Technische Universiteit Delft
Faculteit Elektrotechniek, Wiskunde en Informatica
Delft Institute of Applied Mathematics

Analysis of Microscopic Images
A Gradient Vector Flow Based Approach

(Analyse van Microscoopbeelden
Een op Gradient Vector Flow Gebaseerde Aanpak)

Verslag ten behoeve van het
Delft Institute of Applied Mathematics
als onderdeel ter verkrijging

van de graad van

BACHELOR OF SCIENCE
in
TECHNISCHE WISKUNDE

door

DUNCAN DEN BAKKER

Delft, Nederland
Juni, 2018

BSc verslag TECHNISCHE WISKUNDE

"Analysis of Microscopic Images"
("Analyse van Microscoopbeelden")

DUNCAN DEN BAKKER

Technische Universiteit Delft

Begeleider

Dr. N.V. Budko

Overige commissieleden

Drs. E.M. van Elderen

Dr. J.F.B.M. Kraaijevanger

Juni, 2018

Delft

Abstract

The theory of active contours is applied to microscopic images of cells. A model is developed that approximates cell borders by dynamic curves. This model is based on gradient vector flow (GVF), an external force that acts on the contours. Both active contours and the GVF force field are defined as functions that minimize certain functionals. The corresponding Euler-Lagrange equations are derived and analyzed theoretically. A number of auxiliary algorithms are designed to aid the performance of the main snake algorithm, including a preprocessing algorithm, a method for detecting cell centers and an algorithm that detects areas devoid of cells. Results of the snake algorithm are presented, along with practical considerations regarding parameter choice. Finally, statistical methods are applied to the results to demonstrate their usefulness.

Contents

Machine Specifications

Throughout this thesis, real world computation times will be mentioned. As these times depend on the machine on which the program is performed, it is important to mention the specifications of the machine used. These can be used to predict how fast certain programs run on other machines. The machine in question is a 64-bit machine running Ubuntu 16.04. Its CPU is a quad-core Intel Core i5-4670k clocked at 3.40 GHz. Futhermore, the machine contains 16GB of DDR4 RAM.

1 Introduction

In this thesis the theory of active contours, or snakes, is applied to microscopic images of potatoes. These images come from the company HZPC[1] based in Joure, the Netherlands. HZPC develops breeds of potatoes for many purposes. For example, the potatoes used to make french fries should have different properties from those used to make mashed potato. Properties of potatoes are closely linked to its microscopic structure. For this reason, HZPC is working with the TU Delft in order to develop a tool that analyzes microscopic images of potatoes.

The images in question contain a large number of cells. The aim is to approximate the cell boundaries using curves, or in practice a set of points. When this is achieved, a number of geometric parameters can be extracted from every curve. These parameters can tell us something about the properties of the potato. In particular the isoperimetric quotient, which will be defined in a later chapter, and the diameter of a cell are valuable parameters. Since it is invariant under scaling, images made with different magnification factors can still be compared to each other. The ultimate goal is to use statistical analysis to determine to which class or type a given potato belongs, based on its microscopic structure. We will focus more on building the 'tool' used to convert an image to a collection of curves. However the last chapter contains some basic statistical analysis, to demonstrate that there are in fact differences between images.

Previous attempts at approximating the cell boundaries had mixed succes. Methods that have been tried are the watershed method, curve fitting and a more physical method involving point charges. All methods had their own downsides, the most frequent problem was neglecting vague cell boundaries. This resulted in the method regarding two (or more) cells as being one. Active contours [1] are used to minimize the number of occurences of this phenomenon. They are also quite flexible in use; parameters can be tuned to make curves less or more smooth, larger or smaller, etcetera. The succes of active contours depends on the initial curves and the force field used. Placing initial curves is a problem itself, as cells should be detected beforehand by some other algorithm.

The force field we use is the gradient vector flow field, or GVF field [2]. It is constructed in such a way that it remains smooth in regions that contain little data. Typically, these regions are the cell interiors. The desired consequence of this behaviour is that curves that lie in these low-data regions are nonetheless subject to a force. This means that initial curves can be placed further away from the cell boundary, and still converge. The theory behind active contours and the GVF field is presented in chapter 2.

Active contours and the GVF field are both formulated in a continuous framework. In particular, a snake satisfies an ordinary differential equation, whereas the GVF field satisfies a partial differential equation. The relevant equations should be discretized in order for them to be applicable to (discrete) images. The dimensions of the microscopic images can be quite large, so the computer model should be optimized extensively, or else computation time will be unreasonable. This is done by using sparse matrices and specialized solvers in the programming language Python.

Whenever an image provided by HZPC is used in a figure, its name is mentioned in the figure caption. These names all share the same structure, for example "312;US;2017;FIELD32;T12;CORTEX.REP2" is one possible image title. We differentiate different datasets by the first number, in this case 312. We have access to about 50 datasets.

[1] https://www.hzpc.com/

2 | Theory of Active Contours

A grayscale image $I : \Omega \to [0, 1]$ is a function on a (usually rectangular) region $\Omega \in \mathbb{R}^2$ into the interval $[0, 1]$. Its values represent the gray value in a point; 0 represents black and 1 represents white. In practice an image is identified with a matrix. Each entry of this matrix represents the value of a *pixel*. The dimensions of this matrix determines the size of the image; a matrix of size $n \times m$ represents an image consisting of as many pixels.

$$A = \begin{pmatrix} 0.55 & 0.49 & 0.86 & 0.61 & 0.08 & 0.07 \\ 0.62 & 0.22 & 0.07 & 0.93 & 0.79 & 0.31 \\ 0.90 & 0.95 & 0.07 & 0.00 & 0.43 & 0.57 \\ 0.61 & 0.86 & 0.77 & 0.86 & 0.54 & 0.69 \\ 0.84 & 0.68 & 0.25 & 0.77 & 0.41 & 0.97 \\ 0.46 & 0.61 & 0.07 & 0.05 & 0.96 & 0.54 \end{pmatrix}$$

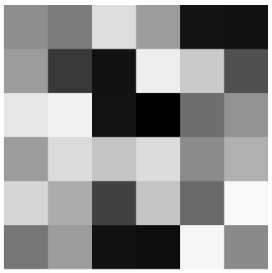

Figure 2.1: Image corresponding to the matrix A. We usually say that A 'is' the image.

An active contour [1], also called snake, is a curve in Ω that gets pulled to features of interest by image forces. These features are usually edges or lines. A snake is subject to two forces, an internal and external force. The external force is derived from the image data, resulting in a force field pointing towards features of interest. The external force is usually defined as the gradient of an external energy function E_{ext}. Common choices of E_{ext} are $-\|\nabla I\|^2$, $-\|G_\sigma * I\|^2$, or simply I or $G_\sigma * I$. Here G_σ represents a Gaussian kernel with standard deviation σ. We will focus on a different external force altogether, called gradient vector flow. It is introduced in section 2.2. The internal forces of a curve are added to hold the curve together. It contains a first and second order term, both controlled by different parameters. The first order term makes the snake act like an elastic membrane, whereas the second order term makes it act like a thin plate. The second order term keeps the snake from bending too much. In the case of biological cells, we expect the second order term to be small compared to the first order term, because cell borders are more elastic than rigid.

The aim of this project is to apply the theory of active contours to analyse cells in microscopic images. Hence an extra property imposed on the snakes is that they must be closed. A consequence is that the contour encloses a region. The area of this region is a statistical parameter examined in section 5.3.

2.1 Snakes

A snake is a curve parameterized by a function $\boldsymbol{x} : [0, 1] \to \Omega$ that minimizes the energy functional

$$E[\boldsymbol{x}] = \int_0^1 \frac{1}{2}\Big(\alpha\|\boldsymbol{x}'(s)\|^2 + \beta\|\boldsymbol{x}''(s)\|^2\Big) + E_{\text{ext}}(\boldsymbol{x}(s))\,\mathrm{d}s \tag{2.1}$$

Note that one curve can be parameterized by infinitely many functions. Nevertheless, we will identify $\boldsymbol{x}$ with the curve it describes. We restrict $\boldsymbol{x}$ to be a four times continuously differentiable closed curve. The parameters α and β control the first and second order terms, or the snake's tension and rigidity, respectively. The external energy E_{ext} is derived from the image. Generally, it takes smaller values at regions of interest, such as boundaries. That way, the gradient ∇E_{ext} points towards these regions.

The functional in (2.1) can be minimized by a greedy algorithm, which moves one point at a time. This is however a very inefficient method. We derive the Euler-Lagrange equation corresponding to E. This turns out to be a fourth order nonlinear ordinary differential equation, which can be solved by making it dynamic.

Assume that E has a local minimum in $\boldsymbol{x}$. Let $\boldsymbol{v} : [0,1] \to \Omega$ be some twice continuously differentiable fixed curve. Write $\boldsymbol{x}(s) = (x(s), y(s))$ and $\boldsymbol{v}(s) = (v(s), w(s))$. Then for all small values of $\varepsilon > 0$ we have $E[\boldsymbol{x}] \leq E[\boldsymbol{x} + \varepsilon \boldsymbol{v}]$. Hence the function $\varphi(\varepsilon) = E[\boldsymbol{x} + \varepsilon \boldsymbol{v}]$ assumes its minimum at $\varepsilon = 0$. From calculus we know $\varphi'(0) = 0$. First calculate its derivative:

$$\varphi'(\varepsilon) = \int_0^1 \alpha(\boldsymbol{x}' + \varepsilon \boldsymbol{v}') \cdot \boldsymbol{v}' + \beta(\boldsymbol{x}'' + \varepsilon \boldsymbol{v}'') \cdot \boldsymbol{v}'' + \nabla E_{\text{ext}}(\boldsymbol{x}) \cdot \boldsymbol{v} \, \mathrm{d}s. \tag{2.2}$$

Substituting $\varepsilon = 0$ yields

$$\varphi'(0) = \int_0^1 \alpha \left(\boldsymbol{x}' \cdot \boldsymbol{v}'\right) + \beta \left(\boldsymbol{x}'' \cdot \boldsymbol{v}''\right) + \nabla E_{\text{ext}}(\boldsymbol{x}) \cdot \boldsymbol{v} \, \mathrm{d}s = 0. \tag{2.3}$$

By linearity of the integral, we can calculate each term individually. Using partial integration[2] we obtain:

$$\int_0^1 \boldsymbol{x}' \cdot \boldsymbol{v}' \, \mathrm{d}s = \Big[\boldsymbol{x}' \cdot \boldsymbol{v}\Big]_{s=0}^1 - \int_0^1 \boldsymbol{x}'' \cdot \boldsymbol{v} \, \mathrm{d}s \overset{(*_1)}{=} - \int_0^1 \boldsymbol{x}'' \cdot \boldsymbol{v} \, \mathrm{d}s, \tag{2.4}$$

and

$$\begin{aligned} \int_0^1 \boldsymbol{x}'' \cdot \boldsymbol{v}'' \, \mathrm{d}s &= \Big[\boldsymbol{x}'' \cdot \boldsymbol{v}'\Big]_{s=0}^1 - \int_0^1 \boldsymbol{x}''' \cdot \boldsymbol{v}' \, \mathrm{d}s \overset{(*_2)}{=} - \int_0^1 \boldsymbol{x}''' \cdot \boldsymbol{v}' \, \mathrm{d}s \\ &= -\Big[\boldsymbol{x}''' \cdot \boldsymbol{v}\Big]_{s=0}^1 + \int_0^1 \boldsymbol{x}'''' \cdot \boldsymbol{v} \, \mathrm{d}s \overset{(*_3)}{=} \int_0^1 \boldsymbol{x}'''' \cdot \boldsymbol{v} \, \mathrm{d}s. \end{aligned} \tag{2.5}$$

For steps $(*_1)$, $(*_2)$ and $(*_3)$ to be valid, we need to impose additional conditions on $\boldsymbol{x}$ and $\boldsymbol{v}$. These conditions are:

$$\boldsymbol{x}'(0) = \boldsymbol{x}'(1), \quad \boldsymbol{x}''(0) = \boldsymbol{x}''(1), \quad \boldsymbol{x}'''(0) = \boldsymbol{x}'''(1), \quad \boldsymbol{v}(0) = \boldsymbol{v}(1), \quad \boldsymbol{v}'(0) = \boldsymbol{v}'(1). \tag{2.6}$$

It makes sense to let $\boldsymbol{v}$ belong to the same class of functions as $\boldsymbol{x}$, by imposing periodic boundary conditions on $\boldsymbol{v}''$ and $\boldsymbol{v}'''$. However, this is not a necessary step to obtain the following equation. Substituting (2.4) and (2.5) into equation (2.3) yields

$$\varphi'(0) = \int_0^1 \big[-\alpha \boldsymbol{x}'' + \beta \boldsymbol{x}'''' + \nabla E_{\text{ext}}(\boldsymbol{x})\big] \cdot \boldsymbol{v} \, \mathrm{d}s = 0. \tag{2.7}$$

Note that equation (2.7) must hold for any $\boldsymbol{v}$, from which it can be concluded that $-\alpha \boldsymbol{x}'' + \beta \boldsymbol{x}'''' + \nabla E_{\text{ext}}(\boldsymbol{x}) = 0$. To summarize: in order for a curve $\boldsymbol{x}$ to minimize the energy functional found in (2.1), it must satisfy the following ordinary differential equation:

$$\alpha \boldsymbol{x}'' - \beta \boldsymbol{x}'''' - \nabla E_{\text{ext}}(\boldsymbol{x}) = 0. \tag{2.8}$$

This is called the Euler-Lagrange equation corresponding to the functional E. It should be noted that a curve $\boldsymbol{x}$ satisfying (2.8) does not necessarily minimize E. The other way around is true.

[2] The reader can verify him- or herself that partial integration is in fact a valid technique when dealing with the dot product.

2.2 Gradient Vector Flow

Let f be a map derived from I such that it takes on larger values near features of interest. In our cases we let f be large near cell boundaries. The exact definition is given in section 4.2. The *gradient vector flow* [2] field is defined as the vector field $\boldsymbol{u}(x,y) = (u(x,y), v(x,y))$ that minimizes the functional

$$\mathcal{E}[\boldsymbol{u}] = \iint_\Omega \mu\left(u_x^2 + u_y^2 + v_x^2 + v_y^2\right) + \|\nabla f\|^2 \|\boldsymbol{u} - \nabla f\|^2 \,\mathrm{d}\Omega. \tag{2.9}$$

Where u_x denotes the partial derivative of u with respect to x. Furthermore, ∇f and $\mu > 0$ are given. This formulation comes from the principle of making the result smooth when there is no data. In other words, if there is no image data in a particular region (constant value), then data at the boundary of this region is used to make the resulting force field smooth. In particular, if $\|\nabla f\|$ is small, the functional $\mathcal{E}$ is dominated by partial derivatives of $\boldsymbol{u}$. This yields a smooth vector field, as these partial derivatives should be small in order to minimize $\mathcal{E}$. Moreover, if $\|\nabla f\|$ is large, the second term dominates $\mathcal{E}$. In this case $\mathcal{E}$ is minimized by setting $\boldsymbol{u} = \nabla f$. The parameter μ determines how strong the smoothing property of the GVF field is.

Finding the corresponding Euler-Lagrange equations is done in a similar manner as in section 2.1. Assume $\mathcal{E}$ reaches its minimum at $\boldsymbol{u}$. Let $\boldsymbol{\nu} = (\xi, \eta)$ be a vector field on Ω. Define $\varphi(\varepsilon) = \mathcal{E}[\boldsymbol{u} + \varepsilon\boldsymbol{\nu}]$ for small values of $\varepsilon > 0$, then $\varphi'(0) = 0$. The derivative of φ is given by

$$\begin{aligned}\varphi'(\varepsilon) = 2\iint_\Omega \mu\big((u_x + \varepsilon\xi_x)\xi_x + (u_y + \varepsilon\xi_y)\xi_y + (v_x + \varepsilon\eta_x)\eta_x + (v_y + \varepsilon\eta_y)\eta_y\big) \\ + \|\nabla f\|^2\big((u - f_x + \varepsilon\xi)\xi + (v - f_y + \varepsilon\eta)\eta\big)\,\mathrm{d}\Omega.\end{aligned} \tag{2.10}$$

Substituting $\varepsilon = 0$ yields

$$\begin{aligned}\varphi'(0) &= 2\iint_\Omega \mu\big(u_x\xi_x + u_y\xi_y + v_x\eta_x + v_y\eta_y\big) + \|\nabla f\|^2\big((u - f_x)\xi + (v - f_y)\eta\big)\,\mathrm{d}\Omega \\ &= 2\iint_\Omega \mu\big(\nabla u \cdot \nabla\xi + \nabla v \cdot \nabla\eta\big) + \|\nabla f\|^2(\boldsymbol{u} - \nabla f)\cdot\boldsymbol{\nu}\,\mathrm{d}\Omega.\end{aligned} \tag{2.11}$$

This integral can be simplified using Green's first identity, which states:

$$\iint_\Omega \nabla f \cdot \nabla g \,\mathrm{d}\Omega = \int_{\partial\Omega} f(\nabla g \cdot \boldsymbol{n})\,\mathrm{d}\Gamma - \iint_\Omega f\nabla^2 g \,\mathrm{d}\Omega.$$

Where $\boldsymbol{n}$ is the outward pointing normal to the boundary $\partial\Omega$. Applying this to the first part of the integral found in (2.11), we obtain:

$$\begin{aligned}\iint_\Omega \nabla u \cdot \nabla\xi + \nabla v \cdot \nabla\eta \,\mathrm{d}\Omega &= \int_{\partial\Omega} \xi(\nabla u \cdot \boldsymbol{n}) + \eta(\nabla v \cdot \boldsymbol{n})\,\mathrm{d}\Gamma - \iint_\Omega \xi\nabla^2 u + \eta\nabla^2 v \,\mathrm{d}\Omega. \\ &\overset{(*)}{=} -\iint_\Omega \begin{pmatrix}\nabla^2 u \\ \nabla^2 v\end{pmatrix} \cdot \boldsymbol{\nu}\,\mathrm{d}\Omega.\end{aligned} \tag{2.12}$$

In order for step $(*)$ to hold, we need either $\boldsymbol{\nu}$, or $\nabla u \cdot \boldsymbol{n}$ and $\nabla v \cdot \boldsymbol{n}$ to vanish on $\partial\Omega$. We will assume the latter, which will be used in section 3. Altogether, the integral in (2.11) becomes

$$\varphi'(0) = 2\iint_\Omega \left[-\mu\begin{pmatrix}\nabla^2 u \\ \nabla^2 v\end{pmatrix} + \|\nabla f\|^2(\boldsymbol{u} - \nabla f)\right] \cdot \boldsymbol{\nu}\,\mathrm{d}\Omega = 0. \tag{2.13}$$

This equality must hold for every $\boldsymbol{\nu}$. Thus we conclude that $\boldsymbol{u}$ must satisfy the following equations in order to minimize $\mathcal{E}$:

$$\mu\nabla^2 u - (f_x^2 + f_y^2)(u - f_x) = 0 \tag{2.14a}$$

$$\mu\nabla^2 v - (f_x^2 + f_y^2)(v - f_y) = 0 \tag{2.14b}$$

We say that a curve $\boldsymbol{x}$ is a *GVF snake* corresponding to the *GVF field* $\boldsymbol{u}$ if it satisfies the equation

$$\alpha\boldsymbol{x}'' - \beta\boldsymbol{x}'''' + \boldsymbol{u}(\boldsymbol{x}) = 0. \tag{2.15}$$

Note that this is similar to equation (2.8); the potential force $-\nabla E_{\text{ext}}$ is replaced by $\boldsymbol{u}$.

2.3 Analysis of Snake Equation

In order to solve equation (2.15), it is made *dynamic*, or time-dependent. That is, we treat $\boldsymbol{x}$ as a function of s as well as t, i.e. $\boldsymbol{x} = \boldsymbol{x}(s,t)$. The partial derivative of $\boldsymbol{x}$ with respect to t is set to equal the left hand side of equation (2.15). Furthermore, a parameter γ is introduced to control the effect of the GVF field compared to the internal forces:

$$\frac{\partial \boldsymbol{x}}{\partial t} = \alpha \frac{\partial^2 \boldsymbol{x}}{\partial s^2} - \beta \frac{\partial^4 \boldsymbol{x}}{\partial s^4} + \gamma \boldsymbol{u}(\boldsymbol{x}). \tag{2.16}$$

Note that the ratio between γ and α and β determines the effect of the external force in comparison to the internal forces. However, the parameters α, β and γ all have an effect on the speed at which the snake is transformed. This equation is called the *dynamic snake equation*. Whenever the solution $\boldsymbol{x}$ stabilizes, its time derivative vanishes, resulting in a solution to (2.15). To formulate a well-posed problem, boundary conditions and an initial condition are required. As mentioned before, periodic boundary conditions are used. The initial curve is called $\boldsymbol{x}_0$, resulting in the completely formulated problem

$$\begin{cases} \dfrac{\partial \boldsymbol{x}}{\partial t} &= \alpha \dfrac{\partial^2 \boldsymbol{x}}{\partial s^2} - \beta \dfrac{\partial^4 \boldsymbol{x}}{\partial s^4} + \gamma \boldsymbol{u}(\boldsymbol{x}), & s \in [0,1],\ t > 0 \\ \dfrac{\partial^k \boldsymbol{x}}{\partial s^k}(0,t) &= \dfrac{\partial^k \boldsymbol{x}}{\partial s^k}(1,t), & t > 0, \quad \text{for } k = 0,1,2,3 \\ \boldsymbol{x}(s,0) &= \boldsymbol{x}_0(s), & s \in [0,1] \end{cases} \cdot \tag{2.17}$$

2.3.1 No Force Field

First set $\gamma = 0$ in (2.17), the snake equation is then given by

$$\frac{\partial \boldsymbol{x}}{\partial t} = \alpha \frac{\partial^2 \boldsymbol{x}}{\partial s^2} - \beta \frac{\partial^4 \boldsymbol{x}}{\partial s^4}, \tag{2.18}$$

which is a linear parabolic partial differential equation with constant coefficients, and can be solved by means of a series expansion. By using a Fourier series, the periodic boundary conditions are automatically satisfied. Note that both components (x and y) of $\boldsymbol{x}$ satisfy the same equation, so it suffices to solve the problem for one of the components, say x. We write $\boldsymbol{x}(s,t) = \boldsymbol{x}_t(s)$ and $\boldsymbol{x}_0(s) = (x_0(s), y_0(s))$, for $t > 0$ let

$$x(s,t) = \sum_{n \in \mathbb{Z}} \widehat{x_t}(n) e^{2\pi i n s}, \quad \text{where } \widehat{x_t}(n) = \int_0^1 x_t(s) e^{-2\pi i n s} \, \mathrm{d}s. \tag{2.19}$$

Then indeed by the 1-periodicity of the functions $s \mapsto e^{2\pi i n s}$, all boundary conditions are satisfied. The initial condition is also written as a Fourier series:

$$x_0(s) = \sum_{n \in \mathbb{Z}} \widehat{x_0}(n) e^{2\pi i n s}, \quad \text{where } \widehat{x_0}(n) = \int_0^1 x_0(s) e^{-2\pi i n s} \, \mathrm{d}s. \tag{2.20}$$

If x_0 and x_t are assumed to be differentiable, then the series expansions above converge pointwise. Note that x_t should be a $C^4([0,1])$ function in order to satisfy (2.18), so it is differentiable. The initial curve x_0 can be chosen in such a way that it is differentiable. Substituting the series expansion for $x(s,t)$ into equation (2.18) yields

$$\begin{aligned} \sum_{n \in \mathbb{Z}} \frac{\partial \widehat{x_t}}{\partial t}(n) e^{2\pi i n s} &= \alpha \sum_{n \in \mathbb{Z}} (2\pi i n)^2 \widehat{x_t}(n) e^{2\pi i n s} - \beta \sum_{n \in \mathbb{Z}} (2\pi i n)^4 \widehat{x_t}(n) e^{2\pi i n s} \\ &= -\sum_{n \in \mathbb{Z}} \left(4\pi^2 n^2 \alpha + 16\pi^4 n^4 \beta\right) \widehat{x_t}(n) e^{2\pi i n s}. \end{aligned}$$

From which it follows that $\widehat{x_t}(n)$ should satisfy

$$\frac{\partial \widehat{x_t}}{\partial t}(n) = -\left(4\pi^2 n^2 \alpha + 16\pi^4 n^4 \beta\right) \widehat{x_t}(n), \quad \text{for all } n \in \mathbb{Z}. \tag{2.21}$$

The solution is given by

$$\widehat{x_t}(n) = c(n) \exp\left[-\left(4\pi^2 n^2 \alpha + 16\pi^4 n^4 \beta\right) t\right]. \tag{2.22}$$

By substituting $t = 0$ we find that $c(n) = \widehat{x_0}(n)$, and thus

$$\widehat{x_t}(n) = \widehat{x_0}(n) \exp\left[-\left(4\pi^2 n^2 \alpha + 16\pi^4 n^4 \beta\right) t\right]. \tag{2.23}$$

Note that for $n = 0$, the Fourier coefficient $\widehat{x_t}(0)$ is constant in time:

$$\widehat{x_t}(0) = \widehat{x_0}(0) = \int_0^1 x_0(s)\, \mathrm{d}s$$

The first component of the solution to (2.18) is then given by

$$x(s,t) = \sum_{n\in\mathbb{Z}} \widehat{x_0}(n) \exp\left[-\left(4\pi^2 n^2 \alpha + 16\pi^4 n^4 \beta\right) t\right] \exp\left[2\pi i n s\right]. \tag{2.24}$$

This series can also be written in terms of sines and cosines, yielding

$$x(s,t) = \widehat{x_0}(0) + \sum_{n=1}^{\infty} \left(a_n \cos(2\pi n s) + b_n \sin(2\pi n s)\right) \exp\left[-\left(4\pi^2 n^2 \alpha + 16\pi^4 n^4 \beta\right) t\right], \tag{2.25}$$

where

$$a_n = 2\int_0^1 x_0(s) \cos(2\pi n s)\, \mathrm{d}s, \quad b_n = 2\int_0^1 x_0(s) \sin(2\pi n s)\, \mathrm{d}s.$$

Given that y satisfies the same equation, the solution to (2.18) can be written as

$$\boldsymbol{x}(s,t) = \int_0^1 \boldsymbol{x}_0(s)\, \mathrm{d}s + \sum_{n=1}^{\infty} \begin{pmatrix} a_n & b_n \\ c_n & d_n \end{pmatrix} \begin{pmatrix} \cos(2\pi n s) \\ \sin(2\pi n s) \end{pmatrix} \exp\left[-\left(4\pi^2 n^2 \alpha + 16\pi^4 n^4 \beta\right) t\right], \tag{2.26}$$

where

$$a_n = 2\int_0^1 x_0(s) \cos(2\pi n s)\, \mathrm{d}s, \quad b_n = 2\int_0^1 x_0(s) \sin(2\pi n s)\, \mathrm{d}s,$$
$$c_n = 2\int_0^1 y_0(s) \cos(2\pi n s)\, \mathrm{d}s, \quad d_n = 2\int_0^1 y_0(s) \sin(2\pi n s)\, \mathrm{d}s.$$

While (2.26) seems very similar to (2.25), the solution is now vector-valued. The obtained representation of the solution given us some insight into its behaviour.

Example 2.1. Let $\boldsymbol{x}_0$ be a circle around a point $\boldsymbol{p} \in \mathbb{R}^2$:

$$\boldsymbol{x}_0(s) = \boldsymbol{p} + \begin{pmatrix} \cos(2\pi s) \\ \sin(2\pi s) \end{pmatrix}, \quad s \in [0,1].$$

Then by the orthogonality of sines and cosines, the solution simply becomes

$$\boldsymbol{x}(s,t) = \boldsymbol{p} + \begin{pmatrix} \cos(2\pi s) \\ \sin(2\pi s) \end{pmatrix} \exp\left[-\left(4\pi^2 \alpha + 16\pi^4 \beta\right) t\right].$$

For $t \to \infty$, the curve converges to the point $\boldsymbol{p}$. This is a singularity, as a point is no longer a curve. The rate of convergence depends on the size of α and β. The larger they are, the faster $\boldsymbol{x}$ converges to $\boldsymbol{p}$.

2.3.2 Simple Force Field

Let $\gamma \neq 0$ in (2.17), and let $\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{x}$. We then get the following snake equation

$$\frac{\partial \boldsymbol{x}}{\partial t} = \alpha \frac{\partial^2 \boldsymbol{x}}{\partial s^2} - \beta \frac{\partial^4 \boldsymbol{x}}{\partial s^4} + \gamma \boldsymbol{x}. \tag{2.27}$$

This field points outward from the origin, its magnitude is proportional to the distance to the origin. The GVF field inside a perfectly round cell around the origin will be somewhat similar to this field, as it also has a greater magnitude near the cell border. While there may not be a one-to-one correspondence between the field $\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{x}$ and the GVF field, examing the solution (2.27) allows us make qualitative predictions on the behaviour of a curve subject to the GVF field. The process of finding the solution is very similar to the one described in section 2.3.1. To avoid repetition, these steps are skipped. The solution is given by

$$\boldsymbol{x}(s,t) = \int_0^1 \boldsymbol{x}_0(s)\, \mathrm{d}s + \sum_{n=1}^{\infty} \begin{pmatrix} a_n & b_n \\ c_n & d_n \end{pmatrix} \begin{pmatrix} \cos(2\pi n s) \\ \sin(2\pi n s) \end{pmatrix} \exp\left[\left(\gamma - 4\pi^2 n^2 \alpha - 16\pi^4 n^4 \beta\right) t\right], \tag{2.28}$$

where again

$$a_n = 2\int_0^1 x_0(s)\cos(2\pi n s)\, \mathrm{d}s, \quad b_n = 2\int_0^1 x_0(s)\sin(2\pi n s)\, \mathrm{d}s,$$
$$c_n = 2\int_0^1 y_0(s)\cos(2\pi n s)\, \mathrm{d}s, \quad d_n = 2\int_0^1 y_0(s)\sin(2\pi n s)\, \mathrm{d}s.$$

Example 2.2. Let $\boldsymbol{x}_0$ be a circle of radius 1 around the origin, then the solution to (2.27) is given by

$$\boldsymbol{x}(s,t) = \begin{pmatrix} \cos(2\pi s) \\ \sin(2\pi s) \end{pmatrix} \exp\left[\left(\gamma - 4\pi^2 \alpha - 16\pi^4 \beta\right) t\right].$$

By the orthogonality of $\sin(2\pi n s)$ and $\cos(2\pi m s)$, all coefficients b_n and c_n are indeed equal to zero. The coefficients a_n and d_n are equal to 1 whenever $n = 1$, and zero otherwise. The curve can behave in three different ways. It can either shrink and converge to the origin, it can grow exponentially, or it can remain stationary. The type of behaviour depends on the choice of α, β and γ, see table 2.1.

Type of behaviour	Condition
decay	$\gamma < 4\pi^2\alpha + 16\pi^4\beta$
growth	$\gamma > 4\pi^2\alpha + 16\pi^4\beta$
stationary	$\gamma = 4\pi^2\alpha + 16\pi^4\beta$

Table 2.1: Different behaviours of solution.

In Figure 2.2 the effect of γ on the convergence speed is visualized. The parameters used are $\alpha = 1$ and $\beta = 0$. The curve is plotted for $t = 0, 0.05, 0.1$, such that the deviation from the initial curve can be seen. The vector field $\boldsymbol{u}(\boldsymbol{x}) = \gamma \boldsymbol{x}$ is also plotted.

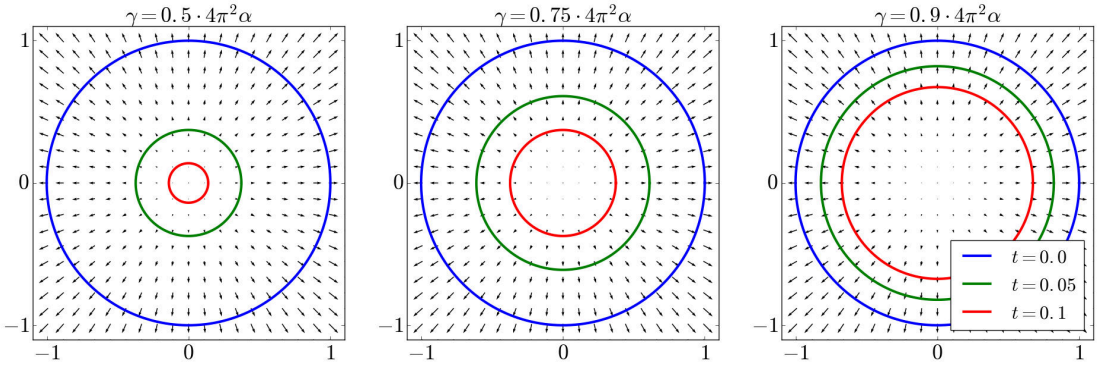

Figure 2.2: Convergence speed for different values of γ.

More general, if $\boldsymbol{x}_0$ consists of a finite amount of modes around the origin, say

$$\boldsymbol{x}_0(s) = \sum_{n=1}^{k} \begin{pmatrix} a_n & b_n \\ c_n & d_n \end{pmatrix} \begin{pmatrix} \cos(2\pi n s) \\ \sin(2\pi n s) \end{pmatrix},$$

then the solution is given by

$$\boldsymbol{x}(s,t) = \sum_{n=1}^{k} \begin{pmatrix} a_n & b_n \\ c_n & d_n \end{pmatrix} \begin{pmatrix} \cos(2\pi n s) \\ \sin(2\pi n s) \end{pmatrix} \exp\left[\left(\gamma - 4\pi^2 n^2 \alpha - 16\pi^4 n^4 \beta\right) t\right].$$

Which follows from the general solution presented in (2.28). Assume that α and β are fixed. If we choose γ such that the first mode is stationary, i.e. $\gamma = 4\pi^2\alpha + 16\pi^4\beta$, then all higher modes will decay. Indeed, for all $n > 1$ we have

$$\gamma = 4\pi^2\alpha + 16\pi^4\beta < 4\pi^2 n^2 \alpha + 16\pi^4 n^4 \beta.$$

This means that the curve converges to the first mode:

$$\boldsymbol{x}(s,t) \to \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \begin{pmatrix} \cos(2\pi s) \\ \cos(2\pi s) \end{pmatrix} \quad \text{as } t \to \infty, \quad \text{for all } s \in [0,1].$$

Moreover, if we choose γ such that the highest mode is stationary, i.e. $\gamma = 4\pi^2 k^2 \alpha + 16\pi^2 k^4 \beta$, then all lower modes will grow exponentially. In this case the curve will diverge.

Example 2.3. Since the initial curves that are used in the model are squares, it makes sense to examine the solution whenever the initial curve is a square. Such a square can be written as a Fourier series, for example, a square around the origin with sides of length 2 is given by

$$\mathrm{SQ}(s) = \frac{8}{\pi^2} \sum_{n=0}^{\infty} \begin{pmatrix} 1 & (-1)^{n+1} \\ 1 & (-1)^n \end{pmatrix} \begin{pmatrix} \cos(2\pi(2n+1)s) \\ \sin(2\pi(2n+1)s) \end{pmatrix}.$$

This can be verified by looking at both coordinate functions; they both represent the Fourier series of a (shifted) traingle wave on $[0,1]$. For practical purposes, we consider only a finite amount of modes of the square:

$$\mathrm{SQ}_k(s) = \frac{8}{\pi^2} \sum_{n=0}^{k} \begin{pmatrix} 1 & (-1)^{n+1} \\ 1 & (-1)^n \end{pmatrix} \begin{pmatrix} \cos(2\pi(2n+1)s) \\ \sin(2\pi(2n+1)s) \end{pmatrix}.$$

We let $\alpha = 0.01, \beta = 0$ and set $\gamma = 4\pi^2\alpha$. We expect the square to converge to a circle, because all modes higher than 1 will decay. This is indeed the case, as one can see in Figure 2.3.

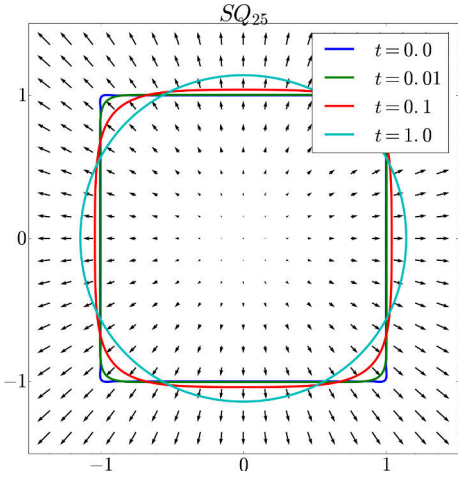

Figure 2.3: Solution to (2.27) using $\boldsymbol{x}_0 = \mathrm{SQ}_{25}$.

Setting γ equal to values higher than $4\pi^2\alpha$ but lower than $16\pi^2\alpha$ results in all modes vanishing except for the first mode, which instead grows. In Figure 2.4 one can see that the higher modes vanish rapidly, leaving a fast growing circle.

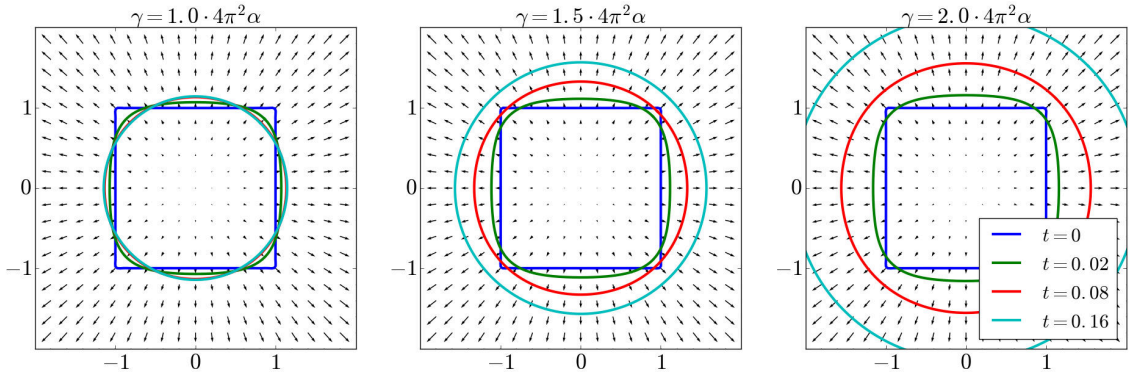

Figure 2.4: Divergence speed for different values of γ, using $\alpha = 0.1$ and $\boldsymbol{x}_0 = \mathrm{SQ}_{25}$.

We conclude that the choice of γ is important in controlling the behaviour of a curve. While the GVF is not such a simple force field as the one used in this section, we expect it to be similar to the simple force field locally. In this case, the obtained criteria for γ can be taken into account when tuning the parameters. While it can not be applied directly, we expect the criteria to give a good estimate of the effective range of γ.

3 Discretization

In order to solve the dynamic snake equation (2.16), it is first discretized in space, and subsequently discretized in time. Both discretizations are done in section 3.2. It is also necessary to compute the GVF field numerically. This is done by discretizing equations (2.14a) and (2.14b), which is described in section 3.3. It turns out that first defining a coordinate transformation yields insight into the effect of the number of discretization points on the parameters α and β, as seen in section 3.1.

3.1 Coordinate Transformation

A curve is discretized by N points, such that each point $\boldsymbol{x}_i(t)$ is an approximation of $\boldsymbol{x}(i/N, t)$ for $i = 0, \ldots, N-1$. This curve can also be parameterized by a function $\widetilde{\boldsymbol{x}} : [0, N] \times \mathbb{R}^+ \to \Omega$ defined by $\widetilde{\boldsymbol{x}}(\widetilde{s}, t) = \widetilde{\boldsymbol{x}}(Ns, t) = \boldsymbol{x}(s, t)$. Due to the coordinate transformation $\widetilde{s} = Ns$, the points $\boldsymbol{x}_i(t)$ now are approximations to $\widetilde{\boldsymbol{x}}(i, t)$. Recall that $\boldsymbol{x}$ satisfies equation (2.16):

$$\frac{\partial \boldsymbol{x}}{\partial t}(s,t) = \alpha \frac{\partial^2 \boldsymbol{x}}{\partial s^2}(s,t) - \beta \frac{\partial^4 \boldsymbol{x}}{\partial s^4}(s,t) + \gamma \boldsymbol{u}(\boldsymbol{x}(s,t)). \tag{3.1}$$

By the chain rule, we have

$$\frac{\partial \widetilde{\boldsymbol{x}}}{\partial \widetilde{s}}(\widetilde{s}, t) = \frac{\partial}{\partial \widetilde{s}} \boldsymbol{x}\left(\widetilde{s}/N, t\right) = \frac{\partial \boldsymbol{x}}{\partial s}\left(\widetilde{s}/N, t\right) \frac{1}{N} = \frac{1}{N} \frac{\partial \boldsymbol{x}}{\partial s}(s, t).$$

From which it follows that $\widetilde{\boldsymbol{x}}$ satisfies the partial differential equation

$$\frac{\partial \widetilde{\boldsymbol{x}}}{\partial t}(\widetilde{s}, t) = \alpha N^2 \frac{\partial^2 \widetilde{\boldsymbol{x}}}{\partial \widetilde{s}^2}(s,t) - \beta N^4 \frac{\partial^4 \widetilde{\boldsymbol{x}}}{\partial \widetilde{s}^4}(s,t) + \gamma \boldsymbol{u}(\widetilde{\boldsymbol{x}}(\widetilde{s}, t)). \tag{3.2}$$

Intuitively, this makes sense. By introducing the coordinate transformation $\widetilde{s} = Ns$, one revolution around the curve takes N times more 'time'. Roughly speaking, the speed at which we move along the curve is N times lower. Hence, the derivatives with respect to $\widetilde{s}$ are N times smaller. To counteract this, we need to multiply the second and fourth partial derivatives with the factors N^2 and N^4 respectively.

From now on we will drop the tildes, and write

$$\frac{\partial \boldsymbol{x}}{\partial t} = \alpha N^2 \frac{\partial^2 \boldsymbol{x}}{\partial s^2} - \beta N^4 \frac{\partial^4 \boldsymbol{x}}{\partial s^4} + \gamma \boldsymbol{u}(\boldsymbol{x}) \tag{3.3}$$

whenever it is clear that $\boldsymbol{x}$ is approximated by N points. Hence, for $i = 0, \ldots, N-1$; $\boldsymbol{x}_i(t)$ is the approximation to $\boldsymbol{x}(i, t)$. However, it turns out that keeping α and β fixed as N grows is more intuitive. If we indeed fix α and β and let N grow, we are in fact keeping the distance between to adjacent points constant. Increasing N means increasing the size of the curve, whereas increasing α and/or β means decreasing the size of the curve as well as making it smoother. For this reason we let α and β be independent of N. The dynamic snake equation for a snake consisting of N points is again given by

$$\frac{\partial \boldsymbol{x}}{\partial t} = \alpha \frac{\partial^2 \boldsymbol{x}}{\partial s^2} - \beta \frac{\partial^4 \boldsymbol{x}}{\partial s^4} + \gamma \boldsymbol{u}(\boldsymbol{x}). \tag{3.4}$$

3.2 Discretization of Dynamic Snake Equation

We write $\boldsymbol{x}_i(t) = (x_i(t), y_i(t))$, and define the vectors $\boldsymbol{X}(t)$ and $\boldsymbol{Y}(t)$ containing the unknowns at time t:

$$\boldsymbol{X}(t) = \begin{pmatrix} x_0(t) \\ x_1(t) \\ \vdots \\ x_{N-1}(t) \end{pmatrix}, \quad \boldsymbol{Y}(t) = \begin{pmatrix} y_0(t) \\ y_1(t) \\ \vdots \\ y_{N-1}(t) \end{pmatrix}.$$

Then the second order partial derivative with respect to s is approximated by a central difference quotient:

$$\frac{\partial^2 \boldsymbol{x}_i}{\partial s^2} \approx \boldsymbol{x}_{i-1} - 2\boldsymbol{x}_i + \boldsymbol{x}_{i+1} \tag{3.5}$$

By the periodicity of $\boldsymbol{x}$, subscripts can be interpreted 'modulo N'. For example $\boldsymbol{x}_{-1} = \boldsymbol{x}_{N-1}$. Note that it seems a Δs^2 is missing in the denominator[3]. However, by the coordinate transformation defined in 3.1, we have $\Delta s = 1$. The fourth order partial derivative with respect to s is also approximated by a central different quotient:

$$\frac{\partial^4 \boldsymbol{x}}{\partial s^4} \approx \boldsymbol{x}_{i-2} - 4\boldsymbol{x}_{i-1} + 6\boldsymbol{x}_i - 4\boldsymbol{x}_{i+1} + \boldsymbol{x}_{i+2} \tag{3.6}$$

Define the $N \times N$ matrices

$$D_2 = \begin{pmatrix} -2 & 1 & & & & 1 \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ 1 & & & & 1 & -2 \end{pmatrix}, \quad D_4 = \begin{pmatrix} 6 & -4 & 1 & & & 1 & -4 \\ -4 & 6 & -4 & 1 & & & 1 \\ 1 & -4 & 6 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 6 & -4 & 1 \\ 1 & & & 1 & -4 & 6 & -4 \\ -4 & 1 & & & 1 & -4 & 6 \end{pmatrix}. \tag{3.7}$$

Note that $D_2^2 = D_4$. This equality is expected, as applying the second derivative twice yields the fourth derivative. Equation (3.4) is then approximated by the following linear system of ODE's:

$$\begin{cases} \boldsymbol{X}'(t) &= (\alpha D_2 - \beta D_4)\, \boldsymbol{X}(t) + \gamma \boldsymbol{U}(\boldsymbol{X}(t), \boldsymbol{Y}(t)) \\ \boldsymbol{Y}'(t) &= (\alpha D_2 - \beta D_4)\, \boldsymbol{Y}(t) + \gamma \boldsymbol{V}(\boldsymbol{X}(t), \boldsymbol{Y}(t)) \end{cases}. \tag{3.8}$$

Where $\boldsymbol{U}_i(\boldsymbol{X}(t), \boldsymbol{Y}(t)) = u(x_i(t), y_i(t))$ and $\boldsymbol{V}_i(\boldsymbol{X}(t), \boldsymbol{Y}(t)) = v(x_i(t), y_i(t))$. The semi-discrete system in (3.8) can be solved by any time stepping method. For simplicity's sake, the explicit Euler method is chosen. Given some inititial curve $(\boldsymbol{X}(0), \boldsymbol{Y}(0))$ and time step $\Delta t > 0$, the method is given by:

$$\begin{cases} \boldsymbol{X}(t + \Delta t) &= \boldsymbol{X}(t) + \Delta t \Big[(\alpha D_2 - \beta D_4)\, \boldsymbol{X}(t) + \gamma \boldsymbol{U}(\boldsymbol{X}(t), \boldsymbol{Y}(t)) \Big] \\ \boldsymbol{Y}(t + \Delta t) &= \boldsymbol{Y}(t) + \Delta t \Big[(\alpha D_2 - \beta D_4)\, \boldsymbol{Y}(t) + \gamma \boldsymbol{V}(\boldsymbol{X}(t), \boldsymbol{Y}(t)) \Big] \end{cases}. \tag{3.9}$$

3.3 Computing the Gradient Vector Flow Field

In order to implement the discretization found in section 3.2, one needs to know the values of the gradient vector flow field $\boldsymbol{u} = (u, v)$. This flow field satisfies equations (2.14a) and (2.14b). We will derive the discrete system for the u-component. Using that u and v satisfy a very similar equation,

[3] See any book on numerical analysis.

one can then immediately produce the discrete system for v.

Recall that it was assumed in section 2.2 that $\nabla u \cdot \boldsymbol{n} = \nabla v \cdot \boldsymbol{n} = 0$ on the boundaries. Hence, u is the solution to the following boundary value problem:

$$\begin{cases} \mu\nabla^2 u - gu &= -gf_x, & \text{in } \Omega \\ \nabla u \cdot \boldsymbol{n} &= 0, & \text{on } \partial\Omega \end{cases}, \quad \text{where } g = f_x^2 + f_y^2. \tag{3.10}$$

Assume that Ω is the domain of an image I consisting of $n \times m$ pixels. That is, there are n pixels in the x-direction and m pixels in the y-direction. Let u_{ij} be the (unknown) value of u at pixel (i, j) for $i \in \{0, \ldots, n-1\}$ and $j \in \{0, \ldots, m-1\}$, see Figure 3.1. The second derivatives of u with

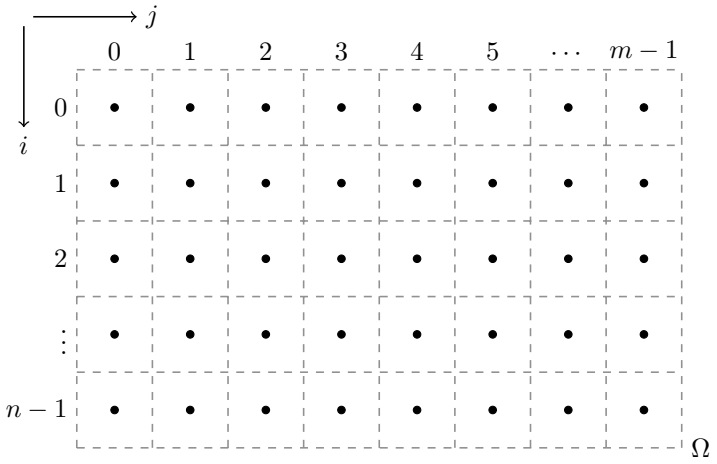

Figure 3.1: Locations of unknowns.

respect to x and y are approximated by a central difference. For a pixel (i, j) that is not adjacent to a boundary, these approximations are given by

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_{ij} \approx u_{i-1,j} - 2u_{ij} + u_{i+1,j}, \quad \left(\frac{\partial^2 u}{\partial y^2}\right)_{ij} \approx u_{i,j-1} - 2u_{ij} + u_{i,j+1}. \tag{3.11}$$

If (i, j) is adjacent to a boundary, the Neumann boundary condition is used. For example, if $i = 0$, we let $u_{-1,j} = u_{0,j}$ in order to approximate the derivative normal to the boundary being 0. Define the matrices

$$L_{xx} = \begin{pmatrix} -1 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -1 \end{pmatrix}, \quad L_{yy} = \begin{pmatrix} -1 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -1 \end{pmatrix}, \tag{3.12}$$

where L_{xx} is an $n \times n$ matrix and L_{yy} is $m \times m$. Then for all $i \in \{0, \ldots, n-1\}$ and $j \in \{0, \ldots, m-1\}$ we have

$$\begin{pmatrix} \left(\frac{\partial^2 u}{\partial y^2}\right)_{i,0} \\ \vdots \\ \left(\frac{\partial^2 u}{\partial y^2}\right)_{i,m-1} \end{pmatrix} \approx L_{yy} \begin{pmatrix} u_{i,0} \\ \vdots \\ u_{i,m-1} \end{pmatrix}, \quad \begin{pmatrix} \left(\frac{\partial^2 u}{\partial x^2}\right)_{0,j} \\ \vdots \\ \left(\frac{\partial^2 u}{\partial x^2}\right)_{n-1,j} \end{pmatrix} \approx L_{xx} \begin{pmatrix} u_{0,j} \\ \vdots \\ u_{n-1,j} \end{pmatrix}. \tag{3.13}$$

The discrete Laplacian L corresponding to the lexicographical order of unknowns[4] is then given by

$$L = I_y \otimes L_{xx} + L_{yy} \otimes I_x, \quad \text{where } \otimes \text{ denotes the Kronecker product,} \tag{3.14}$$

[4]The unknowns are ordered first in the x-direction and then in the y-direction, resulting in the unkown vector $\widehat{\boldsymbol{u}} = (u_{0,0}, \ldots, u_{n-1,0}, u_{0,1}, \ldots, u_{n-1,m-1})^\top$.

and I_y, I_x are the $m \times m$ and $n \times n$ identity matrices respectively. If A is an $m \times n$ matrix and B is a $p \times q$ matrix, the Kronecker product is defined as the $mp \times nq$ block matrix

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}.$$

Thus, L is an $nm \times nm$ matrix. Let g_{ij} be the value of g at pixel (i, j), then we can write $g_{ij} = (f_x)_{ij}^2 + (f_y)_{ij}^2$ where $(f_x)_{ij}$ and $(f_y)_{ij}$ are the values of f_x and f_y at pixel (i, j) respectively. Define $G_j = \operatorname{diag}(g_{0,j}, \ldots, g_{n-1,j})$ for $j = 0, \ldots, m-1$ and let G be the $nm \times nm$ block matrix

$$G = \begin{pmatrix} G_0 & & \\ & \ddots & \\ & & G_{m-1} \end{pmatrix}. \tag{3.15}$$

The discretized systems corresponding to equations (2.14a) and (2.14b), taking into account the boundary conditions, are given respectively by:

$$(\mu L - G)\widehat{\boldsymbol{u}} = -G\boldsymbol{f}_x \tag{3.16a}$$
$$(\mu L - G)\widehat{\boldsymbol{v}} = -G\boldsymbol{f}_y \tag{3.16b}$$

Where $\widehat{\boldsymbol{u}}$ and $\widehat{\boldsymbol{v}}$ are ordered lexicographically. The vectors $\boldsymbol{f}_x$ and $\boldsymbol{f}_y$ contain the values of f_x and f_y at the pixels respectively. These values of f_x and f_y also need to be computed, as only f is given. Central differences are used in pixels away from the border, whereas one-sided differences are used in pixels adjacent to the border:

$$(f_x)_{ij} = \begin{cases} f_{i+1,j} - f_{ij}, & \text{if } i = 0 \\ f_{ij} - f_{i-1,j}, & \text{if } i = n-1 \\ \frac{1}{2}\left(f_{i+1,j} - f_{i-1,j}\right), & \text{otherwise} \end{cases}, \tag{3.17a}$$

$$(f_y)_{ij} = \begin{cases} f_{i,j+1} - f_{ij}, & \text{if } j = 0 \\ f_{ij} - f_{i,j-1}, & \text{if } j = m-1 \\ \frac{1}{2}\left(f_{i,j+1} - f_{i,j-1}\right), & \text{otherwise} \end{cases}. \tag{3.17b}$$

While it seems that the use of '$=$' instead of '$\approx$' is abuse of notation, the function f is only defined at the locations of the pixels. Hence, it doesn't make sense to consider its 'true' partial derivatives. The vectors $\boldsymbol{f}_x$ and $\boldsymbol{f}_y$ contain the elements given in (3.17a) and (3.17b) in lexicographical order. Let F be the $n \times m$ matrix containing the values of f, such that $F_{ij} = f_{ij}$:

$$F = \begin{pmatrix} f_{0,0} & f_{0,1} & \cdots & f_{0,m-1} \\ f_{1,0} & f_{1,1} & \cdots & f_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ f_{n-1,0} & f_{n-1,1} & \cdots & f_{n-1,m-1} \end{pmatrix}. \tag{3.18}$$

Then the $n \times m$ matrices F_x and F_y, defined analogously to F but for f_x and f_y, can be written as $F_x = C_x F$ and $F_y = F C_y^\top$. Here C_x is an $n \times n$ matrix and C_y is an $m \times m$ matrix, both of the form

$$\begin{pmatrix} -1 & 1 & & & \\ -\frac{1}{2} & 0 & \frac{1}{2} & & \\ & \ddots & \ddots & \ddots & \\ & & -\frac{1}{2} & 0 & \frac{1}{2} \\ & & & -1 & 1 \end{pmatrix}. \tag{3.19}$$

Once F_x and F_y have been computed, the vectors $\boldsymbol{f}_x$ and $\boldsymbol{f}_y$ can be created by placing all columns of F_x and F_y on top of each other. This is commonly denoted by $\boldsymbol{f}_x = \operatorname{vec}(F_x)$ and $\boldsymbol{f}_y = \operatorname{vec}(F_y)$.

A note on Implementation

A practical problem that is encountered when implementing the linearized system is that the matrix L will become large. Images with a size of 1000×1000 pixels are nothing out of the ordinary, but in this case L will be a $10^6 \times 10^6$ matrix. Performance will likely suffer drastically if L is implemented as a regular array. Using the fact that L contains many zeroes, it can be implemented as a sparse matrix. In a sparse matrix, only the nonzero elements are stored. More details will follow in section 4. Note that we can write L_{xx} and L_{yy} as a product of two simple (to initialize) matrices: $L_{xx} = -D_x^\top D_x$ and $L_{yy} = -D_y^\top D_y$, where

$$D_x = \underbrace{\begin{pmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \end{pmatrix}}_{n} \Bigg\} n-1, \quad D_y = \underbrace{\begin{pmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \end{pmatrix}}_{m} \Bigg\} m-1\,.$$

To see where these products come from, write $u_{i-1,j} - 2u_{ij} + u_{i+1,j} = (u_{i+1,j} - u_{ij}) - (u_{ij} - u_{i-1,j})$. In essence, the second order central difference formula is the difference of two first order central differences, each approximating the first derivative between two pixels. Multiplying by D_x or D_y gives the central differences between pixels, and subsequently multiplying by $D_x^\top$ or $D_y^\top$ yields (minus) the difference of these differences.

4 Implementation

In this chapter we will describe the required steps to convert a microscopic image to a collection of curves that can be used for statistical analysis. The model is displayed schematically in Figure 4.1, along with the parameters needed at every point and the corresponding section. One starts with the image. If this image is too big, it can be resized to a suitable size. This is done to somewhat reduce the time it takes to find the GVF field. Although the image quality is lowered, we observe that the results are still acceptable. In practice, the resizing step is only done for very large images, those in the order of tens of megabytes.

The model is then split into two. In the left branch, the cell centers are detected and initial curves are placed. Finding the cell centers is not a trivial task, the process is described in section 4.1. In the right branch the GVF field is calculated. The image is first converted to another image f by a simple preprocessing algorithm, given in section 4.2. Calculating the GVF field takes the most amount of time out of all the steps. This is due to the fact that two very large systems of linear equations (3.16a) and (3.16b) must be solved. In section 4.4 a number of different solvers are compared. Finally, the two branches come together and the snake algorithm can be performed. The snake algorithm simply iterates equation (3.9) a number of times for every curve.

In section 4.3 we pay some attention to areas of the image in which no cells are present. Since they do not contain any cells, these areas might as well be omitted. We call this step cropping and present a method of doing this while avoiding discarding valuable information. Note that this step is performed after cell centers have been detected. The aim is to find wronly detected centers. The preprocessing algorithm is also used in this step, accounting for the arrow pointing from the preprocessing step to the cropping step.

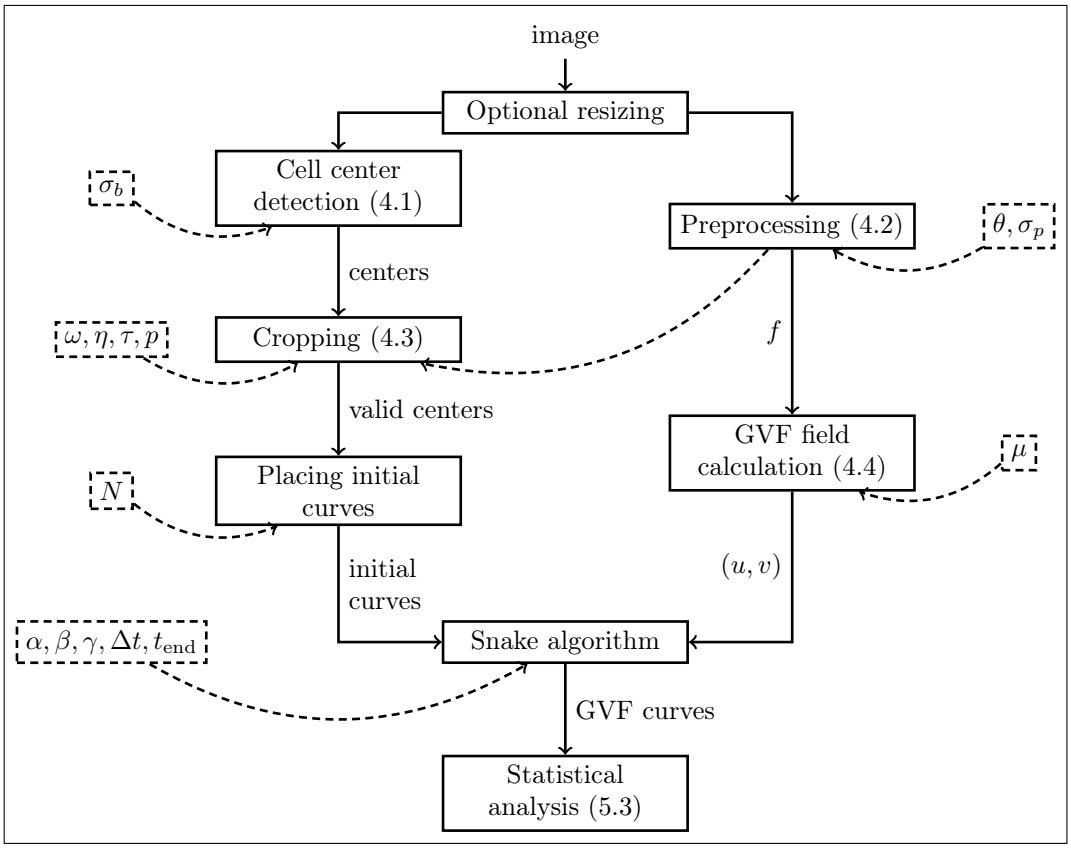

Figure 4.1: Schematic view of model.

4.1 Cell Center Detection

Cell centers are found by detecting local maxima in the image convolved with a Gaussian. By convolving with a Gaussian kernel, the image is made smooth. Every cell has a more or less constant interior. Smoothing the image turns these 'plateaus' into 'hills', roughly speaking. If the smoothing parameter σ is chosen just right, every one of these hills (cells) has exactly one maximum. This process can be seen in Figure 4.2. Finding local maxima can be done by a number of standard algorithms that are generally very fast. The problem lies thus in finding the right value for σ.

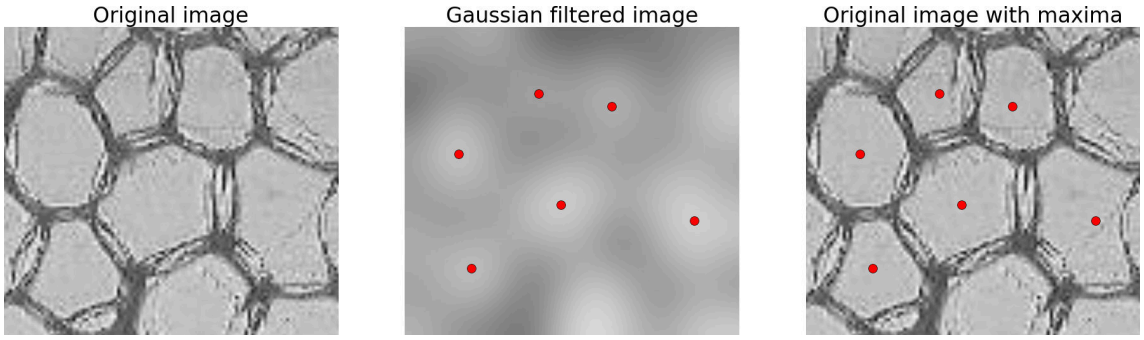

Figure 4.2: Finding cell centers by convolution with Gaussian with a good choice of σ.

We define the 'best choice' for σ to be the value σ_{opt} for which the number of local maxima found in $G_{\sigma_{\text{opt}}} * I$ is equal to the number of cells in I, where I is the original image. This definition seems to only move the problem somewhere else. After all, how does one count the number of cells in I? While the exact number of cells is very hard to count, it can be rather accurately approximated

by algorithm 4.1. This algorithm first slightly smooths I using a Gaussian kernel with $\sigma = \sigma_b$. Then for each row the mean of I is calculated, and the number of times the value of I crosses the mean is counted. This corresponds to the number of times a boundary is crossed, so the number of cells in this row is roughly half the number of counted 'boundary crossings'. By doing this for every row, the average number of cells in the x-direction can be calculated. Repeat this process for every column to obtain the average number of cells in the y-direction. The product of these two averages is the approximated amount of cells in I.

Algorithm 4.1 Approximate number of cells

Require: $n \times m$ image I, $\sigma_b > 0$
  $I^* \leftarrow G_{\sigma_b} * I$ ▷ Convolution with Gaussian to smooth data
  $S_x, S_y \leftarrow 0$
  for $i = 1, \ldots, n$ **do** ▷ First scan all rows
    $\mu \leftarrow \frac{1}{m} \sum_{j=1}^{m} I^*_{ij}$ ▷ Mean of i^{th} row
    for $j = 1, \ldots, m$ **do**
      $b_j \leftarrow 0$
      if $I^*_{ij} < \mu$ **then** $b_j \leftarrow 1$
      end if ▷ Set to 1 if value is below mean
    end for
    $c \leftarrow \frac{1}{2} \sum_{j=1}^{m-1} |b_{j+1} - b_j|$ ▷ Half times the number of 'boundary crossings'
    $S_x \leftarrow S_x + c/n$ ▷ Average number of cells in all rows
  end for
  for $j = 1, \ldots, m$ **do** ▷ Do same thing for every column
    $\mu \leftarrow \frac{1}{n} \sum_{i=1}^{n} I^*_{ij}$
    for $i = 1, \ldots, n$ **do**
      $b_i \leftarrow 0$
      if $I^*_{ij} < \mu$ **then** $b_i \leftarrow 1$
      end if
    end for
    $c \leftarrow \frac{1}{2} \sum_{i=1}^{n-1} |b_{i+1} - b_i|$
    $S_y \leftarrow S_y + c/m$ ▷ Average number of cells in all columns
  end for
  return $S_x S_y$ ▷ Product of average number of cells in both directions

The reason for first slightly smoothing I is to make the number of boundary crossings more accurate. Values of I at cell boundaries can jump a lot, which can result in detecting too many boundary crossings. By smoothing I, these jumps are dampened. We found that $\sigma_b \in [1, 3]$ usually yields good results. Note that the algorithmic complexity of algorithm 4.1 is $\mathcal{O}(nm)$. This means that for large images it can be quite slow. This is easily fixed by not considering all rows and columns, but only a small number of them. For example, say I has dimensions 100×100. Instead of looping over all 100 rows and columns, only take into account rows and columns $1, 11, 21, \ldots, 91$. In this case a $10\times$ speedup is achieved, not considering the time it takes to compute $G_{\sigma_b} * I$.

To summarize, the smoothing parameter σ_{opt} used to find the cell centers, by finding the local maxima in $G_{\sigma_{\text{opt}}} * I$, is given by

$$\sigma_{\text{opt}} = \arg\min_{\sigma > 0} \big| \#\text{LM}\,(G_\sigma * I) - S(I) \big|, \tag{4.1}$$

where $S(I)$ is the approximated number of cells in I obtained by algorithm 4.1, and $\#\text{LM}\,(G_\sigma * I)$ denotes the number of local maxima in $G_\sigma * I$. Searching for σ_{opt} can be time consuming. Its value also depends on the resolution of the image. If the image has a high resolution[5], σ_{opt} will be bigger. In Figure 4.3 the method is applied to an image with dimensions 1091×1267. First $S(I)$ is calculated by considering 8 rows and columns. The approximated number of cells in this case is $S(I) = 1111$. Then σ_{opt} is found using (4.1). In order to speed up this search, σ is incremented by steps of 0.5. The resulting value is $\sigma_{\text{opt}} = 6.0$, which produces 1129 cell centers.

[5]By high resolution we mean more pixels per cell, not just more pixels.

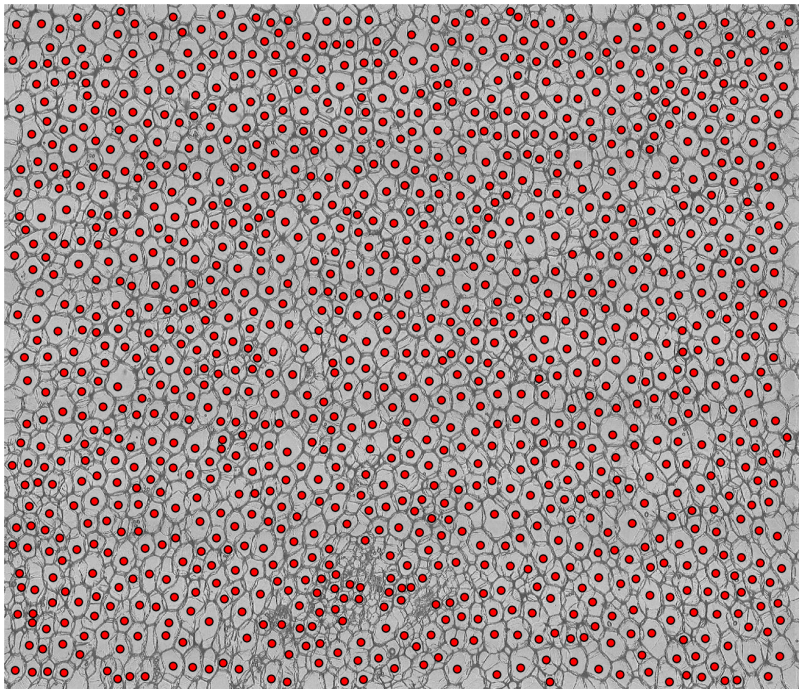

Figure 4.3: Cell center detection in larger image, using $\sigma_{\text{opt}} = 6.0$ as obtained by equation (4.1).

Note that no cell centers are detected at the image border, where a number of cells are cut off. This is a result of the local maximum detection algorithm in conjunction with the applied Gaussian filter. We deem this behaviour welcome. The method also works rather well on more 'messy' images, where cell borders are less clear. In Figure 4.4 we have $S(I) = 343$ by considering 8 rows and columns. This results in $\sigma_{\text{opt}} = 4.5$ producing 337 cell centers. Note that σ is again incremented by steps of 0.5.

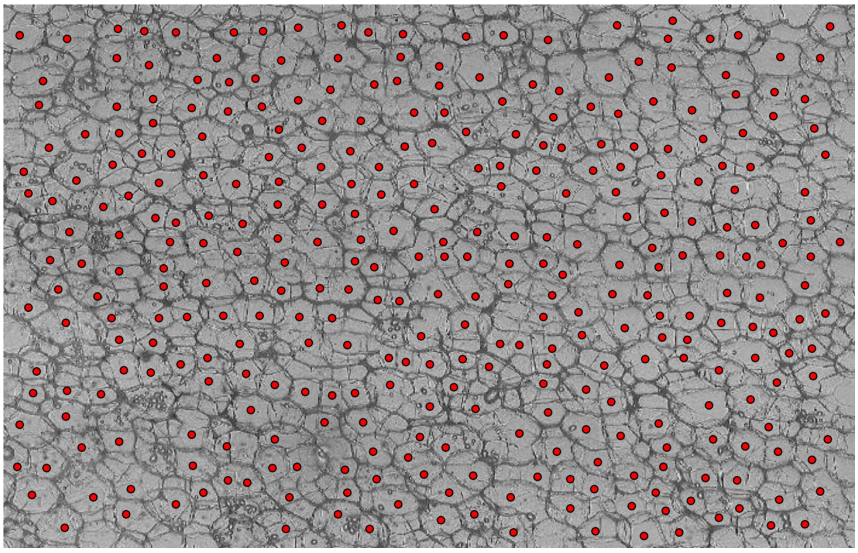

Figure 4.4: Cell center detection in image with dimensions 449×704 containing vague cell boundaries.

As can be seen, not all cells have been detected. However, as a lot of cell boundaries are rather vague, the result is acceptable. In both Figure 4.3 and 4.4 $\sigma_b = 2.5$ is used to slightly smooth the image beforehand. Note that σ_b is the only input parameter of this algorithm. We examine the effect it has on the approximated number of cells $S(I)$, by computing $S(I)$ for a number of values

of σ_b in the range $[0, 10]$. We take into account 50 rows and 50 columns of a small section of a microscopic image. The results can be seen in Figure 4.5.

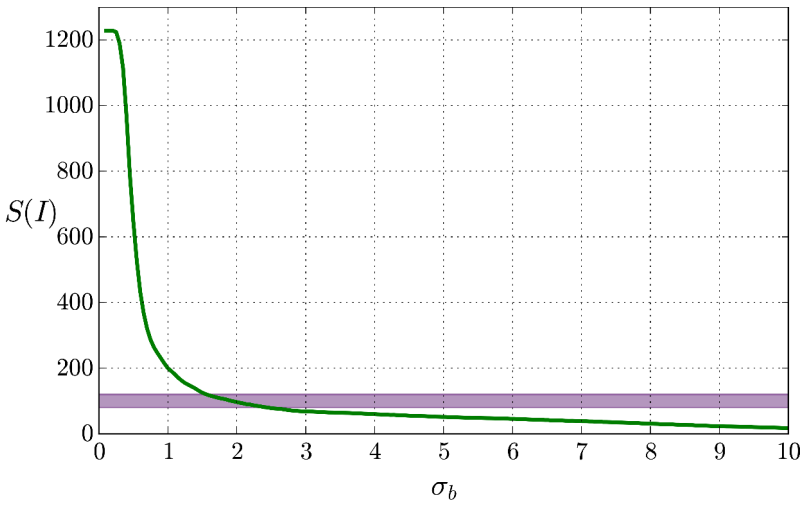

Figure 4.5: Approximated number of cells $S(I)$ as a function of the smoothing parameter σ_b, computed for a section of image 1;US;2017;FIELD1;T14;PITH.REP2.

About 95 cells are counted by hand. The purple band in Figure 4.5 ranges from 80 to 120 on the vertical axis, representing an 'acceptable' number of cells. The corresponding values of σ_b are roughly in the interval $[1.5, 2.5]$. Good values of σ_b depend on the image itself. If a lot of noise is present, a slightly higher value of σ_b should be chosen. If σ_b is too small, multiple centers will be detected in one cell. When the snake algorithm is applied to such a cell, the resulting snakes corresponding to each center will be very similar. It is therefore possible to discard duplicate snakes at the end. For this reason we propose that σ_b should never be too high. If it is too high, many cells will not be detected, which can't be fixed afterwards.

4.2 Preprocessing

Preprocessing is used to convert the original image I to the image f on which the GVF field is calculated. The algorithm is rather simple, and is given in pseudocode in algorithm 4.2. Given a ratio θ and smoothing parameter σ_p, the new image f is calculated by comparing the value of every pixel to $\theta\mu$. Here, μ is the mean of I. If a pixel has a higher value than $\theta\mu$, it will be assigned a new value of 1. Else, it will be assigned a new value of 0. The idea being that the cell boundaries will have value of 0, whereas the interior of the cell will have a value of 1. After this process is completed, a binary image is obtained containing only 1's and 0's. The final step is convolving this binary image with a Gaussian with smoothing parameter $\sigma = \sigma_p$. This is done to make the gradients smaller in the areas where values go from high to low.

Algorithm 4.2 Preprocessing

Require: $n \times m$ image I, $\theta > 0, \sigma_p > 0$
  $\mu \leftarrow \frac{1}{nm} \sum_{i=1}^{n} \sum_{j=1}^{m} I_{ij}$ ▷ Mean of image
  for $i = 1, \ldots, n$ **do**
    for $j = 1, \ldots, m$ **do**
      if $I_{ij} > \theta\mu$ **then**
        $I'_{ij} \leftarrow 1$
      else
        $I'_{ij} \leftarrow 0$
      end if
    end for
  end for
  $f \leftarrow G_{\sigma_p} * I'$ ▷ Convolution with Gaussian to obtain smoother image
  return f

The value of θ should not be too high, otherwise the resulting binary image will only contain zeroes. We found that $\theta \in [0.7, 1.0]$ works well, but bear in mind that it is dependent on the image and the desired result. Furthermore, σ_p should be small. The smoothing step is really only used to soften the high gradients at cell borders, not to make the whole image smooth. The GVF field itself will 'fill in' areas with little data, which are the cell interiors. All $\sigma_p \leq 1.5$ seem to work quite well. Figure 4.6 shows the result of algorithm 4.2 when applied to an image.

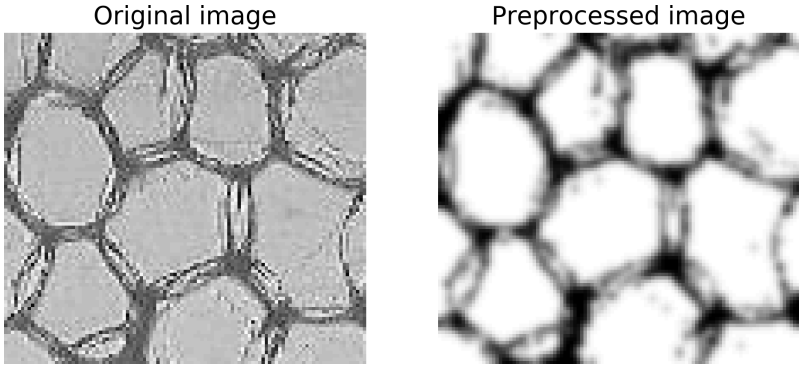

Figure 4.6: Result of preprocessing algorithm, using $\theta = 0.8$ and $\sigma_p = 1.2$.

This algorithm is used to remove some of the distortion inside the cell interiors. The idea is that the whole cell interior has a value of 1, such that the GVF field can work its magic on these homogeneous regions.

4.3 Cropping

Sometimes it happens that cell centers are (wrongly) detected in regions where no cells are present. We call these regions *voids*. See for example the upper right corner of Figure 5.7. The resulting GVF snakes are often near-perfect circles, skewing the data. A way to prevent this from happening is to crop the image such that it doesn't contain any voids. Cropping the image to a new rectangular image often means regions that do contain cells are also discarded, which we want to avoid. We present a method that detects voids, such that wrongly detected cell centers in these regions can be deleted.

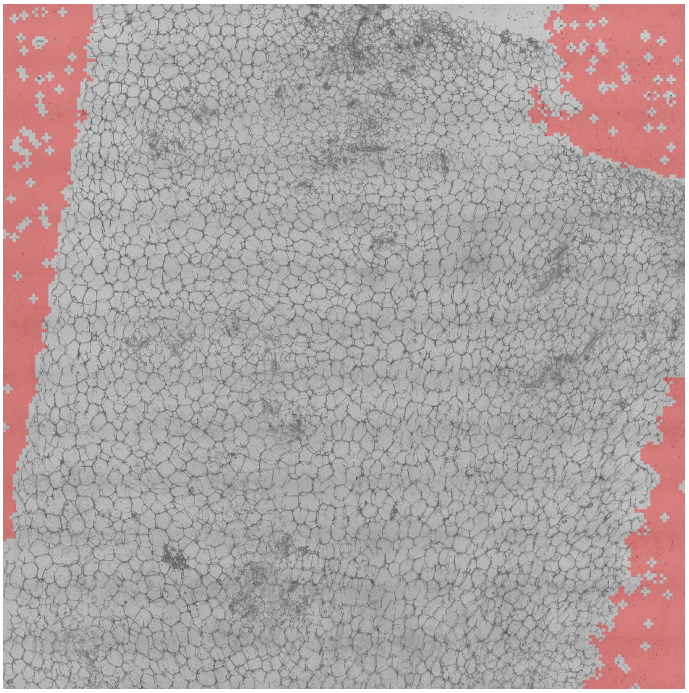

Figure 4.7: The areas shaded red are the detected voids in image 2;US;2017;FIELD1;T9;CORTEX.REP1.

Voids are detected by finding connected regions in which the gradient of the image is small. That is, for a $n \times m$ image I, a binary image B of equal dimensions is created as follows:

$$B_{ij} = \begin{cases} 1, & \text{if } \|\nabla I_{ij}\| \leq \tau \\ 0, & \text{otherwise} \end{cases}, \tag{4.2}$$

for some $\tau > 0$. We are searching for so-called connected components of B, by means of a connected-component labeling algorithm.

Definition 4.1. A connected component $\mathcal{C}$ of a binary matrix A is an equivalence class corresponding to the relation

$$(i,j) \sim (k,l) \iff A_{ij} = A_{kl} = 1 \quad \text{and both entries are connected via a 'Manhattan-path' consisting of only ones.}$$

'Manhattan-path' is a path such that two adjacent entries are either vertical or horizontal neighbours.

Note that by definition 4.1, an isolated '1' is a connected component, as there is always a path from an entry to itself. The matrix

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

has two connected components, a red and blue one. The connected components of B represent regions of I in which the gradient ∇I is small. Typically, cell interiors and voids are such regions. The main difference between cell interiors and voids is that voids are much larger. This is the criterium for a connected component to be deemed a void.

Definition 4.2. A connected component $\mathcal{C}$ of B is called a *void* of I if $|\mathcal{C}| \geq p \cdot nm$, for some $p \in [0,1]$.

In other words, $\mathcal{C}$ is a void if its area - the number of pixels it contains - is greater than some fraction p of the total area of I. Searching for the connected components of B takes a long time in practice. For this reason, we first make I significantly smaller by resizing it by a factor $1/\eta$. Before this is done, a minimum filter with size ω is applied to I to make cell boundaries more prominent. Finally, the preprocessing algorithm from section 4.2 is applied to the resized image. In short:

$$I \xmapsto{\text{Min. Filter}} I' \xmapsto{\text{Resize}} I'' \xmapsto{\text{Preprocessing}} I'''. \tag{4.3}$$

This proces is visualized in Figure 4.8. Note that I'' and I''' have dimensions $n/\eta \times m/\eta$. The binary image B is then made for I'''. When the voids have been found according to definition 4.2, they can be placed in a new $n/\eta \times m/\eta$ matrix V, defined by

$$V_{ij} = \begin{cases} 1, & \text{if } (i,j) \in \mathcal{C} \text{ for some void } \mathcal{C} \\ 0, & \text{otherwise} \end{cases}. \tag{4.4}$$

V is then upscaled to match the dimensions of I by applying a Kronecker product with an $\eta \times \eta$ matrix E_η containing only ones:

$$W = V \otimes E_\eta. \tag{4.5}$$

Every 1 in V is replaced by an $\eta \times \eta$ block of ones, resulting in a $n \times m$ matrix W. Note that in practice η will most likely not divide n and m. In fact, n and/or m may even be prime numbers. For this reason, n/η and m/η are rounded to the nearest integer. This means that W will not be exactly an $n \times m$ image. We therefore resize W to have exact the dimensions $n \times m$.

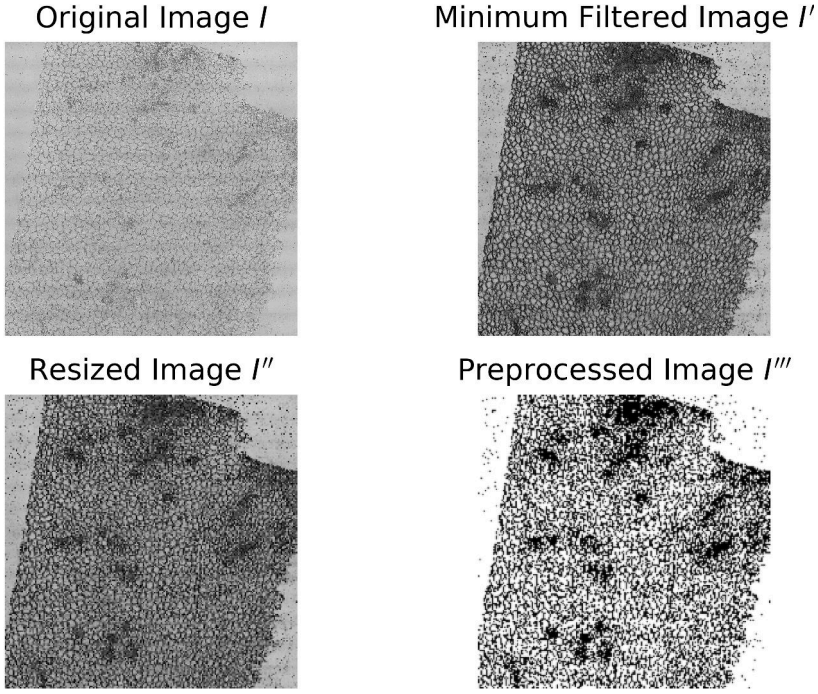

Figure 4.8: The steps of (4.3) visualised for the image 2;US;2017;FIELD1;T9;CORTEX.REP1. The parameters used are $m = 5$, $\eta = 10$, $(\theta, \sigma_p) = (0.8, 0.5)$, $\tau = 0.1$ and $p = 0.01$.

As with most numerical algorithms, a tradeoff is made between speed and accuracy. As η gets larger, the dimensions of I''' get smaller, resulting in a faster execution. However voids are given in terms of big $\eta \times \eta$ blocks, which means the accuracy is lower. We find that choosing η such that I'' roughly has dimensions 200×200 yields good enough results. Depending on the image, it takes about 1 to 2 seconds on a quad-core Linux machine[6].

[6]See the full specifications of the machine used below the table of contents.

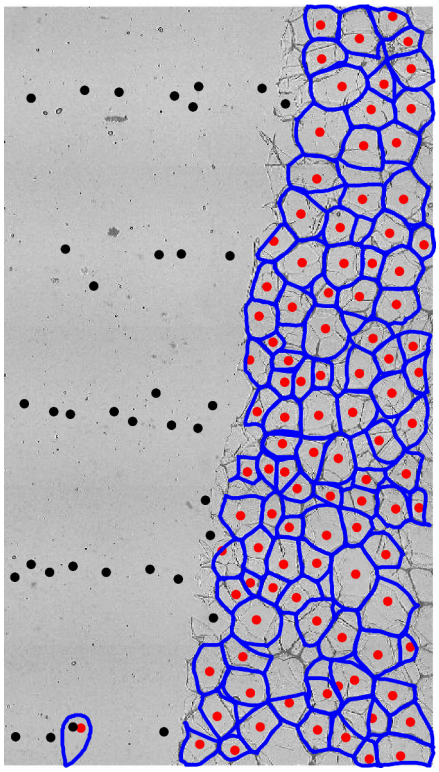

Figure 4.9: Small section of image 2;US;2017;FIELD1;T9;CORTEX.REP1 containing a large void. The black dots are cell centers detected inside this void.

In Figure 4.9 a large void is present. The cell center detection algorithm detects some centers inside this void, they are colored black. No GVF curves has been made for the black centers. Note that there is one red center that should have been black. In some cases this happens due to small dark spots in the image. We still find this to be an acceptable result, because in this case more than 95% of wrongly detected cell centers have been colored black. A possible way to erase the dark spots is to apply an open filter.

4.4 Solving GVF Systems

Recall the GVF systems (3.16a) and (3.16b):

$$(\mu L - G)\widehat{\boldsymbol{u}} = -G\boldsymbol{f}_x \tag{3.16a}$$
$$(\mu L - G)\widehat{\boldsymbol{v}} = -G\boldsymbol{f}_y \tag{3.16b}$$

We examine the performance of a number of linear solvers. Note that the matrix $\mu L - G$ is sparse, but still very large. If the original image has dimensions $n \times m$, then $\mu L - G$ has dimensions $nm \times nm$, which can get very large. We compare the solvers `CG`, `CGS`, `MINRES` and `BiCGSTAB`. In table 4.1 one can see the time it takes to solve both systems for a given image. The clear winner in this case is `MINRES`. Also note that a higher value of μ results in a longer computation time. This is expected, as a high μ means the effect of the diagonal matrix G is smaller. In this case the systems (3.16a) and (3.16b) are more like a discrete Poisson equation.

	`CG`	`CGS`	`MINRES`	`BiCGSTAB`
$\mu = 0.1$	20.7	27.0	16.8	24.4
$\mu = 0.2$	22.3	30.9	19.0	27.0
$\mu = 0.4$	25.5	35.7	21.9	32.6

Table 4.1: Performance in seconds of different solvers for image 1;US;2017;FIELD1;T9;PITH.REP2 with dimensions 1632×2112. The applied tolerance is 10^{-5}.

When solving a system $A\boldsymbol{x} = \boldsymbol{b}$, all four solvers try to minimize the norm of the residual vector $\boldsymbol{r} = A\boldsymbol{x} - \boldsymbol{b}$. The process is stopped whenever $\|\boldsymbol{r}\|$ is below some tolerance ε.

5 Results

In this chapter, we present some results obtained from the snake algorithm and apply a number of statistical tests on these results. The snake algorithm is applied to images provided by HZPC. The names of these images are quite cryptic. The only piece of information that can be extracted is whether it is a microscopic image of the pith or the cortex of a potato. These are two different regions of the potato. Pith being center of the potato, is also called the inner medulla. The cortex is the region just below the skin.

5.1 GVF Field

In Figure 5.1 and 5.2 two different GVF fields are given for one cell. In Figure 5.1 the GVF parameter is set to $\mu = 0.2$ whereas in Figure 5.2 it is set to $\mu = 0.1$. In the latter figure, the magnitude of the field is much larger near the cell boundary, relative to the cell interior. This is expected, as the GVF field $\boldsymbol{u}$ is closer to ∇f for lower values of μ. Indeed, in Figure 5.1 the magnitude near the cell boundary is relatively closer to that in the cell interior.

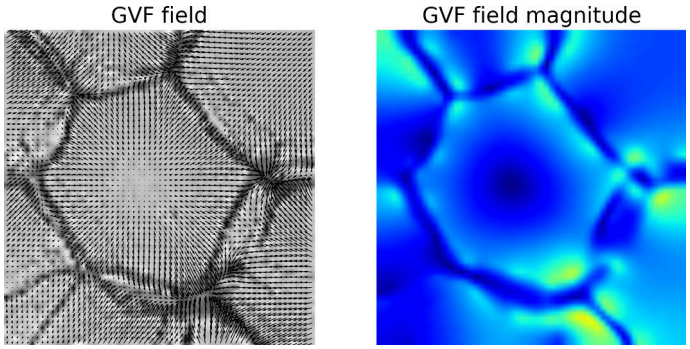

Figure 5.1: GVF field and its (relative) magnitude using preprocessing parameters $(\theta, \sigma_p) = (0.8, 1.2)$ and GVF parameter $\mu = 0.2$.

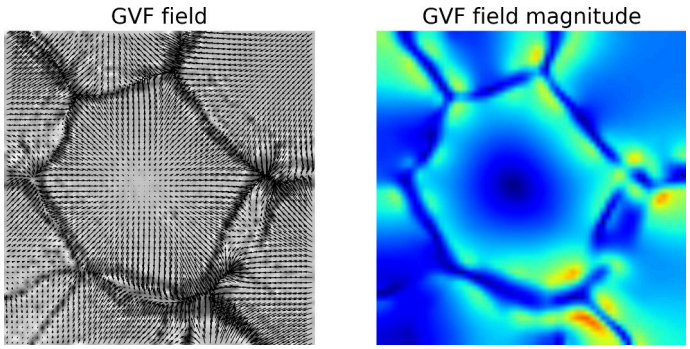

Figure 5.2: GVF field and its (relative) magnitude using preprocessing parameters $(\theta, \sigma_p) = (0.8, 1.2)$ and GVF parameter $\mu = 0.1$.

5.2 Snakes

We present some results of the algorithm applied to different images. In Figure 5.3 the evolution in time of a single snake can be seen. Note that the initial curve is quite large. Every side of the square contains 16 points, with one pixel between two adjacent points.

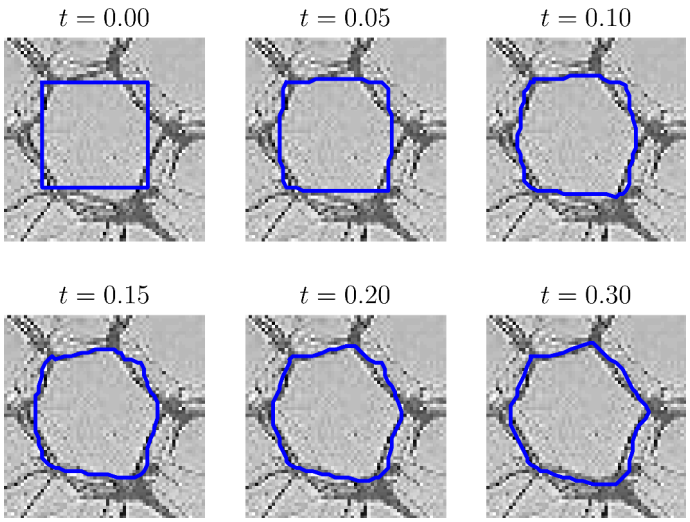

Figure 5.3: The evolution of a snake in time, subject to the GVF field. Parameters that are used are shown in table 5.1.

In Figure 5.4 a number of snakes are developed. Note that a curve can leave the image. The GVF field is extended beyond the image, such that it pushes snakes back towards the image border. The further away a point is from the image, the stronger this pushback force will be.

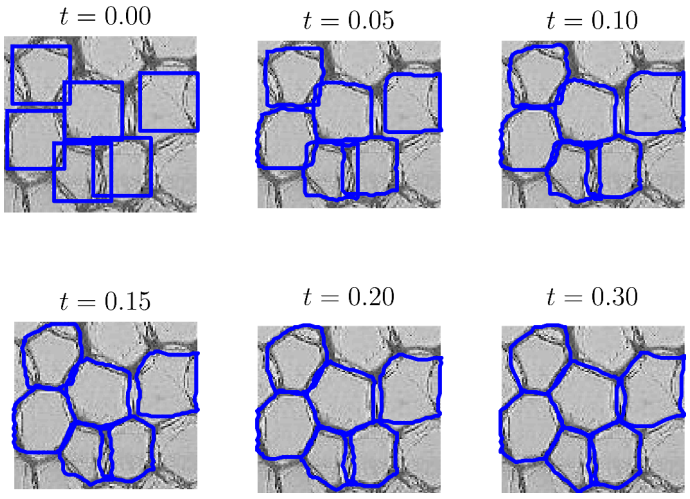

Figure 5.4: The evolution of a number of snakes in time, subject to the GVF field, again using the parameters found in table 5.1.

Table 5.1: Parameters used to generate Figures 5.3 and 5.4.

Init.	GVF			Snakes				
σ_b	μ	θ	σ_p	N	α	β	γ	Δt
3	0.2	0.8	1.2	64	4	0	250	0.01

In Figure 5.5 the result of the snake algorithm for a whole image is shown. The entire process

takes about 1 to 2 minutes to complete. Note that there is almost no space between the curves. This is an indication that the process is succesful, as there is also no space between two cells. In Figure 5.6 a small section of the same image is shown. We consider the cells to be accurately approximated by the GVF curves.

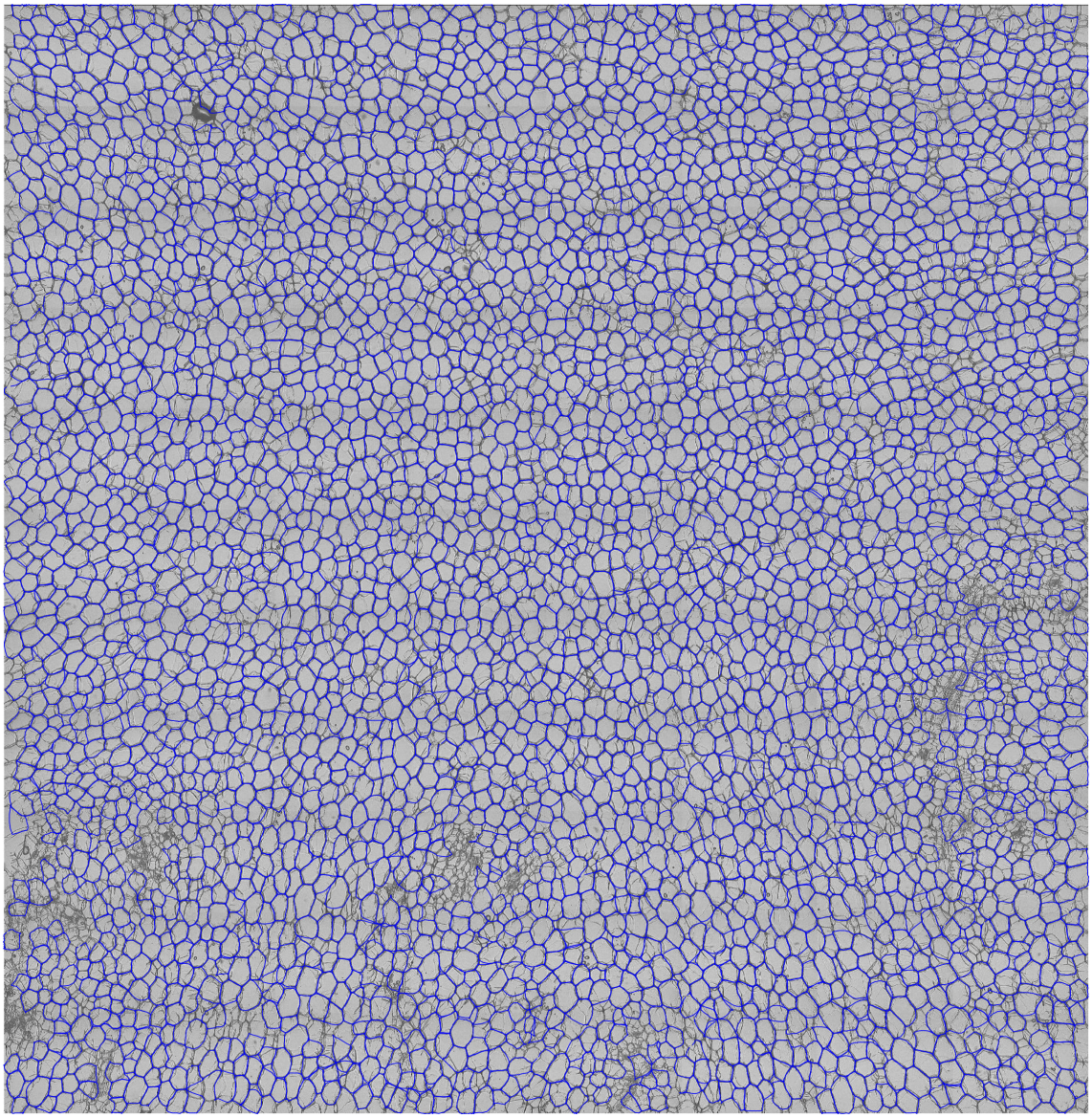

Figure 5.5: Result of snake algorithm applied to the image 1;US;2017;FIELD1;T14;PITH.REP2, using the parameters shown in table 5.2

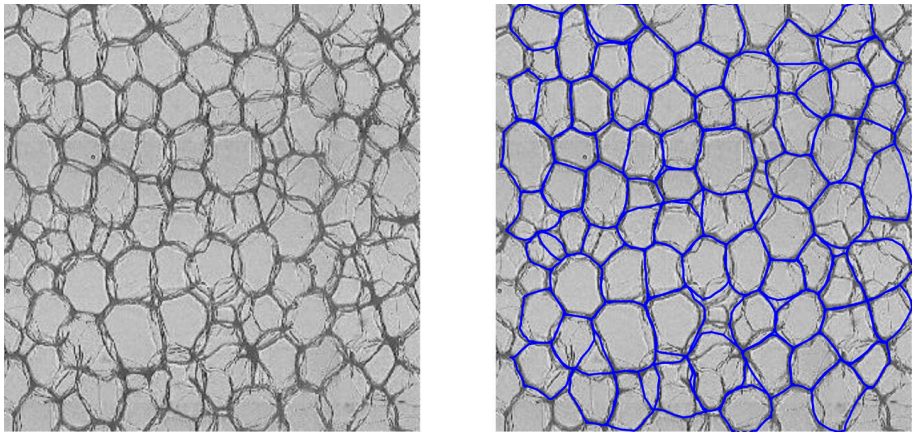

Figure 5.6: Small section of image 5.5, with and without GVF curves.

Table 5.2: Parameters used to generate Figures 5.5 and 5.6.

Init.	GVF			Snakes					
σ_b	μ	θ	σ_p	N	α	β	γ	Δt	t_{end}
3	0.2	0.8	1.2	64	4	0	250	0.04	6.0

Not every result is as good as in Figure 5.5. For example, in Figure 5.7 there is more space between the curves. While certain parameters may be tuned to obtain better results, the biggest bottleneck is the image quality: compare Figure 5.8 to 5.6.

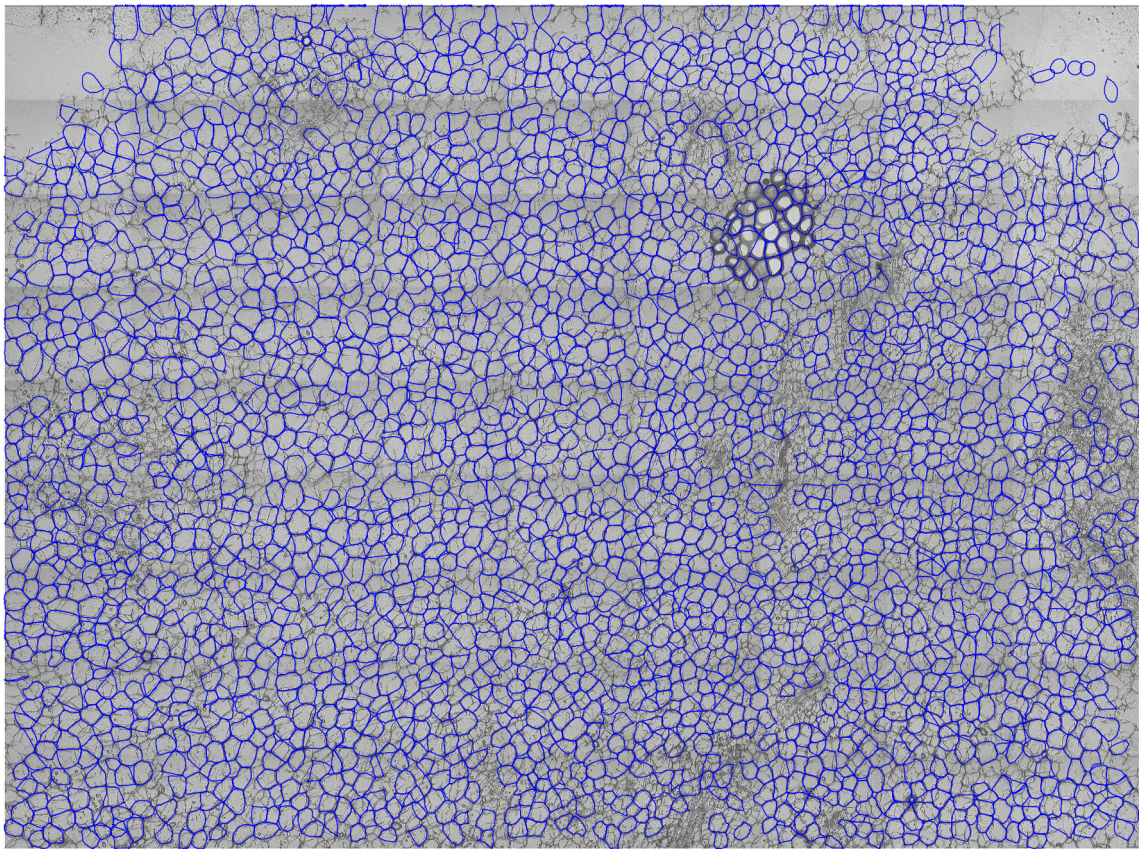

Figure 5.7: Result of snake algorithm applied to the image 1;US;2017;FIELD1;T15;PITH.REP, using the parameters found in table 5.3

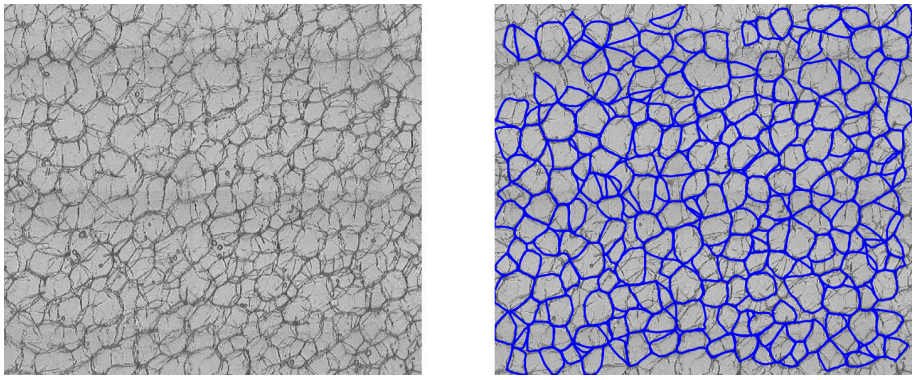

Figure 5.8: Small section of image 5.7, with and without GVF curves.

Table 5.3: Parameters used to generate Figures 5.7 and 5.8.

Init.	GVF			Snakes					
σ_b	μ	θ	σ_p	N	α	β	γ	Δt	t_{end}
3	0.2	0.8	1.2	64	6	0	250	0.03	6.0

Note that all snakes so far have been generated using $\beta = 0$. We found that this choice yields good results. If we set $\beta > 0$, the snakes become too regularized very quickly. Also the time step Δt should be made much smaller in order to avoid numerical instability of the explicit Euler method.

A favourable property of GVF snakes is that incomplete cell boundaries have little effect on the result. For example, in Figure 5.9, the two middle cells are seperated by a very vague boundary. Nonetheless, two cell centers are detected and the resulting GVF snakes are correct; they respect the vague boundary.

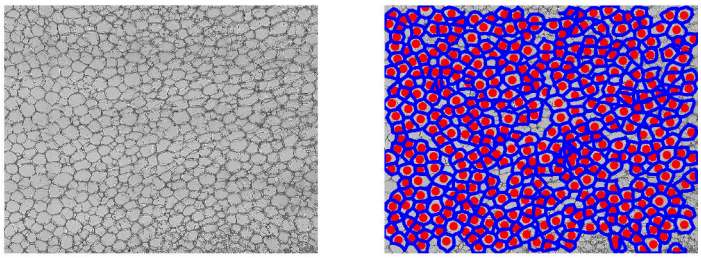

Figure 5.9: GVF snakes in a small section of image 1;US;2017;FIELD1;T19;PITH.REP1.

Table 5.4: Parameters used to generate Figure 5.9.

Init.	GVF			Snakes					
σ_b	μ	θ	σ_p	N	α	β	γ	Δt	t_{end}
4	0.2	0.8	1.2	84	10	0	250	0.01	5.0

Also note that the rightmost curve respects the vague upper boundary. A situation in which the GVF snakes perform worse is when small granules are present. As can be seen in Figure 5.10, curves are pulled towards the clusters of granules.

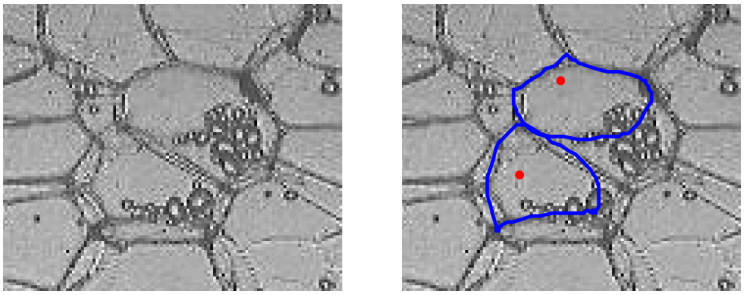

Figure 5.10: GVF snakes in a small section of image 1;US;2017;FIELD1;T9;PITH.REP2 containing clusters of granules.

Table 5.5: Parameters used to generate Figure 5.10.

Init.	GVF			Snakes					
σ_b	μ	θ	σ_p	N	α	β	γ	Δt	t_{end}
6	0.2	0.8	1.2	64	6	0	150	0.01	5.0

If there are only a few of these granules present, their influence can be reduced by tweaking the parameters slightly. In Figure 5.11 the size of the curves is increased, yielding acceptable results. Note that the granule in the leftmost cell still affects the curve.

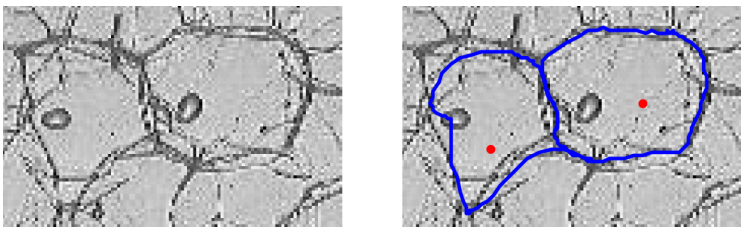

Figure 5.11: GVF snakes in a small section of image 2;US;2017;FIELD1;T14;CORTEX.REP1 containing granules.

Table 5.6: Parameters used to generate Figure 5.11.

Init.	GVF			Snakes					
σ_b	μ	θ	σ_p	N	α	β	γ	Δt	t_{end}
8	0.2	0.8	1.2	96	10	0	250	0.01	5.0

As a final remark, note that the values of σ_b are quite large in the previous simulations. This is done to only detect cell centers in the cells we are interested in.

5.2.1 Degenerate Snakes

In some cases, the snake algorithm produces faulty curves. These degenerate snakes are often squished against the image border or a cell boundary, enclosing only a very small area. Degenerate snakes are not desirable when performing statistical analysis on the output of the snake algorithm, as they can skew the data. In Figure 5.12, some degenerate snakes are shown.

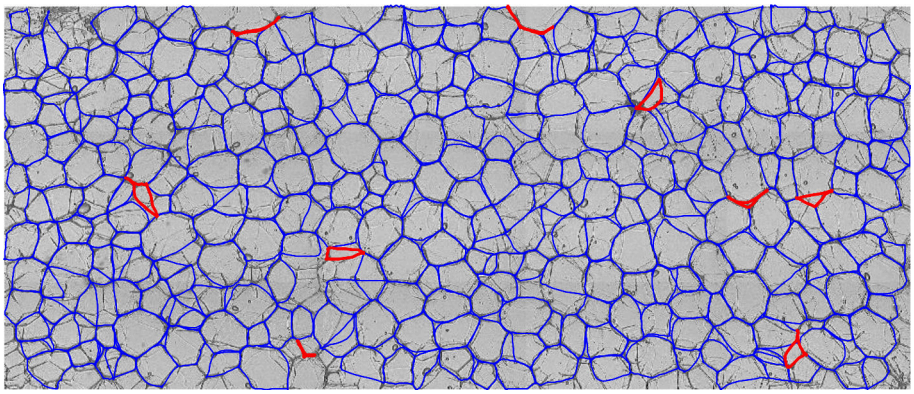

Figure 5.12: Small section of 2;US;2017;FIELD1;T14;CORTEX.REP1, containing degenerate curves that are marked red.

Fortunately, these faulty snakes can be easily detected. Define the isoperimetric ratio IPR and isoperimetric quotient IPQ of a closed curve $\boldsymbol{r}$ by

$$\mathrm{IPR}(\boldsymbol{r}) = \frac{\ell(\boldsymbol{r})^2}{A(\boldsymbol{r})}, \quad \mathrm{IPQ}(\boldsymbol{r}) = \frac{4\pi A(\boldsymbol{r})}{\ell(\boldsymbol{r})^2}, \tag{5.1}$$

where $\ell(\boldsymbol{r})$ denotes the length of $\boldsymbol{r}$, and $A(\boldsymbol{r})$ denotes the area it encloses. A circle is a curve with the smallest IPR; it is equal to 4π [3]. For this reason, the IPQ of any curve is in the range $[0, 1]$. Furthermore, the IPR and IPQ of a curve are invariant under scaling. We expect the IPR of a degenerate curve to be large, as the area it encloses is small. Or equivalently, the IPQ is expected to be small. We use this intuition to define a degenerate curve as follows:

Definition 5.1. A snake $\boldsymbol{x}$ is called *degenerate* if $\mathrm{IPQ}(\boldsymbol{x}) \leq \frac{1}{3}$.

This is also the criterium used to find the faulty snakes in Figure 5.12. To put definition 5.1 into context, consider a rectangle with sides of length 1 and a. Then the IPR of this rectangle, say R_a, is given by

$$R_a = \frac{(2a+2)^2}{a} = 4a + 8 + \frac{4}{a}.$$

If a snake $\boldsymbol{x}$ is degenerate, then we have

$$\mathrm{IPR}(\boldsymbol{x}) = \frac{4\pi}{\mathrm{IPQ}(\boldsymbol{x})} \geq 12\pi \geq 37 = 4 \cdot 7 + 9 \geq 4 \cdot 7 + 8 + \frac{4}{7} = R_7.$$

So the IPR of $\boldsymbol{x}$ is higher than that of a rectangle with sides of length 1 and 7. We therefore assume that $\boldsymbol{x}$ is not the boundary of any cell. In practice, a snake is given by a set of points. Its length and area will have to be approximated. Assume a snake $\boldsymbol{x}$ is given by N points $(x_0, y_0), \ldots, (x_{N-1}, y_{N-1})$. Then the length is calculated by summing the lengths of all line segments between to consecutive points:

$$\ell(\boldsymbol{x}) = \sum_{i=1}^{N} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}. \tag{5.2}$$

The area enclosed by $\boldsymbol{x}$ can be calculated using the the shoelace-formula[4]:

$$A(\boldsymbol{x}) = \frac{1}{2} \left| \sum_{i=1}^{N} (x_{i-1} y_i - x_i y_{i-1}) \right|. \tag{5.3}$$

In the equations above the periodicity of $\boldsymbol{x}$ is used, so $x_N = x_0$ and $y_N = y_0$.

5.3 Statistics

In this section we provide some examples of statistical tests that can be performed on the output of the snake algorithm. No conclusions are drawn from any of these tests; the aim is simply to show that a number of different geometric parameters can be extracted from the snakes. We focus on the isoperimetric quotient, which is already introduced in section 5.2.1, and the diameter of a snake. The diameter of a cell is the longest straight line that can be placed inside the cell. The orientation of the diameter in particular is given some attention.

5.3.1 Isoperimetric Quotient

Assume that for some image the snake algorithm yields K snakes. For a snake $\boldsymbol{x} \in \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_K\}$, its length $\ell(\boldsymbol{x})$ and the area it encloses $A(\boldsymbol{x})$ can be calculated using (5.2) and (5.3) respectively. By calculating the IPQ we can check wether $\boldsymbol{x}$ is degenerate by definition 5.1. If it is, discard the snake. Repeating this process for all snakes, this leaves us with $K' \leq K$ nondegenerate snakes. In Figure 5.13 the relation between length, area and IPQ of all snakes in a particular image are presented.

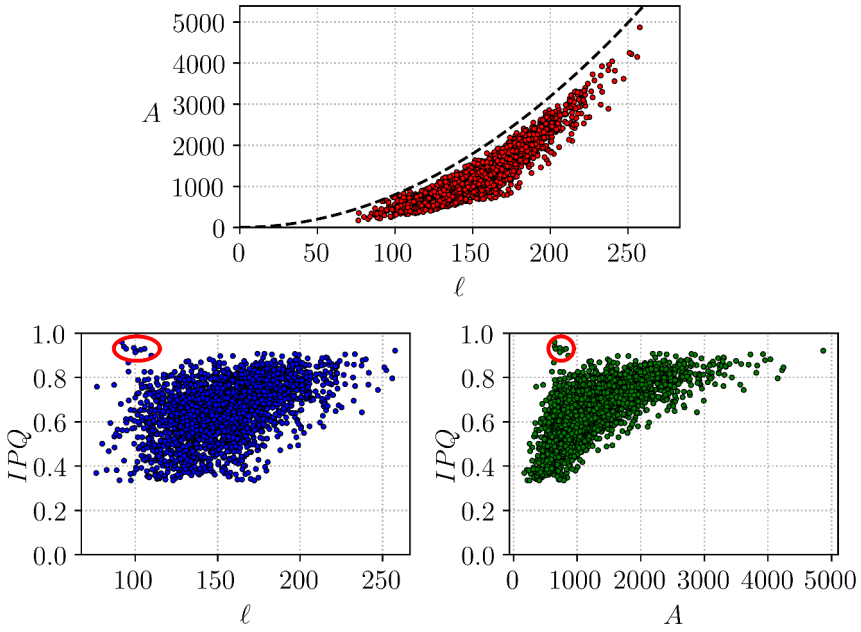

Figure 5.13: Scatterplots of area against length (upper plot), isoperimetric quotient against length (bottom left) and isoperimetric quotient against area (bottom right), based on the results of the snake algorithm applied to image 451;US;2017;FIELD46;T17;PITH.REP1. Length and area are given in terms of pixels. The parameters used are given in table 5.7.

Table 5.7: Parameters used to generate Figure 5.13.

Init.	GVF			Snakes					
σ_b	μ	θ	σ_p	N	α	β	γ	Δt	t_{end}
1.5	0.2	0.8	1.2	64	4	0	250	0.04	6.0

Note that there are no snakes with an IPQ of less than 1/3, which is the result of discarding degenerate snakes. The dashed line in the upper plot represents the parabola $A = \ell^2/4\pi$. By the isoperimetric inequality, no points can be located above this line. The points inside the two red ellipses correspond to snakes with high IPQ's. In this case, these snakes are located in a region of the image where no cells are present. Here, the effect of the GVF field is minimal, so the snakes turn into near perfect circles. These type of snakes are characterized by a high IPQ and small area and length. Nevertheless, they are hard to detect afterwards, as there may be some actual

cells with these characteristics. The solution to this problem is to prohibit cell centers from being detected in regions without cells.

5.3.2 Diameter

Another interesting statistic of the cells is the diameter, which has a length and (almost always) an orientation. The orientation in particular can yield insight in the way cells are placed in the analyzed cross section. In Figure 5.14 the diameters of the cells are plotted. Note that in the lower right corner the general orientation seems to be nearly vertical, whereas in the upper right corner the it is more or less horizontal.

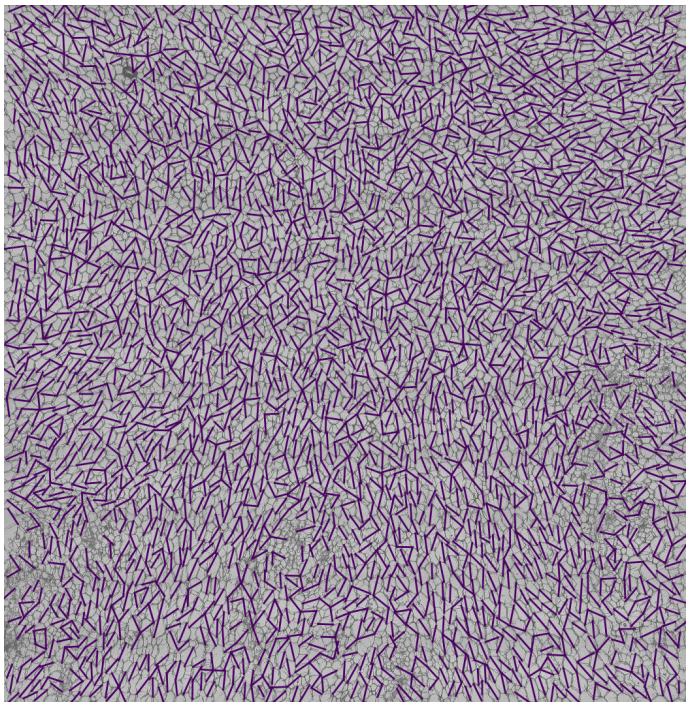

Figure 5.14: Diameters of GVF snakes in image 1;US;2017;FIELD1;T14;PITH.REP2. The average length of all diameters is 47.0 pixels.

This leads us to the matter of finding these diameters. The initial idea was to fit an ellipse to every cell, and let the major axis of this ellipse be the diameter. However, the fitted ellipses were often quite bad, producing even worse diameters. Therefore, this method wass discarded. The alternative method is much simpler. Given a snake $\boldsymbol{x}$ as a list of N points $(x_0, y_0), \ldots, (x_{N-1}, y_{N-1})$, the diameter d is defined to be the length of the longest edge connecting two of these points:

$$d = \max_{i \neq j} \|(x_i, y_i) - (x_j, y_j)\|. \tag{5.4}$$

Which implies that $N(N-1)$ lengths have to be checked for every cell. However, assuming the cell has a nearly convex shape, only points on the opposite side of the cell have to be checked. For example, if we start checking in point (x_0, y_0), we need not calculate the distance to the points $(x_1, y_1), (x_{N-1}, y_{N-1})$ etc, for it is assumed these points are close to (x_0, y_0). Only $N/2$ points on the opposite side of (x_0, y_0) are checked. To avoid calculating the length of lots of edges multiple times, only half the points are checked. This reduces the number of lengths that have to be checked to $N^2/4$. In algorithm 5.1 the procedure is given in pseudocode.

Algorithm 5.1 Find diameter

Require: Snake $\boldsymbol{x}$ consisting of N points $(x_0, y_0), \ldots, (x_{N-1}, y_{N-1})$
  $d \leftarrow 0$ ▷ Diameter length
  $e \leftarrow \{0, 0\}$ ▷ Edge corresponding to diameter
  for $i = 0, \ldots, N/2$ **do** ▷ Check half of all points
    for $j = i + N/4, \ldots, i + 3 \cdot N/4$ **do** ▷ Half of the points opposite to point i
      $j \leftarrow j \mod N$ ▷ Use periodicity of snake
      $d_{ij} \leftarrow \|(x_i, y_i) - (x_j, y_j)\|$
      if $d_{ij} > d$ **then**
        $d \leftarrow d_{ij}$
        $e \leftarrow \{i, j\}$
      end if
    end for
  end for
  return d, e

The orientation of a diameter can be expressed in terms of its angle with respect to the x-axis. Note that in image processing the x-axis points downwards, and the y-axis points to the right. We define the orientation θ of a diameter d connecting two points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ to be the angle obtained from turning anti-clockwise from the positive x-direction to the segment. This is illustrated in Figure 5.15. Since d is an undirected line segment, we can let $\theta \in [0, \pi)$. Given that $y_j \geq y_i$, θ can then be calculated as follows,

$$\theta = \arccos \frac{x_j - x_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|}. \tag{5.5}$$

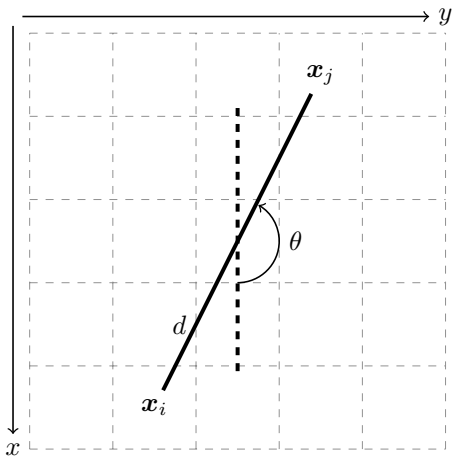

Figure 5.15: Orientation of diameter between points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$.

This allows us to examine the orientation of the diameters of all snakes in an image. In Figure 5.16 this is done using the diameters of Figure 5.14. The data is presented as a histogram on a circle. The size of a bar in a given direction corresponds to the number of diameters that have that direction. Note that the data is π-periodic; bars on opposite sides of the circle have the same size. Because $\theta \in [0, \pi)$, the left half of the circle can be left out. We include it for the sake of presentation.

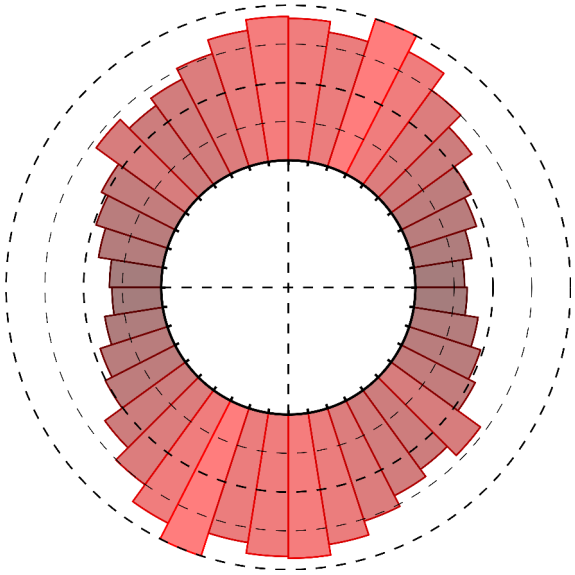

Figure 5.16: Histogram showing the orientations of the diameters of GVF snakes in image 1;US;2017;FIELD1;T14;PITH.REP2.

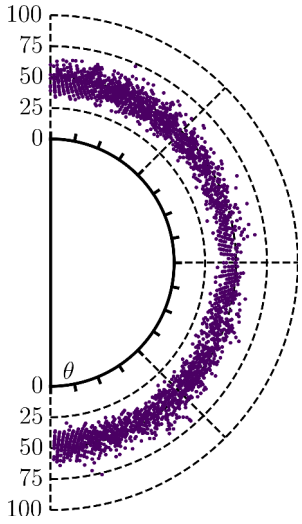

Figure 5.17: Scatterplot showing the length of a diameter against its orientation.

It can be seen that the diameters in Figure 5.14 have a 'vertical bias'; more diameters are orientated rougly vertical than horizontal. However, there is not clear correlation between the orientation and length of a diameter, as can be seen in Figure 5.17. We found that in some images, different areas have a different median direction. In Figure 5.18, two different regions are highlighted. The diameters in the green region are orientated vertically, whereas the diameters in the blue region are oriented more or less horizontally.

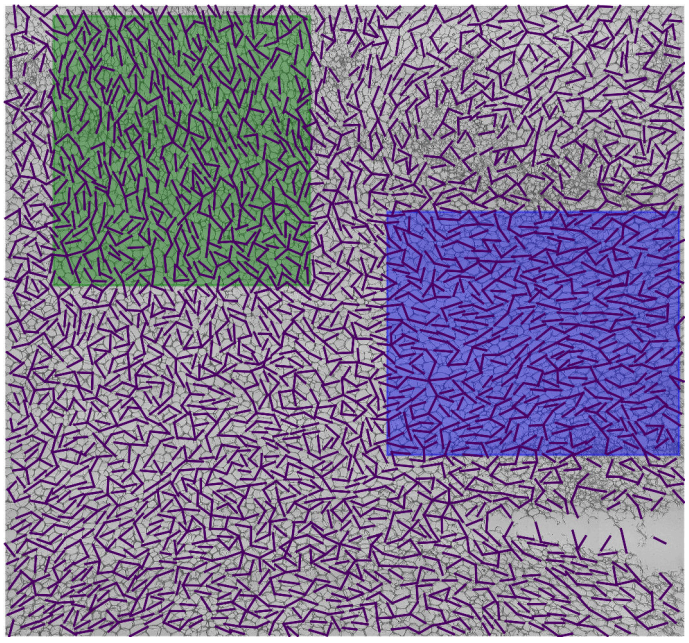

Figure 5.18: Diameters of GVF snakes in image 1;US;2017;FIELD1;T14;PITH.REP1.

These claims are supported by Figures 5.19 and 5.20, where histograms showing the orientation of diameters in both regions are presented. In the green region, a peak at about $\theta = \pi/6$ is present. This means that a lot of diameters have roughly this orientation. In the blue region we can see peaks around $\theta = \pi/2$, meaning that a lot of diameters are roughly horizontal.

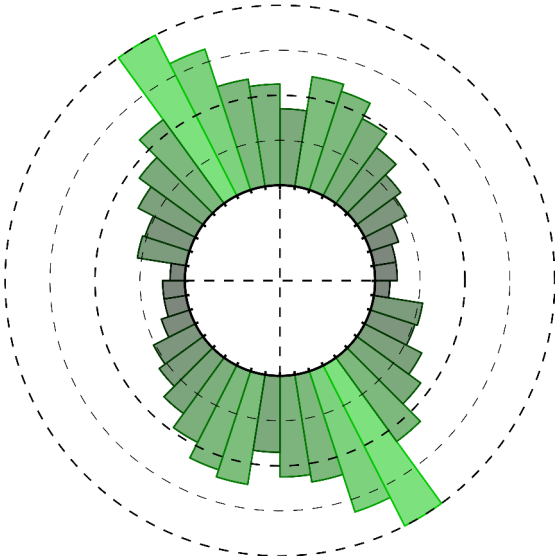

Figure 5.19: Orientations of diameters in green part of Figure 5.18.

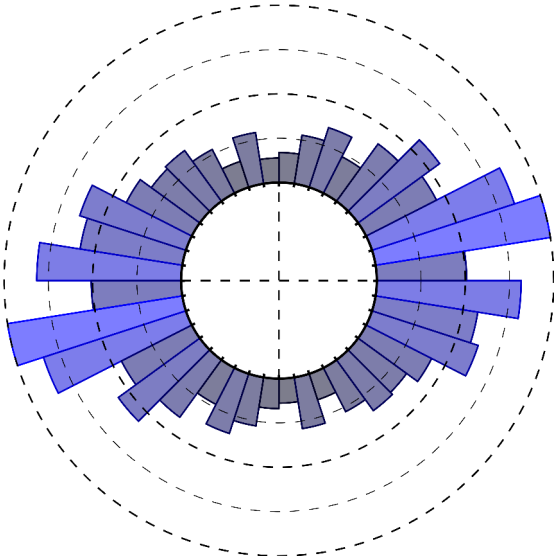

Figure 5.20: Orientations of diameters in blue part of Figure 5.18.

We expect diameters with peak orientation to be slightly longer than other diameters in a given region. This can be seen in Figures 5.21 and 5.22, where the lengths corresponding to peak orientations are indeed slightly above average. The differences are very small however.

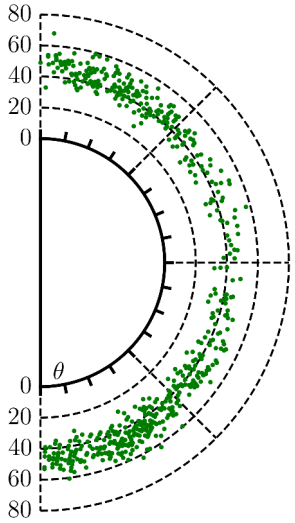

Figure 5.21: Scatterplot showing length of a diameter in green part of Figure 5.18 against its orientation.

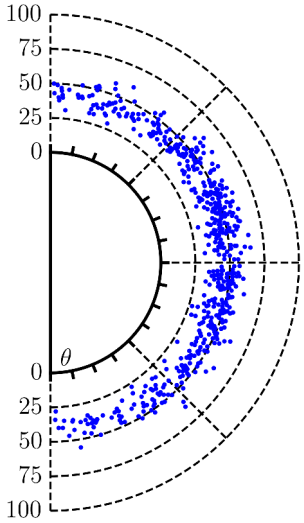

Figure 5.22: Scatterplot showing length of a diameter in blue part of Figure 5.18 against its orientation.

Furthermore, if the reader slightly squints at Figures 5.14 and 5.18, some kind of 'flow' can be seen. While it is beyond the scope of this report, further research can be done in creating an algorithm to find these flows.

5.4 Clustering

The snake algorithm was applied to about 1200 images[7], using the parameters in table 5.7. Note that these parameters might not have been the best choice for certain images. However, finding a good set of parameters for each of the 1200 images by hand would have taken too much time. Based on these simulations, we wish to seperate the images into a number of different classes. This is called clustering. The set of all images is separated into two disjoint sets, one containing microscopic images of the pith, and the other containing microscopic images of the cortex. For every image, the length and area is calculated for every GVF snake, and degenerate curves are discarded. Then the mean of the length and area of all non-degenerate curves can be calculated. Say that the output of the snake algorithm applied to an image I is a set of K' non-degenerate curves $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{K'}\}$. Then the mean of the lenghts and areas are given by respectively

$$\mu_\ell = \frac{1}{K'} \sum_{k=1}^{K'} \ell(\boldsymbol{x}_k), \quad \mu_A = \frac{1}{K'} \sum_{k=1}^{K'} A(\boldsymbol{x}_k). \tag{5.6}$$

Every image now produces a vector (μ_ℓ, μ_A). These vectors are shown in the left plot of Figure 5.23. There is a clear linear relation between μ_A and μ_ℓ. The same thing can be done for the standard deviations. They are calculated as follows:

$$\sigma_\ell = \sqrt{\frac{1}{K'} \sum_{k=1}^{K'} (\ell(\boldsymbol{x}_k) - \mu_\ell)^2}, \quad \sigma_A = \sqrt{\frac{1}{K'} \sum_{k=1}^{K'} (A(\boldsymbol{x}_k) - \mu_A)^2}. \tag{5.7}$$

The vectors (σ_ℓ, σ_A) are shown in the right plot of Figure 5.23. Note that there is again a clear correlation. Based on these means and standard deviations, the images can be clustered.

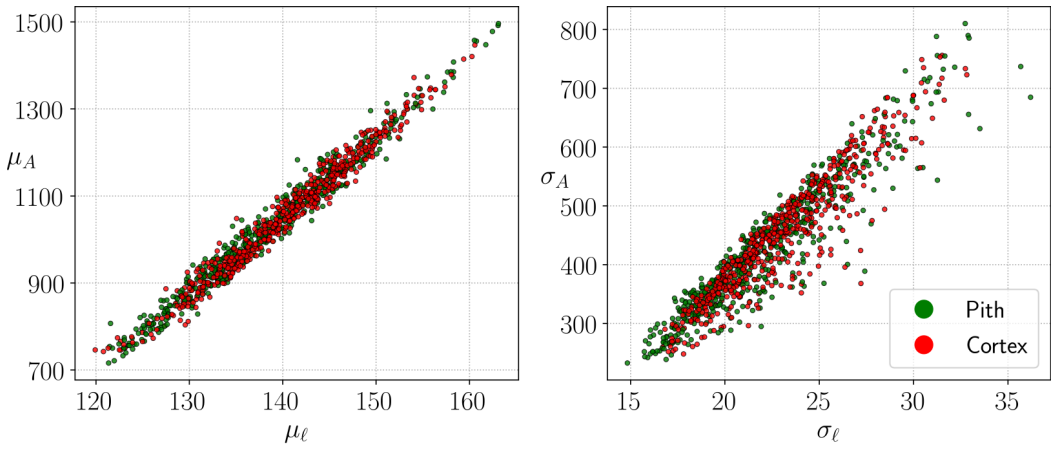

Figure 5.23: Scatterplots showing the points (μ_ℓ, μ_A) and (σ_ℓ, σ_A) for every image, seperated into pith and cortex.

The numbers μ_ℓ, μ_A, σ_ℓ and σ_A can only be compared with each other if the magnification level of the microscope used to make the corresponding images is the same for both images. It is assumed that this is indeed the case. We assume that two potatoes of the same species have a somewhat similar cell structure. In this case we also expect that the statistics $\mu_\ell, \mu_A, \sigma_\ell$ and σ_A are close to each other. If it is known that all potatoes belong to one of k species, a clustering algorithm can be applied to the set of 4-dimensional vectors $(\mu_\ell, \mu_A, \sigma_\ell, \sigma_A)$. Every cluster then represents a species. The k-means method [5] is well known clustering algorithm that aims to partition n vectors into k sets, such that the within-cluster sum of squares is minimized. In Figures 5.24 and 5.25 the pith and cortex images are separated into 3 clusters, using the k-means method. The initial positions of the centroids are chosen at random. Note that the data points are actually

[7] The total computation time was roughly 15 hours.

elements of 4-dimensional space. Both figures shows two projections of the data, onto the (μ_ℓ, μ_A) plane and (σ_ℓ, σ_A) plane. This is the reason why some clusters seem to overlap.

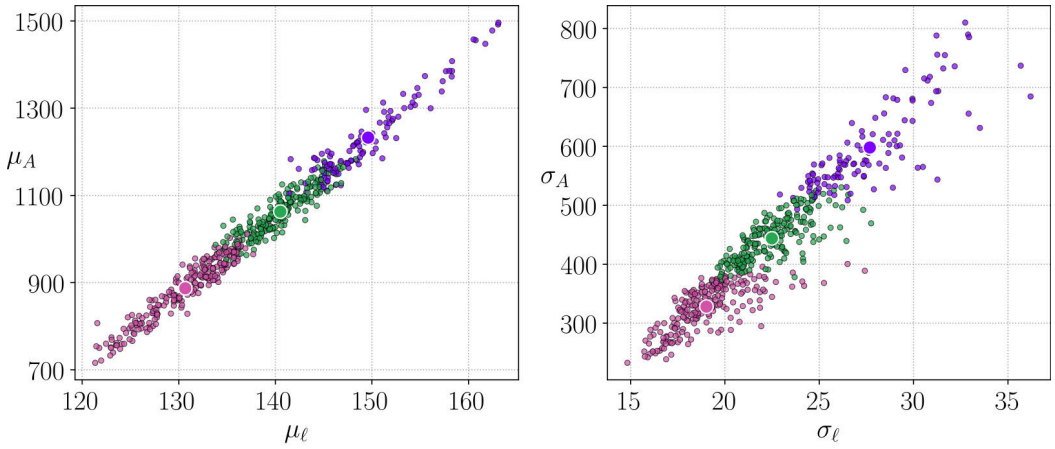

Figure 5.24: Pith images separated into 3 clusters. The big dots represent the centroids of a cluster.

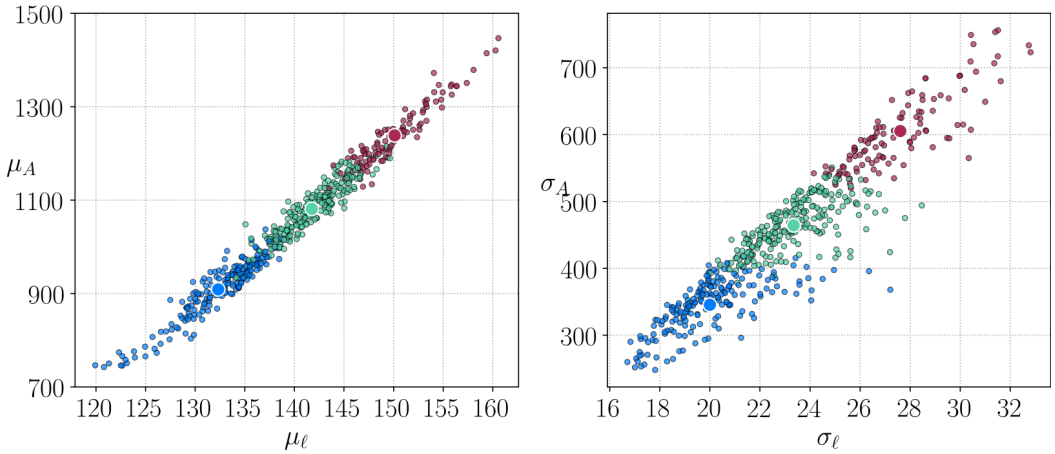

Figure 5.25: Cortex images separated into 3 clusters. The big dots represent the centroids of a cluster.

The clusters produced by the k-means method seem to be rather arbitrary. This might be due to the fact that a lot of information about the curves is lost when calculating means and standard deviations. This problem is avoided by using spectral clustering [6], which makes use of the eigenvalues of a similarity matrix. Let $S_1, \ldots, S_P$ be datapoints. In our case the datapoints are sets that contain lengths and areas of all GVF curves in one image, i.e.

$$S_i = \left\{ \begin{pmatrix} \ell_1^i \\ A_1^i \end{pmatrix}, \ldots, \begin{pmatrix} \ell_{K_i}^i \\ A_{K_i}^i \end{pmatrix} \right\}, \tag{5.8}$$

where ℓ_k^i and A_k^i denote the length and area of curve k in image i, and K_i is the total number of non-degenerate GVF curves in image i. Note that the GVF curves are not inherently ordered, but they can be numbered from 1 to K_i. The similarity matrix A is then a symmetric $P \times P$ matrix such that $A_{ij} \geq 0$ represents a measure of similarity between S_i and S_j. We will discuss the similarity function later. Given A, its laplacian matrix L is given by

$$L = I - D^{-1/2} A D^{-1/2}, \tag{5.9}$$

where D is a diagonal matrix with

$$D_{ii} = \sum_{j=1}^{P} A_{ij}. \tag{5.10}$$

Computing $D^{-1/2}$ can be done by raising every diagonal element of D to the power $-1/2$. The eigenvector v corresponding to the second smallest eigenvalue λ_2 of L is used to partition the data into a number of clusters. For example, the k-means method can be applied to v to partition v into k clusters. Every entry v_i corresponds to a datapoint S_i, so the data has been clustered implicitly. In essence, spectral clustering reduces the P datapoints to a P-dimensional vector, on which conventional clustering techniques can be applied.

The similarity between two sets S_i and S_j is a number that respresents how similar the two sets are, let $\psi(S_i, S_j)$ be that number. ψ should satisfy three properties:

$$\psi(S_i, S_j) = \psi(S_j, S_i), \quad \psi(S_i, S_j) \geq 0, \quad \text{and} \quad \psi(S_i, S_j) = 0 \iff S_i = S_j. \tag{5.11}$$

We propose two different similarity functions,

$$\psi_1(S_i, S_j) = \frac{1}{|S_i| + |S_j|} \left(\sum_{a \in S_i} \min_{b \in S_j} \|a - b\| + \sum_{b \in S_j} \min_{a \in S_i} \|a - b\| \right), \tag{5.12a}$$

$$\psi_2(S_i, S_j) = \frac{1}{2|S_i|} \sum_{a \in S_i} \min_{b \in S_j} \|a - b\| + \frac{1}{2|S_j|} \sum_{b \in S_j} \min_{a \in S_i} \|a - b\|. \tag{5.12b}$$

Where $\|\cdot\|$ can be any norm on $\mathbb{R}^2$. For simplicity, we let $\|\cdot\|$ be the Euclidian norm. Both functions come from the same principle. If S_i is similar to S_j, then for every point $a \in S_i$, we expect there to be a point $b \in S_j$ close to a. In this case, all distances $\min_{b \in S_j} \|a - b\|$ will be small, so ψ_1 and ψ_2 will both be small. However, if S_i is not similar to S_j, then at least a number of these distances will be large. It can be immediatly seen that ψ_1 and ψ_2 satisfy the first two properties of (5.11). The third property is also satisfied. Assume $S_i = S_j$, then for every point, the minimum distance to a point in the other set is the distance to itself. Now assume $S_i \neq S_j$, and WLOG[8] assume there is a point $a \in S_i$ such that $a \notin S_j$. Then $\min_{b \in S_j} \|a - b\| \neq 0$, so $\psi_1(S_i, S_j), \psi_2(S_i, S_j) \neq 0$.

The difference between the two similarity functions lies in the way they treat sets of different sizes. If $|S_i| = |S_j|$, then $\psi_1(S_i, S_j) = \psi_2(S_i, S_j)$. Note that ψ_2 gives both sets the same 'weight'. Even if S_i contains only one element a, the minimum distance from a to S_j is just as important as the average minimum distance of all elements of S_j to a. On the other hand, ψ_1 gives the larger set more weight.

Computing ψ_1 or ψ_2 is very computationally intensive. If all $P = 1200$ datasets are taken into account, we would need to calculate $P(P-1)/2 \approx 720000$ similarities. By extrapolation, we found that this would take roughly 17 hours on a quad-core Linux machine. We test the method on the dataset 382, containing 11 cortex images. We separate this dataset into 2 clusters, using both ψ_1 and ψ_2 as similarity functions. The resulting P-dimensional eigenvector v is separated by the sign of its entries. If $v_i > 0$, then image i belongs to cluster 1. Otherwise it belongs to cluster 0. See Table 5.8 for the results.

[8] Without Loss Of Generality; it could be that $S_i \subseteq S_j$, in this case such a point a would be in S_j.

Table 5.8: Results of spectral clustering method using different similarity functions.

Image name	Cluster (ψ_1)	Cluster (ψ_2)
382;US;2017;FIELD39;T4;CORTEX.REP2	1	1
382;US;2017;FIELD39;T19;CORTEX.REP2	0	1
382;US;2017;FIELD39;T9;CORTEX.REP1	1	1
382;US;2017;FIELD39;T12;CORTEX.REP2	1	0
382;US;2017;FIELD39;T20;CORTEX.REP1	0	0
382;US;2017;FIELD39;T4;CORTEX.REP1	0	1
382;US;2017;FIELD39;T12;CORTEX.REP1	0	0
382;US;2017;FIELD39;T19;CORTEX.REP1	1	0
382;US;2017;FIELD39;T9;CORTEX.REP2	0	0
382;US;2017;FIELD39;T20;CORTEX.REP2	0	1
382;US;2017;FIELD39;T19;CORTEX.REP3	1	1

Note that both functions yield rather different results. Computing time for both functions is about 5 seconds, which is quite bad for such a small collection of images. We propose a new similarity function based on the empirical cumulative distribution function. Split a dataset S_i as in (5.8) into two sets ℓ^i and A^i:

$$\ell^i = \left\{\ell^i_1, \ldots, \ell^i_{K_i}\right\}, \quad A^i = \left\{A^i_1, \ldots, A^i_{K_i}\right\}. \tag{5.13}$$

Let F^i_ℓ and F^i_A be the empirical cumulative distribution functions (ecdf) of the sets ℓ^i and A^i respectively. Define the functions

$$\Delta^{ij}_\ell(t) = \left|F^i_\ell(t) - F^j_\ell(t)\right|, \quad \Delta^{ij}_A(t) = \left|F^i_A(t) - F^j_A(t)\right| \tag{5.14}$$

to be the absolute difference between the two empirical distributions corresponding to the length and area. If S_i and S_j are similar, then the functions Δ^{ij}_ℓ and Δ^{ij}_A will be close to zero. Define

$$I^{ij}_\ell = \int_0^\infty \Delta^{ij}_\ell(t)\,\mathrm{d}t, \quad I^{ij}_A = \int_0^\infty \Delta^{ij}_A(t)\,\mathrm{d}t. \tag{5.15}$$

Finally, the similarity function ψ_3 is defined as

$$\psi_3(S_i, S_j) = \left\|\left(4\pi I^{ij}_\ell, I^{ij}_A\right)\right\|, \tag{5.16}$$

where $\|\cdot\|$ again denotes the Euclidian norm. The integral I^{ij}_ℓ is multiplied by 4π in order to make the effect of the length stronger. As one can see in Figure 5.23, the area enclosed by a snake is significantly larger than its length. As a result, the interval on which Δ^{ij}_A is non-zero is much bigger than that of Δ^{ij}_ℓ, which in turn implies that I^{ij}_A will be much bigger than I^{ij}_ℓ. Multiplying by 4π is done to counteract this effect. One immediate advantage of ψ_3 over ψ_1 and ψ_2 is the reduced computation time. Applying it to the dataset 382 only takes 0.03 seconds. By extrapolation, applying spectral clustering using ψ_3 to all images would take only 6 minutes. See Table 5.9 for the resulting partition.

Table 5.9: Results of spectral clustering method using ψ_3.

Image name	Cluster (ψ_3)
382;US;2017;FIELD39;T4;CORTEX.REP2	0
382;US;2017;FIELD39;T19;CORTEX.REP2	1
382;US;2017;FIELD39;T9;CORTEX.REP1	0
382;US;2017;FIELD39;T12;CORTEX.REP2	1
382;US;2017;FIELD39;T20;CORTEX.REP1	0
382;US;2017;FIELD39;T4;CORTEX.REP1	0
382;US;2017;FIELD39;T12;CORTEX.REP1	0
382;US;2017;FIELD39;T19;CORTEX.REP1	0
382;US;2017;FIELD39;T9;CORTEX.REP2	1
382;US;2017;FIELD39;T20;CORTEX.REP2	1
382;US;2017;FIELD39;T19;CORTEX.REP3	0

Since we have no information on the actual species a potato belongs to, results from the clustering algorithms can't be checked. While it is rather dissatisfying, the aim of this section is to show that results produced by the snake algorithm *can* be used for purposes like clustering.

6 Summary

The aim of this project was to develop a tool to analyze microscopic images of cells. The desired output of this tool was a set of curves that represent the cell boundaries. The theory of active contours, or snakes, is used to achieve this. In particular the gradient vector flow (GVF) method was utilised. GVF snakes possess a number of favourable characteristics. Most notably, the force field can capture a snake from a long range. Thus, initial curves can be quite large. Furthermore, because of its smoothing properties, GVF snakes deal well with missing or vague cell boundaries. A drawback of GVF snakes is that they can be affected by small granules inside the cells. These granules distort the GVF force field, resulting in snakes getting pulled towards them. The snake algorithm relies on a number of parameters. Although a general set of parameters can be utilised for all images within the provided collection, for optimal results these parameters could tuned for every image individually. However, the quality of the resulting curves will depend on the quality of the image itself.

A number of auxiliary algorithms have also been developed. Most important is the cell center detection method. This method is vital to the succes of the snake algorithm, as the locations of cells have to be known before initial curves can be placed. A preprocessing algorithm was implemented in order to remove some of the noise in the image. Lastly, we noticed that some images contained empty regions. To prohibit cell centers from being placed in these regions, a cropping algorithm was implemented.

The results from the snake algorithm can be used to perform statistical analyses. For example, the length of the curves and the area they enclose can vary from image to image. The diameters can be used to determine the orientation of cells in (parts of) the image. The statistics can be used to cluster images into sets, representing different species. Two different clustering methods have been applied to sets of lengths and areas, to show this is indeed possible.

Further work can be done in using a variable number of points to represents the curves. At present, every curve consists of the same (fixed) amount of points. However, as some cells are larger than others, we would prefer the number of points to be variable. The amount of points can be determined beforehand for every curve, as a function of the GVF parameters and for example the area of the cell (if it can be calculated). Another possibility is to add points during the algorithm. For example, if the internal forces are keeping a curve from moving in the direction of the GVF field, another point can be added to relax the internal force.

Bibliography

[1] M. Kass, A. Witkin, D. Terzopoulos. *Snakes: Active Contour Models*. Int. J. Computer Vision, 1(4):321-331, 1987.

[2] J.L. Prince, C. Xu. *A new External Force Model for Snakes*. Image and Multidimensional Signal Processing Workshop, p. 30-31, 1996.

[3] R. Osserman. *The Isoperimetric Inequality*. Bulletin of the Am. Math. Soc., Vol. 84, No. 6, 1978.

[4] Y. Lee, W. Lim. *Shoelace Formula: Connecting the Area of a Polygon with Vector Cross Product*. Math. Teacher, 110(8): 631-636, 2017.

[5] J.A. Hartigan. *Clustering Algorithms*. 1975.

[6] A.Y. Ng, M.I. Jordan, Y. Weiss. *On Spectral Clustering: Analysis and an Algorithm*. Adv. in Neural Inf. Proc. Systems, p. 849-856, 2002.

A | Python Code

The most important programs are presented. All scripts are written in Python 2.7, and make use of open-source libraries such as numpy, matplotlib, scipy and skimage.

A.1 snake.py

This script contains the snake algorithm. Given a set of parameters and an image, the output is a list `GVFcurves` containing all GVF curves. A curve is given as an $N \times 2$ numpy array, where the first column contains all x-coordinates and the second column contains all y-coordinates.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.sparse as sp
import scipy.sparse.linalg as spla
from scipy import ndimage
from skimage import color, io, transform
from skimage.feature import peak_local_max

# ============================== Parameters ==============================
num_points = 64          # should be multiple of 4
alpha = 6                # 2nd order coefficient
beta = 0                 # 4th order coefficient
gamma = 250              # force field strength
dt = 0.03                # time step
t_end = 6.0              # simulation end time

theta = 0.8              # GVF preprocessing parameters
sigma = 1.2
mu = 0.2                 # GVF parameter

resize = False
size = 2400              # rescale if image is high-res

b_sig   = 3              # cell detection parameters
min_sig = 5
max_sig = 20
# ============================== Image choice ==============================
impath = '/path/to/image.jpg'
im = color.rgb2gray(io.imread(impath))
imshape_x, imshape_y = im.shape
if resize:
    new_x, new_y = size, int(float(imshape_y)/imshape_x*size)
    im = transform.resize(im, (new_x, new_y))           # rescale

# ======================= Approximate Number of Cells ====================
num_lines = 8              # number of rows and columns used to compute S(I)
im_gauss = ndimage.gaussian_filter(im, sigma=b_sig)
avg_x_cells, avg_y_cells = 0, 0
for i in range(num_lines):
    n = i*imshape_x/num_lines
    m = i*imshape_y/num_lines
    line_x = im_gauss[n,:]
    line_y = im_gauss[:,m]
    line_x_mean = line_x.mean()
    line_y_mean = line_y.mean()
    boundary_x = (line_x<line_x_mean)
    boundary_y = (line_y<line_y_mean)
    switch_x = abs(np.diff(boundary_x))
```

```
    switch_y = abs(np.diff(boundary_y))
    count_x = switch_x.sum()*0.5
    count_y= switch_y.sum()*0.5
    avg_x_cells += count_x/num_lines
    avg_y_cells += count_y/num_lines
num_cells_scan = avg_x_cells*avg_y_cells

# ======================== Find Center Coordinates ======================
resolution = 0.5
sig_range = np.array([min_sig+resolution*k for k in range(int((max_sig-min_sig)/
    resolution)+1)])
diff = np.infty
coordinates = np.array([])

for i in range(len(sig_range)):
    sigma_i = sig_range[i]
    im_gauss = ndimage.gaussian_filter(im, sigma=sigma_i)
    new_coordinates = peak_local_max(im_gauss)
    num_peaks = new_coordinates.shape[0]
    new_diff = abs(num_cells_scan - num_peaks)
    if new_diff > diff:
        break
    diff = new_diff
    coordinates = new_coordinates

sig_opt = sig_range[i-1]
center_coords = coordinates

# ====================== Place initial Curves ============================
N = num_points/4
r = N + 1
dxrange = range(-r+1, r,   2)
dyrange = range(-r+3, r-2, 2)

initial_curves = []
for point in center_coords:
    i, j = point[0], point[1]
    curve = []
    dy = -r+1
    for dx in dxrange:
        curve.append(np.array([i+dx, j+dy]))
    dx = r-1
    for dy in dyrange:
        curve.append(np.array([i+dx, j+dy]))
    dy = r-1
    for dx in dxrange[-1::-1]:
        curve.append(np.array([i+dx, j+dy]))
    dx = -r+1
    for dy in dyrange[-1::-1]:
        curve.append(np.array([i+dx, j+dy]))
    initial_curves.append(np.array(curve))

# ======================== GVF preprocessing ============================
mean = np.mean(im)
newim = np.zeros(im.shape)
for i in range(im.shape[0]):
    for j in range(im.shape[1]):
        if im[i,j] > theta*mean:
            newim[i,j] = 1

f = -ndimage.gaussian_filter(newim, sigma=sigma)

# ====================== GVF field calculation ==========================
def C(n):
    mat = np.zeros((n,n))
    mat[0,0] = -1
    mat[n-1,n-1] = 1
    mat += np.diag([1]+[0.5 for k in range(n-2)], 1)
    mat += np.diag([-0.5 for k in range(n-2)]+[-1], -1)
    return mat
```

```
Cx, CyT = C(imshape_x),  C(imshape_y).transpose()
Fx, Fy = np.dot(Cx, f), np.dot(f, CyT)
fx, fy = Fx.transpose().ravel(), Fy.transpose().ravel()
gdiag = fx**2 + fy**2
G = sp.diags([gdiag], [0])
Ix = sp.eye(imshape_x)
Iy = sp.eye(imshape_y)
Dx = sp.diags([-np.ones(imshape_x-1), np.ones(imshape_x-1)], [0,1], shape=(
    imshape_x-1,imshape_x))
Dy = sp.diags([-np.ones(imshape_y-1), np.ones(imshape_y-1)], [0,1], shape=(
    imshape_y-1,imshape_y))
Lxx = -Dx.transpose().dot(Dx)
Lyy = -Dy.transpose().dot(Dy)
L = sp.kron(Iy, Lxx, format='csr') + sp.kron(Lyy, Ix, format='csr')
A = mu*L - G
bx = -gdiag*fx
by = -gdiag*fy

u = spla.minres(A, bx, tol=1e-5)[0]
v = spla.minres(A, by, tol=1e-5)[0]

u = u.reshape((-1,imshape_x)).transpose()
v = v.reshape((-1,imshape_x)).transpose()
# ========================= GVF functions ============================
push = 0.1                  # strength of force pushing back to image
def U(X, Y):
    Xr, Yr = np.round(X).astype(int), np.round(Y).astype(int)
    ulist = np.zeros(4*N)
    for i in range(4*N):
        if Xr[i]<0:
            ulist[i] =  push*abs(Xr[i])
        elif Xr[i]>=imshape_x:
            ulist[i] = -push*abs(Xr[i]-imshape_x-1)
        elif Yr[i]<0 or Yr[i]>=imshape_y:
            ulist[i] = 0
        else:
            ulist[i] = u[Xr[i], Yr[i]]
    return ulist

def V(X, Y):
    Xr, Yr = np.round(X).astype(int), np.round(Y).astype(int)
    vlist = np.zeros(4*N)
    for i in range(4*N):
        if Yr[i]<0:
            vlist[i] =  push*abs(Yr[i])
        elif Yr[i]>=imshape_y:
            vlist[i] = -push*abs(Yr[i]-imshape_y-1)
        elif Xr[i]<0 or Xr[i]>=imshape_x:
            vlist[i]=0
        else:
            vlist[i] = v[Xr[i], Yr[i]]
    return vlist

# ========================= Solving for GVF curves =======================
num_it = int(t_end/dt)
GVFcurves = []
num_curves = len(initial_curves)

def create_D2(size):
    diag = [-2 for k in range(size)]
    subdiag = [1 for k in range(size-1)]
    mat = np.diag(subdiag,-1) + np.diag(diag,0) + np.diag(subdiag,1)
    mat[(0,-1),(-1,0)] = 1
    return mat

A = create_D2(4*N)
B = A.dot(A)
M = alpha*A - beta*B
```

```
for curve in initial_curves:
    X, Y = curve[:,0], curve[:,1]
    it = 0
    while it<num_it:
        X = X + dt*(M.dot(X) + gamma*U(X,Y))
        Y = Y + dt*(M.dot(Y) + gamma*V(X,Y))
        it += 1
    X = np.round(X)
    Y = np.round(Y)
    newcurve = np.array([X,Y]).transpose()
    GVFcurves.append(newcurve)
```

A.2 cropping.py

This script contains a function `void` that determines the locations of voids as defined in section 4.3. The output is a binary image of the same dimensions as the input image. The most important parameter is `size`, as it determines the size of the downscaled image. The larger this number, the more accurate the voids become.

```
import numpy as np
from scipy import ndimage
from skimage import color, io, transform

# ======================== Parameters ====================
size = 200          # size of downsized image
tau = 0.1           # threshold for small gradient
p = 0.01            # area percentage
fsize = 5           # minimum filter size
theta = 0.8         # preprocessing theta
sigma = 0.5         # preprocessing sigma

# ========================= Image ========================
def C(n):
    mat = np.zeros((n,n))
    mat[0,0] = -1
    mat[n-1,n-1] = 1
    mat += np.diag([1]+[0.5 for k in range(n-2)], 1)
    mat += np.diag([-0.5 for k in range(n-2)]+[-1], -1)
    return mat

def void(orig_im, size=200, tau=0.1, p=0.01, fsize=5, theta=0.8, sigma=0.5):
    imshape_x, imshape_y = orig_im.shape
    im = ndimage.filters.minimum_filter(orig_im, size=fsize)
    eta = int(float(imshape_x)/size)
    new_x, new_y = int(1./eta*imshape_x), int(1./eta*imshape_y)
    im_small = transform.resize(im, (new_x, new_y))

    # ====================== Preprocessing ===================
    mean = np.mean(im_small)
    newim = np.zeros(im_small.shape)
    for i in range(new_x):
        for j in range(new_y):
            if im_small[i,j] > theta*mean:
                newim[i,j] = 1
    f = -ndimage.gaussian_filter(newim, sigma=sigma)

    # ======================== Gradient ======================
    Cx, CyT = C(new_x),  C(new_y).transpose()
    Fx, Fy = np.dot(Cx, f), np.dot(f, CyT)
    G = np.sqrt(Fx**2+Fy**2)
    B = (G<tau)                          # binary image

    # ========================= Labels =======================
    labelled_arr, num_reg = ndimage.measurements.label(B, structure=np.array
        ([[0,1,0],[1,1,1],[0,1,0]]))
    void = np.zeros((new_x, new_y))
    for num in range(1, num_reg+1):
        if (labelled_arr==num).sum() > p*new_x*new_y:
            void += (labelled_arr==num)
```

```
    big_void = np.kron(void, np.ones((eta, eta)))
    big_void_corrected = np.ceil(transform.resize(big_void, im.shape)).astype(int)
        # make shape equal to image
    return big_void_corrected

if __name__ == "__main__":
    impath = '/path/to/image.jpg'
    orig_im = color.rgb2gray(io.imread(impath))
    voids = void(orig_im)
```

A.3 analysis.py

We present some functions that can be used to calculate different statistics.

```
import numpy as np
from math import sqrt

def length(X, Y):
    """Compute length of curve."""
    l = 0
    N = len(X)
    for i in range(N):
        l += sqrt((X[i-1]-X[i])**2+(Y[i-1]-Y[i])**2)
    return l

def area(X, Y):
    """Compute area enclosed by curve using the Shoelace Formula."""
    A = 0
    N = len(X)
    for i in range(N):
        A += (X[i]+X[i-1])*(Y[i]-Y[i-1])
    A = 0.5*abs(A)
    return A

def IPQ(l, A):
    """Given the length and area, compute Isoperimetric Quotient."""
    return 4*np.pi*A/l**2

def diameter(X, Y):
    """Compute the diameter of a curve and its orientation."""
    N = len(X)
    d = 0
    edge = (0,0)
    for i in range(N/2+1):
        Xi, Yi = X[i], Y[i]
        for j in range(i+N/4, i+3*N/4+1):
            j = j%N
            dist = np.linalg.norm([Xi-X[j], Yi-Y[j]])
            if dist > d:
                d = dist
                edge = (i,j)
    i, j = edge
    if y[j] >= y[i]:
            theta = np.arccos((x[j]-x[i])/np.linalg.norm([x[i]-x[j],y[i]-y[j]]))
        else:
            theta = np.arccos((x[i]-x[j])/np.linalg.norm([x[i]-x[j],y[i]-y[j]]))
    return d, theta

if __name__ == "__main__":
    GVFcurves = ...          # list containing numpy arrays of shape (N,2)
        representing curves
    for curve in GVFcurves:
        X = curve[:,0]
        Y = curve[:,1]
        l = length(X ,Y)
        A = area(X, Y)
        d, theta = diameter(X, Y)
```

A.4 spectral_clustering.py

This script partitions the datasets in the list `collection` into a number of clusters, using spectral clustering.

```
import numpy as np
from scipy.cluster.vq import kmeans2

def psi1(A, B):
    lA = A[:,0]             # lengths in A
    AA = A[:,1]             # areas in A
    lB = B[:,0]             # lengths in B
    AB = B[:,1]             # areas in B
    res = 0
    for elt in A:
        res += (np.sqrt((elt[0]-lB)**2+(elt[1]-AB)**2)).min()
    for elt in B:
        res += (np.sqrt((elt[0]-lA)**2+(elt[1]-AA)**2)).min()
    return 1.0/(A.shape[0]+B.shape[0])*res

def psi2(A, B):
    lA = A[:,0]
    AA = A[:,1]
    lB = B[:,0]
    AB = B[:,1]
    res = 0
    for elt in A:
        res += (np.sqrt((elt[0]-lB)**2+(elt[1]-AB)**2)).min()
    res = 0.5*res/A.shape[0]
    for elt in B:
        res += (np.sqrt((elt[0]-lA)**2+(elt[1]-AA)**2)).min()
    res = 0.5*res/B.shape[0]
    return res

if __name__ == "__main__":
    collection = [...]              # collection of datasets
    names = [...]                   # names of images corresponding to datasets

    P = len(collection)
    A = np.zeros((P,P))             # similarity matrix

    similarity_func = psi1          # choice of similarity function
    for i in range(P):
        for j in range(i+1,P):
            A[i,j] = similarity_func(collection[i],collection[j])
            A[j,i] = A[i,j]

    d = np.sum(A, axis=1)
    Dinvsq = np.diag(1./np.sqrt(d))                         # D^(-1/2)
    L = np.eye(P) - np.dot(Dinvsq, np.dot(A, Dinvsq))       # Laplacian matrix

    lambdas, vectors = np.linalg.eig(L)
    lambda_2 = np.sort(lambdas)[1]                          # second smallest
        eigenvalue
    i = np.argmin(abs(lambdas-lambda_2))
    v = vectors[i]                                          # corresponding
        eigenvector

    k = 3
    points = [np.percentile(v, 0.5*(i/float(k)+(i+1)/float(k))*100) for i in range(
        k)]      # initial location of centroids in k-means method
    centroids, labels = kmeans2(v, points, minit='matrix')
                                        # returns label for each image
    for i in range(P):
        print labels[i], names[i]
```