Technical University of Delft
Faculty of Electrical Engineering, Mathematics and Computer Sciences
Delft Institute of Applied Mathematics

Analysis of Microscopic Images: A Morphological Approach

(Nederlandse titel: Analyse van Microscopische Beelden: Een Morfologische Aanpak)

Thesis submitted to the
Delft Institute of Applied Mathematics
as part of the acquisition of

the degree of

BACHELOR OF SCIENCE
in
APPLIED MATHEMATICS

by

MYRTE VAN BELKOM

Delft, Netherlands
June 2018

BSc thesis Applied Mathematics

"Analysis of Microscopic Images: A Morphological Approach'

(Nederlandse titel: "Analyse van Microscopische Beelden: Een Morfologische Aanpak"

MYRTE VAN BELKOM

Technical University of Delft

Supervisor

Dr. N. V . Budko

Other committee members

Drs. E. van Elderen	Dr. Ir. D. den Ouden-van der Horst
Mathematical Physics	Numerical Analysis

June, 2018 Delft

Analysis of Microscopic Images:
A Morphological Approach

Myrte van Belkom,
Technical University of Delft

June 27, 2018

Supervisor
Dr. N.V. Budko, Numerical Analysis

Committee members
Dr. Ir. D. den Ouden-van der Horst, Numerical Analysis
Drs. E. M. van Elderen, Mathematical Physics

Abstract

In this thesis we apply the numerical method of Morphological Geometric Active Contours as proposed by Alvarez, Baumela and Marquez-Neila [3] to microscopic images of plant cells. The goal is to find all plant cells in the images, and then to find their cell walls. This is done in order to calculate certain properties of these cells automatically, such as area, perimeter and ellipticity. As the name suggests, mathematical morphology plays a big part in morphological GAC. This thesis describes the theory behind morphological GAC, mathematical morphology and the implementation in Python of the actual algorithm. We made two adaptations compared to the original model. These are a change in the conditions of the balloon force, and an extra step at the end of each iteration. The change in the balloon force was made to keep edges stronger. The extra step was added to improve smoothness of the contours, since the smoothing parameter had no effect.

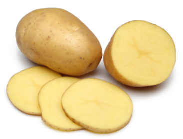
In dit verslag passen we de numerieke methode van morfologische Geometrische Actieve Contours toe, zoals beschreven door Alvarez, Baumela en Marquez-Neila [3], op microscopische afbeeldingen van plantencellen. Het doel is om alle plantencellen te lokaliseren in de afbeeldingen en vervolgens om de celwanden te vinden. Dit doen we om bepaalde eigenschappen van de cellen te achterhalen, zoals oppervlak, omtrek en maximale diameter. Zoals de naam impliceert speelt morfologie een belangrijke rol bij morfologische GAC. Dit verslag beschrijft de theorie achter morfologische GAC, morfologie en de implementatie van het algoritme in Python. We hebben twee aanpassingen gemaakt aan het bestaande algoritme. Dit zijn een extra stap aan het eind van elke iteratie en een verandering van de ballonkracht stap. De ballonkracht is aangepast om randen sterker te houden. De extra stap is toegevoegd om de gladheid van de contours te verbeteren, aangezien de gladheidsparameter verhogen geen invloed had.

Contents

1 Introduction

The company HZPC is a global market leader in not only potato breeding, but also seed potato trade and product development[1]. One of their focuses is quality control and breeding research. They develop breeds of potato to match a certain local environment as best as possible. To see how a certain type of potato is growing, the tuber is sliced very thinly. Then the starch is rinsed out carefully, and these extremely thin slices are placed under a microscope. This way, only one layer of cells should be visible and can be studied extensively. Unfortunately, the cells from different layers may overlap in some places. Decisions have to be made with regard to the goal of the problem. Should every part of a cell excluding overlap get its own contour, or do we want a contour to include all overlapping parts?

These microscopic images have been provided to us to research how the cells can be detected. No information has been provided about the conditions in which the tuber has been growing in order to remain unbiased. The following properties are considered: number of cells per image, area and perimeter of cells, ellipticity, isoparametric ratio and quotient, maximum diameter and its direction.



An average picture contains 2000 to 4000 cells. Since this is too large of a quantity to count by hand, this must be processed by a computer. The goal is to automate retrieving all the previously mentioned parameters of the cells in one microscopic image. The first step is to find out where the cells are located, next to find their cell walls, and lastly to compute all previously mentioned properties.

HZPC recently acquired a new microscope which can produce images with a higher resolution. These images have an average size of 50 megabytes instead of the previous 1.5 megabytes of the other images. The bigger images still contain about 2000 to 3000 cells on average. These require some new techniques to process, but should have less noise and more details in the cells.

In Section 2 we introduce geometric active contours, and after some theory about mathematical morphology in Section 3, we show how this can be rewritten into morphological GAC. Then we give some details about the implementation of this algorithm and show results in Section 5. And lastly we describe the adaptations that were made in Section 6 and our conclusions in Section 7.

2 Background

Humans find it very easy to not only detect objects in an image, but also to identify these objects immediately. For a computer, this task is exceptionally difficult. One method in image analysis that can be used to detect object outlines, are active contours or snakes. These are two-dimensional contours that are generated and moved by a computer algorithm. It is driven by minimizing the energy functional:

$$E_{snake} = E_{internal} + E_{external}$$

Here, $E_{internal}$ maintains the smoothness and continuity of the contour. $E_{external}$ is a function of the image, usually to highlight the interesting areas. This is often a Gaussian filter to smooth out noise, a gradient filter to highlight edges, or a combination of the two.

The main focus of the paper by Alvarez, Bauma and Marquez-Neila [3] lies on the derivation of a morphological version of the geometric active contour framework using a level set implementation. We will show how the GAC formulation is rewritten with morphological operations in the Subsections 4.1 to 4.3. But we will show the GAC framework using the level set method to begin with.

The evolution of a contour using GAC with an added balloon force is described by:

$$\mathcal{C}_t = (g(I)\mathcal{K} + g(I)\nu - \nabla g(I) \cdot \mathcal{N})\mathcal{N} \tag{1}$$

Here, C is a parameterisation of the contour over time:

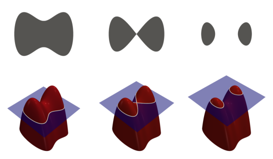
$$\mathcal{C} : \mathbb{R}^+ \times [0,1] \to \mathbb{R}^2 : (t,p) \to \mathcal{C}(t,p).$$

And the evolution of this contour is given by $\mathcal{C}_t$. The $g(I)$ is a certain function g applied to the image I and $\mathcal{K}$ is the Euclidean curvature of the contour. $\mathcal{N}$ is the normal to the contour and ν is the balloon force parameter, which will be explained in Section 2.

In general $\mathcal{C}_t$ can be written as $\mathcal{C}_t = L(\mathcal{C})$ with a differential operator L. Every L can be written as a product of the normal to the contour and a scalar field, $\mathcal{F}$, which might depend on the contour and describes the speed of the curve evolution. We can see that this is also true in our case with: $\mathcal{F} = g(I)\mathcal{K} + g(I)\nu - \nabla g(I) \cdot \mathcal{N}$.

We can rewrite this equation using the Osher-Sethian level set method [2]. Define a function $u : \mathbb{R}^+ \times \mathbb{R}^2 \to \mathbb{R}$ which is 1 inside the contour and 0 outside, so that it embeds $\mathcal{C}$ as its $\frac{1}{2}$ level set. First we look at a simple example of $\mathcal{F}$ equal to one or minus one. Then L is just the normal to the contour (or the negative of this). This means that the contour moves with constant speed in the direction of the normal to the contour. Using the level set method, this results in the following PDE for the evolution of u:

$$\frac{\partial u}{\partial t} = \pm|\nabla u|. \tag{2}$$



Another example is when $\mathcal{F}$ is the Euclidean curvature of the contour. Then L is called the curvature flow, which evolves any closed curves that do not intersect into convex curves. Hence this can be seen as a type of smoothing operation. Application of the level set method to this equation leads to:

$$\frac{\partial u}{\partial t} = \operatorname{div}\left(\frac{\nabla u}{|\nabla u|}\right)|\nabla u| \tag{3}$$

Equation (3) is called the mean curvature motion in general. With these results we can begin to rewrite equation (1). The equation consists of three distinct terms which will be discussed separately in the following Subsections.

Attraction force

The last term $(\nabla g(I) \cdot \mathcal{N})\mathcal{N}$ in equation (1) represents the attraction force. This is an external force which pulls the contour in the direction of interesting parts of the image. Depending on the choice of $g(I)$ we are able to select which parts of an image we are interested in. It is known that when C_t is equal to the normal, the level set method yields equation (2). The difference in this equation, is that the normal is not multiplied by 1 or -1, but with $\nabla g(I) \cdot \mathcal{N}$. So the evolution no longer moves with constant speed, but with a speed depending on the gradient of $g(I)$. $\nabla g(I) \cdot \mathcal{N}$ may no longer be a constant, but it is still a scalar-valued function. Hence this term can be rewritten as: $\frac{\partial u}{\partial t} = \nabla g(I) \cdot \nabla u$.

Balloon force

The second term $g(I)\nu\mathcal{N}$ represents the balloon force [7]. Similarly to the attraction force, we use equation (2) to rewrite it. $\frac{\partial u}{\partial t} = g(I)|\nabla u|\nu$. This extra balloon force is added because sometimes the attraction force itself is not strong enough to pull the contour in the right direction. Since a curve can only evolve inward or outward, the balloon force has a parameter ν which determines direction and strength of the balloon force.

Smoothing force

The first term $g(I)\mathcal{K}\mathcal{N}$ represents the smoothing force. The smoothing force is an internal force which maintains the smoothness and continuity of the contour. Since we just showed how the $g(I)\mathcal{N}$ could be rewritten into $g(I)|\nabla u|$ in Section 2, all that is left is the Euclidean curvature. We saw in equation (3) that the Euclidean curvature multiplied with the normal became the PDE (3) after application of the level set method, we can use this result. Therefore the first term $g(I)\ \mathcal{K}\mathcal{N}$ becomes $\frac{\partial u}{\partial t} = g(I)|\nabla u| \operatorname{div}\left(\frac{\nabla u}{|\nabla u|}\right)$.

When we combine the last three results, we get an expression for the entire equation:

$$\frac{\partial u}{\partial t} = g(I)|\nabla u| \operatorname{div}\left(\frac{\nabla u}{|\nabla u|}\right) + g(I)|\nabla u|\nu + \nabla g(I) \cdot \nabla u. \tag{4}$$

A very useful property of the level-set implementation of GAC is that the contours can split and merge when necessary. Now that we have a level set formulation of the GAC, all that is left is to rewrite it into a morphological GAC. But first, some theory about mathematical morphology.

3 Mathematical morphology

Mathematical morphology is an image analysis and processing technique invented in 1964 [9]. It was originally developed for binary images, but has since been extended to grayscale images and was even generalised to complete lattices. All theory will be explained for binary images first, and then we will explain the difference with grayscale morphology.

A binary image is a digital image which consists of only two different values, usually ones and zeros. Because of this, a binary image can be represented as a set. This fact is employed in mathematical morphology. A few concepts from set theory are used in mathematical morphology: intersection ($\cap$), union ($\cup$) and complements (A^c). And also concepts from topology and geometry such as distance, size, convexity and connectivity. In the following Subsections, the basics of mathematical morphology will be explained.

To begin, some terms and notation are introduced that will be used in the following Sections. Since only binary digital images are considered, each pixel is given a value of 1 or 0. Where a 1 means that a pixel is active, part of the object, or in the foreground of an image. And conversely a 0 means that a pixel is inactive, part of the background, or not part of the object. Active pixels are displayed in white and inactive pixels in black.

3.1 Structuring element

The first ingredient needed for mathematical morphology, is a structuring element. This is used as a type of probe to study a specific image and is nearly always smaller than the image or object being studied. This structuring element is adapted to fit the geometrical properties of an image. It is defined in terms of a center pixel or anchor point and which pixels are considered to be in its neighbourhood. Two of the most widely used structuring elements are depicted in Figure 1.

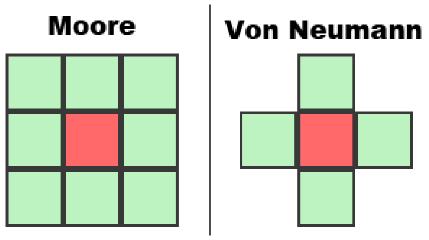


Figure 1: Two neighbourhoods (green) [12].

The red pixels are the anchor points of these structuring elements. The Von Neumann neighbourhood is defined as all pixels with distance 1 to the red pixel using the Manhattan Metric: $(|x_1 - x_2| + |y_1 - y_2|)$. And the Moore neighbourhood is defined as all pixels with distance 1 using the Chebyshev metric: $\max(|x_1 - x_2|, |y_1 - y_2|)$. The x_1 and x_2 represent the x-coordinates of two different pixels and y_1 and y_2 the y-coordinates. Since we measure this in entire pixels x and y can only take integer values, and the distances too. The Moore neighbourhood is often referred to as 8-connected (N_8) and the Von Neumann as 4-connected (N_4). These abbreviations will be used throughout the rest of this thesis.

A third widely used structuring element is a circle with radius r around a center pixel. In the binary case, a perfect circle is not possible, although approximations can be used. But theoretically this circular structuring element can be very useful as will be demonstrated in Section 4.

3.2 Dilation and erosion

Erosion of an object A with structuring element B in an image I is denoted with $\ominus$ and defined as:

$$A \ominus B = \{z \in I | B_z \in A\} = \bigcap_{b \in B} A_b \tag{5}$$

Where $A_b = \{a+b | a \in A\}$ or the translation of A by the vector b and B_z denotes the translation of B by z. In the case of B_z this can be pictured as all pixels in the neighbourhood of z when using B as structuring element. And $B_z \in A$ means that the entire translated B_z falls within A. So even when a single pixel of B_z does not lie within A, this means that $B_z \notin A$. A more intuitive explanation is one where a structuring element is seen as the neighbourhood of a pixel. If a pixel has any neighbours (including itself) that are *not* part of the object, it is removed. Erosion of an object always leaves a smaller or equal object, since it can only remove pixels and cannot add them. A morphological operation that can add pixels is dilation.

Dilation of an object A with a structuring element B is denoted by $A \oplus B$. It does not matter in which order these elements are, since dilation is commutative ($A \oplus B = B \oplus A$) and associative ($(A \oplus B) \oplus C = A \oplus (B \oplus C)$). The definition of binary dilation is as follows:

$$A \oplus B = \{z \in I | (B^*)_z \cap A = \varnothing\} = \bigcup_{b \in B} A_b$$

With $B^* = \{z \in I | -z \in B\}$ and A_b as defined above. If a structuring element is again considered as a neighbourhood, dilation can simply be described as follows. Any pixel that has a neighbour within the object, is added to the object. It can easily be seen that erosion can only change the value of active pixels and dilation only of inactive pixels. This is because the anchor pixels themselves are also part of the structuring element. In the case of dilation the operator checks whether any part of the structuring element is active, and this is already true for the pixel itself and it will remain active. Simply put: erosion can only deactivate pixels and dilation can only activate them. Dilation is not the inverse operation to erosion, but they are each others' dual operation :$A \oplus B = (A^c \ominus B^*)^c$. This is easily shown with an example of a five by five grid with a single pixel in the middle in Figure 2. If it is dilated first, then all nine middle pixels will become active. If an erosion is applied next, then precisely those pixels that were just added will vanish, and only the original pixel is left. But look at what happens if these operations are reversed. When the erosion applied to a single pixel, it removes the pixel. And subsequent dilation of the image will lead to the same empty image. If the operations were each others' inverse, the original picture should have remained in the last window. What is true is that dilation of the foreground is equal to erosion of the background of an image and vice versa. This follows directly from the definitions of dilation and erosion. Adding all neighbours of the foreground pixels (making them active) is the same as removing all pixels from the background which have a neighbour in the foreground. Because in binary images, removing a pixel from the background is the same as adding it to the foreground.

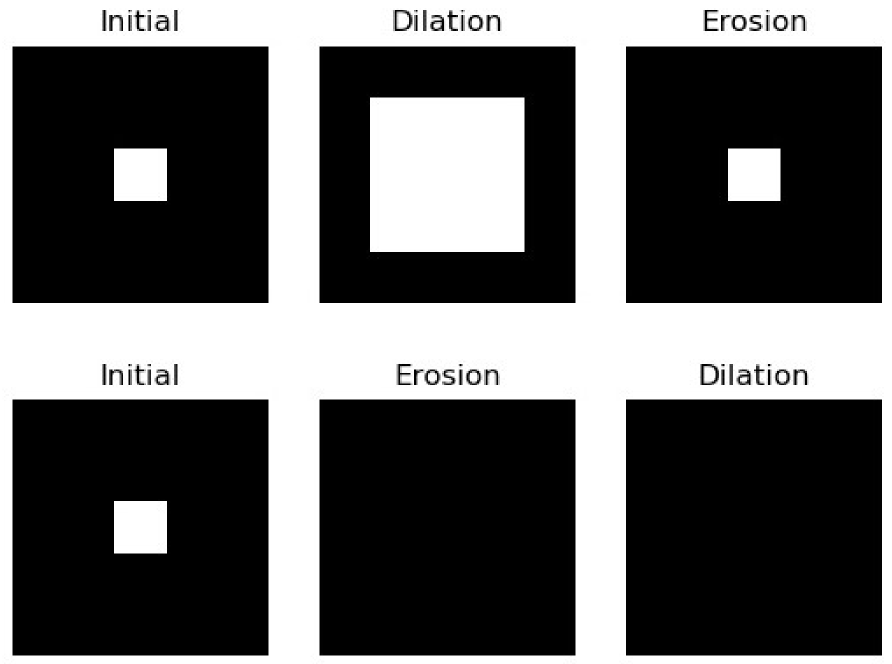


Figure 2: Example of morphological operations

3.3 Opening and closing

Opening and closing are the next most important morphological operations. They both consist of successive applications of dilation and erosion. Opening and closing are defined as follows, where an opening is denoted by $\circ$ and a closing by $\bullet$.

$$A \circ B = (A \ominus B) \oplus B, \tag{6}$$

$$A \bullet B = (A \oplus B) \ominus B. \tag{7}$$

Note that the structuring element is the same for the erosion as the dilation in a single opening or closing. From these definitions, it is quickly visible what the operations do. As the name implies, an opening can *open up* an object, remove speckle noise, remove thin lines or small protrusions and disconnect objects. Because the erosion is applied first, all single pixels are removed. When the image is dilated, the speckle noise is gone, and not affected by the dilation. A closing does exactly what its name suggests: It *closes* objects and it is the dual operation to opening. Closing of an image can remove small holes from objects, connect objects that are close together and also connect lines that are broken up because of noise. When an object has a hole of a single pixel in it, dilation will add this to the object. When erosion is applied, this once inactive pixel, will now remain active because all its neighbours are. Opening and closing also have a few things in common. One very important example is that they both smooth the contour of an object. This fact will be used later in this paper in Section 6. Both opening and closing can be applied more than once. Then an opening consists of multiple erosions followed by the same amount of dilations, and not n times one erosion followed by one dilation. Then larger noise than speckle noise of single pixels can be removed by opening, and also larger holes within objects can be filled by closing.

3.4 Grayscale morphology

In grayscale morphology, the main difference is that images are seen as functions mapping a grid in $\mathbb{R}^2$ to the compactification of the real numbers $\overline{\mathbb{R}}$. The image pixels can now have values of not only 1 and 0 but also every value in between. The grayscale definitions of dilation and erosion when using a flat structuring element are given below.

$$(a \oplus b)(x) = \sup_{z \in B^*} a(x - z),$$

$$(a \ominus b)(x) = \inf_{z \in B} a(x - z).$$

Where a denotes the image and b the structuring element. A structuring element does not need to be flat. Weights can be added to certain pixels, but we will not discuss this. One example of a flat structuring element is a circle with radius h, the third structuring element mentioned in Section 3.1. The theory covered for binary morphology holds for grayscale morphology. So opening and closing are still defined as the application of successive dilations and erosions and do not need to be defined again.

One interesting result in grayscale morphology, is that every morphological operation T can be written as one of the following two sup-inf formulations:

$$(T_h a)(x) = \sup_{B \in \mathcal{B}} \inf_{y \in x + hB} a(y) \tag{8}$$

$$(T_h a)(x) = \inf_{B \in \mathcal{B}} \sup_{y \in x + hB} a(y) \tag{9}$$

Where $\mathcal{B}$ is not one structuring element, but a set of structuring elements and h is a scalar. Dilation and erosion are a simple example, where the set of structuring elements consists of only one.

4 Morphological GAC

With the theory covered in Sections 2 and 3 we can describe the change from equation (4),

$$\frac{\partial u}{\partial t} = g(I)|\nabla u| \operatorname{div}\left(\frac{\nabla u}{|\nabla u|}\right) + g(I)|\nabla u|\nu + \nabla g(I)\nabla u,$$

to a morphological variant. One of the ideas employed to achieve this, is to find the relationship between morphological operations and PDEs [6]. If we denote a dilation of u with the circle of radius h around its center by $D_h u$. And an erosion of u with the same structuring element by $E_h u$. It holds that:

$$\lim_{h \to 0^+} \frac{D_h u - u}{h} = |\nabla u|, \tag{10}$$

$$\lim_{h \to 0^+} \frac{E_h u - u}{h} = -|\nabla u|. \tag{11}$$

So if we keep dilating the level set u by this structuring element with very small radius, it approximates the steady state solution of the PDE $\frac{\partial u}{\partial t} = |\nabla u|$. And the same goes for erosion, except that it approximates $\frac{\partial u}{\partial t} = -|\nabla u|$. As we can see, this term occurs multiple times in the PDE we want to rewrite in terms of morphological operations. This is a very useful result, but there is still one factor we need to rewrite, $\operatorname{div}\left(\frac{\nabla u}{|\nabla u|}\right)$, also know as the curvature operator. This is precisely what Alvarez, Baumela and Marquez-Neila introduced in their paper [3]: The curvature morphological operator. This is an operator which mimics the behaviour of the PDE:

$$\frac{\partial u}{\partial t} = \operatorname{div}\left(\frac{\nabla u}{|\nabla u|}\right). \tag{12}$$

They did not only derive this for the two-dimensional case, but also for the n-dimensional case and they apply it to two- and three-dimensional images.

Catté, Dibos and Koepfler [5] developed a discrete scheme for mean curvature motion also using morphology and proved that applying the mean operator $F_{\sqrt{h}}$ successively to u with h small is equivalent to the solution of (12).

Alvarez, Baumela and Marquez-Neila used this result and derived its morphological equivalent which consists of the composition of supremum and infimum operations denoted by: $SI \circ IS$ using base $\mathcal{B} = \{[-1,1]_\theta \subset \mathbb{R}^2 : \theta \in [0,\pi)\}$. This gives us a morphological expression we can use to rewrite the PDE. In practice u is discretised, and so the morphological operators must be discretised too. This is achieved by using a discrete basis instead of $\mathcal{B}$, consisting of the following structuring elements: The discrete versions of dilation and erosion in

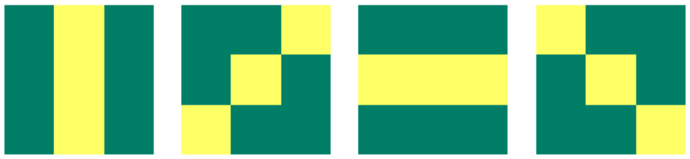


Figure 3: Structuring elements for discrete $SI \circ IS$ operator (yellow)

this PDE are achieved the same way. Instead of the circular base, the Moore neighbourhood is used as structuring element. Now that we have everything we need to rewrite the equation (4) into a morphological one, we will do so for each term. For easier reading, here is the equation one last time:

$$\frac{\partial u}{\partial t} = g(I)|\nabla u| \text{ div}\left(\frac{\nabla u}{|\nabla u|}\right) + g(I)|\nabla u|\nu + \nabla g(I) \cdot \nabla u.$$

4.1 Attraction force

The attraction force term $\nabla g(I)\nabla u$ can be replaced very simply, since this is almost discrete already. When the $\nabla g(I) \cdot \nabla u$ is positive, it adds to the level set. And when it is negative, it takes pixels out of the level set. All that is left is to discretise the gradients. We approximate this by a central difference and denote the result by ∇_d:

$$u^{n+1}(x) = \begin{cases} 1 & \text{if } \nabla_d u^n \cdot \nabla_d g(I)(x) > 0 \\ 0 & \text{if } \nabla_d u^n \cdot \nabla_d g(I)(x) < 0 \\ u^{n+1} & \text{if } \nabla_d u^n \cdot \nabla_d g(I)(x) = 0 \end{cases} \tag{13}$$

Here u^{n+1} is the level set in the next iteration of the morphological GAC algorithm. In each iteration, the result of the previous iteration is used. We also need to select a $g(I)$ for the algorithm. For us the edges of the cells are the regions of interest. The image itself would pose a relatively good $g(I)$, since its values are low in cell walls (dark or black) and high in the centres of the cell (bright or white). We decide to smooth the image with a Gaussian filter for reasons which will be explained in Section 5.2.

4.2 Balloon force

The balloon force term $g(I)\nu|\nabla u|$ can be rewritten using the results in equations (10) and (11). Binarization of the $g(I)\nu$ is handled similarly to the attraction force:

$$u^{n+1}(x) = \begin{cases} (Du^n)(x) & \text{if } g(I)(x) > \theta \text{ and } \nu > 0 \\ (Eu^n)(x) & \text{if } g(I)(x) > \theta \text{ and } \nu < 0 \\ u^n(x) & \text{otherwise} \end{cases} \tag{14}$$

The balloon parameter's strength no longer matters, since we take h to be very small. Only the sign of ν is of importance now, as is visible in the rewritten PDE.

4.3 Smoothing force

For the smoothing force, $g(I)$ could be rewritten in the same way, but Alvarez, Bauma and Marquez-Neila showed that this was unnecessary. Applying results from the first part of Section 4 his leaves us with only the following binarization:

$$u^{n+1}(x) = ((SI \circ IS)^{\mu} u^n)(x). \tag{15}$$

Here, the μ is a smoothness parameter: for larger values of μ, the contour becomes smoother. Note that μ should be an integer, since applying fractions of morphological operations does not exist.

After combining results from each term, the complete morphological GAC algorithm is given by:

$$u^{n+\frac{1}{3}}(x) = \begin{cases} (Du^n)(x) & \text{if } g(I)(x) > \theta \text{ and } \nu > 0 \\ (Eu^n)(x) & \text{if } g(I)(x) > \theta \text{ and } \nu < 0 \\ u^n(x) & \text{otherwise} \end{cases} \tag{16}$$

$$u^{n+\frac{2}{3}}(x) = \begin{cases} 1 & \text{if } \nabla_d u^{n+\frac{1}{3}} \cdot \nabla_d g(I)(x) > 0 \\ 0 & \text{if } \nabla_d u^{n+\frac{1}{3}} \cdot \nabla_d g(I)(x) < 0 \\ u^{n+\frac{1}{3}} & \text{if } \nabla_d u^{n+\frac{1}{3}} \cdot \nabla_d g(I)(x) = 0 \end{cases} \tag{17}$$

$$u^{n+1}(x) = \left((SI \circ IS)^{\mu} u^{n+\frac{2}{3}}\right)(x) \tag{18}$$

We denote intermediate results by $u^{n+\frac{2}{3}}$ and $u^{n+\frac{1}{3}}$, but these are not a one third iteration and two thirds iteration on their own. The iteration is only done when the smoothing force is applied (18) and this is denoted by u^{n+1}.

5 Implementation

Using the derived algorithm (16), we implemented this level set evolution in Python. The Python code can be found in Appendix A. The images used are grayscale images of plant cells, specifically the cross-sections of potato tubers. These are implemented as two-dimensional NumPy arrays filled with float values ranging from 0 to 1, and so are the level sets but with values of only 0 and 1. In every iteration, the different terms have to be applied instantaneously to all pixels of the image. We therefore copy the original level set to store all new calculated values. This is done three times in every iteration, once for each term. A single iteration is defined in one function. This function is then used in a different function which initializes the level set and starts the iterations. Also, new functions were written for dilation and erosion, since the dilation and erosion within Python work on an entire image. In the algorithm we need to determine whether a certain statement is true before we decide to either erode or dilate that specific pixel. We will discuss some specific parts of the implementation in the Sections 5.1 to 5.6. All images used in these Sections are of the smaller type, we will discuss the images made by the new microscope in Section 5.6.

5.1 Locating cell centres

The very first thing we need to do, is locate all the cell centres. We use the fact that the cell interiors have a light colour, and thus very high gray values. And the cell walls are very dark, and have low gray values. We can use this fact to find the cell centres, which are local maxima in terms of gray values. However, these images have high amounts of detail and also noise, which leads to more local maxima being found than there are cell centres. To solve this, first a Gaussian filter is applied with standard deviation σ. For Gaussian filters with very small σ, still too many maxima will be found. But when we increase the value of σ, these details and noise are smoothed out and the amount of maxima found decreases. Of course, this σ can also become too big and eventually only one maximum will be found when the image is completely blurred. In Figure 4 the influence of σ with regard to the detection of cell centres is shown.

This Figure shows a range of σ from 2 to 30. The σ between 0 and 2 are not shown, because these values are very high and would lead to details in the rest of the curve not being visible. We can see that for small σ between 0 and 4, too many cells are detected. And for a σ larger than 15, the amount of cells found is approaching one. Because excessive smoothing will eventually lead to only a single maximum in the entire image. What we are looking for, is precisely in between. In the Figure there is a very noticeable bend in the curve around σ equal to 5. And when we look at the amount of cell centres found, this is very close to what we would expect. As mentioned in the Introduction, most pictures have between 2000 and 4000 cells. From this bend in the curve, the amount of cells found stops decreasing as fast. Another slightly less noticeable bend occurs around a σ of 15. From this value, the decrease slows down even

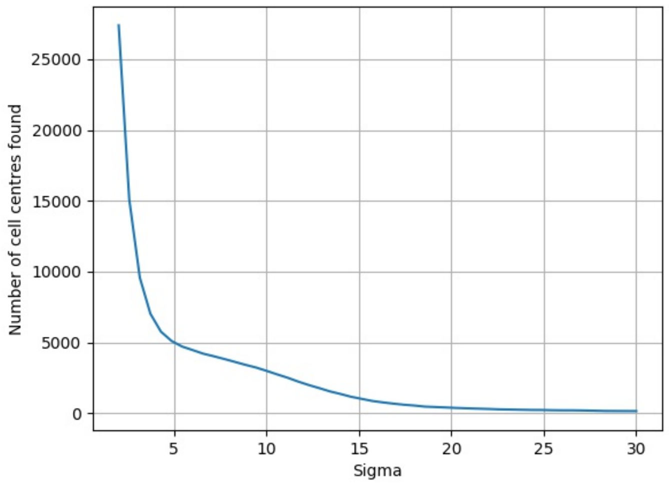


Figure 4: Cell centres found for different σ.

more and far too little cell centres are found. So the σ we are looking for must be somewhere between 5 and 15. When we look very closely at the curve for σ between 10 and 15, we see that it is decreasing a little bit faster than below 10. We also know that the amount of cell centres should lie above 2000, so values from 10 to 15 for σ are also not the best choice. This leads us to the conclusion that σ should have a value larger than 5 and smaller than 10. We decided on a value of 7 for σ, since this corresponds to the expected value. Also, since the level sets have the ability to merge, we would rather have a few cell centres too many than too little.

5.2 Function $g(I)$

The goal of the $g(I)$ function is that it highlights the interesting parts of the images. This means that the values of $g(I)$ should be low in the interesting areas, since active contours are energy minimizing. When the algorithm was derived, we wrote that a Gaussian filter was used for $g(I)$. To show why this was chosen, a few options have been plotted. In Figure 5 the original image is plotted with three different Gaussian filters applied to it. We will describe in Section 5.4 how the contours are extracted from level sets.

The only difference between the three Gaussian filters is which σ was used. In Figure 6 it is shown to which contours each of these options leads after 20 iterations of the algorithm. This is more than enough iterations for the contours to reach the cell walls and adapt according to influences of the cell walls. It is immediately visible that using the image itself is a bad choice for $g(I)$. The contours are very sensitive to the noise in the image.

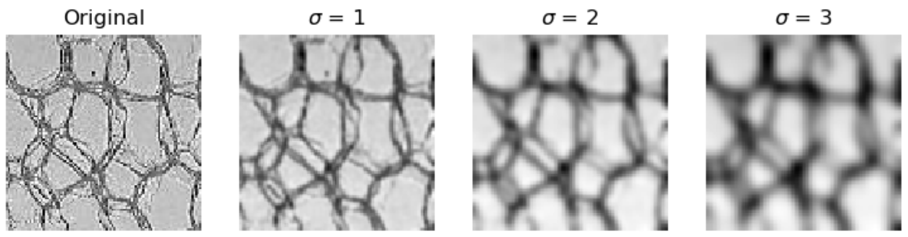


Figure 5: Different σ.

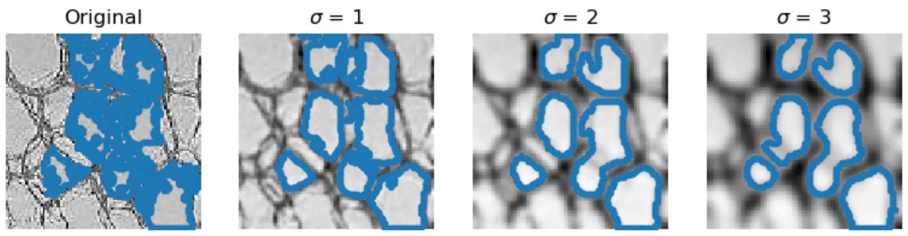


Figure 6: Contours for different σ

It is common to use some type of smoothing applied to the image for $g(I)$ as an edge detection. This smoothes out the unimportant noise and only leaves the more notable properties in an image, in this case cell walls. The only downside of using smoothing, is that the cell walls themselves become less sharp. We can see in Figure 6 that the evolution is quite accurate. But the contours do stop a few pixels before the actual walls are reached. Based on these results, we selected a Gaussian filter with σ 3.

Another option mentioned in the paper by Alvarez, Baumela and Marquez-Neila, uses not only a Gaussian filter, but also the gradient of the image. Using gradients is the second most common option to detect edges. A gradient becomes very high when nearing a change in image or an edge, and is almost zero on the edge itself. The function used is:

$$g(I) = \frac{1}{\sqrt{1 + \alpha|\nabla G_\sigma * I|}}.$$

Some examples of this function for different values of α are shown in image 7.

To be used in this application, the function is 1 divided by the gradient. This means that values close to the edges are very small, and the edges themselves have larger values and stay sharp. This sharpness is the reason that it is such an attractive choice for $g(I)$ Now the contour is attracted to values close to the edges, but not to the edges themselves. Also, the value of α has to be relatively big for these edges to be visible.

Again it is shown in Figure 8 to which contours these different varieties of the $g(I)$ lead for each value of α. It is obvious that this function is not a good

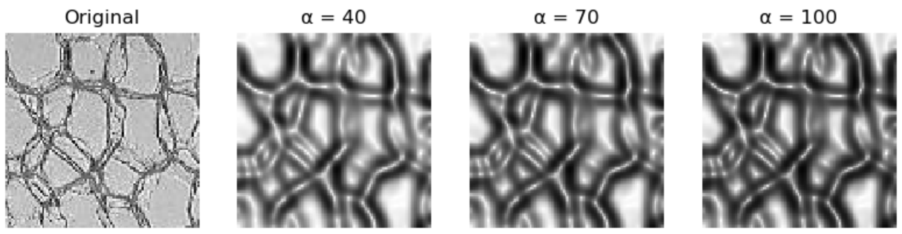


Figure 7: Different α.

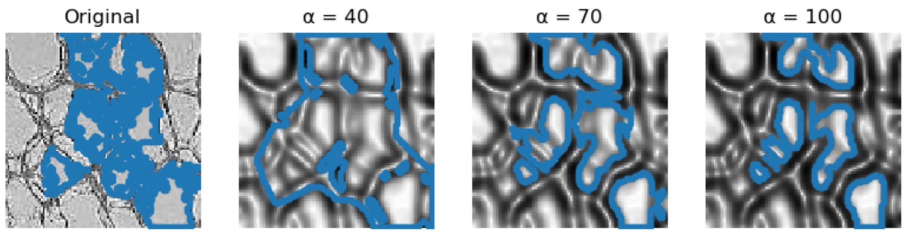


Figure 8: Contours for different α.

choice in itself. Even for an α of 100, there are still some bulges in the contour that we would not expect. And for any lower values, the edges are not strong enough to keep each contour separate and they merge together. Another option that was considered, was combining the previous two functions. However, this always lead to the sharp edges of the second function being blurred by the Gaussian. This was attempted for many different parameters but to no avail. None of these options performed better than a Gaussian filter with σ of 3.

5.3 Determining threshold θ

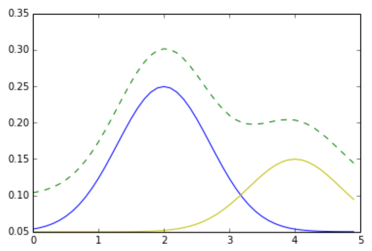


Figure 9: Two Gaussian distributions and their sum (green)

We need to determine an appropriate threshold θ which splits an image into cell walls and cell interiors. This is similar to extracting foreground and background from an image. And since extracting foreground objects from images is an important problem, a lot of theory has been developed for it. The foreground objects' pixels are distributed as a Gaussian, as are the background's pixels, known as a mixture of Gaussians [10]. An example of pixel distributions is depicted in Figure 9, as well as their sum. In a perfect image, this sum is what the histogram of all occurring gray values would look like. It is not immediately visible which Gaussian represents the foreground and which the background, but this can easily be derived. The height of a peak represents the relative amount of either foreground or background. And the location of the peak in terms of gray value represent an average color of that object: either dark or bright.

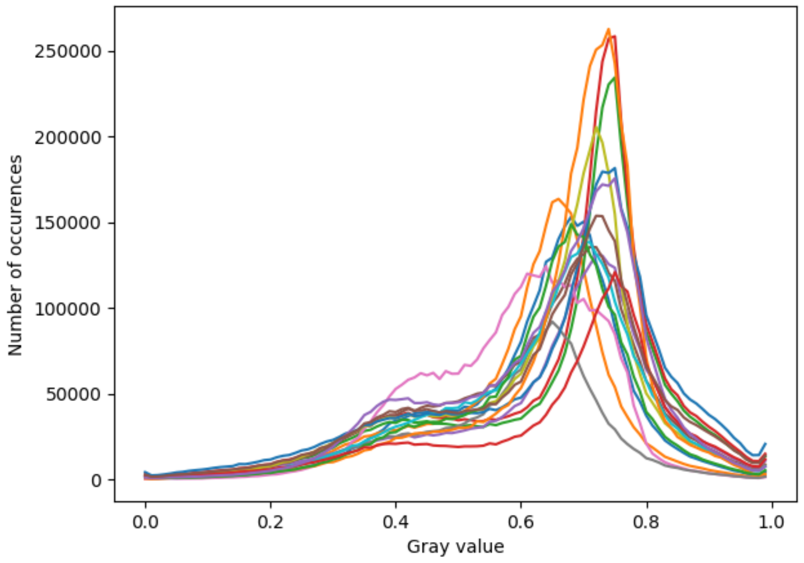


Figure 10: Histograms of 16 different images

In Figure 10 the histograms of sixteen of the smaller tuber cell images are plotted. As we can see, this does appear to take the shape of a sum of two Gaussians for each image. Some images have lower peaks, because they have less pixels in total. We want to pick a threshold which will fit any image. The contour will continue to grow until this threshold in reached, because of the balloon force. Based on this image of randomly selected images, a value somewhere between 0.5 and 0.7 could be a good fit. Because the balloon force can be quite strong, we would rather choose a larger value to be safe. We implemented this by calculating the histogram of an image and then looking up when 40 percent of the gray values have been passed, starting from the left hand side. The 40 percent boundary was taken from the paper by Alvarez, Baumela and Marquez-Neila [3]. We experimented with many different θ, but chosen values did not outperform the automatically calculated threshold.

5.4 Determining contour from level set

When the iterations are done, we are left with a level set whose edges determine the contours. Multiple options are considered and we start out by determining a single contour. The difficulty lies in the fact that sets are unordered and the contour is only represented implicitly. One option that was considered, is tracking which pixels are added in each iteration and always ordering them in a given direction. Unfortunately, this did not work out. Partly because of the amount of time and memory needed for this option. We would need to keep a list for each of the pixels in any contour. And for every operation done, we would need to figure out a way to always connect it to a unique origin pixel. Eventually we decided on a different approach which is faster and takes up no extra memory.

First, the unordered contour is extracted from the level set by eroding the level set with a N_4 structuring element and then calculating the difference with the original. We then have a set of all pixels that are part of the contour in random order. And as long as the contour is smooth, this gives us an 8-connected contour. Then we start by picking a random pixel that is part of the contour and go around it. This is done by finding both neighbouring pixels and then adding one that has not been added yet. Since we know the contour is 8-connected for sufficiently smooth contours, this should always work. Unfortunately, some problems arose because the contour was not smooth enough. And when a contour splits, or is about to split, the edge of the level set at that moment is also not smooth. We made some adaptations to the model in order to solve this problem. This will be further discussed in Section 6.2. The reason that this method is preferable over the other option, is that we only need to do all these calculations once instead of in each iteration. This will save us a lot of time throughout the processing of the images.

5.5 Properties of cells

Using this implementation, it is very easy to determine certain properties of all cells. For example, the area of cells in pixels can be calculated by adding up all values in the level set. The level set consists of only ones and zeros, and the area of a one by one square is also one. And determining the number of cells can already be done after the initialisation described in Section 5.1. Once we have extracted the contour using the method described above, we can easily calculate the perimeter. In the function we designed to find the contour, we also add all lengths together. When two pixel are connected directly, one is added, and when they are connected by their corners, the square root of two is added. This gives us a good approximation of the perimeter of a cell. When we have the area and perimeter of a cell, the isoparametric ratio and quotient can be determined using simple calculus. The maximum diameter and its direction are only slightly harder to calculate. We write a loop to iterate over each two elements in the contour and determine their Euclidean distance. When this distance is larger than any distance previously found, it is updated. In this loop we also keep track of the location the pixels with the biggest distance, so that we can determine their angle with respect to the horizontal axis. This shows that using the morphological GAC framework is definitely a solution to determine all mentioned properties.

5.6 Bigger images

The new microscope from HZPC can make pictures with a higher resolution than the other microscope. The images are then 50 megabytes, instead of the usual 1.5 to 2. Before processing of these images, they are first resized to be 5 times smaller in each direction, thereby making it about the same size as the other images. The values in the smaller image are calculated as the average of gray values in each 5 by 5 block in the original image. This operation does not only make the images smaller, but also smoother. Therefore we tried to use the image itself as $g(I)$, the result of this is shown for four different images in Figure 11.

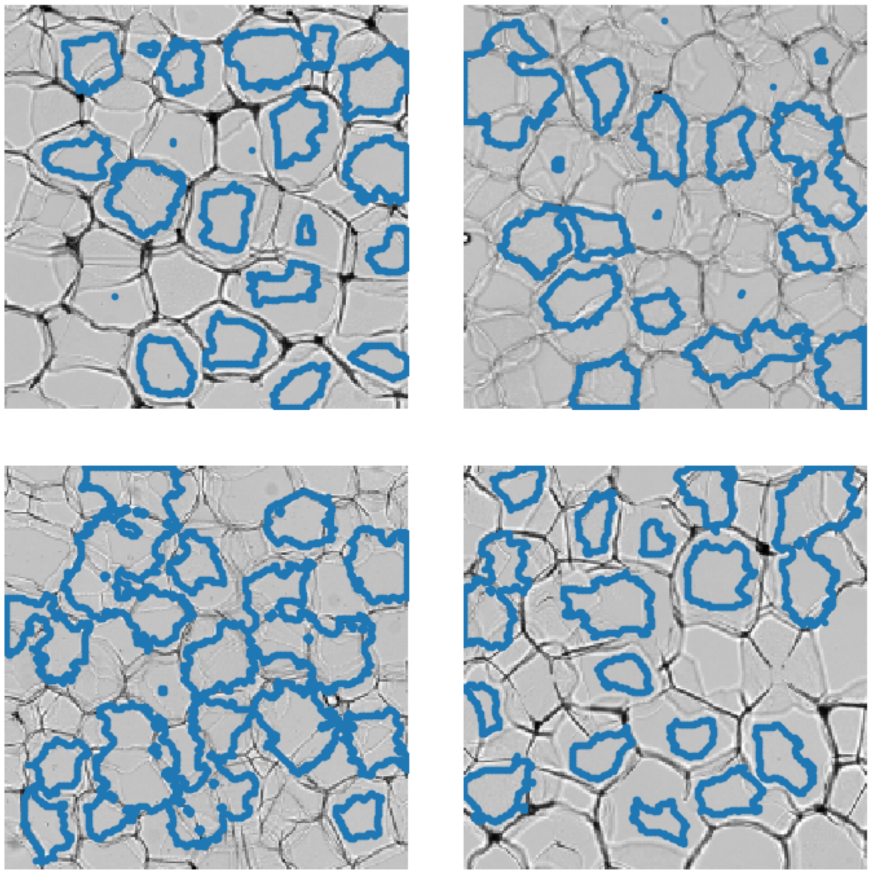


Figure 11: Four images using $g(I) = I$

These bigger images may be smoother, but the image itself is still not smooth enough to pose as a good choice of $g(I)$. Some other options were considered for $g(I)$, but none performed better than $g(I) = I$. We tried the option that worked very well on the smaller images: a Gaussian with σ of 3. The result of this is shown in Figure 12 in three different iterations. It is apparent that the contours all flow out of their cells and merge together. After this, we tried Gaussians with a smaller standard deviation, but the results were the same.

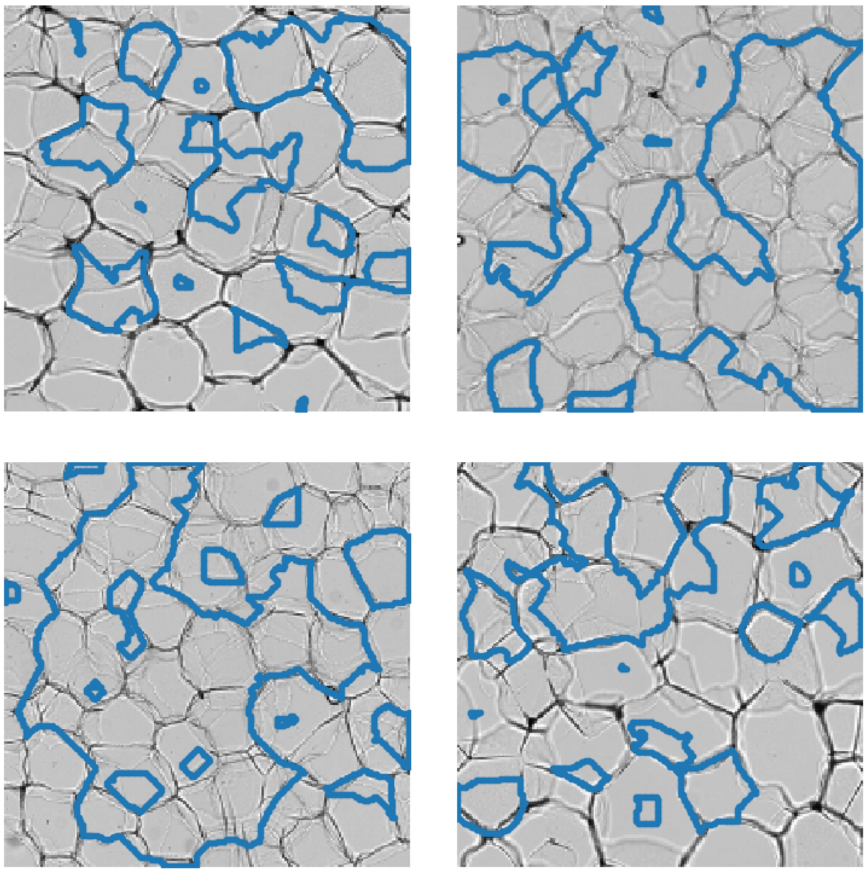


Figure 12: Four images using Gaussian with $\sigma = 3$ as $g(I)$

In Figure 13 it is shown what the histogram of 20 different high-resolution images looks like. In some of the images, the gray value of the cell walls are very light, and are indistinguishable from the larger peak for cell interiors. In other images we can no longer distinguish where the cell wall peak is, because it is too small. From inspecting the bigger images, we know this peak should be around 0.2 to 0.3. We therefore choose to use a fixed threshold of 0.4 instead of the one from Subsection 5.3.

Because the edges are simply too thin or lightly coloured to stop the contours, even after preprocessing, we choose to use the original smaller images only. Even

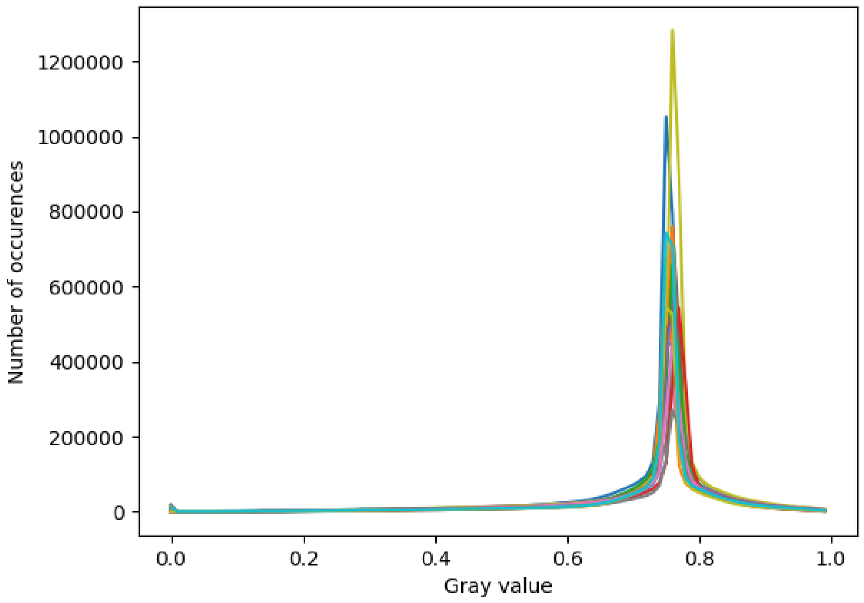


Figure 13: Histogram of bigger pictures

though they are noisier, they have more significant details in them.

6 Adaptations

In this Section all adaptations that were made to the algorithm are discussed.

6.1 Balloon Force Term

The balloon force term should have been implemented as:

$$u^{n+1}(x) = \begin{cases} (Du^n)(x) & \text{if } g(I)(x) > \theta \text{ and } \nu > 0 \\ (Eu^n)(x) & \text{if } g(I)(x) > \theta \text{ and } \nu < 0 \\ u^n(x) & \text{otherwise} \end{cases},$$

but we decided to implement it as:

$$u^{n+1}(x) = \begin{cases} (Du^n)(x) & \text{if } g(I)(x) > \theta \\ (Eu^n)(x) & \text{if } g(I)(x) < \theta \\ u^n(x) & \text{otherwise} \end{cases}, \tag{19}$$

because the balloon force parameter ν no longer has a strength and only its sign matters. The parameter was set to 1 when following the original algorithm, and then the balloon force was very difficult to stop. Most contours flowed out of their cell walls and merged with others. An example of this can be seen in Figure 14. And the same evolution, but now with an adapted balloon force, can be seen in Figure 15.

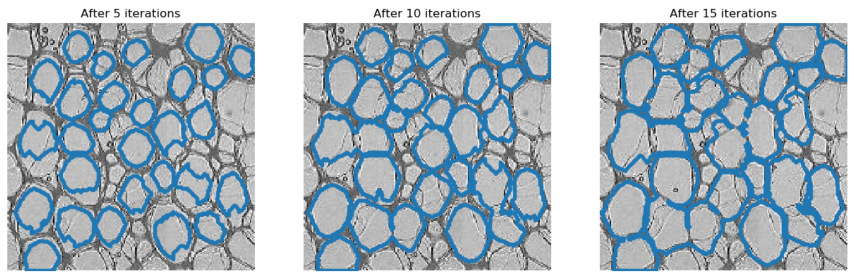


Figure 14: Original balloon force

6.2 Extra step in each iteration

In order to keep the contour smoother, some options were evaluated. The goal was to keep the contour from flowing out of the cells which were damaged or had a torn cell wall. In some cases the contour evolved very precisely around cell walls in 180 degree turns and became a lot less smooth. And when the contour does remain smooth, we can keep using the method described in Section 5.4 to

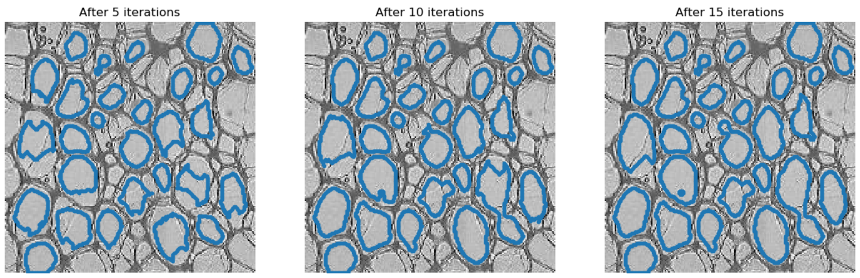
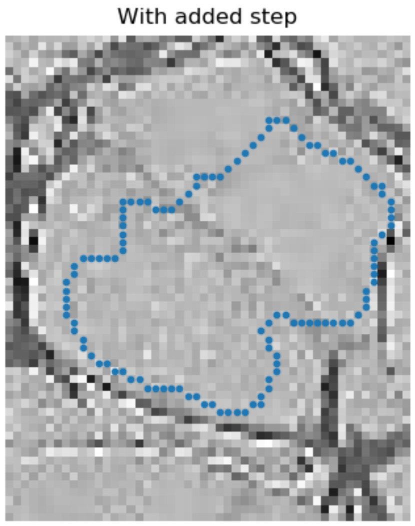


Figure 15: Adapted balloon force

determine contours.

The first and most logical option was to increase the smoothness parameter μ and apply multiple successive $SI \circ IS$ operations in every iteration. The results are seen in Figure 16. What becomes very clear from these images, is that extra operations did not have much effect on the overall smoothness.

Next, the addition of an extra step to the algorithm was considered. It seemed that an opening of the entire level set would be a good candidate because of the qualities discussed in Section 3.3. An opening is not only a smoothing operator, but also has the property that it can remove small protrusions. To test this, we first applied it to the same picture as seen in the image on the right. To study the extra step in more detail, we created a very simple cell with only black and white values. In Figure 17a it is shown what effect an opening with structuring element N_4 had on a contour which was flowing out of the cell.



This extra step does partly improve the smoothness, but did not stop the contour from flowing outward. In the next picture 17b it is depicted what an added opening step resulted in, only now with an N_8 structuring element. It not only smooths the contour, but also stops the contours from moving outside of

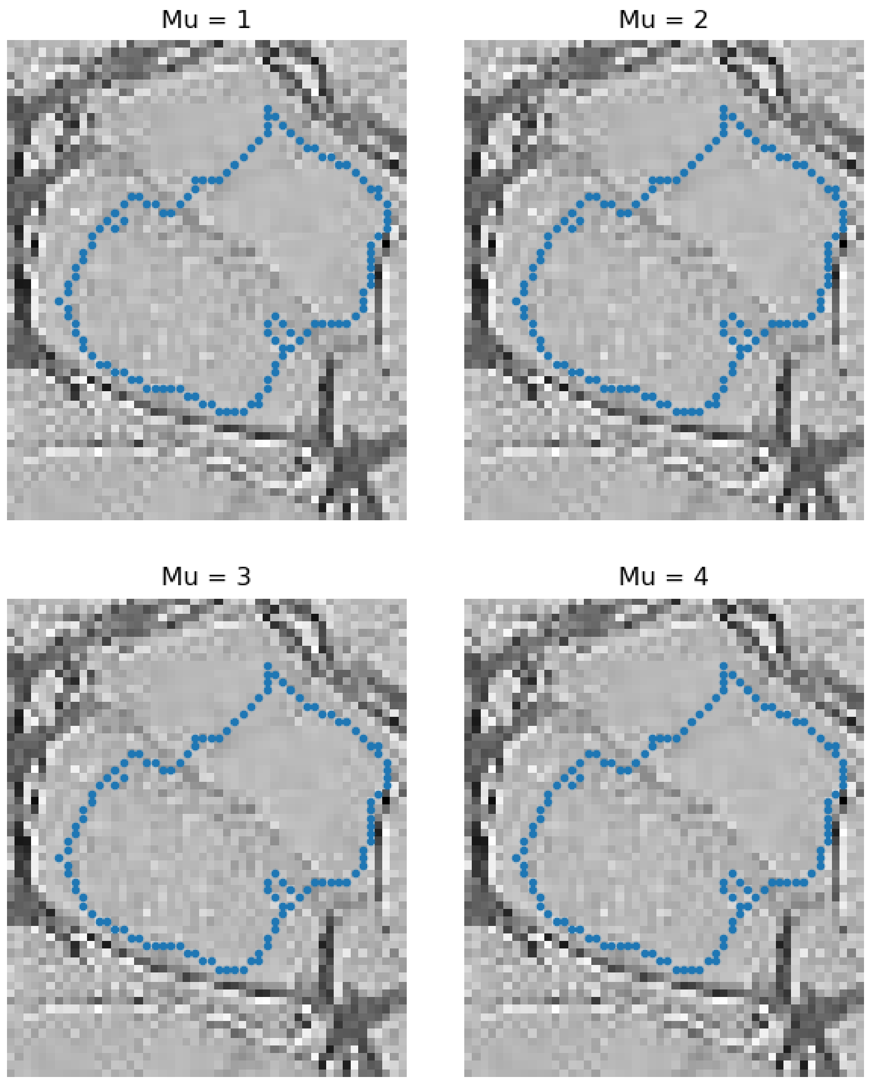
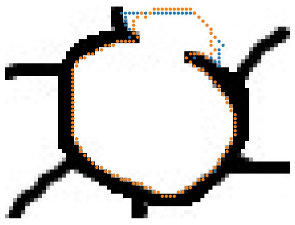


Figure 16: Contour for different μ.


(a) N_4

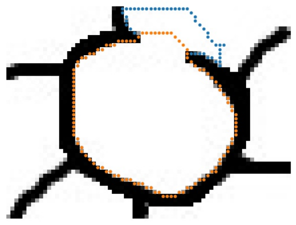

(b) N_8

Figure 17: Previous contours (blue) compared to new contours (orange)

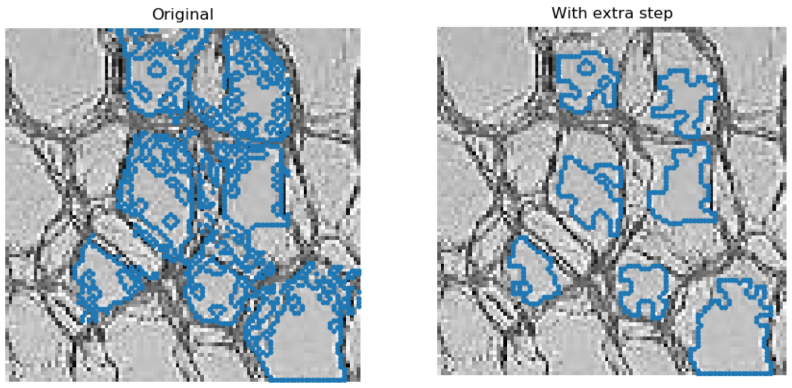


Figure 18: The original contours compared to the new ones.

the cells. On a simplified picture with very thick cell walls, this extra operation has precisely the desired effect.

As far as we tested this on actual cell images, the operation behaved as we wanted it to. We show an example of this adapted algorithm on multiple cells in Figure 18. We have chosen this example to be as non-smooth as possible on purpose to highlight the function of the added step. The original image is an example of choosing $g(I)$ to be equal to the image itself. As we described in Section 5.2 this is a bad choice of $g(I)$, because the contours are very sensitive to noise. However, we also see that the extra step can prevent almost all of the excessive merging. In the original image there are dozens of tiny contours around each noisy pixel which we do not want. Whereas in the new image, we can count the amount of tiny and wrong contours on one hand. This makes it very probable that the extra step can be a good adaptation to the algorithm. But of course there is no guarantee for the behaviour of this operation on every image we may encounter. It was not part of the original algorithm and has only been tested on our images of plant cells.

7 Conclusions

Our goal was to use the method of morphological geometric active contours to detect cells in microscopic images. The objects studied are the cells in cross-sections of potato tubers. These images were provided to us by the company HZPC to determine certain properties of the cells. We succeeded in implementing morphological GAC and were able to determine these properties quite easily as described in Section 5.5. It was shown that this method produces good results in Section 5.

Locating cell centres using Gaussian smoothing and local maxima detection proved to be a fast and efficient solution. The function $g(I)$ from the morphological GAC is a very important factor in the algorithm and determines almost entirely how well the contour evolves. As discussed in Section 5.2, for this specific application choosing a Gaussian with small σ was optimal. Using any type of gradient filter for detection of cell walls performed poorly compared to the Gaussian filter. More research could be done into designing a new function for the detection of cell walls. The sharpness of the edges of the gradient based $g(I)$ has some very attractive properties and could potentially be employed more effectively. We did not look into functions that were neither based on Gaussian nor on gradient filters, because these are the most conventional options. Perhaps there is a more suitable, more unconventional function that could be used or designed.

Determining a contour from an implicit representation proved to be fairly difficult. Our solution to this problem only works in certain cases. When a contour intersects or is simply not smooth, this solution fails. But in cases where the contour remains simple and smooth, it performed well. This smoothness should be preserved because of the smoothness step in every iteration. But even after increasing the parameter μ, we did not observe any change in the smoothness of the contours or in fact any change at all.

For this reason, we decided to adapt the existing algorithm. We added an extra step to be performed at the end of each iteration. This step consists of a single opening operation applied to the entire image. We showed that this performed better than the original algorithm in specific problem cases. These include ruptured cell walls and unexpected non-smooth contours. However, there is no mathematical basis for this addition. So adding this step might have unexpected consequences. Therefore, any contours produced using this new method must always be compared to the original method. More research should be done into the addition of this extra morphological operation.

An advantage, but possibly also a disadvantage, of morphological GAC is the ability to merge and split contours. We saw in Section 5.6 that the higher-resolution images behaved very differently and got worse results than the original ones. This is the opposite of what we expected. In the introduction we noted that these images should have less noise and are generally smoother. We would expect this to result in better edge detection. Unfortunately, the cell walls were too thin or too light to be detected by the algorithm and any type of smoothing only made them harder to detect. Perhaps more research could be done into choosing better parameters for specifically the high-resolution images made by the new microscope.

The method of morphological GAC proved useful in the images with well defined cell walls. Some problems arose for broken cells, because the original algorithm did not keep the contour smooth enough. We therefore introduced an extra step to be executed at the end of each iteration. The first option with an N_4 structuring element opening hardly had any impact, but the opening with an N_8 structuring element did. Even when the cells had tears in them, the additional step could handle this. The adaptations made to the algorithm were necessary and provided good results. The change in implementation of the balloon force described in Section 2 was very natural. It also helped to stop the contours from flowing out of their designated cells. Again this outflow was partly a result of the contours' ability to merge. This merging and splitting of the contours can be useful in certain applications. For us this proved to be both a difficulty, but also an advantage. Keeping the contours smooth enough required some extra work, but the sensitivity we gained turned out to be very useful.

All in all there are many factors which influence the evolution of the morphological GAC. It can be concluded that this algorithm is very sensitive to each parameter. Not only the threshold θ, but especially the $g(I)$ function appeared to have a big impact on the contour evolution. A poor choice of $g(I)$ immediately resulted in contours either overlapping or never reaching the cell walls. And unexpectedly, the balloon parameter ν and smoothing force parameter μ appeared to have little to no influence at all. These results need to be considered carefully when applying morphological GAC and especially the choice of parameters such as $g(I)$ and θ.

8 References

[1] Retrieved 13-06-2018 from `https://www.hzpc.com/`.

[2] Osher, S., & Sethian, J. A. (1988). Fronts propagating with curvature-dependent speed. *Journal of Computational Physics, 79(1).* doi:10.1016/0021-9991(88)90002-2

[3] Marquez-Neila, P., Baumela, L., & Alvarez, L. (2014). A Morphological Approach to Curvature-Based Evolution of Curves and Surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 36(1).* doi:10.1109/tpami.2013.106a

[4] Chenyang, X., Yezzi, A., & Prince, J. (2000). On the relationship between parametric and geometric active contours. *Conference Record of the Thirty-Fourth Asilomar Conference on Signals, Systems and Computers (Cat. No.00CH37154).* doi:10.1109/acssc.2000.911003

[5] Catte, F., Dibos, F., & Koepfler, G. (1994). A morphological scheme for mean curvature motion and applications to anisotropic diffusion and motion of level sets. *Proceedings of 1st International Conference on Image Processing.* doi:10.1109/icip.1994.413268

[6] Alvarez, L., Guichard, F., Lions, P., & Morel, J. (1993). Axioms and fundamental equations of image processing. *Archive for Rational Mechanics and Analysis, 123(3), 199-257.* doi:10.1007/bf00375127

[7] Cohen, L. D. (1991). On active contour models and balloons. *CVGIP: Image Understanding, 53(2).* doi:10.1016/1049-9660(91)90028-n

[8] Sternberg, S. R. (1986). Grayscale morphology. *Computer Vision, Graphics, and Image Processing, 35(3). 333-355.* doi:10.1016/0734-189x(86)90004-6

[9] Matherson, G., & Serra, J. *The Birth of Mathematical Morphology.* Retrieved from `http://cmm.ensmp.fr/~serra/pdf/birth_of_mm.pdf`

[10] Kulkarni, M. (2010). Histogram-based foreground object extraction for indoor and outdoor scenes. *Proceedings of the Seventh Indian Conference on Computer Vision, Graphics and Image Processing.* doi:10.1145/1924559.1924579

[11] Figure in introduction was taken from: `https://porcelainfacespa.com/blog/quick-fix-it/`

[12] Retrieved from `https://www.quora.com/What-is-neighbors-of-a-pixel`

[13] Alvarez, L., Baumela, L., Mrquez-Neila, P., & Henrquez, P. (2012). A Real Time Morphological Snakes Algorithm. *Image Processing On Line, 2.* doi:10.5201/ipol.2012.abmh-rtmsa

[14] Caselles, V., Kimmel, R., & Sapiro, G. (1995). Geodesic active contours. *Proceedings of IEEE International Conference on Computer Vision.* doi:10.1109/iccv.1995.466871

[15] Alvarez, L., Baumela, L., Henriquez, P., & Marquez-Neila, P. (2010). Morphological snakes. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition.* doi:10.1109/cvpr.2010.5539900

[16] Sapiro, G. (2001). Geometric Partial Differential Equations and Image Analysis. doi:10.1017/cbo9780511626319

[17] Morphological Image Processing. (2013). Retrieved from `https://www.slideshare.net/Johnrebel999/morphological-image-processing-22899372`

[18] Mathematical morphology - Encyclopedia of Mathematics. Retrieved from `https://www.encyclopediaofmath.org/index.php/Mathematical_morphology`

[19] Image on title page was retrieved from `https://www.med.muni.cz/biofyz/Image/analyza.html`

A Python code

```
from scipy import ndimage
import matplotlib.pyplot as plt
import numpy as np
from skimage import io, color, feature, transform
import copy
from skimage import morphology as m


## Some often used structures to reduce runtime
struct = np.ones((3,3)).astype(dtype=int)
structure = np.zeros((21,21)).astype(dtype=int)
structure[10][10] = 1
for i in range(7):
    structure = m.binary_dilation(structure)
###############################################

def ready_image(name):
    """Load image, convert to b&w and return it."""
    res = io.imread(str(name)+".jpg")
    res = color.rgb2gray(res)
    return res

def ready_image2(name):
    """Same as ready_image, but makes image 5 times smaller."""
    img = io.imread(str(name)+".jpg")
    img = color.rgb2gray(img)
    res = transform.rescale(img,1/5,anti_aliasing = False)
    return res

def levelset_initial(image,sigma):
    """Finds cell centres given an image and sigma and initialises
                                     level set."""
    gaus = ndimage.gaussian_filter(image, sigma)
    points = feature.peak_local_max(gaus)
    level_set = np.zeros([len(image),len(image[0])])
    for elt in points:
        level_set[elt[0]][elt[1]] = 1
    return m.binary_dilation(level_set,structure).astype(dtype =
                                     int)

def dilatie(u,row,col):
    """Dilation of a single pixel without changing u, only
                                     determines new value."""
    if u[row][col]:
        return 1
    else:
        if row > 0:
            if u[row-1][col]:
                return 1
            if col > 0:
                if u[row-1][col-1]:
                    return 1
            if col < u.shape[1]-1:
                if u[row-1][col+1]:
                    return 1
        if row < u.shape[0]-1:
            if u[row+1][col]:
                return 1
            if col > 0:
                if u[row+1][col-1]:
                    return 1
            if col < u.shape[1]-1:
                if u[row+1][col+1]:
                    return 1
        if col > 0:
            if u[row][col-1]:
                return 1
        if col < u.shape[1]-1:
            if u[row][col+1]:
                return 1
        return 0



def erosie(u,row,col):
    """Erosion of a single pixel without changing u, only
                                      determines new value."""
    if not u[row][col]:
        return 0
    else:
        if row != 0:
            if not u[row-1][col]:
                return 0
            if col != 0:
                if not u[row-1][col-1]:
                    return 0
            if col != u.shape[1]-1:
                if not u[row-1][col+1]:
                    return 0
        if row != u.shape[0]-1:
            if not u[row+1][col]:
                return 0
            if col != 0:
                if not u[row+1][col-1]:
                    return 0
            if col != u.shape[1]-1:
                if not u[row+1][col+1]:
                    return 0
        if col != 0:
            if not u[row][col-1]:
                return 0
        if col != u.shape[1]-1:
            if not u[row][col+1]:
                return 0
        return 1

def central(field, row, col):
    """Calculate central difference of first derivative, one-sided
                                      difference \
    on edges of image. Then returns it as a numpy array. """
    if row == 0:
        dy = field[row][col] - field[row+1][col]
    elif row == field.shape[0] - 1:
        dy = field[row-1][col] - field[row][col]
    else:
        dy = (field[row-1][col] - field[row+1][col])/2.
    if col == 0:
        dx = field[row][col+1] - field[row][col]
    elif col == field.shape[1] - 1:
        dx = field[row][col] - field[row][col-1]
    else:
        dx = (field[row][col+1] - field[row][col-1])/2.
    return np.array([dx,dy])

def SI1(u,row,col,mid):
    """SI operator on a single pixel."""
    if u[row][col] == 0:
        return 0
    if row == 0 or row == u.shape[0]-1 or col == 0 or col == u.
                                        shape[1]-1:
        return mid
    else:
        if u[row][col+1] and u[row][col-1] and mid:
            return 1
        if u[row+1][col+1] and u[row-1][col-1] and mid:
            return 1
        if u[row+1][col] and u[row-1][col] and mid:
            return 1
        if u[row-1][col+1] and u[row+1][col-1] and mid:
            return 1
    return 0

def IS1(u,row,col,mid):
    """IS operator on a single pixel."""
    if u[row][col]:
        return 1
    if row == 0 or row == u.shape[0]-1 or col == 0 or col == u.
                                        shape[1]-1:
        return mid
    else:
        if u[row][col+1] == 0 and u[row][col-1] == 0 and mid == 0:
            return 0
        if u[row+1][col+1] == 0 and u[row-1][col-1] == 0 and mid ==
                                              0:
            return 0
        if u[row+1][col] == 0 and u[row-1][col] == 0 and mid == 0:
            return 0
        if not u[row-1][col+1] and not u[row+1][col-1] and not mid:
            return 0
        return 1

def find_contour(level_set):
    """Finds contours given level set using morphological
                                        operations, \
    return coordinates."""
    eroded = ndimage.binary_erosion(level_set).astype(dtype = int)
    difference = level_set - eroded
    coords = np.where(difference == 1)
    return coords
```

```
def plot_contour_8(f):
    """Orders given sets of coordinates so that a contour can be
                                  plotted. \
        Only works on a single contour. Also returns length of
                                      contour. Assumes\
        contour is 8-connected."""
    x0 = f[1][0]
    y0 = f[0][0]
    xarray = np.array([x0])
    yarray = np.array([y0])
    lengte = 0
    while len(xarray) != len(f[0]):
        if not set(np.where(f[1]==x0)[0]).isdisjoint(set(np.where(f
                                          [0]==y0+1)[0]))\
           and set(np.where(xarray==x0)[0]).isdisjoint(set(np.where
                                             (yarray==y0+1)[0])):
            #and x0 not in xarray and y0+1 not in yarray:
            xarray = np.append(xarray,x0)
            yarray = np.append(yarray,y0+1)
            y0 = y0+1
            lengte += 1
        elif not set(np.where(f[1]==x0)[0]).isdisjoint(set(np.where
                                          (f[0]==y0-1)[0]))\
             and set(np.where(xarray==x0)[0]).isdisjoint(set(np.
                                               where(yarray==y0-1)[
                                               0])):
            #and x0 not in xarray and y0-1 not in yarray:
            xarray = np.append(xarray,x0)
            yarray = np.append(yarray,y0-1)
            y0=y0-1
            lengte += 1
        elif not set(np.where(f[1]==x0+1)[0]).isdisjoint(set(np.
                                          where(f[0]==y0)[0]))\
             and set(np.where(xarray==x0+1)[0]).isdisjoint(set(np.
                                               where(yarray==y0)[0]
                                               )):
            #and  x0+1 not in xarray and y0 not in yarray:
            xarray = np.append(xarray,x0+1)
            yarray = np.append(yarray,y0)
            x0=x0+1
            lengte += 1
        elif not set(np.where(f[1]==x0-1)[0]).isdisjoint(set(np.
                                          where(f[0]==y0)[0]))\
             and set(np.where(xarray==x0-1)[0]).isdisjoint(set(np.
                                               where(yarray==y0)[0]
                                               )):
            #and y0 not in xarray and x0-1 not in yarray:
            xarray = np.append(xarray,x0-1)
            yarray = np.append(yarray,y0)
            x0=x0-1
            lengte += 1

        elif not set(np.where(f[1]==x0-1)[0]).isdisjoint(set(np.
                                          where(f[0]==y0-1)[0]))\
             and set(np.where(xarray==x0-1)[0]).isdisjoint(set(np.
                                               where(yarray==y0-1)[
                                                    0])):
            #and y0 not in xarray and x0-1 not in yarray:
            xarray = np.append(xarray,x0-1)
            yarray = np.append(yarray,y0-1)
            x0=x0-1
            y0=y0-1
            lengte += np.sqrt(2)
        elif not set(np.where(f[1]==x0-1)[0]).isdisjoint(set(np.
                                              where(f[0]==y0+1)[0]))\
             and set(np.where(xarray==x0-1)[0]).isdisjoint(set(np.
                                                   where(yarray==y0+1)[
                                                   0])):
            #and y0 not in xarray and x0-1 not in yarray:
            xarray = np.append(xarray,x0-1)
            yarray = np.append(yarray,y0+1)
            x0=x0-1
            y0=y0+1
            lengte += np.sqrt(2)
        elif not set(np.where(f[1]==x0+1)[0]).isdisjoint(set(np.
                                              where(f[0]==y0+1)[0]))\
             and set(np.where(xarray==x0+1)[0]).isdisjoint(set(np.
                                                   where(yarray==y0+1)[
                                                   0])):
            #and y0 not in xarray and x0-1 not in yarray:
            xarray = np.append(xarray,x0+1)
            yarray = np.append(yarray,y0+1)
            x0=x0+1
            y0=y0+1
            lengte += np.sqrt(2)
        elif not set(np.where(f[1]==x0+1)[0]).isdisjoint(set(np.
                                              where(f[0]==y0-1)[0]))\
             and set(np.where(xarray==x0+1)[0]).isdisjoint(set(np.
                                                   where(yarray==y0-1)[
                                                   0])):
            #and y0 not in xarray and x0-1 not in yarray:
            xarray = np.append(xarray,x0+1)
            yarray = np.append(yarray,y0-1)
            x0=x0+1
            y0=y0-1
            lengte += np.sqrt(2)
        else:
            print("Geen buur")
    xarray = np.append(xarray,f[1][0])
    yarray = np.append(yarray,f[0][0])
    return xarray,yarray,lengte

def balloon(level_set,info,row,col,threshold):
    """Returns result of balloon operator on a single pixel, \
    does not change level_set."""
    if info[row][col] > threshold:
        return dilatie(level_set,row,col)
    elif info[row][col] < threshold:
        return erosie(level_set,row,col)
    else:
        return level_set[row][col]

def attraction(level_set, gI, row, col):
    """Returns result of attraction operator on a single pixel, \
    does not change level_set."""
    var = np.dot(central(level_set,row,col),gI[row][col])
    if var > 0:
        return 1
    elif var < 0:
        return 0
    else:
        return level_set[row][col]

def evolve(u,gI,deriv,t):
    """Computes one iteration of evolution algorithm on entire
                                         image \
    with u level set, and deriv contains central differences in
                                         each pixel, \
    and t is threshold for balloon force."""
    u_b = u.copy()
    for row in range(u.shape[0]):
        for col in range(u.shape[1]):
            u_b[row][col] = balloon(u,gI,row,col,t)

    u_a = u_b.copy()
    for row in range(u.shape[0]):
        for col in range(u.shape[1]):
            u_a[row][col] = attraction(u_b,deriv,row,col)

    u_b = u_a.copy()
    for row in range(u.shape[0]):
        for col in range(u.shape[1]):
            val = IS1(u_a,row,col,u_a[row][col])
            u_b[row][col] = SI1(u_a,row,col,val)

    #Optional fourth step:
    #u_b = m.binary_opening(u_b,struct).astype(dtype=int)
    return u_b

def run(image,gI,steps):
    """Calculates steps*4 iterations of morphGAC of image using gI
                                         as g(I). \
    Returns result after each _steps_ steps."""
    # Determine threshold for balloon force.
    hist,bins = np.histogram(image,bins = 100)
    num = 0
    j = 0
    grootte = (image.shape[0]*image.shape[1])
    while num < 0.4:
        num += hist[j]/grootte
        j += 1
    t = bins[j]

    # Calculate central differences in each pixel.
    diffs = np.zeros((image.shape[0],image.shape[1],2))
    for x in range(image.shape[0]):
        for y in range(image.shape[1]):
            gradi = central(gI,x,y)
            diffs[x][y] = gradi
    h1 = levelset_initial(image,7)
    for i in range(steps):
        h1 = evolve(h1,gI,diffs,t)
    h2 = h1.copy()
    for i in range(steps):
        h2 = evolve(h2,gI,diffs,t)
    h3 = h2.copy()
    for i in range(steps):
        h3 = evolve(h3,gI,diffs,t)
    h4 = h3.copy()
    for i in range(steps):
        h4 = evolve(h4,gI,diffs,t)
    return h1,h2,h3,h4

if __name__ == "__main__":
    # Insert name of image in place of "HERE" without .jpg (should
                                  be jpg type)
    # Select part of image (row1 < row2 and col1 < col2).
    # image = ready_image("HERE")[row1:row2][col1:col2]

    # It is advised to use a maximum of a 200x200 size part of an
                                  image.
    # An example is given below for an image stored under "image1.
                                  jpg".
    image = ready_image("image11")[700:800,1050:1150]

    gI = ndimage.gaussian_filter(image, 3)
    steps = 3
    h1,h2,h3,h4 = run(image,gI,steps)
    g1 = find_contour(h1)
    g2 = find_contour(h2)
    g3 = find_contour(h3)
    g4 = find_contour(h4)

    plt.subplot(221)
    plt.axis("off")
    plt.imshow(image, cmap = 'gray')
    plt.plot(g1[1],g1[0],".")
    #Optional to plot initial contours
    #plt.plot(r[1],r[0],".")
    plt.title("After "+str(steps)+" iterations")

    plt.subplot(222)
    plt.axis("off")
    plt.imshow(image,cmap = "gray")
    plt.plot(g2[1],g2[0],".")
    plt.title("After "+str(2*steps)+" iterations")

    plt.subplot(223)
    plt.axis("off")
    plt.imshow(image,cmap = "gray")
    plt.plot(g3[1],g3[0],".")
    plt.title("After "+str(3*steps)+" iterations")

    plt.subplot(224)
    plt.axis("off")
    plt.imshow(image,cmap = "gray")
    plt.plot(g4[1],g4[0],".")
    plt.title("After "+str(4*steps)+" iterations")

    #Optional: save figure.
    #plt.savefig("name.jpg")
    plt.show()
```