Anisotropic Total Variation Based Image Restoration Using Graph Cuts

Bjørn Rustad

Master of Science in Physics and Mathematics
Submission date: February 2015
Supervisor: Markus Grasmair, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

In this thesis we consider a particular kind of edge-enhancing image restoration method based on total variation. We want to address the fact that the total variation method in some cases leads to contrast loss in thin structures. To reduce the contrast loss a directional dependence is introduced through an anisotropy tensor. The tensor controls the regularization applied based on the position in the image and the direction of the gradient. It is constructed using edge information extracted from the noisy image. We optimize the resulting functional using a graph cut framework; a discretization which is made possible by a coarea and a Cauchy–Crofton formula. In the end we perform numerical studies, experiment with the parameters and discuss the results.

Sammendrag

I denne masteroppgaven ser vi på en spesifikk kant-bevarende støyfjerningsalgoritme basert på «total variation». Vi tar for oss at «total variation» i noen tilfeller fører til tap av kontrast i detaljer og tynne strukturer. For å redusere kontrast-tapet introduserer vi en retningsavhengig anisotropitensor. Denne tensoren kontrollerer støyfjerningen basert på posisjonen i bildet, og retningen til gradienten i punktet. Den blir konstruert basert på kant-informasjon fra det opprinnelige støyete bildet. Vi minimerer den resulterende funksjonalen i et graf-kutt-rammeverk, som er gjort mulig ved hjelp av en coarea- og en Cauchy–Crofton-likning. Vi avslutter med en numerisk studie, eksperimentering med parametrene og diskusjon av resultatene.

Preface

This master thesis concludes my study at the Applied Physics and Mathematics Master's degree program with specialization in Industrial Mathematics at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisor Markus Grasmair at the Department of Mathematical Sciences for invaluable help and discussion throughout my work with my project and this thesis.

Finally I would like to thank my family for their support, and Mats, Lars, Kine, Hager, Edvard and Henrik for productive discussions around the coffee pot.

Bjørn Rustad, February 8, 2015.

Contents

Chapter 1

Introduction

Image processing is becoming an increasingly important part of our modern computerized world. Tasks previously only performed by humans, like detecting edges, recognizing textures and inferring shapes and motions can now be performed algorithmically. The background of these methods spans several fields, including psychology and biology for the study of human vision, statistics and analysis for the mathematical background, and computer science for their implementation and performance analysis.

Image restoration methods are concerned with trying to remove noise or recover otherwise degraded images. Possible noise can result from the physical nature of light traveling to your sensor, dust on your lens, and many other sources. Therefore numerous different approaches to denoising exist, each having their own strengths and weaknesses. Some of these are introduced in Chapter 2, and one of the main challenges they all face is the recovery of edges.

A method well known for recovering edges is the total variation method, as the total variation does not favor smooth gradients over edges. I gave an overview of this method in my project work [1], where I used a graph cut framework to obtain a numerical solution. The method consists of trying to reduce the total variation of the image, while still staying "close" to the original.

A problem with the total varation method is that contrast is often lost, especially in fine details and thin structures. In this thesis we try to alleviate this. We extend the method by introducing an anisotropy tensor into the total variation, thus making it directionally dependent. This means we can control the regularization applied to the image based on position and direction. The main idea is then to reduce the regularization applied across edges in the image, while we still regularize along them.

The variational problem we obtain is a convex minimization problem, and many optimization approaches exist. We choose to discretize in such a way that we can apply the same graph cut framework used in my project work [1]. Through the coarea formula, the functional is decomposed into a sequence of minimization problems, one for each level of the image. These separate level problems are then transformed and discretized further using an anisotropic Cauchy–Crofton formula that we develop. Similar formulas have been presented before in other contexts.

A nice property of this numerical approach is that we can prove that the graph cut framework finds an exact global minimizer of the discrete functional. Additionally we verify that the discrete functional is consistent with the continuous one.

We present and implement two maximum flow algorithms that allow us to find minimum cuts corresponding to minimizers of the discrete functionals. The push-relabel algorithm is considered to be the fastest and most versatile for general graphs, while the Boykov–Kolmogorov algorithm is specially tailored for the type of graphs we find in these kinds of imaging applications. We describe every part of the method in detail such that it can be easily implemented by the reader. In addition, a C++ implementation is attached.

In the end we present numerical results that show how the different parameters affect the restoration, and we look into and explain some artifacts caused by approximations in the discretization. Further we look at how the introduction of the anisotropy in certain cases amend some of the weaknesses of the total variation method. We particularly look at how contrast loss is reduced in images containing thin structures such as fingerprints.

Chapter 2

Methods in image restoration

There are numerous methods in image restoration, but we do not have time nor space to discuss them all. In short overview, which is an extension of the one given in my project [1], we will focus on the methods related to the anisotropic total variation method considered later in this thesis. See [2] and [3] for more background on image processing in general.

In this chapter, and also in the rest of the thesis we will assume that we are given an image $f : \Omega \to \mathbb{R}$ where Ω is a rectangular, open domain. Because of limitations in the numerical method used, the codomain is $\mathbb{R}$ and we are thus restricted to monochrome, or grayscale images. Such images are produced in large numbers by for example ultrasound, X-ray and MRI machines.

The space in which the image f resides in will vary, but since we are looking at image restoration methods, we assume that it includes some kind of noise. Depending on the application and how the image is obtained, one might construct different models describing different types of noise.

We will assume that the given image f is a combination of an underlying, actual image u^*, and some noise δ. The simplest model is additive noise where the assumption is that $f = u^* + \delta$. There is also multiplicative noise where $f = u^* \cdot \delta$. An other much seen noise type is salt and pepper noise, which is when black and white pixels randomly appear in the image.

These are only models, and in the real world the noise might be more complex, and even come from a combination of sources. Depending on the application, the goal might not even be to recover u^*, but rather to obtain an output which fulfills certain smoothness or regularity properties. In any case, we will continue denoting the noisy input image f and use u for the output image in the description of the restoration methods.

2.1 Diffusion filtering

Diffusion filtering is a broad group of filtering and restoration methods based on physical diffusion processes. The basic idea is to take the noisy image as the initial value of some diffusion process, and then let it evolve for some time. The best known method is probably the Gaussian filter or Gaussian blur, in which one convolves the image with the Gaussian function

$$K_\sigma(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right). \tag{2.1}$$

In the discrete setting where the image consists of a grid of pixels, the Gaussian blur amounts to calculating each pixel in the output image as a weighted average of its neighboring pixels in the input image.

The Gaussian function happens to be the fundamental solution of the heat equation $\partial_t u = \Delta u$. Convolving $K_\sigma(x,y)$ with the original image f is therefore equivalent to solving the heat equation with f as initial value, until some time $T > 0$ depending on σ. Boundary conditions have to be specified of course, and one common choice is to symmetrically extend the image in x and y directions, which corresponds to zero flux boundary conditions.

By basic Fourier analysis it is possible to show that the Gaussian filter is a low-pass filter which attenuates high frequencies. Further theory can be found in Weickert's book on anisotropic diffusion [4].

The main concern with the Gaussian filter is that it will, in addition to smoothing out possible noise, remove details from the image, which motivates the next set of methods, where the amount of diffusion can vary between different parts of the image.

2.1.1 Non-linear diffusion filtering

In the theory of the heat equation one can introduce a *thermal diffusivity* α such that the equation becomes

$$\begin{cases} \partial_t u = \operatorname{div}\left(\alpha(\nabla u)\nabla u\right), \\ u|_{t=0} = f. \end{cases} \tag{2.2}$$

The thermal diffusivity $\alpha(\nabla u) = \alpha(x, \nabla u)$ is material dependent, and can also vary throughout the object. It specifies how well heat travels through the specific point in the object. We can make use of this in the image restoration context by specifying different diffusivity in different parts of the image, in an effort to reduce noise without loosing image detail. Optimally, we would like there to be a lot of diffusion in smooth parts of the image, and not so much in areas with a lot of details.

One much-studied non-linear diffusion equation is the Perona–Malik equation

$$\partial_t u = \operatorname{div}\left(\frac{\nabla u}{1+\frac{|\nabla u|^2}{\lambda^2}}\right). \tag{2.3}$$

The thermal diffusivity $\alpha(\nabla u) = (1 + {}^{|\nabla u|^2}/_{\lambda^2})^{-1}$ varies from 1 in smooth areas to 0 as the norm of the gradient $|\nabla u|$ grows.

This particular form of the thermal diffusivity has been shown to be related to how brightness is perceived by the human visual system. The model has some theoretical problems related to well-posedness, for more information see [4].

A different kind of non-linear diffusion model is the total variation flow which can be formulated as

$$\partial_t u = \operatorname{div}\frac{\nabla u}{|\nabla u|}, \tag{2.4}$$

where the diffusivity has a similar effect of reducing the diffusion in areas of high variation. As the name suggests, this model can be related to the variational total variation formulation presented later. One forward Euler time-step in the solution of this partial differential equation corresponds to the Euler–Lagrange equation of the variational formulation.

Note that we follow Weickert's terminology when it comes to the distinction between non-linear and anisotropic diffusion methods. The Perona–Malik equation, and other diffusion equations with non-homogenous diffusivities, are often by others called anisotropic, as the diffusivity depends on the location. We will name these methods non-linear and spare the anisotropy term for the "real" anisotropic methods. These are methods where the diffusivity is a tensor, and thus both location and direction dependent.

2.1.2 Anisotropic diffusion

The diffusivity is made directionally dependent by introducing a diffusion *tensor* $A(u)$ such that the initial boundary value problem becomes

$$\begin{cases} \partial_t u = \operatorname{div}\left(A(u)\nabla u\right) & \text{on } \Omega \times (0,\infty), \\ u|_{t=0} = f & \text{on } \Omega, \\ A(u)\nabla u \cdot \nu = 0 & \text{on } \partial\Omega \times (0,\infty), \end{cases} \tag{2.5}$$

where ν is the outer normal of Ω. The tensor $A(u)$ is constructed such as to diminish the effect of ∇u across what we believe to be edges in the image. This way, there will also be less diffusion through these edges. Weickert [4] suggests constructing $A(u)$ based on the edge estimator ∇u_σ where

$$u_\sigma := K_\sigma * \tilde{u} \tag{2.6}$$

and $\tilde{u}$ is an extension of u from Ω to $\mathbb{R}^2$ made by symmetrically extending u across the boundary of Ω. Assuming we are at an edge in the image, the direction of ∇u_σ should be perpendicular to the edge, while its magnitude will provide information on the steepness of the edge.

To extract this information, and also to identify features on a larger scale, the *structure tensor* is introduced

$$S_\rho(x) := K_\rho * (\nabla u_\sigma \otimes \nabla u_\sigma), \tag{2.7}$$

where the convolution with the Gaussian function K_ρ is done component-wise. The anisotropy tensor $A(u)$ can then be constructed based on the eigenvectors and eigenvalues of $S_\rho(x)$. The structure tensor and its properties will be discussed further when we introduce our anisotropic total variation functional.

Assuming some smoothness, symmetry and uniform positive definiteness on $A(u)$ one can prove well-posedness, regularity and an extremum principle of the problem (2.5) as done in [4].

However, even if the diffusivity tensor was introduced to reduce the amount of smoothing across edges, the solution of (2.5) will still be infinitely differentiable [4], i.e. $u(T) \in C^\infty(\Omega)$ for $T > 0$. Thus there are no real discontinuities, and no real edges in the solution.

Further, the anisotropic diffusion may introduce structure based on noise, when there really was no structure to begin with. This is a problem we aim to avoid in our anisotropic total variation method.

2.2 Total variation

Total variation was initially introduced to the field of image restoration by Rudin, Osher and Fatemi in [5] and is usually formulated as a minimization problem

$$\begin{gathered}\min_{u \in L^p(\Omega)} F(u), \\ F(u) = \underbrace{\int_\Omega |u - f|^p \, dx}_{\text{fidelity term}} + \beta \underbrace{\int_\Omega |\nabla u| \, dx}_{\text{regularization term}},\end{gathered} \tag{2.8}$$

where p is normally taken to be 1 or 2. The fidelity term penalizes images u that are far from the original image f. The regularization term is the total variation of the image, and minimizing it will reduce the variation and thus regularize the image. The β parameter controls the strength of the regularization. Note that $u = f$ is a minimizer of the fidelity term, while a constant image $u = c$ is a minimizer of the regularization term.

As this restoration method is the one which will be extended later in this thesis, we will look a little bit more deeply into the background and the numerical methods relating to it.

Since we do not only want to consider differentiable images $u \in C^1(\Omega)$ for which the gradient exists, we introduce the total variation using the distributional derivative.

Definition 2.1 (Total variation)**.** *Given a function $u \in L^1(\Omega)$, the total variation of u, often written $\int_\Omega |Du|\, dx$, where the D is the gradient taken in the distributional sense, is*

$$\mathrm{TV}(u) = \int_\Omega |Du|\, dx = \sup\left\{ \int_\Omega u \cdot \operatorname{div} \varphi \, dx : \varphi \in C_c^\infty\left(\Omega, \mathbb{R}^2\right), \|\varphi\|_{L^\infty(\Omega)} \leq 1 \right\}. \tag{2.9}$$

The test functions φ are taken from $C_c^\infty\left(\Omega, \mathbb{R}^2\right)$, the space of smooth functions from Ω to $\mathbb{R}^2$ with compact support in Ω.

Note that since Ω is open and bounded, the test functions φ vanish on the boundary of Ω. Thus no variation is measured at the boundary.

As we are searching for an image with low total variation, it is useful to introduce the space of functions of bounded variation.

Definition 2.2 (Functions of bounded variation)**.** *The space of functions of bounded variation* $\mathrm{BV}(\Omega)$ *is the space of functions $u \in L^1(\Omega)$ for which the total variation is finite, i.e.,*

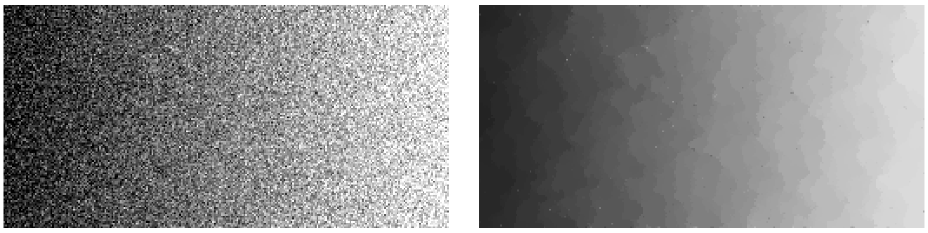
$$\mathrm{BV}(\Omega) = \left\{ u \in L^1(\Omega) : \mathrm{TV}(u) < \infty \right\}. \tag{2.10}$$

Our optimization problem has thus become

$$\min_{u \in \mathrm{BV}(\Omega)} \int_\Omega |u - v|^p \, dx + \beta \mathrm{TV}(u). \tag{2.11}$$

As with any restoration method, the total variation method has its strengths and weaknesses. Its main strength is its ability to recover edges in the input image. The total variation of a section only takes the absolute change into account, and does not favor gradual changes like the diffusion methods.

There is also a theoretical result stating that the set of edges in the solution u is contained in the set of edges in the original image f, thus no new edges are created [6]. However, in the presence of noise, the method may introduce or rather



(a) Noisy gradient **(b)** Total variation restoration

Figure 2.1: Although the original gradient was smooth, the total variation method manages to find structure in the noise, and create edges in the restored image.

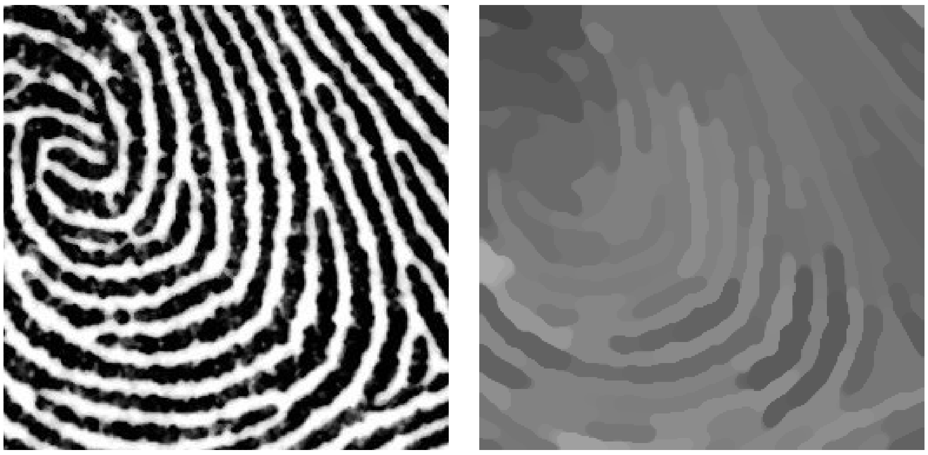


Figure 2.2: A fingerprint heavily regularized using the total variation method. The originally white and black ridges have been brought closer in value, to reduce the total variation.

"find" new edges that were not in the original image, since flat sections of zero variation are encouraged by the functional. This effect is called the *stair-casing effect*, and can be seen in Figure 2.1 where a noisy gradient has been restored using the total variation method.

Fine details, thin objects and corners may suffer from contrast loss since bringing them closer to their surroundings reduces the total variation. An example of this is shown in Figure 2.2, where a not particularly noisy fingerprint image has been strongly regularized. The original black and white levels have been brought closer to yield a lower total variation in the regularized image.

2.2.1 Numerical methods

See [7] for an overview of some of the numerical methods relating to total variation image restoration. Amongst others it describes some dual and primal-dual methods, as well as the graph cut approach we take in this thesis.

Graph cut approach

Using graph cuts is the approach we will be taking later when considering the anisotropic total variation regularization, and it is therefore valuable to briefly look into how graph cuts are used in the case of regular total variation.

A graph cut is a set of edges that when removed will separate the graph into two disconnected parts. A minimum cut is a cut such that the sum of the weight of the edges in the cut is minimal. It has been shown that for some discrete functionals, it is possible to construct graphs for which the minimum cuts correspond to minimizers of the functional.

In the discrete setting our image consists of pixels, and is represented by a function $u : \mathcal{G} \to \mathcal{P}$ where $\mathcal{G}$ is a regular grid over Ω, and $\mathcal{P} = \{0, \dots, L-1\}$ is the discrete set of pixel values, or *levels.* We denote the value in pixel x as $u(x) = u_x$.

For an image u and a level λ we denote the *level set* by $\{u > \lambda\}$, defined as the set $\{x \in \Omega : u_x > \lambda\}$. The thresholded image u^λ, an indicator function, is then defined as

$$u^\lambda = \chi_{u>\lambda}. \tag{2.12}$$

Here, χ_E signifies the characteristic function of the set E, the function which is equal to one in every point in E, and zero elsewhere.

The idea of the graph cut approach is to decompose the minimization problem into one minimization problem for each level of the image, and then solve them separately before combining the results.

Through careful manipulation of the continuous functional in (2.11) it is possible to obtain a discrete functional decomposed as a sum over all the level values on the form

$$F(u) = \sum_{\lambda=0}^{L-2} \sum_{x} F_\lambda^x(u_x^\lambda) + \beta \sum_{\lambda=0}^{L-2} \sum_{(x,y)} F^{x,y}(u_x^\lambda, u_y^\lambda) =: \sum_{\lambda=0}^{L-2} F_\lambda(u^\lambda) \tag{2.13}$$

where the sum over (x, y) is over all pixel pairs (x, y) in a neighbor relation, i.e. pixels that are “close” to each other. The actual form of the functional, and the steps to construct it will be presented later.

The graph cut we find will for each level λ give us the thresholded image u^λ, and they can then be combined to form the complete image u.

When constructing the graph used to find the thresholded image u^λ, we have two special vertices, one representing the set $\{u > \lambda\}$, and one which represents the set $\{u \leq \lambda\}$. The pixels are then connected to these vertices with a weight representing how strongly they are related to the corresponding set. This weight will be based on the value of F^x_λ.

Additionally there are connections between pixels in a neighborhood relation, representing the energy $F^{x,y}$. Thus when finding a cut, we partition the pixels into the sets $\{u > \lambda\}$ and $\{u \leq \lambda\}$. And if in addition the cut is minimal, we know that the edges cut have minimal weight, and can prove that the u^λ found minimizes the functional in (2.13).

Chapter 3

Continuous formulation

In the previous chapter we saw that there are many different approaches to the image restoration problem, all with their own strengths and weaknesses. The method considered in this thesis is an anisotropic total variation formulation, and the aim is to keep the strengths of the anisotropic diffusion and total variation methods, while eliminating some of their weaknesses.

This chapter will be devoted to the continuous formulation of the method. We will look at the functional we want to minimize and its different forms, and briefly discuss its well-posedness. Through the anisotropic coarea formula, the anisotropic total variation is rewritten as an integral of the perimeter of all the level sets of the image.

Following that, the anisotropic Cauchy–Crofton formula is introduced to make it feasible to calculate the perimeter of these level sets. All of this leads up to the discretization of our functional in the next chapter.

3.1 Anisotropic total variation

The method considered will build on the total variation regularization method of Section 2.2. From anisotropic diffusion in Section 2.1.2 we borrow the idea of making the regularization in each point directionally dependent. We introduce the anisotropic total variation

$$\mathrm{TV}_A(u) = \int_\Omega \sqrt{\nabla u(x)^T A(x) \nabla u(x)}\, dx \tag{3.1}$$

for all $u \in C^1(\Omega)$. We assume here that $A(x)$ is continuous and positive definite, and we will later need the eigenvalues of $A(x)$ to be uniformly bounded below and above. If $A(x)$ is the identity matrix we get the regular total variation found in

(2.8). When minimizing the regular total variation, we will also try to reduce the variation over known edges in the image. This can lead to unwanted contrast loss, especially in fine details. By controlling $A(x)$ such that the contribution of $\nabla u(x)$ is reduced across known edges, we hope to retain the regularization properties of the original method while reducing this contrast loss. If the variation across an edge is “ignored” by the functional, there is no gain in reducing the height of the edge as before.

Note that $u(x)$ and $A(x)$ are always dependent on the position in the image x, but we will sometimes drop the x, when the intended meaning is clear.

As we will not always be working with differentiable images, we extend the definition of the total variation functional. Being symmetric positive definite, the matrix A can be factored into two symmetric matrices as $A = A^{1/2}A^{1/2}$. We can then write

$$\begin{aligned} \mathrm{TV}_A(u) &= \int_\Omega \left|A^{1/2}\nabla u\right| \, dx \\ &= \sup_{|\xi(x)|\le 1} \int_\Omega (A^{1/2}\nabla u)^T \xi \, dx \\ &= \sup_{|\xi(x)|\le 1} \int_\Omega \nabla u \cdot A^{1/2}\xi \, dx \\ &= \sup_{|\xi(x)|\le 1} \int_\Omega u \operatorname{div}(A^{1/2}\xi) \, dx \\ &= \sup_{\eta^T A^{-1}\eta \le 1} \int_\Omega u \operatorname{div} \eta \, dx, \end{aligned} \tag{3.2}$$

where ξ and $\eta = A^{1/2}\xi$ are in $C_c^\infty(\Omega, \mathbb{R}^2)$, the space of smooth vector fields with compact support. In the following we define the norms $\|\xi\|_A = \sup_x (\xi^T A \xi)^{1/2}$ and $\|\eta\|_A^* = \sup_x (\eta^T A^{-1} \eta)^{1/2}$, and with that we present the formal definition of the anisotropic total variation.

Definition 3.1 (Anisotropic total variation)**.** *For a function $u \in L^2(\Omega)$ and a continuous symmetric positive definite tensor $A : \Omega \to \mathbb{R}^{2\times 2}$ we define the anisotropic total variation*

$$\mathrm{TV}_A(u) = \sup\left\{ \int_\Omega u \operatorname{div} \xi \, dx : \xi \in C_c^\infty(\Omega, \mathbb{R}^2), \|\xi\|_A^* \le 1 \right\}. \tag{3.3}$$

With this extended definition, we have arrived at a minimization problem where we seek to find a minimizer of the functional

$$F(u) = \int_\Omega (u - f)^2 \, dx + \beta \, \mathrm{TV}_A(u). \tag{3.4}$$

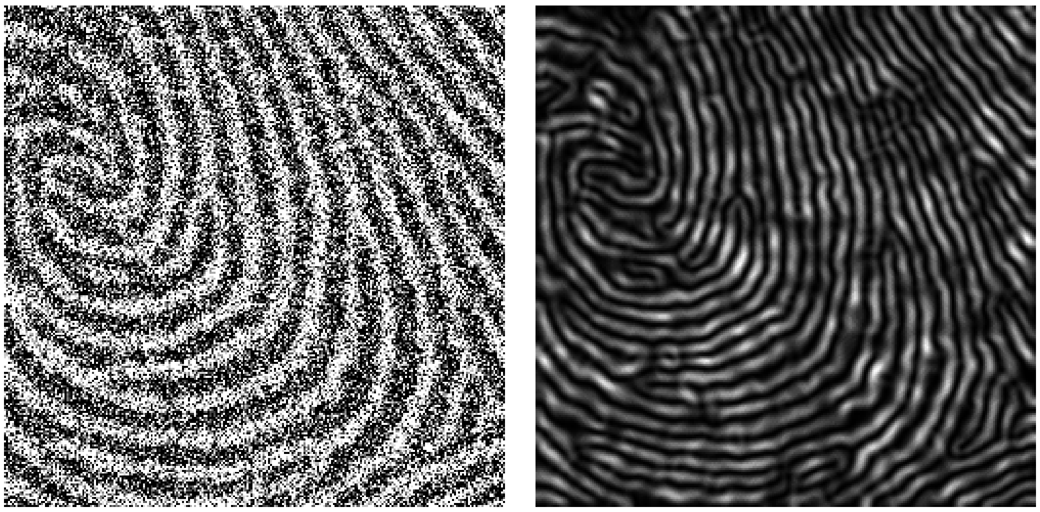


Figure 3.1: A noisy fingerprint on the left, and the largest eigenvalue of the structure tensor is $|\nabla f_\sigma(x)|^2$ on the left, which—as we can see—functions as an edge detector.

Similar functionals have been considered in [8] and [9]. The question is now how to construct the anisotropy tensor $A(x)$ to get the improvements we hope for, and how the introduction of the tensor affects our numerical solution method.

3.1.1 Anisotropy tensor

There are many possible choices for the anisotropy tensor $A(x)$. Our constraints are that we have assumed it to be continuous and symmetric positive definite, and we have some wishes for its properties. We would first and foremost like it to down-weight ∇u in (3.1) across true edges, while maintaining normal regularization properties in smooth sections.

By true edges we mean that that we do not want the tensor to be sensitive to noise in the image, and thus find edges where there are none, so we somehow want to be sure about the edges we find.

Edges can be found in many different ways, but as suggested by Weickert in his book on Anisotropic Diffusion [4], and briefly mentioned in Section 2.1.2, a good starting point is the *edge detector* ∇f_σ. The image is smoothed by a Gaussian filter as described in Section 2.1: $f_\sigma = K_\sigma * \tilde{f}$, where $\tilde{f}$ is the symmetric extension of the initial image f in $\mathbb{R}^2$. The smoothing parameter σ is called the *noise scale*, and it controls the scale at which details are considered to be noise.

As seen in Figure 3.1, the edge detector is fine for detecting edges, but it can not give us information about larger structures, like corners and textures, which is why we introduce the *structure tensor* $S_\rho(x)$. First consider the tensor $S_0(x) = \nabla f_\sigma(x) \otimes \nabla f_\sigma(x)$. It is symmetric positive semi-definite, and obviously contains no more information than the edge detector itself. Its eigenvalues are $\lambda_1 = |\nabla f_\sigma(x)|^2$ and $\lambda_2 = 0$ with corresponding eigenvectors v_1 and v_2 parallel and perpendicular to $\nabla f_\sigma(x)$ respectively.

To detect features in a neighborhood around the point x, such as corners, curved edges and coherent structures we introduce the component-wise convolution with K_ρ such that

$$S_\rho(x) := K_\rho * (\nabla f_\sigma(x) \otimes \nabla f_\sigma(x))(x). \tag{3.5}$$

The parameter ρ, called the *integration scale*, controls the size of the neighborhood which affects the structure tensor. Thus it defines the size of the structures we want our anisotropy tensor to be sensitive to.

The smoothed tensor $S_\rho(x)$ can easily be verified to be symmetric positive semi-definite, just like $S_0(x)$. In addition, when $\rho > 0$, the elements of S_ρ are smooth maps from Ω to $\mathbb{R}$.

We order the two real eigenvalues such that $\lambda_1 \geq \lambda_2$ and denote the corresponding eigenvectors v_1 and v_2. From the characteristic polynomial of $S_\rho(x) = \left(\begin{smallmatrix} s_{11} & s_{12} \\ s_{12} & s_{22} \end{smallmatrix}\right)$ we obtain a closed form expression for the eigenvalues

$$\lambda = \frac{1}{2}\left(s_{11} + s_{22} \pm \sqrt{(s_{11} - s_{22})^2 + 4s_{12}^2}\right). \tag{3.6}$$

The vector v_1 will then indicate the direction of most variation in the neighborhood. An edge will give $\lambda_1 \gg \lambda_2 \approx 0$, while smooth areas will give $\lambda_1 \approx \lambda_2 \approx 0$. In corners we have variation in the direction of v_1 but also perpendicular to v_1, so we will have $\lambda_1 \approx \lambda_2 \gg 0$. Thus the quantity $(\lambda_1 - \lambda_2)^2$ will be large around edges and small in smooth or non-coherent areas.

To extract this information from the structure tensor, we decompose it as

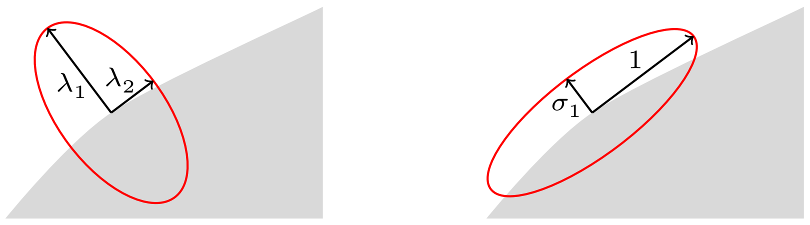
$$S_\rho(x) = U(x)\Lambda(x)U(x)^T, \tag{3.7}$$

where

$$\Lambda(x) = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \tag{3.8}$$

has the eigenvalues $\lambda_1 \geq \lambda_2$ on its diagonal, while $U(x)$ is a rotation matrix and has the eigenvectors of $S_\rho(x)$ as its columns. From this we construct a new matrix $A(x) = U(x)\Sigma(x)U(x)^T$ where

$$\Sigma(x) = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}. \tag{3.9}$$



(a) The structure tensor S_ρ. **(b)** The anisotropy tensor A.

Figure 3.2: An edge with the structure and anisotropy tensors visualized using their eigenvectors and eigenvalues.

and for σ_1 and σ_2 we choose

$$\begin{aligned}\sigma_1 &= \left(1 + \frac{(\lambda_1 - \lambda_2)^2}{\omega^2}\right)^{-1}, \\ \sigma_2 &= 1.\end{aligned} \tag{3.10}$$

Thus the eigenvectors of $A(x)$ and $S_\rho(x)$ are equal, while the eigenvalues are different. A visualization of the two tensors can be seen in Figure 3.2 where the two tensors are shown at an edge in the image.

In smooth areas, $\sigma_1 \approx 1$ and $A(x)$ will be close to the identity matrix. At or around edges, σ_1, which corresponds to the eigenvector perpendicular to the edge, will be small.

Around corners $A(x)$ will be close to the identity matrix, which gives regularization similar to smooth areas. This is one possible down-side of this tensor choice, as rounded corners may occur.

The parameter ω controls the amount of anisotropy in the method, such that if it is very large we are left with the identity matrix and our method becomes the regular total variation method. Note also that changing the parameter ω implicitly affects the amount of regularization applied. For an image u, decreasing ω will, all else being equal, decrease the lowest eigenvalue of $A(x)$ and in turn decrease the anisotropic total variation $\mathrm{TV}_A(u)$.

For the case where $\lambda_1 = \lambda_2$, the $U(x)$ in our decomposition is not well-defined. This is not a problem however, since $\Sigma(x)$ will be the identity matrix, so any orthogonal matrix will suffice for $U(x)$.

Note that the eigenvalues of S_ρ are continuous, and so are the eigenvectors (ignoring their sign) except possibly when $\lambda_1 = \lambda_2$. Thus A is also continuous except possibly in these points. When $\lambda_1 = \lambda_2$ however, the eigenvalues σ_1 and σ_2 of A will both be 1, and A is the identity matrix. Thus we can argue that if $S_\rho(x) \to \lambda I$ then $A(x) \to I$ and A is continuous in all of Ω.

See [10] for a different tensor construction, made to enhance flow structures in the image, relevant in for example fingerprint analysis.

3.2 Well-posedness

The theory of existence and uniqueness for these kinds of variational methods is a minefield of more or less subtle problems. Even if we restrict ourselves to a nice space such as $L^2(\Omega)$ we will at some point run into problems. The discussion here is not meant to give the most rigorous background, but rather an overview of what needs to be shown. Some problems will be worked around, while others will be skipped with a reference to further theory.

The basic things we ask of our functional

$$F(u) = \int_\Omega (u-f)^2 + \beta \mathrm{TV}_A(u) \tag{3.11}$$

to have a *well-posed* problem are lower semi-continuity and coercivity for existence, and convexity for uniqueness. We restrict ourself to $L^2(\Omega)$ which makes sense with our fidelity term, assuming that $f \in L^2(\Omega)$.

We consider the weak topology, as it will allow us to arrive at an existence result relatively easily. We say that a sequence f_n in $L^2(\Omega)$ converges weakly to f if

$$\lim_{n\to\infty} \int_\Omega f_n \,\xi \, dx = \int_\Omega f \,\xi \, dx \tag{3.12}$$

for all $\xi \in L^2(\Omega)$ and we write $f_n \rightharpoonup f$. A weakly convergent sequence is a sequence that converges in the weak topology.

3.2.1 Convexity

We start with convexity as it is the easiest to show. Being quadratic, the fidelity term of our functional

$$\int_\Omega (u-f)^2 \, dx \tag{3.13}$$

is obviously strictly convex. This can be shown by expanding and rearranging the strict convexity condition

$$\int_\Omega (\lambda u_1 + (1-\lambda)u_2 - f)^2 \, dx < \lambda \int_\Omega (u_1 - f)^2 \, dx + (1-\lambda) \int_\Omega (u_2 - f)^2 \, dx \tag{3.14}$$

to obtain that it is equivalent to

$$-\lambda(1-\lambda) \int_\Omega (u_1 - u_2)^2 \, dx < 0 \tag{3.15}$$

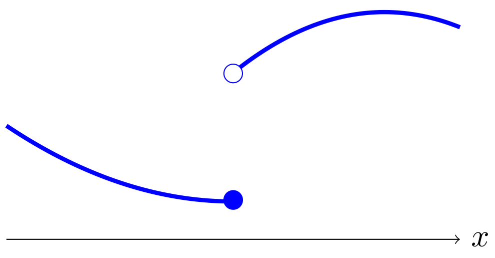


Figure 3.3: A lower semi-continuous function $f : \mathbb{R} \to \mathbb{R}$ can have discontinuities, but for a convergent sequence $x_k \to x$ we always have $f(x) \leq \liminf_{k\to\infty} f(x_k)$.

which is true for $0 < \lambda < 1$ and $u_1 \neq u_2$.

The anisotropic total variation

$$\mathrm{TV}_A(u) = \sup_{\|\xi\|_A^* \leq 1} \int_\Omega u \operatorname{div} \xi \, dx \tag{3.16}$$

can be thought of as—and has the properties of—a semi-norm, and is therefore convex. The sum of the fidelity and regularization terms is thus strictly convex, which, given the existence of a minimizer, implies uniqueness.

3.2.2 Coercivity

Coercivity relates to how the functional behaves when the norm of the image u tends to infinity. What we need in order to conclude with existence is weak sequential coercivity. Thus we need all level sets $F^\alpha = \{u \in L^2(\Omega) : F(u) \leq \alpha\}$ to be *weakly sequentially pre-compact*, meaning that all sequences in the set contain a subsequence weakly converging to an element of the closure of the set.

It is obvious from the fidelity term that for some fixed $f \in L^2(\Omega)$, if $\|u\|_{L^2} \to \infty$ then $F(u) \to \infty$. This implies that all the level sets F^α are bounded. Since $L^2(\Omega)$ is a Hilbert space all bounded sequences contain a weakly convergent subsequence. Thus all the level sets F^α are weakly sequentially pre-compact.

3.2.3 Lower semi-continuity

The lower semi-continuity is the most tricky part, and this is where we will take some shortcuts. Lower semi-continuity for a functional F at a point u means that at points u_ϵ close to u, the functional takes values either close to or above $F(u)$. More specifically, for every sequence u_k converging to u, we have $F(u) \leq \liminf_k F(u_k)$. For a function $f : \mathbb{R} \to \mathbb{R}$ this can be visualized as in Figure 3.3.

Since our space $L^2(\Omega)$ is of infinite dimensions things become a little problematic here. The problem lies in the fact that a functional which is continuous with respect to sequences is not necessarily continuous with respect to the underlying topology. In other words, in these spaces, there can be a difference between sequential continuity and topological continuity. Topological continuity implies sequential continuity, but the converse does not hold. One way to get around this would be to consider topological *nets*, an extension of sequences, but for simplicity, and because it might not add much to the understanding of the restoration method, we will stick to proving sequential lower semi-continuity and referring to further theory. For further reading on the theory of sequential versus topological continuity see for example Megginson's book on Banach space theory [11].

The mapping $u \mapsto \int_\Omega u\xi\, dx$ is weakly continuous for all $\xi \in L^2(\Omega)$. Note that when we write weakly continuous it is not a weaker version of continuity, but rather continuity in the weak topology, and the same goes for weak lower semi-continuity.

Before arguing that our own functional is sequentially weakly lower semi-continuous, we present a needed result.

Lemma 3.2. *Assume that the functional $F : L^2(\Omega) \to \mathbb{R}$ is defined by*

$$F = \sup_i F_i \tag{3.17}$$

where all the F_i are sequentially weakly lower semi-continuous, then F is sequentially weakly lower semi-continuous, meaning that for any sequence $u_k \rightharpoonup u$ we have $F(u) \le \liminf_k F(u_k)$.

Proof. For any sequence $u_k \rightharpoonup u$ in $L^2(\Omega)$ we have

$$F(u) = \sup_i F_i(u) \le \sup_i \liminf_{k\to\infty} F_i(u_k) \tag{3.18}$$

from the sequential weak lower semi-continuity of F_i. Using that $\liminf_{k\to\infty} u_k = \sup_k \inf_{l\ge k} u_l$, we obtain

$$\begin{aligned} F(u) &\le \sup_i \sup_k \inf_{l\ge k} F_i(u_l) \\ &= \sup_k \sup_i \inf_{l\ge k} F_i(u_l) \\ &\le \sup_k \inf_{l\ge k} \sup_i F_i(u_l) \\ &= \liminf_{k\to\infty} F(u_k) \end{aligned} \tag{3.19}$$

which proves that F is sequentially weakly lower semi-continuous. □

In our functional in (3.4), we first consider the fidelity term, and rewrite it as a supremum

$$\int_\Omega (u-f)^2\,dx = \sup\left\{\int_\Omega (u-f)\,\xi\,dx : \xi \in L^2(\Omega), |\xi(x)| \le |u(x)-f(x)|\right\} \quad (3.20)$$

As the map $u \mapsto \int_\Omega (u-v)\xi\,dx$ is continuous in the weak topology, the fidelity term is thus a supremum of weakly continuous functionals, and is thus by Lemma 3.2 sequentially lower semi-continuous.

For the regularization term the approach is similar. With our extended definition from (3.3), we have

$$\mathrm{TV}_A(u) = \sup\left\{\int_\Omega u\,\mathrm{div}\,\xi\,dx : \xi \in C_c^\infty(\Omega,\mathbb{R}^2), \|\xi\|_A^* \le 1\right\} \quad (3.21)$$

This is again a supremum of weakly continuous functionals. Thus the regularization term is by Lemma 3.2 also sequentially weakly lower semi-continuous.

The sum of the two terms is trivially sequentially weakly lower semi-continuous functional since

$$\begin{aligned} F_1(u)+F_2(u) &\le \liminf_{k\to\infty} F_1(u_k) + \liminf_{k\to\infty} F_2(u_k) \\ &= \lim_{k\to\infty}\left(\inf_{l\ge k} F_1(u_l) + \inf_{l\ge k} F_2(u_l)\right) \\ &\le \liminf_{k\to\infty}\left(F_1(u_k)+F_2(u_k)\right), \end{aligned} \quad (3.22)$$

and thus our functional is sequentially weakly lower semi-continuous.

The usual ways of going from coercivity and lower semi-continuity to existence do not work in infinite dimensions. But with sequential coercivity and sequential lower semi-continuity in the weak topology we can conclude that we have existence from [12, Theorem 5.1].

3.3 Anisotropic coarea formula

The anisotropic coarea formula we present here will allow us to write the anisotropic total variation as an integral over the levels of the image. For a similar presentation of the regular coarea formula for all $f \in \mathrm{BV}(\Omega)$ see [13].

First we define the thresholded image at level s.

Definition 3.3 (Thresholded image). *The thresholded image at level s is the function*

$$u^s(x) = \begin{cases} 1 & \textit{if } u(x) > s, \\ 0 & \textit{otherwise.} \end{cases} \quad (3.23)$$

This will be used throughout the rest of the thesis. Note that given the thresholded image for every level, we are able to reconstruct the image as

$$u(x) = \sup \{s : u^s(x) = 1\}. \tag{3.24}$$

The thresholded image definition also allows us to write a non-negative image $u \geq 0$ as an integral over all the layers

$$u(x) = \int_0^\infty u^s(x)\, ds. \tag{3.25}$$

Note that (3.25) only holds for non-negative images, which complicates the proof of the anisotropic coarea formula a little.

Theorem 3.4 (Anisotropic coarea formula)**.** *Given an image $u \in \mathrm{BV}(\Omega)$, the anisotropic total variation can be written as an integral over all the levels*

$$\mathrm{TV}_A(u) = \int_{-\infty}^\infty \mathrm{TV}_A(u^s)\, ds. \tag{3.26}$$

For the proof we will avoid measure theory and follow a proof given in [9], but first we will present a necessary result from measure theory.

Theorem 3.5 (Lebesgue's Dominated Convergence theorem)**.** *Let $\{f_n\}$ be a sequence of real-valued measurable functions on a space S with measure $d\mu$ which converges almost everywhere to a real-valued measurable function f. If there exists an integrable function g such that $|f_n| \leq g$ for all n, then f is integrable and*

$$\lim_{n\to\infty} \int_S f_n \, d\mu = \int_S f \, d\mu. \tag{3.27}$$

For a proof and further background on measure theory and Lebesgue integration theory see for example [14].

Proof of the anisotropic coarea formula. Assume that $u \in C^1(\Omega) \cap \mathrm{BV}(\Omega)$. The extension to all functions $u \in \mathrm{BV}(\Omega)$ will not be considered here, but for the case of regular total variation see [15, Theorem 5.3.3].

Proof of upper bound. Assume that $u \geq 0$ such that the integral representation in (3.25) holds, then inserting (3.25) into the extended total variation definition in (3.3) gives

$$\begin{aligned} \mathrm{TV}_A(u) &= \sup_{\|\xi\|_A^* \leq 1} \int_\Omega \left(\int_0^\infty u^s ds \right) \operatorname{div} \xi \, dx = \sup_{\|\xi\|_A^* \leq 1} \int_\Omega \int_0^\infty u^s \operatorname{div} \xi \, ds \, dx \\ &\leq \int_0^\infty \left(\sup_{\|\xi\|_A^* \leq 1} \int_\Omega u^s \operatorname{div} \xi \, dx \right) ds = \int_0^\infty \mathrm{TV}_A(u^s)\, ds. \end{aligned} \tag{3.28}$$

For $u \le 0$ we use that $\mathrm{TV}_A(-v) = \mathrm{TV}_A(v)$ and that $\mathrm{TV}_A(c+v) = \mathrm{TV}_A(v)$ for any constant c. Note that $-u \ge 0$ and that its thresholded image $(-u)^s$ will be exactly the opposite of u^{-s}, that is $(-u)^s = 1 - u^{-s}$. This allows us to show that

$$\begin{aligned}\mathrm{TV}_A(u) = \mathrm{TV}_A(-u) &\le \int_0^\infty \mathrm{TV}_A((-u)^r)\,dr = \int_0^\infty \mathrm{TV}_A(1-u^{-r})\,dr \\ &= \int_0^\infty \mathrm{TV}_A(u^{-r})\,dr = \int_{-\infty}^0 \mathrm{TV}_A(u^s)\,ds.\end{aligned} \tag{3.29}$$

Following from the supremum definition of the anisotropic total variation in (3.3), we obtain the inequality

$$\begin{aligned}\mathrm{TV}_A(u_1+u_2) &= \sup_{\|\xi\|_A^*\le 1}\int_\Omega (u_1+u_2)\operatorname{div}\xi\,dx \\ &\le \sup_{\|\xi\|_A^*\le 1}\int_\Omega u_1\operatorname{div}\xi\,dx + \sup_{\|\xi\|_A^*\le 1}\int_\Omega u_2\operatorname{div}\xi\,dx \\ &= \mathrm{TV}_A(u_1) + \mathrm{TV}_A(u_2).\end{aligned} \tag{3.30}$$

Next, we write a general u as a difference of two positive functions $u = u_+ - u_-$ where $u_+ = \max\{u,0\}$ and $u_- = -\min\{u,0\}$. Inserting (3.28) and (3.29) into (3.30) we obtain

$$\begin{aligned}\mathrm{TV}_A(u) &\le \mathrm{TV}_A(u_-) + \mathrm{TV}_A(u_+) = \mathrm{TV}_A(-u_-) + \mathrm{TV}_A(u_+) \\ &\le \int_{-\infty}^0 \mathrm{TV}_A((-u_-)^s)\,ds + \int_0^\infty \mathrm{TV}_A(u_+^s)\,ds \\ &= \int_{-\infty}^0 \mathrm{TV}_A(u^s)\,ds + \int_0^\infty \mathrm{TV}_A(u^s)\,ds = \int_{-\infty}^\infty \mathrm{TV}_A(u^s)\,ds.\end{aligned} \tag{3.31}$$

Note that u_+ and u_- will not be differentiable everywhere, but we did not use the differentiability of u in this part of the proof.

Proof of lower bound. Define the function

$$m(t) = \int_{\{x\in\Omega : u(x)\le t\}} \|\nabla u\|_A\,dx, \tag{3.32}$$

and note that $m(\infty) = \mathrm{TV}_A(u)$ and $m(-\infty) = 0$. Since $m(t)$ is non-decreasing with t, we can apply the existence theorems of Lebesgue [16, Thm. 17.12, 18.14] to conclude that $m'(t)$ exists almost everywhere and that the following inequality holds:

$$\int_{-\infty}^\infty m'(t)\,dt \le m(\infty) - m(-\infty) = \mathrm{TV}_A(u). \tag{3.33}$$

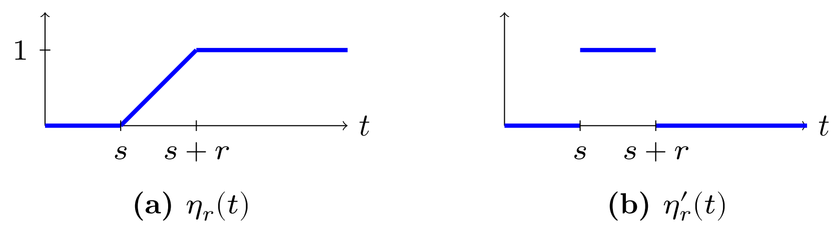


Figure 3.4: Visualization of the cut-off function $\eta_r(t)$ and its derivative.

Next, fix an $s \in \mathbb{R}$ and define the cut-off function

$$\eta_r(t) = \begin{cases} 0 & \text{if } t < s, \\ (t-s)/r & \text{if } s \le t < s+r, \\ 1 & \text{if } t \ge s+r, \end{cases} \quad {\eta_r}'(t) = \begin{cases} 0 & \text{if } t < s, \\ 1 & \text{if } s < t < s+r, \\ 0 & \text{if } t > s+r, \end{cases} \tag{3.34}$$

visualized in Figure 3.4. By composing the function η_r with our image u and using Green's formula, for example from [8, Corollary 9.32] we obtain

$$\int_\Omega -\eta_r(u) \operatorname{div} \xi \, dx = \int_\Omega {\eta_r}'(u) \nabla u \cdot \xi \, dx = \frac{1}{r} \int_{\{s<u<s+r\}} \nabla u \cdot \xi \, dx, \tag{3.35}$$

for all vector fields $\xi \in C_c^\infty(\Omega, \mathbb{R}^2)$. The measure of $\{x : u(x) = \lambda \text{ and } \nabla u(x) \neq 0\}$ is zero for all λ following from [13, Corollary I, Section 3.1.2], thus we can ignore the sets $\{u = s\}$ and $\{u = s + r\}$ in the integrals. Assuming that $\|\xi\|_A^* \le 1$ we obtain from (3.32) and (3.35) that

$$\begin{aligned} \frac{m(s+r) - m(s)}{r} &= \frac{1}{r} \int_{\{s<u\le s+r\}} \|\nabla u\|_A \, dx \\ &\ge \frac{1}{r} \int_{\{s<u\le s+r\}} \nabla u \cdot \xi \, dx = \int_\Omega -\eta_r(u) \operatorname{div} \xi \, dx. \end{aligned} \tag{3.36}$$

As the limit when $r \to 0$ of the left-hand side exists almost everywhere, suppose it exists at $s \in \mathbb{R}$. The integrand on the right-hand side $-\eta_r(u) \operatorname{div} \xi$ approaches $-u^s \operatorname{div} \xi$ pointwise almost everywhere. We also have the bound $|\eta_r(u) \operatorname{div} \xi| \le |u^s \operatorname{div} \xi|$ and know that $\xi \in C_c^\infty(\Omega, \mathbb{R}^2)$ is bounded following from the extreme value theorem. Thus we can apply Lebesgue's dominated convergence theorem, giving that $|u^s \operatorname{div} \xi|$ is integrable and

$$\int_\Omega -\eta_r(u) \operatorname{div} \xi \, dx \to \int_\Omega -u^s \operatorname{div} \xi \, dx \tag{3.37}$$

From (3.36) we then obtain

$$m'(s) \geq -\int_\Omega u^s \operatorname{div} \xi \, dx. \tag{3.38}$$

As this holds for any $\|\xi\|_A^* \leq 1$, we get from the extended total variation definition in (3.3) that $m'(s) \geq \mathrm{TV}_A(u^s)$ almost everywhere and conclude using (3.33) that

$$\mathrm{TV}_A(u) \geq \int_{-\infty}^{\infty} m'(t) \, dt \geq \int_{-\infty}^{\infty} \mathrm{TV}_A(u^s) \, ds. \tag{3.39}$$

Combining the upper and lower bounds just proved, we have equality. □

This coarea formula is our first step in transforming the anisotropic total variation into an easily discretizable expression. It allows us to consider each level λ separately when calculating the anisotropic total variation.

The anisotropic total variation of the thresholded images occurring in the anisotropic coarea formula is very much related to the size of the boundary of the level set, as the only variation in a characteristic function occurs at the boundary of the set. This is why we introduce the following definition of the anisotropic set perimeter.

Definition 3.6 (The anisotropic set perimeter)**.** *Given an anisotropy tensor A the anisotropic perimeter of a set U in Ω is defined as*

$$\mathrm{Per}_A(U; \Omega) = \mathrm{TV}_A(\chi_U). \tag{3.40}$$

The anisotropic set perimeter is not like the regular set perimeter and does not measure the length of the boundary of the set, but it can for sufficiently nice level sets be calculated in the following way

$$\begin{aligned}
\mathrm{Per}_A(\{u > s\}; \Omega) &= \mathrm{TV}_A(u^s) \\
&= \sup_{\|\xi\|_A^* \leq 1} \int_\Omega u^s \operatorname{div} \xi \, dx \\
&= \sup_{\|\xi\|_A^* \leq 1} \int_{\{u>s\}} \operatorname{div} \xi \, dx \\
&= \sup_{\|\xi\|_A^* \leq 1} \int_{\partial\{u>s\}} \nu_s \cdot \xi \, dt \\
&= \sup_{\|\eta\| \leq 1} \int_{\partial\{u>s\}} \nu_s \cdot A^{1/2} \eta \, dt \\
&= \int_{\partial\{u>s\}} \sqrt{\nu_s A \nu_s} \, dt.
\end{aligned} \tag{3.41}$$

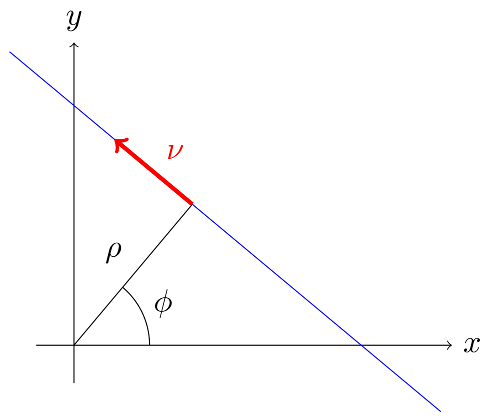


Figure 3.5: The blue line is parametrized by the angle ϕ and the distance from the origin to the line ρ, or alternatively, the pair (ν, ρ).

Here, ν_s is the unit exterior normal of the level set $\{u > s\}$. Note that because of the compact support of ξ in Definition 3.1, the parts of the boundary of U that overlap with the boundary of Ω will not be included in the perimeter.

Exterior normals and perimeters of level sets of any function $u \in \mathrm{BV}(\Omega)$ will not be considered here, but can for the isotropic case be found in for example [15, Section 5.4 and 5.5].

Using the anisotropic coarea formula and inserting the anisotropic perimeter definition we transform the anisotropic total variation and are left with the problem of minimizing the following functional

$$F(u) = \int_\Omega (u - f)^2 \, dx + \beta \int_{-\infty}^{\infty} \mathrm{Per}_A(\{u > \lambda\}; \Omega) \, d\lambda. \tag{3.42}$$

The transformation is motivated by our upcoming anisotropic Cauchy–Crofton integration formula, and the discretization, where an approximation of the perimeter will be computed using a graph cut machinery.

3.4 Anisotropic Cauchy–Crofton formula

In the fields of integral geometry and geometric measure theory there are a number of interesting integral formulas. Several of them fall in a category often referred to as *Cauchy–Crofton style formulas*, and give ways to measure geometric objects using the set of all lines in the plane. The formulas presented here will give a way to measure the length of a curve by counting the times it intersects lines in the set of all lines. The first formula will be for the isotropic case, and we will use it to prove the anisotropic formula following it.

We write $\mathcal{L}$ for the set of all lines in the plane, and parametrize them as shown in Figure 3.5. A line is parametrized by the angle $\phi \in [0, 2\pi)$ of the normal going to the origin, and the distance $\rho \in [0, \infty)$ from origin to the line. Sometimes it is more convenient to consider a unit vector ν giving the direction of the line instead of the angle parameter ϕ. We denote a line by $\ell_{\phi,\rho} = \ell_{\nu,\rho}$ where ν is a unit vector along the line, i.e. $\nu = (-\sin\phi, \cos\phi)^T$. By defining the measure on this set $d\mathcal{L} = d\phi\, d\rho$ we are ready to introduce the Cauchy–Crofton formula. Note that the measure $d\mathcal{L}$ is invariant under rotations.

Theorem 3.7 (The Euclidean Cauchy–Crofton formula)**.** *Given a differentiable curve C in $\mathbb{R}^2$, the length of this curve $|C|$ is related to the set of lines $\mathcal{L}$ as follows*

$$\int_{\mathcal{L}} \#(\ell_{\phi,\rho} \cap C)\, d\mathcal{L}(\ell_{\phi,\rho}) = 2\,|C|, \tag{3.43}$$

where $\#(\ell_{\phi,\rho} \cap C)$ is the number of times the line $\ell_{\phi,\rho}$ intersects the curve C.

Proof. See [17, Theorem 3, Section 1-7]. □

If our space is equipped with a metric tensor $M(x)$ such that the inner product of two vectors a and b in a point x is calculated as $\langle a, b\rangle_M = \langle a, M(x)b\rangle$, then the length of a curve γ parametrized by some parameter t becomes

$$|\gamma|_M = \int_\gamma \sqrt{\langle \dot\gamma, M(\gamma(t))\,\dot\gamma\rangle}\, dt. \tag{3.44}$$

We will now present and prove a Cauchy–Crofton formula in this case where our domain is equipped with a metric tensor in each point. This elegant formula is very useful when we later will discretize our perimeter calculation. The set of lines $\mathcal{L}$ is then discretized in a reasonable way, and the length of the curve C can be approximated by a sum over all these lines.

Theorem 3.8 (The anisotropic Cauchy–Crofton formula)**.** *Assume that our space Ω is equipped with a continuous positive definite metric tensor $M(x)$, whose eigenvalues are bounded by $0 < k \le \lambda_2 \le \lambda_1 \le K < \infty$ for all $x \in \Omega$. The Cauchy–Crofton formula for a differentiable curve C of finite length then becomes*

$$|C|_M = \int_{\mathcal{L}} \sum_{x \in \ell_{\nu,\rho} \cap C} \frac{\det M(x)}{2\left(\nu^T \cdot M(x) \cdot \nu\right)^{3/2}}\, d\mathcal{L}(\ell_{\nu,\rho}). \tag{3.45}$$

Proof of the anisotropic Cauchy–Crofton formula. Assume first that our space is equipped with a constant metric tensor M. The length of a curve in this space can be calculated by transforming the curve and applying the Euclidean Cauchy–Crofton formula

$$|C|_M = \int_C \sqrt{\langle \dot{C}, M\,\dot{C}\rangle}\,dt = \int_C \sqrt{\langle M^{1/2}\dot{C}, M^{1/2}\dot{C}\rangle} = \left|M^{1/2}C\right| \tag{3.46}$$

$$= \int_{\mathcal{L}} \#(\ell_{\phi,\rho} \cap M^{1/2}C)\, d\mathcal{L}(\ell_{\phi,\rho}) \tag{3.47}$$

$$= \int_{\mathcal{L}} \#(M^{-1/2}\ell_{\phi,\rho} \cap C)\, d\mathcal{L}(\ell_{\phi,\rho}) \tag{3.48}$$

$$= \int_{\mathcal{L}} \#(m_{\phi,\rho} \cap C)\left|J_M(\ell_{\phi,\rho})\right|\, d\mathcal{L}(m_{\phi,\rho}). \tag{3.49}$$

Here $J_M(\ell_{\phi,\rho})$ is the Jacobian of the coordinate transformation $F : \mathcal{L} \to \mathcal{L}$, which maps $\ell_{\phi,\rho} \mapsto M^{1/2}\ell_{\phi,\rho}$.

We will now compute the Jacobian $J_M(\ell_{\phi,\rho})$. As $M \in \mathbb{R}^{2\times 2}$ is symmetric, so is $M^{1/2}$, and it admits a decomposition $M^{1/2} = U\Sigma U^T$ where the components correspond to the following coordinate transformations

$$U(\ell_{\nu,\rho}) = \ell_{\phi+\xi,\rho} = \ell_{U\nu,\rho} \tag{3.50}$$

$$U^T(\ell_{\nu,\rho}) = \ell_{\phi-\xi,\rho} = \ell_{U^T\nu,\rho} \tag{3.51}$$

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} = \begin{pmatrix} \sqrt{\lambda_1} & 0 \\ 0 & \sqrt{\lambda_2} \end{pmatrix} \tag{3.52}$$

As U and U^T correspond to rotations and our measure $\mathcal{L}$ is invariant under rotations, U and U^T do not have direct contributions to the Jacobian. They do however affect the input angle of the operator Λ such that $J_M(\ell_{\phi,\rho}) = J_{\Sigma^2}(U^T\ell_{\phi,\rho})$. Thus we will now compute $J_{\Sigma^2}(\ell_{\phi,\rho})$. Given a line

$$\ell_{\phi,\rho} = \begin{pmatrix} \rho \cdot \cos\phi \\ \rho \cdot \sin\phi \end{pmatrix} + \mathbb{R}\begin{pmatrix} -\sin\phi \\ \cos\phi \end{pmatrix}, \tag{3.53}$$

the operator Σ transforms it into

$$\Sigma\ell_{\phi,\rho} = \begin{pmatrix} \sigma_1\rho \cdot \cos\phi \\ \sigma_2\rho \cdot \sin\phi \end{pmatrix} + \mathbb{R}\begin{pmatrix} -\sigma_1 \sin\phi \\ \sigma_2 \cos\phi \end{pmatrix}, \tag{3.54}$$

which equals the line $\ell_{\theta,\eta}$ with

$$\theta = \arctan\left(\frac{\sigma_1}{\sigma_2}\tan\phi\right) \tag{3.55}$$

$$\eta = \left\langle \begin{pmatrix} \sigma_1\rho \cdot \cos\phi \\ \sigma_2\rho \cdot \sin\phi \end{pmatrix}, \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} \right\rangle = \sigma_1\rho \cdot \cos\phi \cdot \cos\theta + \sigma_2\rho \cdot \sin\phi \cdot \sin\theta. \tag{3.56}$$

As $\partial_\rho \theta = 0$, the Jacobian becomes $\left|J_{\Sigma^2}(\ell_{\phi,\rho})\right| = \partial_\phi \theta \cdot \partial_\rho \eta$. Differentiation yields

$$\partial_\phi \theta = \frac{\frac{\sigma_1}{\sigma_2}\sec^2\phi}{1+\frac{\sigma_1^2}{\sigma_2^2}\tan^2\phi} = \frac{\sigma_1\sigma_2}{\sigma_1^2\sin^2\phi+\sigma_2^2\cos^2\phi}, \tag{3.57}$$

$$\partial_\rho \eta = \sigma_1\cos\phi\cdot\cos\theta + \sigma_2\sin\phi\cdot\sin\theta. \tag{3.58}$$

In the expression for $\partial_\rho \eta$ we insert θ from (3.55) and use that $\sin(\arctan(x)) = x/\sqrt{1+x^2}$ and that $\cos(\arctan(x)) = 1/\sqrt{1+x^2}$ to obtain

$$\partial_\rho \eta = \frac{\sigma_1\cos\phi + \sigma_2\sin\phi\,\frac{\sigma_1}{\sigma_2}\tan\phi}{\sqrt{1+\frac{\sigma_1^2}{\sigma_2^2}\tan^2\phi}} = \frac{\sigma_1\sigma_2}{\sqrt{\sigma_1^2\sin^2\phi+\sigma_2^2\cos^2\phi}}. \tag{3.59}$$

If $\nu = (\nu_x, \nu_y)^T$ is a unit vector along the line $\ell_{\phi,\rho} = \ell_{\nu,\rho}$ then

$$\left|J_{\Sigma^2}(\ell_{\nu,\rho})\right| = \frac{\sigma_1^2\sigma_2^2}{\left(\sigma_1^2\sin^2\phi+\sigma_2^2\cos^2\phi\right)^{3/2}} = \frac{\sigma_1^2\sigma_2^2}{\left(\sigma_1^2\nu_x^2+\sigma_2^2\nu_y^2\right)^{3/2}} = \frac{\det\Sigma^2}{(\nu^T\cdot\Sigma^2\cdot\nu)^{3/2}}. \tag{3.60}$$

We are interested in the Jacobian of the whole transformation $J_{\Sigma^2}(U^T\ell_{\nu,\rho})$, so all that is left to do is insert $U^T\ell_{\nu,\rho}$ to obtain

$$\left|J_M(\ell_{\nu,\rho})\right| = \left|J_{\Sigma^2}(U^T\ell_{\nu,\rho})\right| = \frac{\det M}{\left(\nu^T U\cdot\Sigma^2\cdot U^T\nu\right)^{3/2}} = \frac{\det M}{\left(\nu^T\cdot M\cdot\nu\right)^{3/2}} \tag{3.61}$$

We have now proved that for a constant metric tensor M, the length of the differentiable curve C with regards to this tensor can be calculated as

$$|C|_M = \int_C \sqrt{\langle\dot{C}, M\dot{C}\rangle}\,dt = \int_{\mathcal{L}} \#(\ell_{\nu,\rho}\cap C)\frac{\det M}{\left(\nu^T\cdot M\cdot\nu\right)^{3/2}}\,d\mathcal{L}(\ell_{\nu,\rho}). \tag{3.62}$$

Further we argue that the similar formula in (3.45) holds for a non-constant but continuous metric tensor $M(x)$. By partitioning the domain into disjoint sets U_i such that $\Omega = \cup_i U_i$, we make a piecewise constant approximation $M_\pi(x)$ such that if $x \in U_i$ then $M_\pi(x) = M(x_i)$ for some fixed $x_i \in U_i$. We then approximate (3.62) by

$$|C|_{M_\pi} = \sum_i \int_{\mathcal{L}} \#(\ell_{\nu,\rho}\cap C\cap U_i)\,w_i(\nu)\,d\mathcal{L}(\ell_{\nu,\rho}) \tag{3.63}$$

where w_i is the weight-function used in the set U_i, that is,

$$w_i(\nu) = \frac{\det M(x_i)}{\left(\nu^T\cdot M(x_i)\cdot\nu\right)^{3/2}}. \tag{3.64}$$

We further simplify the approximation by introducing the global weight-function $w_\pi(\nu, x)$ which is equal to $w_i(\nu)$ when $x \in U_i$. It can be written as

$$w_\pi(\nu, x) = \frac{\det M_\pi(x)}{\left(\nu^T \cdot M_\pi(x) \cdot \nu\right)^{3/2}}. \tag{3.65}$$

Using this weight in (3.63) we can get rid of the sum over the partition i and form a sum of all intersection point of C and the line $\ell_{\nu,\rho}$ currently being integrated over. The approximation becomes

$$\begin{aligned} |C|_{M_\pi} &= \sum_i \int_{\mathcal{L}} \sum_{x \in \ell_{\nu,\rho} \cap C \cap U_i} w_\pi(\nu, x)\, d\mathcal{L}(\ell_{\nu,\rho}) \\ &= \int_{\mathcal{L}} \sum_{x \in \ell_{\nu,\rho} \cap C} w_\pi(\nu, x)\, d\mathcal{L}(\ell_{\nu,\rho}). \end{aligned} \tag{3.66}$$

Now it only remains to show that the left- and right-hand side of (3.66) converges to the left- and right-hand side of (3.45).

As our partition π is refined, the weight $w_\pi(x)$ converges pointwise to the continuously varying weight

$$w(\nu, x) = \frac{\det M(x)}{\left(\nu^T \cdot M(x) \cdot \nu\right)^{3/2}} \tag{3.67}$$

found in (3.45).

Recall from (3.44) that the left-hand side is calculated as

$$|C|_{M_\pi} = \int_C \left|\dot{C}(t)\right|_{M_\pi} dt = \int_C \sqrt{\dot{C}(t)^T M_\pi(C(t)) \dot{C}(t)}\, dt. \tag{3.68}$$

We know that $M_\pi(x)$ converges pointwise to $M(x)$, and thus $|\dot{C}(t)|_{M_\pi}$ converges pointwise to $|\dot{C}(t)|_M$. We have assumed bounds on the eigenvalues of $M(x)$ such that, according to the Rayleigh principle

$$K \geq \lambda_1 = \max_\xi \frac{\xi^T M_\pi(x) \xi}{\xi^T \xi} \tag{3.69}$$

and therefore we have the bound

$$\xi^T M_\pi(x) \xi \leq K \left\|\xi\right\|^2, \quad \forall \xi. \tag{3.70}$$

Thus the integrand of (3.68) is bounded by $g(t) = (K \cdot \dot{C}(t)^T \dot{C}(t))^{1/2}$. We know that $g(t)$ is integrable as its integral is exactly $\sqrt{K}\,|C|$ and we have assumed that the curve is of finite length. This means we can apply Lebesgue's dominated convergence theorem to see that $|C|_{M_\pi} \to |C|_M$.

We apply the same theorem to show that the right-hand side of (3.66) converges. Recall the definition of w_π in (3.65). The numerator is equal to $\sigma_1^2\sigma_2^2 = \lambda_1\lambda_2$ and is by assumption bounded from above by K^2.

Next we need to bound $\nu^T M_\pi(x)\nu$ away from zero. According to the Rayleigh principle

$$\lambda_2 = \min_{\|\xi\|=1} \xi^T M_\pi(x)\xi \tag{3.71}$$

and thus $\nu^T M_\pi(x)\nu \geq \lambda_2 \geq k$. The weight function w_π is then bounded such that

$$\sum_{x\in\ell_{\nu,\rho}\cap C} w_\pi(\nu,x) \leq \sum_{x\in\ell_{\nu,\rho}\cap C} \frac{K^2}{k^{3/2}} = \frac{K^2}{k^{3/2}} \cdot \#(\ell_{\nu,\rho}\cap C) =: g(\ell_{\nu,\rho}). \tag{3.72}$$

This is integrable following from the Euclidean Cauchy–Crofton formula in Theorem 3.7 and the fact that we assumed C to be of finite length:

$$\int_{\mathcal{L}} g(\ell_{\nu,\rho})\, d\mathcal{L}(\ell_{\nu,\rho}) = \frac{K^2}{k^{3/2}}\,|C| < \infty. \tag{3.73}$$

Thus we can apply the dominated convergence theorem again and conclude that

$$\int_{\mathcal{L}} \sum_{x\in\ell_{\nu,\rho}\cap C} w_\pi(\nu,x)\, d\mathcal{L}(\ell_{\nu,\rho}) \to \int_{\mathcal{L}} \sum_{x\in\ell_{\nu,\rho}\cap C} w(\nu,x)\, d\mathcal{L}(\ell_{\nu,\rho}) \tag{3.74}$$

which—as both sides of the equality in (3.66) have been shown to converge—leaves us with what we wanted to prove

$$|C|_M = \int_{\mathcal{L}} \sum_{x\in\ell_{\nu,\rho}\cap C} \frac{\det M(x)}{2\left(\nu^T\cdot M(x)\cdot\nu\right)^{3/2}}\, d\mathcal{L}(\ell_{\nu,\rho}). \tag{3.75}$$

□

With the anisotropic coarea formula in Theorem 3.4 we have a way to calculate the anisotropic total variation by integrating the anisotropic perimeter of each level set of the image. We will now see how the anisotropic Cauchy–Crofton formula can help us calculate the perimeters of the level sets. In the Euclidean case, which here would amount to setting the anisotropy tensor A equal to the identity matrix I, the perimeter coincides nicely with the length of the boundary curve, assuming some regularity for the boundary. In the general case we need to be more careful. As can be seen in (3.41), the anisotropic perimeter is calculated by integrating the norm of the *normal vector* around the boundary, while the anisotropic curve length in (3.44) is the integral of the norm of the *tangent vector* of the curve. Thus a 90° rotation separates the two.

If P is a 90° rotation matrix we have

$$\begin{aligned}\mathrm{Per}_A(U;\Omega) &= \int_{\partial U} \sqrt{\langle \nu_{\partial U}, A(x)\nu_{\partial U}\rangle}\, dt \\ &= \int_{\partial U} \sqrt{\langle P\nu_{\partial U}, PA(x)P^T P\nu_{\partial U}\rangle}\, dt.\end{aligned} \tag{3.76}$$

We simplify the equation by defining the metric tensor $M(x) = PA(x)P^T$ and letting $\gamma = \partial U \cap \Omega$ be an arclength parametrization of the boundary of U that does not overlap with the boundary of Ω

$$\mathrm{Per}_A(U;\Omega) = \int_\gamma \sqrt{\langle \dot\gamma, M(x)\,\dot\gamma\rangle}\, dt. \tag{3.77}$$

Now we make sure that all the assumptions of the anisotropic Cauchy–Crofton formula in Theorem 3.8 are fulfilled so that it can be applied to the curve length integral we have constructed in (3.77).

The structure tensor is constructed as described in Section 3.1.1

$$S_\rho(x) = \big(K_\rho * (\nabla f_\sigma \otimes \nabla f_\sigma)\big)(x). \tag{3.78}$$

Because of the convolutions with the Gaussian function, this is a smooth map from $\bar\Omega$ to $\mathbb{R}^{2\times 2}$. As we can see in (3.6), the eigenvalues depend continuously on the coefficients of the elements in the structure tensor $S_\rho(x)$. The extreme value theorem states that a continuous real-valued function on a nonempty compact space is bounded above. Thus the eigenvalues $\lambda_1 \geq \lambda_2$ of $S_\rho(x)$ are bounded above. Moreover, by the construction in (3.10), there exists uniform bound k such that the smallest eigenvalue σ_1 of the anisotropy tensor $A(x)$ is bounded away from zero, as

$$\sigma_1 = \left(1 + \frac{(\lambda_1 - \lambda_2)^2}{\omega^2}\right)^{-1} \geq \left(1 + \frac{\lambda_1^2}{\omega^2}\right)^{-1} \geq k > 0. \tag{3.79}$$

Hence our metric tensor $M(x) = PA(x)P^T$ is continuous and positive definite with bounded eigenvalues $k \leq \sigma_1 \leq \sigma_2 \leq K = 1$ and thus the curve length calculation in (3.77) fulfills all the assumptions of the anisotropic Cauchy–Crofton formula in Theorem 3.8. Hence we can apply the formula to calculate the perimeter in (3.77) as

$$\mathrm{Per}_A(U;\Omega) = \int_{\mathcal{L}} \sum_{x \in \ell_{\nu,\rho} \cap \gamma} \frac{\det M(x)}{2\left(\nu^T \cdot M(x) \cdot \nu\right)^{3/2}}\, d\mathcal{L}(\ell_{\nu,\rho})\, ds, \tag{3.80}$$

where $\gamma = \partial U \cap \Omega$. Note that P does not affect the determinant, i.e. $\det A = \det PAP^T = \det M$, and from our decomposition in (3.10) we see that the transformation $PAP^T \to M$ actually amounts to switching the two eigenvalues σ_1 and σ_2 in Σ.

This concludes the treatment of the continuous problem. We have seen how the anisotropic coarea formula in Theorem 3.4 allows us to calculate the anisotropic total variation as an integral of the perimeter of all the level sets. Through the anisotropic Cauchy–Crofton formula in Theorem 3.8 these perimeters are calculated by an integral over the set of all lines. We are then left with the functional

$$F(u) = \int_\Omega (u-f)^2 + \beta \operatorname{TV}_A(u), \tag{3.81}$$

where

$$\operatorname{TV}_A(u) = \int_{-\infty}^{\infty} \int_{\mathcal{L}} \sum_{x \in \ell_{\nu,\rho} \cap \gamma_s} \frac{\det M(x)}{2\left(\nu^T \cdot M(x) \cdot \nu\right)^{3/2}} \, d\mathcal{L}(\ell_{\nu,\rho}) \, ds, \tag{3.82}$$

and $\gamma_s = \partial\{u > s\} \cap \Omega$. Within the restrictions that these theorems put on the tensor $M(x)$, we have chosen a construction where one eigenvalue is always 1, while the other varies from 1 in smooth areas towards 0 around edges, with the corresponding eigenvector perpendicular to the edge.

Chapter 4

Discrete formulation

The whole transformation from the initial functional in (3.4), through the anisotropic coarea formula and the Cauchy–Crofton formula was motivated by the discrete formulation which will be described here. After discretizing the functional, we will see how a graph cut approach can be used to find a global minimizer in polynomial time.

4.1 Discretization

Assume that our discrete images are given on a uniform grid $\mathcal{G}$, where each grid point is called a pixel. The image is a function giving each pixel a value in the set of levels $\mathcal{P} = \{0, \dots, L-1\}$, such that $u : \mathcal{G} \to \mathcal{P}$. This is a reasonable assumption for digital grayscale images. Thus, when discretizing the functional in (3.81), we have to consider that our images now have both discrete domain and co-domain.

The integrals in (3.81) will be approximated by discrete sums. First the fidelity term is discretized without too much trouble, while with the regularization term, there is more choice as to how to discretize the set of lines $\mathcal{L}$. In the end we will verify that our discretization is consistent with the continuous functional.

4.1.1 Fidelity term

Since it is not affected by our introduction of the anisotropy tensor, the fidelity term can be discretized as in my project work [1]. For some pixel position $x \in \mathcal{G}$ and some level value $k \in \mathcal{P}$, we define the following function

$$N_x(k) = |k - f_x|^2 \tag{4.1}$$

which is the value of the fidelity term if we were to give u_x a value of k. This allows us write

$$\int_\Omega |u-f|^2\,dx \approx \sum_{x\in\mathcal{G}} |u_x - f_x|^2\,\Delta x = \sum_{x\in\mathcal{G}} N_x(u_x)\Delta x. \tag{4.2}$$

The reason we introduce the function $N_x(k)$ is that we want to apply the following decomposition formula, which holds for any function $F(k)$ taking values $k \in \mathcal{P}$:

$$\begin{aligned} F(k) &= \sum_{\lambda=0}^{k-1} (F(\lambda+1) - F(\lambda)) + F(0) \\ &= \sum_{\lambda=0}^{L-2} (F(\lambda+1) - F(\lambda)) I(\lambda < k) + F(0), \end{aligned} \tag{4.3}$$

where $I(x)$ is the indicator function that takes the value 1 if x is true, and 0 if x is false. Since $I(\lambda < u_x) = u_x^\lambda$ we rewrite (4.2) and obtain

$$\sum_{x\in\mathcal{G}} |u_x - f_x|^2 = \sum_{x\in\mathcal{G}} N_x(u_x) = \sum_{\lambda=0}^{L-2}\sum_{x\in\mathcal{G}} (N_x(\lambda+1) - N_x(\lambda))\, u_x^\lambda + N_x(0). \tag{4.4}$$

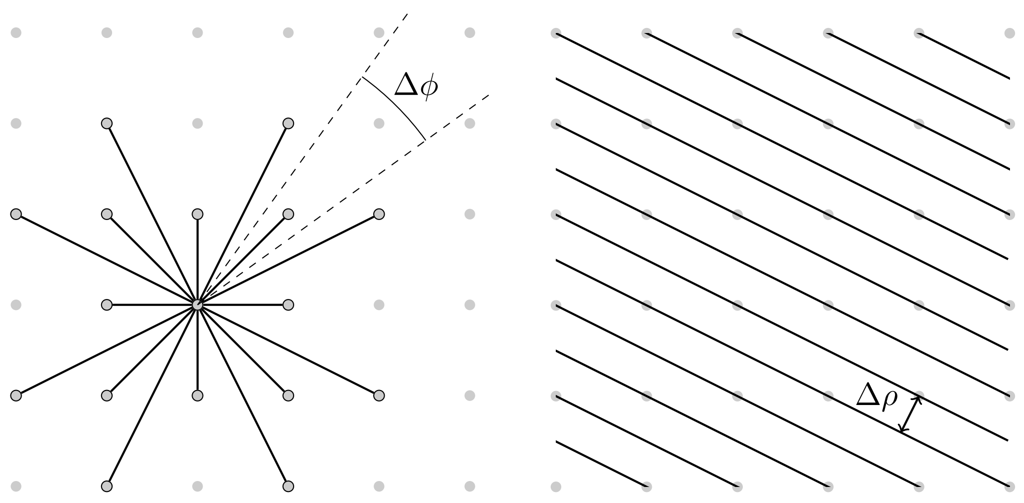
As our domain is discretized uniformly, we drop the constant Δx, and absorb it into our parameter β of (3.81). Note that since our image takes values in $\mathcal{P} = \{0,\dots,L-1\}$, the thresholded image u^{L-1} is equal to zero everywhere.

4.1.2 Regularization term

Discretizing the regularization term is more challenging. We introduce the discrete levels to get

$$\int_{-\infty}^{\infty} \mathrm{Per}_A(\{u>\lambda\};\Omega)\,d\lambda \approx \sum_{\lambda=0}^{L-2} \mathrm{Per}_A(\{u>\lambda\};\Omega)\,\Delta\lambda. \tag{4.5}$$

As with the Δx difference, we can absorb the $\Delta\lambda$ difference into the β parameter of (3.81). The perimeter is then calculated using a discretized version of the Cauchy–Crofton formula introduced in Theorem 3.8. Again, we stop the sum at $L-2$ since the level set $\{u > L-1\}$ is empty and has zero perimeter.



(a) The discrete set of lines $\mathcal{L}_D$ visualized as a neighborhood.

(b) One family of lines having the same ϕ parameter.

Figure 4.1: The set of lines $\mathcal{L}$ is discretized to $\mathcal{L}_D$ where each line belongs to a family given by ϕ, the angle parameter.

Discrete anisotropic Cauchy–Crofton formula

By approximating the integral Theorem 3.8 by a discrete sum we obtain the approximation

$$\begin{aligned}|C|_M &= \int_{\mathcal{L}} \sum_{x \in \ell_{\nu,\rho} \cap C} \frac{\det M(x)}{2\left(\nu^T \cdot M(x) \cdot \nu\right)^{3/2}} \, d\mathcal{L}(\ell_{\nu,\rho}) \\ &\approx \sum_{\ell_{\nu,\rho} \in \mathcal{L}_D} \sum_{x \in \ell_{\nu,\rho} \cap C} \frac{\det M(x)}{2\left(\nu^T \cdot M(x) \cdot \nu\right)^{3/2}} \, \Delta\ell_{\nu,\rho} \\ &= \sum_{\nu} \sum_{\rho} \sum_{x \in \ell_{\nu,\rho} \cap C} \frac{\det M(x)}{2\left(\nu^T \cdot M(x) \cdot \nu\right)^{3/2}} \, \Delta\rho \, \Delta\nu.\end{aligned} \tag{4.6}$$

The set of lines $\mathcal{L}$ has been discretized into the set $\mathcal{L}_D$. Note that C is still a differentiable curve, not yet discretized. Being a difference in the ρ parameter of our line discretization in Figure 3.5, the difference $\Delta\rho$ represents the distance from one line to the next in a line family as shown in Figure 4.1b, and thus depends on the angle ϕ considered. The difference $\Delta\phi$ is taken to be the average of the distance to the two neighboring line families as shown in Figure 4.1a, and thus also depends on ϕ.

The choice of our discrete set of lines $\mathcal{L}_D$ is important, as it will decide the accuracy of our approximation. We need some sensible restrictions on the set

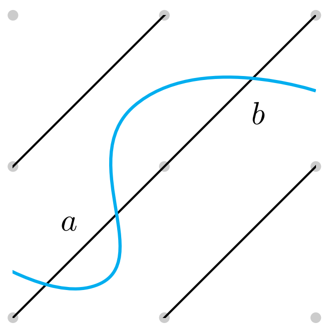


Figure 4.2: Here our intersection approximation would not be correct, as only the intersection with edge b is counted in (4.15), even though the curve intersects edge a twice.

$\mathcal{L}_D$ to simplify the further discussion. All lines intersect at least two grid points, and from the periodicity of our grid they thus intersect an infinite number of grid points. This puts some restrictions on the angles we can choose. For each angle included, we include all possible lines of that family, meaning there are no grid points without a line of that family intersecting it.

The set of lines can then be represented by the neighborhood of a pixel as shown in Figure 4.1a. We write $\mathcal{N}(x)$ for the neighborhood of grid point x. Extending the edges shown in Figure 4.1a gives all lines going through the point considered. Figure 4.1b shows all lines of a given family, i.e. lines having the same angle parameter ϕ.

Thus not only have we discretized the set of lines, but each line is made up of edges going from one grid point to the next. We will denote such an edge by e or e_{ab} when its endpoints are $a, b \in \mathcal{G}$. Thus we rewrite the discretization of (4.6), and sum over all the edges in the discretization $\mathcal{L}_D$ to obtain

$$|C|_M \approx \sum_e \sum_{x \in e \cap C} \frac{\det M(x) \, \|e\|^3}{2 \left(e^T \cdot M(x) \cdot e\right)^{3/2}} \, \Delta\phi \, \Delta\rho. \tag{4.7}$$

This is beginning to look like something we can calculate. One difficulty is finding the intersections $e \cap C$. The exact calculations of these points will not fit into our graph cut framework later, and thus for an edge e we will only consider the question of "did e cross C or not?" This amounts to checking whether the terminals of e lie on different sides of the curve C. This approximation is exact for zero or one intersection points, but will, as we see in Figure 4.2, be wrong when we have more.

The second difficulty is that in the discrete setting, we will only have an approximation of the metric tensor $M(x)$ for each point $x \in \mathcal{G}$, and it is thus not available for arbitrary intersection points in Ω. For an intersection of edge e we will utilize the average of the tensor in the two endpoints of the edge. Thus for an

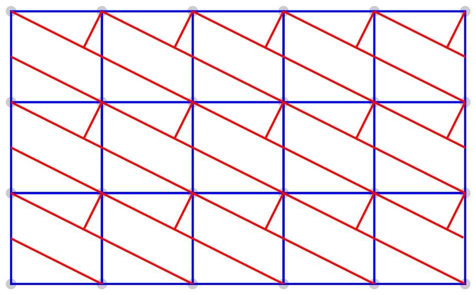


Figure 4.3: A visual argument showing that $\delta^2 = \Delta\rho\,\|e\|$. If extended to the whole plane, there will be the same amount of blue squares as red rectangles as each grid point is the upper left corner of both a blue and a red rectangle. Thus their areas must be equal.

intersection point x somewhere on the edge e_{ab}, we approximate the metric tensor by

$$M(x) \approx M(e_{ab}) = \frac{M(a) + M(b)}{2}, \tag{4.8}$$

the component-wise average of the tensors in the two end points of the edge. Recall that we have already done some spatial smoothing of the structure tensor in (3.5) corresponding to the *integration scale* ρ, and thus we expect the tensors $M(a)$ and $M(b)$ to be similar for edges e of reasonably short length.

We also remark that using the Rayleigh principle, it is easy to conclude that the eigenvalues of the tensor approximation $M(e_{ab})$ are bounded below and above by the smallest and largest eigenvalues of $M(a)$ and $M(b)$.

We have now almost arrived at our final curve length approximation, but we need a way to calculate the inter-line distance $\Delta\rho$ which will be provided by the following lemma.

Lemma 4.1. *For each family of lines given by an angle parameter ϕ in the uniform grid of size δ we have the relation*

$$\delta^2 = \|e\|\,\Delta\rho. \tag{4.9}$$

Proof. Consider a line ℓ intersecting the point in the grid given by the indices $(p, q) \in \mathbb{Z}^2$. The distance $\Delta\rho$ from this line ℓ to the neighboring lines can then be calculated as a minimum over the distance to all other grid points.

The lines are split into edges $e = (\delta s, \delta t)^T$ where $s, t \in \mathbb{Z}$ are coprime such that e does not intersect any other grid points than its two endpoints.

We then calculate the minimal distance to a grid point not on the line ℓ as

$$\begin{aligned}\Delta\rho &= \min_{(p',q')\in\mathcal{G}\setminus\ell}\left\{\left\langle \delta[p-p',q-q'], \frac{e^{\perp}}{\|e^{\perp}\|}\right\rangle\right\} \\ &= \min_{(p',q')\in\mathcal{G}\setminus\ell}\left\{\delta^2\cdot\frac{t(p-p')-s(q-q')}{\|e\|}\right\}.\end{aligned} \tag{4.10}$$

Since s and t are coprime, there exists $a, b \in \mathbb{Z}$ such that $at - bs = 1$, and since the $\Delta\rho$ cannot be zero, we obtain

$$\Delta\rho = \frac{\delta^2}{\|e\|}. \tag{4.11}$$

A visual argument for the same result can be seen in Figure 4.3. □

Inserting $\Delta\rho = \delta^2/\|e\|$ and the tensor approximation of (4.8) into the curve length approximation of (4.7) we obtain

$$|C|_M \approx \sum_{e\cap C} \frac{\det M(e)\,\|e\|^2\,\delta^2\,\Delta\phi}{2\left(e^T\cdot M(e)\cdot e\right)^{3/2}}, \tag{4.12}$$

where the sum is over all edges crossing the curve.

The curve length we initially wanted to calculate was the level set perimeter $\mathrm{Per}_A(\{u>\lambda\};\Omega)$ in (4.5). To find edges that crosses this boundary curve, we identify the edges that have one terminal inside the level set, and the other outside. Thus we rewrite the sum over $e\cap C$ such that

$$\mathrm{Per}_A(\{u>\lambda\};\Omega) \approx \sum_{e_{ab}} \left|u_a^\lambda - u_b^\lambda\right| \frac{\det M(e_{ab})\,\|e_{ab}\|^2\,\delta^2\,\Delta\phi}{2\left(e_{ab}^T\cdot M(e_{ab})\cdot e_{ab}\right)^{3/2}}. \tag{4.13}$$

The absolute value $\left|u_a^\lambda - u_b^\lambda\right|$ is 1 if one of a and b lie inside the level set and the other lies outside, and 0 otherwise. In other words the absolute value is one if e_{ab} crosses the perimeter of $\{u>\lambda\}$ an odd number of times, and zero otherwise.

Thus we have arrived at our final discretization, which takes the form

$$F(u) = \sum_\lambda\sum_x F_x^\lambda(u_x^\lambda) + \beta\sum_\lambda\sum_{(x,y)} F_{x,y}^\lambda(u_x^\lambda, u_y^\lambda) =: F^\lambda(u^\lambda), \tag{4.14}$$

$$\begin{aligned} F_x^\lambda(u_x^\lambda) &= \left(N_x(\lambda+1) - N_x(\lambda)\right)\cdot u_x^\lambda, \\ F_{x,y}^\lambda(u_x^\lambda, u_y^\lambda) &= \left|u_x^\lambda - u_y^\lambda\right| \frac{\det M(e_{xy})\,\|e_{xy}\|^2\,\delta^2\,\Delta\phi}{2\left(e_{xy}^T\cdot M(e_{xy})\cdot e_{xy}\right)^{3/2}}.\end{aligned} \tag{4.15}$$

Recall that $N_x(\lambda) = |\lambda - f_x|^2$.

If we minimize F_λ to obtain u^λ for each level separately, it is obvious that we will also minimize the sum over all F_λ. However, it is not guaranteed that the obtained thresholded images u^λ can be combined to make an output image u. They were defined as $u^\lambda = \chi_{u>\lambda}$, so we need them to be monotonically decreasing in increasing level values, i.e.

$$u_x^\lambda \geq u_x^\mu, \quad \forall \lambda \leq \mu, \quad \forall x \in \mathcal{G}. \tag{4.16}$$

Later we will present two graph cut algorithms that find the thresholded images minimizing each level, *while guaranteeing that they meet this requirement.*

Consistency

Consistency relates to whether a solution to the continuous problem fits in the discretized equation, in other words, whether the discretized equation approximates the continuous one.

It is obvious that the discretization of the fidelity term in (4.2) is consistent. The sum is a midpoint rule approximation of the integral. As the grid is refined and $\delta \to 0$ the sum will converge to the integral.

For the regularization term we will argue that for a differentiable curve C, the discretization of our domain Ω and the set of lines $\mathcal{L}$ gives a discrete Cauchy–Crofton formula that is consistent with the continuous one. We will show that for an increasingly refined discrete domain $\mathcal{G}$, there exists a choice for $\mathcal{L}_D$ that leads to a consistent Cauchy–Crofton formula. For convenience we will use a neighborhood representation of $\mathcal{L}_D$ similar to the one in Figure 4.1a.

If we consider the edges e of each family separately, the curve length approximation in (4.12) can be written

$$\begin{aligned} |C|_M &= \int_\nu \int_\rho \sum_{x \in \ell_{\nu,\rho} \cap C} \frac{\det M(x)}{2\left(\nu^T \cdot M(x) \cdot \nu\right)^{3/2}}\, d\rho\, d\nu \\ &\approx \sum_\nu \sum_\rho \sum_{e_{\nu,\rho} \cap C} \frac{\det M(e_{\nu,\rho}) \left\|e_{\nu,\rho}\right\|^3}{2\left(e_{\nu,\rho}^T \cdot M(e_{\nu,\rho}) \cdot e_{\nu,\rho}\right)^{3/2}}\, \Delta\rho\, \Delta\nu. \end{aligned} \tag{4.17}$$

As described in the construction of this formula, there are four main approximations used. Firstly there is the fact that we do not consider the actual intersection points, but only whether an edge crosses the curve or not. Secondly we have the tensor which is averaged as in (4.8). And then we have the discretizations of our two line parameters ν and ρ.

It is intuitive that if $\sup \|e\| \to 0$, the number of times the differentiable curve C can cross a given edge decreases. We will not prove convergence, but rather assume that the special cases where it might not work, are negligible.

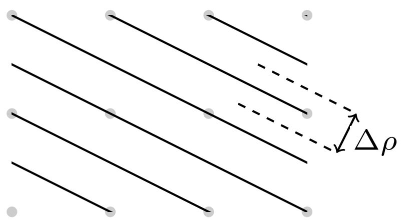


Figure 4.4: The discretization in the ρ dimension can be regarded as a midpoint rule approximation of the integral, since the difference $\Delta\rho$ the same for all lines in one line family.

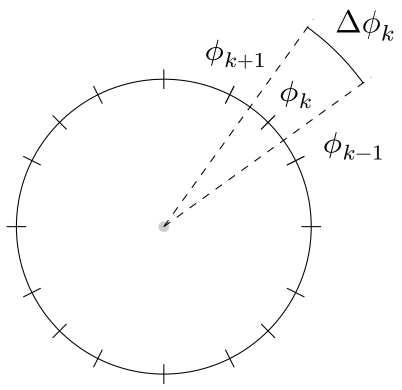


Figure 4.5: We showed that the maximal angle difference $\Delta\phi_k$ goes to zero. The discretization in the ϕ dimension can be viewed as a rectangle approximation rule of the integral, as the summand is evaluated at ϕ_k, somewhere inside the interval $\Delta\phi_k$.

Further, if $\sup \|e\| \to 0$ it is obvious that the tensor average in (4.8) converges to the tensor in the intersection point.

Consider now the discretization in ρ. For each ϕ parameter, our discretization in the ρ dimension can be regarded as a midpoint rule as shown in Figure 4.4. Thus if $\sup \Delta\rho \to 0$, this part of the discretization is consistent.

The discretization in the ϕ dimension can also be regarded as a version of the *rectangle method*, although not the midpoint rule. As shown in Figure 4.5, the circle is split into intervals

$$\left[\frac{\phi_{k-1}+\phi_k}{2}, \frac{\phi_k+\phi_{k+1}}{2}\right] \tag{4.18}$$

of length $\Delta\phi_k = (\phi_{k+1}+\phi_{k-1})/2$. The summand is evaluated at ϕ_k, somewhere inside the interval. Thus if $\sup \Delta\phi_k \to 0$, this discretization is also consistent.

To show that all these properties can be fulfilled, we look at a particular neighborhood stencil construction. Consider a square centered around a grid point with

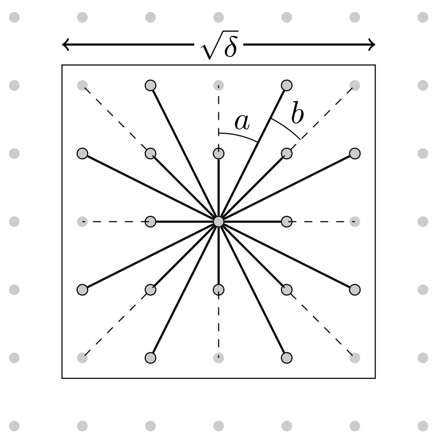


Figure 4.6: To show that we have a consistent discretization of the Cauchy–Crofton integral formula, we construct a discrete set of lines $\mathcal{L}_D$ such that the length of the edges $\|e\|$, the angle differences $\Delta\phi$ (here a and b) and the distance between lines $\Delta\rho$ goes to zero as $\delta \to 0$.

side lengths $\sqrt{\delta}$ as shown in Figure 4.6. As δ goes to zero, the size of this square will go to zero. Inside this square we can fit a square of $n^2 = \lfloor 1/\sqrt{\delta} \rfloor^2$ grid points. This means that the number of grid points along the outer edge of this square n goes to infinity.

We include all grid points inside the square in our neighborhood, except for multiple points that lie on the same line from the origin. If two or more grid points lie on the same line, we include only the one closest to the origin. This implies that for each grid point along the outer edge of this square, we include in our neighborhood a grid point having the same angle ϕ to the x-axis.

This construction can be seen in Figure 4.6 for $n = 5$. The maximal angle between two lines $\phi_k - \phi_{k-1}$ will always be when one of ϕ_k and ϕ_{k-1} is horizontal or vertical, shown in Figure 4.6 as the angle a. Thus the largest $\Delta\phi_k$ will be when $\phi_k = {}^{m\pi}/_{2}$ for $m \in \mathbb{Z}$, so around vertical and horizontal edges. The supremum can then be calculated to be

$$\sup \Delta\phi_k = 2 \cdot \sup \frac{\phi_{k+1} - \phi_k}{2} = \arctan \frac{1/n}{n/2} = \arctan \frac{2}{n^2} \to 0. \tag{4.19}$$

Further we see that the edge length will be bounded by half of the diagonal of the square such that

$$\|e\| \leq \sqrt{\delta/2} \to 0. \tag{4.20}$$

And finally we know from Lemma 4.1 that for each line family $\delta^2 = \Delta\rho \, \|e\|$ and $\|e\| \geq \delta$. Thus for the inter-line distance $\Delta\rho_k$ we have

$$\sup \Delta\rho_k = \sup \frac{\delta^2}{\|e\|} \leq \frac{\delta^2}{\delta} = \delta \to 0. \tag{4.21}$$

Hence the approximation has been shown to be equivalent to well-known, and consistent integral approximations, where the summand converges to the integrand, and the differences $\Delta\phi$ and $\Delta\rho$ go to zero. Thus the perimeter approximation in (4.5) is consistent with the continuous formulation in Theorem 3.8.

Note that as we will work with digital images with fixed resolutions, we do not really have the chance to refine our discretization. We do however have to take these things into account when creating our neighborhood stencil, to make sure that we get a reasonable approximation of the perimeter lengths.

4.2 Graph cut approach

The discretization we arrived at in (4.15) can be minimized using graph cuts. For each level λ, a minimum graph cut is found to produce the corresponding level set $\{u > \lambda\}$. These are then combined to form the final restored image u.

In this section we will look at how these graphs are constructed such that their minimum cuts correspond to the minimizers of the functional F^λ. The description is taken with some small adjustments from my project work [1], and is included here for completeness. An implementation of the described approach can be found in Appendix A.

4.2.1 Graphs

Using the notation of [18] we will denote a directed graph as $G = (V, E)$ where V is a finite set of vertices, and E is a binary relation on V. If $(u, v) \in E$ we say that there is an edge from u to v in the graph G.

We introduce the non-negative capacity function $c : V \times V \to [0, \infty)$. Only edges $(u, v) \in E$ can have a positive capacity $c(u, v) = q > 0$ and it means that it is possible to send a *flow* of maximum q units from u to v. For convenience we will let $c(u, v) = 0$ for any pair $(u, v) \notin E$, and we do not allow self-loops in our graph. When a directed graph G is equipped with capacity function c, one might call it a capacitated graph or a flow network, but as all our graphs will be capacitated from this point, we will just call them graphs and we write $G = (V, E, c)$.

There are two special vertices in the graph, the source s and the sink t. Contrary to other vertices, which can neither produce nor receive excess flow, the source can produce and the sink can receive an unlimited amount of flow. The most basic problem in graph flow theory is the question of how much flow it is possible to send through the graph from the source to the sink.

What we seek in our final graph is a minimum s-t-cut, a “minimal” line through the graph that cuts a set of edges and divides the vertex set in two, separating the source from the sink.

Definition 4.2 (s-t-cut)**.** *Given a graph $G = (V, E, c)$, an s-t-cut (S, T) of G is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. The capacity of the cut is*

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u, v), \tag{4.22}$$

and a minimum s-t-cut is a cut whose capacity is minimum over all s-t-cuts.

Note that there might exist several minimum s-t-cuts in a graph, resulting in different partitions of V. This is why we need to verify later that the cuts we obtain result in stackable thresholded images.

4.2.2 Graph representable functionals

The next step is to find a way to construct a graph such that we can minimize the functional in (4.15) by finding a minimum s-t-cut. We will do this by creating small and simple graphs representing the separate summands of the functional. For these small graphs it will be easy to verify that the minimal cut also minimizes the corresponding part of the functional, and they can then be merged giving a graph representing the complete functional in (4.15).

First we need to establish the definition of a graph representable function, presented by Kolmogorov and Zabih in [19].

Definition 4.3 (Graph representable functions)**.** *A function $\mathcal{E}(x_1, \dots, x_n)$ of n binary variables is graph-representable if there exists a graph $G = (V, E, c)$ with terminals s and t, and a subset of vertices $V_0 = \{v_1, \dots, v_n\} \subseteq V - \{s, t\}$ such that, for any configuration $(x_1, \dots, x_n) \in \{0, 1\}^n$, the value of the energy $\mathcal{E}(x_1, \dots, x_n)$ is equal to a constant plus the cost of the minimum s-t-cut among all cuts $C = (S, T)$ where $x_i = 0 \Leftrightarrow v_i \in S$ and $x_i = 1 \Leftrightarrow v_i \in T$, $\forall\, 1 \leq i \leq n$.*

From this definition we see that if we have a graph-representable function $\mathcal{E}$ it is possible to find an exact global minimum of $\mathcal{E}$ by finding a minimal s-t-cut in a graph representing $\mathcal{E}$.

Furthermore Kolmogorov and Zabih present an important result concerning what kinds of functions are graph-representable.

Theorem 4.4 (Identification of graph representable functions)**.** *Given an energy function $\mathcal{E}$ of n binary variables of the form*

$$\mathcal{E}(x_1, \dots, x_n) = \sum_i \mathcal{E}^i(x_i) + \sum_{i<j} \mathcal{E}^{i,j}(x_i, x_j), \tag{4.23}$$

then $\mathcal{E}$ is graph representable if and only if each term $\mathcal{E}^{i,j}$ satisfies the inequality

$$\mathcal{E}^{i,j}(0,0) + \mathcal{E}^{i,j}(1,1) \leq \mathcal{E}^{i,j}(0,1) + \mathcal{E}^{i,j}(1,0). \tag{4.24}$$

This theorem will allow us to verify that our functional actually is graph representable.

Finally, the following theorem, proved by Kolmogorov and Zabih in [19], will be crucial in our graph construction.

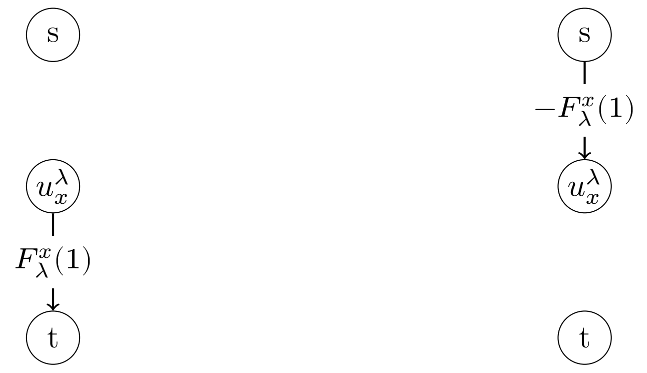
Theorem 4.5 (Additivity). *The sum of a finite number of graph-representable functions*

$$\mathcal{E}(x_1, \dots, x_n) = \sum_k \mathcal{E}^k(x_1, \dots, x_n), \tag{4.25}$$

each represented by a graph $G^k = (V^k, E^k, c^k)$*, is graph-representable by* $G = (V, E, c)$ *where* $V = \cup_k V^k$*,* $E = \cup_k E^k$ *and* $c(u, v) = \sum_k c^k(u, v)$*.*

It allows us to construct small graphs representing the different summands of our functional (4.15), before adding them together to create a final graph representing complete functional.

Note that when we apply this theorem later, we will assume that all the summands of (4.15) have the whole picture as their domain. It is unproblematic to extend $F^x_\lambda(u^\lambda_x)$ and $F^{x,y}(u^\lambda_x, u^\lambda_y)$ such that they take all the pixels their argument and then ignore all pixels except the ones they actually depend on.



(a) The graph when $F^x_\lambda(1) > 0$, with constant equal to 0.

(b) Graph when $F^x_\lambda(1) < 0$, with constant equal to $-F^x_\lambda(1)$.

Figure 4.7: The graph construction for the fidelity term $F^x_\lambda(u^\lambda_x)$. See Table 5.1 for an overview of the different possible cuts, and on why this construction works.

4.2.3 Graph construction

We will construct a graph in such a way that if a vertex u^λ_x ends up on the source side of the cut we set $u^\lambda_x = 1$, and if it ends up on the sink side we set $u^\lambda_x = 0$, as

Table 4.1: Each row represents one of the two possible values of $u_x^\lambda \in \{0,1\}$. The functional $F_\lambda^x(u_x^\lambda)$ and minimum cut obtaining this configuration is shown. The last two columns show the capacities of the cut for each of the two graph constructions in Figure 4.7. We verify that for each of the two graph constructions, the cut capacities are equal to the functional value, plus a constant.

u_x^λ	$F_\lambda^x(u_x^\lambda)$	Min. cut (S,T)	Graph (a) cut cap.	Graph (b) cut cap.
0	0	$(\{s\},\{u_x^\lambda,t\})$	0	$-F_\lambda^x(1)$
1	$F_\lambda^x(1)$	$(\{s,u_x^\lambda\},\{t\})$	$F_\lambda^x(1)$	0

in Definition 4.3. This is an arbitrary choice, but still something we have to keep in mind through the rest of the section.

We will now consider the two kinds of summands in the discrete functional (4.15). The fidelity term coming from our aim to keep the output image close to the original image, and the regularization term coming from our aim to minimize the total variation.

Fidelity term

The fidelity term of our discrete functional (4.15) simplifies to

$$F_\lambda^x(0) = 0 \tag{4.26}$$

$$F_\lambda^x(1) = N_x(\lambda+1) - N_x(\lambda) \tag{4.27}$$

where $F_\lambda^x(1)$ might be positive or negative depending on λ and the pixel value f_x.

Figure 4.7 shows how graphs can be constructed to represent this part of functional. The construction differs depending on whether $F_\lambda^x(1)$ is positive or negative. Table 5.1 shows how the cuts correspond to the values of u_x^λ and we can easily verify that the constructed graph actually represents the fidelity term.

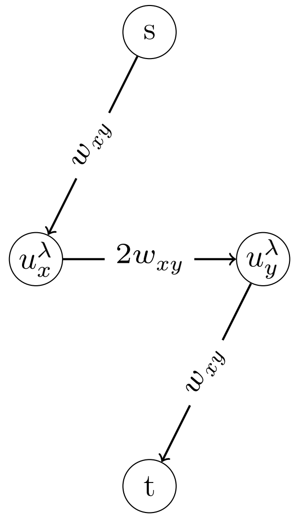
Regularization term

For our regularization term in (4.15) on the form

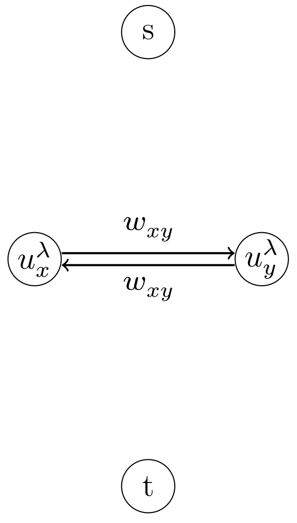
$$F^{x,y}(u_x^\lambda, u_y^\lambda) = w_{xy}\left|u_x^\lambda - u_y^\lambda\right| \tag{4.28}$$

we have

$$\begin{aligned} F^{x,y}(0,0) &= 0,\\ F^{x,y}(0,1) &= w_{xy},\\ F^{x,y}(1,0) &= w_{xy},\\ F^{x,y}(1,1) &= 0, \end{aligned} \tag{4.29}$$



(a) Representing $F^{x,y}(u_x^\lambda, u_y^\lambda)$ with a constant term of w_{xy}.



(b) Representing $F^{x,y}(u_x^\lambda, u_y^\lambda)$ with a constant term of 0.

Figure 4.8: Two alternative ways of constructing a graph representing the energy term $F^{x,y}(u_x^\lambda, u_y^\lambda)$. See Table 4.2 for an overview of the different possible configurations of $(u_x^\lambda, u_y^\lambda)$ and how they correspond to cuts through the graph.

and by Theorem 4.4 our functional is graph representable. In [19], Kolmogorov and Zabih presents a way to construct a graph for any graph representable function on the form shown in Theorem 4.4. Since the energies in (4.29) are especially simple, the construction and presentation is simplified.

Figure 4.8 shows two different ways of how a graph can be constructed to represent the regularization term. See Table 4.2 for an overview of how the values of $(u_x^\lambda, u_y^\lambda)$ corresponds to cuts in the graph.

Figure 4.9 shows a visualization of how the final graph might look with all its edges. The source will have a lot of outgoing edges, one for each pixel, while the sink has one incoming edge from each pixel. The vertices corresponding to the pixels are only connected to the source, the sink, and their neighboring pixels, so their edge degree is much smaller than for s and t.

Table 4.2: An overview of the possible configurations of the variables in the term $F^{x,y}(u_x^\lambda, u_y^\lambda)$. For each configuration the corresponding functional value and the cut yielding this configuration is shown. The last two columns show the capacities of the cut in the two alternative graph constructions shown in Figure 4.8. We verify that for each of the two graph constructions, the cut capacities are equal to the functional value, plus a constant.

$(u_x^\lambda, u_y^\lambda)$	$F^{x,y}(u_x^\lambda, u_y^\lambda)$	Min. cut (S,T)	Alt. (a) cut cap.	Alt. (b) cut cap.
$(0,0)$	0	$(\{s\}, \{u_x^\lambda, u_y^\lambda, t\})$	w_{xy}	0
$(0,1)$	w_{xy}	$(\{s, u_y^\lambda\}, \{u_x^\lambda, t\})$	$2w_{xy}$	w_{xy}
$(1,0)$	w_{xy}	$(\{s, u_x^\lambda\}, \{u_y^\lambda, t\})$	$2w_{xy}$	w_{xy}
$(1,1)$	0	$(\{s, u_x^\lambda, u_y^\lambda\}, \{t\})$	w_{xy}	0

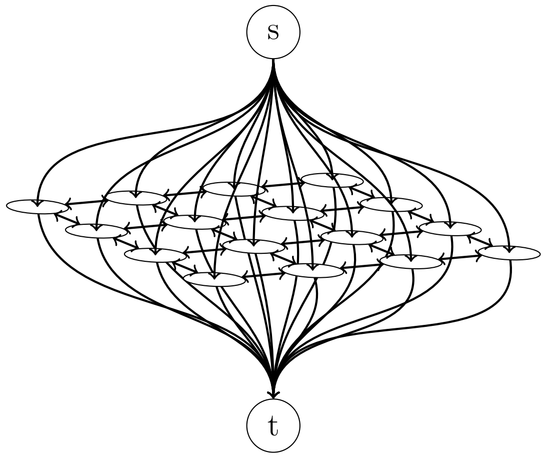


Figure 4.9: A visualization of the final graph, with edges between neighboring pixels, and edges connecting the source and sink to the rest of the graph. The edge capacities are not visualized, and of course some of them might be zero while others might be very high.

Chapter 5

Maximum flow

In the previous section we have seen how finding the minimum cut of carefully constructed graph can give us the thresholded image minimizing the functional for one level value λ. We will in this and the next section see how such a minimum cut can be found by sending flow through the graph and trying to identify the "bottleneck". This chapter, except for the description of the Boykov–Kolmogorov algorithm is taken with some adjustments from my project work [1] and is included here for completeness. Implementations of the push-relabel and Boykov–Kolmogorov algorithms can be found in Appendix A.

5.1 Flow graphs

We have already introduced capacities, and briefly mentioned the notion of flow as something limited by the capacity. In other words, flow is something we can send through our graph, but the capacity limits how much we can send along each edge. It is useful to imagine a water supply graph with pipes of different sizes and a water source and sink.

Formally, we introduce the *flow* as the function $f : V \times V \to [0, \infty)$. This function keeps count of how much flow we are sending through each edge of our graph and must satisfy the following two constraints

Capacity constraint: For all $u, v \in V, 0 \leq f(u,v) \leq c(u,v)$, i.e., for every pair of vertices, the flow is less than or equal to the capacity.

Flow conservation: For all $u \in V - \{s,t\}$

$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v), \tag{5.1}$$

i.e., for any vertex except the source and the sink, the flow into the vertex must be equal to the flow out of the vertex.

Note that we have defined f with all pairs of vertices as its domain, even though it is only non-zero on edges $(u, v) \in E$. This makes it easier to write sums as in the flow conservation constraint. We say that an edge (u, v) is *saturated* if $f(u, v) = c(u, v)$.

We define $|f|$ as the net amount of flow from the source to the sink in the graph. Because of the flow conservation constraint, this can be calculated as the net amount of flow going out of the source

$$|f| = \sum_{v \in V} f(s, v) - \sum_{u \in V} f(u, s). \tag{5.2}$$

Furthermore we denote the *net flow* across an s-t-cut $C = (S, T)$ as

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u). \tag{5.3}$$

Note how this definition differs from the capacity of a cut $c(S, T)$ in Definition 4.2. While the capacity of a cut represents how much flow it is maximally possible to send from S to T, the net flow across a cut represents the net amount of flow going across the cut, counting negatively the flow that goes back from T to S.

For any s-t-cut we have that

$$|f| = f(S, T). \tag{5.4}$$

This is quite intuitive given the flow conservation constraint, and a full chain of arguments can be found in [18].

Following from (5.3) and (5.4), we find that

$$\begin{aligned} |f| &= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\ &\leq \sum_{u \in S} \sum_{v \in T} f(u, v) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u, v) \\ &\leq c(S, T) \end{aligned} \tag{5.5}$$

for any s-t-cut $C = (S, T)$. A very central result in graph flow theory called the max-flow min-cut theorem will be presented later. It states that the inequality of (5.5) becomes an equality when f is a maximum flow for some cut $C = (S, T)$, and that all such cuts are minimum cuts.

But how does this help us? If we know the maximum flow value, and we have an s-t-cut with capacity equal to the maximum flow, we actually have a minimum cut. The question is then, how do we find a maximum flow, and how do we find a minimum cut?

5.2 Augmenting path algorithms

The family of augmenting flow algorithms represent a popular approach to the maximum flow problem. The idea is simply to look for paths from the source to the sink that can carry additional flow, so-called augmenting paths, and then send the maximum possible amount of flow along such a path. When no such path exists anymore, no more flow can be sent from the source to the sink, and a maximum flow has been reached.

5.2.1 Residual graph

When further discussing approaches to solving the maximum flow problem we will need the notion of a residual graph $G_f = (V_f, E_f, c_f)$, which is derived from the original graph G and contains the edges along which it is possible to send additional flow. This means that E_f contains the edges (u, v) from E where $f(u, v) < c(u, v)$. But that is not all; an important realization is that it is also possible to push flow *back* along an edge where the flow is already positive. In other words, sending flow from v to u by canceling some or all of the flow that is already going from u to v.

Thus the capacity function c_f of our residual graph becomes

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{5.6}$$

The vertices V_f of G_f are the same as the original graph G, while the edges E_f are taken to be all pairs of vertices (u, v) with $c_f(u, v) > 0$. Since we can in E_f at most have all the original edges, and their reversals, we have $\left|E_f\right| \leq 2\,|E|$.

Note that there is ambiguity in the definition of the residual graph in the case where the original graph contains anti-parallel edges. One could avoid this by defining $c_f(u, v) = f(v, u) + c(u, v) - f(u, v)$ instead, or as they do in [18], disallow anti-parallel edges altogether. In any case it is not something we will have to think about in the implementation, since we will not actually construct the residual graph.

With the residual graph defined, we are ready to formally present the max-flow min-cut theorem.

Theorem 5.1 (Max-flow min-cut theorem). *If f is a flow in a graph $G = (V, E, c)$ with source s and sink t, then the following statements are equivalent:*

1. *f is a maximum flow in G.*

2. *The residual graph G_f contains no augmenting paths.*

3. $|f| = c(S, T)$ *for some cut* (S, T) *of* G*.*

See [18] for a proof, and remark that because of the inequality in (5.5), the cut in Statement 3 is a minimum cut. The theorem does not tell us how to find such a cut, and there are multiple ways. One possibility is to take S to be all vertices reachable from the source in the residual graph and $T = V - S$.

Figure 5.1 shows a simple graph which already has five units flowing from s to t. The marked path is a possible augmenting path, and note that it follows an edge in E in the reverse direction, made possible by the construction of the residual graph just described.

5.2.2 Ford-Fulkerson

The Ford-Fulkerson algorithm is the most basic augmenting path algorithm, which can be extended to more advanced algorithms. It is stated in pseudocode in Algorithm 1, and the idea is to augment the flow along paths from s to t until it is no longer possible.

There are different ways to find augmenting paths, and a common choice is to do a breadth-first search from the source until the sink is found, as this will yield the shortest possible augmenting path. This version of the algorithm is called Edmonds-Karp and has a running time of $O(|V|\,|E|^2)$. See for example [18] for a description of the breadth-first search, and a formal proof of the running time of the Edmonds-Karp algorithm.

5.3 Other algorithms

There are many different maximum flow algorithms that fall into the augmenting path category, although we will see a different approach in the next section.

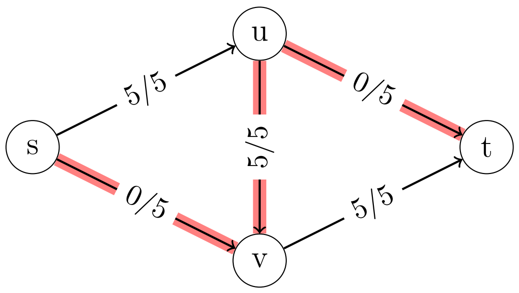


Figure 5.1: A graph with the flow and capacity of each edge shown as flow/capacity. The marked path is a valid augmenting path from s to t, and by increasing the flow along it, we will push flow back from v to u.

Algorithm 1 The Ford-Fulkerson max-flow algorithm

```
function FORD-FULKERSON($G$, $s$, $t$)
    while there exists a path $p$ from $s$ to $t$ in the residual graph $G_f$ do
        $\alpha \leftarrow \min\{c_f(u,v) : (u,v) \in p\}$
        for all $(u,v) \in p$ do
            if $(u,v) \in E$ then
                $f(u,v) \mathrel{+}= \Delta f$
            else
                $f(v,u) \mathrel{-}= \Delta f$                    ▷ Push flow back
            end if
        end for
    end while
end function
```

The algorithm of Dinitz, originally published in 1970, later improved on, and described by the original author in [20], is a variant of the augmenting path algorithm. It maintains a distance labeling $d(u)$ of the vertices $u \in V$ in the graph, where $d(u)$ is the shortest path from the s to u in the residual graph. This can be computed with a simple breadth-first search. The next step is to construct a *blocking flow* f', using only edges in $E'_f = \{(u,v) \in E_f : d(u) + 1 = d(v)\}$. The blocking flow is such that if we augment the flow f by f', there is no longer any paths from s to t following edges in E'_f. After the blocking flow f' has been found and added to f, the distance labels are recalculated, and the label of the sink will be increased by at least one.

Boykov and Kolmogorov present a variant of the augmenting path algorithm in [21], specialized for the kinds of graphs occurring in graphical applications. It will be presented in Section 5.5

5.4 Push–relabel algorithm

The push-relabel algorithm is a different approach to the maximum flow problem, presented by Goldberg and Tarjan in [22]. Unlike the augmenting flow algorithms, it does not maintain a valid flow f in the graph at all times, but still obtains a valid maximum flow when the algorithm terminates.

5.4.1 Preflow

Instead of maintaining a valid flow, we introduce the concept of a *preflow* by relaxing the flow conservation constraint from earlier. We allow positive excess in the vertices and the flow conservation constraint from before then becomes

Preflow conservation: For all $u \in V - \{s, t\}$

$$\sum_{v \in V} f(v,u) \geq \sum_{v \in V} f(u,v), \tag{5.7}$$

i.e., for any vertex except the source and the sink, the flow into the vertex must greater or equal to the flow out of the vertex.

As in most of the cited push-relabel literature, we define $N = |V|$, and for all vertices $u \in V$ we define the excess

$$e(u) = \sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v), \tag{5.8}$$

which represents the amount of flow which disappears in vertex u. Equivalent to the preflow conservation constraint is stating that $e(u) \geq 0$ for all vertices $u \in V - \{s, t\}$.

In addition to the flow, we maintain a height map $d : V \to \mathbb{N}$ that satisfies $d(t) = 0$, and for every edge (u, v) in the residual graph, i.e. every edge with $c_f(u, v) > 0$, we require that $d(u) \leq d(v) + 1$. For all vertices u, the label $d(u)$ will be a lower bound on the length from u to t in G_f which is why it is also often called a distance labeling.

A vertex u is *active* if $u \in V - \{s, t\}$, it has positive excess ($e(u) > 0$) and $d(u) < N$. These are the vertices we want to operate on to increase the preflow.

5.4.2 Basic operations

The algorithm performs two basic operations, the *push* and *relabel* operations, while always maintaining a valid preflow f and a valid distance labeling d.

The push procedure

The push procedure moves excess flow from an active vertex along an edge $(u, v) \in E_f$ for which $d(u) = d(v) + 1$, i.e. to a vertex with a smaller distance label. We call such edges *admissible*. See Algorithm 2 for a pseudocode implementation of the push operation.

Assuming that f is a valid preflow, it is easy to verify that the preflow f and labeling d remain valid after running the push procedure on some admissible edge (u, v).

The relabel procedure

The relabel procedure is our tool for changing the distance labels of the vertices. It changes the label of a vertex to the greatest possible value, which is one more

Algorithm 2 The push procedure of the Push-Relabel algorithm.

```
function PUSH($u$, $v$)
    $\Delta f \leftarrow \min(c_f(u,v), e(u))$
    if $(u,v) \in E$ then
        $f(u,v) \mathrel{+}= \Delta f$
    else
        $f(v,u) \mathrel{-}= \Delta f$                    ▷ Push flow back
    end if                         ▷ Excess $e(u)$ and $e(v)$ will also change
end function
```

than the lowest label among its neighbors in the residual graph. See Algorithm 3 for a pseudocode implementation.

Algorithm 3 The relabel procedure of the Push-Relabel algorithm

```
function RELABEL($u$)
    if there is a $v \in V$ such that $(u,v) \in E_f$ then
        $d(u) \leftarrow \min\{d(v),\ \forall v \in V : (u,v) \in E_f\} + 1$
    else
        $d(u) \leftarrow N$                    ▷ $u$ becomes inactive
    end if
end function
```

If d was a valid labeling before running the relabel procedure, then we still have $d(u) \leq d(v) + 1$ for all neighbors v of u in the residual graph, and d remains a valid labeling. The capacity constraint and preflow constraint remain satisfied assuming they were satisfied before the procedure was started.

5.4.3 Final algorithm

These basic procedures are then applied to active vertices and admissible edges until we obtain our minimum cut. We will see later that when there are no more active vertices, we can extract the minimum cut from the graph.

In the first phase of the algorithm we initialize a valid preflow and distance labeling by saturating all edges out of the source s, and then setting its distance label $d(s) = N$. We then apply the push and relabel procedures where applicable until there are no more active vertices and a maximum preflow is obtained.

A vertex u can only be successfully relabeled to obtain a new label if the outgoing edges of u in the residual graph have changed since the previous relabeling. This is why the push and relabel procedures often are combined into a *discharge* procedure as shown in Algorithm 4. When it is run on an active vertex u, we push as much as possible of the excess flow to other vertices before the vertex is relabeled.

Algorithm 4 The discharge procedure of the Push-Relabel algorithm

```
function DISCHARGE($u$)
    for all $v \in V$ such that $(u, v) \in E_f$ do
        if $c_f(u, v) > 0$ and $d(u) = d(v) + 1$ then PUSH($u$, $v$)
        end if
    end for
    if $e(u) > 0$ then RELABEL($u$)
    end if
end function
```

In the second phase of the algorithm this maximum preflow is turned into a maximum flow by returning excess flow which did not reach the sink from inside the graph back to the source. We can skip this part of the algorithm, as it is possible to identify a minimum cut as soon as the first phase is finished, and the following theorem allows us to do that.

Theorem 5.2 (Cut identification)**.** *Given a graph $G = (V, E, c)$, assume that the first phase of the push-relabel algorithm has terminated so that no more active vertices remain. Then there exists a $k \in \{1, \dots, N-1\}$ such that there is no vertex with label k. For every such k the vertex sets $S = \{u \in V : d(u) > k\}$ and $T = \{u \in V : d(u) < k\}$ define a minimum cut $C = (S, T)$ in G.*

Proof. There are N vertices, the source has label N and the sink has label 0, and the $N-2$ remaining vertices can not occupy all the $N-1$ labels in $\{1, \dots, N-1\}$, so there must exist an k as described.

There can be no edge $(u, v) \in E_f$ such that $u \in S$ and $v \in T$, as this would imply $k \leq d(u) - 1 \leq d(v) < k$. From the construction of E_f we now know that all edges in E from S to T are saturated, and all edges from T to S carry no flow. This implies the capacity of the cut is equal to the flow through the cut, i.e. $c(S, T) = f(S, T)$.

Since the first phase of the algorithm has terminated, there can be no active vertices, and therefore no excess in T, except for the sink. If all flow excess in vertices in S is returned to the source, we can apply the max-flow min-cut theorem to conclude that $C = (S, T)$ is a minimum s-t-cut, since the cut capacity is equal to the flow. □

We will see later that with the gap relabeling heuristic, there will always be a gap at label $k = N - 1$ such that we can construct our cut by taking $S = \{u \in V : d(u) \geq N\}$.

Note that the vertices in S are vertices earlier described as being on the source side of the cut, as no additional flow can go from these vertices to the sink.

5.4.4 Complexity

In their original article [22], Goldberg and Tarjan analyze the complexity of the push-relabel algorithm by considering the maximum number of basic operations we can possibly do before the algorithm terminates.

The number of relabelings is in $O(|V|^2)$ since every time the procedure is applicable to a vertex u, the label $d(u)$ increases by at least one.

The number of saturating pushes is in $O(|V|\,|E|)$. When a push along (u, v) is saturating, the label of v has to increase with at least 2 before a push can saturate the same edge (in the opposite direction). Since the number of relabelings of a single vertex is bounded by $|V|$, and we have $|E|$ edges, this gives the stated number of saturating pushes.

The number of non-saturating pushes is the most complicated to bound, and will also make up the asymptotic running time of the algorithm. The idea is to define

$$\phi = \sum_{u \text{ active}} d(u), \tag{5.9}$$

and look at how much this number changes throughout the algorithm. It starts at zero and ends at zero. Every non-saturating push from u to v makes ϕ decrease by at least one since it makes u inactive (but might activate v). The total increase in ϕ due to relabelings is less than $|V|^2$. A saturating push from u to v increases ϕ by at most $|V|$, since v might become active.

Even if ϕ is always increased by relabelings and saturating pushes, we can bound the number of non-saturating pushes by

$$|V|^2 + |V|\,\underbrace{c\,|V|\,|E|}_{\#(\text{saturating pushes})} \tag{5.10}$$

which means that in the general case, the algorithm has a complexity of $O(|E|\,|V|^2)$.

5.4.5 Vertex selection rules

Until now we have stated that the discharge procedure is run on active vertices until there are no more active vertices left. The choice of the order in which to discharge these active vertices remain, and multiple possibilities exist.

The «First In, First Out» (FIFO) approach is to always maintain a queue of active vertices. When the vertex from the beginning of the queue is discharged, other vertices might become active, and these are added at the end of the queue.

The original article of Goldberg and Tarjan [22] contains a proof that this selection rule gives a complexity of $O(|V|^3)$, which can be very good if you have a dense graph.

Using the highest level selection rule one always discharges the active vertex with the largest distance label. Goldberg and Tarjan state that this rule also gives a complexity of $O(|V|^3)$ while this bound is improved to $O(|V|^2\sqrt{|E|})$ in an article by Cheriyan and Maheshwari [23].

These are complexity bounds, and the actual running time of the algorithm, which can only be determined by implementing it and running it, varies a lot with the structure of the input graph.

Cherkassky and Goldberg describe the algorithm along with different selection rules, heuristics and their implementation in [24].

5.4.6 Heuristics

Different heuristics exist that can speed up the algorithm considerably. Being heuristics, they are not guaranteed to work, and might perform differently on different kinds of graphs. The most used heuristics are the gap and global relabeling heuristics, both aiming to reduce the total number of relabeling steps.

The gap relabeling heuristic aims to find a label k as in Theorem 5.2 such that no vertex has that label. From vertices u with $d(u) > k$, there are no unsaturated edges going to vertices with smaller distance labels, so no more flow can ever find its way from these vertices to the sink. These vertices are therefore given the label N and never considered again as they will never become active. Algorithm 5 shows a pseudocode representation of what is done once a gap k is found.

Algorithm 5 The gap procedure of the Push-Relabel algorithm

```
function GAP($k$)
    for all $u \in V$ such that $d(u) > k$ do
        $d(u) \leftarrow N$
    end for
end function
```

The preflow and capacity constraints are still valid after the gap relabeling procedure, as only the labels d are changed. For the labeling d one has to verify that $d(u) \leq d(v) + 1$ for every edge $(u, v) \in E_f$ in the residual network. If none, both, or only v is relabeled, this is trivial. It is not possible that only u would be relabeled, as this would imply that $d(u) \geq d(v) + 2$ which is not a valid initial labeling.

When running the push-relabel algorithm with the gap heuristic, we can be sure that there will never be a vertex u with label $d(u) = N - 1$ at the end of

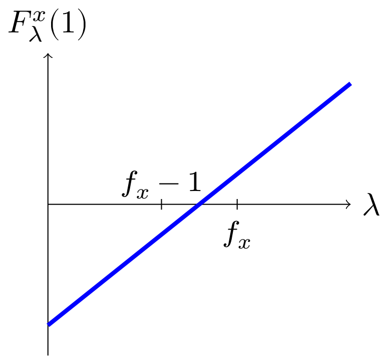


Figure 5.2: A sketch of how the fidelity energy term $F^x_\lambda(1)$ in (4.15) increases monotonically with λ.

the algorithm, i.e. we know that there will always be a gap at label $N-1$. This can be seen using the same reasoning as in Theorem 5.2, because if there was a vertex with label $N-1$, there would only be $N-3$ vertices possibly having labels in $\{1,\dots,N-2\}$, so a gap must exist somewhere in that interval. When using the gap relabeling heuristic, such a gap can not exist, so we can conclude that there is no vertex with label $N-1$.

Using Theorem 5.2 we can then conclude that the sets $S = \{u \in V : d(u) \geq N\}$ and $T = V - S$ form a minimum cut of the graph.

5.4.7 Parametric push-relabel algorithm

Now we have an algorithm for finding a minimum s-t-cut in a graph, so let's return to the graph constructed in Section 4.2.3. For every level $\lambda \in \{0,\dots,L-1\}$ we want to find a minimum s-t-cut which gives us the thresholded image u^λ. These can then hopefully be stacked together to form the final image u.

Graph reuse

Solving L separate minimum cut problems seems like a lot of work, but when using the push-relabel algorithm we will, if we do things in the right order, be able to reuse the graph when going from one label to the next.

Going back to the graph representations in Figure 4.7 and Figure 4.8 we know that only the capacity of edges from sub-graphs representing the fidelity term depend on our level parameter λ. From the expression in (4.26), visualized in Figure 5.2, we see that the energy term $F^x_\lambda(1)$ increases monotonically with increasing λ parameter. Let $u, v \in V - \{s,t\}$. Since the edges in Figure 4.7 now are the only ones depending on λ, the following is true for *decreasing* values of λ

Edges from s to u: As seen in Figure 4.7b the capacity of these edges will increase monotonically with decreasing λ parameter.

Edges from u to v: These edges do not depend on λ and will remain unchanged.

Edges from v to t: As seen in Figure 4.7a the capacity of these edges will decrease monotonically with decreasing λ parameter.

After running the push-relabel algorithm for $\lambda = k$, we are left with a graph $G = (V, E, c)$, a preflow f and a labeling d. To obtain the graph for $\lambda = k - 1$ we have to change the capacity of two different kinds of edges, and this is done in the following way to keep the capacity, preflow and labeling constraints satisfied.

Edges from s to u: The capacity $c(s, u)$ is increased, and the flow is set to be equal to the capacity $f(s, u) = c(s, u)$. The vertex u might have an increased excess $e(u)$, which might in turn make it active.

Edges from v to t: The capacity $c(v, t)$ is decreased. If it is decreased to a value below the current flow value, we set $f(v, t) = c(v, t)$ which will decrease the excess of the sink t, and increase the excess of v.

None of these actions will create new edges in the residual graph, and we do not change the labeling d, so the labeling constraints are also satisfied in the new graph.

Through this procedure we have easily created the graph for $\lambda = k - 1$, and the distance labels remain the same. As the distance labels always increase monotonically, we have a head start compared to the case where we reset the flow and labels.

Output image construction

We mentioned already in Chapter 4 that in order to be able to construct our output image u, the thresholded images u^λ would have to stack one on top of the other. Because of the reuse of the distance labels between the iterations of the push-relabel algorithm, we can guarantee that this is possible.

Consider two subsequent runs of the push-relabel algorithm, for labels λ and $\lambda - 1$ ending with distance labels d^λ and $d^{\lambda-1}$ respectively. We already know that the distance labels d are monotonically increasing. This means that the set $S = \{u \in V : d(u) \geq N\}$ is increasing in size, more precisely, we have the inclusion

$$\{u \in V : d^\lambda(u) \geq N\} \subseteq \{u \in V : d^{\lambda-1}(u) \geq N\}. \tag{5.11}$$

For a pixel $x \in S$ we will set $u_x^\lambda = 1$, which together with the previous inclusion property implies

$$u_x^\lambda \leq u_x^{\lambda-1} \tag{5.12}$$

for all $x \in \mathcal{G}$. Being equivalent with the inequality in (4.16), this means our algorithm produces stackable thresholded images u^λ.

We then construct our output image u by giving each pixel the value

$$u_x = \min\{\lambda \in \{0, \dots, L-1\} : u_x^\lambda = 0\}. \tag{5.13}$$

This marks the end of the description of the implemented algorithm, but we will further discuss some possible improvements.

5.4.8 Divide and conquer

The possibility of re-using the graph between separate level is a very nice property of the push-relabel algorithm, but there are further room for improvements. Consider one pixel x with value u_x, and imagine we only wanted to find the value of this pixel. One could go through all pixel values $\lambda \in (L-1, \dots, 0)$, and see when u_x^λ changes from 1 to 0, just as we do for all the pixels in the algorithm above. Ignoring graph re-use this would have us solve $O(L)$ maximum flow problems.

Improving on this we could employ the idea of binary search to find the value of u_x in only $O(\log_2 L)$ time. After finding one cut, we know whether u_x is above or below the current λ value, and by choosing λ as the midpoint of the current possible range of u_x, we can cut the search space in half for each iteration of the algorithm.

We can extend this idea to the problem of finding all pixel values. Instead of running the algorithm for successively decreasing values of λ, we choose some λ in the middle of the range $\{0, \dots, L-1\}$. The cut we obtain consists of two sets $S = \{u \in V : d(u) \geq N\}$ and $T = V - S$. We know that no more flow can be sent from S to T, even if we decrease the value of λ and adjust the capacities accordingly.

The idea is now that we have halved the possible λ interval for *all* pixels. We continue by considering the two sets S and T separately, and applying the algorithm recursively, at each time halving the λ interval until we have the value of every pixel.

Combining the divide and conquer approach with the parametric push-relabel algorithm is a bit problematic. For all pixels $x \in T$ we know that $u_x \leq \lambda$, and we can reuse the graph when decreasing λ. However for pixels $x \in S$, we seek to find $u_x > \lambda$, meaning we have to increase λ which does not allow graph re-use.

Goldfarb and Yin [25] have found that the divide and conquer approach only yields improved performance when using the L^2 norm in the fidelity term.

See [26], [27] and [25] for more information.

5.5 Boykov–Kolmogorov algorithm

Algorithm 6 The Boykov–Kolmogorov maximum flow algorithm

```
function BoykovKolmogorov($G$, $s$, $t$)
    $A := \{s, t\}$, $O := \emptyset$, $S = \{s\}$, $T = \{t\}$
    $s$.color = $S$, $t$.color = $T$
    while True do
        $e \leftarrow$ Grow($G$, $A$)
        if not $e$ then
            break
        end if
        Augment($G$, $s$, $t$, $e$, $O$)
        Adopt($G$, $s$, $t$, $O$)
    end while
end function
```

A maximum flow algorithm specialized for the type of graphs found in imaging applications is described by Boykov and Kolmogorov in [21]. Their algorithm is an augmenting path algorithm where the paths are found using trees that keep track of possible partial paths.

The idea of the algorithm is to always maintain the structure of two trees, one rooted in the source and one rooted in the sink, which are built up using only edges that can carry additional flow. The algorithm consists of different phases, one being the *grow* phase, where the trees are grown by adding additional edges. When an unsaturated edge connecting the two trees are found, the *augment* phase starts. In this phase, an augmenting path through the trees is recovered, and the flow is augmented. Some of the edges in the trees may then become saturated, and can no longer be a part of their tree. The vertices they connect to the tree become *orphans* which in the *adopt* phase are adopted back into their trees, or alternatively freed from their tree connection.

There is one tree denoted S with the source s as its root, and another tree denoted T with the sink t as its root. These trees are disjoint, and all edges in the tree S can carry flow towards the leafs of the tree, while all edges in T can carry flow towards the sink t.

A vertex can either be part of one of these trees, or be a free vertex, and we write

$$u.\text{color} = \begin{cases} S & \text{if } u \in S \\ T & \text{if } u \in T \\ \texttt{Free} & \text{otherwise.} \end{cases} \tag{5.14}$$

The vertices in the trees S and T are either active or passive, and we write $u \in A$ if u is active. The active vertices are those at the boundary of the tree, which can possibly connect to other vertices to grow the tree. The passive vertices are internal in the tree, and edges to their neighbors are either a part of the tree, or completely saturated.

The main loop of the algorithm is as shown in Algorithm 6. In the grow procedure, the trees are grown from their active vertices until a new path is found. The flow is then augmented along this path in the augment procedure, and orphan vertices might be created. These orphan vertices are then either joined back into their respective trees, or become free, in the adopt procedure.

Algorithm 7 The grow procedure of the Boykov–Kolmogorov maximum flow algorithm

```
function GROW($G$, $A$)
    while $|A| \neq 0$ do
        $u \leftarrow$ one node from $A$
        for all $v$ such that treeCap($u$, $v$) $> 0$ do
            if $v$.color = Free then
                $v$.color $\leftarrow u$.color
                $v$.parent $\leftarrow u$
                $A := A \cup \{v\}$
            else if $v$.color $\neq u$.color then
                return $e := (u, v)$
            end if
        end for
        Remove $u$ from $A$
    end while
    return False
end function
```

The grow procedure is shown in Algorithm 7. It goes through the set of active vertices A to try to expand the trees S and T. When considering an active vertex $u \in S$, we want to grow by finding vertices v such that $c_f(u, v) > 0$, while when considering an active vertex $u \in T$, we want to grow by finding vertices v such that $c_f(v, u) > 0$. This is why the treeCap is introduced which is defined as

$$\text{treeCap}(u, v) = \begin{cases} c_f(u, v) & \text{if } u \in S, \\ c_f(v, u) & \text{if } u \in T. \end{cases} \tag{5.15}$$

Thus if a non-saturated edge is found from an active vertex u to a free vertex v, then v is added to the tree of u. The tree structure is stored by keeping a parent attribute in each non-free node, such that for example v.parent $= u$.

If a non-saturated edge is found from an active vertex u to a vertex v in the other tree, the two trees connect, and we can return the edge e since we have an augmenting path. Note that if this happens, the vertex u is still active. It only becomes passive when all neighbors are considered without yielding an augmenting path.

Algorithm 8 The augment procedure of the Boykov–Kolmogorov maximum flow algorithm

```
function AUGMENT($G$, $s$, $t$, $e$, $O$)
    $p \leftarrow$ path from $s$ to $t$ through $e$            $\triangleright$ through the trees $S$ and $T$
    $\Delta f \leftarrow \min\{c_f(u,v) : (u,v) \in p\}$
    for all $(u,v) \in p$ do
        $f(u,v) \mathrel{+}= \Delta f$
        $f(v,u) \mathrel{-}= \Delta f$
        if $c_f(u,v) = 0$ then
            if $u$.color $= v$.color $= S$ then
                $v$.parent = Null
                $O := O \cup \{v\}$
            else if $u$.color $= v$.color $= T$ then
                $u$.parent = Null
                $O := O \cup \{u\}$
            end if
        end if
    end for
end function
```

The augment procedure finds the augmenting path going through the tree S, the edge e and the tree T. The maximal possible flow is then sent along this path. At least one edge will then become saturated. If a saturated edge occurs in the trees S or T, the edge terminal farthest from the tree root is marked as an orphan. Note that even if a whole subtree is disconnected from the main tree, only the root of this subtree is marked as an orphan. The adopt procedure will take care of either reconnecting, or freeing all the vertices in the disconnected subtree.

The adopt procedure processes all the vertices in the set of orphans O. These vertices are either single vertices, or roots of disconnected subtrees. For an orphaned vertex u we look through its neighbors to find a possible parent vertex v. It should belong to the same tree as u, and also satisfy TREECAP$(v, u) > 0$. In addition, v should be connected to one of the tree roots s or t. This keeps us from connecting to other orphaned vertices or subtrees, and is checked by the procedure TREEORIGIN(v), which follows the parent information until reaching either s, t or an orphaned vertex.

Algorithm 9 The adopt procedure of the Boykov–Kolmogorov maximum flow algorithm

```
function ADOPT($G$, $s$, $t$, $O$)
    while $|O| \neq 0$ do
        $u \leftarrow$ one node from $O$
        Remove $u$ from $O$
        found $\leftarrow$ False
        for all $v$ such that TREECAP($v$, $u$) $> 0$ do
            if $u$.color $\neq$ $v$.color then
                continue
            end if
            orig $\leftarrow$ TREEORIGIN($v$)
            if orig $\neq s$ and orig $\neq t$ then
                continue
            end if
            found $\leftarrow$ True
            $u$.parent $\leftarrow v$
            break
        end for
        if found $\neq$ True then
            for all $v$ such that $u$.color $=$ $v$.color do
                if TREECAP($v$, $u$) $> 0$ then
                    $A := A \cup \{v\}$
                end if
                if $v$.parent $= u$ then
                    $O := O \cup \{v\}$
                    $v$.parent $\leftarrow$ Null
                end if
                $u$.color $\leftarrow$ Free
                $A := A - \{u\}$
            end for
        end if
    end while
end function
```

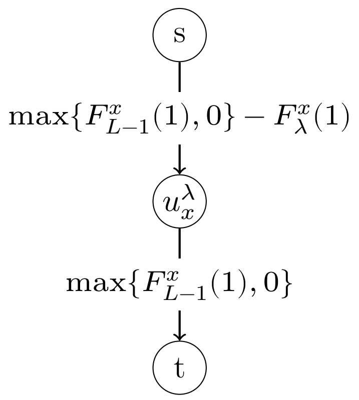

s
$\max\{F^x_{L-1}(1), 0\} - F^x_\lambda(1)$
u^λ_x
$\max\{F^x_{L-1}(1), 0\}$
t

Figure 5.3: A modified version of the subgraph corresponding to the fidelity term $F^x_\lambda(u^\lambda_x)$ that has positive and non-decreasing edges with decreasing λ.

If a potential parent v of u is *not* found, u becomes a free vertex. All vertices that had u as its parent are orphaned, and are thus treated by the adopt procedure later. Vertices that are in one of the trees S or T, and have a non-saturated edge to this newly freed vertex u, become active.

When no more active vertices remain, there are no more possible augmenting paths, and the algorithms terminates with a maximum flow. A proof of correctness can be found in Kolmogorov's PhD thesis [28].

5.5.1 Graph reuse

As with the push-relabel algorithm described earlier, it is not necessary to completely restart the Boykov–Kolmogorov algorithm for every level of the image, but we can reuse the graph and the trees S and T in successive runs.

Since this is algorithm does not deal with the relaxed preflow concept, the updates from one level to the next have to be done in a different way. We can no longer decrease the capacity of edges, as this could break the flow conservation constraint. However, the graph construction in Section 4.2.2 did allow for the addition of an arbitrary constant to all the edges of the sub-graph. Thus the construction in Figure 5.3 is also valid. We have added the non-negative constant $\max\{F^x_{L-1}(1), 0\}$ to all the edges of the graph in Figure 4.7b. As λ goes from $L-1$ to 0 both edges stay non-negative. The edge from u^x_λ to t is non-decreasing with decreasing λ parameter.

This construction allows us to update the capacities of the edges, while retaining the flow and the trees S and T. A nice property that follows is that the partition $(S, V - S)$ is after each run a minimum cut since the tree S has been

Table 5.1: Each row represents one of the two possible values of $u_x^\lambda \in \{0, 1\}$. The functional $F_\lambda^x(u_x^\lambda)$ and minimum cut obtaining this configuration is shown. The last column shows the capacities of the cut for the graph construction in Figure 5.3. We verify that the cut capacities are equal to the functional value, plus a constant.

u_x^λ	$F_\lambda^x(u_x^\lambda)$	Min. cut (S, T)	Graph cut cap.
0	0	$(\{s\}, \{u_x^\lambda, t\})$	$\max\{F_{L-1}^x(1), 0\} - F_\lambda^x(1)$
1	$F_\lambda^x(1)$	$(\{s, u_x^\lambda\}, \{t\})$	$\max\{F_{L-1}^x(1), 0\}$

grown as much as possible. Further, no vertex already in the tree S will leave S when the trees are updated and algorithm is run for lower λ values. This can be seen from the fact that if u is in S and the algorithm has terminated, all paths from s to t going through u have a saturated edge somewhere after u. For all these paths, the capacity will only change for edges before u, as we only change the capacity of edges (s, v). Thus these saturated edges will stay saturated, and u will stay in S.

5.5.2 Performance improvements

There are several open choices in the implementation of the algorithm, for example the order in which active vertices are processed. As recommended in [21], we have implemented a «First-In-First-Out» queue. This ensures that at least the first augmenting path found is a shortest path, although later the distance information is lost in the adoption stage.

In the adoption stage it is possible, and perhaps preferable to seek a possible parent that is closest to the root of the tree, and adopt that vertex as parent, instead of the first one found.

Because of the particular graph construction, all vertices except s and t are connected directly to s and t. Thus there are as many two-edged paths from s to t as there are pixels in the image. When increasing the capacity of edges (v, t), a quick sweep over these two-edged paths to send any possible flow may speed up the algorithm.

Chapter 6

Results

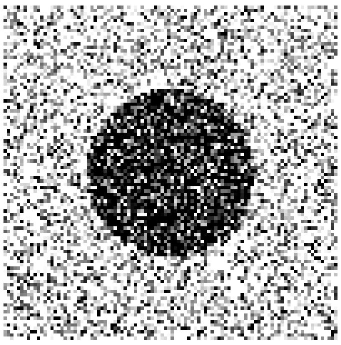
In the previous chapters we have carefully constructed a method always trying to argue how the choices we make can have a positive impact on the image restoration results. The anisotropic total variation is the main advance from my project work [1], and we would like to see how the introduction of the anisotropy affects the performance of the restoration algorithm.

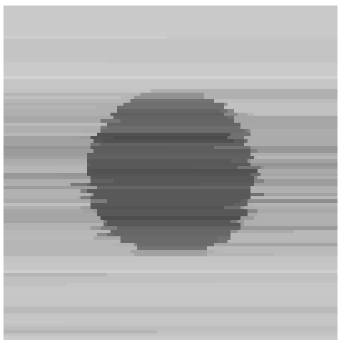
6.1 Tensor parameters

The anisotropy was introduced into the total variation in order to lessen the regularization applied across what we know, or at least are pretty sure to be edges in the image. Before considering the anisotropy tensor construction described earlier, we will look at how a simple pre-described tensor affects the regularization. Imagine a tensor which is a diagonal matrix $A = \text{diag}(1, \epsilon)$, where $\epsilon \ll 1$ is small. This would result in us down-weighting the size of ∇u in the y-direction, and thus regularization mostly in the x-direction.

The results of this experiment can be seen in Figure 6.1, where a noisy picture of a circle has been restored in two different ways, first with a uniform anisotropy tensor $A = \text{diag}(1, \epsilon)$, and then with the tensor described in Section 3.1.1. We see in Figure 6.1b that the uniform tensor gives a strong smoothing in the x-direction, but no apparent smoothing in the y-direction.

It is useful to look at intermediate results to better understand what happens and how the parameters affect our end result. In Figure 6.2 we show the different stages of our image restoration algorithm. We start out with a section of the much-used Lena test image in Figure 6.2a. The first step in the construction of the structure tensor is to apply a Gaussian blur with parameter σ, and the result is shown in Figure 6.2b. The blurring is done so that the edge detector ∇f_σ,


(a) Noisy circle.


(b) Uniform tensor.

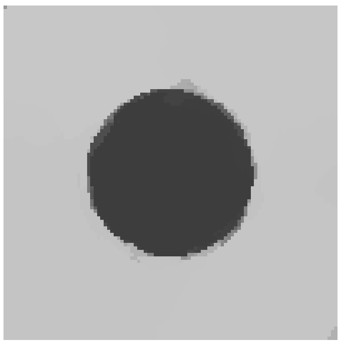
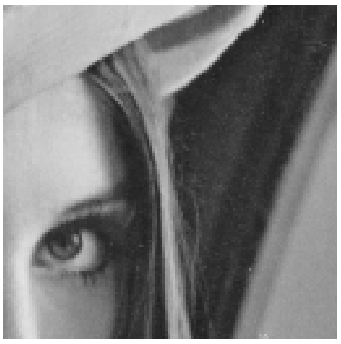
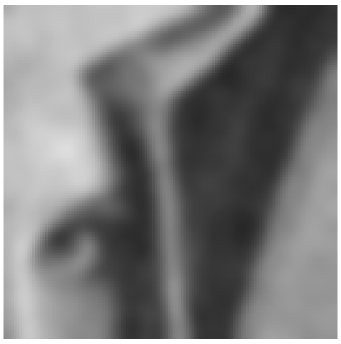
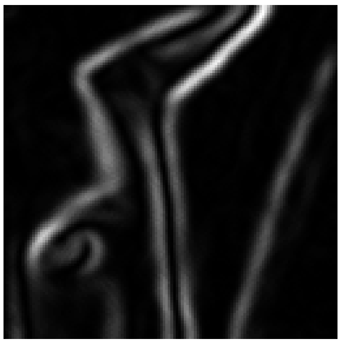
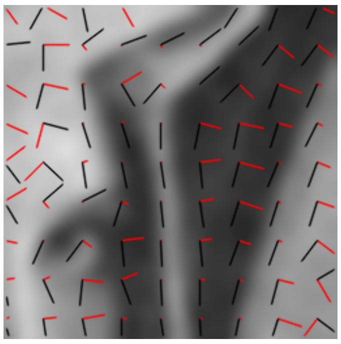

(c) Normal tensor.

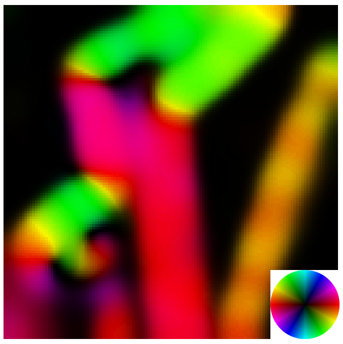
Figure 6.1: A noisy circle first restored using a uniform tensor, resulting mostly in smoothing in the x-direction. Then restored using the tensor described in Section 3.1.1.


(a) Lena's eye.


(b) Smoothed.


(c) Edge detector.


(d) Anisotropy tensor.


(e) Anisotropy tensor.

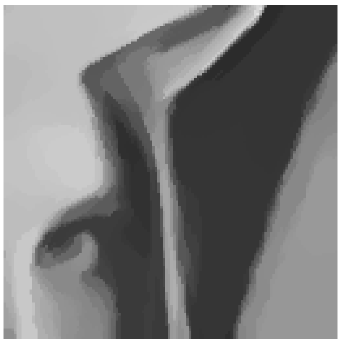

(f) Restored Lena.

Figure 6.2: The different stages of the restoration algorithm, showing the original image, the smoothed image, the edge detector, two visualizations of the anisotropy tensor, and finally the restored image.

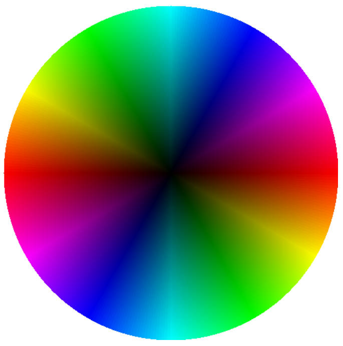


Figure 6.3: Color wheel used for tensor visualization.

visualized as $|\nabla f_\sigma|^2$ in Figure 6.2c, is not too sensitive to noise in the image. The structure tensor is then constructed using ∇f_σ and smoothed according to the integration scale ρ. The resulting anisotropy tensor is visualized in Figure 6.2d. In a selection of points, the eigenvectors of the tensor have been drawn. The length of the vectors have been scaled by the corresponding eigenvalue.

Another way of visualizing the tensor is shown in Figure 6.2e. The brightness and color in each point is decided by the size of the smallest eigenvalue and the direction of its corresponding eigenvector using the color wheel in Figure 6.3. The direction decides the color, while the brightness, which is the radius in the color wheel, is set to $1/\sigma_1 - 1$, where σ_1 is the smallest eigenvalue. Thus the stronger the anisotropy, the brighter the color, while we expect uniform areas in the original image to be black in the tensor visualization.

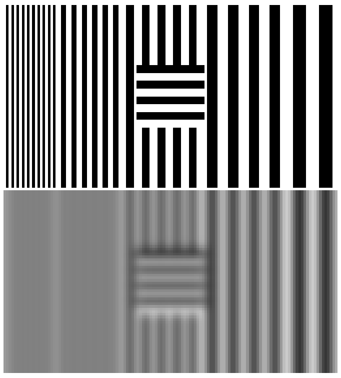
Finally, Figure 6.2f shows the restored image, and we see that it has been heavily regularized. How the anisotropy affected the regularization is not obvious however.

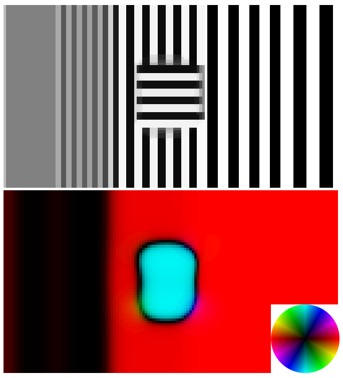
To look further into the effects of the parameters in our tensor construction we have a constructed a zebra pattern of increasing width as shown in Figure 6.4. The anisotropy introduced should in theory help reduce contrast loss in this situation, by reducing the regularization applied in the x-direction across the edges. There is however the question of how the different scales σ and ρ affect the regularization.

In Figure 6.4b the noise scale is increased such that the edge detector, and thus the anisotropy tensor, considers the finest lines to be noise rather than details. Thus this left-most part is regularized to a smooth gray area.

The integration scale ρ of (3.5) controls the size of the structures we want to detect with our anisotropy tensor. In Figure 6.4c a high value for ρ means that the structure found in the inner square is almost completely ignored in favor of the larger, more coherent structure around it.

Another visualization of the anisotropy tensor can be seen for a fingerprint image in Figure 6.5. We see that the gradual changes in the direction of the


(a) Original and blurred with $\sigma = 2$.


(b) High noise scale of $\sigma = 2$ means thinnest lines are regularized.

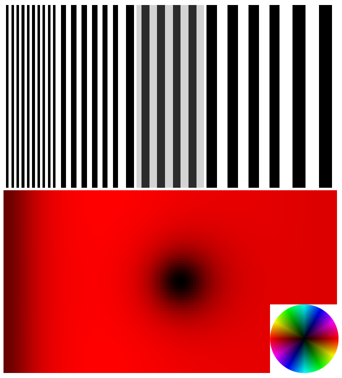
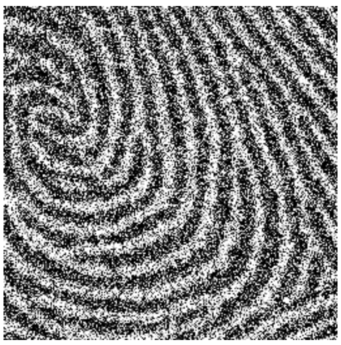
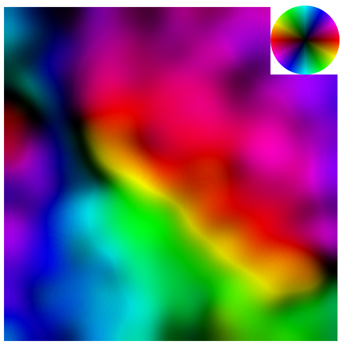

(c) Low noise scale $\sigma = 0.2$, but large integration scale $\rho = 15$ means inner structure is ignored.

Figure 6.4: An example constructed to show the effects of the parameters σ and ρ in the anisotropy tensor. The noise scale σ controls the smoothing done before edge detection. The integration scale ρ controls the size of the structures considered by the tensor. Parameters: $|\mathcal{N}| = 8$, $\beta = 10000$, $\omega = 100$.


(a) Noisy fingerprint.


(b) $\rho = 10$

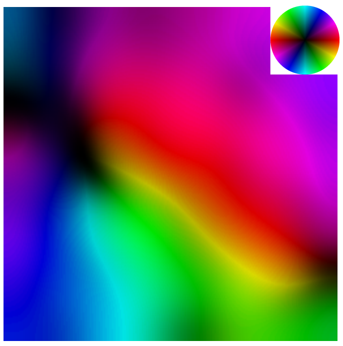

(c) $\rho = 20$

Figure 6.5: A noisy fingerprint with the anisotropy tensor visualized for different integration scales ρ. Parameters: $|\mathcal{N}| = 32$, $\sigma = 3$, $\beta = 15000$, $\omega = 150$.

edges are captured by the tensor. Even the singularities in the fingerprint can be identified by finding the dark spots, as the gradient of the image in these areas do not point in one single direction.

6.2 Neighborhood stencils

The neighborhoods were introduced as a way to describe the discrete set of lines $\mathcal{L}_D$ in the discretization of the regularization term in Section 4.1.2. And as discussed, we want the stencil to have many short edges, such that the angular differences $\Delta\phi$, the inter-line distances $\Delta\rho$ and the edge lengths e are "small."

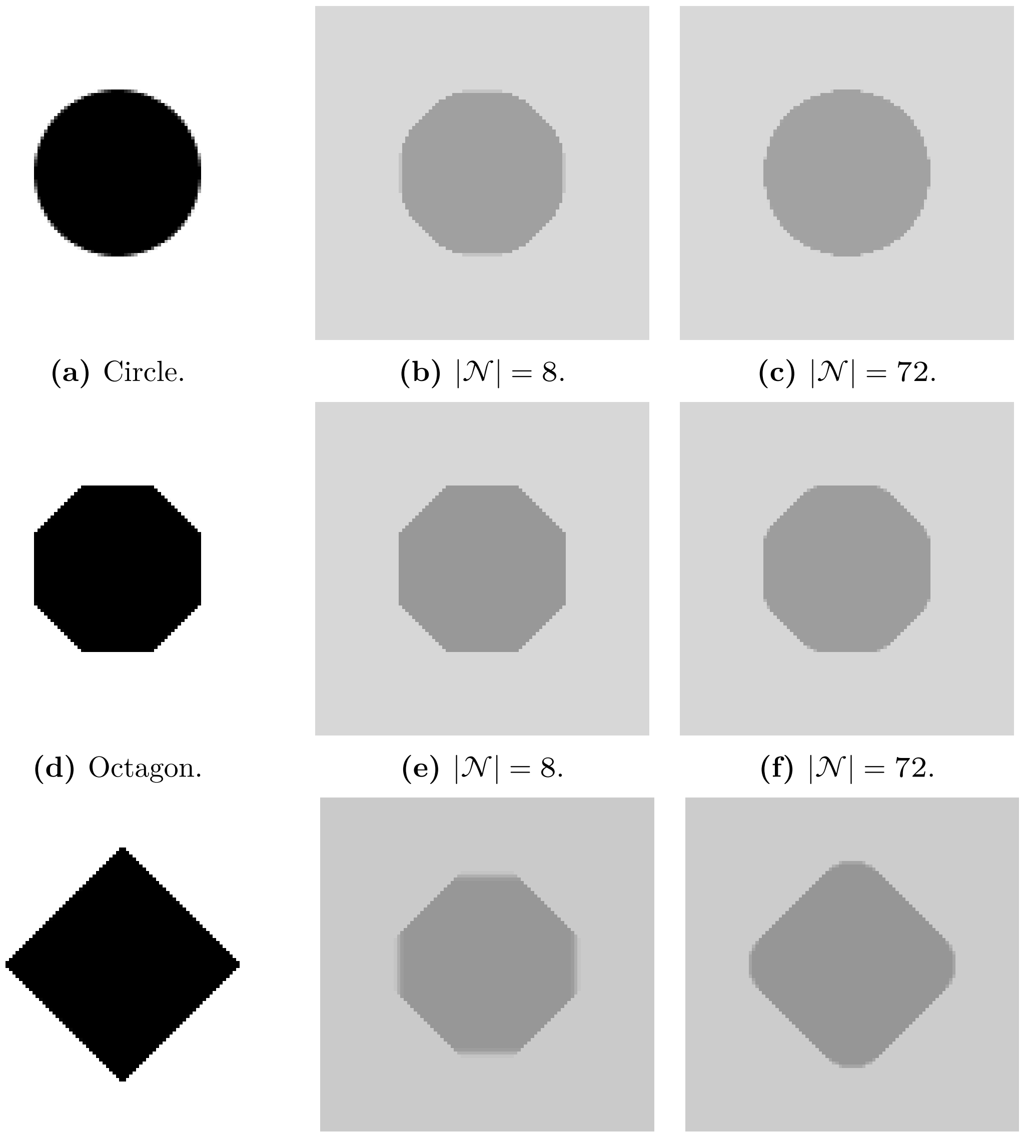
Figure 6.6 shows three images of different shapes that are heavily regularized using different neighborhood stencils. In these results we see some of the artifacts that may occur because of this particular discretization method. It is clear that the stencil of size 8 prefers horizontal, vertical and 45° perimeters, such that the circle in Figure 6.6a is shaped like an octagon when regularized in Figure 6.6b. However the stencil of size 72 manages to keep the circular shape.

In Figure 6.6d, we see that the stencil of size 8 actually favors octagon-like shapes, as the restored shape is the same octagon, just with some contrast loss.

The tilted square in Figure 6.6g, hints that a stencil of size 4 neighborhood favors horizontal and vertical edges only. While the larger stencil of size 72 retains the square shape but rounds the corners some.

As mentioned before, there is a discretization error which relates to the length of the edges in the neighborhood. We approximated the number of times an edge e_{ab} crosses level set boundary by $\left|u_a^\lambda - u_b^\lambda\right|$, an approximation that becomes worse for long edges. Thus a larger neighborhood is not always better, even if it will reduce the artifacts discussed above. In Figure 6.7, a noisy image of Lena has been restored using two neighborhood stencils, and there are obvious differences. For the stencil of size 72, the restored image still contains some pixel-sized noise. An explanation can be found in Figure 6.8. We see why the length of a one-pixel curve is underestimated by the Cauchy–Crofton formula when some of the edges are long. Because many edges e_{ab} cross the curve cross while $\left|u_a^\lambda - u_b^\lambda\right| = 0$, thus these edges are ignored completely in our perimeter approximation.

An additional demonstration that this problem mostly relates to small sized noise is shown in Table 6.1. The table shows how our discrete Cauchy–Crofton formula approximates the circumference of circles of different radii. Note that the circumference approximated is that of an actual continuous circle $u : \mathbb{R}^2 \to \{0, 1\}$ and not a discrete representation. We see that the perimeter of the smallest circle is grossly underestimated by the larger neighborhoods. Thus the perimeter of one-pixel noise will be underestimated, and in turn the contribution to the total variation by one-pixel noise will be smaller for large neighborhoods.

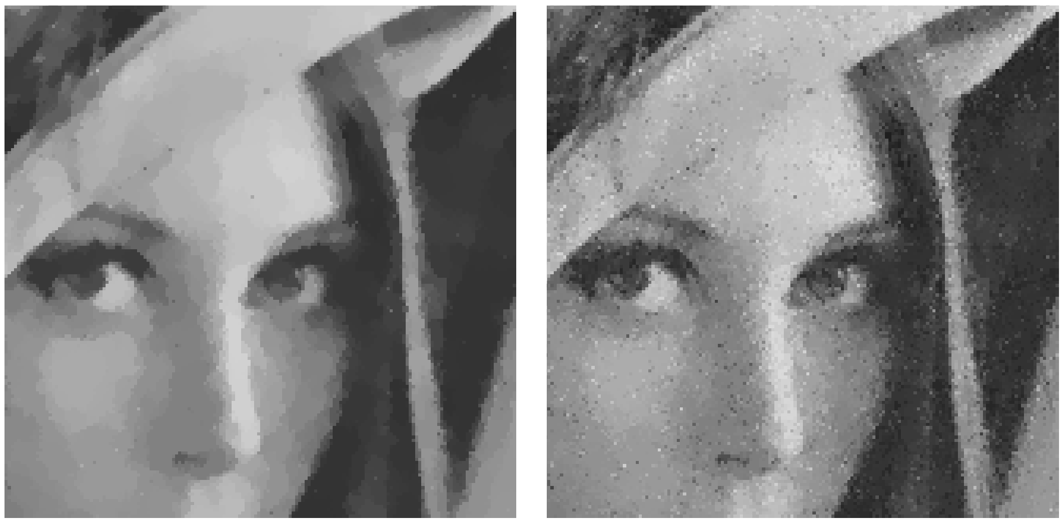


(a) Circle. **(b)** $|\mathcal{N}| = 8$. **(c)** $|\mathcal{N}| = 72$.

(d) Octagon. **(e)** $|\mathcal{N}| = 8$. **(f)** $|\mathcal{N}| = 72$.

(g) Tilted square. **(h)** $|\mathcal{N}| = 4$. **(i)** $|\mathcal{N}| = 72$.

Figure 6.6: Different test images restored without anisotropy, different neighborhoods and a large restoration parameter β. We see how different neighborhood stencils introduce different artifacts.



(a) $|\mathcal{N}| = 16$ **(b)** $|\mathcal{N}| = 72$

Figure 6.7: Noisy Lena restored using different neighborhood stencils. Note how the large neighborhood introduces some pixel size artifacts.

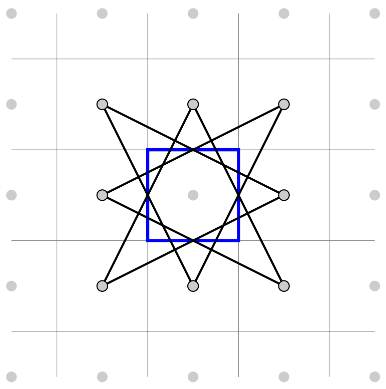


Figure 6.8: A selection of edges contributing to the wrong approximation of the length of a single pixel's perimeter. The edges cross the perimeter twice, but we count zero crossings in our approximation. This leads to residual pixel noise in the restored image, when large neighborhood stencils are used.

Table 6.1: The circumference of circles of different radii r measured by the discretized Cauchy–Crofton formula in (4.15), using different neighborhood stencils.

$\vert\mathcal{N}\vert$	$r = 0.5$	$r = 1.5$	$r = 5.5$	$r = 50.5$
4	3.14	9.42	34.56	317.3
8	2.68	10.27	33.94	317.5
16	2.08	9.97	34.59	316.8
32	1.63	9.24	34.44	317.2
48	1.40	8.40	34.29	317.3
72	1.21	7.45	33.95	317.2
$2\pi r$	3.14	9.42	34.56	317.3

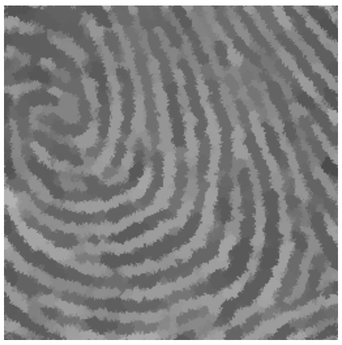
6.3 Restoration

We have claimed that the anisotropic total variation will reduce some of the contrast loss one can encounter with regular total variation regularization. In Figure 6.9 the noisy fingerprint from Figure 6.5a has been restored three times with different parameters. The first using regular total variation and $\beta = 15000$, then the second with $\beta = 15000$ and anisotropy $\omega = 150$. This gives an obvious contrast enhancement, but it is however not obvious what this tells us about the quality of the anisotropic algorithm. As previously mentioned, increasing the anisotropy (decreasing ω) means decreasing the total amount of regularization applied, and less contrast loss is an expected outcome of decreasing the regularization, disregarding the anisotropy.

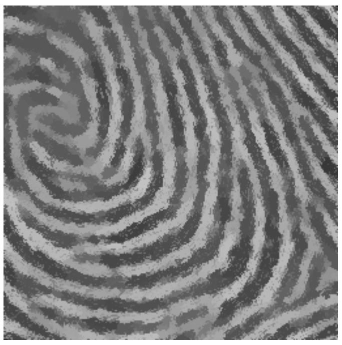
Thus in the last image we have also used $\omega = 150$ but the restoration amount β is increased such that the amount of noise removed $\|u - f\|_{L^2}$ is approximately equal to the noise removed when using regular total variation in Figure 6.9a. By visual inspection, the last image seems to have somewhat higher contrast. This can be confirmed by calculating the standard deviation of the images, one of many possible contrast measures, which gives 22.9, 37.9 and 27.2 respectively.

To further compare the three restored images, a single row has been extracted from the three images and is shown against each other in Figure 6.10. Note that although the results in Figure 6.9c look promising compared to Figure 6.9a, some details in the singularity in the upper left is actually lost. The metric tensor is approximately the identity matrix there, and thus an increased restoration parameter β leads to increased smoothing compared to the regular total variation of Figure 6.9a.

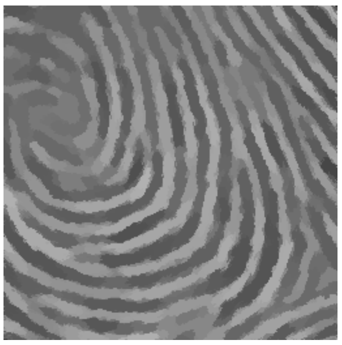
So far we have just inspected the restored image visually to judge the results of the method. Depending on the application, we might have different wishes for the



(a) Regular TV, $\beta = 15000$, $\|u - f\|_{L^2} = 17022$.



(b) Anisotropic TV ($\omega = 150$), $\beta = 15000$, $\|u - f\|_{L^2} = 15272$.



(c) Anisotropic TV ($\omega = 150$), $\beta = 20970$, $\|u - f\|_{L^2} = 17022$.

Figure 6.9: A noisy fingerprint restored using both isotropic and anisotropic total variation to see how the contrast loss compares. Parameters: $|\mathcal{N}| = 32$, $\sigma = 3$, $\rho = 10$.

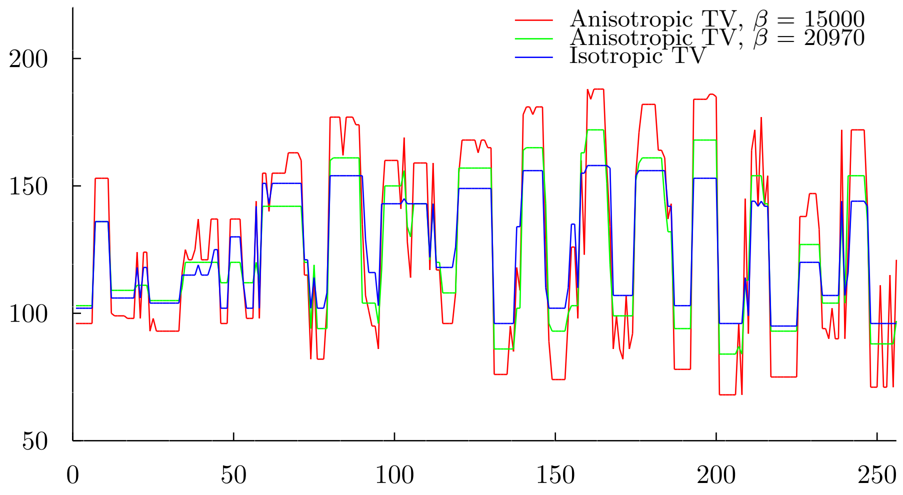
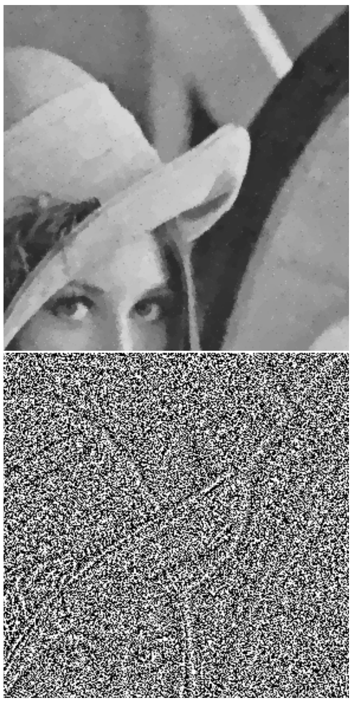
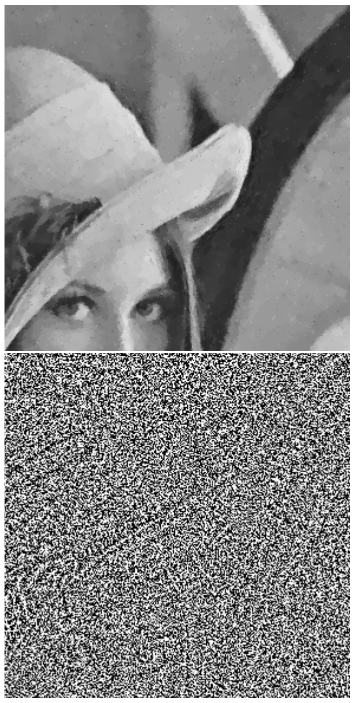


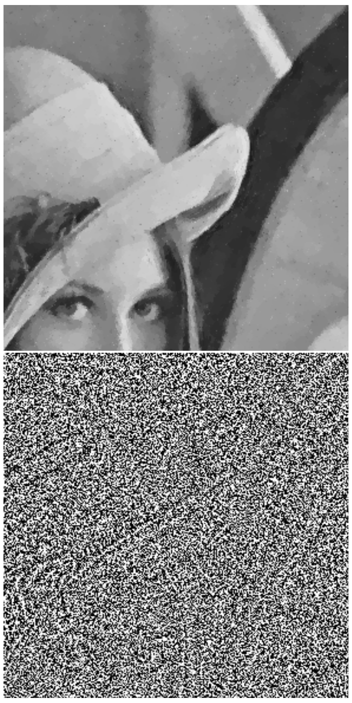
Figure 6.10: A one-dimensional slice through the restored fingerprint images of Figure 6.9, showing a difference in the contrast.



(a) Regular total variation, $\beta = 3000$, $\|u-f\|_{L^2} = 9771$



(b) Anisotropic TV, $\beta = 3000$, $\omega = 110$, $\|u-f\|_{L^2} = 9556$



(c) Anisotropic TV, $\beta = 3191$, $\omega = 110$, $\|u-f\|_{L^2} = 9771$

Figure 6.11: Lena with additive Gaussian noise ($\sigma = 30$) shown restored with different parameters, with corresponding method noise below. A slight improvement in the amount of detail in the method noise is seen from (a) to (c).

results. In many cases we just want to remove the noise δ in $f = u^* + \delta$ where the captured image f is assumed to consist of an actual image u^* and an additional noise term δ. The noise removed by the restoration method, $\widehat{\delta} = f - u^*$ is called the method noise. Any assumptions of properties we have on δ, we optimally want $\widehat{\delta}$ to fulfill as well. Thus if we assumed independently distributed Gaussian noise, we want to remove that kind of noise. An indication of problems can in that case be if the method noise contains a lot of the details of the image.

Figure 6.11 shows the results after restoring a noisy Lena picture with different parameters, and their method noise. We see that the method noise of Figure 6.11a shows slightly more details than the anisotropic counterpart in Figure 6.11c.

Chapter 7

Discussion and conclusion

In this thesis we have seen how the total variation restoration method can be extended using an anisotropy tensor, and how this fits into the discretization and graph cut framework used in my project work [1]. The anisotropy tensor was introduced in hopes of reducing the amount of regularization applied across edges in the image, and by that preventing contrast loss. It was constructed based on the structure tensor which contains local information about edge direction and steepness.

In order to arrive at a discrete functional that could be minimized using the same graph cut framework as in my project work [1], we transformed the continuous functional using an anisotropic coarea and Cauchy–Crofton formula, both described in detail. The resulting discrete functional was shown—under some restrictions—to be consistent with the continuous functional.

The successive graph cuts, each giving a minimizer for a functional of a level in the image, were found using maximum flow algorithms. The push-relabel algorithm is know to have good all-round performance, while the Boykov–Kolmgorov algorithm is tailored for the kinds of graphs appearing in imaging applications. They were both described and implemented.

A goal was to give readers a good understanding of the inner workings of the method, as well as all details necessary for a working implementation. Thus we put effort into describing all steps going from the initial continuous formulation, to the discretization, and finally the maximum flow algorithms.

One part of the thesis consists of studies of how the different parameters affect the performance of the algorithm. We will not give any unified conclusion as to whether this method is "better" or "worse" than the regular total variation method it is based on, or other methods. In different applications the input images have different properties, and one might also have different hopes for, and restrictions on the output image. But we have seen how the anisotropy tensor affects the restoration, and that it can have positive effects on contrast loss as well as method noise.

Further work is possible in the study of the continuous problem, its well-foundedness, and also the anisotropic coarea and Cauchy–Crofton formulas, which can be studied on a measure-theoretic foundation. Also, the construction of the anisotropy tensor offers choices that can be explored further.

Regarding the discretization, the choice of neighborhood stencil also allows for further discussion, as approximation error can be traded for algorithm performance. One particular possibility would be the application of non-uniform stencils with varying stencil size depending on the local level of detail. This could give better performance without sacrificing too much solution accuracy.

Bibliography

[1] Bjørn Rustad. Total variation based image restoration using graph cuts. 2014.

[2] Bernd Jähne. *Digital Image Processing*. Springer-Verlag, Berlin Heidelberg, 2005.

[3] Gilles Aubert and Pierre Kornprobst. *Mathematical problems in image processing*, volume 147 of *Applied Mathematical Sciences*. Springer, New York, second edition, 2006.

[4] Joachim Weickert. *Anisotropic diffusion in image processing*. European Consortium for Mathematics in Industry. B. G. Teubner, Stuttgart, 1998.

[5] Leonid I Rudin, Stanley Osher, and Emad Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, 60(1–4):259–268, 1992.

[6] V. Caselles, A. Chambolle, and M. Novaga. Total variation in imaging. In Otmar Scherzer, editor, *Handbook of Mathematical Methods in Imaging*, pages 1016–1057. Springer, New York, 2011.

[7] Raymond Chan, Tony Chan, and Andy Yip. Numerical methods and applications in total variation image restoration. In *Handbook of Mathematical Methods in Imaging*, pages 1059–1094. Springer, 2011.

[8] Markus Grasmair and Frank Lenzen. Anisotropic total variation filtering. *Appl. Math. Optim.*, 62(3):323–339, 2010.

[9] C. Olsson, M. Byrod, N.C. Overgaard, and F. Kahl. Extending continuous cuts: Anisotropic metrics and expansion moves. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 405–412, Sept 2009.

[10] Joachim Weickert. Coherence-enhancing diffusion filtering. *International Journal of Computer Vision*, 31(2-3):111–127, 1999.

[11] Robert E. Megginson. *An introduction to Banach space theory*, volume 183 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1998.

[12] Otmar Scherzer, Markus Grasmair, Harald Grossauer, Markus Haltmeier, and Frank Lenzen. *Variational methods in imaging*, volume 167 of *Applied Mathematical Sciences*. Springer, New York, 2009.

[13] Lawrence C. Evans and Ronald F. Gariepy. *Measure theory and fine properties of functions*. Studies in Advanced Mathematics. CRC Press, Boca Raton, FL, 1992.

[14] Robert G Bartle. *The Elements of Integration and Lebesgue Measure*. John Wiley & Sons, 1995.

[15] William P. Ziemer. *Weakly differentiable functions*, volume 120 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1989.

[16] Edwin Hewitt and Karl Stromberg. *Real and abstract analysis. A modern treatment of the theory of functions of a real variable*. Springer-Verlag, New York, 1965.

[17] Manfredo P. do Carmo. *Differential geometry of curves and surfaces*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, third edition, 2009.

[19] Vladimir Kolmogorov and Ramin Zabih. What energy functions can be minimized via graph cuts? In *Computer Vision—ECCV 2002*, pages 65–81. Springer, 2002.

[20] Yefim Dinitz. Dinitz' algorithm: the original version and Even's version. In *Theoretical computer science*, volume 3895 of *Lecture Notes in Comput. Sci.*, pages 218–240. Springer, Berlin, 2006.

[21] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(9):1124–1137, 2004.

[22] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. Assoc. Comput. Mach.*, 35(4):921–940, 1988.

[23] Joseph Cheriyan and S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing*, 18(6):1057–1086, 1989.

[24] B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[25] Donald Goldfarb and Wotao Yin. Parametric maximum flow algorithms for fast total variation minimization. *SIAM Journal on Scientific Computing*, 31(5):3712–3743, 2009.

[26] Giorgio Gallo, Michael D. Grigoriadis, and Robert E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.*, 18(1):30–55, 1989.

[27] Dorit S. Hochbaum. An efficient algorithm for image segmentation, Markov random fields and related problems. *J. ACM*, 48(4):686–701 (electronic), 2001.

[28] Vladimir Kolmogorov. *Graph Based Algorithms for Scene Reconstruction from Two or More Views*. PhD thesis, Cornell University, September 2003.

[29] Bjørn Rustad. C++ implementation. `https://github.com/burk/image-restoration`.

[30] G. Bradski. Opencv library. *Dr. Dobb's Journal of Software Tools*, 2000.

List of Figures

List of Tables

List of Symbols

$\lvert C\rvert$	The length of the curve C, page 25
$\lvert C\rvert_M$	The length of the curve C using the metric tensor M, page 25
$\alpha(\nabla u)$	Scalar thermal diffusivity, page 4
β	Regularization parameter, page 6
$\mathrm{BV}(\Omega)$	Functions of bounded variation in Ω, page 7
δ	Noise function, page 3
$\ell_{\nu,\rho}$	Line given by tangent vector ν and distance to origin ρ, page 25
$\ell_{\phi,\rho}$	Line given by angle of normal ϕ and distance to origin ρ, page 25
χ_E	Characteristic function of the set E, page 9
$\lVert\eta\rVert_A^*$	The norm $\sup_x(\eta^T A^{-1}\eta)^{1/2}$, page 12
$\lVert\xi\rVert_A$	The norm $\sup_x(\xi^T A\xi)^{1/2}$, page 12
$\mathcal{G}$	Regular grid of pixels over Ω, page 9
$\mathcal{L}$	The set of all straight lines in the plane, page 25
$\mathcal{N}(x)$	Neighborhood of pixel x, page 35
$\mathcal{P}$	Discrete set of pixel values, or levels, page 9
∇f_σ	Edge detector, page 13
Ω	Rectangular, open domain, page 3

ω	Anisotropy parameter, page 15
$\mathrm{Per}_A(U;\Omega)$	Anisotropic perimeter of set U using anisotropy tensor A, page 23
$\tilde{u}$	Symmetric extension of u from Ω to $\mathbb{R}^2$, page 6
$\mathrm{TV}(u)$	Total variation of u, page 7
$\mathrm{TV}_A(u)$	Anisotropic total variation of u given anisotropy tensor A, page 12
$A(u)$	Thermal diffusivity tensor, page 5
$A(x)$	Anisotropy tensor, page 11
$c(u,v)$	Capacity function $c : V \times V \to [0,\infty)$, page 41
$C^\infty(\Omega)$	Space of infinitely differentiable functions from Ω to $\mathbb{R}$., page 6
$C_c^\infty(\Omega,\mathbb{R}^2)$	Space of smooth functions from Ω to $\mathbb{R}^2$ with compact support in Ω, page 7
$d(u)$	Height map or distance labeling $d : V \to \mathbb{N}$, page 52
$e(u)$	Excess in vertex u, page 52
f	Original, noisy image, page 3
$G = (V,E,c)$	Graph given by the vertices V, edges E, and capacity function c, page 41
$J_M(\ell_{\phi,\rho})$	Jacobian of the coordinate transformation induced by the metric tensor M, page 26
$K_\sigma(x,y)$	The Gaussian kernel, page 4
$L^p(\Omega)$	Real functions f on Ω for which $\int_\Omega \lvert f\rvert^p < \infty$, page 6
$M(x)$	The metric tensor, page 25
$N_x(k)$	Helper function $N_x(k) = \lvert k - f_x\rvert^2$, page 33
s	The source vertex, page 41
$S_\rho(x)$	Structure tensor, page 6
t	The sink vertex, page 41
u^*	Usually unknown, actual image without noise, page 3
u^λ	Thresholded image at level λ, page 9

Appendix A

C++ implementation

A C++ implementation of the method described is included here and can also be found online at [29]. It uses the open computer vision library OpenCV [30] to load and save image files and contains compilation and usage instructions. The implementation has been tested on an installation of the Ubuntu Linux distribution, but it should in theory be portable to other platforms supported by OpenCV. A rudimentary graphics interface has also been made, to make it easier to play with the parameters of the algorithm.

Both the push-relabel and the Boykov–Kolmogorov algorithms have been implemented. Although an effort has been made to improve the performance of both implementations, they are not ment to beat the fastest. The focus has rather been on clarity and understanding.

Note that when implementing maximum flow algorithms it is not a good idea, memory- and performance-wise, to actually construct the residual graph G_f. Instead, every time we update the flow $f(u,v)$ we set the flow in the opposite direction to its negative value $f(v,u) = -f(u,v)$. Then we can at any time, consider the value $c(u,v) - f(u,v)$ in the place of the residual capacity $c_f(u,v)$.

For the gap relabeling heuristic of the push-relabel algorithm, we need to have a easy way of finding when a gap occurs. This is done by keeping track of how many vertices exist with each label.

When the capacities have been updated in the Boykov–Kolmogorov algorithm, flow is sent along all two-edge paths such that they do not have to be considered by the main loop of the algorithm.

A.1 main.cpp

Main function of the command line executable. Here we read and parse command line parameters.

```cpp
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <cstdio>
#include <unistd.h>
#include <opencv2/opencv.hpp>
#include "graph.hpp"
#include "selectionrule.hpp"
#include "neighborhood.hpp"
#include "image.hpp"
#include "anisotropy.hpp"

using namespace std;
using namespace cv;

int main(int argc, char *argv[])
{
        int p = 2;
        double beta = 10;

        int neighbors = 8;

        double sigma = 10.0;
        double rho   = 10.0;
        double gamma = 10000.0;

        int c;

        /* Read command line parameters beta and p. */
        while ((c = getopt(argc, argv, "b:p:r:s:g:n:fh")) != -1) {
                switch (c)
                {
                case 'p':
                        p = atoi(optarg);
                        break;
                case 'b':
                        beta = atof(optarg);
                        break;
                case 'g':
                        gamma = atof(optarg);
                        break;
                case 'r':
                        rho = atof(optarg);
                        break;
                case 's':
                        sigma = atof(optarg);
                        break;
                case 'n':
                        neighbors = atoi(optarg);
                        break;
                case '?':
                        if (optopt == 'p' || optopt == 'b' || optopt == 'g'
                                        || optopt == 'n' || optopt == 'r'
                                        || optopt == 's') {
                                fprintf(stderr, "Option -%c requires an argument.\n",
                                                optopt);
                        }
                        else if (isprint(optopt)) {
                                fprintf(stderr, "Unknown option `-%c'.\n", optopt);
                        }
                        else {
                                fprintf(stderr, "Unknown option character `\\x%x'.\n",
                                                optopt);
                        }
                        return 1;
                default:
                        exit(1);
                }
        }

        /*
         * Non-option arguments are now in argv from index optind
         * to index argc-1
         */
        Mat image;
        image = imread(argv[optind], CV_LOAD_IMAGE_GRAYSCALE);

        if (!image.data) {
                cout << "Loading image failed" << endl;
                return -1;
        }

        cout << "Using gamma = " << gamma << endl;
        cout << "Using rho = " << rho << endl;
        cout << "Using sigma = " << sigma << endl;

        Mat_<Tensor> tensors = Mat_<Tensor>::zeros(image.rows, image.cols);
        Mat blur, edge, structure, color;
        createAnisotropyTensor(tensors, image, sigma, rho, gamma,
                        blur, edge, structure, color);
        imwrite(argv[optind + 1], blur);
        imwrite(argv[optind + 2], edge);
        imwrite(argv[optind + 3], structure);
        imwrite(argv[optind + 4], color);

        /*
         * Network only handles integer edges, so we increase the scale a bit.
         */
        int a;
        int b;
        a = 100;
        b = beta;

        /*
         * Specify the neighbors of a pixel.
         */
        cout << "Creating size " << neighbors << " neighborhood." << endl;
        Neighborhood neigh;
        if (neighbors >= 4) {
                neigh.add( 1, 0, b * 1.0);
                neigh.add( 0, 1, b * 1.0);
                neigh.add(-1, 0, b * 1.0);
                neigh.add( 0,-1, b * 1.0);
        }

        if (neighbors >= 8) {
                neigh.add( 1, 1, b * 1.0/sqrt(2.0));
                neigh.add(-1, 1, b * 1.0/sqrt(2.0));
                neigh.add( 1,-1, b * 1.0/sqrt(2.0));
                neigh.add(-1,-1, b * 1.0/sqrt(2.0));
        }

        if (neighbors >= 16) {
                neigh.add8(1, 2, 1.0);
        }

        if (neighbors >= 32) {
                neigh.add8(3, 1, 1.0);
                neigh.add8(3, 2, 1.0);
        }

        if (neighbors >= 48) {
                neigh.add8(1, 4, 1.0);
                neigh.add8(3, 4, 1.0);
        }

        if (neighbors >= 72) {
                neigh.add8(1, 5, 1.0);
                neigh.add8(2, 5, 1.0);
                neigh.add8(3, 5, 1.0);
        }

        cout << "Neighborhood: " << endl;
        neigh.setupAngles();
        for (Neighborhood::iterator it = neigh.begin(); it != neigh.end(); ++it) {
                cout << it->x << ", " << it->y << ": " << it->dt * 180 / M_PI << endl;
        }

        Mat out = image.clone();
        restoreAnisotropicTV(image, out, tensors, neigh, a, b, p);

        cout << "Writing output to " << argv[optind + 5] << endl;
        imwrite(argv[optind + 5], out);

        return 0;
}
```

A.2 image.hpp

Calculates and sets up graph weights based on the input image and parameters.

```
#pragma once

#include <vector>
#include <set>
#include <opencv2/opencv.hpp>
#include "graph.hpp"
#include "selectionrule.hpp"
#include "neighborhood.hpp"
#include "anisotropy.hpp"

void createEdgesAnisotropic(
                FlowGraph& network,
                Neighborhood& neigh,
                int beta,
                const cv::Mat_<Tensor>& tensors
                );
void setupSourceSink(FlowGraph& network, cv::Mat& in, int alpha, int label, int p);
void setupSource(FlowGraph& network, cv::Mat& in, int alpha, int label, int p);
void setupSink(FlowGraph& network, cv::Mat& in, int alpha, int label, int p);

void restoreAnisotropicTV(
                cv::Mat& in,
                cv::Mat& out,
                cv::Mat_<Tensor>& tensors,
                Neighborhood& neigh,
                int alpha, int beta, int p
                );
```

A.3 image.cpp

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <set>
#include <iterator>
#include <cmath>
#include <opencv2/opencv.hpp>
#include "image.hpp"
#include "neighborhood.hpp"
#include "selectionrule.hpp"
#include "anisotropy.hpp"

using namespace std;
using namespace cv;

int f(int u, int v, int p) {
        return p == 2 ? (u - v) * (u - v) : abs(u - v);
}

/* Fidelity energy term. */
int Ei(int label, int pix, int u, int p) {
        return (f(label+1, pix, p) - f(label, pix, p)) * u;
}

void createEdgesAnisotropic(
                FlowGraph& network,
                Neighborhood& neigh,
                int beta,
                const Mat_<Tensor>& tensors
                ) {

        int rows = tensors.rows;
        int cols = tensors.cols;
        int pixels = rows * cols;

        /*
         * Add sink edges first, so that the first push in discharge
         * will go towards the sink. The capacities are set up in
         * setupSourceSink.
         */
        for (int i = 0; i < pixels; ++i) {
                network.addEdge(i, network.getSink(), 0);
        }

        /*
         * Create internal edges, which do not depend on the current
         * level.
         */
        for (int i = 0; i < rows; ++i) {
                for (int j = 0; j < cols; ++j) {
                        Neighborhood::iterator it;

                        for (it = neigh.begin(); it != neigh.end(); ++it) {

                                /* Only add edges for right half of the neighborhood. */
                                if (it->x < 0)
                                        continue;
                                if (it->x == 0 && it->y > 0)
                                        continue;

                                int x = j + it->x;
                                int y = i + it->y;

                                if (x >= 0 && x < cols && y >= 0 && y < rows) {

                                        Mat ee = (Mat_<double>(2, 1) << it->x, it->y);
                                        Mat M1 = Mat(tensors(i, j));
                                        Mat M2 = Mat(tensors(y, x));

                                        Mat M3 = (M1 + M2) / 2.0;

                                        double w = beta
                                                * norm(ee) * norm(ee)
                                                * determinant(M3)
                                                * it->dt
                                                / pow(ee.dot(M3 * ee), 3.0 / 2.0);

                                        network.addDoubleEdge(
                                                        i*cols + j,
                                                        y*cols + x,
                                                        w
                                                        );
                                }
                        }
                }
        }

        /*
         * Add edges from the source. Capacities are set up in
         * setupSourceSink.
         */
        for (int i = 0; i < pixels; ++i) {
                network.addEdge(network.getSource(), i, 0);
        }
}

/*
 * Change the capacities of the edges connecting the source
 * and the sink to the rest of the network, as these edges
 * are dependent on the current level.
 */
void setupSourceSink(FlowGraph& network, Mat& in, int alpha, int label, int p) {
        std::vector<int> s_caps(in.rows * in.cols);
        std::vector<int> t_caps(in.rows * in.cols);

        for (int j = 0; j < in.rows; ++j) {
                for (int i = 0; i < in.cols; ++i) {
                        int e1 = Ei(label, in.at<uchar>(j, i), 1, p);

                        if (0 < e1) {
                                t_caps[j*in.cols + i] += e1 - 0;
                        }
                        else {
                                s_caps[j*in.cols + i] += 0 - e1;
                        }
                }
        }

        for (size_t i = 0; i < s_caps.size(); ++i) {
                network.changeCapacity(network.getSource(), i, alpha * s_caps[i]);
        }

        for (size_t i = 0; i < t_caps.size(); ++i) {
                network.changeCapacity(i, network.getSink(), alpha * t_caps[i]);
        }
}

/* Update edges connected to the source. Used by Boykov--Kolmogorov. */
void setupSource(FlowGraph& network, Mat& in, int alpha, int label, int p) {
        std::vector<int> s_caps(in.rows * in.cols);

        for (int j = 0; j < in.rows; ++j) {
                for (int i = 0; i < in.cols; ++i) {
                        int e1init = Ei(255, in.at<uchar>(j, i), 1, p);
                        int e1 = Ei(label, in.at<uchar>(j, i), 1, p);

                        s_caps[j*in.cols + i] += max(e1init, 0) - e1;
                }
        }

        for (size_t i = 0; i < s_caps.size(); ++i) {
                if (s_caps[i] != 0)
                        network.changeCapacity(network.getSource(), i, alpha * s_caps[i]);
        }
}

/* Update edges connected to the sink. Used by Boykov--Kolmogorov. */
void setupSink(FlowGraph& network, Mat& in, int alpha, int label, int p) {
        std::vector<int> t_caps(in.rows * in.cols);

        for (int j = 0; j < in.rows; ++j) {
                for (int i = 0; i < in.cols; ++i) {
                        int e1init = Ei(255, in.at<uchar>(j, i), 1, p);

                        t_caps[j*in.cols + i] += max(e1init, 0);
                }
        }

        for (size_t i = 0; i < t_caps.size(); ++i) {
                if (t_caps[i] != 0)
                        network.changeCapacity(i, network.getSink(), alpha * t_caps[i]);
        }
}

/* Restore image. */
void restoreAnisotropicTV(
                Mat& in,
                Mat& out,
                Mat_<Tensor>& tensors,
                Neighborhood& neigh,
                int alpha, int beta, int p
                ) {

        int rows = in.rows;
        int cols = in.cols;
        int pixels = rows * cols;
        int source = pixels;
        int sink   = pixels + 1;

        FIFORule frule(pixels + 2);
        SelectionRule& rule = frule;
        FlowGraph network(rows * cols + 2, source, sink, rule);

        createEdgesAnisotropic(network, neigh, beta, tensors);

```

```
#ifdef BOYKOV_KOLMOGOROV
        setupSink(network, in, alpha, 255, p);
#endif

        for (int label = 255; label >= 0; --label) {
                cout << "Label: " << label << endl;

#ifdef PUSH_RELABEL
                setupSourceSink(network, in, alpha, label, p);
                network.minCutPushRelabel(source, sink);
#else
                setupSource(network, in, alpha, label, p);
                network.minCutBK(source, sink);
#endif

                /* Use the cut to update the output image. */
                for (int j = 0; j < rows; ++j) {
                        for (int i = 0; i < cols; ++i) {
                                if (!network.cut[j*cols + i])
                                        out.at<uchar>(j, i) = label;
                        }
                }
        }
}
```

A.4 anisotropy.hpp

Construction of anisotropy tensor.

```
#pragma once

#include <opencv2/opencv.hpp>

typedef cv::Matx<double, 2, 2> Tensor;

void createAnisotropyTensor(
                cv::Mat_<Tensor>& tensors,
                cv::Mat& in,
                double sigma,
                double rho,
                double gamma,
                cv::Mat& blur,
                cv::Mat& edge,
                cv::Mat& structure,
                cv::Mat& color
                );

void createUniformAnisotropyTensor(cv::Mat_<Tensor>& tensors, cv::Mat& in, double gamma);
```

A.5 anisotropy.cpp

```
#include <opencv2/opencv.hpp>
#include <cstdlib>
#include <cstdio>
#include "anisotropy.hpp"

using namespace cv;
using namespace std;

```

```
void createAnisotropyTensor(
                Mat_<Tensor>& tensors,
                Mat& in,
                double sigma,
                double rho,
                double gamma,
                cv::Mat& blur,
                cv::Mat& edge,
                cv::Mat& structure,
                cv::Mat& color
                ) {

        Mat grad;

        /* Apply blur by sigma. */
        GaussianBlur(in, blur, Size(0,0), sigma, 0, BORDER_REFLECT);

        Mat grad_x, grad_y;
        Mat kernel;

        Point anchor = Point(-1, -1);

               kernel = Mat::zeros(1, 3, CV_16S);

        kernel.at<short>(0, 0) = -1;
        kernel.at<short>(0, 1) = 0;
        kernel.at<short>(0, 2) = 1;

        /* Calculate gradient in x and y. */
        filter2D(blur, grad_x, CV_64F, kernel, anchor, 0, BORDER_REFLECT);
        transpose(kernel, kernel);
        filter2D(blur, grad_y, CV_64F, kernel, anchor, 0, BORDER_REFLECT);

        Mat x_sq, y_sq, xy;
        Mat x_sqr, y_sqr, xyr;

        /* Element of outer product. */
        x_sq = grad_x.mul(grad_x);
        y_sq = grad_y.mul(grad_y);
        xy   = grad_x.mul(grad_y);

        /* Integration scale (rho) smoothing. */
        GaussianBlur(x_sq, x_sqr, Size(0,0), rho, 0, BORDER_REFLECT);
        GaussianBlur(y_sq, y_sqr, Size(0,0), rho, 0, BORDER_REFLECT);
        GaussianBlur(xy  , xyr  , Size(0,0), rho, 0, BORDER_REFLECT);

        Mat evec, eval;

        Mat h = Mat::zeros(in.size(), CV_64F);
        Mat s = Mat::zeros(in.size(), CV_64F);
        Mat v = Mat::zeros(in.size(), CV_64F);
        Mat hsv;
        vector<Mat> channels;

        edge.create(in.size(), CV_64F);

        for (int i = 0; i < in.rows; ++i) {
                for (int j = 0; j < in.cols; ++j) {
                        /* Structure tensor. */
                        Tensor b = Tensor(
                                        x_sqr.at<double>(i,j),
                                        xyr.at<double>(i,j),
                                        xyr.at<double>(i,j),
                                        y_sqr.at<double>(i,j)
                                        );
```

```cpp

            /* Structure tensor without rho smoothing, for the edge detector. */
            Tensor c = Tensor(
                    x_sq.at<double>(i,j),
                    xy.at<double>(i,j),
                    xy.at<double>(i,j),
                    y_sq.at<double>(i,j)
                    );

            /* Returns the eigenvectors as row vectors! */
            eigen(b, eval, evec);

            double s1 = eval.at<double>(0);
            double s2 = eval.at<double>(1);

            if (s2 > s1)
                fprintf(stderr, "OOPS: Wrong ordering of eigenvalues\n");

            double l1 = 1.0;
            double l2 = 1.0 / (1.0 + (s1 - s2) * (s1 - s2) / (gamma*gamma));

            Mat eval2 = eval.clone();
            eval2.at<double>(0) = l1;
            eval2.at<double>(1) = l2;
            tensors(i, j) = Tensor(Mat(evec.t() * Mat::diag(eval2) * evec));

            h.at<double>(i, j) = fmod(atan2(evec.at<double>(1), evec.at<double>(0))
                    * 180.0 / M_PI + 180.0, 180.0);
            s.at<double>(i, j) = 0;
            v.at<double>(i, j) = 1.0/(1.0 + l2);

            /* Returns the eigenvectors as row vectors! */
            eigen(c, eval, evec);

            s1 = eval.at<double>(0);
            s2 = eval.at<double>(1);

            edge.at<double>(i, j) = s1;
        }
    }
    normalize(edge, edge, 0, 255, NORM_MINMAX, CV_8U);

    /* Create tensor visualization, double the size. */
    structure.create(in.size(), CV_64F);
    GaussianBlur(in, structure, Size(0,0), sigma, 0, BORDER_REFLECT);
    cvtColor(structure, structure, CV_GRAY2BGR);
    resize(structure, structure, Size(0, 0), 2, 2);
    for (int i = 0; i < in.rows; i += 15) {
        for (int j = 0; j < in.cols; j += 15) {
            Tensor b = tensors(i, j);

            eigen(b, eval, evec);

            double s1 = eval.at<double>(0);
            double s2 = eval.at<double>(1);

            Point2f p1(evec.row(0));
            Point2f p2(evec.row(1));
            line(structure, 2*Point(j, i), 2*Point2f(j, i) + 20 * s2 * p1,
                    CV_RGB(255,0,0), 1.2, CV_AA);
            line(structure, 2*Point(j, i), 2*Point2f(j, i) + 20 * s1 * p2,
                    CV_RGB(0,0,0), 1.2, CV_AA);
        }
    }

```

```
        Mat ho, so, vo;
        h.convertTo(ho, CV_8U);
        normalize(s, so, 255, 255, NORM_MINMAX, CV_8U);
        normalize(v, vo, 0, 255, NORM_MINMAX, CV_8U);

        channels.push_back(ho);
        channels.push_back(so);
        channels.push_back(vo);
        merge(channels, hsv);

        cvtColor(hsv, color, CV_HSV2BGR);
}

/* Create a uniform tensor, for testing. */
void createUniformAnisotropyTensor(Mat_<Tensor>& tensors, Mat& in, double gamma) {
        Mat evec, eval;
        for (int i = 0; i < in.rows; ++i) {
                for (int j = 0; j < in.cols; ++j) {
                        Tensor b = Tensor(1, 1, 1, 1);

                        /* Returns the eigenvectors as row vectors! */
                        eigen(b, eval, evec);

                        double l1 = 1.0 / gamma;
                        double l2 = 1.0;

                        Mat eval2 = eval.clone();
                        eval2.at<double>(0) = l1;
                        eval2.at<double>(1) = l2;
                        tensors(i, j) = Tensor(Mat::diag(eval2));
                }
        }
}
```

A.6 graph.hpp

Graph class with the maximum flow algorithms.

```
#pragma once

#include <iostream>
#include <queue>
#include <vector>
#include <list>
#include <set>
#include "selectionrule.hpp"

enum Color { FREE = 0, SOURCE, SINK };

class Edge {
private:

public:
        int from, to;
        int cap;
        int flow;
        int index;

        Edge(int f, int t, int c, int i) :
                from(f), to(t), cap(c), flow(0), index(i) {}
};

```

```
class Vertex {
private:

public:
        std::vector<Edge> e;
        Color c;
        bool active;
        Edge *p;
        int height;
        int excess;
        int si;
        int ti;

        Vertex(Color c, bool a) :
                c(c), active(a), p(NULL), height(0), excess(0), si(0), ti(0) {}

        Vertex() :
                c(FREE), active(false), p(NULL), height(0), excess(0), si(0), ti(0) {}
};

class FlowGraph {
private:
        int N;
        int source, sink;
        std::vector<Vertex > G;
        std::vector<int> count;
        SelectionRule& rule;

        /* BK stuff */
        std::queue<int> bkq;
        std::queue<int> orphans;

        int lastGrowVertex;
        size_t lastIndex;
public:
        std::vector<char> cut;

        FlowGraph(int N, int source, int sink, SelectionRule& rule) :
                N(N),
                source(source),
                sink(sink),
                G(N),
                count(N+1),
                rule(rule),
                lastGrowVertex(-1),
                cut(N) {

#ifdef BOYKOV_KOLMOGOROV
                bkq.push(source);
                bkq.push(sink);
                G[source].c = SOURCE;
                G[sink].c   = SINK;
                G[source].active = true;
                G[sink].active   = true;
#endif
        }

        int getSource() const { return source; }
        int getSink() const { return sink; }
        void addEdge(int from, int to, int cap);
        void addDoubleEdge(int from, int to, int cap);
        void changeCapacity(int from, int index, int cap);
        void resetFlow();
        void resetHeights();

        void push(Edge &e);
        void push(Edge &e, int f);
        void relabel(int u);
        void gap(int h);
        void discharge(int u);
        void minCutPushRelabel(int source, int sink);

        void minCutBK(int source, int sink);
        int augment(Edge *e);
        int treeCap(const Edge& e, Color col) const;
        int treeOrigin(int u, int &len) const;
        void adopt();
        Edge *grow();
};
```

A.7 graph.cpp

```
#include <iostream>
#include <queue>
#include <stack>
#include <vector>
#include <algorithm>
#include <cassert>
#include <cstdlib>

#include "graph.hpp"

using namespace std;

/* Add an edge from one vertex to another. */
void FlowGraph::addEdge(int from, int to, int cap) {
        G[from].e.push_back(Edge(from, to, cap, G[to].e.size()));
        if (from == to) G[from].e.back().index++;
        int index = G[from].e.size() - 1;
        G[to].e.push_back(Edge(to, from, 0, index));

        if (from == source)
                G[to].si = index;

        if (to == sink)
                G[from].ti = index;
}

/*
 * Add an edge and at the same time an antiparallel edge
 * with the same capacity.
 */
void FlowGraph::addDoubleEdge(int from, int to, int cap) {
        G[from].e.push_back(Edge(from, to, cap, G[to].e.size()));
        G[to].e.push_back(Edge(to, from, cap, G[from].e.size() - 1));
}

#ifdef PUSH_RELABEL

/*
 * Change the capacity of an edge. Need the from-vertex and
 * the index of the edge in its edge list (returned from addEdge.
 */
void FlowGraph::changeCapacity(int from, int to, int cap) {
        int index;
        if (from == source) {
                index = G[to].si;
        } else if (to == sink) {
                index = G[from].ti;
        } else {
                exit(1);
        }

        int diff = G[from].e[index].flow - cap;

        G[from].e[index].cap = cap;

        /* Check if we need to reduce the flow. */
        if (diff > 0) {
                G[from].excess += diff;
                G[to].excess -= diff;
                G[from].e[index].flow = cap;
                G[to].e[G[from].e[index].index].flow = -cap;
                rule.add(from, G[from].height, G[from].excess);
        }
}

#else

/*
 * Change the capacity of an edge. Need the from-vertex and
 * the index of the edge in its edge list (returned from addEdge)
 */
void FlowGraph::changeCapacity(int from, int to, int cap) {
        int index;
        if (from == source) {
                index = G[to].si;
        } else if (to == sink) {
                index = G[from].ti;
        } else {
                exit(1);
        }

        /* Nodes in the S set can not send any more flow anyways. */
        if (from == source && G[to].c == SOURCE)
                return;

        G[from].e[index].cap = cap;

        if (from != source)
                return;

        /* Push flow along two-edged path. */
        int si = G[to].si;
        int ti = G[to].ti;

        Edge *sv, *vt;
        sv = &G[source].e[si];
        vt = &G[to].e[ti];

        int rs = sv->cap - sv->flow;
        int rt = vt->cap - vt->flow;

        if (rs > 0 && rt > 0) {
                int m = min(rs, rt);
                push(*sv, m);
                push(*vt, m);

                if (m == rs && G[to].p == sv) {
                        G[to].p = NULL;
                        orphans.push(to);
                }

                if (m == rt && G[to].p == vt) {
                        G[to].p = NULL;
                        orphans.push(to);
                }
        }

        /* Activate vertices if the new edge was not saturated. */
        if (G[from].e[si].flow != G[from].e[si].cap) {
                if (!G[from].active) {
                        bkq.push(from);
                        G[from].active = true;
                }
                if (!G[to].active) {
                        bkq.push(to);
                        G[to].active = true;
                }
        }
}

#endif

/* Reset all flow and excess. */
void FlowGraph::resetFlow() {
        for (size_t i = 0; i < G.size(); ++i) {
                for (size_t j = 0; j < G[i].e.size(); ++j) {
                        G[i].e[j].flow = 0;
                }
                G[i].excess = 0;
        }
}

/* Reset all distance labels. */
void FlowGraph::resetHeights() {
        for (size_t i = 0; i < G.size(); ++i) {
                G[i].height = 0;
        }
        fill(count.begin(), count.end(), 0);
}

/* Push along an edge. */
void FlowGraph::push(Edge &e) {
        int flow = min(e.cap - e.flow, G[e.from].excess);
        G[e.from].excess -= flow;
        G[e.to].excess   += flow;
        e.flow += flow;
        G[e.to].e[e.index].flow -= flow;

        rule.add(e.to, G[e.to].height, G[e.to].excess);
}

/* Push given flow along an edge. */
void FlowGraph::push(Edge &e, int f) {
        e.flow += f;
        G[e.to].e[e.index].flow -= f;
}

/* Relabel a vertex. */
void FlowGraph::relabel(int u) {
        count[G[u].height]--;
        G[u].height = 2*N;

        for (size_t i = 0; i < G[u].e.size(); ++i) {
                if (G[u].e[i].cap > G[u].e[i].flow) {
                        G[u].height = min(G[u].height, G[G[u].e[i].to].height + 1);
                }
        }

        if (G[u].height >= N) {
                G[u].height = N;
        }
        else {
                count[G[u].height]++;
                rule.add(u, G[u].height, G[u].excess);
        }
}

/* Relabel all vertices over the gap h to label N. */
void FlowGraph::gap(int h) {
        for (size_t i = 0; i < G.size(); ++i) {
                if (G[i].height < h) continue;
                if (G[i].height >= N) continue;

                rule.deactivate(i);

                count[G[i].height]--;
                G[i].height = N;
        }

        rule.gap(h);
}

/* Discharge a vertex. */
void FlowGraph::discharge(int u) {
        size_t i;
        for (i = 0; i < G[u].e.size() && G[u].excess > 0; ++i) {
                if (G[u].e[i].cap > G[u].e[i].flow
                                && G[u].height == G[G[u].e[i].to].height + 1) {
                        push(G[u].e[i]);
                }
        }

        if (G[u].excess > 0) {
                /* Check if a gap will appear. */
                if (count[G[u].height] == 1)
                        gap(G[u].height);
                else
                        relabel(u);
        }
}

/* Run the push-relabel algorithm to find the min-cut. */
void FlowGraph::minCutPushRelabel(int source, int sink) {
        G[source].height = N;

        rule.activate(source);
        rule.activate(sink);

        for (size_t i = 0; i < G[source].e.size(); ++i) {
                G[source].excess = G[source].e[i].cap;
                push(G[source].e[i]);
        }
        G[source].excess = 0;

        int c = 0;
        /* Loop over active nodes using selection rule. */
        while (!rule.empty()) {
                c++;
                int u = rule.next();
                discharge(u);
        }

        /* Output the cut based on vertex heights. */
        for (size_t i = 0; i < cut.size(); ++i) {
                cut[i] = G[i].height >= N;
        }
}

/* The capacity of the edge, in the direction given by the tree. */
int FlowGraph::treeCap(const Edge& e, Color col) const {
        if (col == SOURCE)
                return e.cap - e.flow;
        else if (col == SINK)
                return G[e.to].e[e.index].cap - G[e.to].e[e.index].flow;
        else
                return 0;
}

/* Try to grow the trees S and T from the active vertices. */
Edge *FlowGraph::grow() {
        while (!bkq.empty()) {
                int p = bkq.front();
                if (!G[p].active) {
                        bkq.pop();
                        continue;
                }

                size_t i = 0;

                /*
                 * If we're growing from the same vertex as before,
                 * continue with the same index, if not restart at 0.
                 */
                if (lastGrowVertex == p)
                        i = lastIndex;

                lastGrowVertex = p;

                for (; i < G[p].e.size(); ++i) {
                        lastIndex = i;
                        Edge *e = &G[p].e[i];
                        int q = e->to;

                        if (G[p].c == G[q].c)
                                continue;

                        if (treeCap(*e, G[p].c) <= 0)
                                continue;

                        if (G[q].c == FREE) {
                                /* Found a free vertex, add it. */
                                G[q].c = G[p].c;

                                int len;
                                if (G[p].c == SOURCE) {
                                        G[q].p = e;
                                        assert(treeOrigin(p, len) == source);
                                } else if (G[p].c == SINK) {
                                        G[q].p = &G[q].e[e->index];
                                        assert(treeOrigin(p, len) == sink);
                                } else {
                                        cout << G[p].c << endl;
                                        exit(1);
                                }
                                (void)len;
```

```cpp

                                G[q].active = 1;
                                bkq.push(q);
                        }
                        else if (G[q].c != G[p].c) {
                                /* The trees meet! */
                                if (G[p].c == SOURCE) {
                                        return e;
                                }
                                else if (G[p].c == SINK) {
                                        return &G[q].e[e->index];
                                }
                                else {
                                        exit(1);
                                }

                                return NULL;
                        }
                }

                bkq.pop();
                G[p].active = 0;
        }

        /* Path is empty */
        return NULL;
}

/* Augment along path given by the edge e, and implicitly by the trees. */
int FlowGraph::augment(Edge* e) {
        int m = e->cap - e->flow;

        /* Find maximum flow we can send. */
        Edge *cur = e;
        while (cur != NULL) {
                m = min(m, cur->cap - cur->flow);
                cur = G[cur->from].p;
        }

        cur = e;
        while (cur != NULL) {
                m = min(m, cur->cap - cur->flow);
                cur = G[cur->to].p;
        }

        cur = e;
        bool back = true;
        int len = 0;
        /* Loop through path and update flow. */
        while (cur != NULL) {
                /* If saturated, we must orphanize. */
                if (cur->cap - cur->flow == m) {
                        int u = cur->from;
                        int v = cur->to;

                        if (G[u].c == SOURCE && G[v].c == SOURCE) {
                                if (v != source && v != sink) {
                                        orphans.push(v);
                                        G[v].p = NULL;
                                }
                        }
                        if (G[u].c == SINK && G[v].c == SINK) {
                                if (u != source && u != sink) {
                                        orphans.push(u);
                                        G[u].p = NULL;
                                }
                        }
                }
                len++;
                push(*cur, m);

                /*
                 * If we reach the source, we must start again
                 * in e, and go towards the sink.
                 */
                if (back) {
                        cur = G[cur->from].p;
                        if (cur == NULL) {
                                back = false;
                                cur = G[e->to].p;
                        }
                } else {
                        cur = G[cur->to].p;
                }
        }
        return len;
}

/* Find the origin of vertex u. */
int FlowGraph::treeOrigin(int u, int &len) const {
        int cur = u;
        len = 0;

        if (G[cur].c == SOURCE) {
                while (G[cur].p != NULL) {
                        cur = G[cur].p->from;
                        len++;
                }
        } else if (G[cur].c == SINK) {
                while (G[cur].p != NULL) {
                        cur = G[cur].p->to;
                        len++;
                }
        } else {
                exit(1);
        }

        return cur;
}

/* Adopt orphans. */
void FlowGraph::adopt() {
        while (orphans.size() > 0) {
                int u = orphans.front();
                orphans.pop();

                assert(G[u].c != FREE);

                int minlen = 1000000000;
                int minidx = -1;
                /* Aim to find parent close to the root of the tree. */
                for (size_t i = 0; i < G[u].e.size(); ++i) {
                        int v = G[u].e[i].to;

                        if (G[u].c != G[v].c)
                                continue;

                        if (treeCap(G[v].e[G[u].e[i].index], G[u].c) <= 0)
                                continue;

```

```
				int len;
				int origin = treeOrigin(v, len);
				if (origin != source && origin != sink)
					continue;

				if (len < minlen) {
					minlen = len;
					minidx = i;
				}
				if (minlen <= 2) break;
				/* Found a possible parent */
			}

			bool found = false;
			if (minidx != -1) {
				int i = minidx;
				int v = G[u].e[i].to;
				int len;
				int origin = treeOrigin(v, len);

				if (origin == source) {
					G[u].p = &G[v].e[G[u].e[i].index];
					found = true;
				} else if (origin == sink) {
					G[u].p = &G[u].e[i];
					found = true;
				} else {
					exit(1);
				}
			}

			/* If not found, free vertex, and orphanize possible children. */
			if (!found) {
				for (size_t i = 0; i < G[u].e.size(); ++i) {
					int v = G[u].e[i].to;

					if (G[u].c != G[v].c)
						continue;

					if (treeCap(G[v].e[G[u].e[i].index], G[v].c) > 0) {
						G[v].active = true;
						bkq.push(v);
					}

					if (v == source || v == sink)
						continue;

					if (G[v].p
							&& (G[v].p->to == u
							||  G[v].p->from == u)) {
						orphans.push(v);
						G[v].p = NULL;
					}
				}

				G[u].c = FREE;

				G[u].active = false;
				/* We might still have u in the queue */
			}
		}
}

void FlowGraph::minCutBK(int source, int sink) {
	lastGrowVertex = -1;
```

```
        adopt();

        int numpaths  = 0;
        double totlen = 0;
        while (true) {
                Edge *e;
                e = grow();

                if (e == NULL) {
                        /* Empty path. */
                        break;
                }

                totlen += augment(e);
                numpaths++;
                adopt();
        }

        cout << "Avg length: " << double(totlen) / double(numpaths) << endl;

        int size1 = 0, size2 = 0;
        for (size_t i = 0; i < cut.size(); ++i) {
                if (G[i].c == SOURCE) size1++;
                else if (G[i].c == SINK) size2++;
                cut[i] = G[i].c == SOURCE;
        }
        cout << "Inbetweeners: " << cut.size() - size1 - size2 << endl;
}
```

A.8 selectionrule.hpp

Highest level and FIFO selection rules for the push-relabel algorithm.

```
#pragma once

#include <iostream>
#include <queue>
#include <vector>
#include <exception>

class EmptyQueueException : public std::exception {
        virtual const char* what() const throw()
        {
                return "Empty Queue Exception";
        }
};

class SelectionRule {

protected:
        int N;

private:
        std::vector<char> active;

public:
        virtual int next(void) = 0;
        virtual void add(int u, int height, int excess) = 0;
        virtual bool empty(void) = 0;
        virtual void gap(int h) = 0;

        void activate(int u) { active[u] = 1; }
        void deactivate(int u) { active[u] = 0; }
        bool isActive(int u) { return active[u]; }

        SelectionRule(int N) : N(N), active(N) {}

        virtual ~SelectionRule() {}
};

class HighestLevelRule : virtual public SelectionRule {

private:
        int highest;
        std::vector<std::queue<int> > hq;

        void updateHighest(void);

public:
        virtual int next(void);
        virtual void add(int u, int height, int excess);
        virtual bool empty(void);
        virtual void gap(int h);

        HighestLevelRule(int N) : SelectionRule(N), highest(-1), hq(N) {}
};

class FIFORule : virtual public SelectionRule {

private:
        std::queue<int> q;

public:
        virtual int next(void);
        virtual void add(int u, int height, int excess);
        virtual bool empty(void);
        virtual void gap(int h);

        FIFORule(int N) : SelectionRule(N) {}
};
```

A.9 selectionrule.cpp

```
#include "selectionrule.hpp"

using namespace std;

void HighestLevelRule::gap(int h) {
        highest = h - 1;
        updateHighest();
}

void HighestLevelRule::updateHighest(void) {
        int s = highest;
        highest = -1;

        for (int i = s; i >= 0; --i) {
                if (hq[i].size() > 0) {
                        highest = i;
                        break;
                }
        }
}

```

```
bool HighestLevelRule::empty(void) {
        return highest < 0;
}

int HighestLevelRule::next(void) {
        if (empty()) throw EmptyQueueException();

        int u = hq[highest].front();
        hq[highest].pop();
        deactivate(u);

        updateHighest();

        return u;
}

void HighestLevelRule::add(int u, int height, int excess) {
        if (isActive(u)) return;
        if (height >= N) return;
        if (excess == 0) return;

        hq[height].push(u);
        if (height > highest) highest = height;
}

void FIFORule::gap(int h) {
}

int FIFORule::next(void) {
        int u;

        if (!q.empty()) {
                u = q.front();
                q.pop();
                deactivate(u);
        }
        else {
                throw EmptyQueueException();
        }

        return u;
}

void FIFORule::add(int u, int height, int excess) {
        if (isActive(u)) return;
        if (height >= N) return;
        if (excess == 0) return;

        activate(u);
        q.push(u);
}

bool FIFORule::empty(void) {
        return q.empty();
}
```

A.10 neighborhood.hpp

Neighborhood class that also calculates angular differences.

```
#pragma once

```

```
#include <iostream>
#include <vector>
#include <cmath>

class Coord {
public:
        int x;
        int y;
        int w;
        mutable double dt;

        Coord() : x(0), y(0), w(0), dt(0.0) {}
        Coord(int x, int y, int w) : x(x), y(y), w(w), dt(0.0) {}

        double angle() {
                return atan2(y, x);
        }
};

class CoordCompare {
public:
        bool operator()(const Coord& lhs, const Coord& rhs) {
                double p1 = copysign(1.0 - lhs.x/(fabs(lhs.x) + fabs(lhs.y)), lhs.y);
                double p2 = copysign(1.0 - rhs.x/(fabs(rhs.x) + fabs(rhs.y)), rhs.y);

                return p1 < p2;
        }
};

class Neighborhood {
private:
        std::set<Coord, CoordCompare> v;

public:
        void add(int x, int y, int w) {
                v.insert(Coord(x, y, w));
        }

        void add8(int x, int y, int w) {
                v.insert(Coord( x, y, w));
                v.insert(Coord(-x, y, w));
                v.insert(Coord( x,-y, w));
                v.insert(Coord(-x,-y, w));
                v.insert(Coord( y, x, w));
                v.insert(Coord(-y, x, w));
                v.insert(Coord( y,-x, w));
                v.insert(Coord(-y,-x, w));
        }

        std::set<Coord, CoordCompare>::iterator begin() { return v.begin(); }
        std::set<Coord, CoordCompare>::iterator end() { return v.end(); }
        std::set<Coord, CoordCompare>::reverse_iterator rbegin() { return v.rbegin(); }
        std::set<Coord, CoordCompare>::reverse_iterator rend() { return v.rend(); }
        size_t size() { return v.size(); }

        typedef std::set<Coord, CoordCompare>::iterator iterator;

        void setupAngles() {
                for (Neighborhood::iterator it = this->begin();
                                it != this->end();
                                ++it) {
                        Coord prev;
                        Coord next;

                        if (it == this->begin()) {
```

```
                                        prev = *(this->rbegin());
                                } else {
                                        prev = *(--it);
                                        it++;
                                }

                                if (++it == this->end()) {
                                        next = *(this->begin());
                                } else {
                                        next = *it;
                                }
                                it--;

                                it->dt = next.angle() - prev.angle();
                                while (it->dt > 2.0 * M_PI)
                                        it->dt -= 2.0 * M_PI;
                                while (it->dt < 0.0)
                                        it->dt += 2.0 * M_PI;
                                it->dt /= 2.0;
                        }
                }
};
```