Bachelor's Thesis

Framework zur computergestützten Analyse von Bildern vaskulärer Netzwerke

Computational Analysis Framework for Vascular Network Images

presented by

Jana Lasser

from Graz, Österreich

at the Max-Planck-Institut für Dynamik und Selbstorganisation

Thesis period:	17th December 2012 until 25th March 2013
First Referee:	Eleni Katifori, Ph.D.
Second Referee:	Prof. Dr. Sarah Köster

Abstract

In this thesis, we develop a framework for the processing of high resolution images of vascular networks. The framework processes high resolution scans of plant leaves and extracts the topological and geometric information of the vein networks present on the leaves. This framework consists of modular Python scripts, modules and classes. The main goal of the work is to provide as fast and highly automated processing of the data as possible while preserving the properties of the networks. The resulting framework can be used cross-platform and on PCs as well as on clusters. With a few adaptations it can be a good basis for the processing of images from many networks in different biological systems.

Keywords: Biological Networks, Image Processing, Python

Abstract

Das Ziel dieser Arbeit ist die Entwicklung eines Frameworks zur Datengewinnung aus hochauflösenden Bildern vaskulärer Netzwerke. Das Framework verarbeitet hochauflösende Scans von Pflanzenblättern, um Informationen über die Topologie und Geometrie der darauf vorhandenen Transportnetzwerke zu gewinnen. Das Framework ist modular aus Python-Skripten, -Modulen und -Klassen zusammengesetzt. Es soll den Nutzer in die Lage versetzen, so automatisiert wie möglich Netzwerkdaten auszulesen, während die Eigenschaften des Netzwerkes weitestgehend erhalten bleiben. Augenmerk liegt auch auf der Plattform- und Hardwareunabhängigkeit des Frameworks, welches, mit einigen Anpassungen, auch als Werkzeug zur Gewinnung von Daten aus Bildern anderer biologischer Netzwerke verwendet werden kann.

Stichwörter: Biologische Netzwerke, Bildverarbeitung, Python

Contents

1. Introduction

1.1. Motivation and Overview

When looking at biological systems, networks are omnipresent. The blood vessels which supply animal and human bodies with nutrients and oxygen, the vascular networks in plants or neuronal networks responsible for the transport of information - networks are the backbone of life. Understanding the topology, geometry and formation of these networks is a main goal of research in biological physics.
The information that can be obtained by understanding biological networks is diverse and ranges from the search for the most efficient way of supplying a system with nutrients to the properties of these networks when exposed to extreme circumstances or inflicted with defects. Understanding how a plant can keep alive a leaf even with a deep cut dividing parts of that leaf can have significant impact on the way we build our transportation and information networks. This results in making them more efficient and more resistant against disturbances.
This work focuses on the networks present in leaves and their characteristics. The goal is to model these networks as accurately as possible to allow analysis of their reaction to defects and dependence on various parameters. To reach that goal, network models need to be supplied with data from natural leaves. Vein networks of leaves can be digitized by scanning the leaf the network belongs to.
Before the leaf is scanned, it is prepared to improve the visibility of its vein network. One way of preparing the leaf is *clearing* it by submerging it into several chemicals until the tissue material between the veins is transparent. The veins will also become transparent after such processing, so the samples are treated (commonly with acid fuchsin) to stain the veins. Clearing and staining the leaves greatly improves the contrast between vein and no-vein regions but most scans still have defects and shadows from non-uniform staining on them. This further complicates recognition of the network. An alternative way of preprocessing leaves is the removal of all tissue material that is not part of the vein network, thus *skeletonizing* the leaf. On scans of skeletonized leaves, the veins are clearly

visible and there are no issues with non-uniform stains, but the highest branching orders are destroyed in the process. When the surrounding tissue is removed, the locations of the different network branches are no longer preserved and the appearance of the network is distorted.
As we want to simulate networks as lifelike as possible, we use scans of cleared leaves to ensure the integrity of the vein network. The samples were provided to us by the New York Botanical gardens and our collaborator Douglas Daly. Thus the need arises to correct the scans for non-uniform staining and improve the network's visibility before information about its properties can be extracted.
The databases of cleared leaves that already exist, for example [15], are very large and contain thousands of scans. Therefore a tool for data extraction from all those scans needs to be able to operate in a highly automated way, independent of user input with regards to network extraction parameters, such as thesholding values.
The aim of this work is to create a tool that is able to handle scans with very high resolutions as well as large amounts of data, improve the scans in a way that the network becomes clearly visible and extract the information that represents the network. A framework that offers the possibility for automated processing of images and creation of data fit for modeling can be adopted to a variety of fields that obtain their experimental data from images.
The written part of this thesis will serve as documentation for the framework so that new users or developers are able to become quickly acquainted with image processing and the algorithms used.

1.2. State of Research

Attempts to extract data from images of biological networks already have been made with various aims concerning usability, quality, scale of data and speed.
A graphical user interface (GUI) for enhancement and processing of images of leaves [7] does already exist, but its possibilities are rather limited where automated processing of large amounts of scans and images with a very high resolution is concerned. A commonly encountered technique for dealing with processing issues concerning connectivity of veins near the main vein or high branching orders is simply discarding those regions (as has been done in the approaches described in [3] and [2]). This is not the path we follow for this framework as our interest lies especially in the leaves as whole systems with high branching orders where conduction of water and nutrients involves multiple scales. The

approach outlined in [6] uses skeletonized leaves which is not viable for our purposes as explained above.
After due consideration of the existing approaches and their drawbacks, it was decided to create a new framework to improve on the weaknesses of the already available tools and to have an adaptable and GPL-licensed framework at hand.

1.3. Framework Requirements

In the following the main considerations for the development of a framework able to extract network data from digital images will be outlined.

Quality

The first and most important goal is the extraction of data with *high quality*. During the enhancement and processing steps, we will try to preserve as much data from as high branching orders as possible. When encountering difficulties with network recognition, we will not discard small details; and to get information about even the highest orders of branches, digital image enhancement methods will be used.

Automation

We want to be able to extract the network data from whole libraries of cleared leaf scans which consist of thousands of images. Therefore, the framework needs to be as independent from user input at the processing time as possible. Methods developed for the framework have to be applicable to a wide range of images, may they be small or large, bright or dark, with high or low contrast.

Efficiency

As the images themselves have a high resolution and therefore their data files are very large, and the amount of images that will be processed is very high, time efficiency is an issue. Quality should not be sacrificed for efficiency, but for the computationally demanding parts of the algorithm, ways to improve performance such as parallelizing or outsourcing of calculations to lower level programming languages are actively searched for.

Usability

For the selection of the language the framework is written in, platform independence as well as hardware independence and availability of public licenses have to be ensured. The framework is intended to be a solid foundation for image processing and data extraction and it is very important that it is expandable and adaptable to other problems and questions. This is not the case for software written in licensed languages such as MATLAB, because to use and expand such a tool, for every machine it is installed on a separate license has to be bought. As this is not feasible in most cases, *Python* is the language of choice. To make the framework adaptable to different hardware environments (especially concerning the random access memory the individual computer has at its disposal), options to dynamically adapt the memory consumption are needed. The target hardware is commonly used desktop computers and the framework is designed with their limited CPU and RAM capacity in mind. All used languages and libraries conform to the GNU General Public License.

1.4. Network Modeling

In the following I will outline one approach to network modeling using the example of the phloem network in plants to identify the data input needed to model real plants.
The phloem network in plants transports sugar away from its production sites in the leaves. We model the network as an assembly of tubes which transport that sugar, solved in water. A point where two different tubes connect is called a *node*. The *water volume current* Q_{ij} through a tube from node i to node j is assumed to be laminar (i.e. the water flows in parallel layers). Under this assumption, in a network with n nodes, the water flow can be described by a system of n^2 *Hagen-Poiseuille equations*

$$Q_{ij} = \frac{C_{ij}}{\ell_{ij}} \cdot (p_j - p_i) \ ,$$

where $C_{ij} \propto r^4$ is the pipe's *conductivity*, r the pipe's *radius*, ℓ_{ij} the pipe's *length* and p_j the hydrostatic pressure at node j.
This already identifies the three types of information we need to model a network:

- The *topology* of the network, i.e. which node is connected to which other nodes.
- The *geometry* of the network, i.e. the length of the tubes (or equivalently the coordinates of the nodes).
- The *conductivity* of the tubes which is identified with their *width* in phloem networks.

This is the information needed to model the network itself. To model transport in the network, we can then designate certain nodes as flow sources (production sites) and sinks, leading to values for the pressures p_i.

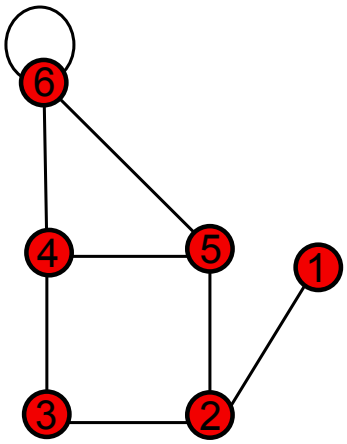


Figure 1.1.: Network with 6 nodes.

A network is represented by a *graph* consisting of *nodes* and *edges* (i.e. connections between nodes). The topology of a graph often is represented by its *adjacency matrix*. The adjacency matrix of a network with n nodes is a $n \times n$ matrix where the diagonal entry a_{ii} is the number of edges from the node to itself (which is usually zero for phloem networks) and the non-diagonal entry a_{ij} is the number of edges from node i to node j. For phloem networks, each node is usually connected to another node by a maximum of one edge, therefore the adjacency matrix for such a network will be a binary *connection matrix*.

The connection matrix A of the small network shown in figure 1.1 is:

$$A = \begin{bmatrix} - & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{6} \\ \mathbf{1} & 0 & 1 & 0 & 0 & 0 & 0 \\ \mathbf{2} & 1 & 0 & 1 & 0 & 1 & 0 \\ \mathbf{3} & 0 & 1 & 0 & 1 & 0 & 0 \\ \mathbf{4} & 0 & 0 & 1 & 0 & 1 & 1 \\ \mathbf{5} & 0 & 1 & 0 & 1 & 0 & 1 \\ \mathbf{6} & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

The *conductivities* of the tubes can be included in the connection matrix, thus creating a *weighted* connection matrix where the entry a_{ij} is the conductivity between nodes i and j. A conductivity of zero signifies that nodes i and j are not connected.

The geometry of the network can be described by the *distance matrix* where the entry a_{ij} is the distance between nodes i and j. Therefore, if the weighted connection matrix and the distance matrix of a network are known, it can be described with the model outlined above.

To create those two matrices from an image of the network, we need to identify the parts of the image that belong to the network and then recognize whether a part of the network is node or edge. This, and the handling of big data sets consisting of very large images are the tasks of the framework.

1.5. Framework Structure and Design

The structure of the framework can be divided into three main parts: image handling, image enhancement and binary processing. The framework therefore is organized in three Python modules that provide functions and custom classes for the tasks the framework has to fulfill. Also included in the framework several scripts that combine the methods provided in the modules to efficiently handle large images, digitally enhance them, vectorize the features present in the images and finally extract the network data.
The course of this thesis will follow the outline of the framework's design, describing theoretical background as well as actual implementation. The description will follow the order naturally given by the way an image passes through the framework, from the raw scan at the beginning to the data extraction at the end.
It has to be emphasized that the result of this work is the *framework* itself and not the data it will eventually produce. As this thesis is a software project, a lot of time was spent on quality assurance and documentation. In the appendix, a complete documentation of the three modules (`tile_functions` (appendix A), `filters` (appendix B) and `triangulation_functions` (appendix C)) representing the implementation of each of the three main parts of the framework can be found.
The main body of the thesis exceeds the standard length of 40 pages. This is due to the inclusion of many pictures. As this is a work about image processing, those pictures are necessary to illustrate the operating principles of the methods introduced.

2. Image Handling

2.1. Language and Libraries

The language the framework is written in is Python. This language was chosen because of its four main advantages which help fulfill the requirements stated in chapter 1.3:

- Scriptability, which supports high development activity.
- Extendability, especially with C and C++ , which helps improve efficiency when used in parts of the framework which do computationally intensive work.
- Platform independence; the standard installation for the language is freely available for all major platforms. This ensures portability and the possibility to modify and adapt the framework.
- The Python Software Foundation (PSF) license is compatible with the GNU Public License. Other languages such as MATLAB require expensive licenses that limit portability.

Python's standard library, although already extensive, is expanded by various third-party libraries, called *modules*, most of them especially designed for scientific computing. The most important ones used in this framework are:

- `NumPy` and `SciPy` [14] for array operations, linear algebra and access to the Python Imaging Library (PIL)
- `PyTables` [1] for reading and writing image data
- `OpenCV` [4] for advanced image processing and computer vision algorithms
- `PyGame` [20] for visualization
- `poly2tri` [10] for triangulations
- `Sphinx` [5] for documentation

2.2. Digital Image Representation

A *raster digital image* is a two-dimensional ordered accumulation of discrete pixels - a matrix. It is defined by its resolution, number of channels and depth. The number of pixels per length is called the *resolution* of the image. The scans that are processed with this framework have a resolution of 6400 dpi (dots per inch) which allows identification of up to five orders of vein-branches (beginning with the main vein as 0th order). Figure 2.1 illustrates how much information an image with such a resolution holds by showing a series of zoomed in details of a raw scan, each subsequent detail enlarged by a factor of four. The image quality is ultimately only limited by the optical resolution of the scanner.

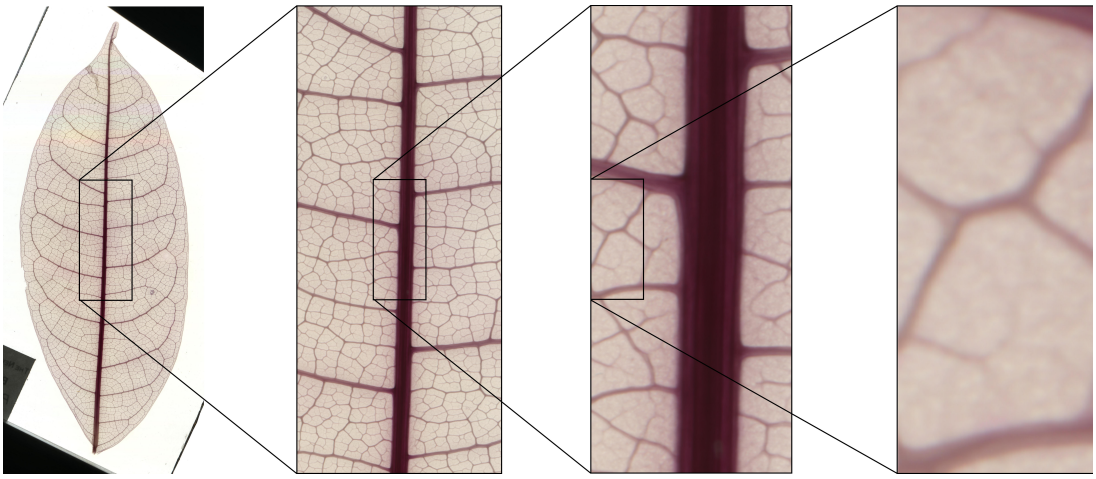


Figure 2.1.: Successively zoomed in details of a leaf scan.

A grayscale image has one single *channel* in which information is stored. A color image usually has three channels.
The range of different values each pixel can hold is called *depth* of the image. During the processing within our framework, three different types of images will be used:

- Color images hold visible color information. The scans natively are color images and are converted to grayscale in the first step of the processing. The scans have a depth of 24 bits which corresponds to $2^8 = 256$ values per pixel for each of the three color channels.

- Grayscale images only hold the brightness information of the image, this requires one single channel which has a depth of 8 bits per pixel in the images we will process with the framework.

- Binary images hold information about the affiliation of each pixel to either foreground or background of features present in the image. Binary images therefore consist of only zeros and ones (one bit) stored in one single channel.

For the internal representation of image data NumPy's data type `ndarray` is used which can store an n-dimensional array containing any commonly used number format and supports a wide range of array and linear algebra operations.

2.3. Dealing with Image Size

Processing an image sometimes involves copying it multiple times and holding all copies in the RAM of the computer. As the images for which this framework is designed typically have a size of up to 3 gigabytes, simply duplicating the image will almost inevitably lead to reading and writing image data from and to the hard drive because on most machines, the available RAM is not large enough. This dramatically slows down processing speed and has to be avoided under all circumstances.
As filter-specific dependencies between pixels are, if existent, only local, we can divide the image into different subregions, which will be called *tiles*; the tiles then are processed independently. The parts of the image which are not currently processed are stored on the hard drive and only loaded when needed, to keep RAM-consumption at a minimum. All routines dealing with efficient tiling, loading and storing of images as well as supplying a socket for the insertion of a filter (see section 3) are gathered in the module `tile_functions`.

2.3.1. Image Tiling

When the loading and storing mechanism only needs to supply data for a filter performing point operations (see section 3.1), the image can simply be "cut" into smaller pieces which can be loaded one by one. The task gets more challenging when dealing with neighborhood operations (see section 3.2). When a pixel situated at a tile boundary is processed and not all neighboring pixels are supplied, the result of the filtering will be distorted. Therefore we also need to supply the overlap between the different tiles to be able to process boundary pixels correctly.
For a filter that requires an overlap of width d, when the tiling mechanism loads the first quadratic tile with edge length a, it can only process the region which has at least distance d to the tile's boundary. A slice of width $2 \cdot d$ is stored for the right and lower

boundary of the tile to be able to process the currently left out regions later on (as can be seen in figure 2.2). The processing starts at the upper left corner of the image and iterates through the image from left to right and top to bottom.
The tiling mechanism combines every tile loaded after the first one with one or two slices from the memory (one if the tile is touching a boundary of the image, two otherwise) containing the overlap from previously processed tiles.
When not touching any boundaries, the processible region is quadratic with edge length a but is shifted from the newly loaded tile by d (as can be seen in figure 2.3).
To be able to also process the pixels at the outermost boundaries of the image without too much error, the image is *padded* with a frame of additional pixels of width d. The padding can either consist of the addition of pixels with a constant value, a reflection of the pixels inside the original image or a linear interpolation from the boundary pixels to a constant value. The method A.7 `tile_functions.NewBoundaries()` handles the padding functionality and at the same time also ensures that the image has dimensions fitting to a discrete number of square tiles.

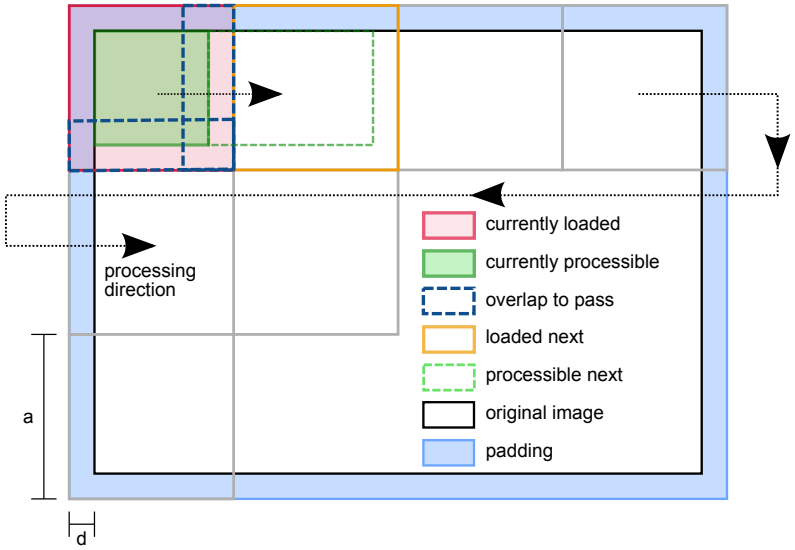


Figure 2.2.: Start of the image processing, beginning at the first tile.

2.3.2. Tiling Implementation

The core of the image tiling implementation is a so-called “Hierarchical Data Format” (in our case HDF5 [11]). The data format supports storing of data in tree-like structures in a file. We do this by creating one node for every line of tiles and then placing every tile under the node as a leaf (the tree-like structure is illustrated in figure 2.4). This ensures that only the data chunk that is currently needed is loaded into memory and makes navigating between different tiles easy. Before we process an image, it is split into different

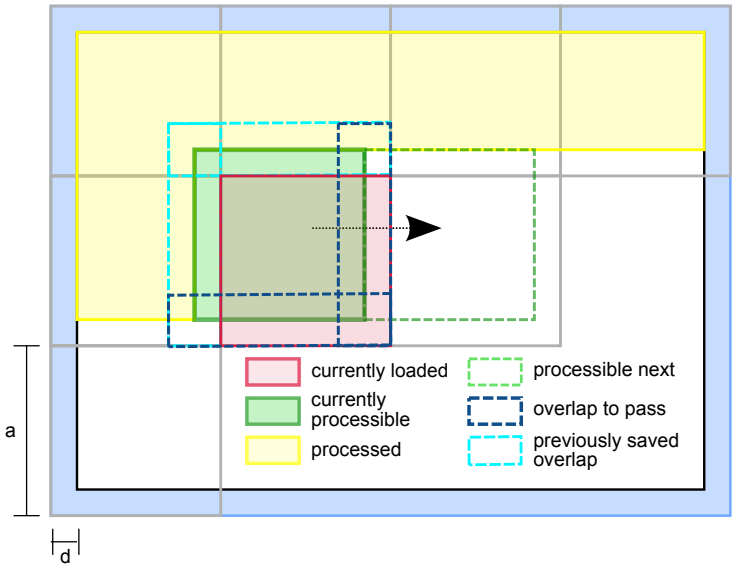


Figure 2.3.: Ongoing iteration showing processing of a tile in the middle of the image.

tiles and stored on the hard drive. For processing, the framework reads every tile one at a time, and stores the result of the processing in another HDF-file for further processing. Dividing the image into chunks and creating an HDF-file can be done by calling the method A.10 `tile_functions.WriteChunkData()`, whereas the method A.4 `tile_functions.getImageFromH5()` handles the recomposition of an image from an HDF-file.

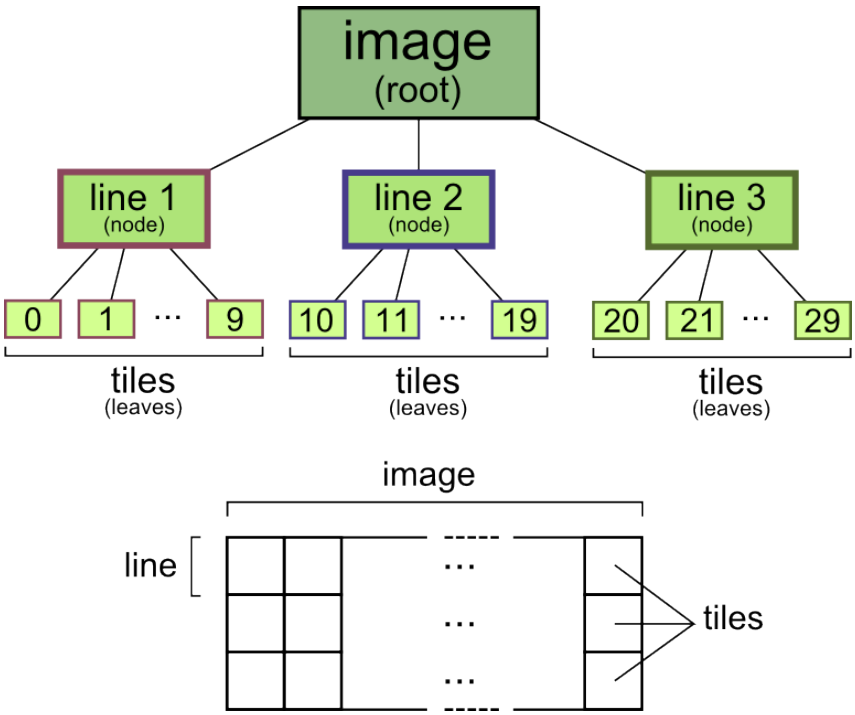
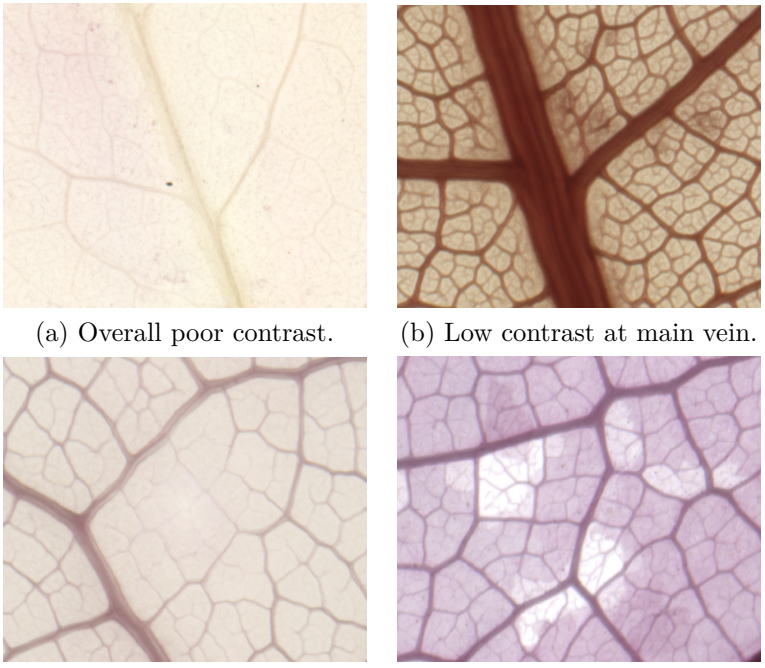


Figure 2.4.: Illustration of a hierarchical data structure for image storing.

3. Image Enhancement

As soon as we have read an image from the hard drive and fed it to the tiling mechanism, the image enhancement itself can be started. The raw scans that we process often are not perfect and parts of the network, especially regions near the main vein and high branching orders, might not be accurately parsed by simple segmentation algorithms. Figures 3.1a to 3.1d show several details of images which illustrate that the vein network often initially is not separable from the background, or where high brightness differences may confuse the processing algorithms. Therefore, before we can extract the network data without too much error, the visibility of the vascular network needs to be increased. The framework offers several filters and methods for image enhancement that can be chosen and combined freely to adapt to a given set of images with characteristic contrast, brightness and defects. All filters are gathered in the `filters`-module documented in B.
In the following I will describe those filters along with the method that worked best for the example images that I processed during the course of the work on this thesis.



(a) Overall poor contrast. (b) Low contrast at main vein.

(c) Regional low contrast. (d) Brightness difference.

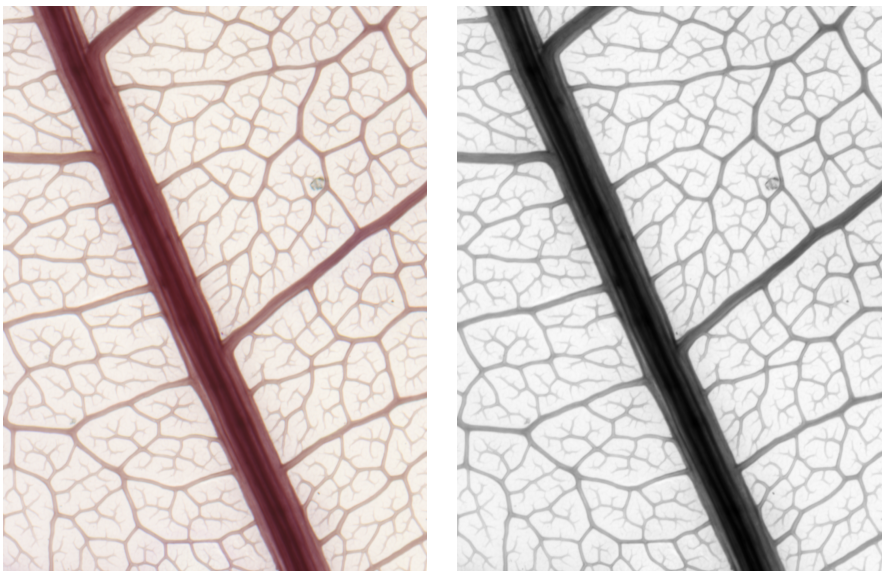
Figure 3.1.: Different occurrences of poor/uneven contrast in raw cleared leaf scans.

3.1. Point Operations

When we talk about *filtering* an image, we need to distinguish two classes of filtering operations: *point operations* and *neighborhood operations*. Point operations are operations on an image that only need information from one single pixel to calculate the resulting pixel. The three main filters implemented in this framework that use point operations are grayscale conversion, histogram equalization and the thresholding filter.

3.1.1. Grayscale Conversion

The scans we process in this framework are color-images. The first thing we need to do before we apply further filters is a conversion from a color-image to a grayscale one. As the color images use the RGB color space for color representation (information about color spaces can be found for example in [19]43), this is done by adding one third of the value of each color channel together to form the *brightness* information of each pixel. Therefore the image now appears in 256 grayscale values. The color information of images could at some point be used to do defect correction for defects that have characteristic color information but at this stage in the framework's development, color information is neglected and the scans are immediately converted to grayscale.



(a) Detail of color-image. (b) Detail after grayscale conversion.

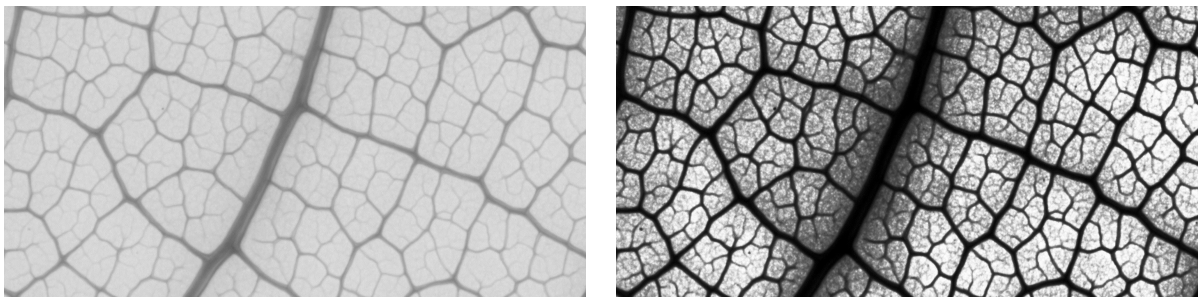
Figure 3.2.: Application of grayscale conversion.

3.1.2. Histogram Equalization

The *histogram* of an image holds information about how many pixels with each color/brightness value are present in the image. When equalizing a histogram globally, it is "stretched" so the whole range of possible values is used.
For each brightness level j in the image, the newly assigned value k is

$$k = \frac{256}{N_{\text{total}}} \cdot \sum_{i=0}^{j} p(i) \ ,$$

where the sum counts the number of pixels in the image (by integrating over each bin of the histogram) with brightness equal to or less than j, normalized by the total number of pixels N_{total}. This enhances the contrast of the image and can make selection of a global threshold for later thresholding easier. The effect of histogram equalization are illustrated in figures 3.3a and 3.3b. As can be seen, subtle brightness differences that are present in the original detail but are not visible to the naked eye are enhanced in the processed detail. However, the overall brightness gradient between vein and no-vein regions is unchanged and it still would be difficult to extract the vein network from this image without disconnected veins or too much noise present.
Histogram equalization is a point operation but in a sense each pixel has a global dependency on each other pixel as the histogram of an image that will be equalized needs to be the histogram of the *whole* image to get a uniform result. As a consequence, we cannot execute the creation of the histogram using the above described tiling mechanism whereas the assignment of new values itself can be executed that way. The framework implements histogram equalization in function B.2 `filters.histeq()`.



(a) Detail with a brightness range of 131 values. (b) Brightness range stretched to 255 values.

Figure 3.3.: Application of global histogram equalization.

3.1.3. Thresholding

Thresholding is used to create a binary image from a grayscale one. When we threshold an image, we select one global value - the threshold -, and map every pixel above it to 1 and every pixel below it to 0, thus distinguishing between foreground and background of the image. For a given threshold T the thresholding function is

$$f_T(x) = \begin{cases} 0 & \text{if } x < T \\ 1 & \text{if } x \geq T \end{cases} .$$

By convention the value "0" (black) is used for foreground pixels and "1" (white) for the background during image enhancement. When we perform operations on a binary image (as described in section 4), this convention is often inverted. The framework implements various thresholding functions which allow thresholding both with a single threshold and within a range of values, as well as mapping to custom values, see for example function B.12 `filters.threshold()`.
While performing the thresholding itself is easy, the selection of a proper threshold can be challenging. When we manually look at an image, after a few iterations of trial and error we are able to find a threshold that does not include too much noise in the foreground selection as well as leaving most of the veins connected. To automate this, the framework implements a method called "Otsu Thresholding" [16].
While searching for the best threshold T we minimize the *intra*-class variance, which is a sum of the weighted variances σ_i of both the foreground and the background class:

$$\sigma^2_{\text{intra}}(T) = \omega_B(T)\sigma^2_B(T) + \omega_F(T)\sigma^2_F(T) .$$

The same result can be achieved by maximizing the *inter*-class variance, using the class means μ_i:

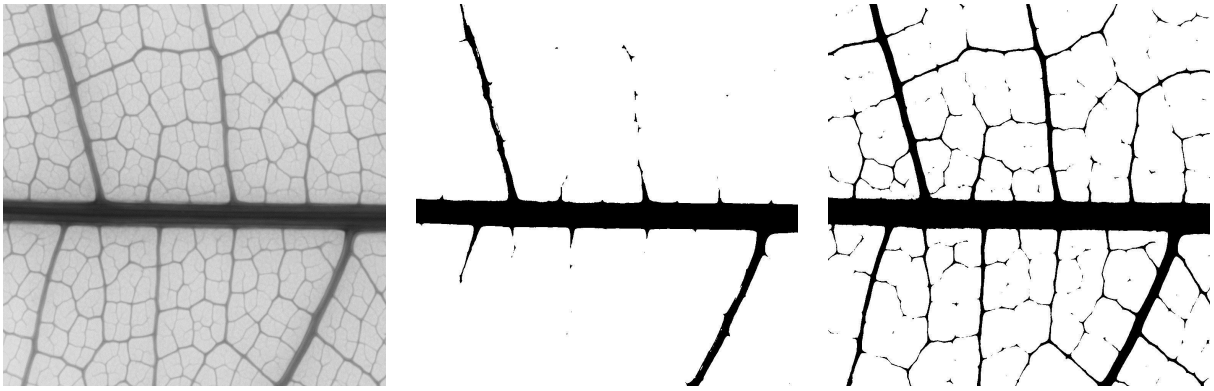
$$\sigma^2_{\text{inter}}(T) = \omega_B(T)\omega_F(T) \cdot [\mu_B(T) - \mu_F(T)]^2$$

The weights and class-means are defined as

$$\omega_B = \sum_{i=0}^{T} p(i) , \qquad \omega_F = 1 - \omega_B$$

$$\mu_B = \left(\sum_{i=0}^{T} p(i) \cdot T \right) \cdot \frac{1}{\omega_B} , \qquad \mu_F = \left(\sum_{i=T+1}^{255} p(i) \cdot T \right) \cdot \frac{1}{\omega_F}$$

where $p(i)$ is the number of pixels in the ith bin of the image's histogram.
As connectedness of veins is very important to be able to later correctly extract the topology of the network and the appearance of noise is acceptable as long as it is disconnected from the network, a constant value c is added to the threshold found by Otsu's method. Trial and error yielded a value of $c = 30$ to be the best choice. This generally works well on images that have a mean brightness in the middle of the spectrum and is implemented in function B.8 `filters.otsu_threshold()`. On images that are either very light or very dark, this can produce rather bad results. This is a problem that cannot really be fixed computationally so the framework gives a warning if it encounters an image with an overall brightness that lies at the outer boundaries of the spectrum and encourages manual threshold selection.
Figures 3.4a to 3.4c show the results of the application of a thresholding filter with two different thresholds applied to a grayscale detail of a raw scan without further processing. It can be seen that in figure 3.4b only the main vein and one branching order is present whereas the retention of higher branching orders is much better in figure 3.4c. Nevertheless, the result is far from perfect. This shows that the images need further processing before the vein network can be extracted properly via thresholding at the end of the image enhancement.



(a) Grayscale detail. (b) Threshold $T_O = 132$. (c) Threshold $T_O + 30 = 162$.

Figure 3.4.: Application of a thresholding filter with two different thresholds to a detail of a grayscale image.

3.2. Neighborhood Operations

Neighborhood operations need information from the neighborhood of the pixel currently processed, taking into account their values for the calculation of the new pixel's value. The majority of neighborhood operations can be described by the discrete *convolution* of the matrix representing the image with a convolution kernel which we calculate by taking the sum of the pixel values in the region of the kernel size, multiplied by weights specified in the convolution kernel (as described in [19]209):

$$P^*_{x,y} = \frac{\sum_{i,j=-m}^{+m} \omega_{i,j} \cdot P_{x+i,y+j}}{\sum_{i,j=-m}^{+m} \omega_{i,j}}$$

The size of the kernel determines the size of the neighborhood which will be taken into account, the values within the kernel determine the properties of the filter. In the following, the filters implemented in the framework that make use of neighborhood operations and their applications will be described. Those filters are a mixture between own implementations and improvements or adaptions of filters that already exist in libraries used for this framework. Some filters are wrappers around filters from `openCV` which set options that best fit the kind of images that are processed and handle the input and output formats.

3.2.1. Blurring

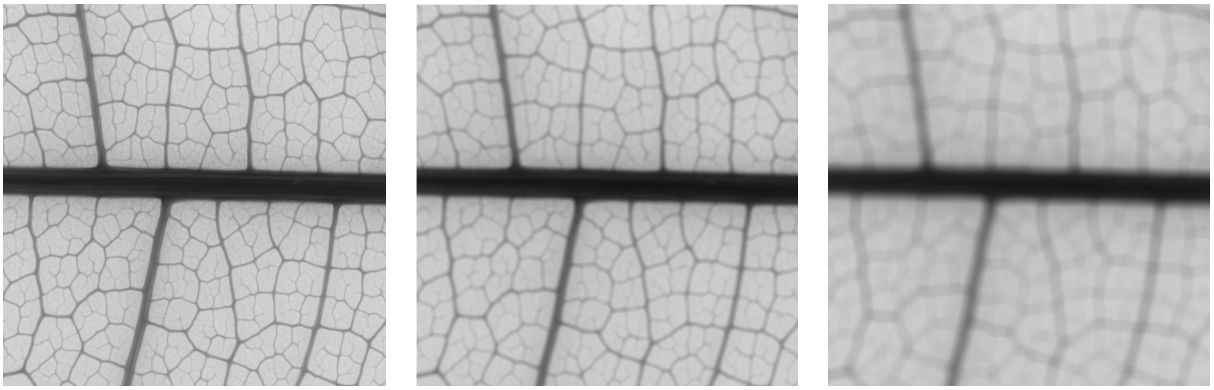
Applying a *blurring* (also called *median*) filter is a convolution with a kernel of the form

$$K = \frac{1}{K_{\text{width}} \cdot K_{\text{height}}} \cdot \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \cdots & 1 \end{bmatrix} .$$

This averages over the values of all neighboring pixels and results in an overall decrease of sharpness, thus blurring the features. The amount of blurring done to the image can be controlled by adjusting the size of the filter kernel. Also, kernel shapes that approximate a circular neighborhood around the processed pixel yield more "natural" results.
There are three functions for blurring filters in the framework:

- Function B.5 **`filters.median_dynamic()`**
- Function B.6 **`filters.nan_median()`**
- Function B.7 **`filters.openCV_median()`**

The `dynamic` filter supports creation and use of kernels with circular shapes, whereas the `nan_median` filter supports the usage of NaN values in the image which specify pixels to be ignored during the processing The `openCV` implementation is generally very fast.
At first glance, one might think that blurring is of limited use for the improvement of contrast and sharpness of the image but this is not the case. Blurring an image with a kernel-size smaller than the smallest features present in the image (in this case the highest branching order) before applying other filters helps in reducing the amount of unwanted noise. The blurring filter also is an essential part of the technique called "Unsharp Masking" [19]288-290 which is described in section 3.3.2. An application of the blurring filter is shown in figures 3.5a to 3.5c.



(a) Grayscale detail. (b) Blurring, kernel size 11. (c) Blurring, kernel size 31.

Figure 3.5.: Application of a blurring filter with two different kernel sizes.

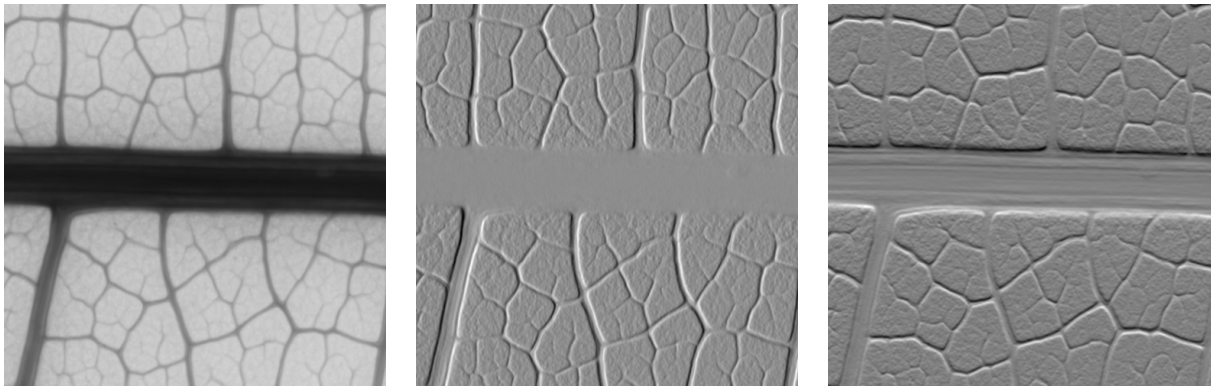
3.2.2. Directional Derivatives

A *directional derivative* filter approximates the directional derivative of the brightness B [19]296. The filter kernels for the horizontal (x) and vertical (y) direction look as follows:

$$\frac{\partial B}{\partial x} \quad \text{corresponds to a convolution with} \quad \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix},$$

$$\frac{\partial B}{\partial y} \quad \text{corresponds to a convolution with} \quad \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}.$$

In the above directional derivative filters, the original central pixel's value does not enter into the calculation at all. Instead the differences are formed between neighbors to the left and right or top and bottom respectively. Therefore brightness changes in one direction are emphasized whereas brightness changes in all other directions are suppressed. The directional derivative filter is implemented in function B.3 `filters.dir_deriv()`. An application of the Directional Derivative filters in x- and y-direction with the addition of a medium gray of value 128 to also make negative values visible is shown in figures 3.6a to 3.6c.



(a) Grayscale detail. (b) Derivative in x-direction. (c) Derivative in y-direction.

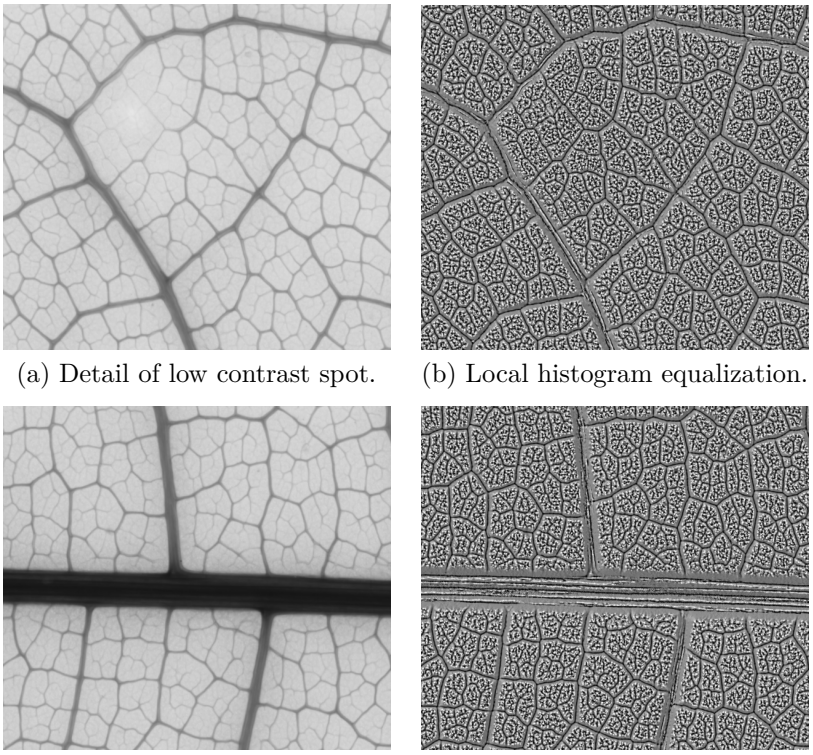
Figure 3.6.: Application of a directional derivative Filter in x- and y-direction.

3.2.3. Local Histogram Equalization

Local histogram equalization is the local counterpart of global histogram equalization (described in section 3.1.2). When performing local histogram equalization, we compare the value of each pixel in a specified neighborhood to the value of the pixel currently processed. The new value is the number of values that are smaller than the processed pixel's value. This cuts down the range of possible values to the number of pixels in the neighborhood but ensures global and brightness independent contrast enhancement. The tradeoff is that this method also enhances noise with the same intensity as it enhances contrast differences between vein and no-vein regions.

As can be seen in figure 3.7b, local histogram equalization works well to enhance visibility of veins in regions with low contrast. The blind spot that can be seen in figure 3.7a vanished after application of the filter. The appearance of noise in between the veins as well as on the main vein can be observed in figure 3.7d. This can be counteracted to some extent by blurring an image with a small blurring kernel before applying local

histogram equalization. The local histogram equalization filter itself can be accessed in the framework via function B.4 **`filters.lochisteq()`**.



(a) Detail of low contrast spot. (b) Local histogram equalization.

(c) Detail near the main vein. (d) Local histogram equalization.

Figure 3.7.: Application of local histogram equalization with an 11×11 neighborhood.

3.3. Filter Combinations

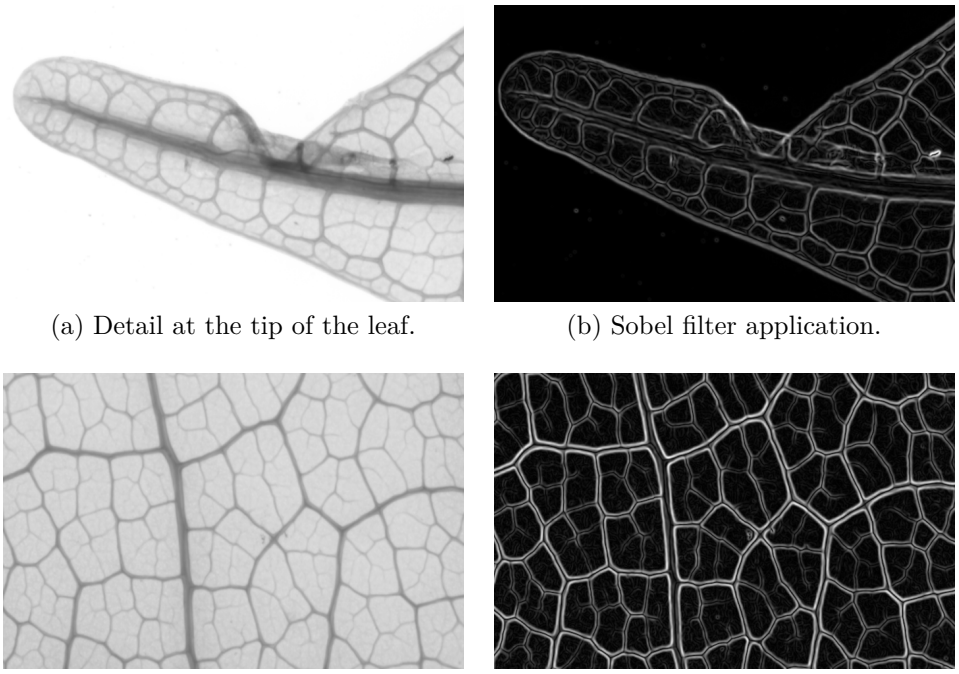
The image enhancement process aims to create an image with as low overall contrast difference and as high local contrast difference between vein and no-vein regions as possible. If we have achieved that, we can select a global threshold value and then create a binary image via application of the thresholding filter which can be used for further processing. The filters introduced in the chapters above, on their own, have very limited possibilities to improve the visibility of veins. To counter the weaknesses of each specific filter and further improve visibility, they need to be combined. Filters can be combined by using them one after another as well as by adding or subtracting images from different processing states to create an enhanced image.
Commonly, the first step in image enhancement is a blurring operation with a relatively small kernel size (smaller than the smallest feature that has to be preserved) to smooth out some of the local noise. Blurring will be applied first in each of the following filter combinations.

3.3.1. Sobel Filter

The *Sobel* filter is based on the directional derivative filter (described in section 3.2.2) and computes the magnitude M of the vector that represents the brightness change, as described in [19]301:

$$M = \sqrt{\left(\frac{\partial B}{\partial x}\right)^2 + \left(\frac{\partial B}{\partial y}\right)^2}$$

A combination of blurring and apllication of the Sobel filter results in an image with greatly emphasized edges and smoothed local details (noise) at the expense of the inner parts of the veins. This goes so far as to keep only the boundaries of the veins when using larger kernel sizes for the Sobel filter. The framework implements the Sobel filter in function B.9 `filters.sobel()` which in turn uses the directional derivative filter.
The fact that only information about the boundaries of the veins is retained gives rise to the problem of how to decide what is *outside* the vein and what is *inside* the vein, a problem which is not trivial to solve. The resulting appearance of contour-like shapes after application of the Sobel filter can be seen in figures 3.8a to 3.8d.



(a) Detail at the tip of the leaf. (b) Sobel filter application.

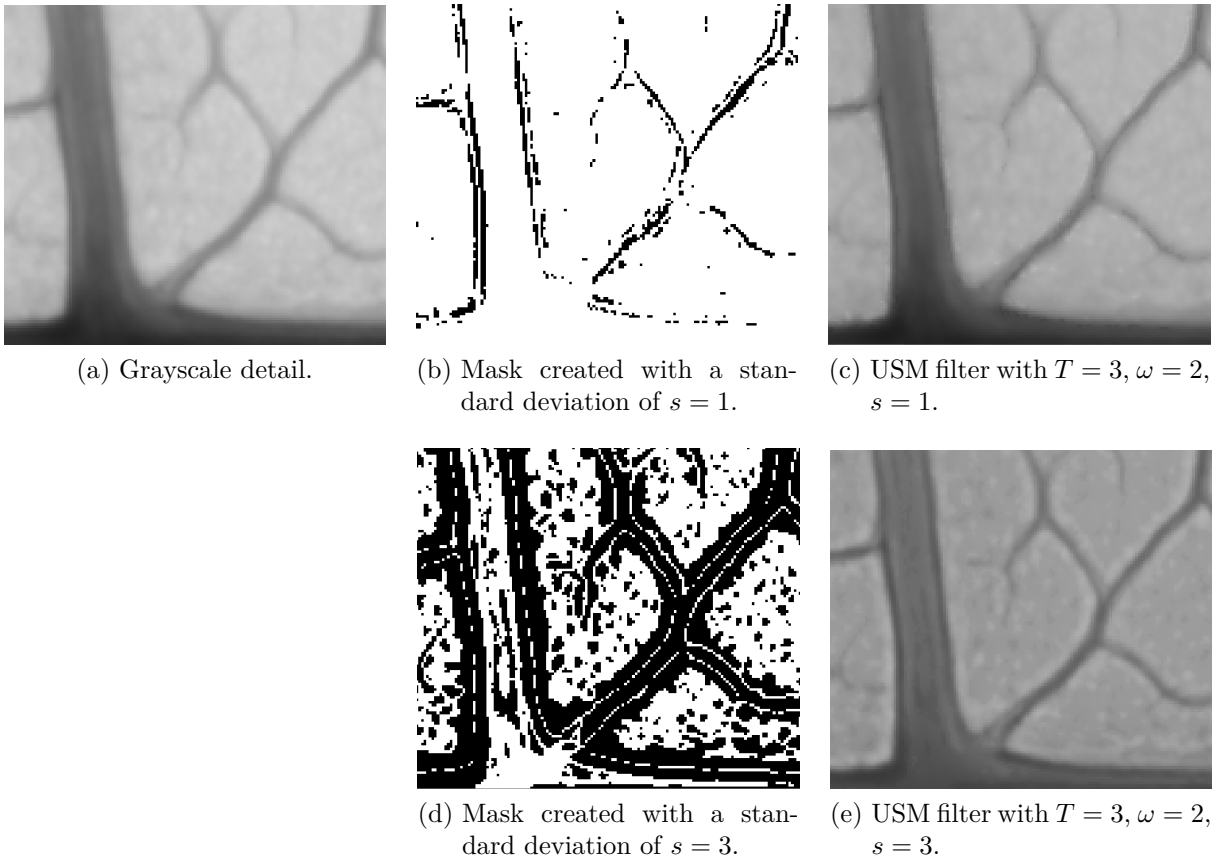
(c) Grayscale detail. (d) Sobel filter application.

Figure 3.8.: Application of blurring and Sobel filter with a kernel size of 7.

3.3.2. Unsharp Masking

Another common processing technique for sharpening edges in images is the *unsharp masking* (USM). In order to sharpen the image, we first create a blurred (unsharp) version of the image. With that we can create a mask by taking the absolute value of the difference between the original and the blurred image and thresholding it (typically with a very small threshold T because the difference will not be too large). The mask is a map filled with Boolean values which are true where brightness changes in the original image happen and false otherwise. We then create the sharpened image by subtracting the blurred image from the original (optionally weighted with a constant value ω) and inserting those newly created values into the original image wherever the mask holds a "true" value.
The unsharp masking process can be modified via the Gaussian standard deviation s of the blurring filter (which is equivalent to the kernel size), the threshold T and the weight ω. This offers a wide range of adjustability but also imposes the need to exhaustively adjust each mask to a given image. Doing this in a mostly automated way proved to be very difficult. Nevertheless the USM method including all parameters to adjust it is implemented in function B.13 `filters.unsharp_mask()`.
The process of unsharp masking as well as the results of two iterations with different

blurring filter sizes are shown in figures 3.9a to 3.9e. As can be seen, the USM with a blurring filter with a standard deviation of $s = 1$ produces nearly no visible effect, whereas the USM with a standard deviation of $s = 3$ emphasized steep brightness changes and suppressed slower ones.



(a) Grayscale detail.

(b) Mask created with a standard deviation of $s = 1$.

(c) USM filter with $T = 3$, $\omega = 2$, $s = 1$.

(d) Mask created with a standard deviation of $s = 3$.

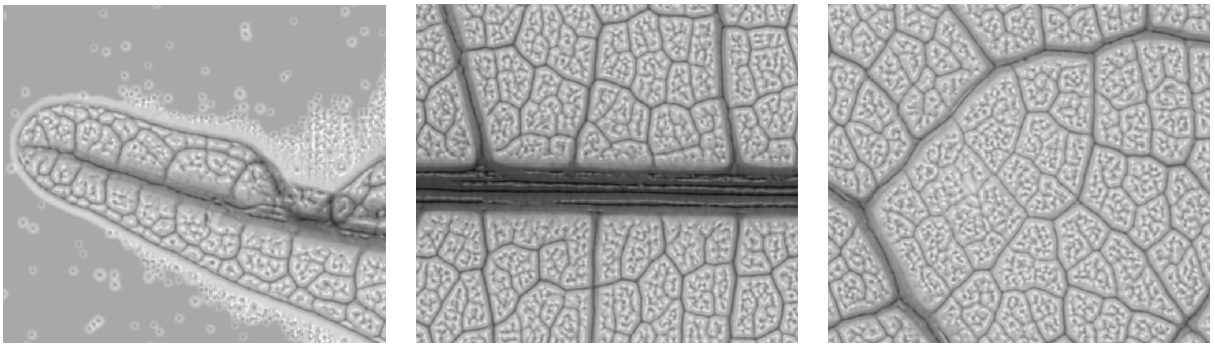
(e) USM filter with $T = 3$, $\omega = 2$, $s = 3$.

Figure 3.9.: Illustration of the application of the unsharp mask filtering method.

3.3.3. Blurring and Local Histogram Equalization

The local histogram equalization introduced in section 3.2.3 combined with a blurring filter of suitable size can be a very powerful tool to emphasize edges regardless of position in the brightness spectrum and large variances in brightness and contrast.
To counteract the appearance of edges where there are none (e.g. within larger veins), we can combine the processed image with the original one. Trial and error has shown that a proportion of one third of the processed image to two thirds of the original to form the resulting image works well in keeping a balance between noise and emphasized edges.
The results of the above described method are shown in figures 3.10a to 3.10c. As can be

seen, edges were emphasized in different regions of the leaf (tip, main vein, region with former low contrast) and all connections between the veins are well preserved. A lot of noise appeared in between the veins as well as on the main vein but the superposition with the original image ensured that the brightness of the noise is not too different from the surrounding regions to compromise correct thresholding.



(a) Detail at the leaf tip. (b) Detail near the main vein. (c) Former low contrast region.

Figure 3.10.: Examples processed with blurring ($s = 5$), local histogram equalization ($k = 11$) and combination of processed and original image (1/3 to 2/3).

3.4. Summary

3.4.1. Comparison and Choice of Method

With the possibilities of digital image enhancement and availability of many different filters and algorithms, countless different, promising ways to reach the goal of sharpening the edges of the vein network and emphasizing contrast are imaginable. As this is a Bachelor's thesis, I have focused on the methods introduced in section 3.3, utilizing the tools introduced in sections 3.1 and 3.2, and proceeded to binary processing as soon as I found a method that suited the needs of the framework.

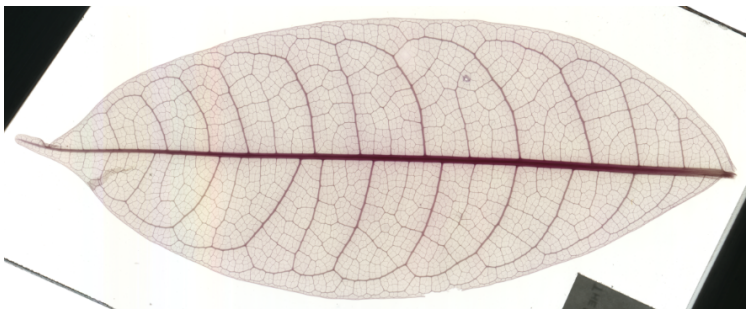
While the Sobel filter emphasized edges very well and might have yielded some very good results, it introduced the additional problem of refilling the interior of the veins. Unsharp masking is a very versatile tool with many different possibilities to adapt and adjust to a given data set and can yield results of high quality when enough time and thought adjusting the parameters are invested. The need for higher time investment and user input conflicts with the requirement for the framework to be able to operate mostly automated, therefore USM is not the method of choice to base binary processing upon.

The properties of local histogram equalization (LHE) fit best into the requirements for

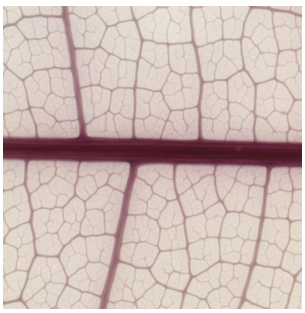
the framework. The fact that local, disconnected noise (the appearance of which is one of the major drawbacks of LHE) can be dealt with in binary processing as well as the advantage of the method being very independent of the specific image it is used on led to the decision of using this method as the basis for further processing.
One other major drawback of LHE which should be mentioned here is its time consumption. Most of the methods mentioned above are adaptions from already existing, optimized methods in libraries. As a method for LHE was not readily available in a form that would fit our workflow, I had to develop it in its entirety. Despite my effort to optimize the code for speed, the method consumes a sizeable portion of the total time needed to complete the image processing from raw scan to extraction of raw data (around 30 min for a 1 GB image).

3.4.2. Intermediate Results

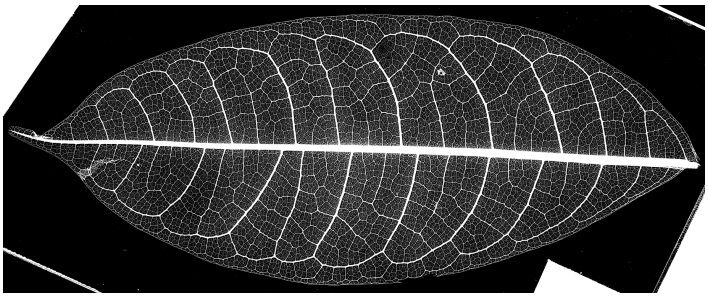
To show a few intermediate results, I selected some scans of leaves belonging to plants of the Burseraceae family which are representative of the plethora of scans in the cleared leaf database, and processed them from the raw state to the binary image. Figure 3.11 shows the scan that is the source of most of the details shown in this thesis to illustrate various concepts. Figure 3.12 (with its size of 1 GB) was one of the biggest images I had at hand for testing. Figure 3.13 was chosen because it is a relatively dark scan with a lot of poor contrast regions near the main vein. Finally figure 3.14 is a relatively bright scan with very sharp contrast between vein and no-vein regions and therefore yielded very appealing results.



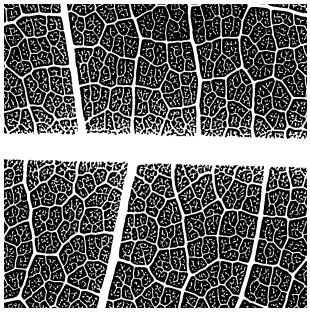
(a) Original Scan: complete leaf.



(b) Original Scan: detail.

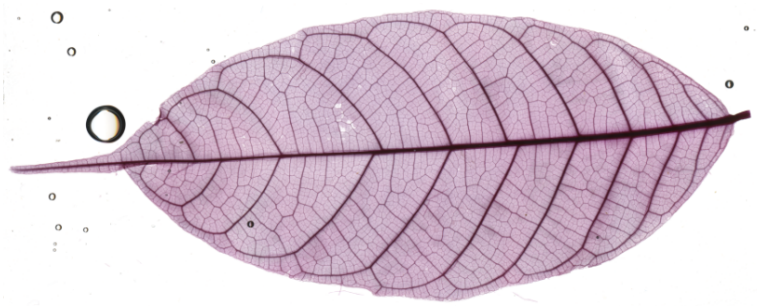


(c) Binary image: complete leaf

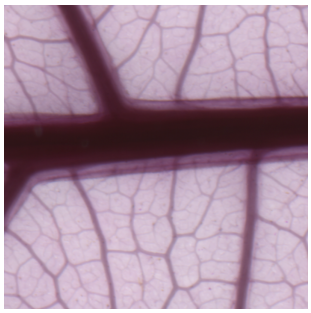


(d) Binary image: detail.

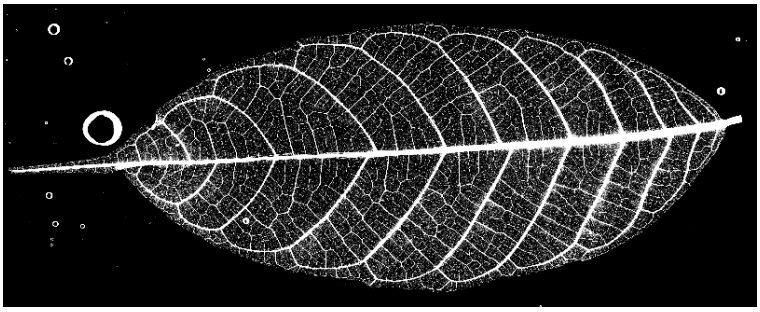
Figure 3.11.: Original and processed versions of a scan of *Protium Grandifolium*.



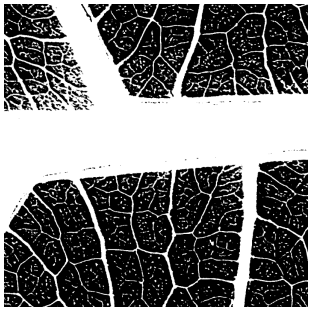
(a) Original Scan: complete leaf.



(b) Original Scan: detail.

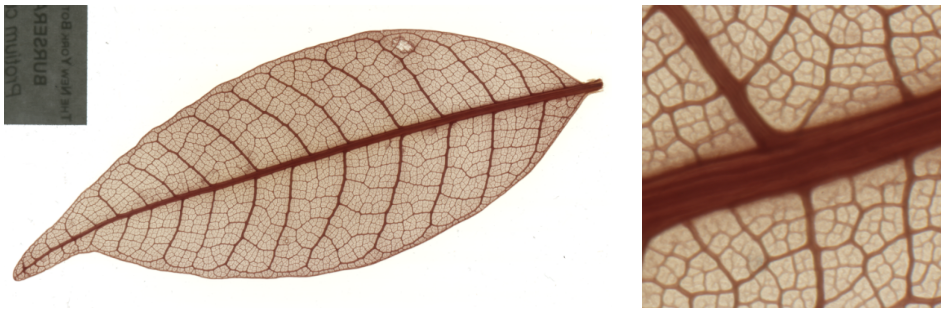


(c) Binary image: complete leaf.

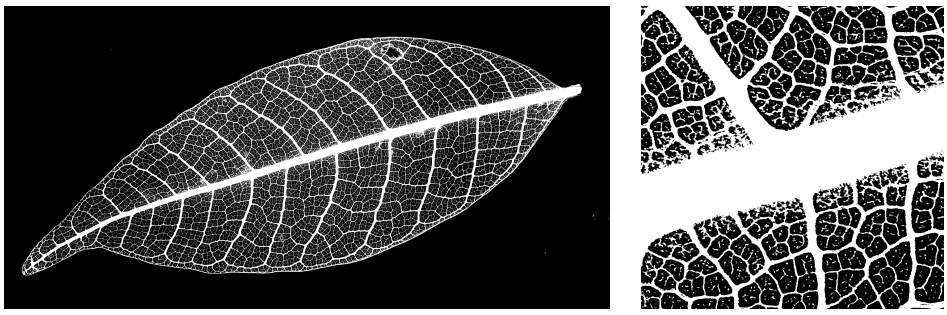


(d) Binary image: detail.

Figure 3.12.: Original and processed versions of a scan of *Brosimum Guianensis*.



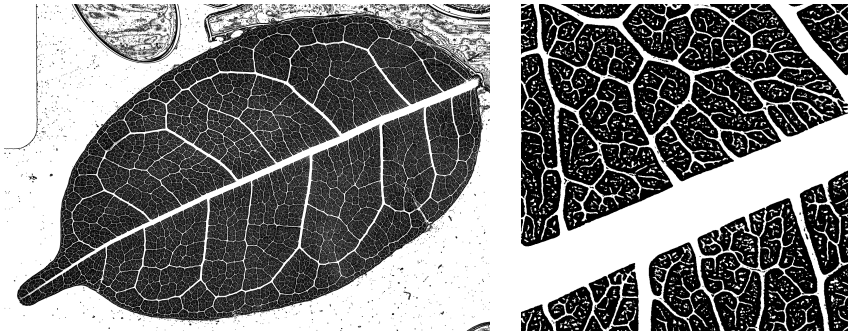
(a) Original Scan: complete leaf. (b) Original Scan: detail.



(c) Binary image: complete leaf. (d) Binary image: detail.

Figure 3.13.: Original and processed versions of a scan of *Protium Cubense*.

(a) Original Scan: complete leaf. (b) Original Scan: detail.



(c) Binary image: complete leaf. (d) Binary image: detail.

Figure 3.14.: Original and processed versions of a scan of *Protium Apiculatum*.

4. Binary Processing: Skeletonization and Distance Maps

After we have created a suitably enhanced image by application of the methods mentioned in section 3 to the scan of the leaf, we create a binary image by application of the thresholding filter (described in section 3.1.3) with a threshold calculated with Otsu's method. A binary image consists of only zeros and ones, representing foreground and background of the image. Such an image, which needs only one bit for the storage of information for every pixel, also consumes notably less memory. The binary versions of the images that I processed during the development of the framework typically only had 1/90 th of their original size, therefore making concerns about memory consumption negligible for the following course of the processing.
It should be noted that in binary processing, the convention of which value represents the foreground often is contrary to the convention used in previous chapters: white (value "1") is now used as *foreground* or *feature* and black (value "0") represents the *background*.
The next step before the nodes and edges of the leaf's network can be extracted is the retrieval of the *skeleton* of the network.

4.1. Skeletonization

The *skeleton* of a feature is a one pixel thin line that has the same distance from all corresponding boundary points and preserves the topology of the shape (i.e. points that are connected in the original image are also connected in the skeleton - no additional loops and endpoints appear) as well as its size. In figure 4.1 an illustration of the skeleton of a vein-network-like shape can be seen. In image processing, skeletons usually are used to save information about objects in an image in a more compressed way and make them comparable. In our case we need the skeleton to identify the nodes and edges of the network.

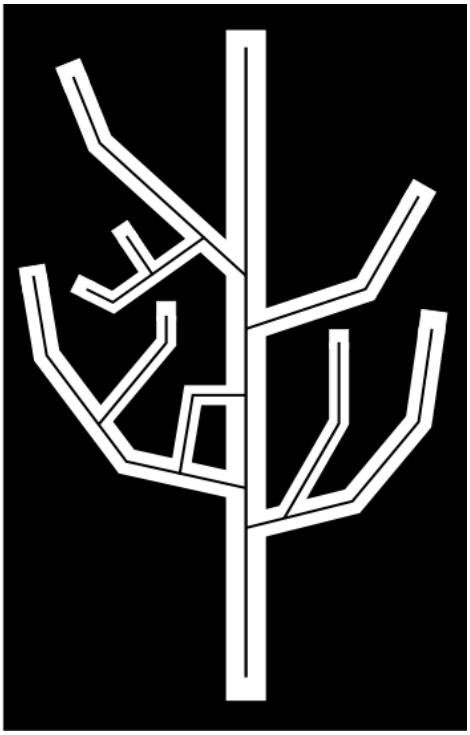


Figure 4.1.: Sketch of a skeleton.

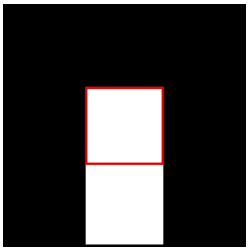
We distinguish between two fundamentally different approaches for skeletonization: *pixel-wise thinning* approaches and *vectorization* approaches that "construct" the skeleton of the features. Pixel-wise thinning is the most general approach and can be applied to any class of features that are represented in a binary image. Vectorization approaches by design presuppose additional information about the shape of the features in an image (such as ribbon-like shapes for vein networks on leaf-images) and vectorize them based on this information. During the development of the framework I have tested both approaches and will describe their operating principles and the results from application to binary images of leaves in the following sections.

4.2. Skeletonization by Thinning

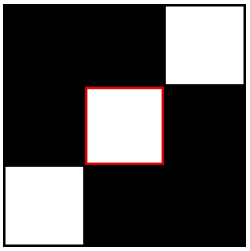
To create the skeleton of a shape via pixel-wise thinning, the algorithm traverses the image multiple times and removes redundant pixels. To decide whether a pixel is redundant, in each iteration it compares the 3×3 neighborhood of the currently processed pixel with the elements of a lookup table containing all possible $2^8 = 256$ binary 3×3 neighborhoods. A pixel is not redundant if it is necessary for the connection of two other pixels or if it is an endpoint. The pixel's removal also must not change the number of connected background or foreground areas in the neighborhood.

It has to be noted that by convention connectedness is defined differently for foreground and background. Foreground is so-called *eight-connected* which means that a foreground pixel is connected to another foreground pixel if it is touching the other pixel either at an edge or at a corner point. Background is *four-connected* which means that background pixels are only connected if they are touching at the edges.

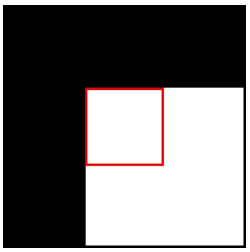
Examples of possible 3×3 neighborhoods can be seen in figures 4.2a to 4.2e. In every example, the figure shows the currently processed pixel at the center of the neighborhood, marked red. Figure 4.2a shows an example of an endpoint, therefore the central pixel is essential and will not be removed by the thinning algorithm. In figure 4.2b the central pixel connects the two pixels at the edges. Removal of the central pixel would create two separate foreground regions, therefore the central pixel is again essential. Figure 4.2c shows an example of a redundant pixel. The central pixel is neither an endpoint nor does its removal change the number of foreground and background regions in the neighborhood. The



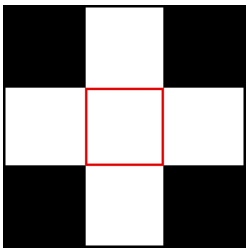
(a) Not redundant.



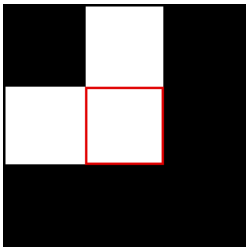
(b) Not redundant.



(c) Redundant.



(d) Not redundant.

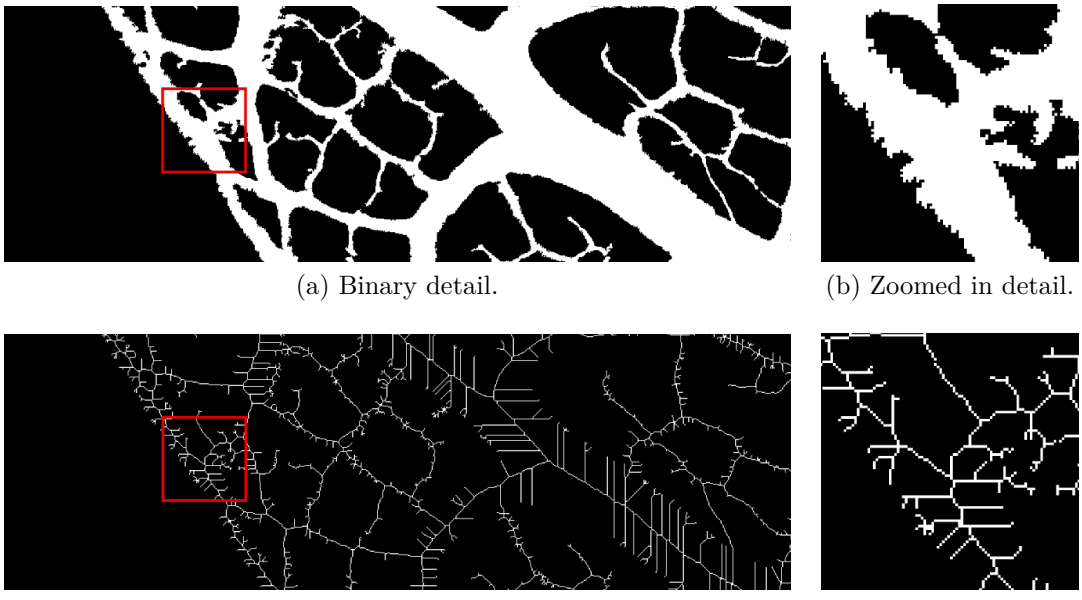


(e) Redundant.

Figure 4.2.: 3×3 neighborhood examples.

example also illustrates one of the major drawbacks of this method: depending on the rest of the shape the central pixel is part of, during further processing its removal might lead to the creation of additional end points, thus creating branches in regions that were one single vein before the thinning.
Figure 4.2d shows an example where removal of the central pixel would not change the number of foreground regions but increase the number of background regions by one, as background pixels are only four-connected. Finally, figure 4.2e shows an example where the central pixel is redundant and its removal will not cause the appearance of additional endpoints either.

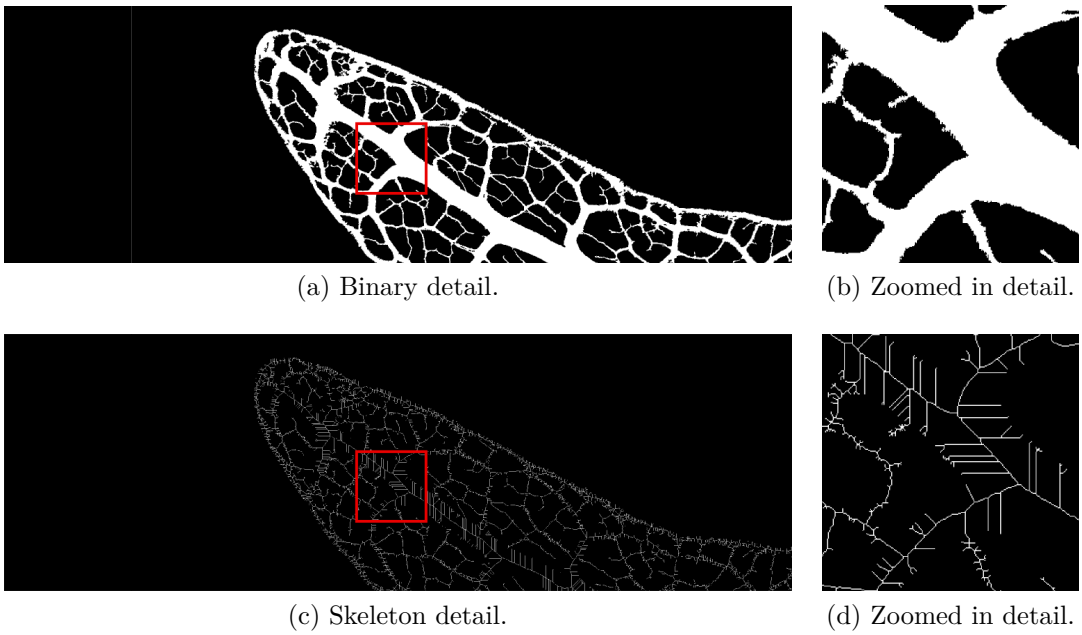
The thinning algorithm "melts" the features in the image layer by layer and needs as many iterations over the whole image as the width of the widest feature in pixels. Thinning is very prone to irregularities and small protrusions at the vein-boundaries.
The results of a skeletonization of network parts with a MATLAB implementation of thinning are shown in figures 4.3a to 4.3d and figures 4.4a to 4.4d. It can be seen that a lot of additional small branches appear - especially at the boundaries of larger veins - and therefore the topology of the network is distorted.



(a) Binary detail. (b) Zoomed in detail.

(c) Skeleton detail. (d) Zoomed in detail.

Figure 4.3.: Skeletonization of a binary detail via thinning implemented in MATLAB.



(a) Binary detail. (b) Zoomed in detail.

(c) Skeleton detail. (d) Zoomed in detail.

Figure 4.4.: Skeletonization of a binary detail via thinning implemented in MATLAB.

Countering this without losing information of the highest branching orders proves very challenging. Features can be "smoothed" to reduce the number of origins of surplus branches. If we do not want to manually look at problematic regions, every "fix" for a regional problem also affects the whole image and it is very difficult to find operations that improve the image in a region without destroying information in other regions as well.

Another way to get rid of surplus branches is the "pruning" of the network, i.e. cutting off all branches with a length shorter than a given threshold. This naturally also affects the small branches belonging to the highest branching orders, which will be lost in the process. One would need to determine in an automated way whether a short branch originates in a very large vein and therefore is an unwanted artifact or is situated in a region of very delicate branches and actually belongs to the network. Such an operation, though feasible, would be error prone and computationally cumbersome. In contrast, a vectorization approach natively makes it possible to selectively remove branches according to their neighborhood.

As the appearance of surplus branches could not be dealt with in a satisfactory way, we decided to take a less general approach to the problem of skeletonization by using the information that the features we are interested in are ribbon-like and implement a skeletonization algorithm based on vectorization of the features.

4.3. Skeletonization based on Constrained Delaunay Triangulation

The second class of approaches to the problem of skeletonization is the *vectorization* of a shape and subsequent construction of the skeleton using information about the boundaries of the shape by iterating only a few times or even only once over the whole image. As the shape of the vein network consists of ribbon-like parts with many junctions and crossings and of various widths, the approach proposed by Zou and Yan [12] is a good fit for our data set.

For the construction of the skeleton, we first need the contours of the shape. On these contours we can perform a Constrained Delaunay Triangulation (CDT) with the inclusion of the contour segments as constraints. The result of the CDT-algorithm is a mesh of triangles. By classifying those triangles into end triangles, normal triangles and junction triangles, we can find the skeleton segments which approximate the exact skeleton. In the following, I will describe a customized skeletonization algorithm based on a CDT: I adapted the algorithm to the problem of the skeletonization of large vein networks, and I will detail both its operating principles and implementational particulars.

4.3.1. Contour Extraction and Linearization

As a first step, we extract the *contours* of the shapes that are to be skeletonized. This can be done in various ways and may already influence the resulting skeleton by a large degree. The first and simplest approach is to extract every pixel that has at least one background and one foreground pixel as a neighbor and build the contour with those pixels. This results in a boundary that consists of a very large amount of pixels. As the triangulation (described in section 4.3.2) triangulates every contour point, this leads to a very large amount of triangles with extremely sharp angles, as one of their sides, by construction, has a length of one pixel. This, in turn, slows down the triangulation significantly. Also, if we select every pixel with one background and one foreground neighbor, this copies all irregularities, small variations, curves and noise at the boundaries into the contour. The result is a distorted skeleton that may even contain additional branches originating in former noise-pixels. To prevent that, the contour has to be linearized before continuing to work with it: we need to select *dominant points* representing the contour. The retrieval of the contour, including several modes for dominant point detection is already available via OpenCV's function `cv2.findContours()`. OpenCV implements the contour retrieval as suggested in [21]; the available contour approximation modes are:

- No approximation - all contour points are treated as dominant points and returned.
- Simple approximation - compresses horizontal, vertical and diagonal segments and yields the endpoints of the segments as dominant points.
- The two flavors of the *Teh-Chin Dominant Point Detection Algorithm* (DPDA) which differ by their curvature measure and further compress the contours, reducing the number of dominant points.

The journal article [22] offers a thorough description of the DPDA as well as an upper boundary for the error between the original contour and the approximating polygon formed by the dominant points. In the framework the default value for the DPDA is the selection of the 1-curvature measure because it yields good compression results and is the faster one of the two flavors of the algorithm. The framework's function for contour creation is function C.7 `triangulation_functions.getContours()` and it offers several useful functions for the conversion of the list of contour coordinates into other formats (functions C.10, C.11 and C.12).

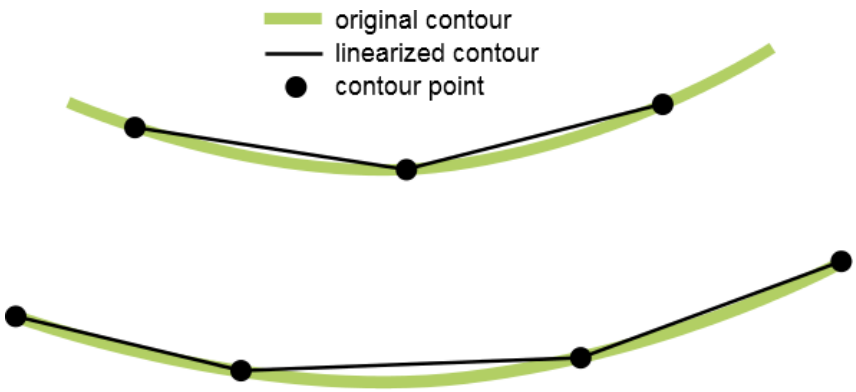
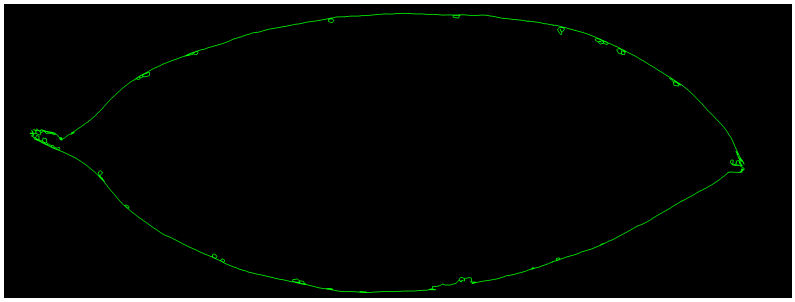
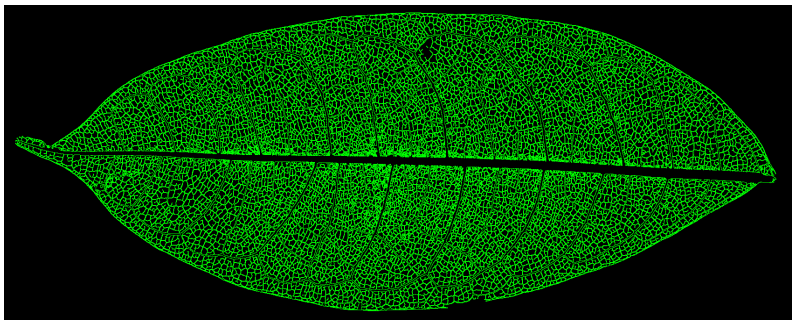


Figure 4.5.: Linearization of a contour.

Figure 4.5 illustrates the principle of the selection of dominant points and approximation of the original contour by a polygonal line. (The dominant points in the sketch do not claim to satisfy the conditions of the DPDA.)
The result of the contour finding process is a hierarchical list of contours and sub-contours with the outermost contour of the leaf at the top. The inner parts of the network are formed by *holes* in the outermost contour. A plot of the outermost contour of a leaf as well as a plot of the full hierarchy can be seen in figures 4.6a and 4.6b.
A contour technically is a list of points which form a polygonal line. The length of this



(a) Outermost contour of the leaf.

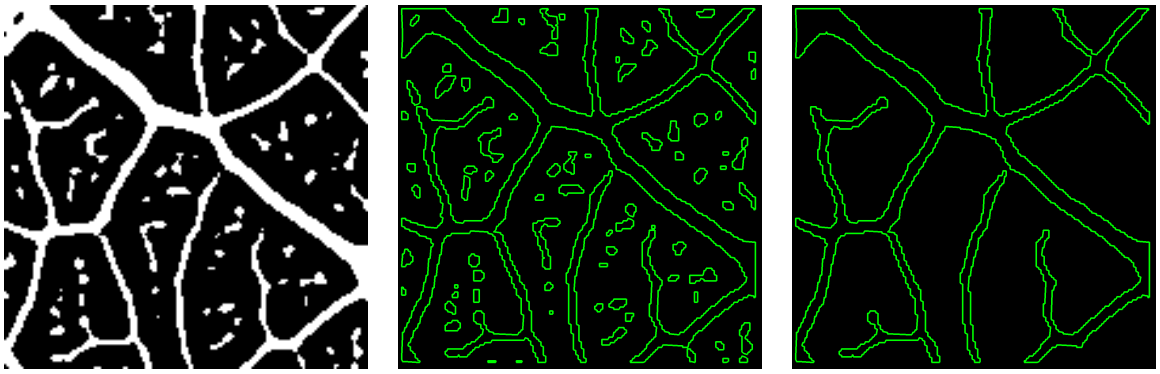


(b) Holes added to the longest contour.

Figure 4.6.: Illustration of the contour hierarchy of the vein network.

list (i.e. the number of contour points) is by trend proportional to the size of the contour, thus shapes that appear larger in the image have a tendency to have longer contours. During the course of the processing, we always accepted the appearance of *disconnected* noise (small neighborhoods of connected pixels that are disconnected from the leaf vein network) as a tradeoff for the advantages of local histogram equalization. This is because at this stage of the processing, the contours belonging to noise can be removed by thresholding the list of contours by length. This is possible as long as the noise-contour is *not* connected to the network itself and therefore is represented by an *additional* entry in the list of contours. As noise is usually smaller than network-forming holes, contours belonging to noise will be the shortest in the list of contours. This not only applies to positive noise (i.e. "isles" in the holes between the veins) but also to negative noise (i.e. holes in the large veins). Nevertheless, we have to choose the length threshold for the contours carefully because if the threshold is too high, thresholding will also remove the smallest holes that form the highest branching orders. This results in the appearance of "blobs" at the end of the vein network instead of small branches. The framework's implementation of contour thresholding is function C.8 `triangulation_functions.thresholdContours()`. The selection of a length threshold has not been automated yet. This could be done by looking for a minimum in the length distribution among the contours with low length since

noise as well as the highest branching order each should have a characteristic length. Figures 4.7a to 4.7c show an example of contour retrieval and removal of noise by thresholding of the contour-length for a binary detail of an image. It should be noted that in OpenCV's implementation of the contour retrieval, the original image is changed and the outermost pixels of the image (i.e. a one pixel thin rectangle around the image) are substituted with boundary pixels. This explains the appearance of artificial boundary lines at the edges of the example images.



(a) Binary detail. (b) All contours. (c) Thresholded contours.

Figure 4.7.: Contour retrieval and subsequent noise removal.

4.3.2. Constrained Delaunay Triangulation

After we have extracted the contours and thresholded them in such a way that only contours belonging to the leaf are left, the resulting shape can be triangulated using a Constrained Delaunay Triangulation. Triangulations and Constrained Delaunay Triangulations are defined as follows:

- A *triangulation* is a subdivision of an n-dimensional geometric object into simplices. (A simplex is an n-dimensional generalization of a polygon with $n+1$ edges.) In the two-dimensional case of images, it is a subdivision into triangles.

- A triangulation $T(P)$ of a *discrete set of points* $P \subset \mathbb{R}^2$ is a subdivision of the *complex hull* of the points into triangles. The triangles have to intersect either in a common face (i.e. a side) or not at all and the set of all vertices of the triangles has to coincide with P.

- A *Delaunay Triangulation* $DT(P)$ of a set of points is a triangulation where no point $p \in P$ lies within the inside of any triangle's circumcircle. Delaunay Triangulations

maximize the minimum angle of all angles of the triangles in $DT(P)$. Therefore, an optimal triangle in this sense is an equilateral one.

- A *Constrained Delaunay Triangulation* $CDT(P)$ is a generalization of the Delaunay Triangulation. In a CDT, required segments (such the line segments between the contour points) can be forced into the triangulation. A CDT might contain edges that do not satisfy the Delaunay condition and therefore itself often is not a Delaunay Triangulation ($CDT(P) \supsetneq DT(P)$).

Figure 4.8 illustrates the functionality of the CDT applied to a ribbon-like contour.

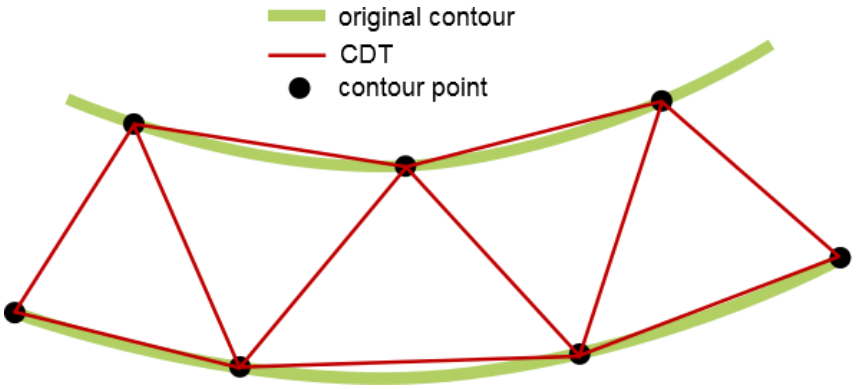
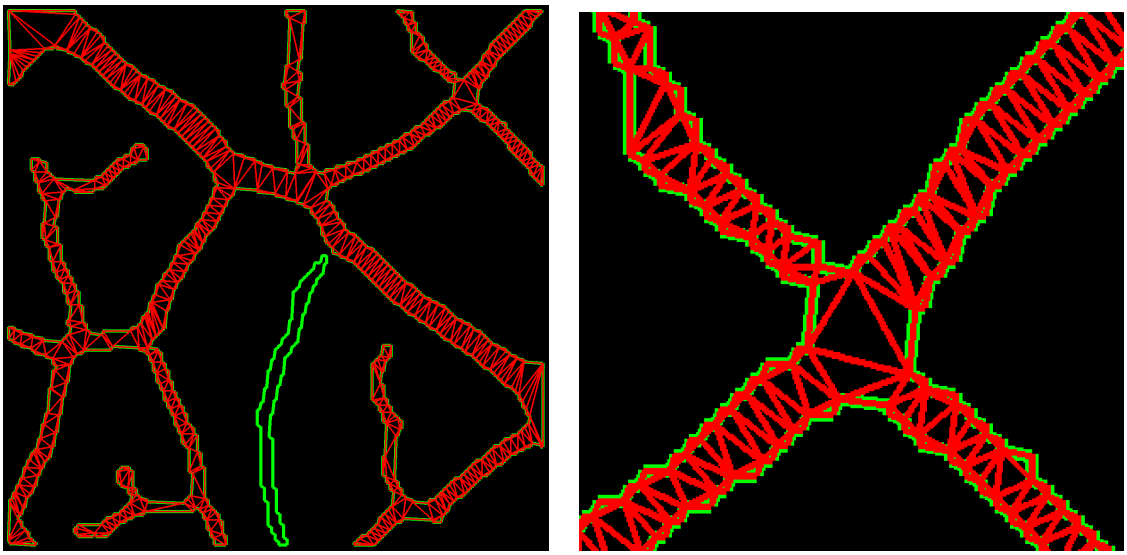


Figure 4.8.: Sketch of a Constrained Delaunay Triangulation.

For the actual execution of the CDT on the set of contour points of the network with the edges between contour points as constraints, there were three different libraries for Python available to me: `MeshPy`, `poly2tri` and `Python Triangle`. After looking into all three of them, I chose `poly2tri` because it was the only library with some available documentation and usage-examples to work with. The other two choices might perform better, be more versatile or yield better results but since they were undocumented I did not use them for the purpose of this thesis.
To perform the triangulation, we have to add the polygon representing the outermost contour to a triangulation object and then add all smaller contours to the same triangulation object as holes. Performing the triangulation yields a list of all triangles the triangulation consists of. The triangulation function which yields the list of triangles is function C.9 `triangulation_functions.getTriangulation()`.
Figure 4.9a shows a CDT of the contours from figure 4.7c. We can observe that the disconnected shape in the lower half of the image is not part of the triangulation, which is another safeguard against the transfer of disconnected components into the network representation. In this case, the disconnected feature only is disconnected because the

detail of the image was chosen that way and is very likely connected to the network if a larger detail is examined.
The additional boundaries at the detail's edges somewhat distort the triangulation, but the triangles filling the veins quickly "stabilize" and the principle of the triangulation can be seen clearly.

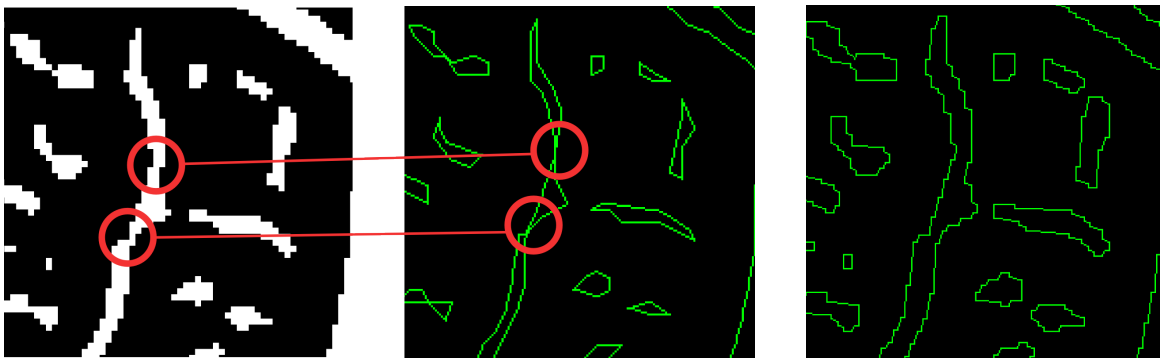


(a) CDT of thresholded contours. (b) Triangulation at a junction.

Figure 4.9.: Constrained Delaunay Triangulation of a thresholded contour.

4.3.3. Technical Triangulation Limitations

As the triangulation library is written for triangulations over much shorter and better behaved contours with a much smaller number of holes, several problems arose during the application of the CDT to the contours of whole leaves.
Singular contour points (or "bottlenecks") cause the poly2tri algorithm to crash. Bottlenecks are points where both boundaries of the contour meet in a single pixel and afterward split again and continue. The contour extraction and linearization algorithm does not prevent the appearance of those points and bottlenecks can appear wherever in the original binary image the shape is only connected by a single pixel. We can observe the formation of such a bottleneck from a binary image in figure 4.10a.
We solve this problem by resizing the binary image by an odd factor ≥ 3 before the contours are created. This simply triplicates (or more) the size of every pixel in the binary image and does not create additional edges. Afterward the algorithm for contour creation is now able to trace *around* the formerly singular point and find two separate,



(a) Formation of singular points from a binary feature.

(b) Solution by inflating the image.

Figure 4.10.: Feature in a binary image that is only connected by one single pixel and resulting contour with singular point. "Bloating" the feature solves the problem.

non-intersecting contour courses. "Inflating" the image by a factor of three also triples its size but even in the worst case of a 3 GB image the size of the inflated binary image is still far below a size that threatens to exceed a usual RAM.

Another potential problem is the appearance of *collinear points* during the triangulation. Collinear points are points that lie on a straight line between two other points. As the triangulation algorithm does not create new points, those points must already be contained in the contours fed to it. However, the DPDA algorithm should inherently eliminate all those points and thorough checks of the triangulated point sets that caused crashes did not reveal any collinearities in the contours. This leads to the assumption that crashes of the triangulation algorithm due to collinear points might originate in a bug in the library itself which I was not able to fix in the given time.

As a consequence, the framework can only process sub-regions of whole leaves so far. Nevertheless the results gained by processing details of images show that in principle, the triangulation does work well for these kinds of shapes when the contour points are selected properly.

4.3.4. Skeleton Construction

After we have created a contour and a CDT of the network shape and therefore vectorized it, the last and final step is the construction of the skeleton based on the triangles contained in the triangulation.

As suggested in [13], we divide the edges forming the triangles into two classes: an edge

is called *external edges* if it is a contour segment and *internal edge* otherwise. This leads to a distinction between four species of triangles that form the triangulation from which the skeleton will be constructed:

- An *end triangle* (ET) is a triangle that has one *internal edge*.
- A *normal triangle* (NT) is a triangle that has two *internal edges*.
- A *junction triangle* (JT) is a triangle that has three *internal edges*.
- An *isolated triangle* (IT) is a triangle that has three *external edges*.

ITs do not appear in the triangulation as only connected components of the network are triangulated. Figure 4.9b illustrates the appearance of a junction triangle in the previously shown triangulation.
The skeleton segments of each of the triangle species are formed as follows:

- ET: a straight line from the centroid of the triangle to the midpoint of the internal edge.
- NT: a straight line that connects the midpoints of the internal edges.
- JT: three straight lines that connect the centroid of the triangle with each of the midpoints of its edges.

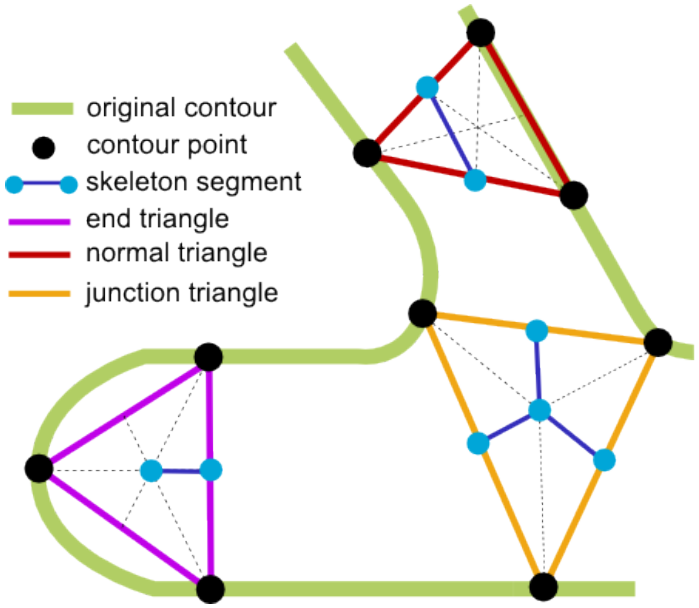


Figure 4.11.: Different triangle species and their skeleton segments.

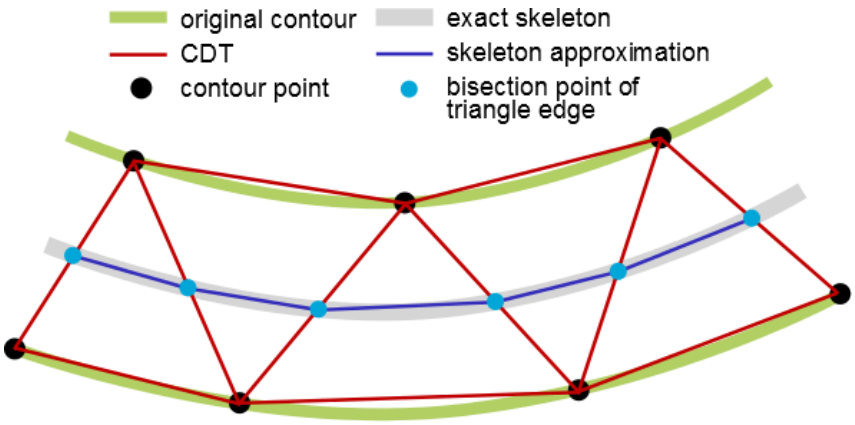


Figure 4.12.: Construction of the approximation of the exact skeleton using the midpoints of internal edges.

The skeleton segments of all triangles together form the skeleton of the network. Figure 4.11 shows the three different triangle species and their skeleton segments and figure 4.12 shows the approximation of the exact skeleton by the skeleton segments of several NTs.

Figures 4.13a to 4.13l demonstrate the triangle classification based on the CDT as well as the extraction of the skeleton segments. At this point, all information about the position of the nodes and edges and the connections between the nodes is present in the list of triangles and the list of contour segments. The last thing left to do is to gather this information and compile it into suitable data formats.

For this purpose, the module `triangulation_functions` introduces several custom classes for the handling of points, segments and triangles:

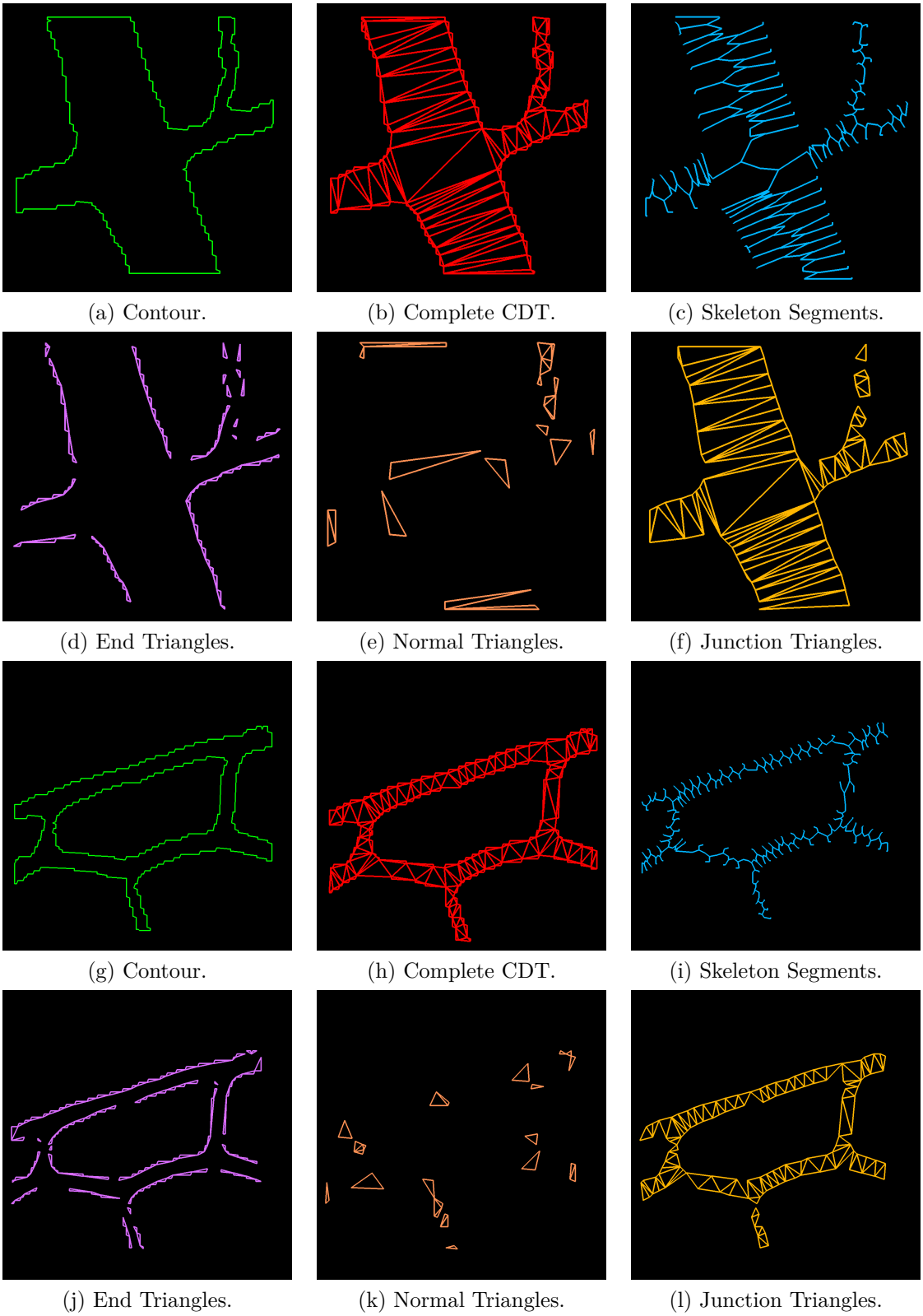
- The point class C.1 **`triangulation_functions.point`**
- The segment class C.2 **`triangulation_functions.segment`**
- The ET class C.4 **`triangulation_functions.end_triangle`**
- The NT class C.5 **`triangulation_functions.normal_triangle`**
- The JT class C.6 **`triangulation_functions.junction_triangle`**

The characteristic of the three triangle classes is that each triangle knows its skeleton segment, which reduces the problem of skeleton segment extraction to a classification of the triangles in the triangulation to the three triangle classes. We do this by comparing the edges of the triangles with the contour segments. Function C.13 **`triangulation_functions.triangle_classify()`** handles the classification process and function C.14 **`triangulation_functions.extract_skeleton()`** extracts the skeleton

segments from all triangles and adds them to a global list of skeleton segments. The classified triangles also offer the possibility to extract all nodes and/or all endpoints present in the network simply by looking at the centroids of the triangles in the corresponding class.
The (skeleton) segments export a `segment.connects()` method which makes it easier to determine whether two segments share a common point and therefore trace the skeleton from node to node.
It should be noted that the pixels in the image are discrete but calculating the centroids of triangles and midpoints of edges will produce non-integer values in many cases. To prevent that a point that belongs to two skeleton segments is not equal in both segments and therefore ensure connection of the segments, rounding of the non-integer values to the nearest integer is done right after the calculation of the point.

4.3.5. Possibilities for Error Minimization and Stabilization

As we can see in figures 4.13d to 4.13f and 4.13j to 4.13l, the triangulation and therefore the classification of the triangles is largely dependent on the selection of contour points. We can observe (e.g. figure 4.13h), that the triangles in the middle of the veins are "padded" with tiny triangles at the vein's boundaries. The triangulation produces those triangles because the "bloating" of the image (as described in section 4.3.3) somewhat fools the DPDA algorithm and tricks it into selecting more contour points than needed. Normally, the triangulation should mostly consist of NTs but the padding results in a lot of ETs (figure 4.13j) at vein boundaries. In turn a lot of JTs (figure 4.13l) appear in the middle of veins. This leads to the conclusion that in concept, the vectorization of the vein network via CDT and triangle classification does work but needs refinement before the resulting skeletons represent the real topology of the network without too much error. As bloating of the image seems to be the cause for imprecise skeleton creation later on, we have to think of a better solution for singular points in contours.
Apart from that, future improvements will have three main targets: the *smoothing* of the vein's boundaries, the *stabilization* of the triangles at critical points, and *error minimization* (i.e. the overall reduction of the deviation from the true skeleton).
To smooth the vein's boundaries, the previously mentioned "padding" ETs have to be removed or merged into the triangles that lie in the middle of the vein. An approach to detect and remove these so-called "protrusions" can be found in [13]. In this approach, a protrusion (when viewed as a ribbon-like shape) is considered *perceptually insignificant* if its length is smaller than its width which is by trend the case for irregularities of the vein-boundaries.



(a) Contour. (b) Complete CDT. (c) Skeleton Segments.

(d) End Triangles. (e) Normal Triangles. (f) Junction Triangles.

(g) Contour. (h) Complete CDT. (i) Skeleton Segments.

(j) End Triangles. (k) Normal Triangles. (l) Junction Triangles.

Figure 4.13.: Triangle classification and skeleton extraction based on a CDT.

The course of the skeleton near and at junctions can differ by a large degree from the true skeleton if the junction is represented by more than one junction triangle. Clarifying the course of the skeleton in this region is called *stabilization* of the triangles and is done by merging them into junction *polygons* rather than triangles [13]. [18] also proposes the refinement of the triangulation near junctions and sharp bends by insertion of additional contour points at strategic places.
The overall deviation of the skeleton from the true skeleton can be reduced by controlling the "skeletonization error" as described in [17]. This approach tries to adapt the number of contour points chosen in a specific region dynamically to the curvature of the contour.

4.4. Euclidean Distance Map

The only information that we need to fully describe the network that we have not been extracted yet is information about the *conductivity* of the veins which corresponds to the *thickness* of the edges that connect the nodes. The approach to retrieve this information is to create a so-called *Distance Map* from the binary image and superimpose it with the skeleton. Then every pixel in the skeleton has a corresponding conductivity value. Pixels not belonging to the skeleton are discarded.

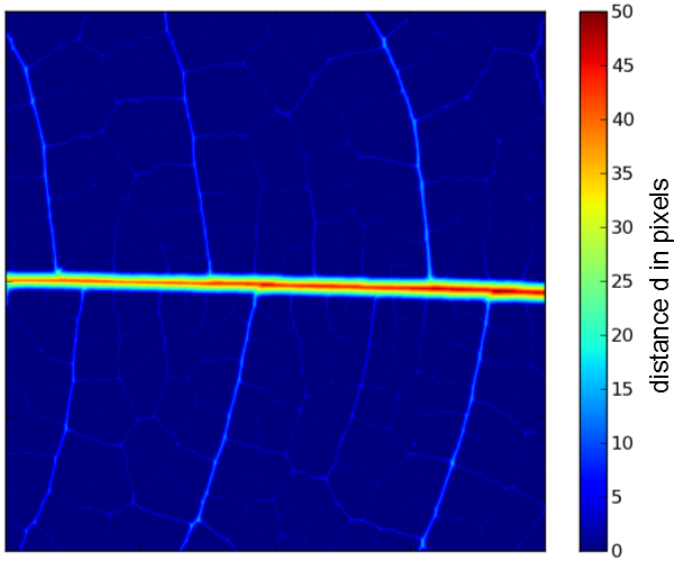


Figure 4.14.: Distance Map of an image detail.

A Distance Map of a binary image holds information about the distance each foreground pixel has to the nearest background pixel. As the image is discrete, such a map can only be an approximation of the actual Euclidean distance. As a first approach for the creation of the Distance Map, we search originating at each foreground pixel, for the nearest background pixel by "walking" in each of the four directions (up, down, left, right) until we encounter a background pixel. We then save the shortest number of steps that is needed to reach a background pixel in a matrix at the coordinates of the origin-pixel. The matrix

in turn can be represented by an image. To refine this approach we can increase the number of directions which are searched for the nearest background pixel until the searched regions with each additional step approximate concentric circles around the origin pixel. To reduce the error that results from the discrete size of "one-pixel-steps", diagonal steps are counted as a length of $\sqrt{2}$ pixels.
We create the Distance Map by utilizing openCV's function `cv2.distanceTransform()` (the algorithm used is described in [9] and [8]) which supports various modes for distance types and search area shapes. The framework's function C.15 `triangulation_functions.cv_distanceMap()` handles input and output formats and sets default values for the parameters.
The Distance Map is the final result of the framework regarding conductivity of the veins because it allows for easy extraction of the vein thickness at any given point of the skeleton: the thickness/conductivity C of the vein at pixel $p_{i,j}$ is

$$C = 2 \cdot d_{i,j}$$

where $d_{i,j}$ is the distance to the nearest background pixel found in the Distance Map.

4.5. Summary and Results

Skeletonizing the vein network using a pixel-wise thinning algorithm yielded non-satisfactory results. Therefore I chose a vectorization approach which makes use of the generally ribbon-like structure of vascular networks. The skeletonization of the vein network is done via an approach based on a constrained Delaunay Triangulation of the contours forming the network-features. The triangulation yet needs to be refined to yield qualitatively fully satisfying results but in concept the approach works for the skeletonization of a vein-network.
The first result of the binary processing of the image is the list of edges of the network which we extract from the classified triangles the CDT yielded. Based on this list, a connection matrix and a distance matrix can be constructed by tracing connected skeleton segments.
The Distance Map of the binary image holds information about the distance from each foreground pixel to the nearest background pixel. We create the Distance Map by utilizing OpenCV's `cv2.distTransform()` function with parameters fitting the problem set. The Distance Map is the second result of binary processing which, superimposed with the previously created skeleton, yields information about the thickness of the veins and therefore makes the construction of a weighted connection matrix possible.

5. Results

5.1. Summary

In this thesis, I developed an adaptable and portable framework for the processing of images containing vascular networks. The data we want to extract from the vascular networks present on the images includes the network's topology, geometry and conductivity (edge thickness). The tools provided by the framework span the whole range from efficient loading and saving of images, to enhancing those images in various ways, vectorization of the features on the image and extraction of network data.
The goal was to allow as little quality losses during the processing as possible and emphasize possible automation of the methods used. As the images processed are very large and the number of images to be processed is very high, time efficiency also was a factor. Last but not least the framework was developed to be portable and open for adaptations and further development. In the following I will briefly summarize the three main parts of the framework and to what extend they comply with the stated requirements.
The first part of the work on the framework was committed to the handling of large images. I was able to develop a stable and versatile tiling mechanism which makes it possible to process images of a size that would otherwise exceed the computer's RAM capacity. The tiling mechanism divides the image into tiles which are loaded from the hard drive independently and handles the overlap between those tiles so no artifacts are generated. The images are saved in a hierarchical data format which makes hard drive access very efficient. The tiling mechanism in this sense fully met the goals set for the framework's development.
Subsequent to the tiling of the image, in the second part of the framework I developed and implemented several filters and filter combinations to enhance the images. During the work on the image enhancement I encountered a lot of possibilities and different approaches to image enhancement and had to focus on one approach due to the limited time of the thesis. The image enhancement method chosen reduces noise in the image by blurring it and then performs a local histogram equalization to minimize the overall

contrast of the image and maximize the local contrast between vein and no-vein regions. After the image is filtered in this way, it is combined with the original to counter some of the tradeoffs of local histogram equalization. This approach yields results of high quality when applied to images differing in brightness and/or contrast, but its execution time is rather long. A rough value for the processing time of a 1 GB image from the raw state to its binary version is 30 minutes. Processing time as well as variety of available filters in the framework need further improvement. Quality of the processed images and the possibility to automate the processing due to the robustness of the used filters fulfill the requirements we set for the framework.
The third and last part of work on the framework was committed to skeletonization and subsequent data extraction from the image. The approach of pixel-wise thinning to create a skeleton of the network did not work out well, therefore I chose an approach of vectorization based on Constrained Delaunay Triangulation. The available triangulation libraries do not work for images of the size we are considering, so for the purpose of this thesis I demonstrated the method using only image details. The skeletons produced by the vectorization algorithm demonstrate current pitfalls of the method but serve as proof of principle for the CDT method and as basis for further development.

5.2. Outlook

During the course of the thesis I already mentioned several possibilities for improvements and further development of the tools present in the framework. In the following I will give an outlook on the most important and promising targets for further development categorized into efficiency improvements, quality improvements, solutions for current problems and quality of life improvements.

Efficiency Improvements

Algorithms that work more efficiently shorten processing times and enable us to process more images in a shorter time. Time efficiency can be improved in the following ways:

- In principle, image enhancement is completely parallelizable. To make parallelized processing possible, a non-sequential variant of the tiling mechanism needs to be implemented.

- Parts of the framework that are computationally intensive can be coded in C using Python's C-interface `Cython` to speed up their execution.

- To reduce processing times of image enhancement methods, instead of in the spatial domain they can be performed in frequency space.

Quality Improvements

Better quality of the enhanced images lead to a better quality and less errors in the extracted skeletons, which in turn lead to simulations that resemble naturally grown networks more closely. Quality improvements can be made at various points in the framework:

- The possibilities of image enhancement can be explored further and techniques that promise to produce images with an even better quality can be implemented, for example gradient based thresholding.

- So far, defects in the images such as holes and bends in the leaves, and stains and artifacts from the scanning process have been ignored. The implementation of heuristic methods to remove or reduce such defects may further improve the quality of the image.

- At the moment images processed in this framework are immediately converted to grayscale images. In future attempts to correct defects, color information can be used to develop heuristics for this purpose.

Current Problems and Possible Solutions

The most severe problems the framework has in its current state lie in the vectorization algorithm. To enable the skeletonization to work properly, we have to solve various problems:

- The problems caused by triangulation of singular contour points. Solving this problem by inflating the image caused further problems with the triangulation so a different solution has to be found.

- Triangulation library issues that prevent us from skeletonizing whole images.

- The output of a triangulation that does not correctly represent the topology of the network by improving the contour point selection algorithm and stabilizing the triangles in the triangulation.

Quality of Life Improvements

To facilitate the handling of the framework, quality of life improvements for the user can be made:

- Development of more intuitive ways for users to interact with the framework, possibly even development of a GUI.
- Implementation of functions to display the extracted network data in a more accessible way, e.g. conversion of the currently present list of segments and Distance Map to a weighted connection matrix and a distance matrix.

In the following months, the framework presented in this thesis will be expanded and streamlined, as part of a larger project interfacing with major cleared leaf collections at Yale and the Bronx Botanical Gardens.

A. Module: `tile_functions`

The tile_functions module contains a collection of functions to deal with large images in the form of numpy arrays and openCV matrices. It contains methods to efficiently dump data on the hard drive and read it again using the HDF-5 format. It also provides wrapper methods for filtering the images in which filters from the `filters` module B can be plugged in.

`tile_functions.`**`ChunksizeOptimizer`**(*image, max_chunk_size*)

Tries to find the optimal size for the data chunks so the least expanding of image boundaries is needed

Parameters:

`image:` `ndarray` array containing the image data

`max_chunk_size:` `integer` maximum allowed size of data chunks

Returns:

`best_chunk_size:` `integer` best chunk_size with the lowest rest on the edges of the image

`tile_functions.`**`FilterImage`**(*image*, *chunk_size*, *data_path*, *filter*, *flag1*)

Filters an image with a given filter. Does not return an image but creates an HDF5 file with the filtered image instead.

Parameters:

`image:` `ndarray` array containing the image data in 8 bit grayscale

`chunk_size:` `integer` size of the chunks the data is divided into

`data_path:` `string` directory of the .h5 file containing the image data

`filter:` `function` filter from the `filters` module B

`filter_arguments:list` list of arguments for the filter

`tile_functions.`**`getImage`**(*image_path*)

Helper for opening an image and converting it to a numpy array filled with floats

Parameter:

`image_path:` `string` path to the file containing the image

Returns:

`image:` `ndarray` array containing the image data read from the file as a grayscale image

tile_functions.**getImageFromH5**(*image_path*, *image_name*)

Recomposes an image from data chunks in an .h5 file

Parameters:

image_path: `string` path to the file containing the image

image_name: `string` name of the file containing the image

Returns:

image: `ndarray` array containing the image data read from the file

tile_functions.**getExtrema**(*image*)

Wrapper for getting the maximum and minimum value of pixels in an image, using numpy's amin() and amax() functions

Parameter:

image: `ndarray` image containing the pixels which will be searched for extrema

Returns:

extrema: `tuple` tuple containing the minimum and maximum values of pixels

tile_functions.**MakeFile**(*file_name*, *file_path*)

Checks if given file at given path exists, deletes existing file and creates a new file with the given name.

Parameters:

file_name: `string` name of the file

file_path: `string` path to the file

tile_functions.**NewBoundaries**(*image*, *chunk_size*)

Expands the image at the edges so the chunks fit in without any rest at the borders. numpy's arraypad method does all the work.

Parameters:

image: `ndarray` array containing the image data

chunk_size: `integer` size of the data chunks

Returns:

image: `nparray` expanded image

tile_functions.**ResizeImage**(*image*, *scale_factor*)

Wrapper for the openCV resize method.

Parameters:

image_name: `string` name of the image to be resized

cource_image_path: `string` path to the directory containing the source image

scale_factor: `float` factor by which the image will be resized, can be > 1.0 as well as between 0 and 1.0.

Returns:

thumbnail: `ndarray` array containing the data of the resized image

`tile_functions.`**`RGBtoGray`**(*image*)

Helper function for opening an image and converting it to a numpy array filled with floats

Parameter:

`image_path:` `string` path to the file containing the image

Returns:

`image:` `ndarray` array containing the image data read from the file

`tile_functions.`**`WriteChunkData`**(*image*, *chunk_size*, *image_path*)

Divides the array with the image data into square chunks and writes them into a tree-like structure in an .h5 file

Parameters:

`image:` `ndarray` array containing the image data

`chunk_size:` `integer` size of the data chunks

`image_path:` `string` path to the image

B. Module: `filters`

The `filters` module introduces a basic filter-class which can be extended to create custom filters. It already implements a few useful filters needed to perform the image processing in this project. Custom filters can be imported by calling the script of choice with the filter's name as argument.

`filters.blur`(*chunk*, *filter_arguments*)

Wrapper for openCV's cv2.blur. Blurs an image using a kernel filled with ones.
(See http://docs.opencv.org/modules/imgproc/doc/filtering.html#cv2.blur.)

Parameters:

`chunk:` `ndarray` currently processed tile

`filter_arguments:` `list`, has to consist of: overlap (integer)

Returns:

`chunk:` `ndarray` resulting chunk which is smaller than original chunk by overlap

`filters.histeq`(*image*)

Wrapper for openCV's cv2.equalizeHist which creates the histogram of an image, equalizes it and returns the equalized image. Wrapper only handles type conversion into uint8.
(See http://docs.opencv.org/modules/imgproc/doc/histograms.html#cv2.equalizeHist.)

Parameter:

`image:` `ndarray` image to be equalized

Returns:

`image:` `ndarray` resulting image

Notes:

Does NOT work within the tiling mechanism as first a histogram of the WHOLE image needs to be created which gets distorted if only parts of the image are looked at.

`filters.dir_deriv`(*chunk*, *filter_arguments*)

Wrapper for openCV's cv2.Sobel function,
(see http://docs.opencv.org/modules/imgproc/doc/filtering.html#cv2.Sobel.)
which selects either the x or y directional derivative and kernel size. For processing of one single image pad image before using this function.

Parameters:

`chunk:` `ndarray` currently processed tile

`filter_arguments:` `list`, has to consist of: kernel size which has to be 1,3,5 or 7 (integer), direction 'x' or 'y' (string) (in this exact order)

Returns:

`chunk:` `ndarray` resulting chunk which is smaller than original chunk by overlap

`filters.lochisteq`(*chunk*, *filter_arguments*)

Local histogram equalization, blurring before using this function to get rid of local noise is highly recommended.

Parameters:

`chunk:` `ndarray` currently processed tile

`filter_arguments:` `list` has to consist of: overlap (integer)

Returns:

`chunk:` `ndarray` resulting chunk which is smaller than original chunk by overlap

Notes:

Computationally intensive, might take long!

`filters.median_dynamic`(*chunk*, *filter_arguments*)

Wrapper that uses the `_median_footprint()` function for creation of a footprint to feed into scipy's `ndimage.median_filter()`.
(see: http://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.filters.median_filter.html)

Parameters:

`chunk:` `ndarray` currently processed tile

`filter_arguments:` `list` has to consist of: overlap (integer)

Returns:

`chunk:` `ndarray` resulting chunk which is smaller as original chunk by overlap

Notes:

As for now, only circular footprint shapes are supported.

`filters.nan_median`(*chunk*, *filter_arguments*)

Median filter that ignores values of np.nan and calculates the median value only with the values that are not np.nan.

Parameters:

`chunk:` `ndarray` currently processed tile

`filter_arguments:` `list` has to consist of: overlap (integer)

Returns:

`chunk:` `ndarray` resulting chunk which is smaller than original chunk by overlap

`filters.openCV_median`(*chunk*, *filter_arguments*)

Wrapper for openCV's cv2.medianBlur
(see:http://docs.opencv.org/modules/imgproc/doc/filtering.html#cv2.medianBlur)

Parameters:

`chunk:` `ndarray` currently processed tile

`filter_arguments:` `list` has to consist of: overlap (integer)

Returns:

`chunk:` `ndarray` resulting chunk which is smaller than original chunk by overlap

`filters.otsu_threshold`(*image*)

Implementation of "Otsu Thresholding" which will calculate the threshold of the image that has the least intra class variance.

Parameter:

`image:` `ndarray` image for which the threshold will be calculated, needs to be single-channel and grayscale

Returns:

`t:` `integer` optimal threshold for the image

`filters.sobel`(*chunk*, *filter_arguments*)

Sobel filter implementation utilizing directional derivative filter.

Parameters:

`chunk:` `ndarray` currently processed tile

`filter_arguments:` `list`, size of the directional derivative kernel which has to be 1,3,5 or 7 (integer)

Returns:

`chunk:` `ndarray` resulting chunk which is smaller than original chunk by overlap.

`filters.thresh_range`(*chunk*, *filter_arguments*)

Thresholding filter which supports an upper and lower thresholding value and sets all values bigger than the lower threshold and smaller than the upper threshold to 255 and all other values to 0

Parameters:

`chunk:` `ndarray` currently processed tile

`filter_arguments:` `list`, has to consist of: overlap (integer), lower threshold (integer), upper threshold (float), (in this exact order)

Returns:

`chunk:` `ndarray` resulting chunk which is smaller than original chunk by overlap

Notes:

Convention change: zeros are now background and ones (255) foreground!

`filters.thresh_value`(*chunk*, *filter_arguments*)

Standard thresholding filter which sets all values below a given threshold to a given value and all values equal or above the threshold to a second given value

Parameters:

`chunk:` `ndarray` currently processed tile

`filter_arguments:` `list`, has to consist of: overlap (integer), threshold (integer), value1 (integer), value2 (integer) (in this exact order)

Returns:

`chunk:` `ndarray` resulting chunk which is smaller than original chunk by overlap

`filters.threshold`(*chunk*, *filter_arguments*)

Standard thresholding filter which sets all values below a specified threshold to 0 and all values equal or above the threshold to 255

Parameters:

`chunk:` `ndarray` currently processed tile

`filter_arguments:` `list` has to consist of: overlap (integer), threshold (float) (in this exact order!)

Returns:

`chunk:` `ndarray` resulting chunk which is smaller than original chunk by overlap

`filters.unsharp_mask`(*chunk*, *filter_arguments*)

Implementation of unsharp masking. The unsharp mask is created by blurring the original image and subtracting it from the original one.

Parameters:

`chunk:` `ndarray` currently processed tile

`filter_arguments:` `list` has to consist of: size of the blurring kernel (integer), σ of the gaussian blur (integer), threshold for unsharp mask creation (integer), weight for the superposition of original and masked image (float) (in this exact order)

Returns:

`sharpened:` `ndarray` resulting sharpened chunk which is smaller than original chunk by overlap

C. Module: `triangulation_functions`

The triangulation_functions module provides custom classes for the handling of points, segments and triangles, along with methods to perform a triangulation, classify its triangles and extract skeleton segments from each triangle.

`triangulation_functions.`**`point`**(*x*, *y*)

Simple custom point class.

Members:

`x:` `integer` x-coordinate

`y:` `integer` y-coordinate

Functions:

`__eq__(point):` True if coordinates of two points are equal, false otherwise

`__ne__(point):` inverse of `__eq__(point)`

`triangulation_functions.`**`segment`**(*p1*, *p2*)

Custom segment class.

Members:

`p1:` `point`

`p2:` `point`

`midpoint:` `point` midpoint between p1 and p2, rounded to the nearest integer

Functions:

`__eq__(p1,p2):` true if both points are equal

`__ne__(p1,p2):` inverse of `__eq__(p1,p2)`

`connects(segment):` returns True if the segments are not equal and share one point, False otherwise

`get_cp(segment):` returns the connecting point between two segments or `None` if there is no connecting point

`triangulation_functions.`**`shape_triangle`**(*p1*, *p2*, *p3*)

Custom triangle class consisting of three points and three edges.

Members:

`p1:` `point`

`p2:` `point`

`p3:` `point`

`s1:` `segment(p1,p2)`

s2: segment(p2,p3)
s3: segment(p3,p1)

triangulation_functions.**end_triangle**(*int_edge*, *ext_edge1*, *ext_edge2*)

End triangle class consisting of one internal and two external edges.

Members:

int_edge: segment internal edge
ext_edge1: segment first external edge
ext_edge2: segment second external edge
p1: point
p2: point
p3: point
centroid: point centroid of the triangle, coordinates rounded to the neares integer
sks: segment skeleton segment from the centroid to the midpoint of the internal edge

triangulation_functions.**normal_triangle**(*int_edge1*, *int_edge2*, *ext_edge*)

Normal triangle class consisting of two internal and one external edges.

Members:

int_edge1: segment first internal edge
int_edge2:segment second internal edge
ext_edge: segment external edge
p1: point
p2: point
p3: point
sks: segment skeleton segment from the midpoint of the first internal edge to the midpoint of the second internal edge

triangulation_functions.**junction_triangle**(*int_edge1*, *int_edge2*, *int_edge3*)

Junction triangle class consisting of three internal edges.

Members:

int_edge1: segment first internal edge
int_edge2: segment second internal edge
int_edge3: segment third internal edge
p1: point
p2: point
p3: point
sks1: segment skeleton segment from the centroid to the midpoint of the first internal edge
sks2: segment skeleton segment from the centroid to the midpoint of the second internal edge
sks3: segment skeleton segment from the centroid to the midpoint of the third internal edge

triangulation_functions.**getContours**(*image*)

Wrapper around openCV's cv2.findContours() function.
(See: http://docs.opencv.org/modules/imgproc/doc/
structural_analysis_and_shape_descriptors.html#cv2.findContours.)
Sets suitable options and converts contours to a list of ndarrays.

Parameter:

`image:` `ndarray` input image

Returns:

`contours:` `list` list of contours

`triangulation_functions.`**`thresholdContours`**(*contours*, *threshold*)

Thresholds a given list of contours by length.

Parameters:

`contours:` `list` list of contours

`threshold:` `integer` length-threshold

Returns:

`filtered_cont:` `list` list of contours

`triangulation_functions.`**`getTriangulation`**(*filtered_cont*)

Creates a Constrained Delaunay Triangulation of a given list of contours using the Poly2Tri library.

Parameter:

`filtered_cont:` `list` list of contours

Returns:

`triangles:` `list` list of `p2t.triangle`

`triangulation_functions.`**`flatten_contours`**(*contours*)

Helper function for flattening contours consisting of nested ndarrays.

Parameter:

`contours:` `list` list of nested `ndarrays`

Returns:

`flattened_contours:` `list` list of flattened ndarrays

`triangulation_functions.`**`cont_to_points`**(*contours*)

Converts a list of contour point coordinates to a list of `triangulation_function.points`

Parameter:

`contours:` `list` list of contours consisting of coordinates

Returns:

`new_contours:` `list` list of contours consisting of points

`triangulation_functions.`**`cont_to_segment`**(*contours*)

Converts a list of contour point coordinates to a list of `triangulation_function.segments`

Parameter:

contours: `list` list of contour points

Returns:

new_contours: `list` list of contour segments

`triangulation_functions.`**`triangle_classify`**(*triangles,contour_segments*)

Classifies a given list of lists of `p2t.triangle`s into end triangles, normal triangles and junction triangles by comparing their segments to a given list of contour segments.

Parameters:

triangles: `list` list of `p2t.triangle`s

contour_segments: `list` list of `triangle_function.segment`s

Returns:

triangle_classes: `list` triangle_classes: list of lists of classified triangles consisting of (end_triangles, normal_triangles, junction_triangles) (in exactly that order!)

`triangulation_functions.`**`extract_skeleton`**(*triangle_classes*)

Extracts the skeleton segments from a list of triangles depending on the triangle class.

Parameter:

triangle_classes: `list` list of triangle lists, consisting of one list respectively for end triangles, normal triangles and junction triangles (in exactly that order!)

Returns:

skeleton: `list` list of skeleton segments

`triangulation_functions.`**`cv_distanceMap`**(*image*)

Wrapper for openCV's DistTransform function.
(See http://docs.opencv.org/modules/imgproc/doc/miscellaneous_transformations.html #cv.DistTransform.)
which sets all options with values that fit the processing of leaves.

Parameter:

image: `ndarray` image for which the distance map will be created, has to be single channel grayscale

Returns:

dst: `ndarray` distance map of the image with precisely calculated distances

Bibliography

[1] Francesc Alted, Ivan Vilata, et al. PyTables: Hierarchical datasets in Python, 2002–2012. URL: `http://www.pytables.org/`.

[2] Hauser M. Baumgarten W. Detection, extraction, and analysis of the vein network of the slime mould physarum polycephalum. *Journal of Computational Interdisciplinary Sciences*, 2010.

[3] Hauser M. Baumgarten W. Computational algorithms for extraction and analysis of two-dimensional transportation networks. *Journal of Computational Interdisciplinary Sciences*, 2012.

[4] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[5] Georg Brandl. Spinx 1.1.3, oct 2012. URL: `http://sphinx-doc.org/`.

[6] Bohn S. et al. Constitutive property of the local organization of leaf venation networks. *Physical Review E, Vol. 65*, 2002.

[7] Price A. et al. Leaf extraction and analysis framework graphical user interface: Segmenting and analyzing the structure of leaf veins and areoles. *Plant Physiology*, 2010.

[8] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Distance transforms of sampled functions. Technical report, Cornell Computing and Information Science, 2004.

[9] Borgefors G. Distance transformations in digital images. *Computer Vision, Graphics, and Image Processing Vol. 34*, 1986.

[10] Mason Green. poly2tri 0.3.3, may 2011. URL: `http://code.google.com/p/poly2tri/`.

[11] The HDF Group. Hierarchical data format version 5, 2000-2010. URL: `http://www.hdfgroup.org/HDF5`.

[12] Zou J. and Yan H. A new skeletonization algorithm based on constrained delaunay triangulation. *Fifth International Symposium on Signal Processing and its Applications,*, 1999.

[13] Zou J. and Yan H. Skeletonization of ribbon-like shapes based on regularity and singularity analyses. *IEEE Transactions on Systems, Man, and Cybernetics, —Part B: Cybernetics, Vol. 31, No. 3*, 2001.

[14] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–2012. URL: `http://www.scipy.org/`.

[15] University of California Museum of Paleontology. Ucmp cleared leaf collection. URL: `www.ucmp.berkeley.edu/science/clearedleaf.php`.

[16] Nobuyuki Otsu. A tlreshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-9, No. 1*, 1979.

[17] Morrison P. and Zou J. Skeletonization based on error reduction. *Pattern Recognition Vol. 39*, 2006.

[18] Morrison P. and Zou J. Triangle refinement in a constrained delaunay triangulation skeleton. *Pattern Recognition Vol. 40*, 2007.

[19] John C. Russ. *Image Processing Handbook*. CRC Press, Taylor & Francis Group, 2011.

[20] Pete Shinners. Pygame, January 2013. URL: `http://www.pygame.org`.

[21] Satoshi Suzuki and Keichi Abe. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing Vol. 30*, 1985.

[22] Cho-Huak Teh and Roland T. Chin. On the detection of dominant points on digital curves. *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 11, No. 8*, 1989.

Erklärung nach §13(8) der Prüfungsordnung für den Bachelor-Studiengang Physik und den Master-Studiengang Physik an der Universität Göttingen:

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe.
Darüberhinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, im Rahmen einer nichtbestandenen Prüfung an dieser oder einer anderen Hochschule eingereicht wurde.

Göttingen, den 12. März 2013

(Jana Lasser)