, referred as the Main Pipeline, is described and justified.

Chapter 5 presents and discusses the evolution of the image processing techniques and the experiments performed to obtain the final code version. Also, results of manual and automatic classification and counting of *S. cerevisiae* cells are compared.

Finally, Chapter 6 contains the most relevant conclusions as well as some perspectives for further work.

2. Context

In this chapter, the theoretical concepts which will serve as basis for the execution of this work are presented.

A brief description of global aspects of the yeast *Saccharomyces cerevisiae* focusing on aspects of their life cycle and morphology is given. First, a history context will be introduced followed by the description of it cell cycle presenting some of the processes occurring in the cell that leads to its division and duplication (replication). Next, some aspects concerning *S. cerevisiae* cells morphology will be presented, focusing on all aspects and shapes that this unicellular organism may adopt during their division phase (budding).Then the concept of image processing and analysis is presented. A brief introduction to digital images acquisition and sensing system will be taken. Subsequently, image processing and analysis techniques will be distinguished and some examples are presented. A short presentation of some practical cases of image analysis of *S. cerevisiae* cells is also presented.

2.1. *Saccharomyces cerevisiae*

Saccharomyces cerevisiae is a well-known, non-pathogenic, unicellular yeast (diameter usually vary between 4 and 10 µm). Its use dates back to ancient times which appear associated with processes like vinification, brewing and baking in ancient Egypt, Sumeria, Babylonia and Georgia. It is of main knowledge that its first appearance dates back to 1680 when van Leeuwenhoek using his primitive microscope saw yeast flocs fermenting wort [2]. However, it is only in the 19th century that it is associated with the alcoholic fermentation. Then, the name *Saccharomyces cerevisiae* is assigned due to its connection with beer production. The genre name, derived from Latinized Greek, means "sugar fungus" and *cerevisiae* means beer in Latin.

Due to their unique advantages like being single celled; can be easily cultured; have capacity to grown readily in any type of bioreactor; having, more lately, its genome completely sequenced and *etc*, *S. cerevisiae* has received increasing attention since the early days of laboratory research. During the last 60 years, this yeast became one of the most studied organisms in the world, where its genetics, molecular biology, and metabolism have been studied extensively, as well its "cousin", the bacteria *Escherichia coli*. Nowadays, this biological system is one of the preferentially election for molecular biotechnologists for high yield and quality of the final product.

One of these tasks is the synthesis of functioning recombinant proteins, by engineered *S. cerevisiae* cells as expression systems. In other cases, the genetically modified yeast itself is the desired product [3]. Furthermore, yeast is probably the most widely used unicellular eukaryotic organism in large-scale production of chemicals, pharmaceuticals, proteins, alcoholic beverages and bread [4].

2.1.1. Cell cycle

One of the most interesting aspects of *S. cerevisiae* **life cycle** is perhaps its ability to alternate between eukaryotic haploid (16 chromosomes) and eukaryotic diploid (32 chromosomes). But first, is also important to refer the two existing different types of haploid cells, denominated by *a* and α mating types. The difference between cell types is found only at the genetic material level, where at certain sites of the genome (MAT locus) can be inserted silent genes (genes of the genome which are not expressed) *a* and α. When located in the MAT locus, the function of these genes is to regulate the production of the major contributor for the mating process, the proteins factor *a* and factor α. These hormones bind to cells of opposite mating and promote changes in their cell surface so that fusion can take place. These different mating types of cells show primitive aspects of sexual differentiation. Furthermore, *S. Cerevisiae* cells can switch between *a* and α type through a reversible mechanism known as *cassette mechanism* [5].

When two different haploid cells merge through the process called matting, new combinations of chromosomes are produced by genetic recombination yielding a diploid cell of a/α type, which are incapable of mating with other cells. Mating of haploid cells yields diploid cells and meiosis of diploid cells produces a structure called *ascus* with four haploid products within, the gametes. Digestion of the *ascus* releases the four haploid meiotic products (ascospores) which, after germination, become haploid yeast cells again [4]–[6].

As well as many species of fungus and most of the microorganisms, in addition to the undifferentiated sexual reproduction, yeasts also reproduce asexually. All cell types (*a*, α and a/α) are capable of proliferating by mitosis. According to Herskowitz [6] given sufficient nutrients, the number of yeast cells doubles at approximately 100 minutes. During the **cell cycle**, the genetic material of the mother cell is duplicated and distributed by each mother cell and daughter. Unlike the classical process of cellular division in which a cell enlarges and divides by cell fission, yeast grows through a process referred to as **budding**. In this case, a daughter cell appears as a protuberance (bud) grows in the mother cell wall. Usually the mother cell is greater than the daughter cell, and this need to increase in size in order to start the process of duplication of chromosomes. At some point, when the bud reaches a certain size, the previously replicated genetic material migrates to the daughter cell. After that, the disconnection of the bud from its mother cell takes place producing a characteristic bud scar on the wall of each cell [7]. Chant and Pringle [8] state that the site where the bud appears is different between haploid and diploid cells. In the first case, the formation of new buds follow an axial pattern, while in the second case both cell poles are potential bud sites. Another study aimed at identifying nonessentials genes involved in bipolar budding [9]. Through the analysis of bud scars on 127 mutants of yeast, three types of phenotypes were identified: unipolar, axial-like, and random.

The life cycle of eukaryotic cells is usually divided into four phases: first gap phase (G1 phase), synthesis phase (S phase), mitosis phase (M) and second gap phase (G2 phase) (Figure 2.1 [10]). If left without nutrients the proliferation is abandoned at G1 phase resuming growth when nutrients become available again [6], [11].

2.1.2. Morphology

Many of the existing studies on *S. cerevisiae* fall into the mechanisms of how cells morphology is regulated and protein localization [9], [11]–[14]. Researchers had focused their interest in know specific morphologic characteristics of cells such as cell size, roundness, bud neck, and position. In the past, these studies were made by manually analysing images from microscopy. Nowadays, the reduction of these activities time is one of the main reasons for developing programs which automatically acquire information from images of microorganism's cells.

Yu *et al.* [15] designed and developed a classification algorithm that is able to distinguish the

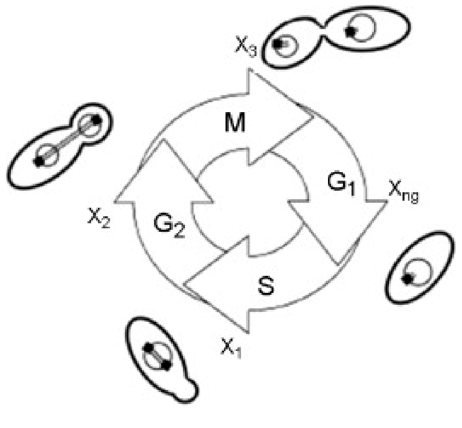


Figure 2.1. Schematic diagram for cell morphological changes used in cell cycle description (S phase: plaque duplication; G2 phase: nuclear migration; M phase: nuclear division; G1 phase: bud detachment) [10].

yeast cells in different phases of the cell cycle through image analysis. After a set of operations, the classifier distinguished between non budding cells, small bud and large bud cells which were associated to cell cycle phases G1, S and G2/M, respectively.

Saito *et al.* [16] presented the *S. cerevisiae Morphological Database* (SCMD), providing data related to yeast cells morphology in different stages of the cell cycle and information about the location of nuclear DNA and actin (http://yeast.gi.k.u-tokyo.ac.jp/datamine/). The authors developed a program for automatic image analysis [17] and analysed thousands of micrographs of yeast mutant cells and results were stored at SCMD. In addition, the database allowed the user to find similar yeast mutants in a particular number of morphological features, through a query search. The cell shape parameters recognized by the program were: long axis length; mother cell and bud roundness (long axis length / short axis length), bud neck position angle, bud growth direction angle, and the fitness value that showed how the cell line fits to the ellipse. The program was also able to quantify more 501 morphological parameters.

The *Phenomics of yeast Mutants* (PhenoM) is another database similar to SCMD [18]. In this database, morphological quantitative data acquired from 78194 morphological images of different yeast mutants are stored. In addition, characteristics obtained from other studies (bud size, cell area, roundness and *etc*), temperature data, cellular components of 775 yeast mutants, as well as other 865 morphological parameters are also presented. The database also provided several data-mining tools including a module for phenotypic comparison between mutant strains and a Gene Ontology module for functional enrichment analysis of gene sets showing similar morphological changes.

The *Yeast Resource Center Public Image Repository* (YRC PIR) is a large database of *S. cerevisiae* and *Schizosaccharomyces pombe* cell images (http://images.yeastrc.org) depicting the subcellular localization and colocalization of proteins [19]. The user can select one or more of 1259833 stored TIFF (Tagged Image File Format) images, by searching the data according to a set of experimental options, such as: data source; number of channels; number of z-sections; number of time-series; objective; binning; image size, and also according to a set of biological options, namely: proteins of interest and selected strain of the available species.

One of the most important aspects of yeast cell morphology is perhaps the size of the bud. Usually, a distinction is made according to bud size groups, including: no bud; small bud; medium bud and large bud. Each one of the categories may be associated to a range of possible mother and bud cell sizes ratio values. Another set of parameters important to mention is the cell shape parameters. Figure 2.2 displays some of these features. It should be noted that all these parameters do not take into account the three-dimensional plane of the cells and that sometimes another possible morphological configuration could be presented. These configurations, such as elongated cells, mother cells with more than one bud, fusing haploid cells during the mating process as well the presence of diploid buds during a specific process, are called complexed cells. Complexed cells may appear in certain conditions of stress. Furthermore, Zalewski and Buchholz [20] categorized the cells of yeast in four distinct groups according to the morphology: (1) proportion of single cells with bud, (2) complexed cells with two or more cells of identical size, (3) single cells with large vacuole, and (4) dead cells, indicating the culture growth phase and the nutrient richness of the medium.

2.2. Image processing and analysis

Image analysis and **processing** is the extraction of quantitative and qualitative information from images by appropriate techniques.

An image can be viewed as a set of distributed data in a lattice or array, with positions defined by a set of elements from the *Cartesian* plane (Figure 2.3 [21]). Thus, the data values could be described as a two-dimensional function, $f(x, y)$, where x and y, coordinates of the *Cartesian* plane, assume integer values from 0 to $N - 1$ and $M - 1$, respectively, in images with size $N \times M$.

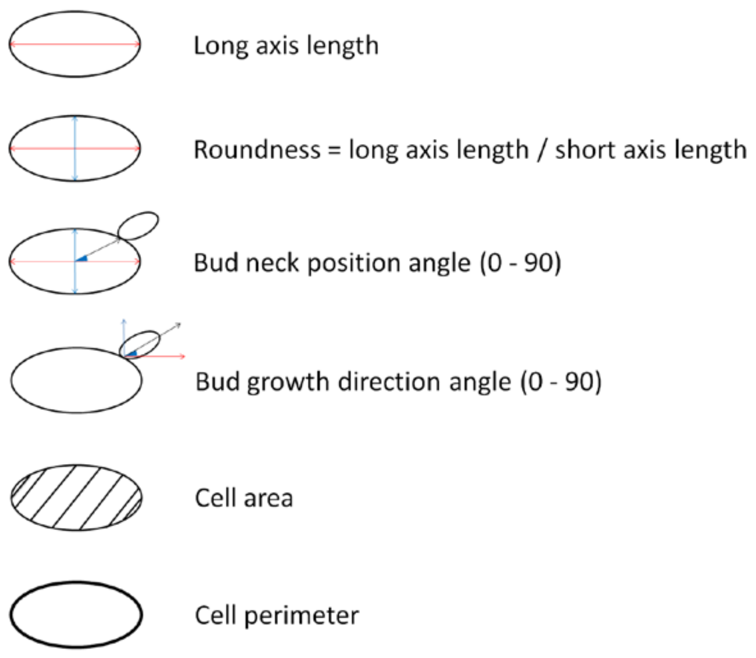

Long axis length
Roundness = long axis length / short axis length
Bud neck position angle (0 - 90)
Bud growth direction angle (0 - 90)
Cell area
Cell perimeter

Figure 2.2. Cell shape measurements of *Saccharomyces cerevisiae* cells (adapted) [16], [18].

Occasionally, images may be squares, making so, *M* equal to *N*. Another frequently used nomenclature is the given spaces generated by the intersection of rows and columns of a matrix. In this case, *i* and *j* represents the row and column numbers, respectively, within the same range of integer values, in a similar way, of the previous example. Each element composing this matrix, grid or lattice if preferred, is named by *picture element*, *image element*, *pixel* or *pel*, being *pixel* the best known denomination [21]. Data values, *f*, is a discrete representation of the intensity or amount of visible light (or any other radiation in the electromagnetic spectrum) reflected (or transmitted) by an object. High intensities are typically translated into a high, or near the maximum values, whereas is usually to translate low intensities into low, or near the minimum values. It should be noted that due to the reflective nature of the objects (the light is not totally reflect or absorbed by any physical object) as well as to image sensing and acquisition processes (described with more detail in the next chapter), the quantization does not distinguish between different colours of the visible spectrum (this is another area of image processing), but the intensity of the monochromatic light. Thus, the quantization of light can be achieved from a range of values, where white is represented by the maximum value and black represented by the minimum value, or *vice versa*. All intermediate values of this scale represent variations of the

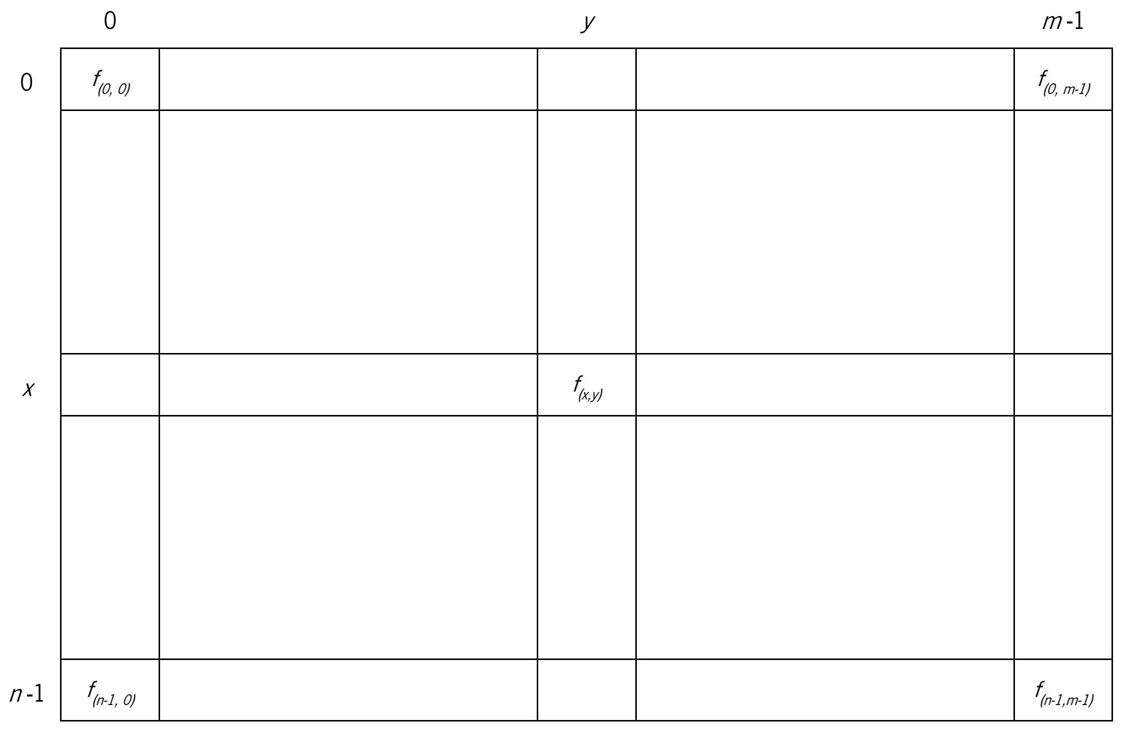


Figure 2.3. Mathematical notation for a digital image (adapted) [21].

monochromatic intensity. The most common scale used in this process is the *grayscale*. The intermediate values of the grayscale represent different intensities of gray – *gray levels*. Due to hardware considerations, the number of levels (L) of the grayscale for quantifying the intensity is limited, taking integer values in the order of powers of 2:

$$L = 2^k \tag{2.1}$$

The number of bits required to store a digital picture is:

$$b = M \times N \times k \tag{2.2}$$

As may be seen from equation 2.2, the number of bits required for storing a digital image is proportional to *k*. Therefore, it is common to refer to images with 2^k gray levels as a "*k-bit image*". For example, the most popular types of images, 8, 16 and 32-bit, are images with 256, 65536 and 4294967296 gray levels, respectively. From this point of view, it becomes obvious the relationship between the size of the grayscale and the memory needed for storage. The image size, *M* × *N*, which best known and most used designation is *spatial resolution*, also influence the number of memory bits required to allocate an image [22].

The collection of significant information from images becomes a little more elucidated after being provided a detailed description of the concepts related to the composition of the image. With informatics background, an image is visualized as an array of integers, and the application of basic algorithms for array manipulation is a common practice. However, image processing and analysis needs several techniques available with the assistance of computer programs. Thus, it is possible to extract qualitative and quantitative information from images using mathematical theorems ranging from the most simple to the most complex. The results of applying these sets of techniques are commonly divided into quantitative and qualitative. For example, the groups of techniques, whose results are images, are classified as **image processing techniques**. Whereas in groups of techniques that extract information from the images by "making sense", similarly to what happens with the human eye shape, are classified as **image analysis techniques**. This definition was adopted by Gonzalez & Woods [22] which divided the sets of image manipulation operations, according to the type of result returned by them.

2.2.1. Image sensing and acquisition

An image recording device responsible for acquiring digital images for the purpose of computational processing is also referred to as image digitizer. In most cases, a camera is commonly used for image acquisition.

The principle of acquiring digital images is applicable to all different types of digitizers. A source of electromagnetic radiation continuously brightens a scene or object and is reflected (or transmitted such as in X-rays) and captured by a sensing material, responsible for transforming the intensity of incoming light into voltage. In the case of digital cameras, the sensor is in reality predominantly a set of sensors arranged in the form of a 2-D array. The output waveform is the response of the sensors and a digital quantity is obtained by digitizing its response. The signal produced by each sensor of the camera leads to the formation of an array of picture elements with values directly proportional to the intensity of light passing through the lens [23]. This involves two processes: sampling and quantization. Digitizing the coordinate values of each one of the sensors is called sampling while digitizing the amplitude values of the voltage waveform is called quantization.

Visible light, as well as any other radiation of the electromagnetic spectrum, vary continuously in terms of wavelength and frequency values, never ending abruptly in a band and starting in another. For example, in this way the sensed data contained in the output waveform could be represented in a graph *f (x, y) versus x* whose function is continuous, except perhaps in the coordinates corresponding to peripheral regions or boundaries of objects. Notice that the value of a coordinate of the *Cartesian* plane is set to 1 to make it possible to do a graphical representation of a row of the image of the scanned object. Equally spaced samples are collected along the coordinate varying then the value of the other variable. Such proceeding is called *sampling*. Sampling limitations are imposed by the number of sensors and by the number of variations of the fixed coordinate [22]. The resultant continuous value of the sampling process is associated to a discrete level of the color scale used. For example, in 8-bit images the value of light intensity is equally divided into 32 to 32 between the ranges from 0 to 255, corresponding to each one of these ranges of values, different intensities of gray of the grayscale. The process of discretization of the values of brightness is called *quantization*.

Figure 2.5 shows an example of the pixel values distribution in the first line of a picture of a cell colony. The original image is shown in Figure 2.4. In this case the values are already discretized (after quantization) within a range of values between 0 and 255 (8-bit). However, the curve of the continuous function is easily perceptible. It is noted that the peak values of the function corresponding to white dots in the image represent cells. The variability in the intensity of the peaks demonstrates the perceptibility of the cells in the image. The higher the peaks are the

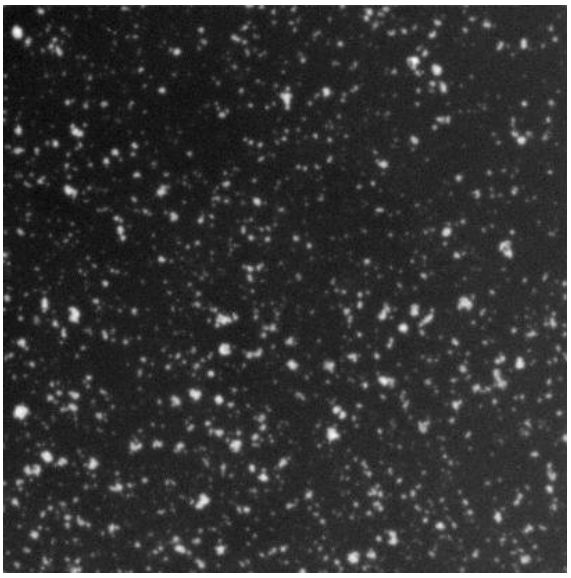


Figure 2.4. Cell colony image (available at http://imagej.nih.gov/ij/images).

more noticeable are the cells in the picture due to the contrast with the background, being this perceptibility reduced as the peaks size decreases.

2.2.2. Image processing techniques

The main objective of processing techniques resides essentially in obtaining an image by modifying or transforming an original image by the action of specific operators. These operators (T) are applied directly on the values of certain pixels, $f(x, y)$, resulting in a new pixel value with

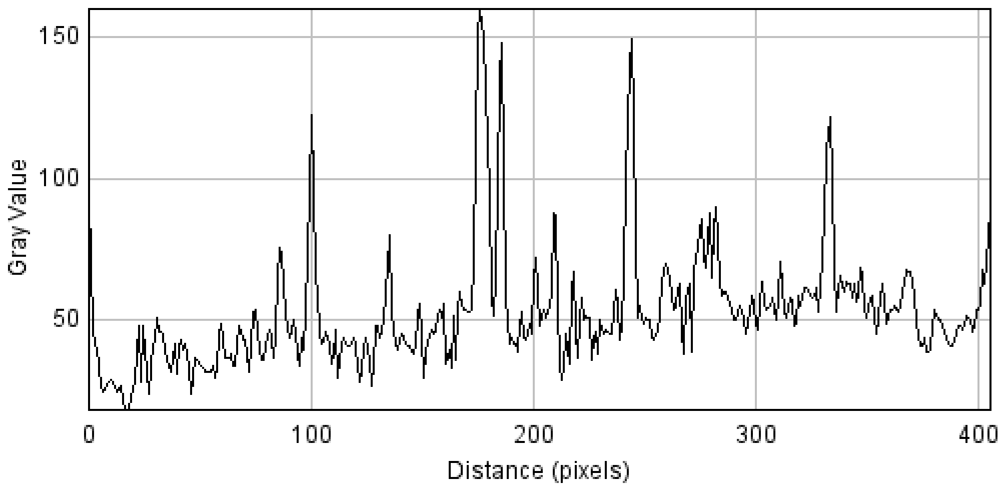


Figure 2.5. Plot of the first line ($0 \leq x \leq 406$ and $y = 0$) of a cell colony picture. The values, discretized to have a range corresponding to 8-bit images, demonstrate a good approximation of what would be the function curve for the values with continuous amplitude.

position coordinates *x* and *y*:

$$g(x, y) = T[f(x, y)] \qquad (2.3),$$

or also on the values of pixels of entire regions including whether or not the value of the target picture element, *f*. The shape and size of these areas may be diverse, ranging from the most direct, vertical, horizontal or diagonal neighbors, to a set of neighboring elements with radius of a specified number of times of the central member, and may assume the most varied formats, including square, round, diamond, elongated, *etc*.

For example, a processing operation that aims to enhance images with high amounts of dark pixels whose areas of interest are the clearest ones is creating the negative image. The negative transformation of an image is easily translated by the following expression:

$$g(x, y) = L - 1 - f(x, y) \qquad (2.4),$$

whereas stated earlier, L is the level of grayscale image. Reversing the intensity levels of an image is the same as the equivalent of producing a photographic negative. In Figure 2.6 is shown how this operation applied to all the picture elements of the cell colony causes an increase in the ease of cells viewing.

This type of operation is included in the group of operations that aims the image enhancement. There are other operations whose quality results vary according with the definition given by the user. Several methodologies such as *power-law* and *log transformations*, *histograms processing*,

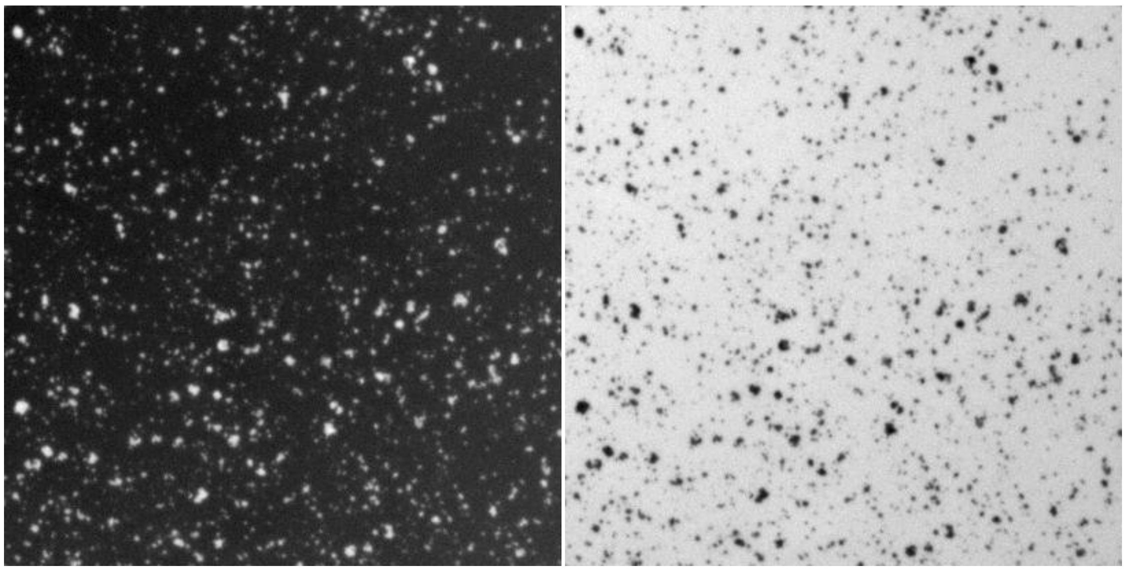


Figure 2.6. Image of a cell colony (left picture) and their negative transformation (right picture) (available at http://imagej.nih.gov/ij/images). The transformation was possible using the *Image Inverter* plugin from software ImageJ.

filters, and also the combination of different enhancement methods can be found. Subsequently, image improvement thorough pre-processing techniques, it is common practice before separating objects from the background making image *segmentation*.

Image enhancement techniques arise frequently associated with tasks such as automated cell counting, cell size and morphology characterization, determination of cell content, determination and quantification of protein localization and many others [24]–[30].

Filters improve images through the application of changes on groups of pixels. These are efficient computational methods that have the aim of reducing the level of "noise" and to emphasize edges in images. Filters provide an improvement to visual interpretation of images and can be a precursor to subsequent digital processing techniques such as segmentation [21]. The filtering process consists essentially to move, pixel by pixel, the centre of a "mask" of variable size (usually 3 × 3 pixel region) changing the value of the centre pixel according to a relationship between the values of the remaining pixels covered by the mask. Within this type of process there are different filters. Smoothing filters are primarily used for noise reduction and blurring while sharpening filters are used with the goal of highlight fine detail in an image to enhance detail that was been blurred. In general, filtering of an image of size $M \times N$ with a filter mask of size $m \times n$ is given by the expression:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t) \qquad (2.5),$$

where $a = (m - 1) / 2$, $b = (n - 1) / 2$, $w(s, t)$ is a filter coefficient for the corresponding mask pixels coordinates s, t. In order to achieve a completely filtered image it is necessary to apply eq. 2.5 to each one of the pixels of the image. Two considerations must be taken when the centre of the mask reaches a pixel in the image border. One of them, called padding, is to add replicated rows and columns, or with pixel values of 0, while the other consists in prevents the filter to advance to outside the image.

Segmentation or **thresholding** is the separation of pixels in a given range of values from other pixels with a specific range of values, different from the previous one. In practice, segmentation is the separation of regions of pixels corresponding to objects or part of objects of interest from the rest of the image. Each pixel of the image is allocated to one of these regions or categories. Thresholding can be a very effective method of measuring disjunctive and complex characteristics of an image. However, it could make wrong assignments of pixels to distinguish and separate an area of interest. Thus, segmentation is typically considered a critical step in image analysis.

Segmentation is a process usually preceded of the image binarization. Pixels values below a certain value or range of values are set equal to zero, while the pixels values above assume the maximum grayscale value. Thus, the image becomes represented by white and black pixels. Another type of thresholding result is highlighted as selected area which falls into a determined range of pixel value. In this case, two limit values are selected.

Figure 2.7 shows the results of the cell colony image segmentation as well as the results of filtering the original image followed by its segmentation. A median filter with radius of 2 pixels (Fig. 2.7c) was applied to the original image, where a slight reduction of background noise as well as some general image blurring is observed. This can be visualized by the image

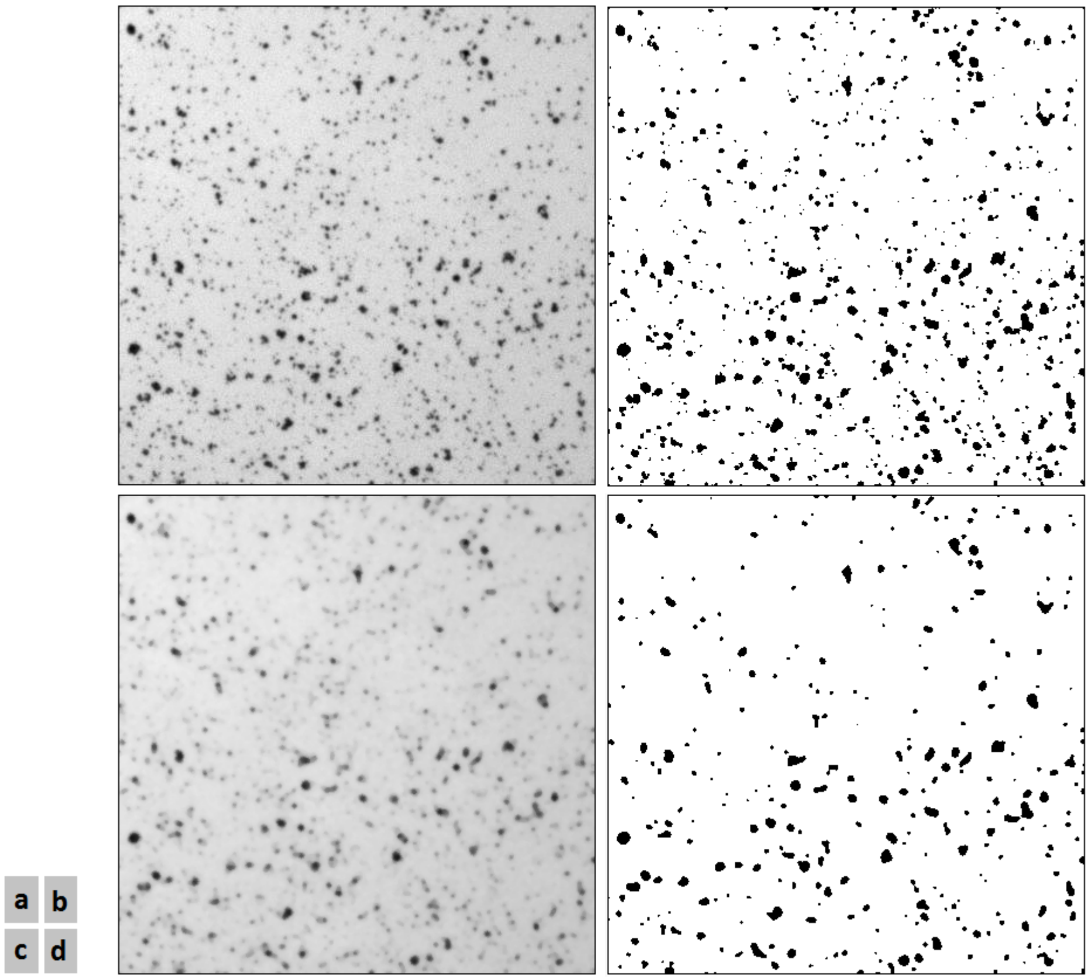


Figure 2.7. Binarization of the cell colony image without filtering and after filtering. Figure 2.7a is the original cell colony image. Figure 2.7b is the result of thresholding the image of Figure 2.7a at pixel value 171. The image of Figure 2.7c is the result of applying a 3x3 median filter to the image of Figure 2.7a. The image of Figure 2.7d is the result of thresholding the image of Figure 2.7c at pixel value 171. All filter and threshold operations were performed with the support of the software ImageJ (available at [1]).

segmentation after the thresholding at pixel value of 171. More visible black spots can be observed in Figure 2.7b when compared to Figure 2.7d. This shows that the filter applied had an effect of image blur resulting in a cleansing of the image noise and also of some cells. This is also corroborated by the analysis of the histograms of the pixels distribution.

In this manner, it is shown that the filter caused a decrease of the maximum value assigned to a pixel, and a decrease in the average of pixel values.

2.2.3. Image analysis techniques

Image analysis is a field of digital image manipulation that differs from image processing. The purpose of image analysis is to emulate human vision, including learning and be able to make decisions based on vision inputs [22]. This technique is also associated with a high level of digital imaging computerized processes. The level of processing involves the "making sense" of a recognized set of objects where cognitive functions normally associated with vision are applied. While the human vision system collects information in a qualitative way, image analysis extracts quantitative information from the data sets assembled in images.

It can be stated that this group of techniques have, or could have, images as inputs, but whose outputs are attributes of images and not images as what happens with the application of image processing techniques. Usually, image analysis techniques are applied to images resulting from processing techniques. The most common image analysis techniques are morphological analysis, measurements, recognition, representation and description. Segmentation could also be included in this group of techniques. As stated earlier, segmentation results in image partition into its objects or constituents. The recognition based on objects segmentation, like assigning a label (e.g. cells) to objects, could be included in the group of image analysis techniques.

Morphological analysis relies commonly on the geometric aspects of an object. The parameters related to objects morphology are diameter, area, number, perimeter length, width, eccentricity, roundness, extension, convexity, solidity and compactness [31], fitting perfectly in the area of microscopic images analysis of yeast cells, or any other microorganism.

Besides the most basic mathematical morphological operations, such as *translation*, *reflection*, *complement* and *difference*, three types of operations characterize the morphological analysis technique: operations for size and shape, operations for connectivity and operations for texture [21]. The morphological analysis described in this chapter is applicable to binary images, usually derived from thresholding.

Operations for size and shape are the most direct and easiest morphological operations to use and are related with the properties, local shapes and sizes of objects in images. The components of the images are affected by how the structural elements of this technique, usually disks or squares are accommodated. Afterwards, several morphological operators are briefly described: *dilation*, *erosion*, *opening*, *closing*, *hit-or-miss transform*, *boundary extraction*, *region filling* and *connected components*.

Dilation is the operation in which object boundaries are expanded according to a structural element. The central pixel of the structural element, generally smaller in number of pixels, cycles through all pixels of the target object resulting in a wider object. This increase in size is related to the number of pixels of the structural element which exceeds the limits of the target object when the center pixel reaches the edge of the target object.

Erosion is the opposite of dilation. This technique causes a contraction of the boundaries of an object in a similar manner to the dilation process. In this case, the object is "reduced" according to the shape and size of the structural element. Once the limit of this element reaches the boundary of the object, the pixels between the object boundary and the central pixel of the structural element “drop out” from the constitution of the object. Once the center of the structural element to pass through all the pixels of the target object that would result in a contraction limits.

Opening and **closing** are intrinsic to dilation and erosion techniques. The result of opening is the same as the result of erosion followed by dilation of an object, causing smoothed contours, broken narrows isthmuses and eliminated small islands and sharp peaks. In an analogous way, the result of closing is the same result of dilation followed by erosion of an object, causing smoothed contours, breaks and fused narrowed longed thin gulfs and eliminated small holes.

Hit-or-miss transform is an iterative morphological shape detection tool. Actually, this operation involves a number of basic operations such as erosion, complement, intersection and difference. After a number of iterations where each of these techniques is implemented according to a specific mathematical equation, the final result includes the coordinates of the object of interest in the image.

Boundary extraction is a technique which returns a region of pixels corresponding to the boundary of an object of interest. The quality of the resulting boundary is related to the shape of the structural element since the result of this operation is equal to the difference between the original object and the same object after erosion.

Region filling is the filling of a boundary defined region, given a point in the same region.

The **connected components** allow determining the number of connected pixels of a particular region. It is important to notice that the type of connection is based on a square structural element, e.g., they are considered connected to a certain pixel all the eight pixels that are around it (8-connectivity).

The techniques that constitute the group of morphological analysis operations for connectivity of objects are *thinning*, *thickening*, *skeletons* and *pruning*. This set of operations considers the connectivity between objects unlike the operations described above which only interact with the "local" size and not the global size of objects. For example, it could be important to know not only if an object is only connected but also with its topology, particularly with properties such as how many branches and hole it contains [21]. **Thinning**. is an iterative process based in the hit-or-miss transform technique, but which characteristic structural element rotates 90 degrees, after each iteration. The final result is a thinner object than the original but whose connectivity is preserved. **Thickening** is the morphological dual of thinning. It is similar to the thinning process applied to the object background. Actually, the process begins with a pixel inverting and the final result is achieved by another inversion.

2.2.4. Image analysis of *S. cerevisiae* cells

Segmentation and classification procedures have been gaining large importance in the characterization of *S. cerevisiae* cells. The characterization of yeasts has been considered quit relevant for many chemical processes.

The determination of yeast population size distribution and kinetics is a relevant aspect on the characterization of yeast cultures at live operation. A fully automation of this process can avoid for instance several of sampling for image scanning or manual recognition of the ellipsoid shape of the cells, as well as minor and major half axes [30]. Zalewski and Buchholz [20] developed a fully-automated system for sampling and analysis of *S. cerevisiae* in order to control their growth. After processing the samples through a purpose-built system, a scanned image was submitted to a process of image analysis. This program was composed of a sequence of steps, which in addition to the classical cell count, estimated the physiological conditions of the yeast culture. The principal steps of the program after sampling were color filtering, contour treatment, object recognition, classification and analysis.

Subsequently, to improve the segmentation process an image analysis for yeast cells was developed was developed [26]. It was found that the performance of bioreactors and other chemical processes were greatly influenced by the morphological character of the cells. Therefore, the authors developed a method capable of segmenting yeast cells based on a *Watersheed Transformation* and in a tree representation called the *Tree of Critical Lakes* preserving cell contour. Yet Chen *et al.* [32] used a machine learning approach to automatically find cell contours in the graphical model representation. The authors developed an automatic image analysis of yeasts for the purpose of recognizing protein localization. The system was trained and tested for features selection by a support vector machine classifier. The automated method provided an objective, quantitative and repeatable assignment of protein locations applied to new collections of yeast images.

Taking into account that yeast cell morphology has an impact on fermentations productivity, image analysis techniques have been developed for morphological characterization [30]–[35]. Various groups presented computational tools for automatic analysis of microscopic fluorescent images of budding yeast. Miroshita *et al.* developed an image analysis program for cell morphology characterization of gene-deletion strains [13]. However, the algorithm required the cell membrane to be stained with a fluorescent dye. The open-source software Cell-ID [14], on the other hand, used bright field images for cell recognition. The actual image analysis methods used have not yet been published.

A good cell contours identification can lead to an effective morphological parameters quantification as well as the proteins subcellular localization from fluorescent date [36]. Logg *et al.* developed a fast method for cell contours recognition. This method exhibited great performance for pictures of yeast cells not too clustered.

Many advances have been made in the field of image analysis of cells for morphological characterization. Yu *et al.* [15] developed an automatic processing algorithm for microscopic images classification of yeast cells in the microfluidic channel. The image quality was improved and the background noise removed through an enhancement process. Then, a segmentation algorithm converted the images into a binary matrix containing the shape of the cells. The morphological characteristics – compactness and cell axis ratio – capable of classifying between non budding, small and large buds were extracted by methods of extracting features. Finally, a self-learning classifier distinguished the four types of cells morphologies. Three machine learning methods for classification were tested being them, *k'th nearest neighbors* (kNN), *linear-kernel support vector machine* (SVM) *and Mahalanobis distance* classification.

More recently, Doncic *et al*. [28] presented an algorithm with a fully automated segmentation and tracking of budding yeast cells within growing colonies. The algorithm was developed carrying out considerable amounts of thresholds, leading the outcome of the entire set of thresholds to be used to perform a final robust segmentation. Each threshold divided the image area into two types: putative "cell" and "no-cell" areas. Then, with the aid of one watershed algorithm, *Ferdinand Meyer watershed* algorithm via the MATLAB function [37], the separation of the two putative areas in various regions was performed. In practice, this was achieved by computing the distance transform of the complement of binary image defined above and assigning minus infinity to each pixel that does not belong to any object. Thus, a "topographic map" was created for the object to which the watershed algorithm could be applied. To determine whether a region is part of the cell or not, score each region as 1 (cell) if it meets certain criteria relating to their area. Otherwise, score the region as 0 (no-cell). This process was applied to all sub-regions. Finally, a composite image was created whose pixel values depends on the number of times the cell was found (value 1) after watershed segmentation.

3. ImageJ

Until a short time ago, image processing and analysis were only possible through expensive commercial tools, or through software packages developed by those who needed it. Moreover, the acquisition and save of digital pictures to an image processing system (e.g. personal computers) has never been as commonplace as it is today. The ubiquity of digital technology has made it digital images part of numerous areas from astronomy to microscopy passing by medicine. In fact, with the increasing flow of acquisition and circulation of digital images, new powerful software packages has been raised, enabling the regular user to manipulate and process digital images as a professional.

In this context, finer distinctions of certain concepts related to digital images processing performance appeared, among them *image processing* and *image editing* [38]. **Image editing** is the set of processes for manipulation of images by regular users with the help of programs such as *Photoshop* or *Paint*. **Image processing** is the conception, design, development, and improvement of digital images performed by specific groups of people such as scientists or engineers. As previously mentioned, **image analysis** comprises all the techniques with the purpose of extracting meaningful information about picture contents.

ImageJ is a software package containing many of the techniques listed in the above mentioned concepts. It is able to compute areas and statistics of pixel values, measure distances or even create histograms and line profile plots. Moreover, it includes the most basic image processing techniques such as: background subtraction, brightness & contrast adjustment, image type conversion, smoothing, sharpening, filtering, and binarization. It also features other basic manipulations such as geometric transformations (scaling, zooming, rotation, *etc*.). It can display, edit, analyze, process, save and print 8–bit, 16–bit and 32–bit images. It can read many image formats including TIFF, GIF, JPEG, BMP, DICOM, FITS and 'raw'. It supports 'stacks' (and hyperstacks), a series of images that share a single window. It is multithreaded, so time-consuming operations such as image file reading can be performed in parallel with other operations [39].

This program is a public domain (meaning that its source code is openly available and its use is license-free) Java image processing and analysis program inspired by NIH Image (developed at the U.S. National Institutes of Health and available on the Internet at http://rsb.info.nih.gov/nih-image/) for the Macintosh. It runs, either as an online applet or as a downloadable application, on any computer with a Java 1.5 or later virtual machine. Downloadable distributions are available for Windows, Mac OSX and Linux at **http://imagej.nih.gov/ij/index.html**. Also, ImageJ was designed with an open architecture that provides extensibility via Java plugins. Custom acquisition, analysis and processing plugins can be developed using ImageJ's built in editor and Java compiler. User-written plugins make it possible to solve almost any image processing or analysis problem.

The following chapters are intended to make a simple presentation of the ImageJ software. Their menu commands will be presented as well its features will be briefly explained. Its principal commands, such as *Particle Anlayzer* or *Subctract Background* will be discussed with a little more detail. The main classes and its methods of ImageJ´s API will be presented, taking into account the usefulness they had in the preparation of this work, continuing to focus the form of how they are applied in Java. Finally, a brief introduction to coding plugins in Java for ImageJ will be performed. For more information about menu commands or building plugins it is possible to consult the documentation available at **http://imagej.nih.gov/ij/developer/**.

3.1. Image processing with ImageJ

Similar to the most of the existing programs, ImageJ provides to users a graphical interface containing a bar with commands menus. Actually, the program main window is not much more than a bar with the **Menu Commands** plus a **Toolbar**, **Status Bar** and **Progress Bar** (Figure 3.1). Additional windows are opened to show images, histograms, plots, results, *etc*. The toolbar (squares with figures in the main window) contains the basic tools to make the following selections:

1. Rectangular and Rounded Rectangular;
2. Oval, Elliptical and Brush;
3. Polygon;
4. Freehand;
5. Straight, Segmented, Freehand Arrow Line;
6. Angle;
7. Point and Multi-point.

Besides these tools, it also has seven free slots to select any of the +60 tools and +15 toolsets available on the website ImageJ. The menu bar is constituted by eight menus. The **File** menu

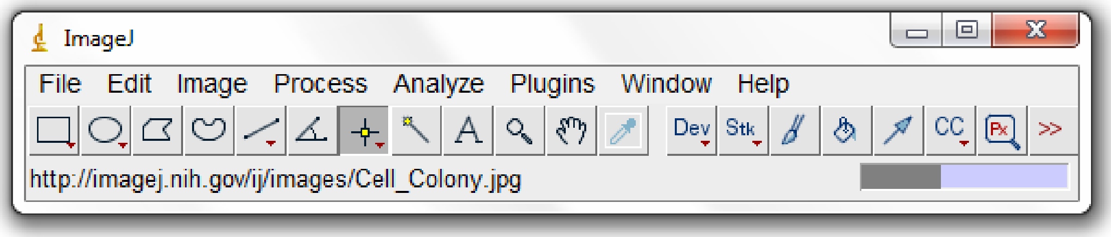

ImageJ
File Edit Image Process Analyze Plugins Window Help
Dev Stk CC >>
http://imagej.nih.gov/ij/images/Cell_Colony.jpg

Figure 3.1. The ImageJ main window (version 1.48k).

contains the basic operations (opening, saving, creating new images) where most of them are self-explanatory. Then, the **Edit** menu contains operations for edition and drawing as well as global configurations. The following menu, **Image**, has operations for image modification and conversion including geometric transformations. The **Process** menu includes all kinds of image processing operations, while the **Analyze** menu has statistical measurements, histogram and profile plotting, among other operations related to image analysis. The **Plugins** menu contains commands for creating, editing and managing add-ons, as well the lists of all installed macros, scripts and plugins. Finally, **Window** and **Help** menus feature the selection and management of windows, and the documentation and version information, respectively.

With this menu bar short description, it is possible to conclude that the most important and/or most used menus for image analysis and processing are Image, Process and Analyze menus (*Analyze Particles* mainly). For this reason, next is presented a more detailed description of the main content present in each of them.

3.1.1. Image menu

The **Image** menu is divided into several sub-menus, some of which are: **Type**, **Adjust**, **Color**, **Stacks**, **Hyperstacks**, **Crop**, **Duplicate**, **Rename**, **Scale**, **Transform**, **Zoom**, **Overlay** and **Lookup Tables**.

The sub menu **Type** determines the type of the active image or converts to another type. Image types supported by ImageJ are 8-bit, 16-bit, 32-bit, 8-bit color, RGB color, RGB stack and HSB stack. Any tentative to run an unsupported conversion causes a dialog box to be displayed that lists possible conversions.

The **Adjust** submenu contains commands that adjust brightness/contrast, adjust threshold levels, window leveling, color balance, image size and canvas size. The **Brightness/Contrast...** command allows the user to change the brightness and contrast of the active image. When the command button is selected, a window opens where it's possible to view the distribution histogram of the active image pixel values. A diagonal line crosses the histogram indicating how the pixels values are mapped to 8-bit (0-255) display values. The two values below the graph indicate the minimum and maximum values of the displayed pixels. Below the histogram, four sliders allow the user to interactively change the minimum and maximum value of pixels as well as the brightness and contrast of the active image. **Threshold** allows the user to automatically or interactively set an upper and/or a lower limit to target areas of interest or background of grayscales images. This tool presents a graphical representation of upper and lower sliders enabling to manual adjust the thresholding window across the range of values. There is given to users the possibility to choose between displays the threshold values in red, black (with a white background) and green (pixels below threshold). Also, there are 16 different methods of automatic threshold. The **Auto** button automatically sets the threshold levels based on an analysis of the histogram of the current image or selection. **Apply** sets thresholded pixels to black and all other pixels to white. However if **Process > Binary > Options > Black Background** is checked, the thresholded pixels are set to white and all other pixels to black. **Reset** disables thresholding and updates the **histogram.**

The **Color** submenu contains several commands that allow the user to manipulate color images. The ImageJ is able to handle three types of color images: *Pseudocolor* images, *RGB* images and *composite images*. **Pseudocolor** images are images with a single gray channel (8, 16 or 32-bit) which is associated with a color via a lookup table (LUT). A LUT may be viewed as a table of gray values with matching values of red, green and blue in a way that the gray values are presented as colorized pixels. **RGB** images are color images that have 256 values in the Red, Green and Blue channels. Hence, these images are 24-bit (2^{2x8}). Like the RGB images, **composite images** are formed by channels of pixel values. However, each channel is separated from the other and can be switched on and off, allowing the visualization of the channels separately. Each channel can be 8, 16 or 32-bit and run on any LUT. Furthermore, more than three channels can be kept together or separately. Returning to sub-menu Color, the command **Convert Stack to RGB** converts up to 2 or 3 slice of a stack in an RGB image, assuming that each of slices has 8 or 16-bit grayscales and sorted into R, G and B. **Convert to Composite** converts an RGB image or a stack with 2-7 slices into a composite color stack. The separation of R, G and B components of an RGB image into three 8-bit grayscale images is possible via the **RGB Split** command. The opposite is possible through the **RGB Merge** command, where 1, 2 or 3 8-bit grayscales images are merged in an RGB image.

The **Stacks** submenu contains a set of commands that allows various manipulations of image stacks. Image stacks are images displayed in a single window, being desirable that these images are spatially or temporally related, wherein that all images must have the same size and bit depth. The constituent images of a stack are called slices. Many of the existing commands in ImageJ have the option of processing all the slices in a stack. Stacks submenu already has the basic commands for manipulating slices in stacks, such as **Add Slice**, **Delete Slice**, **Next Slice**, **Previous Slice** and **Set Slice...** all of them with self-explanatory names. The commands **Images to Stack** and **Stack to Images** create a stack of all images that are displayed in separate windows and *vice versa*. All other commands of this submenu are related to hyperstacks, virtualstacks and 3D animations.

Crop and **Duplicate** functions allows cutting or duplicating, respectively, of an image or stack based on the current rectangular selection. **Rename** allows the user to change the title of an active image or stack. The **Scale** submenu opens a dialog box allowing the user to resize or to scale at its discretion an image or a selected area. The **Transform** submenu contains commands that enable the user to apply geometric transformations on an image or a stack of images. It is possible to flip horizontally and vertically, rotate a specific number of degrees in any direction, translate in a specified number of pixels the image in the x and y directions, and to reduce the size of an image. It is also possible to control how the current image is displayed through the **Zoom** submenu. Finally, **The Lookup Tables** submenu contains a selection of color lookup tables that can be applied to grayscale images to produce Pseudocolor Images.

3.1.2. Process menu

The **Process** menu has the following image processing techniques: **Smooth** and **Sharpen**, **Find Edges** and **Maxima**, **Enhance Contrast**, **Noise**, **Shadows**, **Binary**, **Filters**, **Bach**, ***Image Calculator*** and ***Subtract Background***.

Smooth is actually a filtering function via an average filter with a 3×3 mask, which blurs an image or a selected area. Filtering is a concept explained with more detail in section 2.2.3. **Sharpen** is a command to a processing technique that accentuates and enhances detail of an image, particularly the edges of objects. However, the noise enhancement is considered a drawback.

The filter **Find Edges** uses a Sobel edge detector to highlight sharp changes in intensity in the active image or selection.

On the other hand, the **Noise** submenu contains commands to add or remove noise from images. The **Shadows** submenu has a group of commands which produce a shadow effect in image objects, with a light appearing from a corresponding direction to the command name.

The **Make Binary** command converts one image in black and white in a similar way to that described in section 2.2.3. Furthermore, **Binary** submenu is constituted by commands for morphological analyses of binary images, such as those described in section 2.2.4. Watershed segmentation is a relevant operation present in this submenu. This operation is an automatic mode to separate or cut objects or particles that touch each other and which are not supposed to be together.

The **Filters** submenu contains several filters differing from each other essentially by the method applied: *Convolution*, *Gaussian*, *Median*, *Mean*, and *Unsharp*. **Unsharp Mask** command sharpens and enhances image objects edges by subtracting a blurred version of the image (the unsharp mask) from the original. The Unsharp mask is created by Gaussian blurring the original image. The Gaussian blur filter radius is a parameter which must be defined. Increasing the Gaussian blur radius sigma will increase the contrast. The Unsharp mask is multiplied by the Mask Weight parameter, where higher values mean additional edge enhancement. The Mask Weight determines the strength of filtering.

Image Calculator performs arithmetic operations like addition, subtraction, multiply, divide, *etc*, between two images.

Subtract Background is a process useful for images with high background, but even images with lower background may benefit from the subtraction of background. The process is based on the "rolling ball" concept in which a ball with a given radius roll over the bottom surface of a three-dimensional grayscale 2D image. Figure 3.2c and Figure 3.2d are cross segments of the yellow line in Figure 3.2a (available at http://imagej.nih.gov/ij/images) and Figure 3.2b, respectively. The graphs are profiles of pixel values ranging from 0 to 406 of the line number 200 of each one of the figures. The role of the "rolling ball" is to cause a smoothing in pixel values, as shown by the change that took place between the plots of Figures 3.2c and 3.2d. Figures 3.2a and 3.2b shows the effect of the passage of a rolling ball with 50 pixels of radius over the image surface. In this type of process, it is important that the rolling ball radius is as large as the radius of the largest object present in the image. For that reason, rolling balls with very small radius will cause greater subtraction than rolling balls with larger radius. The ability to choose to make the subtraction of dark backgrounds with lighter objects and choose to create background (not subtract), e.g. the output of the process would be the background itself rather than the image with the background subtracted is another feature present in this application of ImageJ. It is also given to the user the possibility making a preview of the process effect in the image without being permanently altered. This tool has two additional options: **Sliding Paraboloid** replaces the rolling ball with a paraboloid that has the same curvature at the apex as a ball of that radius. This option typically produces more reliable corrections since the rolling ball, a legacy algorithm, is prone to edge artifacts. For calculating the background (rolling the *Subtract Background* ball), images are filtered with an average filter (3 x 3 pixels). With **Disable Smoothing** the unmodified image data are used for creating the background.

The analysis measurements which allow the user to determine areas, average, minimum and maximum areas of interest previously defined, for example by threading, are other tools present in the Process menu.

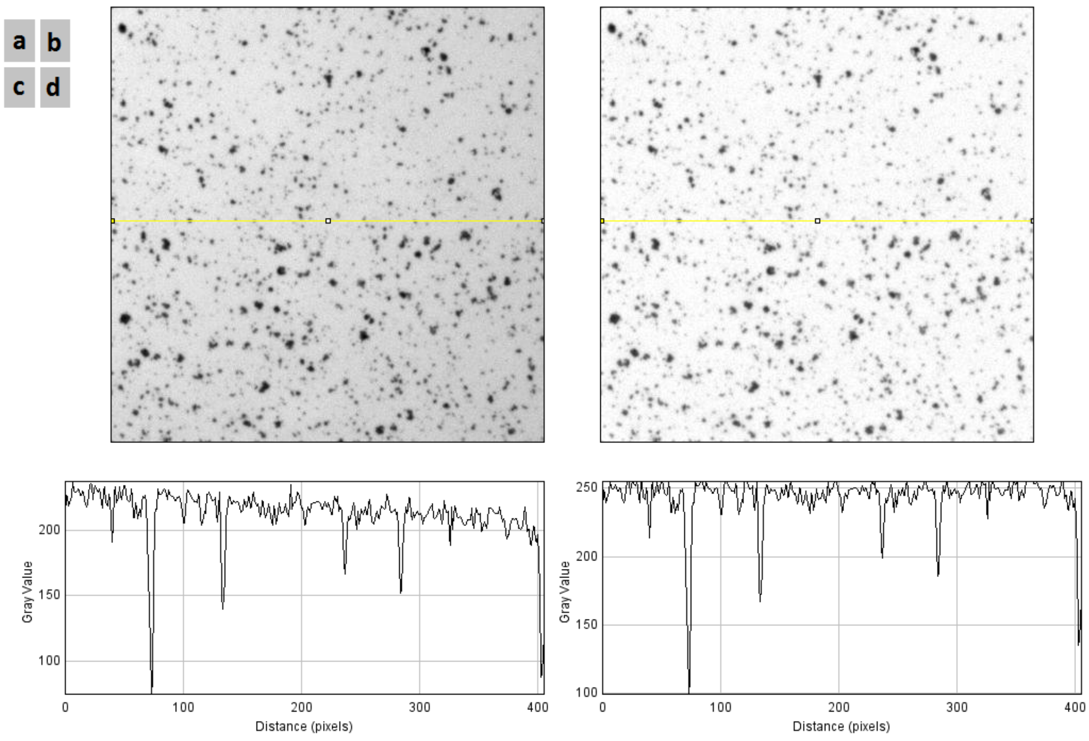


Figure 3.2. Background subtraction of the cell colony image. Figure 3.2a is the original cell colony image (available at http://imagej.nih.gov/ij/images). Figure 3.2b results from the background subtraction of Figure 3.2a, with a rolling ball with 50 pixels of radius. The profile of Figure 3.2c is the plot of the yellow segment in the image of Figure 3.2a. The profile of Figure 3.2d is the plot of the yellow segment in the image of Figure 3.2b.

3.2. Object counting with ImageJ

A particular feature of ImageJ is the possibility of counting objects, such as for example fluorescent cells from 2D images. Although automatic counting is a program application of greater interest for its speed and ease of execution, manual counting also has some features that make it relevant to mention. The automatic counting is interesting, especially in the presence of a large amount of objects, but may be less accurate than manual counting. To count objects manually, the Cell Counter plugin, available on the homepage of ImageJ (http://imagej.nih.gov/ij/plugins/cell-counter.html) is needed.

Before the automatic counting of objects in an image (e.g. in cell colony image), it should be applied in the image a set of processes, to make the count more reliable. The background subtraction is one of these processes. This process results in a higher evidence of the objects of interest, increasing the gap between background pixels and objects of interest pixels.

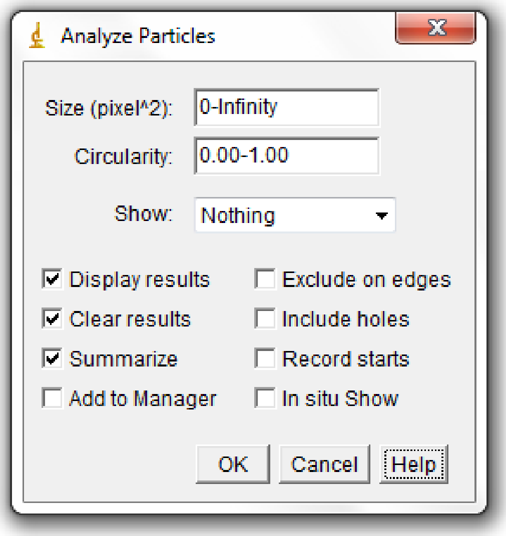
Prior to automatic counting, thresholding is a fundamental procedure. The purpose of this process is the image binarization thus specifying what signal must be included in the analysis based on the intensity of the pixels. A manual adjustment with the sliders may be required to decrease the overlap as much as possible, but is also possible to trust in the automatic adjustment provided by the software. In this case, it is of great importance that all areas of interest are included with or without overlap. The final result is a binarized image which the white areas (pixel value 0) has no interest for the count.

Even the most careful threshold can place two objects or more together, particularly if they are nearby or overlapping. One operation to fix this is “cut” these objects with the watershed tool. However, this process is not completely perfect. The comparison of the final result with the original image to assess the accuracy of the watershed algorithm is always imperative.

Finally, the image is followed by the automatic counting with **Analyze Particles...** tool. A control menu window of the particle analysis function is opened, where could be encountered some particularities that might be included in the final result set (Figure 3.3). The type of objects is the most important among them. It is important to notice that a prior image scaling is required. In the first line of the control menu showed in Figure 3.3, is allowed to specify the range of the size of objects to be counted. This is particularly interesting since it allows to bright noise pixels that have not been eliminated in previous process to not be included in the counting. It’s also possible to set which objects are included in the counting based on its circularity. This is possible by varying the range of circularity, where the minimum extent (0) corresponds to a straight line and the maximum measure (1) corresponds to a perfect circle. If this range is defined then all types of objects are counted. The Show menu allows specifying which image (or overlay) should be displayed after analysis. One is **Outlines** where an 8-bit image is generated containing the numbered outlines of the measured objects. Other is **Masks** where the filled outlines of the measured objects are displayed. The combination of the outlines results with the original image, in order to create an overlay after applying the correct conversions of images, is possible via the *Image Calculator* command available in the Process menu. Other options presentation of results include: "display of results" which lists information of each counted object in extra results that can be saved and opened in MS Excel window; "summarize" that shows the summary results such as total number of objects counted in a window apart which can be saved in Excel; among others such as "exclude edges" that will make the objects that touch the image boundary are not counted.

A particularity of ImageJ is the possibility of applying operations to selected areas or regions of an image. The *Analyze Particles* operation becoming is truly useful in counting objects of different colors in RGB images. This feature allows the differentiation of different types of objects and provides count results of different types.

To illustrate possible differences between objects counting without any image enhancement and with different combinations of operations described, Table 3.1 shows the results of several cell counts in a cell colony image. Test 1 are the results of the cell counting after auto-thresholding


Analyze Particles

Size (pixel^2): 0-Infinity

Circularity: 0.00-1.00

Show: Nothing

[✓] Display results
[✓] Clear results
[✓] Summarize
[ ] Add to Manager
[ ] Exclude on edges
[ ] Include holes
[ ] Record starts
[ ] In situ Show

Figure 3.3. *Analyze Particles* window for particle counting operation of a thresholded image or 8-bit binary image.

(at pixel value 171) of the original image. In Test 2 are presented the results with background subtraction after passage of a "rolling ball" with a radius of 50 pixels. In this case, the auto-thresholding was applied at pixel value of 202. The results of Test 3 are related to the prior procedures, however, with the application of the watershed algorithm. The following results (Test 4 and 5) are identical to the previous two procedures with a background subtraction by the application of 2 pixels radius "rolling ball" and an auto-threshold at pixel value 223. All tests were performed for a circularity of objects ranging from 0 to 1 and from a range of sizes from 0 to infinity.

3.3. Writing ImageJ plugins

Since it was introduced in 1997, ImageJ has been successively grown by developing short add-on programs that provide new functionality to the core program. These additional files can be incorporated into the program in the form of **plugins**, **macros** or **scripts**. It can be written in Java or ImageJ's macro programming language and after being recorded to ImageJ plugins folder it can be accessed through the Plugins menu at the Menu Commands. These loadable code modules turned the simple program for image processing ImageJ on a framework that anyone can use to develop their own imaging solutions. Today are available +500 plugins, +300 macros and +20 scripts. Another way to extend ImageJ is using specific programs such plugins and imaging capabilities, e.g. technically through the use of a library of imaging methods.

The extension of ImageJ through plugins is a much more powerful, flexible and faster concept than with macros and scripts. Actually, most of the menu commands of the core program are implemented by plugins. Moreover, implementing with plugins brings some advantages like having full access to ImageJ and Java APIs, and since ImageJ´s plugins are written in Java programming language (.java source files) which is a richer language with extensive documentation, and compiled to .class files [40]. These files are listed in the ImageJ/plugins/

Table 3.1. Object counting example results.

Test	Count	Total Area	Average Size	Area
		pixels	pixels	%
1	710	10979	15.463	6.628
2	690	10126	14.675	6.113
3	710	10079	14.196	6.085
4	881	8120	9.217	4.902
5	893	8094	9.064	4.886

folder and must have at least one underscore in their name. The acquisition, analysis and processing of plugins can be made using ImageJ's built in editor and Java compiler. Knowing the ImageJ API is extremely important to write plugins. ImageJ documentation can be found at http://rsb.info.nih.gov/ij/docs, including links to further documentation. Furthermore, at http://developer.imagej.net/ides can be accessed instructions for setting up the development environment (command line, *Eclipse*, *Netbeans*, *IntelliJ IDEA*) for ImageJ resources.

3.3.1. Compiling and running plugins

After developing a plugin, it is possible to compile and run it from ImageJ. For this, it is necessary first to copy the java file with the source code to the folder named Plugins, or one of its subfolders in the directory of ImageJ installation. If the Java runtime environment includes an installed Java compiler it is possible to compile and run plugins inside ImageJ. So, to do this, select the **Plugins > Compile and Run...** menu, which opens a dialog box that lets the user select a java file that will be compiled in a .class file. The plugin will run normally. If compiled successfully, go to **Help > Refresh Menus** or (better) restart ImageJ. If the plugin has an underscore "_" in the name, it will appear in the Plugins menu (or submenu).
Additionally, ImageJ has an integrated editor that allows code modification and editing, as well as compile and run plugins. For this, the command **Plugins > New...** shows a dialog box that allows the user to choose a name for the new plugin, while the command **Plugins> Edit...** shows a file open dialog to open a selected plugin in a text editor.

3.3.2. ImageJ API

To program plugins, a brief knowledge of ImageJ´s general class structure is required. The most important classes for the development of plugins are listed and described briefly below. Furthermore, ImageJ class diagram is available in Figure 3.4.
ImageJ – This class is an extension of the functionality of Java's Advanced Windowing Toolkit (AWT) for the implementation of user interface and presentation of image data. Hence, is the main class of ImageJ application, containing the main entry point of the program and the main window of ImageJ.
ImagePlus – *ImagePlus* is a Java object (java.lang.Object) which represents an image via a set of variables such as a class *ImageProcessor* which contains the pixel data array of the 2D image and some basic methods to manipulate it. Other variables are image types represented by

declared constants, as for example COLOR_256 to represent a 8-bit color image; COLOR_RGB representing a color RGB image; GRAY16, GRAY32 and GRAY8 images for 16-bit, 32-bit and 8-bit grayscale, respectively. Other variables, which can be accessed by *get* methods, also represent other important characteristics of images, as well as *Width* and *Height*.

To display images, ImageJ uses a class called *ImageWindow*, an extension of the class *Frame* from AWT. To build an object of the class *ImagePlus*, one of the following constructors may be used:

`ImagePlus ()` – default constructor which creates an empty *ImagePlus* object and does no initialization;

`ImagePlus (java.lang.String pathOrURL)` – an image constructor from the

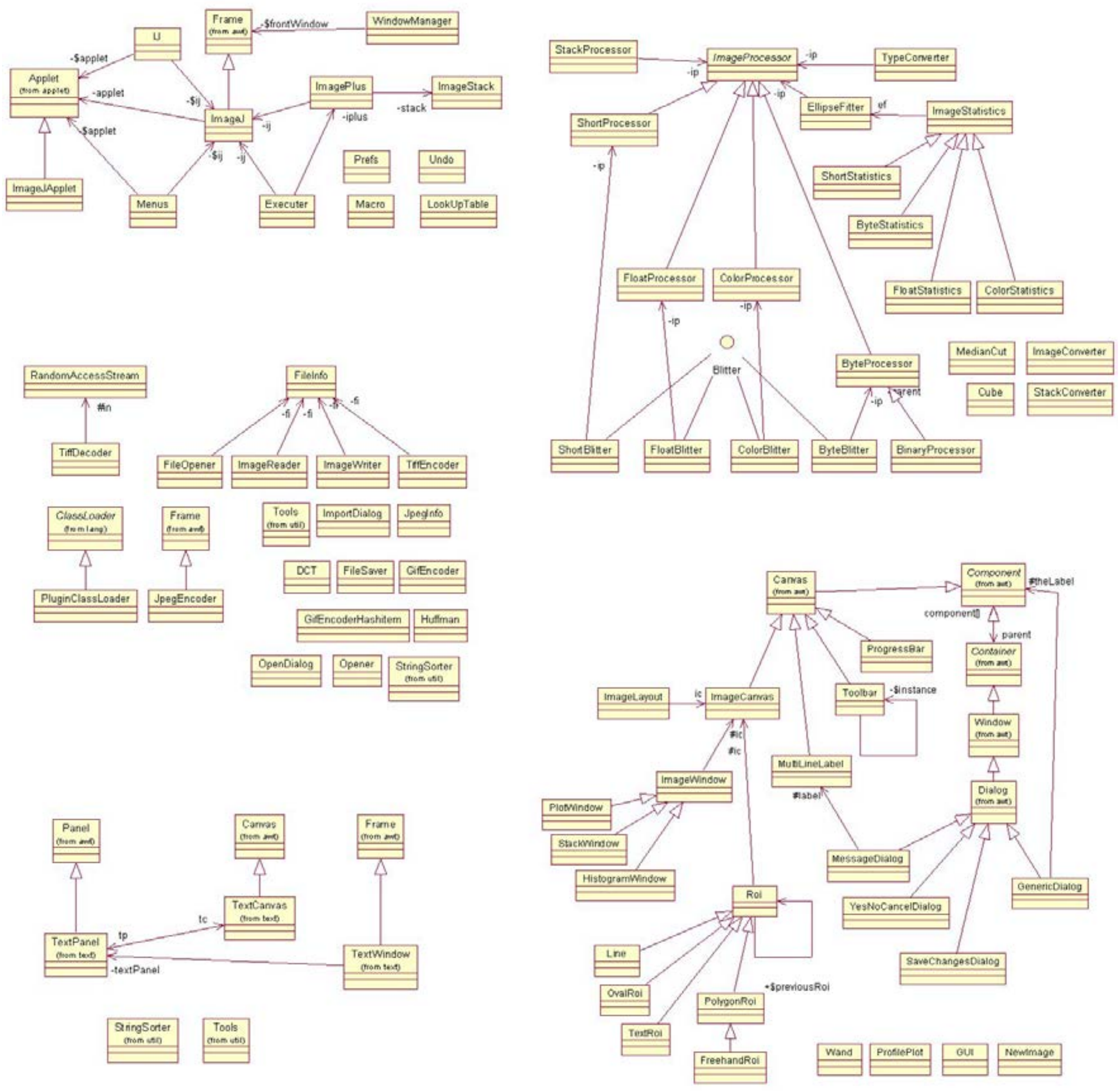


Figure 3.4. ImageJ class diagram.

load the same image through a specific path or URL;

`ImagePlus (java.lang.String title, ImageProcessor ip)` – contructor from a *ImageProcessor* and a title for the *ImageWindow* that shows the image;

`ImagePlus (java.lang.String title, ImageStack stack)` – constructor of a new *ImagePlus* from a stack of images.

In addition to the *get* methods mentioned above, *getWidth* and *getHeight*, this class has many more basic manipulation methods. Below are described some of the most important:

`int getBitDepth ()`- returns the image bit depth, e.g. the type of picture that represents, being 8, 16, 24 or 32, for 8-bit, 16-bit, RGB, or 32-bit, respectively;

`ImageProcessor` `getProcessor ()`-returns a reference to the current *ImageProcessor*;

`void show ()` – opens a window and displays the image of this *ImagePlus*;

`void setTitle (java.lang.String title)` – sets the image name;

`void setProcessor (ImageProcessor ip)` – restores the *ImageProcessor* by a specific other and updates the display;

`void close ()` – closes this image and sets is *ImageProcessor* to null;

`ImagePlus duplicate ()` – returns a copy (clone) of this *ImagePlus*.

ImageProcessor – this class is actually an abstract superclass for subclasses that process the four data types (byte, short, float and RGB): *ByteProcessor* used for 8-bit and color images; *ShortProcessor* used for 16-bit grayscale images; *ColorProcessor* for 32-bit integer images, RGB or 8-bit per channel; and *FloatProcessor* used for 32-bit floating point images.

A *ImageProcessor* has the pixel data of a 2D image as a set of basic methods to manipulate it:

`int getAutoThreshold ()` – returns the pixel value (threshold) that can be used to split the image into objects and background;

`void threshold (int level)` – sets the pixels with a value less than or equal to the level with value 0 and the others with value 255;

`void invertLut ()` – inverts the values of the LUT of this image;

`ImageProcessor convertToRGB ()` – returns an RGB version of this image as a *ColorProcessor*;

`ImageProcessor resize (int dstWidth, int dstHeight)` – creates a new *ImageProcessor* containing a copy of the current with the width and height changed;

`void invert ()` – inverts the pixel values of the image or selection;

`ImageProcessor duplicate ()` – returns a duplicate of this image;

`getPixels ()` - returns a reference to the byte array containing this image's pixel data.

Opener – This is a class to open TIFF, DICOM, FITS, JPEG, BMP or GIF images through a file selected through a dialog box, or a path. Calls the plugin *HandleExtraFileTypes* if the file type is not recognized. It is composed of a set of static variables of type *int* to represent each recognized image types. It has a constructor `Opener ()`. From the set of methods that it possesses, the following are worthy of mention:

`ImagePlus openImage (java.lang.String path)` - attempts to open the specified file as a TIFF, BMP, DICOM, FITS, GIF or JPEG as a *ImagePlus*;

`ImagePlus openImage (java.lang.String directory, java.lang.String name)` – try to open a TIFF, BMP, DICOM, FITS, GIF or JPEG file specified by the directory and name as an *ImagePlus*.

DirectoryChooser – This class shows a dialog box allowing the user to select a directory. It has only one featured method that returns a *String* with the name of the directory chosen by the user. The constructor requires only a *String* as a parameter and displays a dialog box using the specified title.

Analyzer – This Java language object implements the interfaces *Measurements* and *PlugInFilter*, which means that implements the ImageJ commands **Analyze / Measurements** and **Analyze / Set Measurements**. It features a simple constructor by default, the `Analyzer ()`, and another that constructs a new *Analyzer* using an object of class *ImagePlus* using the current measurements and default *ResultsTable* – `Analyzer (`*`ImagePlus`*` imp)`. Finally, the constructor `Analyzer (`*`ImagePlus`*` imp, int measurements, ResultsTable rt)` constructs a *Analyzer* from a given *ImagePlus* image, measurements and *ResultsTable*. These measurements are *static final int* belonging to the field values in the *Measurements* interface. They represent the 26 measurements possible to select through the **Analyze < Set Measurements** command and among them we can find some more important such as AREA, CIRCULARITY, ELLIPSE and PERIMETER. AREA, as the name suggests, represents the area of a selection in square pixels, or pre-calibrated square units through the **Analyze < Set Scale** command. PERIMETER is the perimeter of the outer edge of a specific selection. ELLIPSE represents the ellipse that best fits a selection using the Major, Minor and Angle measures. The first two represent the major and minor axis of the ellipse, respectively, while ANGLE is the angle between the major axis of the ellipse and the line parallel to the X-axis of the image. CIRCULARITY calculates and displays a set of shape descriptors, including *Circularity*, *Aspect ratio*, *Roundness* and *Solidity*. **Circularity** (*Circ*) is given by the following equation:

$$Circ. = 4 \times \pi \times \frac{Area}{Perimeter\ ^2} \tag{3.1}$$

If the result of the equation 3.1 is 1, it indicates a perfect circle. As a result approaches to 0, this means that the shape elongates. The **Aspect Ratio** (*AR*) of the ellipse that fits the object is calculated by the equation 3.2:

$$AR = \frac{Major}{Minor} \tag{3.2}$$

Roundness (*Round.*) is calculated by the inverse of *AR* or with recourse to equation 3.3:

$$Round. = 4 \times \frac{Area}{\pi \times Major\ ^2} \tag{3.3}$$

Solidity is the result of dividing the Area by the Convex Area, where this latter is the area resulting from the application of the Convex Hull command.

ParticleAnalyzer – This class also uses the *PlugInFilter* and *Measurements* interfaces. It is the implemented representation in Java of the ImageJ *Analyze Particles* command.

An object of this class is defined by a set of state variables, where the most important from the point of view of this dissertation are an instance of the class *Analyzer* and other instance of the class *ResultsTable*, which in turn represents the tables Results and Summary returned by this command. Other variables of relative importance are the integers that when counted activate the following functions: EXCLUDE_EDGE_PARTICLES defines that the particles touching the edge of the image are not accounted for; SHOW_MASKS shows an image containing the binary masks of the measured particles; SHOW_NONE shows no picture with the outlines of the measured particles; SHOW_OUTLINES shows an image with the outlines of the measured particles; SHOW_RESULTS shows the results with measurements selected on the class *Measurements*; DISPLAY_SUMMARY shows in another table the results of particle counts, total area, mean particle size and other parameters defined on the object *Measurements*.

The construction of an instance of this class is possible with empty constructor `ParticleAnalyzer ()` and the `ParticleAnalyzer (int options, int measuremts, ResultsTable rt, double minSize, double maxSize, double minCirc, double maxCirc)`. The value of options is determined by the sum of all variables values of integer type from *ParticleAnalyzer* class, thus defining what the desired parameters in the final result. As the value options, the measurements value defines the measures appearing in the resulting table. The result table *rt* passed as parameter in the constructor, could be an empty table and receives the values of the taken measurements. If this table is already initialized, the measurements are appended to the original table. The values *minSize*, *maxSize*, *minCirc* and *maxCirc* define respectively the minimum and maximum values of size and circularity of detected particles.

Among the methods implemented in this class, three of them are highlighted below:

`boolean Analyze (`*`ImagePlus`*` imp)` – performs particle analyze on the specified image;

`ImagePlus getOutputImage ()` – returns the "Outlines", "Masks", "Ellipses" or "Count Masks" image, or null if "Nothing" is selected in the *Measurements* class;

`void setHideOutputImage (Boolean hideOutputImage)` – if true does not displays the image selected in *Measurements* class.

ResultsTable – This class represents the table that stores the measurement results as columns of values.

The state variables represent each possible measurement that can be displayed, from AREA, AREA_FRACTION, and CIRCULARITY among others, as well as some representative of the state of the table, such as COLUMN_IN_USE, TABLE_FULL or COLUMN_NOT_FOUND.

The constructor for this class is a simple empty constructor `ResultsTable ()` that generates a *ResultsTable* with a counter = 0, no columns and with a precision of decimal places equal to 3. It has a large number of methods that permit access, modification, and different types of table management. Among these, the most prominent are:

`float [] getColumn (int column)` – returns a copy of the column as a float array with the values in the column with index passed as a parameter, or null if the column is empty;

`int getColumnIndex (java.lang.String heading)` – returns the index of the first column with the name passed as a parameter;

`java.lang.String getStringValue (int column, int row)` – returns the *String* value of the column and row with indices passed as parameter, where these indices cannot be less than zero or greater than the number of columns and rows, respectively;

`java.lang.String getStringValue (java.lang.String column, int row)` - returns the *String* value of the column with name passed as a parameter and the line with index passed as a parameter, where this index cannot be less than zero or greater than the number of lines;

`void setValue (int column, int row, double value)` – puts the value passed as parameter in the column and row with indices passes as parameter, where the index values must be greater than zero and less than or equal to the number of columns and rows, respectively;

`void setValue (java.lang.String column, int row, double value)` – puts the value passed as a parameter in the column with the given name and row with the given index, where the index value must be greater than zero and less than or equal to the number of lines;

`void setPrecision (int decimalPlaces)` – select the number of decimal places, passed as parameter, of the values presented in ResultsTable;

`void show (java.lang.String windowTitle)` – shows *ResultsTable* contents in a window with title specified by the passed parameter, or update an existing results window;

`void showRowNumbers (boolean showNumbers)` – if the parameter is true, the line numbers are displayed;

`void reset ()` – deletes the entire contents of the *ResultsTable*.

ImageStack – This class represents the expandable array of images that ImageJ provides to store as images stacks multiple images of the same size. For this very reason and beyond the default constructor – *`ImageStack ()`* – to create a new empty stack is necessary to pass as constructor parameters the width and height values of the stack – *`ImageStack (int width, int height)`*. In addition, it is possible *a priori* to set the number of possible images to store in a new stack – *`ImageStack (int width, int height, int size)`*. Besides the constructors presented, the class *ImageStack* also has methods to create new stacks:

`ImageStack createEmptyStack ()` – returns an empty stack image with the same resolution and color table of the opened image;

`void setStack (java.lang.String title, ImageStack stack)` – sets a newly created *ImageStack* as a stack of an *ImagePlus*.

To add and remove images from a stack the following methods should be selected:

`void addSlice (java.lang.String sliceLabel, ImageProcessor ip)` – add an image represented by the class *ImageProcessor* to the end of the stack;

`void addSlice (java.lang.String sliceLabel, ImageProcessor ip, int n)` – add an image represented by the class *ImageProcessor* to the stack following slice n;

`void deleteLastSlice ()` – eliminate the last slice of the stack;

`void deleteSlice (int n)` – eliminate a specific slice, where n ranges from 1 to the number of slices in the stack.

IJ – This is a class that contains many very useful utility static methods. Requiring no constructor, just being imported from ImageJ API, this class allows almost everything to be done, from the invocation of menu commands, macros and plugins, through its methods. To access all menu commands there are two possible methods:

`static void doCommand (java.lang.String command)` – executes a menu command, and proceed immediately without waiting for the command to be executed;

`static void run (java.lang.String command)` – executes a command and continues as soon as the command has finished;

`static void run (java.lang.String command, java.lang.String options)` – executes a command, passing additional options as a parameter;

`static void run (ImagePlus imp, java.lang.String command, java.lang.String options)` – executes a command on an image by passing additional options as parameter.

Among all these methods, it is the latter that has significant advantages. Although, it does not have the advantage of not blocking while the query is executed as with the *doCommand* method, it has the advantage of running the command on a specific image. The fact that the image is represented by *ImagePlus* class does not imply it to be opened, making the method lightweight at visual and memory level. Moreover, this way opens up the possibility of implementing plugins that it is not necessary that images are visually available, as we shall see later on.

3.3.3. Types of Plugins

The plugin coding for ImageJ comprises implementing classes, e.g., for each plugin a new class is created. This only implies that each plugin is implemented by a single class, not excluding the fact that this class has other classes implemented in its own or imported. The use of classes of the ImageJ API is achieved through their imports according as they are organized into packages. Thus, any ImageJ plugin code starts with the proper imports of the classes from ImageJ API or Java libraries which it uses. More details about Java libraries and their imports are available at **http://docs.oracle.com/javase/7/docs/api/**. Among the most common classes in ImageJ plugins, the imports that usually appear are:

```
import ij.*;
import ij.gui.*;
import ij.io.*;
import ij.measure.*;
import.ij.plugin.*;
import.ij.process.*;
import java.io.*;
import java.util.*;
```

The asterisk (*) at the end of each import is substituted by the name of the class that the programmer wants to use, or else, all the classes contained in the respective package will be accessible. For example, any image to process must inevitably be represented by *ImagePlus* and *ImageProcessor* classes. For this, the necessary imports would be as follows:

```
import.ij.ImagePlus;
import.ij.process.ImageProcessor;
```

A final mandatory import must be performed concerning the type of plugin required. Basically there are two types of plugins: without images as input, implemented by the interface *PlugIn*; and with images as input, implemented by the interface *PlugInFilter*. After deciding the type of plugin to encode, one or other of the following imports must be made:

```
import ij.plugin.PlugIn;
import ij.plugin.filter.PlugInFilter;
```

These two interfaces apply a considerable number of classes. As stated in the API documentation to developers, plugins for receiving images or Windows displaying should be implemented by the interface *PlugIn*, while plugins that process images should be implemented by the interface *PluginInFilter*. Nevertheless, classes implemented by these two interfaces are not exclusive, being the developer free to use a class from one or another interface regarding the type of plugin. Images must be represented by the *ImagePlus* or *ImageProcessor* class for plugins implementation.

The interface *PlugIn* comprises only one method - `void run (java.lang.String arg)`. This method is invoked when the plugin is started. “arg” is the argument specified for this plugin in the file IJ_Props.txt. This configuration file is available at **http://imagej.nih.gov/ij/developer/**. The interface *PlugInFilter* also has a `run ()` method, but however it is passed as a parameter an instance of the class *ImageProcessor*, referring to the image that will be processed by the plugin - `void run (ImageProcessor ip)`. The processor can directly be modified, or a new image or processor could be based on its data, leaving intact the original image. Unlike the `run ()` method of the *PlugIn* interface, it has no *String* argument, which may be passed in the method `int setup (java.lang.String arg, ImagePlus imp)`. This method runs once the plugin is started or when the filter is loaded. “arg”, which may be blank, is the argument specified for this plugin in IJ_Props.txt or in the plugins.config file of a jar archive containing the plugin. “imp” is the currently active image. This method should return a flag word that specifies the filters capabilities. For Plugin-filters specifying the FINAL_PROCESSING flag, the setup method will be called again, this time with arg = "final" after all other processing is done.

This method sets up the plugin filter for use. The *String* arg has the same function as in the run method of the *PlugIn* interface. The programmer does not have to care for the argument imp—this is handled by ImageJ and the currently active image is passed. The setup method returns a flag word that represents the filters capabilities (e.g. which types of images it can handle). In the method’s code, the image argument is stored in the instance variable, and the capability flag of the filter plugin is returned. The following capability flags are defined in *PlugInFilter*:

`static int CONVERT_TO_FLOAT` – set this flag to have the *ImageProcessor* that is passed to the `run()` method converted to a *FloatProcessor*;

`static int DOES_16` – Set this flag if the filter handles 16-bit images;

`static int DOES_32` – Set this flag if the filter handles *float* images;

`static int DOES_8C` – Set this flag if the filter handles 8-bit indexed color images;

`static int DOES_8G` – Set this flag if the filter handles 8-bit grayscale images;

`static int DOES_ALL` – Set this flag if the filter handles all types of images;

`static int DOES_RGB` – Set this flag if the filter handles RGB images;

`static int DOES_STACKS` – Set this flag if the filter wants its `run()` method to be called for all the slices in a stack;

`static int DONE` – Set this flag if the filter does not want its run method called;

`static int FINAL_PROCESSING` – Set this flag if the setup method of the filter should be called again after the calls to the `run (ip)` have finished;

`static int KEEP_THRESHOLD` – Set this flag to keep the invisible binary threshold from being reset;

`static int NO_CHANGES` – Set this flag if the filter makes no changes to the pixel data and does not require undo;

`static int NO_IMAGE_REQUIRED` – Set this flag if the filter does not require that an image be open;

`static int NO_UNDO` – Set this flag if the filter does not require undo;

`static int PARALLELIZE_IMAGES` – Set this flag if images may be processed in parallel threads;

`static int PARALLELIZE_STACKS` – Set this flag if the slices of a stack may be processed in parallel threads ;

`static int ROI_REQUIRED` – Set this flag if the filter requires an ROI;

`static int SNAPSHOT` – Set this flag if the filter requires a snapshot (copy of the pixels array);

`static int STACK_REQUIRED` – Set this flag if the filter requires a stack;

`static int SUPPORTS_MASKING` – Set this flag if the filter wants ImageJ, for non-rectangular ROIs, to restore that part of the image that's inside the bounding rectangle but outside of the ROI.

3.3.4. Coding new plugins

Presented the two types of plugins it remains to describe the implementation of each one in Java code.

The first code is for a plugin that implements the interface *PlugInFilter* and calculates the number of pixels in a ROI selection of an image showing the result in a message window. For that, a mask is created through the method `getMask ()` for irregular ROI's. The count of pixels with value equal to 255 is made through the mask histogram. The result is returned in a window through the command `IJ.showMessage (java.lang.String msg)`. The filter of this plugin accepts all kinds of images and requires the presence of ROI selections, so the DOES_ALL and ROI_REQUIRED flags are selected in the setup method.

```
import ij.IJ;
import ij.ImagePlus;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;
public class Mask_area implements PlugInFilter {
     public int setup(String arg, ImagePlus imp) {
          return DOES_ALL+ROI_REQUIRED;
     }
     public void run(ImageProcessor ip) {
          ImageProcessor mask = ip.getMask();
          int area = mask.getHistogram()[255];
          IJ.showMessage(String.valueOf(area) + "\n");
     }
}
```

The second Java code is for a plugin that implements the interface *PlugIn* and counts the number of pixels of an image. An *ImagePlus* instance is created through the path of the directory of the image. Then, the number of pixels of the image is returned by combining the methods `getProcessor ()`, from *ImagePlus* class, and `getPixelCount ()` from *ImageProcessor* class. The final value is displayed in a window.

```
import ij.IJ;
import ij.ImagePlus;
import ij.plugin.PlugIn;
public class Image_area implements PlugIn {
     public void run(String arg) {
          String path = "C:\\pathExample\\roi.tif";
          ImagePlus image = new ImagePlus(path);
          int area = image.getProcessor().getPixelCount();
          IJ.showMessage(String.valueOf(area) + "\n");
     }
}
```

Despite the similar operations performed by the two codes, only in the first one is necessary that an image is opened. The operation performed can be accomplished by loading a previously stored ROI. However, becomes more pleasant to perform interactively and directly onto an open image, allowing the user to manually make a selection. In the second code, the user only needs to indicate the directory and name of the image to be analyzed or to select the folder images through a dialog box. These reasons lead to the choice between types of plugins presented here, which will make the plugin more efficient both in terms of programming and user-program interface level.

4. Methodology and algorithms

Building a plugin for ImageJ to collect and process microscopic images of *Saccharomyces cerevisiae* cultures is not a trivial process by itself. Based on this, methodologies have been developed to improve the efficiency of the algorithm included in the plugin as well as the user-program interface. This chapter presents all steps taken to develop a final version of the plugin called **ScerevisiCounter**.

Since ImageJ core program consists of plugins implemented in the informatics language Java, the *Eclipse IDE SDK*, version 4.2.1, build id M20120914-1800, was used for the development of ScerevisiCounter. Furthermore, it was necessary to setting up *Eclipse* to create and debug plugins for ImageJ. For this, it was created a Java Project named IJ with default output folder set to IJ / bin. Finally, it was copied to the source folder a ij folder and subfolders withdrawal from the official webpage of ImageJ (http://rsb.info.nih.gov/ij/download/src/). The extracted version of ImageJ source code was the 1.48f. All experiments were performed on a PC with 2.13 GHz Intel(R) Core(TM) i3 CPU, 4GB RAM, running Windows 7 Home Premium (SP1),

4.1. ScerevisiCounter´s Main Pipeline

The plugin ScerevisiCounter consists of a set of processes from choosing the directory of the image or images to be processed and to display the final result. To determine which processes would be part of the plugin code, a series of plugins were created in order to test and try the full potential offered by the ImageJ API. In the next sections, the implementation of the plugin as well as the proposed solutions and features for some drawbacks are presented.

4.1.1. Opening images

As previously discussed, there are two main types of plugins: requiring images as input (opened images) implemented by the interface *PlugIn*, and requiring images as input, implemented by the interface *PluginFilter*. One of the first issues that emerged was what kind of actions the user would need to take to indicate the image he wants to analyze. The possibility of the plugin simultaneously process more than one image at time arises, being a more attractive option from user´s point of view.

In the case of ScerevisiCounter implemented by the interface *PluginFilter*, the user needs to open previously the images to analyze, through one of the possible commands existing in the File menu of the program interface. So, right from the start this solution was not attractive, remaining the hypothesis implemented by the interface *PlugIn*.

The ImageJ´s API contains an *IJ* class which has implemented some very useful static methods (Chapter 3.3.2). One of them, *IJ.open ()*, opens a dialog box that allows the user to select and open an image. From here, the image would easily associated to the class *ImagePlus* through the method *getCurrentImage ()* from the *WindowManager* class. The *open ()* method is the simplest existing for these purposes, and cannot open more than one image, unless the plugin has a prior information on the number of images that the user wants to open. Another possibility of this command is to pass as parameter a *String* denominating the directory and image name. Similarly, the *ImagePlus* class has a constructor that creates an instance of this class passing as parameter the path of image location. None of these hypotheses solve by itself the question of access to multiple images, unless the number of images to open is known in advance.

Via a dialog box of the class *DirectoryChooser* or by calling the method *IJ.getDirectory (String title)*, enables the user to select a folder containing the image or images to be analyzed. A *String* with the path of the folder could be passed as parameter in the constructor of the class *File* (*java.io.File*) creating an instance of *File* and converting the *String path* in an abstract path. Through the method *list ()*, an array of *String*s naming the files and directories in the directory denoted by the abstract path name is returned. From here, the number of images in the directory would be easily obtained and through a cycle would be generated instances of the class *ImagePlus* with the directory and image titles saving as parameters for the constructors.

4.1.2. Methodologies configuration

The counting and classification of *S. cerevisiae* cells were the main objectives for the development of this plugin. ImageJ *Analyze Particles* is a key command, where the command is implemented in ImageJ´s API by the class *ParticleAnalyzer*, and giving quantities of particles in black and white images. It counts and measure particles in thresholded or binarized images, and one of these two methods, *Threshold* and *Binaryze*, or even both, would also have to be present in the ScerevisiCounter code.

The chosen command for images binarization was the *AutoThreshold*, which automatically uses the default method to set the threshold levels based on the analysis of the image histogram. This selection appeared to be the most reasonable since the threshold level greatly varies from image to image. The threshold method, from *ImageProcessor* class, was used passing as a parameter the integer resulting from the method *getAutoThreshold*, from the same class.

The effect caused by this command can be seen in Figure 4.1. Figure 4.1a shows an 8-bit image of two yeast cells submitted to an auto threshold resulting in a binary image (Figure 4.1b), where the clearer edges of the original image are in black. After binarization, the resulting image was subjected to a particles analysis. This analysis resulted in a mask (Figure 4.1c) where the interior of the particles of the original image (Figure 4.1b) were filled with black being the rest represented with white. Another option can be an image where the particles of the binary image are numbered (Figure 4.1d). According to the desired results, as can be seen with more detail in the following chapter, the *Analyze Particles* command should include the options *exclude edge particles*, *show masks* and *show outlines*. The *ParticleAnalyzer* minimum and maximum area limits of the detected particles were 50 and 6000 pixels, respectively. The measurements selected to appear in the final results were area, ellipse, circularity, perimeter and centroid of the

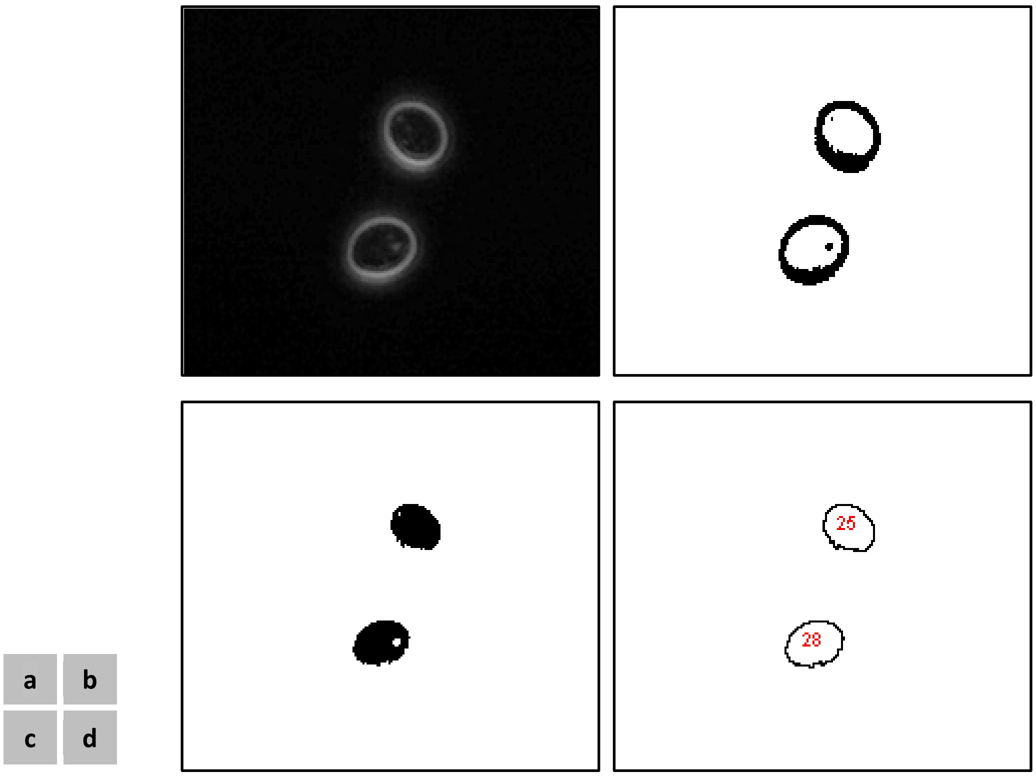


Figure 4.1. Particle analysis of a microscopic picture of yeast cells. Figure 4.1a is an 8-bit picture of two yeast cells. The image in Figure 4.1b is the result from auto thresholding the image from Figure 4.1a.Figures 4.1c and 4.1d are the mask and outlines resulting from the particle analysis of the image from Figure 4.1b, respectively.

particles.

Despite the minimum and maximum particle size is given in pixels, it was desirable that the final results were given in actual sizes. Given the small sizes of *S. cerevisiae* cells, micrometer (µm) was the unit selected for the particles measurement. In this case it was necessary to calibrate the images so that the width of one pixel (or height since it were square pixels) corresponded to 0.160 µm. In case of areas 1 pixel corresponds to 0.160 x 0.160 µm^2.

Given the possibility to process more than one image simultaneously, one of the potential desired final results would be to display the outlines of the detected particles superimposed on the original images processed by the plugin. To avoid the large number of windows opened by the plugin in end of each analyzed image, an instance of the class *ImageStack* was created to accommodate all the images. The height and width were passed as parameters in the class constructor to accommodate at least the major analyzed image. From then on, all images would be included in the stack without generating a compilation error.

This set of commands form the basis of ScerevisiCounter **Main Pipeline**. All other commands or methods that could be added to the code were subjected, as described in the following chapters, to a more detailed study in order improve the plugin performance. In a simplified view, and given the pipeline presented until this point of this report, the plugin has a general algorithm with the following structure:

```
Stack – ImageStack class instance
Path – images folder String directory
List – array of titles
FOR (each Title in List)
     Image := instance of the class ImagePlus (Path, Title)
     AutoThreshold (Image)
     ParticleAnalyzer (Image)
     Calibrate (Image)
     Add Image to Stack
Return Stack
```

4.1.3. Type of results

ScerevisiCounter proposes to count and classify *S. cerevisiae* cells in microscopy images. It was previously stated that the counting task is easily achieved by the *Analyze Particles* command already implemented in ImageJ software. However, based on cells morphology four types were defined: **small**, **normal**, **large** and **bud** cells.

The first three are easily classified by the size of their areas. For this, twenty microscopic images of yeast cells were analyzed and the data for the area, centroid, ellipse and circularity were collected. Each group of data for each cell was associated with a manual classification of bud cell or non-bud cell. The values for the areas of small, normal and large cells were determined based on the areas of the non-bud cells. Given the sample size, it was adopted an equal distribution of sizes, where the value of the first tercile corresponds to the limit between small and normal cells, and the limit value between normal and large cells, determined in a similar way, corresponds to the second tercile of the areas data. The classification of bud cells was determined by the analysis of the area, ellipse, centroid and circularity values.

The ImageJ´s *Analyze Particles* tool has the option to show a result table whose contents can be specified from a range of measurements already presented in Chapter 3.2. It would be interesting to include in this table the cells areas and respective classification. Being the Results Table displayed at the end, it would be interesting, as well, to present the title of the image which each respective cell belongs. The class of objects *ResultsTable* from ImageJ API has all the necessary methods to add a *Group* column and an *Image* column, with the classification assigned to each cell and it respective image title. To this table was assigned the name of "Results of" + folder name, where this folder is the folder with the analyzed images.

The cells count results were presented as a table including the cell count per group of cells, a total number of cells per image, a total number of cells in each group for all images and a total of cells detected on all images. Again, the class *ResultsTable* served to manipulate the Summary Table coming from the *Analyzer*, to include any desired information relating to the cell count. To this table was assigned the name of "Summary of" + folder name.

Finally, after the classification and measurement of each cell in each image, a correspondence between the values shown in Results Table and each cell present in each image was missing. For this, the numeric identifier at the beginning of each line of the Results Table, being independent of the analyzed images, could serve to solve this demand. For this purpose, it was necessary to overlap each cell in each image by the corresponding number given to the respective measurements in the Results Table. For this, the outlines from particle analysis were superimposed on each image from the final stack, preserving and maintaining the original images untouched and adding a corresponding number to each cell.

The process of delimiting the outlines of the cells in the final images is achieved by performing the following set of instructions, being this process called by *Drawing Outlines*:

```
options = ParticleAnalyzer.EXCLUDE_EDGE_PARTICLES +
ParticleAnalyzer.SHOW_OUTLINES;
particleAnalyzer.analyze(image);
ImagePlus outlines = particleAnalyzer.getOutputImage();
ImageProcessor outlinesProcessor = outlines.getProcessor();
outlinesProcessor.invertLut();
outlines.setProcessor(outlinesProcessor);
IJ.run(outlines, "Blue", "");
outlinesProcessor = outlines.getProcessor();
outlinesProcessor.invertLut();
outlinesProcessor = outlinesProcessor.convertToRGB();
ImagePlus image = open.openImage(path, imageTitle);
ImageProcessor ImageProcessor =
image.getProcessor().convertToRGB();
image = new ImagePlus(imageTitle, ImageProcessor);
```

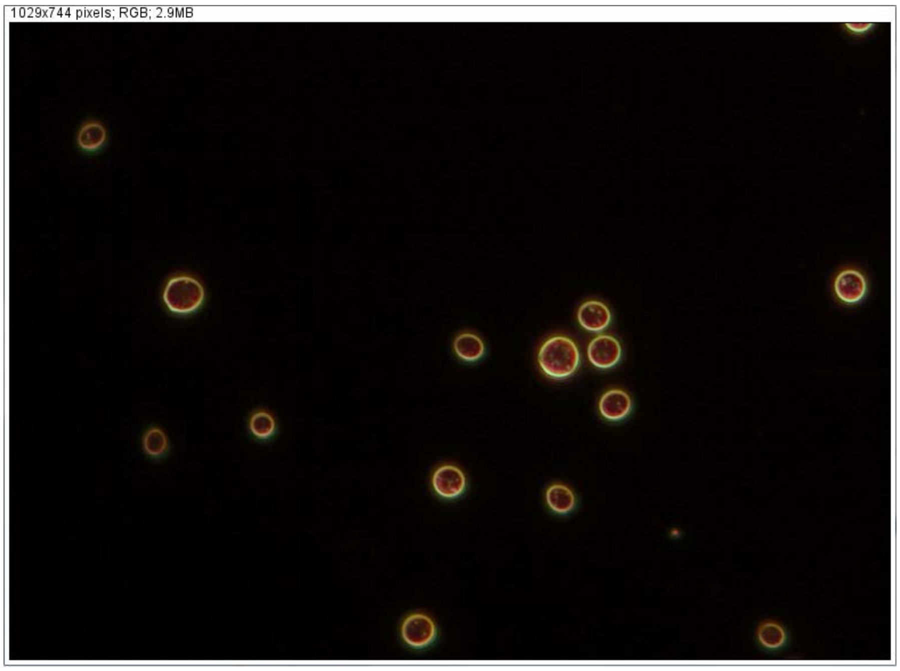


Figure 4.2. Microscopic image (200 ms of exposure time) of a group of *S. cerevisiae* cells. One of the cells is cut by the edge of the picture and one of the visible spheres is not counted as a cell due to the reduced size, both of which are not accounted by ScerevisiCounter plugin.

```
ImageCalculator ic = new ImageCalculator();
ImagePlus final = ic.run("Add create", image, outlines);
ImageStack.addSlice(imp3.getTitle(),imp3.getProcessor());
```

After the application of the *Particle Analyzer* to a specific image, the resulting outlines are instantiated as objects of the class *ImagePlus* and the class *ImageProcessor*, through the method *getOutPutImage ()* of the class *ParticleAnalyzer* and the method *getProcessor ()* from the class *ImagePlus*. The LUT of this image should be inverted so that the application of a blue LUT has an effect on the image background, remaining the particles edges in white. This image should be re-inverted and converted to RGB so that the edges become at the color of the LUT (blue) while the background is converted to white. Finally, the resulting image is added to an instance of the original image, and finally added to the final stack of images.

Figure 4.2 and 4.3 shows two original images where the ScerevisiCounter plugin was applied. The final processed images provided by the plugin are shown in Figures 4.4 and 4.5. In these Figures it is possible to observe the same cells from the original images numbered and delimited by a blue outline. The numerations are able to identify the corresponding measurements and classifications of each cell (Figure 4.6). It is also possible to create a detailed counting of the

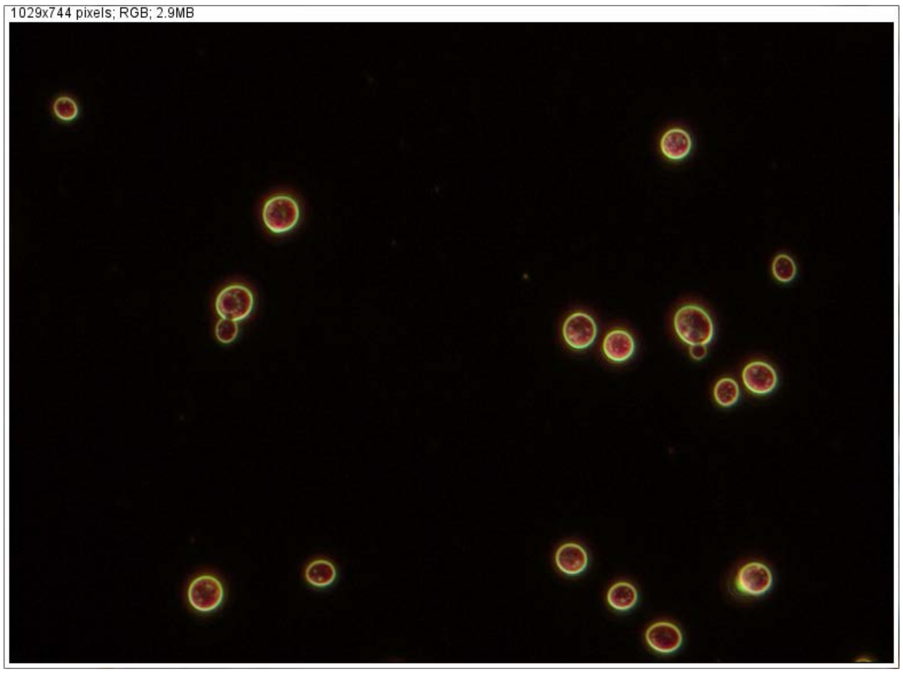


Figure 4.3. Microscopic image (200 ms of exposure time) of a group of *S. cerevisiae* cells. Two cells may be considered as buds due to the protuberance in its edge.

number of cells present in each image and the number of cells per group and these results are presented in Figure 4.7. *Image1* and *image2*, are related to the original images from Figures 4.2 and 4.3, respectively.

The Main Pipeline for the ScerevisiCounter plugin is now shown combining the methods described in the previous chapters with the results described in this chapter. In the following chapters, methods and algorithms added to the plugin in order to make it effective counting and detecting all the cells in the images of yeast as well as detecting bud cells are discussed.

4.2. Methodology optimization

So far, it was introduced, explained and justified the Main Pipeline of the plugin ScerevisiCounter for the processing and analysis of *S. cerevisiae* images. As seen, the pipeline consists in a selection of images to be processed followed by a cycle in which is assigned a threshold level to each image and a particle analysis of the thresholded image. At the end, two tables are generated, one with the cell by cell measurements and classifications results and another with the summary results consisting in an accounting of cells per group and per image. Furthermore, any images that have been processed are presented with the outlines drawn on the detected

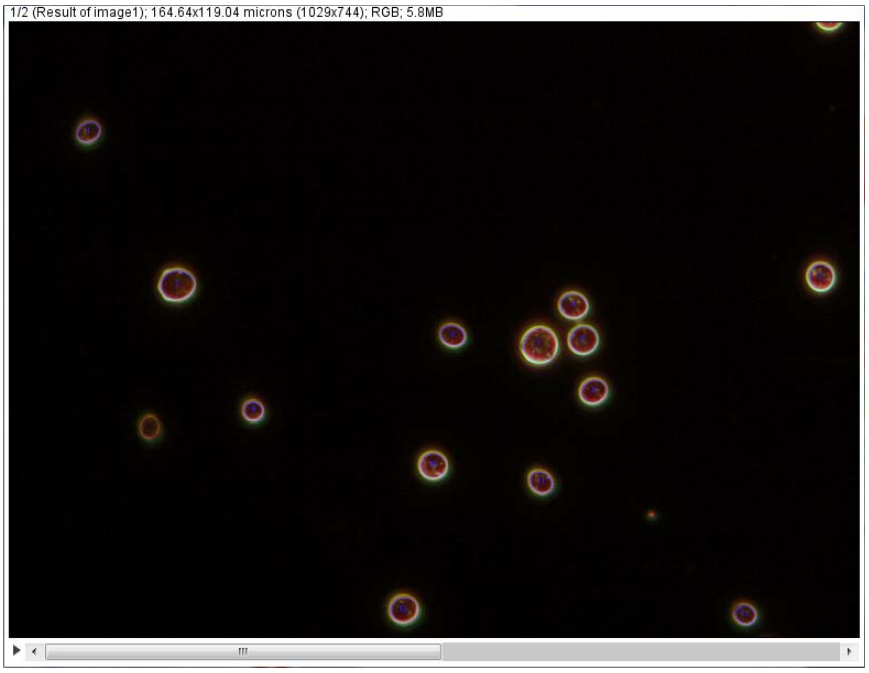

1/2 (Result of image1); 164.64x119.04 microns (1029x744); RGB; 5.8MB

Figure 4.4. Image 1 of 2 from the stack returned after processing the images of Figures 4.2 and 4.3 with the ScerevisiCounter plugin. This image refers to the image in Figure 4.2. It can be verified that all the cells were enumerated and properly marked by a blue outline, except the cell cropped by the edge of the image and near the lower left corner of the image. Small particles or debris were not processed.

cells, as well as an identification number per cell.

However, this set of methodologies can be considered insufficient for a reliable quantification and qualification of *S. cerevisiae* cells. Figure 4.8 shows the effect of applying an automatic threshold, followed by an analysis of particles with sizes above 200 pixels and circularity between 0 and 1, and finally the application of the *Drawing Outlines*. Regarding Figure 4.8b, and despite the detection of all cells, a poor image analysis was performed. The main disadvantage of the present processing methodology is the incorrect distinction between touching cells and bud cells. Due to the unsuccessful classification of *S. cerevisiae* cells, in the present work the addition of pre-processing methods to obtain the maximum efficiency from the Main Pipeline was required.

In the following sections, the experiments performed to determine which and how the combination of image processing methods whose result gives a more effective analysis by the Main Pipeline are presented. Although this work is very dependent on the tested images, thus requiring a large number of images, the experiments were carried out on the image of Figure 4.9. This image acquired with an exposure time of 178.1 ms, 400x of magnification and dark field contrast was a good example as it features a large number of cells, including bud cells, isolated cells, cells in group and high number of debris. It is also advanced that since the evolution of the

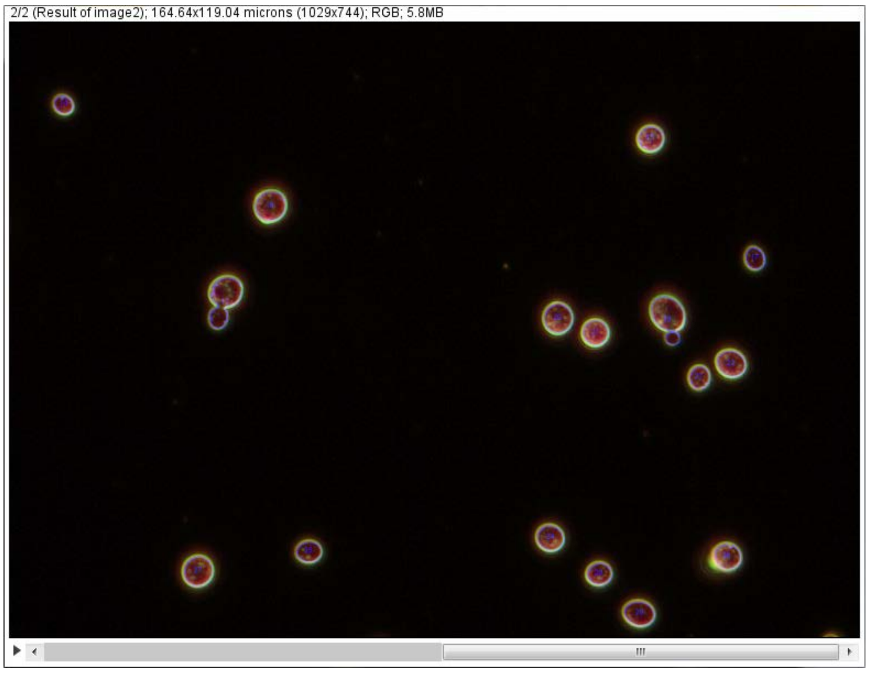
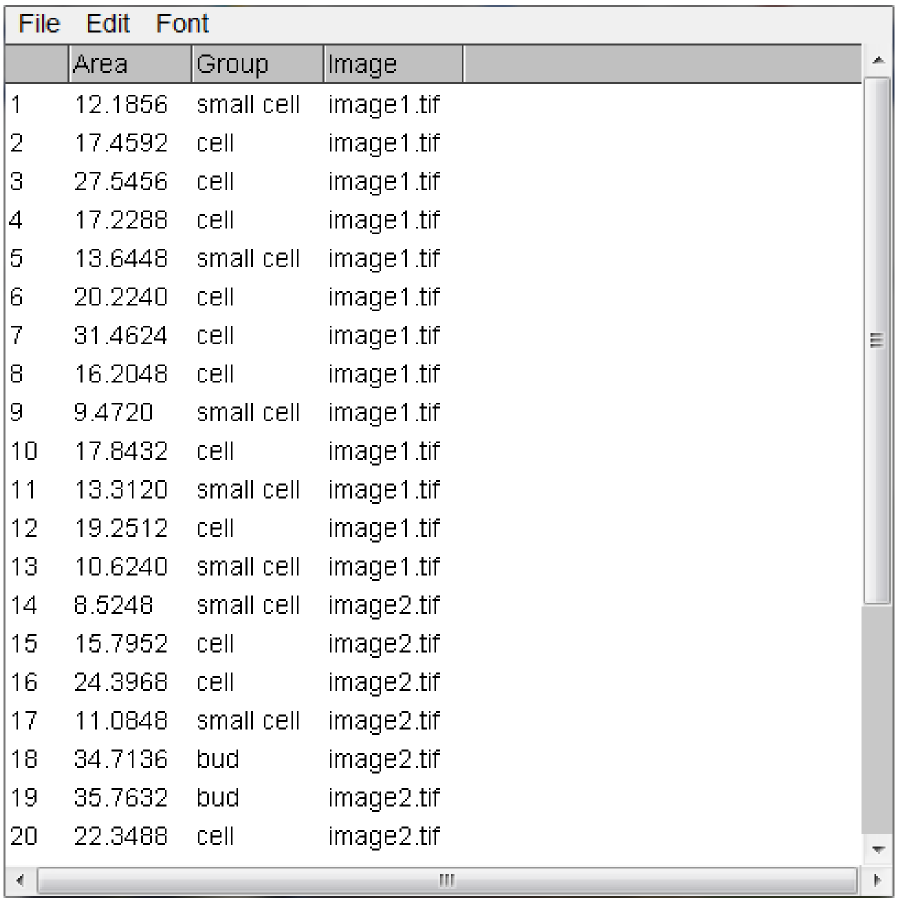

2/2 (Result of image2); 164.64x119.04 microns (1029x744); RGB; 5.8MB

Figure 4.5. Image 2 of 2 from the stack returned after processing the images of Figures 4.2 and 4.3 with the ScerevisiCounter plugin. This image refers to the image of Figure 4.3. All cells were properly enumerated and marked by a blue outline. The numbering was done synchronously with the numbering of the picture 1 of 2 of the final stack.

experiments depended on a general analysis of the results of the experiments already performed, it is possible to immediately state that the latest described experiments were those programmed in function of the previous experiments results on the image in Figure 4.9. The image of Figure 4.10 shows the resulting mask from a manual analysis of the image of Figure 4.9.

4.2.1. Processes

The methods used for pre-processing images of *S. cerevisiae* cells are present in the ImageJ menu commands and are available in ImageJ API. Among all, the *Subtract Background* received special attention due to the wide range of options and features that allow a visual enhancement of the edges of the cells. The *Unsharp Mask* tool also allowed the cells edges brightening. The tools established for the pre-processing optimization of cell cultures images were performed as follows:



	Area	Group	Image
1	12.1856	small cell	image1.tif
2	17.4592	cell	image1.tif
3	27.5456	cell	image1.tif
4	17.2288	cell	image1.tif
5	13.6448	small cell	image1.tif
6	20.2240	cell	image1.tif
7	31.4624	cell	image1.tif
8	16.2048	cell	image1.tif
9	9.4720	small cell	image1.tif
10	17.8432	cell	image1.tif
11	13.3120	small cell	image1.tif
12	19.2512	cell	image1.tif
13	10.6240	small cell	image1.tif
14	8.5248	small cell	image2.tif
15	15.7952	cell	image2.tif
16	24.3968	cell	image2.tif
17	11.0848	small cell	image2.tif
18	34.7136	bud	image2.tif
19	35.7632	bud	image2.tif
20	22.3488	cell	image2.tif

Figure 4.6. Results Table of the analysis of two images of *S. cerevisiae* cells, *image1* and *image2*. Each table row corresponds to the measurements of each cell present in the images. First column is the cell identifier. Second column is for the cells areas, in square micrometers. In column *Group* is the classification given by the plugin to each cell.

- *Subtract Background*;
- *Unsharp Mask*;
- *Split Channels*;
- *Enhance Contrast (saturated= 0.35)*;
- *Convert to 32 bit*.

Despite the small number of processes, it is important to notice that the *Split Channels* process (*Split Blue*, *Split Red*, *Split Green*) and considering that all are always tested simultaneously, there are 5040 possible combinations (factorial 7) of these methods. For this reason, the set of experiments was gradually programmed depending on the obtained results.

Both *Subtract Background* and *Unsharp Mask* commands were implemented in the code using the method *run (ImagePlus image, String commands, String options)* from the class *IJ* implemented in ImageJ´s source code, previously installed in *Eclipse*. The instance *image* from the class *ImagePlus* passed as parameter was referred to Figure 4.9. The *String commands* passed as parameters were "Subtract Background..." and "Unsharp Mask..." intuitively for the commands *Subtract Background* and *Unsharp Mask*. In the first case, the *String options* included

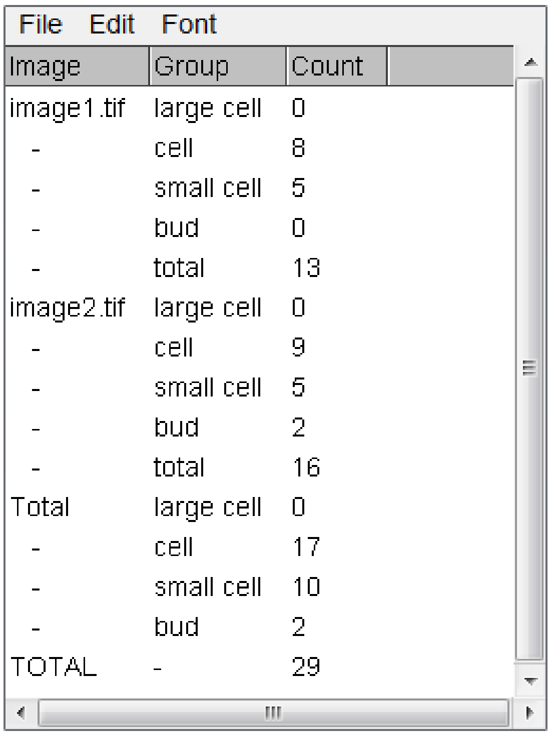


Image	Group	Count
image1.tif	large cell	0
-	cell	8
-	small cell	5
-	bud	0
-	total	13
image2.tif	large cell	0
-	cell	9
-	small cell	5
-	bud	2
-	total	16
Total	large cell	0
-	cell	17
-	small cell	10
-	bud	2
TOTAL	-	29

Figure 4.7. Summary Table of two images, image1 and image2, analyzed by the plugin ScerevisiCounter. For each image is discriminated the number of cells per group. The total number of cells per image and per group is also presented.

one of the following *String*s “”, “ sliding”, “ disable” and “ sliding disable”, preceded by the *String* “rolling =” plus *String rol*, where *rol* defined the *Subtract Background* rolling ball radius. In the second case, the *String options* was defined by the set of *String*s "radius =" plus *radius* plus "mask =" plus *mask*, where *radius* and *mask* were *String*s defining the radius and weight of the *Unsharp Mask* tool mask, respectively.

The command *32-bit* from the submenu Type... from ImageJ menu Image, was implemented using the class *Convert* from *IJ* API, passing as parameter an instance of the class *ImagePlus*. After creating an instance of this class, the method *convertToGray32 ()* allowed the conversion of the RGB image to a signed 32-bit grayscale image.

The conversion of RGB images into 8-bit grayscales images containing the blue, green and red components of the original, was programmed by creating a method *private ArrayList <ByteProcessor> splitChannels (ImagePlus image)*, with an *ImagePlus* instance as input parameter whose result was an array with the respective three *byteprocessors*. The method consisted in a cycle that run all the pixels of the RGB image and returned an array of bytes for

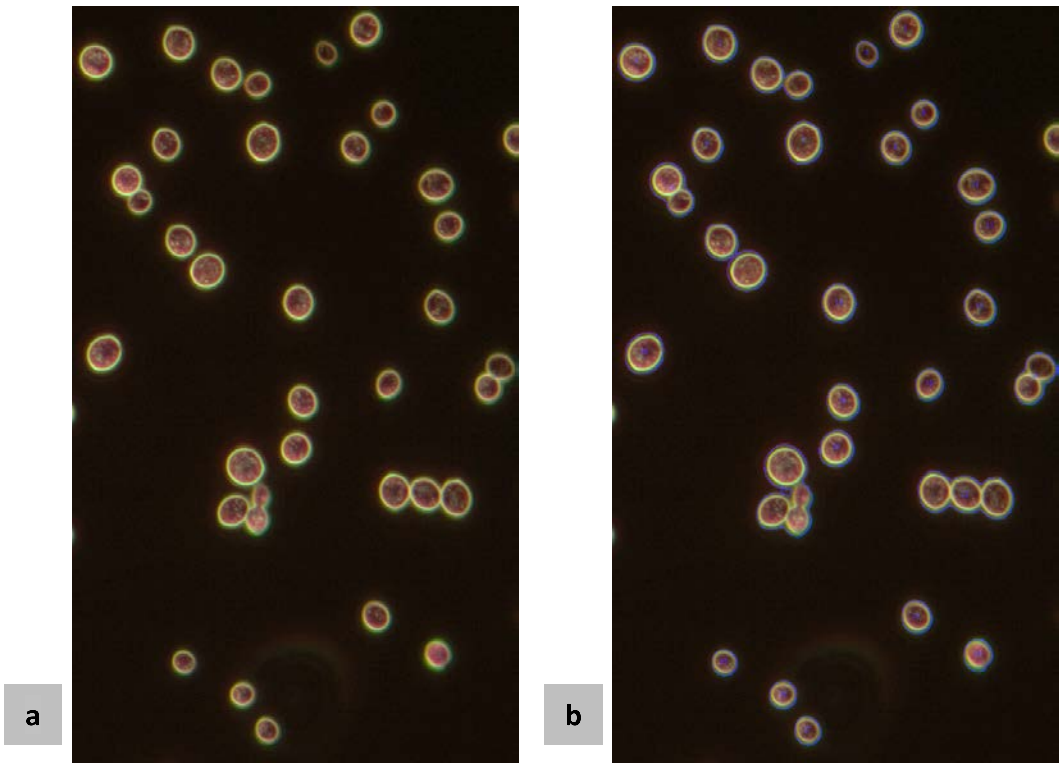


Figure 4.8. Processing an image of *S. cerevisiae* cells by a pipeline constituted by an auto threshold; followed by an analysis of particles larger than 200 pixels and circularity between 0 and 1; finished by the *Drawing Outlines* process. Figure 4.8b is a result of the described processing on the image of Figure 4.8a. As can be seen, a poor detection of the cells took place mainly in cells with touching edges.

the corresponding values of red, green and blue. Subsequently, the *bytePocessors* were built from these arrays with width and height equal to the original image.

Enhance Constrast was also implemented using the method *run ()* of the class *IJ*. The *String commands* was "Enhance Contrast" and the *String options* was "saturated = 0.35", since the percentage of pixels allowed to became saturated was 0.35%.

4.2.2. Image Process Optimization

All experiments were programmed in Java code and carried out using the IDE *Eclipse* SDK, as if it were ImageJ´s plugins. The Main Pipeline was slightly modified so that the results would include an image Mask and a Results Table with the area of each detected particle. Coupled with the Results Table, it was also returned the percentage of the true area relative to the area, manually determined, of the mask of the Figure 4.10. This parameter was used for comparison with the total areas returned in each experiment. The percentage of **True Area** was calculated using the following equation:

$$True\ Area = \frac{Positive\ \ Pixels}{Manual\ \ Area} \times 100 \tag{4.1}$$

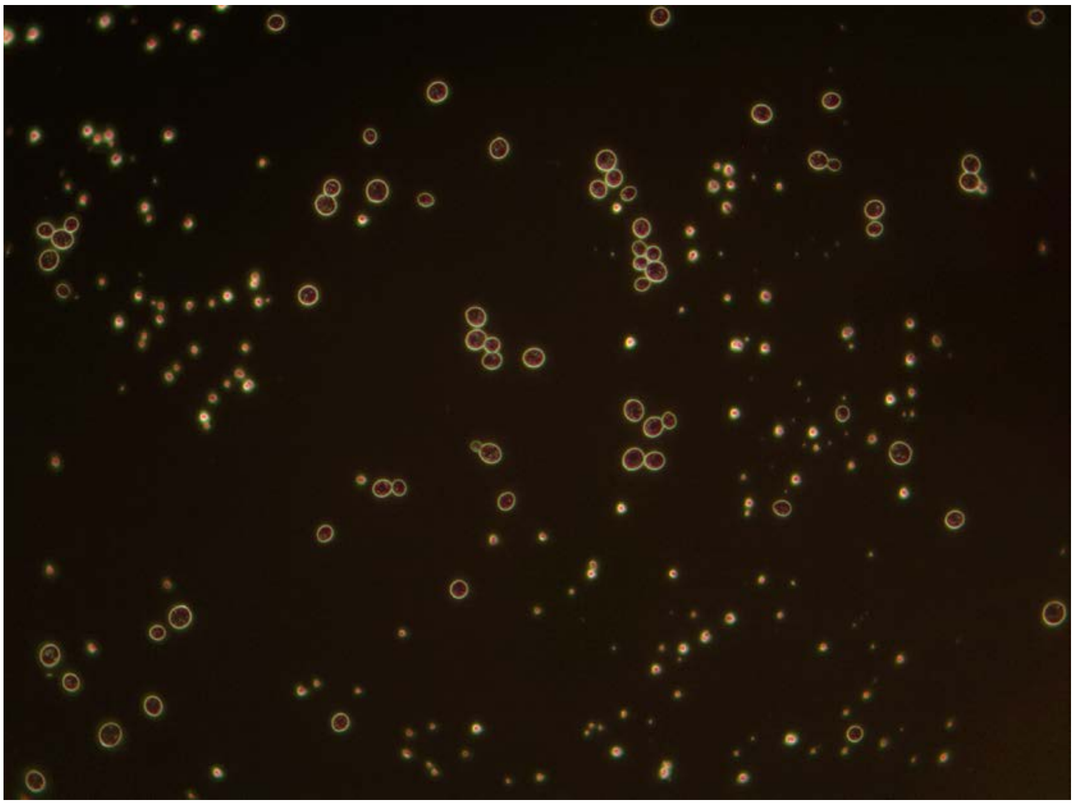


Figure 4.9. Image used for the methodology optimization experiments. It is a microscopic image of a culture of *S. cerevisiae* cells. It was taken with exposure time of 178.1 ms, 400x of magnification and dark field contrast

The **Positive Pixels** were the number of pixels with value equal to 255 (black pixels) of the resulting masks from the experiments, where the corresponding pixel in the Manual Mask must had the same value (black pixel as well). The **Manual Area** is the number of pixels with value equal to 255 in the mask manual determined (Figure 4.10).

Other data included the results were:

- **Negative Pixels** – difference between the number of black pixels of the resulting mask and the positive pixels ;
- **Black Pixels** – resulting mask pixels with value equal to 255;
- **White Pixels** – resulting mask pixels with value equal to 0.

Based on this methodology the user is able to determine the proportion of correct cells area (*True Area*), the detected area outside the borders of the area manually determined (**Extra Area**) and the percentage of resulting mask undetected area given by the *Manual Area* (**Missing Area**):

$$Extra\ Area = \frac{Negative\ \ Pixels}{Manual\ \ Area} \times 100 \tag{4.2}$$

$$Missing\ Area = \frac{Manual\ \ Area - Positive\ \ Pixels}{Manual\ \ Area} \times 100 \tag{4.3}$$

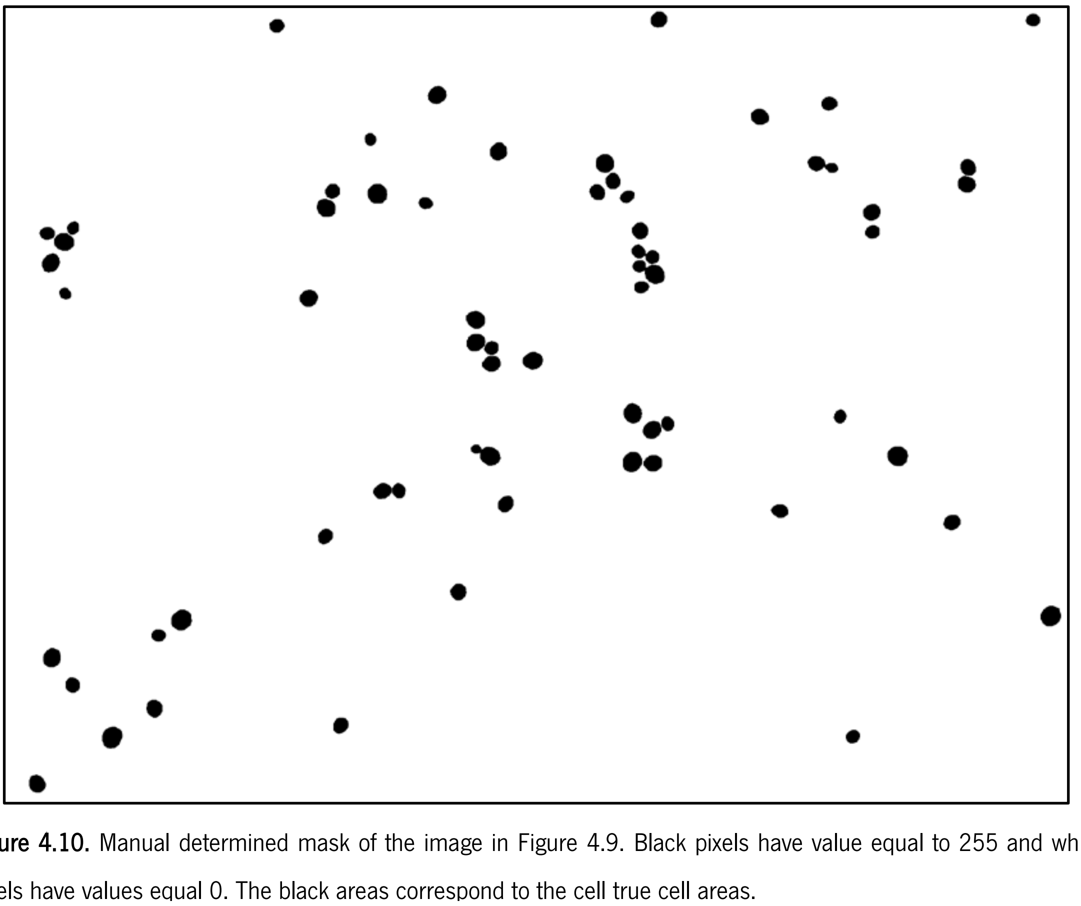


Figure 4.10. Manual determined mask of the image in Figure 4.9. Black pixels have value equal to 255 and white pixels have values equal 0. The black areas correspond to the cell true cell areas.

Thus, the Main Pipeline for the optimization methodologies was represented by the following algorithmic structure:

```
ManualMask – instance of the class ImagePlus to represent the mask in Figure 4.10
Path – String with the directory of the folder with the image in Figure 4.9
Title – String with name of the image in Figure 4.9
Image := instance of the class ImagePlus (Path, Title)
Methods (Image)
AutoThreshold (Image)
ParticleAnalyzer (Image)
Return Image Mask, Results Table
```

The exclusive difference of this algorithm from each experiment was solely based on the step *methods (image)* (Table 4.1).This step divided the experiments into two different groups: **Group 1** and **Group 2**. These two groups were selected based on the different options used in the tools *Subtract Background* and *Unsharp Mask*.

Thereby, in each experiment of the *Group 1* it was tested a *Subtract Background* rolling ball radius with values included in a range between 1 to 30 pixels. When the *Unsharp Mask* tool was used, the radius and weight of the mask were 5 pixels and 0.90, respectively. The configuration of each experiment (test) of this group is detailed in Table 4.2.

In *Group 2*, the *Unsharp Mask* tool had a radius with values between 1 and 20 pixels. For each value of radius tested, a range of mask weight between 0.1 and 0.9 was also tested. In the 6 experiments of this group, the radius of the rolling ball of the *Subtract Background* tool was 1, 2, 3, 10, 20 and 30 pixels, respectively.

Table 4.1. Constitution of the experiments based on the groups.

Method	*Group 1*	*Group 2*
1	*Subtract Background*	*Split Grenn*
2	*Convert to 32 bit*	*Subtract Background*
3	*Enhance Contrast*	*Unsharp Mask 1*
4	*Split Grenn*	-
5	*Split Red*	-
6	*Split Blue*	-
7	*Unsharp Mask*	-

Table 4.2. Configuration of *Group 1* experiments.

Order	*M_001*	*M_002*	*M_003*	*M_004*	*M_005*	*M_006*	*M_007*	*M_008*	*M_009*
1st	1	2	1	2	4	4	6	6	5
2nd	2	1	3	1	2	2	2	2	2
3rd			2	3	1	1	1	1	1
4th						3		3	
5th									

Table 4.2. Configuration of *Group 1* experiments (continuation).

Order	*M_010*	*M_011*	*M_012*	*M_013*	*M_014*	*M_015*	*M_016*	*M_017*	*M_018*
1st	5	3	4	4	4	4	3	4	4
2nd	2	4	3	1	1	3	4	7	7
3rd	1	1	1	3	3	1	1	1	3
4th	3				7	7	7	3	1
5th								7	7

Table 4.2. Configuration of *Group 1* experiments (continuation).

Order	*M_019*	*M_020*	*M_021*	*M_022*	*M_023*	*M_024*	*M_025*	*M_026*	*M_027*
1st	3	7	7	3	7	4	4	3	4
2nd	4	4	4	7	3	7	7	4	1
3rd	7	1	3	4	4	1	3	7	7
4th	1	3	1	1	1	3	1	1	
5th	7								

All the experiments of the two groups were performed in quadruplicate in order to test all possible combinations of *Sliding Paraboloid* and *Disable Smoothing* options of the *Subtract Background* tool. Only particles with areas greater than 300 pixels were detected by the *Particle Analyzer* in all experiments.

4.2.3. Refining methodologies

Refinement methodologies have been added to the Main Pipeline of the plugin ScerevisiCounter. These methodologies were implemented after the study of cells that were not properly processed. Figure 4.11 shows two cases where the correct detection is particularly difficult. After most of the test process methodologies were applied, the auto threshold resulted in a cell having an open edge (Figure 4.11b) generating a non-detection by the *Particle Analyzer*, and in a cell with a *septum* that should not exist (Figure 4.11e) as this bud cell will be considered by the *Analyzer* as two joined cells.

One way to solve this, at least for the image used in the optimization experiments (Figure 4.9), was to found the experiments that these cells were correctly detected and then incorporate those experiments into the Main Pipeline. However, this approach did not ensure an increased efficiency for others cells in other pictures, since the edges defining the cell varies from cell to cell. The image used in these experiments does not represent a global sample for all examples of cells. Furthermore, the inclusion of so many methodologies on the Main Pipeline would substantially increase the runtime of the plugin.

Another possible solution would be to find the optimal threshold level that would return the desired edges for each cell. However this approach has proved to be rather trivial since the threshold values greatly vary depending on the image, the cell, and the processing method applied.

Based on this hypothesis, a method that would apply at any point of the set of processing methods, a limited number of thresholds with different levels was created. Subsequently, the *Particle Analyzer* would be applied to each one of the resulting image of these limited number of thresholds and each *Analyzer* resulting mask superimposed in a final mask, using for that a

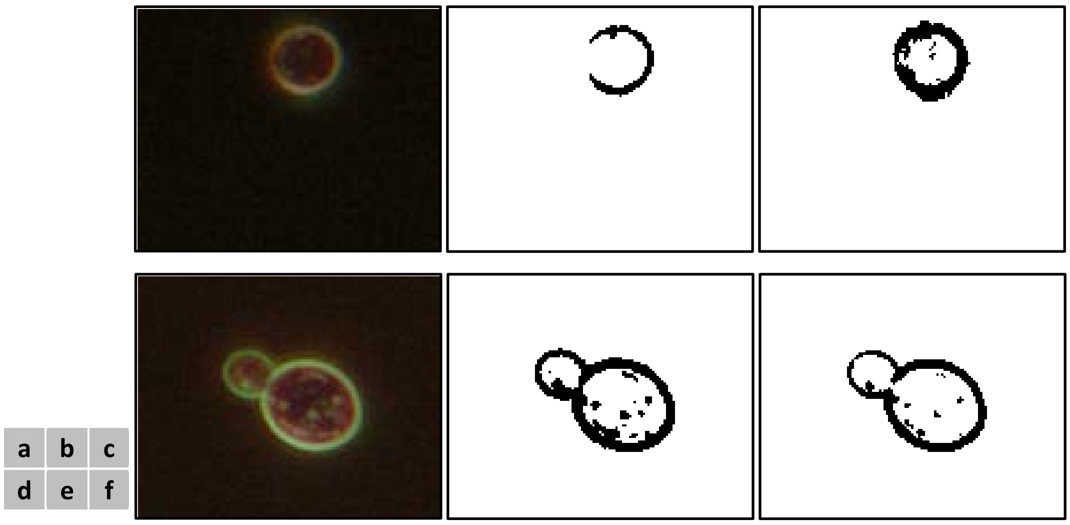


Figure 4.11. Effect of the method *thresholdRange* on two images of cells. Figures 4.11a and 4.11d are the ROI of the original image. Figures 4.11b and 4.11e are the resulting masks from the application of the Main Pipeline of the plugin ScerevisiCounter on the images of Figures 4.11a and 4.11d, respectively. Figures 4.11c and 4.11f are the resulting masks from the application of the Main Pipeline of the plugin ScerevisiCounter with the method *thresholdRange* included in it configuration.

method designed for this purpose with the class *ImageCalculator*. This method, `private ImagePlus thresholdRange (ImageProcessor imageProcessor, int tresAmplitude, int autoTres, int tresStep, String imageTitle)` has as input parameters, the integer value used to auto threshold the image (**autoTres**), the integer value (**tresStep**) defining the increment to the threshold level after each iteration of the loop with limits defined by the integer **tresAmplitude**. This cycle began with a threshold value equal to the difference between *autoTres* and *tresAmplitude* and ended when the threshold value was equal, and not greater, than the sum of the previous two variables, and in this way, the *tresAmplitude* and *tresStep* defined number of iterations of each cycle. If the difference or the sum of *autoTres* with *tresAmplitude* was less than 20 or greater than 250, the cycle limits would set to 20 and 250, respectively.

4.3. Debugging

The program test phase demonstrated the existence of peculiar situations. All occurrences were verified and revised, and all solutions are presented in this section.

The first action that the plugin ScerevisiCounter has after the user select it in the ImageJ Plugins menu command is the selection of the folder that contains the image or images to be analyzed. For this, a dialog box for folders and files browsing is opened. However, if the option to "Cancel" instead of "Select" is choose, a *java.lang.NullPointerException* at *java.io.File <init> (File.java: 251)* is executed. The *String path* created by the method *getDirectory* inherent to the instance of the class *DirectoryChooser* of *ij.io* API, is null with the cancellation of the operation, and the instantiation of the class *File* throws a *NullPointerException* because the *String pathname* is null. The solution for this problem was to test the condition of the *String pathname* to null before the instantiation of the class *File*, and return if it is true, stopping the plugin without any exception is thrown.

Once the directory of the folder with the images is selected and the pathname assigned to the *String path*, a list with the names of all files within that folder is created. To make the program without attaching all types of files to this list, it was necessary to verify if the files have extension TIFF, PNG, JPEG or RAW before adding their names to the list of *String*s. Moreover, if it were returned a null list, a dialog box with the following message is shown: "*there are no images with TIFF, PNG, RAW or JPEG extensions*".

It was also essential to verify the process before the *splitChannels* method was called. In this case, for the method not return one *java.lang.ClassCastException*, it was necessary to verify if the analyzed image was RGB color. Otherwise an error message is returned with the following message: "*RGB image required*".

Depending on the ImageJ version installed on the computer, by default, the "Dark background" option of the *Threshold* command can or cannot be selected. In the present case, a method was created to invert the image LUT depending on the ImageJ version, in order to the rest of the implemented code be compatible with the *Threshold* result.

Finally, due to the large number of images that the plugin can process, a statement showing that the plugin is actually working was introduced. This statement was implemented as a Progress Bar that appears in the ImageJ main window.

5. Results and discussion

The optimizing experiments (*Group 1* and *Group 2*) of the set of all methodologies presented in the plugin code was performed using only one image (N-Hefe_400x_dark Field_1 – *Image 1*) (Figure 4.9). Subsequently, the next set of images (Table 5.1 image 2 to 7) was also tested using the methodologies of the experiments of *Group 2*. Due to differences in the exposure time (between 300 and 1200 ms) during images acquisition, the final results were quite different. However, it was found that the processing methodologies were better applied to images with 400x total magnification and dark field contrast.

In the following sections all results leading to the development of the final version of the plugin ScerevisiCounter Main Pipeline are presented and discussed. First, the results of the optimization experiments for processing methodologies are shown. Next, the boundaries determination results and the classification of small, normal, large cells are discussed. The classification of cells along the classification of bud cells is also explained. After comparing manual and automatic counting and classification results, the final version of the plugin ScerevisiCounter is presented, with a more detailed explanation of the parameters and steps involved in the image processing and analyzing methods assembled to the plugin code.

The final version of the plugin ScerevisiCounter was achieved after analyzing and processing multiple microscopic images of the yeast *S. cerevisiae*. Table 5.1 shows all the images used to test the final version of the plugin.

5.1. Methodology optimization results

The optimization of the methods present in the processing methodology included in the ScerevisiCounter code was divided into two groups of experiments (*Group 1* and *Group 2*). The nomenclature used to identify each group of experiments was performed according to Table 4.2 for the experiments of *Group 1*. The configuration of all experiments was repeated four times, once for each combination of *Sliding Paraboloid* and *Disable Smoothing* options of the *Subtract Background* tool. To solve the naming problem, an identifier of each option was added to the experience title, in the form of an underscore (_) followed by the numeric identifier:

1 – No options;

2 – *Sliding Paraboloid* option;

Table 5.1. *S. cerevisiae* images title and receiving date.

Image	Number	Date received	Image	Number	Date received
N-Hefe_400x_dark-field_1	1	21-March	N Hefe 200ms	11	21-July
P hefe 800ms DF 400	2	20-May	N Hefe 250ms	12	21-July
P hefe 1200ms DF 400	3	20-May	Bild_227	13	21-August
N_Yeast 400x DF 300ms	4	16-June	Bild_229	14	21-August
N_Yeast 400x DF 500ms	5	16-June	Bild_281	15	21-August
N_Yeast 400x DF 700ms	6	16-June	Bild_289	16	21-August
N_Yeast 400x DF 1000ms	7	16-June	1B 1_100	17	27-August
N Hefe 100ms	8	21-July	1CV1 1_10	18	27-August
N Hefe 150ms	9	21-July	1N 1_100	19	27-August
N Hefe 200ms 2	10	21-July	1P 1_100	10	27-August

3 – *Disable Smoothing* option;

4 – *Sliding Paraboloid* plus *Disable Smoothing* options.

It should be noticed that all experiments results were comparable, despite the difference between the configurations and options defined. For the same image, each methodology present in each experiment generated the same result.

5.1.1. *Group 1* experiments – *Subtract Background* study

In *Group 1*, it was proposed to study the effect of varying the size of the rolling ball radius of the *Subtract Background* tool in different combinations of methods (Table 4.2). Table 5.2 shows the results of this set of experiments for the four possible combinations of *Sliding Paraboloid* and *Disable Smoothing* of the *Subtract Background* method. Using 30 values of rolling ball radius tested, the maximum area as well as its correspondent rolling ball radius were found. The area values presented in Table 5.2 are related to the *True Area* whenever was close to 100 %. The last column contains the area values obtained in each experiment without the *Subtract Background* method. The area *vs* rolling ball radius from *Group 1* experiments can be found in Appendix A (Figure A.1 – A.27).

In some cases, the resulting area values corresponded to *True Areas* far below 100, showing a large amount of pixels outside the manually defined area (see mask of Figure 4.10). For that reason, some caution is recommended for the analysis of the Figures presented in Appendix A and the areas presented in Table 5.2, since they are the maximum references in these experiences. For instance, in experience *M_007_2*, the maximum obtained area was equal to 840 pixels. However, as may be conferred by the mask in Figure 5.1, the experience *M_007_2* with *Subtract Background* rolling ball radius equal to 12 pixels, resulted in a mask with 235555 Black Pixels, being this area mostly outside to the manually determined red borders (borders of the mask in Figure 4.10). The mask in Figure 5.1 resulted from a set of methodologies configured in the following way: the blue channel image was converted to 32-bit, and the *Subtract Background* method was set with a rolling ball radius of 12 pixels with an active *Disable Smoothing* option. Then, the auto threshold was applied, and, as previously stated in Chapter 3, the threshold level chosen by the *AutoThreshold* is based on the distribution histogram of the pixel values for a 8, 16 or 32-bit grayscale image. The reason for the low *True Area* value was related to the automatic threshold applied after the processing methodology. Usually, for this image, the percentile of thresholded pixels without *Unsharp Mask* is higher than 90 %. In this

Table 5.2. Group 1 experiments maximum results (/pixels).

Test	*Area_1*	*Rolling Ball Radius_1*	*Area _2*	*Rolling Ball Radius_2*	*Area _3*	*Rolling Ball Radius_3*	*Area _4*	*Rolling Ball Radius_4*	Area
M_001	36751	3	35834	3	37136	6	35332	2	23676
M_002	36985	6	34925	5	39138	3	35405	1	23676
M_003	37497	4	35932	2	35932	5	35196	2	21152
M_004	36985	6	34925	5	39138	3	35405	1	23676
M_005	39822	7	38415	10	39059	6	38097	4	35604
M_006	39822	7	38415	10	39059	6	38097	4	35604
M_007	31046	11	840	29	34744	5	32310	1	0
M_008	31046	11	840	29	34744	5	32310	1	0
M_009	30364	16	30463	10	31488	6	28790	9	9521
M_010	30364	16	30463	10	31488	6	29578	12	9521
M_011	39305	1	38981	1	39217	4	37645	2	29530
M_012	39671	2	38748	3	38649	8	38260	4	35604
M_013	39671	2	38748	3	38568	9	38260	4	35604
M_014	42413	3	42433	2	41662	5	41134	1	38586
M_015	42413	3	42433	2	41662	5	41134	1	38586
M_016	41951	3	41145	1	40917	5	39636	1	37911
M_017	41494	3	40716	1	40881	3	40679	1	40648
M_018	41494	3	40716	1	40881	3	40679	1	40648
M_019	39131	6	38697	1	2172	2	0	Na	45884
M_020	41171	10	40575	2	41144	6	41188	3	38586
M_021	41171	10	40575	2	41144	6	41188	3	38586
M_022	38952	10	37869	1	38550	6	38990	3	37911
M_023	41171	10	40575	2	41144	6	41188	3	38586
M_024	41171	10	40575	2	41144	6	41188	3	38586
M_025	41171	10	40575	2	41144	6	41188	3	38586
M_026	38952	10	37869	1	38550	6	38990	3	37911
M_027	42413	3	42433	2	41662	5	41134	1	38586

case, the *AutoTheshold* defined a threshold level where less than 90 % of the pixels were thresholded.

Analyzing the results shown in Table 5.2 is concluded that it is necessary to include in the image pre-processing a linearization of the background. In all experiments, except in rare situations, where a substantial reduction of the detected area was verified, the introduction of the *Subtract Background* in the Main Pipeline causes an increase in the maximum area. In experience *M_019*, the area shown is acceptably higher than the maximum areas obtained in the same experiments with background subtraction. However, this area has a low *True Area* (84%), as demonstrated by the resulting mask (Figure 5.2), thus making invalid the comparison with the other results from the same experiment.

Analyzing the results of the experiments with background subtraction included in the pipeline (Table 5.2 and Figures A.1 – A.27), it is possible to verify the occurrence of 247 results with *True Areas* with percentage lower than 95% and results with undetected area (Table 5.3). Taking into account that the *True Area* value is strongly dependent on the total detected area, e.g., this value

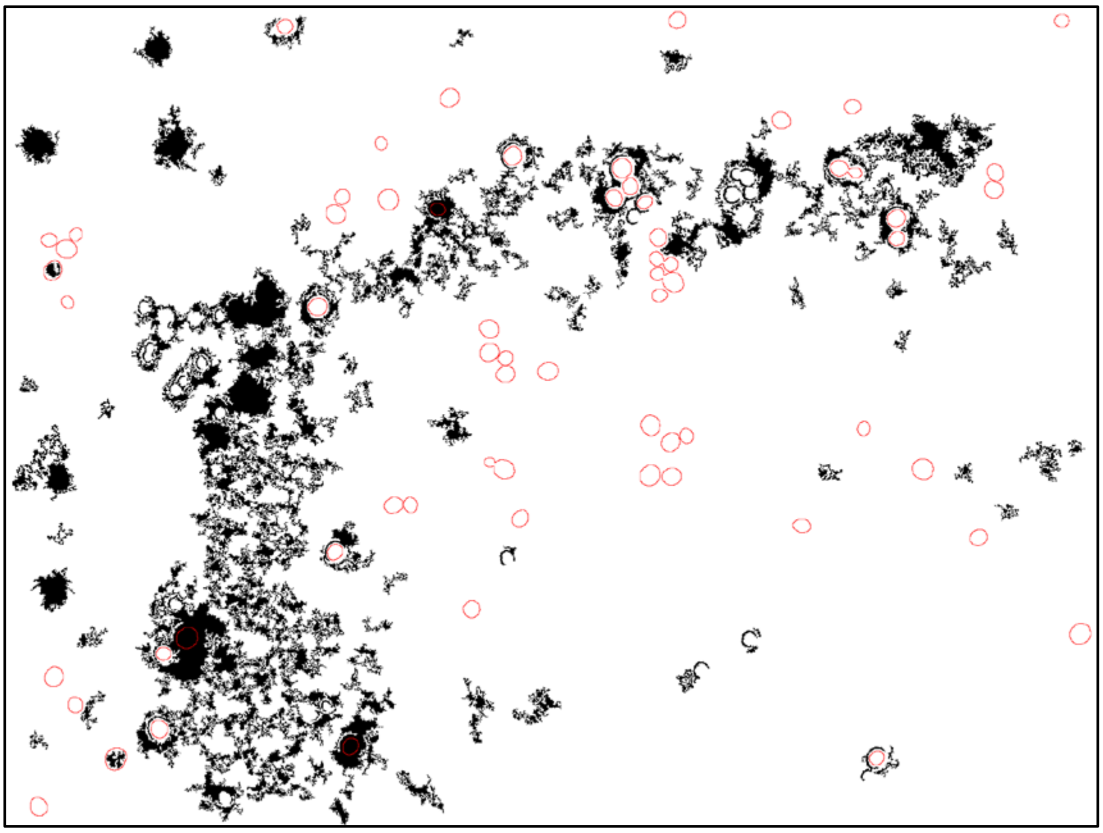


Figure 5.1. Experiment *M_007_2* resulting mask. The *Subtract Background* rolling ball radius was equal 12 pixels. The red borders are for the area manually determined.

is very sensitive to oscillations of the total detected area value, a visual analysis of all the obtained results was taken, in order to find situations of significant pixels areas outside the manually defined edges. In this way, it was stated that a *True Area* value below 95 percent is an optimal indicator for to identify the cases of false considerably large areas.

Of the 247 cases presented in Table 5.3, the experiments *M_005* to *M_008* with active *Sliding Paraboloid* option and the experiments *M_019_1* to *M_019_3* were the only cases where the below 95 % *True Areas* not resulted from experiments with rolling ball starting from 1 pixel to the value given in the respective cell in same table. In these cases, only one of the intermediate radius values, of the tested range of values starting at 1 pixel, did not resulted in a *True Area* with percentage less than 95 %. Supported in these observations, it can be said that regardless of the used methodologies, a greater *Subtract Background* rolling ball radius will result in a higher incidence of correct results, e.g., there will be more certainty in obtaining correct results when using greater rolling ball radius values.

However, the analysis of the graphs in Figures A.1 – A.27 shows a slight reduction in detected

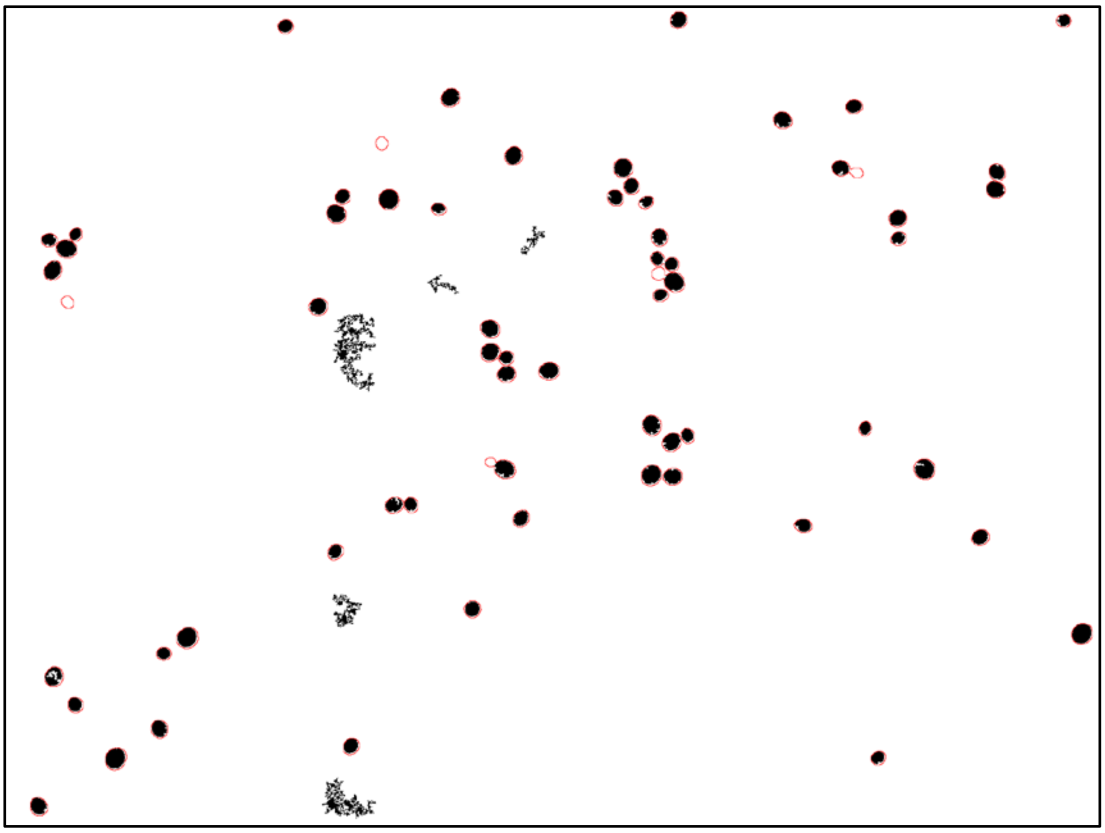


Figure 5.2. Experiment M_019, without background subtraction, resulting mask. The red borders are for the manually determined area.

area with the increase of radius. In experiments *M_001*, *M_002*, *M_003*, *M_004*, *M_005*, *M_006*, *M_014*, *M_015*, *M_016*, *M_026* and *M_027*, low radius values generated larger area values. In experiments *M_007*, *M_008*, *M_009* and *M_010* the maximum values of detected area were verified in tests with intermediate radius values. Above and below these values, a gradual decrease in results values was verified. In these cases, the red and blue channels were processed, indicating that the intermediate values of radius were ideal for detecting the correct maximum area. In experiments *M_011* and *M_025*, the radius variation did not cause significant fluctuations in the detected area, only noticing occasional cases of lower area for lower radius and some descended steps of area as the radius increases.

Some experiments, despite the difference in methodologies configuration presented equal results for the same rolling ball radius and *Sliding Paraboloid* and *Disable Smoothing* options of the

Table 5.3. Number of *True Areas* with percentage below 95% of detected area.

Test	*_1*	*_2*	*_3*	*_4*
M_001	0	0	0	0
M_002	5	4	0	0
M_003	0	0	0	0
M_004	5	4	0	0
M_005	5	6	0	0
M_006	5	6	0	0
M_007	10	26	0	0
M_008	10	26	0	0
M_009	3	5	0	0
M_010	3	5	0	0
M_011	0	0	0	0
M_012	0	0	0	0
M_013	0	0	0	0
M_014	0	0	0	0
M_015	0	0	0	0
M_016	0	0	0	0
M_017	0	0	1	0
M_018	0	0	1	0
M_019	20	29	29	30
M_020	0	0	1	0
M_021	0	0	1	0
M_022	1	0	1	0
M_023	0	0	1	0
M_024	0	0	1	0
M_025	0	0	1	0
M_026	1	0	1	0
M_027	0	0	0	0

Subtract Background method. The methods set in experiment *M_004* is the same in experiment *M_002* with the exception that the later one has the *Enhance Contrast (0.35%)* tool included in its methodologies pipeline. The same is applied to the experiments *M_005* and *M_006*; *M_007* and *M_008*; *M_009* and *M_010*; *M_012* and *M_013*; *M_014*, *M_015* and *M_027*; *M_017* and *M_018*; *M_020* and *M_021*; *M_023*, *M_024* and *M_025*. However, the results of these experiments are the same demonstrating that the contrast enhancement does not modify the final results. Despite this, experiments with equal methodology configurations, but with different *Enhance Contrast* positions had different final results. This was verified in experiments *M_001* and *M_003*; *M_011*, *M_012* and *M_013*; *M_015* and *M_016*; *M_017*, *M_018* and *M_019*; *M_022* and *M_023*; *M_024*, *M_025* and *M_026*, where the *Enhance Contrast* tool was situated in the first position of one of these experiment configurations. This showed that somehow this method changes the pixels values when being the first applied method. Another possibility is that when this method is applied to a grayscale image, the image is visually changed, but not the pixel values. This reason caused the same results in the experiments first described. Most of these experiments showing differences in the results, demonstrated that the use of contrast enhancement before the image type conversion from RGB to grayscale, causes a decrease in the detected area value.

The *Split Channel* tool divides the channels of an RGB image into three 8-bit grayscale images. The naming *Split Green*, *Split Blue* and *Split Red* were used when the resulting image of the *Split Channels* image was for the green, blue and red channel, respectively. The *Convert 32* method converts any image into a 32-bit grayscale image. The experiments *M_002* and *M_005* to *M_013* were set up to compare the use of *Convert 32* with each one of the *Split* methods. Besides these, it was also planned a few extra experiments to support the results of the main experiments. For a rolling ball radius equal to 10, the *M_005_1* to *M_005_4*, *M_007_1* to *M_007_4* and *M_009_1* to *M_009_4* experiments were repeated with and without *Convert 32* method. The extra experiments without *Convert 32* received an identical naming as the original experiments but with a "*32*" at the end of its name. The results of these experiments were also compared with the results of experiment *M_002* under the same conditions, thereby seeking to test the effect of the processing with each one of the referred methods with each separately and together. The detected area values are shown in Table 5.4. Taking off the experiment *M_007* in which were not obtained valid results for the two first cases of possible combinations of *Subtract Background* options, in general, there were not significant differences between the results for

Table 5.4. Results from different combinations of *Convert 32* and *Split Channels*.

Test	*_1*	*_2*	*_3*	*_4*
M_002	34586	34475	34560	34360
M_005	38259	38415	38180	37432
M_005_32	38704	37774	38313	37508
M_007	0	0	31091	31608
M_007_32	30261	29494	30330	31538
M_009	29152	30463	29756	28461
M_009_32	26823	28924	29715	28585

different combinations of *Disable Smoothing* and *Sliding Paraboloid* in the same experiments. Generally, it can be assumed that processing the green channel of an RGB image rather than the other channels, with or without *Convert 32*, resulted in larger areas. Through the use of the *Convert 32* together with *Split Channels* did not significantly change this trend, where the best results were obtained in the methodologies with the processing method *Split Green*. With the exception of using the *Sliding Paraboloid* option, the experiment *M_005_32* results were slightly better than the results of the experiment *M_005*. Despite the slight improvement of detected area, this was not reflected in an increase in detected cells, and only in a slightly increased area of the already detected cells. In this way and by Table 5.2 analysis, it can be seen that the largest areas were obtained in experiments *M_012* and *M_013*. Therefore, the methodology with *Split Green* without *Convert 32* was adopted for the following experiments.

Different positions of the *Unsharp Mask* tool in the pipeline methods were tested in experiments *M_014* to *M_027*. In the experiments *M_014* to *M_016* were tested different configurations (different position of the *Enhance Contrast* tool in the pipeline) with the *Unsharp Mask* method in the last position of pipeline methods, just after the *Subtract Background*. The experiments *M_017* to *M_019* had identical settings to the previous experiments, but with an additional *Unsharp Mask* immediately after the *Split Green*. In experiments *M_020* to *M_023*, the effect of an *Unsharp Mask* prior to the *Split Green* was tested. In the experiments *M_024* to *M_026* it was sought to determine the performance of the same settings, but with the *Unsharp Mask* after the *Split Green*.

Of the sets of experiments above described, removing the ones with *Enhance Contrast* in the beginning of its configuration, the values in Table 5.2 shows that the maximum results did not have significantly differences between them. Taking the lower and higher maximum results of this set of experiments, obtained in experience *M_020_2* (or *M_021_2*) and *M_027_2*, respectively, it is possible to confirm that the difference between these areas is 1858 pixels. As this difference is not calculated over the same pixel position in both masks, it is also necessary to take into account the number of detected cells, and both the differences between the masks obtained in these experiments. In the left mask of Figure 5.3, it is possible to confirm that the maximum detected area in experience *M_020_2*, and not in experience *M_027_2*, had an extra cell (420 extra pixels), while in the right mask of the same figure, the maximum detected area in experience *M_027_2*, and not in experience *M_020_2*, has more area and two extra cells (2268 extra pixels). With this, it can be stated that the use of an adequate rolling ball radius, a larger area will be detected when using an *Unsharp Mask* at the end of the methodologies configuration. However, the use of methodology which detects a larger area, does not guarantee the detection of cells which are detected by methodologies configurations that generate lower areas. Besides this, the inclusion of at least one *Unsharp Mask* method in the configuration is relevant.

The difference between the higher result of the experiments without *Unsharp Mask*, obtained in experience *M_005_1*, and the higher result of the experiments with *Unsharp Mask*, obtained in experience *M_027_2*, was 2611 pixels (see Table 5.2). As can be seen by analyzing Figure 5.4, the inclusion of this method leads to an improvement in the detected area as well as the detection of another cell. As can be seen, most of the difference found between the two methods results is verified when the inclusion of the *Unsharp Mask* method is taken.

From all the *Group 1* experiments maximum valid areas obtained, excepting the null verified on experience *M_019_4*, the minimums were 30364, 840, 2172 and 28790 pixels in the experiments *M_009_1* (*roI* = 16), *M_007_2* (*roI* = 29), *M_019_3* (*roI* = 2) and *M_009_4* (*roI* = 9), respectively. The overlapping of the masks, related to these results, yields a mask with a counting of 56 cells with a total area equal to 33494 pixels (Figure 5.5). The resulting mask of Figure 5.5 shows that the blue and red channels image processing from methodologies *M_007* and *M_009*, plus green channel processing with contrast enhancement at the configuration beginning and two *Unsharp Mask* methods from experience *M_019* generates masks far from the expectations. The combination of the "best" configuration of these experiments failed to detect 5 cells and 17289 pixels. Similarly, for different combinations of *Sliding Paraboloid* and *Disable Smoothing* options, the largest valid areas obtained were 42413, 42433, 41662 and 41188 pixels in the experiments *M_027_1* (*roI* = 3), *M_027_2* (*roI* = 2), *M_027_3* (*roI* = 5) and *M_023_4* (*roI* = 3), respectively. The addition of each of these masks resulted in the mask

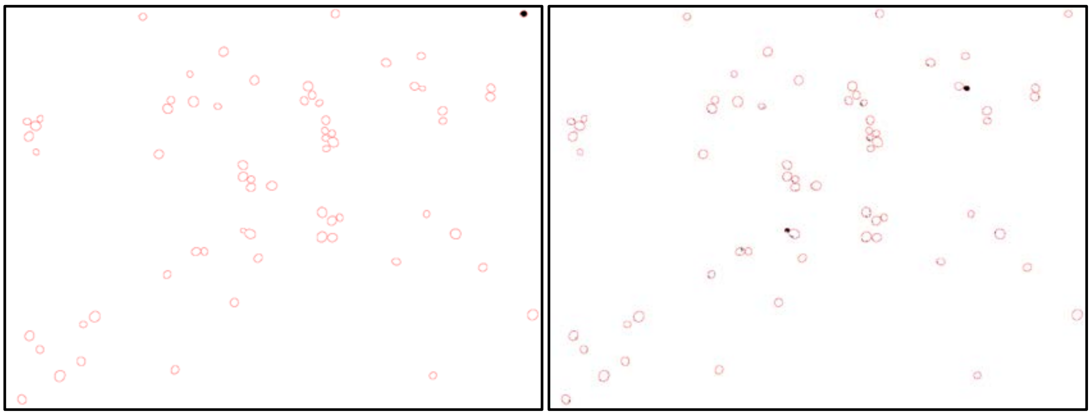


Figure 5.3. Left image is the resulting mask from the difference (420 pixels) between the masks of the higher results from *M_020_2* and *M_027_2* experiments, respectively. Right image is the resulting mask from the difference (2268 pixels) between the masks of the higher results from *M_027_2* and *M_020_2* experiments, respectively.

shown in Figure 5.6. This mask has an area of 43576 pixels and 64 cells. As can be seen, the area shown is almost entirely contained in the manually area defined for this picture, whose edges are drawn in red. The difference in the number of cells detected between the mask of the overlapping masks and the mask handily defined is due to the fact that the manual mask has more bud cells than the overlapping mask. This causes that one counted bud cell in the manual mask to be counted as two separated cells in the overlapping mask. Furthermore, the overlapping mask has a very good fit to the manual defined edges, where of the 43576 pixels of the overlapping mask, only 77 pixels outsides the red borders, and only 7154 pixels are missing for an adjustment 100% effective.

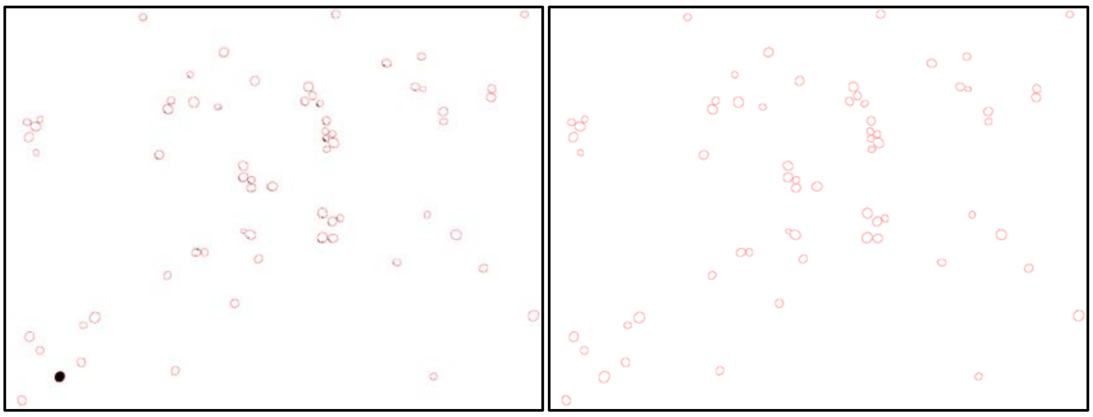


Figure 5.4. Left image is the resulting mask from the difference (2678 pixels) between the masks of the higher results from *M_027_2* and *M_005_1*, experiments, respectively. Right image is the resulting mask from the difference (54 pixels) between the masks of the higher results from *M_005_1* and *M_027_2*, experiments, respectively.

5.1.2. *Group 2* experiments – *Unsharp mask* study

Given the results obtained in the *Group 1* experiments, the *Group 2* experiments were programmed. Since the largest areas were obtained in experiment *M_027*, the *Group 2* experiments intended to verify the effect of the *Unsharp Mask* options in the results. Thereby for each possible combination of *Sliding Paraboloid* and *Disable Smoothing*, as well as to a rolling radius of the ball rolling radius of the *Subtract Background* method equal to 1, 2, 3, 10, 20 and 30 pixels, the experiments *M_028* to *M_033* were planned. In each one of them it was tested an *Unsharp Mask* radius mask between 1 and 20 pixels, and for each one of these values was tested each possible mask weights, between 0.1 and 0.9, totalizing 4320 experiments. Similarly to the results of *M_027*, the *Group 2* experiments did not generated areas with *True Area* less than 95%, which is an indicator that all tested options are secure to generate valid areas.

Figures B.1 – B.20, C.1 – C.20, D.1 – D.20, E.1 – E.20, F.1 – F.20 and G.1 – G.20 (Appendix B to G) have the graphics area (*I* pixels) *vs* mask weight for each tested mask radius and for each combination of *Sliding Paraboloid* and *Disable Smoothing*. In each graph is also present a line

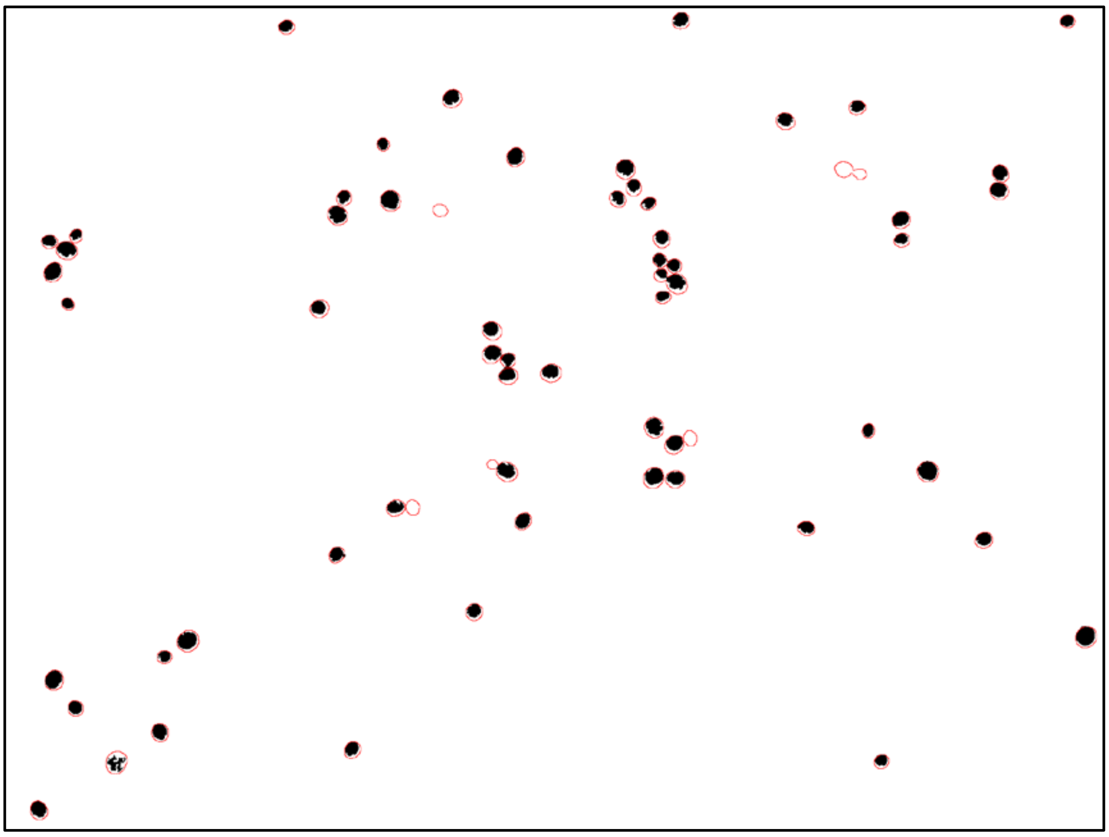


Figure 5.5. Result of the overlapping of the masks obtained in experiments *M_009_1* (*roI* = 16), *M_007_2* (*roI* = 29), *M_019_3* (*roI* = 2) and *M_009_4* (*roI* = 9). The red borders delimit are for the valid area defined manually.

representative of the area value for the respective experience without the *Unsharp Mask* method. Thus, in each *Group 2* experiment without the *Unsharp Mask* tool in its pipeline, is obtained a valid area (in pixels) equal to:

M_028_1 – 34876; *M_028_2* – 36863; *M_028_3* – 36507; *M_028_4* – 37714;
M_029_1 – 39671; *M_029_2* – 29236; *M_029_3* – 30631; *M_029_4* – 37224;
M_030_1 – 36626; *M_030_2* – 38748; *M_030_3* – 16981; *M_030_4* – 36704;
M_031_1 – 38704; *M_031_2* – 37774; *M_031_3* – 38313; *M_031_4* – 37508;
M_032_1 – 37507; *M_032_2* – 37084; *M_032_3* – 36512; *M_032_4* – 37190;
M_033_1 – 37200; *M_033_2* – 38000; *M_033_3* – 36159; *M_033_4* – 36374.

Because not all utilized mask radius and weight options generated a higher amount of valid area than the same experience without the *Unsharp Mask* tool, adding this line in the graphs allows identifying which settings generated a detected area increase. This increase, although not

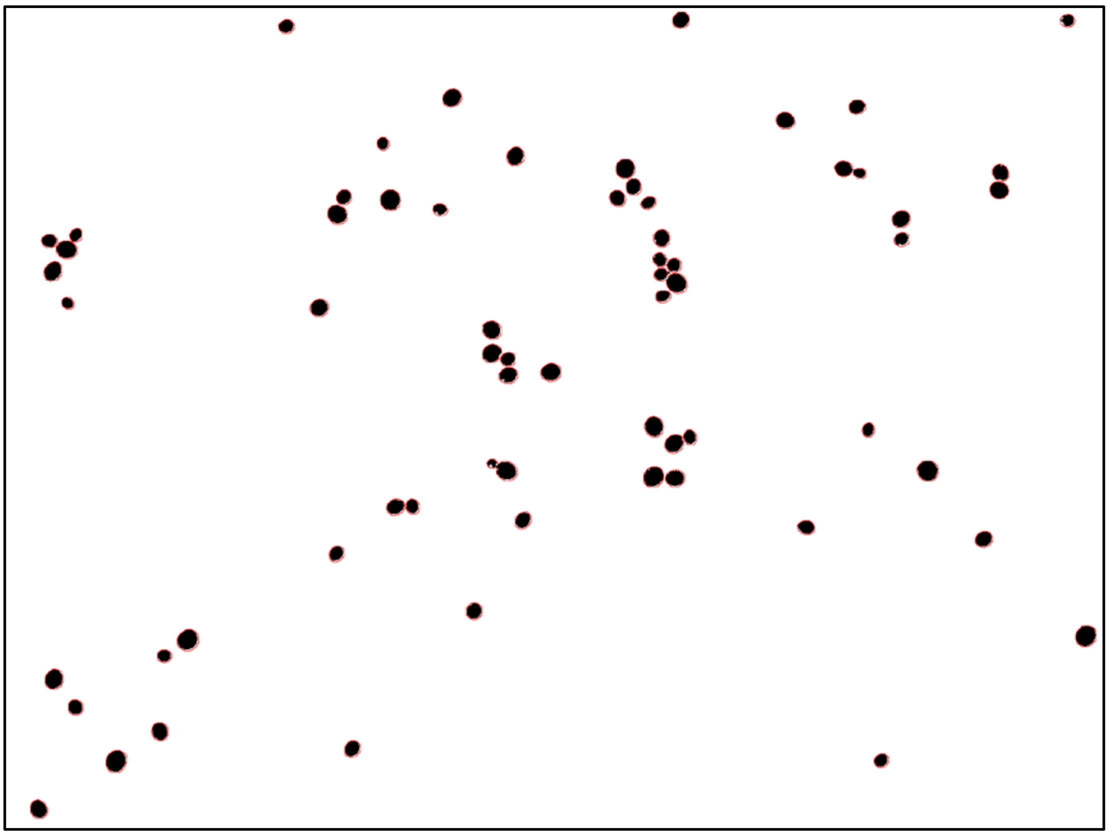


Figure 5.6. Result of the overlapping of the masks obtained in experiments *M_027_1* (*roi*=3), *M_027_2* (*roi*=2), *M_027_3* (*roi*=5) and *M_023_4* (*roi*=3). The red borders delimit are for the valid area defined manually.

significant in some cases, in others it can mean the difference between detect and not detect various cells. As an example, the left images on Figures 5.7 and 5.8 show the masks generated by the experiments *M_030_2* and *M_032_2*, respectively, for a mask radius and weight equal to 4 pixels and 0.9, and 7 pixels and 0.9, respectively. The right images on the same figures show the resulting masks of the same experiments without *Unsharp Mask*. Again, the borders of the *Manual Area* were drawn in red. These masks show that between using or not using *Unsharp Mask* had differences in the resulting areas equal to 3920 pixels in Figure 5.7 and 2754 pixels in Figure 5.8, meaning the detection failure of several cells. Despite this, the observed difference of 2830 pixels between the areas of these two experiments with *Unsharp Mask* (*M_032_2* and *M_030_2*) showed no differences between the total detected cells, only differences in which cells were detected and in the area "quality" of the detected cells.

Tables H.1 – H.24 (Appendix H) contains informative support on *Group 2* experiments results whose areas were larger than the areas obtained in the same experiments without *Unsharp Mask* (area without *Unsharp*). For each mask radius are shown which and how many weights generated area values above area without *Unsharp* (columns *Mask weights* and *Total*, respectively), the average value of the areas above area without Unsharp (column *Area mean*) and the difference between this average and the area without *Unsharp* (column *Difference*), the value of the largest area obtained (column *Max area*) and the respective mask weight used in the experience which generated the largest area (column *Mask weight*).

The values gathered in Table 5.5 intended to make a support to information contained in tables and figures of Appendices from B to H. The total number (*n*) and frequency (*f*) of mask weights

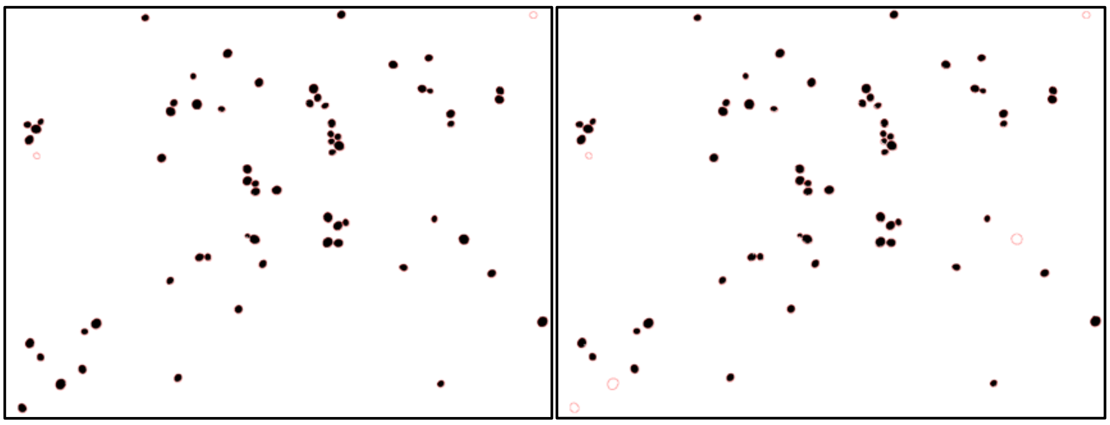


Figure 5.7. Left image is the mask resulting from experiment *M_030_2* with a mask radius and weight equal to 4 pixels and 0.9, respectively (42668 pixels). Right image is the mask resulting from *M_030_2* experiment without the *Unsharp Mask* tool (38748 pixels).

that generated values of area above the area without *Unsharp* were included, for each used mask radius. As can be seen, the experiments with no active *Sliding Paraboloid* and *Disable Smoothing* options (*_1*) were those that, less frequently generated areas with values above the value of the area without *Unsharp*. Of the 1080 experiments, the results were positive in only 452 (41.85 %). This low efficiency, for a methodology with an *Unsharp Mask* method in its configuration, is mainly due to experiments *M_028_1* and *M_029_1*, where only seven results of these experiments were positive. To a mask weight equal to 0.1 and a mask radius equal to 1 and 5 to 9 in test *M_028_1*, the resulted area were increased only 290, 62, 17, 5, 1 and 1 pixels, while for a mask weight equal to 0.1 and a mask radius equal to 1 in test *M_029_1*, the resulted area were increased only 266 pixels (Table H.1 and H.5). The low efficiency, caused by the results of experiments *M_028_1* and *M_029_1*, is balanced with the increase in the rolling ball radius value of the *Subtract Background* method. For each mask radius in experiments *M_032_1* and *M_033_1*, average values of 7.5 were verified for the number of mask weights that generated areas above the area without *Unsharp*. As can be seen in the graph of Figure 5.9, only experiments *M_030_1* to *M_033_1* had acceptable average area values above the area without *Unsharp*. For radius mask values above 2 pixels, these four experiments showed reasonable means, presenting a small decrease in the average quality for radius mask highest values in experiments with *Subtract Background* rolling ball radius equal 20 and 30 pixels. Finally, it can be stated that most of the largest areas observed in these experiments number *_1*, were generated in a configuration with an *Unsharp Mask* with a mask weight equal to 0.9. Nevertheless, the increase of radius mask value decreases the bests mask weight values to

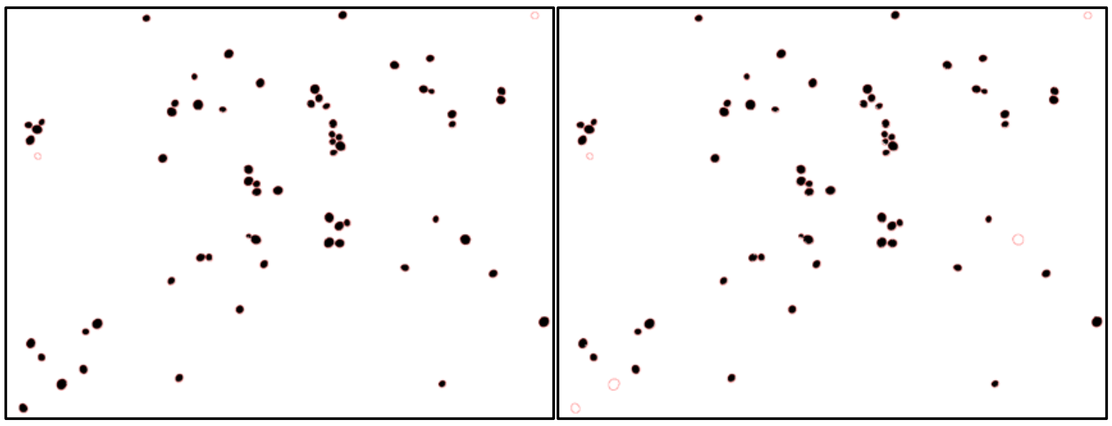


Figure 5.8. Left image is the mask resulting from experiment *M_032_2* with a mask radius and weight equal to 7 pixels 0.9, respectively (39838 pixels Right image is the mask resulting from *M_032_2*, experiment without the *Unsharp Mask* tool (37084 pixels).

Table 5.5. Frequencies of mask weight values that generate an increase in the detected area.

Radius	*_1*		*_2*		*_3*		*_4*		Total	
pixels	*n*	*f*	*n*	*f*	*n*	*f*	*n*	*f*	*n*	*f*
1	23	0.43	37	0.69	40	0.74	40	0.74	140	64.81
2	25	0.46	44	0.81	42	0.78	47	0.87	158	73.15
3	26	0.48	44	0.81	44	0.81	47	0.87	161	74.54
4	26	0.48	44	0.81	45	0.83	49	0.91	164	75.93
5	29	0.54	45	0.83	45	0.83	48	0.89	167	77.31
6	30	0.56	44	0.81	40	0.74	48	0.89	162	75.00
7	27	0.50	44	0.81	40	0.74	48	0.89	159	73.61
8	24	0.44	45	0.83	38	0.70	45	0.83	152	70.37
9	28	0.52	44	0.81	37	0.69	44	0.81	153	70.83
10	24	0.44	45	0.83	38	0.70	48	0.89	155	71.76
11	23	0.43	45	0.83	37	0.69	43	0.80	148	68.52
12	21	0.39	44	0.81	37	0.69	42	0.78	144	66.67
13	22	0.41	43	0.80	37	0.69	44	0.81	146	67.59
14	18	0.33	40	0.74	36	0.67	40	0.74	134	62.04
15	17	0.31	40	0.74	32	0.59	39	0.72	128	59.26
16	17	0.31	41	0.76	32	0.59	40	0.74	130	60.19
17	17	0.31	39	0.72	32	0.59	39	0.72	127	58.80
18	18	0.33	38	0.70	32	0.59	40	0.74	128	59.26
19	19	0.35	38	0.70	33	0.61	39	0.72	129	59.72
20	18	0.33	37	0.69	33	0.61	39	0.72	127	58.80
Total	452	0.42	841	0.78	750	0.69	869	0.80	–	–

between 0.6 and 0.8 (Tables H.1, H.5, H.9, H.13, H.17 and H.21).

In experiments *_2*, the *Sliding Paraboloid* option together with the *Unsharp Mas*k method worked as a catalyst to increase the detected area. Of the 1080 experiments, 841 (77.87 %) resulted in an increase in the detected area, regarding the detected area of the same methodology without the unsharp filter. For any tested rolling ball radius of the *Subtract Background*, were high values of detected area for mask radius between 2 and 12 pixels (Table 5.5). However, analyzing Tables H.2, H.6, H.10, H.14, H.18 and H.22, it is verified that in all experiments, except in *M_028_2* and *M_030_2*, there were an increase of all detected area for 90 % of all used mask weight values. The decrease of this frequency to 80 %, is practically due to the low profit occurred in experiments *M_028_2* and *M_030_2* wherein for any tested mask radius mask, only 50 % of the mask weight values were responsible for an increase of the detected area. The maximum areas observed in these experiments are significantly higher than those observed in experiments *_1*, but like these first, the weights that generate the maximum areas were mainly equal to 0.9. Despite the low averages observed in experience *M_029_2*, especially for low mask radius values, this experience was the one that generated greater increases in detected areas (Table

H.6). This fact is due to the very low area detected in the same experiment without unsharp (29236 pixels), but yet the maximum area values were obtained within the range of the other experiments, for mask radius values mainly between 4 and 9 pixels.

The performance of the experiments with the *Disable Smoothing* option of the *Subtract Background* active, regarding the frequency of detected area increasing face the detected area in the same methodologies without unsharp filter, as in the experiments *_2*, were higher comparing to experiments *_1*. However, comparing with the performance of experiments with an active *Sliding Paraboloid* option, these experiments had a lower rate of area increase. This frequency reduction is observed mainly for higher mask radius values, because for the first five mask radius, these two sets of experiments, *_2* and *_3*, had similar frequencies. Furthermore, in experiments *M_028_3*, *M_029_3* and *M_031_3*, the number of mask weights which resulted in an increase in the detected areas, similarly to experiments *_1*, was greatly reduced. Opposing this, the experiment *M_030_3* (rolling ball radius equal to 3 pixels) had an increase of detected face area face the area without *Unsharp*, for any used mask radius and weight (Tables H.3, H.7, H.11, H.15, H.19 and H.23). In this experiment the differences between the average area and the area without *Unsharp* were larger, reaching values of 19000 pixels, mainly, because the value of the detected area without *Unsharp* is very low (16981 pixels). Despite the increase of these large values of detected area, the averages were the lowest among the other experiment *_3*, for any mask radius tested (Figure 5.11). In these experiments, the highest average and the highest maximum areas were obtained in the three experiments with larger radius of rolling ball of the *Subtract Background*. These maximum values were observed for a mask radius equal to 2 – 4 pixels and for high mask weights, 0.8 and 0.9.

The highest frequency of mask weight values, which generated largest areas than the area without *Unsharp*, were observed in the experiments with the two options, *Sliding Paraboloid* and *Disable Smoothing*, active. For any tested mask radius, were more mask weight values that generated an increase in the detected area than in the other *Group 2* experiments (Table 5.5) area. Apart from experiment *M_028_4*, for any value of used mask radius in other experiments there was a profit in 7 to 9, of the 9 possible mask weight values (Tables H.4, H.8, H.12, H.16, H.20 and H.24). As can be seen in the graph of Figure 5.12, areas larger than the respective area without *Unsharp* were detected in these experiments for any value of radius mask. In all experiments, the mean areas were consistently higher, noting a slight decrease with the increasing of the radius mask value. Once again, mask radius values between 2 and 7 pixels

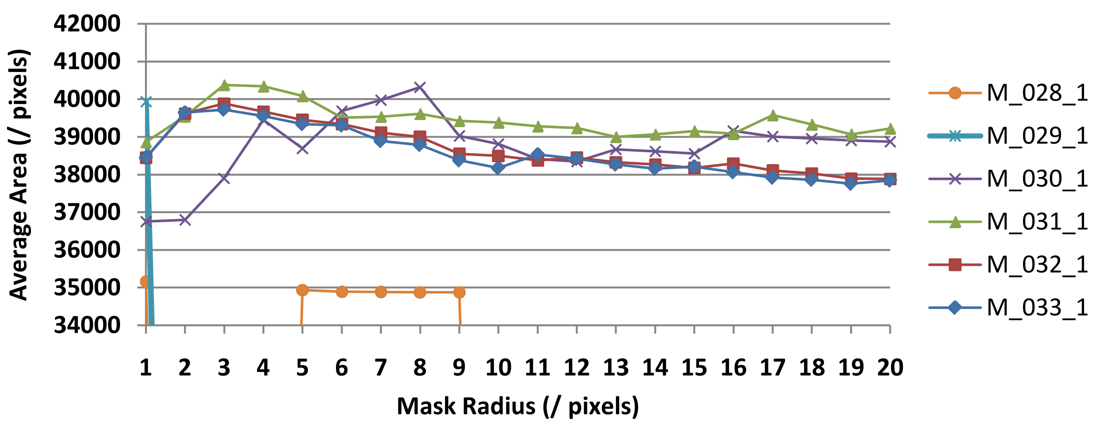


Figure 5.9. Average area values (/ pixels), of the areas higher than the area resulted from the same experiment without unsharp filter, for each mask radius (/ pixels) in the experiments _1.

resulted in higher averages, while higher maximum areas were obtained for any of the tested rolling ball radius. Likewise, mask weights between 0.7 and 0.9 resulted in largest detected areas.

In Table 5.6, the maximum area values obtained in each *Group 2* experiment are presented. For each maximum area it is indicated the mask radius and weight of used in the *Unsharp Mask* tool options. Excepting the experiments *M_028_1*, *M029_1*, *M_028_3*, *M_29_3 M_030_3*, all areas have maximum values above 42000 pixels. Mask radius and weight values equal to 2 – 5 pixels and 0.9, respectively, were used in these experiments. The experiments with higher rolling ball radius were those that resulted in larger maximum area values. In particular, the experience *M_031* (*rol* = 10 pixels) yielded very positive results for any combination of the options *Sliding Paraboloid* and *Disable Smoothing*.

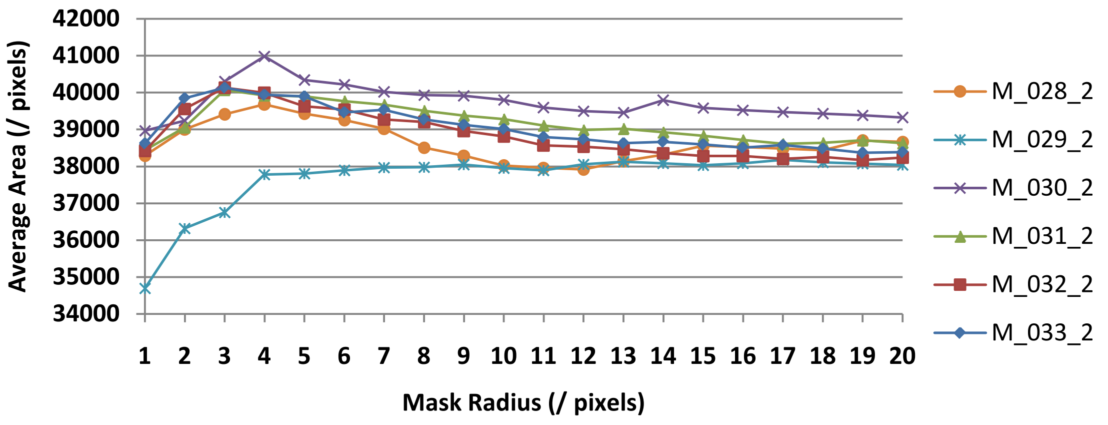


Figure 5.10. Average area values (/ pixels), of the areas higher than the area resulted from the same experiment without unsharp filter, for each mask radius (/ pixels) in the experiments _2.

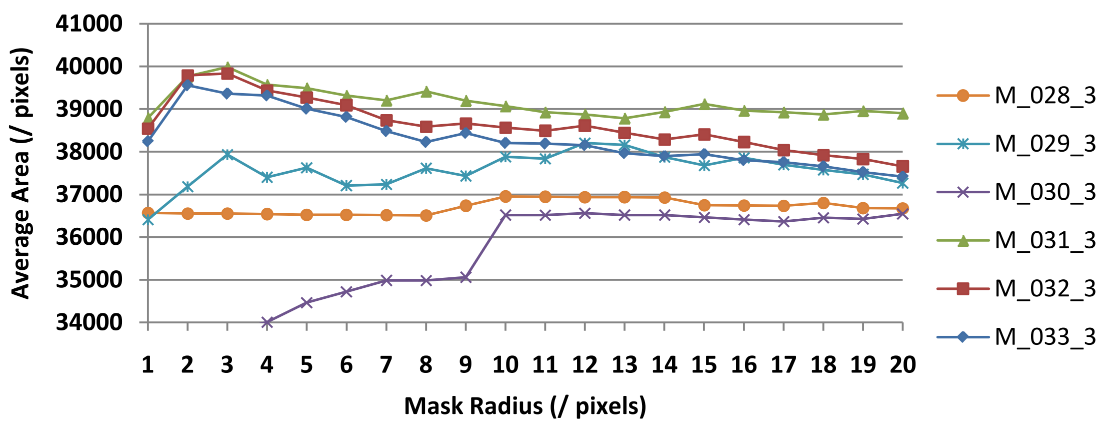


Figure 5.11. Average area values (/ pixels), of the areas higher than the area resulted from the same experiment without unsharp filter, for each mask radius (/ pixels) in the experiments _3.

5.1.3. Other images *Group 2* experiments

As previously mentioned, *Group 1* and *Group 2* experiments were performed using image N-Hefe_400x_dark-Field_1 (*Image 1*). This image captured with an exposure time near to 200 ms contains *S. cerevisiae* cells magnified 400x over a dark field background. For this image, it was determined that the methodologies which yield larger areas and a greater number of cells were represented by the experiments *M_031_1* and *M_031_2*, with mask radius and weight equal to 3 pixels and 0.9, respectively. Two other tests that resulted in considerable large detected areas were the *M_031_3* and *M_030_4* experiments, both with a masks radius and a weight equal to 2 pixels and 0.9, respectively. Nevertheless, as can be seen in Figure 5.13, each one of these tests did not detect all the cells present at the respective original picture. The masks resulting

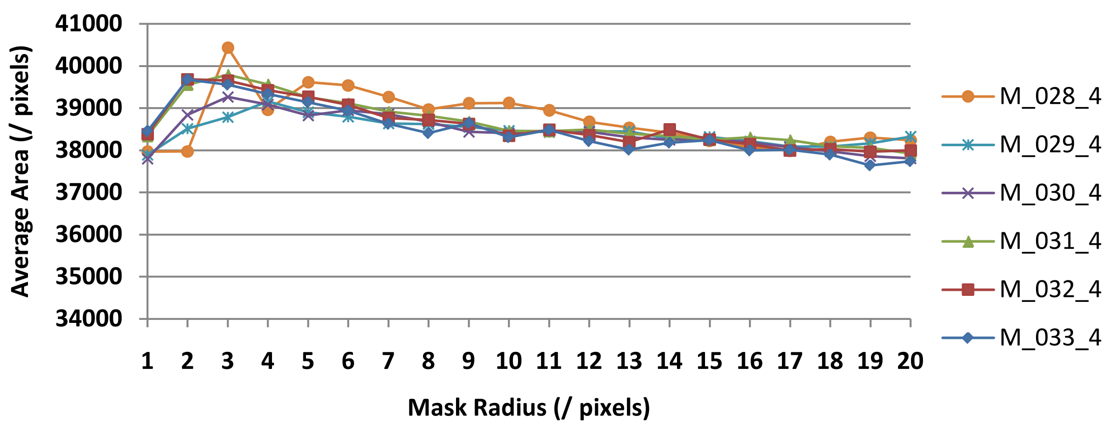


Figure 5.12. Average area values (/ pixels), of the areas higher than the area resulted from the same experiment without unsharp filter, for each mask radius (/ pixels) in the experiments _4.

Table 5.6. *Group 2* experiments maximum areas.

Test	Max area	Mask radius	Mask weight
	pixels	pixels	
M_028_1	35166	1	0.1
M_028_2	42759	4	0.9
M_028_3	36951	9	0.3
M_028_4	42790	3	0.9
M_029_1	39937	1	0.1
M_029_2	42433	5	0.9
M_029_3	41603	3	0.4
M_029_4	42771	2	0.9
M_030_1	42413	5	0.9
M_030_2	42668	4	0.9
M_030_3	41376	3	0.9
M_030_4	42931	2	0.9
M_031_1	43018	3	0.9
M_031_2	43021	3	0.9
M_031_3	42760	2	0.9
M_031_4	42766	2	0.9
M_032_1	42603	3	0.9
M_032_2	42716	3	0.9
M_032_3	42554	2	0.9
M_032_4	42627	2	0.9
M_033_1	42524	3	0.9
M_033_2	42671	3	0.9
M_033_3	42522	2	0.9
M_033_4	42619	2	0.9

from these tests shown that, missing cells in one test were detected in another test, and *vice versa*. However, the overlapping of the masks of these tests resulted in a mask (Figure 5.14) in which all cells of the original image are present.

It should be mentioned that the image processing methods may not result in an analysis 100 % effective, where always some cells will not be detected. The achievement of each possible processing methodology for an image takes time, being impracticable the realization of all different processing methodologies on an image, and subsequent joining of the resulting masks. In order to get a higher precision in which methods result in larger areas and greater number of detected cells, the *Group 2* experiments were carried out in six more images of yeast cultures; *Image 2* to *Image 7* (Figure I.2 – I.7 in Appendix I). Those images have the same characteristics of N-Hefe_400x_dark-field_1. The main difference is at the brightness level, which is caused by different exposure times that the camera took to capture the images. Indeed the first two images are identical but with different contrasts caused by different exposure times. The same is true for the following four images. In these images, all cells except a few, are in the same positions only differing its brightness feature.

The results for the maximum areas observed in all the experiments of the *Group 2* for the new images are presented in Tables 5.7 – 5.9. Due to the greater number of cells as well as the contrast presented by the cells for different exposure times, the maximum results show a much larger variability than the results for the first image. Therefore, is more reliable to choose the methods that return better results for this group of images. A quick analysis at these pictures maximum results, it appears that most of them were achieved in experiments with rolling ball radius equal to 10, 20 and 30 pixels. It can also be noted that the ideal mask weight was equal to 0.9 for low mask radius values, reinforcing the observations made in the experiments with the first image.

For *Image 2*, the methodologies that resulted in larger areas were the present in the configuration of experiments *M_031_2*, for a mask radius and weight equal to 3 pixels and 0.9,

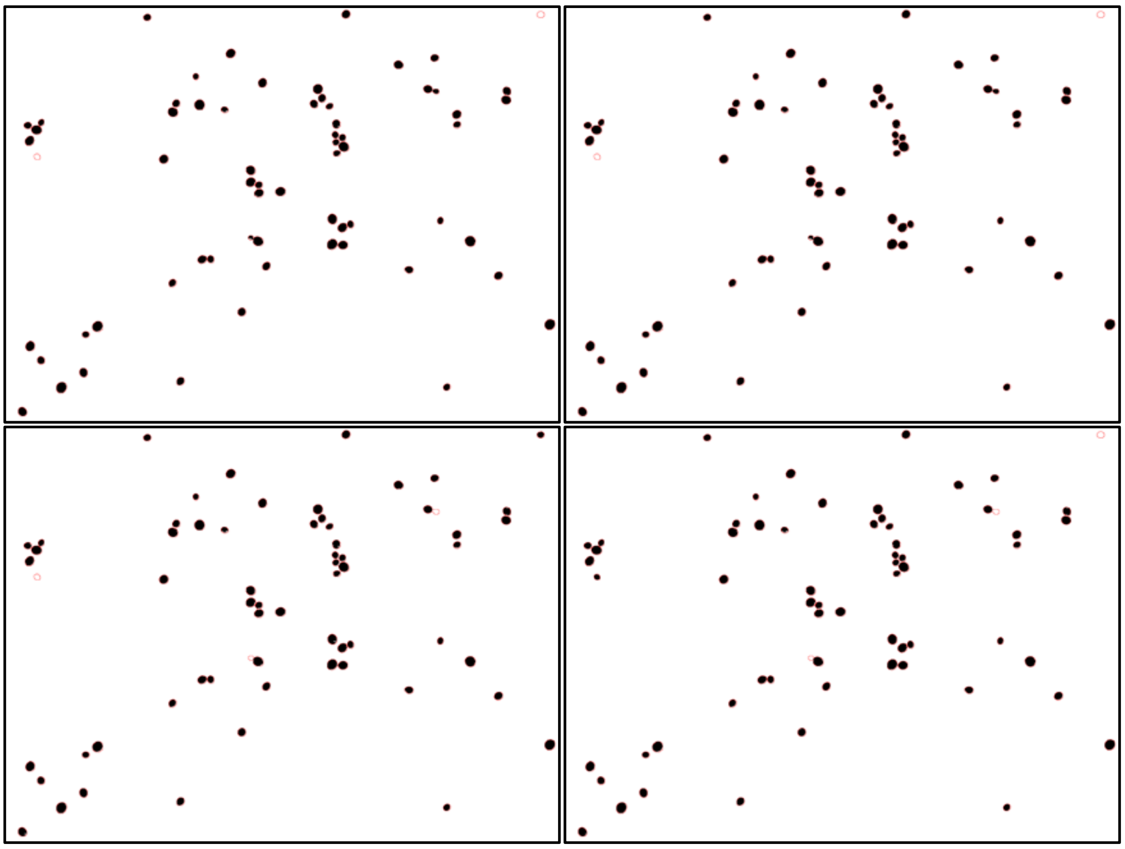


Figure 5.13. Resulting masks from experiments *M_031_1* and *M_031_2*, both with radius mask and weight equal to 3 pixels and 0.9, respectively (top left to right), and resulting masks from experiments *M_031_3* and *M_030_4*, both with radius mask and weight equal to 2 pixels and 0.9, respectively (bottom left to right).

respectively, and, *M_031_4*, *M_032_1* and *M_032_3* for a mask radius and weight equal to 2 pixels and 0.9, respectively (Table 5.7). The sum of the masks obtained in these tests results in the mask of Figure J.1 (Appendix J), whose area is equal to 150073 pixels and the total number of detected cells is 220. However if adding the masks obtained in the methodologies with better performances for the *Image 1*, the experiments *M_031_1* and *M_031_2*, for a mask radius and weight equal to 3 pixels and 0.9, respectively, and the experiments *M_031_3* and *M_030_4*, for a mask radius and weight equal to 2 pixels and 0.9, respectively, it is generated a mask (Figure J.2 - Appendix J) whose combined area of 155457 pixels is for 226 particles detected. Yet, analyzing with detail the masks of Figures J.1 and J.2 it is possible to see that some cells still not detected, and cells present in one mask and missing in another mask, and *vice versa*. Thereby, the mask of Figure J.3 (Appendix J) was the result of the overlapping of the mask of Figures J.1 and J.3, and have a total area of 157241 pixels and 228 detected cells.

Analyzing the tests that resulted in larger areas for the *Image 3*, it is possible to verify that they had lower values of rolling ball radius than in the previous cases. This may be due to the higher

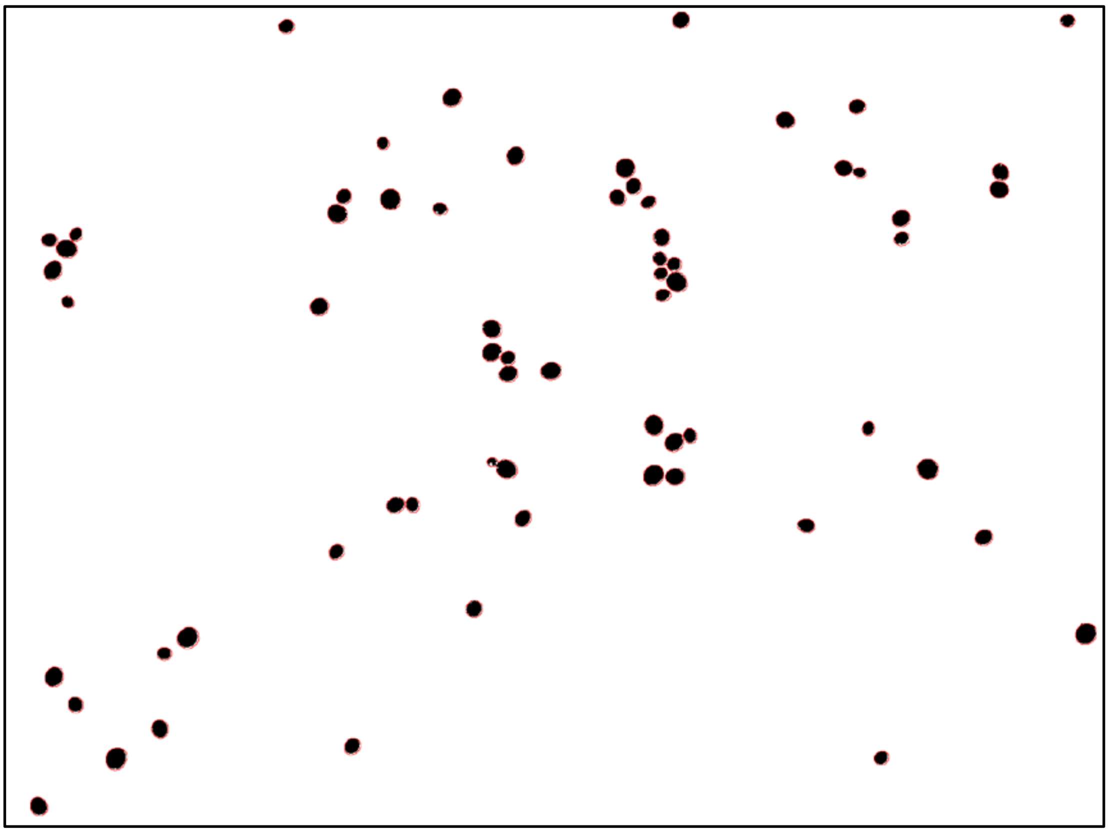


Figure 5.14. Resulting mask from the overlapping of the resulting masks from experiments *M_031_1* and *M_031_2*, both with radius mask and weight equal to 3 pixels and 0.9, respectively, and the resulting masks from experiments *M_031_3* and *M_030_4*, both with radius mask and weight equal to 2 pixels and 0.9, respectively.

Table 5.7. *Image 2* and *Image 3* maximum areas.

	P hefe 800ms DF 400			P hefe 1200ms DF 400		
Test	Max area	Mask radius	Mask weight	Max area	Mask radius	Mask weight
	pixels	pixels		pixels	pixels	
M_028_1	108547	1	0.10	118526	16	0.90
M_028_2	124051	5	0.90	146969	3	0.90
M_028_3	61238	1	0.10	96847	1	0.10
M_028_4	129149	5	0.90	147368	2	0.90
M_029_1	97787	20	0.90	136184	16	0.90
M_029_2	137529	4	0.90	149459	2	0.90
M_029_3	94660	20	0.90	138017	14	0.90
M_029_4	136942	3	0.90	149690	2	0.90
M_030_1	113111	11	0.90	143016	4	0.90
M_030_2	139597	3	0.90	150637	2	0.90
M_030_3	113385	12	0.90	145078	3	0.90
M_030_4	138710	3	0.90	149234	2	0.90
M_031_1	133328	3	0.90	148966	2	0.90
M_031_2	140207	3	0.90	144060	2	0.90
M_031_3	134200	3	0.90	149829	2	0.90
M_031_4	140171	2	0.90	142302	2	0.90
M_032_1	138584	2	0.90	145199	2	0.90
M_032_2	134688	2	0.90	134378	2	0.90
M_032_3	138904	2	0.90	143721	2	0.90
M_032_4	132723	2	0.90	132756	2	0.90
M_033_1	129028	2	0.90	130524	2	0.90
M_033_2	129490	3	0.90	129157	3	0.90
M_033_3	128045	2	0.90	128825	2	0.90
M_033_4	129438	3	0.90	128356	3	0.90

brightness presented by the particles of this image. Seeing this, the methods that resulted in larger areas were present in the experiments *M_031_1*, *M_030_2*, *M_031_3* and *M_029_4*, for a mask radius and weight, in all of them, equal to 2 pixels and 0.9, respectively. Applying the same workout described in the previous paragraph for the image *Image 2* the junction of the resulting masks from the methodologies that originated maximum areas for this image, results in a mask with 228 detected cells and a total area of 152218 pixels (Figure J.4). In the case of the junction of the resulting masks of the application, in *Image 3* (P hefe 1200ms DF 400), of the best methodologies for *Image 1* (N-Hefe_400x_dark Field_1), the resulting mask would have 8 more detected cells and a total area of 155978 pixels (Figure J.5). The sum of these two masks results in a mask having the same number of cells as the previous mask, but with an area of 157029 pixels (Figure J.6).

Table 5.8. *Image 4* and *Image 5* maximum areas.

	N_Yeast 400x DF 300ms			N_Yeast 400x DF 500ms		
Test	Max area	Mask radius	Mask weight	Max area	Mask radius	Mask weight
	pixels	pixels		pixels	pixels	
M_028_1	89571	1	0.10	78537	6	0.50
M_028_2	93827	5	0.90	90837	3	0.90
M_028_3	82932	4	0.40	74953	3	0.20
M_028_4	98948	4	0.90	92102	3	0.90
M_029_1	82600	13	0.90	73171	19	0.90
M_029_2	101908	3	0.90	96474	4	0.90
M_029_3	78558	19	0.90	68836	20	0.90
M_029_4	101491	3	0.90	98084	4	0.90
M_030_1	91788	5	0.90	85628	12	0.90
M_030_2	103198	3	0.90	98392	4	0.90
M_030_3	91789	6	0.90	84914	11	0.90
M_030_4	101477	3	0.90	97809	3	0.90
M_031_1	102560	2	0.90	96969	3	0.90
M_031_2	104378	2	0.90	100311	3	0.90
M_031_3	103179	2	0.90	96722	3	0.90
M_031_4	103794	2	0.90	99604	2	0.90
M_032_1	104043	2	0.90	99052	2	0.90
M_032_2	103563	2	0.90	100486	2	0.90
M_032_3	103203	2	0.90	99029	2	0.90
M_032_4	103122	2	0.90	99603	2	0.90
M_033_1	104642	2	0.90	99096	2	0.90
M_033_2	103513	2	0.90	99463	2	0.90
M_033_3	104284	2	0.90	98043	2	0.90
M_033_4	104483	2	0.90	99181	2	0.90

Of the six new tested images, the *Image 4* (N_Yeast 400x DF 300ms), is the most similar to *Image 1*, much due to the similarity in exposure times. As can be seen by the obtained maximums, the methodologies with *Subtract Background* rolling ball radius equal to 10, 20 and 30 pixels resulted in larger areas, all obtained with mask radius between 2 and 3 pixels, and mask weights equal to 0.9. Due to the lower contrast of the cells, the less aggressive *Subtract Background* options (*_2* and *_4*) also returned acceptable maximums for lower rolling ball radius, namely, 2 and 3 pixels. This time the sum of the resulting masks of the methods that returned larger areas, *M_031_2*, *M_033_1*, *M_033_3* and *M_033_4* (all with mask radius and weight equal to 2 pixels and 0.9) resulted in a not much different mask of the mask resulted from the sum of the masks of best methods for the *Image 1* (Figure J.7 and J.8). Both masks account for 157 detected cells and total areas of 106083 and 107001 pixels. Again, the combination of these two masks resulted in an improvement in the total number of detected cells, but this time

Table 5.9. *Image 6* and *Image 7* maximum areas.

	N_Yeast 400x DF 700ms			N_Yeast 400x DF 1000ms		
Test	Max area	Mask radius	Mask weight	Max area	Mask radius	Mask weight
	pixels	pixels		pixels	pixels	
M_028_1	79985	3	0.20	74920	1	0.10
M_028_2	89281	4	0.90	88347	4	0.90
M_028_3	48498	2	0.30	42692	2	0.30
M_028_4	92659	4	0.90	88475	3	0.90
M_029_1	72258	17	0.90	69272	20	0.90
M_029_2	96717	5	0.90	97045	4	0.90
M_029_3	68882	20	0.90	65169	9	0.90
M_029_4	98456	4	0.90	95566	4	0.90
M_030_1	84791	11	0.90	80612	6	0.90
M_030_2	99984	4	0.90	97076	3	0.90
M_030_3	83423	5	0.90	78917	6	0.90
M_030_4	100850	4	0.90	98442	4	0.90
M_031_1	96928	3	0.90	95113	3	0.90
M_031_2	100965	2	0.90	98690	3	0.90
M_031_3	96124	3	0.90	95381	3	0.90
M_031_4	100082	2	0.90	97585	2	0.90
M_032_1	100077	2	0.90	96372	2	0.90
M_032_2	96664	2	0.90	96873	2	0.90
M_032_3	99755	2	0.90	96662	2	0.90
M_032_4	96814	2	0.90	95060	2	0.90
M_033_1	94612	2	0.90	92289	2	0.90
M_033_3	92471	2	0.90	91577	2	0.90
M_033_4	94023	2	0.90	91681	2	0.90

only one more cell was detected, and an increase of 875 pixels of detected area was observed (Figure J.9).

Just as the previous image, the *Image 5*, mainly due to the shorter exposure time taken to capture it, has maximum areas in experiments with higher rolling ball radius and in experiments with smaller rolling ball radius but with the less aggressive, at least for low contrast images, *Subtract Background* options (Table 5.8). The maximum areas were observed in the experiments *M_031_4*, *M_032_2*, *M_032_3* and *M_033_1*, all with mask radius and weights equal to 2 pixels and 0.9, respectively. The overlap of the masks obtained in these experiments resulted in the mask of Figure J.10. It has 158 cells with a total area of 104190 pixels. The overlap of the masks resulted from the experiments *M_033_1*, *M_033_2*, *M_033_3* and *M_032_4*, with the mask radius and weights which originated the best mask for the first image generated the mask in Figure J.11. In it are present 160 cells with a total area of 107003 pixels. The junction of the masks of Figures J.10 and J.11 resulted in the mask of Figure J.12. With this final mask it is possible to conclude that the mask of Figure J.11 is the main responsible for the improvement in cell number and area value, since the number of final cells is equal to the number of cells in Figure J.11, with a small increase of 347 pixels in the total area.

Finally, the analysis of *Image 6* and *Image 7* (N_Yeast 400x DF 700ms and N_Yeast 400x DF 1000ms), resulted in masks with maximum areas on the same experiments, particularly *M_032_1*, *M_031_ 2* and *M_032_3*, for a mask radius and weight and equal to 2 pixels and 0.9, respectively, and *M_030_4* for a mask radius and weight and equal to 4 and 0.9, respectively (Table 5.9). The overlap of these masks resulted in the images of Figures J.13 and J.16, respectively for *Image 6* and *Image 7*. These masks contain a number of cells equal to 163 and 159, respectively, and a total area equal to 106310 and 104602, respectively. Similarly, the masks in Figures J.14 and J.17 resulted from joining the masks from the experiments with these two images with the methodologies which give higher areas in *Image 1*. With the given masks, it can be seen that even though the number of cells between Figures J.13 and J.14 remain unchanged, all detected cells are not the same, existing an increase in the final area of 1121 pixels, between the mask of Figure J.13 and the mask of Figure J.14. In the case of Figures J.16 and J.17, it is verified an increase in cell number from 159 to 160 cells and an increase in the total area from 104602 to 106177 pixels. The overlapping of the masks from Figures J.13 and J.16 with the masks of Figures J.4 and J.17, respectively, it is obtained the masks of Figures J.15 and J.18, respectively. These two final masks contain a number of cells equal to 165 and 163 and a total area equal to 109333 and 108224 pixels, respectively.

5.2. Final Version

As can be seen, the previous short exercise intended to show that the inclusion of different methods in the plugin code may not always be completely effective in detecting all the cells present in an image. Given the results obtained with the realization of all experiments, it can be stated that for dark-field contrast yeast images with 400x magnification and exposure time between 150 and 300 ms, it becomes essential to use methodologies that contain an *Unsharp Mask* method with mask radius between 2 and 3 pixels and mask weight equal to 0.9. It was also verified that for images taken with exposure time above the last mentioned range, the use of higher rolling ball radius regarding a more efficient cell and area detection is the most appropriate.

Since most of the received images were captured with exposure times above mentioned, it was decided to include in the Main Pipeline the processing methods used in the experiments *M_030_1*, *M_030_2*, *M_030_3* and *M_030_4* with and without unsharp filter. In case using *Unsharp Mask*, it was decided to use mask radius equal to 2 and 3 pixels, and mask weight equal to 0.9. With these methods included in the plugin code, for each analyzed image, it will be generated 12 masks which should be overlapped in a final mask through the *Image Calculator* method. To test this first final version (**Version 1**) of the Main Pipeline, all images used in the experiments were analyzed: *Image 1* to *Image 7*, plus a new set of images: *Image 8* to *Image 12* (Appendix I). The resulting masks and the respective manually defined borders (red edges) of the *Manual Area* can be viewed in Figures K.1 – K.12 (Appendix K).

The same images were tested with a second version of the Main Pipeline (**Version 2**). This time the *Version 2* includes the methods of the *Version 1*, with the addition of methods from experiments *M_030_3* and *M_030_4* with the inclusion of the *Split Red* method instead of *Split Green* method. Just as in the first version, the method *Unsharp Mask* options were 2 and 3 pixels of mask radius and 0.9 of mask weight. E.g., with this version, 4 masks resulting from the red channel processing are added to the other 12 masks resulting from *Version 1* methodologies. The masks resulting from the processing with the *Version 2* methodologies, of the abovementioned images together with it manually defined red borders are in Figures K.13 – K.24 (Appendix K).

Finally, the analysis of the same images with the inclusion of the refinement methodology *ImagePlus thresholdRange (ImageProcessor ImageProcessor, int tresAmplitude, int autoTres, int tresStep, String imageTitle)* at the end of each one of the 16 sets of methods used in *Version 2* was performed with the **Version 3**. In the configurations without *Unsharp Mask*, values of *autoTres* and *tresStep* equal to 10 and 2, respectively, were used. In the configurations with *Unsharp Mask* and *Split Green*, values of *autoTres* and *tresStep* equal to 20 and 4, respectively were used, and for the configurations with *Unsharp Mask* and *Split Red*, values of 6 and 2 were used for *autoTres* and *tresStep*, respectively. The masks generated by experiments with *Version 3* are shown in Figures K.25 – K.36. The values of *autoTres* and *tresStep* shown until now, were chosen in order to guarantee a certain safety in cell images thresholding. Very high values of *autoTres* could imply a threshold upper bound which would eliminate edges between together cells, leading to an inadequate joining of two cells as a bud cell. Likewise, very low values of *autoTres* could imply a threshold lower limit which would cause the appearance of particles not desirable to be detected, as can be seen in Figure 5.1. The selection of *tresStep* values was made taking into account the collection of various masks, resulting from the application of multi thresholds with different threshold levels, taking careful to not make a plugin run process too "heavy". The decision to use multiple threshold levels between the upper and lower limits defined by the *tresAmplitude* variable was due to the intention to soften the edges, thereby increasing the detected area. The using of the upper and lower values alone as threshold levels might imply the opening or extreme widening of the cells edges, resulting in incorrect cell detection.

In the images of Figure 5.15, a detailed example of the result of the implementation of *Version 2* and *Version 3* to various particles present in some tested images. As can be seen with the transformation of the left masks, resulted from the application of *Version 2*, in the right masks, resulted from the applications of *Version 3*, the threshold levels variation had a smoothing effect in some cells area borders. This effect caused the total detected area in multiple images to increase substantially. It also allowed an improved bud cells classification, whereas with *Version 3*, two detected cells by *Version 2* that should be a bud cell, could be detected as a bud cell. This is all possible to verify with the values of cell counting and area size in pixels shown in Table 5.10. For all analyzed images there was an increase in the total detected area with the application of the final versions of the plugin, e.g. from *Version 1* to *Version 3* were an improvement in the area detection. Regarding this, in most cases there was a slight increase in the number of detected cells. In some cases there was a decrease in the number of detected cells. This was mainly due to the case described above, and exemplified with some particles of the masks in Figure 5.15. The combination of two detected cells in a cell bud was slightly common with the variation of the final versions application.

5.3. Boundaries determination

The classification of cells detected by the plugin was made after the analysis of the cells present in 20 microscopic images of *S. cerevisiae* cultures (Figure I.1 – I.20 in Appendix I). Applying the methodologies of the final version of the plugin ScerevisiCounter, ImageJ available variables, like area, ellipse, centroid and circularity, were collected, namely the data respective to *Area*, *Major*, *Minor*, *Circularity*, *Roundness* and *X* and *Y* coordinates. After the collection of the data about cell morphology, a manual classification of the 2603 detected cells in non-bud and bud was taken. This manual classification was based only on the observation of the borders of the detected cells. Thus, the particles which had a circular or ellipsoid geometry were classified as non-buds, being the remainder classified as buds. Completed this task, it was verified the existence of 2453 non-bud cells and 147 bud cells.

For the cell classification in large, normal and small, the data relating to the areas size of the cells manually classified as non-buds were sufficient. From an area size analysis it was found that the lowest and highest observed areas had a value equal to 2.56 and 45.98 μm^2, respectively. These two values are strongly influenced by the minimum and maximum values defined to be the limits of the size of the particles detected the *Particle Analyzer*. The area values limiting the classification between small, normal or large cells were determined by the quantiles 33 % and 66 % of the areas values. So the limit between small and normal cells was determined to be equal to 13.41 μm^2 (quantile 33 %), while the limit of classification between normal cells and large cells was determined as being equal to 19.35 μm^2 (quantile 66 %).

As for the bud cells classification, the *Major/Minor* (or *Aspect Ratio*) was a good classifier parameter. For an *AR* value higher than 1.65, from the 2603 detected cells, only 2 were not manually considered as non-buds. While, from the 147 manually considered as bud cells, only 7 had *AR* values lower or equal to 1.65. The morphological data showed that smallest area of these seven cells was equal to 19.30 μm^2, while the highest circularity and roundness values of the

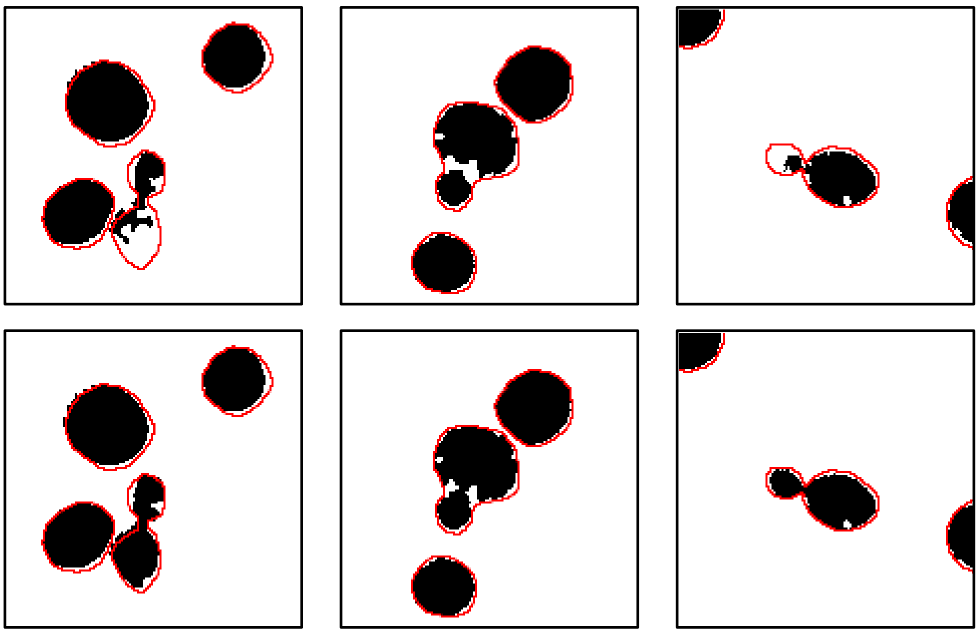


Figure 5.15. Resulting masks from applying two different ScerevisiCounter final versions to three sets of cells. The masks above are the result of the implementation of *Version 2*, while the masks below are the result of applying the *Version 3*. The images above correspond to the same images directly below these. The red manually defined borders are also visible.

Table 5.10. Final versions results on cell counting and area detection.

Image		Version	Count	Total area pixels
N-Hefe_400x_dark-field_1	1	1	63	44858
		2	59	45640
		3	59	46040
P hefe 800ms DF 400	2	1	235	159748
		2	241	171250
		3	240	175021
P hefe 1200ms DF 400	3	1	244	159840
		2	243	161607
		3	244	163992
N_Yeast 400x DF 300ms	4	1	162	108976
		2	162	112622
		3	163	114583
N_Yeast 400x DF 500ms	5	1	166	109576
		2	164	113036
		3	165	115005
N_Yeast 400x DF 700ms	6	1	167	109614
		2	167	114033
		3	168	116210
N_Yeast 400x DF 1000ms	7	1	165	108666
		2	166	114382
		3	165	116186
N Hefe 100ms	8	1	41	27446
		2	56	34880
		3	56	35318
N Hefe 150ms	9	1	57	34219
		2	57	34329
		3	57	34720
N Hefe 200ms 2	10	1	60	35184
		2	63	36873
		3	63	37375
N Hefe 200ms	11	1	56	32491
		2	56	32662
		3	58	33785
N Hefe 250ms	12	1	56	31989
		2	57	32858
		3	57	33362

same cells were equal to 0.65 and 0.82, respectively. Introducing this classification system, bud cells have values of *AR* higher than 1.65, at least 7 cells that would be expected to be classified as buds, would be classified as non-buds. Therefore, following this classification, a verification of the cell type was introduced. Of these cells, already classified as buds, small, normal or large with the above proposed system, those which had areas greater than 19.29 μm^2, and circularity and roundness values lower than 0.66 and 0.83, respectively, would be reclassified as bud cells. Thus, those 7 bud cells with *AR* < 1.65 would be correctly reclassified as bud cells.

However, following the classification system above described, 13 non-bud cells that had been correctly classified as non-buds, would be wrongly reclassified as buds. In order to reduce this high number of normal cells wrongly reclassified as bud cells, it was decided to lower the values of circularity and roundness. With this action, only 2 cells, of those 7 with *AR* lower than 1.65, would not be reclassified as buds. Thereby, for areas greater than 19.29 μm^2, and circularity and roundness values lower than 0.64 and 0.74, respectively, only 1 cell correctly classified as non-bud was wrongly classified as bud.

In addition to this system, a re-classification of bud cells was also added according to the proximity between large and small cells. So, if a large or a small cell presented a distance lower than 6 μm to another small or large cell, respectively, it would be proceeded the removal of a large cell and a small cell and the adding of a bud cell to the final total counting. The calculation of the distance between two cells was carried out by using the *X* and *Y* coordinates and the following equation:

$$d = \sqrt{(x1 - x2)^2 + (y1 - y2)^2} \qquad (5.1),$$

where *x1*, *x2*, *y1* and *y2* are the *X* and *Y* Cartesian coordinates of a cell 1 and a cell 2.

Therefore, the classification system inserted in the final version was structured as follows:

```
FOR EACH cell
     IF cell (Major) > 1.65 * cell (Minor) AND cell (Area)
>= 11.78
     THEN cell = bud cell
     IF cell (Area) > 19.35
     THEN cell = large cell
     IF cell (Area) > 13.41 AND cell (Area) <= 19.35
     THEN cell = normal cell
     IF cell (Area) <= 13.41
     THEN cell = small cell
 FOR EACH cell
     IF cell != bud
          IF cell (Circularity) < 0.64 AND cell (Roundness)
< 0.74 AND cell (Area) > 19.29
          THEN cell = bud
```

5.4. Manual and automatic counting

Included the classification systems in the plugin final version, a small test with the images of Figures I.13 – I.16 (Appendix I) was set up. For this, with the aid of others, a manual counting of bud and total cells was been made, for comparison with the counts taken by ScerevisiCounter.

The total cells and bud cells counts, automatically and manually done, for the 4 pictures chosen, are in Table 5.11. Generally, the plugin efficiency detecting bud cells was inferior to the manual counting. This efficiency is heavily compensated by the higher accounting of bud cells in Bild_229 (*Image 14*). However, in general it can be said that the accounting of total cells by the plugin was very similar to the manual counting.

Deciding whether, if two cells are mother and daughter cells or just two non-bud cells together, it is sometimes difficult to take, and visually deciding this may result differently from the decision taken by the algorithm inserted the plugin. This algorithm is quite efficient when the septum that separates the two cells is weak, or poorly defined, causing the detected area to cover this division, making the cell to be easily counted as bud. A large cell usually means that enough carbohydrates were stored and likely to start a new bud cycle. Small cells are usually one product of *S. cerevisiae* cell cycle, and are common to be identified as the daughter cells that had just separated from the mother cell. Sometimes the septum that separates a mother cell from a daughter cell is difficult to "break" and for this reason it was introduced in the algorithm the reclassification of very close large and small cells as a bud cell.

Generally the implemented classification system counted cells fairly well. In almost all the images were fewer bud cells detected by the plugin than manually. The total bud cell count is balanced in the second image, Bild_229, wherein the automatically detected bud cells were the double from the manual counted. In this image are present many more cells than in the other images. Moreover, many groups of cells are also visible together. This may have influenced the manual counting.

Table 5.11. Automatic *vs* manual cell count.

Picture		Manual		ScerevisiCounter	
		Bud cells	Total cells	Bud cells	Total cells
Bild_227	**13**	12	106	11	105
Bild_229	**14**	16	198	31	184
Bild_281	**15**	44	135	37	146
Bild_289	**16**	42	91	36	95
Total		114	530	115	530

6. Conclusions and further work

In this dissertation, the implementation of a Java language code that emulates some ImageJ commands (organized and/or modified) to investigate *Saccharomyces cerevisiae* cells from microscopy was successfully performed.

Images containing groups of different cells size and budding cells were counted automatically through a set of pre-processing methods specifically implemented for the improvement of particles recognition and analysis. The ScerevisiCounter was able to identify budding cells and individual cells, as well as to discriminate cells depending on their size (small, normal, large). The classification of bud cells, in addition to the morphological parameters chosen for this task, was also based on the distance between the large and small cells types, where an exemplary of each of these cells types may be classified as a bud, thereby increasing the bud cells count closest to the real number.

The plugin also provided the area results in square micrometer, to classify each cell identified into a group, and to count the cells presented in each image. At the same time, the code included in the plugin was able to separate budding cells based on *septum* detection and to select only one cell when overlapping cells were present. Whenever one cell was covered by another cell, the binarization process was quite difficult for edges detection, resulting in two cells. It was found quite relevant to remove some particles from all images provided by the background noise. Thus, particles with lower and higher area than cells were removed. The plugin was also able to provide a morphological analysis where area, circularity, roundness and aspect ratio were selected as the main parameters. Cells were correctly classified in non-bud and bud with an efficiency of 99.8 %.

The code included in the plugin has no specific algorithm for the distinction of overlapping cells from bud cells. Despite this, he is able to perform this task quite accurately since the detection of bud cells rely mostly on breaking the *septum* that separates the mother cell from the daughter cell. It was assumed that this boundary, at least for most cases, is less defined, or with a lower contrast than the rest of the cells edges, being easily included, unlike the rest of the outer edges, in the cells area after image processing and binarization. In the case of overlapping cells, if the cell that is behind is totally eclipsed by the front cell then the plugin will easily count one cell. If the cell that is behind is partially eclipsed, then the edges of the two cells would be hardly broken by the binarization, resulting in the detection of two separate cells. However, the classification based on the proximity between cells could make an addition to bud cells case these two overlapping and partially eclipsed back cell were small and large cells.

During the development of the ScerevisiCounter, it was found that depending on the image, masks resulting from different processing methods were added to the final mask image, which was quite close to the original image. The ScerevisiCounter plugin automatically selected the threshold levels from each image for processing and it was found that the red channel was quite relevant in the processing step. To improve the plugin performance, two main processing methods were studied: *Subtract Background* and *Unsharp Mask*. The number of processing methodologies was minimized due to the previous knowledge of the type of images. The counting and classification process in the ScerevisiCounter plugin provided reliable results when compared to a visual examination of each image.

In conclusion, the plugin code developed during this dissertation can be found as an alternative method to manual cell counting and to distinguish normal from bud cells in *S. cerevisiae* cultures. Furthermore, the ScerevisiCounter is able to display the final results in tables which are easily converted to Excel files, including the classification assigned to each cell and the actual areas of each one.

For further work, it important to notice that depending on the type of images to analyze, the application of different methodologies will improve the final mask image presenting cell counts and detected area as close as possible to the original image. To improve the performance of a specific plugin will require a number of methodologies added that can be minimized knowing *a priori* the most effective methods.

References

[1] W. S. Rasband, "ImageJ," 1997. [Online]. Available: http://rsbweb.nih.gov/ij/.

[2] D. E. Briggs, C. A. Boulton, P. A. Brookes, and R. Steven, *Brewing: science and practice*. Woodhead Publishing Limited, 2004, pp. 363–365.

[3] B. R. Glick and J. J. Pasternak, *Molecular biotechnology: principles and applications of recombinant DNA*. ASM Press, 2003, pp. 256–308.

[4] H. Feldmann, *Yeast : molecular and cell biology*. Wiley-Blackwell, 2009, pp. 1–35.

[5] M. T. Madigan, J. M. Martinko, P. V. Dunlap, and D. P. Clark, *Brock Biology of Microorganisms*. Pearson International Edition, 2008, pp. 456–457.

[6] I. Herskowitz, "Life cycle of the budding yeast Saccharomyces cerevisiae," *Microbiol. Rev.*, vol. 52, no. 4, pp. 536–53, Dec. 1988.

[7] A. A. Barton, "Some aspects of cell division in Saccharomyces cerevisiae," *J. Gen. Microbiol.*, vol. 4, no. 1, pp. 84–6, Jan. 1950.

[8] J. Chant and J. R. Pringle, "Patterns of bud-site selection in the yeast Saccharomyces cerevisiae," *J. Cell Biol.*, vol. 129, no. 3, pp. 751–65, May 1995.

[9] L. Ni and M. Snyder, "A genomic study of the bipolar bud site selection pattern in Saccharomyces cerevisiae," *Mol. Biol. Cell*, vol. 12, no. 7, pp. 2147–70, Jul. 2001.

[10] M. a. Z. Coelho, J. a. P. Coutinho, E. C. Ferreira, M. Mota, and I. Belo, "Analysis of the effects of hyperbaric gases on S. cerevisiae cell cycle through a morphological approach," *Process Biochem.*, vol. 42, no. 10, pp. 1378–1383, Oct. 2007.

[11] N. Slavov and D. Botstein, "Coupling among growth rate response, metabolic cycle, and cell division cycle in yeast," *Mol. Biol. Cell*, vol. 22, no. 12, pp. 1997–2009, Jun. 2011.

[12] Y. Ohya, J. Sese, M. Yukawa, F. Sano, Y. Nakatani, T. L. Saito, A. Saka, T. Fukuda, S. Ishihara, S. Oka, G. Suzuki, M. Watanabe, A. Hirata, M. Ohtani, H. Sawai, N. Fraysse, J.-P. Latgé, J. M. François, M. Aebi, S. Tanaka, S. Muramatsu, H. Araki, K. Sonoike, S. Nogami, and S. Morishita, "High-dimensional and large-scale phenotyping of yeast mutants," *Proc. Natl. Acad. Sci. U. S. A.*, vol. 102, no. 52, pp. 19015–20, Dec. 2005.

[13] T. L. Saito, J. Sese, Y. Nakatani, F. Sano, M. Yukawa, Y. Ohya, and S. Morishita, "Data mining tools for the Saccharomyces cerevisiae morphological database," *Nucleic Acids Res.*, vol. 33, no. Web Server issue, pp. W753–7, Jul. 2005.

[14] S. Huh, D. Lee, and R. F. Murphy, "Efficient framework for automated classification of subcellular patterns in budding yeast.," *Cytometry. A*, vol. 75, no. 11, pp. 934–40, Nov. 2009.

[15] B. Yang Yu, C. Elbuken, C. L. Ren, and J. P. Huissoon, “Image processing and classification algorithm for yeast cell morphology in a microfluidic chip,” *J. Biomed. Opt.*, vol. 16, no. 6, p. 066008, Jun. 2011.

[16] T. L. Saito, M. Ohtani, H. Sawai, F. Sano, A. Saka, D. Watanabe, M. Yukawa, Y. Ohya, and S. Morishita, “SCMD: Saccharomyces cerevisiae Morphological Database,” *Nucleic Acids Res.*, vol. 32, no. Database issue, pp. D319–22, Jan. 2004.

[17] M. OHTANI, A. SAKA, F. SANO, Y. OHYA, and S. MORISHITA, “Development of Image Processing Program for Yeast Cell Morphology,” *J. Bioinform. Comput. Biol.*, vol. 01, no. 04, pp. 695–709, Jan. 2004.

[18] K. Jin, J. Li, F. S. Vizeacoumar, Z. Li, R. Min, L. Zamparo, F. J. Vizeacoumar, A. Datti, B. Andrews, C. Boone, and Z. Zhang, “PhenoM: a database of morphological phenotypes caused by mutation of essential genes in Saccharomyces cerevisiae,” *Nucleic Acids Res.*, vol. 40, no. Database issue, pp. D687–94, Jan. 2012.

[19] M. Riffle and T. N. Davis, “The Yeast Resource Center Public Image Repository: A large database of fluorescence microscopy images,” *BMC Bioinformatics*, vol. 11, p. 263, Jan. 2010.

[20] Kerstin Zalewski and R. Buchholz, “Morphological analysis of yeast cells using an automated image processing system,” vol. 1656, no. 96, 1996.

[21] Chris A Glasbey and Graham W Horgan, *Image analysis for the biological sciences*. New York: John Wiley & Sons, Inc., 1995, pp. 47–184.

[22] R. C. Gonzalez, *Digital Image Processing*, vol. 14, no. 3. 2002, p. 256.

[23] J. C. . Russ, “Acquiring Images,” in in *The Image Processing Handbook, Fourth Edition*, CRC Press, 2002.

[24] N. Bandekar, A. Wong, D. Clausi, and M. Gorbet, “A novel approach to automated cell counting for studying human corneal epithelial cells,” *Conf. Proc. IEEE Eng. Med. Biol. Soc.*, vol. 2011, pp. 5997–6000, Jan. 2011.

[25] P. Tibayrenc, L. Preziosi-Belloy, J.-M. Roger, and C. Ghommidh, “Assessing yeast viability from cell size measurements?,” *J. Biotechnol.*, vol. 149, no. 1–2, pp. 74–80, Aug. 2010.

[26] M. a. G. de Carvalho, R. D. a. Lotufo, and M. Couprie, “Morphological segmentation of yeast by image analysis,” *Image Vis. Comput.*, vol. 25, no. 1, pp. 34–39, Jan. 2007.

[27] G. Cahill, P. K. Walsh, and D. Donnelly, “Determination of yeast glycogen content by individual cell spectroscopy using image analysis,” *Biotechnol. Bioeng.*, vol. 69, no. 3, pp. 312–22, Aug. 2000.

[28] A. Doncic, U. Eser, O. Atay, and J. M. Skotheim, “An algorithm to automate yeast segmentation and tracking,” *PLoS One*, vol. 8, no. 3, p. e57970, Jan. 2013.

[29] W. Chen and X. Zhang, “A New Watershed Algorithm for Cellular Image Segmentation Based on Mathematical Morphology,” *2010 Int. Conf. Mach. Vis. Human-machine Interface*, pp. 653–656, 2010.

[30] M. N. Pons, H. Vivier, J. F. Rémy, and J. A. Dodds, “Morphological characterization of yeast by image analysis,” *Biotechnol. Bioeng.*, vol. 42, no. 11, pp. 1352–1359, 1993.

[31] A. L. P. do Amaral, “Image analysis in biotechnological processes: applications to wastewater treatment,” University of Minho, 2003.

[32] S.-C. Chen, T. Zhao, G. J. Gordon, and R. F. Murphy, “Automated image analysis of protein localization in budding yeast,” *Bioinformatics*, vol. 23, no. 13, pp. i66–71, Jul. 2007.

[33] a Vicente, J. M. Meinders, and J. a Teixeira, “Sizing and counting of saccharomyces cerevisiae floc populations by image analysis, using an automatically calculated threshold.,” *Biotechnol. Bioeng.*, vol. 51, no. 6, pp. 673–8, Sep. 1996.

[34] G. CAHILL, D. M. MURRAY, P. K. WALSH, and D. DONNELLY, “Effect of the concentration of propagation wort on yeast cell volume and fermentation performance,” *J. Am. Soc. Brew. Chem.*, vol. 58, no. 1, pp. 14–20.

[35] M. A. Priede, J. J. Vanags, U. E. Viesturs, K. G. Tucker, W. Bujalski, and C. R. Thomas, “Hydrodynamic, physiological, and morphological characteristics of Fusarium moniliforme in geometrically dissimilar stirred bioreactors,” *Biotechnol. Bioeng.*, vol. 48, no. 3, pp. 266–277, 1995.

[36] K. Logg, A. Diez, K. Mikael, and C. S. Park, “Image analysis algorithms for cell contour recognition in budding yeast Mats Kvarnstr o Abstract :,” vol. 16, no. 17, pp. 1035–1042, 2008.

[37] F. Meyer, “Topographic distance and watershed lines,” *Signal Processing*, vol. 38, no. 1, pp. 113–125, Jul. 1994.

[38] W. Burger and M. J. Burge, “Digital Image Processing: An Algorithmic Introduction using Java.” Springer, p. 566, 2008.

[39] W. Rasband, “The ImageJ User Guide,” *Cah. Agric.*, vol. 38, no. April, p. 186, 2010.

[40] W. Bailer, “Writing ImageJ Plugins — A Tutorial,” pp. 1–57, 2006.

Appendix A – *Group 1* Results

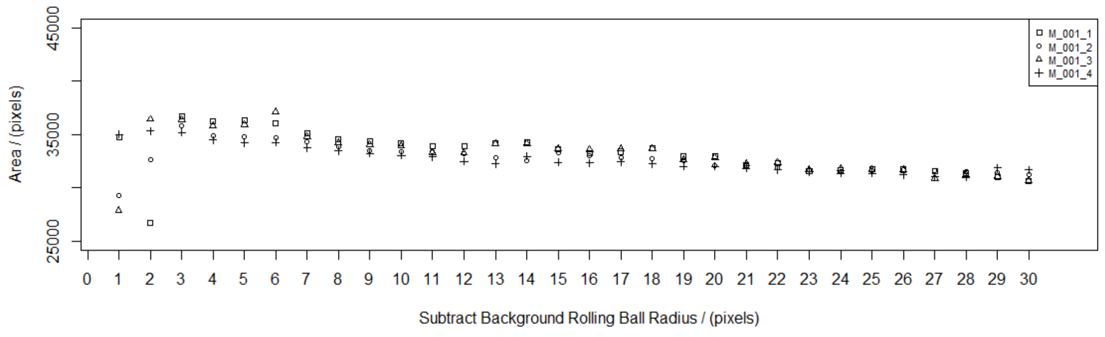


Figure A.1. *M_001* results.

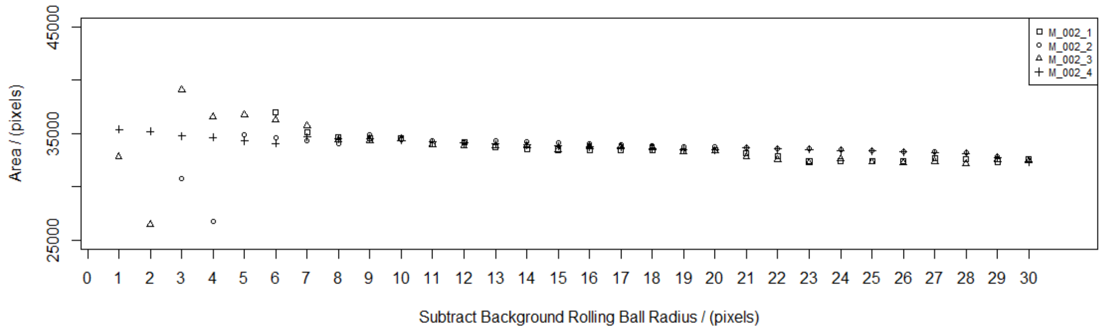


Figure A.2. *M_002* results.

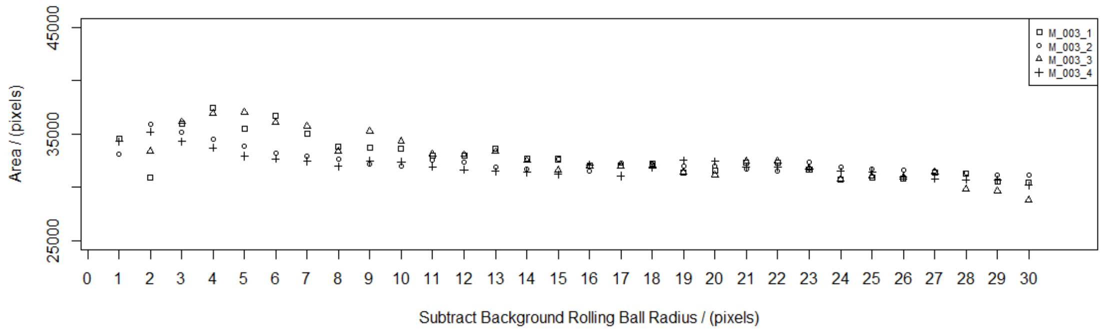


Figure A.3. *M_003* results.

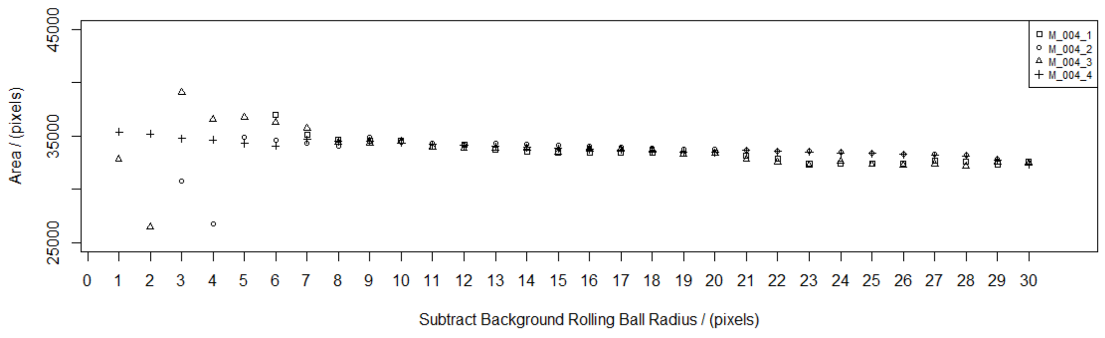


Figure A.4. *M_004* results.

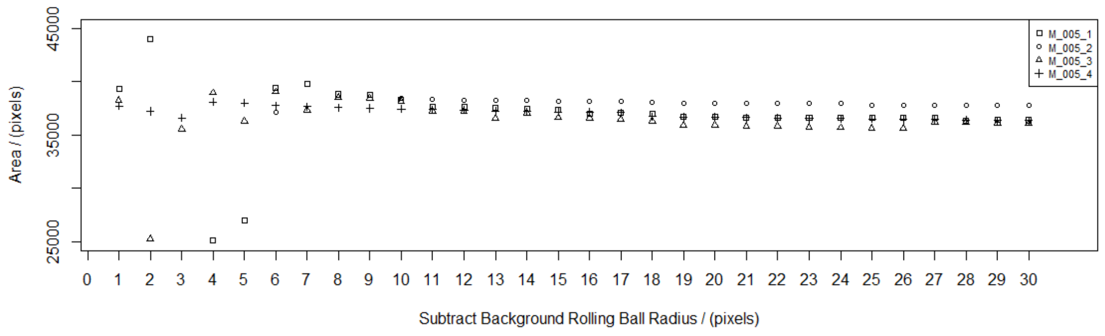


Figure A.5. *M_005* results.

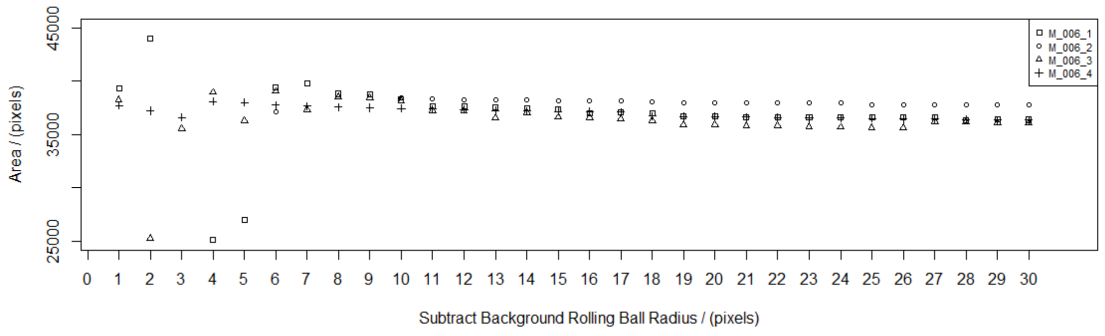


Figure A.6. *M_006* results.

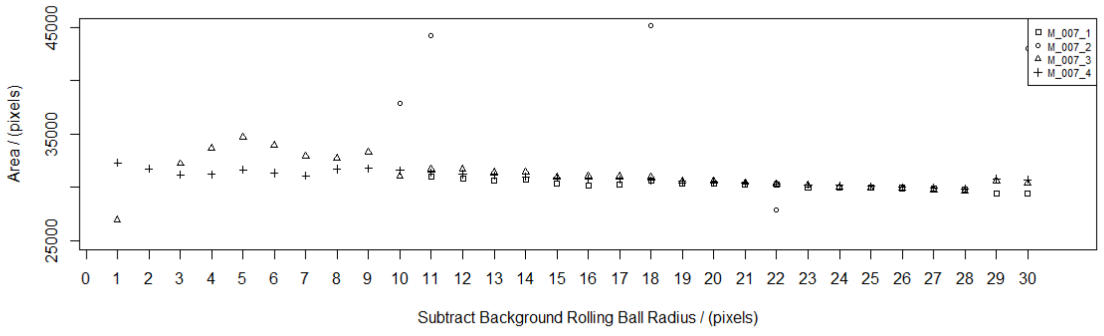


Figure A.7. *M_007* results.

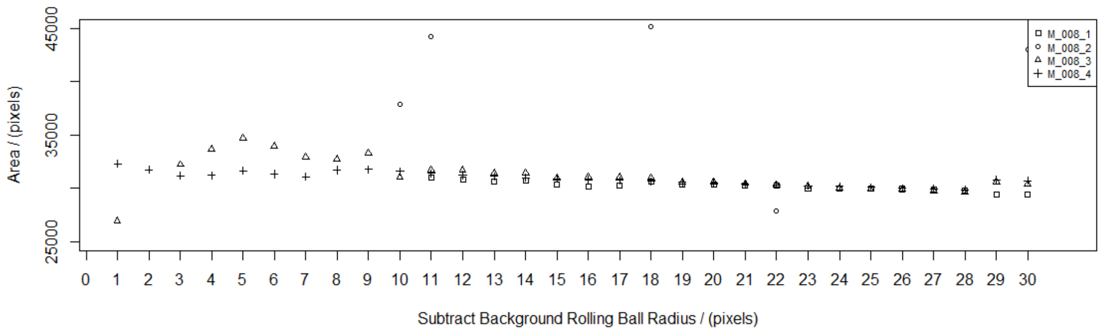


Figure A.8. *M_008* results.

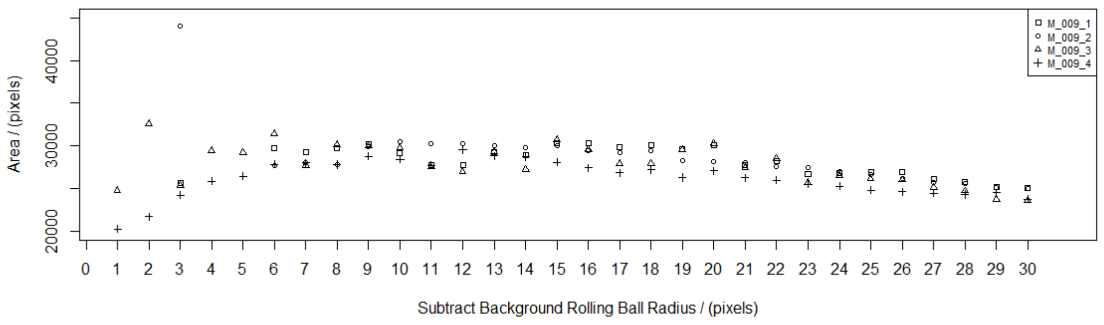


Figure A.9. *M_009* results.

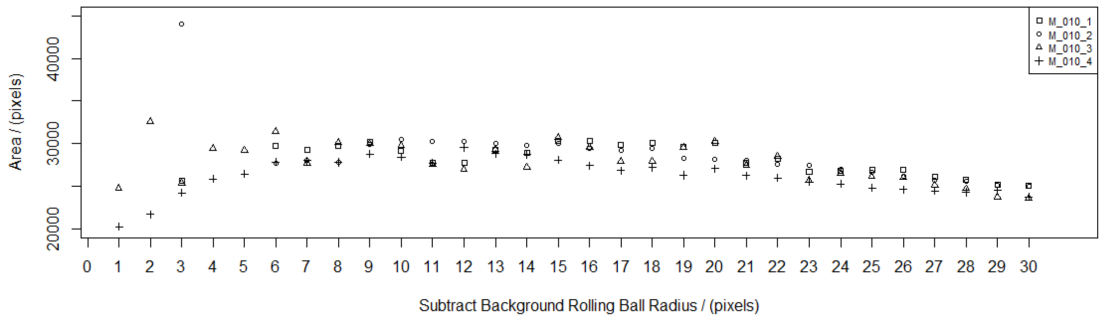


Figure A.10. *M_010* results.

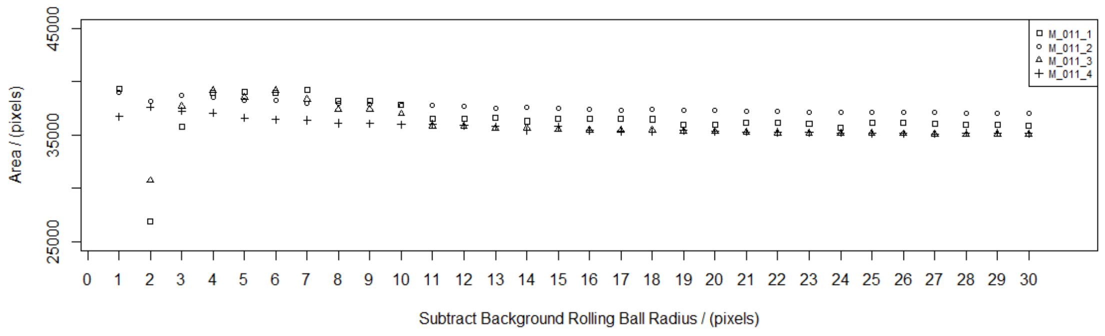


Figure A.11. *M_011* results.

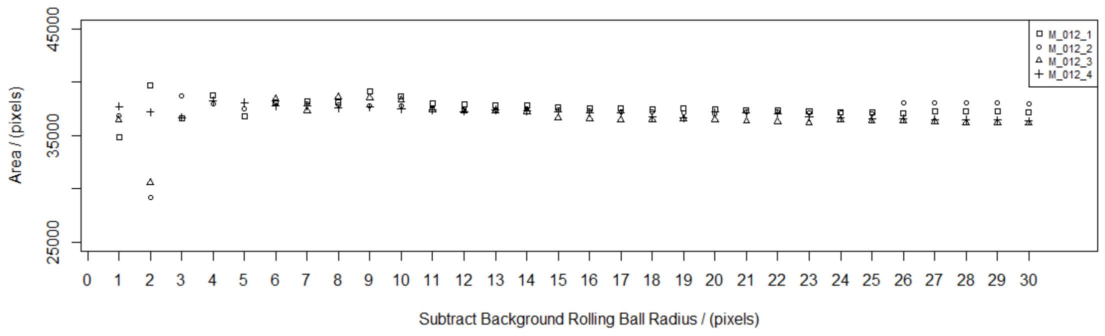


Figure A.12. *M_012* results.

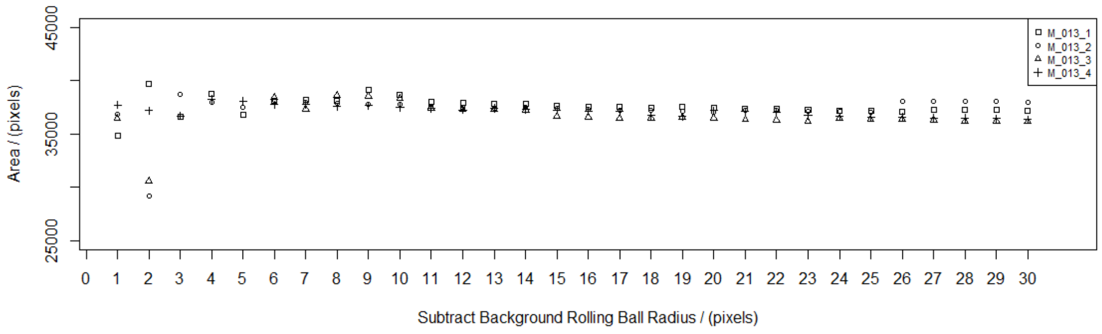


Figure A.13. *M_013* results.

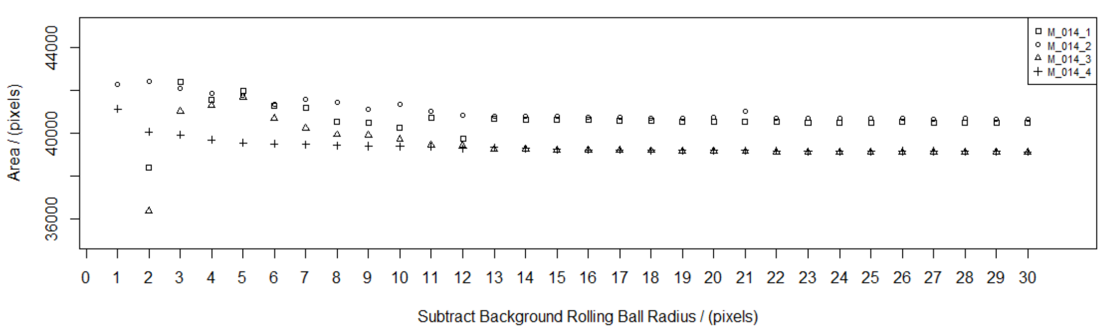


Figure A.14. M_014 results.

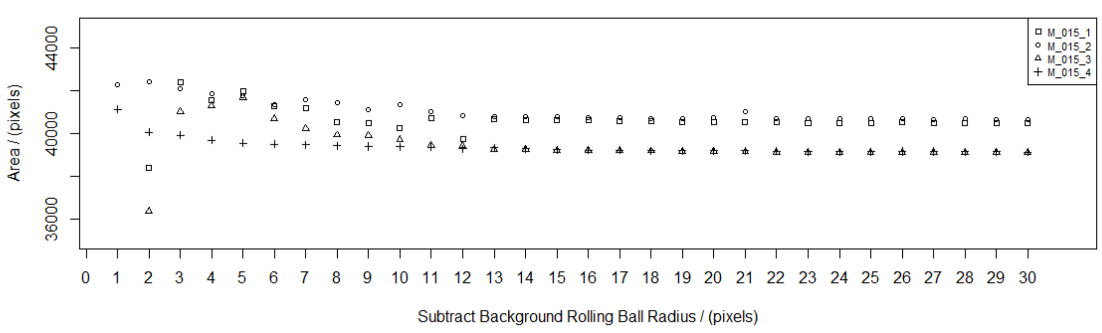


Figure A.15. *M_015* results.

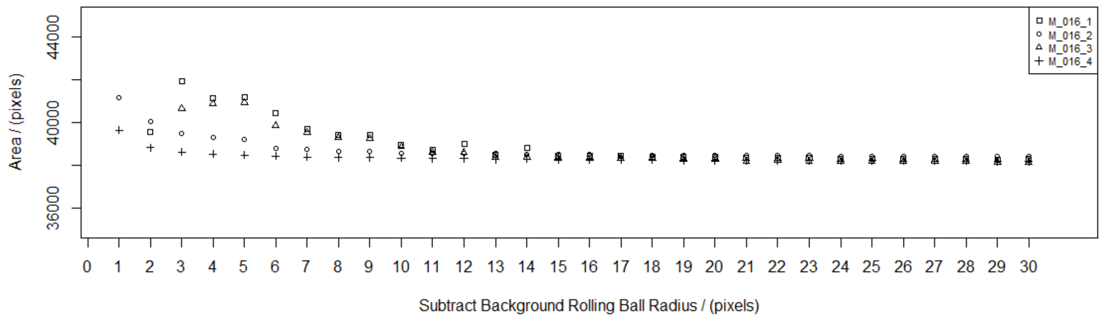


Figure A.16. *M_016* results.

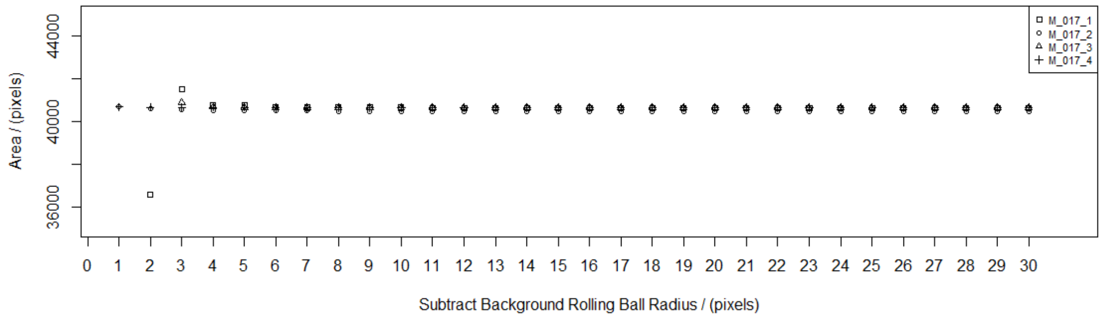


Figure A.17. *M_017* results.

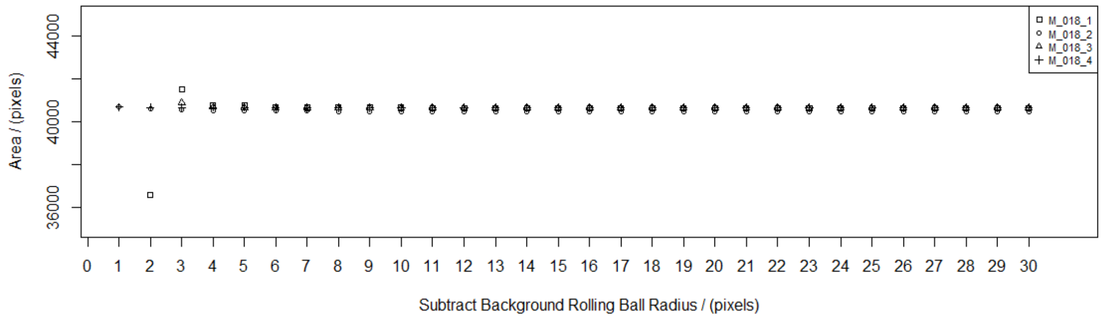


Figure A.18. *M_018* results.

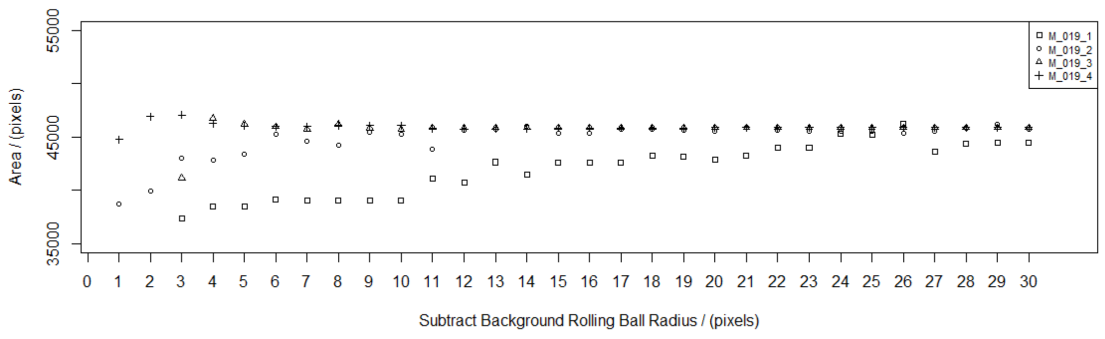


Figure A.19. *M_019* results.

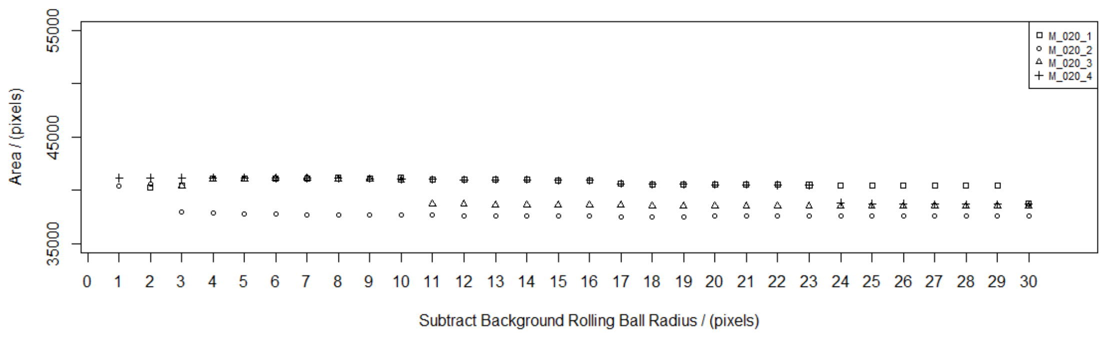


Figure A.20. *M_020* results.

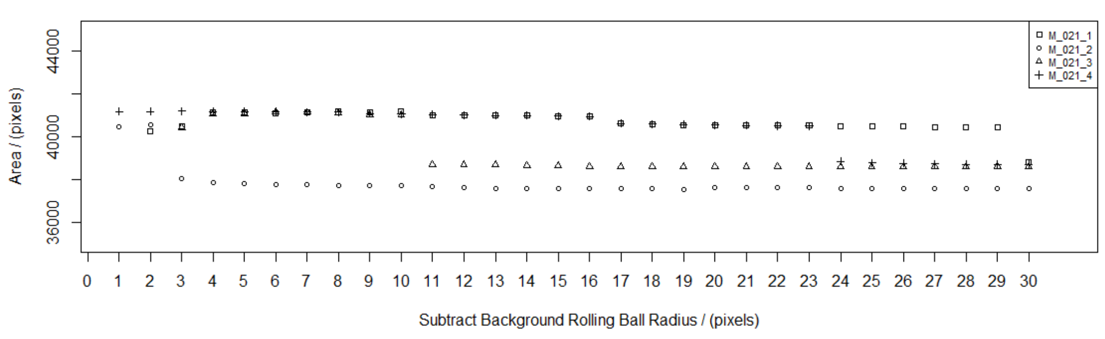


Figure A.21. *M_021* results.

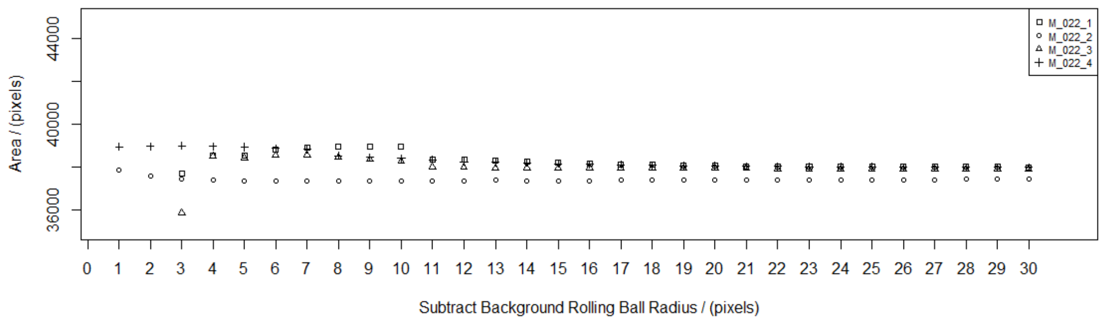


Figure A.22. *M_022* results.

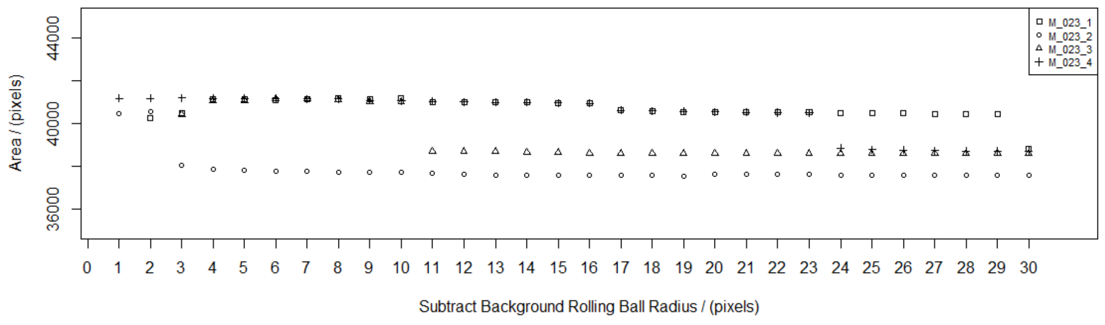


Figure A.23. *M_023* results.

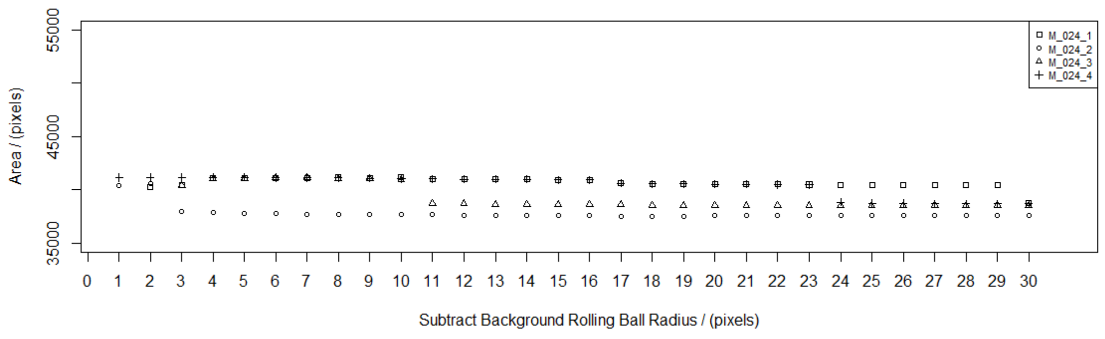


Figure A.24. *M_024* results.

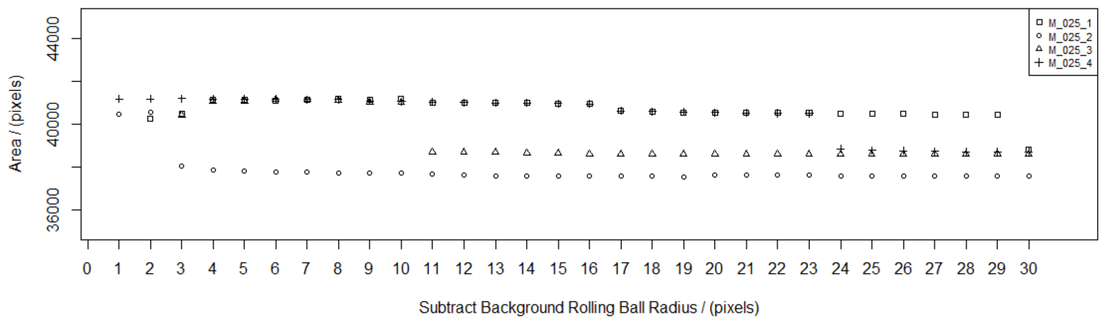


Figure A.25. *M_025* results.

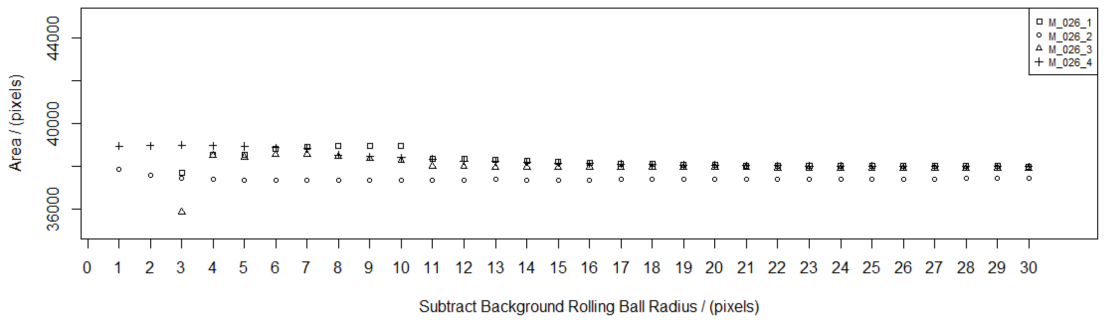


Figure A.26. *M_026* results.

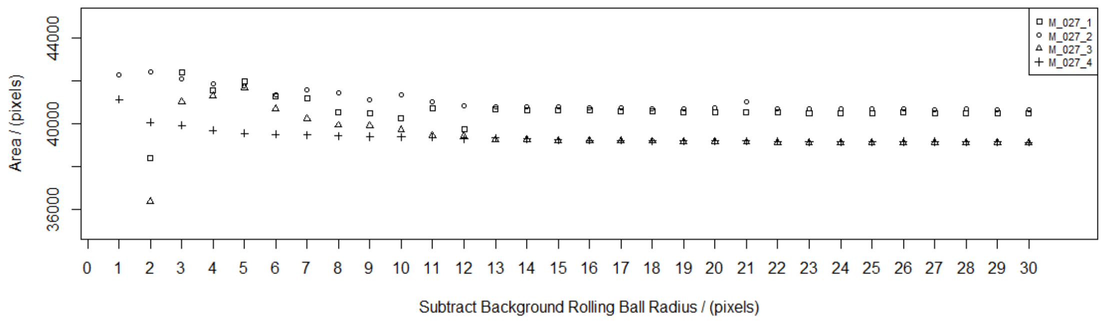


Figure A.27. *M_027* results.

Appendix B – *Group 2* Results 1

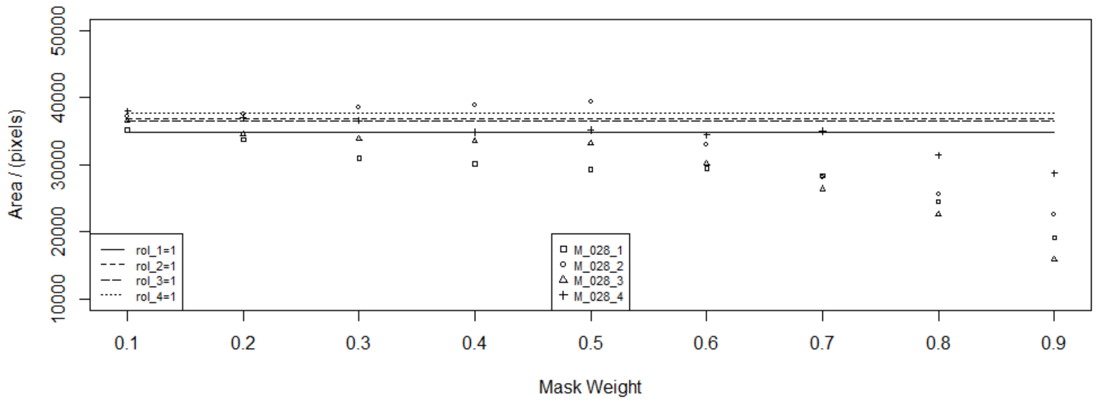


Figure B.1. *M_028* area results. *Unsharp Mask* radius equal to 1 pixel.

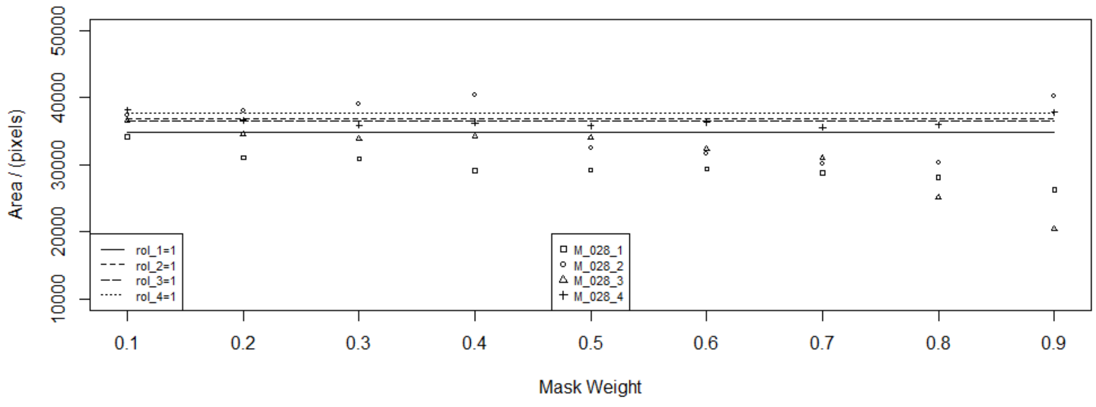


Figure B.2. *M_028* area results. *Unsharp Mask* radius equal to 2 pixels.

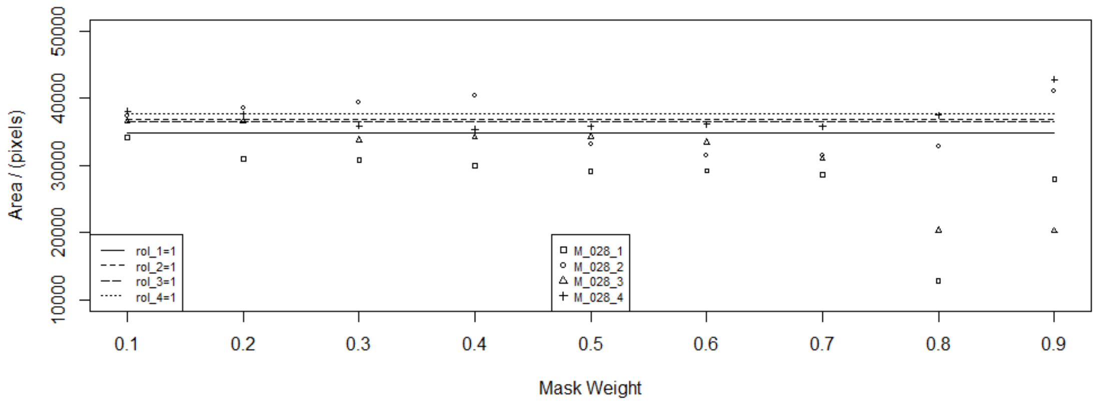


Figure B.3. *M_028* area results. *Unsharp Mask* radius equal to 3 pixels.

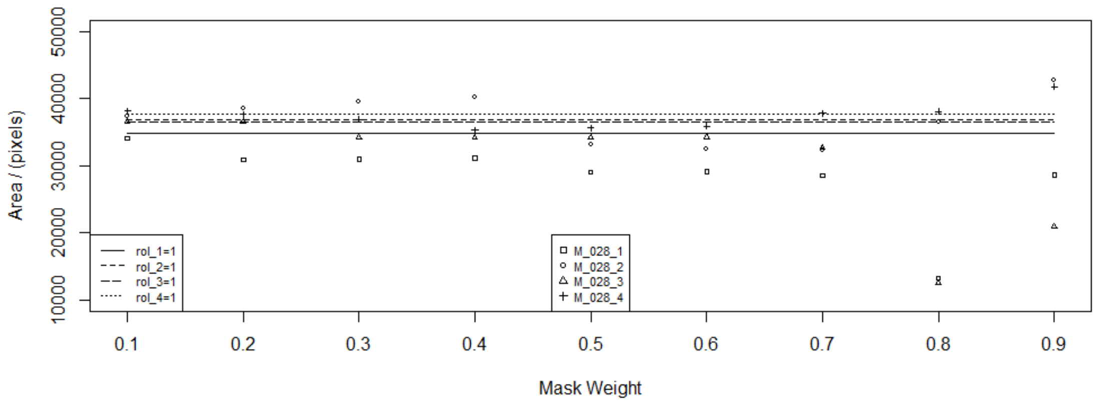


Figure B.4. *M_028* area results. *Unsharp Mask* radius equal to 4 pixels.

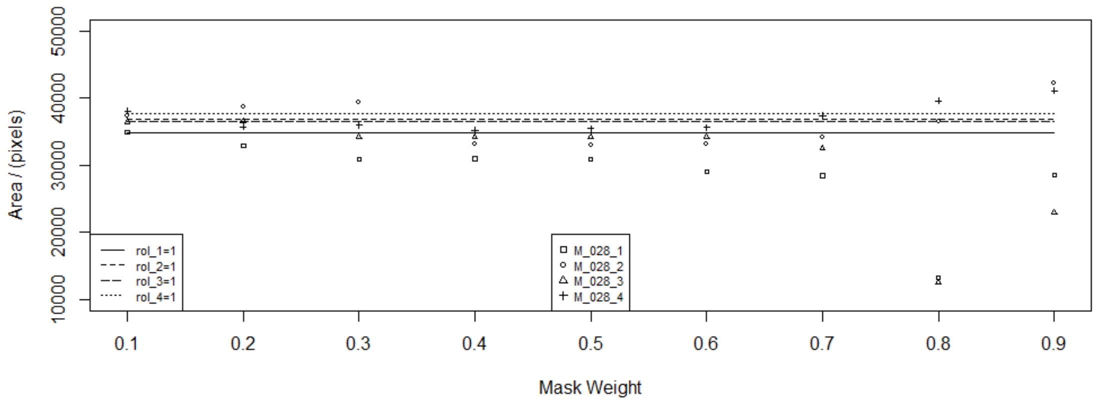


Figure B.5. *M_028* area results. *Unsharp Mask* radius equal to 5 pixels.

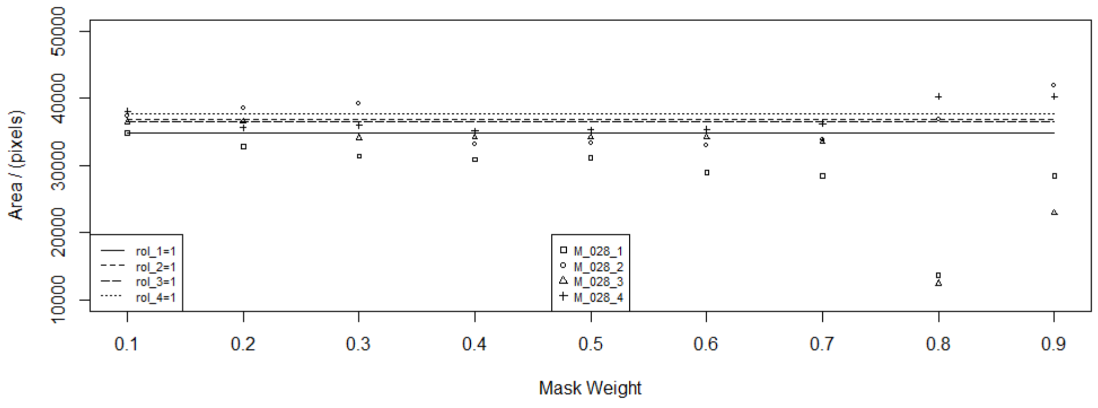


Figure B.6. *M_028* area results. *Unsharp Mask* radius equal to 6 pixels.

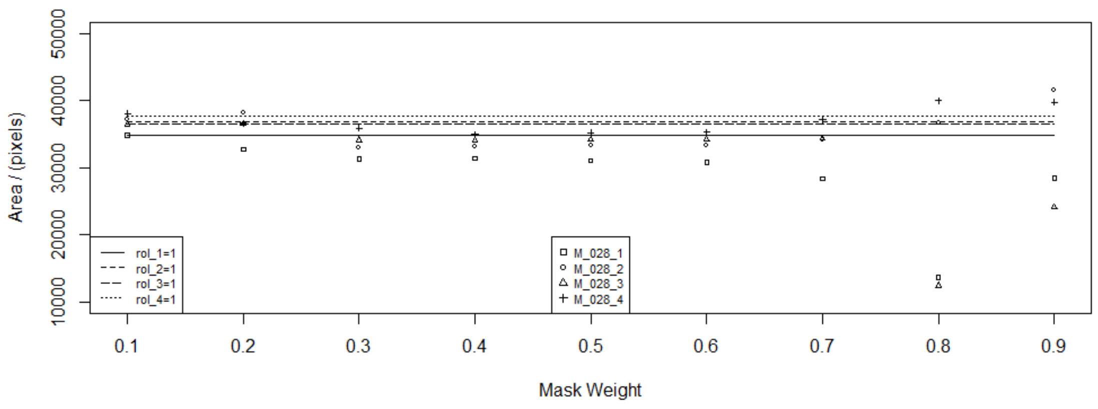


Figure B.7. *M_028* area results. *Unsharp Mask* radius equal to 7 pixels.

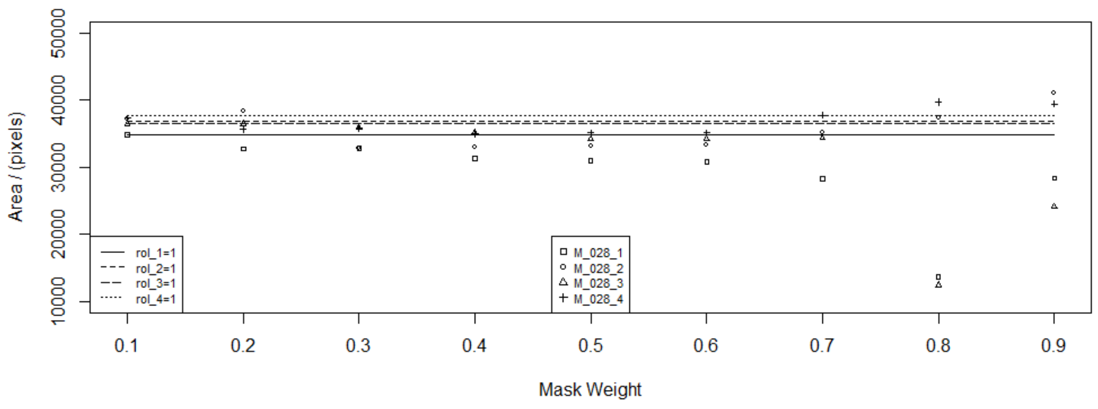


Figure B.8. *M_028* area results. *Unsharp Mask* radius equal to 8 pixels.

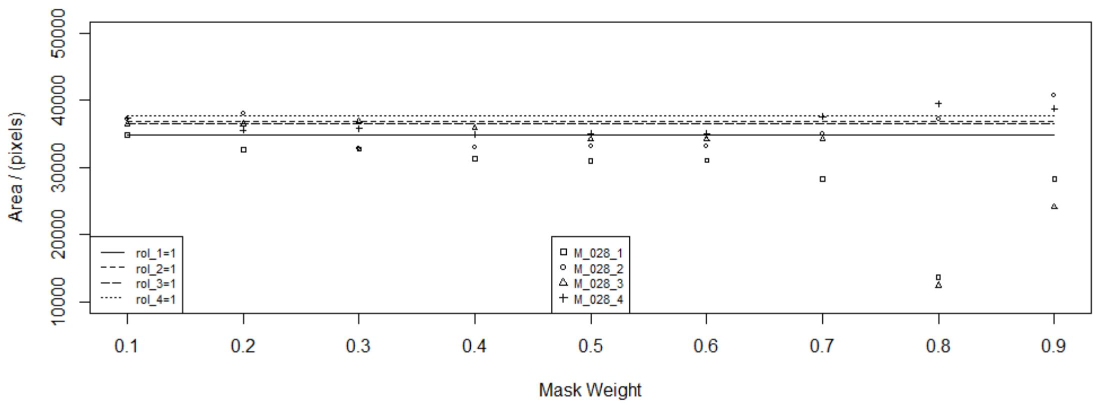


Figure B.9. *M_028* area results. *Unsharp Mask* radius equal to 9 pixels.

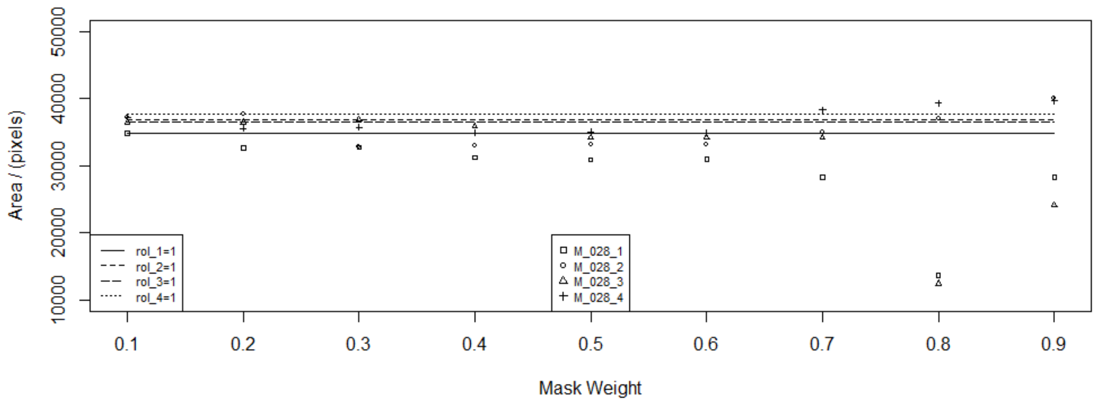


Figure B.10. *M_028* area results. *Unsharp Mask* radius equal to 10 pixels.

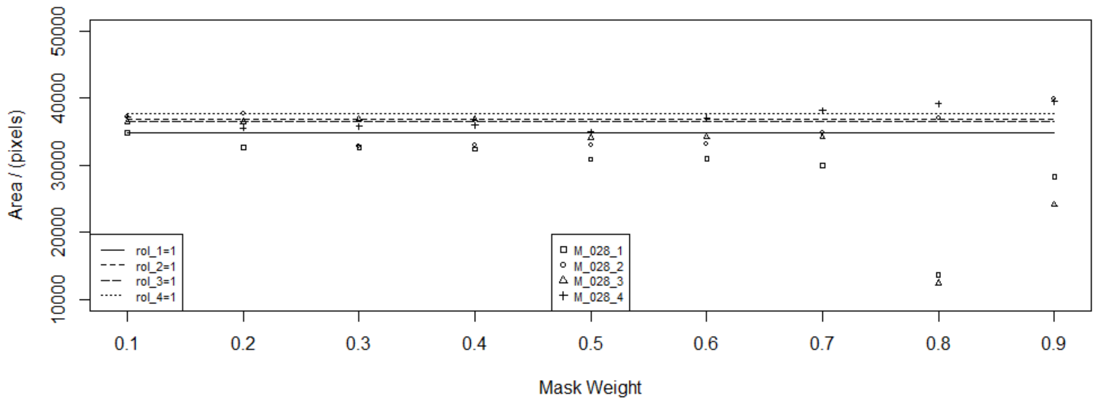


Figure B.11. *M_028* area results. *Unsharp Mask* radius equal to 11 pixels.

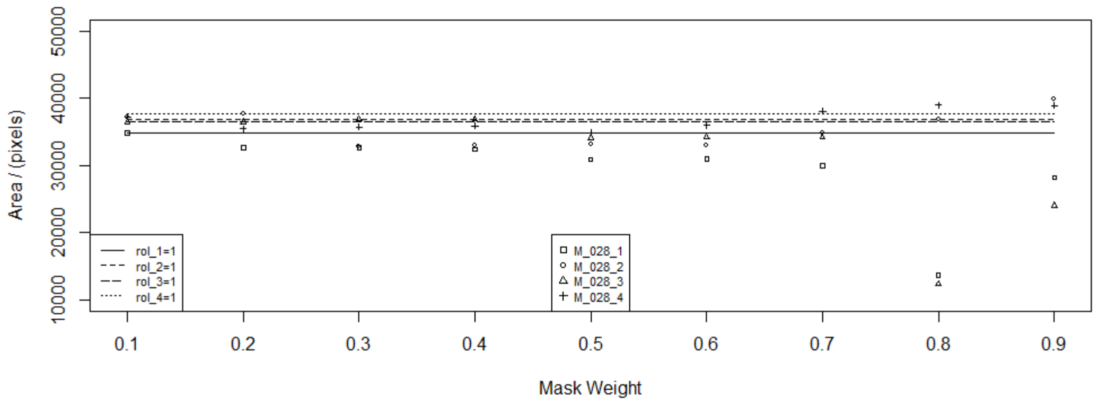


Figure B.12. *M_028* area results. *Unsharp Mask* radius equal to 12 pixels.

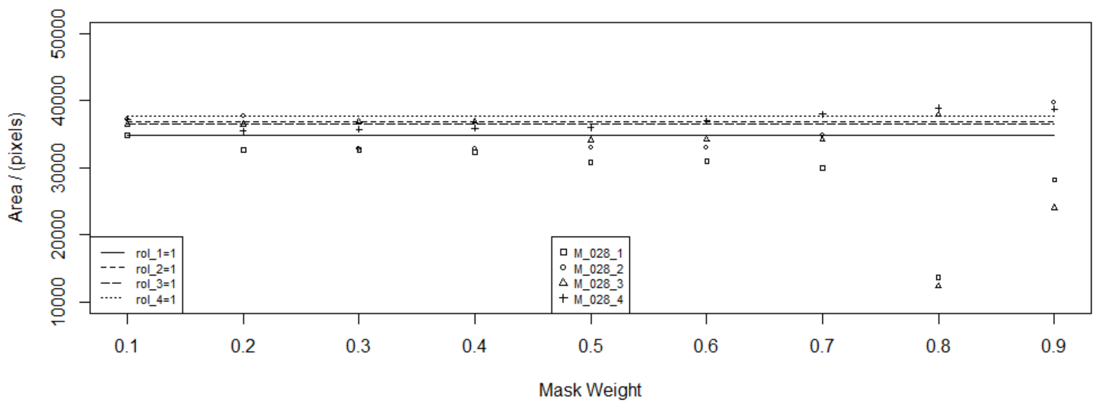


Figure B.13. *M_028* area results. *Unsharp Mask* radius equal to 13 pixels.

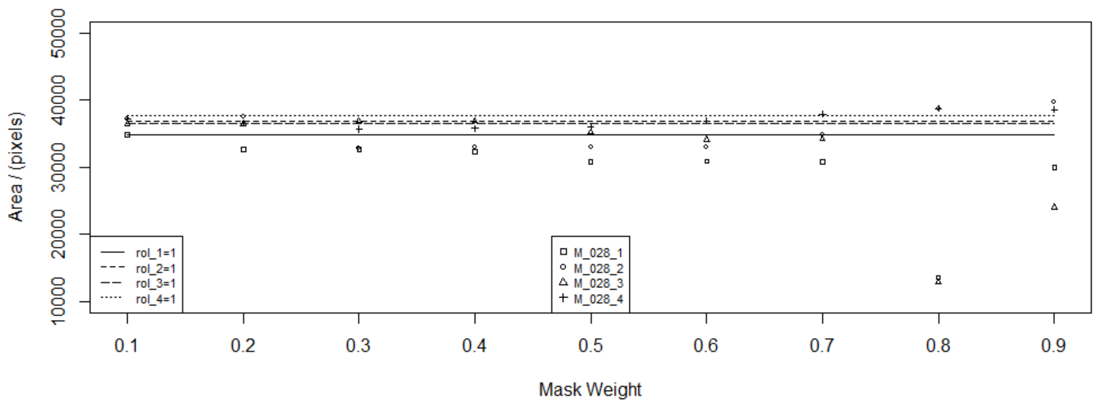


Figure B.14. *M_028* area results. *Unsharp Mask* radius equal to 14 pixels.

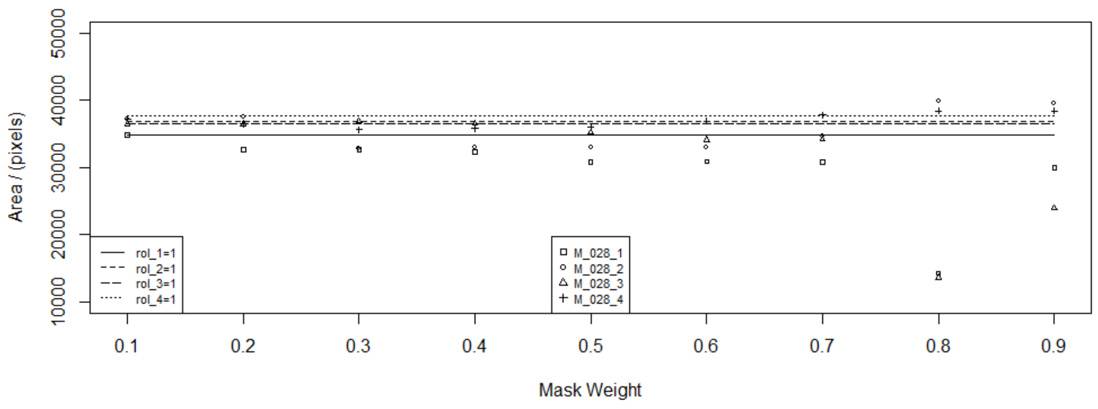


Figure B.15. *M_028* area results. *Unsharp Mask* radius equal to 15 pixels.

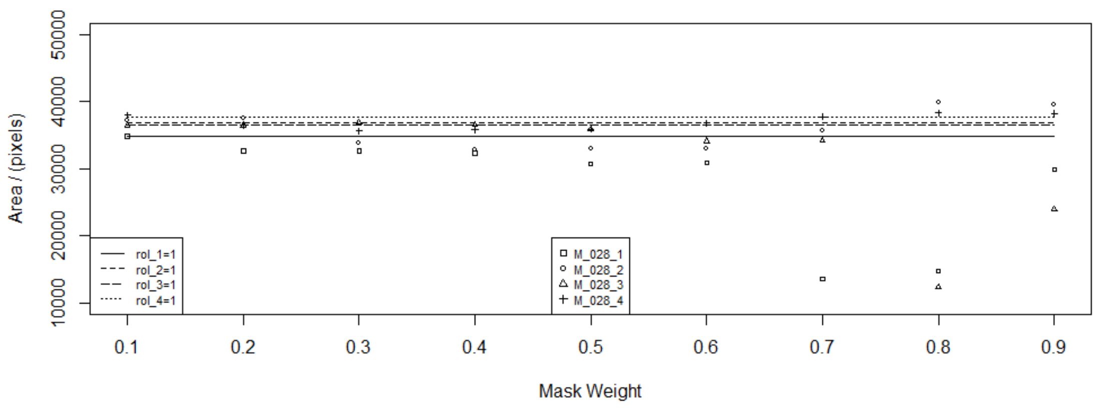


Figure B.16. *M_028* area results. *Unsharp Mask* radius equal to 16 pixels.

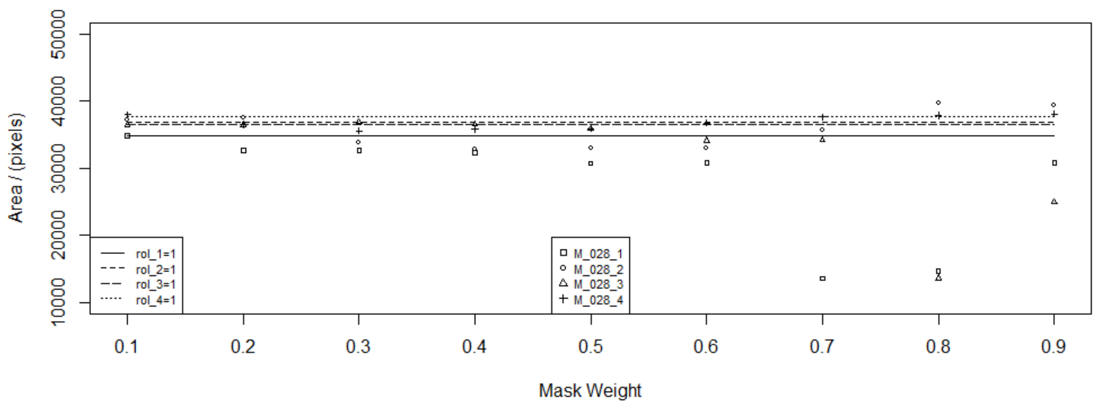


Figure B.17. *M_028* area results. *Unsharp Mask* radius equal to 17 pixels.

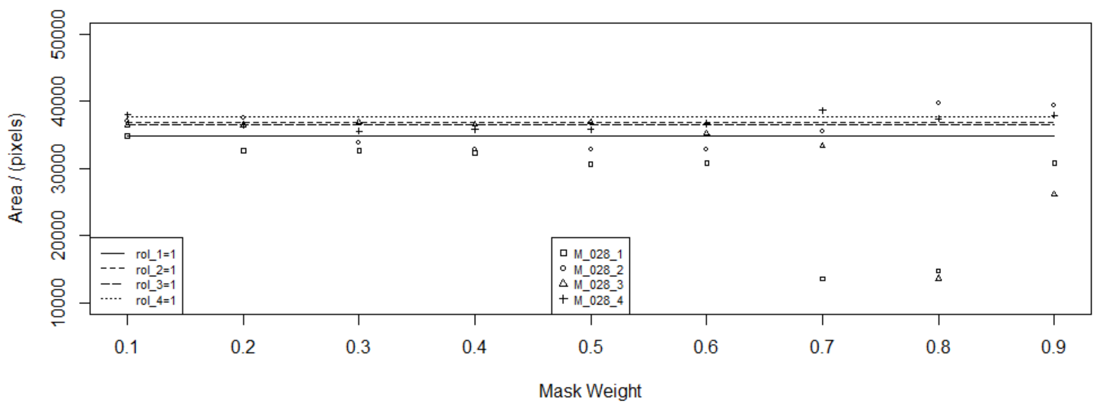


Figure B.18. *M_028* area results. *Unsharp Mask* radius equal to 18 pixels.

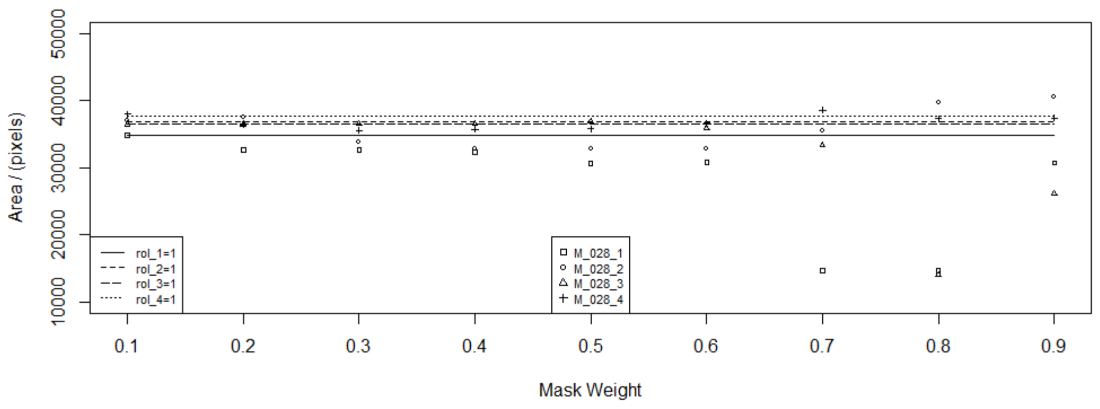


Figure B.19. *M_028* area results. *Unsharp Mask* radius equal to 19 pixels.

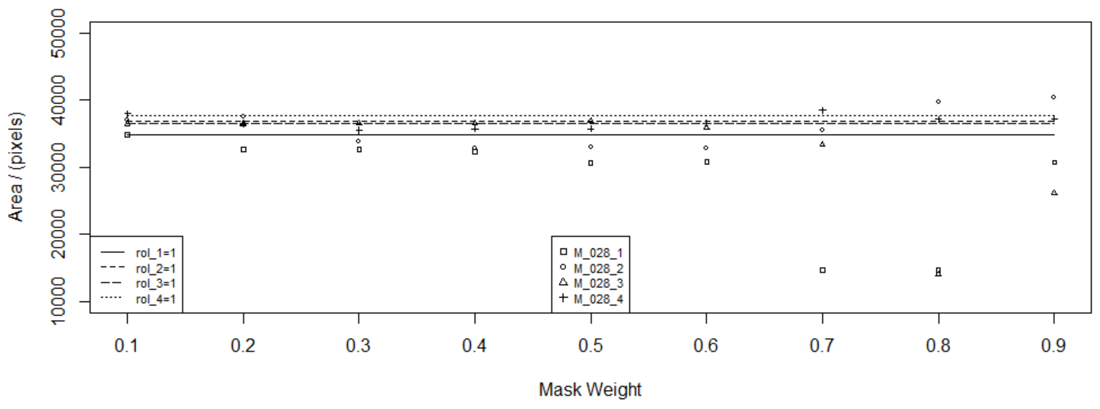


Figure B.20. *M_028* area results. *Unsharp Mask* radius equal to 20 pixels.

Appendix C – *Group 2* Results 2

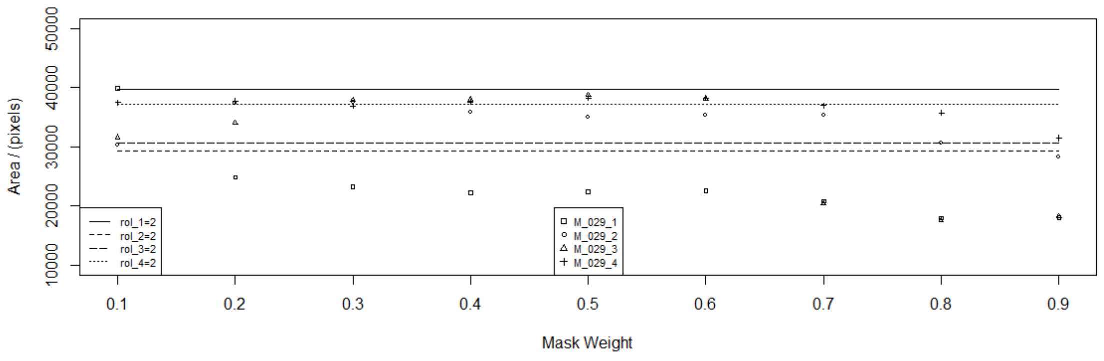


Figure C.1. *M_029* area results. *Unsharp Mask* radius equal to 1 pixels.

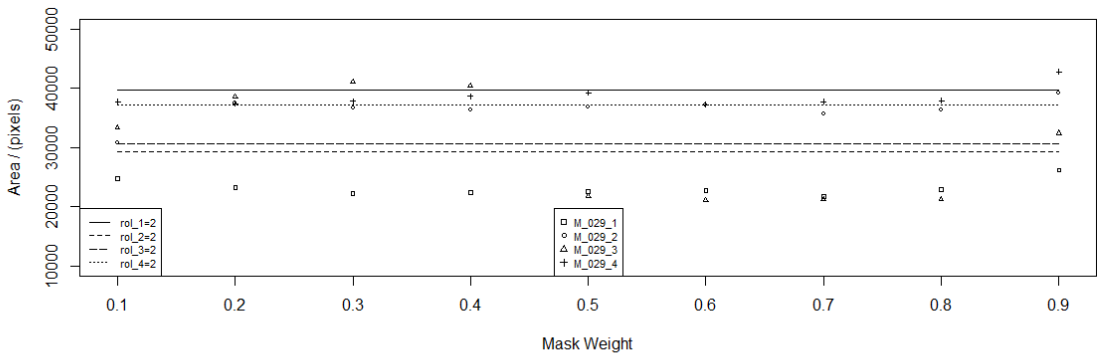


Figure C.2. *M_029* area results. *Unsharp Mask* radius equal to 2 pixels.

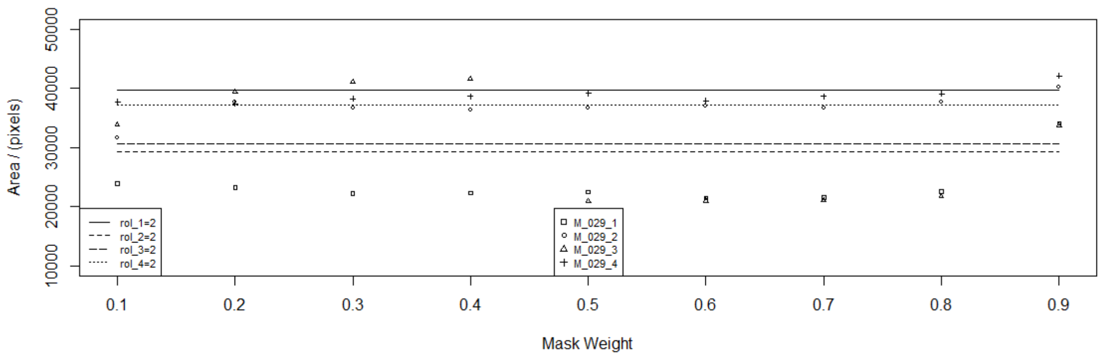


Figure C.3. *M_029* area results. *Unsharp Mask* radius equal to 3 pixels.

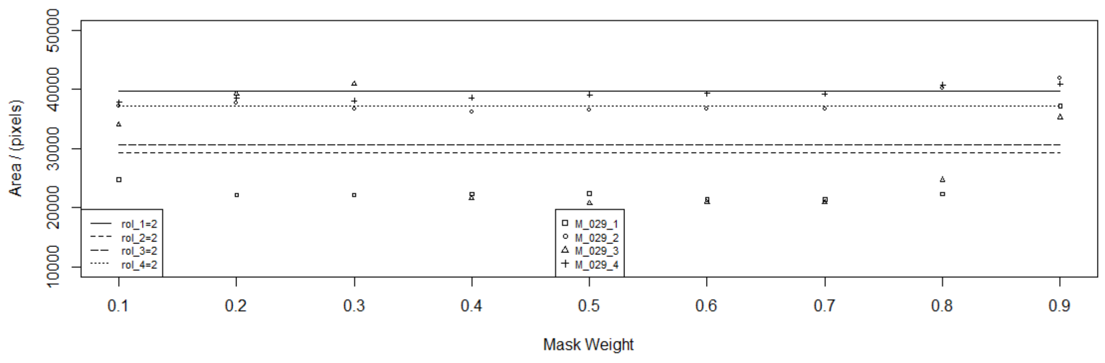


Figure C.4. *M_029* area results. *Unsharp Mask* radius equal to 4 pixels.

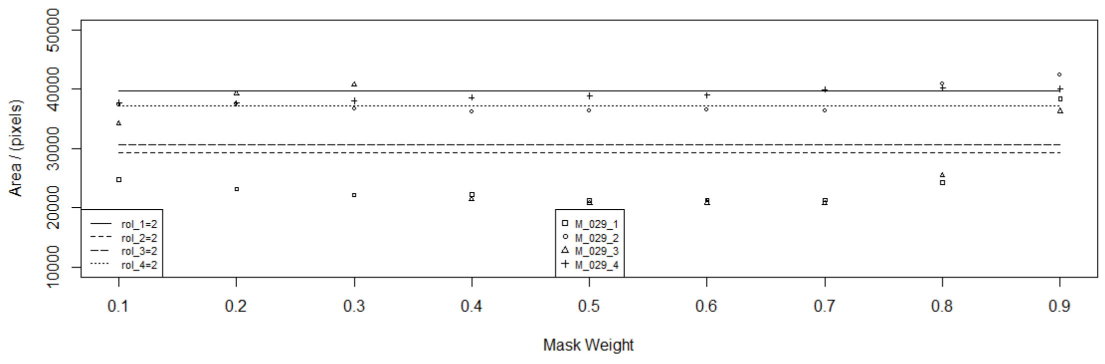


Figure C.5. *M_029* area results. *Unsharp Mask* radius equal to 5 pixels.

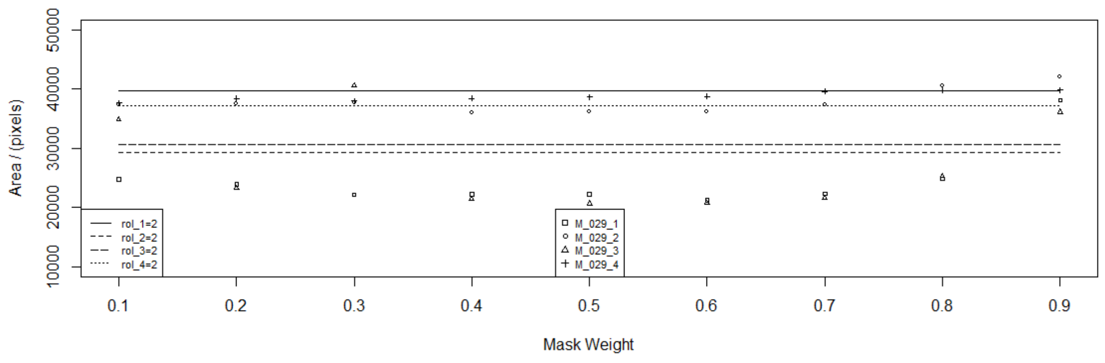


Figure C.6. *M_029* area results. *Unsharp Mask* radius equal to 6 pixels.

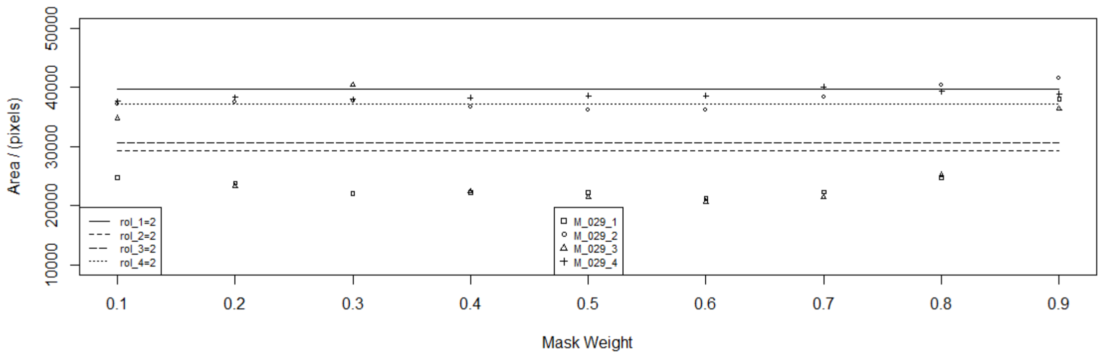


Figure C.7. *M_029* area results. *Unsharp Mask* radius equal to 7 pixels.

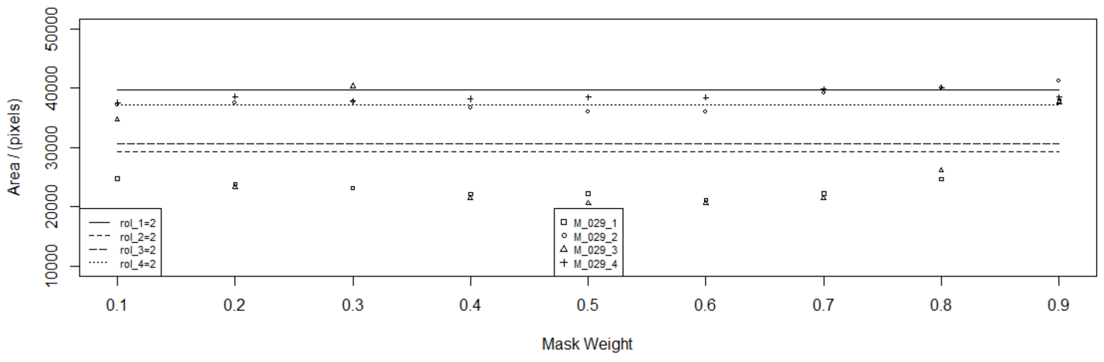


Figure C.8. *M_029* area results. *Unsharp Mask* radius equal to 8 pixels.

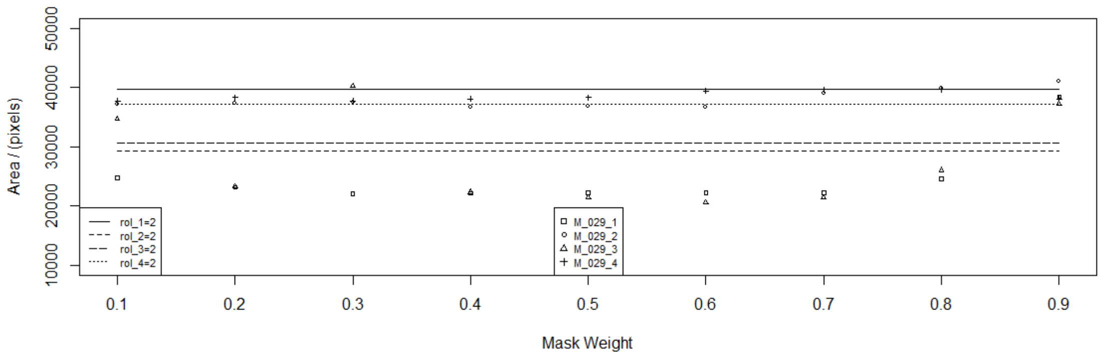


Figure C.9. *M_029* area results. *Unsharp Mask* radius equal to 9 pixels.

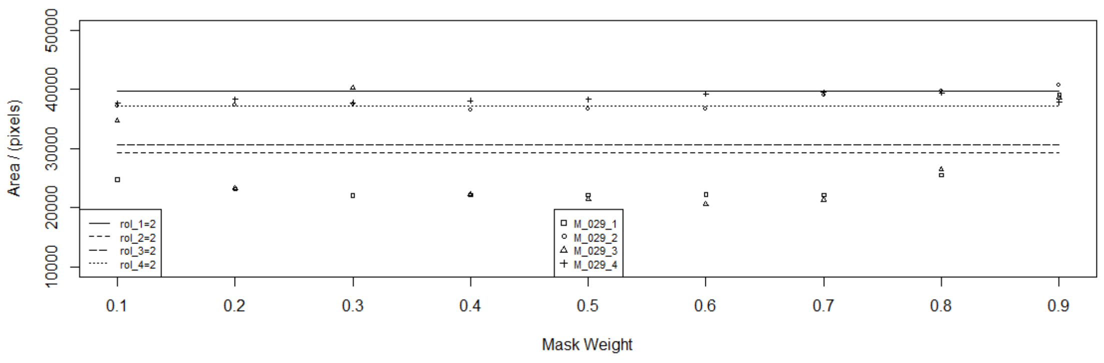


Figure C.10. *M_029* area results. *Unsharp Mask* radius equal to 10 pixels.

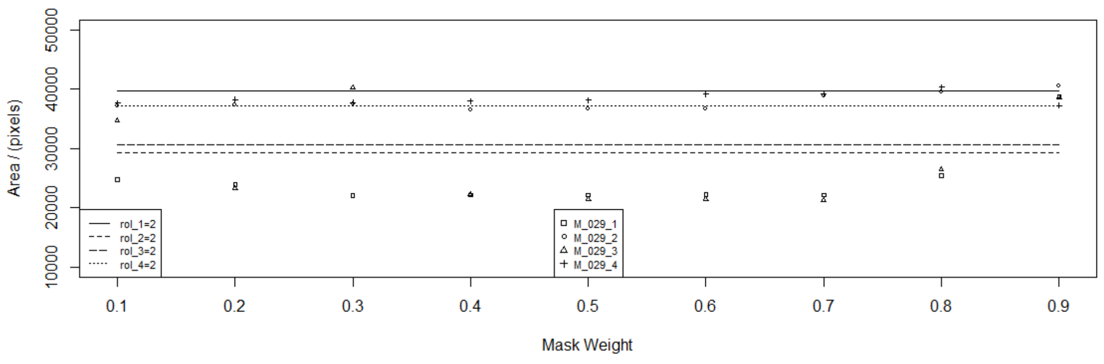


Figure C.11. *M_029* area results. *Unsharp Mask* radius equal to 11 pixels.

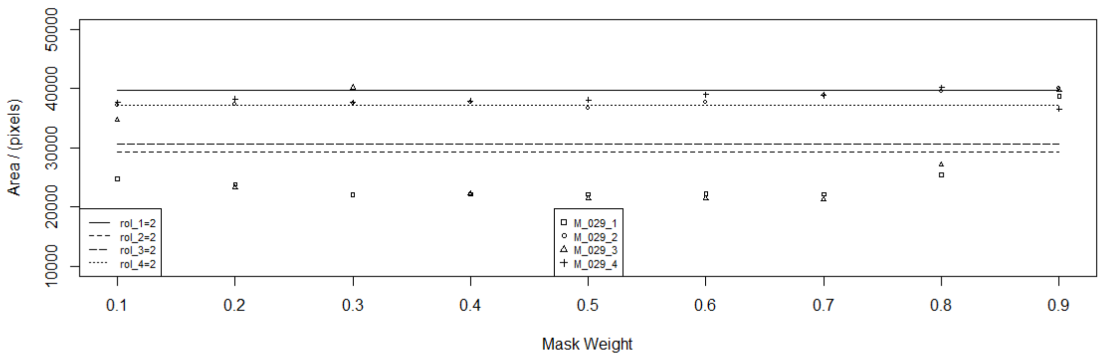


Figure C.12. *M_029* area results. *Unsharp Mask* radius equal to 12 pixels.

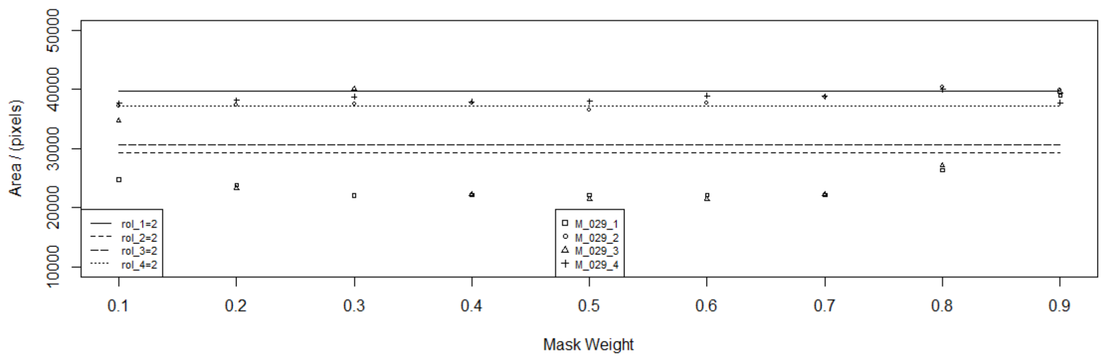


Figure C.13. *M_029* area results. *Unsharp Mask* radius equal to 13 pixels.

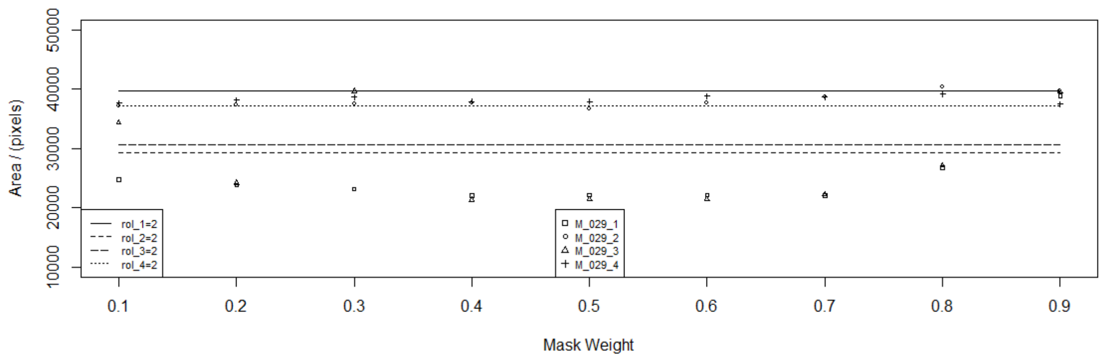


Figure C.14. *M_029* area results. *Unsharp Mask* radius equal to 14 pixels.

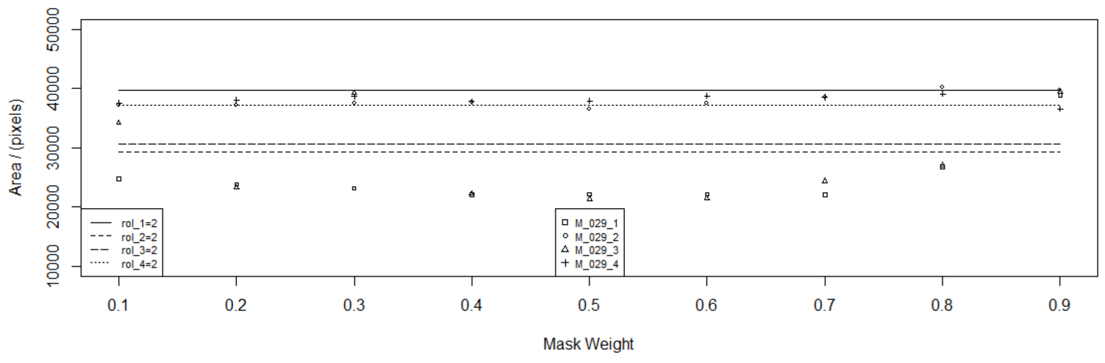


Figure C.15. *M_029* area results. *Unsharp Mask* radius equal to 15 pixels.

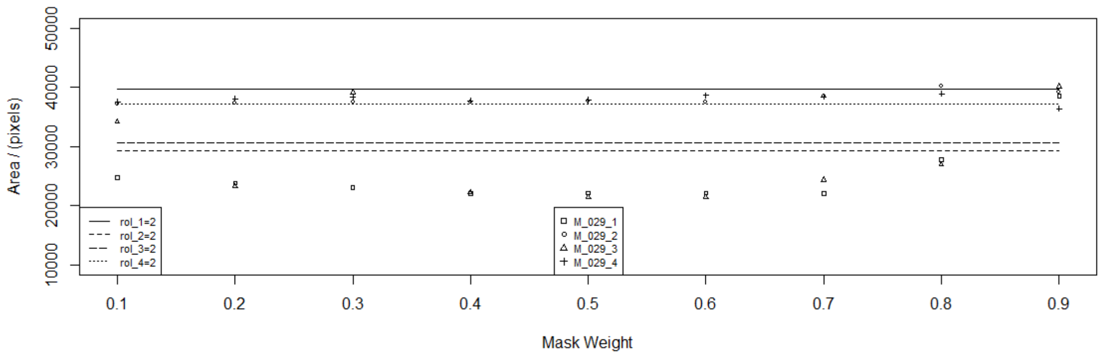


Figure C.16. *M_029* area results. *Unsharp Mask* radius equal to 16 pixels.

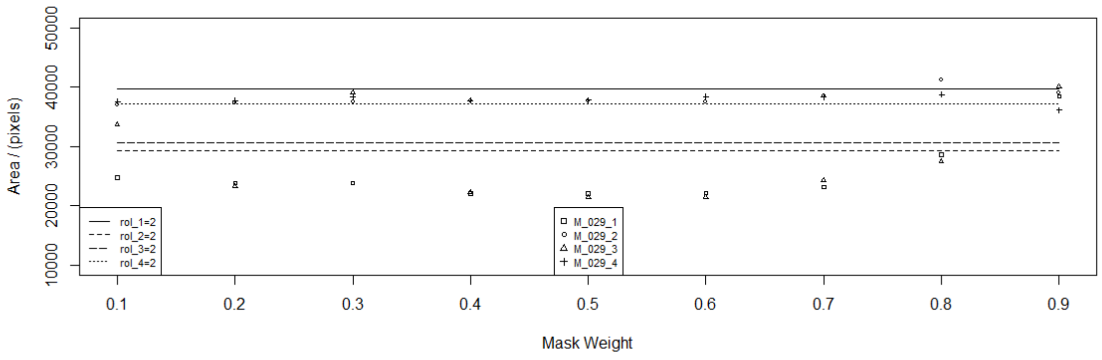


Figure C.17. *M_029* area results. *Unsharp Mask* radius equal to 17 pixels.

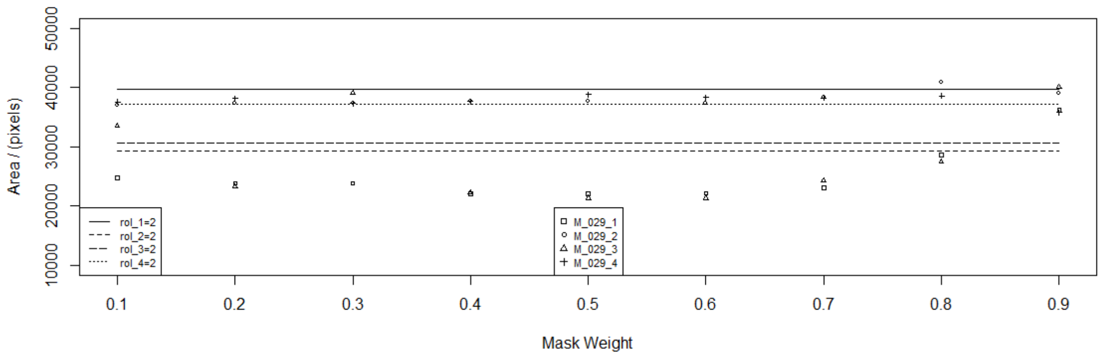


Figure C.18. *M_029* area results. *Unsharp Mask* radius equal to 18 pixels.

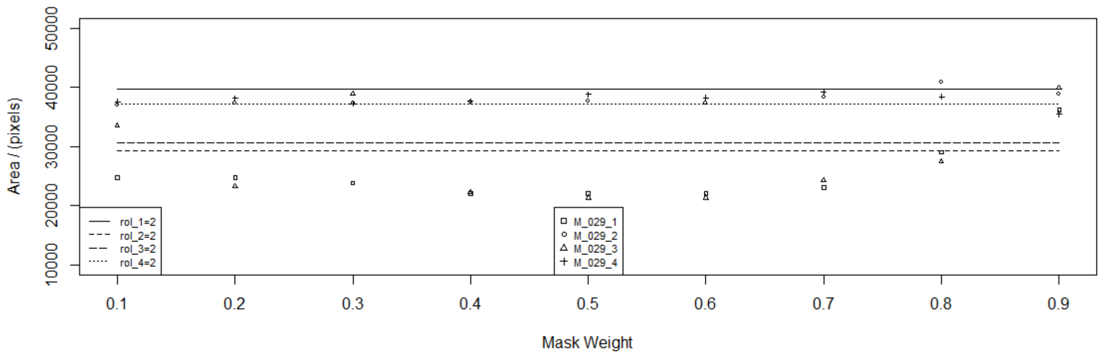


Figure C.19. *M_029* area results. *Unsharp Mask* radius equal to 19 pixels.

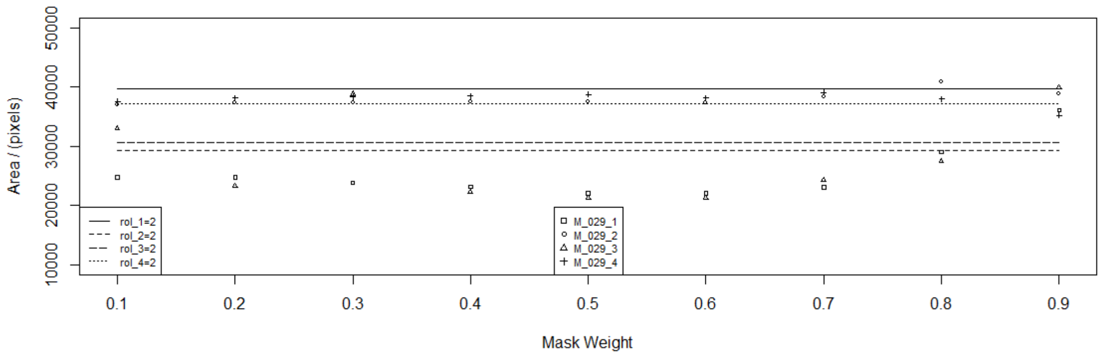


Figure C.20. *M_029* area results. *Unsharp Mask* radius equal to 20 pixels.

Appendix D – *Group 2* Results 3

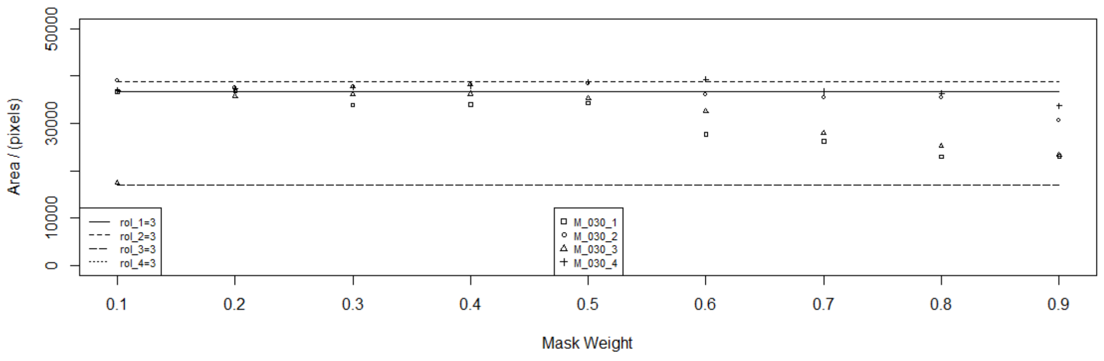


Figure D.1. *M_030* area results. *Unsharp Mask* radius equal to 1 pixels.

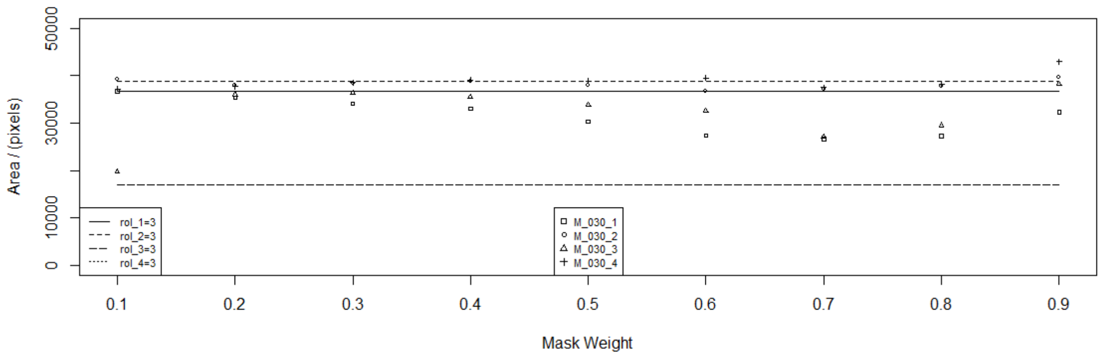


Figure D.2. *M_030* area results. *Unsharp Mask* radius equal to 2 pixels.

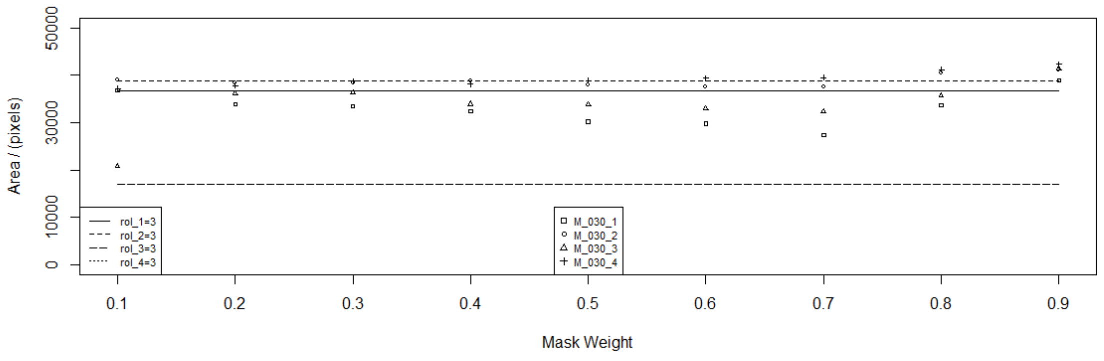


Figure D.3. *M_030* area results. *Unsharp Mask* radius equal to 3 pixels.

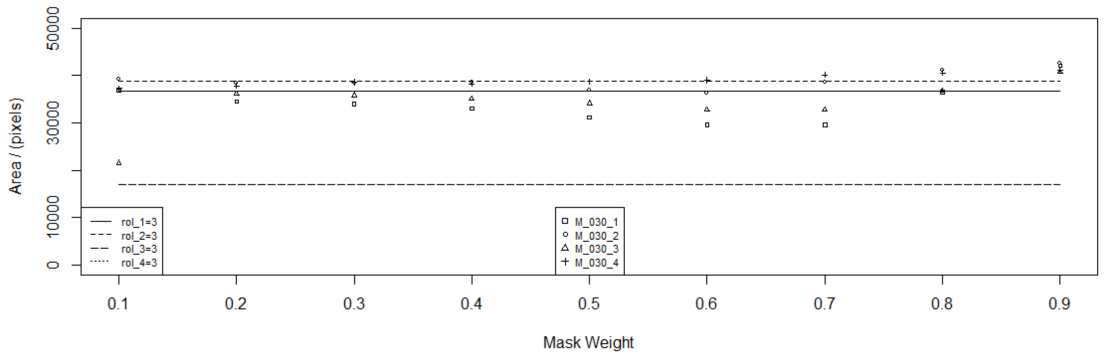


Figure D.4. *M_030* area results. *Unsharp Mask* radius equal to 4 pixels.

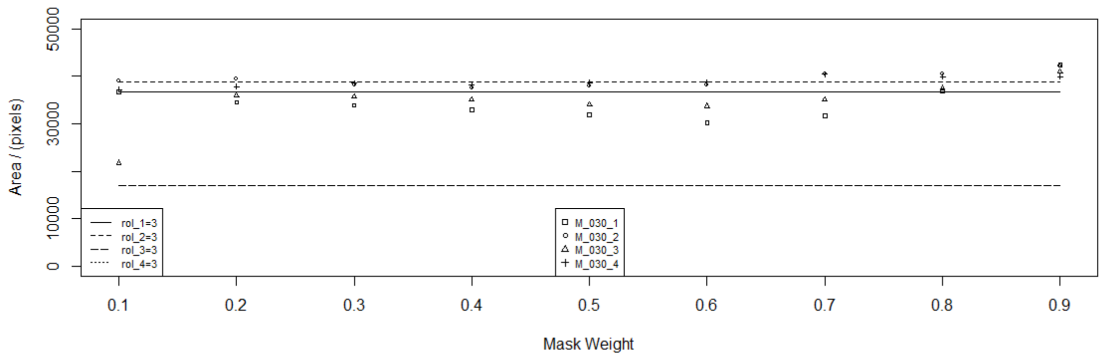


Figure D.5. *M_030* area results. *Unsharp Mask* radius equal to 5 pixels.

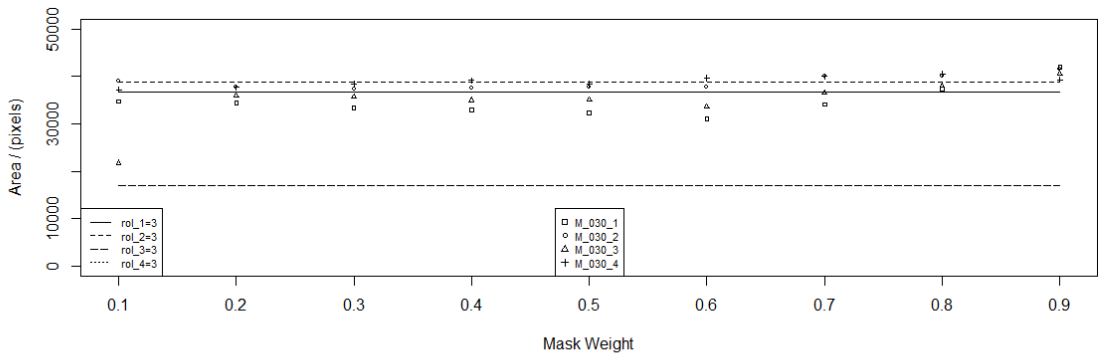


Figure D.6. *M_030* area results. *Unsharp Mask* radius equal to 6 pixels.

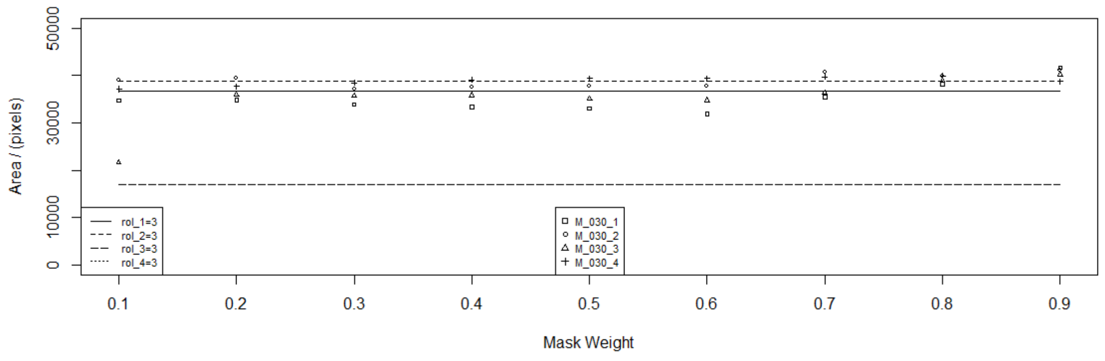


Figure D.7. *M_030* area results. *Unsharp Mask* radius equal to 7 pixels.

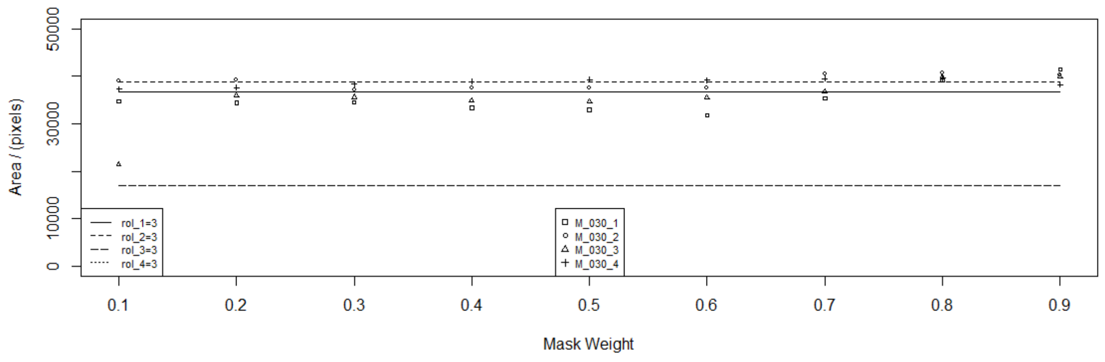


Figure D.8. *M_030* area results. *Unsharp Mask* radius equal to 8 pixels.

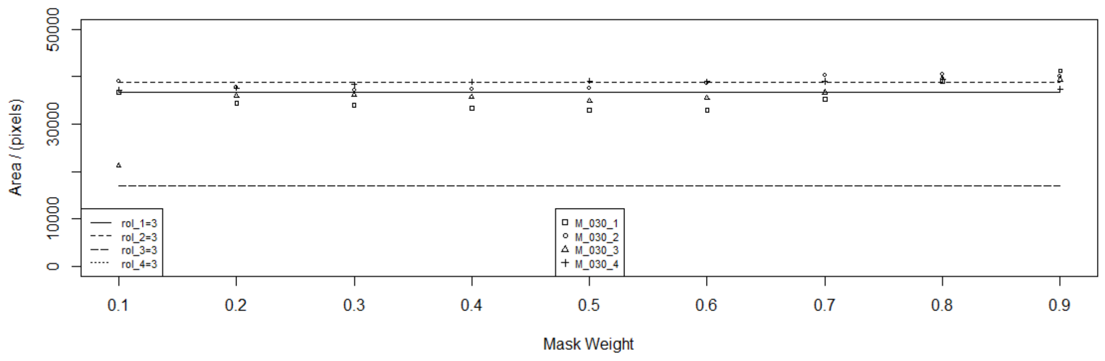


Figure D.9. *M_030* area results. *Unsharp Mask* radius equal to 9 pixels.

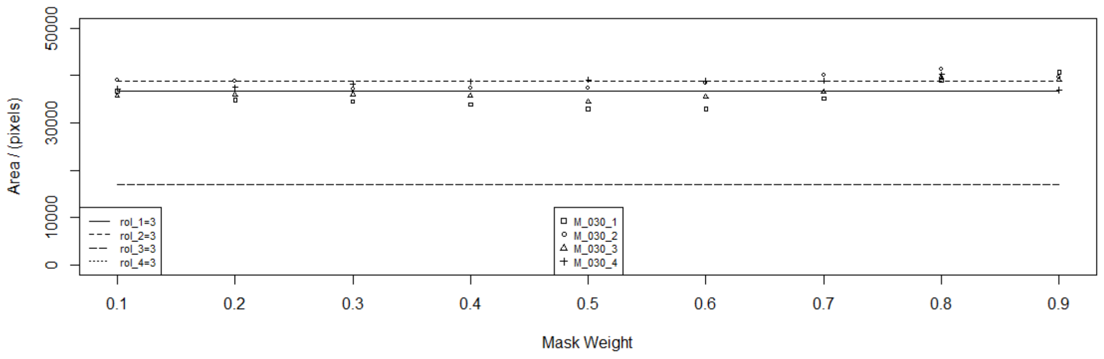


Figure D.10. *M_030* area results. *Unsharp Mask* radius equal to 10 pixels.

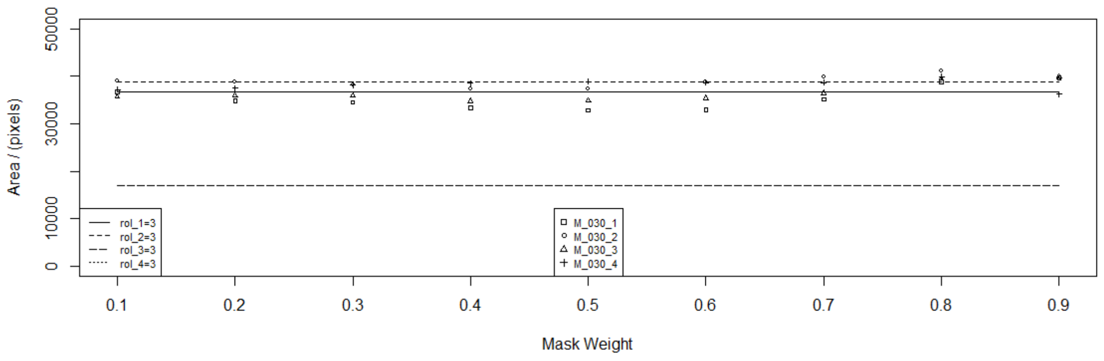


Figure D.11. *M_030* area results. *Unsharp Mask* radius equal to 11 pixels.

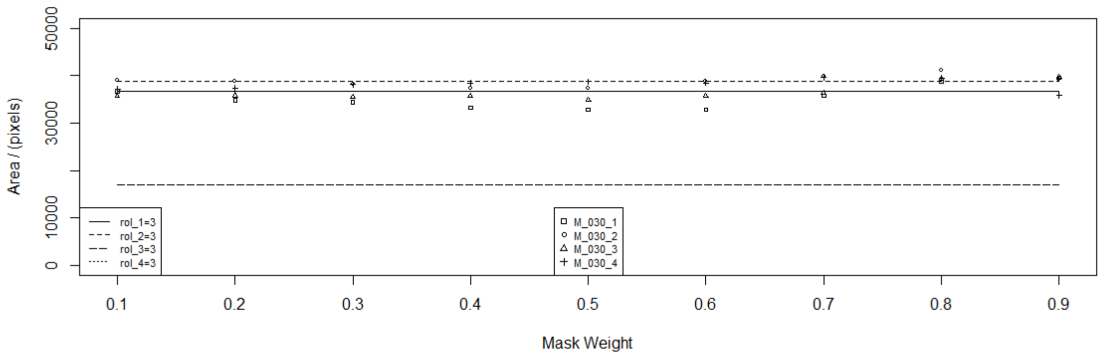


Figure D.12. *M_030* area results. *Unsharp Mask* radius equal to 12 pixels.

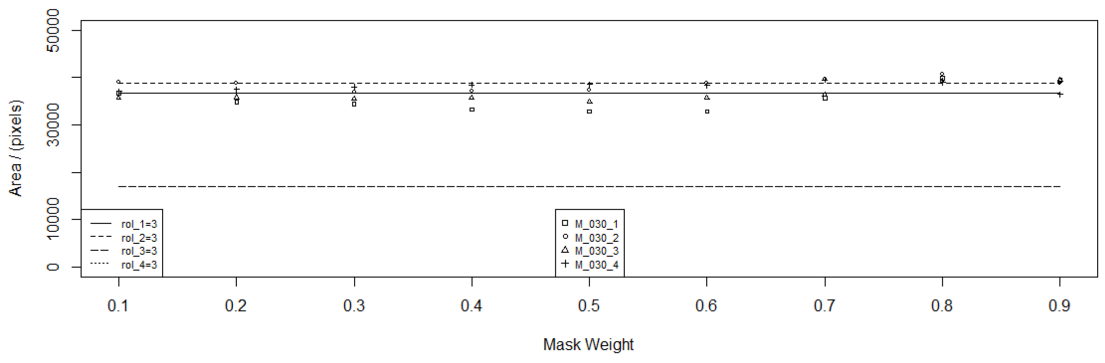


Figure D.13. *M_030* area results. *Unsharp Mask* radius equal to 13 pixels.

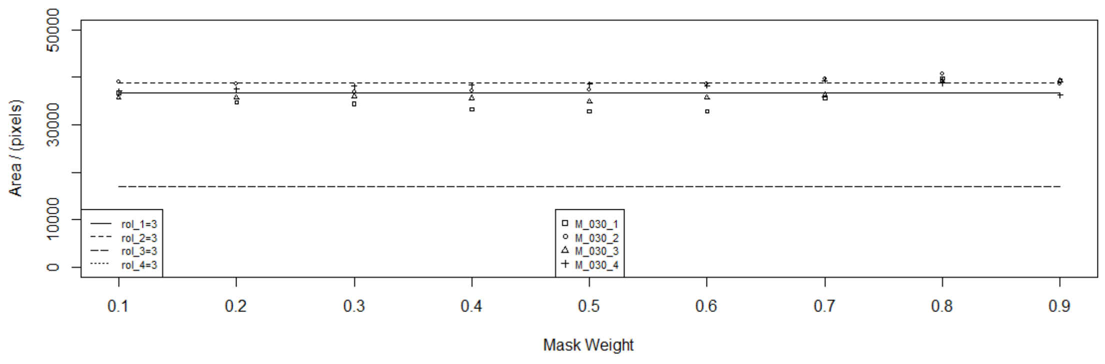


Figure D.14. *M_030* area results. *Unsharp Mask* radius equal to 14 pixels.

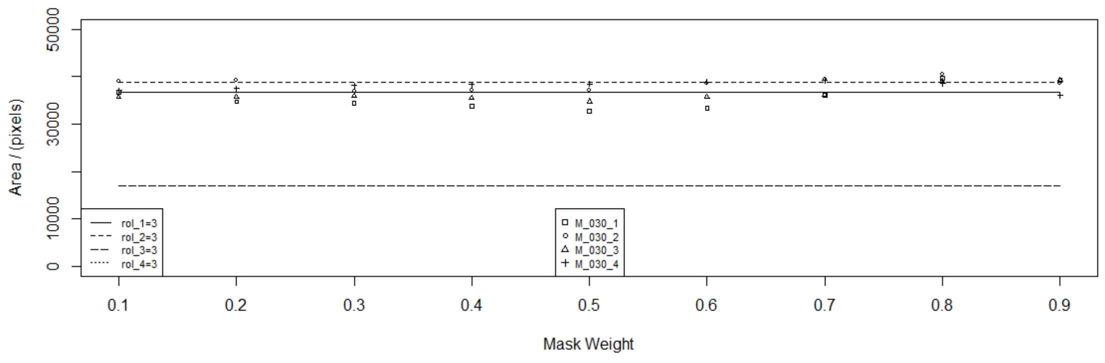


Figure D.15. *M_030* area results. *Unsharp Mask* radius equal to 15 pixels.

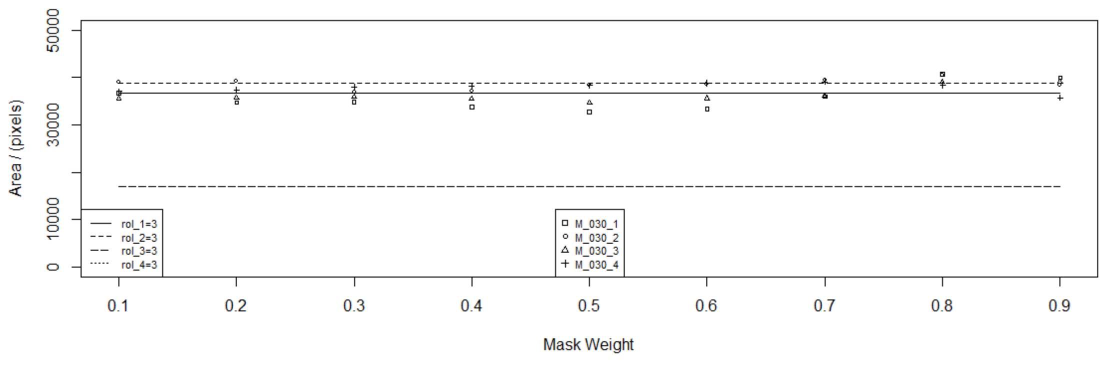


Figure D.16. *M_030* area results. *Unsharp Mask* radius equal to 16 pixels.

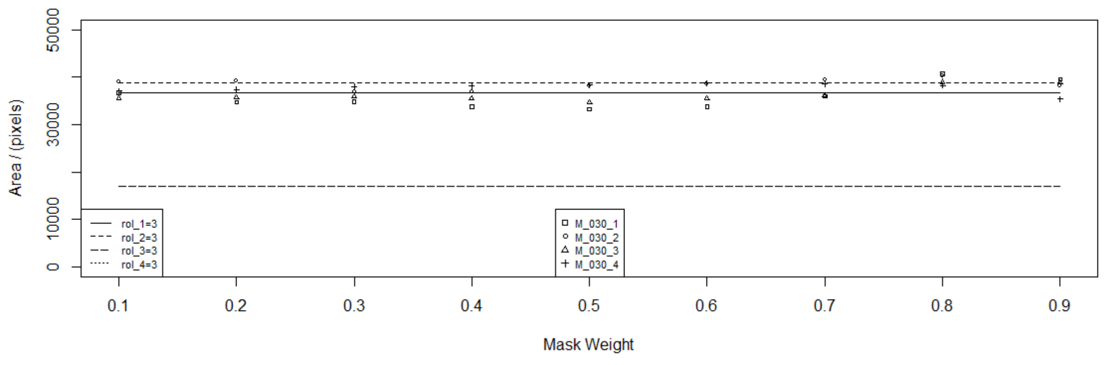


Figure D.17. *M_030* area results. *Unsharp Mask* radius equal to 17 pixels.

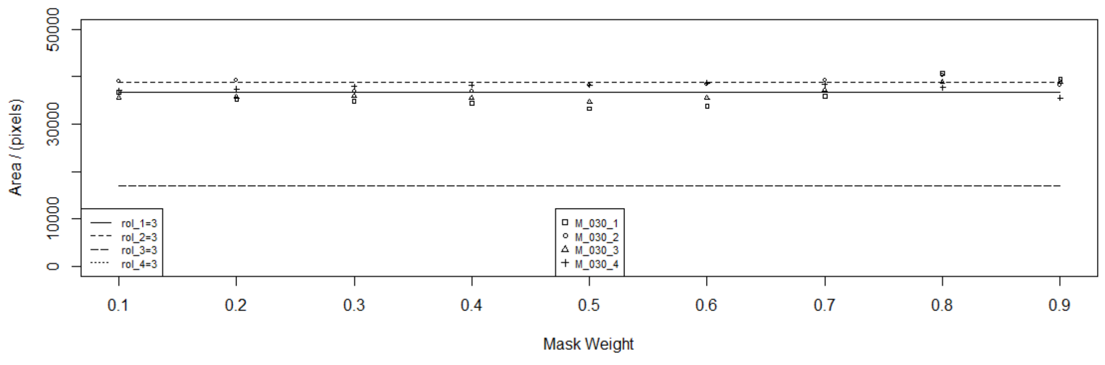


Figure D.18. *M_030* area results. *Unsharp Mask* radius equal to 18 pixels.

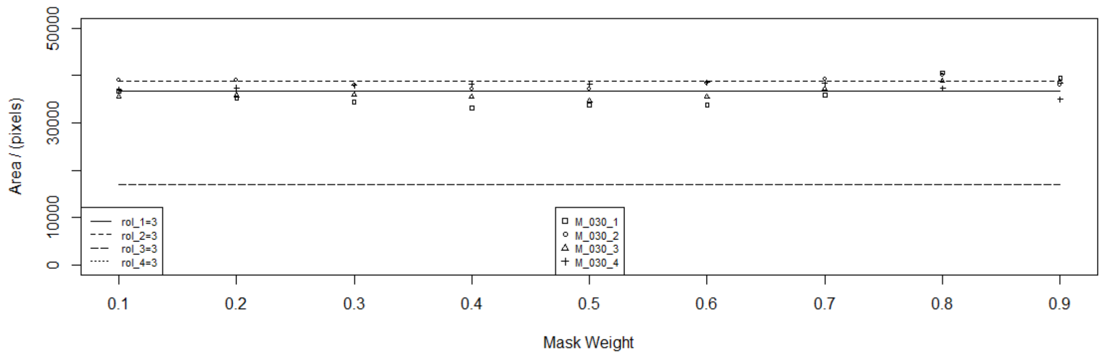


Figure D.19. *M_030* area results. *Unsharp Mask* radius equal to 19 pixels.

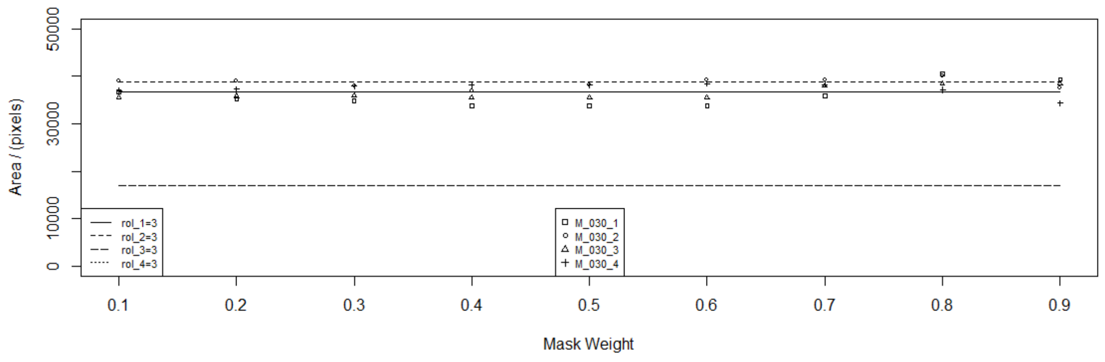


Figure D.20. *M_030* area results. *Unsharp Mask* radius equal to 20 pixels.

Appendix E – *Group 2* Results 4

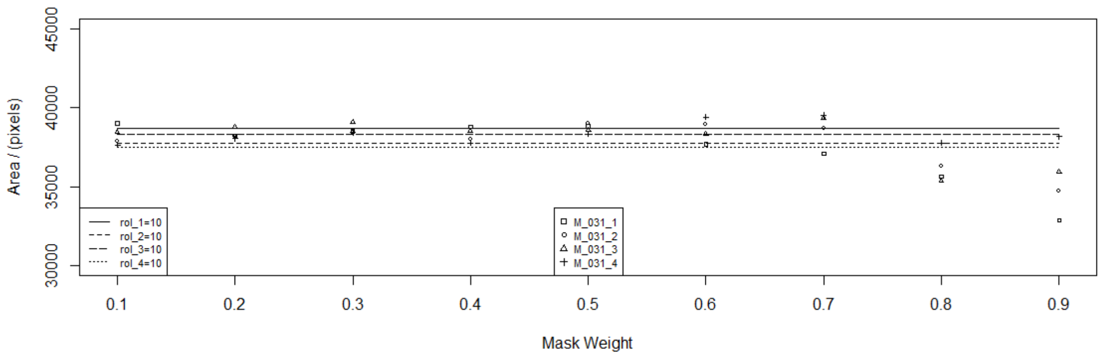


Figure E.1. *M_031* area results. *Unsharp Mask* radius equal to 1 pixels.

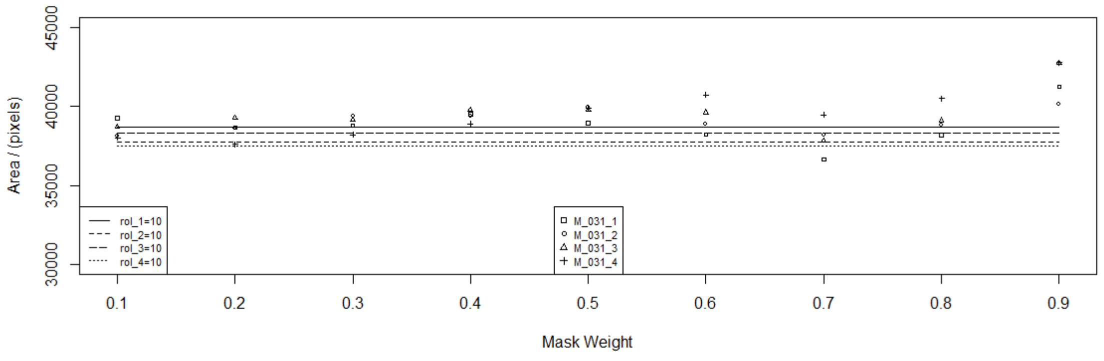


Figure E.2. *M_031* area results. *Unsharp Mask* radius equal to 1 pixels.

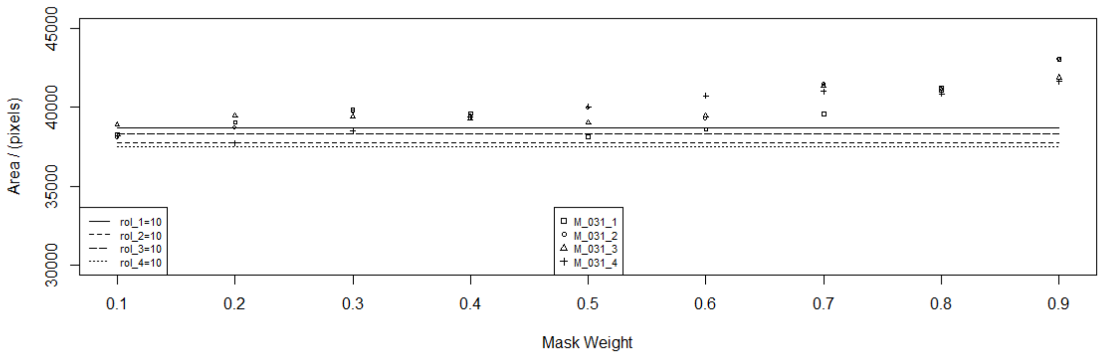


Figure E.3. *M_031* area results. *Unsharp Mask* radius equal to 3 pixels.

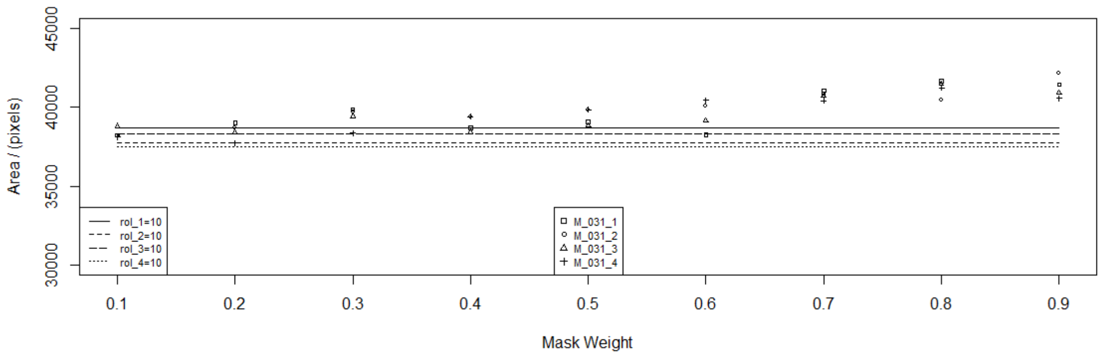


Figure E.4. *M_031* area results. *Unsharp Mask* radius equal to 4 pixels.

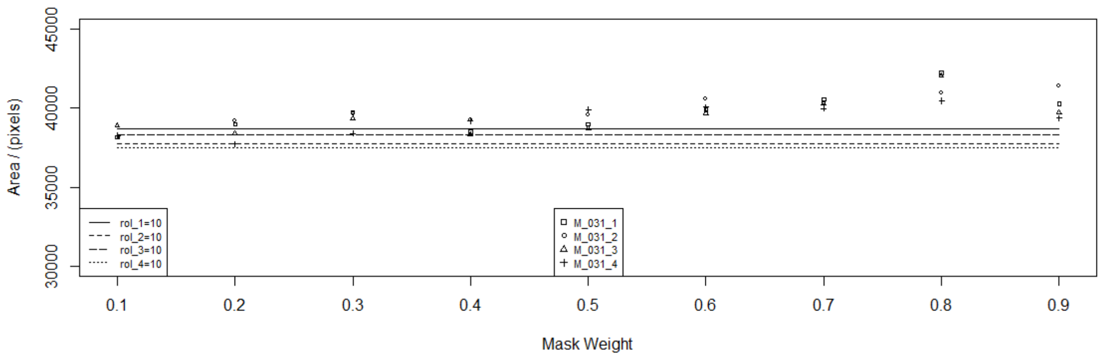


Figure E.5. *M_031* area results. *Unsharp Mask* radius equal to 5 pixels.

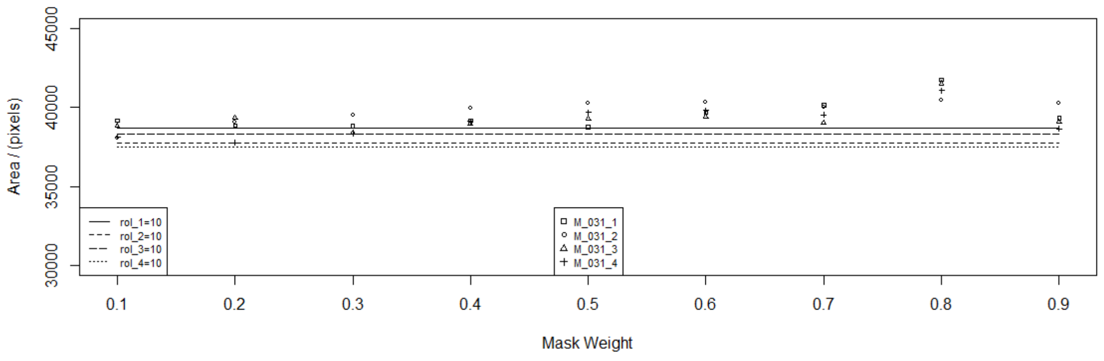


Figure E.6. *M_031* area results. *Unsharp Mask* radius equal to 6 pixels.

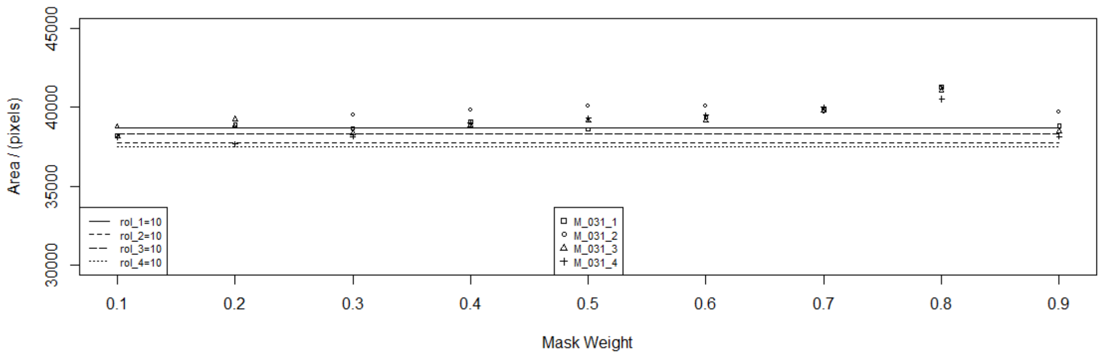


Figure E.7. *M_031* area results. *Unsharp Mask* radius equal to 7 pixels.

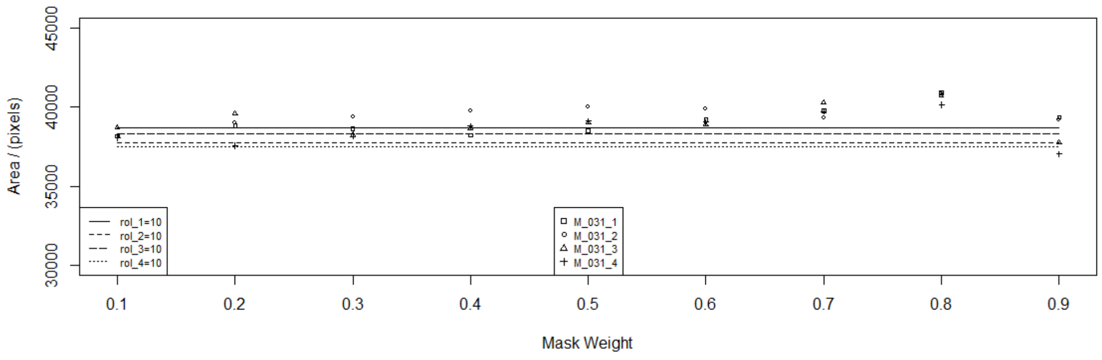


Figure E.8. *M_031* area results. *Unsharp Mask* radius equal to 8 pixels.

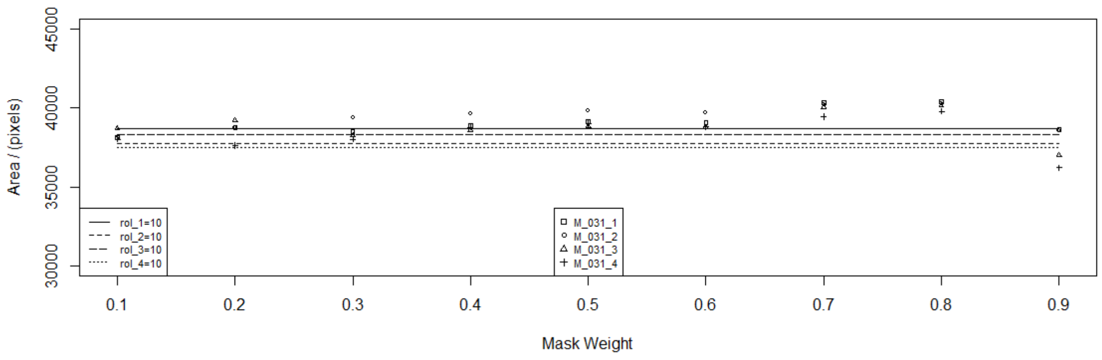


Figure E.9. *M_031* area results. *Unsharp Mask* radius equal to 9 pixels.

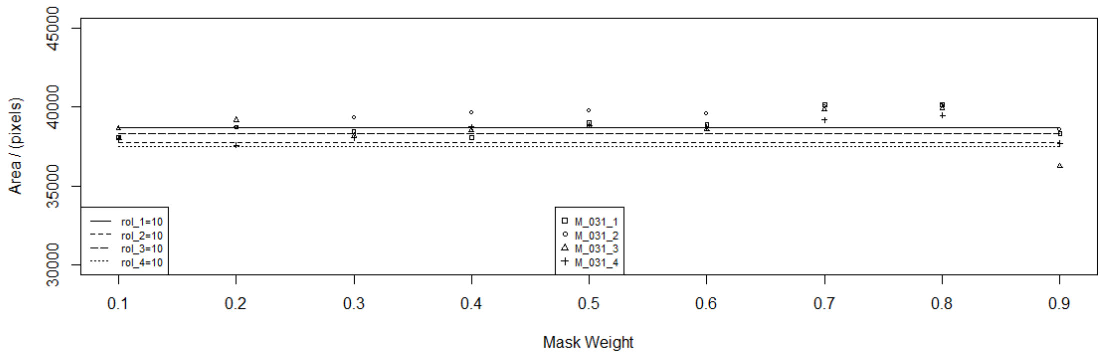


Figure E.10. *M_031* area results. *Unsharp Mask* radius equal to 10 pixels.

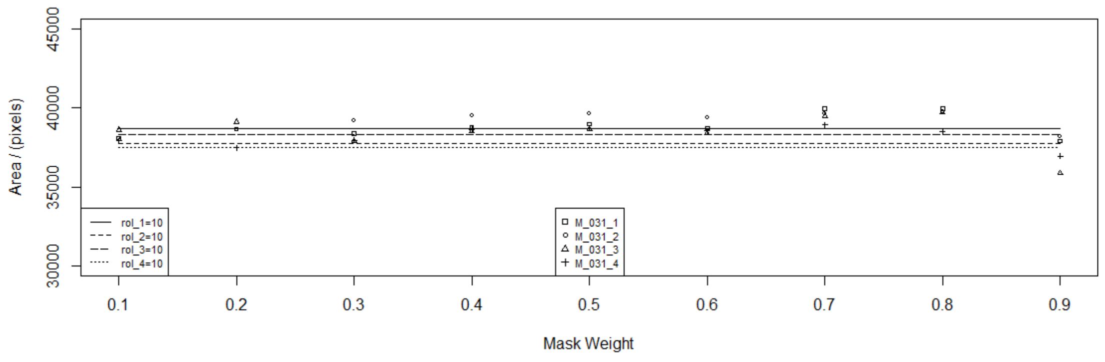


Figure E.11. *M_031* area results. *Unsharp Mask* radius equal to 11 pixels.

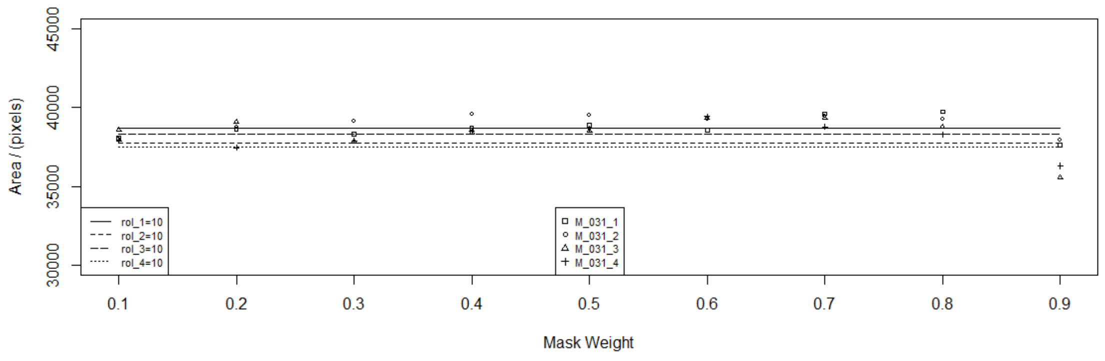


Figure E.12. *M_031* area results. *Unsharp Mask* radius equal to 12 pixels.

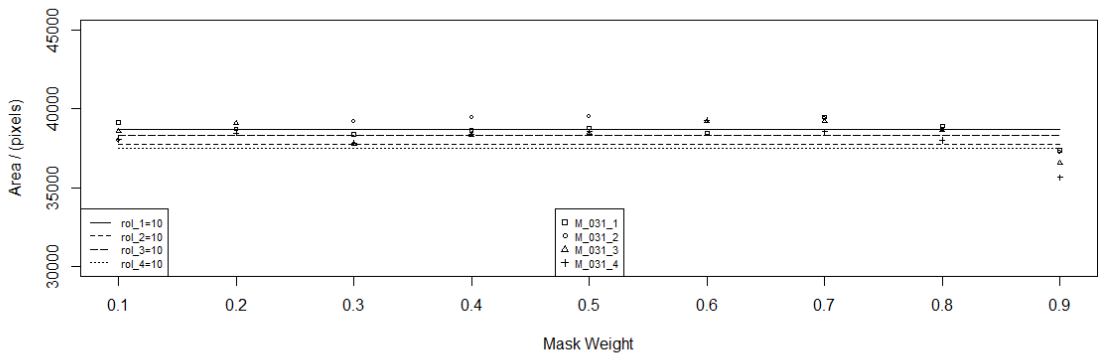


Figure E.13. *M_031* area results. *Unsharp Mask* radius equal to 13 pixels.

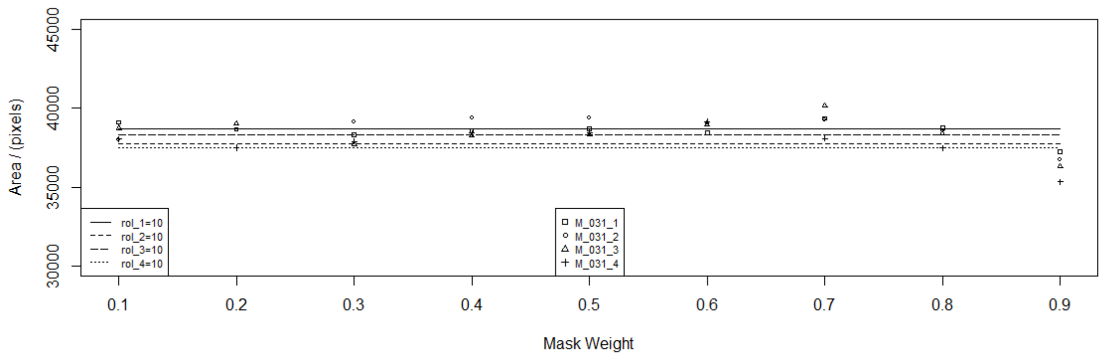


Figure E.14. *M_031* area results. *Unsharp Mask* radius equal to 14 pixels.

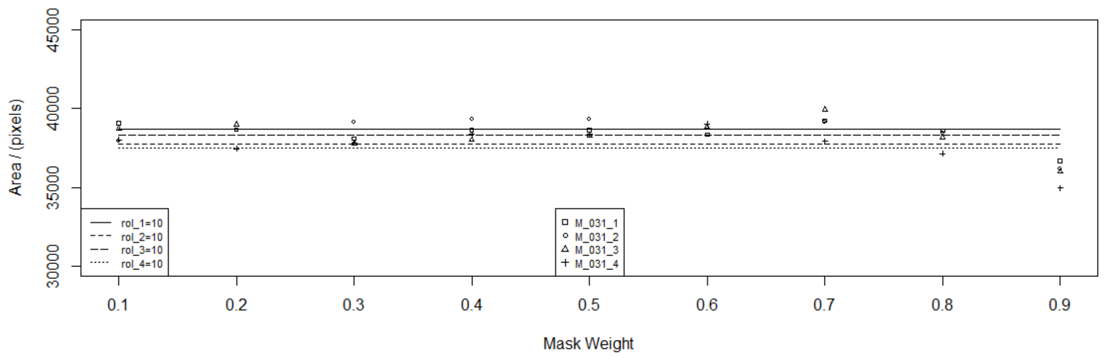


Figure E.15. *M_031* area results. *Unsharp Mask* radius equal to 15 pixels.

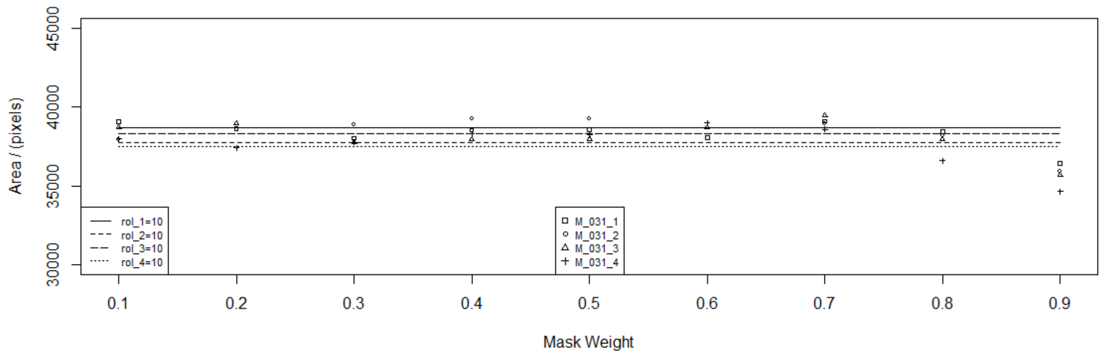


Figure E.16. *M_031* area results. *Unsharp Mask* radius equal to 16 pixels.

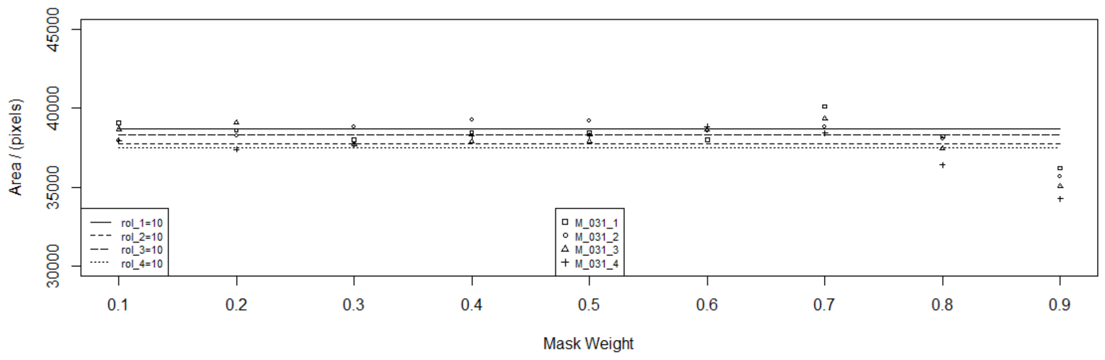


Figure E.17. *M_031* area results. *Unsharp Mask* radius equal to 17 pixels.

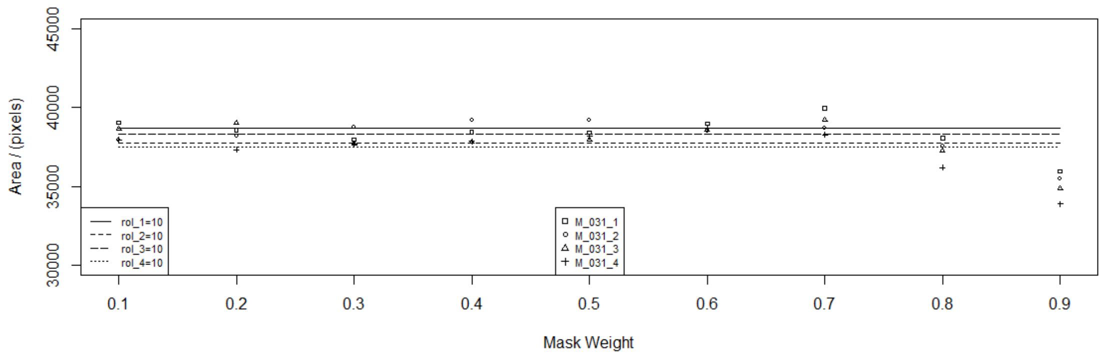


Figure E.18. *M_031* area results. *Unsharp Mask* radius equal to 18 pixels.

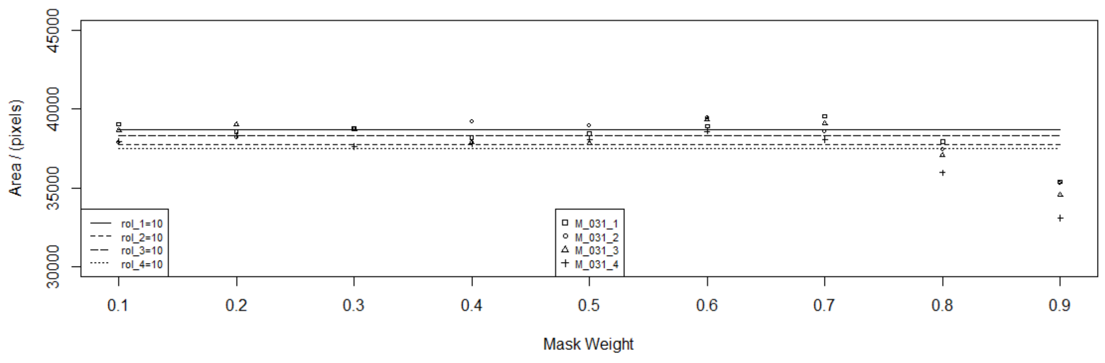


Figure E.19. *M_031* area results. *Unsharp Mask* radius equal to 19 pixels.

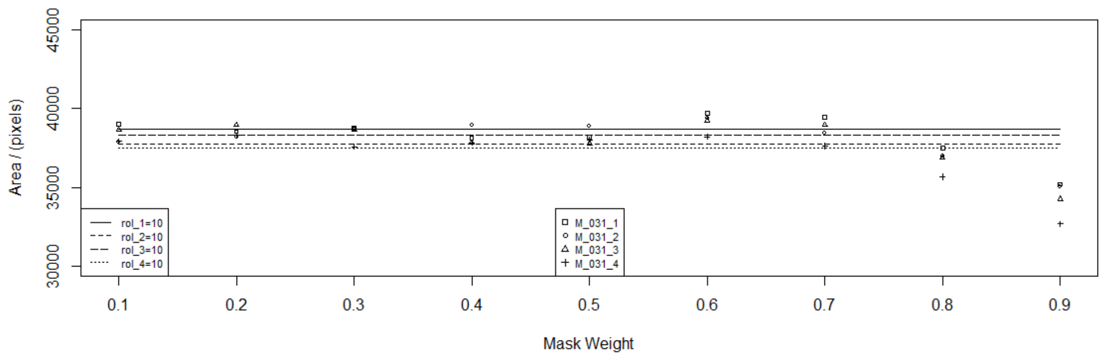


Figure E.20. *M_031* area results. *Unsharp Mask* radius equal to 20 pixels.

Appendix F – *Group 2* Results 5

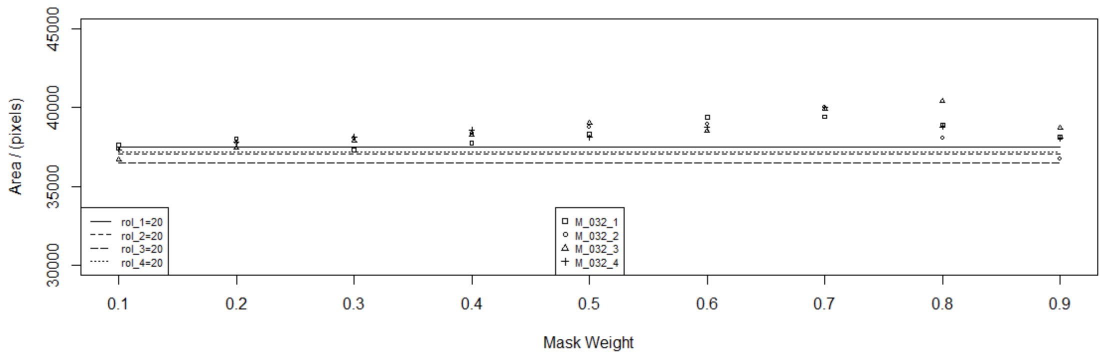


Figure F.1. *M_032* area results. *Unsharp Mask* radius equal to 1 pixels.

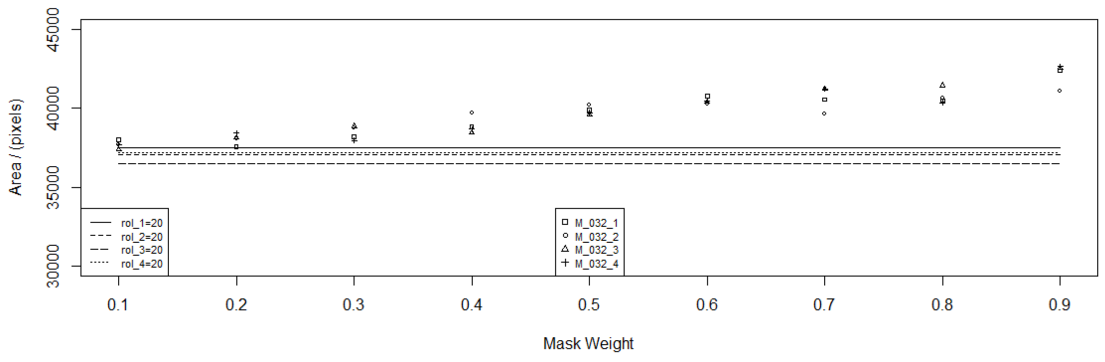


Figure F.2. *M_032* area results. *Unsharp Mask* radius equal to 2 pixels.

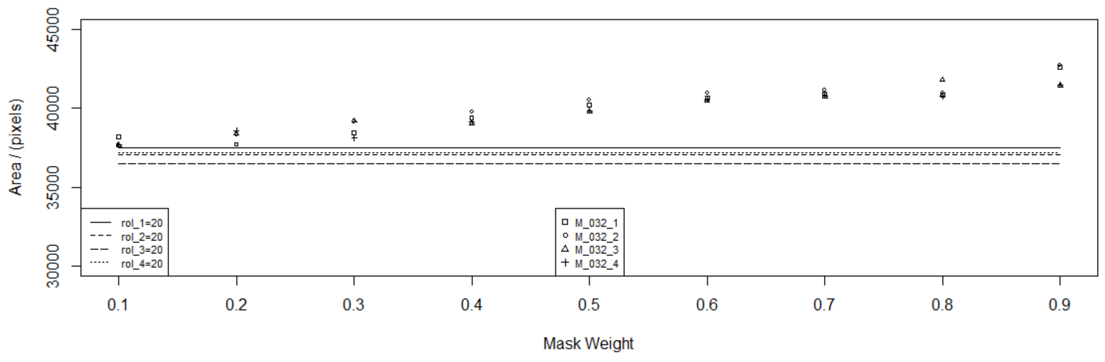


Figure F.3. *M_032* area results. *Unsharp Mask* radius equal to 3 pixels.

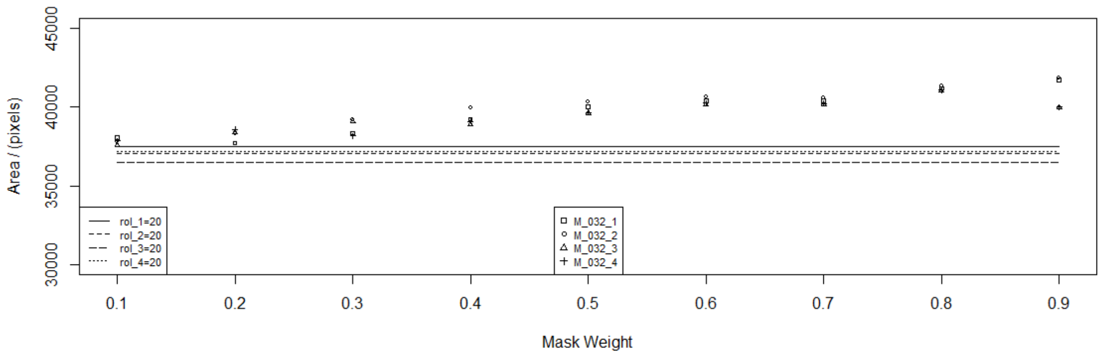


Figure F.3. *M_032* area results. *Unsharp Mask* radius equal to 4 pixels.

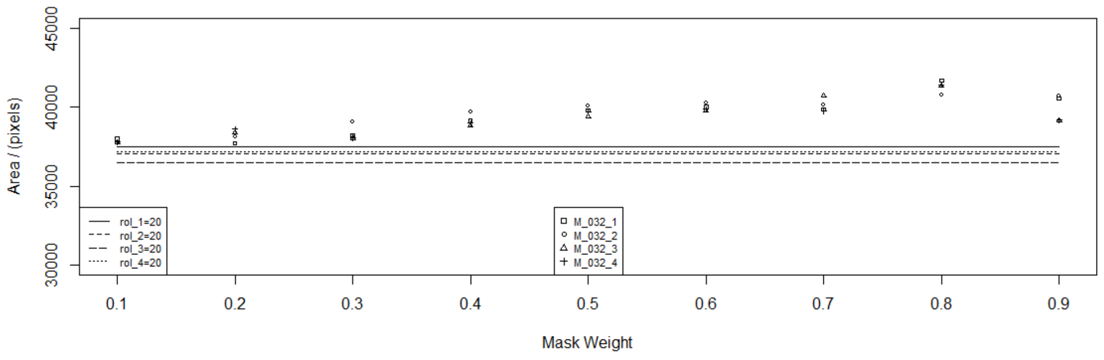


Figure F.5. *M_032* area results. *Unsharp Mask* radius equal to 5 pixels.

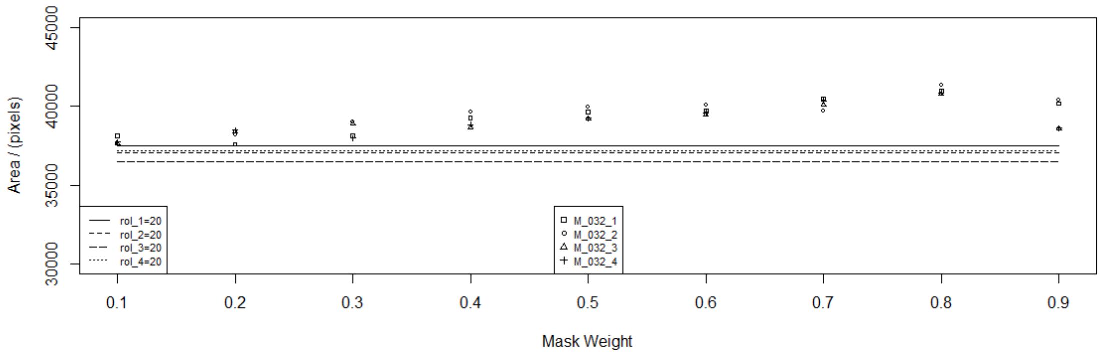


Figure F.6. *M_032* area results. *Unsharp Mask* radius equal to 6 pixels.

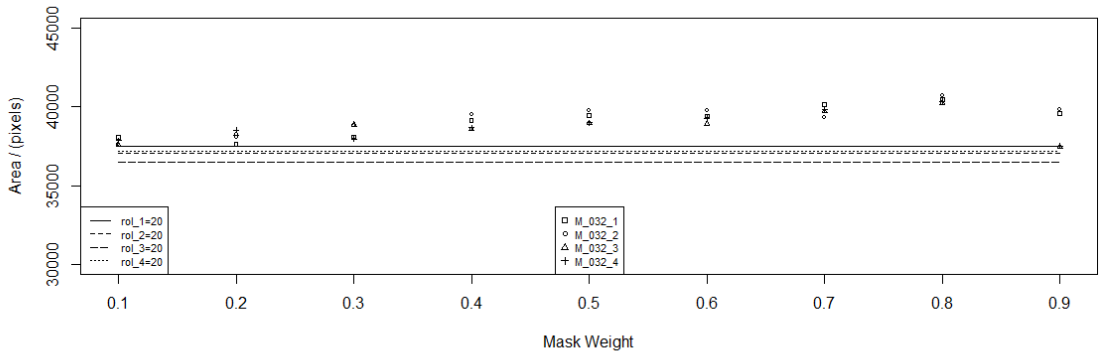


Figure F.7. *M_032* area results. *Unsharp Mask* radius equal to 7 pixels.

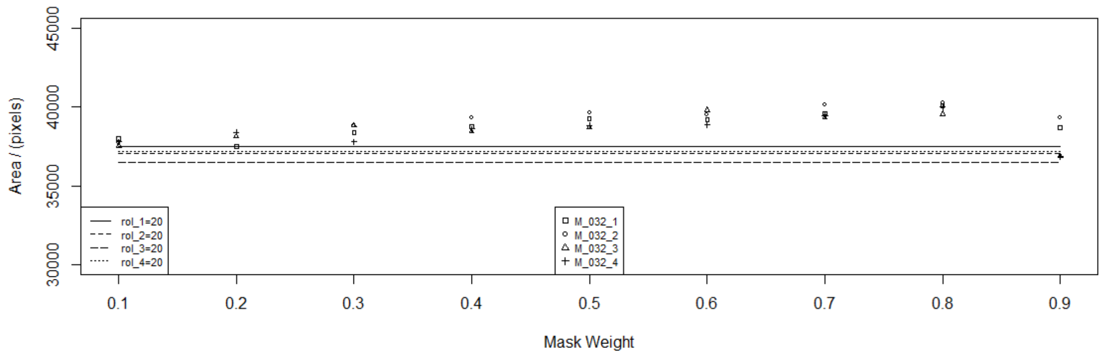


Figure F.8. *M_032* area results. *Unsharp Mask* radius equal to 8 pixels.

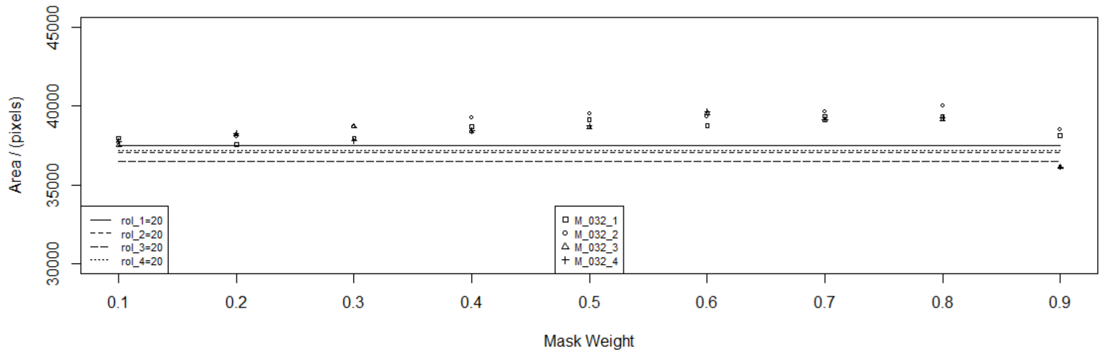


Figure F.9. *M_031* area results. *Unsharp Mask* radius equal to 9 pixels.

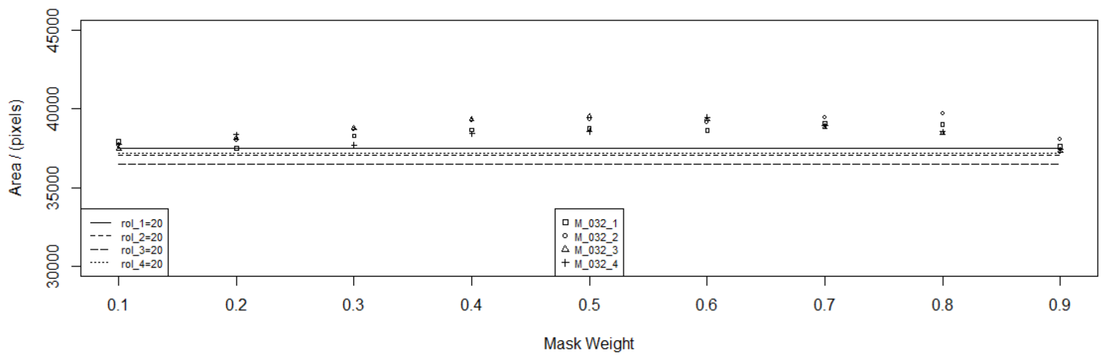


Figure F.10. *M_032* area results. *Unsharp Mask* radius equal to 10 pixels.

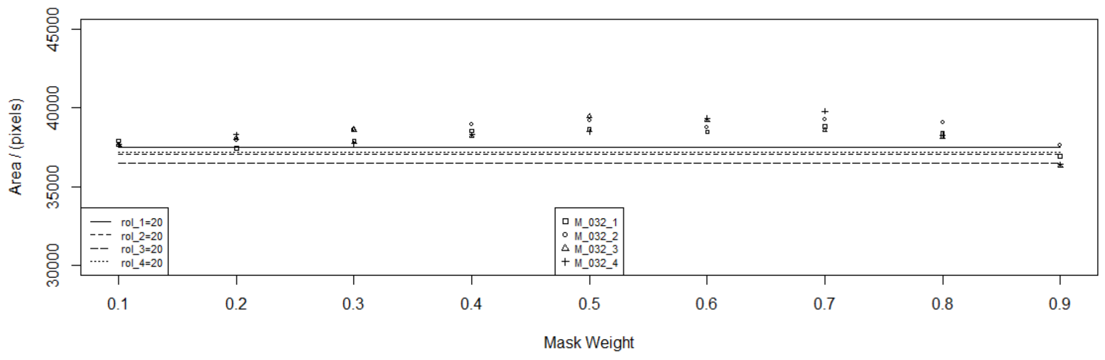


Figure F.11. *M_032* area results. *Unsharp Mask* radius equal to 11 pixels.

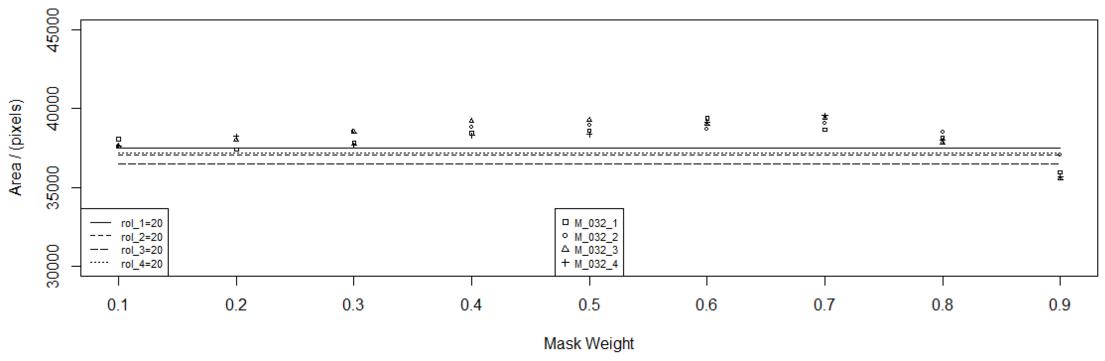


Figure F.12. *M_032* area results. *Unsharp Mask* radius equal to 12 pixels.

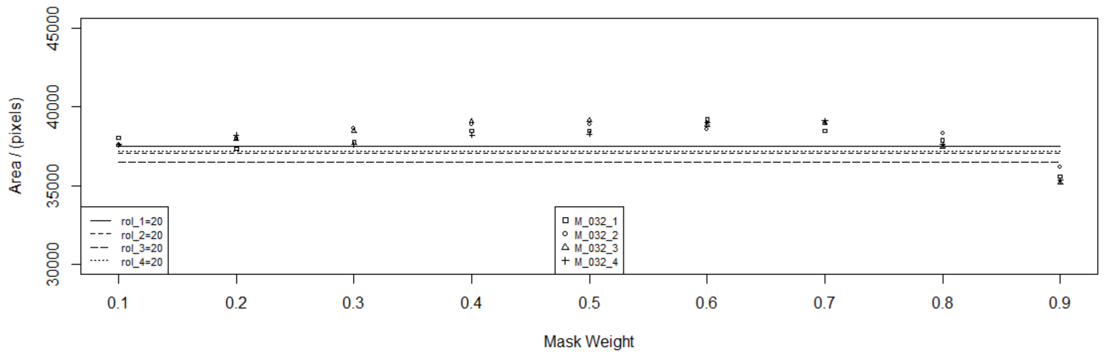


Figure F.13. *M_032* area results. *Unsharp Mask* radius equal to 11 pixels.

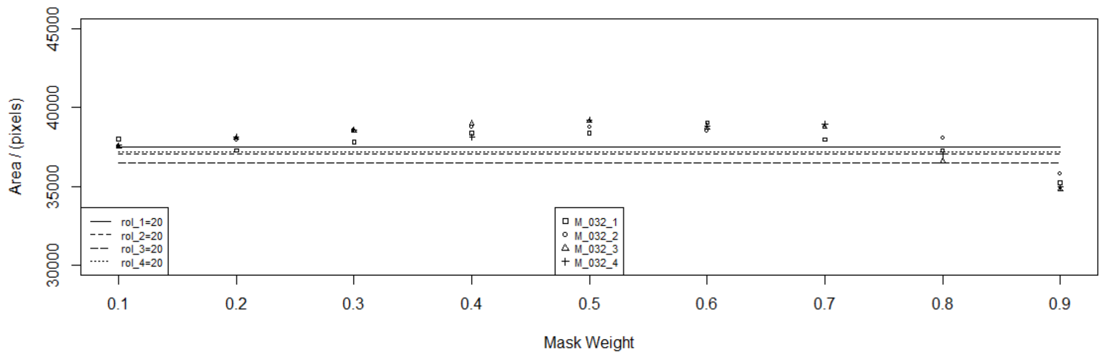


Figure F.14. *M_032* area results. *Unsharp Mask* radius equal to 14 pixels.

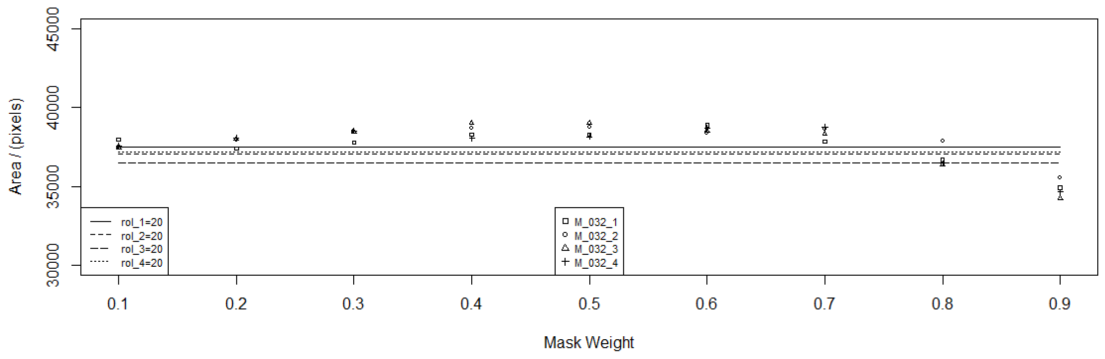


Figure F.15. *M_032* area results. *Unsharp Mask* radius equal to 15 pixels.

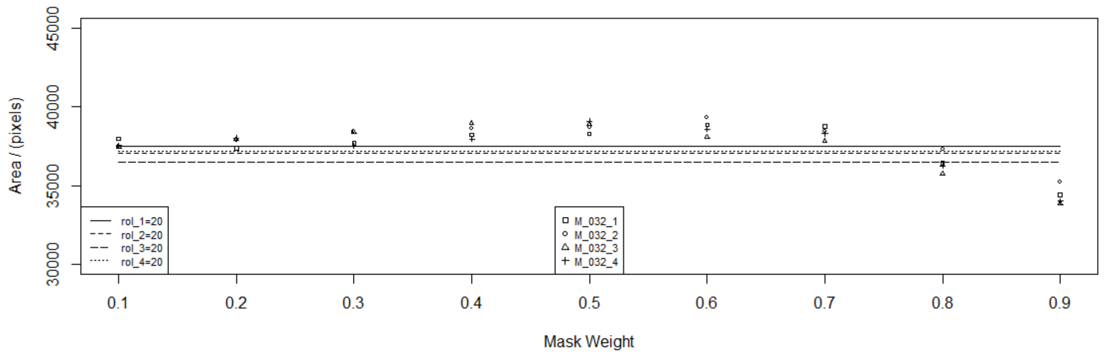


Figure F.16. *M_032* area results. *Unsharp Mask* radius equal to 16 pixels.

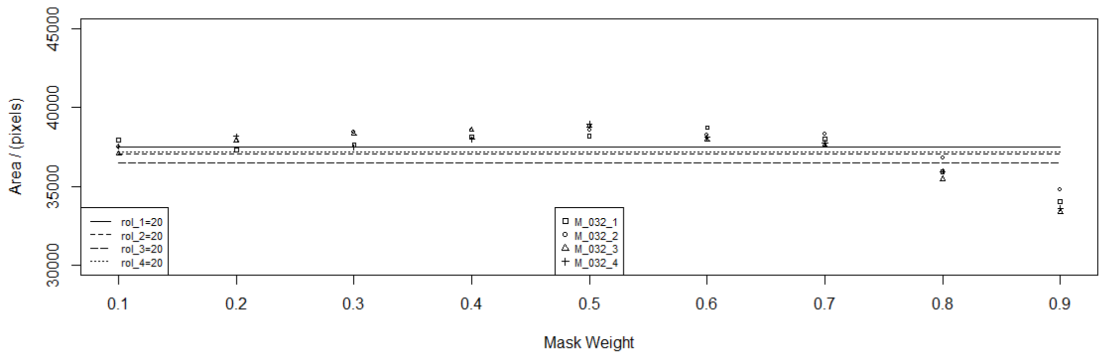


Figure F.17. *M_032* area results. *Unsharp Mask* radius equal to 17 pixels.

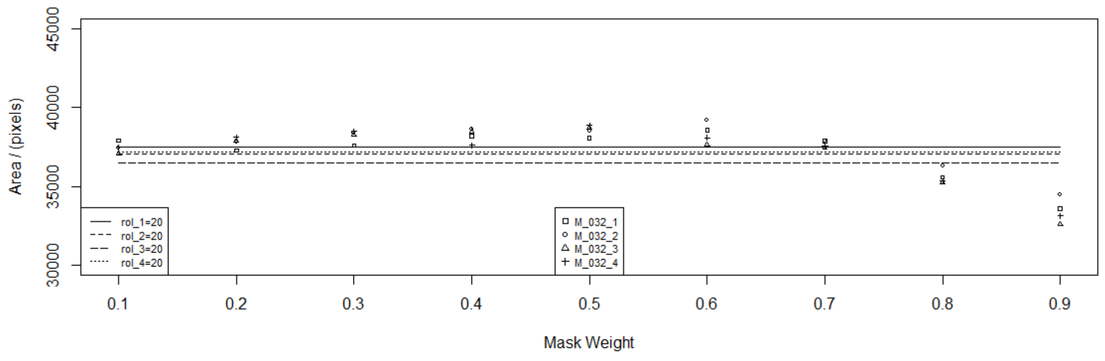


Figure F.18. *M_032* area results. *Unsharp Mask* radius equal to 18 pixels.

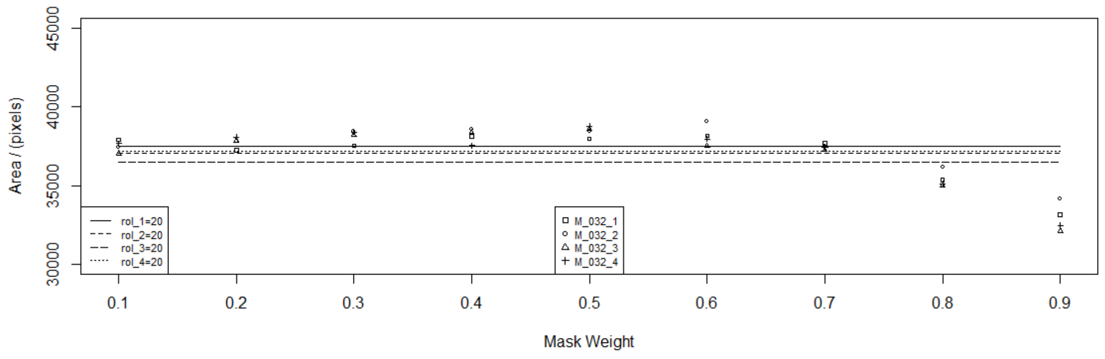


Figure F.19. *M_032* area results. *Unsharp Mask* radius equal to 19 pixels.

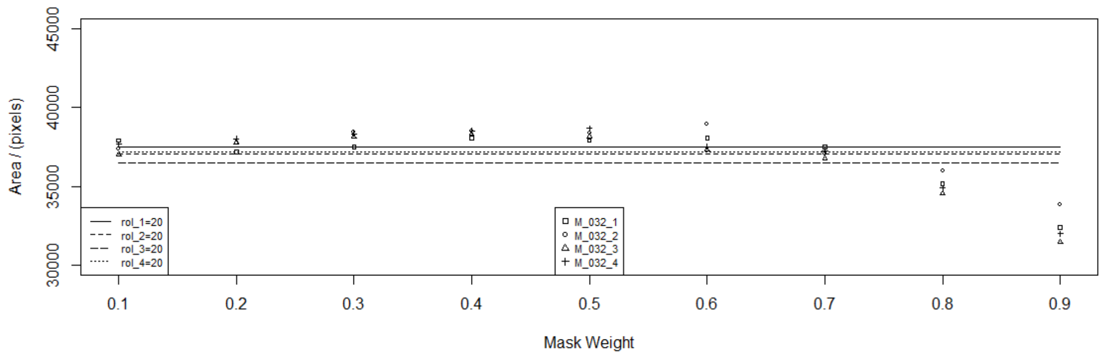


Figure F.20. *M_032* area results. *Unsharp Mask* radius equal to 20 pixels.

Appendix G – *Group 2* Results 6

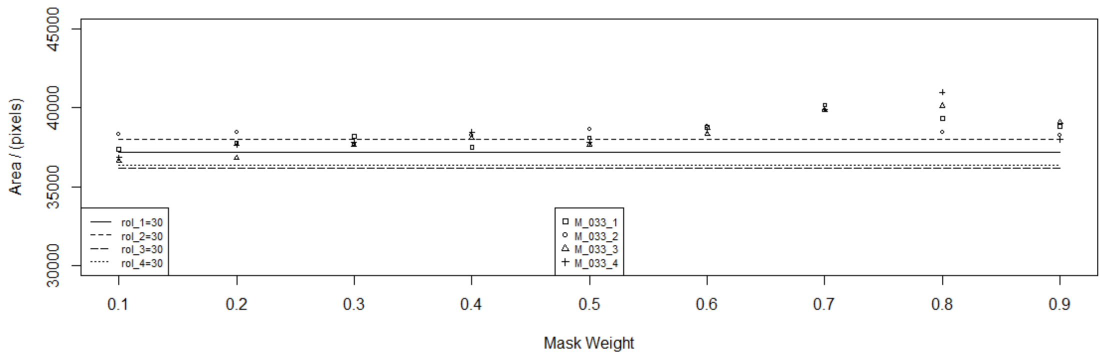


Figure G.1. *M_033* area results. *Unsharp Mask* radius equal to 1 pixels.

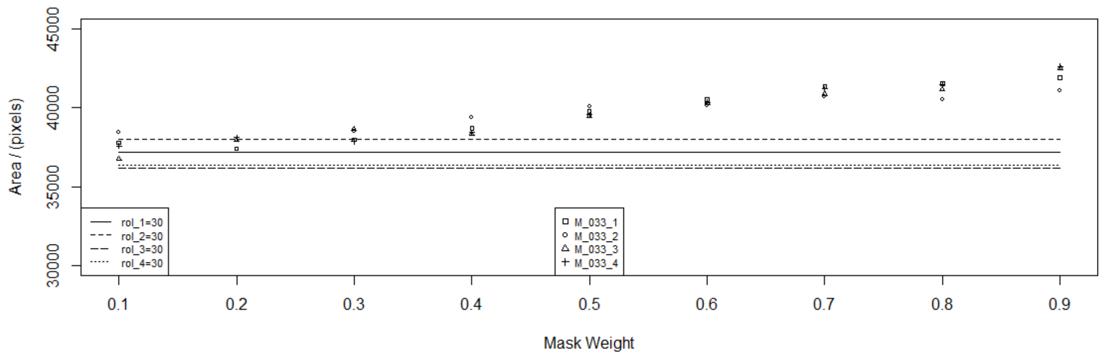


Figure G.2. *M_033* area results. *Unsharp Mask* radius equal to 2 pixels.

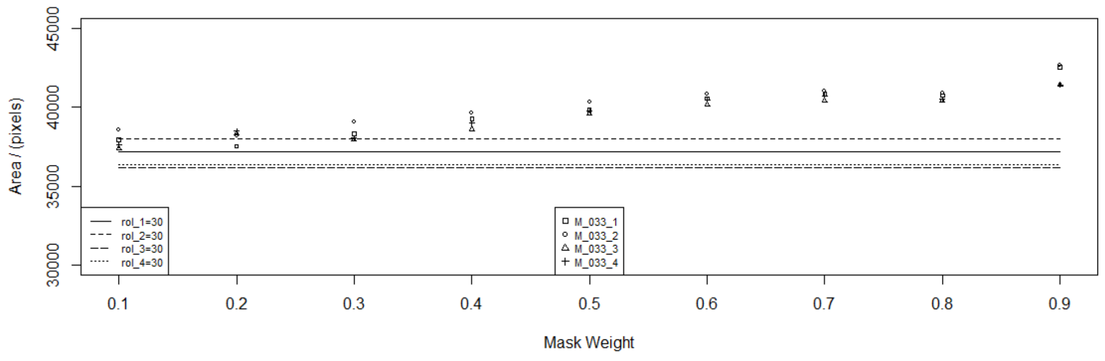


Figure G.3. *M_033* area results. *Unsharp Mask* radius equal to 3 pixels.

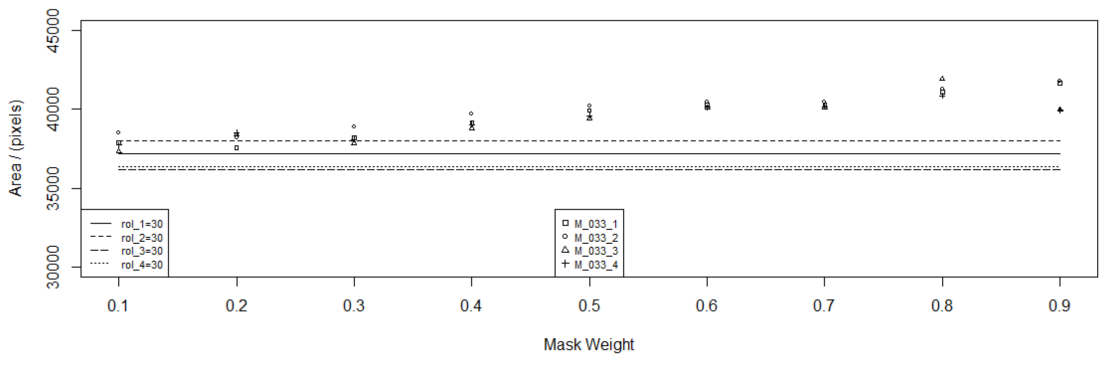


Figure G.4. *M_033* area results. *Unsharp Mask* radius equal to 4 pixels.

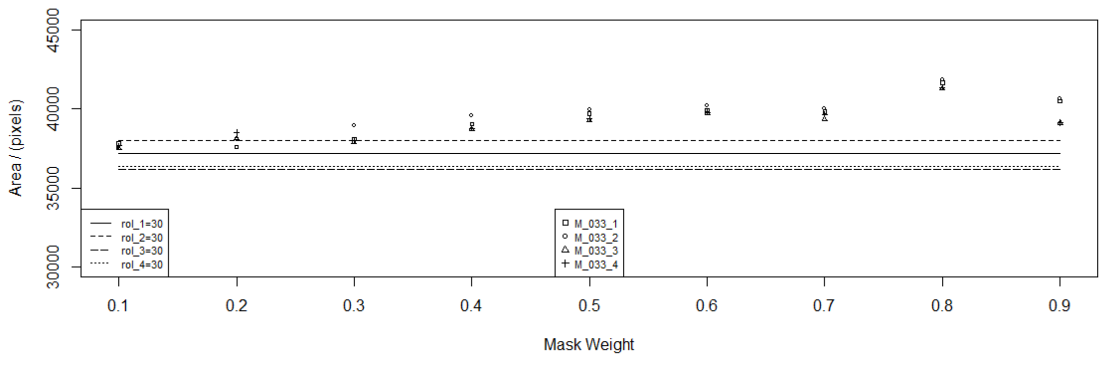


Figure G.5. *M_033* area results. *Unsharp Mask* radius equal to 5 pixels.

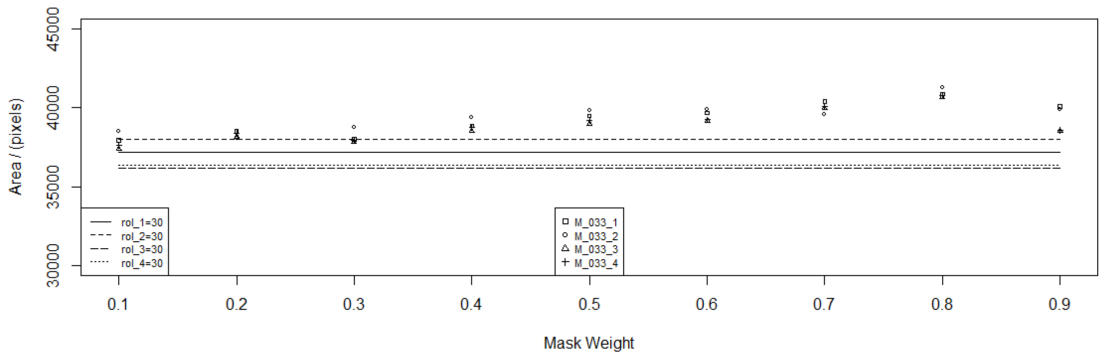


Figure G.6. *M_033* area results. *Unsharp Mask* radius equal to 6 pixels.

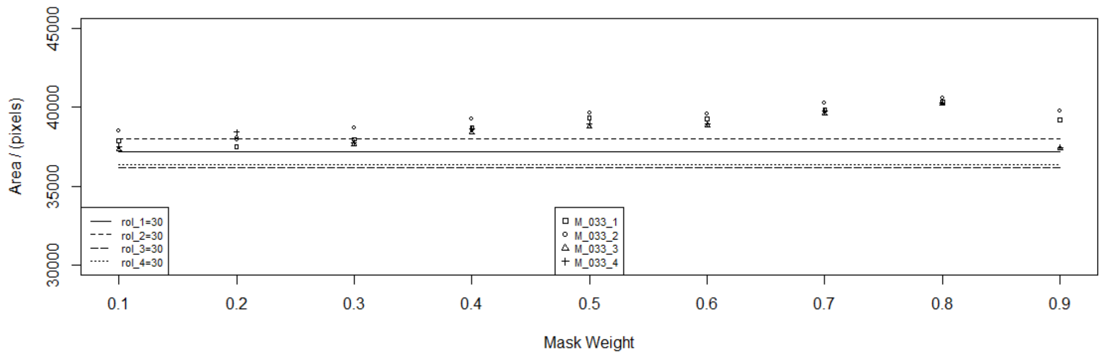


Figure G.7. *M_033* area results. *Unsharp Mask* radius equal to 7 pixels.

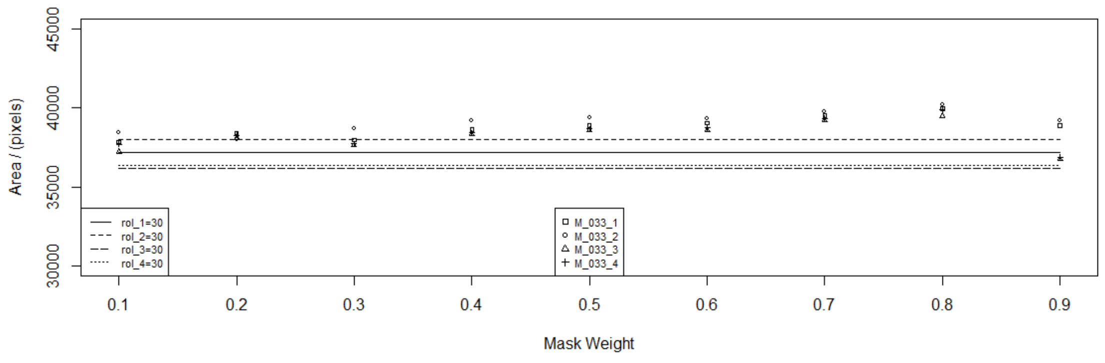


Figure G.8. *M_033* area results. *Unsharp Mask* radius equal to 8 pixels.

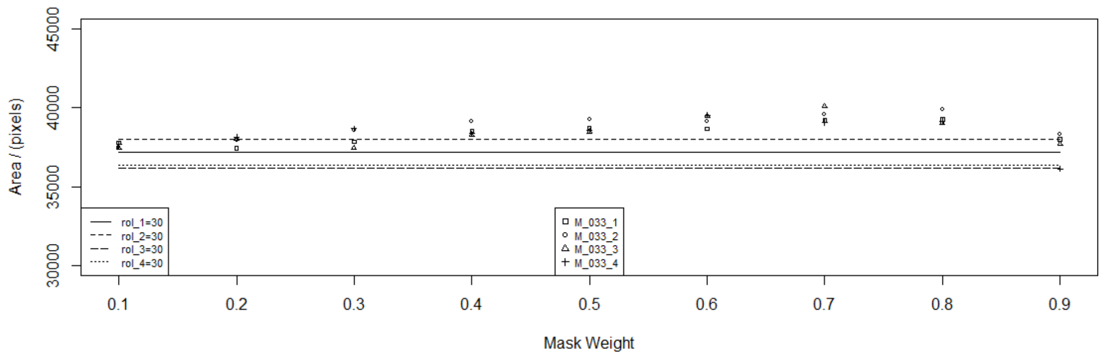


Figure G.9. *M_033* area results. *Unsharp Mask* radius equal to 9 pixels.

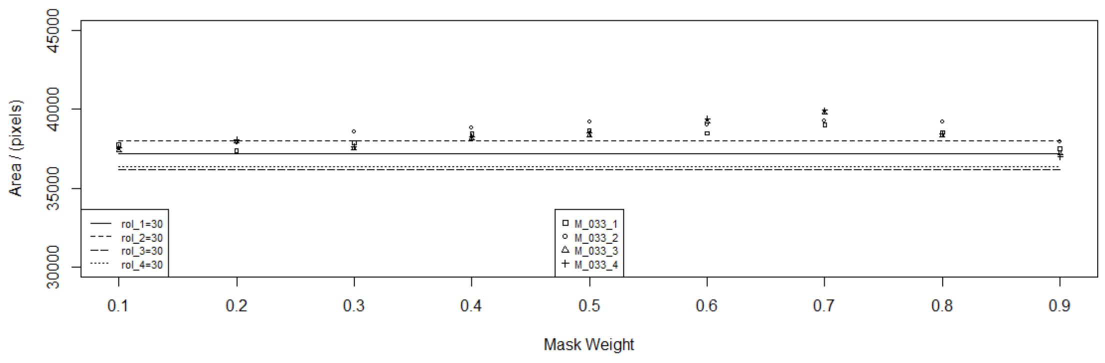


Figure G.10. *M_033* area results. *Unsharp Mask* radius equal to 10 pixels.

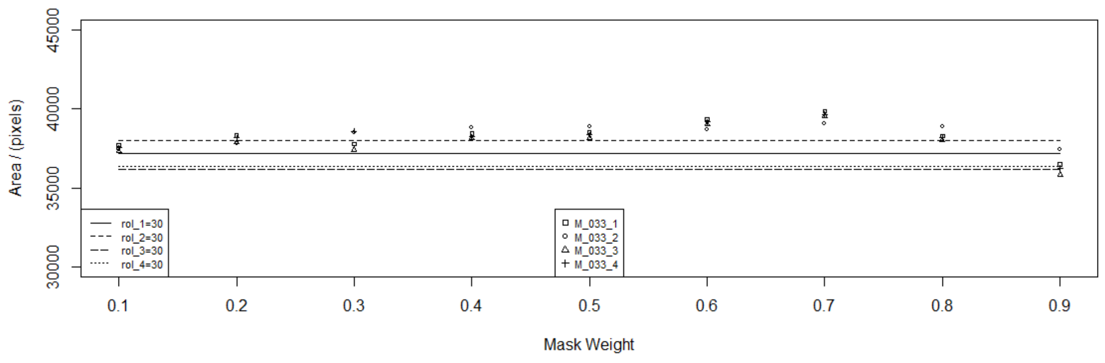


Figure G.11. *M_033* area results. *Unsharp Mask* radius equal to 11 pixels.

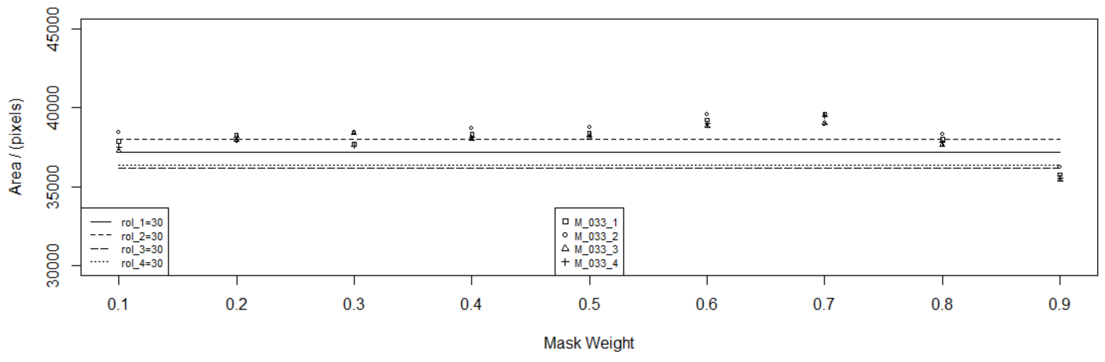


Figure G.12. *M_033* area results. *Unsharp Mask* radius equal to 10 pixels.

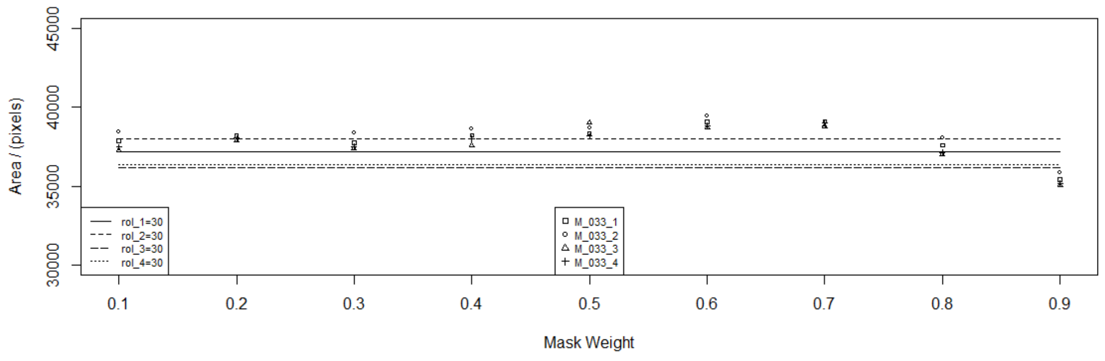


Figure G.13. *M_033* area results. *Unsharp Mask* radius equal to 13 pixels.

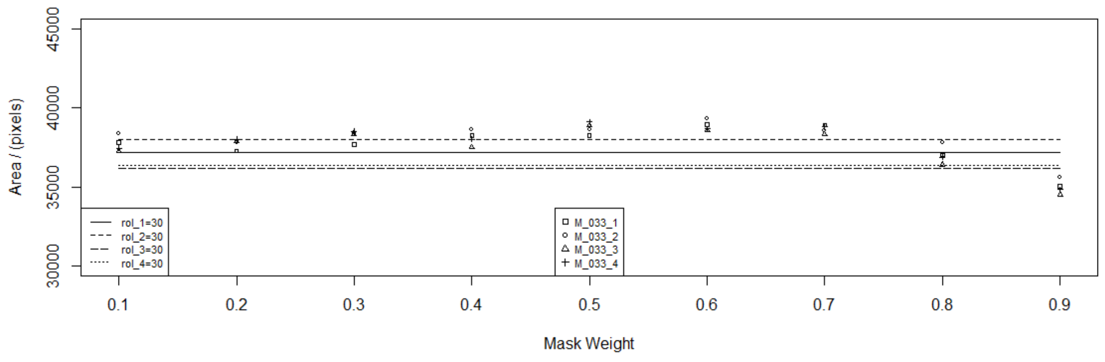


Figure G.14. *M_033* area results. *Unsharp Mask* radius equal to 14 pixels.

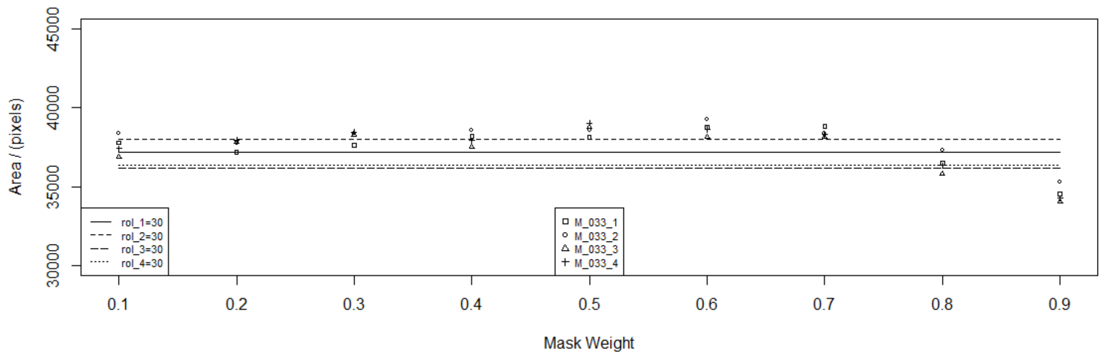


Figure G.15. *M_033* area results. *Unsharp Mask* radius equal to 15 pixels.

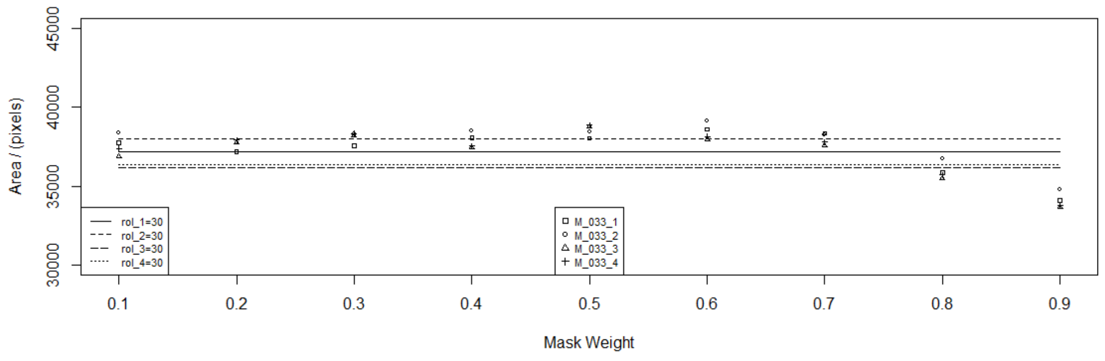


Figure G.16. *M_033* area results. *Unsharp Mask* radius equal to 16 pixels.

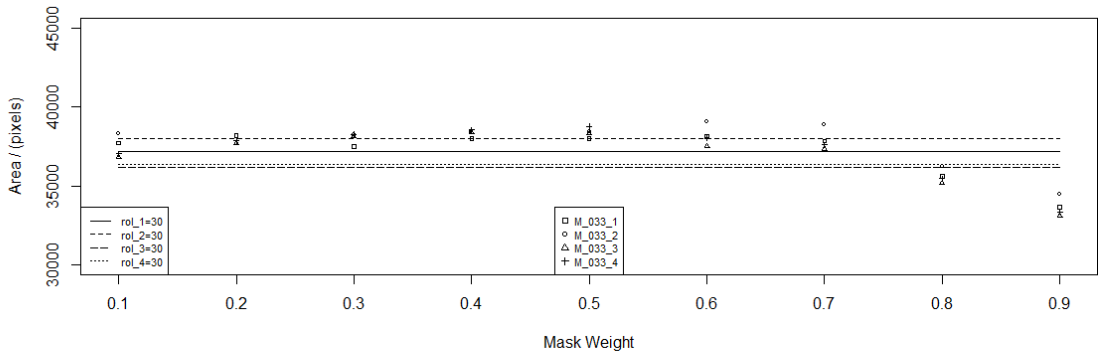


Figure G.17. *M_033* area results. *Unsharp Mask* radius equal to 17 pixels.

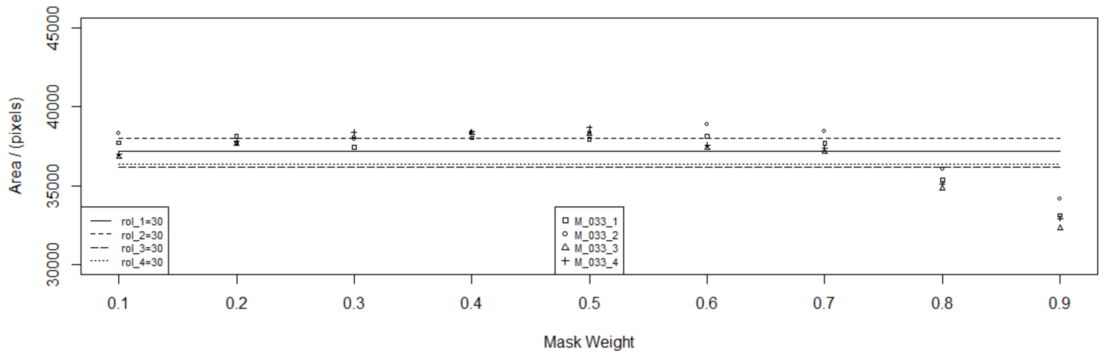


Figure G.18. *M_033* area results. *Unsharp Mask* radius equal to 18 pixels.

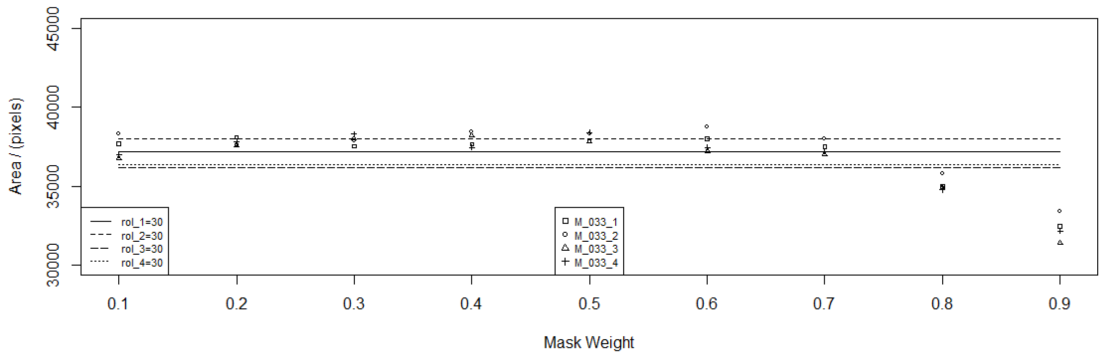


Figure G.19. *M_033* area results. *Unsharp Mask* radius equal to 19 pixels.

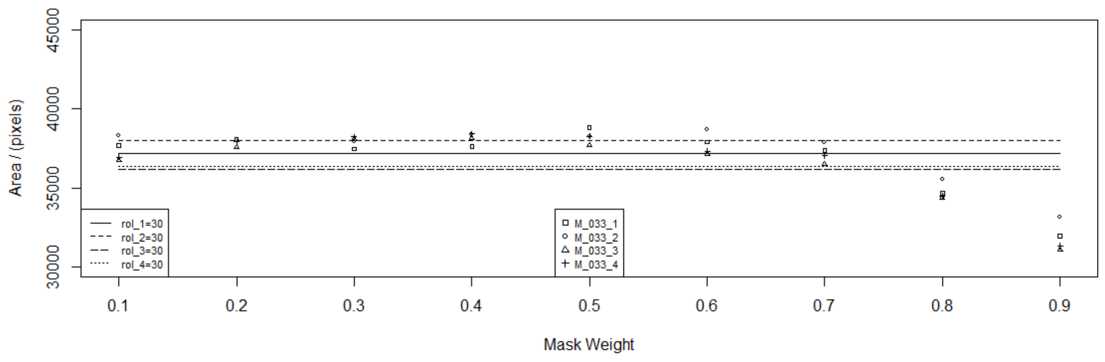


Figure G.20. *M_033* area results. *Unsharp Mask* radius equal to 20 pixels.

Appendix H – *Group 2* table results

Table H.1. *M_028_1* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1	1	35166.0	290.0	35166	0.1
2	-	0	NaN	NaN	34202	-
3	-	0	NaN	NaN	34150	-
4	-	0	NaN	NaN	34938	-
5	0.1	1	34938.0	62.0	34938	0.1
6	0.1	1	34893.0	17.0	34893	0.1
7	0.1	1	34881.0	5.0	34881	0.1
8	0.1	1	34877.0	1.0	34877	0.1
9	0.1	1	34877.0	1.0	34877	0.1
10	-	0	NaN	NaN	34876	-
11	-	0	NaN	NaN	34876	-
12	-	0	NaN	NaN	34876	-
13	-	0	NaN	NaN	34876	-
14	-	0	NaN	NaN	34876	-
15	-	0	NaN	NaN	34876	-
16	-	0	NaN	NaN	34876	-
17	-	0	NaN	NaN	34876	-
18	-	0	NaN	NaN	34876	-
19	-	0	NaN	NaN	34876	-
20	-	0	NaN	NaN	34876	-

Table H.2. *M_028_2* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1 to 0.5	5	38286.2	1423.2	39399	0.5
2	0.1 to 0.4 and 0.9	5	39001.8	2138.8	40321	0.4
3	0.1 to 0.4 and 0.9	5	39407.6	2544.6	41117	0.9
4	0.1 to 0.4 and 0.9	5	39679.2	2816.2	42759	0.9
5	0.1 to 0.3 and 0.9	4	39423.8	2560.8	42294	0.9
6	0.1 to 0.3 and 0.9	4	39251.8	2388.8	41900	0.9
7	0.1, 0.2 and 0.9	3	39018.0	2155.0	41589	0.9
8	0.1, 0.2, 0.8 and 0.9	4	38510.8	1647.8	41048	0.9
9	0.1, 0.2, 0.8 and 0.9	4	38293.5	1430.5	40676	0.9
10	0.1, 0.2, 0.8 and 0.9	4	38018.5	1155.5	40046	0.9
11	0.1, 0.2, 0.8 and 0.9	4	37963.5	1100.5	39949	0.9
12	0.1, 0.2, 0.8 and 0.9	4	37912.3	1049.3	39852	0.9
13	0.1, 0.2, 0.8 and 0.9	4	38141.0	1278.0	39744	0.9
14	0.1, 0.2, 0.8 and 0.9	4	38315.0	1452.0	39686	0.9
15	0.1, 0.2, 0.8 and 0.9	4	38561.8	1698.8	39873	0.8
16	0.1, 0.2, 0.8 and 0.9	4	38521.3	1658.3	39814	0.8
17	0.1, 0.2, 0.8 and 0.9	4	38479.0	1616.0	39761	0.8
18	0.1, 0.2, 0.8 and 0.9	4	38439.3	1576.3	39710	0.8
19	0.1, 0.2, 0.8 and 0.9	4	38693.8	1830.8	40487	0.9
20	0.1, 0.2, 0.8 and 0.9	4	38662.8	1799.8	40421	0.9

Table H.3. *M_028_3* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1	1	36568.0	61.0	36568	0.1
2	0.1	1	36554.0	47.0	36554	0.1
3	0.1 and 0.2	2	36553.0	46.0	36578	0.2
4	0.1 and 0.2	2	36535.5	28.5	36557	0.2
5	0.1 and 0.2	2	36524.0	17.0	36538	0.2
6	0.2	1	36522.0	15.0	36522	0.2
7	0.2	1	36516.0	9.0	36516	0.2
8	0.2	1	36511.0	4.0	36511	0.2
9	0.2 and 0.3	2	36730.0	223.0	36951	0.3
10	0.3	1	36949.0	442.0	36949	0.3
11	0.3 and 0.4	2	36942.0	435.0	36948	0.3
12	0.3 and 0.4	2	36939.0	432.0	36946	0.3
13	0.3 and 0.4	2	36934.0	427.0	36944	0.3
14	0.3 and 0.4	2	36927.5	420.5	36940	0.3
15	0.3 and 0.4	2	36749.0	242.0	36939	0.3
16	0.3 and 0.4	2	36740.5	233.5	36933	0.3
17	0.3 and 0.4	2	36736.0	229.0	36927	0.3
18	0.3 to 0.5	2	36801.7	294.7	36946	0.5
19	0.3 to 0.5	2	36681.3	174.3	36942	0.5
20	0.3 to 0.5	2	36677.0	170.0	36936	0.5

Table H.4. *M_028_4* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1	1	37963.0	249.0	37963	0.1
2	0.1 and 0.9	2	37976.5	262.5	38149	0.1
3	0.1 and 0.9	2	40440.0	2726.0	42790	0.9
4	0.1 and 0.7 to 0.9	4	38961.0	1247.0	41781	0.9
5	0.1, 0.8 and 0.9	3	39618.0	1904.0	41134	0.9
6	0.1, 0.8 and 0.9	3	39541.0	1827.0	40304	0.8
7	0.1, 0.8 and 0.9	3	39268.7	1554.6	40000	0.8
8	0.7 to 0.9	3	38971.0	1257.0	39723	0.8
9	0.8 and 0.9	2	39110.0	1396.0	39490	0.8
10	0.7 to 0.9	3	39117.7	1403.7	39726	0.9
11	0.7 to 0.9	3	38943.3	1229.3	39509	0.9
12	0.7 to 0.9	3	38669.7	955.7	39024	0.8
13	0.7 to 0.9	3	38535.3	821.3	38875	0.8
14	0.7 to 0.9	3	38410.7	696.7	38763	0.8
15	0.7 to 0.9	3	38223.0	509.0	38437	0.8
16	0.1 and 0.7 to 0.9	4	38072.5	358.5	38299	0.8
17	0.1, 0.8 and 0.9	3	37991.0	277.0	38091	0.9
18	0.1 and 0.7 to 0.9	4	38202.0	488.0	38681	0.7
19	0.1 and 0.7	2	38299.5	585.5	38624	0.7
20	0.1 and 0.7	2	38238.5	524.5	38509	0.7

Table H.5. *M_029_1* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1	1	39937.0	266.0	39937	0.1
2	-	0	NaN	NaN	26229	-
3	-	0	NaN	NaN	34118	-
4	-	0	NaN	NaN	37176	-
5	-	0	NaN	NaN	38379	-
6	-	0	NaN	NaN	38154	-
7	-	0	NaN	NaN	38008	-
8	-	0	NaN	NaN	37869	-
9	-	0	NaN	NaN	38485	-
10	-	0	NaN	NaN	39024	-
11	-	0	NaN	NaN	38765	-
12	-	0	NaN	NaN	38711	-
13	-	0	NaN	NaN	38985	-
14	-	0	NaN	NaN	38923	-
15	-	0	NaN	NaN	38896	-
16	-	0	NaN	NaN	38523	-
17	-	0	NaN	NaN	38479	-
18	-	0	NaN	NaN	36182	-
19	-	0	NaN	NaN	36154	-
20	-	0	NaN	NaN	36119	-

Table H.6. *M_029_2* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1 to 0.8	8	34693.1	5457.1	37510	0.3
2	0.1 to 0.9	9	36316.4	7080.4	39223	0.9
3	0.1 to 0.9	9	36752.4	7516.4	40285	0.9
4	0.1 to 0.9	9	37778.9	8542.9	41932	0.9
5	0.1 to 0.9	9	37806.6	8570.6	42433	0.9
6	0.1 to 0.9	9	37885.9	8649.9	41998	0.9
7	0.1 to 0.9	9	37967.8	8731.8	41598	0.9
8	0.1 to 0.9	9	37977.6	8741.6	41271	0.9
9	0.1 to 0.9	9	38044.3	8808.3	41010	0.9
10	0.1 to 0.9	9	37950.6	8714.6	40645	0.9
11	0.1 to 0.9	9	37893.2	8657.2	40478	0.9
12	0.1 to 0.9	9	38058.9	8822.9	40044	0.9
13	0.1 to 0.9	9	38121.0	8885.0	40443	0.8
14	0.1 to 0.9	9	38076.7	8840.7	40343	0.8
15	0.1 to 0.9	9	38025.2	8789.2	40236	0.8
16	0.1 to 0.9	9	38085.0	8849.0	40161	0.8
17	0.1 to 0.9	9	38173.0	8937.0	41215	0.8
18	0.1 to 0.9	9	38111.1	8875.1	40954	0.8
19	0.1 to 0.9	9	38069.3	8833.3	40883	0.8
20	0.1 to 0.9	9	38034.1	8798.1	40811	0.8

Table H.7. *M_029_3* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1 to 0.6	6	36413.7	5782.7	38725	0.6
2	0.1 to 0.4 and 0.9	6	37185.2	6554.2	41076	0.9
3	0.1 to 0.4 and 0.9	6	37936.4	7305.4	41603	0.9
4	0.1 to 0.3 and 0.9	7	37398.5	6767.5	40956	0.9
5	0.1 to 0.3 and 0.9	7	37627.0	6996.0	40713	0.8
6	0.1, 0.3 and 0.9	3	37204.0	6573.0	40601	0.9
7	0.1, 0.3 and 0.9	3	37239.0	6608.0	40501	0.7
8	0.1, 0.3 and 0.9	3	37608.7	6977.7	40348	0.8
9	0.1, 0.3 and 0.9	3	37430.0	6799.0	40312	0.8
10	0.1, 0.3 and 0.9	3	37883.0	7252.0	40261	0.7
11	0.1, 0.3 and 0.9	3	37840.0	7209.0	40220	0.8
12	0.1, 0.3 and 0.9	3	38208.3	7577.3	40183	0.8
13	0.1, 0.3 and 0.9	3	38161.7	7530.7	40122	0.8
14	0.1, 0.3 and 0.9	3	37876.3	7245.3	39690	0.8
15	0.1, 0.3 and 0.9	3	37680.0	7049.0	39501	0.8
16	0.1, 0.3 and 0.9	3	37862.7	7231.7	40191	0.8
17	0.1, 0.3 and 0.9	3	37691.7	7060.7	40144	0.8
18	0.1, 0.3 and 0.9	3	37572.3	6941.3	40046	0.5
19	0.1, 0.3 and 0.9	3	37467.7	6836.7	39952	0.7
20	0.1, 0.3 and 0.9	3	37267.0	6636.0	39908	0.7

Table H.8. *M_029_4* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1,0.2 and 0.4 to 0.6	5	37888.8	664.8	38292	0.6
2	0.1 to 0.9	9	38513.3	1289.3	42771	0.9
3	0.1 to 0.9	9	38787.6	1563.6	42106	0.9
4	0.1 to 0.9	9	39161.4	1937.4	40918	0.9
5	0.1 to 0.9	9	38899.6	1675.6	40207	0.8
6	0.1 to 0.9	9	38797.8	1573.8	39807	0.9
7	0.1 to 0.9	9	38634.2	1410.2	40079	0.7
8	0.1 to 0.9	9	38617.4	1393.4	40074	0.8
9	0.1 to 0.9	9	38554.9	1330.9	39659	0.8
10	0.1 to 0.9	9	38468.9	1244.9	39475	0.7
11	0.1 to 0.9	9	38426.6	1202.6	40346	0.8
12	0.1 to 0.8	8	38450.5	1226.5	40150	0.8
13	0.1 to 0.9	9	38441.3	1217.3	39975	0.8
14	0.1 to 0.9	9	38281.3	1057.3	39214	0.8
15	0.1 to 0.8	8	38312.9	1088.9	39069	0.8
16	0.1 to 0.8	8	38215.9	991.9	38925	0.8
17	0.1 to 0.8	8	38084.6	860.6	38738	0.8
18	0.1 to 0.8	8	38087.3	863.3	38871	0.5
19	0.1 to 0.8	8	38160.9	936.9	39171	0.7
20	0.1 to 0.8	8	38321.9	1097.9	39056	0.7

Table H.9. *M_030_1* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1 and 0.2	2	36757.0	131.0	36844	0.2
2	0.1	1	36792.0	166.0	36792	0.1
3	0.1 and 0.9	2	37900.5	1274.5	38934	0.9
4	0.1 and 0.9	2	39444.5	2818.5	42086	0.9
5	0.1, 0.8 and 0.9	3	38693.7	2067.7	42413	0.9
6	0.8 and 0.9	2	39689.5	3063.5	42026	0.9
7	0.8 and 0.9	2	39974.0	3348.0	41694	0.9
8	0.8 and 0.9	2	40319.0	3693.0	41469	0.9
9	0.1, 0.8 and 0.9	3	39020.0	2394.0	41272	0.9
10	0.1, 0.8 and 0.9	3	38814.0	2188.0	40772	0.9
11	0.1, 0.8 and 0.9	3	38408.7	1782.7	39657	0.9
12	0.1, 0.8 and 0.9	3	38340.7	1714.7	39537	0.9
13	0.1, 0.8 and 0.9	3	38670.0	2044.0	39823	0.8
14	0.1, 0.8 and 0.9	3	38617.7	1991.7	39762	0.8
15	0.1, 0.8 and 0.9	3	38312.9	1088.9	39069	0.8
16	0.1, 0.8 and 0.9	3	38215.9	991.9	38925	0.8
17	0.1, 0.8 and 0.9	3	38084.6	860.6	38738	0.8
18	0.1, 0.8 and 0.9	3	38087.3	863.3	38871	0.5
19	0.1, 0.8 and 0.9	3	38160.9	936.9	39171	0.7
20	0.1, 0.8 and 0.9	3	38321.9	1097.9	39056	0.7

Table H.10. *M_030_2* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1	1	38967.0	219.0	38967	0.1
2	0.1, 0.4 and 0.9	3	39235.3	487.3	39714	0.9
3	0.1, 0.8 and 0.9	3	40306.7	1558.7	41207	0.9
4	0.1, 0.8 and 0.9	3	40982.3	2234.3	42668	0.9
5	0.1, 0.2 and 0.7 to 0.9	5	40338.0	1590.0	42081	0.9
6	0.1 and 0.7 to 0.9	4	40219.3	1471.3	41555	0.9
7	0.1, 0.2 and 0.7 to 0.9	5	40016.0	1268.0	41102	0.9
8	0.1, 0.2 and 0.7 to 0.9	5	39932.2	1184.2	40633	0.8
9	0.1 and 0.7 to 0.9	4	39912.8	1164.8	40418	0.8
10	0.1, 0.2 and 0.7 to 0.9	5	39803.6	1055.6	41382	0.8
11	0.1, 0.2 and 0.7 to 0.9	5	39589.0	841.0	41214	0.8
12	0.1, 0.2 and 0.6 to 0.9	6	39500.2	752.2	41087	0.8
13	0.1, 0.2 and 0.6 to 0.9	6	39450.2	702.2	40781	0.8
14	0.1, 0.7 and 0.8	3	39788.7	1040.7	40682	0.8
15	0.1, 0.7 and 0.8	3	39586.3	838.3	40548	0.8
16	0.1, 0.2, 0.7 and 0.8	4	39527.0	779.0	40423	0.8
17	0.1, 0.2, 0.7 and 0.8	4	39471.8	723.8	40315	0.8
18	0.1, 0.2, 0.7 and 0.8	4	39423.3	675.3	40217	0.8
19	0.1, 0.2, 0.7 and 0.8	4	39383.5	635.5	40152	0.8
20	0.1, 0.2 and 0.6 to 0.8	5	39320.6	1097.9	40076	0.8

Table H.11. *M_030_3* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1 to 0.9	9	30025.0	13044.0	36274	0.4
2	0.1 to 0.9	9	32148.7	15167.7	38250	0.9
3	0.1 to 0.9	9	33770.2	16789.2	41376	0.9
4	0.1 to 0.9	9	34010.2	17029.2	40739	0.9
5	0.1 to 0.9	9	34468.7	17487.7	41036	0.9
6	0.1 to 0.9	9	34723.9	17742.9	40602	0.9
7	0.1 to 0.9	9	34980.6	17999.6	40241	0.9
8	0.1 to 0.9	9	34983.1	18002.1	39969	0.9
9	0.1 to 0.9	9	35056.0	18075.0	39765	0.8
10	0.1 to 0.9	9	36513.0	19532.0	39654	0.8
11	0.1 to 0.9	9	36519.3	19538.3	39900	0.9
12	0.1 to 0.9	9	36560.6	19579.6	39734	0.9
13	0.1 to 0.9	9	36514.0	19533.0	39583	0.9
14	0.1 to 0.9	9	36512.7	19531.7	39379	0.9
15	0.1 to 0.9	9	36466.2	19485.2	39247	0.9
16	0.1 to 0.9	9	36413.4	19432.4	39085	0.8
17	0.1 to 0.9	9	36367.2	19386.2	39006	0.8
18	0.1 to 0.9	9	36455.6	19474.6	38935	0.8
19	0.1 to 0.9	9	36429.3	19448.3	38872	0.8
20	0.1 to 0.9	9	36549.1	19568.1	38541	0.9

Table H.12. *M_030_4* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1 to 0.7	7	37801.4	1097.4	39348	0.6
2	0.1 to 0.9	9	38840.9	2136.9	42931	0.9
3	0.1 to 0.9	9	39264.7	2560.7	42355	0.9
4	0.1 to 0.9	9	39081.4	2377.4	41130	0.9
5	0.1 to 0.9	9	38828.6	2124.6	40466	0.7
6	0.1 to 0.9	9	38945.8	2241.8	40541	0.8
7	0.1 to 0.9	9	38853.3	2149.3	39971	0.8
8	0.1 to 0.9	9	38669.2	1965.2	39631	0.8
9	0.1 to 0.9	9	38439.1	1735.1	39375	0.8
10	0.1 to 0.9	9	38408.0	1704.0	40227	0.8
11	0.1 to 0.8	8	38461.6	1757.6	39969	0.8
12	0.1 to 0.8	8	38439.0	1735.0	39605	0.7
13	0.1 to 0.8	8	38304.4	1600.4	39484	0.7
14	0.1 to 0.8	8	38246.9	1542.9	39331	0.7
15	0.1 to 0.8	8	38266.8	1562.8	39195	0.7
16	0.1 to 0.8	8	38189.9	1485.9	39077	0.7
17	0.1 to 0.8	8	38075.5	1371.5	38752	0.6
18	0.1 to 0.8	8	37970.6	1266.6	38648	0.6
19	0.1 to 0.8	8	37861.1	1157.1	38555	0.6
20	0.1 to 0.8	8	37806.6	1102.6	38488	0.6

Table H.13. *M_031_1* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1, 0.4 and 0.5	3	38862.3	158.3	38989	0.1
2	0.1 to 0.5 and 0.9	6	39539.8	835.8	41230	0.9
3	0.2 to 0.4 and 0.7 to 0.9	6	40373.7	1669.7	43018	0.9
4	0.2, 0.3, 0.5, 0.7 to 0.9	6	40337.7	1633.7	41639	0.8
5	0.2, 0.3, 0.5 to 0.9	7	40083.3	1379.3	42208	0.8
6	0.1 to 0.9	9	39510.9	806.9	41740	0.8
7	0.2, 0.4, 0.6 to 0.9	6	39535.5	831.5	41282	0.8
8	0.2, 0.6 to 0.9	5	39610.0	906.0	40898	0.8
9	0.2, 0.4 to 0.8	6	39425.2	721.2	40402	0.8
10	0.2, 0.5 to 0.8	5	39377.6	673.6	40154	0.8
11	0.4 to 0.8	5	39278.4	574.4	39955	0.8
12	0.4, 0.5, 0.7 and 0.8	4	39231.5	527.5	39729	0.8
13	0.1, 0.2, 0.5, 0.7 and 0.8	5	39000.8	296.8	39469	0.7
14	0.1, 0.7 and 0.8	3	39067.0	363.0	39347	0.7
15	0.1 and 0.7	2	39147.5	443.5	39214	0.7
16	0.1 and 0.7	2	39080.0	376.0	39087	0.7
17	0.1 and 0.7	2	39575.0	871.0	40099	0.7
18	0.1, 0.6 and 0.7	3	39327.7	623.7	39967	0.7
19	0.1, 0.3, 0.6 and 0.7	4	39063.8	359.8	39541	0.7
20	0.1, 0.3, 0.6 and 0.7	4	39222.0	518.0	39691	0.6

Table H.14. *M_031_2* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1 to 0.7	7	38438.7	664.7	38984	0.5
2	0.1 to 0.9	9	39055.0	1281.0	40133	0.9
3	0.1 to 0.9	9	40070.8	2296.8	43021	0.9
4	0.1 to 0.9	9	39923.3	2149.3	42170	0.9
5	0.1 to 0.9	9	39892.7	2118.7	41369	0.9
6	0.1 to 0.9	9	39763.8	1989.8	40470	0.8
7	0.1 to 0.9	9	39668.6	1894.6	41211	0.8
8	0.1 to 0.9	9	39502.2	1728.2	40825	0.8
9	0.1 to 0.9	9	39378.2	1604.2	40293	0.8
10	0.1 to 0.9	9	39283.7	1509.7	40048	0.8
11	0.1 to 0.9	9	39107.6	1333.6	39789	0.8
12	0.1 to 0.9	9	38986.2	1212.2	39543	0.4
13	0.1 to 0.8	8	39016.0	1242.0	39522	0.5
14	0.1 to 0.8	8	38927.1	1153.1	39397	0.5
15	0.1 to 0.8	8	38824.5	1050.5	39314	0.5
16	0.1 to 0.8	8	38714.8	940.8	39236	0.5
17	0.1 to 0.8	8	38611.4	837.4	39270	0.4
18	0.1 to 0.7	7	38640.1	866.1	39220	0.4
19	0.1 to 0.7	7	38708.0	934.0	39476	0.6
20	0.1 to 0.7	7	38628.1	854.1	39403	0.6

Table H.15. *M_031_3* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1 to 0.5 and 0.7	6	38788.5	475.5	39323	0.7
2	0.1 to 0.6, 0.8 and 0.9	8	39769.6	1456.6	42760	0.9
3	0.1 to 0.9	9	39983.9	1670.9	41887	0.9
4	0.1 to 0.9	9	39574.9	1261.9	41482	0.8
5	0.1 to 0.9	9	39485.8	1172.8	42025	0.8
6	0.1 to 0.9	9	39320.1	1007.1	41512	0.8
7	0.1 to 0.9	9	39207.1	894.1	41073	0.8
8	0.1, 0.2, 0.4 to 0.8	7	39414.7	1101.7	40701	0.8
9	0.1, 0.2, 0.4 to 0.8	7	39193.1	880.1	40158	0.8
10	0.1, 0.2, 0.4 to 0.8	7	39068.1	755.1	39942	0.8
11	0.1, 0.2, 0.4 to 0.8	7	38926.9	613.9	39697	0.8
12	0.1, 0.2, 0.4 to 0.8	7	38871.9	558.9	39352	0.6
13	0.1, 0.2, 0.4 to 0.8	7	38786.6	473.6	39235	0.6
14	0.1, 0.2, 0.5 to 0.8	6	38931.7	618.7	40154	0.7
15	0.1, 0.2 , 0.6 and 0.7	4	39120.5	807.5	39933	0.7
16	0.1, 0.2 , 0.6 and 0.7	4	38965.0	652.0	39475	0.7
17	0.1, 0.2 , 0.6 and 0.7	4	38929.5	616.5	39326	0.7
18	0.1, 0.2 , 0.6 and 0.7	4	38872.8	559.8	39216	0.7
19	0.1 to 0.3, 0.6 and 0.7	5	38956.8	643.8	39339	0.6
20	0.1 to 0.3, 0.6 and 0.7	5	38901.0	588.0	39242	0.6

Table H.16. *M_031_4* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1 to 0.9	9	38336.1	828.1	39507	0.7
2	0.1 to 0.9	9	39557.4	2049.4	42766	0.9
3	0.1 to 0.9	9	39786.6	2278.6	41632	0.9
4	0.1 to 0.9	9	39558.9	2050.9	41228	0.8
5	0.1 to 0.9	9	39257.6	1749.6	40451	0.8
6	0.1 to 0.9	9	39115.6	1607.6	41054	0.8
7	0.1 to 0.9	9	38919.9	1411.9	40516	0.8
8	0.1 to 0.8	8	38818.3	1310.3	40136	0.8
9	0.1 to 0.8	8	38680.9	1172.9	39788	0.8
10	0.1 to 0.9	9	38463.2	955.2	39449	0.8
11	0.1, 0.3 to 0.8	7	38454.9	946.9	38920	0.7
12	0.1, 0.3 to 0.8	7	38492.3	984.3	39441	0.6
13	0.1 to 0.8	8	38393.3	885.3	39287	0.6
14	0.1, 0.3 to 0.7	6	38341.2	833.2	39159	0.6
15	0.1, 0.3 to 0.7	6	38248.3	740.3	39028	0.6
16	0.1, 0.3 to 0.7	6	38311.8	803.8	38979	0.6
17	0.1, 0.3 to 0.7	6	38237.2	729.2	38833	0.6
18	0.1, 0.3 to 0.7	6	38095.2	587.2	38690	0.6
19	0.1 to 0.7	7	38058.1	550.1	38589	0.6
20	0.1 to 0.7	7	37923.0	415.0	38285	0.2

Table H.17. *M_032_1* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1, 0.2, 0.4 to 0.9	8	38442.9	935.9	39404	0.7
2	0.1 to 0.9	9	39620.7	2113.7	42383	0.9
3	0.1 to 0.9	9	39881.8	2374.8	42603	0.9
4	0.1 to 0.9	9	39668.1	2161.1	41695	0.9
5	0.1 to 0.9	9	39444.6	1937.6	41689	0.8
6	0.1 to 0.9	9	39341.1	1834.1	40958	0.8
7	0.1 to 0.9	9	39105.8	1598.8	40463	0.8
8	0.1 and 0.3 to 0.9	8	38993.9	1486.9	40075	0.8
9	0.1 to 0.9	9	38543.0	1036.0	39362	0.8
10	0.1 and 0.3 to 0.9	8	38498.6	991.6	39100	0.7
11	0.1 and 0.3 to 0.8	7	38378.4	871.4	38822	0.7
12	0.1 and 0.3 to 0.8	7	38443.0	936.0	39365	0.6
13	0.1 and 0.3 to 0.8	7	38327.3	820.3	39203	0.6
14	0.1 and 0.3 to 0.7	6	38267.0	760.0	39047	0.6
15	0.1 and 0.3 to 0.7	6	38171.5	664.5	38877	0.6
16	0.1 and 0.3 to 0.7	6	38291.0	784.0	38839	0.6
17	0.1 and 0.3 to 0.7	6	38102.7	595.7	38710	0.6
18	0.1 and 0.3 to 0.7	6	38028.0	521.0	38577	0.6
19	0.1 and 0.3 to 0.7	6	37893.7	386.7	38152	0.6
20	0.1 and 0.4 to 0.7	6	37886.6	379.6	38074	0.4

Table H.18. *M_032_2* results.

Radius pixels	Mask weights	Total	*Mean* pixels	*Difference* pixels	*Max area* pixels	Mask weight
1	0.1 to 0.8	8	38416.3	1332.3	39991	0.7
2	0.1 to 0.9	9	39559.8	2475.8	41106	0.9
3	0.1 to 0.9	9	40129.2	3045.2	42716	0.9
4	0.1 to 0.9	9	39990.3	2906.3	41833	0.9
5	0.1 to 0.9	9	39625.3	2541.3	40780	0.8
6	0.1 to 0.9	9	39541.7	2457.7	41333	0.8
7	0.1 to 0.9	9	39271.4	2187.4	40701	0.8
8	0.1 to 0.9	9	39199.4	2115.4	40287	0.8
9	0.1 to 0.9	9	38960.4	1876.4	39985	0.8
10	0.1 to 0.9	9	38813.1	1729.1	39701	0.8
11	0.1 to 0.9	9	38563.7	1479.7	39249	0.7
12	0.1 to 0.8	8	38529.3	1445.3	39062	0.7
13	0.1 to 0.8	8	38462.6	1378.6	38936	0.7
14	0.1 to 0.8	8	38359.9	1275.9	38781	0.4
15	0.1 to 0.8	8	38282.5	1198.5	38759	0.5
16	0.1 to 0.8	8	38278.3	1194.3	39324	0.6
17	0.1 to 0.7	7	38202.0	1118.0	38580	0.5
18	0.1 to 0.7	7	38257.6	1173.6	39181	0.6
19	0.1 to 0.7	7	38169.6	1085.6	39071	0.6
20	0.1 to 0.6	6	38233.8	1149.8	38952	0.6

Table H.19. *M_032_3* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1 to 0.9	9	38544.6	2032.6	40386	0.8
2	0.1 to 0.9	9	39789.0	3277.0	42554	0.9
3	0.1 to 0.9	9	39837.9	3325.9	41792	0.8
4	0.1 to 0.9	9	39435.9	2923.9	41043	0.8
5	0.1 to 0.9	9	39274.1	2762.1	41365	0.8
6	0.1 to 0.9	9	39088.7	2576.7	40789	0.8
7	0.1 to 0.9	9	38738.8	2226.8	40255	0.8
8	0.1 to 0.9	9	38590.6	2078.6	39809	0.6
9	0.1 to 0.8	8	38664.3	2152.3	39561	0.6
10	0.1 to 0.9	9	38565.9	2053.9	39508	0.5
11	0.1 to 0.8	8	38487.6	1975.6	39441	0.5
12	0.1 to 0.8	8	38609.3	2097.3	39432	0.7
13	0.1 to 0.8	8	38445.8	1933.8	39151	0.5
14	0.1 to 0.8	8	38291.0	1779.0	39131	0.5
15	0.1 to 0.7	7	38410.6	1898.6	39032	0.5
16	0.1 to 0.7	7	38225.7	1713.7	38941	0.4
17	0.1 to 0.7	7	38042.0	1530.0	38810	0.5
18	0.1 to 0.7	7	37922.4	1410.4	38710	0.5
19	0.1 to 0.7	7	37831.3	1319.3	38601	0.5
20	0.1 to 0.7	7	37654.9	1142.9	38323	0.4

Table H.20. *M_032_4* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1 to 0.9	9	38383.3	1193.3	39998	0.7
2	0.1 to 0.9	9	39684.8	2494.8	42627	0.9
3	0.1 to 0.9	9	39656.6	2466.6	41483	0.9
4	0.1 to 0.9	9	39428.7	2238.7	41076	0.8
5	0.1 to 0.9	9	39263.7	2073.7	41417	0.8
6	0.1 to 0.9	9	39073.4	1883.4	40852	0.8
7	0.1 to 0.9	9	38762.1	1572.1	40324	0.8
8	0.1 to 0.8	8	38717.0	1527.0	39984	0.8
9	0.1 to 0.8	8	38631.0	1441.0	39656	0.6
10	0.1 to 0.9	9	38346.3	1156.3	39456	0.6
11	0.1 to 0.8	8	38479.8	1289.8	39777	0.7
12	0.1 to 0.8	8	38363.1	1173.1	39556	0.7
13	0.1 to 0.8	8	38197.0	1007.0	39123	0.7
14	0.1 to 0.7	7	38487.9	1297.9	39205	0.5
15	0.1 to 0.7	7	38255.6	1065.6	38767	0.7
16	0.1 to 0.7	7	38137.3	947.3	39088	0.5
17	0.1 to 0.7	7	37995.6	805.6	38969	0.5
18	0.1 to 0.7	7	38029.0	839.0	38865	0.5
19	0.1 to 0.7	7	37967.1	777.1	38770	0.5
20	0.1 to 0.7	7	37996.4	806.4	38667	0.5

Table H.21. *M_033_1* results.

Radius pixels	Mask weights	Total	Mean pixels	Difference pixels	Max area pixels	Mask weight
1	0.1 to 0.8	8	38443.3	1243.3	40181	0.7
2	0.1 to 0.9	9	39646.3	2446.3	41905	0.9
3	0.1 to 0.9	9	39722.1	2522.1	42524	0.9
4	0.1 to 0.9	9	39547.8	2347.8	41643	0.9
5	0.1 to 0.9	9	39340.2	2140.2	41663	0.8
6	0.1 to 0.9	9	39304.0	2104.0	40856	0.8
7	0.1 to 0.9	9	38890.0	1690.0	40345	0.8
8	0.1 and 0.3 to 0.9	8	38787.2	1587.2	39967	0.8
9	0.1 to 0.9	9	38376.3	1176.3	39240	0.8
10	0.1 and 0.3 to 0.9	8	38171.2	971.2	38993	0.7
11	0.1 and 0.3 to 0.9	8	38527.1	1327.1	39842	0.7
12	0.1 and 0.3 to 0.8	7	38418.4	1218.4	39591	0.7
13	0.1 and 0.3 to 0.8	7	38262.8	1062.8	39093	0.7
14	0.1 and 0.3 to 0.7	6	38152.6	952.6	38928	0.6
15	0.1 and 0.3 to 0.7	6	38208.3	1008.3	38802	0.7
16	0.1 and 0.3 to 0.7	6	38058.3	858.3	38602	0.6
17	0.1 and 0.3 to 0.7	6	37917.0	717.0	38194	0.2
18	0.1 and 0.3 to 0.7	6	37861.6	661.6	38134	0.6
19	0.1 and 0.3 to 0.7	6	37756.9	556.9	38086	0.2
20	0.1 and 0.4 to 0.7	5	37837.7	637.7	38813	0.5

Table H.22. *M_033_2* results.

Radius	Mask weights	Total	Mean	Difference	Max area	Mask weight
pixels			pixels	pixels	pixels	
1	0.1 to 0.8	8	38619.1	619.1	39847	0.7
2	0.1 to 0.9	9	39848.0	1848.0	41054	0.9
3	0.1 to 0.9	9	40130.7	2130.7	42671	0.9
4	0.1 to 0.9	9	39938.6	1938.6	41783	0.9
5	0.1 to 0.9	9	39900.5	1900.5	41828	0.8
6	0.1 to 0.9	9	39459.9	1459.9	41261	0.8
7	0.1 to 0.9	9	39534.5	1534.5	40572	0.8
8	0.1 to 0.9	9	39267.0	1267.0	40171	0.8
9	0.1 to 0.9	9	39121.3	1121.3	39860	0.8
10	0.1 to 0.9	9	39011.7	1011.7	39263	0.7
11	0.1 to 0.9	9	38791.7	791.7	39058	0.7
12	0.1 to 0.8	8	38736.0	736.0	39590	0.6
13	0.1 to 0.8	8	38626.7	626.7	39472	0.6
14	0.1 to 0.8	8	38662.7	662.7	39327	0.6
15	0.1 to 0.8	8	38589.5	589.5	39284	0.6
16	0.1 to 0.8	8	38505.3	505.3	39159	0.6
17	0.1 to 0.7	7	38575.5	575.5	39062	0.6
18	0.1 to 0.7	7	38480.8	480.8	38886	0.6
19	0.1 to 0.7	7	38364.2	364.2	38769	0.6
20	0.1 to 0.6	6	38388.3	388.3	38677	0.6

Table H.23. *M_033_3* results.

Radius	Mask weights	Total	Mean	Difference	Max area	Mask weight
pixels			pixels	pixels	pixels	
1	0.1 to 0.9	9	38245.6	2086.6	40119	0.8
2	0.1 to 0.9	9	39555.6	3396.6	42522	0.9
3	0.1 to 0.9	9	39358.6	3199.6	41393	0.9
4	0.1 to 0.9	9	39318.2	3159.2	41941	0.8
5	0.1 to 0.9	9	39006.4	2847.4	41286	0.8
6	0.1 to 0.9	9	38813.6	2654.6	40680	0.8
7	0.1 to 0.9	9	38480.0	2321.0	40200	0.8
8	0.1 to 0.9	9	38224.4	2065.4	39499	0.8
9	0.1 to 0.8	8	38438.9	2279.9	40090	0.7
10	0.1 to 0.9	9	38201.9	2042.9	39762	0.7
11	0.1 to 0.8	8	38188.6	2029.6	39563	0.7
12	0.1 to 0.8	8	38156.5	1997.5	39042	0.7
13	0.1 to 0.8	8	37965.3	1806.3	39035	0.5
14	0.1 to 0.8	8	37898.1	1739.1	38908	0.5
15	0.1 to 0.7	7	37938.9	1779.9	38770	0.5
16	0.1 to 0.7	7	37800.1	1641.1	38751	0.5
17	0.1 to 0.7	7	37752.4	1593.4	38380	0.4
18	0.1 to 0.7	7	37655.6	1496.6	38307	0.4
19	0.1 to 0.7	7	37519.7	1360.7	38226	0.4
20	0.1 to 0.7	7	37416.1	1257.1	38152	0.4

Table H.24. *M_033_4* results.

Radius (pixels)	Mask weights	Total	Mean (pixels)	Difference (pixels)	Max area (pixels)	Mask weight
1	0.1 to 0.9	9	38449.6	2075.6	40964	0.8
2	0.1 to 0.9	9	39675.0	3301.0	42619	0.9
3	0.1 to 0.9	9	39554.8	3180.8	41419	0.9
4	0.1 to 0.9	9	39333.8	2959.8	40857	0.8
5	0.1 to 0.9	9	39135.2	2761.2	41339	0.8
6	0.1 to 0.9	9	38936.7	2562.7	40774	0.8
7	0.1 to 0.9	9	38624.3	2250.3	40285	0.8
8	0.1 to 0.8	8	38403.2	2029.2	39904	0.8
9	0.1 to 0.8	8	38641.1	2267.1	39522	0.6
10	0.1 to 0.9	9	38308.8	1934.8	39937	0.7
11	0.1 to 0.8	8	38483.5	2109.5	39653	0.7
12	0.1 to 0.8	8	38213.6	1839.6	39473	0.7
13	0.1 to 0.8	8	38013.5	1639.5	38977	0.7
14	0.1 to 0.7	7	38179.5	1805.5	39116	0.5
15	0.1 to 0.7	7	38238.4	1864.4	39015	0.5
16	0.1 to 0.7	7	37993.7	1619.7	38867	0.5
17	0.1 to 0.7	7	38013.6	1639.6	38775	0.5
18	0.1 to 0.7	7	37896.0	1522.0	38679	0.5
19	0.1 to 0.7	7	37641.7	1267.7	38382	0.5
20	0.1 to 0.7	7	37739.1	1365.1	38427	0.4

Appendix I – *Sacharomyces cerevisiae* pictures

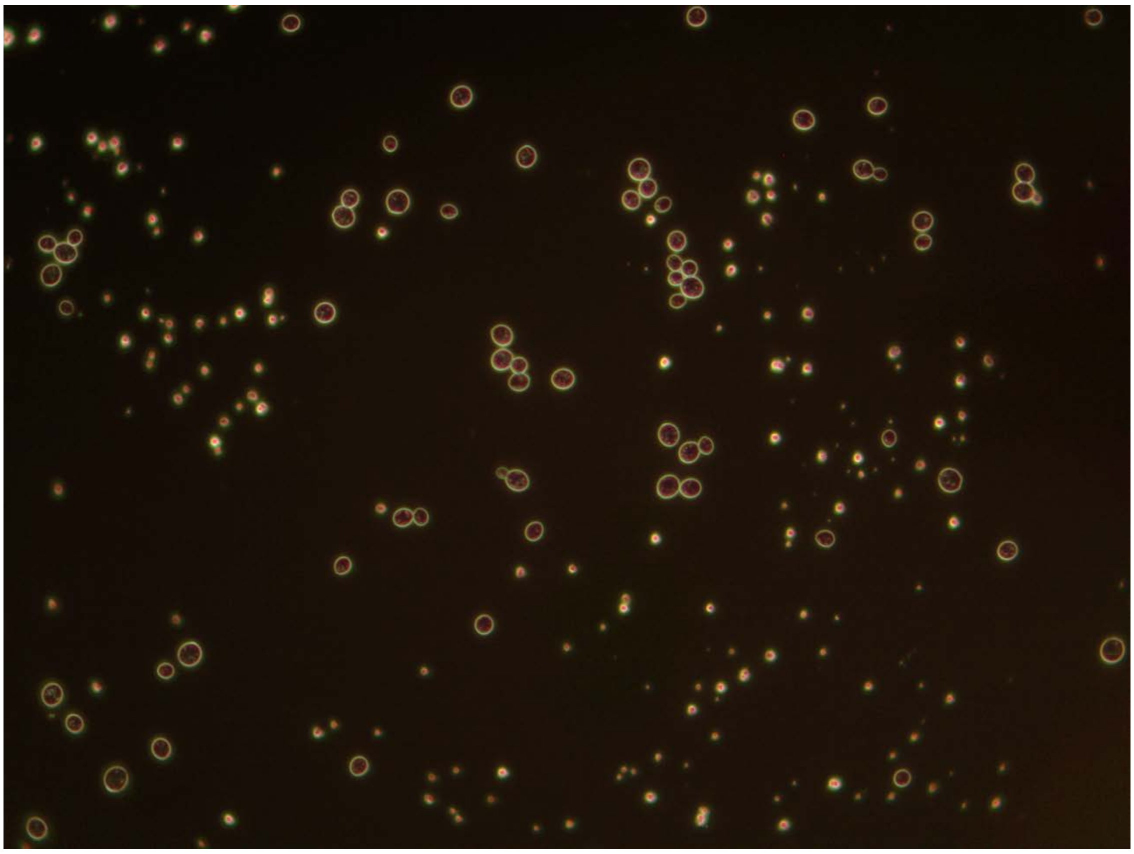


Figure I.1. N-Hefe_400x_dark-field_1 (*Image 1*).

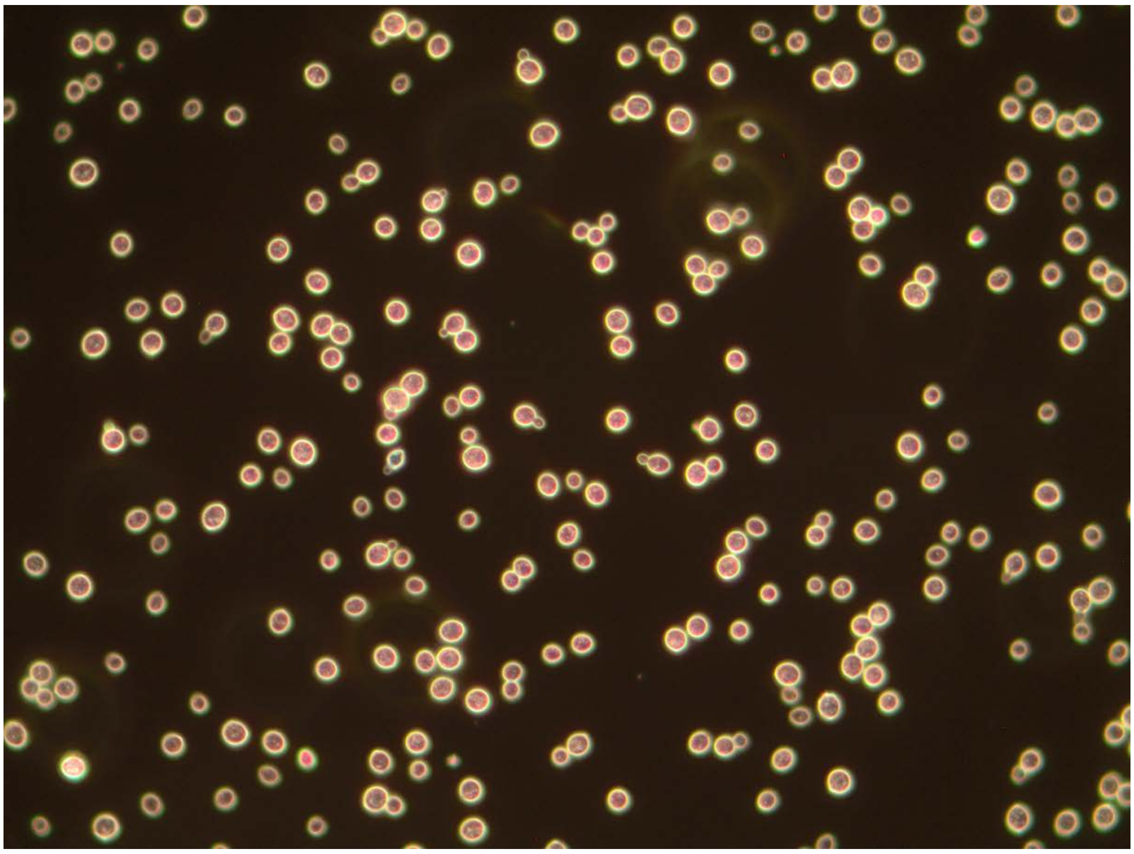


Figure I.2. P hefe 800ms DF 400 (*Image 2*).

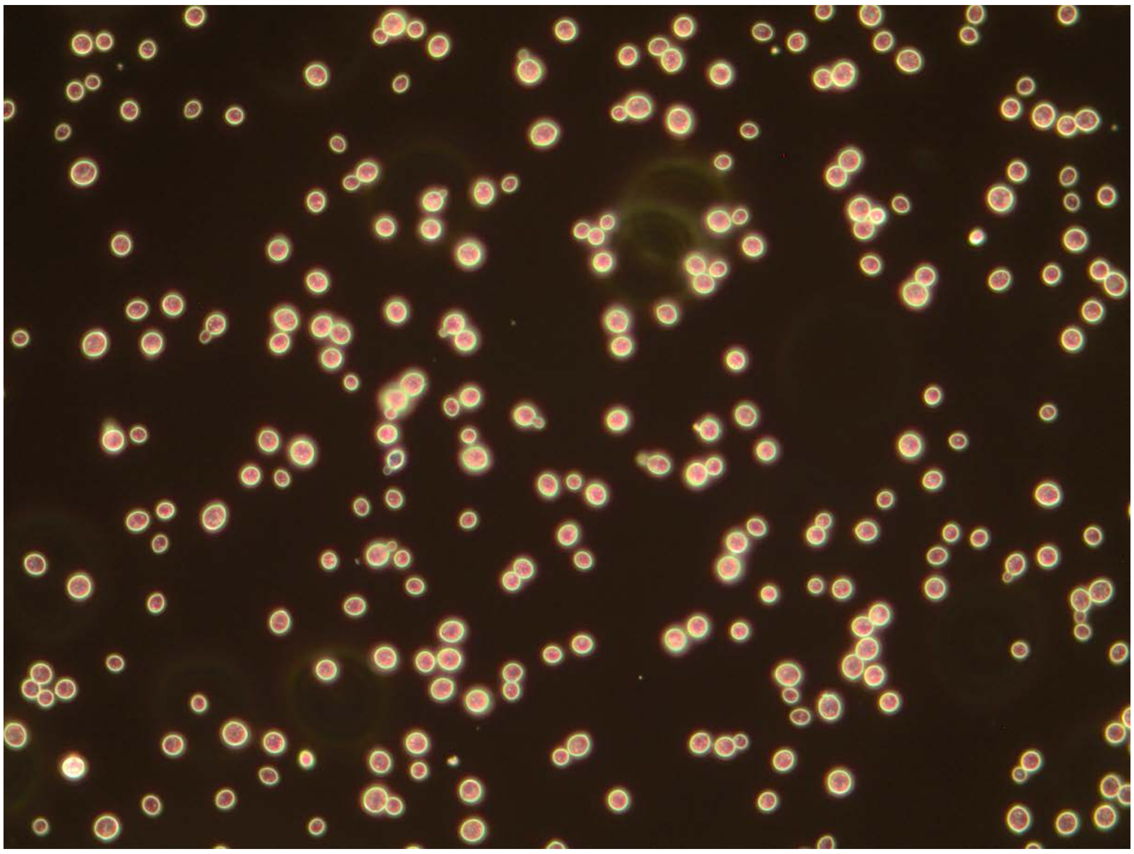


Figure I.3. P hefe 1200ms DF 400 (*Image 3*).

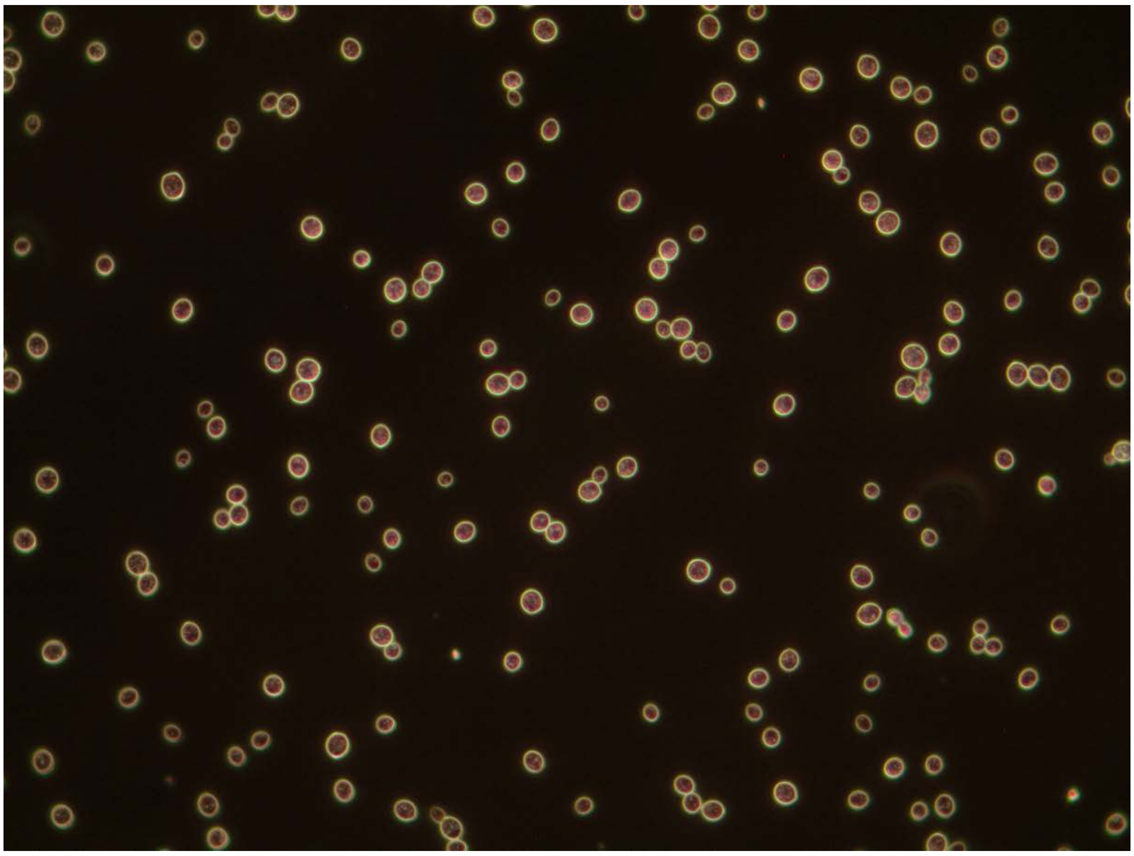


Figure I.4. N_Yeast 400x DF 300ms (*Image 4*).

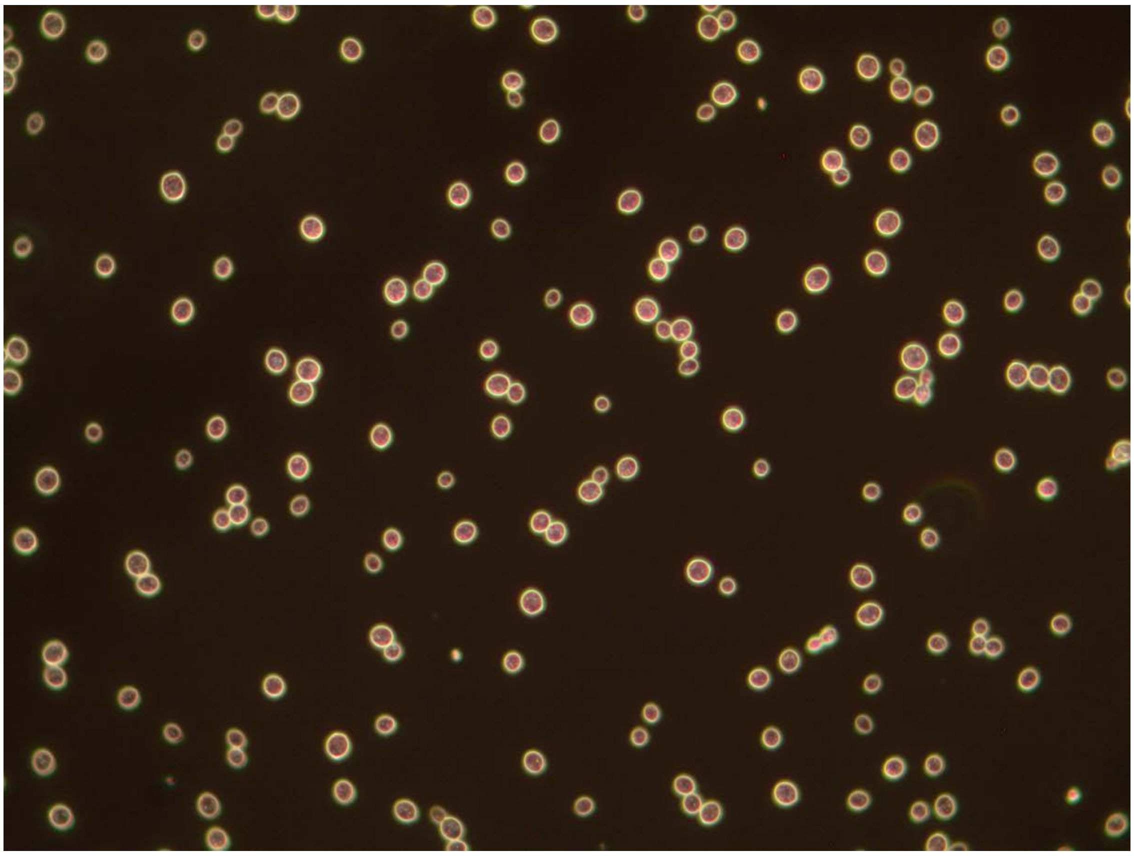


Figure I.5. N_Yeast 400x DF 500ms (*Image 5*).

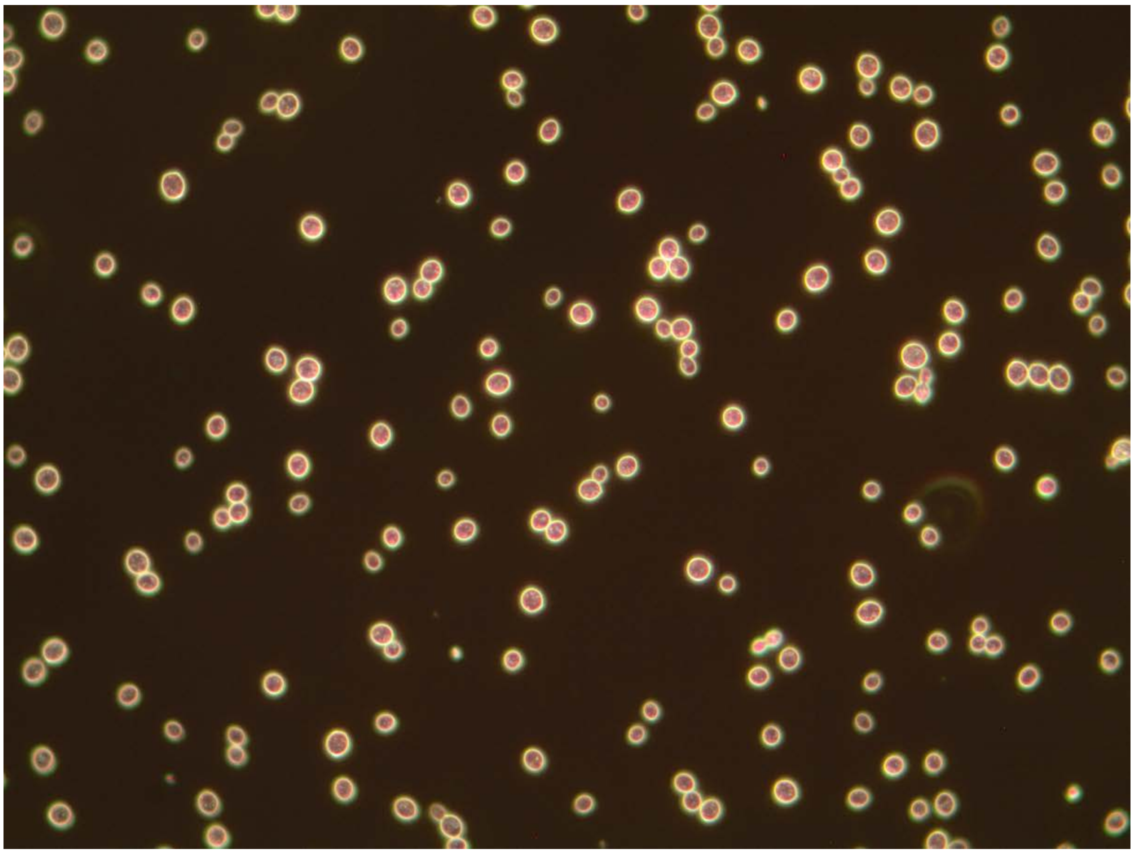


Figure I.6. N_Yeast 400x DF 700ms (*Image 6*).

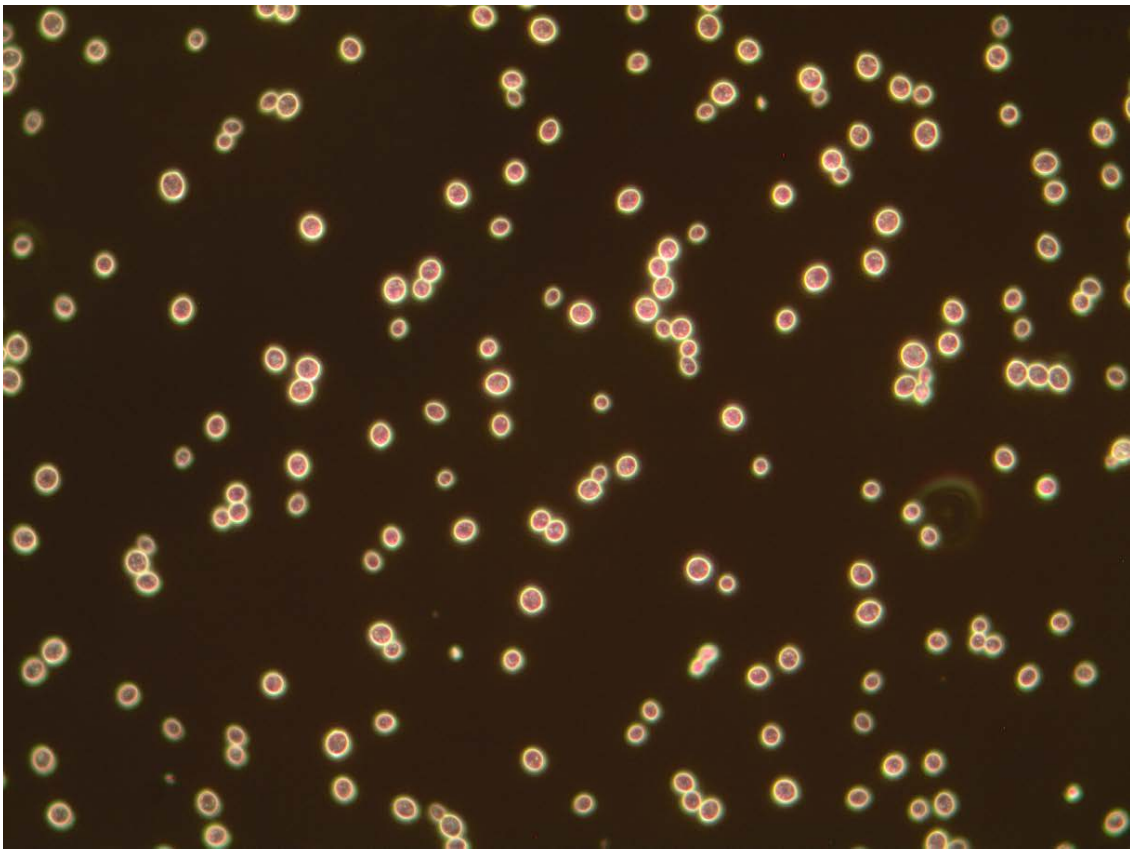


Figure I.7. N_Yeast 400x DF 1000ms (*Image 7*).

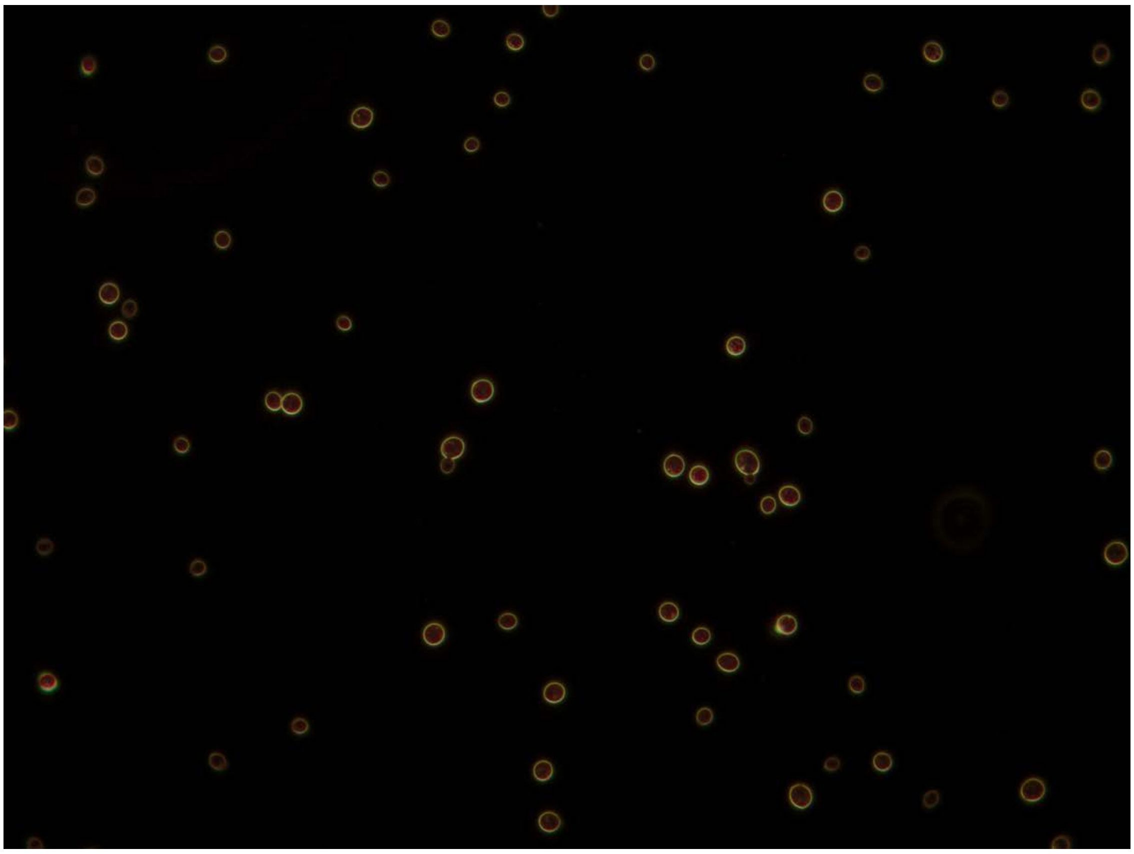


Figure I.8. N Hefe 100ms (*Image 8*).

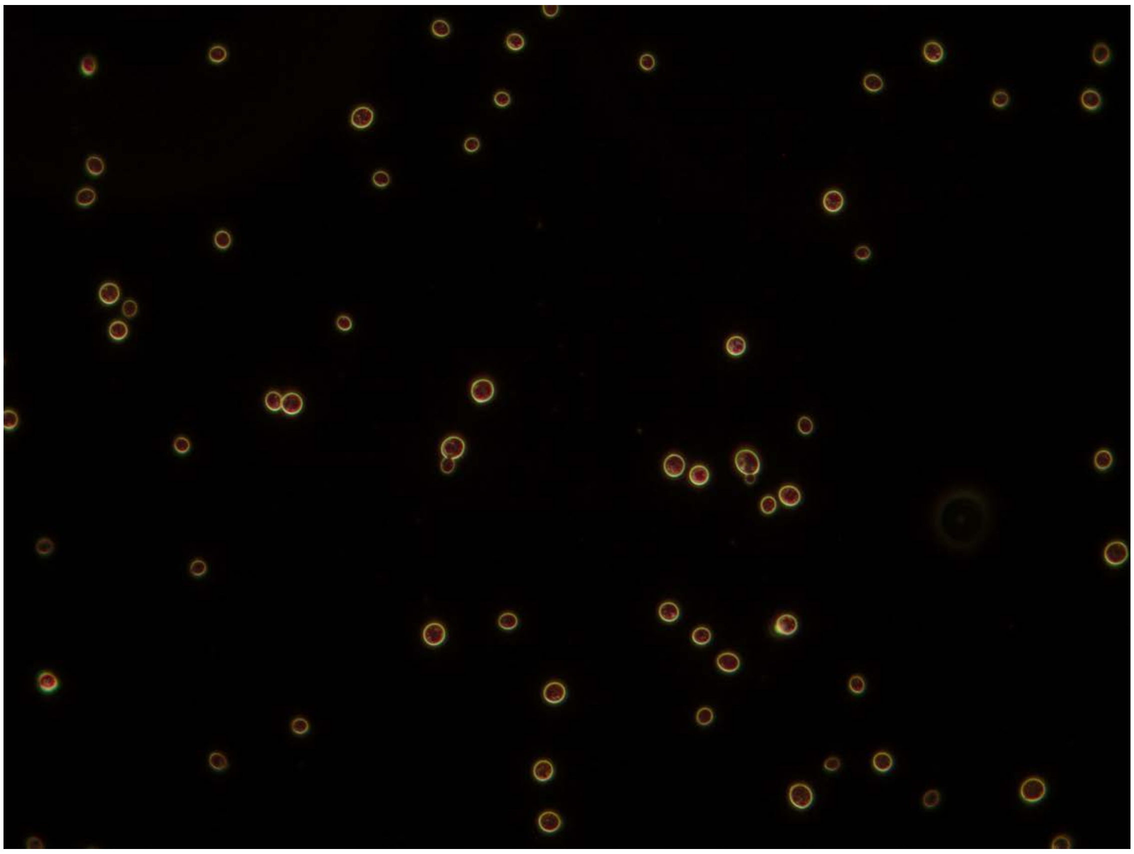


Figure I.9. N Hefe 150ms (*Image 9*).

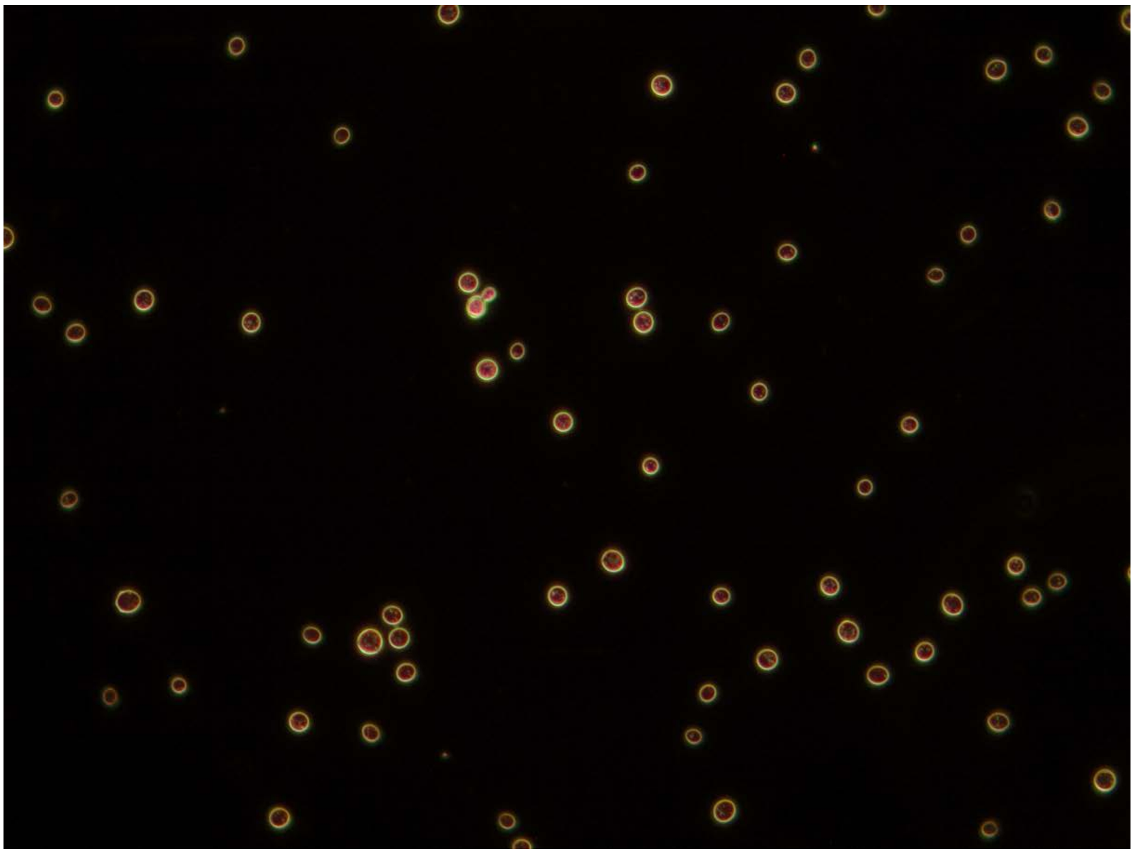


Figure I.10. N Hefe 200ms 2 (*Image 10*).

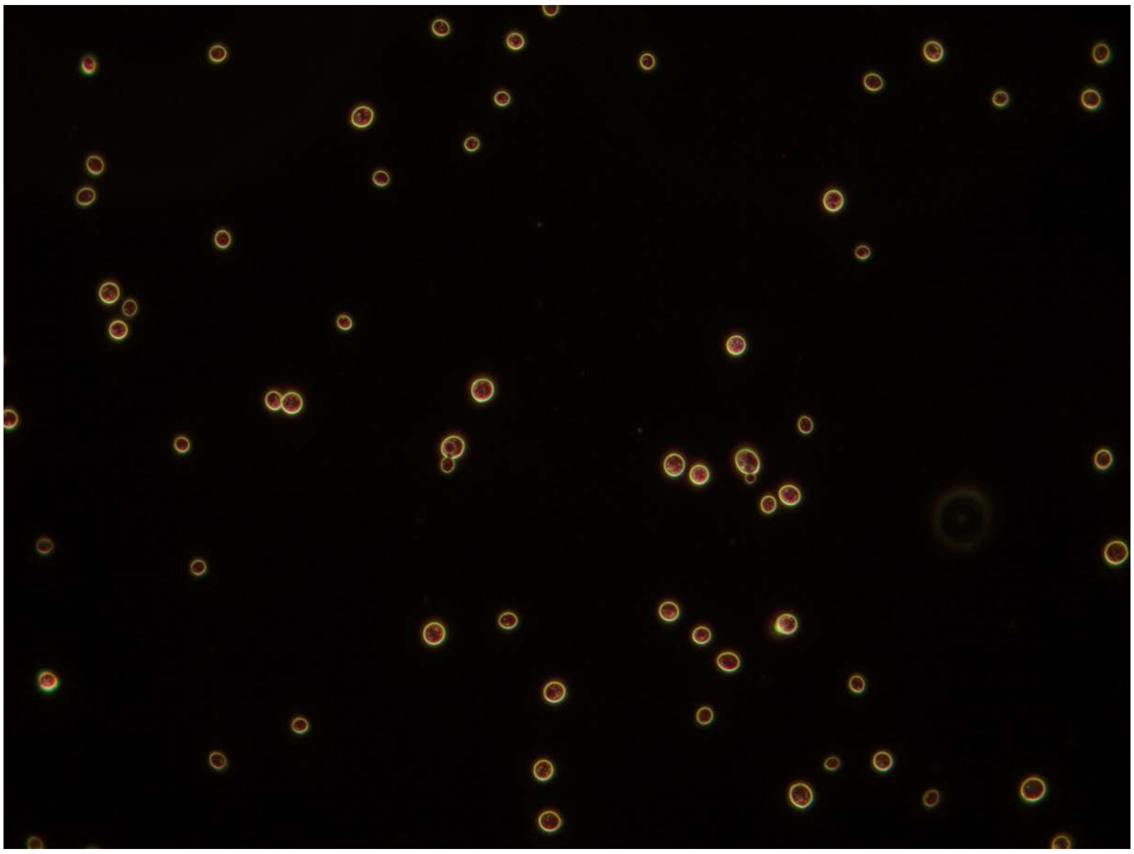


Figure I.11. N Hefe 200ms (*Image 11*).

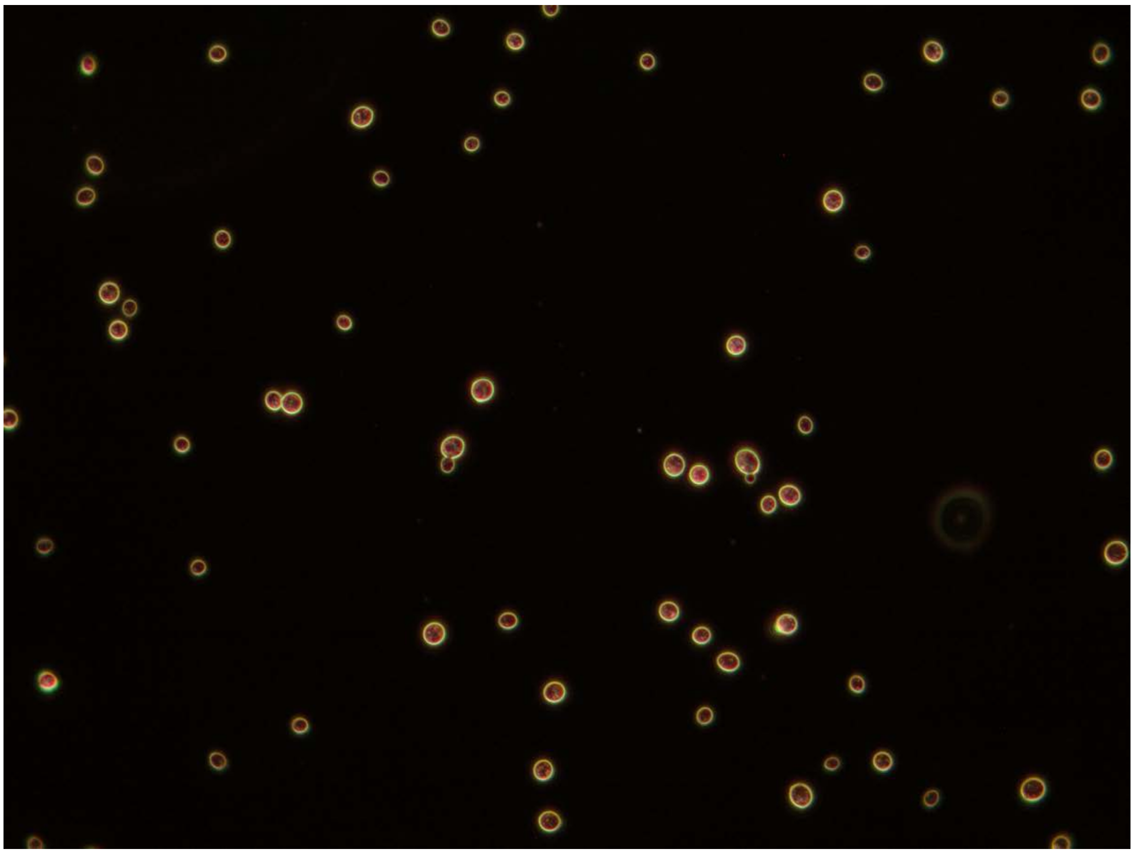


Figure I.12. N Hefe 250ms (*Image 12*).

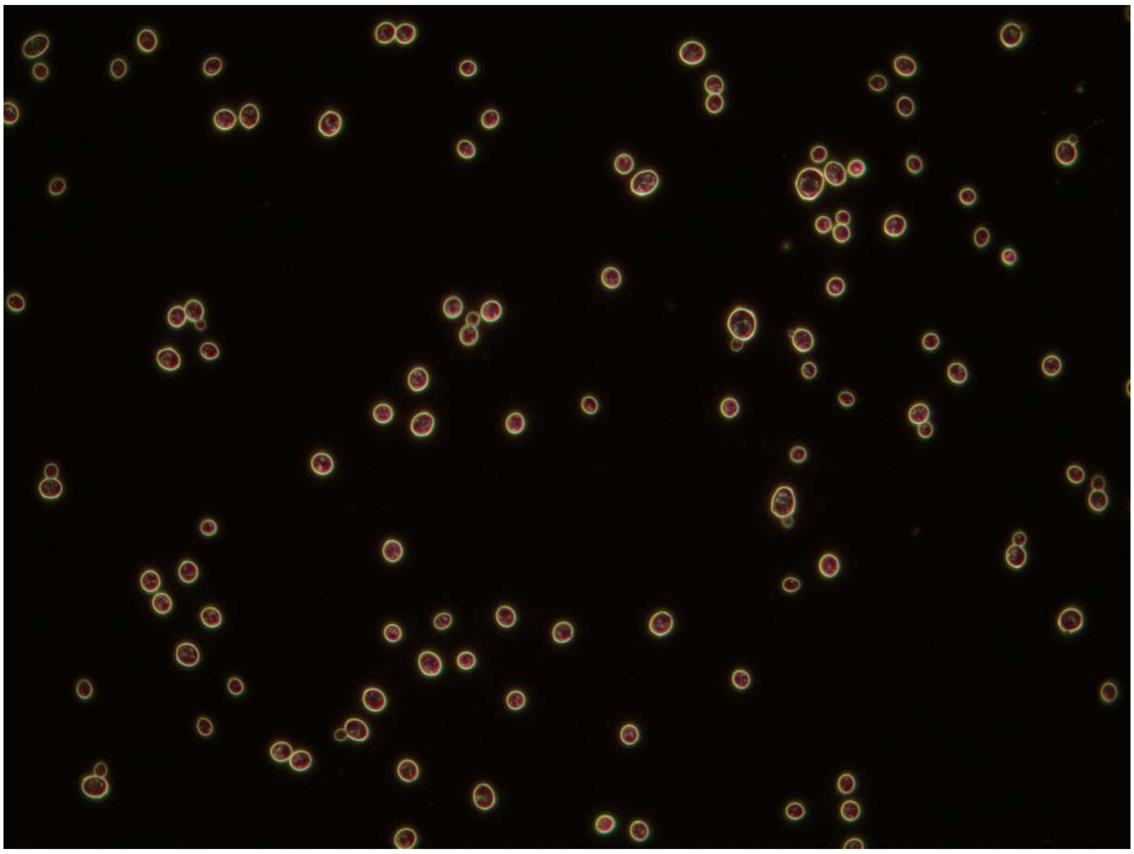


Figure I.13. Bild_227 (*Image 13*).

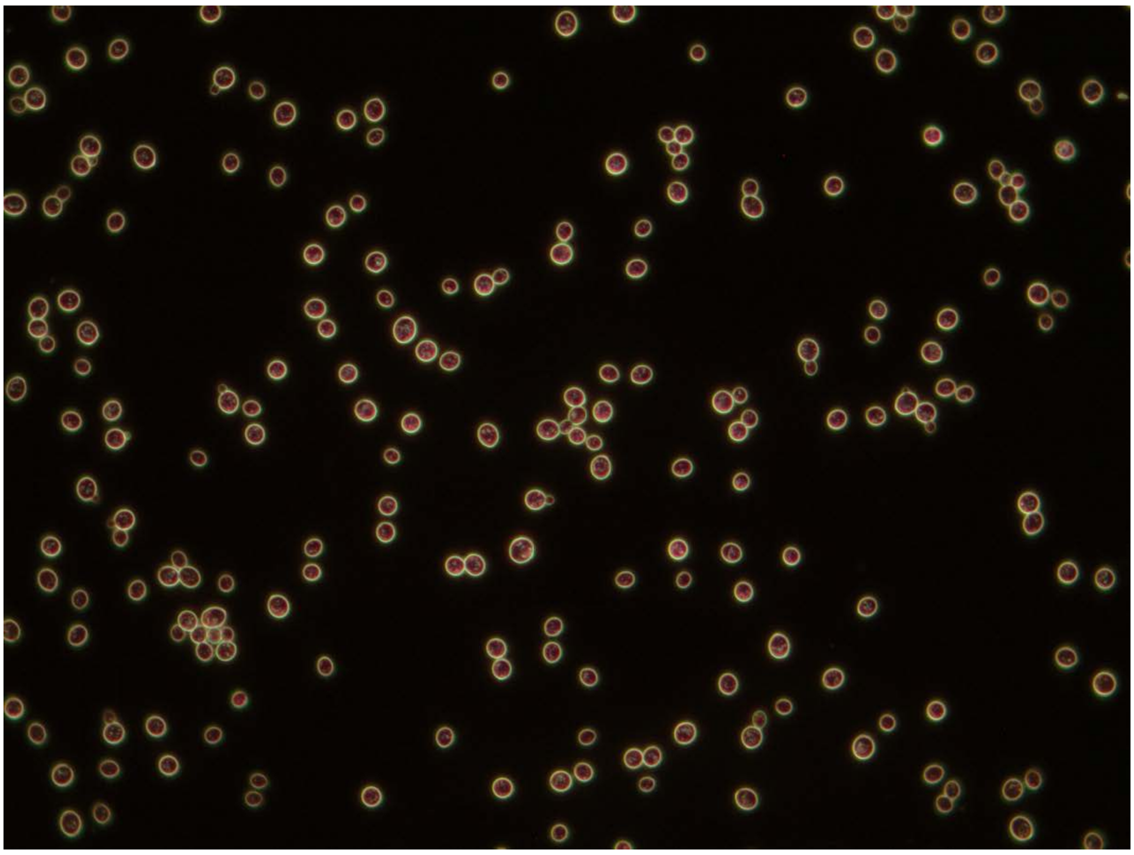


Figure I.14. Bild_229 (*Image 14*).

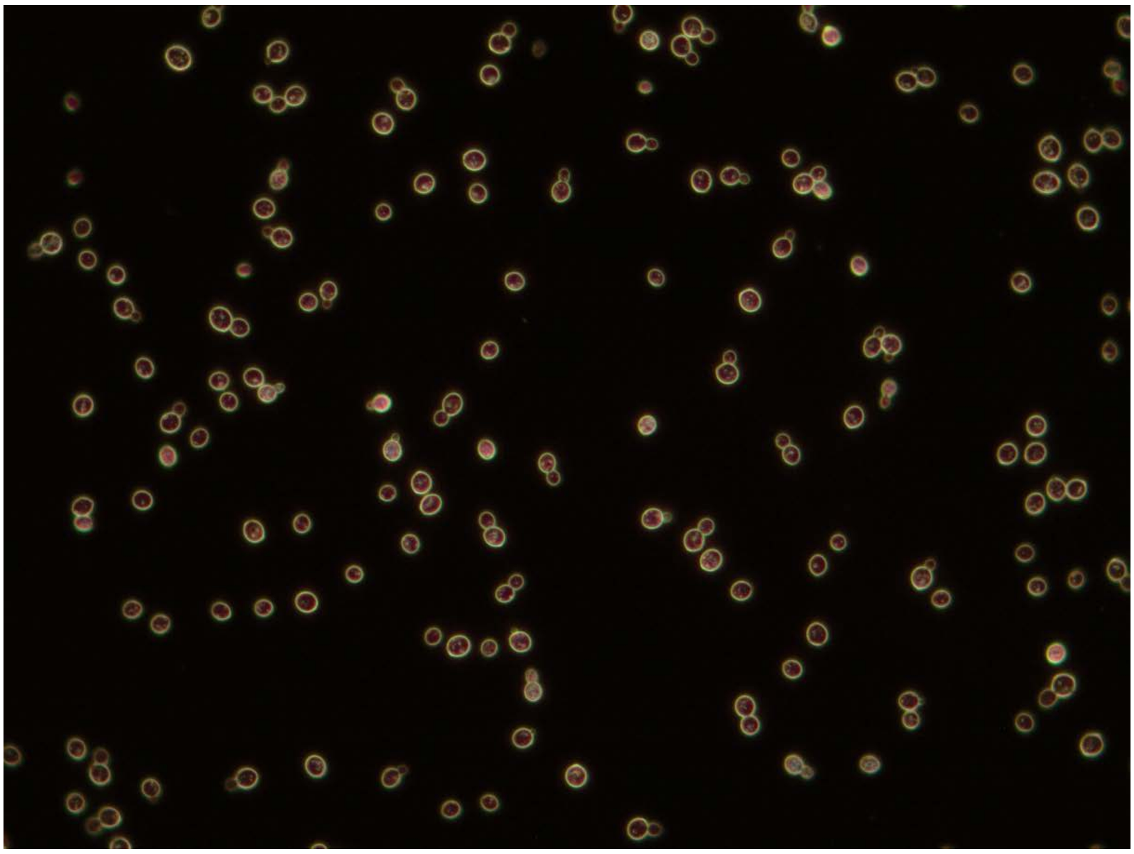


Figure I.15. Bild_281 (*Image 15*).

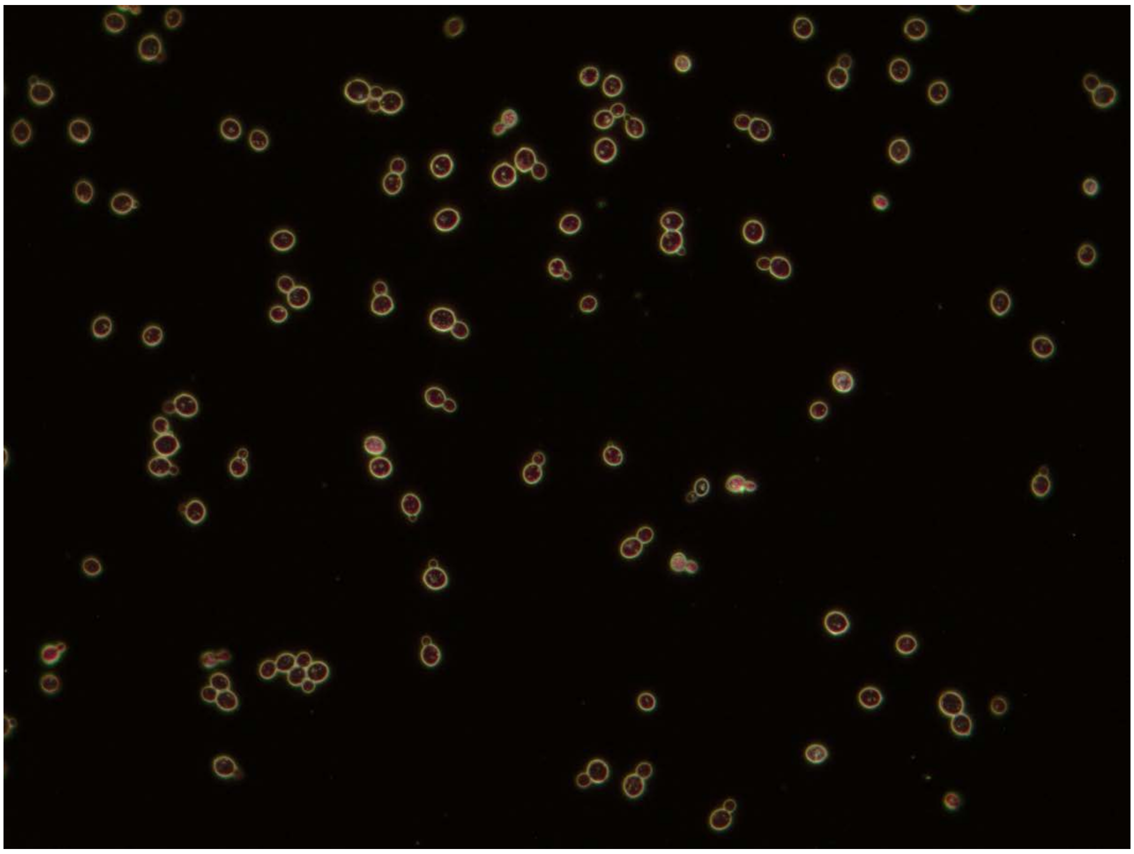


Figure I.16. Bild_289 (*Image 16*).

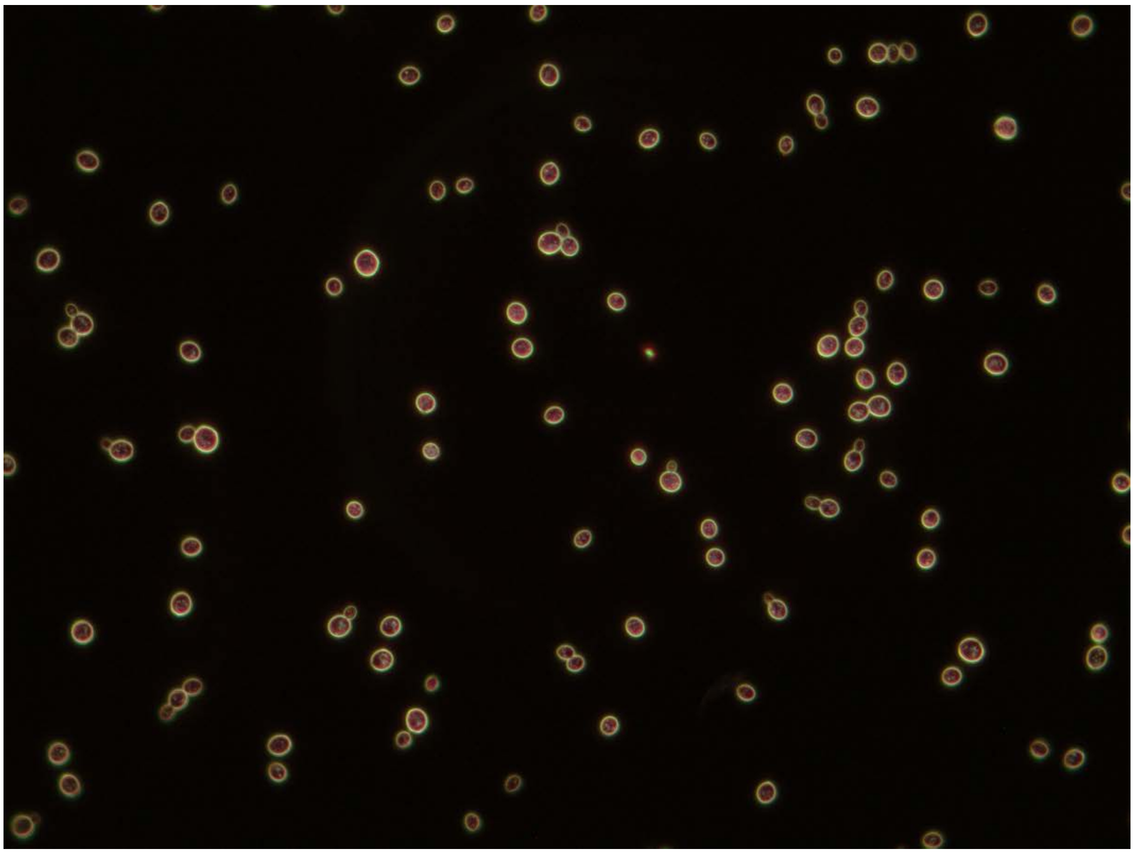


Figure I.17. 1B 1_100 (*Image 17*).

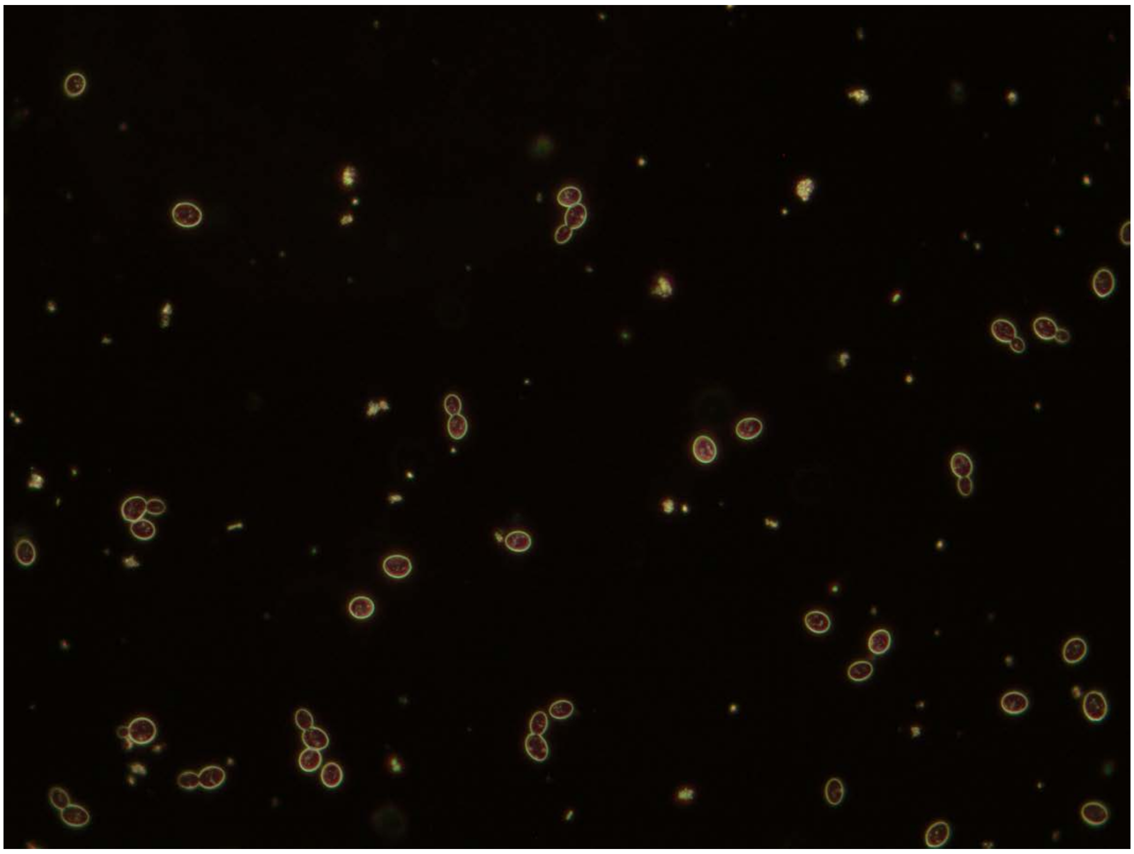


Figure I.18. 1CV1 1_10 (*Image 18*).

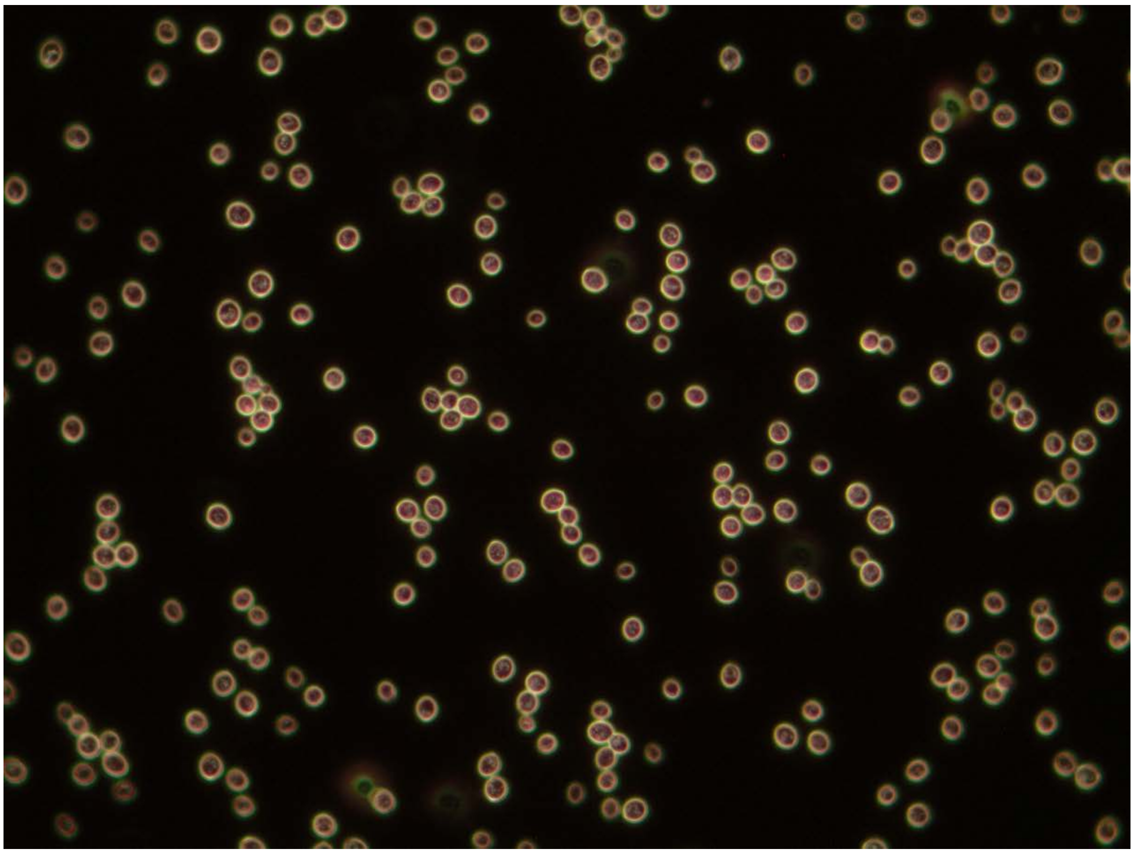


Figure I.19. 1N 1_100 (*Image 19*).

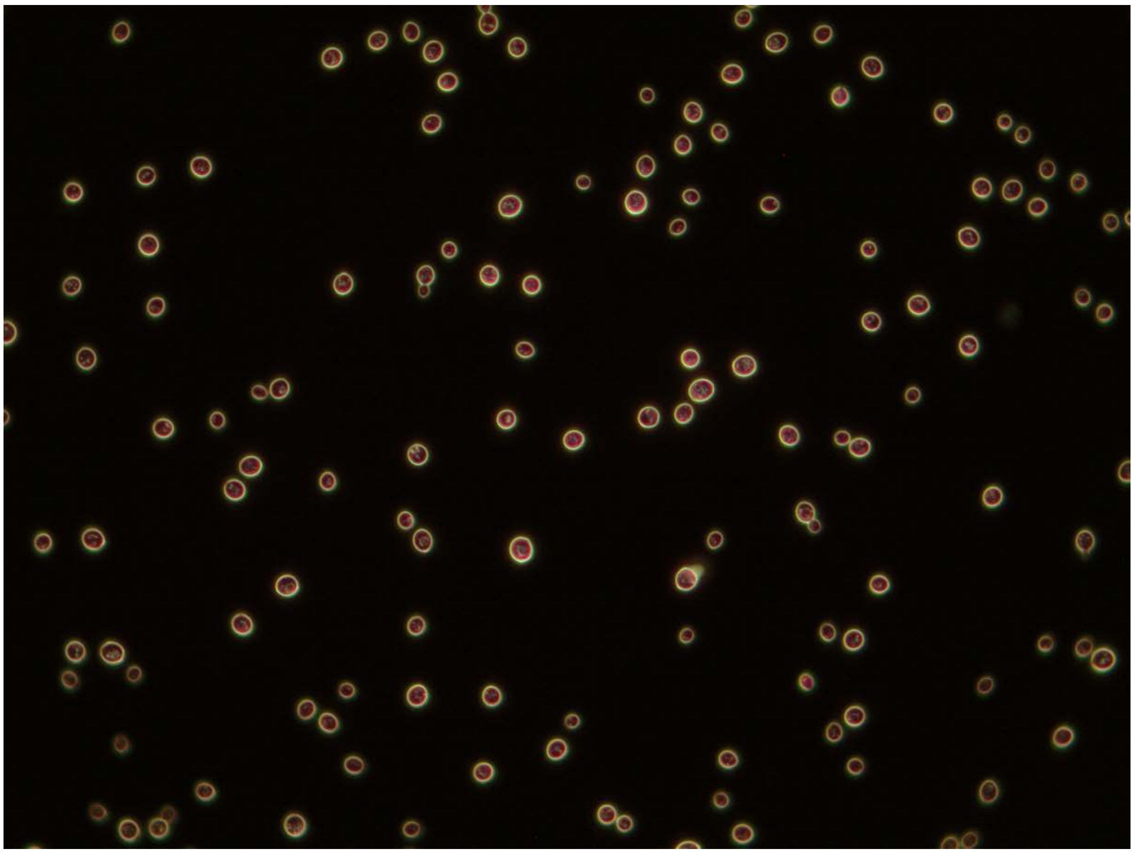


Figure I.20. 1P 1_100 (*Image 20*).

Appendix J – Other images *Group 2* results

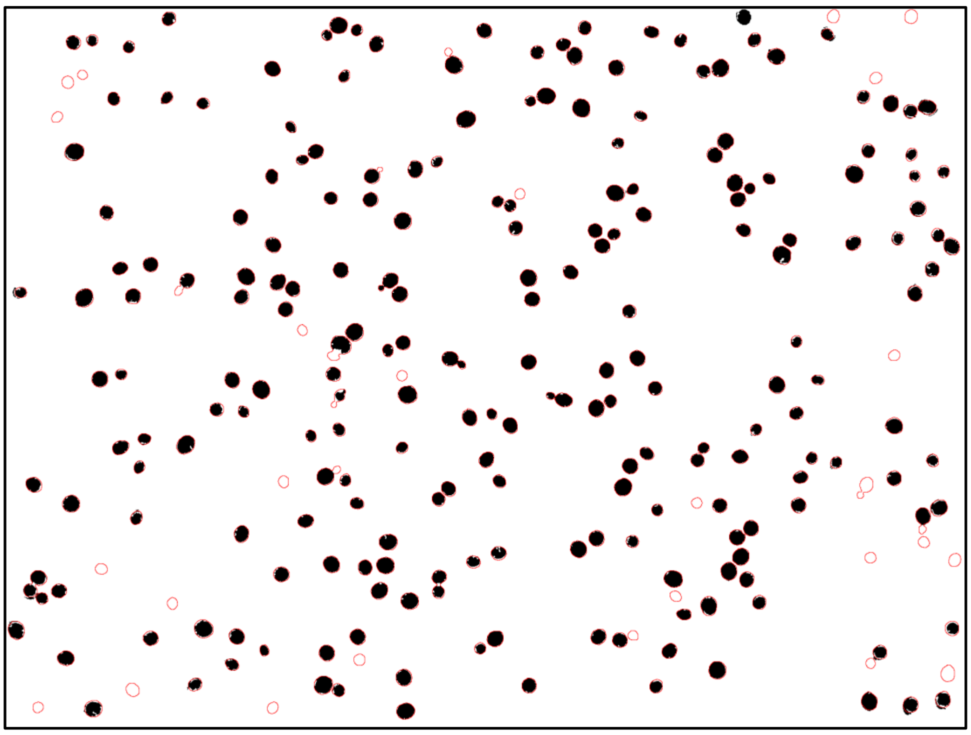


Figure J.1. Resulting mask from the sum of masks from experiments *M_032_1*, *M_032_3* and *M_031_4* with mask radius and weight equal to 2 pixels and 0.9, respectively, and experiment *M_031_2*, with mask radius and weight equal to 3 pixels and 0.9, respectively. All experiments were performed with *Image 2*.

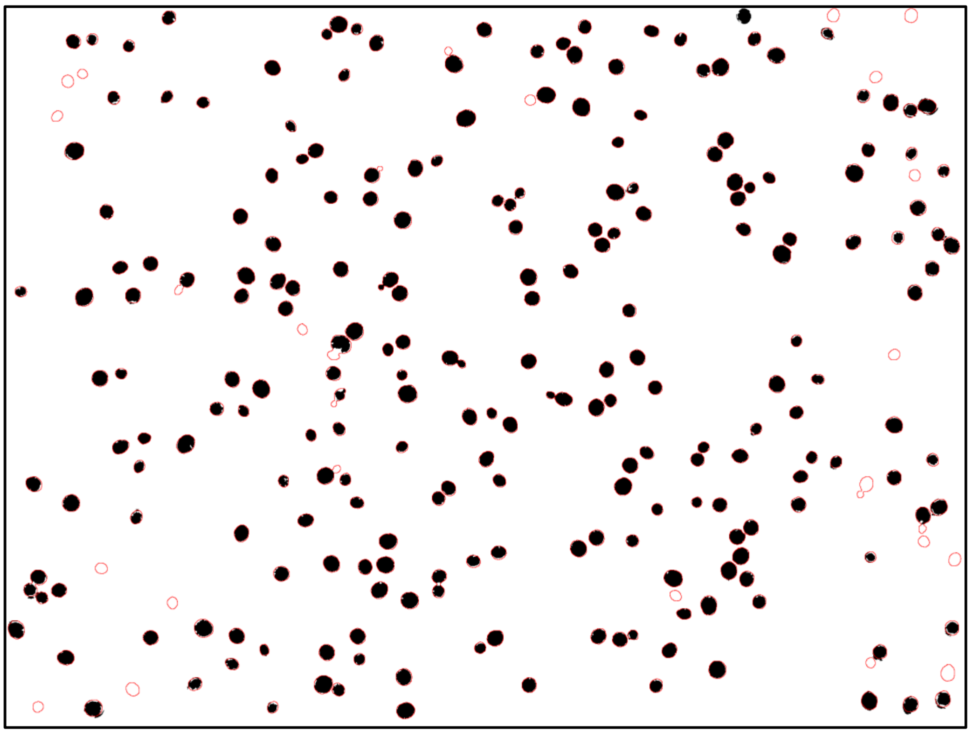


Figure J.2. Resulting mask from the sum of masks from experiments *M_031_1* and *M_031_2*, with mask radius and weight equal to 3 pixels and 0.9, respectively, and experiments *M_031_3* and *M_030_4*, with mask radius and weight equal to 2 pixels and 0.9, respectively. All experiments were performed with *Image 2*.

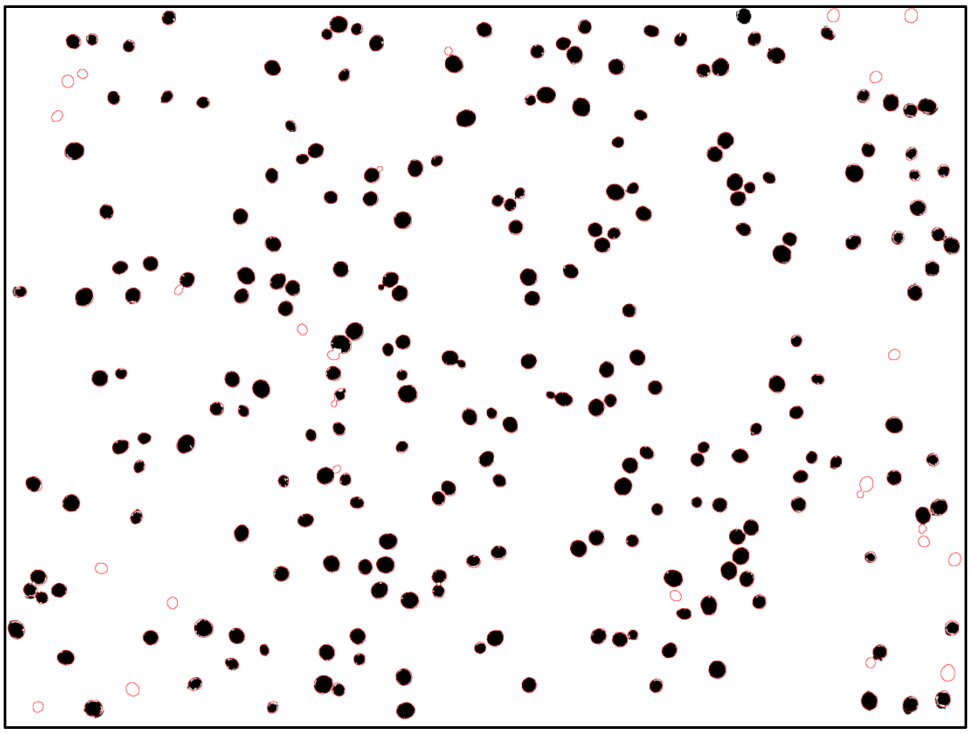


Figure J.3. Resulting mask from the sum of the masks in Figures J.1 and J.2.

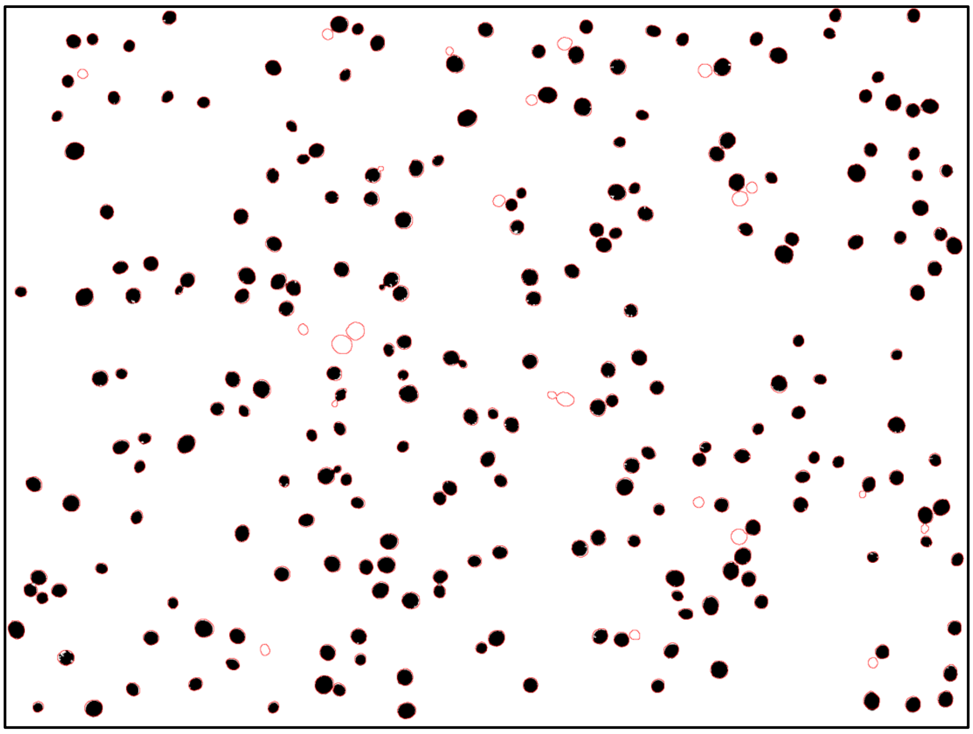


Figure J.4. Resulting mask from the sum of masks from experiments *M_029_4*, *M_030_2*, *M_031_1* and *M_031_3*, with mask radius and weight equal to 2 pixels and 0.9, respectively. All experiences were performed with *Image 3*.

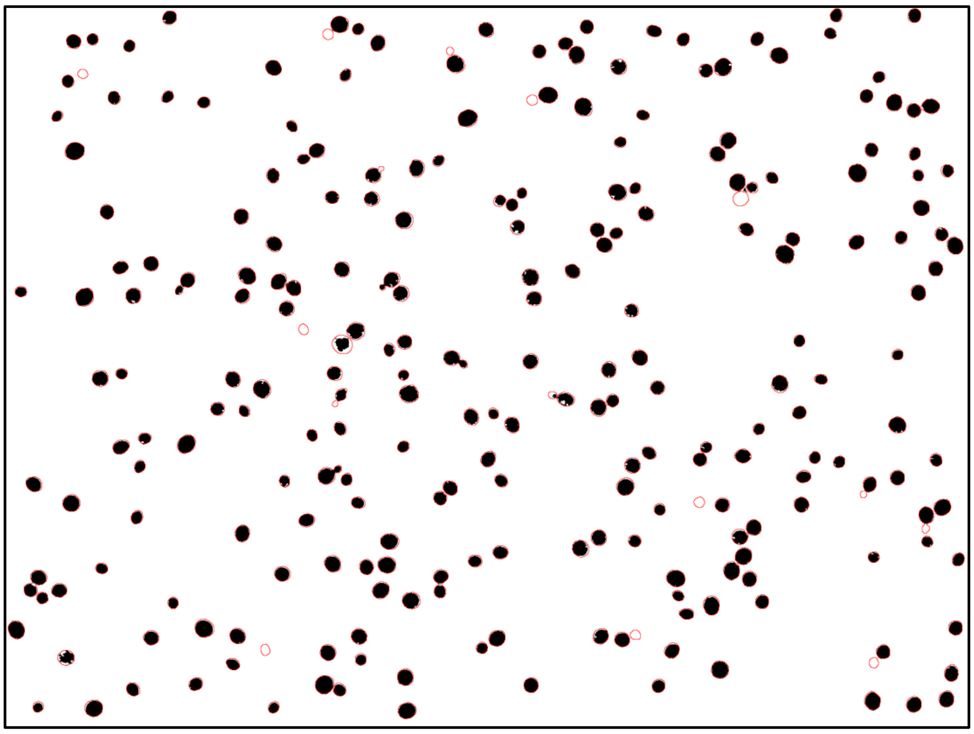


Figure J.5. Resulting mask from the sum of masks from experiments *M_031_1* and *M_031_2*, with mask radius and weight equal to 3 pixels and 0.9, respectively, and experiments *M_031_3* and *M_030_4*, with mask radius and weight equal to 2 pixels and 0.9, respectively. All experiments were performed with *Image 3*.

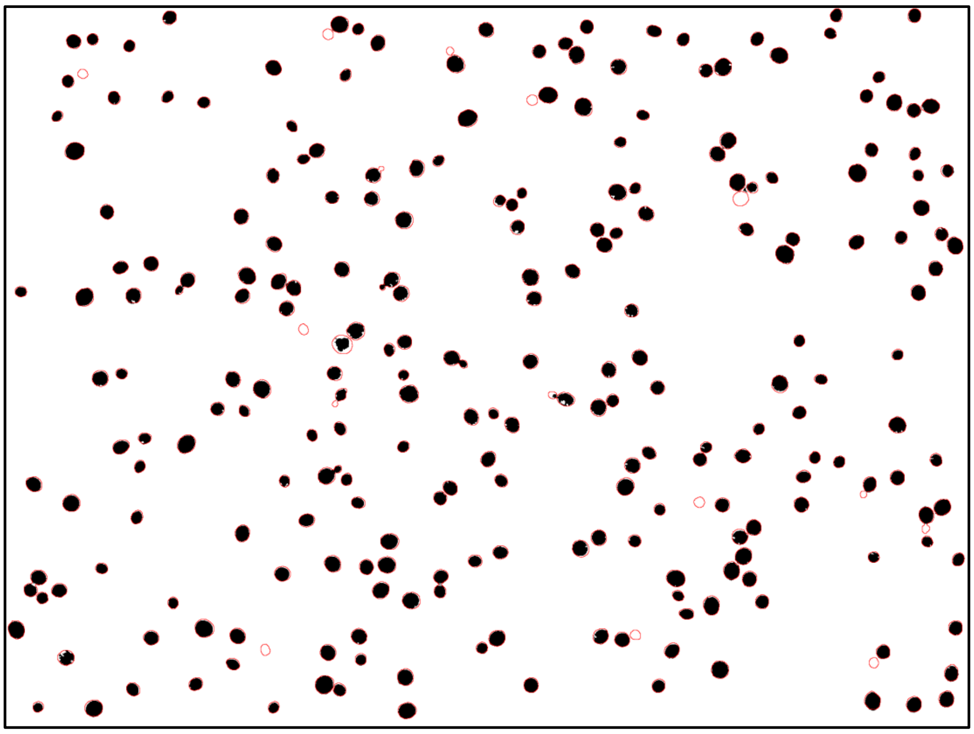


Figure J.6. Resulting mask from the sum of the masks in Figures J.4 and J.5.

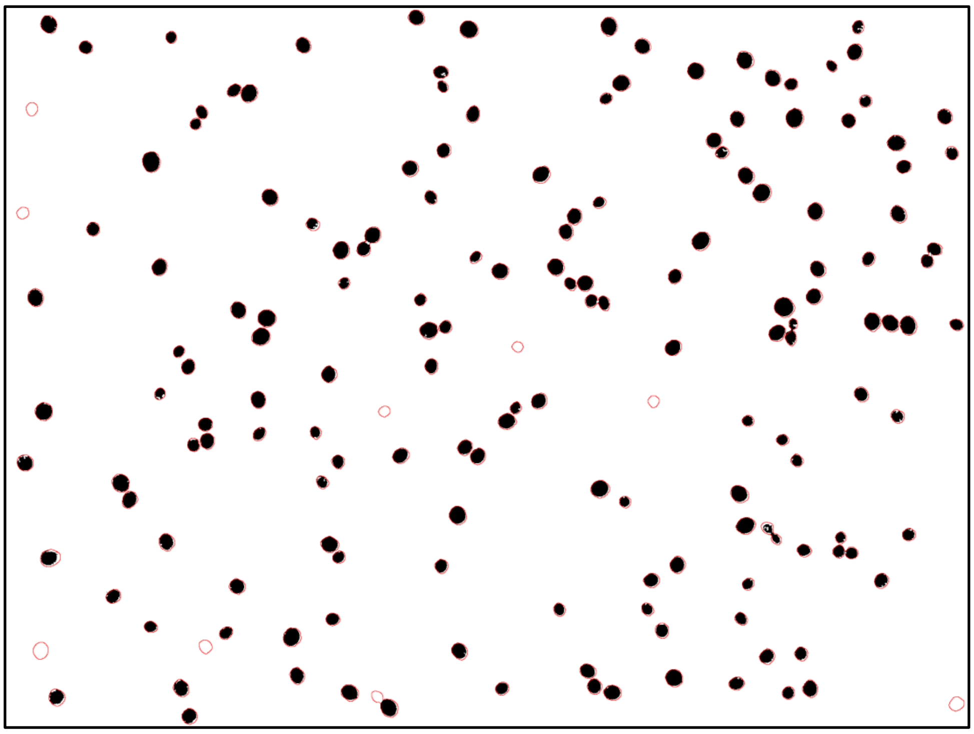


Figure J.7. Resulting mask from the sum of masks from experiments *M_033_1*, *M_033_3*, *M_033_4* and *M_031_2*, with mask radius and weight equal to 2 pixels and 0.9, respectively. All experiments were performed with *Image 4*.

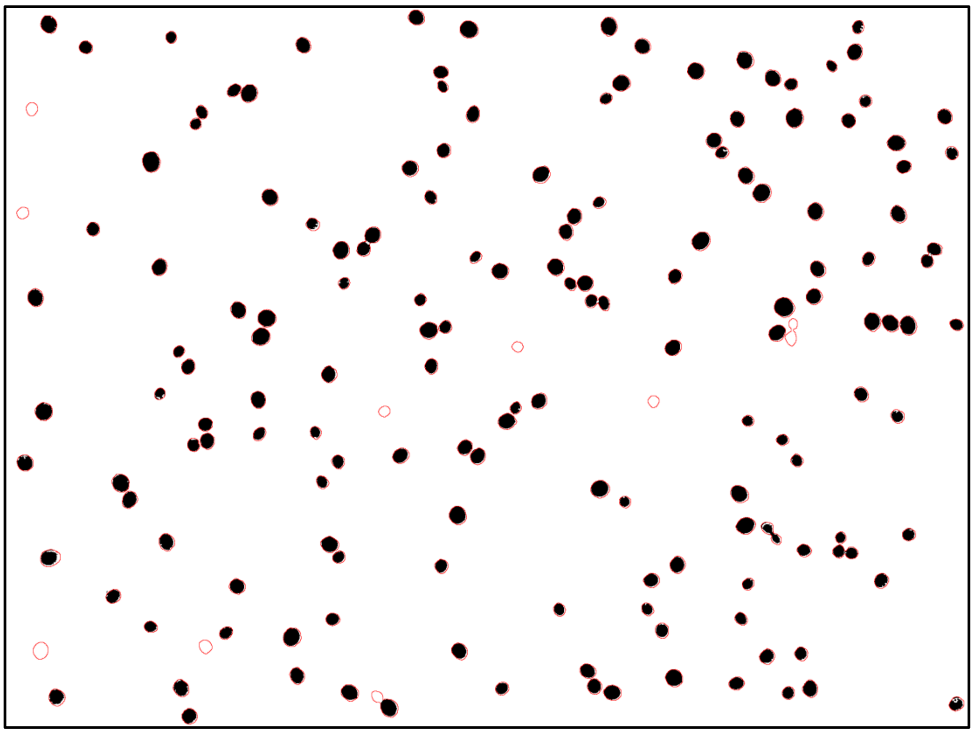


Figure J.8. Resulting mask from the sum of masks from experiments *M_031_1* and *M_031_2*, with mask radius and weight equal to 3 pixels and 0.9, respectively, and experiments *M_031_3* and *M_030_4*, with mask radius and weight equal to 2 pixels and 0.9, respectively. All experiments were performed with *Image 4*.

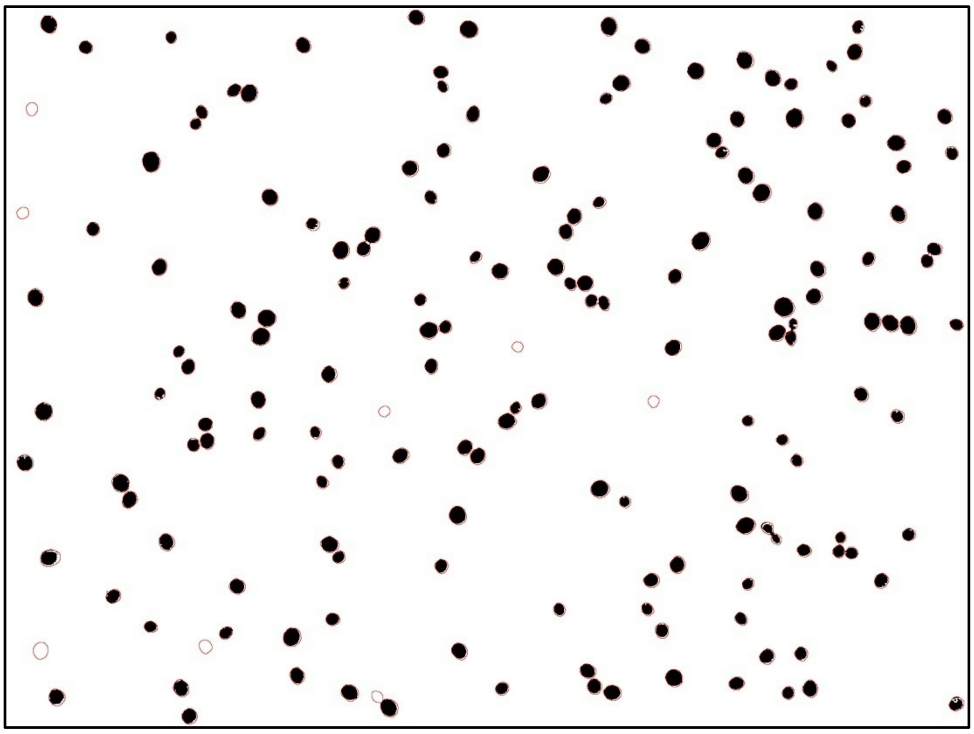


Figure J.9. Resulting mask from the sum of the masks in Figures J.7 and J.8.

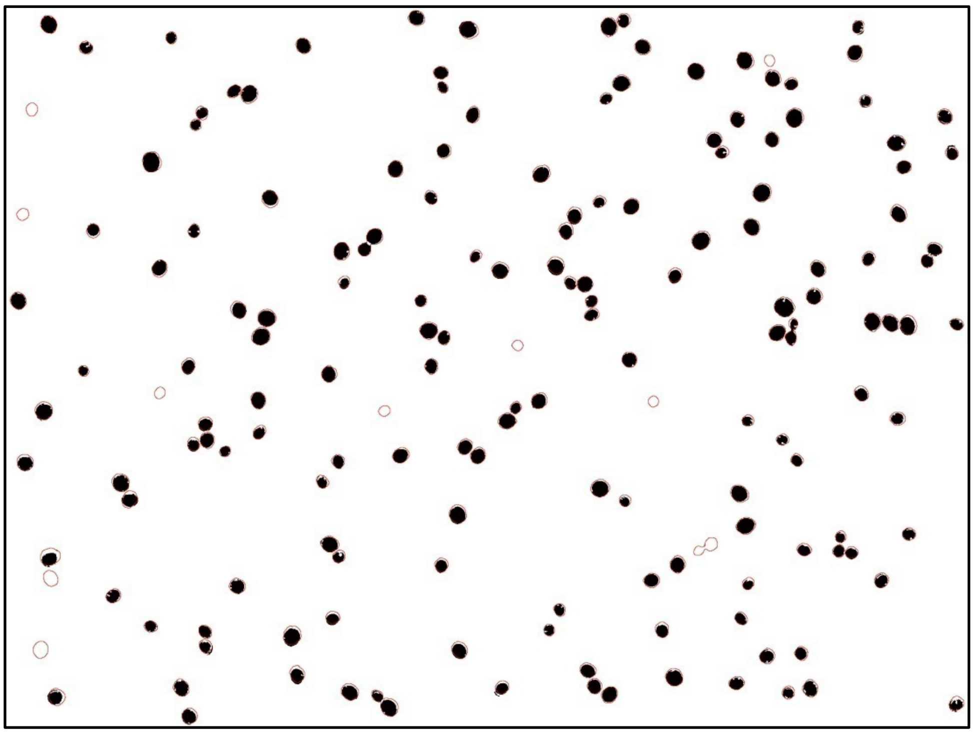


Figure J.10. Resulting mask from the sum of masks from experiments *M_033_1*, *M_032_2*, *M_032_3* and *M_031_4*, with mask radius and weight equal to 2 pixels and 0.9, respectively. All experiments were performed with *Image 5*.

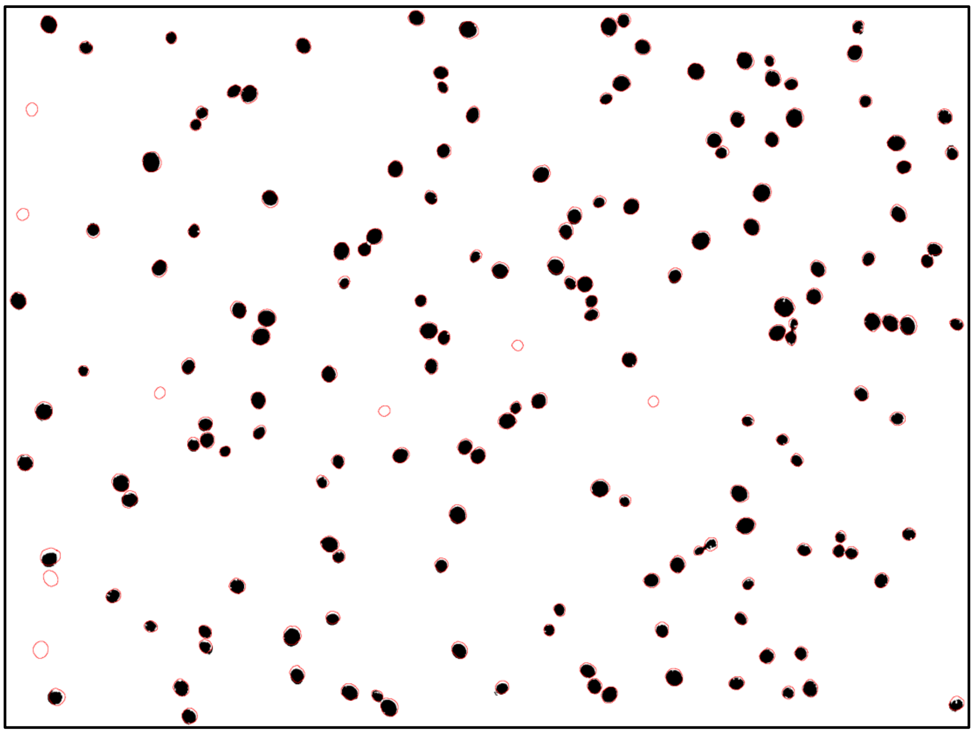


Figure J.11. Resulting mask from the sum of masks from experiments *M_031_1* and *M_031_2*, with mask radius and weight equal to 3 pixels and 0.9, respectively, and experiments *M_031_3* and *M_030_4*, with mask radius and weight equal to 2 pixels and 0.9, respectively. All experiments were performed with *Image 5*.

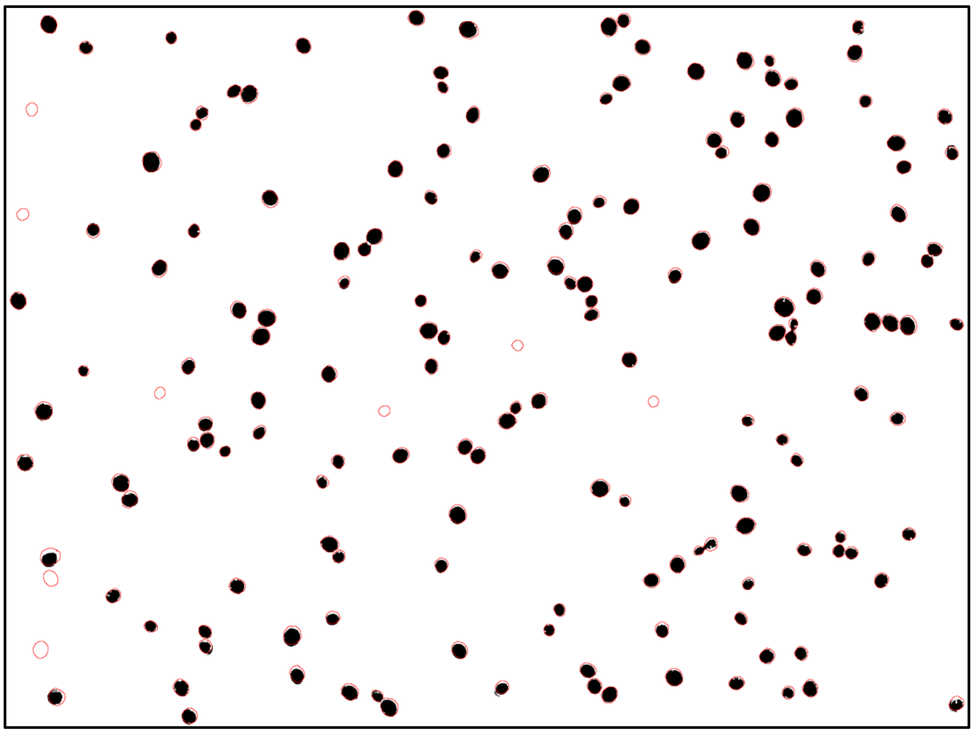


Figure J.12. Resulting mask from the sum of the masks in Figures J.10 and J.11.

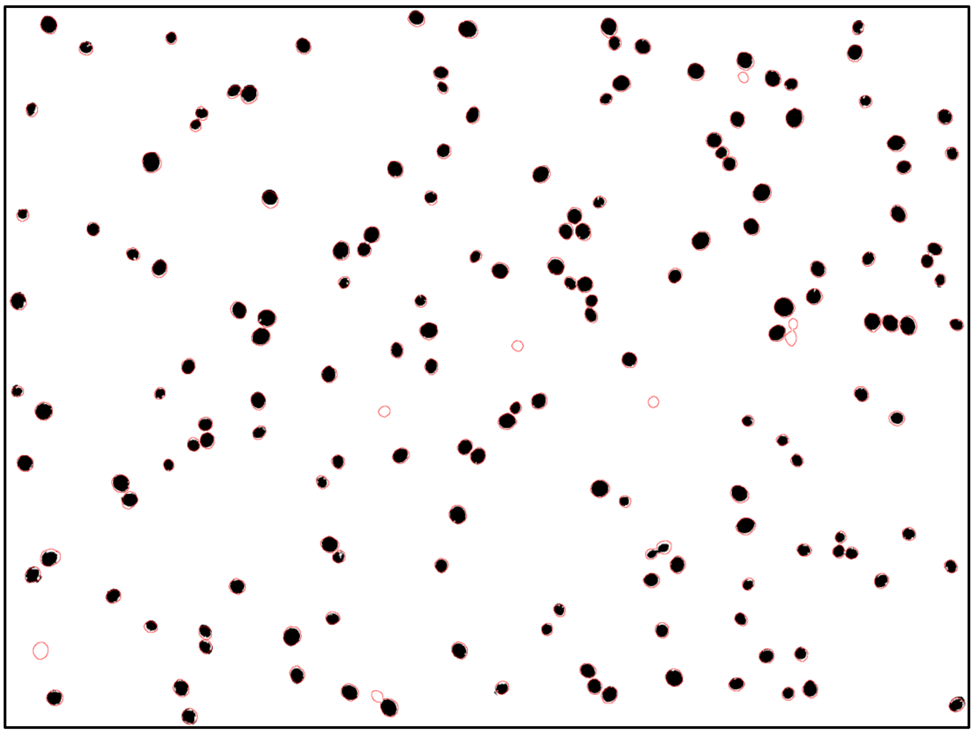


Figure J.13. Resulting mask from the sum of masks from experiments *M_032*_1, *M_031*_2 and *M_032*_3, with mask radius and weight equal to 2 pixels and 0.9, respectively, and experiment *M_030*_4, with mask radius and weight equal to 4 pixels and 0.9, respectively. All experiments were performed with *Image 6*.

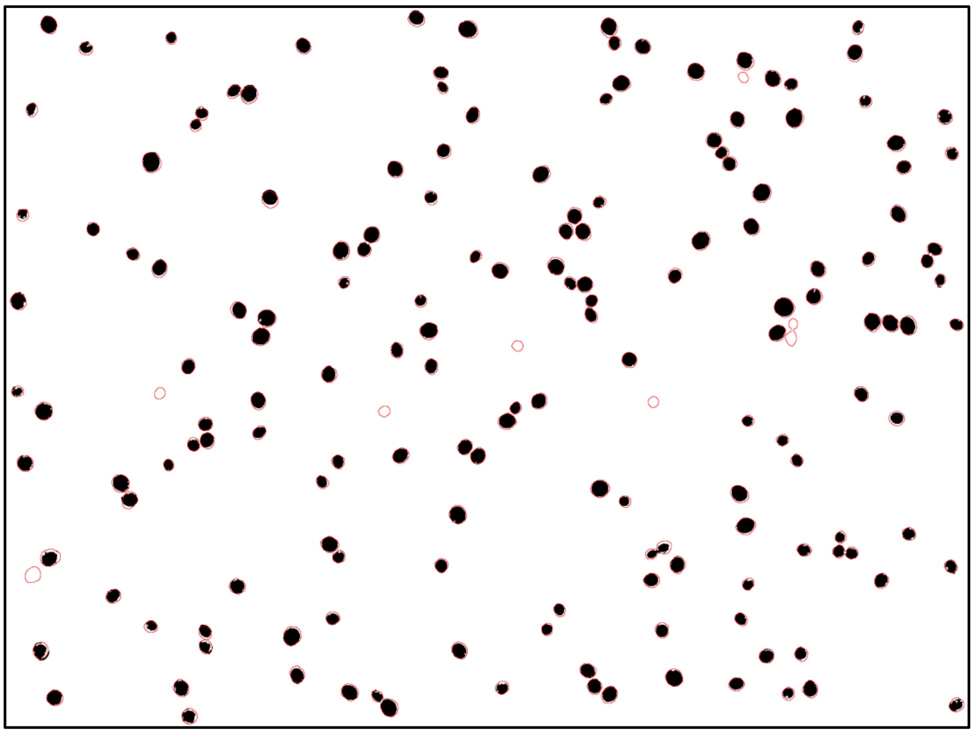


Figure J.14. Resulting mask from the sum of masks from experiments *M_031_1* and *M_031_2*, with mask radius and weight equal to 3 pixels and 0.9, respectively, and experiments *M_031_3* and *M_030_4*, with mask radius and weight equal to 2 pixels and 0.9, respectively. All experiments were performed with *Image 6*.

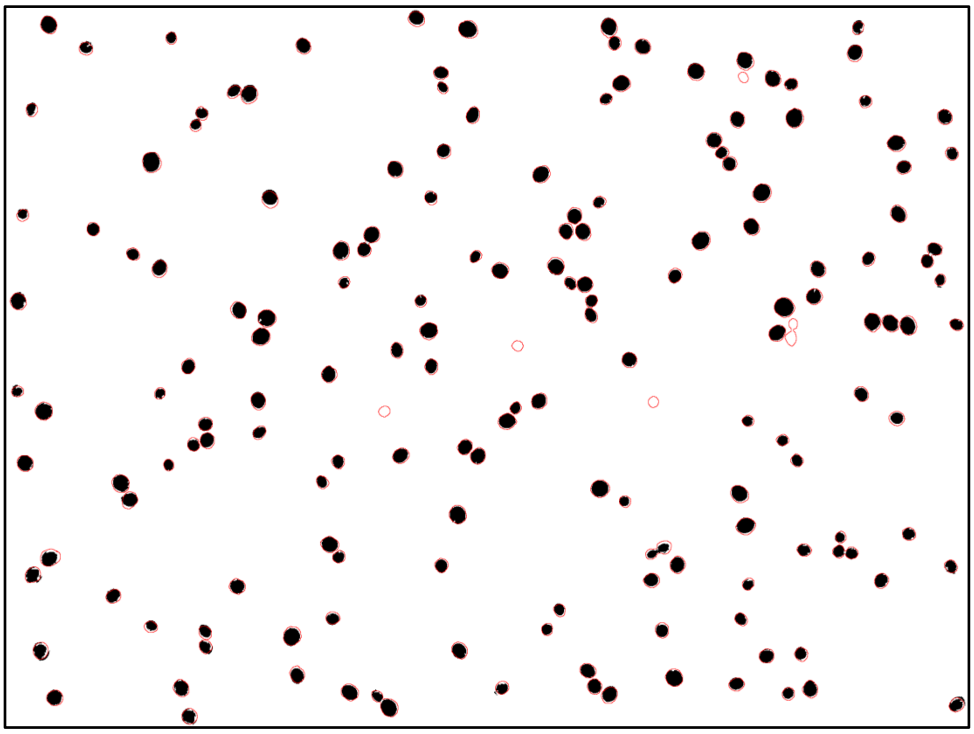


Figure J.15. Resulting mask from the sum of the masks in Figures J.13 and J.14.

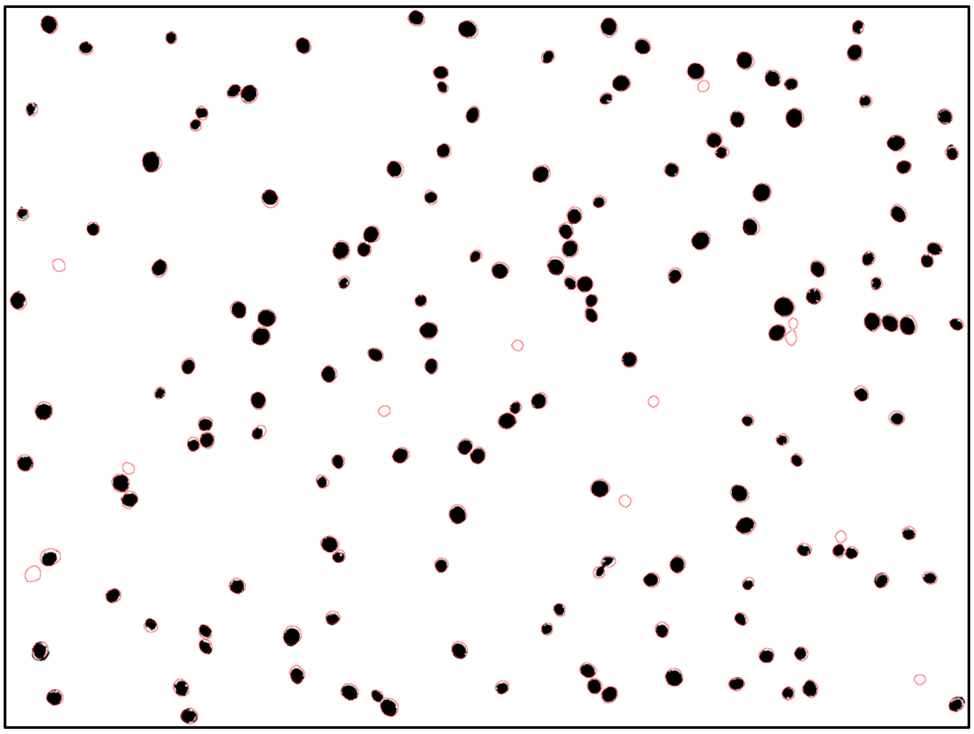


Figure J.16. Resulting mask from the sum of masks from experiments *M_032_1*, *M_031_2* and *M_032_3*, with mask radius and weight equal to 2 pixels and 0.9, respectively, and experiment *M_030_4*, with mask radius and weight equal to 4 pixels and 0.9, respectively. All experiments were performed with *Image 7*.

.

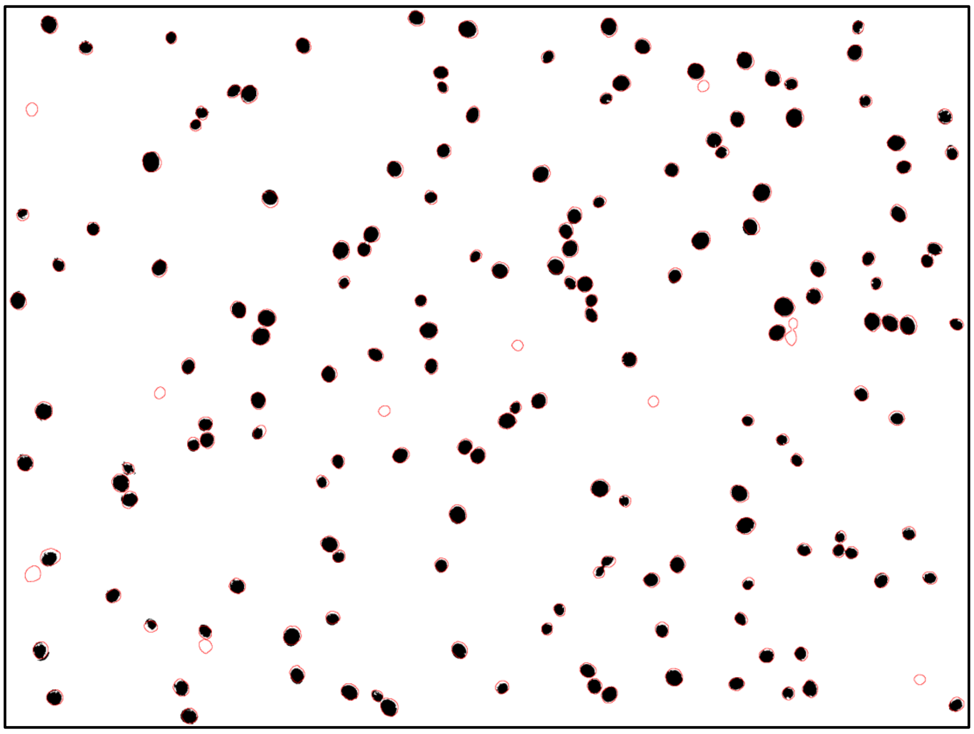


Figure J.17. Resulting mask from the sum of masks from experiments *M_031_1* and *M_031_2*, with mask radius and weight equal to 3 pixels and 0.9, respectively, and experiments *M_031_3* and *M_030_4*, with mask radius and weight equal to 2 pixels and 0.9, respectively. All experiments were performed with *Image 7*.

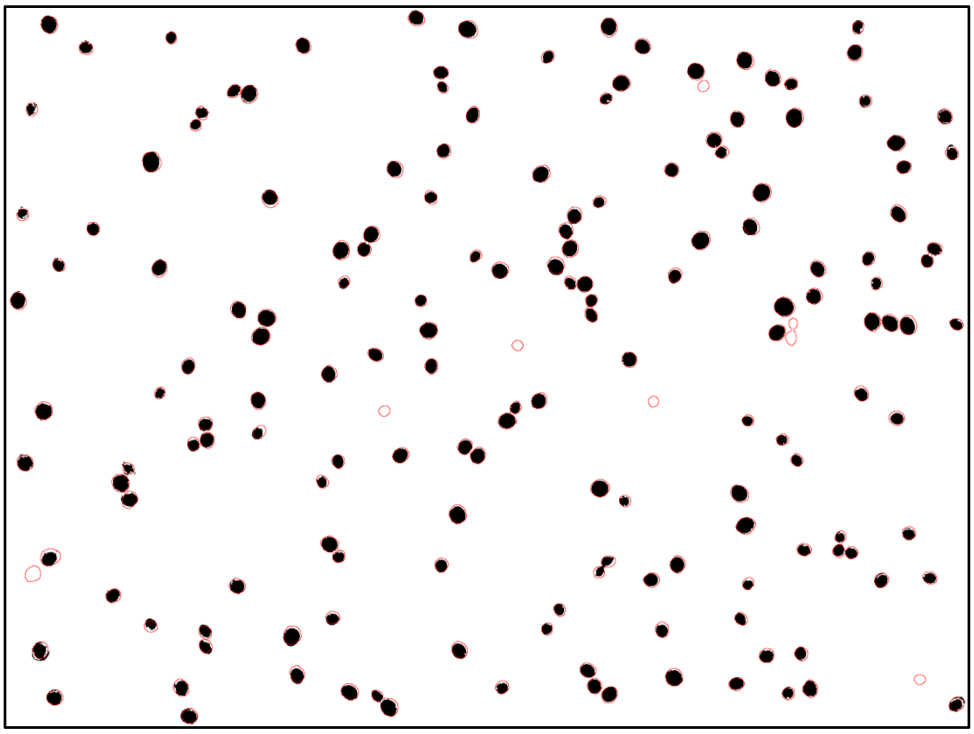


Figure J.18. Resulting mask from the sum of the masks in Figures J.16 and J.17.