Delft University of Technology
Faculty of Electrical Engineering, Mathematics & Computer Science
Delft Institute of Applied Mathematics

Analysis of Microscopic Images: Boundary Detection Using a Modified Mumford-Shah Model

Analyse van Microscopische Afbeeldingen: Grensdetectie met behulp van een Gemodificeerd Mumford-Shah Model

Bachelor thesis submitted to the
Delft Institute of Applied Mathematics
as part to obtain

the degree of

BACHELOR OF SCIENCE
in
APPLIED MATHEMATICS

by

ARTHUR KERST

Delft, the Netherlands
June 2018

BSc thesis APPLIED MATHEMATICS

"Analysis of Microscopic Images: Boundary Detection Using a Modified Mumford-Shah Model "

"Analyse van Microscopische Afbeeldingen: Grensdetectie met behulp van een Gemodificeerd Mumford-Shah Model"

Arthur Kerst

Delft University of Technology

Supervisor

Dr. N.V. Budko

Other members of the committee

Drs. E.M. van Elderen

Dr. J.G. Spandaw

June, 2018

Delft

1 Abstract

Deformable energy-minimizing contours are used in image analysis, particularly to detect boundaries from objects. From the Mumford-Shah model we derive an equation to move such contours towards the boundary, which minimizes the functional. The external force is computed using the mean value of the inside and outside area of the contour. Various image filtering methods are proposed to enhance this model. We introduce an improved Mumford-Shah model to improve the computational time without losing accuracy. An artificial force is added at the beginning of the simulation to quickly yield an approximation of the contour. Finally, we consider a simplified version of the model, which we solve analytically. The solution is used to find suitable parameters for the model.

Vervormbare energie-minimaliserende contouren worden gebruikt in beeldanalyse, in het bijzonder om randen van objecten te detecteren. Van het Mumford-Shah model ontlenen we een vergelijking om contouren naar de grens te verplaatsen, zodat de functionaal wordt geminimaliseerd. De externe kracht wordt berekend met behulp van de gemiddelde waarde van het binnen- en buitengebied van de contour. Verschillende beeldbewerking methoden worden voorgesteld om dit model te verbeteren. We introduceren een verbeterd Mumford-Shah model om de rekentijd te verbeteren zonder de nauwkeurigheid te verliezen. Aan het begin van de simulatie wordt een kunstmatige kracht toegevoegd om snel een benadering van de contour te verkrijgen. Ten slotte beschouwen we een vereenvoudigde versie van het model, die we analytisch oplossen. De oplossing wordt gebruikt om geschikte parameters voor het model te vinden.

Contents

2 Introduction

The aim of this project is to improve the development of seed potato by analyzing microscopic images for the company HZPC. HZPC is an innovative global market leader in potato breeding, seed potato trade and product concept development[1]. Breeders develop seed potato varieties that optimally match local growing conditions. In order to improve the development of seed potato, microscopic images are used which contain useful information about the plant phenotype. The images are studied to find parameters, such as the number of cells, areas and diameters of cells, ellipticity of cells and other geometrical parameters. Our goal is to locate the boundary of the cells; this can be used to determine the previously mentioned parameters.

An important problem in image analysis is to find the boundary of objects in an image. Different models with deformable contours have been proposed, each with a different approach. Every model makes use of a functional, which describes the forces working on the contour. In [5] a gradient vector flow (GVF) has been introduced to solve the problem using the vector field of the image. The boundary can also be found by inflating the contour, as has been done in the balloon model [4].
In this study, the Mumford-Shah model [1] is discussed and applied, which uses the image date from the inside and outside of the contour to locate the boundary. With this model, we wish to find the boundaries of potato cells in a large image. We assume that these object are smooth as in [1], [4] and [5], such that we can use smooth deformable contours. The contours can move under the influence of internal and external forces. The external force is designed to attract the contour towards the boundary, and the internal force is necessary to keep the contour smooth. These forces are described by the Mumford-Shah functional. The optimal solution of the problem corresponds to the minimum of the functional, thus our goal is to minimize this functional.

Besides the model, other techniques from graph theory and image filtering are discussed, which have been used to yield a desired result. Such as; locating the center of a cell using Euler's formula and removing noise using an Gaussian blur filter.

Since our problem consists of finding the boundaries in a large image, the computational time is fairly important. We introduce a new improved Mumford-Shah model which does not necessarily improves the accuracy, but rather the execution time.
Furthermore, we have considered a simplified Mumford-Shah model. The model is solved analytically with Fourier series. The solution is useful for investigating the influence of parameters of the model on the time evolution of the contour

[1] www.hzpc.com

3 Functional Minimization

3.1 Introduction

For this problem we define a contour in a domain. We adapt the shape of this contour in every time step until the contour is at the boundary of the cell. In order to do this, we have to define a functional $E(u, C)$, where u is an approximation function of a given image f and C the contour. The idea of this method is to minimize this functional, by optimizing the contour C and the approximation function u. In order to use the functional for our model, we first have to find the Euler-Lagrange equation by computing its variation with respect to the unknowns u and C. The Euler-Lagrange can be used in our model to adapt the shape of the contour.

3.2 Mumford-Shah Model

The image domain is denoted by $\Omega \subset \mathbb{R}^2$. Let $f : \Omega \to \mathbb{R}$ be a given gray-scale image. The domain Ω is split up in the inside and outside area of the contour C, denoted by Ω_i and Ω_o. The contour C is given by $\boldsymbol{r}(s)$, with $0 \leq s < 1$.

$$\Omega = \Omega_i \cup \Omega_o \cup C.$$

The functional E of the modified Mumford-Shah model [2] to be minimized is given by,

$$E(u, C) = \int_\Omega (f-u)^2 dxdy + \int_{\Omega \setminus C} |\nabla u|^2 dxdy + \frac{1}{2}\nu \int_0^1 |\boldsymbol{r}_s|^2 ds. \tag{1}$$

The original Mumford-Shah functional [1] has the term $\frac{1}{2}\nu \int_0^1 |\boldsymbol{r}_s| ds$ instead of $\frac{1}{2}\nu \int_0^1 |\boldsymbol{r}_s|^2 ds$. We choose to use the latter term, such that the Euler-Lagrange equation can be found in a more convenient manner [2]. The Euler-Lagrange equation will be discussed later in this section.

The idea of the Mumford-Shah model is to compute the optimal approximation of the general image function $f(x, y)$ by a piecewise-smooth function $u(x, y)$. However, we will be using a more simple case of the Mumford-Shah model, where $u(x, y)$ is a piecewise-constant function. This is called the cartoon model [2]. The functional E then simplifies to

$$E(u, C) = \int_\Omega (f-u)^2 dxdy + \frac{1}{2}\nu \int_0^1 |\boldsymbol{r}_s|^2 ds. \tag{2}$$

We can rewrite this as

$$E(u, C) = \int_{\Omega_i} (f-u_i)^2 dxdy + \int_{\Omega_o} (f-u_o)^2 dxdy + \frac{1}{2}\nu \int_0^1 |\boldsymbol{r}_s|^2 ds. \tag{3}$$

where u_i and u_o are constant functions inside Ω_i and Ω_0 respectively. Here, Ω_i is the area inside and Ω_o the area outside the contour C.

3.2.1 Minimizing in u with fixed C

We want to find the optimal function u such that the functional E is minimized for that function u. Let us assume that C is fixed and that u is optimal. We define a new functional $\epsilon \mapsto A(\epsilon) = E(u + \epsilon v)$ for all $\epsilon \in \mathbb{R}_{>0}$ and all real valued constant functions v. This is

$$A(\epsilon) = \int_{\Omega_i} (f-u_i-\epsilon v)^2 dxdy + \int_{\Omega_o} (f-u_o-\epsilon v)^2 dxdy + \frac{1}{2}\nu \int_0^1 |\boldsymbol{r}_s|^2 ds. \tag{4}$$

Taking the derivative with respect to ϵ yields

$$\frac{d}{d\epsilon} A(\epsilon) = -2 \int_{\Omega_i} v(f-u_i-\epsilon v) dxdy - 2 \int_{\Omega_o} v(f-u_o-\epsilon v) dxdy.$$

We must have $E(u, C) = A(0) \leq A(\epsilon) = E(u + \epsilon v, C)$ for all real valued functions v and all $\epsilon \in \mathbb{R}_{>0}$, since u is optimal for E. Thus we impose for all such v the following, as has been done in [3]:

$$\frac{d}{d\epsilon} A(0) = 0. \tag{5}$$

Then we get

$$-2v \int_{\Omega_i} (f - u_i) dxdy - 2v \int_{\Omega_o} (f - u_o) dxdy = 0.$$

Now we use that u_i, u_0 and v are constant functions. Then we yield

$$u_i \int_{\Omega_i} 1 dxdy + u_o \int_{\Omega_o} 1 dxdy = \int_{\Omega_i} f dxdy + \int_{\Omega_o} f dxdy.$$

This equation holds if

$$u_i = \frac{\int_{\Omega_i} f dxdy}{\int_{\Omega_i} 1 dxdy} =: \text{mean}_{\Omega_i}(f), \quad u_0 = \frac{\int_{\Omega_0} f dxdy}{\int_{\Omega_0} 1 dxdy} =: \text{mean}_{\Omega_o}(f), \tag{6}$$

thus we find that the functional is minimized by setting $u_i = \text{mean}_{\Omega_i}(f)$ and $u_o = \text{mean}_{\Omega_o}(f)$. Therefore it is sufficient to minimize

$$E(C) = \int_{\Omega_i} (f - u_i)^2 dxdy + \int_{\Omega_o} (f - u_o)^2 dxdy + \frac{1}{2}\nu \int_0^1 |\boldsymbol{r}_s|^2 ds, \tag{7}$$

with u_i and u_o as described as in (6).

3.2.2 Minimizing in C

In the previous section we minimized with respect to u and with fixed C, but now we want to minimize the functional with respect to C. We assume that (u, C) is a minimizer of E and we vary C. The variation C_ϵ of C is shown in Figure 1 and will be defined later in this section.

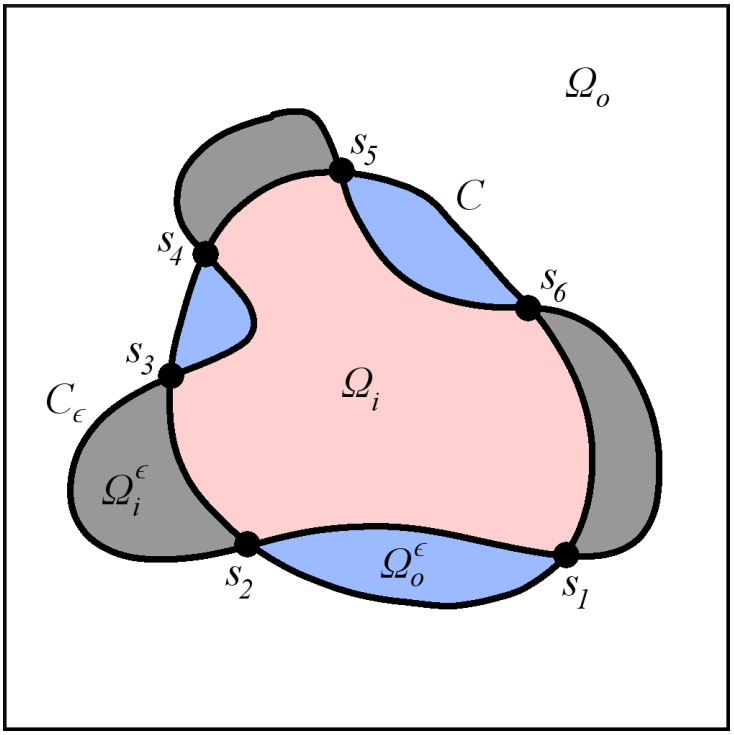


Figure 1: The variation C_ϵ of the contour C.

Here, Ω_i^ϵ is the intersection of the inside area of C_ϵ and the outside area of C and Ω_o^ϵ is the intersection of the outside area of C_ϵ and the inside area of C.

The set of intersection points of C and C_ϵ is denoted by S. Let S_i be the set of pairs of intersection points, such that for an element of S_i, say $s_j = (s_{j,1}, s_{j,2})$, the contour C_ϵ is outside the contour C for every $s \in [s_{j,1}, s_{j,2}]$. And S_o is the set of pairs of intersection points, such that the contour C_ϵ is inside the contour C for every $s \in [s_{j,1}, s_{j,2}]$. For the example in Figure 1 that would be $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$, $S_i = \{(s_2, s_3), (s_4, s_5), (s_6, s_1)\}$ and $S_o = \{(s_1, s_2), (s_3, s_4), (s_5, s_6)\}$.

The variation C_ϵ is for all positive piecewise smooth functions v, such as in Figure 2.

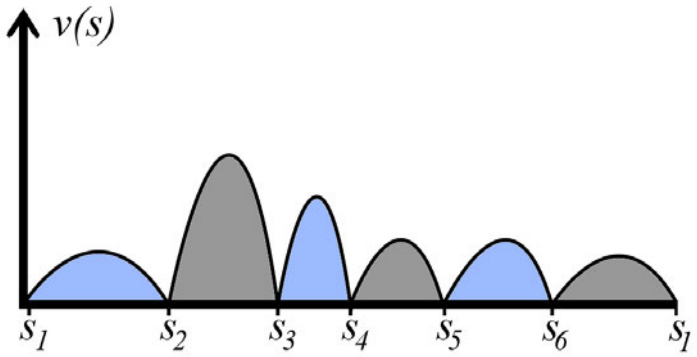


Figure 2: Positive piecewise smooth function v.

For the variation C_ϵ we have to find an expression. The contour C is given by $\boldsymbol{r}(s)$ and the variation of the contour $C_\epsilon = C + \widetilde{C}_\epsilon$ is described by the following representation:

$$\boldsymbol{r}_\epsilon(s) = \boldsymbol{r}(s) + \epsilon v(s)\widetilde{\mathbf{n}}(s), \quad 0 \leq s < 1, \tag{8}$$

for all $\epsilon \in \mathbb{R}_{>0}$ and all positive piecewise smooth functions v. Here, $\widetilde{\mathbf{n}}$ is defined by

$$\widetilde{\mathbf{n}}(s) = \begin{cases} \mathbf{n}, & s \in [s_{j,1}, s_{j,2}] \text{ with } s_j \in S_i \\ -\mathbf{n}, & s \in [s_{j,1}, s_{j,2}] \text{ with } s_j \in S_o \end{cases}, \tag{9}$$

where $\mathbf{n}$ is the unit normal vector to the contour C.

This modified unit normal vector $\widetilde{\mathbf{n}}$ is used, in order to get a single expression for the variation C_ϵ. If we decided to use $\mathbf{n}$, we would get $\boldsymbol{r}(s) + \epsilon v(s)\mathbf{n}(s)$ and $\boldsymbol{r}(s) - \epsilon v(s)\mathbf{n}(s)$ for elements of S_i and S_o, respectively. See Figure 3. We also define a modified function $\widetilde{v}$ given by

$$\widetilde{v}(s) = \begin{cases} v, & s \in [s_{j,1}, s_{j,2}] \text{ with } s_j \in S_i \\ -v, & s \in [s_{j,1}, s_{j,2}] \text{ with } s_j \in S_o \end{cases}, \tag{10}$$

such that

$$\widetilde{v}(s)\mathbf{n}(s) = v(s)\widetilde{\mathbf{n}}(s), \quad \forall s \in [0, 1). \tag{11}$$

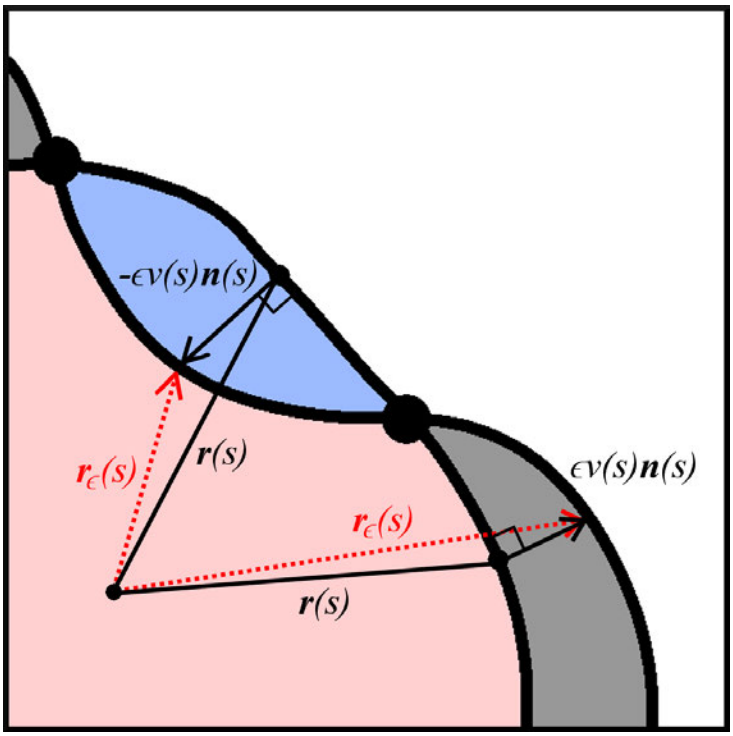


Figure 3: Vector representation of the variational contour C_ϵ.

Again, we define a new functional $\epsilon \mapsto B(\epsilon) = E(u, C + \widetilde{C}_\epsilon) = E(u, C_\epsilon)$. The functional then becomes

$$B(\epsilon) = \iint_{\Omega_i} (f-u_i)^2 dxdy + \iint_{\Omega_o} (f-u_o)^2 dxdy + \iint_{\Omega_i^\epsilon} \left[(f-u_i)^2 - (f-u_o)^2\right] dxdy + \iint_{\Omega_o^\epsilon} \left[(f-u_o)^2 - (f-u_i)^2\right] dxdy + \frac{1}{2}\nu \int_0^1 |(\boldsymbol{r} + \epsilon\widetilde{v}\mathbf{n})_s|^2 ds.$$

For the next part we introduce a coordinate transformation, since the integrals over Ω_i^ϵ and Ω_o^ϵ are difficult to evaluate with Cartesian coordinates. This coordinate transformation will only be used for the the integrals over Ω_i^ϵ and Ω_o^ϵ.

$$\begin{matrix} x(s,\xi) \\ y(s,\xi) \end{matrix} = \begin{cases} x_c(s) + \xi\widetilde{n}_x(s) \\ y_c(s) + \xi\widetilde{n}_y(s) \end{cases}, \quad s \in [0,1),\ \xi \geq 0.$$

The determinant of the Jacobian matrix is

$$\begin{aligned} \left|J(s,\xi)\right| &= \begin{vmatrix} \frac{\partial x}{\partial s} & \frac{\partial x}{\partial \xi} \\ \frac{\partial y}{\partial s} & \frac{\partial y}{\partial \xi} \end{vmatrix} = \frac{\partial x}{\partial s}\frac{\partial y}{\partial \xi} - \frac{\partial x}{\partial \xi}\frac{\partial y}{\partial s} \\ &= (x_C' + \xi\widetilde{n}_x')\widetilde{n}_y - (y_C' + \xi\widetilde{n}_y')\widetilde{n}_x. \end{aligned}$$

The area $\{(x,y) \in \Omega_{i/o}^\epsilon(s_j)\}$ transforms to $\{(s,\xi)|s_{j,1} \leq s \leq s_{j,2}, 0 \leq \xi \leq \epsilon v(s)\}$. Eventually we get the

following functional $B(\epsilon)$:

$$B(\epsilon) = \iint_{\Omega_i} (f-u_i)^2 dxdy + \iint_{\Omega_o} (f-u_o)^2 dxdy + \sum_{s_j \in S_i} \int_{s_{j,1}}^{s_{j,2}} \int_0^{\epsilon v} \left[(f-u_i)^2 - (f-u_o)^2\right] \Big|_{\boldsymbol{r}+\xi\widetilde{\mathbf{n}}} J(s,\xi) d\xi ds + \sum_{s_j \in S_o} \int_{s_{j,1}}^{s_{j,2}} \int_0^{\epsilon v} \left[(f-u_o)^2 - (f-u_i)^2\right] \Big|_{\boldsymbol{r}+\xi\widetilde{\mathbf{n}}} J(s,\xi) d\xi ds + \frac{1}{2}\nu \int_0^1 |(\boldsymbol{r}+\epsilon\widetilde{v}\mathbf{n})_s|^2 ds.$$

First we will look at the first part of the functional $B_1(\epsilon)$, which consists of the first four integrals. We begin by taking the derivative with respect to ϵ;

$$\frac{d}{d\epsilon}B_1(\epsilon) = \sum_{s_j \in S_i} \int_{s_{j,1}}^{s_{j,2}} \frac{d}{d\epsilon} \int_0^{\epsilon v} \left[(f-u_i)^2 - (f-u_o)^2\right] \Big|_{\boldsymbol{r}+\xi\widetilde{\mathbf{n}}} J(s,\xi) d\xi ds + \sum_{s_j \in S_o} \int_{s_{j,1}}^{s_{j,2}} \frac{d}{d\epsilon} \int_0^{\epsilon v} \left[(f-u_o)^2 - (f-u_i)^2\right] \Big|_{\boldsymbol{r}+\xi\widetilde{\mathbf{n}}} J(s,\xi) d\xi ds.$$

For the first two integrals we use the Leibniz's rule.

$$\frac{d}{d\epsilon}B_1(\epsilon) = \sum_{s_j \in S_i} \int_{s_{j,1}}^{s_{j,2}} \left[(f-u_i)^2 - (f-u_o)^2\right] \Big|_{\boldsymbol{r}+\epsilon v\widetilde{\mathbf{n}}} J(s,\xi)\Big|_{\xi=\epsilon v} v(s) ds + \sum_{s_j \in S_o} \int_{s_{j,1}}^{s_{j,2}} \left[(f-u_o)^2 - (f-u_i)^2\right] \Big|_{\boldsymbol{r}+\epsilon v\widetilde{\mathbf{n}}} J(s,\xi)\Big|_{\xi=\epsilon v} v(s) ds.$$

The function inside the integral is evaluated at $\xi = \epsilon v$. For the first integral, that is

$$\int_{s_{j,1}}^{s_{j,2}} \left[(f-u_i)^2 - (f-u_o)^2\right] \Big|_{\boldsymbol{r}+\epsilon v\widetilde{\mathbf{n}}} J(s,\xi)\Big|_{\xi=\epsilon v} v(s) ds = \int_{s_{j,1}}^{s_{j,2}} \left[(f-u_i)^2 - (f-u_o)^2\right] \Big|_{\boldsymbol{r}+\epsilon v\widetilde{\mathbf{n}}} \left[(x'_C + \epsilon v\widetilde{n}'_x)\widetilde{n}_y - (y'_C + \epsilon v\widetilde{n}'_y)\widetilde{n}_x\right] v(s) ds.$$

This can also be done for the second integral.

We assumed that (u, C) is a minimizer of E. We must have $E(u, C) = B(0) \leq B(\epsilon) = E(u, C_\epsilon = C + \widetilde{C}_\epsilon)$ for all real valued function $v(s)$ and all $\epsilon > 0$. Therefore we impose $B'(0) = 0$. We write this as

$$\frac{d}{d\epsilon}B(\epsilon)\Big|_{\epsilon=0} = 0. \tag{12}$$

For $\epsilon = 0$ the functional B_1 becomes

$$\frac{d}{d\epsilon}B_1(\epsilon)\Big|_{\epsilon=0} = \sum_{s_j \in S_i} \int_{s_{j,1}}^{s_{j,2}} \left[(f-u_i)^2 - (f-u_o)^2\right] \Big|_{\boldsymbol{r}} \left[x'_C\widetilde{n}_y - y'_C\widetilde{n}_x\right] v(s) ds + \sum_{s_j \in S_o} \int_{s_{j,1}}^{s_{j,2}} \left[(f-u_o)^2 - (f-u_i)^2\right] \Big|_{\boldsymbol{r}} \left[x'_C\widetilde{n}_y - y'_C\widetilde{n}_x\right] v(s) ds.$$

Now we start using the unit normal vector $\mathbf{n}$ to the contour C again instead of $\widetilde{\mathbf{n}}$ from eq. (9). For $s_j \in S_i$ we have $\mathbf{n} = \widetilde{\mathbf{n}}$ and for $s_j \in S_o$ we have $\mathbf{n} = -\widetilde{\mathbf{n}}$, with $\boldsymbol{n} = \begin{pmatrix} n_x \\ n_y \end{pmatrix}$. The integrals can be added together, which results in

$$\frac{d}{d\epsilon}B_1(\epsilon)\bigg|_{\epsilon=0} = \sum_{s_j \in S_i \cup S_o} \int_{s_{j,1}}^{s_{j,2}} \left[(f-u_i)^2 - (f-u_o)^2\right] \bigg|_{\boldsymbol{r}} \begin{pmatrix} -y_c' \\ x_c' \end{pmatrix} \cdot \mathbf{n} v(s) ds.$$

The summation is now over all elements of S_i and S_o, this means that the integration is over all $s \in [0,1)$. Thus, instead of integrating over small parts of the domain $[0,1)$, we can simply integrate from 0 to 1;

$$\frac{d}{d\epsilon}B_1(\epsilon)\bigg|_{\epsilon=0} = \int_0^1 \left[(f-u_i)^2 - (f-u_o)^2\right] \bigg|_{\boldsymbol{r}} \begin{pmatrix} -y_c' \\ x_c' \end{pmatrix} \cdot \mathbf{n} v(s) ds.$$

Now we look at the remaining part of the functional $B_2(\epsilon) = \frac{1}{2}\nu \int_0^1 |(\boldsymbol{r} + \epsilon \widetilde{v}\mathbf{n})_s|^2 ds$. This can be rewritten as

$$B_2(\epsilon) = \frac{1}{2}\nu \int_0^1 \left(\left| \frac{d\boldsymbol{r}}{ds} \right|^2 + 2\epsilon \frac{d\boldsymbol{r}}{ds} \cdot \frac{d(\widetilde{v}\mathbf{n})}{ds} + \epsilon^2 \left| \frac{d(\widetilde{v}\mathbf{n})}{ds} \right|^2 \right) ds.$$

Then, we take the derivative with respect to ϵ;

$$\frac{d}{d\epsilon}B_2(\epsilon) = \frac{1}{2}\nu \int_0^1 \left(2\frac{d\boldsymbol{r}}{ds} \cdot \frac{d(\widetilde{v}\mathbf{n})}{ds} + 2\epsilon \left| \frac{d(\widetilde{v}\mathbf{n})}{ds} \right|^2 \right) ds.$$

We can also apply eq. (12) to the remaining part of the functional B_2,

$$\begin{aligned} \frac{d}{d\epsilon}B_2(\epsilon)\bigg|_{\epsilon=0} &= \nu \int_0^1 \frac{d\boldsymbol{r}}{ds} \cdot \frac{d(\widetilde{v}\mathbf{n})}{ds} ds \\ &= \frac{d\boldsymbol{r}}{ds} \cdot \widetilde{v}\mathbf{n}\bigg|_0^1 - \nu \int_0^1 \frac{d^2\boldsymbol{r}}{ds^2} \cdot \widetilde{v}\mathbf{n} ds. \end{aligned}$$

The contour C and its variation C_ϵ are both continuous curves, and therefore the following must hold:

$$\begin{aligned} \widetilde{v}\mathbf{n}(0) &= \widetilde{v}\mathbf{n}(1), \\ \frac{d\boldsymbol{r}}{ds}\bigg|_{s=0} &= \frac{d\boldsymbol{r}}{ds}\bigg|_{s=1}. \end{aligned}$$

This results in $\frac{d\boldsymbol{r}}{ds} \cdot \widetilde{v}\mathbf{n}\Big|_0^1 = 0$ and then we get

$$\frac{d}{d\epsilon}B_2(\epsilon)\bigg|_{\epsilon=0} = -\nu \int_0^1 \frac{d^2\boldsymbol{r}}{ds^2} \cdot \widetilde{v}\mathbf{n} ds.$$

Finally we put the two parts of the functional back together $B = B_1 + B_2$. This gives us

$$\begin{aligned} \frac{d}{d\epsilon}B(\epsilon)\bigg|_{\epsilon=0} &= \int_0^1 \left[(f-u_i)^2 - (f-u_o)^2\right] \bigg|_{\boldsymbol{r}} \begin{pmatrix} -y_c' \\ x_c' \end{pmatrix} \cdot v(s)\mathbf{n} ds + -\nu \int_0^1 \frac{d^2\boldsymbol{r}}{ds^2} \cdot \widetilde{v}(s)\mathbf{n} ds \\ &= \int_0^1 \left(\left[(f-u_i)^2 - (f-u_o)^2\right] \bigg|_{\boldsymbol{r}} \begin{pmatrix} -y_c' \\ x_c' \end{pmatrix} v(s) - \nu \frac{d^2\boldsymbol{r}}{ds^2} \widetilde{v}(s) \right) \cdot \mathbf{n} ds = 0. \end{aligned}$$

This is equal to 0, because $\frac{d}{d\epsilon}B(0) = 0$. Since equation (13) must hold for every $v(s)$ and $\widetilde{v}(s)$, we can conclude that

$$\left[(f-u_i)^2 - (f-u_o)^2\right] \bigg|_{\boldsymbol{r}} \begin{pmatrix} -y_c' \\ x_c' \end{pmatrix} - \nu \frac{d^2\boldsymbol{r}}{ds^2} = 0.$$

We write

$$\mathbf{n}_C = \begin{pmatrix} -y_c' \\ x_c' \end{pmatrix}, \tag{13}$$

where y_c' and x_c' are evaluated at the original contour C. This gives us the equation

$$\left[(f-u_i)^2-(f-u_o)^2\right]\Big|_{\boldsymbol{r}}\mathbf{n}_C-\nu\frac{d^2\boldsymbol{r}}{ds^2}=0. \tag{14}$$

This is the Euler-Lagrange equation of the minimization problem. In contrast to the original functional, this equation can be used to actually find the contour C.

3.3 Energy Minimization by Gradient Descent

The Euler-Lagrange equation is time independent, but since we want to iteratively minimize the energy with respect to the contour C and the approximation function u, it is useful to make it time dependent by using an artificial time parameter. The minimization problem can be solved by gradient descent, leading to the following equation

$$\frac{\partial\boldsymbol{r}}{\partial t}=-\frac{\partial E}{\partial\boldsymbol{r}}=[e^+(s,t)-e^-(s,t)]\mathbf{n}_C(s,t)+\nu\frac{\partial^2\boldsymbol{r}}{\partial s^2}(s,t). \tag{15}$$

The terms e^+ and e^- denote the energy densities

$$e^+=(f-u_o)^2,\quad e^-=(f-u_i)^2$$

on the contour, and $\mathbf{n}_C$ is the outer normal vector given by eq. (13). This gives us an equation which we can use to find the optimal contour. Equation (15) consists of two parts; an external force normal to contour C which moves the contour to the boundary, and an internal force which discourages stretching. The internal force is needed to make the contour smooth. The external force is given by

$$\boldsymbol{F}=[e^+-e^-]\mathbf{n}_C.$$

The behavior of this force is made clear in the next part. Let us consider an example with a given gray-scale image $f:\Omega\to[0,1]$, an approximation function u and a contour C at a certain time step. The values of u_i and u_o are evaluated by computing the mean value of f of the area inside and outside the contour, respectively. For this example we take the values $u_i=0.6$ and $u_o=0.2$. See Figure 4.

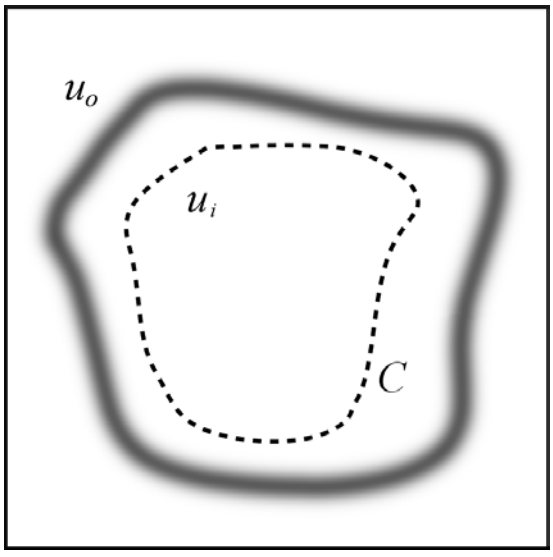


Figure 4: Image of a cell with contour C.

The contour has not reached the cell boundary yet. Let us compute the external force for a point on the contour with value $f=0.6$:

$$\begin{aligned}\boldsymbol{F}&=\left[(f-u_o)^2-(f-u_i)^2\right]\mathbf{n}_C\\&=\left[(0.6-0.2)^2-(0.6-0.6)^2\right]\mathbf{n}_C\\&=0.16\mathbf{n}_C.\end{aligned}$$

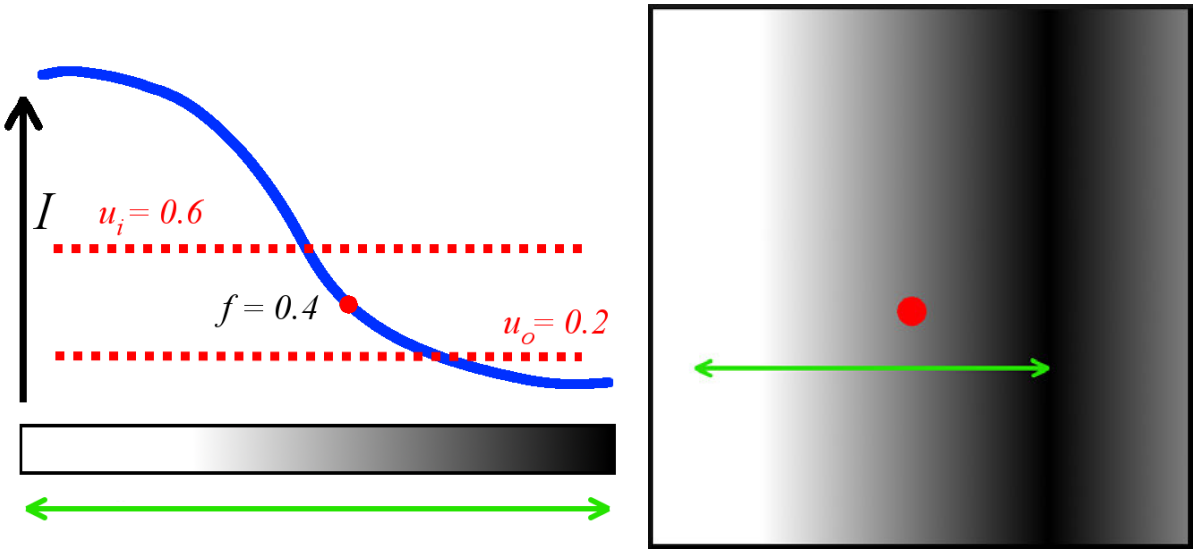
The force is positive, in the direction of the normal vector. This causes the contour to move towards the boundary.

In each time step we determine the values of u_i and u_o, such that the external force can be computed. We also determine the internal force. These two forces makes the contour move.

If a point on the contour reaches the boundary, the intensity of that points will decrease. Consider the following situation where $f = 0.4$ for a certain point. See figure 5. The external force then becomes:

$$\begin{aligned} \boldsymbol{F} &= \left[(f-u_o)^2 - (f-u_i)^2\right]\mathbf{n}_C \\ &= \left[(0.4-0.2)^2 - (0.4-0.6)^2\right]\mathbf{n}_C \\ &= \mathbf{0}. \end{aligned}$$

We find that the external force is $\mathbf{0}$. Thus, if the value of a point is halfway between u_i and u_o, then there is no external force.



(a) The intensity of f, u_i and u_o.

(b) Point inside a cell near the boundary.

Figure 5: The intensity is given by the blue line. The external force at this point is $\mathbf{0}$.

Throughout this report we will use an additional internal force; the bending force. This force prevents the contour from bending too much, its weighting parameter is denoted by κ. The equation with this bending force is given by:

$$\frac{\partial \boldsymbol{r}}{\partial t} = -\frac{\partial E}{\partial \boldsymbol{r}} = [e^+ - e^-]\mathbf{n}_C + \nu\frac{\partial^2 \boldsymbol{r}}{\partial s^2} - \kappa\frac{\partial^4 \boldsymbol{r}}{\partial s^4}. \tag{16}$$

Since this is an improvement to the modified Mumford-Shah model, this will be discussed in Section 6.1.

4 Numerical Methods

In the previous section we derived a equation which we can numerically solve by using an artificial time parameter. In this section we will discuss the numerical methods used to solve the problem. First we will discretize the original problem in space, then we will dicretize the obtained system in time.

4.1 Space Discretization

Equation (15) needs to be discretized in space. For this discretization we use Finite Difference. The contour has periodic boundary conditions and the contour $\boldsymbol{r}$ is equal to the initial contour $\boldsymbol{r}_0$ at $t = 0$. The equation with boundary and initial conditions then becomes

$$(P)\begin{cases} \boldsymbol{r}_t = [e^+(s,t) - e^-(s,t)]\mathbf{n}_C(s,t) + \nu \boldsymbol{r}_{ss}(s,t), & t > 0, s \in [0,1) \\ \boldsymbol{r}(0,t) = \boldsymbol{r}(1,t), & t > 0 \\ \boldsymbol{r}_s(0,t) = \boldsymbol{r}_s(1,t), & t > 0 \\ \boldsymbol{r}(s,0) = \boldsymbol{r}_0(s), & s \in [0,1) \end{cases}, \tag{17}$$

As stated, we will discretize problem (P) using Finite Difference. We will use N equidistant unknowns for s. We decide to scale the values of s, such that the distance between s_i and s_{i+1} is equal to 1 for every i. Then we get $s_i = i$ for $i = 1, 2, ..., N$ and with $h = \Delta s = 1$. We will derive the Finite Difference equation for an internal node, and the two edge nodes for $i = 1$ and $i = N$.

Since the outer normal vector $\mathbf{n}_C$ is of the form $\begin{pmatrix} f(y) \\ f(x) \end{pmatrix}$, we will get a mixed system for the x and y-component. Therefore it is convenient to divide the equation into two part; the x-component and the y-component for $\boldsymbol{r} = \begin{pmatrix} x \\ y \end{pmatrix}$. We will discretize these two components separately and put these at the end together. We rewrite problem (P) as

$$\begin{cases} x_t = [e^+(s,t) - e^-(s,t)](-y') + \nu x_{ss}(s,t), & t > 0, s \in [0,1] \\ y_t = [e^+(s,t) - e^-(s,t)]x' + \nu y_{ss}(s,t), & t > 0, s \in [0,1] \end{cases}, \tag{18}$$

For this part we only consider the equation for the x-component of $\boldsymbol{r}$ and afterwards we will also apply this to the y-component. First we look at the internal force term x_{ss}, where the discretized value of x is denoted by $\widetilde{x}$. This is discretized using central difference.

$$f_i'' = \frac{1}{h^2}(f_{i-1} - 2f_i + f_{i+1}) + O(h^2). \tag{19}$$

In our case we have $h = \Delta s = 1$, thus for an internal node $i = 2, ..., N-1$ we have

$$\widetilde{x}_i'' = \widetilde{x}_{i-1} - 2\widetilde{x}_i + \widetilde{x}_{i+1}.$$

For the edge nodes, we introduce two virtual points s_0 and s_{N+1}. Since $\boldsymbol{r}$ is periodic, we have $\boldsymbol{r}(s_0) = \boldsymbol{r}(s_N)$ and $\boldsymbol{r}(s_{N+1}) = \boldsymbol{r}(s_1)$. Consider $i = 1$, then we get

$$\begin{aligned} \widetilde{x}_1'' &= \widetilde{x}_0 - 2\widetilde{x}_1 + \widetilde{x}_2 \\ &= \widetilde{x}_N - 2\widetilde{x}_1 + \widetilde{x}_2, \end{aligned}$$

because $x(s_0) = x(s_N)$. For $i = N$, we have

$$\begin{aligned} \widetilde{x}_N'' &= \widetilde{x}_{N-1} - 2\widetilde{x}_N + \widetilde{x}_{N+1} \\ &= \widetilde{x}_{N-1} - 2\widetilde{x}_N + \widetilde{x}_1, \end{aligned}$$

because $x(s_{N+1}) = x(s_1)$. The discretization of $\boldsymbol{x}_{ss}$ is $\boldsymbol{x}_{ss} = A\boldsymbol{x}$, where A is the matrix

$$A = \begin{pmatrix} -2 & 1 & & & 1 \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ 1 & & & 1 & -2 \end{pmatrix}. \tag{20}$$

For the discretization of $-y'$ we also use central difference. In this way, the order of both numerical schemes is 2. The numerical scheme for the first derivative is as follows:

$$f_i' = \frac{1}{2h}(f_{i+1} - f_{i-1}) + O(h^2). \tag{21}$$

For an internal point s_i, this is

$$-\widetilde{y}_i' = \frac{1}{2}(\widetilde{y}_{i-1} - \widetilde{y}_{i+1}).$$

For the edge nodes we use the virtual points again. Then we get:

$$\widetilde{y}_1 = \frac{1}{2}(\widetilde{y}_N - \widetilde{y}_2),$$

and

$$\widetilde{y}_N = \frac{1}{2}(\widetilde{y}_{N-1} - \widetilde{y}_1).$$

This results in the discretization $\widetilde{\boldsymbol{y}}' = B\widetilde{\boldsymbol{y}}$, where B is the matrix

$$B = \begin{pmatrix} & -1 & & & 1 \\ 1 & & -1 & & \\ & \ddots & & \ddots & \\ & & 1 & & -1 \\ -1 & & & 1 & \end{pmatrix}. \tag{22}$$

Now we look at the factor $[e^+(s,t) - e^-(s,t)]$. For a point s_i, this is

$$(e^+ - e^-)_i = [(f(x(s_i), y(s_i)) - u_o)^2) - f(x(s_i), y(s_i)) - u_i)^2],$$

where u_o and u_i are defined as in eq. (6). If we put these values in a diagonal matrix, we get

$$E = \begin{pmatrix} (e^+ - e^-)_1 & & & & \\ & (e^+ - e^-)_2 & & & \\ & & \ddots & & \\ & & & (e^+ - e^-)_{n-1} & \\ & & & & (e^+ - e^-)_n \end{pmatrix}$$

The total space discretization of the x-component is

$$\frac{d\widetilde{\boldsymbol{x}}}{dt} = EB\widetilde{\boldsymbol{x}} + \nu A\widetilde{\boldsymbol{x}} \tag{23}$$

The only part of the discretization of the y-component that differs form the x-component is the first derivative x'. For x' we have the discretization $\widetilde{\boldsymbol{x}}' = C\widetilde{\boldsymbol{x}}$, where $C = -B$. The total discretization of the y-component is

$$\frac{d\widetilde{\boldsymbol{y}}}{dt} = EC\widetilde{\boldsymbol{y}} + \nu A\widetilde{\boldsymbol{y}}. \tag{24}$$

Combining eq. (23) and (24) yields;

$$\frac{d}{dt}\begin{pmatrix}\widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}}\end{pmatrix} = \left(\begin{array}{c|c} \nu A & EB \\ \hline EC & \nu A \end{array}\right)\begin{pmatrix}\widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}}\end{pmatrix}. \tag{25}$$

This is the obtained system after discretizing problem (P) in space.

4.2 Time Discretization

In the previous section we discretized the problem in space. The next step is to discretize the system in time. In this section we will discuss two different time integration methods; Forward Euler and Backward Euler. We will compare the results of both methods for different time steps. For this we take a small image where we want to find the boundary of one cell. This can be seen in Figure 6. The contour has 30 points, and the simulation starts with a given initial contour. In this section, a bending force is already applied, which will be introduced later in Section 6.1.

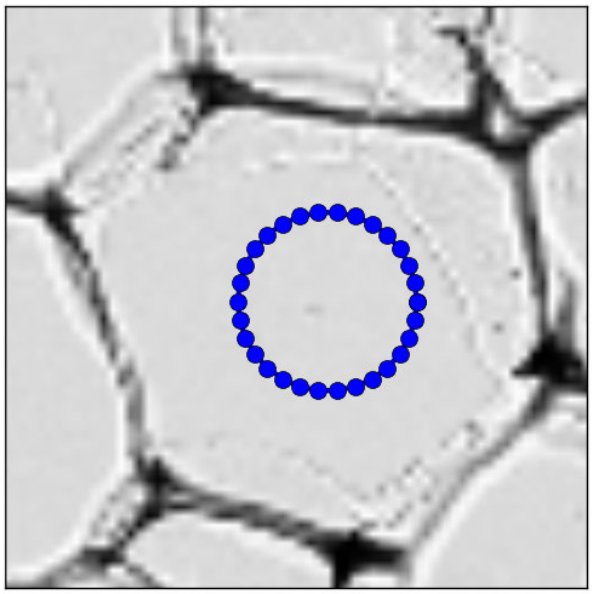


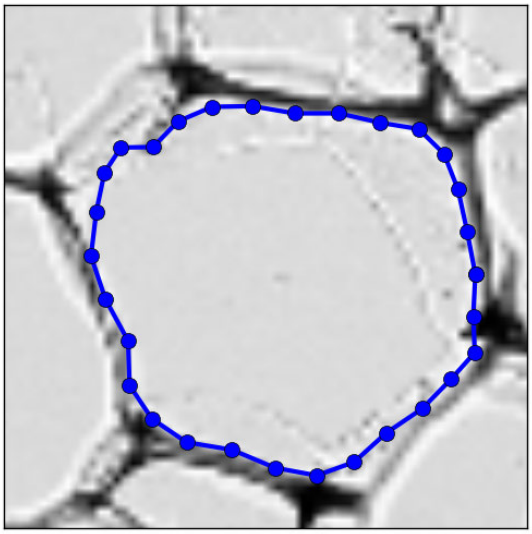
Figure 6: Image with one cell and a initial contour.

4.2.1 Forward Euler

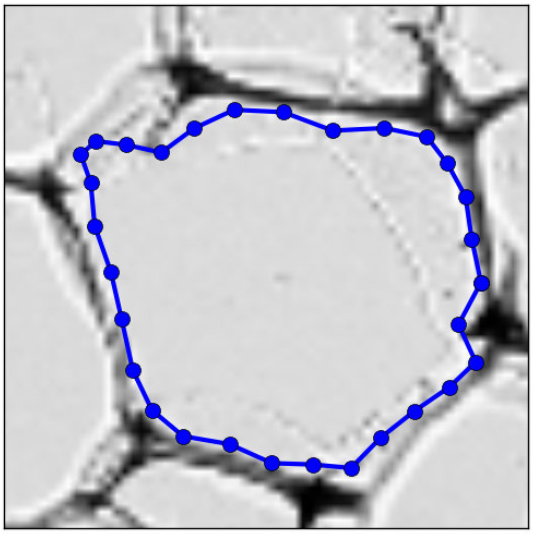
First we will discretize in time using Forward Euler. That is, given the solution $\widetilde{\boldsymbol{r}}^k = \begin{pmatrix} \widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}} \end{pmatrix}^k$ at time $t_k = k\Delta t$, the solution at time t_{k+1} can be found by computing the following sytem

$$\begin{pmatrix} \widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}} \end{pmatrix}^{k+1} = \begin{pmatrix} \widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}} \end{pmatrix}^k + \Delta t \left(\begin{array}{c|c} \nu A & EB \\ \hline EC & \nu A \end{array} \right) \begin{pmatrix} \widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}} \end{pmatrix}^k. \tag{26}$$

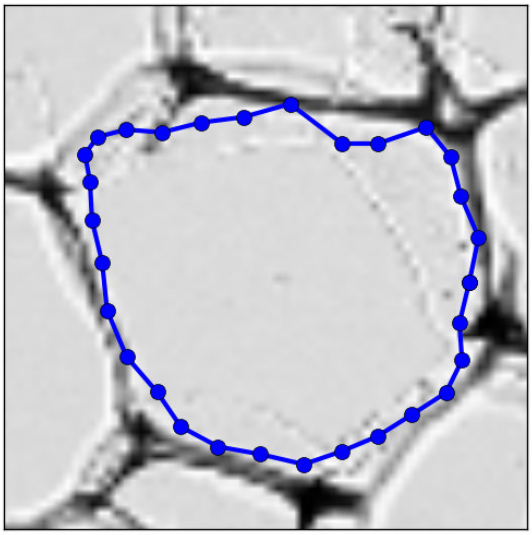
We simulate this method for different time steps to see if this method works correctly for our model. Since Forward Euler is not conditionally stable, not every choice of time step Δt leads to a good result. However, we only have to perform a matrix multiplication for this method, which has a small computational time. This is an advantage of Forward Euler. We simulate this method for one cell for different time steps. The results can be seen in Figure 7.For points of the contour we use negative orientation, such that the normal vector is pointed outwards.



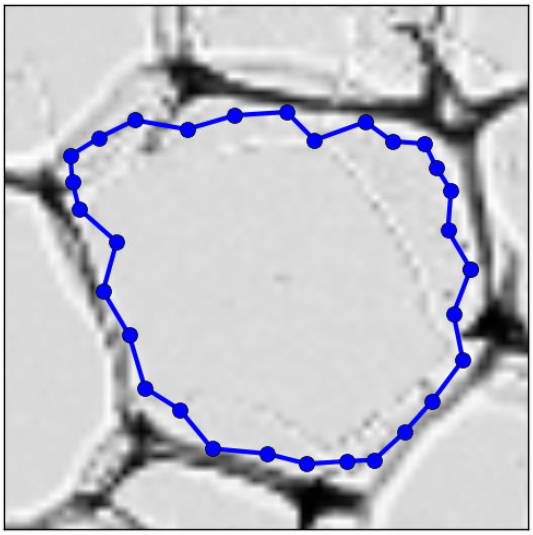
(a) $\Delta t = 1.0$ and steps $= 10000$.



(b) $\Delta t = 5.0$ and steps $= 2000$.



(c) $\Delta t = 8.0$ and steps $= 1250$.



(d) $\Delta t = 10.0$ and steps $= 1000$.

Figure 7: Mumford-Shah model with Forward Euler.

As expected, this method is not stable for every Δt. For $\Delta t = 10.0$ the contour is not smooth anymore, but this method still works quite well for $\Delta t = 5.0$. The method gives different results for different time steps Δt, even though $t_{end} = \Delta t * steps = 10000$ is the same for each simulation. This can be explained by the fact that the method is not linear.

The execution time of the simulations can be seen in the Table 1.

Δt	Time
1.0	5.98 s
5.0	3.52 s
8.0	1.05 s
10.0	0.97 s

Table 1: Execution time of Forward Euler for different time steps Δt.

The execution time is very large. Eventually we want to find the boundaries of all cells in a full image, which contains about 500 cells. If we use Forward Euler with $\Delta t = 5.0$, we will have a total execution time of 30 minutes. This is of course not acceptable, so we have to find a way to solve this problem.

4.2.2 Backward Euler

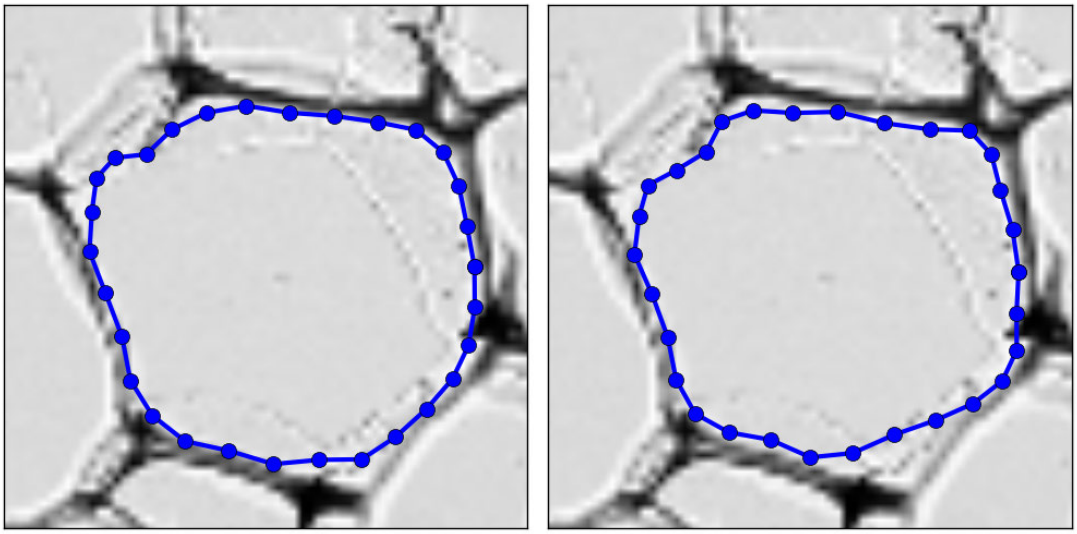
Secondly we will discretize in time using Backward Euler. That is, given the solution $\widetilde{\boldsymbol{r}}^k$ at time $t_k = k\Delta t$, the solution at time t_{k+1} can be found by solving the following system for $\widetilde{\boldsymbol{r}}^{k+1}$;

$$\begin{pmatrix} \widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}} \end{pmatrix}^{k+1} = \begin{pmatrix} \widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}} \end{pmatrix}^{k} + \Delta t \left(\begin{array}{c|c} \nu A & EB \\ \hline EC & \nu A \end{array} \right) \begin{pmatrix} \widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}} \end{pmatrix}^{k+1}. \tag{27}$$

We can rewrite this as

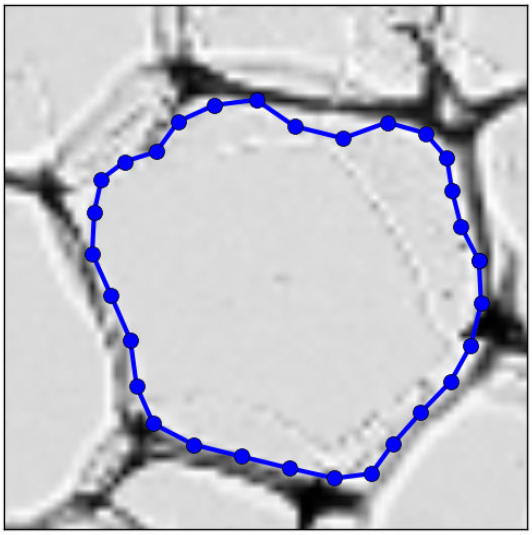
$$\begin{pmatrix} \widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}} \end{pmatrix}^{k+1} = \left(I_{2N} - \Delta t \left(\begin{array}{c|c} \nu A & EB \\ \hline EC & \nu A \end{array} \right) \right)^{-1} \begin{pmatrix} \widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}} \end{pmatrix}^{k}, \tag{28}$$

where I_{2N} is the $2N \times 2N$ identity matrix. For this method we have to compute an inverse matrix. Unfortunately, this matrix is not constant in time, since we have to update E in each step. This increases the computational time, but since Backward Euler is unconditionally stable, we can take larger time steps than with Forward Euler. We also simulate this method for one cell for different time steps. The results are shown in Figure 8.

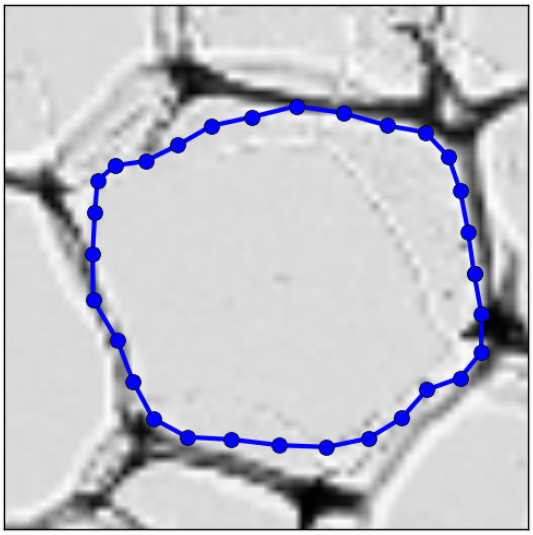


(a) $\Delta t = 1.0$ and steps $= 10000$.

(b) $\Delta t = 5.0$ and steps $= 2000$.



(c) $\Delta t = 10.0$ and steps $= 1000$.



(d) $\Delta t = 15.0$ and steps $= 667$.

Figure 8: Mumford-Shah model with Backward Euler.

As expected, we find that Backward Euler behaves more stable, which leads to smoother contours. For $\Delta t = 10.0$ we still have an acceptable result in contrast to Forward Euler. The execution time of the simulations can be seen in Table 2.

Δt	Time
1.0	15.94 s
5.0	3.5 s
10.0	1.87 s
15.0	1.43 s

Table 2: Execution time of Backward Euler for different time steps Δt.

The execution time for Backward Euler is larger than for Forward Euler, but since we can take a larger time step $\Delta t = 10.0$, this method is preferred over Forward Euler.

5 Image Filtering

5.1 Locating the Center of a Cell

For this modified Mumford-Shah model a initial contour is needed to find the boundary of the cells. The initial contour has to be a good choice in order for the method to work properly. A suitable choice would be inside the cells. Therefore we need to locate the centers of the cells. In this section we will discuss a method to find the centers.

5.1.1 Cell Center Method

The Cell Center methods finds the centers of the cell. This methods consists of the following steps;

- Applying Gaussian blur
- Finding local maxima
- Dividing the local maxima in slices
- Calculating the center of each slice

First we will apply an Gaussian blur filter [5] defined by $\widetilde{I}(x,y) = G_\sigma(x,y) * I(x,y)$, where G_σ is the two-dimensional Gaussian distribution

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-[x^2+y^2]}{2\sigma^2}}, \tag{29}$$

and $I(x,y)$ the original image. And the operation $*$ is the convolution operation. For the 2D discrete case, this is

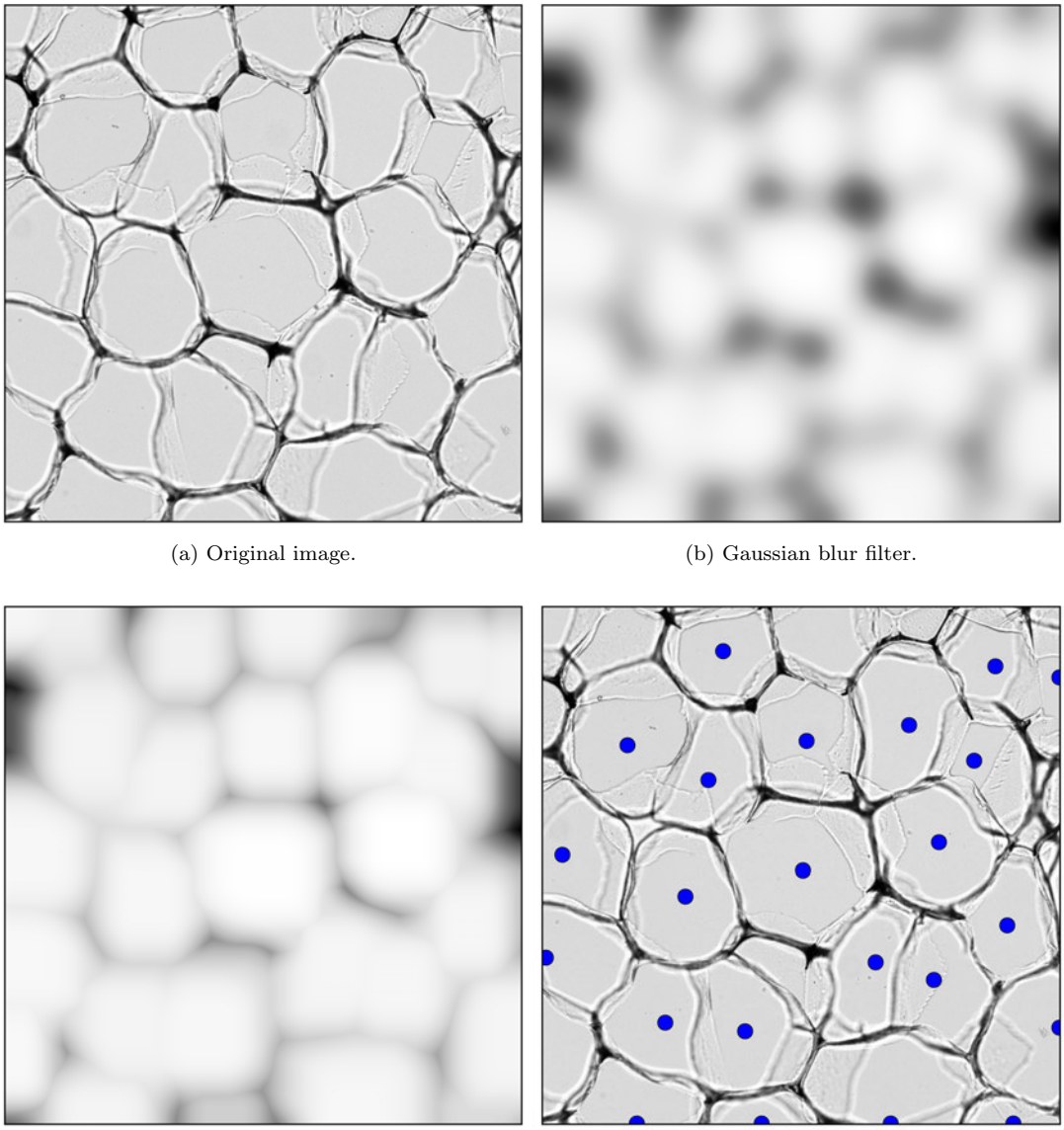
$$\widetilde{I}(x,y) = \sum_{j=0}^{m}\sum_{i=0}^{n} \frac{1}{2\pi\sigma^2} e^{\frac{-[(i-x)^2+(j-x)^2]}{2\sigma^2}} I(i,j). \tag{30}$$

However, this is not the way it is implemented. We make use of the function *ndimage.gaussian_filter* from the library *scipy*. This function uses Fast Fourier Transform [6] to compute the convolution operation much faster.

Secondly, we find the local maxima of the blurred image. For each pixel in the image we search in the neighborhood with a certain size for the maximum value. Then we set the value of the pixel to that maximum value. As a result, we have a image with many connected components containing pixels with the same values.

In the next step each connected components gets a number. And for each numbered component the center is calculated. This gives us the center of each cell in the image.

The steps of this operation are shown in Figure 9.



(a) Original image.

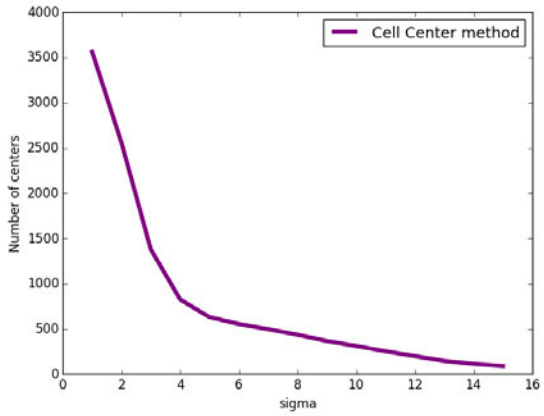
(b) Gaussian blur filter.

(c) Local maxima for each pixel.

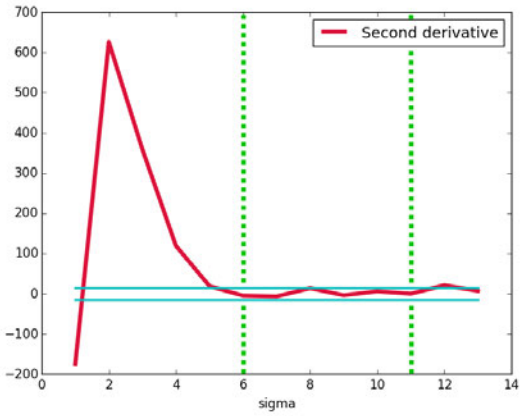
(d) Cell centers.

Figure 9: The steps of Cell Center method.

In order for this method to work correctly, we have to make a suitable choice for σ. If the parameter σ is too large, we will get a very blurred image with only a few centers, but if σ is too small, we will get many local maxima. This means that there are multiple centers located in one cell.
We applied this method for several σ's and calculated the number of centers found. The results are shown in Figure 10.



(a) Number of centers for different σ.



(b) Second derivative of the number of centers. The interval J with $M = 15$ is indicated by the dotted green lines.

Figure 10: Results of the Cell Center method for different σ.

We see that for some values of σ the number of centers does not change much. We look for an interval J such that $\forall \sigma \in J$ we have $\left|\frac{d^2 N_{center}}{d\sigma^2}\right| < M$, where N_{center} is the number of centers and M a positive constant scalar. The results can be seen in Figure 10(b). This only gives us an interval for an suitable choice for σ. We need a more specific approach to find the correct σ. This is done in the next section.

5.1.2 Cell Vertex Method

The Cell Center method gives a good estimate of the total number of centers for the right choice of σ. We will discuss another method to find the number of centers for a given image; the Cell Vertex method. This method does not look for centers of cells, but it uses the confluences of the boundaries of the cells. Then Euler's formula for plane graphs is used to calculate the number of cells. Since this method behaves slightly different than the Cell Center method, we can compare both methods to find the correct choice of σ.

An image is represented as a graph as follows: the confluences of the boundaries of the cells are the vertices of the graph and the boundaries are the edges between the vertices. An example of such a graph can be seen in Figure 11.

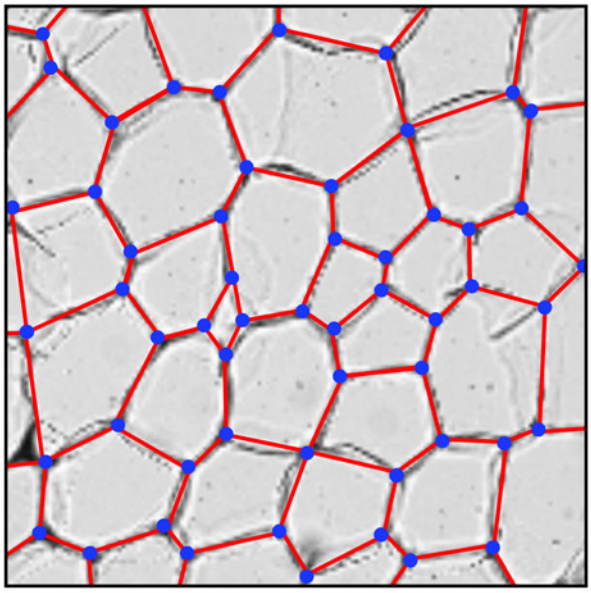
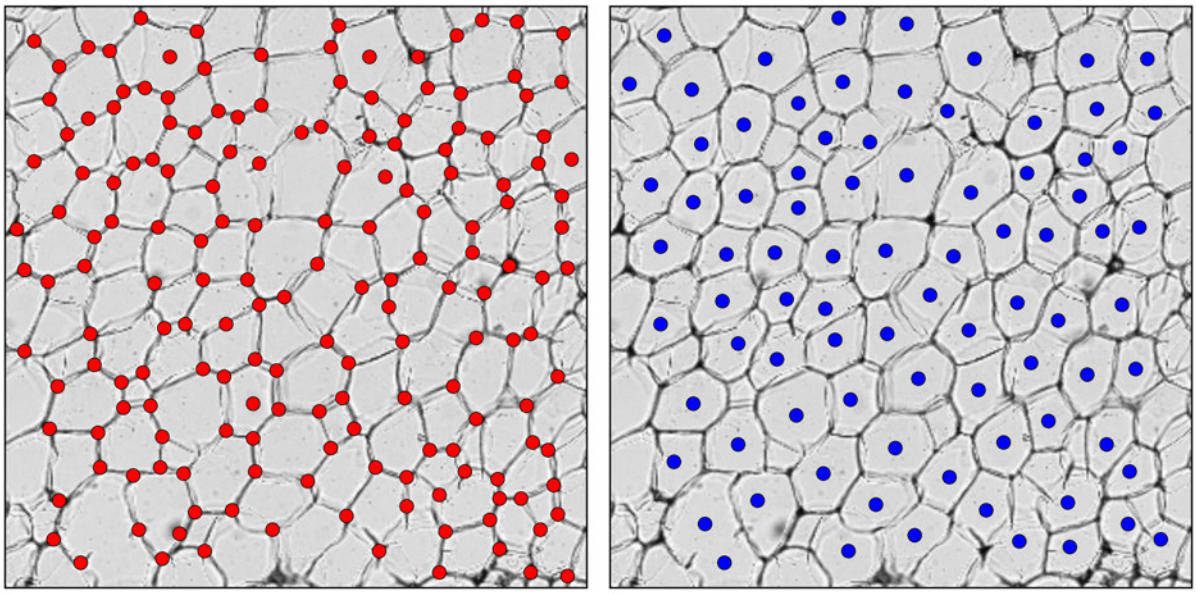


Figure 11: An image of cells represented as a graph with vertices and edges.

We have to find the location of the vertices. This method works the same as the Cell Center method, but instead of finding local maxima, this method searches for local minima. Since the diameter of the cells are about twice as large as the edges, we use $\widetilde{\sigma} = \sigma/2$ for the Gaussian blur filter where we would use σ for the Cell Center method. For the image in Figure 12 $\sigma = 7.0$ was used to locate the vertices.



(a) 171 vertices found with Cell Vertex method

(b) 84 centers found with the Cell Center method

Figure 12: Image with vertices given in red and the centers given in blue.

The graph in Figure 11 can be considered to be a 3-regular graph, meaning each vertex has 3 adjacent edges. Thus each vertex has degree 3. The graph is connected and it is also a plane graph since it can be drawn on the plane without crossing edges. Now we can use Euler's formula [7], which is stated as follows;

Theorem 1 (Euler's formula). *If G is a finite, connected plane graph with vertex set V, edge set E and face set F, then*

$$|V| - |E| + |F| = 2. \tag{31}$$

The number of vertices $|V|$ can be computed by the method discussed above and our goal is to find the number of faces $|F|$. The number of edges can be found using the Handshaking lemma [7].

Theorem 2 (Handshaking lemma). *For a graph with vertex set V and edge set E, we have*

$$\sum_{v \in V} degree(v) = 2|E|. \tag{32}$$

Since our graph is a 3-regular graph, we have $degree(v) = 3$ for each vertex v. This gives us

$$\begin{aligned} \sum_{v \in V} degree(v) &= 2|E| \\ 3|V| &= 2|E| \\ \implies |E| &= \frac{3}{2}|V|. \end{aligned}$$

If we substitute this in Euler's formula, we get

$$\begin{aligned} |V| - \frac{3}{2}|V| + |F| &= 2 \\ \implies |F| &= 2 + \frac{1}{2}|V|. \end{aligned} \tag{33}$$

We use eq. (33) to calculate the number of centers given the the number of vertices from the example in Figure 12. This gives us $|F| = 2 + \frac{1}{2}171 = 87.5 \approx 88$ faces. This is reasonably similar to the number of centers found with the Cell Center method. Then we apply this method for several σ as we did with the previous method. The results are shown in Figure 13.

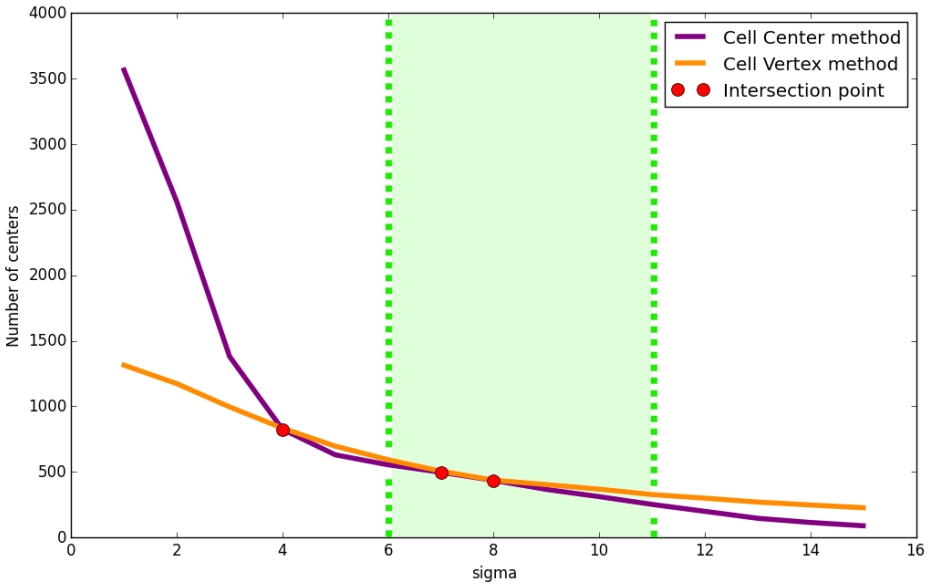


Figure 13: Number of centers found by both methods.

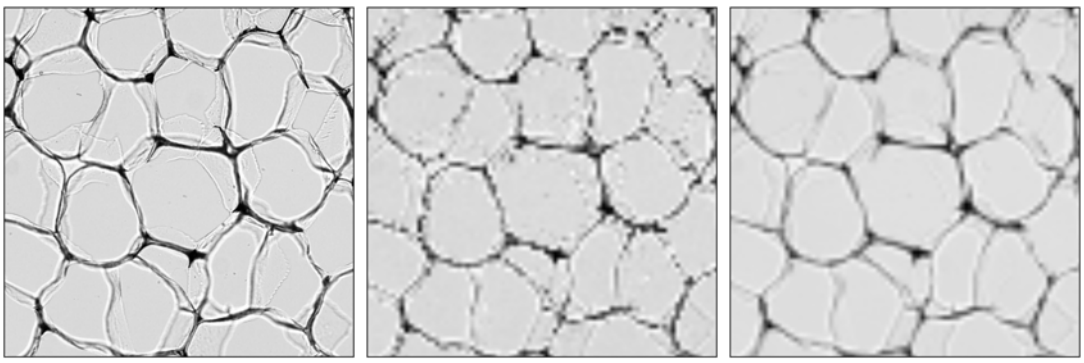
The two lines intersects at $\sigma_1 = 4.0, \sigma_2 = 7.0$ and $\sigma_3 = 8.0$. Since σ_1 lies outside the interval $J = [6.0, 11.0]$, only $\sigma_2 = 7.0$ and $\sigma_3 = 8.0$ are valid choices for σ. It is possible that we have multiple solution for σ; if this occurs, then we take the average of the solution, thus $\bar{\sigma} = 7.5$.

With this value for σ we can determine the location of the centers of the cells.

5.2 Down-sampling

The images of the cells have very high resolution. This is unnecessary and it only makes the computational time much higher. Therefore we choose to apply down-sampling on the image by only taking a fifth of the pixels in the x-direction and the y-direction. This will make the image 25 times smaller.

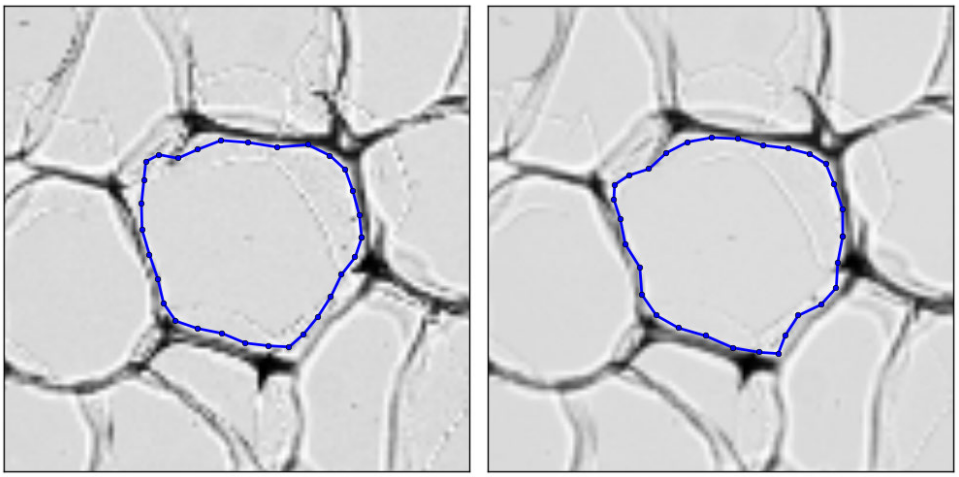
This is not the only way to down-sample the image. We can also take the mean value of small blocks of the image. We divide the image in 5×5 blocks and for each block we compute the mean value. This value becomes the intensity of one pixel in the new image. Since we take all points in consideration, the down-sampling method by taking mean values is preferred over the other method. Figure 14 shows that this way of down-sampling clearly gives a smoother result.



(a) Original image (b) Down-sampled image (c) Down-sampled image by taking mean values

Figure 14: Images of the two down-sampling methods.

We tried both down-sampling methods with the Mumford-Shah model. See Figure 15. As expected, the second method performs better than the first method.



(a) Down-sampled image (b) Down-sampled image by taking mean values

Figure 15: Mumford-Shah model with both down-sampling methods.

Since the down-sampling method by taking mean values leads to a better result, we decide to continue using this method throughout this report.

5.3 Binary Image

The main problem with our current model is that it is very time consuming. The external force in each time step is very small, thus it is required to take many steps. If we choose to scale the external force with a parameter, the method becomes instable.

The external force in the Euler-Lagrange equation is given by

$$F = [(f - u_o)^2 - (f - u_i)^2]\mathbf{n}_C,$$

where u_i and u_o are the mean value of the image of the area inside and outside the contour, respectively, and f is the intensity of the image at the contour.
Consider a point of the contour. At the beginning of the simulation u_i and f are equally large, and u_o is a little bit smaller, because Ω_o contains the boundary of the cell, which has a lower intensity than the rest of the cell. Since the boundary of the cell is only a small part of Ω_o, this does not have a large effect on u_o, resulting in a small difference between u_i and u_o. This means that the external force can be approximated by

$$\begin{aligned} \boldsymbol{F} &= [(f - u_o)^2 - (f - u_i)^2]\mathbf{n}_C \\ &\approx [(u_i - u_o)^2 - (u_i - u_i)^2]\mathbf{n}_C \\ &= [(u_i - u_o)^2]\mathbf{n}_C. \end{aligned}$$

Since $u_i - u_o$ is very small, $(u_i - u_o)^2$ is also very small. In the image from Figure 17 we have $u_i = 0.770$ and $u_o = 0.722$. This results in an external force;

$$\begin{aligned} \boldsymbol{F} &= [(u_i - u_o)^2]\mathbf{n}_C \\ &= 0.0023\mathbf{n}_C. \end{aligned}$$

With such a small force, many steps are required to reach the boundary.

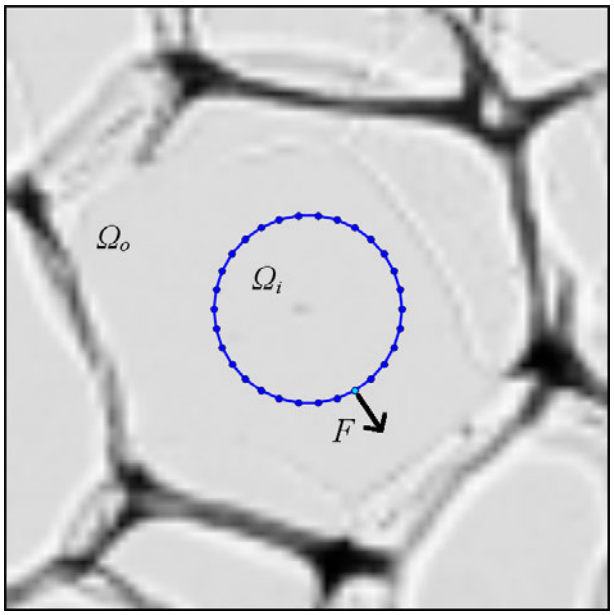


Figure 16: Initial contour of original image.

This problem can be solved by making the image binary, meaning the image only has values 0 and 1. For this operation we need a threshold, such that if the intensity of a pixel if larger than the threshold, then the pixel gets value 1, otherwise 0. The binary image I_{bin} is defined by

$$I_{bin} = \begin{cases} 1, & I(p) \geq I_{threshold} \\ 0, & I(p) < I_{threshold} \end{cases}, \tag{34}$$

for all pixels p in the image. In Section 5.4 we will discuss a method to determine $I_{threshold}$.

As a result, the difference between u_i and u_o is larger. This leads to a larger force. Now we do not have to take many steps in order to find the boundary. In the image below we have $u_i = 1.00$ and $u_o = 0.836$, which is a larger difference than in the previous example.

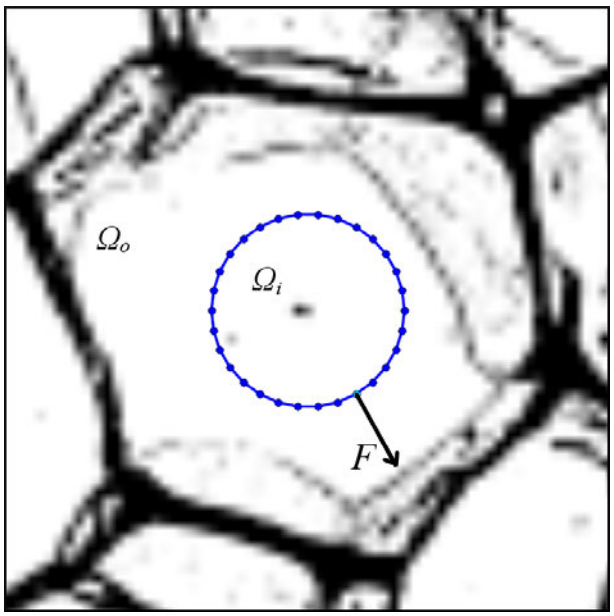
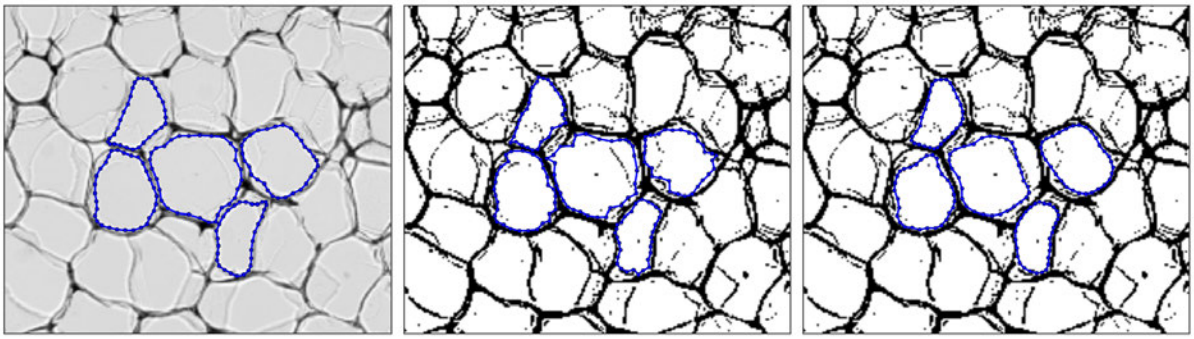


Figure 17: Initial contour of binary image.

For the original image, we need take 3000 time steps and time step $\Delta t = 5.0$. For the binary image it is sufficient to take 250 time steps, but Δt can not be to too large. We use $\Delta t = 2.0$, because larger Δt leads to instabilities, since we have a larger force. For the simulations we use $\nu = 0.001$. The results are shown in Figure 18.



(a) Original image with $\kappa = 0.01$. (b) Binary image with $\kappa = 0.01$. (c) Binary image with $\kappa = 0.05$.

Figure 18: Mumford-Shah model with Backward Euler for original and binary image.

The Mumford-Shah model seems to work quite well for the binary image, however, the contours are not as smooth as the contours of the original image. It can be made smoother by increasing the bending parameter κ, but this leads to inaccurate results, as can be seen in Figure 18(c).

We test this approach for different images. Results show that this does not work correctly for every image. See Figure 19.

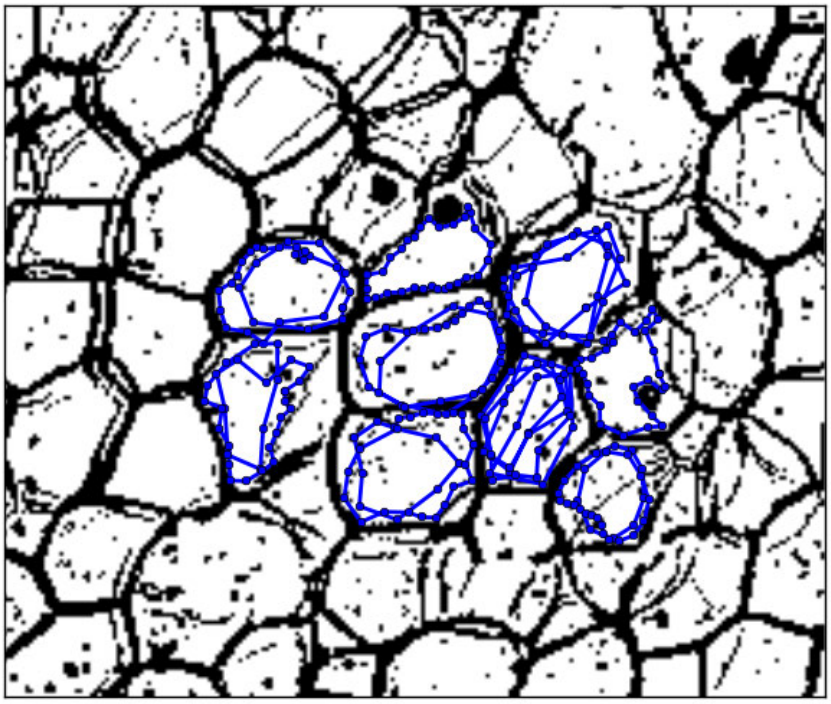


Figure 19: Mumford-Shah model with Backward Euler for a binary image.

The problem is caused by the small points inside each cell. Varying the parameters does not solve the problem.

5.4 Removing Noise from Binary Images

In the previous section we found that making the image binary improves the execution time, but unfortunately this approach does not work for every image, especially when there are many black points inside each cell, which we call noise. We are looking for a way to remove those points. In this section we will discuss a method to remove unwanted black points in the binary image.

This approach consists the following steps

- Converting to binary image
- Applying Gaussian blur
- Checking the difference in intensity for each pixel
- Removing the points with a large difference in intensity
- Removing small points

First the image will be converted to an binary image, as has been done in the previous section. For the operation a threshold is needed. Then we will apply a Gaussian blur filter. The black points surrounded by a lot of white space will become a lot more gray than a black point in the boundary. This can be used to remove the noise.
Next, we will remove the points that changed a lot in intensity by setting the value of the pixel to 1 (white). In the next part we will go more into detail.

In order to make the image binary, we have to find a suitable threshold. Therefore we look at the histogram of the intensity of the image. See Figure 20.

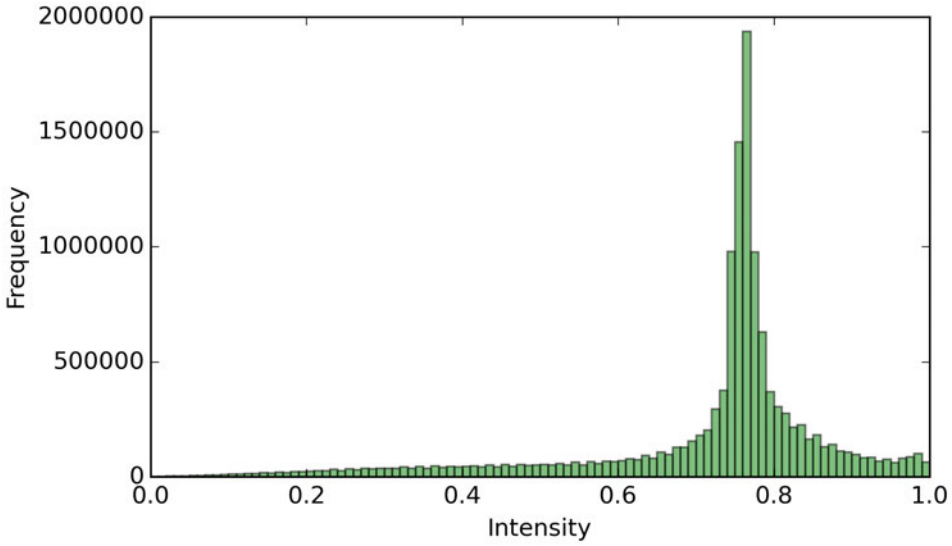


Figure 20: Histogram of the intensity of pixels in the image.

The aim is to find a threshold, such that only the light gray points will get value 1, and the dark gray points value 0. As can be seen in the histogram, there is large peak at around the intensity of 0.8. This peak represents the light gray points, which forms the inside of the cell, thus not the points from the boundary. Our goal is to remove such points. We want to find the intensity, where the peak begins. This can be achieved by looking at the acceleration of the frequency, thus the second derivative of the frequency with respect to the intensity. For the numerical calculation of the second derivative, we use first order forward difference formula:

$$f_i'' = \frac{1}{h^2}(f_{i+2} - 2f_{i+1} + f_i) + O(h). \tag{35}$$

The result can be seen in Figure 21.

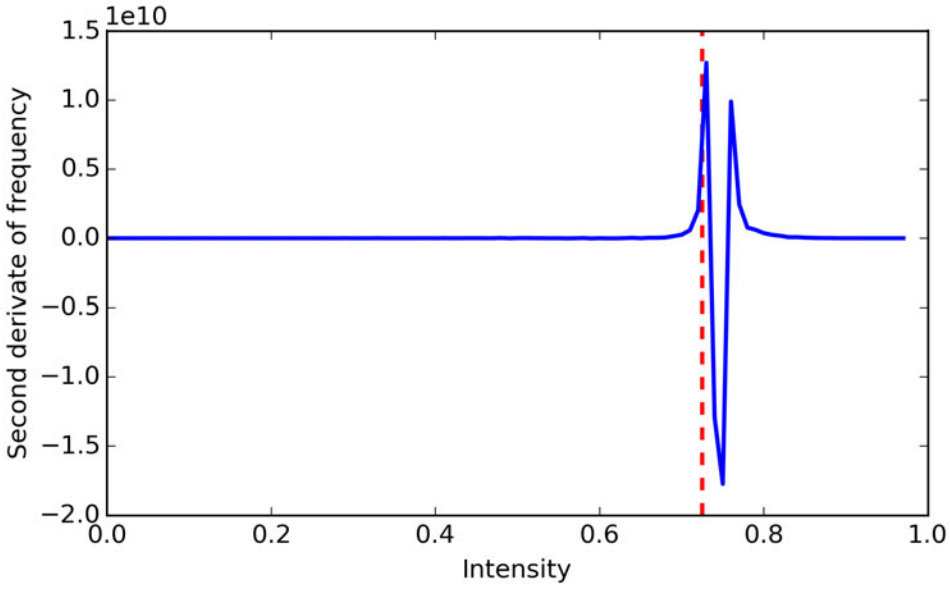


Figure 21: Second derivative of the frequency with respect to the intensity. The threshold $I_{threshold} = 0.72$ is given by the red dotted line.

We find that that the maximum is located at 0.72. Thus, we set $I_{threshold} = 0.72$. In Figure 22 a histogram is shown from an image, where we also calculated the threshold value. In the background the gray-scale of the intensity is shown to illustrate which values will be set to 0 and to 1. The pixels of the histogram left to the red dotted line will be set to 0, and the others will be set to 1. Thus more pixels will get value 1.

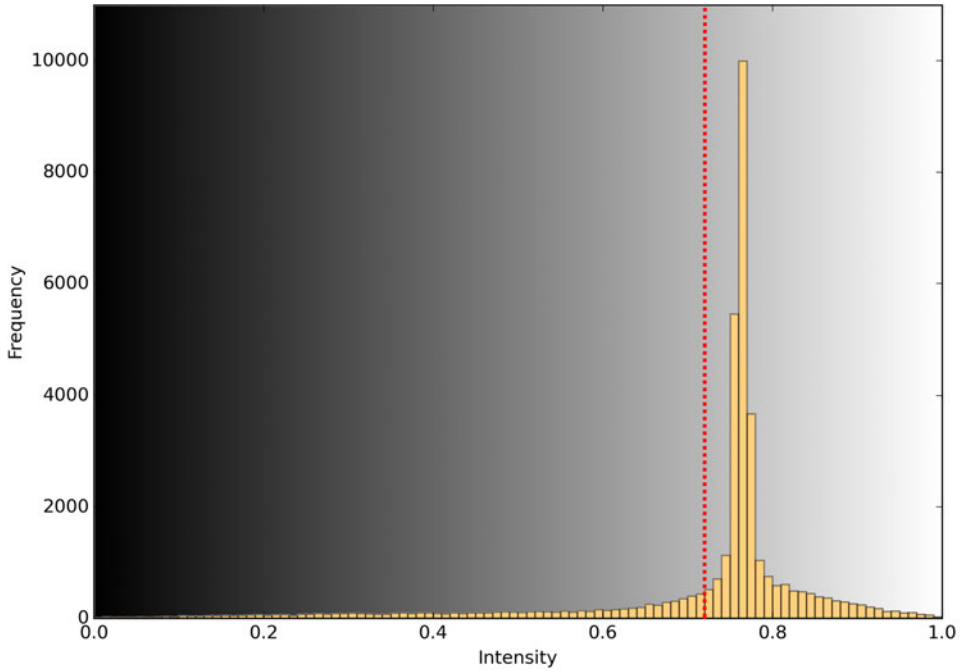
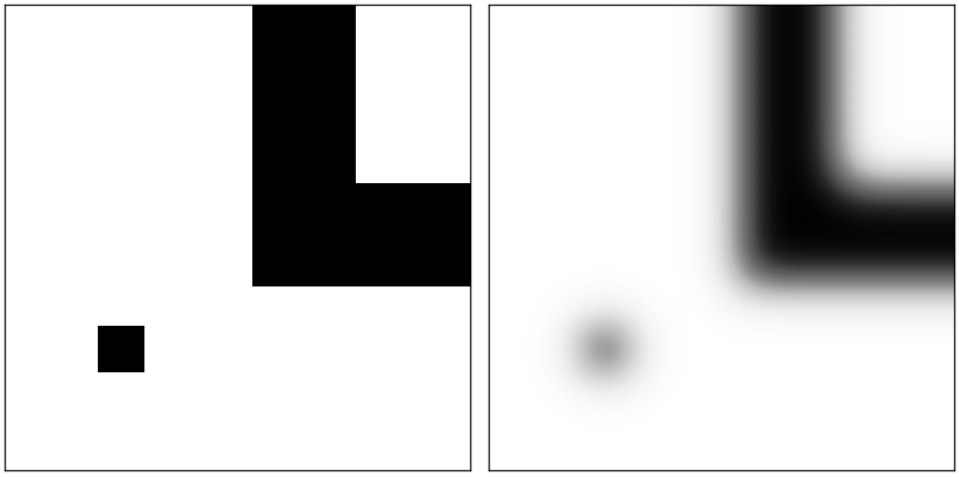


Figure 22: Histogram of the intensity of pixels in the image, with in the background the gray-scale of the intensity. The red dotted line is the threshold value.

In the next step we apply a Gaussian blur filter. The Gaussian blurred image is denoted by $\widetilde{I} = G_\sigma * I$. In Figure 23 a blurred image and its original image is shown, where the smaller block represents the noise inside a cell, which we wish to remove. This block is more faded than the larger block, which represents the boundary of a cell.



(a) Original image. (b) Blurred image.

Figure 23: Original and blurred image.

For this operation we first have to find a suitable σ. We want to choose σ such that a point at edge of the boundary will not be influenced by the white area at the other side of the boundary by applying Gaussian blur. Thus the dark point will not become more gray because of the white area at the other side, only because of the adjacent white area. See Figure 24.

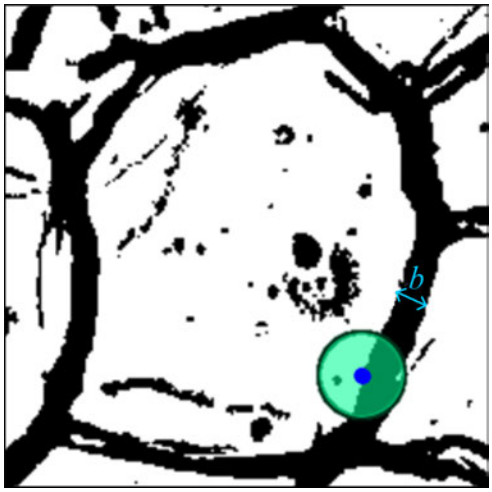


Figure 24: Binary image with a point at the edge of a boundary. The green circle is shown to illustrate the influences of the surrounding pixels by applying a Gaussian blur filter, with $\sigma = b/2$.

This can be achieved by setting $\sigma = b/2$, where b denotes the boundary width. In this case the value of the point at the edge of the boundary is mostly influenced by the boundary and the adjacent white area.

For this approach we need to determine the boundary width b. In order to estimate b, we need to

make some assumptions for the binary image, that is; every cell is a square $(R_{in} + b) \times (R_{in} + b)$ with equally large boundaries, and the cells are located, such as in Figure 25.

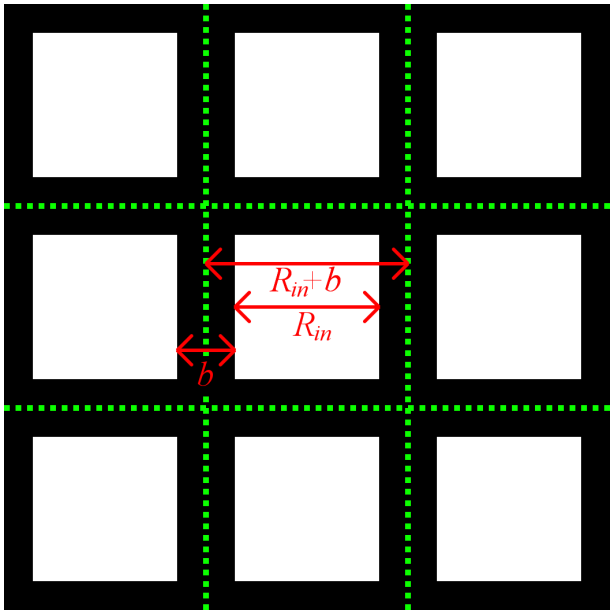


Figure 25: Simplified image of cells.

As can be seen in Figure 25, the cells are divided such that it contains half the width of the boundary. In this way, the cells are equally divided. The diameter of the white area inside the cell is denoted by R_{in}.

First, we calculate the area of a cell and thereafter $(R_{in} + b)$ can be calculated. This is done by dividing the total area of the image by the number of cells, which we already found in Section 5.1. Let $m \times n$ be the shape of the image. Then,

$$A_{cell} = \frac{mn}{N_{cell}}, \tag{36}$$

where A_{cell} is the area of one cell, and N_{cell} the total number of cells.
Since the cells are squares $(R_{in} + b) \times (R_{in} + b)$, we have

$$\begin{aligned} A_{cell} &= (R_{in} + b)^2 \implies \\ R_{in} + b &= \sqrt{A_{cell}}. \end{aligned} \tag{37}$$

If we are also able to calculate R_{in}, then we can determine the width b of the boundary.

Each cell in the binary image consists of a black and white part. In the previous part, we have found a threshold for the binary image. Thus, now we can calculate the part black and white points. The part white point is denoted by ρ_{white}, which is defined by,

$$\rho_{white} = \frac{1}{mn} \sum_p \mathbf{1}_p, \quad with\ \mathbf{1}_p = \begin{cases} 1, & I(p) \geq I_{threshold} \\ 0, & I(p) < I_{threshold} \end{cases}, \tag{38}$$

where p are the pixels in the image.

Consider one cell with area A_{cell}, we are now able to calculate the white area $A_{cell,white}$ of the cell:

$$A_{cell,white} = \rho_{white} A_{cell}. \tag{39}$$

And again, we find for the white area $R_{in} \times R_{in}$:

$$\begin{aligned} A_{cell,white} = (R_{in})^2 \implies \\ R_{in} = \sqrt{A_{cell,white}}. \end{aligned} \tag{40}$$

Combining eq. (37) and (40) yields an approximation for b:

$$\begin{aligned} b &= (R_{in} + b) - (R_{in}) \\ &= \sqrt{A_{cell}} - \sqrt{A_{cell,white}}. \end{aligned} \tag{41}$$

Note that this value for b is an over estimation of the actual boundary width, since we have black noise inside the cells, which results in a smaller ρ_{white}. Eventually, with eq. (36), (39) and (41) we find an expression for σ:

$$\begin{aligned} \sigma = \frac{b}{2} &= \frac{\sqrt{A_{cell}} - \sqrt{A_{cell,white}}}{2} \\ &= \frac{(1 - \sqrt{\rho_{white}})\sqrt{\frac{mn}{N_{cell}}}}{2}. \end{aligned} \tag{42}$$

For the the next part we look at the difference between the original image I and the Gaussian blurred image $\widetilde{I}$ defined by

$$I_\Delta = \widetilde{I} - I. \tag{43}$$

For the images in Figure 26, I_Δ is determined and shown in Figure 26.

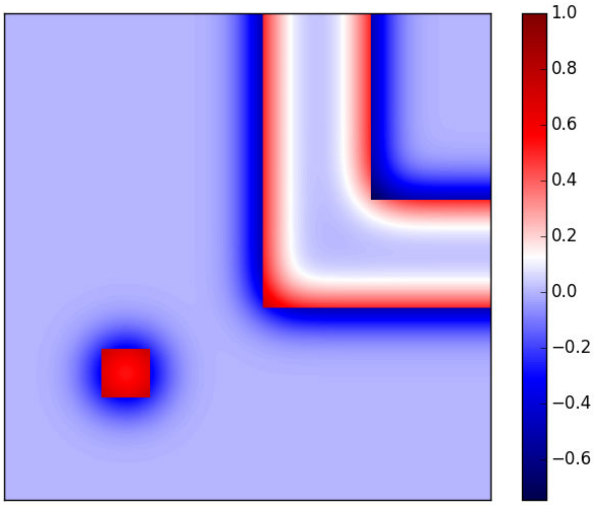


Figure 26: Difference of blurred image and original image.

The difference is very large at the block, but also at the corner of the boundary. We wish to find a threshold denoted by $\hat{I}_\Delta$, such that the smaller block will be removed and none of the points from the boundary. If the difference in intensity of a point is larger than the threshold, then that point will be removed. Consider a point at the corner of the boundary, such as in Figure 27.

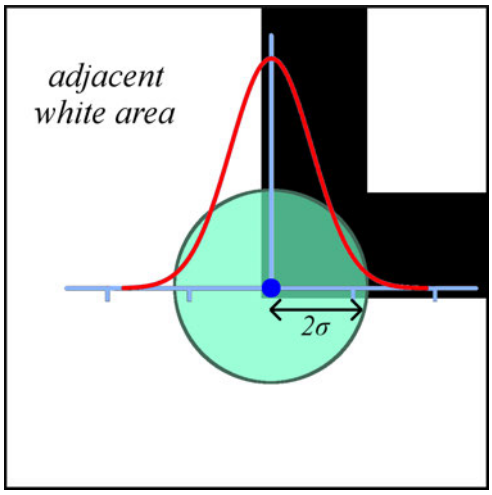


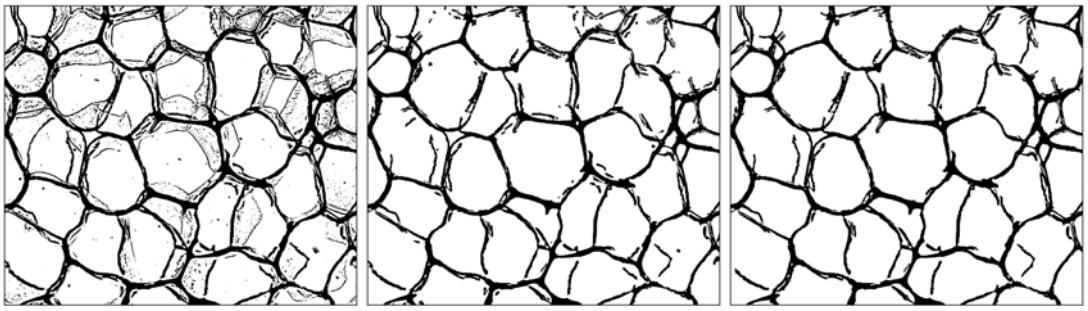
Figure 27: Point at the corner of the boundary, which we do not want to remove.

We do not want to delete this point, thus we have to find a large enough threshold. We calculate the new value of the point after the Gaussian blur. Assume that this is located at $(0,0)$ and that the upper right part of the image is black and the remaining part white. Then, we can evaluate the intensity after the Gaussian blur. This is approximately

$$\begin{aligned}\widetilde{I}_{corner}(0,0) &= (G_\sigma * I)(0,0)\\ &\approx G_\sigma[\text{black part}] * 0 + G_\sigma[\text{white part}] * 1\\ &\approx \int_0^\infty \int_0^\infty G_\sigma(x,y)dxdy * 0 + \left(1 - \int_0^\infty \int_0^\infty G_\sigma(x,y)dxdy\right) * 1\\ &= 1 - \int_0^\infty \int_0^\infty G_\sigma(x,y)dxdy. \end{aligned} \tag{44}$$

We have $\widetilde{I}_\Delta(0,0) = (\widetilde{I} - I)(0,0) = \widetilde{I}(0,0)$, because $I(0,0) = 0$. Since we do not want to delete this point, the threshold has to be greater than $\widetilde{I}_{corner}$. If we make the threshold too large, no points will be deleted, thus we set the threshold $\hat{I}_\Delta$ to $\widetilde{I}_{corner}$. If $b = 4.0$, then we get $\sigma = 2.0$, and now we can determine the threshold with eq. (44), that is $\hat{I}_\Delta = 0.64$.

In the last step, we delete the small remote points inside the cells. If a small area of black points is surrounded by white points, then we remove these black points. Finally, we get the following result, which can be seen in Figure 28.



(a) Original image (b) Removing noise with Gaussian blur. (c) Removing noise with a small area.

Figure 28: Results of removing noise.

This approach removes a lot of noise inside the cells, which is convenient for the simulation of the model. In Figure 29 we simulated a small image, where we first removed the noise in the image.

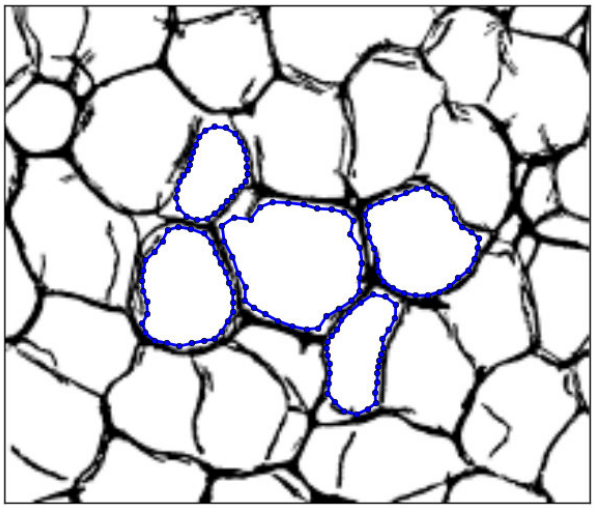


Figure 29: Mumford-Shah model for an image, where the noise is removed.

This gives a better and more stable result than the image from Figure 18.

We also simulated this for a larger image. The Mumford-Shah model performs better with the with the image where the noise has been with the binary image. However, this method can create holes in the boundary, where the boundary was either thin or very light in the original image. This can be seen in Figure 30.

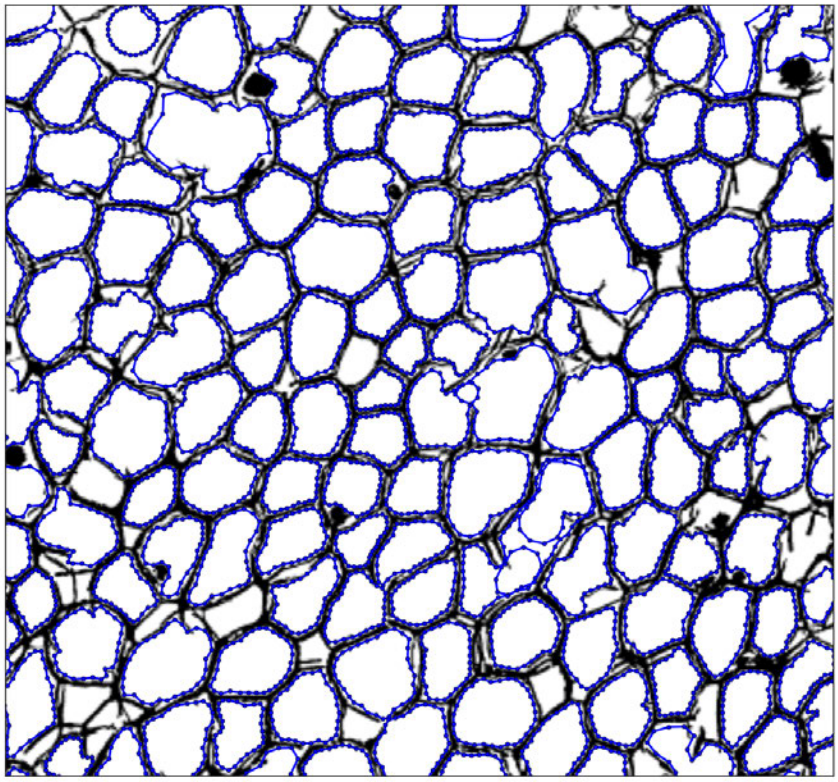


Figure 30: Mumford-Shah model for a large image, where the noise is removed.

The simulation took 34.0 seconds, which is much better than the simulation with the original image. However, not every contour has successfully reached the boundary of the cell. Some contours went through a hole in the boundary and others stopped at a black line or point, which is not part of the boundary cell. For some cells this methods seems to work quite well, but for other cells this is a too aggressive approach.

6 Improvements to the Mumford-Shah Model

In this section we will introduce some improvements to the Mumford-Shah model, which will either improve the accuracy or the computational time of the simulation.

6.1 Extra Internal Force

Our current modified Mumford-Shah model has two types of forces. An external force $(e^+ - e^-)$ and an internal force $|\boldsymbol{r}_s|^2$. In many other contour models, an extra internal force is used $|\boldsymbol{r}_{ss}|^2$. This is a force designed to keep the contour from bending too much, while our current internal force is designed to keep the contour from stretching too much. The new functional then becomes

$$\widetilde{E}(u, C) = \int_\Omega (f-u)^2 dxdy + \frac{1}{2}\int_0^1 \nu|\boldsymbol{r}_s|^2 + \kappa|\boldsymbol{r}_{ss}|^2 ds,$$

where ν and κ are weighting parameter. These parameters control the contour's tension and rigidity [2]. Again, we can apply variational calculus to find the Euler-Lagrange equation which the contour must satisfy in order to minimize $\widetilde{E}$. However, this is analog to the derivation of the Euler-Lagrange equation without the extra internal force, which has been done in Section 3.2.2. We get

$$\left[(f-u_i)^2 - (f-u_o)^2\right]\Big|_{\boldsymbol{r}} \begin{pmatrix} -y' \\ x' \end{pmatrix} - \nu\frac{d^2\boldsymbol{r}}{ds^2} + \kappa\frac{d^4\boldsymbol{r}}{ds^4} = 0.$$

As we did before, the minimization problem can be solved by gradient descent. This gives us

$$\frac{\partial \boldsymbol{r}}{\partial t} = -\frac{\partial E}{\partial \boldsymbol{r}} = [e^+ - e^-]\mathbf{n}_C + \nu\frac{d^2\boldsymbol{r}}{ds^2} - \kappa\frac{d^4\boldsymbol{r}}{ds^4}. \tag{45}$$

The next step is to discretize this in space. The numerical scheme of the fourth derivative with central difference is

$$f_i^{(4)} = \frac{1}{h^4}(f_{i-2} - 4f_{i-1} + 6f_i - 4f_{i+1} + f_{i+2}) + O(h^2). \tag{46}$$

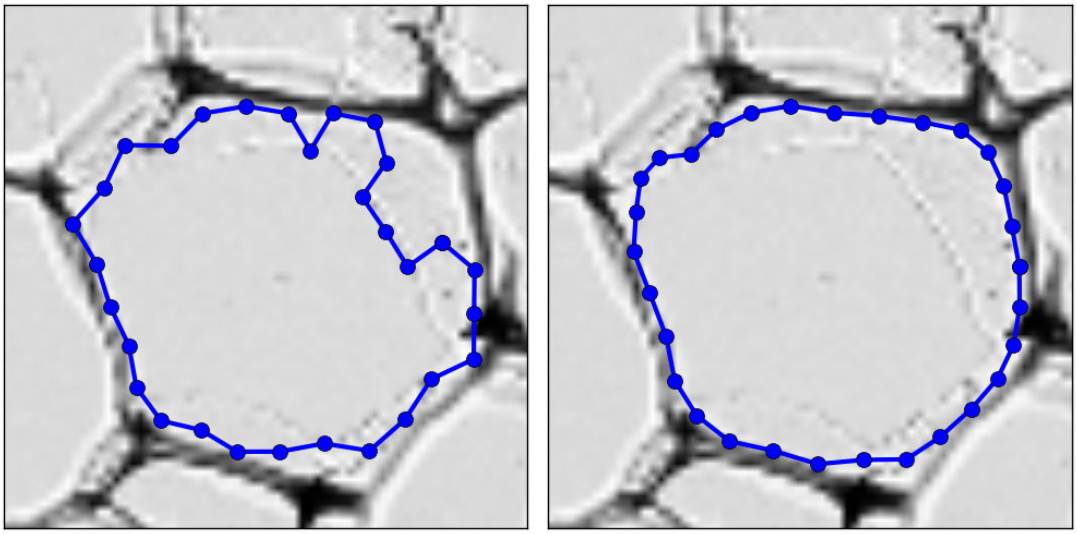
Using the periodicity of $\boldsymbol{r}$ we find the discretized system of $\boldsymbol{r}''''$. That is $\boldsymbol{r}'''' = D\boldsymbol{r}$, where D is the matrix

$$D = \begin{pmatrix} 6 & -4 & 1 & & & 1 & -4 \\ -4 & 6 & -4 & 1 & & & 1 \\ 1 & -4 & 6 & -4 & 1 & & \\ & & \ddots & \ddots & \ddots & & \\ & & 1 & -4 & 6 & -4 & 1 \\ 1 & & & 1 & -4 & 6 & -4 \\ -4 & 1 & & & 1 & -4 & 6 \end{pmatrix}. \tag{47}$$

Using Backward Euler as time integration we get the following system

$$\begin{pmatrix} \widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}} \end{pmatrix}^{k+1} = \left(I_{2N} - \Delta t \left(\begin{array}{c|c} \nu A - \kappa D & EB \\ \hline EC & \nu A - \kappa D \end{array}\right)\right)^{-1} \begin{pmatrix} \widetilde{\boldsymbol{x}} \\ \widetilde{\boldsymbol{y}} \end{pmatrix}^{k}, \tag{48}$$

In Figure 31 we see the results of two simulation, one without and one with the extra bending force. The same image is used for both simulations.



(a) Without bending force

(b) With bending force, $\kappa = 0.01$.

Figure 31: Mumford-Shah model with Backward Euler, $\nu = 0.001$ and $\Delta t = 5.0$.

The model with only the stretching force clearly performs worse than the model with the extra internal bending force, so this is good improvement to the modified Mumford-Shah model.

6.2 Moving Area

Constant areas were used in the previous modified Mumford-Shah models. For each cell, an large enough rectangle for the area Ω is defined, such that the cell is fully contained in the area. As a result, some areas were way too large for the concerned cell and this led to a mediocre result. In this section we will be using moving areas instead of constant areas, so that we no longer encounter that problem. At the end of the simulation, the variable area will only be slightly larger than the contour itself.

We start with a given initial contour represented by $\boldsymbol{r}$, and we define the boundary of the area by

$$\boldsymbol{r}_\Omega = \boldsymbol{r} + h\mathbf{n}, \tag{49}$$

where $h > 0$ is a constant scalar and $\mathbf{n}$ is the unit normal vector to the initial contour. The area Ω is the inside of $\boldsymbol{r}_\Omega$. Since $\boldsymbol{r}$ changes in each time step, we also change $\boldsymbol{r}_\Omega$ in each time step. If the contour becomes larges, the area also becomes larger. The initial situation of this method can be seen in Figure 32.

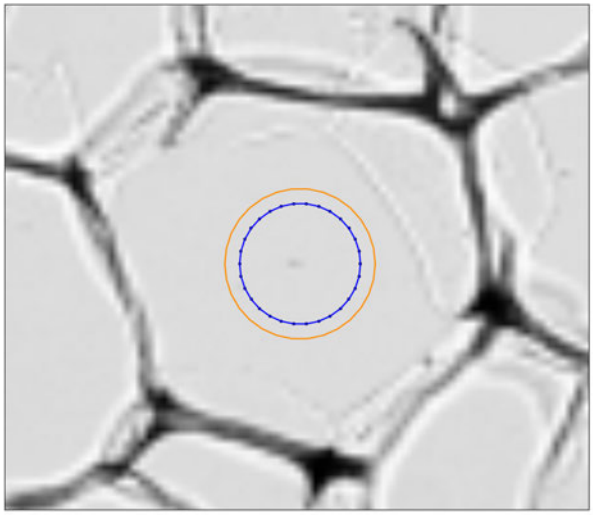


Figure 32: Initial contour with its moving area give by the orange line.

The method works the same as the modified Mumford-Shah model except for one thing. At the beginning of the simulation we will have $u_o \approx u_i$, because the boundary of the cell is not inside the area yet. This means that

$$\begin{aligned} F &= \left[(f-u_o)^2 - (f-u_i)^2\right]\mathbf{n}_C \\ &\approx \left[(f-u_i)^2 - (f-u_i)^2\right]\mathbf{n}_C \\ &= \mathbf{0}, \end{aligned}$$

thus we have a force almost equal to $\mathbf{0}$. We solve this problem by using an artificial force.

$$\text{If } u_o \approx u_i, \text{ then } u_o^* = u_o - 0.05. \tag{50}$$

During the simulation the difference between u_i and u_o becomes greater. This happens when parts of the boundary comes to lie in the current moving area. In Figure 33 the average of u_i and u_o of 392 contours can be seen. This figure shows that the artificial force is needed for about the first hundred steps. This reduces the amount of steps needed and thus the computational time.

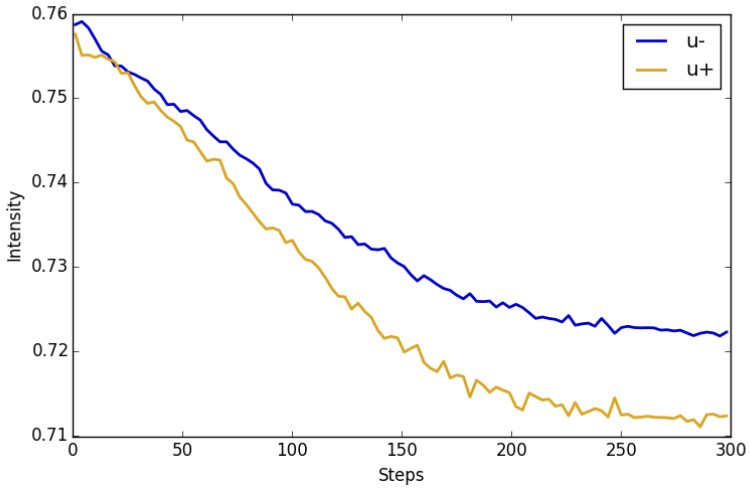
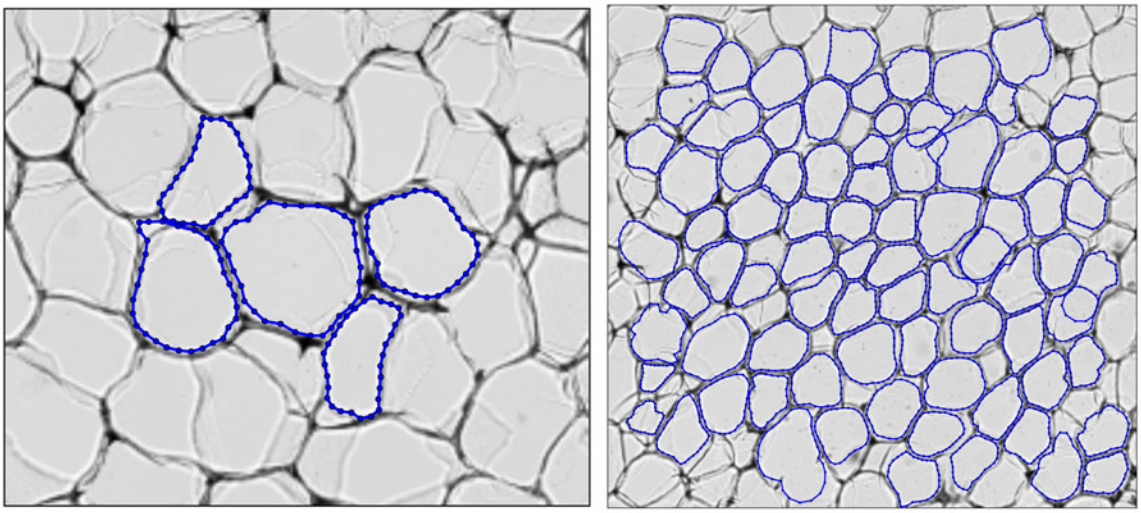


Figure 33: Average u_i and u_o of 392 contour during a simulation.

We simulate this method for two images. We take 300 steps with time step $\Delta t = 10.0$. We use the parameters $\nu = 0.004, \kappa = 0.004$ and $h = 2.0$. The results can be seen in Figure 34.



(a) 5 contours (b) 73 contours

Figure 34: Results of Mumford-Shah with moving areas.

For the image in Figure 34(a) this methods seems to work quite well, but the methods does not work properly for some cells in the image Figure 34(b). This can be caused by a small difference between u_i and u_o, even though the contour is at the boundary of the cell. This occurs when only few points has reached the boundary. Then the contour will get an artificial force, which causes the contour to jump past the boundary.

The moving area method has a better computational time than the modified Mumford-Shah model, but adding an artificial force can lead to jumping past boundaries.

6.3 Improved Mumford-Shah Model

The main problem with the modified Mumford-Shah model is the computational time. We have tried solving this problem using image filtering and moving area. These methods indeed improved the computational time, but they do not perform optimal. In this section we will discuss an improved version of the Mumford-Shah model. This method consists of two phases; the first phase gives an approximation and the second phase gives an accurate result.

6.3.1 Jump Phase

Before we use the modified Mumford-Shah model, we first want to find an approximation of the boundary of the cell. This approximation need not to be accurate, it only needs to be near the boundary. In this way we do not have to take many steps with the modified Mumford-Shah model. This is done in the first phase of the method; the jump phase. We try to quickly move the contour towards the boundary with the aim to reduce the computational time.

For this method we use a moving area defined by

$$\boldsymbol{r}_{\Omega_o} = \boldsymbol{r} + h_{jump}\mathbf{n}, \tag{51}$$

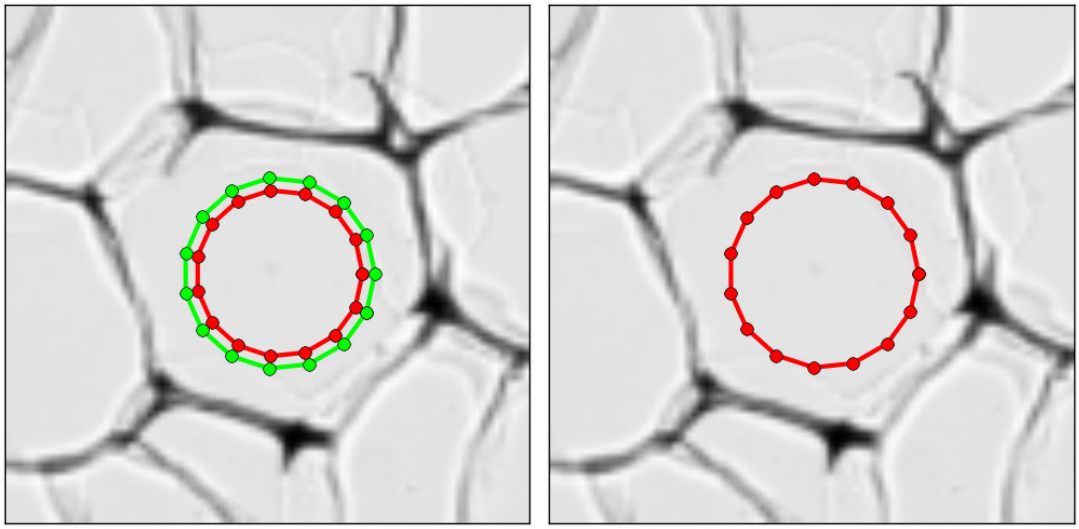
where h_{jump} is a positive constant and $\mathbf{n}$ the unit normal vector to the contour. At the beginning of the simulation, when the contour has not reached the boundary yet, the difference between u_i and u_o is very small. This results in a very small external force, and this causes the contour to shrink due to the

internal forces. A way to counter this is by jumping over such areas, where the difference of the outside and inside area is small.
Consider the case where $u_o \approx u_i$. Thus we can assume that the outside area does not contains a part of the boundary of the cell. Now we can jump to the boundary of the area Ω_o without jumping over the boundary of the cell. Thus we apply the following rule: if

$$\nabla u < k, \text{ with } \nabla u \approx u^+ - u^-, \tag{52}$$

then we jump to boundary of the area. Here, k is a positive constant. This value can not be too large, since this will result in approximation contour, which has jumped past the boundary of the cell. However, if this value is not large enough, the approximation contour will not come near the boundary.

In Figure 35 an example is shown, where Ω_o is the area between $\boldsymbol{r}$ and $\boldsymbol{r}_{\Omega_o}$, and Ω_i is the area inside the contour $\boldsymbol{r}$. In this example, we have $\nabla u < k$, thus the contours jumps to the boundary of the area Ω, see Figure 35(b).



(a) We check if $\nabla u < k$.

(b) The contour jump towards the boundary of the area Ω.

Figure 35: An iteration step of the contour in the jump phase. The contour is given in red and the boundary of the area Ω in green.

This approach causes the contour to move quickly to the boundary of the cell, but for many cells this leads to unwanted results. If just one point of the contour reaches the boundary, the difference between u_i and u_o is not large enough to stop the entire contour from jumping. Eventually, the contour will jump past the boundary, as can be seen in Figure 36.

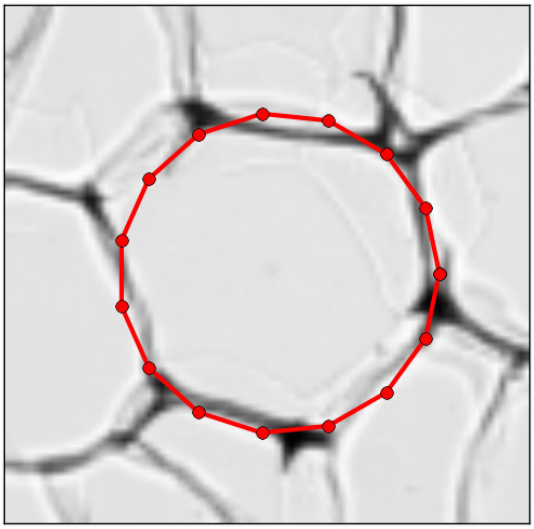


Figure 36: Result of jump phase when using a collective outside and inside area.

The aspect that causes the problem is checking the difference globally. One point at the boundary will simply not be noticed with this approach. One way to solve this problem is by checking the jump statement from eq. (52) locally for an individual point instead of the entire contour. This requires to evaluate the mean value u_o for smaller areas. We have to define such areas for each point, this is shown in Figure 37.

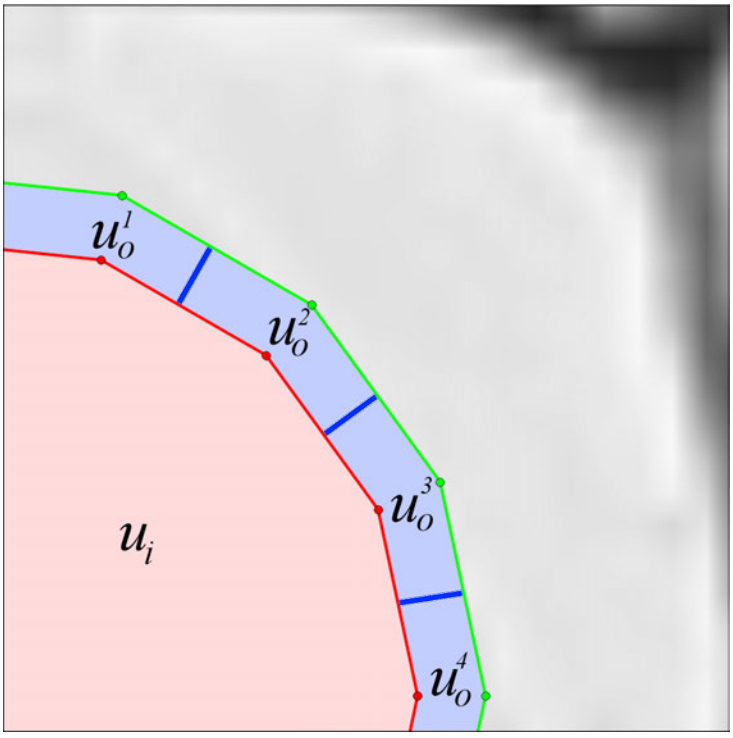
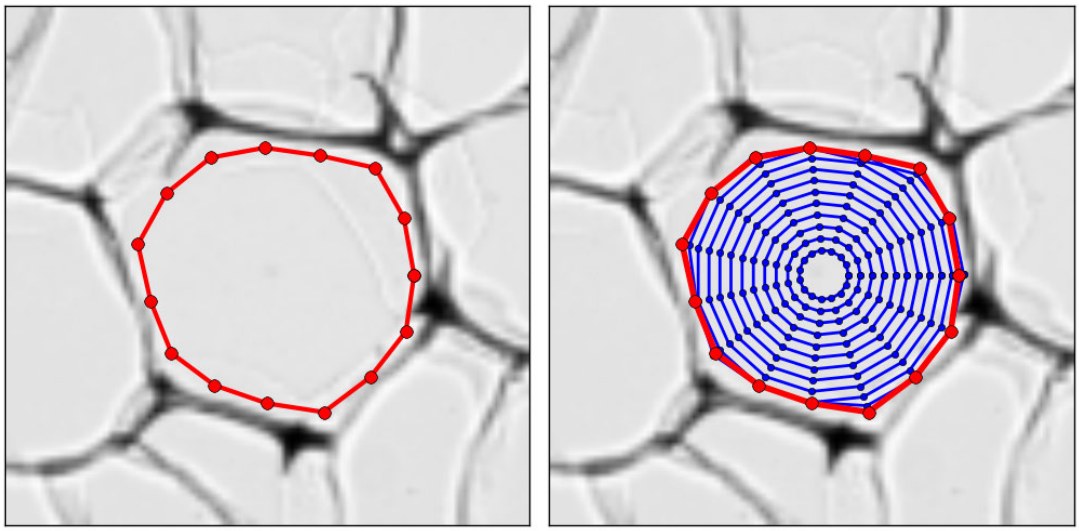


Figure 37: Each point has its own outside area u_o^j.

In equation (53) u_o^j is defined for each point s_j, this will be discussed in the next phase of the method.

Now, for every u_o^j we check if $\nabla u^j < k$. This results in a much better approximation of the boundary.

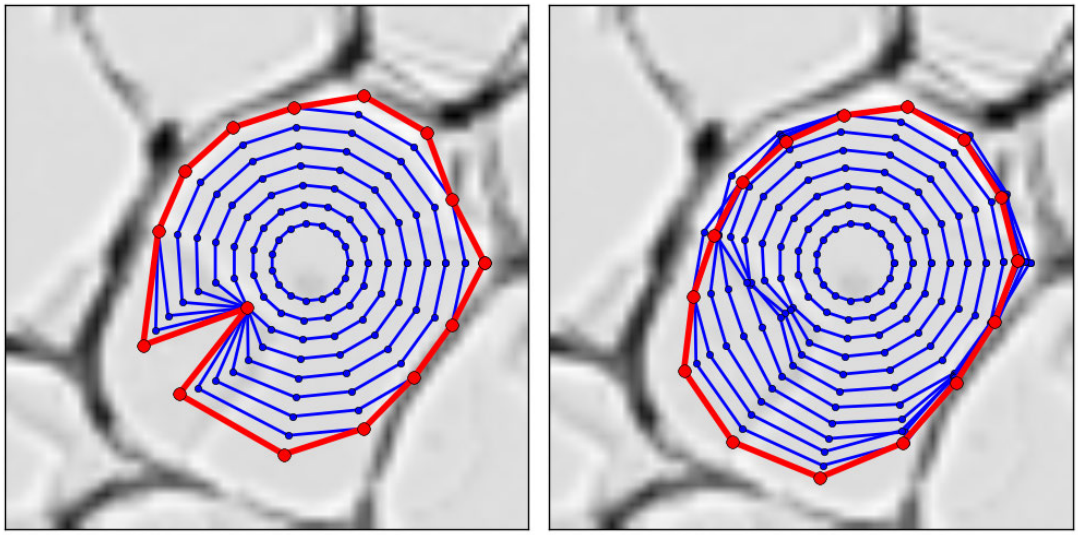
This can be seen in Figure 38, where the contour has 15 points.



(a) Final result of the approximation contour. (b) Time evolution of the jump phase.

Figure 38: Results of the jump phase with small areas u_o^j for each point.

Figure 38 shows the result and the time evolution of the contour. We see that the contour makes a jump in each time step at the beginning. As the contour comes close to the boundary, some points stop jumping. If eventually 75% of the points stops jumping, then we stop every point from jumping.
This particular cell is quite simple, so we will get a desired result. However, if a cell has a dark area inside the cell, it can occur that one point of the contour does not jump, because $\nabla u^j \geq k$ for a point s_j. In this case, the final contour will have inward spikes. This can be seen in Figure 39(a). Therefore it is necessary to still use the Mumford-Shah model with large enough weighting parameters ν and κ. This will pull the point that has stopped jumping towards its adjacent points, since its internal stretching and bending force are very large. We simulate this method with and without the Mumford-Shah model for one cell. Figure 39 clearly shows that using the Mumford-Shah model yields a better result.

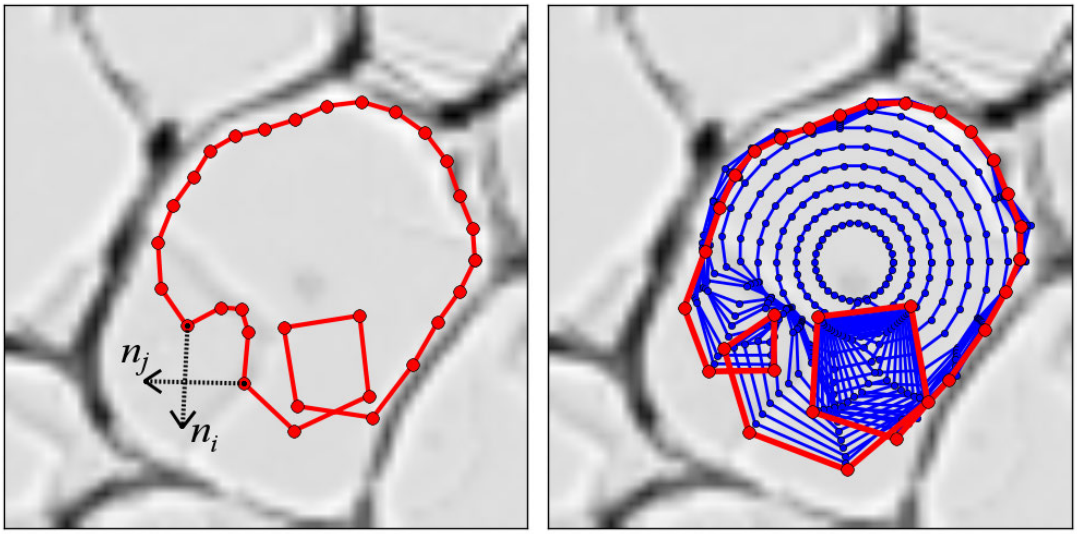


(a) Without Mumford-Shah. (b) With Mumford-Shah, $\nu = 0.005$ and $\kappa = 0.005$.

Figure 39: Jump phase without and with Mumford-Shah model.

In Figure 39(b) we see that some point have stopped jumping, while not being close to the boundary. The internal forces of the Mumford-Shah model causes the points to move past the darker area. Eventually these points will start jumping again.

For the jump phase of the method we only use 15 points, which is a small amount compared to the 30 points used in the previous sections. The reason for this is to get a more stable approximation, because of the following two reasons: if there is a dark area inside the cell, the contour with 15 point will not be influenced much, because it takes the mean value of a larger area than the contour with 30 points. Also, if a point does not jump because of a dark area, the adjacent point will probably pull it past the dark area. If the contour has 30 point, then it is more likely that 2 points will get stuck behind the dark area. Then the internal forces will not have the same effect. Therefore it is convenient to use few points for the first phase of the method. The jump phase is simulated to illustrate this for a contour with 30 points and it can be seen in Figure 40.



(a) Contour after 8 steps.

(b) Contour and its time evolution after 15 steps.

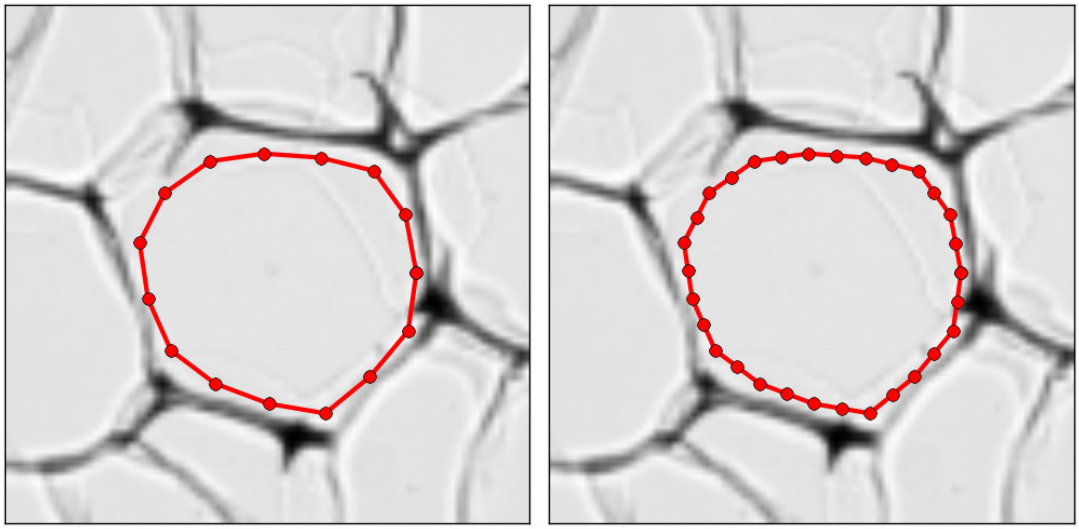
Figure 40: The contour in the jump phase after 8 and 15 steps.

In Figure 40 an example is shown where a loop is formed, this is caused by a dark area inside the cell. For some points their normal vector points towards each other, due to points that has stopped jumping, while the other points continue to jump. Since the external force is in the direction of the normal vector, the points will cross past each other, which creates a loop. Since we want to avoid this kind of situation, we decide to use 15 points for the contours for this phase of the method.

6.3.2 Mumford-Shah Phase

In the jump phase of our method we have found an approximation. In the next phase we use the modified Mumford-Shah model to get an accurate result from the approximation.

As discussed in the previous section, it is useful to use few points for the contour in the jump phase. However, in the next part; the Mumford-Shah phase, it is necessary to use more points to get an accurate result. Therefor we first double the amount of points, simply by putting the extra points between each original point of the contour, see Figure 41.



(a) Contour with 15 points. (b) Contour with 30 points.

Figure 41: Doubling the number of points for a contour.

For this phase we again use smaller areas for each point for the outside area, but also for the inside area. These area are denoted by Ω_o^j and Ω_i^j, respectively and the mean values of the areas are denoted by u_o^j and u_i^j, see Figure 42.

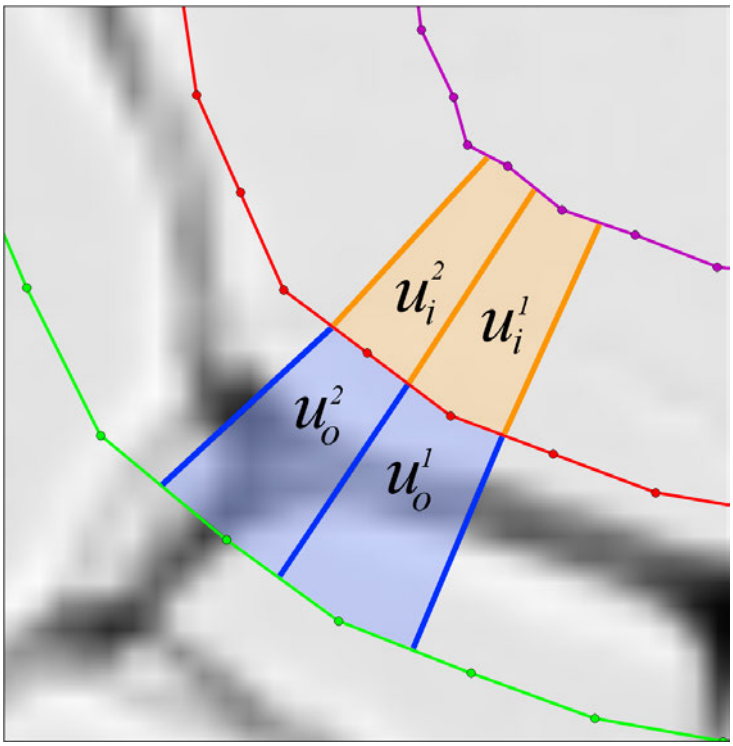


Figure 42: Each point has its own outside and inside area, with mean values u_o^j and u_i^j for $j = 1, ..., N$.

The boundary of the inside and outside area are denoted by $\boldsymbol{r}_{\Omega_i}$ and $\boldsymbol{r}_{\Omega_o}$. This is defined as

$$\begin{aligned} \boldsymbol{r}_{\Omega_o} &= \boldsymbol{r} + h_{MS}\mathbf{n}, \\ \boldsymbol{r}_{\Omega_i} &= \boldsymbol{r} - h_{MS}\mathbf{n}. \end{aligned} \tag{53}$$

In the next part we define the inside and outside area of a point s_j. For a point between two points of a contour C with representation $\boldsymbol{r}$, we have $\bar{\boldsymbol{r}}^j = \frac{\boldsymbol{r}^j + \boldsymbol{r}^{j+1}}{2}$. The area Ω_o^j is the area between $\boldsymbol{r}$, bounded by $\bar{\boldsymbol{r}}^{j-1}$ and $\bar{\boldsymbol{r}}^j$, and $\boldsymbol{r}_{\Omega_o}$, bounded by $\bar{\boldsymbol{r}}_{\Omega_o}^{j-1}$ and $\bar{\boldsymbol{r}}_{\Omega_o}^{j}$. And the area Ω_i^j is the area between $\boldsymbol{r}$, bounded by $\bar{\boldsymbol{r}}^{j-1}$ and $\bar{\boldsymbol{r}}^j$, and $\boldsymbol{r}_{\Omega_i}$, bounded by $\bar{\boldsymbol{r}}_{\Omega_i}^{j-1}$ and $\bar{\boldsymbol{r}}_{\Omega_i}^{j}$.

In each iteration we calculate u_i^j and u_o^j for every point s_j with $j = 1, ..., N$. For this phase of the method we use a larger inside and outside area defined in eq. (53) than the area in the jump phase defined in eq. (51), because we have to make sure that the boundary is contained in the outside area.

With these mean values, we calculate the external force for each point, that is

$$\boldsymbol{F}^j = [(f^j - u_o^j)^2 - (f^j - u_i^j)^2]\mathbf{n}_C. \tag{54}$$

In the previous sections f^j was evaluated by $f^j = f(x^j, y^j)$, but this approach can sometimes lead to an unstable simulation. This happens when a point s^j with coordinates (x^j, y^j) accidentally is located at a dark area inside the cell. This causes the point to get stuck behind the dark area, while the other points move forward. It can also occur that a point is accidentally at a lighter area, even though the contour is already located at the boundary. Then this point will jump past the boundary. This results in instabilities, which is shown in Figure 43(a).
This problem can be solved by choosing a different way to evaluate f^j, such that the method is more stable. A convenient choice would be

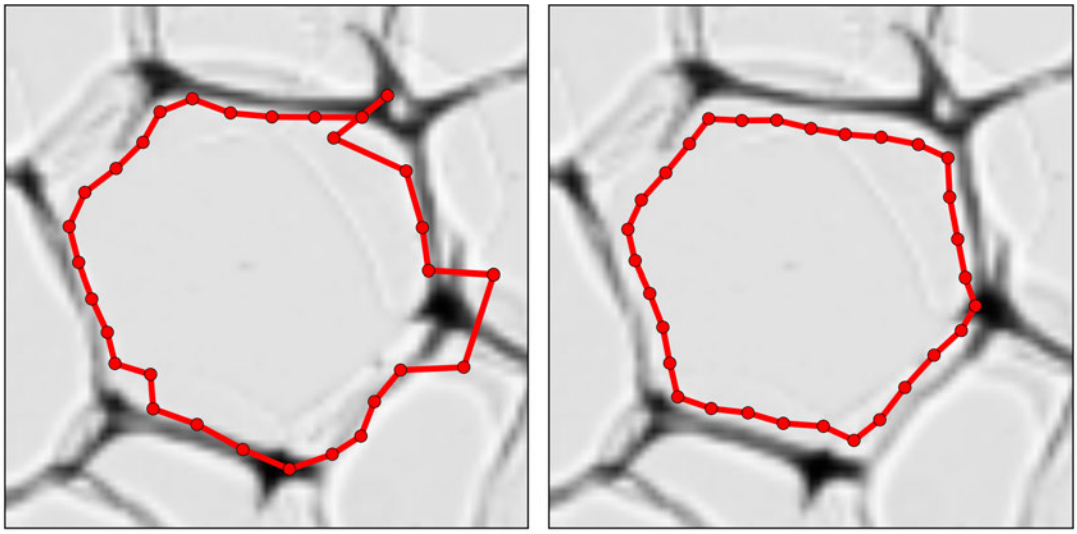
$$f^j = mean_{\Omega_d^j}(f), \tag{55}$$

where $\Omega_{j,d}$ denotes the square area around point s^j with size $2d \times 2d$. This is defined as

$$\Omega_{j,d} = [x^j - d, x^j + d] \times [y^j - d, y^j + d]. \tag{56}$$

The size of the square influences the stability of the method. If d is large, then the method will be more stable, but not very accurate. The simulation will stop before the contour reaches the boundary of the cell.

In Figure 43 an example is shown where we simulated the method using $f^j = f(x^j, y^j)$ as has been done in the previous sections and also using the new approach where f^j is evaluated as in eq. 55.



(a) Result with $f^j = f(x^j, y^j)$.

(b) Result with $f^j = mean_{\Omega_{j,d}}(f)$.

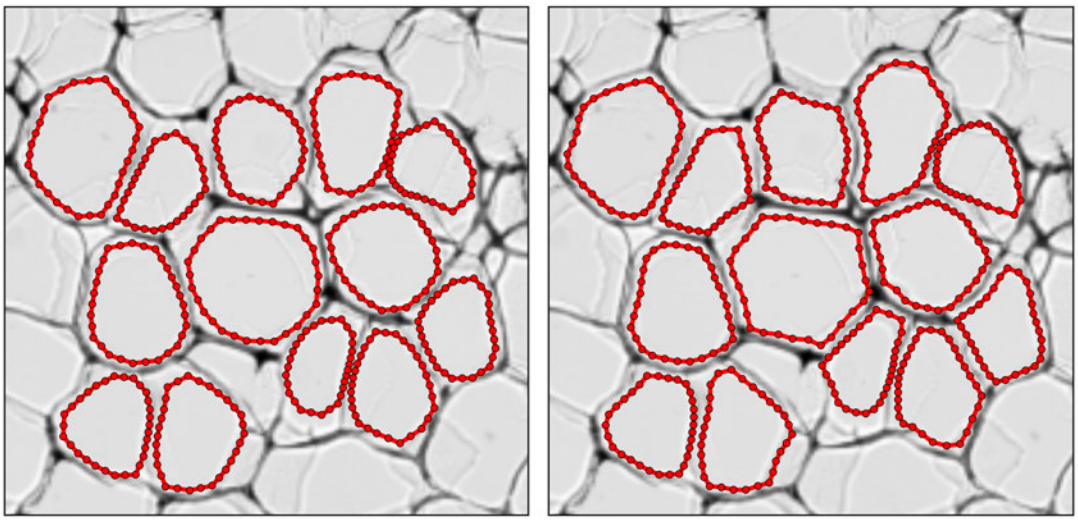
Figure 43: Results of the Mumford-Shah phase with different ways to evaluate f^j.

This result shows that using f^j defined as in eq. (55) leads to a more stable result. Despite being less accurate, we will use this evaluation of f^j. At the end of this section we will introduce a procedure to improve the accuracy of the contour.

For the jump phase we used weighting parameters ν_{jump} and κ_{jump}. For the Mumford-Shah phase we use different values for the weighting parameters ν_{MS} and κ_{MS}, since we take smaller steps than in the previous phase.

An advantage of the Mumford-Shah model is that the contours do not have to be stopped when it reaches the boundary of the cell. The contour stops by itself. However, if the external force is almost zero for more than 75% of the total amount of points of the contour, the contour will be stopped. This improves the computational time.

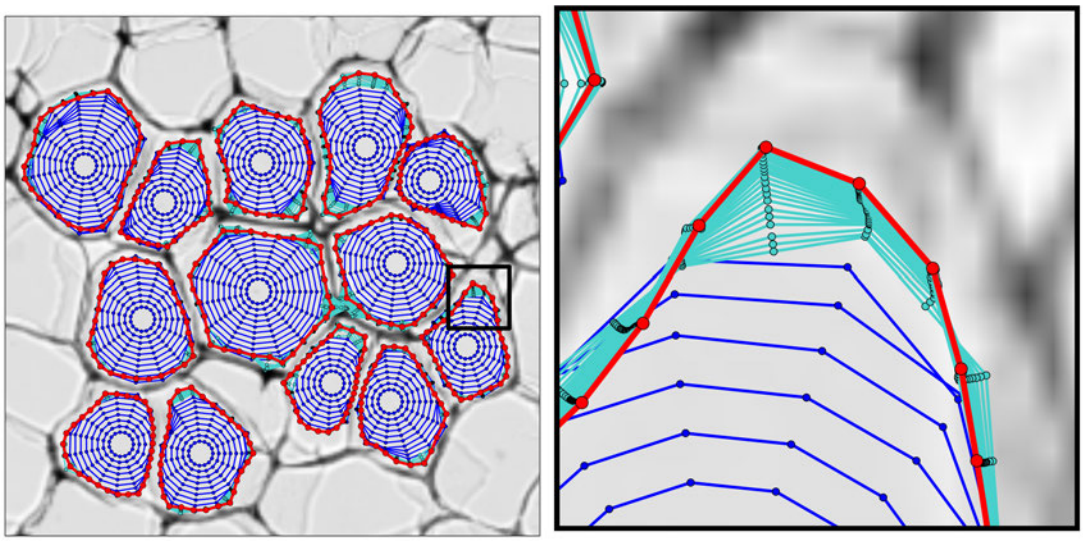
For the next part we simulate this method with two phases for an image with 13 contours. We use the weighting parameters $\nu_{jump} = \kappa_{jump} = 0.005$ and $\nu_{MS} = \kappa_{MS} = 0.002$. We start with 15 points for each contour and time step $\Delta t = 10.0$ is being used for 40 steps. For the inside and outside areas we choose $h_{jump} = 2.0$ and $h_{MS} = 4.0$ and for the square block in (56) we choose the value $d = 3.0$. The results can be seen in Figure 44.



(a) Result of the jump phase. (b) Final result of the Mumford-Shah phase.

Figure 44: Results of the two phases.

This shows the behavior of the two phases. The first phase gives an approximation, and the second phase an accurate result. In the latter phase the contour takes the form of its cell. The size of the contours do not change much; this confirms that the results from the jump phase are indeed good approximations. The execution time of this simulation is 4.85 seconds, which is an improvement compared to the previous methods.



(a) Result of the method for 13 contours. (b) Enlarged image of the indicated part.

Figure 45: The time evolution of the jump and Mumford-Shah phase given in blue and turquoise, respectively. The final result is given in red.

In Figure 45 the time evolution of the two phases is shown. The latter phase makes very small steps. Figure 45(b) shows that the method is stable. If the point is not close to the boundary, the point will move towards it. If a point is already at the boundary, then that point will not move much.

6.3.3 Inflation

As discussed before, the contours are not as close to the boundaries of the cells as we would like to be. This is caused by the size of the square area as defined in (56). However, the contours are similar in shape to the boundaries of the cells. Thus we only have to inflate the cells to get the desired result. This can be performed by moving the points of the contour in the direction of the unit normal vector to the contour. This can be described by

$$\boldsymbol{r}_{infl} = \boldsymbol{r} + h_{infl}\mathbf{n}. \tag{57}$$

For this this value h_{inf} we have to make a suitable choice. Consider a point s_j of a contour at the end of the simulation. The block Ω_d^j belongs to that point. If the simulation has come to an end, then this means that the value of f^j is the average of u_i^j and u_i^j, such that the external force is $\mathbf{0}$.

Now consider the simplified case, where Ω_i^j, Ω_o^j and Ω_d^j are all equal in size $d \times d$. Let the image I be a binary image, which consists a small part of the boundary with a width given by b. This simplified case can be seen in Figure 46(a). Next, u_i^j and u_o^j can be calculated.

$$u_i^j = \frac{1(d \times d)}{d \times d} = 1,$$
$$u_o^j = \frac{1(d \times (d-b)) + 0(d \times b)}{d \times d} = \frac{d \times d}{d \times d} - \frac{b \times d}{d \times d} = 1 - \frac{b \times d}{d \times d}$$

Since f^j is the average of the two values, we get

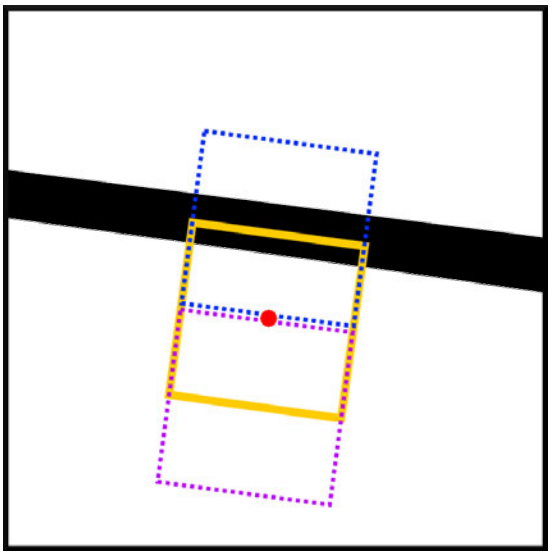
$$f^j = \frac{u_i^j + u_0^j}{2} = \frac{1 + 1 - \frac{b \times d}{d \times d}}{2} = 1 - \frac{\frac{1}{2}(b \times d)}{d \times d}. \tag{58}$$

We can conclude that the block Ω_d^j must contain half the width of the cell boundary, because the mean value then becomes

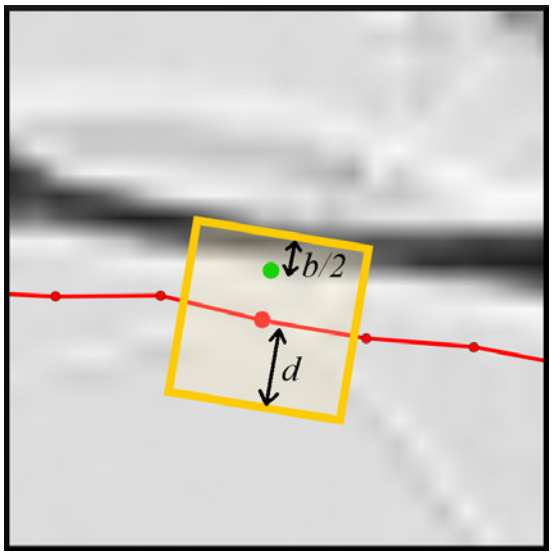
$$mean_{\Omega_d^j}(f) = \frac{1(d \times (d - b/2)) + 0(d \times b/2)}{d \times d} = 1 - \frac{\frac{1}{2}(b \times d)}{d \times d}$$

and this is equal to the average form eq. (58).

Thus if we set $h_{inflate} = d$, then the contour comes to lie in the middle of the boundary of the cell. However, we want to locate the edge of the boundary, thus we have to correct this with half the width of the boundary $b/2$. See Figure 46(b). Then we get $h_{inflate} = d - b/2$. The value for b can be determined by eq. (41) .An example of this procedure is shown in Figure 47.

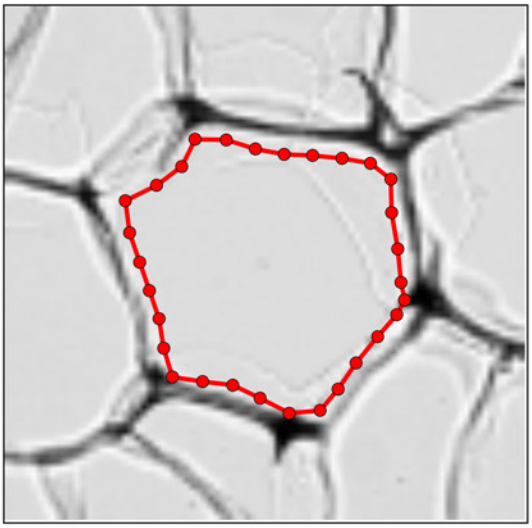


(a) Simplified case with Ω_i^j and Ω_o^j given in purple and blue, respectively.

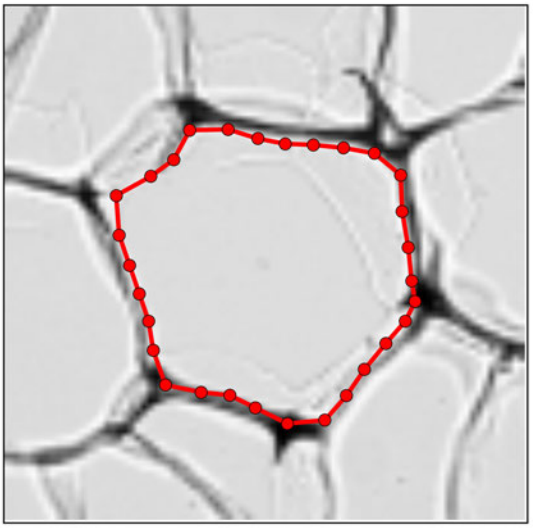


(b) Original case, where the green point is the point where we want to move the red point to.

Figure 46: The simplified and original case of the image to illustrate how we want to inflate the contour. The block Ω_d^j is given by yellow.



(a) Contour after the Mumford-Shah phase.



(b) Inflation with $h_{infl} = d - b/2$.

Figure 47: Inflation of the contour.

Figure 47 shows that the inflation of the contour in the last step leads to a good result. We also applied this for the result in Figure 44. This can be seen in Figure 48.

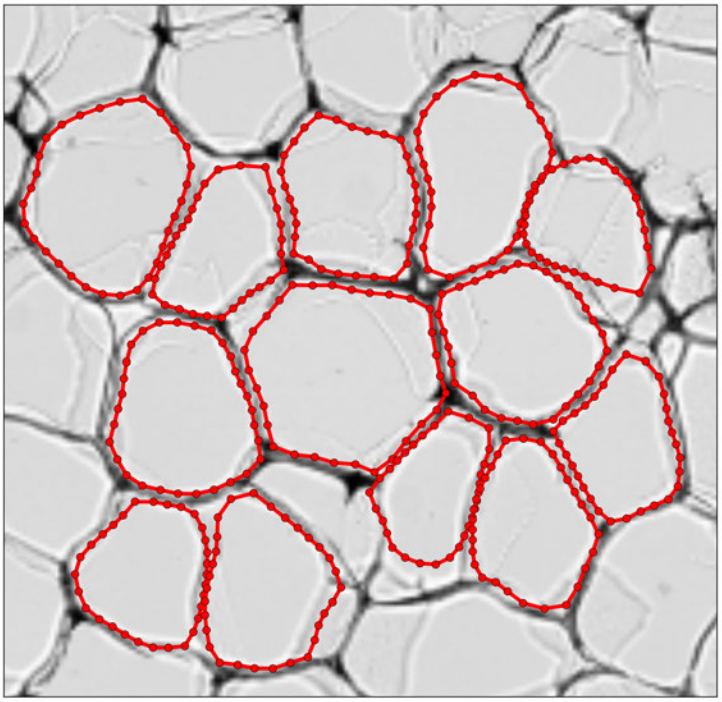


Figure 48: Result after inflation with $h_{infl} = d - b/2$.

We see that this approach leads to a good result. With the use of the block Ω_d^j and thereafter the inflation we get a stable and accurate result. The execution time of this method for the example in Figure 48 is 4.94 seconds.

6.3.4 Results

We simulate this model for a larger image, with $\nu_{jump} = \kappa_{jump} = 0.005$, $\nu_{MS} = \kappa_{MS} = 0.002$, $h_{jump} = 2.0$, $h_{MS} = 4.0$, $n_{iter} = 40$ and $\Delta t = 10.0$. The initial contour consists of 15 points. An part of the image with the results can be seen in Figure 49.

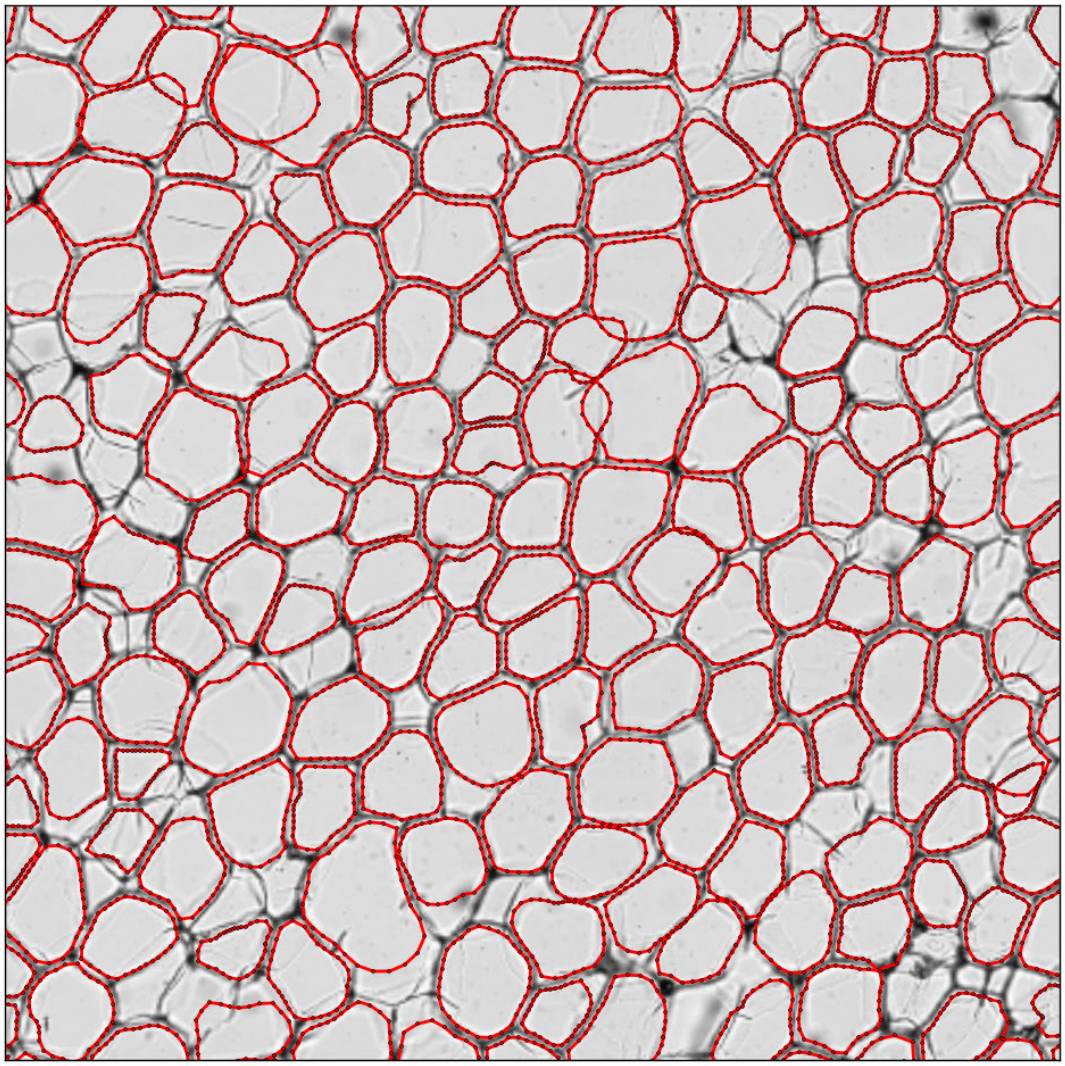
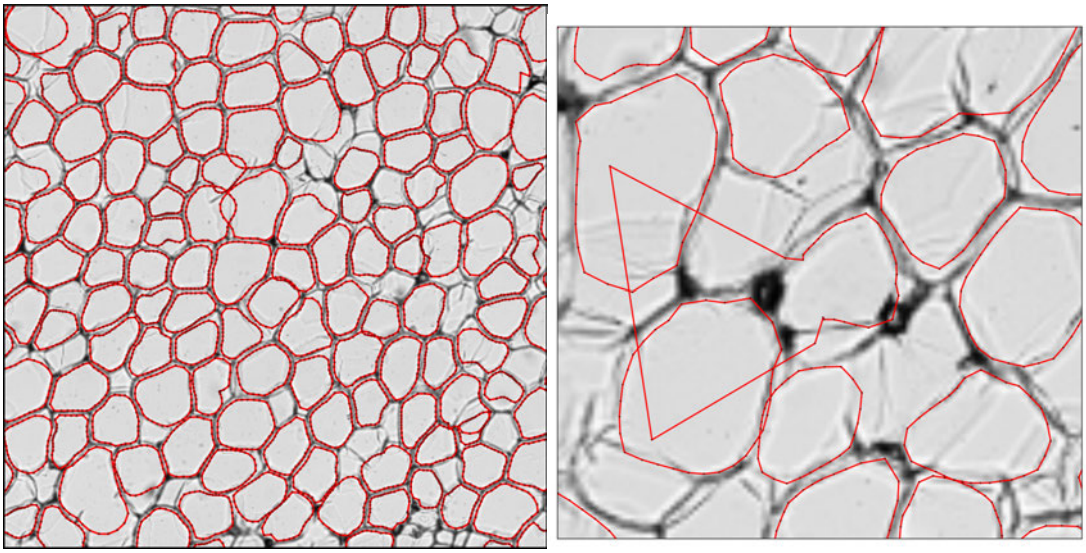


Figure 49: Results of the improved Mumford-Shah model.

This result show that most of the cell boundaries are detected as wished. Some contours only give an approximation of the cell boundary and does not take on the shape of the boundary. This is of course not a desired result, but the contours do not become unstable.

The execution time of the image in Figure 49 with 436 contours is 123.9 seconds. Thus, approximately 0.28 seconds per cell is needed to locate its boundary. Calculating the mean values u_i and u_o takes up the most time, this is 91.7 seconds. Thus, roughly 70% of the total computational time is needed for calculating the mean values, which is used to determine the force. Compared to the execution time of the integration method, this is a large amount time.

If we decide to evaluate u_i and u_o only once in the Mumford-Shah phase, then the execution time will be improved. For most cells, the result will be the same, see Figure 50(a).



(a) Part of an image with 183 contours.

(b) Unstable contour.

Figure 50: Results of the improved Mumford-Shah model with evaluating u_i and u_o only once.

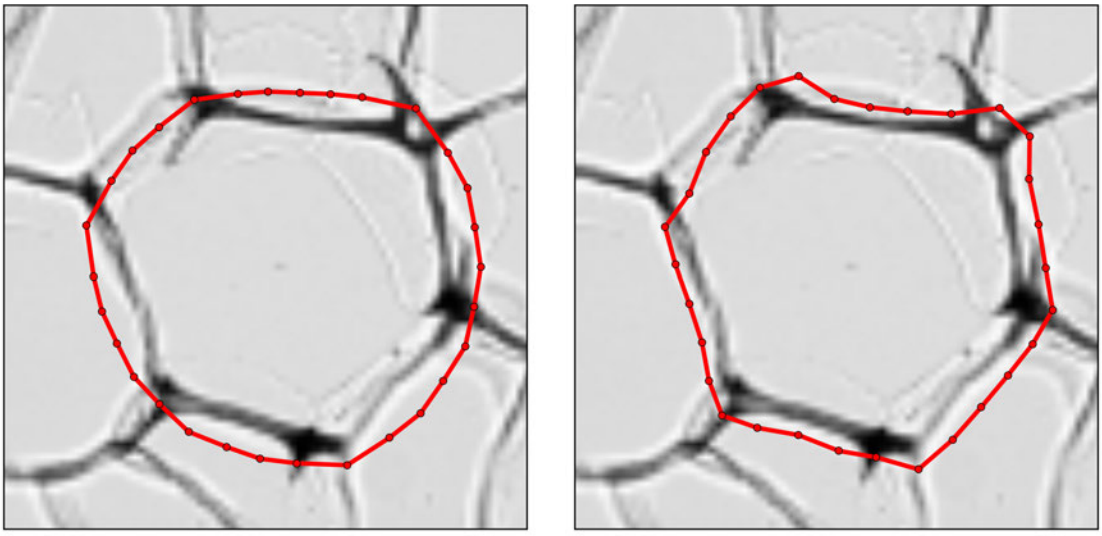
The execution time of this simulation with 183 contours is 21.7 seconds, where only 4.2 seconds is used to evaluate the mean values. Thus, we have reduced the execution time of the mean evaluation from 70% to 5.1 %. Unfortunately, this can lead to unstable results, as can be seen in Figure 50(b). We can conclude that this is not an acceptable approach for reducing the execution time.

For the next part, assume that a contour jumps past the boundary over the cell in the first phase, as in Figure 51(a). We investigate what result the second phase of the method will yield.

Note that the the Mumford-Shah model is able to work in both directions. For a point s_j on the contour in Figure 51(a), we have $u_i^j < u_o^j$, and f^j is approximately equal to u_o^j, since the boundary in contained in the areas Ω_i^j. The force then becomes;

$$\begin{aligned} \boldsymbol{F} &= [(f^j - u_o^j)^2 - (f^j - u_i^j)^2]\mathbf{n}_C \\ &\approx [(u_o^j - u_o^j)^2 - (u_o^j - u_i^j)^2]\mathbf{n}_C \\ &= [-(u_o^j - u_i^j)^2]\mathbf{n}_C. \end{aligned}$$

We see that the force is directed in the negative direction to the normal vector $\mathbf{n}_C$. Thus, this means that the force is pointed towards the boundary.



(a) Contour after the jump phase.

(b) Contour after the Mumford-Shah phase.

Figure 51: Result of the improved model, where the contour has jump past the boundary.

Even though the contour jumped past the boundary in the first phase, the second phase still manages to capture the shape of the cell on the opposite side of the boundary. Thus instead of inflating the contour in the next step, we deflate the contour;

$$r_{defl} = \boldsymbol{r} - h_{delf}\mathbf{n}. \tag{59}$$

The value for h_{defl} can be determined in a similar fashion as h_{infl} in eq. (57). In order to move the contour towards the outside edge of the boundary, we have to choose $h_{defl} = d - b/2$. Since we wish to locate the inside edge of the boundary, we have take the width b in account, then h_{delf} becomes:

$$h_{delf} = d - b/2 + b = d + b/2. \tag{60}$$

See Figure 52.

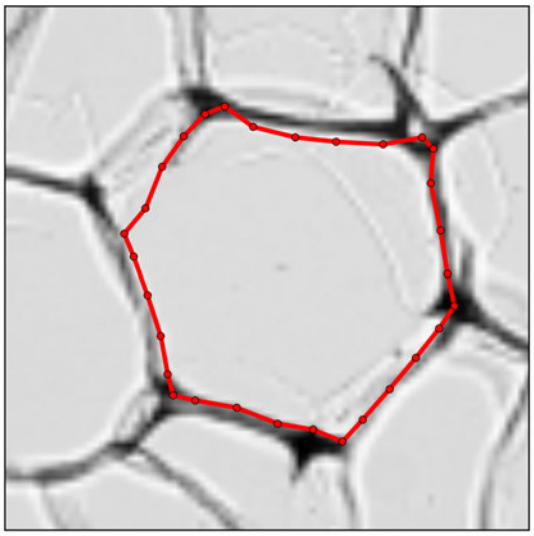
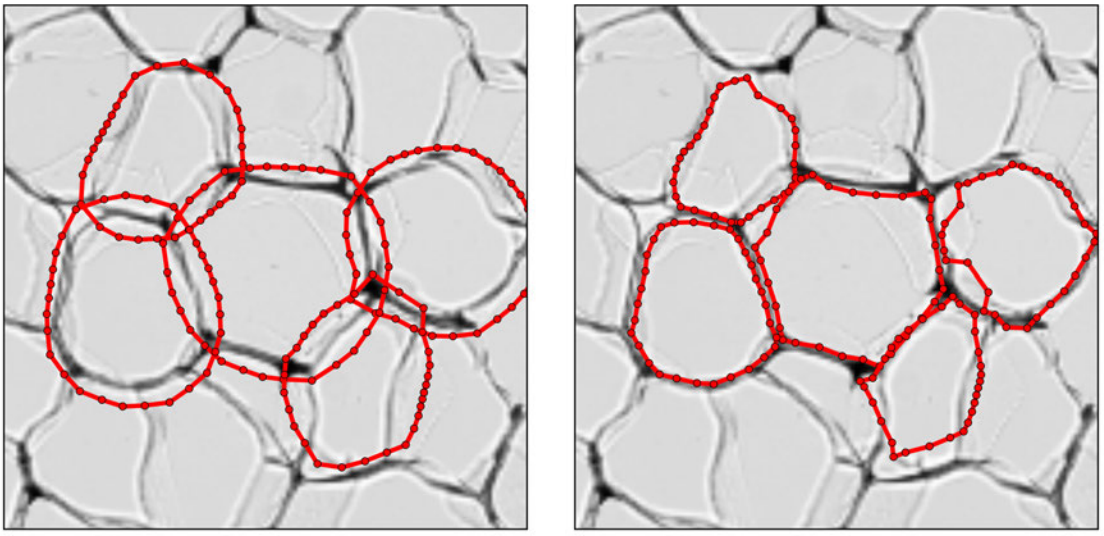


Figure 52: Result after deflation with $h_{defl} = d + b/2$.

We also simulated this for multiple cells, where we artificially moved past the boundary, see Figure 53.



(a) Contours after the jump phase.

(b) Contours after the Mumford-Shah phase and deflation.

Figure 53: Results of the improved model, where the contours have jump past the boundary.

The results show that this does not work properly for each cell, but it still gives a decent approximation of the boundary. If this situation occurs, we want to determine whether to inflate or deflate. We check this by looking at the mean value of u_i^j and u_o^j for all j. If $mean_j(u_i^j) < mean_j(u_o^j)$, then the contour deflates, otherwise it inflates.

In this section the parameters are chosen such that the simulation gives a good result. The aim is to automatically calculate the parameters beforehand for each image. This will be discussed in the next section.

7 Analytical Solution of the Simplified Mumford-Shah Model

In the previous sections we solved eq. (15) numerically with Finite Difference and Backward Euler. In this section we will analytically solve an simplified version of the Mumford-Shah model and study the time evolution of a contour by using the analytical solution.

7.1 Simplified Mumford-Shah Model

Consider the equation for minimizing the modified Mumford-Shah functional;

$$\frac{\partial \boldsymbol{r}}{\partial t} = [e^+ - e^-]\widetilde{\mathbf{n}}_C + \nu \frac{\partial^2 \boldsymbol{r}}{\partial s^2} - \kappa \frac{\partial^4 \boldsymbol{r}}{\partial s^4}, \tag{61}$$

with $\widetilde{\mathbf{n}}_C = -\mathbf{n}_C = \begin{pmatrix} y' \\ -x' \end{pmatrix}$. For the derivation of the analytic solution we use positive orientation, since this is convenient for the Fourier series. In the modified Mumford-Shah model, however, we use the negative direction, therefore we have to take $\widetilde{\mathbf{n}}_C$ for this part.

We want to study the influence of the weighting parameters ν and κ, which control the contour's tension and rigidity, respectively. This can be done by looking at the eigenvalues and eigenvectors of the problem. In order to do this, we simplify the problem, such that we have a constant external force γ instead of $[e^+ - e^-]$. This simplifies to

$$\frac{\partial \boldsymbol{x}}{\partial t} = \gamma \frac{\partial}{\partial s} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \boldsymbol{x} + \nu \frac{\partial^2 \boldsymbol{x}}{\partial s^2} - \kappa \frac{\partial^4 \boldsymbol{x}}{\partial s^4}. \tag{62}$$

7.2 Derivation of the Analytical Solution

Our goal is to write the solution in the form $\boldsymbol{x}(s,t) = \sum_n T_n(t)\boldsymbol{v}_n(s)$, then we can study the time evolution of the solution. This can be used to find suitable choices for the weighting parameters.

First we write eq. (62) as $\frac{\partial \boldsymbol{x}}{\partial t} = D\boldsymbol{x}$, where operator D is

$$D\boldsymbol{x} := \left[\begin{array}{c|c} \alpha \frac{\partial^2}{\partial s^2} - \kappa \frac{\partial^4}{\partial s^4} & \gamma \frac{\partial}{\partial s} \\ \hline -\gamma \frac{\partial}{\partial s} & \alpha \frac{\partial^2}{\partial s^2} - \kappa \frac{\partial^4}{\partial s^4} \end{array}\right] \tag{63}$$

Now, consider:

$$D\boldsymbol{v}_n = \lambda_n \boldsymbol{v}_n, \quad \boldsymbol{v}_n(s) = \begin{bmatrix} v_n(s) \\ u_n(s) \end{bmatrix}. \tag{64}$$

We assume that v_n and u_n are of the form

$$\begin{aligned} v_n(s) &= a_n \cos(2\pi n s) + b_n \sin(2\pi n s), \\ u_n(s) &= c_n \cos(2\pi n s) + d_n \sin(2\pi n s). \end{aligned} \tag{65}$$

Now we need to find the coefficient a_n, b_n, c_n and d_n. Substituting (65) into (64)-(63) yields

$$\begin{aligned} D\boldsymbol{v}_n &= \left[\begin{array}{c} \left[\left(-\nu(2\pi n)^2 - \kappa(2\pi n)^4\right) a_n + \gamma(2\pi n) d_n\right] \cos(2\pi n s) + \\ \left[\left(-\nu(2\pi n)^2 - \kappa(2\pi n)^4\right) b_n - \gamma(2\pi n) c_n\right] \sin(2\pi n s) \\ \hline \left[\left(-\nu(2\pi n)^2 - \kappa(2\pi n)^4\right) c_n - \gamma(2\pi n) b_n\right] \cos(2\pi n s) + \\ \left[\left(-\nu(2\pi n)^2 - \kappa(2\pi n)^4\right) d_n + \gamma(2\pi n) a_n\right] \sin(2\pi n s) \end{array}\right] \\ &= \lambda_n \begin{bmatrix} a_n \cos(2\pi n s) + b_n \sin(2\pi n s) \\ c_n \cos(2\pi n s) + d_n \sin(2\pi n s) \end{bmatrix} \\ &= \begin{bmatrix} \lambda_n a_n \cos(2\pi n s) + \lambda_n b_n \sin(2\pi n s) \\ \lambda_n c_n \cos(2\pi n s) + \lambda_n d_n \sin(2\pi n s) \end{bmatrix} \end{aligned} \tag{66}$$

The coefficients must be chosen, such that $D\boldsymbol{v}_n = \lambda_n \boldsymbol{v}_n$ holds. Hence, we want:

$$\begin{cases} \left(-\nu(2\pi n)^2 - \kappa(2\pi n)^4\right) a_n + \gamma(2\pi n) d_n = \lambda_n a_n \\ \left(-\nu(2\pi n)^2 - \kappa(2\pi n)^4\right) b_n - \gamma(2\pi n) c_n = \lambda_n b_n \\ \left(-\nu(2\pi n)^2 - \kappa(2\pi n)^4\right) c_n - \gamma(2\pi n) b_n = \lambda_n c_n \\ \left(-\nu(2\pi n)^2 - \kappa(2\pi n)^4\right) d_n + \gamma(2\pi n) a_n = \lambda_n d_n \end{cases}$$

This can be written as a system;

$$\begin{bmatrix} -\phi & 0 & 0 & \psi \\ 0 & -\phi & -\psi & 0 \\ 0 & -\psi & -\phi & 0 \\ \psi & 0 & 0 & -\phi \end{bmatrix} \begin{bmatrix} a_n \\ b_n \\ c_n \\ d_n \end{bmatrix} = \lambda_n \begin{bmatrix} a_n \\ b_n \\ c_n \\ d_n \end{bmatrix}, \tag{67}$$

where $\phi = \nu(2\pi n)^2 + \kappa(2\pi n)^4$ and $\psi = \gamma(2\pi n)$. Since this is a symmetric matrix, we have $\lambda_n \in \mathbb{R}$ for all eigenvalues. The eigenvectors can be easily found:

$$\boldsymbol{w}_n^{(1)} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \boldsymbol{w}_n^{(2)} = \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{w}_n^{(3)} = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \boldsymbol{w}_n^{(4)} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{68}$$

These eigenvectors gives suggestions for a_n, b_n, c_n and d_n for the eigenfunctions from (65). We are also interested in the eigenvalues of the problem (64) to study the time evolution of the contour. The eigenvalues are:

$$\begin{cases} \lambda_n^{(1)} &= -\nu(2\pi n)^2 - \kappa(2\pi n)^4 + \gamma(2\pi n) \\ \lambda_n^{(2)} &= \lambda_n^{(1)} \\ \lambda_n^{(3)} &= -\nu(2\pi n)^2 - \kappa(2\pi n)^4 - \gamma(2\pi n) \\ \lambda_n^{(4)} &= \lambda_n^{(3)}. \end{cases} \tag{69}$$

The eigenvectors from (68) imply the following eigenfunctions for the operator D:

$$\begin{aligned} \boldsymbol{v}_n^{(1)} &= \begin{bmatrix} \cos(2\pi n s) \\ \sin(2\pi n s) \end{bmatrix}, \quad \boldsymbol{v}_n^{(2)} = \begin{bmatrix} -\sin(2\pi n s) \\ \cos(2\pi n s) \end{bmatrix}, \\ \boldsymbol{v}_n^{(3)} &= \begin{bmatrix} -\cos(2\pi n s) \\ \sin(2\pi n s) \end{bmatrix}, \quad \boldsymbol{v}_n^{(4)} = \begin{bmatrix} \sin(2\pi n s) \\ \cos(2\pi n s) \end{bmatrix}. \end{aligned} \tag{70}$$

The eigenvectors $\boldsymbol{v}_n^{(1)}, \boldsymbol{v}_n^{(2)} \boldsymbol{v}_n^{(3)}$ and $\boldsymbol{v}_n^{(4)}$ are mutually orthogonal for each $n \in \mathbb{N}$. Namely,

$$\begin{aligned} \int_0^1 \sin(2\pi n s)\cos(2\pi m s) ds = 0, \quad \forall n, m = 0, 1, ..., \\ \int_0^1 \sin(2\pi n s)\sin(2\pi m s) ds = \delta_{nm} \int_0^1 \sin^2(2\pi n s) ds = \frac{1}{2}\delta_{nm}, \\ \int_0^1 \cos(2\pi n s)\cos(2\pi m s) ds = \delta_{nm} \int_0^1 \cos^2(2\pi n s) ds = \frac{1}{2}\delta_{nm}, \end{aligned}$$

and

$$\int_0^1 \sin^2(2\pi n s) ds = \int_0^1 \cos^2(2\pi n s) ds = \frac{1}{2}, \tag{71}$$

because

$$\begin{aligned} \int_0^1 \cos^2(2\pi n s) ds &= \int_0^1 \frac{1}{2}(1 + \cos(4\pi n s)) ds \\ &= \frac{1}{2}\left[s\right]_0^1 + \frac{1}{2}\left[\frac{\sin(4\pi n s)}{4\pi n}\right]_0^1 \\ &= \frac{1}{2} + \frac{1}{8}\left(\frac{\sin(4\pi n)}{4\pi n}\right) = \frac{1}{2}, \end{aligned}$$

since n is a integer. We have the following

$$\int_0^1 \boldsymbol{v}_n^{(k)}(s) \cdot \boldsymbol{v}_m^{(q)}(s) ds = \frac{1}{2} 2\delta_{nm}\delta_{kq} = \delta_{nm}\delta_{kq}, \tag{72}$$

thus $\boldsymbol{v}_n^{(k)}(s)$ and $\boldsymbol{v}_m^{(q)}(s)$ are orthogonal for each $m, n \in \mathbb{N}$ and $k, q = 1, ..., 4$. Note that

$$D\boldsymbol{v}_n^{(k)} = \lambda_n^{(k)} \boldsymbol{v}_n^{(k)}, \quad for\ k = 1, ..., 4.$$

For each $n \in \mathbb{N}$, we have 4 eigenvectors $\boldsymbol{v}_n^{(k)}$. Thus, we seek $\boldsymbol{x}(s,t)$ in the form:

$$\boldsymbol{x}(s,t) = \sum_n \sum_{k=1}^{4} T_n^{(k)}(t) \boldsymbol{v}_n^{(k)}(s). \tag{73}$$

Substitution in eq. (62) yields the following equations for $T_n^{(k)}(t)$ (after projection):

$$\begin{aligned} \frac{dT_n^{(k)}}{dt} &= \lambda_n^{(k)} T_n^{(k)}, \quad k = 1, ..., 4;\ n = 0, 1, ... \\ T_n^{(k)} &= T_n^{(k)}(0) e^{\lambda_n^{(k)} t}. \end{aligned} \tag{74}$$

The initial conditions are being used to determine $T_n^{(k)}(0)$, that is;

$$\boldsymbol{x}(s,0) = \boldsymbol{x}_0(s),$$

with

$$\boldsymbol{x}(s,t) = \sum_n \sum_{k=1}^{4} T_n^{(k)}(0) e^{\lambda_n^{(k)} t} \boldsymbol{v}_n^{(k)}(s). \tag{75}$$

At $t = 0$ we have

$$\boldsymbol{x}(s,0) = \boldsymbol{x}_0(s) = \sum_n \sum_{k=1}^{4} T_n^{(k)}(0) \boldsymbol{v}_n^{(k)}(s). \tag{76}$$

Projection yields the following expression for $T_n^{(k)}(0)$:

$$T_n^{(k)}(0) = \int_0^1 \boldsymbol{x}_0(s) \cdot \boldsymbol{v}_n^{(k)}(s) ds, \quad k = 1, ..., 4;\ n = 0, 1, ... \tag{77}$$

This gives us the analytical solution for the simplified problem.

7.3 Implementation

In the next part we implement the Fourier series (75), such that the time evolution of the contour can be studied. In order to calculate $T_n^{(k)}(0)$ for each k and n, we have to evaluate the integral in (77). For the approximation of the definite integral we use the numerical integration technique trapezoidal rule, which is given by

$$\int_0^1 f(x) dx \approx \sum_{j=1}^{M} \frac{f(x_{j-1}) + f(x_j)}{2} \Delta x_j. \tag{78}$$

We want to evaluate the following integral:

$$\int_0^1 \boldsymbol{x}_0(s) \cdot \boldsymbol{v}_n^{(k)}(s) ds = \int_0^1 x_0(s) v_n^{(k)}(s) + y_0(s) u_n^{(k)}(s) ds. \tag{79}$$

Now, let $g_n^{(k)}(s) = x_0(s)v_n^{(k)}(s) + y_0(s)u_n^{(k)}(s)$, then we can approximate (79) as

$$\int_0^1 \boldsymbol{x}_0(s) \cdot \boldsymbol{v}_n^{(k)}(s)ds \approx \sum_{j=1}^{M} \frac{g_n^{(k)}(s_{j-1}) + g_n^{(k)}(s_j)}{2} \Delta s_j. \tag{80}$$

For the numerical integration we use M equidistant points $s_j = \frac{j-1}{M}$ for $j = 1, ..., M$. For the series from eq. (75) we calculate up to P terms. Thus,

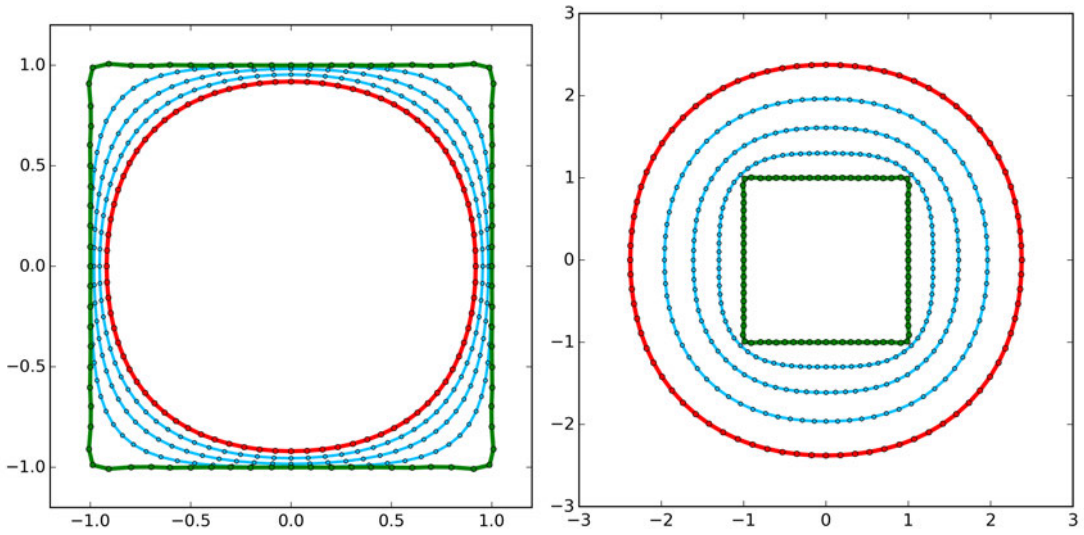
$$\widetilde{\boldsymbol{x}}(s,t) = \sum_{n}^{P} \sum_{k=1}^{4} T_n^{(k)}(0) e^{\lambda_n^{(k)} t} \boldsymbol{v}_n^{(k)}(s), \tag{81}$$

where $\widetilde{\boldsymbol{x}}(s,t)$ is the approximation of $\boldsymbol{x}(s,t)$.

7.4 Results

We simulate this with $N = 80$ points, $M = N$ and $P = 25$. The initial contour is a square rectangle $[-1,1] \times [-1,1]$. For the figures in this section, the initial contour $\widetilde{\boldsymbol{x}}(s,0)$ is given by the green line, the contour $\widetilde{\boldsymbol{x}}(s,5.0)$ at $t = 5.0$ by the red line and the contours $\widetilde{\boldsymbol{x}}(s,\tau)$, for $\tau = 1.0, 2.0, 3.0, 4.0$ by the blue lines, unless indicated otherwise. If a parameter is not explicitly stated, then its value is zero.

In Figure 54 the solution is shown for the model with $\nu = 0.001$ and for the model where we have an external force $\gamma = 0.03$.



(a) Solutions with $\nu = 0.001$.

(b) Solutions with $\nu = 0.001$ and $\gamma = 0.03$.

Figure 54: Solutions of the model without and with an external force.

We find that the stretching force makes the contour shrink as can be seen in Figure 54(a), but if we add a large enough external force, the contour grows, see Figure 54(b). Note that the contour slowly transforms into a circle.

For the next model we also make use of the bending force with the weighting parameter $\kappa = 0.0001$. The result can be seen in Figure 55.

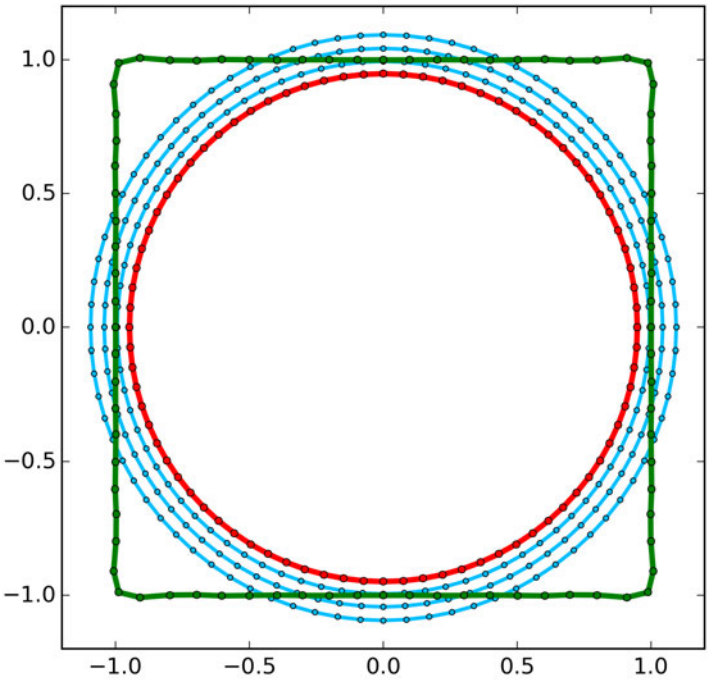
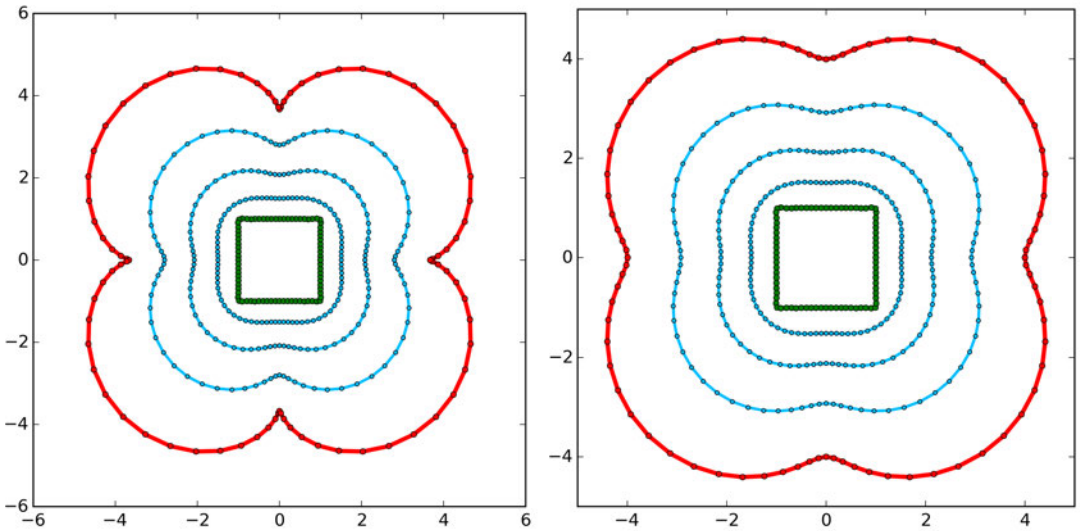


Figure 55: Solutions with $\nu = 0.001, \kappa = 10^{-4}$ and $\gamma = 0.03$.

This result shows that the contour first transforms into a circle and eventually starts shrinking. This is caused by the internal bending force, which prevents the contour from bending too much. Both forces share the same properties; preventing the contour from stretching and bending, but the stretching force has more influence on the contour's tension and the bending force on the contour's rigidity.

For the next model we increase the external force $\gamma = 0.05$, which can be seen in Figure 56.



(a) Solutions with $\nu = 0.001$ and $\gamma = 0.05$.

(b) Solutions with $\nu = 0.001, \kappa = 10^{-7}$ and $\gamma = 0.05$.

Figure 56: Solutions of the model without and with an internal bending force.

The solution from Figure 56 grows faster at the corners towards the outer direction. This situation occurs, since the normal vector $\mathbf{n}_C$ from our model is not normalized, which results in larger normal

vectors at the corners. This leads to protrusions and inward spikes in the contour. In order to prevent the emergence of protrusions and inward spikes, the bending force can be used, see Figure 56(b).

If the external force is too large compared to the internal forces, then loops will occur in the solution. The spikes will eventually turn into loops, because the external force is pointed in the direction of the normal vector. The points of a spike in the contour will have normal vectors pointed towards each other, and this results in points crossing past each other, which creates a loop. This can be seen in Figure 57.

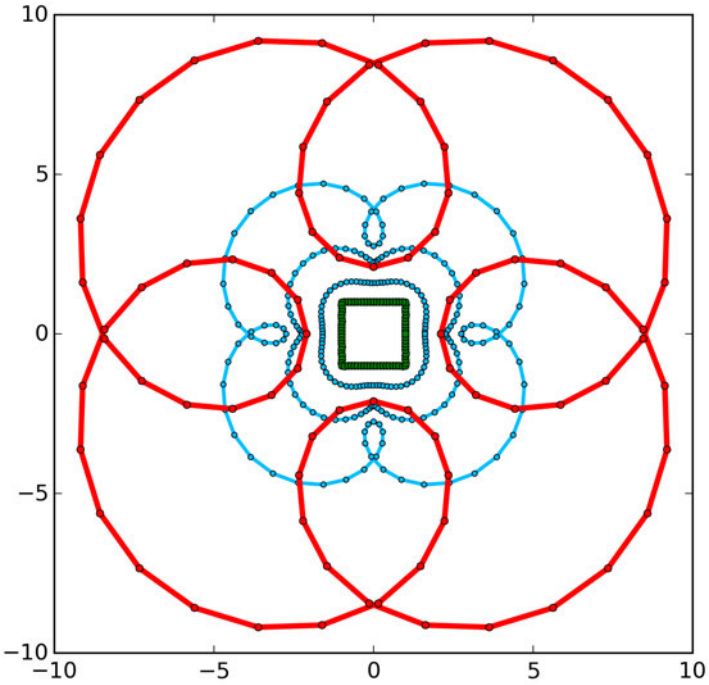


Figure 57: Solution at $t = 10.0$ given by the red line, with $\nu = 0.001$ and $\gamma = 0.06$.

7.5 Parameters for a Steady Contour

We are interested in the parameter for which the contour is steady, that is; the contour does not move for $t > t_{st}$ for a certain $t_{st} > 0$. We study the eigenvalues of the problem to get a clearer understanding of its behavior. The eigenvalues of the problem are:

$$\begin{cases} \lambda_n^{(1)} &= -\nu(2\pi n)^2 - \kappa(2\pi n)^4 + \gamma(2\pi n) \\ \lambda_n^{(2)} &= \lambda_n^{(1)} \\ \lambda_n^{(3)} &= -\nu(2\pi n)^2 - \kappa(2\pi n)^4 - \gamma(2\pi n) \\ \lambda_n^{(4)} &= \lambda_n^{(3)} \end{cases} \tag{82}$$

We assume that $\gamma > 0$, and we already have $\nu, \kappa \geq 0$. Note that

$$\lambda_n^{(k)} < \lambda_1^{(k)}, \quad for\ n > 1,\ k = 1, ..., 4. \tag{83}$$

Hence, if we obtain $\lambda_1^{(k)} = 0$ by choosing the parameters correctly, then the ground mode of the contour will not grow. We investigate for which parameters this holds by looking at the eigenvalues for $n = 1$. Since, $\lambda_1^{(3)}$ and $\lambda_1^{(4)}$ are always negative, because $\nu, \kappa, \gamma > 0$, we do not have to consider the eigenvalues for $k = 3, 4$, we only have to deal with $\lambda_1^{(1)}$ and $\lambda_1^{(2)}$.

Consider $n = 1$; ν and κ have to be chosen, such that

$$\lambda_1^{(1)} = \lambda_1^{(2)} = -\nu(2\pi)^2 - \kappa(2\pi)^4 + \gamma(2\pi) = 0. \tag{84}$$

This is an equation with two unknowns ν and κ, thus we have infinite many solutions. We choose ν and determine which value for κ has to be chosen, such that eq. (84) holds. Thus, we get

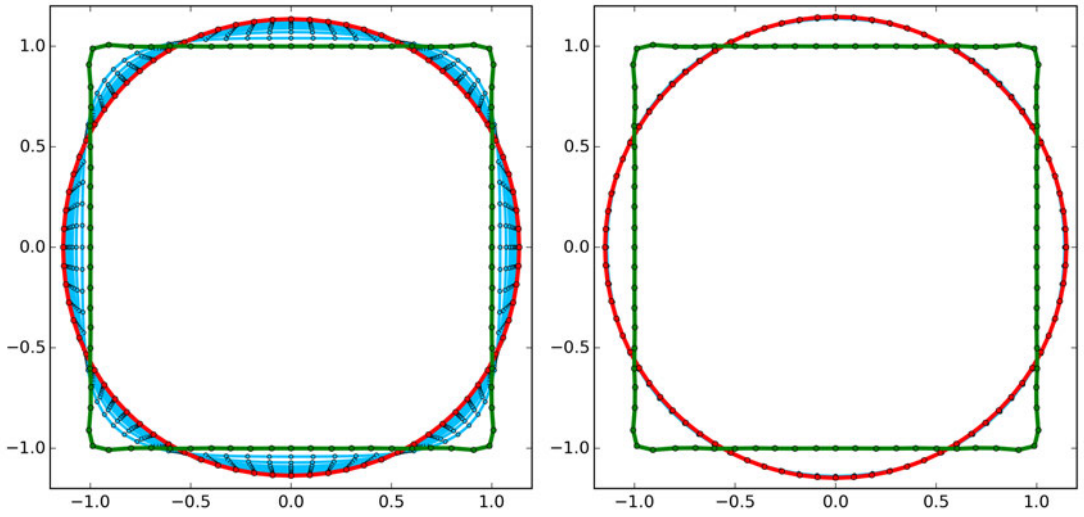
$$\begin{aligned} -\kappa(2\pi)^4 = \nu(2\pi)^2 - \gamma(2\pi) \iff \\ \kappa = \frac{\gamma}{(2\pi)^3} - \frac{\nu}{(2\pi)^2}. \end{aligned} \tag{85}$$

This gives us an expression for κ. If ν is small, then κ has to be large and vice versa. This is to be expected, since both internal forces has a similar influence on the contour, and the internal forces are needed to stop the contour from growing too much. Since $\kappa > 0$ must hold, not every choice for ν is valid. Stating this condition for κ yields the condition for ν:

$$\begin{aligned} \kappa = \frac{\gamma}{(2\pi)^3} - \frac{\nu}{(2\pi)^2} > 0 \iff \\ \frac{\gamma}{2\pi} - \nu > 0 \iff \\ \nu < \frac{\gamma}{2\pi}. \end{aligned} \tag{86}$$

Hence, ν can not be too large.

In Figure 58 the solution is shown, where κ is determined as in eq. (85).



(a) Solution for $t = 10.0$.

(b) Solution for $t = 100.0$.

Figure 58: Results of the jump phase with small areas u_o^j for each point.

We find that the solution converges to a steady state; a circle. The terms from eq. (75) for $n > 1$ are diminished, such that only $\widetilde{\boldsymbol{x}}(s,t) = \sum_{k=1}^{4} T_1^{(k)}(0) e^{\lambda_1^{(k)} t} \boldsymbol{v}_1^{(k)}(s)$ remains, which describes the circle from Figure 58.

The analytic solution of the simplified Mumford-Shah model gives insight in the behavior of the model and the influence of the parameters. This can be used as a guideline to determine the parameters of the Mumford-Shah model.

7.6 Choosing the Parameters

In the first phase of the improved Mumford-Shah model from Section 6.3.1 we used a constant force (51), just like in the simplified Mumford-Shah model. We jumped with step size h_{jump} in the direction of

the unit normal vector to the contour. However, for the simplified model, the normal vector $\mathbf{n}_C$ is not normalized. For this part we want to use the $\mathbf{n}_C$, but in order to keep the same step size, we first have to find the value $\widetilde{h}$, such that $\widetilde{h}\|\mathbf{n}_C\| = h_{jump}\|\mathbf{n}\|$, where $\mathbf{n}$ is the unit normal vector to the contour and $\mathbf{n}_C$ is the normal vector defined in eq. (13).

Consider the initial contour C, which is a circle, consisting of N points. First we calculate $\mathbf{n}_C = \begin{pmatrix} y' \\ -x' \end{pmatrix}$. Consider the points s_j of the contour with $\boldsymbol{r}_j = (x_j, y_j) = (R\cos(\theta_0), R\sin(\theta_0))$ for $j = 1, ..., N$, where $\theta_j = \frac{2\pi(j-1)}{N}$ and R is the radius of the initial contour. Without loss of generality, since the initial contour is rotationally symmetrical, we can consider the point $\boldsymbol{r}_0 = (R\cos(\theta_0), R\sin(\theta_0))$. For this point we calculate y_0' and $-x_0'$. See Figure 59.

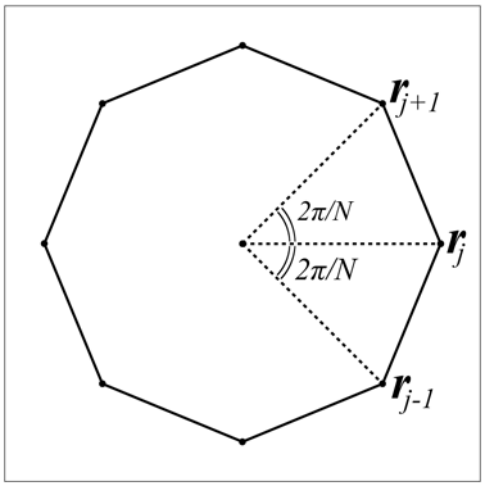


Figure 59: Initial contour with 8 points.

For x_j' and y_j' we use the first order central difference formula;

$$\widetilde{x}_j' = \frac{x_{j+1} - x_{j-1}}{2},$$

where $\widetilde{x}$ is the numerical approximation of x. Thus, we have

$$\begin{aligned} \widetilde{x}_0' &= \frac{\widetilde{x}_1 - \widetilde{x}_N}{2} \\ &= \frac{R\cos(\frac{2\pi}{N}) - R\cos(-\frac{2\pi}{N})}{2} \\ &= \frac{R\cos(\frac{2\pi}{N}) - R\cos(\frac{2\pi}{N})}{2} = 0. \end{aligned}$$

And for y', we have

$$\begin{aligned} \widetilde{y}_0' &= \frac{\widetilde{y}_1 - \widetilde{y}_N}{2} \\ &= \frac{R\sin(\frac{2\pi}{N}) - R\sin(-\frac{2\pi}{N})}{2} \\ &= \frac{2R\sin(\frac{2\pi}{N})}{2} = R\sin\left(\frac{2\pi}{N}\right). \end{aligned}$$

Then, we get

$$\begin{aligned}\|(\mathbf{n}_C)_0\| &= \sqrt{(\widetilde{x}_0')^2 + (-\widetilde{y}_0')^2} \\ &= \sqrt{0^2 + (-\widetilde{y}_0')^2} \\ &= \sqrt{(-\widetilde{y}_0')^2} = R\sin\left(\frac{2\pi}{N}\right).\end{aligned}$$

This gives us, using $\|\mathbf{n}\| = 1$ since $\mathbf{n}$ is a unit normal vector, the following:

$$\begin{aligned}\widetilde{h}\|\mathbf{n}_C\| = h_{jump}\|\mathbf{n}\| \implies \\ \widetilde{h} = \frac{h_{jump}}{\|\mathbf{n}_C\|} \implies \\ \widetilde{h} = \frac{h_{jump}}{R\sin\left(\frac{2\pi}{N}\right)}.\end{aligned}$$

Now we can apply this to the simplified model from (62), using $\gamma = \widetilde{h}$. With the expression for κ, we can find ν and κ, such that the contour is steady. We choose to take $\kappa = \nu$, such that the stretching force weights the same as the bending force. Substituting this in eq. (85) yields

$$\begin{aligned}\nu = \frac{\gamma}{(2\pi)^3} - \frac{\nu}{(2\pi)^2} \iff \\ \frac{\nu[1+(2\pi)^2]}{(2\pi)^2} = \frac{\gamma}{(2\pi)^3} \iff \\ \nu = \frac{\gamma}{2\pi[1+(2\pi)^2]}.\end{aligned} \tag{87}$$

In the improved Mumford-Shah model we used radius $R = 5.0$ for the initial contour with $N = 15$ points and step size $h_{jump} = 2.0$. Now we can finally calculate the weighting parameters ν and κ;

$$\begin{aligned}\nu &= \frac{\widetilde{h}}{2\pi[1+(2\pi)^2]} \\ &= \frac{\frac{h_{jump}}{R\sin\left(\frac{2\pi}{N}\right)}}{2\pi[1+(2\pi)^2]} \approx 0.0048.\end{aligned}$$

Thus, we get $\nu = \kappa = 0.0048$. For the jump phase of the improved model we have used $\nu_{jump} = \kappa_{jump} = 0.005$ and this indeed led to a good result.

For the second phase of the improved Mumford-Shah model, we have a different force than in the first phase; $[e^+ - e^-]\mathbf{n}_C$. This external force is in the direction of the normal vector $\mathbf{n}_C$, however, $[e^+ - e^-]$ is not constant. In order to insure that the contour does not stretch and bend too much, we use the upper estimate of the force.

Since the image of the cell can be seen as a function $f : \Omega \to [0,1]$, we have $u_i, u_o, f^j \in [0,1]$ for every point s_j on the contour. Thus, we yield

$$\begin{aligned}\left|[e^+ - e^-]\right| &= \left|(f^j - u_o)^2 - (f^j - u_i)^2\right| \\ &\leq \left|(f^j - 0)^2 - (f^j - 1)^2\right| \\ &= \left|(f^j)^2 - (f^j)^2 + 2f^j - 1\right| \\ &= \left|2f^j - 1\right| \leq 1,\end{aligned}$$

since $f^j \in [0, 1]$.

Now, we take $\gamma = 1.0$, then we get from eq. (87);

$$\begin{aligned}\nu &= \frac{\gamma}{2\pi[1 + (2\pi)^2]} \\ &= \frac{1.0}{2\pi[1 + (2\pi)^2]} \approx 0.0039,\end{aligned}$$

and thus $\nu_{MS} = \kappa_{MS} = 0.0039$.

We simulated the improved Mumford-Shah model with the same parameters as in Section 6.3.4, except for the internal weighting parameters ν and κ found in this section. The result can be seen in Figure 60.

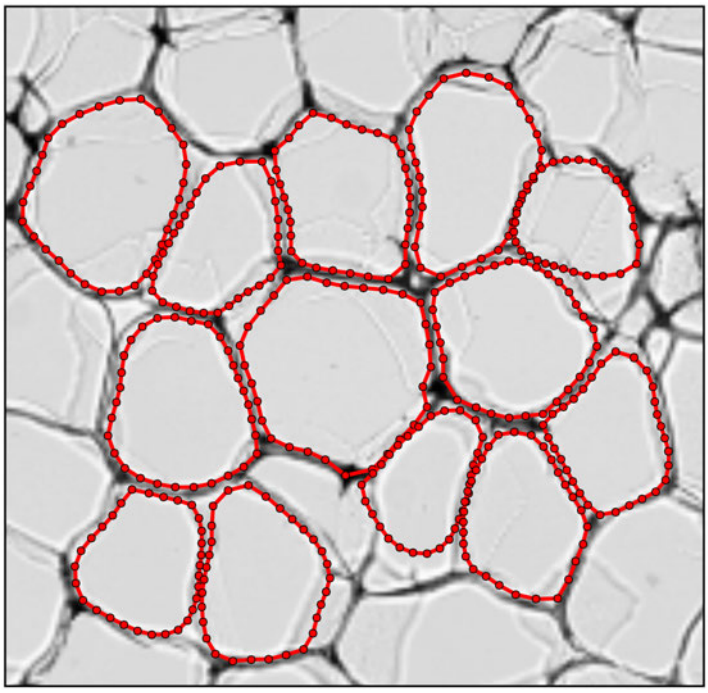


Figure 60: Results from the improved Mumford-Shah model with the weighting parameters of this section.

The weighting parameters give a decent result, however, ν_{MS} and κ_{MS} can be chosen smaller.

8 Conclusion

We discussed the modified Mumford-Shah functional, from which we derived an equation by determining the Euler-Lagrange equation and applying gradient descent in order to minimize the functional. This equation has been fully discretized in space and time, which yields a system to move the contour under the influences of the internal and external force.

Techniques from graph theory have been applied to determine the number of cells; this helped us to locate the centers of the cells. We tried improving the computational time by converting the image into a binary image. Although this operation improved the computational time, the results were less accurate caused by noise inside the cells. An image filtering method has been applied to remove the noise. This enhanced the performance of the Mumford-Shah model with the binary image, but it could occur that this filtering method removed parts of the boundary, which led to contours passing through the boundaries.

We introduced a new internal force, which also have been applied in other deformable contour models. This discourages bending of the contour and proved to be a useful addition to our model.

We presented an improved Mumford-Shah model, which consists of two phases; the jump phase and the Mumford-Shah phase. The first phase improved the execution time by quickly approximating the location of the boundary of a cell. The latter phase adapted the approximation into an accurate result. We modified the evaluation of the force in order to get a more stable result. In the last step of the model, the contour performed an inflation to reach the edge of the boundary.

An simplified version of the Mumford-Shah model has been discussed, from which we derived the analytical solution. This was used to study the influences of the stretching and bending parameters on the contour. The time evolution of the contour was also investigated to understand the model and to determine suitable choices for the parameters.

9 Discussion

The modified Mumford-Shah models performs as expected; the initial contour will eventually reach the boundary, but this takes a long time. This gives the suspicion that the model is not appropriate for this kind of problem. That is; the cartoon model without the gradient term in the functional, and where we determined u_i and u_o as in eq. (6). For next studies, the Mumford-Shah functional with extra gradient term can be considered. However, this operational will also increase the computational time per time step, but perhaps this model allows us to take less steps in order to get the desired results, such that the total computational time will be improved.

For the improved Mumford-Shah model we have used a jump condition to move the contour quickly to the boundary; this gives us an approximation of the boundary. A similar approach has been applied to the balloon model [4], where the contour is inflated such that it moves towards the boundary until it reaches the boundary. Thus this model is likely to give a decent result for this problem in a small amount of time. The balloon model should be considered and examined in order to verify its accuracy.

As stated before, the main problem with the Mumford-Shah model is the computational time. We have tried solving this problem by improving the model and image filtering, however, the code of the program can also be optimized. It can be improved by using sparse matrices, an alternate approach for evaluating inverse matrices and a different way for calculating mean values. Adapting the code, such that it can be executed parallel instead of sequential will also improve the computational time.

References

[1] Mumford, D.B. & Shah, J. (1989). Optimal approximations by piecewise smooth functions and associated variational problems. *Communications on Pure and Applied Mathematics 42(5): 577-685.* doi=10.1002/cpa.3160420503

[2] Cremers, D., Schnörr, C., & Weickert, J. (2001). Diffusion-Snakes: Combining Statistical Shape Knowledge and Image Information in a Cariational Framework. *Variational and Level Set Methods in Computer Vision, 13: 137-144.* doi=10.1109/VLSM.2001.938892

[3] Bar, L., Chan, T.F., Chung, G., Jung, M., Kiryati, M., Mohieddine, R., Sochen, N. & Vese, L.A. (2011). *Mumford and Shah Model and its Applications to Image Segmentation and Image Restoration, Handbook of Mathematical Methods in Imaging.* doi=10.1007/978-0-387-92920-0_25

[4] Cohen, L.D. (1991). On active contour models and balloons. *CVGIP: Image Understanding, 53(2): 211 - 218.* doi=10.1016/1049-9660(91)90028-N

[5] Chenyang, X., Prince, J. (1997). Gradient Vector Flow: A New External Force for Snakes. *Proc Cvpr Ieee, 07: 66 - 71.*

[6] Fialka, O., & Cadik, M. (2006). FFT and Convolution Performance in Image Filtering on GPU. *In Proceedings of the Tenth International Conference on Information Visualisation, 08: 609 - 614.*

[7] Diestel, R. (2017). *Graph Theory.* doi=10.1007/978-3-662-53622-3

A Code

Below a python script that performs the improved Mumford-Shah model described in Section 6.3.

```
import numpy as np
import scipy.misc
from scipy import ndimage
import scipy.ndimage.filters as filters
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from PIL import Image, ImageDraw
import time

#Reads image
def Read(img):
    return scipy.misc.imread(img, 'L')

#Scales the image to valus in [0,1]
def Normalize(I):
    return (I-np.min(I))/(np.max(I)-np.min(I))

#Crops image
def Crop(I,h_l,h_r,v_t,v_b):
    return I[v_t:v_b,h_l:h_r]

#Down-samples the image
#by evaluating the mean value of a block of size factxfact
def downsamplingMean(ar_orig, fact):
    sx_orig, sy_orig = ar_orig.shape
    sx = sx_orig - sx_orig%fact
    sy = sy_orig - sy_orig%fact
    ar = ar_orig[0:sx,0:sy]
    assert isinstance(fact, int), type(fact)
    X, Y = np.ogrid[0:sx, 0:sy]
    regions = sy/fact * (X/fact) + Y/fact
    res = ndimage.mean(ar, labels=regions, index=np.arange(regions.max() + 1))
    res.shape = (sx/fact, sy/fact)
    return res

#Applies Gaussian blur filter
def Gaussianfilter(I, sigma):
    return ndimage.gaussian_filter(I, sigma=sigma)

#Determines the location of the maxima (or cell centers)
#far enough away (bdr) from the edge
def croppedMaxima(I,sigma,bdr):
    m,n = I.shape
    x_list,y_list = findMaxima(I,sigma)
    x_list_crop,y_list_crop = [],[]
    for i in range(len(x_list)):
        if (bdr<x_list[i]<=n-bdr) and (bdr<y_list[i]<=m-bdr):
            x_list_crop.append(x_list[i])
            y_list_crop.append(y_list[i])
    return x_list_crop,y_list_crop

#Finds the location of the maxima (or cell centers)
```

```
def findMaxima(I,sigma):
    neighborhood_size = 10
    threshold = 0.8
    I_gauss = Gaussianfilter(I,sigma)
    I_gauss_min = filters.minimum_filter(I_gauss, neighborhood_size)
    I_gauss_max = filters.maximum_filter(I_gauss, neighborhood_size)
    maxima = (I_gauss == I_gauss_max)
    diff = ((I_gauss_max - I_gauss_min) < threshold)
    maxima[diff == 0] = 0

    labeled, num_objects = ndimage.label(maxima)
    slices = ndimage.find_objects(labeled)
    x, y = [], []
    for dy,dx in slices:
        x_center = (dx.start + dx.stop - 1)/2
        x.append(x_center)
        y_center = (dy.start + dy.stop - 1)/2
        y.append(y_center)
    return x,y

#Determines the location of the minima (or cell vertices)
#far enough away (bdr) from the edge
def croppedMinima(I,sigma,bdr):
    m,n = I.shape
    x_list,y_list = findMinima(I,sigma)
    x_list_crop,y_list_crop = [],[]
    for i in range(len(x_list)):
        if (bdr<x_list[i]<=n-bdr) and (bdr<y_list[i]<=m-bdr):
            x_list_crop.append(x_list[i])
            y_list_crop.append(y_list[i])
    return x_list_crop,y_list_crop

#Finds the location of the minima (or cell vertices)
def findMinima(I,sigma):
    neighborhood_size = 10
    threshold = 0.8
    I_gauss = Gaussianfilter(I,sigma)
    I_gauss_min = filters.minimum_filter(I_gauss, neighborhood_size)
    I_gauss_max = filters.maximum_filter(I_gauss, neighborhood_size)
    minima = (I_gauss == I_gauss_min)
    diff = ((I_gauss_max - I_gauss_min) < threshold)
    minima[diff == 0] = 0

    labeled, num_objects = ndimage.label(minima)
    slices = ndimage.find_objects(labeled)
    x, y = [], []
    for dy,dx in slices:
        x_center = (dx.start + dx.stop - 1)/2
        x.append(x_center)
        y_center = (dy.start + dy.stop - 1)/2
        y.append(y_center)
    return x,y

#Matrix for the second derivative
def matrixA(N):
```

```
    A = np.zeros([N,N])
    for i in range(1,N-1):
        A[i,i] = -2
        A[i,i+1] = 1
        A[i,i-1] = 1
    A[0,0] = A[N-1,N-1] = -2
    A[0,1] = A[0,N-1] = A[N-1,0] = A[N-1,N-2] = 1
    return A

#Matrix for the first derivative
def matrixB(N):
    B = np.zeros([N,N])
    for i in range(1,N-1):
        B[i,i-1] = -1
        B[i,i+1] = 1
    B[N-1,0] = B[0,1] = 1
    B[N-1,N-2] = B[0,N-1] = -1
    return 0.5*B

#Matrix for the n(egative) first derivative
def matrixC(N):
    C = np.zeros([N,N])
    for i in range(1,N-1):
        C[i,i-1] = 1
        C[i,i+1] = -1
    C[N-1,0] = C[0,1] = -1
    C[N-1,N-2] = C[0,N-1] = 1
    return 0.5*C

#Matrix for the fourth derivative
def matrixD(N):
    D = np.zeros([N,N])
    for i in range(2,N-2):
        D[i,i] = 6
        D[i,i-1] = D[i,i+1] = -4
        D[i,i-2] = D[i,i+2] = 1
    D[0,0] = D[N-1,N-1] = D[1,1] = D[N-2,N-2] = 6
    D[0,1] = D[0,N-1] = D[N-1,0] = D[N-1,N-2] = -4
    D[1,0] = D[1,2] = D[N-2,N-1] = D[N-2,N-3] = -4
    D[0,2] = D[0,N-2] = D[N-1,1] = D[N-1,N-3] = 1
    D[1,3] = D[1,N-1] = D[N-2,0] = D[N-2,N-4] = 1
    return D

#Matrix for the external force for u_plus_arr (array) and u_min (integer)
def matrixE(xy_arr,u_min,u_plus_arr,I,d):
    x_arr = xy_arr[:len(xy_arr)/2]
    y_arr = xy_arr[len(xy_arr)/2:]
    Ediag = []
    m,n = I.shape
    for i in range(len(x_arr)):
        x_elt = int(round(x_arr[i]))
        y_elt = int(round(y_arr[i]))
        f = np.mean(I[y_elt-d:y_elt+d,x_elt-d:x_elt+d])
        e_plus = (f-u_plus_arr[i])**2
        e_min = (f-u_min)**2
```

```
        Ediag.append((e_plus-e_min))
    E = np.diagflat(Ediag)
    return E,Ediag

#Matrix for the external force for u_plus_arr (array) and u_min_arr (array)
def matrixE2(xy_arr,u_min_arr,u_plus_arr,I,d):
    x_arr = xy_arr[:len(xy_arr)/2]
    y_arr = xy_arr[len(xy_arr)/2:]
    Ediag = []
    for i in range(len(x_arr)):
        x_elt = int(round(x_arr[i]))
        y_elt = int(round(y_arr[i]))
        f = np.mean(I[y_elt-d:y_elt+d,x_elt-d:x_elt+d])
        e_plus = (f-u_plus_arr[i])**2
        e_min = (f-u_min_arr[i])**2
        Ediag.append((e_plus-e_min))
    E = np.diagflat(Ediag)
    return E,Ediag

#Give the initial contour; a circle of N points with radius 'radius'
def initContour(x_middle,y_middle,radius,N):
    x_start = np.zeros(N)
    y_start = np.zeros(N)
    for i in range(N):
        x_start[i] = x_middle + radius * np.cos(i/float(N)*2*np.pi)
        y_start[i] = y_middle + radius * np.sin(-i/float(N)*2*np.pi)
    xy_start = np.concatenate((x_start,y_start))
    return xy_start

#Makes an array periodic
def makePeriodic(arr):
    arr_per = np.insert(arr,0,arr[-1])
    return arr_per

#Finds the average value between two values of an array
def findAverage(arr):
    arr_per = makePeriodic(arr)
    arr_avg = np.zeros(len(arr))
    for i in range(len(arr_avg)):
        arr_avg[i] = (arr_per[i]+arr_per[i+1])/2.0
    return np.insert(arr_avg,len(arr_avg),arr_avg[0])

#Finds a suitable choice for sigma
def findSigma(I):
    threshold = 10
    bdr = 5
    maxima_list = []
    minima_list = []
    sigma_list = []

    for i in np.arange(1,20,1):
        x_max,y_max = croppedMaxima(I,float(i),bdr)
        sigma_list.append(float(i))
        maxima_list.append(len(x_max))
```

```
        IGauss = Gaussianfilter(I,float(i)/2.0)
        I_fil = Normalize(filters.minimum_filter(IGauss,threshold/2.0))
        xl,yl = croppedMinima(I_fil,0.0,bdr)
        minima_list.append(2+0.5*len(xl))

    maxima_list = np.array(maxima_list)
    minima_list = np.array(minima_list)

    der2 = np.zeros(len(maxima_list)-2)
    s_arr = np.zeros_like(der2)

    valid = []
    for u in range(0,len(maxima_list)-2):
        s_arr[u] = u+1
        der2[u] = maxima_list[u+2]-2*maxima_list[u+1]+maxima_list[u]
    for r in range(len(s_arr)):
        if der2[r]<20:
            valid.append(s_arr[r])
    validmin = np.min(valid)
    validmax = np.max(valid)
    sigma_list = np.array(sigma_list)
    plt.figure()
    plt.plot(sigma_list,maxima_list,'forestgreen',linewidth=3,
             label='Cell Center method')
    plt.plot(sigma_list,minima_list, 'darkorange',linewidth=3,
             label='Cell Vertex method')
    idx = np.argwhere(np.diff(np.sign(maxima_list - minima_list)) != 0)\
    .reshape(-1) + 1
    idxvalid = []
    for idxelt in idx:
        if validmin<=sigma_list[idxelt]<=validmax:
            idxvalid.append(sigma_list[idxelt])
    sigmacorrect = np.mean(idxvalid)
    plt.plot(sigma_list[idx], maxima_list[idx], 'ro',markersize=5)
    plt.legend(loc='upper right')
    plt.xlabel('sigma')
    plt.ylabel('Number of cell centers')
    plt.show()

    return sigmacorrect

#Finds a suitable threshold for converting to a binary image
def findIthreshold(arr,b):
    plt.figure()
    n, bins, patches = plt.hist(arr.flatten(),np.arange(0,1.0+b,b),
                                facecolor='green',alpha=0.5)
    plt.show()

    der2 = []
    for i in range(0,len(n)-2):
        f2 = 1/(b**2)*(n[i+2]-2*n[i+1]+n[i])
        der2.append(f2)
    bins2 = bins[0:-3]

    min_der2 = np.min(der2)
```

```
    max_der2 = np.max(der2)
    optimal = bins2[np.argmax(der2)-1]
    plt.figure()
    plt.plot([optimal+b/2.0,optimal+b/2.0],[min_der2,max_der2],'--r',
             linewidth=2.0)
    plt.plot(bins2,der2,linewidth=2.0)
    plt.ylim([min_der2,max_der2])
    plt.ylabel('Second derivate of frequency')
    plt.xlabel('Intensity')
    plt.show()
    return optimal

#Finds the average width of a boundary
def findWidthBoundary(Ncell,m,n,rho,bdr):
    return (1-np.sqrt(rho))*np.sqrt(((m-bdr)*(n-bdr))/float(Ncell))

#Finds the average diameter of a boundary
def findDiameter(Ncell,m,n,bdr):
    return np.sqrt(((m-bdr)*(n-bdr))/float(Ncell))

#Converts image to a binary image
def binary_image(I,Itr):
    I_bin = np.zeros_like(I)
    I_bin[I > Itr] = 1
    return I_bin

#Evaluates the mean values of the jump phase
def MeanJump(xy_arr,x_out_orig,y_out_orig,I,h,xmin,ymin):
    xy_arr_copy = np.array(xy_arr,copy=True)
    x_out = np.array(x_out_orig,copy=True)
    y_out = np.array(y_out_orig,copy=True)
    x_arr = xy_arr[:len(xy_arr_copy)/2]-xmin
    y_arr = xy_arr[len(xy_arr_copy)/2:]-ymin
    x_out -= xmin
    y_out -= ymin

    x_arr_avg = findAverage(x_arr)
    y_arr_avg = findAverage(y_arr)
    x_out_avg = findAverage(x_out)
    y_out_avg = findAverage(y_out)

    m,n = I.shape

    u_plus_arr = np.zeros(len(x_arr))
    u_min = 0

    block_in = []
    for i in range(len(x_arr)):
        block_out = [(x_arr_avg[i],y_arr_avg[i]),(x_arr[i],y_arr[i]),
                     (x_arr_avg[i+1],y_arr_avg[i+1]),
                     (x_out_avg[i+1],y_out_avg[i+1]),(x_out[i],y_out[i]),
(x_out_avg[i],y_out_avg[i])]
        block_in.append((x_arr[i],y_arr[i]))

        img_out = Image.new('L', (n,m), 0)
```

```
        ImageDraw.Draw(img_out).polygon(block_out, outline=1, fill=1)
        mask_out = np.array(img_out)
        I_plus = np.ma.array(I, mask = mask_out == 1)
        u_plus = np.mean(I_plus)
        u_plus_arr[i] = u_plus

    img_in = Image.new('L', (n,m), 0)
    ImageDraw.Draw(img_in).polygon(block_in, outline=1, fill=1)
    mask_in = np.array(img_in)
    I_min = np.ma.array(I, mask = mask_in == 1)
    u_min = np.mean(I_min)

    return u_min, u_plus_arr

#Evaluates the mean values of the Mumford-Shah phase
def MeanMS(xy_arr,x_out_orig,y_out_orig,I,h,xmin,ymin):
    xy_arr_copy = np.array(xy_arr,copy=True)
    x_out = np.array(x_out_orig,copy=True)
    y_out = np.array(y_out_orig,copy=True)
    x_in,y_in = findAreaContour(xy_arr_copy,-h)
    x_arr = xy_arr[:len(xy_arr_copy)/2]-xmin
    y_arr = xy_arr[len(xy_arr_copy)/2:]-ymin
    x_in -= xmin
    y_in -= ymin
    x_out -= xmin
    y_out -= ymin

    x_arr_avg = findAverage(x_arr)
    y_arr_avg = findAverage(y_arr)
    x_out_avg = findAverage(x_out)
    y_out_avg = findAverage(y_out)
    x_in_avg = findAverage(x_in)
    y_in_avg = findAverage(y_in)

    m,n = I.shape

    u_plus_arr = np.zeros(len(x_arr))
    u_min_arr = np.zeros(len(x_arr))

    block_in = []
    for i in range(len(x_arr)):
        block_out = [(x_arr_avg[i],y_arr_avg[i]),(x_arr[i],y_arr[i]),\
        (x_arr_avg[i+1],y_arr_avg[i+1]),(x_out_avg[i+1],y_out_avg[i+1]),\
        (x_out[i],y_out[i]),(x_out_avg[i],y_out_avg[i])]
        block_in = [(x_arr_avg[i],y_arr_avg[i]),(x_arr[i],y_arr[i]),\
        (x_arr_avg[i+1],y_arr_avg[i+1]),(x_in_avg[i+1],y_in_avg[i+1]),\
        (x_in[i],y_in[i]),(x_in_avg[i],y_in_avg[i])]

        img_out = Image.new('L', (n,m), 0)
        ImageDraw.Draw(img_out).polygon(block_out, outline=1, fill=1)
        mask_out = np.array(img_out)
        if mask_out.size!=1:
            I_plus = np.ma.array(I, mask = mask_out == 1)
            u_plus = np.mean(I_plus)
            u_plus_arr[i] = u_plus
```

```
        else:
            u_plus_arr[i] = np.mean(I)

        img_in = Image.new('L', (n,m), 0)
        ImageDraw.Draw(img_in).polygon(block_in, outline=1, fill=1)
        mask_in = np.array(img_in)
        if mask_out.size!=1:
            I_min = np.ma.array(I, mask = mask_in == 1)
            u_min = np.mean(I_min)
            u_min_arr[i] = u_min
        else:
            u_min_arr[i] = np.mean(I)

    return u_min_arr, u_plus_arr

#Finds the contour normal to the current contour
def findAreaContour(xy_arr,h):
    x_arr = xy_arr[:len(xy_arr)/2]
    y_arr = xy_arr[len(xy_arr)/2:]
    B = matrixB(len(x_arr))
    C = matrixC(len(x_arr))
    x_dev = np.matmul(C,y_arr)
    y_dev = np.matmul(B,x_arr)
    a = abs(y_dev/x_dev)
    dx = h/np.sqrt(1+a**2)
    dy = a*dx
    return x_arr + dx*np.sign(x_dev), y_arr + dy*np.sign(y_dev)

#Doubles the points of an contour
def doubleContour(xy_arr):
    size = len(xy_arr)/2
    x_arr = xy_arr[:size]
    y_arr = xy_arr[size:]
    x_arr_per = np.insert(x_arr,size,x_arr[0])
    y_arr_per = np.insert(y_arr,size,y_arr[0])
    x_arr_new = np.zeros(2*size)
    y_arr_new = np.zeros(2*size)
    for i in range(size):
        x_arr_new[2*i] = x_arr[i]
        x_arr_new[2*i+1] = (x_arr_per[i]+x_arr_per[i+1])/2.0
        y_arr_new[2*i] = y_arr[i]
        y_arr_new[2*i+1] = (y_arr_per[i]+y_arr_per[i+1])/2.0
    xy_arr_new = np.concatenate((x_arr_new,y_arr_new))
    return xy_arr_new

#Finds boundaries of the cells using the improved Mumford—Shah model
def findBoundary(x_list,y_list,Radius,N,I,IGauss_jump,dt,nu_jump,kappa_jump,
                 nu_MS,kappa_MS,n_iter,h_jump,h_MS,d):
    #Generate matrices
    A = matrixA(N)
    B = matrixB(N)
    C = matrixC(N)
    D = matrixD(N)

    A2 = matrixA(2*N)
```

```
B2 = matrixB(2*N)
C2 = matrixC(2*N)
D2 = matrixD(2*N)

contour_list = []

#Initial contour
for l in range(len(x_list)):
    xy_arr = initContour(x_list[l],y_list[l],Radius,N)
    contour_list.append(xy_arr)

JUMP = np.array([True]*len(contour_list))
STOP = np.array([False]*len(contour_list))

for i in range(n_iter):
    for j in range(len(contour_list)):
        xy_arr = contour_list[j]

        if JUMP[j]:
            #Calculate mean values inside and outside the contour.
            x_out_orig, y_out_orig = findAreaContour(xy_arr,h_jump)
            xmin,xmax = int(np.min(x_out_orig)),int(np.max(x_out_orig))+1
            ymin,ymax = int(np.min(y_out_orig)),int(np.max(y_out_orig))+1
            IGauss_part = IGauss_jump[ymin:ymax,xmin:xmax]
            u_min, u_plus_arr = MeanJump(xy_arr,x_out_orig,y_out_orig,
                                         IGauss_part,h_jump,xmin,ymin)

            #Determine which points have to jump
            jump = (u_min-u_plus_arr<2*10**-2).astype(int)
            jump2 = np.concatenate((jump,jump))
            xy_arr[jump2==1] = np.concatenate((x_out_orig,y_out_orig))\
            [jump2==1]

            #If more than 75% of the point don't jump,
            #we stop the jump phase
            if (N-np.count_nonzero(jump))>0.75*N:
                JUMP[j] = False

            #Performs Mumford-Shah model
            E,Ediag = matrixE(xy_arr,u_min,u_plus_arr,IGauss_jump,d)
            MS1 = np.concatenate((nu_jump*A-kappa_jump*D,np.matmul(E,C)),
                                 axis=1)
            MS2 = np.concatenate((np.matmul(E,B),nu_jump*A-kappa_jump*D),
                                 axis=1)
            MS = np.concatenate((MS1,MS2),axis=0)
            BW = np.linalg.inv(np.identity(len(xy_arr))-dt*MS)
            xy_arr = np.matmul(BW,xy_arr)

            #Check if the jump phase stops
            if not JUMP[j]:
                xy_arr = doubleContour(xy_arr)

            contour_list[j] = xy_arr

        else:
```

```
                #If the contours stops movinfg, we continue with the others
                if STOP[j]:
                    continue
                else:
                    #Calculate mean values inside and outside the contour.
                    x_out_orig, y_out_orig = findAreaContour(xy_arr,h_MS)
                    xmin,xmax = int(np.min(x_out_orig)),\
                    int(np.max(x_out_orig))+1
                    ymin,ymax = int(np.min(y_out_orig)),\
                    int(np.max(y_out_orig))+1
                    I_part = I[ymin:ymax,xmin:xmax]
                    u_min_arr, u_plus_arr = MeanMS(xy_arr,x_out_orig,\
                    y_out_orig,I_part,h_MS,xmin,ymin)

                    #Performs Mumford—Shah model
                    E2,Ediag2 = matrixE2(xy_arr,u_min_arr,u_plus_arr,I,d)
                    MS1 = np.concatenate((nu_MS*A2—kappa_MS*D2,
                                          np.matmul(E2,C2)),axis=1)
                    MS2 = np.concatenate((np.matmul(E2,B2),
                                          nu_MS*A2—kappa_MS*D2),axis=1)
                    MS = np.concatenate((MS1,MS2),axis=0)
                    BW = np.linalg.inv(np.identity(len(xy_arr))—dt*MS)
                    xy_arr = np.matmul(BW,xy_arr)

                    f_tr = (abs(np.array(Ediag2))<10**—3).astype(int)
                    n_stop = np.count_nonzero(f_tr)
                    if n_stop>=N:
                        STOP[j]= True

                    contour_list[j] = xy_arr
    return contour_list

#Performs the inflation of the contour
def Inflate(curve_list,h):
    for i in range(len(curve_list)):
        xy_arr = curve_list[i]
        x,y = findAreaContour(xy_arr,h)
        xy_arr_step = np.concatenate((x,y))
        curve_list[i] = xy_arr_step
    return curve_list

#Plots the results of the contours
def plotResult(curve_list,I):
    m,n = I.shape
    plt.figure()
    plt.imshow(I, cmap = cm.Greys_r)
    plt.autoscale(False)
    for i in range(len(curve_list)):
        xy_arr = curve_list[i]
        x_arr = xy_arr[:len(xy_arr)/2]
        y_arr = xy_arr[len(xy_arr)/2:]
        x_arr_per = np.insert(x_arr,0,x_arr[len(x_arr)—1])
        y_arr_per = np.insert(y_arr,0,y_arr[len(x_arr)—1])
        plt.plot(x_arr_per,y_arr_per,'—or',markersize = 3, linewidth = 2)
    plt.axes().set_aspect('equal')
```

```
    m,n = I.shape
    plt.xticks([])
    plt.yticks([])
    plt.savefig("Result.png",bbox_inches='tight',dpi=300)
    plt.show()



# ======================== Main ==============================
if __name__ == "__main__":

    plt.close('all')
    STARTtime = time.time()

    #Beforehand determined parameters
    downsamplingfactor = 5
    N = 15
    dt = 10.0
    n_iter = 40
    #Consider only the part that is bdr removed from the edge of the image
    bdr = 100

    #Normalization and downsampling of image
    img = '../../../Images/img.jpg'
    Image_orig = Read(img)
    Image_normalized = Normalize(Image_orig)
    I = Normalize(downsamplingMean(Image_normalized, downsamplingfactor))
    m,n = I.shape

    #Finding the correct parameters and values
    #Sigma
    sigma = findSigma(I)
    print 'Sigma:', sigma

    #Ithreshold
    IGauss = Gaussianfilter(I,sigma/10.0)
    Ithreshold = findIthreshold(IGauss,0.01)
    print 'Threshold for binary image:', Ithreshold

    #Part white points
    Ibin = binary_image(I,Ithreshold)
    rho = np.mean(Ibin)
    print 'Part white points:', rho

    #Locating cell centers
    x_list, y_list = croppedMaxima(I,sigma,bdr)
    Ncell = len(x_list)
    print 'Number of cells:',Ncell

    #Diameter
    diameter = findDiameter(Ncell,m,n,bdr)
    print 'Diameter of cell:',diameter

    #Boundary width
    b_overestimation = findWidthBoundary(Ncell,m,n,rho,bdr)
```

```
b = 0.75*b_overestimation
print 'Width of boundary:', b

#Radius
Radius = 0.2*diameter
print 'Radius of initial contour:',Radius

#h_jump and h_MS
h_jump = b/2.0
h_MS = 2.0*b
print 'h_jump:', h_jump
print 'h_MS:', h_MS

#Weighting parameters
nu_jump1 = h_jump/(Radius*np.sin(2*np.pi/float(N)))
nu_jump2 = 2*np.pi*(1+(2*np.pi)**2)
nu_jump = nu_jump1/nu_jump2
kappa_jump = nu_jump

nu_MS = 1.0/(2*np.pi*(1+(2*np.pi)**2))
kappa_MS = nu_MS
print 'nu_jump, kappa_jump:',nu_jump
print 'nu_MS, kappa_MS:',nu_MS

#Blocksize d
d = int(b)
print 'Blocksize:',d

#Inflate size h_infl
h_infl = d-b/2.0
print 'h_infl:', h_infl

#Improved Mumford-Shah model
IGauss_jump = Gaussianfilter(I,0.8)
contour_list = findBoundary(x_list,y_list,Radius,N,I,IGauss_jump,dt,
                            nu_jump,kappa_jump,nu_MS,kappa_MS,n_iter,
                            h_jump,h_MS,d)
contour_list_inflated = Inflate(contour_list,h_infl)
plotResult(contour_list_inflated,I)

ENDtime = time.time()
print 'Execution time:', ENDtime-STARTtime
```

The list *contour_list_inflated* contains the solution of the contours at the last iteration $n_{iter} = 40$.